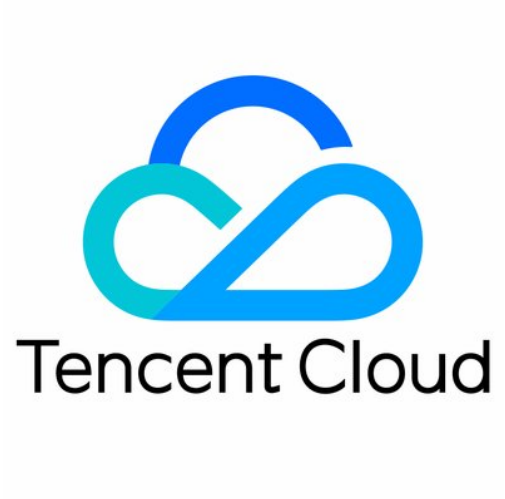
Instant Messaging

SDK Documentation

Product Documentation



Contents

SDK Documentation
SDK Download

Last updated : 2021-09-13 16:37:03

You can download the latest source codes of IM SDKs and demos.

SDK Download

Native SDK	Download Address		Integration Guide	Update Log
Android	GitHub (recommended)	Gitee	Import TUIKit (Android) SDK Integration (Android)	Update Log (Native)
iOS	GitHub (recommended)	Gitee	Import TUIKit (iOS) SDK Integration (iOS)	
Mac	GitHub (recommended)	Gitee	SDK Integration (Mac)	
Windows	GitHub (recommended)	Gitee	SDK Integration (Windows)	

Web SDK	Download Address			Integration Guide	Update Log
Web & HTML5	npm (recommended)	GitHub	ZIP	SDK Integration (Web & Mini Program)	Update Log (Web & Mini Program)
Mini Program	npm (recommended)	GitHub			
Web SDK upload plugin	npm (recommended)	GitHub		-	

Demo and Solution Download

IM demos demonstrate IM features in various scenarios. You can install and try the demos for different platforms and use cases.

Category	Description	Platform	Download Address		Reference
IM demo	Includes all IM features and the capability to co-anchor in a live stream in a group.	Android	GitHub (recommended)	ZIP	Demo Quick Start
		iOS	GitHub (recommended)		
		Web	GitHub (recommended)		
		WeChat Mini Program	GitHub (recommended)		
IM demo (Flutter)	Includes the main features of IM.	Flutter	GitHub (recommended)	ZIP	See GitHub README.
Live commerce demo	Includes live commerce capabilities such as on-screen comments, coupons, and shopping carts.	WeChat Mini Program	GitHub (recommended)	ZIP	Mini Program Live Commerce Mini Program Live Streaming SDK APIs
Interactive live streaming demo	Includes interactive capabilities such as starting live streaming and giving likes.	HTML5 and Web	GitHub (recommended)	ZIP	Quick TWebLive Run TWebLive Component

Relevant Documentation

- Pricing

Quick Integration (Including UI Library) Overview

Last updated : 2020-08-05 12:16:32

Introduction to TUIKit

TUIKit is a UI component library based on the IM SDK. It provides universal UI components such as the conversation list, chat UI, and contact list, enabling developers to quickly build custom IM apps according to their business needs. TUIKit can not only provide UI features through components, but also implement related IM logic and data processing by calling relevant IM SDK APIs, allowing developers to focus on their own business needs or custom extensions.

Related Documentation

- Pricing

Step 1: Import TUIKit
Import TUIKit (Android)

Last updated : 2021-09-02 11:24:24

Environment Requirements

- Android Studio 3.6.1
- Gradle-5.1.1

Integration Description

`TUIKit` can be integrated using the `module` source code.

Integrating the module source code

TUIKit source code download address

1. Download the `Demo` source code from `GitHub`. Note that you need to copy the `tuikit` folder to a directory of your project and use it as a module.

2. Add the following to `settings.gradle`:

```
include ':tuikit'
```

3. Add the following to `build.gradle` in `APP`:

```
dependencies {
implementation project(':tuikit')

......
}
```

4. Modify the `build.gradle` file in `tuikit`: replace the version numbers in the file with those in `build.gradle` in `app`. The following is an example:

```
android {
compileSdkVersion 30
defaultConfig {
minSdkVersion 19
targetSdkVersion 30
versionCode 1
versionName "1.0"
testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}
......
}
```

5. Add the following to the `gradle.properties` file to replace the class in `AndroidX` with that in `support`:

```
android.enableJetifier=true
```

6. Add the following to the `build.gradle` file of the `root` project to add the `maven` repository:

```
allprojects {
repositories {
maven { url "https://mirrors.tencent.com/nexus/repository/maven-public/" }
......
}
}
```

7. Sync the project, and compile and run it.

Initialization

Perform initialization in `onCreate` of `Application`:

```
public class DemoApplication extends Application {
public static final int SDKAPPID = 0; // Your SDKAppID
@Override
public void onCreate() {
super.onCreate();
// Set parameters as needed
TUIKitConfigs configs = TUIKit.getConfigs();
configs.setSdkConfig(new V2TIMSDKConfig());
```

```
configs.setCustomFaceConfig(new CustomFaceConfig());
configs.setGeneralConfig(new GeneralConfig());
TUIKit.init(this, SDKAPPID, configs);
}
}
```

The `init` method is described as follows:

```
/**
* Initializes TUIKit
*
* @param context //App context, usually corresponds to ApplicationContext
* @param sdkAppID //`SDKAppID` assigned to you when you register the app in Tencent Cloud
* @param configs //Relevant configuration items of TUIKit. Usually, you can use the default confi
guration.
*/
public static void init(Context context, int sdkAppID, TUIKitConfigs configs)
```

Import TUIKit (iOS)

Last updated : 2021-08-18 10:15:29

Environment Requirements

- Xcode 10 or later
- iOS 8.0 or later

Integration Description

CocoaPods integration (recommended)

TUIKit supports CocoaPods integration and manual integration. We recommend that you use CocoaPods integration to ensure that you can update to the latest version at any time.

1. Add the following content in the Podfile.

```
// TUIKit uses a third-party static library. This setting needs to be blocked.
#use_frameworks!
// TXIMSDK_TUIKit_live_iOS uses the *.xcassets resource file. You need to add this statement to prevent it from conflicting with other resource files in the project.
install! 'cocoapods', :disable_input_output_paths => true
// Integrate the chat, relationship chain, and group features.
pod 'TXIMSDK_TUIKit_iOS'
// Integratevoice and video calls, group livestreaming and livestreaming plazas, using the TXLiteAVSDK_TRTC library as the default dependency.
pod 'TXIMSDK_TUIKit_live_iOS'
// Integrate voice and video calls, group livestreaming, and livestreaming plazas, using the TXLiteAVSDK_Professional TRTC library as the default dependency.
// pod 'TXIMSDK_TUIKit_live_iOS_Professional'
```

Note :
1. `TXIMSDK_TUIKit_live_iOS` and `TXIMSDK_TUIKit_iOS` must have consistent versions. Otherwise, logic exceptions may occur.
2. Do not integrate different Tencent Cloud audio and video libraries at the same time to avoid symbol conflicts. If you use a library not of the TRTC version, we recommend that you remove it and integrate the `TXIMSDK_TUIKit_iOS_Professional` version. The audio and video library of the LiteAV_Professional version contains all basic audio and video capabilities.

2. Run the following command to install TUIKit.

```
pod install
```

If the latest SDK version cannot be installed, run the following command to update the local CocoaPods repository list.

```
pod repo update
```

Manual integration (not recommended)

1. Add the ImSDK file path to Framework Search Path and manually add the TUIKit and ImSDK directories to your project.
2. Manually add the third-party library used by TUIKit to your project:
   - MMLayout - Tag : 0.2.0
   - SDWebImage - Tag : 5.9.0
   - ReactiveObjC - Tag : 3.1.1
   - Toast - Tag : 4.0.0
   - TXLiteAVSDK_TRTC

Importing TUIKit

1. Introduce TUIKit in the AppDelegate.m file and initialize it.

```
#import "TUIKit.h"
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {
[[TUIKit sharedInstance] setupWithAppId:sdkAppid]; // SDKAppID can be obtained from the IM con
sole.
}
```

2. Compile and save the file.
   If compilation is successful, integration has been completed. If compilation fails, check the cause of the error or perform integration again based on this document.

FAQs

1 target has transitive dependencies that include statically linked binaries

If this error occurs during the pod process, this is because `TUIKit` is using a third-party static library. You need to comment out `use_frameworks!` in the podfile.

If you need to use `use_frameworks!` , use `cocoapods 1.9.0` or a later version for `pod install` and modify it as follows:

```
use_frameworks! :linkage => :static
```

If you use `swift` , change the reference of the header file to the reference format of @import module name.

Step 2: Build Quickly
Build Quickly (Android)

Last updated : 2021-10-21 14:24:43

Instant messaging software usually consists of basic interfaces such as the chat window and conversation list. TUIKit provides a set of basic UI implementations to simplify IM SDK integration. It takes only a few lines of code to use the IM SDK to provide communication features in your project.

Creating the Conversation List Interface

ConversationLayout of the conversation list is inherited from LinearLayout. The acquisition, synchronization, display, and interaction are all encapsulated in TUIKit. You can use the conversation list UI with the same convenience as a normal Android view.

1. Set the layout in any layout.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent">
<com.tencent.qcloud.tim.uikit.modules.conversation.ConversationLayout
android:id="@+id/conversation_layout"
android:layout_width="match_parent"
android:layout_height="match_parent" />
</LinearLayout>
```

2. Reference in code:

```
// Obtain the conversation list panel from the layout file
ConversationLayout conversationLayout = findViewById(R.id.conversation_layout);
// Initialize the conversation list panel
conversationLayout.initDefault();
```

Opening the Chat UI

1. Set the layout in any layout.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
```

```
android:layout_width="match_parent"
android:layout_height="match_parent">
<com.tencent.qcloud.tim.uikit.modules.chat.ChatLayout
android:id="@+id/chat_layout"
android:layout_width="match_parent"
android:layout_height="match_parent"/>
</LinearLayout>
```

2. Reference in code:

```
// Obtain the chat panel from the layout file
ChatLayout chatLayout = findViewById(R.id.chat_layout);
// Initialize the default UI and interaction of the one-to-one chat panel
chatLayout.initDefault();
// Passes in a ChatInfo instance. This instance must contain required chat information and is usu
```

Adding the Contact UI

1. Set the layout in any layout.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent">
<com.tencent.qcloud.tim.uikit.modules.contact.ContactLayout
android:id="@+id/contact_layout"
android:layout_width="match_parent"
android:layout_height="match_parent" />
</LinearLayout>
```

2. Reference in code:

```
// Obtain the contact panel from the layout file
ContactLayout contactLayout = findViewById(R.id.contact_layout);
// Initialize the default UI and interaction of the contact panel
contactLayout.initDefault();
```

Build Quickly (iOS)

Last updated : 2021-08-18 10:15:29

Instant messaging software usually consists of several basic UIs such as the chat window and conversation list. TUIKit provides a set of basic UI implementations to simplify IM SDK integration. It only takes a few lines of code to use the IM SDK to provide communication in your project.

Creating the Conversation List Interface

To create a conversation list, you only need to create a TUIConversationListController object. The conversation list reads recent contacts from the database. When a user clicks a contact, TUIConversationListController calls back the event to the upper layer.

```
// Configure conversation listening
[[TUIKitListenerManager sharedInstance] addConversationListControllerListener:self];
// Create a conversation list
TUIConversationListController *vc = [[TUIConversationListController alloc] init];
[self.navigationController pushViewController:vc animated:YES];
- (void)conversationListController:(TUIConversationListController *)conversationController didSel
ectConversation:(TUIConversationCell *)conversation
{
// Conversation list click event, typically, openning the chat interface
}
```

Opening the Chat Interface

During chat interface initialization, the upper layer needs to pass in the conversation information of the current chat interface. The sample code is as follows:

```
TUIConversationCellData *data = [[TUIConversationCellData alloc] init];
data.groupID = @"groupID"; // For a group conversation, pass in the corresponding group ID
data.userID = @"userID"; // For a one-to-one conversation, pass in the peer user ID
TUIChatController *vc = [[TUIChatController alloc] initWithConversation:data];
[self.navigationController pushViewController:vc animated:YES];
```

TUIChatController will automatically pull and display the historical messages of the user.

Adding the Contacts Interface

The contacts interface does not require other dependencies. You only need to create the object and display it.

```
TUIContactController *vc = [[TUIContactController alloc] init];
[self.navigationController pushViewController:vc animated:YES];
```

Step 3: Set Styles
Set Styles (Android)

Last updated : 2021-10-21 14:20:55

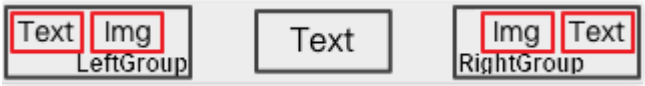
This document describes how to set styles for Android.

Setting the Conversation List Style

The conversation list layout consists of TitleBarLayout and ConversationListLayout. Each part provides UI styles and event registration APIs that can be modified.

Modifying the TitleBarLayout style

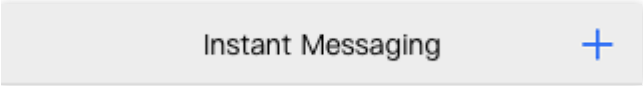
The title bar itself has all the features of a view. In addition, it is divided into three parts: left group, middle group, and right group.



To make custom modifications, see ITitleBarLayout.
For example, the following code hides LeftGroup, sets the title in the middle, and hides the text and image buttons on the right in ConversationLayout:

```
// Get TitleBarLayout
TitleBarLayout titleBarLayout = mConversationLayout.findViewById(R.id.conversation_title);
// Set the title
titleBarLayout.setTitle(getResources().getString(R.string.conversation_title), TitleBarLayout.POS
ITION.MIDDLE);
// Hide the left group
titleBarLayout.getLeftGroup().setVisibility(View.GONE);
// Set the menu icon on the right
titleBarLayout.setRightIcon(R.drawable.conversation_more);
```

The effect is shown below:



You can also customize click events:

```
// Menu class
mMenu = new Menu(getActivity(), titleBarLayout, Menu.MENU_TYPE_CONVERSATION);
// Click event that responds to a menu button
titleBarLayout.setOnRightClickListener(new View.OnClickListener() {
@Override
public void onClick(View view) {
if (mMenu.isShowing()) {
mMenu.hide();
} else {
mMenu.show();
}
}
});
```

Modifying the ConversationListLayout style

The custom conversation list layout is inherited from RecyclerView. After the user logs in, TUIKit reads the user's conversation list from the SDK.
You can customize common features for the conversation list. For example, you can confgure the background, font size, click event, "click and hold" event, and whether the profile photo has rounded corners. The sample code is as follows:

```
public static void customizeConversation(final ConversationLayout layout) {
// Get the conversation list from ConversationLayout
ConversationListLayout listLayout = layout.getConversationList();
listLayout.setItemTopTextSize(16); // Set the font size of top text in items
listLayout.setItemBottomTextSize(12); // Set the font size of bottom text items
listLayout.setItemDateTextSize(10); // Set the font size of timeline text in items
listLayout.setItemAvatarRadius(5); // Set the size of the rounded corners of the adapter item pro
file photo
listLayout.disableItemUnreadDot(false); // Set whether the item displays an unread badge or not.
Badge is displayed by default.
// Click and hold to pop up the menu
listLayout.setOnItemLongClickListener(new ConversationListLayout.OnItemLongClickListener() {
@Override
public void OnItemLongClick(View view, int position, ConversationInfo conversationInfo) {
startPopShow(view, position, conversationInfo);
}
});
}
```

For more information, please see ConversationLayoutSetting.java.

Setting the profile photo

The IM SDK does not store profile photos, so the developer needs to have a profile photo storage API in order to get profile photo URLs. TUIKit uses a random profile photo API as an example to show how to set a profile photo.
First, you need to upload a profile photo to your personal profile page and call the API to modify the profile.

```
HashMap<String, Object> hashMap = new HashMap<>();
// The profile photo. `mIconUrl` is the URL of the uploaded profile photo. See the example of a r
andom profile photo in Demo.
if (!TextUtils.isEmpty(mIconUrl)) {
hashMap.put(TIMUserProfile.TIM_PROFILE_TYPE_KEY_FACEURL, mIconUrl);
}
TIMFriendshipManager.getInstance().modifySelfProfile(hashMap, new TIMCallBack() {
@Override
public void onError(int i, String s) {
DemoLog.e(TAG, "modifySelfProfile err code = " + i + ", desc = " + s);
ToastUtil.toastShortMessage("Error code = " + i + ", desc = " + s);
}
@Override
public void onSuccess() {
DemoLog.i(TAG, "modifySelfProfile success");
}
});
```

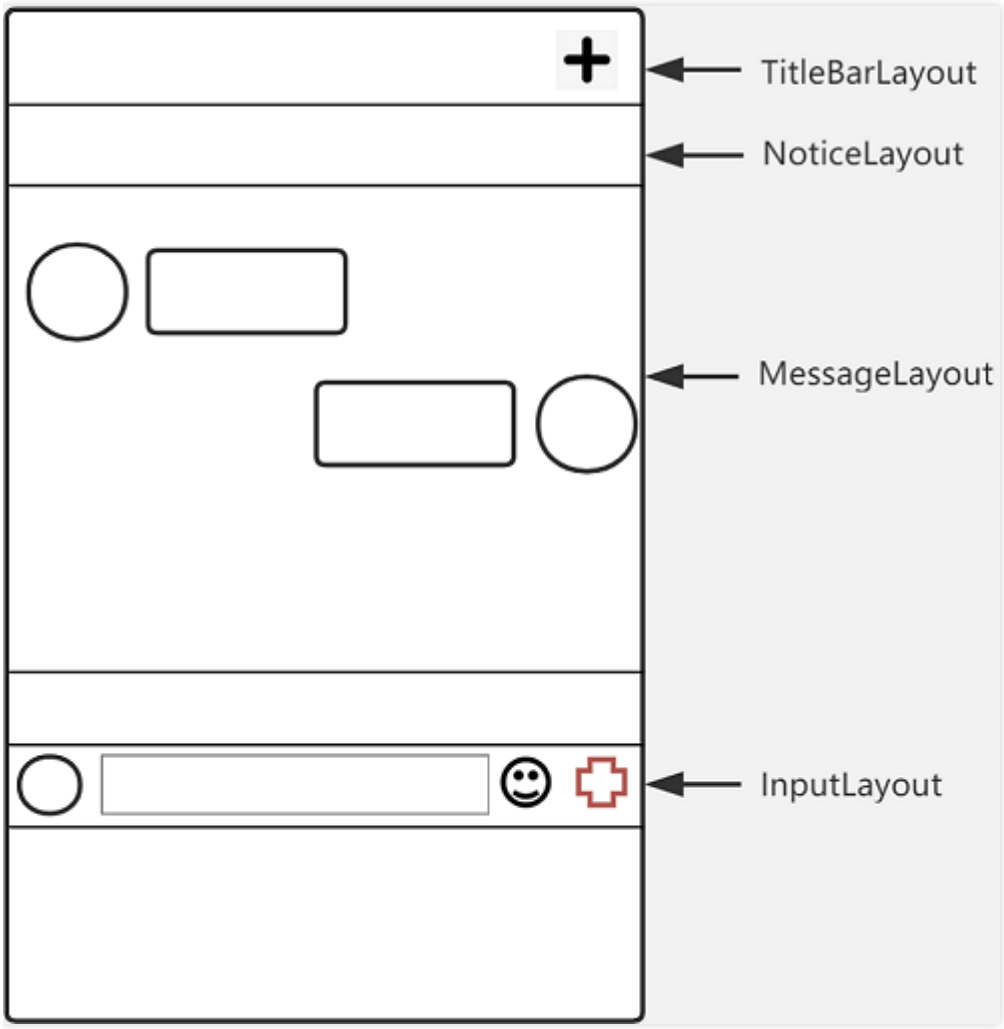
Get and display conversation list profile photos in `ConversationCommonHolder.java`:

```
if (!TextUtils.isEmpty(conversation.getIconUrl())) {
List<String> urllist = new ArrayList<>();
urllist.add(conversation.getIconUrl());
conversationIconView.setIconUrls(urllist);
urllist.clear();
}
```

Setting the Chat Interface

The chat interface includes TitleBarLayout, which is the same as that of the conversation list interface. The chat interface also includes NoticeLayout, MessageLayout, and InputLayout, as shown

in the following figure:




```
/**
* Get the notice layout in the chat interface
* @return
*/
NoticeLayout getNoticeLayout();
/**
* Get the message layout in the chat interface
* @return
*/
MessageLayout getMessageLayout();
/**
* Get the input layout in the chat interface
* @return
*/
InputLayout getInputLayout();
```

Modifying the NoticeLayout style

NoticeLayout consists of two TextViews, as shown in the following figure:


```
// Get NoticeLayout from ChatLayout
NoticeLayout noticeLayout = layout.getNoticeLayout();
// You can always display the notice area
noticeLayout.alwaysShow(true);
// Set the notice title
noticeLayout.getContent().setText("This is an ad");
// Set notice text
noticeLayout.getContentExtra().setText("Click to view your gift");
// Set the click event of notice
noticeLayout.setOnNoticeClickListener(new View.OnClickListener() {
@Override
public void onClick(View v) {
ToastUtil.toastShortMessage("You've received a bonus");
}
});
```

Modifying the MessageLayout style

MessageLayout is inherited from RecyclerView. This document describes how to customize the chat background, bubbles, text, and nicknames. For more information, please see IMessageProperties.java.

Modifying the chat background

You can customize the settings of the chat background.

```
// Get MessageLayout from ChatLayout
MessageLayout messageLayout = layout.getMessageLayout();
////// Set the chat background //////
messageLayout.setBackground(new ColorDrawable(0xB0E2FF00));
```

Modifying profile photo properties

To display a user, TUIKit reads the profile photo URL from the user profile and displays it.

```
// Set the profile photo and nickname on the chat interface
TIMUserProfile profile = TIMFriendshipManager.getInstance().queryUserProfile(msg.getFromUser());
```

```
if (profile == null) {
usernameText.setText(msg.getFromUser());
} else {
usernameText.setText(!TextUtils.isEmpty(profile.getNickName()) ? profile.getNickName() : msg.getF
romUser());
if (!TextUtils.isEmpty(profile.getFaceUrl()) && !msg.isSelf()) {
List<String> urllist = new ArrayList<>();
urllist.add(profile.getFaceUrl());
leftUserIcon.setIconUrls(urllist);
urllist.clear();
}
}
TIMUserProfile selfInfo = TIMFriendshipManager.getInstance().queryUserProfile(TIMManager.getInsta
nce().getLoginUser());
if (profile != null && msg.isSelf()) {
if (!TextUtils.isEmpty(selfInfo.getFaceUrl())) {
List<String> urllist = new ArrayList<>();
urllist.add(profile.getFaceUrl());
rightUserIcon.setIconUrls(urllist);
urllist.clear();
}
}
```

If the user does not set a profile photo, the default profile photo is displayed. You can customize the default profile photo, whether the profile photo has rounded corners, and the profile photo size.

```
// Get MessageLayout from ChatLayout
MessageLayout messageLayout = layout.getMessageLayout();
////// Set the profile photo //////
// Set the default profile photo. The recipient uses the same profile photo by default.
messageLayout.setAvatar(R.drawable.ic_chat_input_file);
// Set rounded corners for the profile photo. No rounded corners by default.
messageLayout.setAvatarRadius(50);
// Set the profile photo size
messageLayout.setAvatarSize(new int[]{48, 48});
```

Modifying bubbles

The recipient’s bubbles are on the left and your own bubbles are on the right. You can customize the bubble background for both parties.

```
// Get MessageLayout from ChatLayout
MessageLayout messageLayout = layout.getMessageLayout();
// Set your own bubble background
messageLayout.setRightBubble(context.getResources().getDrawable(R.drawable.chat_opposite_bg));
```

```
// Set the bubble background for the recipient
messageLayout.setLeftBubble(context.getResources().getDrawable(R.drawable.chat_self_bg));
```

Modifying the nickname style

You can customize the nickname style, including the font size and color. The nickname styles of both parties must be the same.

```
// Get MessageLayout from ChatLayout
MessageLayout messageLayout = layout.getMessageLayout();
////// Set the nickname style (the recipient uses the same style) //////
messageLayout.setNameFontSize(12);
messageLayout.setNameFontColor(0x8B5A2B00);
```

Modifying the chat content style

You can customize the font size and color for both parties, but the sender and recipient must use the same font size.

```
// Get MessageLayout from ChatLayout
MessageLayout messageLayout = layout.getMessageLayout();
// Set the chat content font size. The sender and the recipient use the same font size.
messageLayout.setChatContextFontSize(15);
// Set your own chat content font color
messageLayout.setRightChatContentFontColor(0xA9A9A900);
// Set the chat content font color for the recipient
messageLayout.setLeftChatContentFontColor(0xA020F000);
```

Modifying the chat timeline style

You can customize the background, font size, and font color of the chat timeline.

```
// Get MessageLayout from ChatLayout
MessageLayout messageLayout = layout.getMessageLayout();
// Set the background of the chat timeline
messageLayout.setChatTimeBubble(new ColorDrawable(0x8B691400));
// Set the font size of the chat timeline
messageLayout.setChatTimeFontSize(20);
// Set the font color of the chat timeline
messageLayout.setChatTimeFontColor(0xEE00EE00);
```

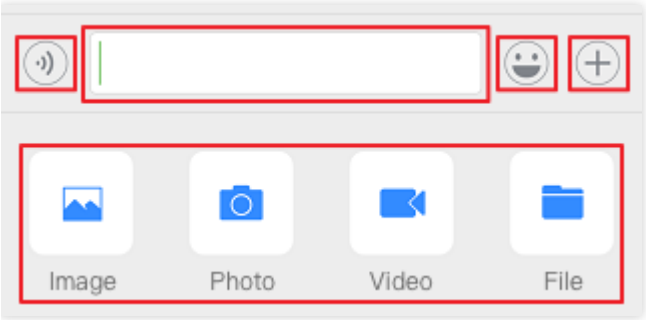
Modifying the tips message style

You can customize the background, font size, and font color of tips messages in chats.

```
// Get MessageLayout from ChatLayout
MessageLayout messageLayout = layout.getMessageLayout();
// Set the background of tips
messageLayout.setTipsMessageBubble(new ColorDrawable(0xA020F000));
// Set the font size of tips
messageLayout.setTipsMessageFontSize(20);
// Set the font color of tips
messageLayout.setTipsMessageFontColor(0x7CFC0000);
```

Setting InputLayout

InputLayout contains audio, text, emoji, and more (+) input options.



Hiding undesired features

You can hide or show the image sharing, photo taking, video recording, and file sending features on the "+" panel.

```
// Get InputLayout from ChatLayout
InputLayout inputLayout = layout.getInputLayout();
// Hide "take photo and send"
inputLayout.disableCaptureAction(true);
// Hide "send file"
inputLayout.disableSendFileAction(true);
// Hide "send image"
inputLayout.disableSendPhotoAction(true);
// Hide "record video and send"
inputLayout.disableVideoRecordAction(true);
```

Adding custom features (method 1)

You can customize and add action units to the "+" panel to provide more features.
The following code shows how to hide the "send file" feature and add an action unit which sends a message:

```
// Get InputLayout from ChatLayout
InputLayout inputLayout = layout.getInputLayout();
// Hide "send file"
inputLayout.disableSendFileAction(true);
// Define an action unit
InputMoreActionUnit unit = new InputMoreActionUnit();
unit.setIconResId(R.drawable.default_user_icon); // Set the unit icon
unit.setTitleId(R.string.profile); // Set the text title of the unit
unit.setOnClickListener(unit.new OnActionClickListener() { // Define the click event
@Override
public void onClick() {
ToastUtil.toastShortMessage("Custom more features");
MessageInfo info = MessageInfoUtil.buildTextMessage("Who am I");
layout.sendMessage(info, false);
}
});
// Add the action unit to the "+" panel
inputLayout.addAction(unit);
```

Adding custom features (method 2; added in version `5.4.666`)

The final effect is the same as method 1. The sample code is as follows:

```
class CustomChatController implements TUIChatControllerListener {
// This method is called to add a action unit each time the "+" button is clicked
@Override
public List<IBaseAction> onRegisterMoreActions() {
InputMoreActionUnit action = new InputMoreActionUnit() {
// Method triggered upon clicks
@Override
public void onAction(String chatInfoId, int chatType) {
// Create a text message
MessageInfo info = MessageInfoUtil.buildTextMessage("Who am I");
IBaseMessageSender messageSender = TUIKitListenerManager.getInstance().getMessageSender();
if (messageSender != null) {
// Send the message
messageSender.sendMessage(info, null, chatInfoId,
chatType == V2TIMConversation.V2TIM_GROUP, false, new IUIKitCallBack() {
@Override
public void onSuccess(Object data) {
Log.i("CustomChatController", "send success");
}
@Override
public void onError(String module, int errCode, String errMsg) {
Log.i("CustomChatController", "send failed");
}
```

```
});
}
}
};
action.setTitleId(R.string.profile);
action.setIconResId(R.drawable.default_user_icon);
List<IBaseAction> list = new ArrayList<>();
list.add(action);
return list;
}
......
}
// Register as soon as possible
TUIKitListenerManager.getInstance().addChatListener(new CustomChatController());
```

Replacing the "+" click event

You can customize features to replace the action units on the "+" panel.

```
// Get InputLayout from ChatLayout
InputLayout inputLayout = layout.getInputLayout();
// Replace the feature entry on the "+" panel with a custom event
inputLayout.replaceMoreInput(new View.OnClickListener() {
@Override
public void onClick(View v) {
ToastUtil.toastShortMessage("Custom  "+"  button event");
MessageInfo info = MessageInfoUtil.buildTextMessage("Custom message");
layout.sendMessage(info, false);
}
});
```

Replacing the panel displayed upon "+" clicking

You can customize the style of the "+" panel, the action units, and their features.

```
// Get InputLayout from ChatLayout
InputLayout inputLayout = layout.getInputLayout();
// Use a custom fragment to replace more features
inputLayout.replaceMoreInput(new CustomInputFragment());
```

The implementation of the new panel CustomInputFragment is the same as that of a common Fragment. Inflate the view at onCreateView and set the event. The following sample code shows how to add two buttons and pop up a toast when clicked:

```
public static class CustomInputFragment extends BaseInputFragment {
@Nullable
@Override
public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, Bundle savedInst
anceState) {
View baseView = inflater.inflate(R.layout.test_chat_input_custom_fragment, container, false);
Button btn1 = baseView.findViewById(R.id.test_send_message_btn1);
btn1.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View v) {
ToastUtil.toastShortMessage("Send a hyperlink message");
}
});
Button btn2 = baseView.findViewById(R.id.test_send_message_btn2);
btn2.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View v) {
ToastUtil.toastShortMessage("Send a message containing video and text");
}
});
return baseView;
}
}
```

Set Styles (iOS)

Last updated : 2020-12-01 11:30:08

This document describes how to set styles on iOS.

Modifying the Profile Photo

Modifying the default profile photo

To display a user on the interface, TUIKit reads the profile photo URL from the user's profile and displays the profile photo. If the user has not set a profile photo, the default profile photo is displayed.

You can set a custom image as the default profile photo.

```
TUIKitConfig *config = [TUIKitConfig defaultConfig];
// Modify the default profile photo
config.defaultAvatarImage = [UIImage imageNamed:@"Your Image"];
// Modify the default group profile photo
config.defaultGroupAvatarImage = [UIImage imageNamed:@"Your Image"];
```

Modifying the profile photo type

There are three profile photo types available: rectangle, rounded, and rounded rectangle.

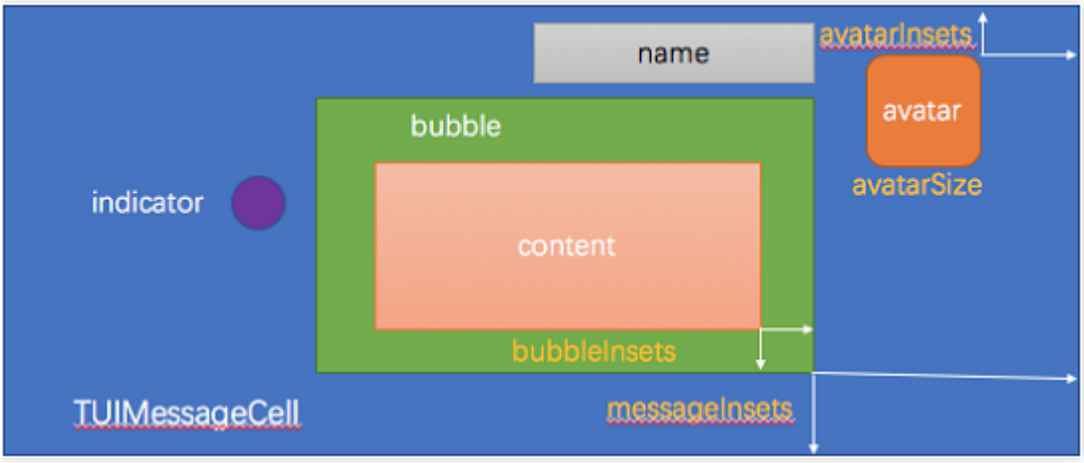
```
typedef NS_ENUM(NSInteger, TUIKitAvatarType) {
TAvatarTypeNone, /*Rectangle profile photo*/
TAvatarTypeRounded, /*Rounded profile photo*/
TAvatarTypeRadiusCorner, /*Rounded rectangle profile photo*/
};
```

You can modify the profile photo type in the same way you modify the default profile photo. The following is a code sample:

```
TUIKitConfig *config = [TUIKitConfig defaultConfig];
// Set rounded rectangle profile photo, with a corner radius of 5
config.avatarType = TAvatarTypeRadiusCorner;
config.avatarCornerRadius = 5.f;
```

Configuring the Chat Interface

The following figure shows how different views are arranged on the chat interface:



Setting the chat interface background

```
TUIChatController *vc = ...; // Get the chat interface object.
vc.messageController.view.backgroundColor = [UIColor greenColor];
```

Configuring messages

Setting bubble images

The images displayed in the bubble cells are obtained from TUIBubbleMessageCellData, which provides class methods to set the images.

```
// Set the outgoing bubble, which can be in normal status or highlighted status. The incoming bub
ble is set in the same way.
[TUIBubbleMessageCellData setOutgoingBubble:[UIImage imageNamed:@"bubble"]];
[TUIBubbleMessageCellData setOutgoingHighlightedBubble:[UIImage imageNamed:@"bubble_highlight"]];
```

Setting bubble margins

In TUIKit, text and audio messages are displayed in bubbles. TUIMessageCellLayout provides class methods to set bubbleInsets.

```
// Set the margins of the outgoing bubble. The incoming bubble is set in the same way.
[TUIMessageCellLayout outgoingTextMessageLayout].bubbleInsets = UIEdgeInsetsMake(10, 10, 20, 20);
```

Modifying the message font and color

The data of text messages comes from the TUITextMessageCellData class, through whose APIs you can modify the font and color of text messages.

```
// Set the font and color of outgoing text messages. The incoming text messages are set in the sa
me way.
[TUITextMessageCellData setOutgoingTextFont:[UIFont systemFontOfSize:20]];
[TUITextMessageCellData setOutgoingTextColor:[UIColor redColor]];
```

Configuring the profile photo

A profile photo is a common element to every message. To set the size and position of the profile photo, you need to first obtain the `layout` instance of the message. Take a text message as an example:

Setting the profile photo size

```
// Set profile photo size for the sender. The receiver's profile photo is set in the same way.
[TUIMessageCellLayout outgoingMessageLayout].avatarSize = CGSizeMake(100, 100);
```

Setting the profile photo location

```
// Set the profile photo location for the sender. The receiver's profile photo is set in the sam
e way.
[TUIMessageCellLayout outgoingTextMessageLayout].avatarInsets = UIEdgeInsetsMake(10, 10, 20, 20);
```

For other messages, obtain their `layout` instances to set the size and position of the profile photos.

Configuring the nickname font and color

Set the nickname font and color by modifying related properties in TUIMessageCellLayout in the same way the profile photo location is set.

```
// Set the nickname font for the receiver. The sender's nickname is set in the same way, but it
is not displayed by default.
[TUIMessageCellData setIncommingNameFont:[UIFont systemFontOfSize:20]];
[TUIMessageCellData setIncommingNameColor:[UIColor redColor]];
```

Configuring the More Menu

Clicking the "+" button next to the input box opens the More panel. By default, the More panel displays 4 options. You can add or delete options by modifying the moreMenus property in TUIChatController.
The following is a code sample to delete the File option:

```
TUIChatController *vc = [[TUIChatController alloc] initWithConversation:conv];
NSMutableArray *array = [NSMutableArray arrayWithArray:vc.moreMenus];
[array removeLastObject]; // Delete the last menu
vc.moreMenus = array; // Reset the property and apply it immediately
```

When the user clicks a button in the menu, TUIChatController notifies the upper layer with a callback event.

Note :

When the user clicks the default menu, you are also notified by a callback, but you do not need to process it.

Step 4: Enable Video Call
Enable Video Call (Android)

Last updated : 2021-10-21 10:13:29

TUIKit 4.8.50 and later versions provide audio/video call features for one-to-one and group chats based on TRTC and support interconnection between iOS and Android platforms. It should be noted that the integration method varies depending on the version:

Note :

- TUIKit versions **4.8.50 to 5.1.60** are integrated with the TRTC UI components and TRTC audio/video library by default. And therefore they support audio/video call related features by default.
- TUIKit **5.4.666** and later versions are not integrated with the TRTC UI components and TRTC audio/video library by default. The related audio/video logic is moved to the TUIKitLive component.
- **TUIKit-Live and TUIKit must have consistent versions. Otherwise, the audio/video call feature will experience an exception and cannot work properly.**

Step 1: Activate the TRTC Service

1. Log in to the IM console and click the target app card to go to the basic configuration page of the app.
2. Click **Activate** under **Activate Tencent Real-Time Communication (TRTC)** .
3. Click **Confirm** in the pop-up dialog box.
   A TRTC app with the same SDKAppID as the IM app will be created in the TRTC console. You can use the same account and authentication information for IM and TRTC.

Step 2: Configure Project Files

You are advised to use the source code to integrate TUIKit and TUIKit-Live. In this way, you can modify the source code to meet your business needs.

```
implementation project(':tuikit')
implementation project(':tuikit-live')
```

Step 3: Initialize TUIKit

To initialize TUIKit, enter the SDKAppID generated in Step 1.

```
TUIKitConfigs configs = TUIKit.getConfigs();
TUIKit.init(this, SDKAPPID, configs);
```

Step 4: Log In to TUIKit

Call the `login` API provided by TUIKit to log in to IM. For more information on how to generate UserSig, see How to Generate Usersig.

```
TUIKit.login(userID, userSig, new IUIKitCallBack() {
@Override
public void onSuccess(Object data) {
// Login succeeded
}

@Override
public void onError(String module, final int code, final String desc) {
// Login failed
}
});
```

Step 5: Initiate an Audio/Video call

When you tap **Video** or **Voice** on the chat UI, TUIKit automatically displays the call invitation UI and sends a call request to the peer.

Step 6: Answer an Audio/Video call

- When an **online** user receives a call invitation with **the app running in the foreground**, TUIKit automatically displays the call receiving UI, where the user can answer or reject the call.
- When an **offline** user receives a call invitation, offline push is required if the app call UI needs to be woken up. For more information about offline push, see Step 7.

Step 7: Offline Push

To implement offline push for audio/video calls, follow these steps:

1. Configure offline push for the app. For more information, see Offline Push Configuration.
2. Upgrade TUIKit to 4.9.1 or later.
3. Use TUIKit to initiate a call invitation. An offline push message will be generated by default. For more information about the message generation logic, see the `sendOnlineMessageWithOfflinePushInfo` method in the TRTCCallingImpl.java class.
4. After the peer receives the offline push message, refer to the `redirect` method in the OfflineMessageDispatcher.java class to wake up the call UI.

FAQs

1. What should I be aware of if I have created the TRTC and IM SDKAppIDs and want to integrate the IM SDK and TRTC SDK at the same time?

If you have created the TRTC and IM SDKAppIDs, you cannot use the same account or authentication information for these two apps. You need to generate a UserSig corresponding to the TRTC SDKAppID to perform authentication. For more information on how to generate UserSig, see How to Generate Usersig.
After obtaining the TRTC SDKAppID and UserSig, you need to replace the corresponding values in the `TRTCAVCallImpl` source code.

```
private void enterTRTCRoom() {
...
TRTCCloudDef.TRTCParams TRTCParams = new TRTCCloudDef.TRTCParams(mSdkAppId, mCurUserId, mCurUserS
ig, mCurRoomID, "", "");
...
}
```

2. How long is the default call invitation timeout duration? How can I modify the default timeout duration?

The default call invitation timeout duration is 30s. You can modify the `TIME_OUT_COUNT` field in `TRTCAVCallImpl` to customize the timeout duration.

3. Will an invitee receive a call invitation if the invitee goes offline and then online within the call invitation timeout duration?

- If the call invitation is initiated in a one-to-one chat, the invitee can receive the call invitation.
- If the call invitation is initiated in a group chat, the invitee cannot receive the call invitation.

Enable Video Call (iOS)

Last updated : 2021-09-02 12:16:58

TUIKit 4.8.50 and later versions provide audio/video call features and support interconnection between iOS and Android platforms. It should be noted that the integration method varies depending on the version:
TUIKit versions **4.8.50 to 5.1.60** are integrated with the TRTC UI components and TRTC audio/video library by default. And therefore they support audio/video call related features by default.
TUIKit versions **5.4.666** and later are not integrated with the TRTC UI components and TRTC audio/video library by default. The related audio/video logic is moved to the TUIKitLive component. If you need to use the audio/video call feature, integrate TUIKitLive by referring to Step 2.

Step 1: Activate the TRTC Service

1. Log in to the IM console and click the target app card to go to the basic configuration page of the app.
2. Click **Activate** under **Activate Tencent Real-Time Communication (TRTC)**.
3. Click **Confirm** in the pop-up dialog box.
   A TRTC app with the same SDKAppID as the IM app will be created in the TRTC console. You can use the same account and authentication information for IM and TRTC.

Step 2: Integrate TUIKitLive

1. Add the following content to the podfile file.

```
// You need to integrate TUIKit_live separately only for TUIKit 5.4.666 or later versions.
pod 'TXIMSDK_TUIKit_live_iOS' // By default, the audio and video library of the TXLiteAVSDK_TRTC version is integrated.
// pod 'TXIMSDK_TUIKit_live_iOS_Professional' // By default, the audio and video library of the TXLiteAVSDK_Professional version is integrated.
```

Do not integrate different Tencent Cloud audio and video libraries at the same time to avoid symbol conflicts. If you use a library not of the TRTC version, we recommend that you remove it and integrate the `TXIMSDK_TUIKit_iOS_Professional` version. The audio and video library of the LiteAV_Professional version contains all basic audio and video capabilities.

2. Run the following command to download the third-party library to the current project:

```
pod install
```

If you cannot install the latest TUIKit version, run the following command to update the local CocoaPods repository list:

```
pod repo update
```

Step 3: Initialize TUIKit

To initialize TUIKit, enter the SDKAppID generated in Step 1.

```
[[TUIKit sharedInstance] setupWithAppId:SDKAppID];
```

Step 4: Log In to TUIKit

Call the `login` API provided by TUIKit to log in to IM. For more information on how to generate UserSig, see How to Generate Usersig.

```
[[TUIKit sharedInstance] login:@"userID" userSig:@"userSig" succ:^{
NSLog(@"-----> login succeeds");
} fail:^(int code, NSString *msg) {
NSLog(@"-----> login fails");
}];
```

Step 5: Enable/Disable Audio/Video Call

In TUIKitLive, audio/video call is enabled by default. If you do not need audio/video call, use the `enableVideoCall` and `enableAudioCall` attributes in `TUIKitLive.h` to disable it. The code is as follows:

```
// Values of `enableVideoCall`: YES (enable); NO (disabled). Default value: Yes
[TUIKitLive shareInstance].enableVideoCall = YES;
```

```
// Values of `enableAudioCall`: YES (enable); NO (disabled). Default value: Yes
[TUIKitLive shareInstance].enableAudioCall = YES;
```

Step 6: Initiate an Audio/Video call

When you tap **Video** or **Voice** on the chat UI, TUIKit automatically displays the call invitation UI and sends a call request to the peer.

Step 7: Answer an Audio/Video call

- When an **online** user receives a call invitation, TUIKit automatically displays the call receiving UI, where the user can answer or reject the call.
- When an **offline** user receives a call invitation, offline push is required if the app call UI needs to be woken up. For more information about offline push, see Step 8.

Step 8: Offline Push

To implement offline push for audio/video calls, follow these steps:

1. Configure offline push for the app. For more information, see Offline Push Configuration.
2. Upgrade TUIKit to 4.9.1 or later.
3. Use TUIKit to initiate a call invitation. An offline push message will be generated by default. For more information about the message generation logic, see the `sendAPNsForCall` function in the `TUICall+Signal.m` class.
4. After the peer receives the offline push message, the peer can call the `didReceiveRemoteNotification` callback in the AppDelegate source code to wake up the call UI.

FAQs

1. What should I be aware of if I have created the TRTC and IM SDKAppIDs and want to integrate the IM SDK and TRTC SDK at the same time?

If you have created the TRTC and IM SDKAppIDs, you cannot use the same account or authentication information for these two apps. You need to generate a UserSig corresponding to the TRTC SDKAppID to perform authentication. For more information on how to generate UserSig, see How to Generate Usersig.

After obtaining the TRTC SDKAppID and UserSig, modify the following code in the `TUICall+TRTC.m` source code:

```
- (void)enterRoom {
TRTCParams *param = [[TRTCParams alloc] init];
// TRTC SDKAppID
param.sdkAppId = 1000000000
// UserSig generated based on the TRTC SDKAppID
param.userSig = "userSig"
}
```

2. How long is the default call invitation timeout duration? How can I modify the default timeout duration?

The default call invitation timeout duration is 30s. You can modify the `SIGNALING_EXTRA_KEY_TIME_OUT` field in `TUICallModel.m` to customize the timeout duration.

3. Will an invitee receive a call invitation if the invitee goes offline and then online within the call invitation timeout duration?

- If the call invitation is initiated in a one-to-one chat, the invitee can receive the call invitation.
- If the call invitation is initiated in a group chat, the invitee cannot receive the call invitation.

4. What can I do if TUIkitLive conflicts with the integrated audio and video library?

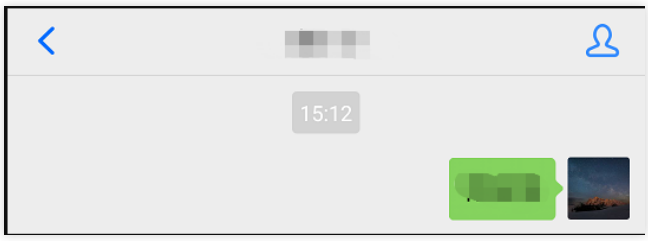
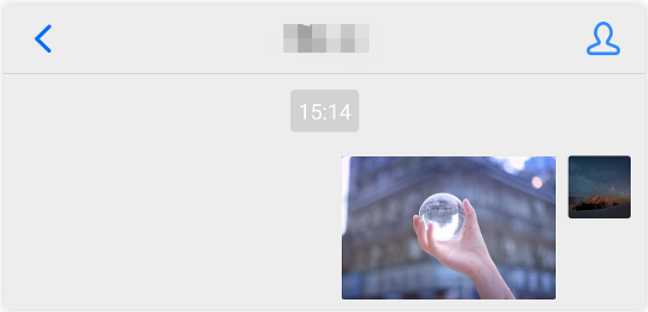
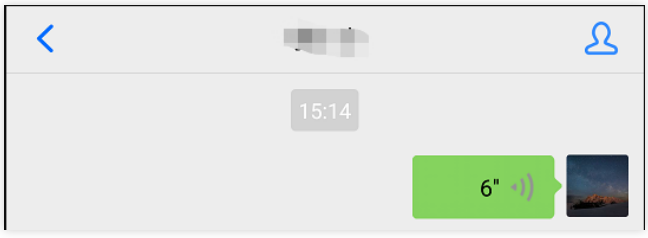
Do not integrate different Tencent Cloud audio and video libraries at the same time to avoid symbol conflicts. If you use a library not of the TRTC version, we recommend that you remove it and integrate the `TXIMSDK_TUIKit_iOS_Professional` version. The audio and video library of the LiteAV_Professional version contains all basic audio and video capabilities.
The audio and video library of the LiteAV_Enterprise version cannot coexist with TUIkitLive.

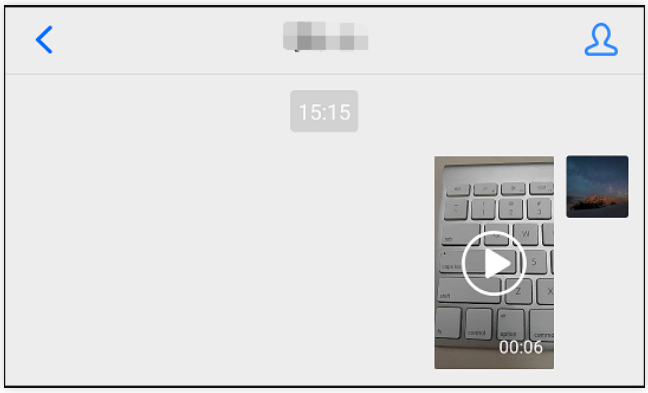
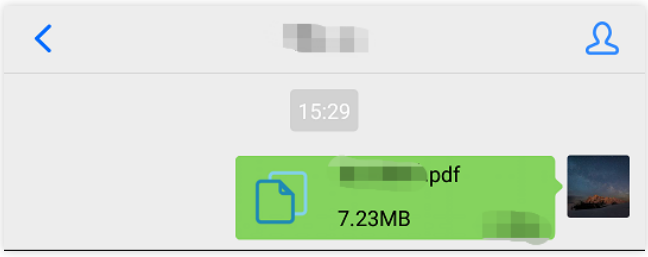
Step 5: Customize Messages
Customize Messages (Android)

Last updated : 2021-11-25 15:05:14

'TUIKit' has already rendered basic messages internally. You can easily adjust the message presentation style by setting attributes, or you can re-customize the message style.

Basic Message Types

Message Type	Renderings
Text message	
Image message	
Voice message	
Video message	

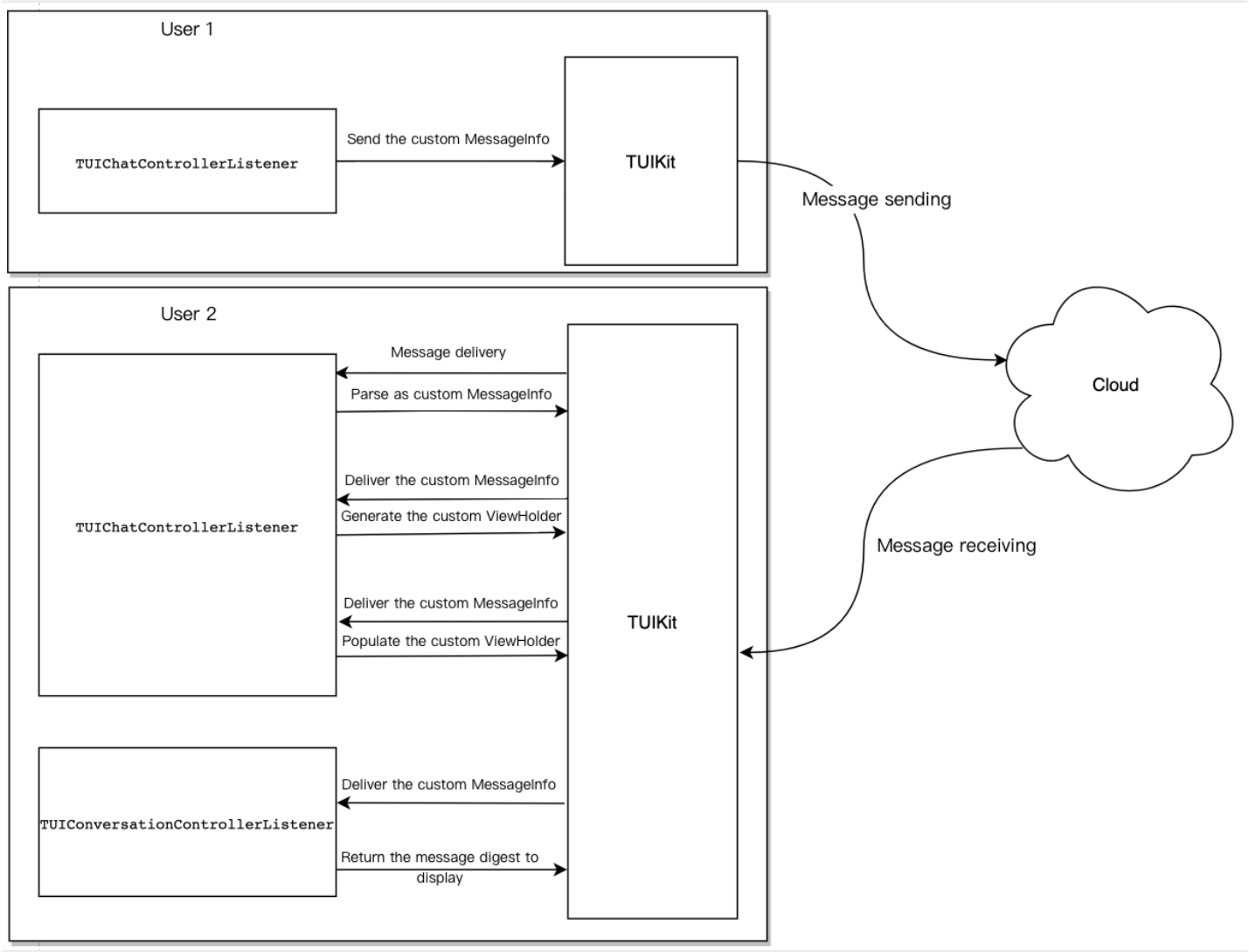
	
File message	

Customizing Messages

- If the basic message types do not meet your requirements, you can customize messages as needed.
- This document uses sending a custom hypertext message that can redirect to the browser as an example to help you quickly understand the implementation process. **This document uses version '5.4.666' as an example, which is different from previous versions.**

Process of implementing message customization

Message customization process



As shown in the figure above, `TUIChatControllerListener` and `TUIConversationControllerListener` must be implemented for custom messages. `TUIChatControllerListener` is used to parse messages and generate the custom `MessageInfo` and create and populate the custom `ViewHolder` to be displayed on the chat page. `TUIConversationControllerListener` is used to generate the message abstract to be displayed on the conversation list.

- `TUIKit` uses `RecyclerView` to display messages. To display custom messages, you need to create a custom message `ViewHolder` to store the `view` for displaying the content of custom messages.
- You need to register your own `TUIChatControllerListener` and `TUIConversationControllerListener` with `TUIKitListenerManager` as soon as possible.

Creating a custom welcome message

Customize `MessageInfo` and implement the custom message parsing method.

```
public class HelloChatController implements TUIChatControllerListener {
// Define that `HelloMessageInfo` is inherited from `MessageInfo`
static class HelloMessageInfo extends MessageInfo {
// Specify the message type ID, which cannot be repeated, including not repeated with the ID of a
built-in message type. A number greater than 100002 is recommended.
public static final int MSG_TYPE_HELLO = 100002;
}
// Implement the `createCommonInfoFromTimMessage` method of TUIChatControllerListener to parse me
ssages and generate `MessageInfo`
// If the message is customized by yourself, the custom `MessageInfo` is generated and returned.
Otherwise, `null` is returned, indicating that processing cannot be performed.
@Override
public IBaseInfo createCommonInfoFromTimMessage(V2TIMMessage timMessage) {
if (timMessage.getElemType() == V2TIMMessage.V2TIM_ELEM_TYPE_CUSTOM) {
V2TIMCustomElem customElem = timMessage.getCustomElem();
if (customElem == null || customElem.getData() == null) {
return null;
}
CustomHelloMessage helloMessage = null;
try {
helloMessage = new Gson().fromJson(new String(customElem.getData()), CustomHelloMessage.class);
} catch (Exception e) {
DemoLog.w(TAG, "invalid json: " + new String(customElem.getData()) + " " + e.getMessage());
}
if (helloMessage != null && TextUtils.equals(helloMessage.businessID, TUIKitConstants.BUSINESS_ID
_CUSTOM_HELLO)) {
MessageInfo messageInfo = new HelloMessageInfo();
// Set the message type ID, which is required
messageInfo.setMsgType(HelloMessageInfo.MSG_TYPE_HELLO);
// Settings
MessageInfoUtil.setMessageInfoCommonAttributes(messageInfo, timMessage);
Context context = TUIKit.getAppContext();
if (context != null) {
messageInfo.setExtra(context.getString(R.string.custom_msg));
}
return messageInfo;
}
}
return null;
}
......
}
```

Register `TUIChatControllerListener` with `TUIKitListenerManager` as soon as possible.

```
TUIKitListenerManager.getInstance().addChatListener(new HelloChatController());
```

You can create a custom message via a JSON string.

```
Gson gson = new Gson();
CustomHelloMessage customHelloMessage = new CustomHelloMessage();
customHelloMessage.version = TUIKitConstants.version;
customHelloMessage.text = DemoApplication.instance().getString(R.string.welcome_tip);
customHelloMessage.link = "https://cloud.tencent.com/document/product/269/3794";
String data = gson.toJson(customHelloMessage); // data = {"businessID":"text_link","link":"http
s://cloud.tencent.com/document/product/269/3794","text":"Welcome to IM","version":4}
// Creating a custom message based on a JSON string will call the rewritten `createCommonInfoFrom
TimMessage` method to parse messages and generate `MessageInfo`.
MessageInfo info = MessageInfoUtil.buildCustomMessage(data);
```

Sending a custom message

You can use the `ChatLayout` instance to send a custom message:

```
MessageInfo info = MessageInfoUtil.buildCustomMessage(data);
chatLayout.sendMessage(info)
```

You can also use `MessageSender` to send the message.

```
MessageInfo info = MessageInfoUtil.buildCustomMessage(data);
IBaseMessageSender messageSender = TUIKitListenerManager.getInstance().getMessageSender();
if (messageSender != null) {
// Send the message
messageSender.sendMessage(info, null, chatInfoId,
chatType == V2TIMConversation.V2TIM_GROUP, false, new IUIKitCallBack() {
@Override
public void onSuccess(Object data) {
Log.i("CustomChatController", "send success");
}
@Override
public void onError(String module, int errCode, String errMsg) {
Log.i("CustomChatController", "send failed");
}
});
}
```

Displaying a custom message in the chat box

Creating ViewHolder

`TUIKit` uses `RecyclerView` to display messages. To display custom messages, you need to create a custom message `ViewHolder` to store the `view` for displaying the content of custom messages.

```
// Define that `HelloViewHolder` is inherited from `MessageCustomHolder`
class HelloViewHolder extends MessageCustomHolder{
public HelloViewHolder(View itemView) {
super(itemView);
}
}
// Implement the `createCommonViewHolder` method of `TUIChatControllerListener` to create a ViewH
older
// Create a custom ViewHolder based on `viewType`. For a non-custom message, `null` is returned
@Override
public IBaseViewHolder createCommonViewHolder(ViewGroup parent, int viewType) {
if (viewType != HelloMessageInfo.MSG_TYPE_HELLO) {
return null;
}
if (parent == null) {
return null;
}
LayoutInflater inflater = LayoutInflater.from(TUIKit.getAppContext());
View contentView = inflater.inflate(R.layout.message_adapter_item_content, parent, false);
return new HelloViewHolder(contentView);
}
```

Setting the display content for a custom message

When displaying a message, `TUIKit` calls the `bindCommonViewHolder` method to set the custom message content.

```
// Implement the `bindCommonViewHolder` method of `TUIChatControllerListener` to populate the Vie
wHolder
// If the ViewHolder is a custom one, set the display content and return `true`, indicating that
processing is completed. Otherwise, `false` is returned.
@Override
public boolean bindCommonViewHolder(IBaseViewHolder baseViewHolder, IBaseInfo baseInfo, int posit
ion) {
if (baseViewHolder instanceof ICustomMessageViewGroup && baseInfo instanceof HelloMessageInfo) {
ICustomMessageViewGroup customHolder = (ICustomMessageViewGroup) baseViewHolder;
MessageInfo msg = (MessageInfo) baseInfo;
new CustomMessageDraw().onDraw(customHolder, msg, position);
return true;
}
```

```
return false;
}
```

Displaying the custom message abstract in the conversion list

```
// Define that `HelloConversationController` implements the `TUIConversationControllerListener` A
PI
public static class HelloConversationController implements TUIConversationControllerListener {
// Obtain the abstract string to be displayed in the conversion list. If the message is not the c
urrent user's custom message, `null` is returned, indicating no processing.
@Override
public CharSequence getConversationDisplayString(IBaseInfo baseInfo) {
if (baseInfo instanceof HelloChatController.HelloMessageInfo) {
return DemoApplication.instance().getString(R.string.welcome_tip);
}
return null;
}
}
// Register `TUIConversationControllerListener` with `TUIKitListenerManager` as soon as possible
so that the custom message abstract can be properly displayed in the conversion list.
TUIKitListenerManager.getInstance().addConversationListener(new HelloChatController.HelloConversa
tionController());
```

Customize Messages (iOS)

Last updated : 2021-08-18 10:35:54

In TUIChatController, every message is internally stored as TUIMessageCellData or a subclass object. When a user scrolls down the message list, the TUIMessageCellData is converted into TUIMessageCell for display.
You can set the delegate callback of TUIChatController to control specific TUIMessageCell instances to achieve message customization.
The hyperlink custom message in the above red box is an example. As TUIKit does not internally implement this effect, you only need to add two UILabels in container of TUIMessageCell to quickly implement this display effect. This document describes the realization process in detail.

Customizing Messages

Step 1: implement a custom `cellData` class

Customize a `cellData` class inherited from `TUIMessageCellData`, to be used for storing displayed texts and links.

```
@inerface MyCustomCellData : TUIMessageCellData
@property NSString *text;
@property NSString *link;
@end
```

TUIMessageCellData needs to calculate the size of the displayed content, so that TUIChatController can reserve sufficient space to display the message.

```
@implement MyCustomCellData : TMessageCellData
- (CGSize)contentSize
{
CGRect rect = [self.text boundingRectWithSize:CGSizeMake(300, MAXFLOAT) options:NSStringDrawingUs
esLineFragmentOrigin | NSStringDrawingUsesFontLeading attributes:@{ NSFontAttributeName : [UIFont
systemFontOfSize:15] } context:nil];
CGSize size = CGSizeMake(ceilf(rect.size.width)+1, ceilf(rect.size.height));

// Add bubble margins
size.height += 60;
size.width += 20;

return size;
```

```
}
@end
```

Step 2: implement a custom cell class

Customize a cell class inherited from TUIMessageCell.

```
@interface MyCustomCell : TUIMessageCell
@property UILabel *myTextLabel;
@property UILabel *myLinkLabel;
@end
```

In the implementation file, you need to create the myTextLabel and myLinkLabel objects and add them to the container.

```
@implementation MyCustomCell
- (instancetype)initWithStyle:(UITableViewCellStyle)style reuseIdentifier:(NSString *)reuseIdenti
fier
{
self = [super initWithStyle:style reuseIdentifier:reuseIdentifier];
if (self) {
_myTextLabel = [[UILabel alloc] init];
_myTextLabel.numberOfLines = 0;
_myTextLabel.font = [UIFont systemFontOfSize:15];
[self.container addSubview:_myTextLabel];

_myLinkLabel = [[UILabel alloc] initWithFrame:CGRectZero];
_myLinkLabel.text = @"View details>>";
_myLinkLabel.font = [UIFont systemFontOfSize:15];
_myLinkLabel.textColor = [UIColor blueColor];
[self.container addSubview:_myLinkLabel];

self.container.backgroundColor = [UIColor whiteColor];
[self.container.layer setMasksToBounds:YES];
[self.container.layer setBorderColor:[UIColor lightGrayColor].CGColor];
[self.container.layer setBorderWidth:1];
[self.container.layer setCornerRadius:5];
}
return self;
}
- (void)fillWithData:(MyCustomCellData *)data;
{
[super fillWithData:data];
self.customData = data;
self.myTextLabel.text = data.text;
}
```

```
- (void)layoutSubviews
{
[super layoutSubviews];
self.myTextLabel.mm_top(10).mm_left(10).mm_flexToRight(10).mm_flexToBottom(50);
self.myLinkLabel.mm_sizeToFit().mm_left(10).mm_bottom(10);
}
@end
```

Step 3: register the TUIChatController callback

The purpose of registering the TUIChatController callback is to notify TUIChatController of how to display custom messages. Registering this callback needs to implement the following callbacks:

- When receiving a message, convert V2TIMMessage into TUIMessageCellData objects.
- Before display, convert TUIMessageCellData into TUIMessageCell objects to be used for ultimate display.

```
@implement MyChatController
- (id)init
{
self = [super init];
// Add a listener
[[TUIKitListenerManager sharedInstance] addChatControllerListener:self];
// Initialize
chat = [[TUIChatController alloc] initWithConversation:conversationData]; // conversationData is
the current conversation data, including groupID, userID, and so on, which can be obtained from t
he conversation list
[self addChildViewController:chat]; // Add the chat interface internally
// Configure the navigation bar
...

return self;
}
// TChatController callback function
- (TUIMessageCellData *)chatController:(TUIChatController *)controller onNewMessage:(V2TIMMessage
*)msg
{
if (msg.elemType == V2TIM_ELEM_TYPE_CUSTOM) {
MyCustomCellData *cellData = [[MyCustomCellData alloc] initWithDirection:msg.isSelf ? MsgDirectio
nOutgoing : MsgDirectionIncoming];
cellData.text = @"View details>>";
cellData.link = @"https://cloud.tencent.com/product/im";
return cellData;
}
return nil
}
```

```
- (TUIMessageCell *)chatController:(TUIChatController *)controller onShowMessageData:(TUIMessageC
ellData *)data
{
if ([data isKindOfClass:[MyCustomCellData class]]) {
MyCustomCell *myCell = [[MyCustomCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdent
ifier:@"MyCell"];
[myCell fillWithData:(MyCustomCellData *)data];
return myCell;
}
return nil
}
@end
```

Sending Custom Messages

TUIChatController provides an API for sending messages. Users can use code to control message sending. The type of a custom message must be inherited from TUIMessageCellData. For example, to send a text message, you can create a TUITextMessageCellData object.
To send custom data, you need to initialize the innerMessage attribute. Please refer to the following code:

```
MyCustomCellData *cellData = [[MyCustomCellData alloc] initWithDirection:MsgDirectionOutgoing];
cellData.innerMessage = [[V2TIMManager sharedInstance] createCustomMessage:data]; // Data is cust
om binary data
[chatController sendMessage:cellData];
```

General Integration (No UI Library)
Quick Import to Projects
SDK Integration (Android)

Last updated : 2021-10-21 14:48:00

This document describes how to quickly integrate the Tencent Cloud IM SDK to your projects. To configure and integrate the SDK, follow these steps.

Environment Requirements

- JDK 1.6.
- Android 4.1 (SDK API level 16) or above

Integrating the SDK (AAR)

You can use Gradle to automatically load the AAR file or manually download the AAR file and import it into your project.

Method 1: automatic loading (AAR)

Because the JCenter service will be deprecated, subsequent IM SDKs will be published to the Maven Central repository. You can configure Gradle to automatically download updates.
Use Android Studio to open your project and modify the `app/build.gradle` file in three simple steps to integrate the SDK to your project, as shown below:

- **Step 1: add SDK dependencies**

  Find `build.gradle` of the app and add `mavenCentral()` dependencies to `repositories`.

```
repositories {
google()
jcenter()
// Add the `mavenCentral` repository.
mavenCentral()
}
```

Add the IM SDK dependencies to `dependencies`.
If the IM SDK basic edition is used, add the following dependencies:

```
dependencies {
api 'com.tencent.imsdk:imsdk:version number'
}
```

If the IM SDK enhanced edition is used, add the following dependencies:

```
dependencies {
api 'com.tencent.imsdk:imsdk-plus:Version number'
}
```

Note :
Replace `version number` with the actual version number of the SDK. You are advised to use the latest version.
Take the version number `5.4.666` as an example:

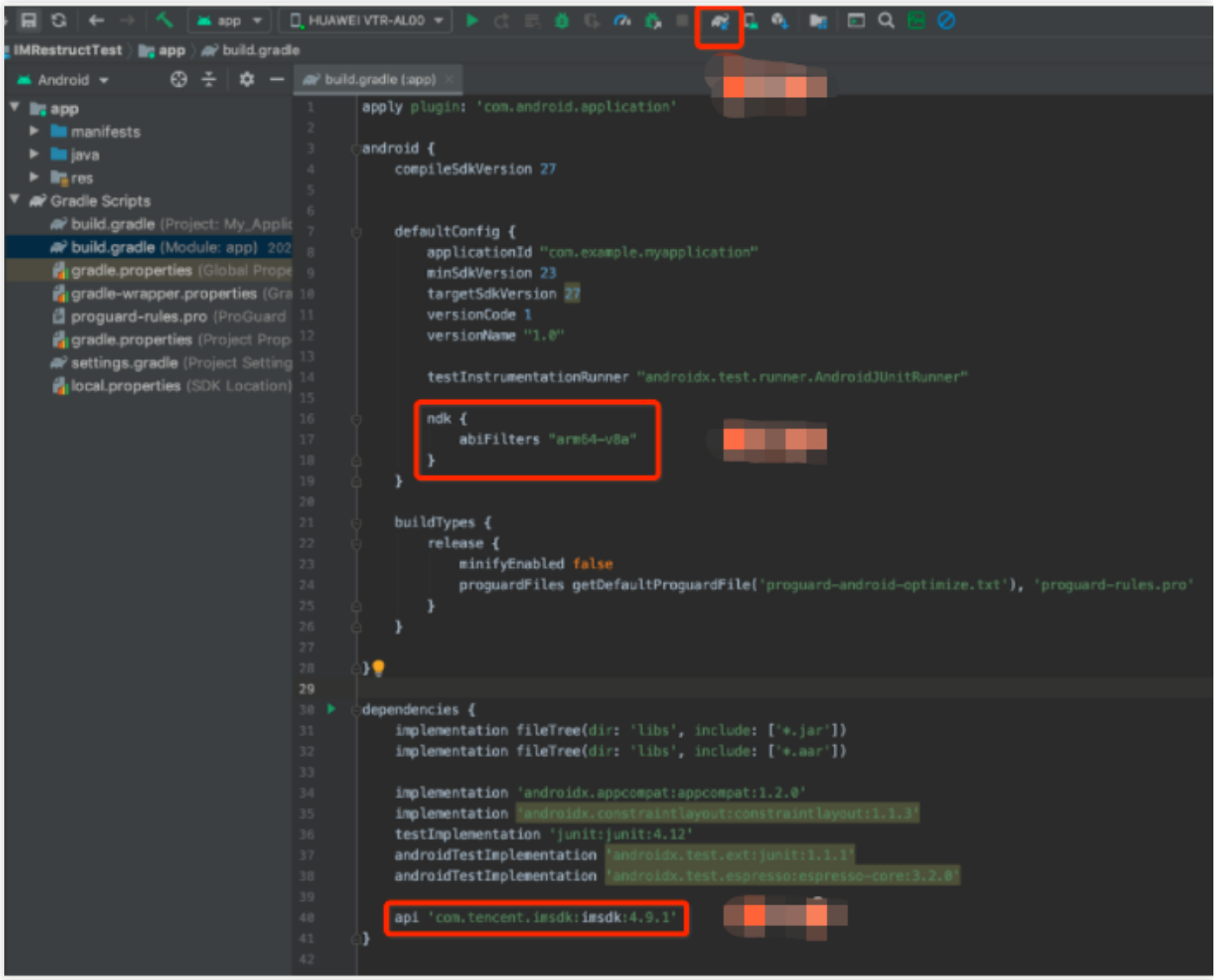
```
dependencies {
api 'com.tencent.imsdk:imsdk-plus:5.4.666'
}
```

- **Step 2: specify the app architecture**
  In `defaultConfig`, specify the CPU architecture used by the app (armeabi-v7a, arm64-v8a, x86, and x86_64 are supported starting from IM SDK v4.3.118).

```
defaultConfig {
ndk {
abiFilters "arm64-v8a"
}
}
```

- **Step 3: sync the SDK**
  Click the `Sync` icon. If the connection to JCenter is normal, the SDK will be automatically

downloaded and integrated to your project.

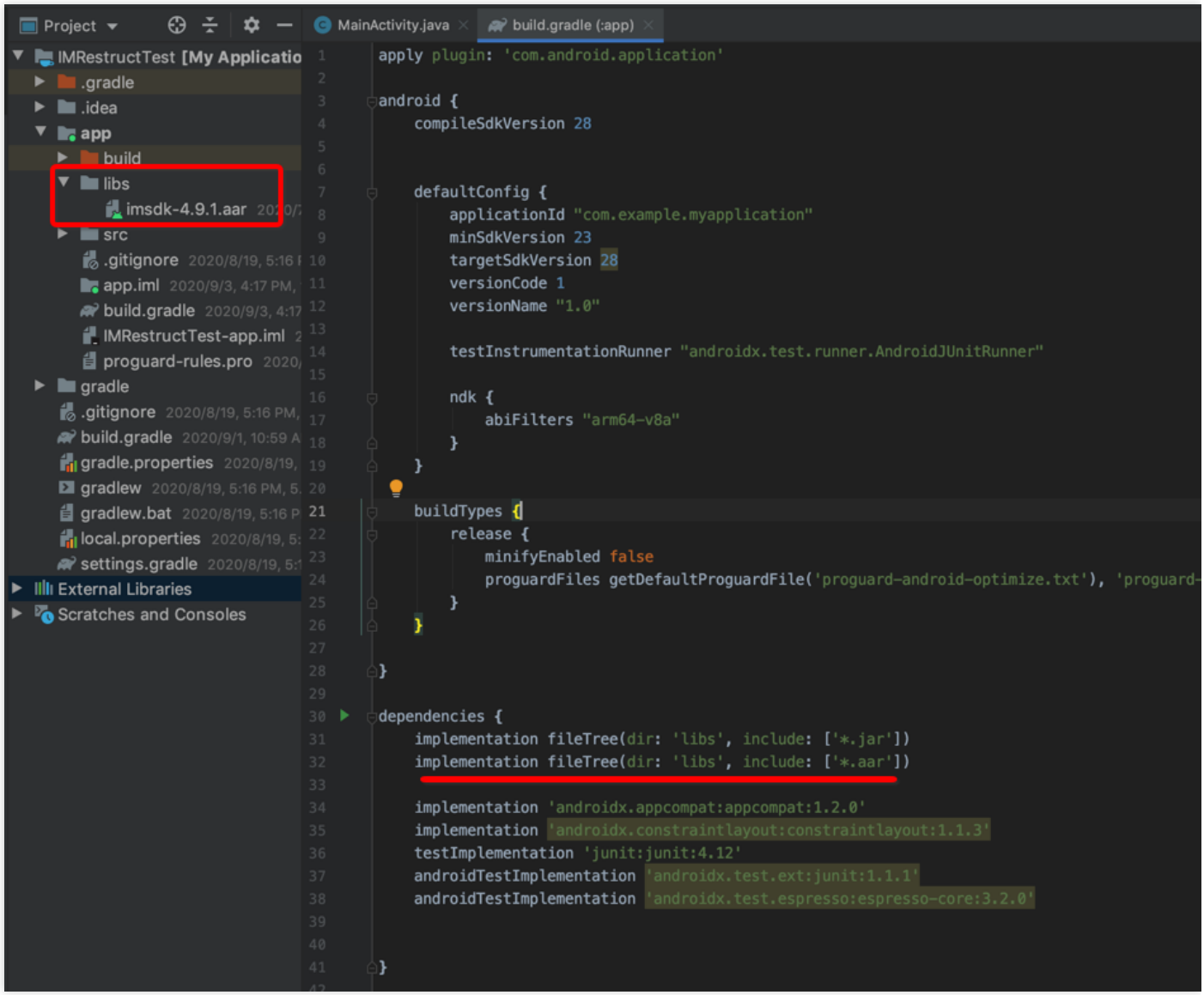


Method 2: manual download (AAR)

If JCenter cannot be accessed, you can manually download the SDK and integrate it into your project:

- **Step 1: download the IM SDK**
  Download the latest version of the IM SDK from GitHub.

- **Step 2: copy the IM SDK to the project directory**
  Copy the downloaded AAR file to the **/libs** directory of the project.



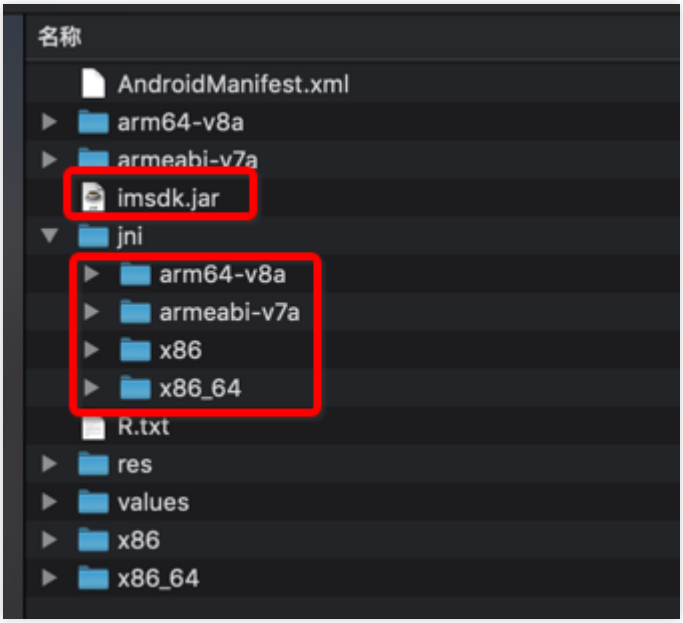
- **Step 3: specify the architecture used by the app and compile and run the architecture**
  In the `defaultConfig` of `app/build.gradle`, specify the CPU architecture used by the app (armeabi-v7a, arm64-v8a, x86, and x86_64 are supported starting from IM SDK v4.3.118).

```
defaultConfig {
ndk {
abiFilters "arm64-v8a"
}
}
```

Integrating SDK

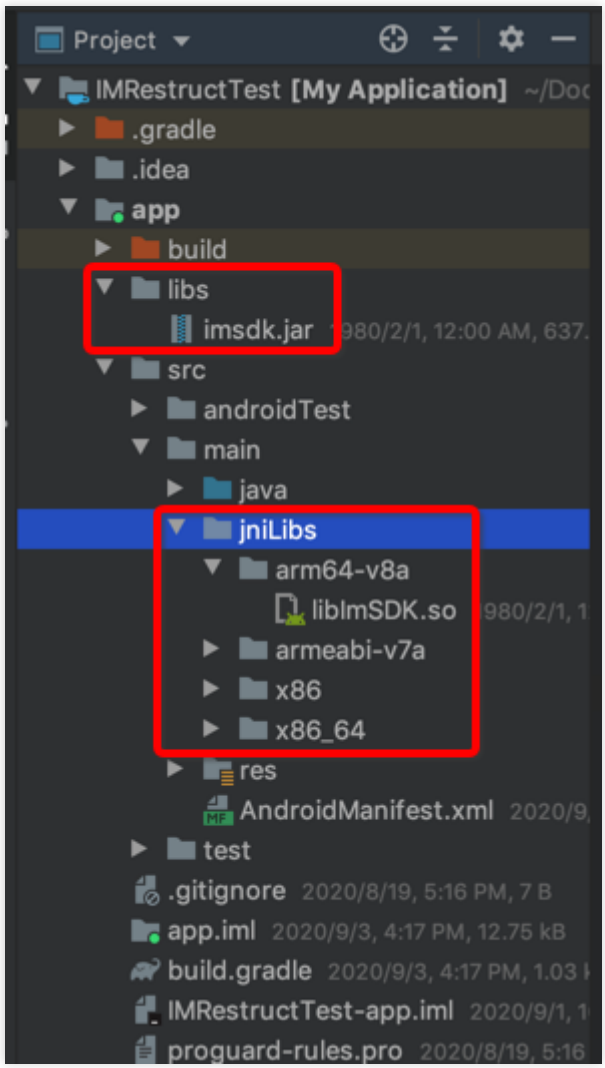
If you do not want to integrate the AAR library, you can integrate the IM SDK by importing the JAR and SO libraries.

- **Step 1: download and decompress the IM SDK**
  Download the latest version of the AAR file from GitHub and decompress it. The extracted folder contains a JAR file and an SO subfolder. Rename **classes.jar** to **imsdk.jar**.



- **Step 2: copy the SDK files to the project directory**
  Copy the renamed JAR file and SO files of different architectures to the default loading directories

of Android Studio.



- **Step 3: specify the architecture used by the app and compile and run the architecture**
  In the `defaultConfig` of `app/build.gradle`, specify the CPU architecture used by the app (armeabi-v7a, arm64-v8a, x86, and x86_64 are supported starting from IM SDK v4.3.118).

```
defaultConfig {
ndk {
abiFilters "arm64-v8a"
}
}
```

Configuring App Permissions

To configure app permissions in `AndroidManifest.xml`, the IM SDK requires the following permissions:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
```

Configuring Obfuscation Rules

In the `proguard-rules.pro` file, add the IM SDK classes to the "do not obfuscate" list.

```
-keep class com.tencent.imsdk.** { *; }
```

SDK Integration (iOS)

Last updated : 2021-09-02 11:41:20

This document describes how to quickly integrate the Tencent Cloud IM SDK (iOS) to your projects. To configure and integrate the SDK, follow these steps.

Environment Requirements

- Xcode 9.0+.
- iPhone or iPad on iOS 8.0 or above.
- Your project has a valid developer signature.

Integrating the IM SDK

You can either automatically integrate the IM SDK using CocoaPods, or manually download the SDK and import it to your current project.

Automatic loading using CocoaPods

1. Install CocoaPods

Enter the following command in the terminal window (you need to install the Ruby environment on your macOS in advance):

```
sudo gem install cocoapods
```

2. Create a Podfile

Go to the path where the project is located and run the following command. Then, a Podfile will appear under the project path.

```
pod init
```

3. Edit the Podfile

If you are using the SDK basic edition, edit the Podfile as follows:

```
platform :ios, '8.0'
source 'https://github.com/CocoaPods/Specs.git'
```

```
target 'App' do
pod 'TXIMSDK_iOS'
end
```

If you are using the SDK enhanced edition, edit the Podfile as follows:

```
platform :ios, '8.0'
source 'https://github.com/CocoaPods/Specs.git'
target 'App' do
pod 'TXIMSDK_Plus_iOS'
end
```

If you are using the SDK bitcode enhanced edition, edit the Podfile as follows:

```
platform :ios, '8.0'
source 'https://github.com/CocoaPods/Specs.git'
target 'App' do
pod 'TXIMSDK_Plus_iOS_Bitcode'
end
```

If you are using the SDK XCFramework enhanced edition, edit the Podfile as follows:

```
platform :ios, '8.0'
source 'https://github.com/CocoaPods/Specs.git'
target 'App' do
pod 'TXIMSDK_Plus_iOS_XCFramework'
end
```

If you are using the SDK XCFramework enhanced edition (bitcode supported), edit the Podfile as follows:

```
platform :ios, '8.0'
source 'https://github.com/CocoaPods/Specs.git'
target 'App' do
pod 'TXIMSDK_Plus_iOS_Bitcode_XCFramework'
end
```

4. Install the SDK or update the local repository.

Run the following command in the terminal window to update the local library file and install the TXIMSDK:

```
pod install
```

Or, run the following command to update the local repository:

```
pod update
```

After the pod command is executed, an .xcworkspace project file integrated with the SDK will be generated. Double-click this file to open it.

Note :
If the pod search fails, you are advised to update the local repo cache of the pod by running the following commands:

```
>pod setup
pod repo update
rm ~/Library/Caches/CocoaPods/search_index.json
```

Manual integration

1. Download the SDK

Download the latest SDK version from GitHub:

- `ImSDK.framework` and `ImSDK_Plus.framework` are the core dynamic library files of the IM SDK.

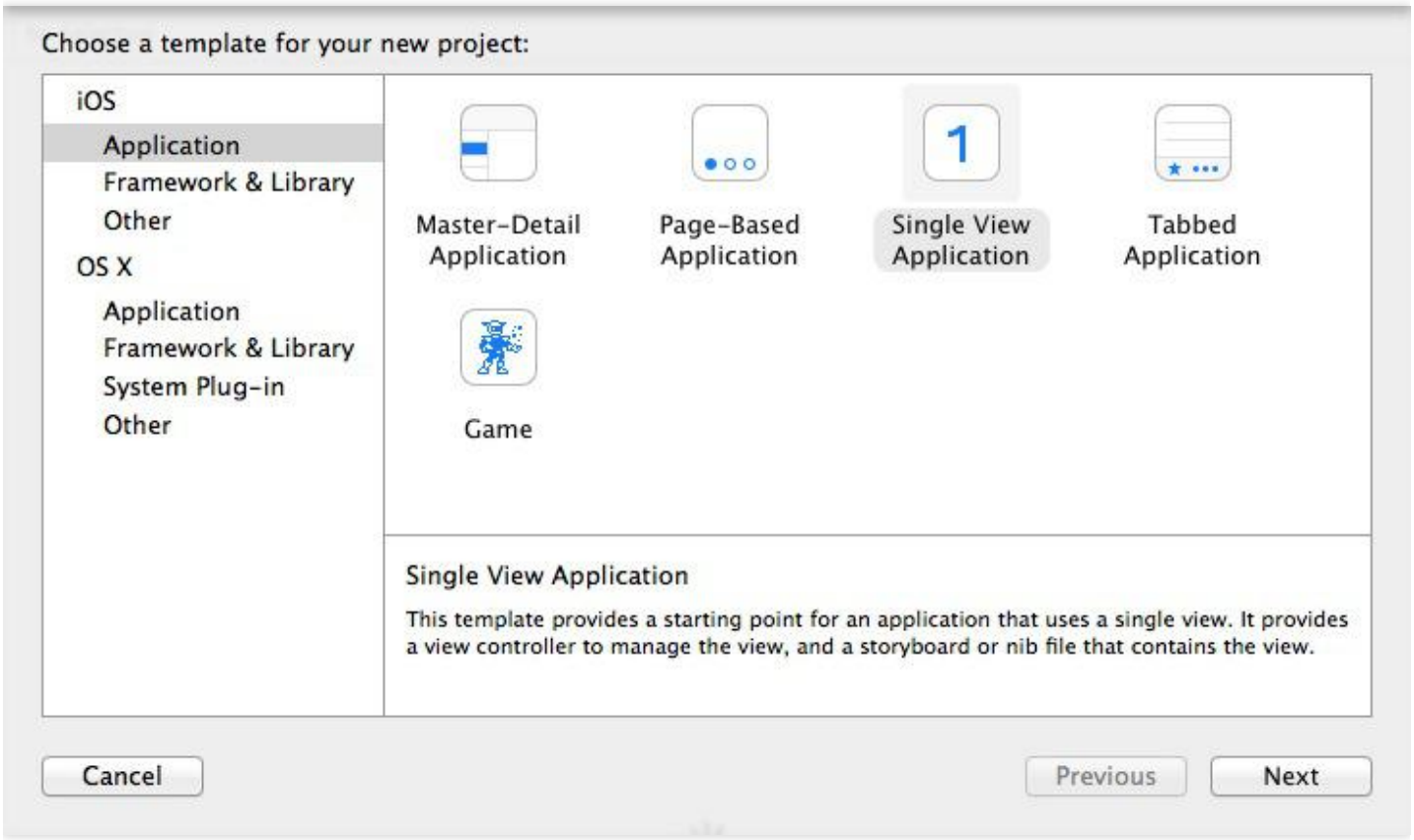
Package Name	Description
ImSDK.framework	IM SDK basic edition
ImSDK_Plus.framework	IM SDK enhanced edition

- TXLiteAVSDK_UGC.framework is the Tencent Cloud UGSV SDK that implements short-video sending and receiving in IM. It is an optional component.

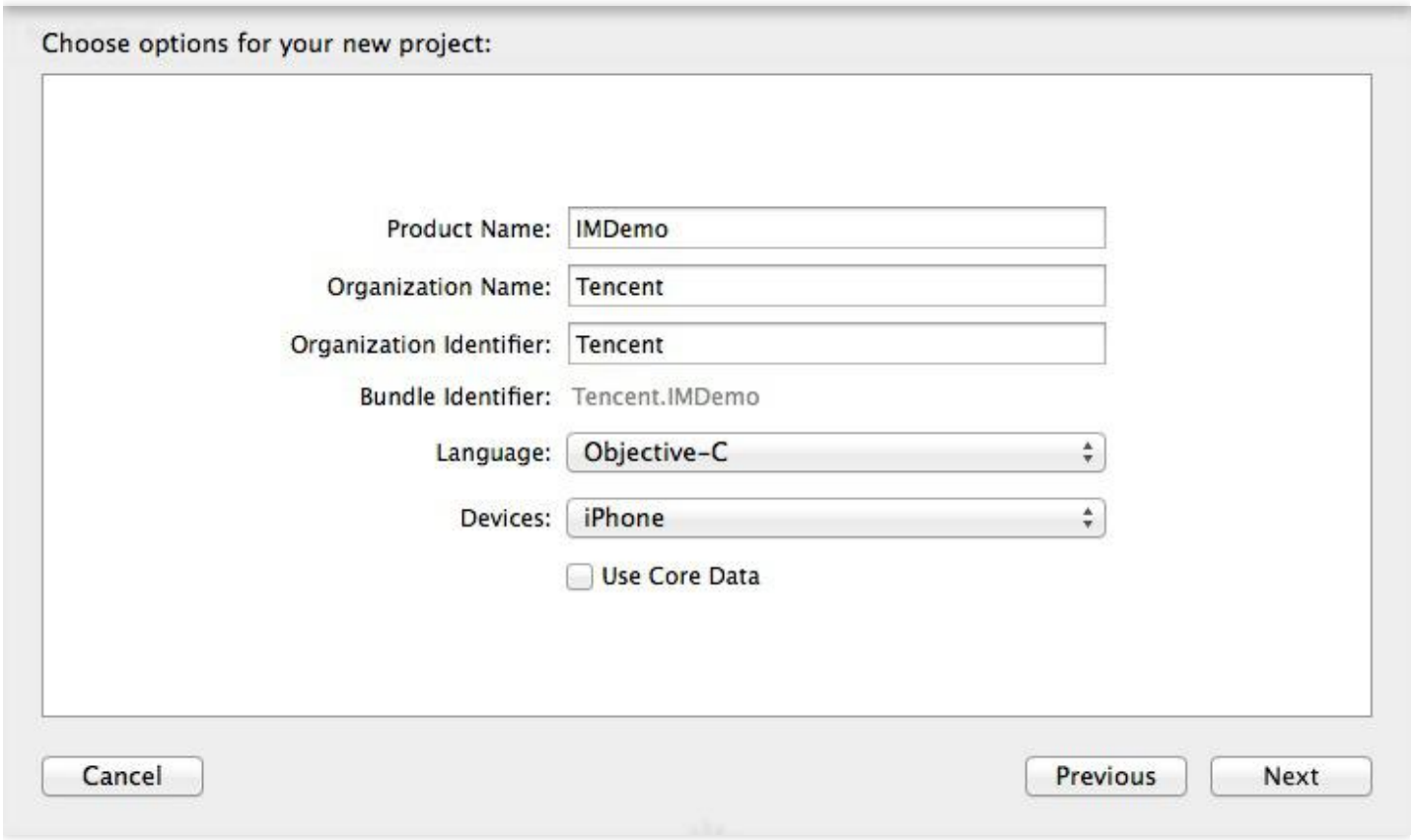
Package Name	Description	Feature
TXLiteAVSDK_UGC.framework	Extension package for recording and editing short videos	This package provides short video recording and editing features. For more information, see UGSV SDK Documentation.

2. Create a project

Create a project.

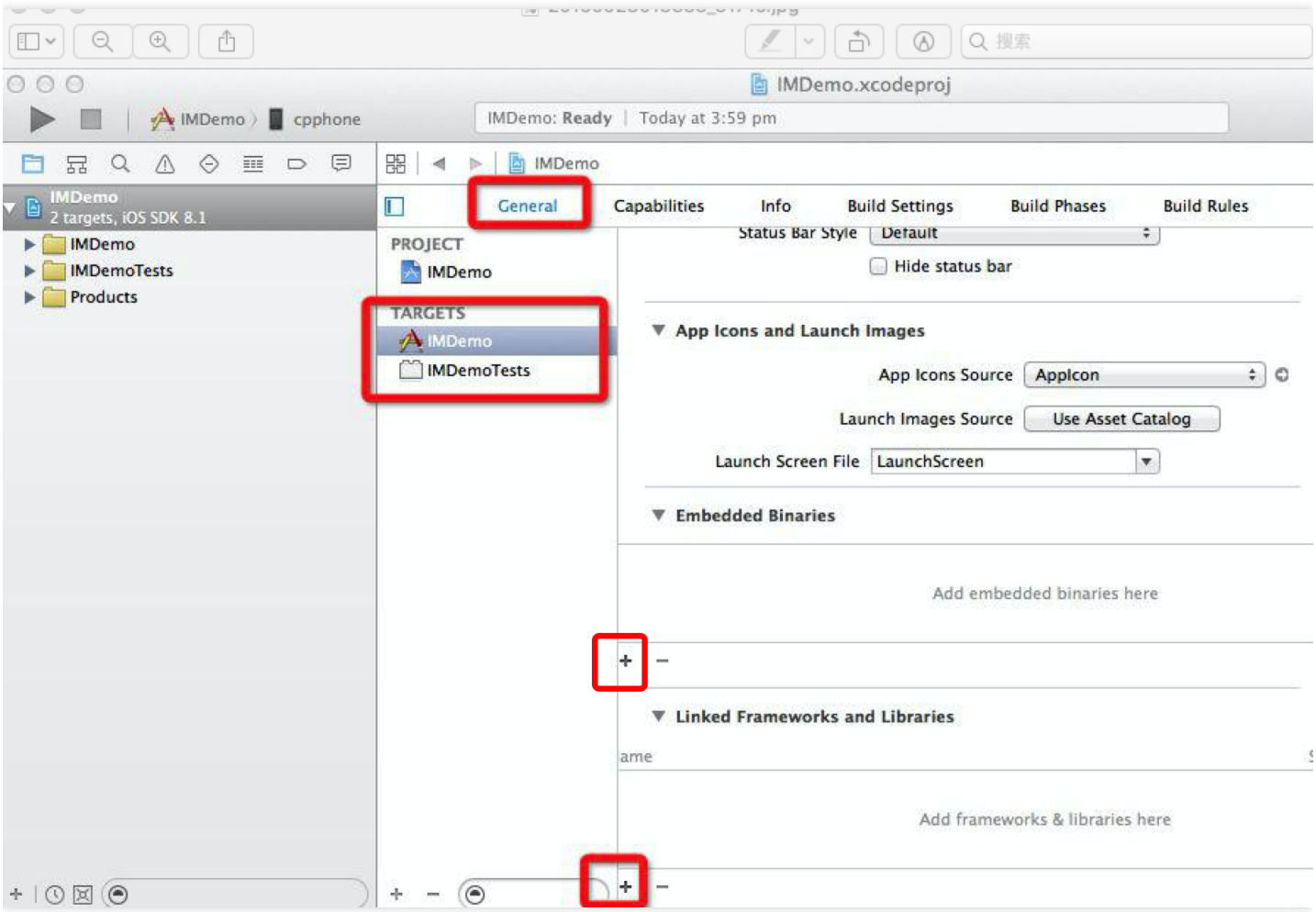


Enter a project name, for example, IMDemo.



3. Integrate the IM SDK

Add the dependency library: select **Target** for IMDemo. On the **General** panel, add the dependency library under **Embedded Binaries** and **Linked Frameworks and Libraries**. If you use the SDK basic edition, select **ImSDK.framework**. If you use the SDK enhanced edition, select **ImSDK_Plus.framework**.



Set link parameters: add `-ObjC` in **Build Setting -> Other Linker Flags.**

> Note :
> For manual integration, you need to change `ImSDK.framework` to `Embed&Sing` in **Target -> General -> Frameworks -> Libraries and Embedded Content**.

Referencing the IM SDK

There are two ways to use the SDK in your project code.

Method 1

Choose **Xcode -> Build Setting -> Header Search Paths**, and set the SDK header file path. In files that require the SDK API, reference the corresponding header file.

- If you use the SDK basic edition, reference the header file as follows:

```
#import "ImSDK.h"
```

- If you use the SDK enhanced edition, reference the header file as follows:

```
#import "ImSDK_Plus.h"
```

Method 2

In files that require the SDK API, reference the corresponding header file.

- If you use the SDK basic edition, reference the header file as follows:

```
#import <ImSDK/ImSDK.h>
```

- If you use the SDK enhanced edition, reference the header file as follows:

```
#import <ImSDK_Plus/ImSDK_Plus.h>
```

SDK Integration (Mac)

Last updated : 2021-03-05 16:55:30

This document describes how to quickly integrate the Tencent Cloud IM SDK (Mac) into your projects. To configure and integrate the SDK, follow these steps.

Development Environment Requirements

- Xcode 9.0+.
- Mac device running OS X 10.10 or later.
- The project has been configured with a valid developer signature.

Integrating the IM SDK

You can either automatically integrate the IM SDK by using CocoaPods, or manually download the SDK and import it to your current project.

Automatically loading CocoaPods

1. Install CocoaPods

Run the following command in a terminal window (you need to install the Ruby environment on your Mac device in advance):

```
sudo gem install cocoapods
```

2. Create a Podfile

Navigate to the path where the project is located and run the following command. Then, a Podfile will appear under the project path.

```
pod init
```

3. Edit the Podfile

Edit the Podfile as follows:

```
platform :macos, '10.10'
source 'https://github.com/CocoaPods/Specs.git'
```

```
target 'mac_test' do
pod 'TXIMSDK_Mac'
end
```

4. Update and install the SDK

Run the following command in a terminal window to update the local library file and install TXIMSDK_Mac:

```
pod install
```

Alternatively, run the following command to update the local library version:

```
pod update
```

After the pod command is executed, a project file integrated with the SDK and suffixed .xcworkspace will be generated. Double-click the file to open it.

Manual integration

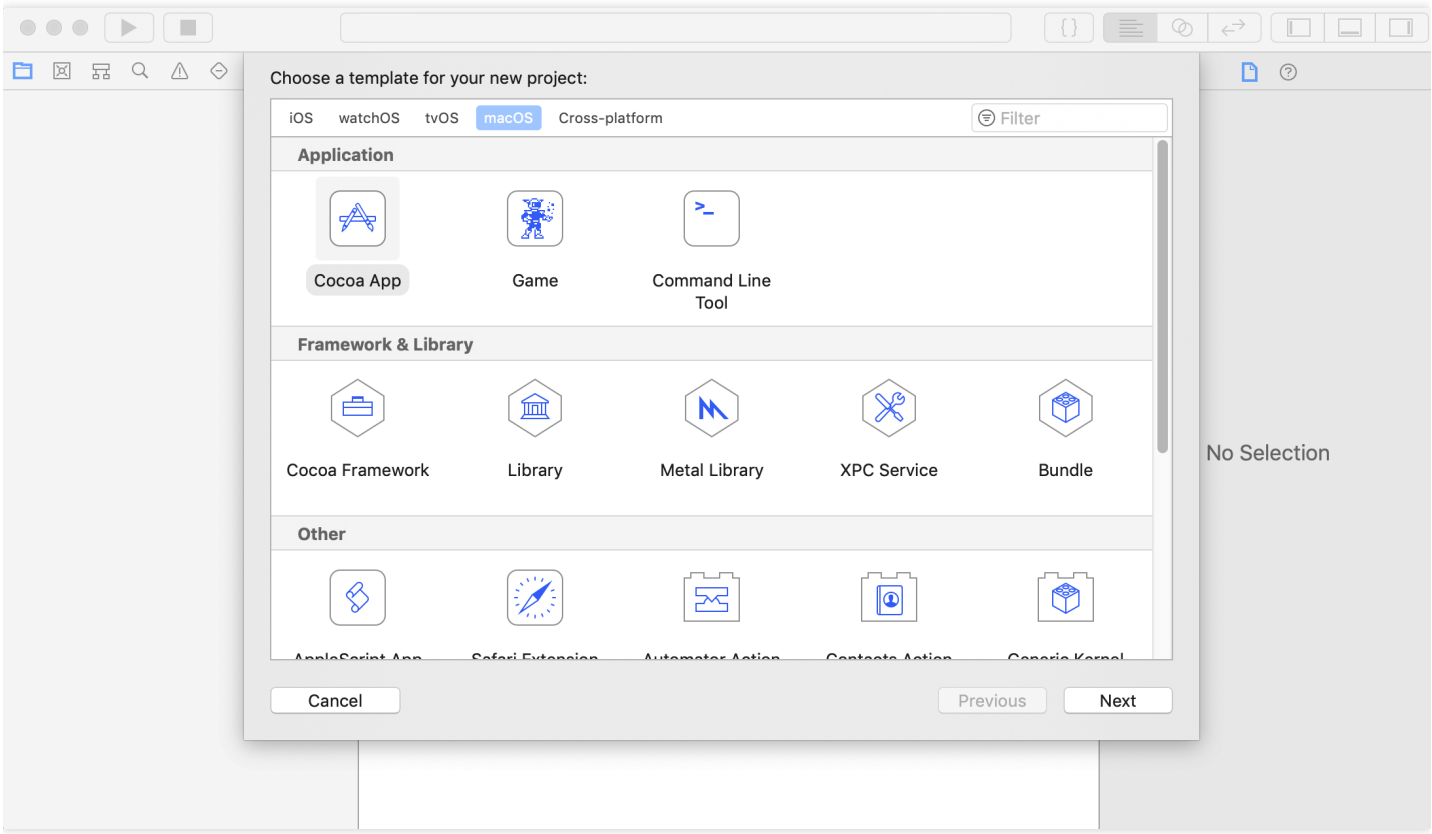
1. Obtain the SDK download URL from Github:

- ImSDKForMac.framework is the core dynamic library file of the IM SDK.

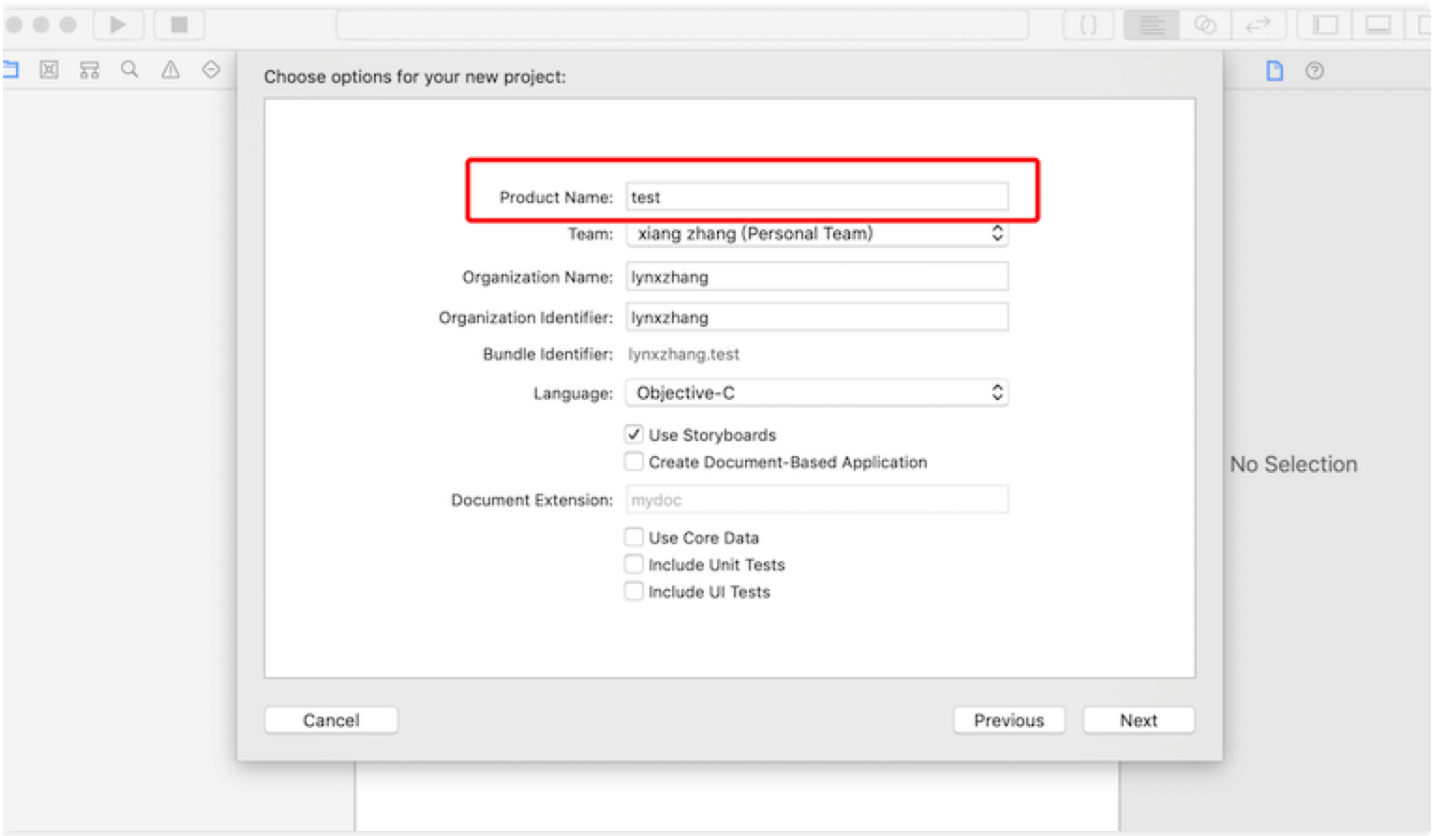
Pack Name	Description	ipa Increment
ImSDKForMac.framework	IM feature pack	1.4 MB

2. Create a project

Create a project:

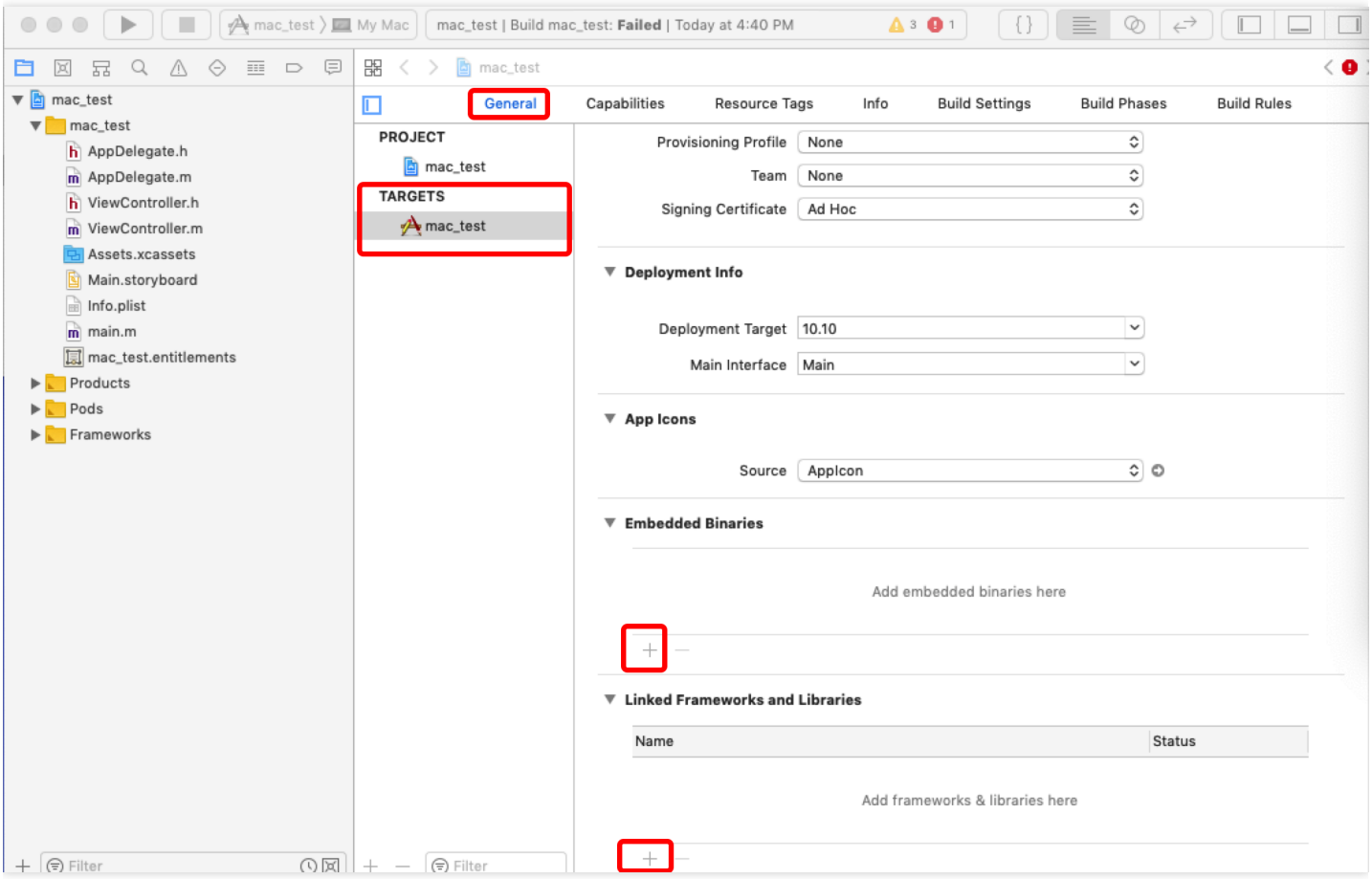


Enter a project name:



2. Integrate the IM SDK

Add the dependent library: select the **Target** of Demo. On the **General** panel, add the dependent library under **Embedded Binaries** and **Linked Frameworks and Libraries**.



Add the dependent library:

```
ImSDKForMac.framework
```

> **⚠ Note :**
>
> You need to add `-ObjC` in **Build Setting** > **Other Linker Flags**.

Referencing the IM SDK

Use the SDK in project code in two ways:

- Method 1: Navigate to **Xcode** > **Build Setting** > **Header Search Paths**, and set the ImSDKForMac.framework/Headers path. In files that require the SDK API, directly reference the header file "ImSDK.h".

```
#import "ImSDK.h"
```

- Method 2: in the files that require the SDK API, import the header file <ImSDKForMac/ImSDK.h>.

```
#import <ImSDKForMac/ImSDK.h>
```

SDK Integration (Web & Mini Program)

Last updated : 2021-10-14 11:09:01

This document describes how to quickly integrate the Tencent Cloud IM SDK into your web or Mini Program project.

- You can integrate the IM SDK into your web project by using npm (recommended) or script.
- You can integrate the IM SDK into your Mini Program project by using npm.
- You can integrate the SDK upload plugin for faster and safer upload of rich text message resources. For more information, see SDK Upload Plugin Integration (Web & Mini Program).

Preparations

- You have already created an IM app and obtained the `SDKAppID`.
- You have obtained the key information.

Relevant Documents

- Demo Quick Start
- Running the IM SDK (Mini Program) TUIKit
- Running the IM SDK (Web) Demo
- SDK Upload Plugin Integration (Web & Mini Program)

Integrating SDK

Integrating via npm (recommended)

Use npm to install appropriate IM SDK dependencies in your project.

Web project

```
// IM Web SDK
// SDK v2.11.2 and later versions support WebSocket, and you are advised to integrate such SDK versions. SDK v2.10.2 and earlier versions use HTTP.
npm install tim-js-sdk --save
// The Tencent Cloud IM upload plugin is required to send messages such as images and files.
npm install tim-upload-plugin --save
```

Note :

If a problem occurs during dependency synchronization, change the npm source and try again.

```
>// Change cnpm source
>npm config set registry http://r.cnpmjs.org/
>
```

Import the module into the project script.

```
// SDK v2.11.2 and later versions support WebSocket, and you are advised to integrate such SDK versions. SDK v2.10.2 and earlier versions use HTTP.
import TIM from 'tim-js-sdk';
import TIMUploadPlugin from 'tim-upload-plugin';
let options = {
SDKAppID: 0 // Replace `0` with the `SDKAppID` of your IM app during access.
};
// Create an SDK instance. The `TIM.create()` method returns the same instance for the same `SDKAppID`.
let tim = TIM.create(options); // The SDK instance is usually represented by `tim`.
// Set the SDK log level for output. For more information on each level, see <a href="https://web.sdk.qcloud.com/im/doc/zh-cn//SDK.html#setLogLevel">setLogLevel API Description</a>.
tim.setLogLevel(0); // Common level. You are advised to use this level during connection as it covers more logs.
// tim.setLogLevel(1); // Release level, at which the SDK outputs important information. You are advised to use this log level in a production environment.
// Register the Tencent Cloud IM upload plugin.
tim.registerPlugin({'tim-upload-plugin': TIMUploadPlugin});
```

Mini Program project

```
// IM Mini Program SDK
// SDK v2.11.2 and later versions support WebSocket, and you are advised to integrate such SDK versions. SDK v2.10.2 and earlier versions use HTTP.
npm install tim-wx-sdk --save
// The Tencent Cloud IM upload plugin is required to send messages such as images and files.
npm install tim-upload-plugin --save
```

Note :

If a problem occurs during dependency synchronization, change the npm source and try again.

```
>// Change cnpm source
>npm config set registry http://r.cnpmjs.org/
>
```

Import modules to the project script and initialize the modules.

```
// SDK v2.11.2 and later versions support WebSocket, and you are advised to integrate such SDK versions. SDK v2.10.2 and earlier versions use HTTP.
import TIM from 'tim-wx-sdk';
import TIMUploadPlugin from 'tim-upload-plugin';
let options = {
SDKAppID: 0 // Replace `0` with the `SDKAppID` of your IM app during access.
};
// Create an SDK instance. The `TIM.create()` method returns the same instance for the same `SDKAppID`.
let tim = TIM.create(options); // The SDK instance is usually represented by `tim`.
// Set the SDK log level for output. For more information on each level, see <a href="https://web.sdk.qcloud.com/im/doc/zh-cn//SDK.html#setLogLevel">setLogLevel API Description</a>.
tim.setLogLevel(0); // Common level. You are advised to use this level during connection as it covers more logs.
// tim.setLogLevel(1); // Release level, at which the SDK outputs important information. You are advised to use this log level in a production environment.
// Register the Tencent Cloud IM upload plugin.
tim.registerPlugin({'tim-upload-plugin': TIMUploadPlugin});
```

For more information on how to initialize the SDK and use APIs, see SDK Initialization.

Integrating via script

Import the SDK to your project by using the script tag and initialize the SDK.

```
<!-- `tim-js.js` and `tim-upload-plugin.js` can be obtained at https://github.com/tencentyun/TIMSDK/tree/master/Web/Demo/sdk. -->
<script src="./tim-js.js"></script>
<script src="./tim-upload-plugin.js"></script>
<script>
var options = {
SDKAppID: 0 // Replace `0` with the `SDKAppID` of your IM app during access.
};
// Create an SDK instance. The `TIM.create()` method returns the same instance for the same `SDKAppID`.
var tim = TIM.create(options);
// Set the SDK log output level. For details on each level, see **setLogLevel API description**.
tim.setLogLevel(0); // Common level. You are advised to use this level during connection as it co
```

```
vers more logs.
// tim.setLogLevel(1); // Release level, at which the SDK outputs important information. You are advised to use this log level in a production environment.
// Register the Tencent Cloud IM upload plugin.
tim.registerPlugin({'tim-upload-plugin': TIMUploadPlugin});
// Next, you can use `tim` to bind events and build IM apps.
</script>
```

Note :
Set the SDK log output level. For more information on each level, see setLogLevel API Description.

For more information on how to initialize the SDK and use APIs, see SDK Initialization.

Relevant resources

- SDK Update Log
- SDK API Documentation
- FAQs
- IM Web Demo
- Download URL of Tencent Cloud IM Upload Plugin

FAQs

1. What should I do if I want to launch a Mini Program and deploy a production environment?
Configure domain names by navigating to **WeChat Official Accounts Platform** -> **Development** -> **Development Settings** -> **Server Domain Name**:

Add the following domain names to **request valid domain names**:

SDK v2.11.2 and later versions support WebSocket, and the following domain names must be added for them:

Domain Name	Description	Required
`wss://wss.im.qcloud.com`	Web IM service domain	Yes
`wss://wss.tim.qq.com`	Web IM service domain	Yes

Domain Name	Description	Required
`https://web.sdk.qcloud.com`	Web IM service domain	Yes
`https://webim.tim.qq.com`	Web IM service domain	Yes

SDK v2.10.2 and earlier versions use HTTP, and the following domain names must be added for them:

Domain Name	Description	Required
`https://webim.tim.qq.com`	Web IM service domain	Yes
`https://yun.tim.qq.com`	Web IM service domain	Yes
`https://events.tim.qq.com`	Web IM service domain	Yes
`https://grouptalk.c2c.qq.com`	Web IM service domain	Yes
`https://pingtas.qq.com`	Web IM statistical domain	Yes

Add the following domain name to the **uploadFile valid domain name**:

Domain Name	Description	Required
`https://cos.ap-shanghai.myqcloud.com`	File upload domain	Yes

Add the following domain name to **downloadFile valid domain name**:

Domain Name	Description	Required
`https://cos.ap-shanghai.myqcloud.com`	File download domain	Yes

SDK Upload Plugin Integration (Web & Mini Program)

Last updated : 2021-09-15 17:03:21

Overview

tim-upload-plugin is an upload plugin of Tencent Cloud IM. It upload resources in a way based on the Tencent Cloud COS pre-signed URL method. When integrating Tencent Cloud IM, you can use tim-upload-plugin instead of "cos-js-sdk" or "cos-wx-sdk" to upload resource. This plugin not only helps improve the security of application data, but also has benefits such as fast upload speed, small size, and support for Mini Programs on multiple platforms.

Advantages

- **Improved security of application data**

  You receive a new pre-signed URL every time you upload a resource file. The pre-signed URL is bound with the current file type and file information. In addition, pre-signed URLs have an expiration time and cannot be used after they expire.

- **10% to 50% increase in average upload speed**
  - The average speed of uploading a resource file within 5 MB is 50% faster than that of "cos-js-sdk" and "cos-wx-sdk".
  - The average speed of uploading a resource file between 5 MB and 12 MB is 30% faster than that of "cos-js-sdk" and "cos-wx-sdk".

- **Support for Mini Programs on multiple platforms**

  "tim-upload-plugin" provides better compatibility with Mini Programs on different platforms. Currently, it can be used when you are integrating Tencent Cloud IM into WeChat, QQ, Baidu, Toutiao, and Alipay Mini Programs, while "cos-wx-sdk" only supports integration into WeChat Mini Programs.

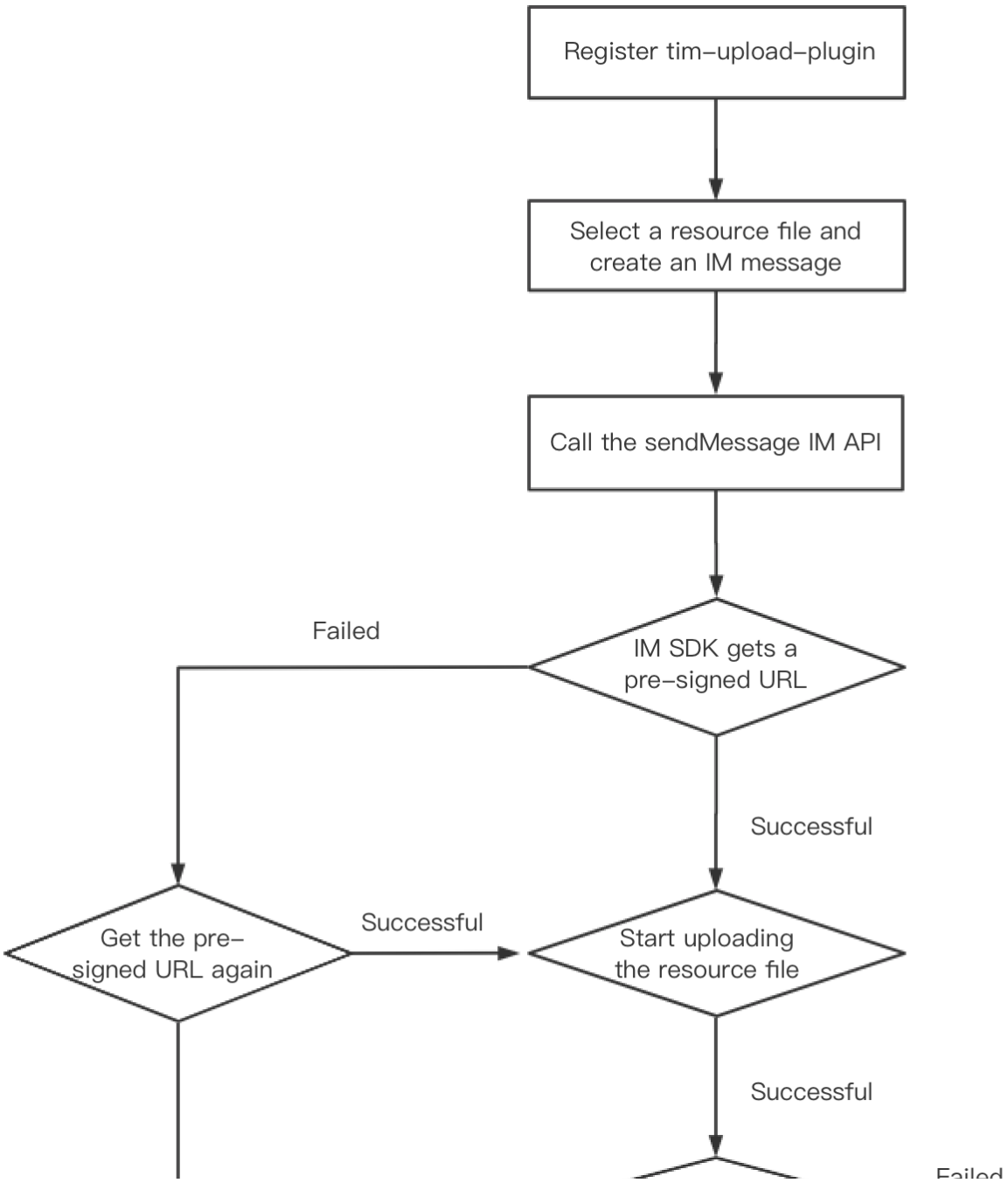
- **Support for various file formats**

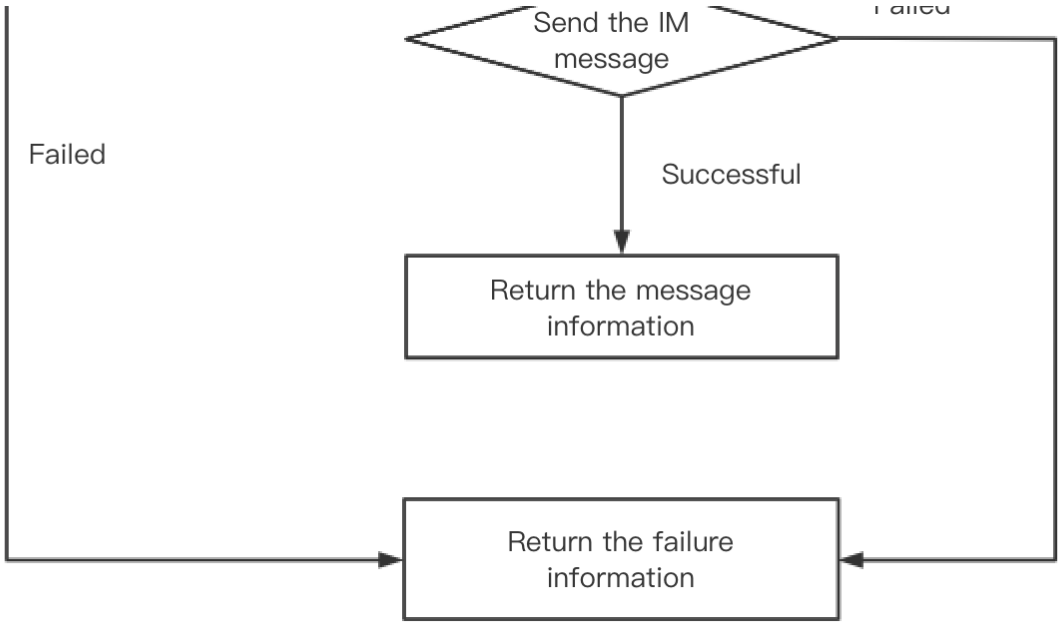
Currently, it supports JPG, JPEG, PNG, BMP, and GIF images, MP4 videos, audio files, as well as Word, Excel, and PDF files.

- **Lightweight**

  Its size is less than 10 KB. Compared with "cos-js-sdk" (1.8 MB) and "cos-wx-sdk" (1.2 MB), "tim-upload-plugin" is 98% smaller in size, which helps reduce your application size.

How It Works





Notes:

- Before using "tim-upload-plugin", please upgrade "tim-js-sdk" or "tim-wx-sdk" to v2.9.2 or above.
- To use "tim-upload-plugin" on a Mini Program terminal, you need to configure the valid domain (https://cos.ap-shanghai.myqcloud.com) of `uploadFile` and `downloadFile` in the Mini Program console.
- Do not register "tim-upload-plugin" and "cos-js-sdk" or "cos-wx-sdk" at the same time. IM SDK will check "tim-upload-plugin" first.
- Currently, "tim-upload-plugin" cannot be used in the Node.js environment.

Access

Before accessing tim-upload-plugin, you need to upgrade your Tencent Cloud IM SDK to v2.9.2 or above.

1. Access via npm

```
// Download the dependency.
npm i tim-upload-plugin --save
// To integrate "tim-upload-plugin", your "tim-js-sdk" or "tim-wx-sdk" version should be v2.9.2 o
r above.
npm i tim-js-sdk@latest --save // Used in web environment.
// npm i tim-wx-sdk@latest --save // Used in Mini Program environment.
```

```
// Import the modules to the project script and initialize it.
import TIM from 'tim-js-sdk' // Used in web environment.
// import TIM from 'tim-wx-sdk'; // Used in Mini Program environment.
import TIMUploadPlugin from 'tim-upload-plugin';
let options = {
SDKAppID: 0 // Replace `0` with the `SDKAppID` of your IM app during access.
};
// Create an SDK instance. The `TIM.create()` method returns the same instance for the same `SDKA
ppID`.
let tim = TIM.create(options); // The SDK instance is usually represented by `tim`.
// Set the SDK log output level. For details on each level, see **setLogLevel API description**.
tim.setLogLevel(0); // Common level. You are advised to use this level during access as it covers
more logs.
// tim.setLogLevel(1); // Release level, at which the SDK outputs important information. You are
advised to use this log level in a production environment.
// Register the Tencent Cloud IM upload plugin.
tim.registerPlugin({'tim-upload-plugin': TIMUploadPlugin});
```

2. Access via script

```
<!-- `tim-js.js` and `tim-upload-plugin.js` can be obtained at https://github.com/tencentyun/TIMS
DK/tree/master/Web/Demo/sdk. -->
<script src='./tim-js.js'></script>
<script src='./tim-upload-plugin.js'></script>
<script>
let options = {
SDKAppID: 0 // Replace `0` with the `SDKAppID` of your IM app during access.
};
// Create an SDK instance. The `TIM.create()` method returns the same instance for the same `SDKA
ppID`.
let tim = TIM.create(options);
// Set the SDK log output level. For details on each level, see **setLogLevel API description**.
tim.setLogLevel(0); // Common level. You are advised to use this level during access as it covers
more logs.
// tim.setLogLevel(1); // Release level, at which the SDK outputs important information. You are
advised to use this log level in a production environment.
// Register the Tencent Cloud IM upload plugin.
tim.registerPlugin({'tim-upload-plugin': TIMUploadPlugin});
// Next, you can use `tim` to bind events and build IM apps.
</script>
```

SDK Integration (Windows)

Last updated : 2021-03-29 15:41:36

This document describes how to quickly integrate the Tencent Cloud IM SDK into your projects. Follow these steps to integrate the SDK easily.

Development Environment Requirements

- Operating system: Windows 7 or above.
- Development environment: Visual Studio 2010 or above. Visual Studio 2015 is recommended.

Integrating the IM SDK

The following describes how to integrate the SDK into a Visual Studio 2015 project by creating a simple MFC project.

Step 1. Download the IM SDK

Download the Windows IM SDK from GitHub.

Download and decompress the IM SDK file folder, which contains the following:

Directory Name	Description
includes	API header files
lib\Win32\Debug	**32-bit Debug mode**, using /MTd to link to library files
lib\Win32\Release	**32-bit Release mode**, using /MT to link to library files
lib\Win64\Debug	**64-bit Debug mode**, using /MTd to link to library files
lib\Win64\Release	**64-bit Release mode**, using /MT to link to library files

Step 2. Create a project

Open Visual Studio and create an MFC application named IMDemo.

For quick integration, on the **Application Type** page of the wizard, select the simple **Dialog-based type**. Do not change the defaults of other configuration items.

Step 3. Copy files

Copy the IM SDK files to the directory where `IMDemo.vcxproj` is located.

Step 4. Modify the project configuration

The IM SDK provides two compiled static libraries, **Debug** and **Release**, which require some special configurations. To do this, go to the IMDemo property page by clicking **Solution Resource Manager** > **Right-Click Menu of the IMDemo Project** > **Properties**.

Using **32-bit Debug mode** as an example, configure the project as follows:

1. Add the inclusion directory
   In **C/C++** > **General** > **Additional Inclusion Directories**, add the IM SDK header file directory `$(ProjectDir)ImSDK¥includes` .
2. Add the library directory
   In **Linker** > **General** > **Additional Library Directories**, add the IM SDK library directory `$(ProjectDir)ImSDK¥lib¥Win32¥Debug` .
3. Add the library file
   In **Linker** > **Input** > **Additional Dependencies**, add the IM SDK library file `imsdk.lib` .
4. Copy the DLL file to the execution directory
   In **Build Events** > **Pre-Built Events** > **Command Line**, enter and run `xcopy /E /Y "$(ProjectDir)ImSDK¥lib¥Win32¥Debug" "$(OutDir)"` to copy the dynamic library file `imsdk.dll` to the application generation directory.
5. Specify the encoding format of the source file
   The IM SDK header file uses the UTF-8 encoding format, whereas some compilers compile source files in the default system encoding format. This may lead to compilation failure. Set this parameter to instruct compilers to compile source files using UTF-8 encoding.
   In **C/C++** > **Command Line** > **Additional Options**, enter `/source-charset:.65001` .

Configure the **Release mode** as follows:

1. Add the library directory
   In **Linker** > **General** > **Additional Library Directories**, add the IM SDK library directory `$(ProjectDir)ImSDK¥lib¥Win32¥Release` .
2. Copy the DLL file to the execution directory
   In **Build Events** > **Pre-Built Events** > **Command Line**, enter and run `xcopy /E /Y "$(ProjectDir)ImSDK¥lib¥Win32¥Release" "$(OutDir)"` to copy the dynamic library file `imsdk.dll` to the application generation directory.

The settings for **64-bit Debug/Release** and **32-bit** are similar, but their library directories of the IM SDK are different, as shown below:

1. Add the library directory

- For the **Debug mode**: in **Linker** > **General** > **Additional Library Directories**, add the IM SDK library directory `$(ProjectDir)ImSDK¥lib¥Win64¥Debug` .
- For the **Release mode**: in **Linker** > **General** > **Additional Library Directories**, add the IM SDK library directory `$(ProjectDir)ImSDK¥lib¥Win64¥Release` .

2. Copy the DLL file to the execution directory

- For the **Debug mode**: in **Build Events** > **Pre-Built Events** > **Command Line**, enter and run `xcopy /E /Y "$(ProjectDir)ImSDK¥lib¥Win64¥Debug" "$(OutDir)"` to copy the dynamic library file `imsdk.dll` to the application generation directory.
- For the **Release mode**: in **Build Events** > **Pre-Built Events** > **Command Line**, enter and run `xcopy /E /Y "$(ProjectDir)ImSDK¥lib¥Win64¥Release" "$(OutDir)"` to copy the dynamic library file `imsdk.dll` to the application generation directory.

Step 5. Print the IM SDK version number

- In the file `IMDemo.cpp` , add the header file:

```
#include "TIMCloud.h"
```

- In the function `CIMDemoDlg::OnInitDialog` , add the following test code:

```
std::string version = TIMGetSDKVersion();
CString szText;
szText.Format(L"SDK version: %hs", version.c_str());
CWnd* pStatic = GetDlgItem(IDC_STATIC);
pStatic->SetWindowTextW(szText);
```

- Press **F5** to run the code and print the IM SDK version number.

FAQs

- If the following error occurs, check whether the directory of the IM SDK header file has been correctly added according to the preceding project configuration:

```
fatal error C1083: unable to open the header file: "TIMCloud.h": No such file or directory
```

- If the following error occurs, check whether the IM SDK library directory and library file have been correctly added according to the preceding project configuration:

```
LINK : fatal error LNK1104: unable to open the file "imsdk.lib"
```

```
error LNK2019: unable to parse the external symbol `__imp__TIMGetSDKVersion`; this symbol is r
eferenced in `protected: virtual int __thiscall CIMDemoDlg::OnInitDialog(void)" (?OnInitDialog
@CIMDemoDlg@@MAEHXZ)`
```

- If the following error occurs, check whether the DLL file of the IM SDK has been copied to the execution directory according to the preceding project configuration.

Initialization and Login
Initialization and Login (Android)

Last updated : 2021-10-15 14:53:14

Initialization

V2TIMManager is a core class and also an entry class of the IM SDK. It implements features such as IM SDK initialization and login, message sending/receiving, group creation, and group leaving. To complete initialization, call the initSDK API.

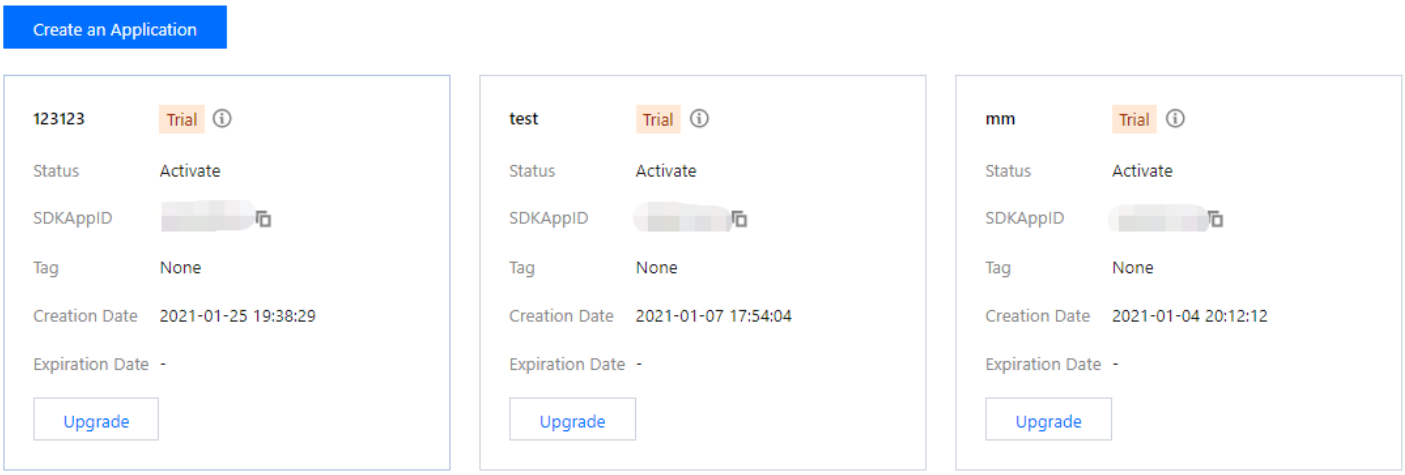
```
// 1. Obtain the SDKAppID of the application from the IM console. For more information, see SDKAp
pID.
// 2. Initialize the `config` object.
V2TIMSDKConfig config = new V2TIMSDKConfig();
// 3. Specify the log output level. For more information, see SDKConfig.
config.setLogLevel(V2TIMSDKConfig.V2TIM_LOG_INFO);
// 4. Initialize the SDK and set the listening object of `V2TIMSDKListener`.
// After you call `initSDK`, the SDK automatically connects to the network. The network connectio
n status can be listened to in the `V2TIMSDKListener` callback.
V2TIMManager.getInstance().initSDK(context, sdkAppID, sdkConfig, new V2TIMSDKListener() {
// 5. Listen to the `V2TIMSDKListener` callback.
@Override
public void onConnecting() {
// The SDK is connecting to the Tencent CVM instance.
}
@Override
public void onConnectSuccess() {
// The SDK is successfully connected to the Tencent CVM instance.
}
@Override
public void onConnectFailed(int code, String error) {
// The SDK fails to connect to the Tencent CVM instance.
}
});
```

The initialization API `initSDK` contains three required parameters, `SDKAppID` , `SDKConfig` , and `listener` .

SDKAppID

`SDKAppID` is a unique ID that the IM service uses to identify a customer account. We recommend that you apply for a new `SDKAppID` for every independent app to automatically isolate messages

between `SDKAppIDs` .
You can view all `SDKAppIDs` in the IM console or click **Add Application** to create an `SDKAppID` .



SDKConfig

The `V2TIMSDKConfig` parameter is used for SDK initialization configuration. It is often used to set the log level, that is, the setLogLevel API. The following table lists the log levels:

Log Level	Log Output
V2TIM_LOG_NONE	No log is output.
V2TIM_LOG_DEBUG	Logs of the DEBUG, INFO, WARNING, and ERROR levels are output.
V2TIM_LOG_INFO	Logs of the INFO, WARNING, and ERROR levels are output.
V2TIM_LOG_WARN	Logs of the WARNING and ERROR levels are output.
V2TIM_LOG_ERROR	Logs of the ERROR level are output.

- IM SDK logs are stored by default in the `/sdcard/tencenet/imsdklogs/<application package="" name="">` directory for versions earlier than 4.8.50 and in the `/sdcard/Android/data/<package name="">/files/log/tencent/imsdk` directory for version 4.8.50 or later.
- Starting from v4.7.1, the xlog module of the WeChat team is used to output IM SDK logs. The xlogs are decompressed by default and must be decompressed using the Python script.
  - To obtain the script for decompression, click Decode Log 27 if you are using Python 2.7, or click Decode Log 30 if you are using Python 3.0.
  - In the Windows or Mac console, you can run the following command to decompress the log files. After decompression, the file names end with "xlog.log", and you can use the text editor to open these files.

```
python decode_mars_nocrypt_log_file.py imsdk_yyyyMMdd.xlog
```

Listener

V2TIMSDKListener is used to listen to the network status and changes to user information.

Event Callback	Event Description	Recommended Operation
onConnecting	The SDK is connecting to the CVM instance.	The "Connecting" status can be displayed on the UI.
onConnectSuccess	The SDK is successfully connected to the CVM instance.	-
onConnectFailed	The SDK fails to connect to the CVM instance.	The user can be notified that the network connection is currently unavailable.
onKickedOffline	The current user is kicked offline.	The "You have already logged in to the SDK on another device using the current account. Are you sure you want to log in again?" message can be displayed on the UI.
onUserSigExpired	The UserSig expires.	Use the new UserSig for login.
onSelfInfoUpdated	The information of the current user is updated.	Update your own profile photo and nickname on the UI.

Note：
If you receive the `onUserSigExpired` callback, the UserSig that you use for login has expired. In this case, you need to update the UserSig and then log in again. If you continue to use the expired UserSig, the SDK falls into an endless login loop.

Login

You can call the login(userID, userSig) function of `V2TIMManager` to log in to the SDK. The features of the IM SDK are available to you only after you successfully log in to the SDK.

- UserID: you are advised to enter only letters (a-z and A-Z), digits (0-9), underscores (_), and hyphens (-). Its length cannot exceed 32 bytes.
- UserSig: login ticket of the IM SDK. It is calculated by your business server to ensure security. For more information on the calculation method, see Generating UserSig.

> Note :
> After you log in successfully by calling `IM SDK Login`, DAU will be calculated. Please use `IM SDK Login` appropriately according to the business scenario to avoid an excessively high DAU.

Login scenarios

You need to call the `login` function in the following scenarios:

- When you need to use features of the IM SDK for the first time after the app is started.
- When the IM SDK triggers an `onUserSigExpired` callback. That is, when the UserSig expires, you need to use the new UserSig for login.
- When the IM SDK triggers an `onKickOffline` callback. That is, when the current user is kicked offline, the "You have already logged in to the SDK on another device using the current account. Are you sure you want to log in again?" message can be displayed on the UI. In this case, you can select "Yes" to log in again.

You do not need to call the `login` function in the following scenarios:

- When your network is disconnected and then reconnected, you do not need to call the `login` function as the SDK automatically goes online.
- When a login process is running, you do not need to log in to the SDK again.

Multi-device login

You cannot use the same account to log in on two mobile phones of the same model. For example, you cannot use the same account for login on two Apple mobile phones. However, one Android mobile phone and one Apple mobile phone will be considered as two different devices, and you can use the same account to log in on these two devices. For more information on configurations related to multi-device login, see the **Login settings** section in Feature Configuration.

Logout

To log out of the SDK, call the logout function.

Initialization and Login (iOS)

Last updated : 2021-10-15 14:55:12

Initialization

V2TIMManager is a core class and also an entry class of the IM SDK. It implements features such as IM SDK initialization and login, message sending/receiving, group creation, and group leaving. To complete initialization, call the initSDK API.

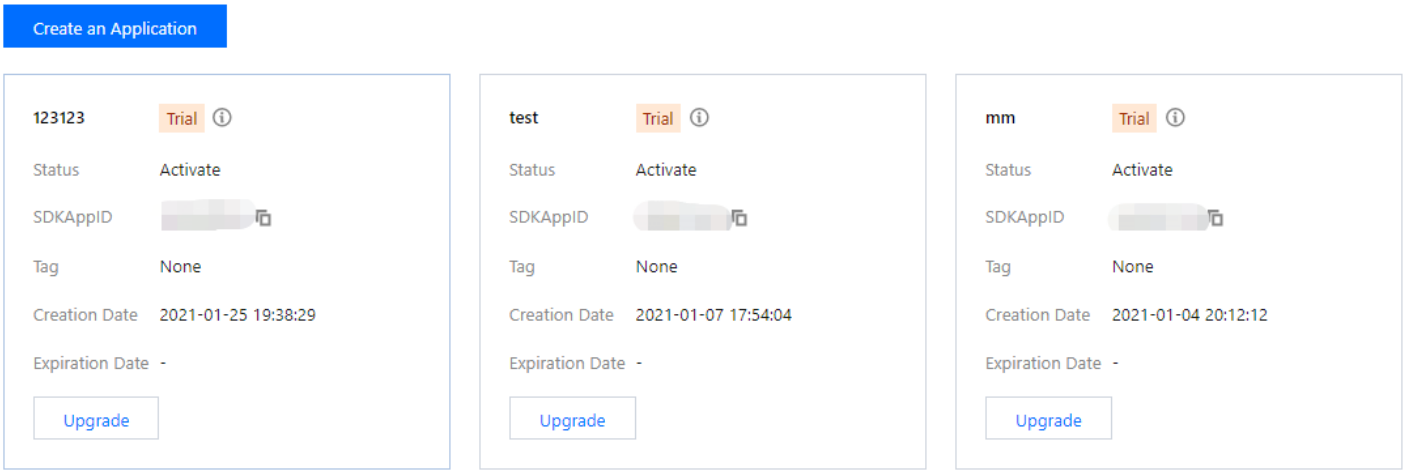
```
// 1. Obtain the SDKAppID of the application from the IM console. For more information, see SDKAppID.
// 2. Initialize the `config` object.
V2TIMSDKConfig *config = [[V2TIMSDKConfig alloc] init];
// 3. Specify the log output level. For more information, see [SDKConfig](#SDKAppID).
config.logLevel = V2TIM_LOG_INFO;
// 4. Initialize the SDK and set the listening object of `V2TIMSDKListener`.
// After you call `initSDK`, the SDK automatically connects to the network. The network connection status can be listened to in the `V2TIMSDKListener` callback.
[[V2TIMManager sharedInstance] initSDK:1400000123 config:config listener:self];

// 5. Listen to the `V2TIMSDKListener` callback.
- (void)onConnecting {
// The SDK is connecting to the Tencent CVM instance.
}
- (void)onConnectSuccess {
// The SDK is successfully connected to the Tencent CVM instance.
}
- (void)onConnectFailed:(int)code err:(NSString*)err {
// The SDK fails to connect to the Tencent CVM instance.
}
```

The initialization API `initSDK` contains three required parameters, including `SDKAppID`, `Config`, and `listener`.

SDKAppID

`SDKAppID` is a unique ID that the IM service uses to identify a customer account. We recommend that you apply for a new `SDKAppID` for every independent app to automatically isolate messages between `SDKAppIDs`.

You can view all `SDKAppIDs` in the IM console or click **Add Application** to create an `SDKAppID` .



SDKConfig

The `V2TIMSDKConfig` parameter is used for initialization configuration of the SDK. It is often used to set the log level, that is, the logLevel parameter. The following table lists the log levels.

Log Level	Log Output
V2TIM_LOG_NONE	No log is output.
V2TIM_LOG_DEBUG	Logs of the DEBUG, INFO, WARNING, and ERROR levels are output.
V2TIM_LOG_INFO	Logs of the INFO, WARNING, and ERROR levels are output.
V2TIM_LOG_WARN	Logs of the WARNING and ERROR levels are output.
V2TIM_LOG_ERROR	Logs of the ERROR level are output.

- Logs of the IM SDK are stored in the `/Library/Caches/` directory by default.
- Starting from V4.7.1, the xlog module of the WeChat team is used to output IM SDK logs. The xlogs are decompressed by default and must be decompressed using the Python script.
  - To obtain the script for decompression, click Decode Log 27 if you are using Python 2.7, or click Decode Log 30 if you are using Python 3.0.
  - In the Windows or Mac console, you can run the following command to decompress the log files. After decompression, the file names end with "xlog.log", and you can use the text editor to open these files.

```
python decode_mars_nocrypt_log_file.py imsdk_yyyyMMdd.xlog
```

Listener

V2TIMSDKListener is used to listen to the network status and changes to the user information.

Event Callback	Event Description	Recommended Operation
onConnecting	The SDK is connecting to the CVM instance.	The "Connecting" status can be displayed on the UI.
onConnectSuccess	The SDK is successfully connected to the CVM instance.	-
onConnectFailed	The SDK fails to connect to the CVM instance.	The user can be notified that the network connection is currently unavailable.
onKickedOffline	The current user is kicked offline.	The "You have already logged in to the SDK on another device using the current account. Are you sure you want to log in again?" message can be displayed on the UI.
onUserSigExpired	The UserSig expires.	Use the new UserSig for login.
onSelfInfoUpdated	The information of the current user is updated.	Update your own profile photo and nickname on the UI.

Note：
If you receive the `onUserSigExpired` callback, the UserSig that you use for login has expired. In this case, you need to update the UserSig and then log in again. If you continue to use the expired UserSig, the SDK falls into an endless login loop.

Login

You can call the login(userID, userSig) function of `V2TIMManager` to log in to the SDK. Features of the IM SDK are available to you only after you successfully log in to the SDK.

- UserID: we recommend that UserID contain only uppercase or lowercase letters, digits, underscores, and hyphens. Its length cannot exceed 32 bytes.
- UserSig: login ticket of the IM SDK. It is calculated by your business server to ensure security. For more information on the calculation method, see Generating UserSig.

> Note :
> After you log in successfully by calling `IM SDK Login`, DAU will be calculated. Please use `IM SDK Login` appropriately according to the business scenario to avoid an excessively high DAU.

Login scenarios

You need to call the `login` function in the following scenarios:

- When you need to use features of the IM SDK for the first time after the app is started.
- When the IM SDK triggers an `onUserSigExpired` callback. That is, when the UserSig expires, you need to use the new UserSig for login.
- When the IM SDK triggers an `onKickOffline` callback. That is, when the current user is kicked offline, the "You have already logged in to the SDK on another device using the current account. Are you sure you want to log in again?" message can be displayed on the UI. In this case, you can select "Yes" to log in again.

You do not need to call the `login` function in the following scenarios:

- When your network is disconnected and then reconnected, you do not need to call the `login` function as the SDK automatically goes online.
- When a login process is running, you do not need to log in to the SDK again.

Multi-device login

You cannot use the same account to log in on two mobile phones of the same model. For example, you cannot use the same account for login on two Apple mobile phones. However, one Android mobile phone and one Apple mobile phone will be considered as two different devices, and you can use the same account to log in on these two devices. For more information on configurations related to multi-device login, see the **Login settings** section in Feature Configuration.

Logout

To log out of the SDK, call the logout() function.

Initialization and Login (Web & Mini Program)

Last updated : 2021-05-31 11:27:19

Creating an SDK Instance

Web project

```
import TIM from 'tim-js-sdk';
// The Tencent Cloud IM upload plugin is required to send messages such as images and files.
import TIMUploadPlugin from 'tim-upload-plugin';

let options = {
SDKAppID: 0 // Replace `0` with the `SDKAppID` of your IM app during connection.
};
// Create an SDK instance. The `TIM.create()` method returns the same instance for the same `SDKAppID`.
let tim = TIM.create(options); // The SDK instance is usually represented by `tim`.

// Set the SDK log output level. For more information on each level, see setLogLevel API Description.
tim.setLogLevel(0); // Common level. You are advised to use this level during connection as it covers more logs.
// tim.setLogLevel(1); // Release level, at which the SDK outputs important information. You are advised to use this log level in a production environment.

// Register the Tencent Cloud IM upload plugin.
tim.registerPlugin({'tim-upload-plugin': TIMUploadPlugin});
```

Mini Program project

```
import TIM from 'tim-wx-sdk';
// The Tencent Cloud IM upload plugin is required to send messages such as images and files.
import TIMUploadPlugin from 'tim-upload-plugin';


let options = {
SDKAppID: 0 // Replace `0` with the `SDKAppID` of your IM app during connection.
};
// Create an SDK instance. The `TIM.create()` method returns the same instance for the same `SDKAppID`.
```

```
let tim = TIM.create(options); // The SDK instance is usually represented by `tim`.

// Set the SDK log output level. For more information on each level, see setLogLevel API Description.
tim.setLogLevel(0); // Common level. You are advised to use this level during connection as it covers more logs.
// tim.setLogLevel(1); // Release level, at which the SDK outputs important information. You are advised to use this log level in a production environment.

// Register the Tencent Cloud IM upload plugin.
tim.registerPlugin({'tim-upload-plugin': TIMUploadPlugin});
```

Setting the Log Level

```
// Set the SDK log output level. For more information on each level, see setLogLevel API Description.
tim.setLogLevel(0);
```

Binding Events

```
// Listened-to events, for example:
tim.on(TIM.EVENT.SDK_READY, function(event) {
// A notification is received, indicating that the offline message and conversation lists are synchronized successfully. The access side can call APIs that require authentication, such as `sendMessage`.
// event.name - TIM.EVENT.SDK_READY
});
tim.on(TIM.EVENT.MESSAGE_RECEIVED, function(event) {
// A newly pushed one-to-one message, group message, group tip, or group system notification is received. You can traverse `event.data` to obtain the message list and render it to the UI.
// event.name - TIM.EVENT.MESSAGE_RECEIVED
// event.data - An array that stores the Message objects - [Message]
});
tim.on(TIM.EVENT.MESSAGE_REVOKED, function(event) {
// The message recall notification is received.
// event.name - TIM.EVENT.MESSAGE_REVOKED
// event.data - An array that stores the Message objects - [Message] - The `isRevoked` value of each Message object is `true`.
});
tim.on(TIM.EVENT.MESSAGE_READ_BY_PEER, function(event) {
```

```
// SDK has received a notification indicating that the opposite end has read the message. This is
the read receipt. Before using it, you need to upgrade the SDK version to V2.7.0 or higher. Only
one-to-one conversations are supported.
// event.name - TIM.EVENT.MESSAGE_READ_BY_PEER
// event.data - event.data - An array that stores the Message objects - [Message] - The `isPeerRe
ad` value of each Message object is `true`.
});
tim.on(TIM.EVENT.CONVERSATION_LIST_UPDATED, function(event) {
// A conversation list update notification is received. You can traverse `event.data` to obtain t
he conversation list and render it to the UI.
// event.name - TIM.EVENT.CONVERSATION_LIST_UPDATED
// event.data - An array that stores the Conversation objects - [Conversation]
});
tim.on(TIM.EVENT.GROUP_LIST_UPDATED, function(event) {
// A group list update notification is received. You can traverse `event.data` to obtain the grou
p list and render it to the UI.
// event.name - TIM.EVENT.GROUP_LIST_UPDATED
// event.data - An array that stores the Group objects - [Group]
});
tim.on(TIM.EVENT.PROFILE_UPDATED, function(event) {
// A notification is received, indicating that your own profile or your friend's profile is updat
ed.
// event.name - TIM.EVENT.PROFILE_UPDATED
// event.data - An array that stores the Profile objects - [Profile]
});
tim.on(TIM.EVENT.BLACKLIST_UPDATED, function(event) {
// A blocklist update notification is received.
// event.name - TIM.EVENT.BLACKLIST_UPDATED
// event.data - An array that stores userIDs - [userID]
});
tim.on(TIM.EVENT.ERROR, function(event) {
// An SDK error notification is received. The error code and error message can be obtained from t
his notification.
// event.name - TIM.EVENT.ERROR
// event.data.code - Error code
// event.data.message - Error message
});
tim.on(TIM.EVENT.SDK_NOT_READY, function(event) {
// An SDK not ready notification is received. At this time, the SDK cannot function normally.
// event.name - TIM.EVENT.SDK_NOT_READY
});
tim.on(TIM.EVENT.KICKED_OUT, function(event) {
// A kicked offline notification is received.
// event.name - TIM.EVENT.KICKED_OUT
// event.data.type - Reason for being kicked offline, such as:
// - TIM.TYPES.KICKED_OUT_MULT_ACCOUNT: kicked offline due to multi-instance login.
// - TIM.TYPES.KICKED_OUT_MULT_DEVICE: kicked offline due to multi-device login.
// - TIM.TYPES.KICKED_OUT_USERSIG_EXPIRED: kicked offline due to UserSig expiration (supported fr
```

```
om V2.4.0).
});
tim.on(TIM.EVENT.NET_STATE_CHANGE, function(event) {
// The network status changes (supported from V2.5.0).
// event.name - TIM.EVENT.NET_STATE_CHANGE
// event.data.state indicates the current network status. The enumerated values are described as
follows:
// ¥- TIM.TYPES.NET_STATE_CONNECTED: already connected to the network
// ¥- TIM.TYPES.NET_STATE_CONNECTING: connecting. This often occurs when the SDK must reconnect d
ue to network jitter. The prompt "The current network is unstable" or "Connecting..." can be disp
layed at the access side based on this status.
// ¥- TIM.TYPES.NET_STATE_DISCONNECTED: disconnected. The prompt "The current network is unavaila
ble" can be displayed at the access side based on this status. The SDK will continue to try to re
connect. If the user network recovers, the SDK will automatically synchronize messages.
});
// Start to log in to the SDK.
tim.login({userID: 'your userID', userSig: 'your userSig'});
```

The `options` parameter is of the `Object` type:

Name	Type	Description
`options`	`Object`	App configuration

`options` includes the following attribute value:

Name	Type	Description
`SDKAppID`	`Number`	`SDKAppID` of the IM app

For more information on how to initialize the SDK and use APIs, see SDK Initialization.

Login

You can send and receive messages in the IM console only after logging in to the IM SDK. To log in to the IM SDK, you need to provide information such as the UserID and UserSig. For more information, see Login Authentication. After successful login, to call APIs that require authentication, such as sendMessage, you must wait until the SDK enters the ready state. You can obtain the status of the SDK by listening to events. For more information, see TIM.EVENT.SDK_READY.

Note :

By default, multi-instance login is not supported. If you use an account that has been logged in on another page to log in on the current page, the account may be forcibly logged out on the other page, which will trigger the `TIM.EVENT.KICKED_OUT` event. You can proceed accordingly after detecting the event through listening. An example of listening for multi-instance login is shown below:

```
javascript
let onKickedOut = function (event) {
console.log(event.data.type); // mutipleAccount (The same account that is used to log in on multi
ple pages on the same device is forcibly logged out.)
};
tim.on(TIM.EVENT.KICKED_OUT, onKickedOut);
```

To support multi-instance login (allowing the use of the same account to log in concurrently on multiple pages), log in to the IM console and click **Feature Configuration** -> **Login and Messages**. In the drop-down list next to **Login and Messages**, select the corresponding `SDKAppID`. Then, in the **Login settings** module, click **Edit** and use the drop-down list next to **Online Web Instances** to configure the number of instances. The configuration will take effect within 5 minutes.

API

```
javascript
tim.login(options);
```

Request parameters

Name	Type	Description
UserID	String	The ID of the user.
UserSig	String	The password with which the user logs in to the IM console. It is essentially the ciphertext generated by encrypting information such as the UserID. For the detailed generation method, see Generating UserSig.

Response

This API returns a `Promise` object.

Example

```
javascript
let promise = tim.login({userID: 'your userID', userSig: 'your userSig'});
promise.then(function(imResponse) {
```

```
console.log(imResponse.data); // Login succeeds.
if (imResponse.data.repeatLogin === true) {
// Indicates that the account has logged in and that the current login will be a repeated login.
This feature is supported from V2.5.1.
console.log(imResponse.data.errorInfo);
}
}).catch(function(imError){
console.warn('login error:', imError); // Information about the login failure.
});
```

Logout

This API is used to log out of the IM console. It is usually called when you switch between accounts. This API clears the login status of the current account and all the data in the memory.

Note :

- When calling this API, the instance publishes the SDK_NOT_READY event. In this case, the instance is automatically logged out and cannot receive or send messages.
- Assume that the value of the **Online Web Instances** configured in the IM console is greater than 1, and the same account has been used to log in to instances `a1` and `a2` (including a Mini Program instance). After `a1.logout()` is executed, `a1` is automatically logged out and cannot receive or send messages, whereas `a2` is not affected.
- Assume that the **Online Web Instances** is set to 2, and your account has been used to log in to instances `a1` and `a2` . When you use this account to log in to instance `a3` , either `a1` or `a2` will be forcibly logged out. In most cases, the instance that first entered the login state is forcibly logged out. This is called **kicked offline due to multi-instance login**. If `a1` is forcibly logged out, a logout process is executed within `a1` and the KICKED_OUT event is triggered. The access side can listen to this event and redirect to the login page when the event is triggered. At this time, `a1` is forcibly logged out, whereas instances `a2` and `a3` can continue to run properly.

API

```
js
tim.logout();
```

Request parameters

N/A

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data` is a null object, indicating that logout succeeded.
- The callback parameter of `catch` is IMError.

Example

```
js
let promise = tim.logout();
promise.then(function(imResponse) {
console.log(imResponse.data); // Logout succeeds.
}).catch(function(imError){
console.warn('logout error:', imError);
});
```

Message Sending and Receiving
Message Sending and Receiving (Android)

Last updated : 2021-10-22 18:15:36

Message Classification

IM messages are classified by message destination into two types: one-to-one messages (also called C2C messages) and group messages.

Message Type	API Keyword	Description
One-to-one message	C2CMessage	Also called C2C message. When sending a one-to-one message, you must specify the UserID of the message recipient, and only the recipient can receive this message.
Group message	GroupMessage	When sending a group message, you must specify the groupID of the target group, and all users in this group can receive this message.

IM messages can also be classified by content into text messages, custom (signaling) messages, image messages, video messages, voice messages, file messages, location messages, combined messages, and group tips.

Message Type	API Keyword	Description
Text message	TextElem	It refers to a common text message. Sensitive words of text messages will be filtered out in the IM service. If a message containing sensitive words is sent, the 80001 error code is returned.
Custom message	CustomElem	It is a section of the binary buffer, which is often used to transfer custom signaling in your application. Its content is not filtered for sensitive words.

Message Type	API Keyword	Description
Image message	ImageElem	When the IM SDK sends an original image, it automatically generates two images in different sizes. The three images are called the original image, large image, and thumbnail.
Video message	VideoElem	A video message contains a video file and an image.
Voice message	SoundElem	Supports displaying a red dot upon playback of the voice message.
File message	FileElem	A file message cannot exceed 100 MB.
Location message	LocationElem	A location message contains three fields: location description, longitude, and latitude.
Combined message	MergerElem	Up to 300 messages can be combined.
Group tip	GroupTipsElem	A group tip is often used to carry a system notification in a group, for example, a notification indicating that a member joins or leaves the group, the group description is modified, or the profile of a group member is changed.

Sending and Receiving Simple Messages

V2TIMManager provides a set of simple APIs for sending and receiving messages. Although these APIs can be used to send or receive text messages and custom (signaling) messages, they are easy to use and only a few minutes are needed to complete interfacing.

Sending text and signaling messages (simplified APIs)

To send text messages, call sendC2CTextMessage or sendGroupTextMessage. Text messages will be filtered by IM for sensitive words. If a message containing sensitive words is sent, the 80001 error code is returned. To send one-to-one custom (signaling) messages, call sendC2CCustomMessage or sendGroupCustomMessage. A custom message is essentially a section of the binary buffer, and is often used to transfer custom signaling in your application. Its content is not filtered for sensitive words.

Receiving text and signaling messages (simplified APIs)

To listen to simple text and signaling messages, call addSimpleMsgListener. To listen to image, video, and voice messages, call addAdvancedMsgListener defined in V2TIMMessageManager.

> Note :
> Do not use addSimpleMsgListener together with addAdvancedMsgListener; otherwise, logic bugs may occur.

Typical example: sending and receiving on-screen comments in an audio-video group

In the live streaming scenario, it is a common way of communication to send or receive on-screen comments in an audio-video group. This can be easily implemented through the simple message APIs.

1. The anchor can call createGroup to create an audio-video group (AVChatRoom) and record the group ID in the list of rooms in "Broadcasting" state.
2. A viewer can select an anchor that he/she likes, and call joinGroup to join the AVChatRoom created by this anchor.
3. The message sender can call sendGroupTextMessage to send a group text message as an on-screen comment.
4. The message recipient can call addSimpleMsgListener to register a simple message listener, and use the listener callback function onRecvGroupTextMessage to obtain text messages.

"FlyHeart" is an instruction. To configure the "FlyHeart" feature for a live room, perform the steps below:

1. Define a custom message type, for example, a JSON string `{ "command": "favor", "value": 101 }`.
2. Call sendGroupCustomMessage to send a message, and call onRecvGroupCustomMessage to receive the message.

Sending and Receiving Rich Media Messages

Image, video, voice, file, and location messages are called rich media messages. Compared with simple messages, it is more complex to send or receive rich media messages.

- Before sending a rich media message, use the `create` function to create a V2TIMMessage object. Then, call the corresponding `send` API to send this message.
- When receiving the rich media message, check `elemType` and perform secondary parsing on `Elem` obtained based on `elemType`.

Sending rich media messages

The following takes an image message as an example to describe the process of sending a rich media message.

1. The sender calls createImageMessage to create an image message, and obtain the V2TIMMessage message object.
2. The sender calls sendMessage to send the created message object.

Receiving rich media messages

1. The recipient calls addAdvancedMsgListener to set the advanced message listener.
2. The recipient obtains the image message V2TIMMessage through the listener callback onRecvNewMessage.
3. The recipient parses `elemType` in V2TIMMessage, and performs secondary parsing based on the message type to obtain the content of `Elem` in the message.

Typical example: sending and receiving image messages

The sender creates and sends an image message.

```
// Create an image message
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createImageMessage("/sdcard/test.pn
g");
// Send the image message
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "toUserID", null, V2TIMMessage.V2TIM_P
RIORITY_DEFAULT, false, null, new V2TIMSendCallback<V2TIMMessage>() {
@Override
public void onError(int code, String desc) {
// The image message fails to be sent
}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {
// The image message is successfully sent
}
@Override
public void onProgress(int progress) {
// Image upload progress (0-100)
}
});
```

The recipient identifies the image message, and parses the message to obtain the original image, large image, and thumbnail contained in the message.

```
@Override
public void onRecvNewMessage(V2TIMMessage msg) {
int elemType = msg.getElemType();
if (elemType == V2TIMMessage.V2TIM_ELEM_TYPE_IMAGE) {
V2TIMImageElem v2TIMImageElem = msg.getImageElem();
// An image message contains an image in three different sizes: original image, large image, and thumbnail. (The SDK automatically generates a large image and a thumbnail.)
// A large image is an image obtained after the original image is proportionally compressed. After the compression, the smaller one of the height and width is equal to 720 pixels.
// A thumbnail is an image obtained after the original image is proportionally compressed. After the compression, the smaller one of the height and width is equal to 198 pixels.
List<V2TIMImageElem.V2TIMImage> imageList = v2TIMImageElem.getImageList();
for (V2TIMImageElem.V2TIMImage v2TIMImage : imageList) {
String uuid = v2TIMImage.getUUID(); // Image ID
int imageType = v2TIMImage.getType(); // Image type
int size = v2TIMImage.getSize(); // Image size (bytes)
int width = v2TIMImage.getWidth(); // Image width
int height = v2TIMImage.getHeight(); // Image height
// Set the image download path `imagePath`. Here, `uuid` can be used as an identifier to avoid repeated download.
String imagePath = "/sdcard/im/image/" + "myUserID" + uuid;
File imageFile = new File(imagePath);
if (imageFile.exists()) {
v2TIMImage.downloadImage(imagePath, new V2TIMDownloadCallback() {
@Override
public void onProgress(V2TIMElem.V2ProgressInfo progressInfo) {
// Image download progress. `v2ProgressInfo.getCurrentSize()` indicates the downloaded size, and `v2ProgressInfo.getTotalSize()` indicates the total file size.
}
@Override
public void onError(int code, String desc) {
// The image fails to be downloaded
}
@Override
public void onSuccess() {
// The image download is completed
}
});
} else {
// The image already exists
}
}
}
}
```

Note :
For more information on the message parsing sample code, see FAQs > 5. How can I parse different types of messages.

Sending and Receiving Group @ Messages

For a group @ message, the sender can listen to the input of the @ character in the input box and call the group member selection interface. After selection is completed, the input box displays the content in the format of `"@A @B @C....."` ,and then the sender can continue to edit the message content and send the message. On the group chat list of the recipient's conversation interface, the identifier `"someone@me"` or `"@all members"` will be displayed to remind the user that the user was mentioned by someone in the group.

Note :
Currently, only text @ messages are supported.

Sending group @ messages

1. The sender listens to the text input box on the chat interface and launches the group member selection interface. After selection is completed, the ID and nickname of the selected member are returned. The ID is used to construct the message object V2TIMMessage, and the nickname is displayed in the text box.
2. The sender calls createTextAtMessage of V2TIMMessageManager to create an @ text message and obtain the message object V2TIMMessage.
3. The sender calls sendMessage to send the created @ message object.

Receiving group @ messages

1. During conversation loading and update, call the getGroupAtInfoList API of V2TIMConversation to obtain the @ data list of the conversation < V2TIMGroupAtInfo >.
2. Obtain and update the @ data type to the @ information of the current conversation through the getAtType API of the V2TIMGroupAtInfo object on the list.

Typical example: sending and receiving group @ messages

- **Sending a group @ message**:
  The sender creates and sends a group @ message:

```
// Obtain the group member ID data
List<String> atUserList = updateAtUserList(mTextInput.getMentionList(true));
// Create a group @ message
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createTextAtMessage(message, atUserList);
// Send the group @ message
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, null, "toGroupID", V2TIMMessage.V2TIM_PRIORITY_DEFAULT, false, null, new V2TIMSendCallback<V2TIMMessage>() {
@Override
public void onError(int code, String desc) {
// The group @ message fails to be sent
}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {
// The group @ message is sent successfully
}
@Override
public void onProgress(int progress) {
}
});
```

- **Receiving a group @ message**:
  During conversation loading and update, obtain the group @ data list:

```
boolean atMe = false;
boolean atAll = false;
// Obtain the group @ data list
List<V2TIMGroupAtInfo> atInfoList = conversation.getGroupAtInfoList();
if (atInfoList == null || atInfoList.isEmpty()){
return V2TIMGroupAtInfo.TIM_AT_UNKNOWN;
}
// Obtain the @ data type
for(V2TIMGroupAtInfo atInfo : atInfoList){
if (atInfo.getAtType() == V2TIMGroupAtInfo.TIM_AT_ME){
atMe = true;
continue;
}
if (atInfo.getAtType() == V2TIMGroupAtInfo.TIM_AT_ALL){
atAll = true;
continue;
}
```

```
}
if (atAll && atMe){
atInfoType = V2TIMGroupAtInfo.TIM_AT_ALL_AT_ME;
} else if (atAll){
atInfoType = V2TIMGroupAtInfo.TIM_AT_ALL;
} else if (atMe){
atInfoType = V2TIMGroupAtInfo.TIM_AT_ME;
} else {
atInfoType = V2TIMGroupAtInfo.TIM_AT_UNKNOWN;
}
// Update the @ type to the current conversation
switch (atInfoType){
case V2TIMGroupAtInfo.TIM_AT_ME:
Log.d(TAG, "update to the current conversation to display [someone@me]");
break;
case V2TIMGroupAtInfo.TIM_AT_ALL:
Log.d(TAG, "update to the current conversation to display [@all members]");
break;
case V2TIMGroupAtInfo.TIM_AT_ALL_AT_ME:
Log.d(TAG, "update to the current conversation to display [someone@me][@all members]");
break;
default:
break;
}
```

Sending and Receiving Combined Messages (Only Available in Lite Edition v5.2.210 and Above)

To implement the combined forward feature similar to that in WeChat, it is necessary to create a combined message according to the original message list, and then send the combined message to the opposite end. After the opposite end receives the combined message, it will parse out the original message list. The display of the combined message also requires the title and abstract information.

- **Sending combined messages**
  Usually when we receive a combined message, the chat screen will look like this:

Chat History of Vinson and Lynx	Title
Vinson: When is the new version of SDK scheduled to go online?	abstract1

Chat History of Vinson and Lynx	Title
Lynx: Next Monday. The specific time depends on the system test result in these two days.	abstract2
Vinson: OK	abstract3

The chat interface will display only the title and abstract information of the combined message, and the combined message list will be displayed only when the user clicks the combined message. When we create a combined message, we need to set not only the combined message list, but also the title and abstract information. The implementation process is as follows:

1. Call the createMergerMessage API to create a combined message.
2. Call the sendMessage API to send the combined message.

- **Receiving combined messages**
  When receiving a combined message V2TIMMessage, use V2TIMMergerElem to get title and abstractList for UI display. When a user clicks the combined message, call the downloadMergerMessage API to download the combine message list for UI display.

Typical example: sending and receiving combined messages

- **Sending combined messages**
  The sender creates a combined message and sends it.

```
// List of messages that need to be forwarded, which can contain combined messages and cannot con
tain group tips
List<V2TIMMessage> msgs = new ArrayList<>();
msgs.add(message1);
msgs.add(message2);
// Title of the combined message
String title = "Chat History of Vinson and Lynx";
// Abstract list of the combined message
List<String> abstactList = new ArrayList<>();
msgs.add("abstract1");
msgs.add("abstract2");
msgs.add("abstract3");
// Combined messages are compatible with text. SDKs of early versions do not support combined mes
sages, and they will send a text message with the content `compatibleText` by default.
String compatibleText = "Please upgrade to the latest version to check the combined message";
// Create a combined message
V2TIMMessage mergeMessage = V2TIMManager.getMessageManager().createMergerMessage(msgs, title, abs
tractList, compatibleText);
```

```
// Send the combined message to the user Denny
V2TIMManager.getMessageManager().sendMessage(mergeMessage, "denny", null, V2TIMMessage.V2TIM_PRIO
RITY_NORMAL, false, null, new V2TIMSendCallback<V2TIMMessage>() {
@Override
public void onProgress(int progress) {}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {}
@Override
public void onError(int code, String desc) {}
})
```

- **Receiving combined messages**

  The recipient receives the combined message and parses it:

```
@Override
public void onRecvNewMessage(V2TIMMessage msg) {
if (msg.getElemType() == V2TIMMessage.V2TIM_ELEM_TYPE_MERGER) {
// Get the combined message elements
V2TIMMergerElem mergerElem = msg.getMergerElem();
// Get the title
String title = mergerElem.getTitle();
// Get the abstract list
List<String> abstractList = mergerElem.getAbstractList();
// Download the combined message list when a user clicks the combined message
mergerElem.downloadMergerMessage(new V2TIMValueCallback<List<V2TIMMessage>>() {
@Override
public void onSuccess(List<V2TIMMessage> v2TIMMessages) {
// Download succeeded. `v2TIMMessages` is the combined message list
for (V2TIMMessage subMsg : v2TIMMessages) {
// If the combined message list still contains combined messages, you can continue parsing
if (subMsg.getElemType() == V2TIMMessage.V2TIM_ELEM_TYPE_MERGER) {
V2TIMMergerElem mergerElem = subMsg.getMergerElem();
// Get the title
String title = mergerElem.getTitle();
// Get the abstract list
List<String> abstractList = mergerElem.getAbstractList();
// Download the combined message list when a user clicks the combined message
......
}
}
}
@Override
public void onError(int code, String desc) {
// Download failed
}
});
}
```

Sending Messages That Are Excluded from the Unread Count (Only Available in Lite Edition v5.3.425 and Above)

Normally, when you send one-to-one chat messages and group messages, the messages are included in the unread count (you can get the unread message count of a conversation via the getUnreadCount API of the V2TIMConversation conversation object). If you need to send messages that are excluded from the unread count, such as tips and control messages, send them as follows:

```
// Create the message object
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createTextMessage(content);
// Set the identifier for excluding from the unread message count
v2TIMMessage.setExcludedFromUnreadCount(true);
// Send the message
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "userA", null, V2TIMMessage.V2TIM_PRIO
RITY_DEFAULT, false, null, new V2TIMSendCallback<V2TIMMessage>() {
@Override
public void onError(int code, String desc) {
// The message fails to be sent
}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {
// The message is sent successfully
}
@Override
public void onProgress(int progress) {
}
});
```

Sending Messages That Are Excluded from the Conversation lastMsg (Only Available in Enhanced Edition v5.4.666 and Above)

In certain scenarios, if you need to send messages that are excluded from the conversation `lastMsg`, send them as follows:

```
// Create the message object
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createTextMessage(content);
// Set the identifier for excluding from the conversation lastMsg
v2TIMMessage.setExcludedFromLastMessage(true);
// Send the message
```

```
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "userA", null, V2TIMMessage.V2TIM_PRIO
RITY_DEFAULT, false, null, new V2TIMSendCallback<V2TIMMessage>() {
@Override
public void onError(int code, String desc) {
// The message fails to be sent
}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {
// The message is sent successfully
}
@Override
public void onProgress(int progress) {
}
});
```

Setting Offline Push (offlinePushInfo)

When the recipient's app is killed, the IM SDK cannot receive new messages through the normal network connection. In this scenario, the offline push service provided by mobile phone manufacturers must be used to notify the recipient of new messages. For more information, see Offline Push (Android).

Setting the title and content for offline push

When sending messages, you can use the `offlinePushInfo` field in the sendMessage API to set the title and content for offline push.

```
// Create and send a text message to groupA, and customize the title and content for offline push
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createTextMessage(content);
V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
// Set the title of the notification bar
v2TIMOfflinePushInfo.setTitle("offline_title");
// Set the content of the notification bar
v2TIMOfflinePushInfo.setDesc("offline_desc");
// Send the message
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, null, "groupA", V2TIMMessage.V2TIM_PRI
ORITY_DEFAULT, false, v2TIMOfflinePushInfo, new V2TIMSendCallback<V2TIMMessage>() {
@Override
public void onError(int code, String desc) {
// The message fails to be sent
}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {
// The message is sent successfully
```

```
}
@Override
public void onProgress(int progress) {
}
});
```

Clicking a pushed message to go to the corresponding chat window

To implement this feature, the sender needs to set the extended field `ext` of the offline push object `offlinePushInfo`, when sending a message. When the recipient opens the app, the recipient can get `ext` by using different methods provided by mobile phone vendors for obtaining custom content, and then go to the corresponding chat window based on the content of `ext`.

The following example assumes that Denny sends a message to Vinson.
Sender: Denny needs to set `ext` before sending a message.

```
// Denny sets `offlinePushInfo` and specifies `ext` before sending a message
JSONObject jsonObject = new JSONObject();
try {
jsonObject.put("action", "jump to denny");
} catch (JSONException e) {
e.printStackTrace();
}
String extContent = jsonObject.toString();
V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
v2TIMOfflinePushInfo.setExt(extContent.getBytes());
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "vinson", null, V2TIMMessage.V2TIM_PRI
ORITY_DEFAULT, false, v2TIMOfflinePushInfo, new V2TIMSendCallback<V2TIMMessage>() {
@Override
public void onError(int code, String desc) {}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {}
@Override
public void onProgress(int progress) {}
});
```

Recipient: although Vinson's app is not online, it can still receive the notification message pushed offline by mobile phone vendors (for example, OPPO). When Vinson clicks the pushed message, the app is started.

```
// After starting the app, Vinson obtains the custom content from the opened `Activity`
Bundle bundle = intent.getExtras();
Set<String> set = bundle.keySet();
if (set != null) {
for (String key : set) {
```

```
// `key` and `value` correspond to `extKey` and `ext content` set at the sender
String value = bundle.getString(key);
if (value.equals("jump to denny")) {
// Go to the chat window with Denny
...
}
}
}
```

Setting onlineUserOnly so that Messages Can Be Received Only Online

In some scenarios, you may wish that sent messages can only be received by online users or that a recipient is not aware of the message when the recipient is offline. For this purpose, you can set `onlineUserOnly` to `true` when calling sendMessage. After the setting, the sent messages differ from common messages in the following ways:

- Messages cannot be stored offline. That is, the recipient cannot receive messages unless he/she is online.
- Messages do not support multi-device roaming. That is, if the recipient has received messages on one terminal, these messages cannot be received on any other terminal no matter whether these messages are read or not.
- Messages cannot be stored locally. That is, these messages cannot be retrieved from the local historical messages in the cloud.

**Typical example: displaying "The other party is typing..."
In the one-to-one chat scenario, you can call sendMessage to send the "I am typing..." message. When the recipient receives this message, "The other party is typing..." is displayed on the UI. The sample code is as follows:

```
// Send the "I am typing..." message to userA
JSONObject jsonObject = new JSONObject();
try {
jsonObject.put("command", "textInput");
} catch (JSONException e) {
e.printStackTrace();
}
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createCustomMessage(jsonObject.toStr
ing().getBytes());
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "userA", null, V2TIMMessage.V2TIM_PRIO
RITY_DEFAULT, true, v2TIMOfflinePushInfo, new V2TIMSendCallback<V2TIMMessage>() {
```

```
@Override
public void onError(int code, String desc) {}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {}
@Override
public void onProgress(int progress) {}
});
```

Setting "Mute Notifications" for Message Receiving (Only Available in Lite Edition v5.3.425 and Above)

The SDK supports the following types of message receiving options:

- V2TIM_RECEIVE_MESSAGE: messages will be received when the user is online, and offline push notifications will be received when the user is offline.
- V2TIM_NOT_RECEIVE_MESSAGE: messages will not be received no matter whether the user is online or offline.
- V2TIM_RECEIVE_NOT_NOTIFY_MESSAGE: messages will be received when the user is online, and offline push notifications will not be received when the user is offline.

You can call the setC2CReceiveMessageOpt API to set the Mute Notifications option for one-to-one messages and call the setGroupReceiveMessageOpt API to set the Mute Notifications option for group messages.

Recalling Messages

The sender can call the revokeMessage API to recall a successfully sent message. By default, the sender can recall a message that is sent within 2 minutes. You can change the time limit for message recall. For detailed operations, see Message recall settings.
Message recall requires cooperation of the UI code at the recipient side. When the sender recalls a message, the recipient will receive a message recall notification, onRecvMessageRevoked. This notification contains the `msgID` of the recalled message. Based on this `msgID`, you can identify the message that has been recalled and change the corresponding message bubble to the "Message recalled" state on the UI.

The sender recalls a message

```
V2TIMManager.getMessageManager().revokeMessage(v2TIMMessage, new V2TIMCallback() {
@Override
public void onError(int code, String desc) {
// The message fails to be recalled
}
@Override
public void onSuccess() {
// The message is successfully recalled
}
});
```

The recipient learns that the message is recalled

1. Call addAdvancedMsgListener to set the advanced message listener.
2. Call onRecvMessageRevoked to receive the message recall notification.

```
@Override
public void onRecvMessageRevoked(String msgID) {
// `msgList` is the message list on the current chat interface
for (V2TIMMessage msg : msgList) {
if (msg.getMsgID().equals(msgID)) {
// `msg` is the recalled message. You need to change the corresponding message bubble state on th
e UI
}
}
}
```

Adding Read Receipts for Messages

In the one-to-one chat scenario, when the recipient calls the markC2CMessageAsRead API to mark an incoming message as read, the message sender will receive a read receipt, indicating that the recipient has read his/her message.

> Note :
> Currently, only one-to-one chats support the read receipt feature, and group chats do not support this feature. Although the markGroupMessageAsRead API is also available to group chats, the group message senders currently cannot receive any read receipts.

The recipient marks messages as read

```
// Mark messages coming from Haven as read
V2TIMManager.getMessageManager().markC2CMessageAsRead("haven", new V2TIMCallback() {
@Override
public void onError(int code, String desc) {
// Messages fail to be marked as read
}
@Override
public void onSuccess() {
// Messages are successfully marked as read
}
});
```

The sender learns that the messages are read

The event notification of the message receipt is located in the advanced message listener V2TIMAdvancedMsgListener. To learn that the message is already read, the sender must call addAdvancedMsgListener to set the listener. Then, the sender can receive a read receipt from the recipient through the onRecvC2CReadReceipt callback.

```
@Override
public void onRecvC2CReadReceipt(List<V2TIMMessageReceipt> receiptList) {
// The sender may receive multiple read receipts at a time. Therefore, the array callback mode is
used here
for (V2TIMMessageReceipt v2TIMMessageReceipt : receiptList) {
// Message recipient
String userID = v2TIMMessageReceipt.getUserID();
// Time of the read receipt. A message is considered as read if the timestamp in the chat window
is not later than `timestamp` here
long timestamp = v2TIMMessageReceipt.getTimest
}
}
```

Viewing Historical Messages

You can call getC2CHistoryMessageList to obtain historical messages of one-to-one chats, or call getGroupHistoryMessageList to obtain historical messages of group chats. If the network connection of the current device is normal, the IM SDK pulls historical messages from the server by default. If the network connection is unavailable, the IM SDK directly reads historical messages from the local database.

Pulling historical messages by page

The IM SDK supports the feature of pulling historical messages by page. The number of messages pulled per page cannot be too large; otherwise, the pulling speed is affected. We recommend that you pull 20 messages per page.
The following example assumes that historical messages of `groupA` are pulled by page, and the number of messages per page is 20. The sample code is as follows:

```
// The value `null` of `lastMsg` is passed in for the first pulling, indicating that starting fro
m the latest message, a total of 20 messages are pulled
V2TIMManager.getMessageManager().getGroupHistoryMessageList("groupA", 20, null, new V2TIMValueCal
lback<List<V2TIMMessage>>() {
@Override
public void onError(int code, String desc) {
// Message pulling failed
}
@Override
public void onSuccess(List<V2TIMMessage> v2TIMMessages) {
// Messages that are pulled by page are listed from new to old by default
if (v2TIMMessages.size() > 0) {
// Obtain the start message for the next pulling by page
V2TIMMessage lastMsg = v2TIMMessages.get(v2TIMMessages.size() - 1);
// Pull the remaining 20 messages
V2TIMManager.getMessageManager().getGroupHistoryMessageList("groupA", 20, lastMsg, new V2TIMValue
Callback<List<V2TIMMessage>>() {
@Override
public void onError(int code, String desc) {
// Messages fail to be pulled
}
@Override
public void onSuccess(List<V2TIMMessage> v2TIMMessages) {
// Message pulling is completed
}
});
}
}
});
```

In actual scenarios, pulling by page is often triggered by your swipe operation. Each time when you swipe on the message list, pulling by page is triggered once. However, the principle is similar to the preceding sample code. In either case, `lastMsg` specifies the start message for pulling, and `count` specifies the number of messages pulled each time.

Precautions

- The storage period of historical messages is as follows:
  - Trial edition: free storage for 7 days, no extension supported.

  - Pro edition: free storage for 7 days, extension supported.
  - Flagship edition: free storage for 30 days, extension supported.

  It is a value-added service to extend the storage period of historical messages. You can log in to the IM console to modify the relevant configuration. For information about billing, see Value-added Service Pricing.
- Only the meeting group (corresponding to the ChatRoom of the earlier version) supports pulling historical messages of members **before they join the group**.
- Messages in an audio-video group (AVChatRoom) do not support local storage and multi-device roaming. Therefore, the getGroupHistoryMessageList API does not take effect on an audio-video group.

Deleting Messages

You can call the deleteMessages API to delete historical messages. After deletion, historical messages cannot be recovered.

Setting Message Permissions

Allowing message sending and receiving only among friends

By default, the IM SDK does not prevent message sending and receiving among strangers. If you wish that one-to-one messages can be sent or received only among friends, you can log in to the IM console, choose **Feature Configuration** -> **Login and Message** -> **Relationship Check**, and enable **Check Relationship for One-to-One Messages**. After this feature is enabled, you can send messages only to friends. When you try to send messages to strangers, the IM SDK returns the 20009 error code.

Not receiving messages from a specific user

To avoid receiving messages from a specific user, you can blocklist the user or set the Mute Notifications option for messages from the user. After setting the Mute Notifications option, you can change the Mute Notifications status.

Blocklisting a user:

Call the addToBlackList API to add the user to the blocklist.

When a user is blocklisted, the user does not know that he/she is in the blocklist by default. That is, after this user sends a message, the prompt still indicates that the message is sent successfully, but in fact the recipient will not receive the message. If you want a user on the blocklist to know that his/her message fails to be sent, you can log in to the IM console, choose **Feature Configuration** -

> **Login and Message** -> **Blocklist Check**, and disable **Show "Sent successfully" After Sending Messages**. After this feature is disabled, the IM SDK will return the 20007 error code when a user in the blocklist sends a message.

Setting "Mute Notifications" for messages from a specified user (only available in Lite Edition v5.3.425 and above):

Call the setC2CReceiveMessageOpt API to set the message receiving option to `V2TIM_NOT_RECEIVE_MESSAGE`.

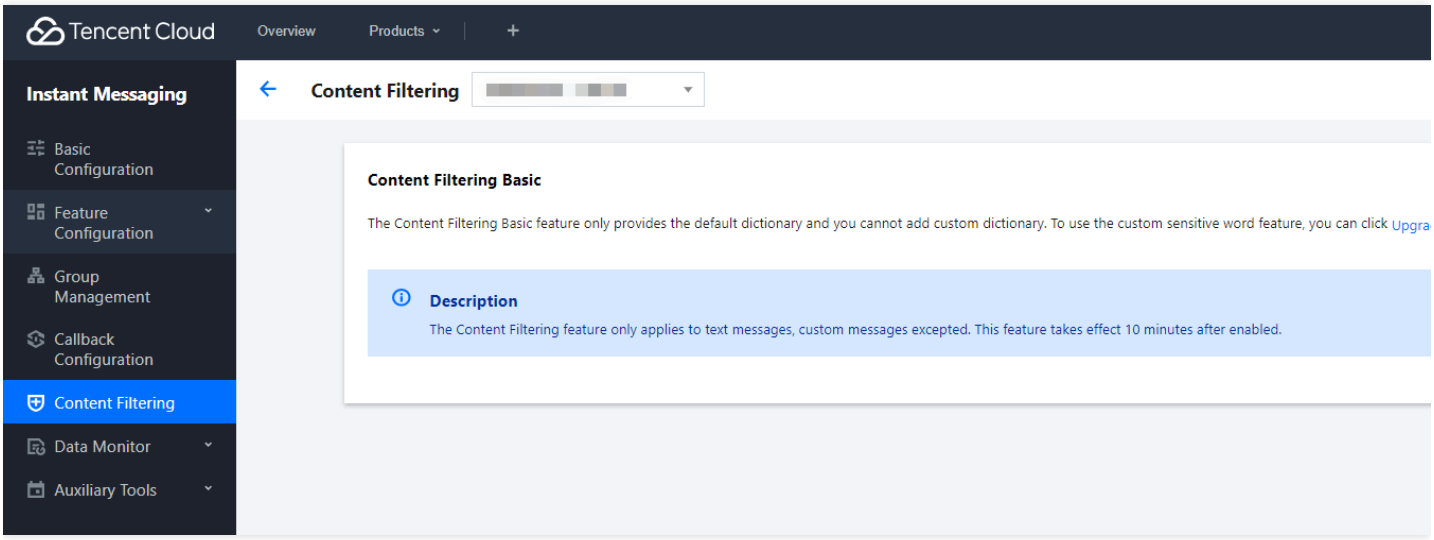
Not receiving messages from a specified group

For Lite Edition v5.3.425 and above, call the setGroupReceiveMessageOpt API to set the message receiving option to `V2TIM_NOT_RECEIVE_MESSAGE`.

For SDKs of other versions, call the `setReceiveMessageOpt` API to set the group message receiving option to `V2TIM_GROUP_NOT_RECEIVE_MESSAGE`.

Filtering Sensitive Words

Text messages sent by the IM SDK are filtered by IM for sensitive words. If a sent text message contains sensitive words, the IM SDK will return the 80001 error code.



FAQs

1. Why am I receiving duplicate messages?

Check the service logic as follows:

- Check whether addSimpleMsgListener is used together with addAdvancedMsgListener. If yes, when text or custom messages are received, both listeners trigger a callback, and consequently duplicate messages are received.
- Check whether the same listener object is added repeatedly. If a listener object is no longer needed, call the corresponding removeSimpleMsgListener or removeAdvancedMsgListener API to remove this listener.

2. Why do the read receipts become invalid after the app is uninstalled and then reinstalled?

In the one-to-one chat scenario, if the recipient calls markC2CMessageAsRead to mark a message as read, the read receipt received by the sender contains `timestamp`. Based on `timestamp`, the SDK determines whether the other party reads the message. Currently, `timestamp` is stored locally, and will be lost when the app is reinstalled.

3. How can I send a message containing multiple `Elem` objects?

You can call appendElem after creating a `Message` object via the `Elem` member of the `Message` object to add the next `Elem` member.
Below is an example of text message + custom message:

```
V2TIMMessage message = V2TIMManager.getMessageManager().createTextMessage("test");
V2TIMCustomElem customElem = new V2TIMCustomElem();
customElem.setData("custom message".getBytes());
message.getTextElem().appendElem(customElem);
```

4. How can I parse a message containing multiple `Elem` objects?

1. Use the `Message` object to parse the first `Elem` object.
2. Use the getNextElem method of the first `Elem` object to obtain the next `Elem` object. If the next `Elem` object exists, the `Elem` object instance is returned. Otherwise, `null` is returned.

```
@Override
public void onRecvNewMessage(V2TIMMessage msg) {
// View the first `Elem` object
int elemType = msg.getElemType();
if (elemType == V2TIMMessage.V2TIM_ELEM_TYPE_TEXT) {
// Text message
V2TIMTextElem v2TIMTextElem = msg.getTextElem();
String text = v2TIMTextElem.getText();
// Check whether `v2TIMTextElem` is followed by more `Elem` objects
V2TIMElem elem = v2TIMTextElem.getNextElem();
while (elem != null) {
```

```
// Identify the `Elem` type. Here, `V2TIMCustomElem` is used as an example
if (elem instanceof V2TIMCustomElem) {
V2TIMCustomElem customElem = (V2TIMCustomElem) elem;
byte[] data = customElem.getData();
}
// Continue to check whether the current `Elem` is followed by more `Elem` objects
elem = elem.getNextElem();
}
// If `elem` is `null`, all `Elem` objects have been parsed
}
}
```

5. How are different types of messages parsed?

It is complex to parse a message. We provide the sample code for parsing different types of messages. You can copy the code to your project, and perform secondary development based on your actual needs.

Message Sending and Receiving (iOS)

Last updated : 2021-11-15 10:03:05

Message Classification

IM messages are classified by message destination into two types: one-to-one messages (also called C2C messages) and group messages.

Message Type	API Keyword	Description
One-to-one message	C2CMessage	Also called C2C message. When sending a one-to-one message, you must specify the UserID of the message recipient, and only the recipient can receive this message.
Group message	GroupMessage	When sending a group message, you must specify the groupID of the target group, and all users in this group can receive this message.

IM messages can also be classified by content into text messages, custom (signaling) messages, image messages, video messages, voice messages, file messages, location messages, combined messages, and group tips.

Message Type	API Keyword	Description
Text message	TextElem	It refers to a common text message. Sensitive words of text messages will be filtered out in the IM service. If a message containing sensitive words is sent, the 80001 error code is returned.
Custom message	CustomElem	It is a section of the binary buffer, which is often used to transfer custom signaling in your application. Its content is not filtered for sensitive words.
Image message	ImageElem	When the IM SDK sends an original image, it automatically generates two images in different sizes. The three images are called the original image, large image, and thumbnail.
Video message	VideoElem	A video message contains a video file and an image.

Message Type	API Keyword	Description
Voice message	SoundElem	Supports displaying a red dot upon playback of the voice message.
File message	FileElem	A file message cannot exceed 100 MB.
Location message	LocationElem	A location message contains three fields: location description, longitude, and latitude.
Combined message	MergerElem	Up to 300 messages can be combined.
Group tip	GroupTipsElem	A group tip is often used to carry a system notification in a group, for example, a notification indicating that a member joins or leaves the group, the group description is modified, or the profile of a group member is changed.

Sending and Receiving Simple Messages

V2TIMManager.h provides a set of simple APIs for sending and receiving messages. Although these APIs can be used to send or receive text messages and custom (signaling) messages, they are easy to use and only a few minutes are needed to complete interfacing.

Sending text and signaling messages (simplified APIs)

To send text messages, call sendC2CTextMessage or sendGroupTextMessage. Text messages will be filtered by IM for sensitive words. If a message containing sensitive words is sent, the 80001 error code is returned.
To send one-to-one custom (signaling) messages, call sendC2CCustomMessage or sendGroupCustomMessage. A custom message is essentially a section of the binary buffer, and is often used to transfer custom signaling in your application. Its content is not filtered for sensitive words.

Receiving text and signaling messages (simplified APIs)

To listen to simple text and signaling messages, call addSimpleMsgListener. To listen to image, video, and voice messages, call addAdvancedMsgListener defined in V2TIMManager + Message.h.

Note :

> Do not use addSimpleMsgListener together with addAdvancedMsgListener; otherwise, logic bugs may occur.

Typical example: sending and receiving on-screen comments in an audio-video group

In the live streaming scenario, it is a common way of communication to send or receive on-screen comments in an audio-video group. This can be easily implemented through the simple message APIs.

1. The anchor can call createGroup to create an audio-video group (AVChatRoom) and record the group ID in the list of rooms in "Broadcasting" state.
2. A viewer can select an anchor that he/she likes, and call joinGroup to join the AVChatRoom created by this anchor.
3. The message sender can call sendGroupTextMessage to send a group text message as the on-screen comment.
4. The message recipient can call addSimpleMsgListener to register a simple message listener, and use the listener callback function onRecvGroupTextMessage to obtain text messages.

"FlyHeart" is an instruction. To configure the "FlyHeart" feature for a live room, perform the steps below:

1. Define a custom message type, for example, a JSON string `{ "command": "favor", "value": 101 }`.
2. Call sendGroupCustomMessage to send a message, and call onRecvGroupCustomMessage to receive the message.

Sending and Receiving Rich Media Messages

Image, video, voice, file, and location messages are called rich media messages. Compared with simple messages, it is more complex to send or receive rich media messages.

- Before sending a rich media message, use the `create` function to create a V2TIMMessage object. Then, call the corresponding `send` API to send this message.
- When receiving the rich media message, check `elemType` and perform secondary parsing on `Elem` obtained based on `elemType`.

Sending rich media messages

The following takes an image message as an example to describe the process of sending a rich media message.

1. The sender calls createImageMessage to create an image message and obtain the V2TIMMessage message object.
2. The sender calls sendMessage to send the previously created message object.

Receiving rich media messages

1. The recipient calls addAdvancedMsgListener to set the advanced message listener.
2. The recipient obtains theV2TIMMessage image message through the onRecvNewMessage listener callback.
3. The recipient parses `elemType` in V2TIMMessage, and performs secondary parsing based on the message type to obtain the content of `Elem` in the message.

Typical example: sending and receiving image messages

The sender creates and sends an image message.

```
// Obtain the local image path
NSString *imagePath = [[NSBundle mainBundle] pathForResource:@"test" ofType:@"png"];
// Create an image message.
V2TIMMessage *msg = [[V2TIMManager sharedInstance] createImageMessage:imagePath];
// Send the image message
[[V2TIMManager sharedInstance] sendMessage:msg receiver:@"userA" groupID:nil
priority:V2TIM_PRIORITY_DEFAULT
onlineUserOnly:NO offlinePushInfo:nil progress:^(uint32_t progress) {
// Image upload progress (0-100)
} succ:^{
// The image message is successfully sent
} fail:^(int code, NSString *msg) {
// The image message fails to be sent
}];
```

The recipient identifies the image message, and parses the message to obtain the original image, large image, and thumbnail contained in the message.

```
- (void)onRecvNewMessage:(V2TIMMessage *)msg {
if (msg.elemType == V2TIM_ELEM_TYPE_IMAGE) {
V2TIMImageElem *imageElem = msg.imageElem;
// An image message contains an image in three different sizes: original image, large image, and
thumbnail. (The SDK automatically generates a large image and a thumbnail.)
// A large image is an image obtained after the original image is proportionally compressed. Afte
r the compression, the smaller one of the height and width is equal to 720 pixels.
// A thumbnail is an image obtained after the original image is proportionally compressed. After
the compression, the smaller one of the height and width is equal to 198 pixels.
NSArray<V2TIMImage *> *imageList = imageElem.imageList;
for (V2TIMImage *timImage in imageList) {
```

```
NSString *uuid = timImage.uuid; // Image ID
V2TIMImageType type = timImage.type; // Image type
int size = timImage.size; // Image size (bytes)
int width = timImage.width; // Image width
int height = timImage.height; // Image height
// Set the image download path `imagePath`. Here, `uuid` can be used as an identifier to avoid re
peated download.
NSString *imagePath = [NSTemporaryDirectory() stringByAppendingPathComponent:
[NSString stringWithFormat: @"testImage%@",timImage.uuid]];
if (![[NSFileManager defaultManager] fileExistsAtPath:imagePath]) {
[timImage downloadImage:imagePath
progress:^(NSInteger curSize, NSInteger totalSize) {
NSLog(@"Image download progress: curSize: %lu,totalSize:%lu",curSize,totalSize);
} succ:^{
NSLog(@"Image download completed");
} fail:^(int code, NSString *msg) {
NSLog(@"Image download failed: code: %d,msg:%@",code,msg);
}];
} else {
// The image already exists
}
}
}
}
```

Note :

For more information on the message parsing sample code, see FAQs > 5. How can I parse different types of messages.

Sending and Receiving Group @ Messages

For a group @ message, the sender can listen to the input of the @ character in the input box and call the group member selection interface. After selection is completed, the input box displays the content in the format of `"@A @B @C....."`, and then the sender can continue to edit the message content and send the message. On the group chat list of the recipient's conversation interface, the identifier `"someone@me"` or `"@all members"` will be displayed to remind the user that the user was mentioned by someone in the group.

Note :

Currently, only text @ messages are supported.

Sending group @ messages

1. The sender listens to the text input box on the chat interface and launches the group member selection interface. After selection is completed, the ID and nickname of the selected member are returned. The ID is used to construct the message object V2TIMMessage, and the nickname is to be displayed in the text box.
2. The sender calls createTextAtMessage of V2TIMManager+Message to create an @ text message and obtain the message object V2TIMMessage.
3. The sender calls sendMessage to send the created @ message object.

Receiving group @ messages

1. During conversation loading and update, you need to call the groupAtInfolist API of V2TIMConversation to obtain the @ data list of the conversation.
2. Obtain and update the @ data type to the @ information of the current conversation through the atType API of the V2TIMGroupAtInfo object on the list.

Typical examples: sending and receiving group @ messages

- **Sending a group @ message:**
  The sender creates and sends a group @ message.

```
// Obtain the ID data of the @ group member
TUITextMessageCellData *text = (TUITextMessageCellData *)data;
NSMutableArray<NSString *> *atUserList = text.atUserList;
// Create a group @ message
V2TIMMessage *atMsg = [[V2TIMManager sharedInstance] createTextAtMessage:text.content atUserList:
atUserList];
// Send the group @ message
[[V2TIMManager sharedInstance] sendMessage:atMsg
receiver:nil
groupID:@"toGroupId"
priority:V2TIM_PRIORITY_DEFAULT
onlineUserOnly:NO
offlinePushInfo:nil
progress:nil
succ:^{
NSLog(@"group @ message is sent successfully");
}
fail:^(int code, NSString *desc) {
```

```
NSLog(@"group @ message fails to be sent");
}];
```

- **Receiving a group @ message:**
  During conversation loading and update, obtain the group @ data list, parse the current @ type, and display the prompt text based on the @ type.

```
// Obtain the group @ data list
NSArray<V2TIMGroupAtInfo *> *atInfoList = conversation.groupAtInfolist;
// Parse the @ type (@me, @all members, @me and @all members)
BOOL atMe = NO; // Whether it's @me
BOOL atAll = NO; // Whether it's @all members
NSString *atTipsStr = @"";
for (V2TIMGroupAtInfo *atInfo in atInfoList) {
switch (atInfo.atType) {
case V2TIM_AT_ME:
atMe = YES;
break;
case V2TIM_AT_ALL:
atAll = YES;
break;
case V2TIM_AT_ALL_AT_ME:
atMe = YES;
atAll = YES;
break;
default:
break;
}
}
// Based on the @ type, prompt:
if (atMe && !atAll) {
atTipsStr = @"[someone@me]";
}
if (!atMe && atAll) {
atTipsStr = @"[@all members]";
}
if (atMe && atAll) {
atTipsStr = @"[someone@me][@all members]";
}
```

Sending and Receiving Combined Messages (Only Available in Lite Edition v5.2.210 and Above)

To implement the combined forward feature similar to that in WeChat, it is necessary to create a combined message according to the original message list, and then send the combined message to the opposite end. After the opposite end receives the combined message, it will parse out the original message list. The display of the combined message also requires the title and abstract information.

- **Sending combined messages**
  Usually when we receive a combined message, the chat screen will look like this:

Chat History of Vinson and Lynx	Title
Vinson: When is the new version of SDK scheduled to go online?	abstract1
Lynx: Next Monday. The specific time depends on the system test result in these two days.	abstract2
Vinson: OK	abstract3

The chat interface will display only the title and abstract information of the combined message, and the combined message list will be displayed only when the user clicks the combined message. When we create a combined message, we need to set not only the combined message list, but also the title and abstract information. The implementation process is as follows:

1. Call the createMergerMessage API to create a combined message.
2. Call the sendMessage API to send the combined message.

- **Receiving combined messages**
  When receiving a combined message V2TIMMessage, use V2TIMMergerElem to get title and abstractList for UI display. When a user clicks the combined message, call the downloadMergerMessage API to download the combine message list for UI display.

Typical examples: sending and receiving combined messages

- **Sending combined messages**
  The sender creates a combined message and sends it.

```
// List of messages that need to be forwarded, which can contain combined messages and cannot con
tain group tips
NSArray *msgs = @[message1,message2...];
// Title of the combined message
NSString *title = @"Chat History of Vinson and Lynx";
// Abstract list of the combined message
```

```
NSArray *abstactList = @[@"abstract1",@"abstract2",@"abstract3"];
// Combined messages are compatible with text. SDKs of early versions do not support combined mes
sages, and they will send a text message with the content `compatibleText` by default.
NSString *compatibleText = @"Please upgrade to the latest version to check the combined message";

// Create a combined message
V2TIMMessage *mergeMessage = [[V2TIMManager sharedInstance] createMergerMessage:msgs title:title
abstractList:abstactList compatibleText:compatibleText];
// Send the combined message to the user Denny
[[V2TIMManager sharedInstance] sendMessage:mergeMessage receiver:@"denny" groupID:nil
priority:V2TIM_PRIORITY_NORMAL onlineUserOnly:NO offlinePushInfo:nil progress:nil succ:nil fail:n
il];
```

- **Receiving combined messages**

  The recipient receives the combined message and parses it:

```
- (void)onRecvNewMessage:(V2TIMMessage *)msg {
if (msg.elemType == V2TIM_ELEM_TYPE_MERGER) {
// Get the combined message elements
V2TIMMergerElem *mergerElem = msg.mergerElem;
// Get the title
NSString *title = mergerElem.title;
// Get the abstract list
NSArray *abstractList = mergerElem.abstractList;
// Download the combined message list when a user clicks the combined message
[msg.mergerElem downloadMergerMessage:^(NSArray<V2TIMMessage *> *msgs) {
// Download succeeded. `msgs` is the combined message list
for (V2TIMMessage *subMsg in msgs) {
// If the combined message list still contains combined messages, you can continue parsing
if (subMsg.elemType == V2TIM_ELEM_TYPE_MERGER) {
V2TIMMergerElem *mergerElem = subMsg.mergerElem;
// Get the title
NSString *title = mergerElem.title;
// Get the abstract list
NSArray *abstractList = mergerElem.abstractList;
// Download the combined message list when a user clicks the combined message
[msg.mergerElem downloadMergerMessage:nil fail:nil];
}
}
} fail:^(int code, NSString *desc) {
// Download failed
}];
}
}
```

Sending Messages That Are Excluded from the Unread Count (Only Available in Lite Edition v5.3.425 and Above)

Normally, when you send one-to-one chat messages and group messages, the messages are included in the unread count (you can get the unread message count of a conversation via the unreadCount API of the V2TIMConversation conversation object). If you need to send messages that are excluded from the unread count, such as tips and control messages, send them as follows:

```
// Create the message object
V2TIMMessage *message = [[V2TIMManager sharedInstance] createTextMessage:@"This is a signaling message"];
// Set the identifier for excluding from the unread message count
message.isExcludedFromUnreadCount = YES;
// Send the message
[[V2TIMManager sharedInstance] sendMessage:msg receiver:@"userA" groupID:nil priority:V2TIM_PRIORITY_DEFAULT onlineUserOnly:YES offlinePushInfo:nil progress:^(uint32_t progress) {
} succ:^{
// The message is sent successfully
} fail:^(int code, NSString *msg) {
// The message fails to be sent
}];
```

Sending Messages That Are Excluded from the Conversation lastMsg (Only Available in Enhanced Edition v5.4.666 and Above)

In certain scenarios, if you need to send messages that are excluded from the conversation `lastMsg`, send them as follows:

```
// Create the message object
V2TIMMessage *message = [V2TIMManager.sharedInstance createTextMessage:content];
// Set the identifier for excluding from the conversation lastMsg
message.isExcludedFromLastMessage = YES;
// Send the message
[V2TIMManager.sharedInstance sendMessage:message receiver:@"userA" groupID:nil priority:V2TIM_PRIORITY_NORMAL onlineUserOnly:NO offlinePushInfo:nil progress:^(uint32_t progress) {
// Sending progress
} succ:^{
// The message is sent successfully
```

```
} fail:^(int code, NSString *desc) {
// The message fails to be sent
}];
```

Setting APNs Offline Push (offlinePushInfo)

When the recipient's app is killed or when the recipient switches to the backend, the IM SDK cannot receive new messages through the normal network connection. In this scenario, the APNs service provided by Apple must be used to notify the recipient of new messages. For more information, see Offline Push (iOS).

Setting the title and voice for APNs offline push

When sending messages, you can use the **offlinePushInfo** field in the sendMessage API to set the title and voice for APNs offline push.

```
// Create and send an image message to groupA, and customize the title and voice for offline pus
h.
NSString *imagePath = [[NSBundle mainBundle] pathForResource:@"test" ofType:@"png"];
// Create an image message.
V2TIMMessage *msg = [[V2TIMManager sharedInstance] createImageMessage:imagePath];
V2TIMOfflinePushInfo *pushInfo = [[V2TIMOfflinePushInfo alloc] init];
// Customize the title and voice for offline push. `01.caf` is a sample file, which must be linke
d to the Xcode project. Here, you only need to enter the file name with the extension.
pushInfo.title = @"Customize the title displayed";
pushInfo.iOSSound = @"01.caf";
[[V2TIMManager sharedInstance] sendMessage:msg receiver:nil groupID:@"groupA" priority:V2TIM_PRIO
RITY_DEFAULT
onlineUserOnly:NO offlinePushInfo:pushInfo progress:^(uint32_t progress) {
} succ:^{
// The message is sent successfully
} fail:^(int code, NSString *msg) {
// The message fails to be sent
}];
```

Clicking a pushed message to go to the corresponding chat window

To implement this feature, the sender needs to set the extended field `ext` of the offline push object `offlinePushInfo`, when sending a message. When the recipient opens the app, the recipient can obtain `ext` through the `didReceiveRemoteNotification` system callback, and then go to the corresponding chat window based on the content of `ext`.

The following example assumes that Denny sends a message to Vinson.

- Sender: Denny needs to set `ext` before sending a message.

```
// Denny sets `offlinePushInfo` and specifies `ext` before sending a message
V2TIMMessage *msg = [[V2TIMManager sharedInstance] createTextMessage:@"Text message"];
V2TIMOfflinePushInfo *info = [[V2TIMOfflinePushInfo alloc] init];
info.ext = @"jump to denny";
[[V2TIMManager sharedInstance] sendMessage:msg receiver:@"vinson" groupID:nil priority:V2TIM_P
RIORITY_DEFAULT
onlineUserOnly:NO offlinePushInfo:info progress:^(uint32_t progress) {
} succ:^{
} fail:^(int code, NSString *msg) {
}];
```

- Recipient: although Vinson's app is not online, it can still receive an APNs offline message notification. When Vinson clicks this notification, the app is started.

```
// Vinson receives the following callback after starting the app.
- (void)application:(UIApplication *)application didReceiveRemoteNotification:(NSDictionary *)
userInfo
fetchCompletionHandler:(void (^)(UIBackgroundFetchResult result))completionHandler {
// Parse `desc`, which is the extended field for online push.
if ([userInfo[@"ext"] isEqualToString:@"jump to denny"]) {
// Go to the chat window with Denny.
}
}
```

Setting `onlineUserOnly` so that Messages Can Be Received Only Online

In some scenarios, you may wish that sent messages can only be received by online users or that a recipient is not aware of the message when the recipient is offline. For this purpose, you can set `onlineUserOnly` to `YES` when calling sendMessage. After the setting, the sent messages differ from common messages in the following ways:

- Messages cannot be stored offline. That is, the recipient cannot receive messages unless he/she is online.
- Messages do not support multi-device roaming. That is, if the recipient has received messages on one terminal, these messages cannot be received on any other terminal no matter whether these messages are read or not.

- Messages cannot be stored locally. That is, these messages cannot be retrieved from the local historical messages in the cloud.

Typical example: displaying "The other party is typing..."
In the one-to-one chat scenario, you can call sendMessage to send the "I am typing..." message. When the recipient receives this message, "The other party is typing..." is displayed on the UI. The sample code is as follows:

```
// Send the "I am typing..." message to userA
NSString *customStr = @"{¥"command¥": ¥"textInput¥"}";
NSData *customData = [customStr dataUsingEncoding:NSUTF8StringEncoding];
V2TIMMessage *msg = [[V2TIMManager sharedInstance] createCustomMessage:customData];
[[V2TIMManager sharedInstance] sendMessage:msg receiver:@"userA" groupID:nil
priority:V2TIM_PRIORITY_DEFAULT onlineUserOnly:YES offlinePushInfo:nil progress:^(uint32_t progre
ss) {
} succ:^{
// The message is sent successfully
} fail:^(int code, NSString *msg) {
// The message fails to be sent
}];
```

Setting "Mute Notifications" for Message Receiving (Only Available in Lite Edition v5.3.425 and Above)

The SDK supports the following types of message receiving options:

- V2TIM_RECEIVE_MESSAGE: messages will be received when the user is online, and offline push notifications will be received when the user is offline.
- V2TIM_NOT_RECEIVE_MESSAGE: messages will not be received no matter whether the user is online or offline.
- V2TIM_RECEIVE_NOT_NOTIFY_MESSAGE: messages will be received when the user is online, and offline push notifications will not be received when the user is offline.

You can call the setC2CReceiveMessageOpt API to set the Mute Notifications option for one-to-one messages and call the setGroupReceiveMessageOpt API to set the Mute Notifications option for group messages.

Recalling Messages

The sender can call the revokeMessage API to recall a successfully sent message. By default, the sender can recall a message that is sent within 2 minutes. You can change the time limit for message recall. For detailed operations, see Message recall settings.
Message recall requires cooperation of the UI code at the recipient side. When the sender recalls a message, the recipient will receive a message recall notification, onRecvMessageRevoked. This notification contains the `msgID` of the recalled message. Based on this `msgID`, you can identify the message that has been recalled and change the corresponding message bubble to the "Message recalled" state on the UI.

The sender recalls a message

```
[[V2TIMManager sharedInstance] revokeMessage:msg succ:^{
// The message is successfully recalled
} fail:^(int code, NSString *msg) {
// The message fails to be recalled
}];
```

The recipient learns that the message is recalled

1. Call addAdvancedMsgListener to set the advanced message listener.
2. Call onRecvMessageRevoked to receive the message recall notification.

```
- (void)onRecvMessageRevoked:(NSString *)msgID {
// `msgList` is the message list on the current chat interface
for(V2TIMMessage *msg in msgList){
if ([msg.msgID isEqualToString:msgID]) {
// `msg` is the recalled message. You need to change the corresponding message bubble state on th
e UI.
}
}
}
```

Adding Read Receipts for Messages

In the one-to-one chat scenario, when the recipient calls the markC2CMessageAsRead API to mark an incoming message as read, the message sender will receive a read receipt, indicating that the recipient has read his/her message.

Note :

> Currently, only one-to-one chats support the read receipt feature, and group chats do not support this feature. Although the markGroupMessageAsRead API is also available to group chats, the group message senders currently cannot receive any read receipts.

The recipient marks messages as read

```
// Mark messages coming from Haven as read
[[V2TIMManager sharedInstance] markC2CMessageAsRead:@"haven" succ:^{
} fail:^(int code, NSString *msg) {
}];
```

The sender learns that the messages are read

The event notification of the message receipt is located in the advanced message listener V2TIMAdvancedMsgListener. To learn that a message is already read, the sender must call addAdvancedMsgListener to set the listener. Then, the sender can receive a read receipt from the recipient through the onRecvC2CReadReceipt callback.

```
- (void)onRecvC2CReadReceipt:(NSArray<V2TIMMessageReceipt *> *)receiptList {
// The sender may receive multiple read receipts at a time. Therefore, the array callback mode is
used here.
for (V2TIMMessageReceipt *receipt in receiptList) {
// Message recipient
NSString * receiver = receipt.userID;
// Time of the read receipt. A message is considered as read if the timestamp in the chat window
is not later than `timestamp` here
time_t timestamp = receipt.timestamp;
}
}
@end
```

Viewing Historical Messages

You can call getC2CHistoryMessageList to obtain historical messages of one-to-one chats, or call getGroupHistoryMessageList to obtain historical messages of group chats. If the network connection of the current device is normal, the IM SDK pulls historical messages from the server by default. If the network connection is unavailable, the IM SDK directly reads historical messages from the local database.

Pulling historical messages by page

The IM SDK supports the feature of pulling historical messages by page. The number of messages pulled per page cannot be too large; otherwise, the pulling speed is affected. We recommend that you pull 20 messages per page.
The following example assumes that historical messages of `groupA` are pulled by page, and the number of messages per page is 20. The sample code is as follows:

```
// The value `nil` of `lastMsg` is passed in for the first pulling, indicating that starting from the latest message, a total of 20 messages are pulled
[[V2TIMManager sharedInstance] getGroupHistoryMessageList:@"groupA" count:20
lastMsg:nil succ:^(NSArray<V2TIMMessage *> *msgs) {
// Messages that are pulled by page are listed from new to old by default
if (msgs.count > 0) {
// Obtain the start message for the next pulling by page
V2TIMMessage *lastMsg = msgs.lastObject;
// Pull the remaining 20 messages
[[V2TIMManager sharedInstance] getGroupHistoryMessageList:@"groupA" count:20
lastMsg:lastMsg succ:^(NSArray<V2TIMMessage *> *msgs) {
// Message pulling is completed
} fail:^(int code, NSString *msg) {
// Messages fail to be pulled
}];
}
} fail:^(int code, NSString *msg) {
// Messages fail to be pulled
}];
```

In actual scenarios, pulling by page is often triggered by your swipe operation. Each time when you swipe on the message list, pulling by page is triggered once. However, the principle is similar to the preceding sample code. In either case, `lastMsg` specifies the start message for pulling, and `count` specifies the number of messages pulled each time.

Precautions

- The storage period of historical messages is as follows:
  - Trial edition: free storage for 7 days, no extension supported.
  - Pro edition: free storage for 7 days, extension supported.
  - Flagship edition: free storage for 30 days, extension supported.

  It is a value-added service to extend the storage period of historical messages. You can log in to the IM console to modify the relevant configuration. For information about billing, see Value-added Service Pricing.
- Only the meeting group (corresponding to the ChatRoom of the earlier version) supports pulling historical messages of members **before they join the group**.

- Messages in an audio-video group (AVChatRoom) do not support local storage and multi-device roaming. Therefore, the getGroupHistoryMessageList API does not take effect on an audio-video group.

Deleting Messages

You can call the deleteMessages API to delete historical messages. After deletion, historical messages cannot be recovered.

Setting Message Permissions

Allowing message sending and receiving only among friends

By default, the IM SDK does not prevent message sending and receiving among strangers. If you wish that one-to-one messages can be sent or received only among friends, you can log in to the IM console, choose **Feature Configuration** -> **Login and Message** -> **Relationship Check**, and enable **Check Relationship for One-to-One Messages**. After this feature is enabled, you can send messages only to friends. When you try to send messages to strangers, the IM SDK returns the 20009 error code.

Not receiving messages from a specific user

To avoid receiving messages from a specific user, you can blocklist the user or set the Mute Notifications option for messages from the user. After setting the Mute Notifications option, you can change the Mute Notifications status.

Blocklisting a user:

Call the addToBlackList API to add the user to the blocklist. When the user is blocklisted, the user does not know that he/she is in the blocklist by default. That is, after this user sends a message, the prompt still indicates that the message is sent successfully, but in fact the recipient will not receive the message. If you want a user on the blocklist to know that his/her message fails to be sent, you can log in to the IM console, choose **Feature Configuration** -> **Login and Message** -> **Blocklist Check**, and disable **Show "Sent successfully" After Sending Messages**. After this feature is disabled, the IM SDK will return the 20007 error code when a user in the blocklist sends a message.

Setting "Mute Notifications" for messages from a specified user (only available in Lite Edition v5.3.425 and above):

Call the setC2CReceiveMessageOpt API to set the message receiving option to `V2TIM_NOT_RECEIVE_MESSAGE` .

Not receiving messages from a specified group

For Lite Edition v5.3.425 and above, call the setGroupReceiveMessageOpt API to set the message receiving option to `V2TIM_NOT_RECEIVE_MESSAGE`.
For SDKs of other versions, call the `setReceiveMessageOpt` API to set the message receiving option to `V2TIM_GROUP_NOT_RECEIVE_MESSAGE`.

Filtering Sensitive Words

Text messages sent by the IM SDK are filtered by IM for sensitive words. If a sent text message contains sensitive words, the IM SDK will return the 80001 error code.

FAQs

1. Why am I receiving duplicate messages?

- Check whether addSimpleMsgListener is used together with addAdvancedMsgListener. If yes, when text or custom messages are received, both listeners trigger callback, and consequently duplicate messages are received.
- Check whether the same listener object is added repeatedly. If a listener object is no longer needed, call the corresponding removeSimpleMsgListener or removeAdvancedMsgListener API to remove this listener.

2. Why do the read receipts become invalid after the app is uninstalled and then reinstalled?

In the one-to-one chat scenario, if the recipient calls markC2CMessageAsRead to mark a message as read, the read receipt received by the sender contains `timestamp`. Based on `timestamp`, the SDK determines whether the other party reads the message. Currently, `timestamp` is stored locally, and will be lost when the app is reinstalled.

3. How can I send a message containing multiple `Elem`?

You can call appendElem after creating a `Message` object via the `Elem` member of the `Message` object to add the next `Elem` member.
Below is an example of text message + custom message:

```
V2TIMMessage *msg = [[V2TIMManager sharedInstance] createTextMessage:@"text"];
V2TIMCustomElem *customElem = [[V2TIMCustomElem alloc] init];
```

```
customElem.data = [@"custom message" dataUsingEncoding:NSUTF8StringEncoding];
[msg.textElem appendElem:customElem];
```

4. How can I parse a message containing multiple `Elem` objects?

1. Use the `Message` object to parse the first `Elem` object.
2. Use the nextElem method of the first `Elem` object to obtain the next `Elem` object. If the next `Elem` object exists, the `Elem` object instance is returned. Otherwise, `nil` is returned.

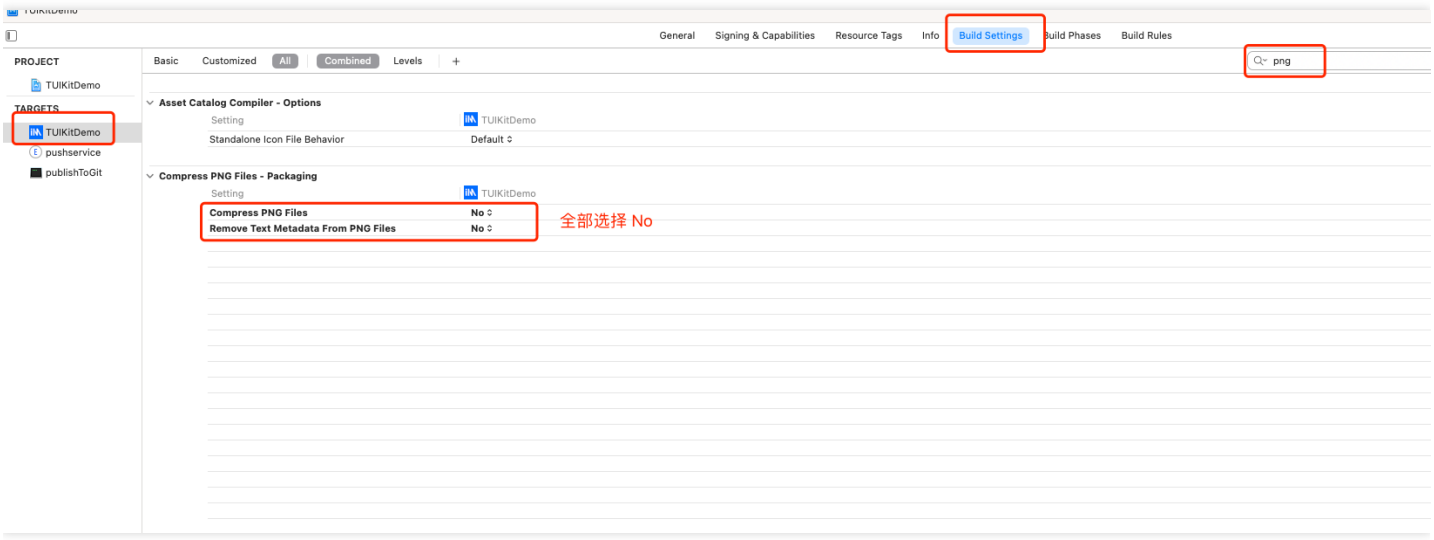
```
- (void)onRecvNewMessage:(V2TIMMessage *)msg {
// View the first `Elem` object
if (msg.elemType == V2TIM_ELEM_TYPE_TEXT) {
V2TIMTextElem *textElem = msg.textElem;
NSString *text = textElem.text;
NSLog(@"Text information: %@", text);
// Check whether `textElem` is followed by more `Elem` objects
V2TIMElem *elem = textElem.nextElem;
while (elem != nil) {
// Identify the `Elem` type
if ([elem isKindOfClass:[V2TIMCustomElem class]]) {
V2TIMCustomElem *customElem = (V2TIMCustomElem *)elem;
NSData *customData = customElem.data;
NSLog(@"Custom information: %@",customData);
}
// Continue to check whether the current `Elem` is followed by more `Elem` objects
elem = elem.nextElem;
}
// If `elem` is `nil`, all `Elem` objects have been parsed
}
}
```

5. How are different types of messages parsed?

It is complex to parse a message. We provide the sample code for parsing different types of messages. You can copy the code to your project, and perform secondary development based on your actual needs.

6. Why did sending the PNG images in an Xcode project fail?

When you use a PNG image in an Xcode project to create an image message and send it, a sending failure message will be displayed. The reason is that Xcode compresses the PNG images in projects and modifies the file headers by default, and as a result, the images cannot be identified by IM. To solve the problem, configure your Xcode project as shown in the following figure.



Message Sending and Receiving (Web & Mini Program)

Last updated : 2021-05-19 16:37:03

This document describes how to send and receive messages through Web and mini programs.

Sending Messages

Creating a text message

This API is used to create a text message. It returns a message instance. If you need to send a text message, call sendMessage to send the message instance.

API name

```
tim.createTextMessage(options)
```

Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Attributes	Default	Description
`to`	`String`	-	-	userID or group the recipient
`conversationType`	`String`	-	-	Conversation ty Valid values: `TIM.TYPES.CONV` (C2C conversat and `TIM.TYPES.CONV` (group convers
`priority`	`String`	`<optional>`	`TIM.TYPES.MSG_PRIORITY_NORMAL`	Message priorit
`payload`	`Object`	-	-	Message conte container

`payload` has the following properties:

Name	Type	Description
`text`	`String`	Text content of the message.

Example

```
// Send a text message. This process is the same for web applications and WeChat Mini Programs.
// 1. Create a message instance. The instance returned by the API can be displayed on the screen.
let message = tim.createTextMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
// Message priority applicable to group chats (supported since v2.4.2). If the message sending fr
equency of a group exceeds the limit, the backend delivers high-priority messages first. For more
information, see https://intl.cloud.tencent.com/document/product/1047/33526.
// Valid values: TIM.TYPES.MSG_PRIORITY_HIGH, TIM.TYPES.MSG_PRIORITY_NORMAL (default), TIM.TYPES.
MSG_PRIORITY_LOW, and TIM.TYPES.MSG_PRIORITY_LOWEST
// priority: TIM.TYPES.MSG_PRIORITY_NORMAL,
payload: {
text: 'Hello world!'
}
});
// 2. Send a message.
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// The message is sent successfully.
console.log(imResponse);
}).catch(function(imError){
// The message fails to be sent.
console.warn('sendMessage error:', imError);
});
```

Response

This API returns a message instance Message.

Creating an image message

This API is used to create an image message. It returns a message instance. If you need to send an image message, call sendMessage to send the message instance.

> 注意：
> File objects are supported since v2.3.1. If you need to use file objects, upgrade your SDK to v2.3.1 or later.

API

```
tim.createImageMessage(options)
```

Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Attributes	Default	Description
`to`	`String`	-	-	Message reci
`conversationType`	`String`	-	-	Conversation Valid values: `TIM.TYPES.CO` and `TIM.TYPES.CO`
`priority`	`String`	`<optional>`	`TIM.TYPES.MSG_PRIORITY_NORMAL`	Message pric
`payload`	`Object`	-	-	Message con container
`onProgress`	`function`	-	-	Callback func used to quer upload progr

`payload` has the following properties:

Name	Type	Description
file	`HTMLInputElement` or `Object`	It is used to select a DOM node or file object of the image in a web application, or the `success` callback parameter for the `wx.chooseImage` API of a WeChat Mini Program. The SDK reads the data contained in this parameter and uploads the image.

Web example

```
// Example 1 for sending an image message in a web application - Passing into a DOM node
// 1. Create a message instance. The instance returned by the API can be displayed on the screen.
let message = tim.createImageMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
// Message priority applicable to group chats (supported since v2.4.2). If the message sending fr
equency of a group exceeds the limit, the backend delivers high-priority messages first. For more
```

```
information, see Message Priority and Frequency Control.
// Valid values: TIM.TYPES.MSG_PRIORITY_HIGH, TIM.TYPES.MSG_PRIORITY_NORMAL (default), TIM.TYPES.
MSG_PRIORITY_LOW, and TIM.TYPES.MSG_PRIORITY_LOWEST
// priority: TIM.TYPES.MSG_PRIORITY_NORMAL,
payload: {
file: document.getElementById('imagePicker'),
},
onProgress: function(event) { console.log('file uploading:', event) }
});
// 2. Send a message.
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// The message is sent successfully.
console.log(imResponse);
}).catch(function(imError) {
// The message fails to be sent.
console.warn('sendMessage error:', imError);
});
```

```
// Example 2 for sending an image message in a web application - Passing in a file object
// Add a message input box with the ID set to "testPasteInput", for example, <input type="text" i
d="testPasteInput" placeholder="Take a screenshot and paste it in the input box" size="30" />
document.getElementById('testPasteInput').addEventListener('paste', function(e) {
let clipboardData = e.clipboardData;
let file;
let fileCopy;
if (clipboardData && clipboardData.files && clipboardData.files.length > 0) {
file = clipboardData.files[0];
// After the image message is successfully sent, the content pointed by `file` may be cleared by
the browser. If you has extra rendering requirements, copy the data in advance.
fileCopy = file.slice();
}
if (typeof file === 'undefined') {
console.warn('file is undefined. Check compatibility of the code or browser.');
return;
}
// 1. Create a message instance. The instance returned by the API can be displayed on the screen.
let message = tim.createImageMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
payload: {
file: file
},
onProgress: function(event) { console.log('file uploading:', event) }
});
// 2. Send a message.
let promise = tim.sendMessage(message);
```

```
promise.then(function(imResponse) {
// The message is sent successfully.
console.log(imResponse);
}).catch(function(imError) {
// The message fails to be sent.
console.warn('sendMessage error:', imError);
});
});
```

Mini Program example

```
// Send an image message in a Mini Program.
// 1. Select an image.
wx.chooseImage({
sourceType: ['album'], // Select an image from the album.
count: 1, // You can select only one image. The SDK does not support sending multiple images at a time.
success: function (res) {
// 2. Create a message instance. The instance returned by the API can be displayed on the screen.
let message = tim.createImageMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
payload: { file: res },
onProgress: function(event) { console.log('file uploading:', event) }
});
// 3. Send the image.
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// The message is sent successfully.
console.log(imResponse);
}).catch(function(imError){
// The message fails to be sent.
console.warn('sendMessage error:', imError);
});
}
})
```

Response

This API returns a message instance Message.

Creating a voice message

This API is used to create a voice message. It returns a message instance. If you need to send a voice message, call sendMessage. Currently, createAudioMessage is applicable only to WeChat Mini Programs.

API

```
tim.createAudioMessage(options)
```

Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Attributes	Default	Description
`to`	`String`	-	-	Message recipi
`conversationType`	`String`	-	-	Conversation t Valid values: `TIM.TYPES.CONV` and `TIM.TYPES.CONV`
`priority`	`String`	`<optional>`	`TIM.TYPES.MSG_PRIORITY_NORMAL`	Message priorit
`payload`	`Object`	-	-	Message conte container

`payload` has the following properties:

Name	Type	Description
file	`Object`	Recorded file.

Mini Program example

```
// Example: Record a voice message by using the official WeChat RecorderManager. For more informa
tion, see RecorderManager.start(Object object).
// 1. Obtain the globally unique RecorderManager.
const recorderManager = wx.getRecorderManager();


// Some parameters related to recording
const recordOptions = {
duration: 60000, // Recording duration in ms. The maximum value is 600000 ms, that is, 10 minute
s.
sampleRate: 44100, // Sample rate.
numberOfChannels: 1, // Number of recording channels.
encodeBitRate: 192000, // Encoding rate.
format: 'aac' // Format of the voice message. A voice message created using this format can be us
ed in all IM platforms, including the Android, iOS, WeChat Mini Programs, and web platforms.
};
```

```
// 2.1 Listen for voice recording errors.
recorderManager.onError(function(errMsg) {
console.warn('recorder error:', errMsg);
});
// 2.2 Listen for the recording end event. After recording is completed, call createAudioMessage
to create a voice message instance.
recorderManager.onStop(function(res) {
console.log('recorder stop', res);

// 4. Create a message instance. The instance returned by the API can be displayed on the screen.
const message = tim.createAudioMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
payload: {
file: res
}
});

// 5. Send the message.
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// The message is sent successfully.
console.log(imResponse);
}).catch(function(imError) {
// The message fails to be sent.
console.warn('sendMessage error:', imError);
});
});

// 3. Start recording.
recorderManager.start(recordOptions);
```

Response

This API returns a message instance Message.

Creating a file message

This API is used to create a file message. It returns a message instance. If you need to send a file message, call sendMessage to send the message instance.

注意：

! File objects are supported since v2.3.1. If you need to use file objects, upgrade your SDK to v2.3.1 or later.

! Since v2.4.0, the maximum size of a file to upload is 100 MB.
WeChat Mini Programs currently do not support the selection of files. Therefore, this API is not applicable to WeChat Mini Programs.

API

```
tim.createFileMessage(options)
```

Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Attributes	Default	Description
`to`	`String`	-	-	userID or gro the recipient
`conversationType`	`String`	-	-	Conversation Valid values: `TIM.TYPES.CO` (C2C convers and `TIM.TYPES.CO` (group conve
`priority`	`String`	`<optional>`	`TIM.TYPES.MSG_PRIORITY_NORMAL`	Message pric
`payload`	`Object`	-	-	Message con container
`onProgress`	`function`	-	-	Callback func used to quer upload progr

`payload` has the following properties:

Name	Type	Description
file	`HTMLInputElement`	It is used to select a DOM node or file object of the image in a web application. The SDK reads the data contained in this parameter and uploads the file.

Example

```
// Example 1 for sending a file message in a web application - Passing in a DOM node
// 1. Create a file message instance. The instance returned by the API can be displayed on the sc
reen.
let message = tim.createFileMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
// Message priority applicable to group chats (supported since v2.4.2). If the message sending fr
equency of a group exceeds the limit, the backend delivers high-priority messages first. For more
information, see Message Priority and Frequency Control.
// Valid values: TIM.TYPES.MSG_PRIORITY_HIGH, TIM.TYPES.MSG_PRIORITY_NORMAL (default), TIM.TYPES.
MSG_PRIORITY_LOW, and TIM.TYPES.MSG_PRIORITY_LOWEST
// priority: TIM.TYPES.MSG_PRIORITY_NORMAL,
payload: {
file: document.getElementById('filePicker'),
},
onProgress: function(event) { console.log('file uploading:', event) }
});
// 2. Send a message.
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// The message is sent successfully.
console.log(imResponse);
}).catch(function(imError) {
// The message fails to be sent.
console.warn('sendMessage error:', imError);
});
```

```
// Example 2 for sending a file message in a web application - Passing in a file object
// Add a message input box with the ID set to "testPasteInput", for example, <input type="text" i
d="testPasteInput" placeholder="Take a screenshot and paste it in the input box" size="30" />
document.getElementById('testPasteInput').addEventListener('paste', function(e) {
let clipboardData = e.clipboardData;
let file;
let fileCopy;
if (clipboardData && clipboardData.files && clipboardData.files.length > 0) {
file = clipboardData.files[0];
// After the image message is successfully sent, the content pointed by `file` may be cleared by
the browser. If you has extra rendering requirements, copy the data in advance.
fileCopy = file.slice();
}
if (typeof file === 'undefined') {
console.warn('file is undefined. Check compatibility of the code or browser.');
return;
}
// 1. Create a message instance. The instance returned by the API can be displayed on the screen.
let message = tim.createFileMessage({
```

```
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
payload: {
file: file
},
onProgress: function(event) { console.log('file uploading:', event) }
});
// 2. Send a message.
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// The message is sent successfully.
console.log(imResponse);
}).catch(function(imError) {
// The message fails to be sent.
console.warn('sendMessage error:', imError);
});
});
```

Response

This API returns a message instance Message.

Creating a custom message

This API is used to create a custom message. It returns a message instance. If you need to send a custom message, call sendMessage to send the message instance.
If the SDK does not provide the capability you need, use custom messages to customize features, for example, the dice rolling feature.

API

```
tim.createCustomMessage(options)
```

Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Attributes	Default	Description
`to`	`String`	-	-	userID or group the recipient

Name	Type	Attributes	Default	Description
`conversationType`	`String`	-	-	Conversation t Valid values: `TIM.TYPES.CONV` (C2C conversat and `TIM.TYPES.CONV` (group convers
`priority`	`String`	`<optional>`	`TIM.TYPES.MSG_PRIORITY_NORMAL`	Message priorit
`payload`	`Object`	-	-	Message conte container

`payload` has the following properties:

Name	Type	Description
`data`	`String`	Data field of the custom message
`description`	`String`	Description field of the custom message
`extension`	`String`	Extension field of the custom message

Example

```
// Example: Implement the dice rolling feature by using a custom message.
// 1. Customize a random function.
function random(min, max) {
return Math.floor(Math.random() * (max - min + 1) + min);
}
// 2. Create a message instance. The instance returned by the API can be displayed on the screen.
let message = tim.createCustomMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
// Message priority applicable to group chats (supported since v2.4.2). If the message sending fr
equency of a group exceeds the limit, the backend delivers high-priority messages first. For more
information, see Message Priority and Frequency Control.
// Valid values: TIM.TYPES.MSG_PRIORITY_HIGH, TIM.TYPES.MSG_PRIORITY_NORMAL (default), TIM.TYPES.
MSG_PRIORITY_LOW, and TIM.TYPES.MSG_PRIORITY_LOWEST
// priority: TIM.TYPES.MSG_PRIORITY_HIGH,
payload: {
data: 'dice', // Identify the message as a dice message.
description: String(random(1,6)), // Obtain the outcome of dice rolling.
```

```
extension: ''
}
});
// 3. Send the message.
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// The message is sent successfully.
console.log(imResponse);
}).catch(function(imError) {
// The message fails to be sent.
console.warn('sendMessage error:', imError);
});
```

Response

This API returns a message instance Message.

Creating a video message

This API is used to create a video message. It returns a message instance. If you need to send a video message, call sendMessage to send the message instance.

注意：

- This API requires SDK v2.2.0 or later.
- createVideoMessage is applicable to WeChat Mini Programs and can be used in web applications if the SDK version is v2.6.0 or later.
- You can use WeChat Mini Programs to record a video message, or select a video from your album. However, Mini Programs do not return a thumbnail of the video. To improve user experience, the SDK sets a default thumbnail for each video message during creation. If you do not wish to display the default thumbnail, skip the information related to the thumbnail during rendering.
- To make your video messages compatible with all platforms, use the latest TUIKit or SDK to develop your mobile client.

API

```
tim.createVideoMessage(options)
```

Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Attributes	Default	Description
`to`	`String`	-	-	Message recipi
`conversationType`	`String`	-	-	Conversation t Valid values: `TIM.TYPES.CONV` and `TIM.TYPES.CONV`
`priority`	`String`	`<optional>`	`TIM.TYPES.MSG_PRIORITY_NORMAL`	Message priorit
`payload`	`Object`	-	-	Message conte container

`payload` has the following properties:

Name	Type	Description
`file`	`HTMLInputElement`, `File`, or `Object`	This is used to select a DOM node or file object of the video file in a web application, or to record a video file or select a video file from the album in a WeChat Mini Program. The SDK reads and uploads the data contained in this parameter.

Example

```
// Example of sending a video message in a Mini Programwx.chooseVideo
// 1. Call the Mini Program API to select a video file.
wx.chooseVideo({
sourceType: ['album', 'camera'], // Source of the video file, which is the album or camera
maxDuration: 60, // Maximum duration, which is 60s
camera: 'back', // Rear camera
success (res) {
// 2. Create a message instance. The instance returned by the API can be displayed on the screen.
let message = tim.createVideoMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
payload: {
file: res
},
onProgress: function(event) { console.log('video uploading:', event) }
})
// 3. Send the message.
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
```

```
// The message is sent successfully.
console.log(imResponse);
}).catch(function(imError) {
// The message fails to be sent.
console.warn('sendMessage error:', imError);
});
}
})

// Example of sending a video message in a web application (supported since v2.6.0):
// 1. Obtain the video file, and pass in the DOM node.
// 2. Create a message instance.
const message = tim.createVideoMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
payload: {
file: document.getElementById('videoPicker') // Alternatively, use event.target.
},
onProgress: function(event) { console.log('file uploading:', event) }
});
// 3. Send the message.
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// The message is sent successfully.
console.log(imResponse);
}).catch(function(imError) {
// The message fails to be sent.
console.warn('sendMessage error:', imError);
});
```

Response

This API returns a message instance Message.

Creating an emoji message

This API is used to create an emoji message. It returns a message instance. If you need to send an emoji message, call sendMessage to send the message instance.

> 注意：
> This API requires SDK v2.3.1 or later.

API

```
tim.createFaceMessage(options)
```

Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Attributes	Default	Description
`to`	`String`	-	-	userID or group the recipient
`conversationType`	`String`	-	-	Conversation ty Valid values: `TIM.TYPES.CONV` (C2C conversat and `TIM.TYPES.CONV` (group convers
`priority`	`String`	`<optional>`	`TIM.TYPES.MSG_PRIORITY_NORMAL`	Message priorit
`payload`	`Object`	-	-	Message conte container

`payload` has the following properties:

Name	Type	Description
`index`	`Number`	Emoji index, which is customized by the user
`data`	`String`	Extra data

Example

```
// Send an emoji. This process is the same for web applications and WeChat Mini Programs.
// 1. Create a message instance. The instance returned by the API can be displayed on the screen.
let message = tim.createFaceMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
// Message priority applicable to group chats (supported since v2.4.2). If the message sending fr
equency of a group exceeds the limit, the backend delivers high-priority messages first. For more
information, see Message Priority and Frequency Control.
// Valid values: TIM.TYPES.MSG_PRIORITY_HIGH, TIM.TYPES.MSG_PRIORITY_NORMAL (default), TIM.TYPES.
MSG_PRIORITY_LOW, and TIM.TYPES.MSG_PRIORITY_LOWEST
// priority: TIM.TYPES.MSG_PRIORITY_NORMAL,
payload: {
index: 1, // The number indicates the emoji index, which is customized by the user.
```

```
data: 'tt00' // The string indicates the extra data.
}
});
// 2. Send a message.
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// The message is sent successfully.
console.log(imResponse);
}).catch(function(imError) {
// The message fails to be sent.
console.warn('sendMessage error:', imError);
});
```

Response

This API returns a message instance Message.

Sending a message

This API is used to send a message. Before sending a message, call the following APIs to create a message instance and then call this API to send the message instance.

- createTextMessage
- createImageMessage
- createAudioMessage
- createVideoMessage
- createCustomMessage
- createFaceMessage
- createFileMessage

> 注意：
> The SDK needs to be `ready` in order to call this API to send message instances. The following events can be listened for to learn the SDK status:

- TIM.EVENT.SDK_READY: this event is triggered when the SDK status is ready.
- TIM.EVENT.SDK_NOT_READY: this event is triggered when the SDK status is not ready.

TIM.EVENT.MESSAGE_RECEIVED must be listened for in order to receive the pushed new one-to-one messages, group messages, group notifications, or system group notifications.
Messages sent by this API do not trigger TIM.EVENT.MESSAGE_RECEIVED. Messages sent by the same account from other clients (or through the RESTful API) trigger

TIM.EVENT.MESSAGE_RECEIVED. Offline push is applicable only to Android or iOS terminals and is not supported by web applications or WeChat Mini Programs.

API

```
tim.sendMessage(options)
```

Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	**Attributes**	Description
`message`	`Message`	-	Message instance
`options`	`Object`	`optional`	Message sending option (message content container)

`options` is described in the following table.

Name	Type	**Attributes**	Description
`onlineUserOnly`	`Boolean`	`optional`	Whether the message is sent to online users only. This parameter is supported since v2.6.4. The default value is false. If this parameter is set to true, the message is not stored in the roaming server, not counted as an unread message, and not pushed to the recipient offline. It is applicable to sending of unimportant prompts or messages, such as broadcast notifications. This parameter does not apply to messages sent in an AVChatRoom.
`offlinePushInfo`	`Object`	`optional`	Offline push information. This parameter is supported since v2.6.4. For more information, see Offline Push.

`offlinePushInfo` is described in the following table.

Name	Type	**Attributes**	Description
`disablePush`	`Boolean`	`optional`	true: offline push is disabled. false (default): offline push is enabled.
`title`	`String`	`optional`	Title of offline push. This parameter is used by both iOS and Android.

Name	Type	**Attributes**	Description
`description`	`String`	`optional`	Offline push content. This field will overwrite the offline push display text of the message instance. If the sent message is a custom message, this field overwrites `message.payload.description`. If both `description` and `message.payload.description` are left unspecified, the recipient cannot receive the offline push notification of the custom message.
`extension`	`String`	`optional`	Passthrough content of offline push
`ignoreIOSBadge`	`Boolean`	`optional`	Whether the badge count is ignored (applicable to iOS only). If this parameter is set to true, the unread count icon of the application will not increase when the message is received by an iOS device.
`androidOPPOChannelID`	`String`	`optional`	Channel ID for offline push configured on OPPO mobile phones that run Android 8.0 or later.

Example

```
// If the recipient is offline, the message will be stored in the roaming server and pushed offli
ne (when the recipient's application switches to the backend or the process is killed). The defau
lt title and content of offline push are kept.
// For more information about offline push, see Offline Push.
tim.sendMessage(message);
// The message sending option is supported since v2.6.4.
tim.sendMessage(message, {
onlineUserOnly: true// If the recipient is offline, the message is neither stored in the roaming
server nor pushed offline.
});
// The message sending option is supported since v2.6.4.
tim.sendMessage(message, {
offlinePushInfo: {
onlineUserOnly: true// If the recipient is offline, the message is stored in the roaming server,
but is not pushed offline.
}
});
// The message sending option is supported since v2.6.4.
tim.sendMessage(message, {
```

```
// If the recipient is offline, the message will be stored in the roaming server and pushed offli
ne (when the recipient's application switches to the backend or the process is killed). The title
and content of offline push can be customized at the access end.
offlinePushInfo: {
title: '', // Title of offline push
description: '', // Content of offline push
androidOPPOChannelID: '' // Channel ID for offline push configured on OPPO mobile phones that run
Android 8.0 or later
}
});
```

Response

Type： `Promise`

Recalling a message

This API is used to recall a one-to-one message or a group message. If the recall is successful, the value of `isRevoked` for the message is set to `true`.

注意：

- This API requires SDK v2.4.0 or later.
- The time limit for message recall is 2 minutes by default. You can log in to the IM console to change this limit.
- You can call the getMessageList API to pull recalled messages from the one-to-one or group message roaming list. Recalled messages are displayed based on `isRevoked` of the message object. For example, "The other party has recalled a message" can be displayed if a message is recalled during a one-to-one conversation, or "Tom has recalled a message" can be displayed if a message is recalled during a group conversation.
- You also use a RESTful API to recall one-to-one messages or recall group messages.

API

```
tim.revokeMessage(options)
```

Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Description
`message`	`Message`	Message instance

Example

```
// Actively recall a message.
let promise = tim.revokeMessage(message);
promise.then(function(imResponse) {
// The message is successfully recalled.
}).catch(function(imError){
// The message fails to be recalled.
console.warn('revokeMessage error:', imError);
});
```

```
tim.on(TIM.EVENT.MESSAGE_REVOKED, function(event) {
// The message recall notification is received. Before using this API, upgrade the SDK to v2.4.0 or later.
// event.name - TIM.EVENT.MESSAGE_REVOKED
// event.data - An array that stores the Message objects - [Message] - The `isRevoked` value of each Message object is `true`.
});
```

```
// Obtain a message list which contains recalled messages.
let promise = tim.getMessageList({conversationID: 'C2Ctest', count: 15});
promise.then(function(imResponse) {
const messageList = imResponse.data.messageList; // Message list
messageList.forEach(function(message) {
if (message.isRevoked) {
// Handle the recalled messages.
} else {
// Handle common messages.
}
});
});
```

Response

Type： `Promise`

Resending a message

This API is used to resend a message. When a message fails to be sent, call this API to resend the message.

API

```
tim.resendMessage(options)
```

Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Description
`message`	`Message`	Message instance

Example

```
// Resend a message.
let promise = tim.resendMessage(message); // Pass in the message instance that needs to be resen
t.
promise.then(function(imResponse) {
// The message is successfully resent.
console.log(imResponse.data.message);
}).catch(function(imError){
// The message fails to be resent.
console.warn('resendMessage error:', imError);
});
```

Response

This API returns a `Promise` object:

- The callback parameter of `then` is IMResponse. You can obtain the group list from `IMResponse.data.groupList` .
- The callback parameter of `catch` is IMError.

Receiving Messages

Receiving a message

For more information, see MESSAGE_RECEIVED.

This API is used to receive messages by means of listening for events.

Example

```
let onMessageReceived = function(event) {
// event.data - Array that stores the Message objects - [Message]
};
tim.on(TIM.EVENT.MESSAGE_RECEIVED, onMessageReceived);
```

Parsing a text message

- **Simple version**
  If your text message contains only text, the message is rendered as `'xxxxxxx'` on the UI.
- **If the message contains special formatted text such as [Toothy], the formatted text is rendered as the corresponding emoji 😬.**

```
const emojiMap = { // Map the formatted text to emojis.
'[Grin]': 'emoji_0.png',
'[Toothy]': 'emoji_1.png',
'[Rain]': 'emoji_2.png'
}
const emojiUrl = 'http://xxxxxxxx/emoji/' // URL of <img src="https://main.qcloudimg.com/raw/6be88c30a4552b5eb93d8eec243b6593.png" style="margin:0;">
function parseText (payload) {
let renderDom = []
// Text message
let temp = payload.text
let left = -1
let right = -1
while (temp !== '') {
left = temp.indexOf('[')
right = temp.indexOf(']')
switch (left) {
case 0:
if (right === -1) {
renderDom.push({
name: 'text',
text: temp
})
temp = ''
} else {
let _emoji = temp.slice(0, right + 1)
if (emojiMap[_emoji]) { // If you want to render text as emojis, you need to map the text to the URLs of the emojis.
renderDom.push({
name: 'img',
src: emojiUrl + emojiMap[_emoji]
})
temp = temp.substring(right + 1)
} else {
renderDom.push({
name: 'text',
text: '['
})
temp = temp.slice(1)
}
}
break
```

```
case -1:
renderDom.push({
name: 'text',
text: temp
})
temp = ''
break
default:
renderDom.push({
name: 'text',
text: temp.slice(0, left)
})
temp = temp.substring(left)
break
}
}
return renderDom
}
// The final structure of renderDom is [{name: 'text', text: 'XXX'}, {name: 'img', src: 'http://xxx'}.....].
// Render the array to obtain the desired UI result, for example, XXX<img src="https://main.qcloudimg.com/raw/6be88c30a4552b5eb93d8eec243b6593.png" style="margin:0;">XXX<img src="https://main.qcloudimg.com/raw/6be88c30a4552b5eb93d8eec243b6593.png" style="margin:0;">XXX[Toothy XXX].
```

Parsing system notifications

```
function parseGroupSystemNotice (payload) {
const groupName =
payload.groupProfile.groupName || payload.groupProfile.groupID
switch (payload.operationType) {
case 1:
return `${payload.operatorID} requests to join ${groupName}`
case 2:
return `You have successfully joined ${groupName}`
case 3:
return `Your request to join ${groupName} has been denied.`
case 4:
return `You have been removed from ${groupName} by ${payload.operatorID}`
case 5:
return `${groupName} has been disbanded by ${payload.operatorID}`
case 6:
return `${payload.operatorID} has created ${groupName}`
case 7:
return `${payload.operatorID} invites you to join ${groupName}`
case 8:
return `You have withdrawn from ${groupName}`
```

```
case 9:
return `${payload.operatorID} has made you an admin of ${groupName}`
case 10:
return `${payload.operatorID} has removed you as an admin of ${groupName}.`
case 255:
return 'Custom system group notification'
}
}
```

Parsing group notifications

```
function parseGroupTipContent (payload) {
switch (payload.operationType) {
case this.TIM.TYPES.GRP_TIP_MBR_JOIN:
return `${payload.userIDList.join(',')} joined the group`
case this.TIM.TYPES.GRP_TIP_MBR_QUIT:
return `${payload.userIDList.join(',')} left the group`
case this.TIM.TYPES.GRP_TIP_MBR_KICKED_OUT:
return `${payload.operatorID} removed ${payload.userIDList.join(',')} from the group`
case this.TIM.TYPES.GRP_TIP_MBR_SET_ADMIN:
return `The following member(s) are now admins: ${payload.userIDList.join(',')}`
case this.TIM.TYPES.GRP_TIP_MBR_CANCELED_ADMIN:
return `The following member(s) are no longer admins: ${payload.userIDList.join(',')}`
default:
return '[Group notification]'
}
}
```

Conversation APIs

Obtaining the message list of a conversation

For more information, see Conversation.

This API is used to pull by page the message list of a specified conversation. It is called when the message list is rendered for the first time after the user joins the conversation, or when the user pulls down the list to see more messages.

API

```
tim.getMessageList(options)
```

注意：
This API can be used to fetch history messages.

Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Attributes	Description
`conversationID`	`String`	`<optional>`	ID of the conversation, which is in the following format: C2C+userID (for a one-to-one conversation); GROUP+groupID (for a group conversation); or @TIM#SYSTEM (for a system notification).
`nextReqMessageID`	`String`	`<optional>`	Message ID, which is used to continue pulling messages by page. This parameter can be left unspecified the first time messages are pulled. Every time the API is called, this parameter is returned, and you need to specify it for the next pulling.
`count`	`Number`	`<optional>`	Number of messages to be pulled. Both the default value and maximum value are 15. This value indicates that a maximum of 15 messages can be pulled at a time.

Example

```
// Pull the message list for the first time when a conversation is opened.
let promise = tim.getMessageList({conversationID: 'C2Ctest', count: 15});
promise.then(function(imResponse) {
const messageList = imResponse.data.messageList; // Message list.
const nextReqMessageID = imResponse.data.nextReqMessageID; // This parameter must be passed in fo
r the next pulling by page.
const isCompleted = imResponse.data.isCompleted; // It indicates whether all messages have been p
ulled.
});
```

```
// Pull the message list for the first time when a conversation is opened.
// Pull down to see more messages.
let promise = tim.getMessageList({conversationID: 'C2Ctest', nextReqMessageID, count: 15});
promise.then(function(imResponse) {
```

```
const messageList = imResponse.data.messageList; // Message list.
const nextReqMessageID = imResponse.data.nextReqMessageID; // This parameter must be passed in fo
r the next pulling by page.
const isCompleted = imResponse.data.isCompleted; // It indicates whether all messages have been p
ulled.
});
```

Response

This API returns a `Promise` object:

- The callback parameter of `then` is IMResponse. You can obtain the group list from `IMResponse.data.groupList` .
- The callback parameter of `catch` is IMError.

Marking conversations as read

This API is used to set the unread messages of a conversation to the read state. Messages set to the read status are not counted as unread messages. This API is called when you open or switch a conversation. If this API is not called when you open or switch a conversation, the corresponding messages remain in the unread state.

API

```
tim.setMessageRead(options)
```

Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Description
`options`	`Object`	Message content container

`payload` has the following properties:

Name	Type	Description
`conversationID`	`String`	ID of the conversation, which is in the following format: C2C+userID (for a one-to-one conversation); GROUP+groupID (for a group conversation); or @TIM#SYSTEM (for a system notification).

Example

```
// Mark all unread messages of a conversation as read.
tim.setMessageRead({conversationID: 'C2Cexample'});
```

Obtaining the conversation list

This API is used to obtain the conversation list. It will pull the latest 100 conversations. You can call this API when you want to refresh the conversion list.

注意：

- The profile in the conversation list obtained by this API is incomplete. It contains only information such as profile photos and nicknames, which is sufficient to meet the requirements for rendering the conversation list. To query the detailed conversation profile, call getConversationProfile.
- The conversation retention time is consistent with the storage time of the last message, which is 7 days by default. That is, conversations will be stored for 7 days by default.

API

```
tim.getConversationList()
```

Example

```
// Pull the conversation list.
let promise = tim.getConversationList();
promise.then(function(imResponse) {
const conversationList = imResponse.data.conversationList; // This conversation list will overwri
te the original conversation list.
}).catch(function(imError){
console.warn('getConversationList error:', imError); // Information related to the failure to obt
ain the conversation list
});
```

Response

This API returns a `Promise` object:

- The callback parameter of `then` is IMResponse. You can obtain the group list from `IMResponse.data.groupList` .
- The callback parameter of `catch` is IMError.

Obtaining a conversation profile

This API is used to obtain the profile of a conversation. When you click a conversation in the conversation list, this API is called to obtain the detailed information of the conversation.

API

```
tim.getConversationProfile(conversationID)
```

Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Description
`conversationID`	`String`	ID of the conversation, which is in the following format: C2C+userID (for a one-to-one conversation); GROUP+groupID (for a group conversation); or @TIM#SYSTEM (for a system notification).

Example

```
let promise = tim.getConversationProfile(conversationID);
promise.then(function(imResponse) {
// The conversation profile is successfully obtained.
console.log(imResponse.data.conversation); // Conversation profile
}).catch(function(imError){
console.warn('getConversationProfile error:', imError); // Information related to the failure to obtain the conversation profile
});
```

Response

This API returns a `Promise` object:

- The callback parameter of `then` is IMResponse. You can obtain the group list from `IMResponse.data.groupList` .
- The callback parameter of `catch` is IMError.

Deleting a conversation

This API is used to delete a conversation based on the conversation ID. It deletes only the conversation but not the messages. For example, if the conversation with user A is deleted, the previous chat messages will still be available next time when you initiate a conversation with user A.

API

```
tim.deleteConversation(conversationID)
```

Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Description
`conversationID`	`String`	ID of the conversation, which is in the following format: C2C+userID (for a one-to-one conversation); GROUP+groupID (for a group conversation); or @TIM#SYSTEM (for a system notification).

Example

```
let promise = tim.deleteConversation('C2CExample');
promise.then(function(imResponse) {
// The conversation is deleted successfully.
const { conversationID } = imResponse.data;// ID of the deleted conversation
}).catch(function(imError){
console.warn('deleteConversation error:', imError); // Information related to the failure to dele
te the conversation
});
```

Response

This API returns a `Promise` object:

- The callback parameter of `then` is IMResponse. You can obtain the group list from `IMResponse.data.groupList` .
- The callback parameter of `catch` is IMError.

Conversation
Conversation (Android)

Last updated : 2021-10-15 15:19:03

Displaying the Conversation List

After logging in to the app, you can display a list of recent conversions like WeChat. The entire display process includes the following steps: **pulling the conversation list**, **processing the change notifications**, and **updating the UI content (including the unread count)**. This document describes these steps in detail.

Pulling the conversation list

After logging in to the app, you can call getConversationList() to pull the local conversation list and display the list on the UI. The conversation list is a list of V2TIMConversation objects, and every object represents a conversation.

The number of local conversations may be large, for example, more than 500. It may take a long time to load all conversations at a time, which results in slow display of the list on the UI. To improve user experience, the `getConversationList()` API is provided to support the feature of pulling by page.

1. When the `getConversationList()` API is called for the first time, you can set the `nextSeq` parameter to 0, indicating that the conversation list is pulled from the beginning, and set `count` to 50, indicating that 50 conversation objects are pulled at a time.
2. The IM SDK pulls the conversation list in the new-to-old order. After the conversation list is pulled successfully for the first time, V2TIMConversationResult, which is the callback result of `getConversationList()`, will contain the `nextSeq` field for the next pulling by page, and the `isFinish` field that indicates whether the conversation pulling is completed.
   - If the returned value of `isFinished` is `true`, all conversations have been pulled.
   - If the returned value of `isFinished` is `false`, more conversations can be pulled. This does not mean that the next page of the conversation list will be pulled immediately. In common communications software, pulling by page is often triggered by the swipe operation of the user. Each time when you swipe on the conversation list, pulling by page is triggered once.
3. When you continue to swipe on the conversation list, if more content of the conversation list is yet to be pulled, the IM SDK can continue to call the getConversationList API and pass in the `nextSeq`

and `count` parameters again. The values of the two parameters come from the V2TIMConversationResult object returned in the previous pulling.

4. The IM SDK repeats step 3 until the returned value of isFinished is `true` .

Displaying the conversation information

After obtaining the V2TIMConversation object, the IM SDK can display the conversation information on the UI. `V2TIMConversation` contains the following key fields, which are often used to construct the conversation list.

Field	Description
getShowName ()	The conversion name: • For the one-to-one chat, the API preferentially returns the friend's remark. If the remark is unavailable or if the peer is not a friend, the API returns the nickname of the peer. If the nickname is also unavailable, the API returns the UserID of the peer. • For a group chat, the group name is displayed.
getFaceUrl ()	The profile photo for the conversation: • For a one-to-one chat, the other party's profile photo is displayed. • For a group chat, the group's profile photo is displayed.
getRecvOpt ()	The message receiving option, which is generally used for a group conversation. It indicates whether the **Mute notifications** mode is enabled for the group.
getUnreadCount ()	The unread count, which indicates the number of unread messages
getLastMessage ()	The last message, which displays the message digest of the conversation

Updating the conversation list

After the IM SDK is successfully logged in, or the user goes online, or the connection is re-established after being interrupted, the IM SDK will automatically update the conversation list. The update process is as follows:

- When any conversation is updated, for example, when a new message is received, the SDK will notify you by using the onConversationChanged event in `V2TIMConversationListener` .

- When any conversation is added, the SDK will notify you by using the onNewConversation event in `V2TIMConversationListener` .

Note :
To ensure that the order of the conversation list complies with the sequencing principle specified in the last message, the data source must be re-sequenced based on getTimestamp in getLastMessage.

Sample code

The sample code shows how to pull, display, and update the conversation list.

```
// 1. Set the conversation listener.
V2TIMManager.getConversationManager().setConversationListener(this);
// 2. Pull 50 local conversation to display them on the UI. Set the value of `nextSeq` that is pa
ssed in for the first pulling to 0.
V2TIMManager.getConversationManager().getConversationList(0, 50,
new V2TIMValueCallback<V2TIMConversationResult>() {
@Override
public void onError(int code, String desc) {
// The attempt to pull the conversation list fails.
}
@Override
public void onSuccess(V2TIMConversationResult v2TIMConversationResult) {
// Pulling is successful, and the UI conversation list is updated.
updateConversation(v2TIMConversationResult.getConversationList(), false);
if (!v2TIMConversationResult.isFinished()) {
V2TIMManager.getConversationManager().getConversationList(
v2TIMConversationResult.getNextSeq(), 50,
new V2TIMValueCallback<V2TIMConversationResult>() {
@Override
public void onError(int code, String desc) {}
@Override
public void onSuccess(V2TIMConversationResult v2TIMConversationResult) {
// Pulling is successful, and the UI conversation list is updated.
updateConversation(v2TIMConversationResult.getConversationList(), false);
}
});
}
}
// 3.1 Receive the callback for adding a conversation.
@Override
public void onNewConversation(List<V2TIMConversation> conversationList) {
updateConversation(conversationList, true);
```

```
}
// 3.2 Receive the callback for updating the conversation.
@Override
public void onConversationChanged(List<V2TIMConversation> conversationList) {
updateConversation(conversationList, true);
}
private void updateConversation(List<V2TIMConversation> convList, boolean needSort) {
for (int i = 0; i < convList.size(); i++) {
V2TIMConversation conv = convList.get(i);
boolean isExit = false;
for (int j = 0; j < uiConvList.size(); j++) {
V2TIMConversation uiConv = uiConvList.get(j);
// If the conversation exists in the UI conversation list, replace this conversation.
if (uiConv.getConversationID().equals(conv.getConversationID())) {
uiConvList.set(j, conv);
isExit = true;
break;
}
}
// If the conversation does not exist in the UI conversation list, add this conversation.
if (!isExit) {
uiConvList.add(conv);
}
}
// 4. Based on the timestamp in the lastMessage of the conversation, sort the UI conversation lis
t, and refresh the UI.
if (needSort) {
Collections.sort(uiConvList, new Comparator<V2TIMConversation>() {
@Override
public int compare(V2TIMConversation o1, V2TIMConversation o2) {
if (o1.getLastMessage().getTimestamp() > o2.getLastMessage().getTimestamp()) {
return -1;
} else {
return 1;
}
}
});
}
...
mAdapter.setDataResource(uiConvList);
mAdapter.notifyDataSetChanged();
}
```

Obtaining the Total Unread Message Count of All Conversations (Only Available in Lite Edition v5.3.425 and Above)

You can call the getTotalUnreadMessageCount API to get the total unread count of all conversations. You don't need to go through the conversation list and add up the unread count of each conversation to get the total unread count. When the total unread count changes, the SDK will proactively call back onTotalUnreadMessageCountChanged to your app to notify you of the latest unread count.

Pinning a Conversation on Top (Only Available in Lite Edition v5.3.425 and Above)

You can pin a one-to-one or group conversation on the top of the conversation list so you can quickly find it. The new SDK version has added the pinConversation API to pin/unpin conversations to/from top. It also supports roaming and multi-client synchronization.
The V2TIMConversation conversation object has added the isPinned API, which is used to determine whether a conversion is pinned on top. When the pinned-on-top status of a conversation changes, the SDK will call back onConversationChanged to your app.

Deleting a Conversation

You can call the deleteConversation API to delete a conversation. Conversation deletion cannot be synchronized across multiple clients. When a conversation is deleted, the message history of this conversation will be deleted from the local storage and the server by default, and cannot be recovered.

Drafts

When sending a message, you may need to switch to another chat window before message editing is completed. In this case, you can call the setConversationDraft API to save the unfinished message. Later, you can return to the original chat window and call getDraftText to continue editing the message.

Note :

- Only text content can be stored in Drafts.
- Drafts are stored only locally, instead of on the server. Therefore, drafts cannot be synchronized across multiple devices. If the program is uninstalled, drafts cannot be reloaded.

FAQs

1. What is the upper limit for the number of conversations that can be stored in a conversation list?

A locally stored conversation list can have unlimited number of conversations. A conversation list stored in the cloud can have up to 100 conversations.
If the information of a conversation has not been updated for a long time, this conversation can be stored in the cloud for at most 7 days. To adjust the period for storing the conversation, contact us.

2. Why are the conversation lists pulled on different mobile phones by using the same account inconsistent?

Locally stored conversations may not always be consistent with those stored in the cloud. If you do not call the deleteConversation API to delete the local conversations, these conversations will always exist. However, at most 100 conversations can be stored in the cloud. In addition, if the information of a conversation has not been updated for a long time, this conversation can be stored in the cloud for at most 7 days. Therefore, local conversations displayed on different mobile phones may be inconsistent with each other.

3. Why are repeated conversations pulled?

Conversations that are pulled by the `getConversationList` API may have already been added to the data source of the UI conversation list through the `onNewConversation` callback API. Therefore, to avoid adding the same conversation repeatedly, you need to find and replace the same conversations based on getConversationID.

4. Does the IM SDK support the feature of pinning a conversation to the top?

IM SDK supports pinning a conversation to the top and sync to the cloud starting from v5.3.425.

Conversation (iOS)

Last updated : 2021-10-15 15:21:25

Displaying the Conversation List

After logging in to the app, you can display a list of recent conversions like WeChat. The entire display process includes the following steps: **pulling the conversation list**, **processing the change notifications**, and **updating the UI content (including the unread count)**. This document describes these steps in detail.

Pulling the conversation list

After logging in to the app, you can call getConversationList() to pull the local conversation list and display the list on the UI. The conversation list is a list of V2TIMConversation objects, and every object represents a conversation.

The number of local conversations may be large, for example, more than 500. It may take a long time to load all conversations at a time, which results in slow display of the list on the UI. To improve user experience, the `getConversationList()` API is provided to support the feature of pulling by page.

1. When the `getConversationList()` API is called for the first time, you can set the `nextSeq` parameter to 0, indicating that the conversation list is pulled from the beginning, and set `count` to 50, indicating that 50 conversation objects are pulled at a time.
2. The IM SDK pulls the conversation list in the new-to-old order. After the conversation list is pulled successfully for the first time, V2TIMConversationResult, which is the callback result of `getConversationList()`, will contain the `nextSeq` field for the next pulling by page and the `isFinish` field that indicates whether the conversation pulling is completed.
   - If the returned value of `isFinished` is `true`, all conversations have been pulled.
   - If the returned value of `isFinished` is `false`, more conversations can be pulled. This does not mean that the next page of the conversation list will be pulled immediately. In common communications software, pulling by page is often triggered by the swipe operation of the user. Each time when you swipe on the conversation list, pulling by page is triggered once.
3. When you continue to swipe on the conversation list, if more content of the conversation list is yet to be pulled, the IM SDK can continue to call the getConversationList() API and pass in the `nextSeq` and `count` parameters again. The values of the two parameters come from the V2TIMConversationResult object returned in the previous pulling.
4. The IM SDK repeats step 3 until the returned value of `isFinished` is `true`.

Displaying the conversation information

After obtaining the V2TIMConversation object, the IM SDK can display the conversation information on the UI. `V2TIMConversation` contains the following key fields, which are often used to construct the conversation list.

Field	Description
showName	The conversion name: • For the one-to-one chat, the API preferentially returns the friend's remark. If the remark is unavailable or if the peer is not a friend, the API returns the nickname of the peer. If the nickname is also unavailable, the API returns the UserID of the peer. • For a group chat, the group name is displayed.
faceUrl	The profile photo for the conversation: • For a one-to-one chat, the other party's profile photo is displayed. • For a group chat, the group's profile photo is displayed.
unreadCount	The unread count, which indicates the number of unread messages.
recvOpt	The message receiving option, which is generally used for a group conversation. It indicates whether the **Mute Notifications** mode is enabled for the group.
lastMessage	The last message, which displays the message digest of the conversation.
groupAtInfolist	The @ information of the conversation, which shows "someone @ me", "@ all members", etc.
isPinned	Whether the conversation is pinned on top. This is only available in the lite edition.

Updating the conversation list

After the IM SDK is successfully logged in, or the user goes online, or the connection is re-established after being interrupted, the IM SDK will automatically update the conversation list. The update process is as follows:

- When any conversation is updated, for example, when a new message is received, the SDK will notify you by using the onConversationChanged event in `V2TIMConversationListener`.
- When any conversation is added, the SDK will notify you by using the onNewConversation event in `V2TIMConversationListener`.

Note :
To ensure that the order of the conversation list complies with the sequencing principle specified in the last message, the data source must be re-sequenced based on timestamp in lastMessage.

Sample code

The sample code shows how to pull, display, and update the conversation list.

```
// 1. Set the conversation listener.
[[V2TIMManager sharedInstance] setConversationListener:self];
// 2. Log in to the IM SDK.
[[V2TIMManager sharedInstance] login:@"yahaha" userSig:@"Pass in the actual UserSig" succ:^{
// 3. Pull 50 local conversation to display them on the UI. Set the value of `nextSeq` that is pa
ssed in for the first pulling to 0.
__weak __typeof(self) weakSelf = self;
[[V2TIMManager sharedInstance] getConversationList:0 count:50
succ:^(NSArray<V2TIMConversation *> *list, uint64_t nextSeq, BOOL isFinished) {
__strong __typeof(weakSelf) strongSelf = weakSelf;
// Pulling is successful, and the UI conversation list is updated.
[strongSelf updateConversation:list];
// 4. If more conversations need to be pulled, the value of `nextSeq` returned in the previous pu
lling will be passed in.
if(!isFinished) {
[[V2TIMManager sharedInstance] getConversationList:nextSeq count:50
succ:^(NSArray<V2TIMConversation *> *list, uint64_t nextSeq, BOOL isFinished) {
// Pulling is successful, and the UI conversation list is updated.
[strongSelf updateConversation:list];
} fail:^(int code, NSString *msg) {
// The attempt to pull the conversation list fails.
}];
}
} fail:^(int code, NSString *msg) {
// The attempt to pull the conversation list fails.
}];
} fail:^(int code, NSString *msg) {
// Logn fails.
}];
// Receive the callback for adding a conversation.
- (void)onNewConversation:(NSArray<V2TIMConversation*> *) conversationList {
[self updateConversation:conversationList];
}
// Receive the callback for updating the conversation.
- (void)onConversationChanged:(NSArray<V2TIMConversation*> *) conversationList {
```

```
[self updateConversation:conversationList];
}
// Update the UI conversation list.
- (void)updateConversation:(NSArray *)convList
{
// If the updated conversation exists in the UI conversation list, replace this conversation. Oth
erwise, add this conversation.
for (int i = 0 ; i < convList.count ; ++ i) {
V2TIMConversation *conv = convList[i];
BOOL isExit = NO;
for (int j = 0; j < self.uiConvList.count; ++ j) {
V2TIMConversation *uiConv = self.localConvList[j];
// If the updated conversation exists in the UI conversation list, replace this conversation.
if ([uiConv.conversationID isEqualToString:conv.conversationID]) {
[self.uiConvList replaceObjectAtIndex:j withObject:conv];
isExit = YES;
break;
}
}
// If the updated conversation does not exist in the UI conversation list, add this conversation.
if (!isExit) {
[self.uiConvList addObject:conv];
}
}
// Sort the UI conversation list based on the timestamp in the lastMessage of the conversation.
[self.uiConvList sortUsingComparator:^NSComparisonResult(V2TIMConversation *obj1, V2TIMConversati
on *obj2) {
return [obj2.lastMessage.timestamp compare:obj1.lastMessage.timestamp];
}];
}
```

Obtaining the Total Unread Message Count of All Conversations (Only Available in Lite Edition v5.3.425 and Above)

You can call the getTotalUnreadMessageCount API to get the total unread count of all conversations. You don't need to go through the conversation list and add up the unread count of each conversation to get the total unread count. When the total unread count changes, the SDK will proactively call back onTotalUnreadMessageCountChanged to your app to notify you of the latest unread count.

Pinning a Conversation on Top (Only Available in Lite Edition v5.3.425 and Above)

You can pin a one-to-one or group conversation on the top of the conversation list so you can quickly find it. The new SDK version has added the pinConversation API to pin/unpin conversations to/from top. It also supports roaming and multi-client synchronization.
The V2TIMConversation conversation object has added the isPinned API, which is used to determine whether a conversation is pinned on top. When the pinned-on-top status of a conversation changes, the SDK will call back onConversationChanged to your app.

Deleting a Conversation

You can call the deleteConversation API to delete a conversation. Conversation deletion cannot be synchronized across multiple clients. When a conversation is deleted, the message history of this conversation will be deleted from the local storage and the server by default, and cannot be recovered.

Drafts

When sending a message, you may need to switch to another chat window before message editing is completed. In this case, you can call the setConversationDraft API to save the unfinished message. Later, you can return to the original chat window and call draftText to continue editing the message.

Note :

- Only text content can be stored in Drafts.
- Drafts are stored only locally, instead of on the server. Therefore, drafts cannot be synchronized across multiple devices. If the program is uninstalled, drafts cannot be reloaded.

FAQs

1. What is the upper limit for the number of conversations that can be stored in a conversation list?

A locally stored conversation list can have unlimited number of conversations. A conversation list stored in the cloud can have up to 100 conversations.
If the information of a conversation has not been updated for a long time, this conversation can be stored in the cloud for at most 7 days. To adjust the period for storing the conversation, contact us.

2. Why are the conversation lists pulled on different mobile phones by using the same account inconsistent?

Locally stored conversations may not always be consistent with those stored in the cloud. If you do not call the deleteConversation API to delete the local conversations, these conversations will always exist. However, at most 100 conversations can be stored in the cloud. In addition, if the information of a conversation has not been updated for a long time, this conversation can be stored in the cloud for at most 7 days. Therefore, local conversations displayed on different mobile phones may be inconsistent with each other.

3. Why are repeated conversations pulled?

Conversations that are pulled by the `getConversationList` API may have already been added to the data source of the UI conversation list through the `onNewConversation` callback API. Therefore, to avoid adding the same conversation repeatedly, you need to find and replace the same conversations based on getConversationID.

4. Does the IM SDK support the feature of pinning a conversation to the top?

IM SDK supports pinning a conversation to the top and sync to the cloud starting from v5.3.425.

Unread Count (Web & Mini Program)

Last updated : 2021-04-28 17:55:51

Unread messages are messages that have not been reported as read by users. This does not indicate whether the recipient has actually read the messages. To display the correct unread count, developers need to explicitly call read reports to notify the IM SDK whether the messages in a conversation are read. For example, you can mark all messages as read when the user enters the chat UI.

Obtaining the Current Unread Count

Every time you use getConversationList(), you will obtain a [Conversation, Conversation, ...] array. Each `Conversation` has an unread count for the current conversation, which is represented by `unreadCount`.
The unread count of all conversations is the sum of the `unreadCount` values of each conversation.

Read Reports

When the user reads a message in a conversation, a read report is sent, and the IM SDK sets all messages before the last read message as read. We recommend that you send read reports when the user clicks to switch between conversations.

Note :

Read reports will change the unread count of a conversation. If you set `C2C` messages to read in SDK v2.7.0 or later versions, read receipts will be pushed to the message sender. For more information, see TIM.EVENT.MESSAGE_READ_BY_PEER.

API

```
tim.setMessageRead(options);
```

Parameters

The `options` parameter is of the `Object` type. The attribute it contains is described in the following table:

Name	Type	Description

`conversationID`	`String`	Conversation ID

Example

```
//Send read reports for all unread messages in a conversation
let promise = tim.setMessageRead({conversationID: 'C2Cexample'});
promise.then(function(imResponse) {
//Read reports sent successfully
}).catch(function(imError) {
//Failed to send read reports
console.warn('setMessageRead error:', imError);
});
```

Group
Group Management (Android)

Last updated : 2021-10-15 16:01:02

Group Types

Instant Messaging (IM) supports the following group types:

- A **work group for friends (Work)** is like an ordinary WeChat group. After a work group is created, a user can only join the group by being invited by a friend who is a member of the group. The invitation does not need to be accepted by the invitee or approved by the group owner.
- A **public group (Public)** is like a QQ group. After a public group is created, the group owner can specify group admins. When a user searches the group ID and initiates a request to join the group, the request must be approved by the group owner or admin before the user can join the group.
- A **temporary meeting group (Meeting)** allows users to join and exit freely and supports viewing historical messages from before the user joined the group. Meeting groups can integrate Tencent Real-Time Communication (TRTC), such as in audio and video conference and online education scenarios.
- A **live streaming group (AVChatRoom)** allows users to join and exit freely, supports an unlimited number of members, and does not store message history. Live streaming groups can be used with Live Video Broadcasting (LVB) to support the on-screen comments.

The following table describes the features and limitations of each group type:

Feature Item	Work	Public	Meeting	AVChatRoom
Available member roles	Group owner and ordinary member	Group owner, group admin, and ordinary member	Group owner, group admin, and ordinary member	Group owner and ordinary member
Requesting to join a group	Unsupported	Supported with group owner or group admin approval required	Supported with no approval required	Supported with no approval required

Joining the group via invitation by a member	Supported	Unsupported	Unsupported	Unsupported
Group owner can quit the group	Supported	Unsupported	Unsupported	Unsupported
Who can modify the group profile	Any group member	Group owner and group admin	Group owner and group admin	Group owner
Who can kick group members out of the group	Group owner	Group owner and group admin, but group admins can only kick ordinary group members out of the group		Group members cannot be removed. The same effect can be achieved by muting members.
Who can mute members	Muting members is not supported	Group owner and group admin, but group admin can only mute ordinary group members		Group owner
Unread message count	Supported	Supported	Unsupported	Unsupported
Viewing historical messages from before a user joined	Unsupported	Unsupported	Supported	Unsupported
Storage of historical messages on the cloud	• Trial Edition: 7 days • Pro Edition: 7 days by default, up to 360 days via value-added service • Flagship Edition: 30 days by default, up to 360 days via value-added service			Unsupported
Number of groups	• Trial Edition: up to 100 existing groups, and disbanded groups do not count against the quota • Pro Edition or Flagship Edition: unlimited			• Trial Edition: up to 10 existing groups, and disbanded groups do not count against the quota • Pro Edition: up to 50 existing groups, and disbanded groups do not count against the quota You can upgrade to an

		unlimited number of live streaming groups by purchasing value-added service • Flagship Edition: unlimited
Number of group members	• Trial Edition: 20 per group • Pro Edition: 200 per group by default, which can be increased to 2,000 per group via value-added service • Flagship Edition: 2,000 per group by default, which can be increased to 6,000 per group via value-added service	Unlimited number of group members

Note：
In the Pro Edition or Flagship Edition SDKAppID, the maximum net increase in group quantity per day is 10,000 for all group types by default. The maximum number of free groups is 100,000 per month, and you will need to pay for groups exceeding the free quota.

Group Management

Creating a group

Simple API

You can quickly create a group by calling the createGroup API and specifying `groupType`, `groupID`, and `groupName`.

Advanced API

If you want to initialize group information (for example, group introduction, group profile photo, and initial group members) when creating a group, call the createGroup API in the `V2TIMGroupManager` management class. The `V2TIMGroupManager` management class can be obtained via `V2TIMManager.getGroupManager`.

```
// Sample code: create a work group using the advanced createGroup API
V2TIMGroupInfo v2TIMGroupInfo = new V2TIMGroupInfo();
v2TIMGroupInfo.setGroupName("testWork");
v2TIMGroupInfo.setGroupType("Work");
v2TIMGroupInfo.setIntroduction("this is a test Work group");
```

```
List<V2TIMCreateGroupMemberInfo> memberInfoList = new ArrayList<>();
V2TIMCreateGroupMemberInfo memberA = new V2TIMCreateGroupMemberInfo();
memberA.setUserID("vinson");
V2TIMCreateGroupMemberInfo memberB = new V2TIMCreateGroupMemberInfo();
memberB.setUserID("park");
memberInfoList.add(memberA);
memberInfoList.add(memberB);
V2TIMManager.getGroupManager().createGroup(
v2TIMGroupInfo, memberInfoList, new V2TIMValueCallback<String>() {
@Override
public void onError(int code, String desc) {
// Failed to create
}
@Override
public void onSuccess(String groupID) {
// Created successfully
}
});
```

- The value of `groupType` is a string. Valid values: "Work", "Public", "Meeting", and "AVChatRoom". For more information on the differences among group types, see Group Types.
- `groupID` specifies the group ID, which uniquely identifies a group. Do not create groups with the same `groupID` in a single SDKAppID. If you set `groupID` to null, an ID is assigned for your group by default.
- `groupName` specifies the group description, which has a maximum length of 30 bytes.

Joining a group

The group joining processes for different group types are described below:

Type	Work Group (Work)	Social Networking Group (Public)	Temporary Meeting Group (Meeting)	Live Streaming Group (AVChatRoom)
How to join the group	Must be invited by a group member	User joins the group after request is approved by group owner or admin	User can join freely	User can join freely

Scenario 1: users can join and quit the group freely

Temporary meeting groups (Meeting) and live streaming groups (AVChatRoom) can be used for interactive scenarios where users join and exit the group frequently, such as online conferences and fashion show live streams. These groups have the simplest join procedure.

After a user successfully joins a group by calling joinGroup, all group members (including the joining user) receive the onMemberEnter callback.

Scenario 2: users must be invited to the group

Similar to WeChat and WeChat Work groups, work groups (Work) are suitable for communication in a work environment. The interaction pattern is designed to disable proactive group joining and only allows users to be invited to the group by group members.

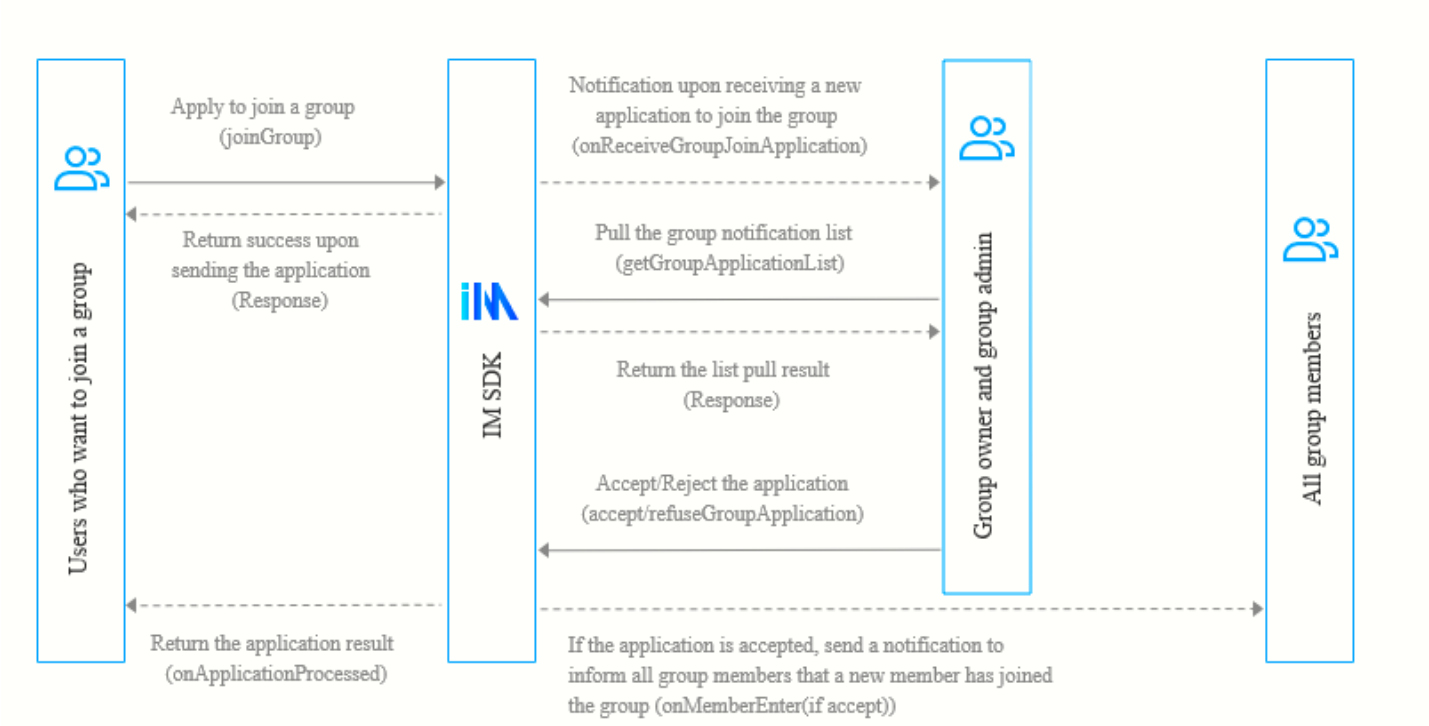
A group member calls inviteUserToGroup to invite a user to the group, and then all group members (including the inviter) receive the onMemberInvited callback.

Scenario 3: users join the group after their requests are approved

Social networking groups (Public) are similar to the interest groups and tribes in QQ. Any user can request to join the group, but will not become a member of the group until the request is approved by the group owner or admin. While approval is required by default, the group owner or admin can call the setGroupInfo API to set the group joining option (`V2TIMGroupAddOpt`) to “forbid anyone to join” or “disable the approval process”.

- V2TIM_GROUP_ADD_FORBID: forbid anyone to join the group.
- V2TIM_GROUP_ADD_AUTH: (default) group owner or admin approval is required to join the group.
- V2TIM_GROUP_ADD_ANY: disable the approval process to allow any user to join the group.

The following diagram illustrates the process of group joining that requires approval:



1. **The user sends a request to join the group**
   The user calls joinGroup to request to join the group.
2. **The group owner or admin processes the group joining application**
   After the onReceiveJoinApplication callback is received, the group owner or admin calls getGroupApplicationList to get the list of group joining requests, and approves or rejects a request with acceptGroupApplication or refuseGroupApplication.
3. **The user receives the result**
   The user receives the onApplicationProcessed callback in V2TIMGroupListener. If `isAgreeJoin` is `true`, the request is approved. Otherwise, the request is rejected. If the request is approved, all members (including the requestor) receive the onMemberEnter callback.

Quitting a group

Call quitGroup to quit a group. The user who quits the group then receives the onQuitFromGroup callback and other group members receive the onMemberLeave callback.

> Note :
> The group owners of social networking groups (Public), temporary meeting groups (Meeting), and live streaming groups (AVChatRoom) are not allowed to quit the group. Instead, the group owner can disband the group.

Disbanding a group

Call dismissGroup to disband a group. Then all group members receive the onGroupDismissed callback.

> Note :
>
> - For social networking groups (Public), temporary meeting groups (Meeting), and live streaming groups (AVChatRoom), the group owner can disband the group at any time.
> - For work groups (Work), the group owner does not have the permission to disband the group. To disband the group, you must have your business server call the RESTful API for disbanding groups.

Getting the list of joined groups

Call getJoinedGroupList to get a list of work groups (Work), social networking groups (Public), and temporary meeting groups (Meeting) the current user has joined. Live streaming groups

(AVChatRoom) will not be included in this list.

Group Profiles and Group Settings

Getting group profiles

Call getGroupsInfo to get the group profile of one or more groups at a time. To get the group profiles of multiple groups with a single call, pass in multiple `groupIDs` at one time.

Modifying group profiles

Call setGroupInfo to modify the group profile. When the modification is complete, all group members receive the onGroupInfoChanged callback.

Note :

- For work groups (Work), all group members can modify the basic group profile.
- For social networking groups (Public) and temporary meeting groups (Meeting), only the group owner and admin can modify the basic group profile.
- For live streaming groups (AVChatRoom), only the group owner can modify the group profile.

```
// Sample code: modify the group profile
V2TIMGroupInfo v2TIMGroupInfo = new V2TIMGroupInfo();
v2TIMGroupInfo.setGroupID("the ID of the group for which you want to modify the group profile");
v2TIMGroupInfo.setFaceUrl("http://xxxx");
V2TIMManager.getGroupManager().setGroupInfo(v2TIMGroupInfo, new V2TIMCallback() {
@Override
public void onError(int code, String desc) {
// Failed
}
@Override
public void onSuccess() {
// Successful
}
});
```

Setting the group message receiving option

Any group member can call the setGroupReceiveMessageOpt API to modify the group message receiving option. Available values are as follows:

- V2TIMGroupInfo.V2TIM_GROUP_RECEIVE_MESSAGE: messages will be received when the user is online and push notifications will be received when the user is offline.
- V2TIMGroupInfo.V2TIM_GROUP_NOT_RECEIVE_MESSAGE: no group messages will be received.
- V2TIMGroupInfo.V2TIM_GROUP_RECEIVE_NOT_NOTIFY_MESSAGE: messages will be received when the user is online and no push notification will be received when the user is offline.

The group message receiving option allows you to mute group messages:

- **No group message will be received**
  With the group message receiving option set to `V2TIMGroupInfo.V2TIM_GROUP_NOT_RECEIVE_MESSAGE`, no group message will be received, and the conversation list will not be updated.
- **Group messages will be received but the user will not be notified. A badge without the unread count will be displayed on the conversation list interface**

  > Note :
  > The unread count feature needs to be enabled for this to work. Therefore it only applies to work groups (Work) and social networking groups (Public).

  With the group message receiving option set to `V2TIMGroupInfo.V2TIM_GROUP_RECEIVE_NOT_NOTIFY_MESSAGE`, when new group messages are received and the conversation list needs to update, get the unread count through getUnreadCount. Use getRecvOpt to verify that the group message receiving option is `V2TIMGroupInfo.V2TIM_GROUP_RECEIVE_NOT_NOTIFY_MESSAGE` and then display a badge without the unread count.

Group Attributes (Custom Group Fields)

Based on API 2.0, we designed new custom group fields called "group attributes". Their features are as follows:

1. Clients can directly add, delete, modify, and query group attributes without the need for console configuration.
2. A maximum of 16 group attributes is supported. The maximum size supported for each group attribute is 4 KB, and the maximum size supported for all group attributes is 16 KB.
3. Currently, only AVChatRooms are supported.
4. The SDK can call initGroupAttributes, setGroupAttributes, and deleteGroupAttributes simultaneously up to 10 times every 5 seconds. If this limit is exceeded, the 8511 error code is

returned via callback. The backend can call these APIs simultaneously up to 5 times per second. If this limit is exceeded, the 10049 error code is returned.

5. The SDK can call getGroupAttributes up to 20 times every 5 seconds.

Based on group attributes, we can manage the mic for voice chat rooms. When a member turns the mic on, you can set a group attribute to manage the information of the mic-on member. When the member turns the mic off, you can delete the corresponding group attribute. Other members can obtain the group attribute list to show the mic position list.

Initializing group attributes

Call initGroupAttributes to initialize group attributes. The original group attributes, if any, will be cleared.

Setting group attributes

Call setGroupAttributes to set group attributes. If a set group attribute does not exist, it will be automatically added.

Deleting group attributes

Call deleteGroupAttributes to delete the specified group attribute. If the `keys` field is set to `null`, all group attributes will be cleared.

Obtaining group attributes

Call getGroupAttributes to obtain the specified group attribute. If the `keys` field is set to `null`, all group attributes will be obtained.

Updating group attributes

If any group attribute is updated, all group attributes will be updated via the onGroupAttributeChanged callback.

Managing group members

Getting group member list

Call getGroupMemberList to get the list of members of a specified group. The list contains profile information about individual members, such as user ID (`userID`), group name card (`nameCard`), profile photo (`faceUrl`), nickname (`nickName`), and time of joining the group (`joinTime`).
As a group may have a large number of members (even over 5,000), so this API supports two advanced properties: `filter` and `nextSeq` .

Filters (filter)

When calling getGroupMemberList, you can specify `filter` to pull the information list of certain group roles.

Filter	Description
V2TIMGroupMemberFullInfo.V2TIM_GROUP_MEMBER_FILTER_ALL	Pull the information list of all group members
V2TIMGroupMemberFullInfo.V2TIM_GROUP_MEMBER_FILTER_OWNER	Pull the information list of the group owner
V2TIMGroupMemberFullInfo.V2TIM_GROUP_MEMBER_FILTER_ADMIN	Pull the information list of the group admin
V2TIMGroupMemberFullInfo.V2TIM_GROUP_MEMBER_FILTER_COMMON	Pull the information list of ordinary group members

```
// Sample code: pull the profile of the group owner using the filter parameter
int role = V2TIMGroupMemberFullInfo.V2TIM_GROUP_MEMBER_FILTER_OWNER;
V2TIMManager.getGroupManager().getGroupMemberList("testGroup", role, 0,
new V2TIMValueCallback<V2TIMGroupMemberInfoResult>() {
@Override
public void onError(int code, String desc) {
// Failed to pull
}
@Override
public void onSuccess(V2TIMGroupMemberInfoResult v2TIMGroupMemberInfoResult) {
// Pulled successfully
}
});
```

Pulling paginated results with nextSeq

In many cases, it makes more sense for the user interface to display the first page of the group member list instead of the complete list. More group members can be pulled when the user clicks "Next Page" or pull the list to refresh. For this scenario, you can apply the method of pulling paginated results.

The getGroupMemberList API returns a maximum of 50 members at a time. You can use the pagination flag `nextSeq` to pull a paginated group member list. In the first attempt to pull the group member list, enter 0 for `nextSeq` . When the first pull succeeds, `getGroupMemberList` 's callback result V2TIMGroupMemberInfoResult contains the getNextSeq() API.

- If getNextSeq() returns 0, the complete group member list has been pulled.
- If getNextSeq() returns a value greater than 0, there remains group member information to be pulled. You can then decide whether to make another call to pull group member information based on the user's action on the UI. In the second pull, you need to pass the getNextSeq() in the `V2TIMGroupMemberInfoResult` returned from the previous pull as parameter to the getGroupMemberList API.

```
// Sample code: pull paginated group member list using nextSeq
{
...
long nextSeq = 0;
getGroupMemberList(nextSeq);
...
}
public void getGroupMemberList(long nextSeq) {
int filterRole = V2TIMGroupMemberFullInfo.V2TIM_GROUP_MEMBER_FILTER_ALL;
V2TIMManager.getGroupManager().getGroupMemberList("testGroup", filterRole, nextSeq,
new V2TIMValueCallback<V2TIMGroupMemberInfoResult>() {
@Override
public void onError(int code, String desc) {
// Failed to pull
}
@Override
public void onSuccess(V2TIMGroupMemberInfoResult groupMemberInfoResult) {
if (groupMemberInfoResult.getNextSeq() != 0) {
// Make another pull
getGroupMemberList(groupMemberInfoResult.getNextSeq());
...
} else {
// Pull ends
}
}
});
}
```

Getting the profiles of group members

To obtain the profile of a group member, call the getGroupMembersInfo API. You can pass in multiple `userID` values at one time to obtain profiles of groups, which improves network transmission efficiency.

Modifying group member profiles

The group owner or admin can call the setGroupMemberInfo API to modify group-related information of members, including group name card (`nameCard`), group member role (`role`), and muting duration (`muteUntil`).

Muting

The group owner or admin can mute a group member and set a muting duration (in seconds) via muteGroupMember. Muting information is stored in the muteUtil field of the group member. After the group member is muted, all group members (including the muted member) receive the onGroupMemberInfoChanged callback.
The group owner or admin can mute the entire group via the setGroupInfo API by setting allMuted to `true` . There is no time limit for muting the group. The group can be unmuted through `setAllMuted(false)` in the group profile.

Removing group members

The group owner or admin can call the kickGroupMember API to remove a group member. As a live streaming group (AVChatRoom) can have unlimited members, it does not support this API. You can use muteGroupMember to achieve the same effect instead.
After the member is removed, all group members (including the removed member) receive the onMemberKicked callback.

Changing group member roles

The group owner can call setGroupMemberRole to change the role of a member of a social networking group (Public) or temporary meeting group (Meeting). Roles available for changing are ordinary member and group admin.

- After a member is set as group admin, all group members (including the new admin) receive the onGrantAdministrator callback.
- After the admin role is removed for a member, all group members (including the member with admin role removed) receive the onRevokeAdministrator callback.

Transferring a group

The group owner can call transferGroupOwner to transfer the ownership of the group to another group member.
After the group ownership is transferred, all group members receive the onGroupInfoChanged callback, where the type of `V2TIMGroupChangeInfo` is `V2TIMGroupChangeInfo.V2TIM_GROUP_INFO_CHANGE_TYPE_OWNER` and the value is the UserID of the new group owner.

FAQs

1. Can a live streaming group (AVChatRoom) continue to receive messages after it is disconnected and then reconnected?

Yes, but since live streaming groups (AVChatRoom) do not support storing message history in the cloud, it cannot pull the messages that were sent when it was disconnected.

2. Why doesn’t the group receive notifications when a user joins or quits the group?

Verify the group type:

- Temporary meeting groups (Meeting) do not support member change notifications.
- Live streaming groups (AVChatRoom) can receive up to 40 messages per second, and it prioritizes the receiving and sending of high-priority messages and discards messages with the lowest priority first once the frequency limit is exceeded.

3. Why does the unread count of temporary meeting groups (Meeting) remain at 0?

Temporary meeting groups (Meeting) and live streaming groups (AVChatRoom) are designed for conference and live streaming scenarios respectively, and they do not support the unread count feature.

Group Management (iOS)

Last updated : 2021-11-11 11:01:35

Group Types

Instant Messaging (IM) supports the following group types:

- A **work group for friends (Work)** is like an ordinary WeChat group. After a work group is created, a user can only join the group by being invited by a friend who is a member of the group. The invitation does not need to be accepted by the invitee or approved by the group owner.
- A **public group (Public)** is like a QQ group. After a public group is created, the group owner can designate group admins. To join the group, a user needs to search for the group ID and send a request, and the request needs to be approved by the group owner or an admin before the user can join the group.
- A **meeting group (Meeting)** allows users to join and exit freely and supports viewing message history from before the user joined the group. Meeting groups are ideal for scenarios that integrate Tencent Real-Time Communication (TRTC), such as audio and video conferences and online education.
- A **live streaming group (AVChatRoom)** allows users to join and exit freely, supports unlimited number of members and does not store message history. Livestreaming groups can be used with Live Video Broadcasting (LVB) to support the on-screen comment chatting scenario.

The following table describes the features and limitations of each group type:

Feature	Work Group for Friends (Work)	Public Group (Public)	Meeting Group (Meeting)	Live Streaming Group (AVChatRoom)
Available member roles	Group owner and ordinary member	Group owner, group admin, and ordinary member	Group owner, group admin, and ordinary member	Group owner and ordinary member
Requesting to join a group	Not supported	Supported with group owner or group admin	Supported with no approval required	Supported with no approval required

		approval required		
Joining group via invitation by a member	Supported	Not supported	Not supported	Not supported
Group owner quitting group	Supported	Not supported	Not supported	Not supported
Who can modify group profile	Any group member	Group owner and group admin	Group owner and group admin	Group owner
Who can kick group members out of group	Group owner	Group owner and group admin. Group admin can only remove ordinary group members.		Group members cannot be removed. The same effect can be achieved by muting members.
Who can mute members	Muting members is not supported	Group owner and group admin. Group admin can only mute ordinary group members.		Group owner
Unread count	Supported	Supported	Not supported	Not supported
Viewing message history earlier than user's entry time	Not supported	Not supported	Supported	Not supported
Retaining message history in the cloud	• Trial Edition: 7 days • Pro Edition: 7 days by default, up to 360 days via value-added service • Flagship Edition: 30 days by default, up to 360 days via value-added service			Not supported
Number of groups	• Trial Edition: up to 100 existing groups, and disbanded groups do not count against the quota • Pro Edition or Flagship Edition: unlimited			• Trial Edition: up to 10 existing groups, and disbanded groups do not count against the quota • Pro Edition: up to 50 existing groups, and disbanded groups do not

		count against the quota You can upgrade to unlimited number of live streaming groups by purchasing value-added service • Flagship Edition: unlimited
Number of group members	• Trial Edition: 20 per group • Pro Edition: 200 per group by default, can be increased to 2,000 per group via value-added service • Flagship Edition: 2,000 per group, can be increased to 6,000 per group via value-added service	Unlimited number of group members

Note：
In the Pro Edition or Flagship Edition SDKAppID, the maximum net increase in group quantity per day is 10,000 for all group types. Free peak group count is 100,000 per month, and you will need to pay for usage exceeding the free quota.

Group Management

Creating a group

Simple API

You can quickly create a group by calling the createGroup API and specifying `groupType`, `groupID`, and `groupName`.

Advanced API

If you want to initialize group information (for example, group introduction, group profile photo, and initial group members) when creating a group, call the createGroup API in the `V2TIMManager+Group.h` management class.

```
// Sample code: create a work group using the advanced createGroup API
V2TIMGroupInfo *info = [[V2TIMGroupInfo alloc] init];
info.groupName = @"testWork";
info.groupType = @"Work";
NSMutableArray *memberList = [NSMutableArray array];
```

```
V2TIMCreateGroupMemberInfo *memberInfo = [[V2TIMCreateGroupMemberInfo alloc] init];
memberInfo.userID = @"vinson";
[memberList addObject:memberInfo];
[[V2TIMManager sharedInstance] createGroup:info memberList:memberList succ:^(NSString *groupID) {
// Group created successfully
} fail:^(int code, NSString *msg) {
// Failed to create the group
}];
```

- The value of `groupType` is a string and can be one of "Work", "Public", "Meeting", and "AVChatRoom". To learn about the differences among group types, see Group Types.
- `groupID` specifies the group ID, which uniquely identifies a group. Do not create groups with the same `groupID` in a SDKAppID. If you set `groupID` to nil, you will be assigned a group ID by default.
- `groupName` specifies the group description, which has a maximum length of 30 bytes.

Joining a group

The processes for joining groups of different types are described as follows:

Type	Work Group (Work)	Social Networking Group (Public)	Temporary Meeting Group (Meeting)	Live Streaming Group (AVChatRoom)
How to join group	Must be invited by group member	User joins group after request is approved by group owner or admin	User can join freely	User can join freely

Scenario 1: users can join and quit the group freely

Meeting groups (Meeting) and live streaming groups (AVChatRoom) can be used for interactive scenarios where users join and exit groups frequently, such as online conference and runway live streaming. The group joining procedure is therefore the simplest.

After a user successfully joins a group by calling joinGroup, all group members (including the joined user), receive the onMemberEnter callback.

Scenario 2: users must be invited to join the group

Resembling WeChat and WeCom groups, work groups (Work) are suitable for communication in work environments. The interaction pattern is designed to disable proactive group joining and only allows users to be invited to join the group by group members.

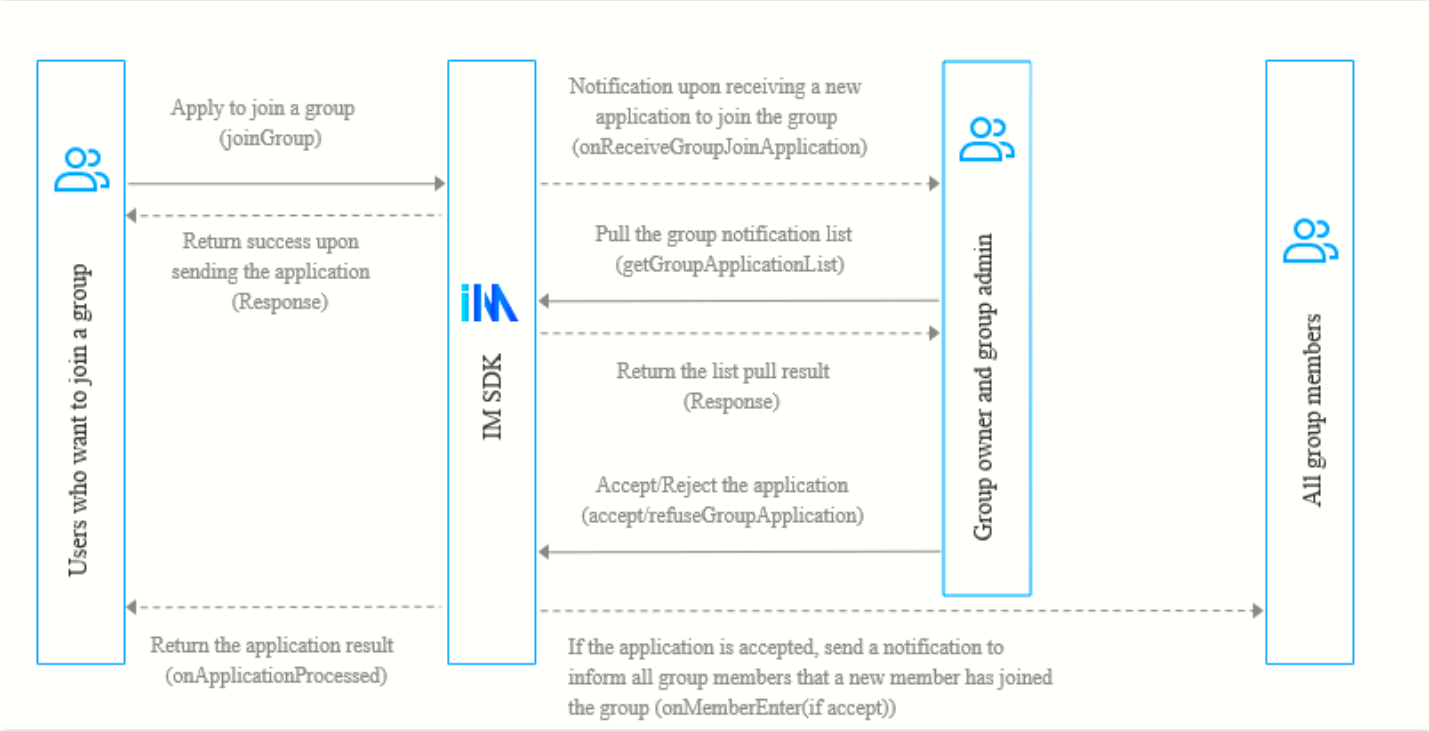
A group member calls inviteUserToGroup to invite a user to group, then all group members (including the inviter) receive the onMemberInvited callback.

Scenario 3: users join the group after requests are approved

Social networking groups (Public) are similar to the interest groups and tribes in QQ. Any user can request to join the group, but will not become a member of the group until the request is approved by the group owner or admin. While approval is required by default, the group owner or admin can call the setGroupInfo API to set the group joining option (`V2TIMGroupAddOpt`) to "forbid anyone to join" which is tighter, or to "disable the approval process", which is more flexible.

- V2TIM_GROUP_ADD_FORBID: forbid anyone to join the group
- V2TIM_GROUP_ADD_AUTH: (default) group owner or admin approval is required for group joining.
- V2TIM_GROUP_ADD_ANY: disable the approval process to allow any user to join the group.

The following diagram illustrates the process of group joining that requires approval:



1. **The user sends a request to join the group**
   The user calls joinGroup to join the group.
2. **The group owner or admin process the group joining application**
   After the onReceiveJoinApplication callback is received, the group owner or admin calls getGroupApplicationList to get the list of group joining applications, and approves or rejects an application with acceptGroupApplication or refuseGroupApplication.

3. **The user receives the result**
   The user receives the onApplicationProcessed callback in V2TIMGroupListener. If `isAgreeJoin` is `true`, the application is approved, otherwise the application is rejected. If the application is approved, all members (including the requestor) receive the onMemberEnter callback.

Quitting a group

Call quitGroup to quit a group. The user who quits the group then receives the onQuitFromGroup callback and other group members receive the onMemberLeave callback.

> Note：
> For group owner of social networking group (Public), temporary meeting group (Meeting), and live streaming group (AVChatRoom) who is not allowed to quit the group, the group owner can disband the group.

Deleting groups

Call dismissGroup to disband a group. Then all group members receive the onGroupDismissed callback.

> Note：
>
> - For a social networking group (Public), temporary meeting group (Meeting), and live streaming group (AVChatRoom), the group owner can disband the group at any time.
> - For a work group (Work), the group owner does not have the privilege to disband the group. To disband the group, you must have your service server call the RESTful API Disbanding a Group.

Getting the list of joined groups

Call getJoinedGroupList to get a list of work groups (Work), social networking groups (Public), and temporary meeting groups (Meeting) the current user has joined. Live streaming groups (AVChatRoom) will not be included in this list.

Group Profiles and Group Settings

Getting group profiles

Call getGroupsInfo to get the group profile of one or more groups at a time. To get the group profiles of multiple groups by a single call, pass in multiple `groupID` at a time.

Modifying group profiles

Call setGroupInfo to modify the group profile. When the modification is complete, all group members receive the onGroupInfoChanged callback.

> Note :
>
> - For work groups (Work), all group members can modify the basic group profile.
> - For social networking groups (Public) and temporary meeting groups (Meeting), only the group owner and admin can modify the basic group profile.
> - For live streaming groups (AVChatRoom), only the group owner can modify the group profile.

```
// Sample code: modify group profile
V2TIMGroupInfo *info = [[V2TIMGroupInfo alloc] init];
info.groupID = @"the ID of the group for which you want to modify the group profile";
info.faceURL = @"http://xxxx";
[[V2TIMManager sharedInstance] setGroupInfo:info succ:^{
// Group profile modified successfully
} fail:^(int code, NSString *msg) {
// Failed to modify the group profile
}];
```

Setting the group message receiving option

Any group member can call the setGroupReceiveMessageOpt API to modify the group message receiving option. Available values are as follows:

- V2TIM_GROUP_RECEIVE_MESSAGE: messages will be received when the user is online and APNs push notifications will be received when the user is offline.
- V2TIM_GROUP_NOT_RECEIVE_MESSAGE: no group messages will be received.
- V2TIM_GROUP_RECEIVE_NOT_NOTIFY_MESSAGE: messages will be received when the user is online and no push notification will be received when the user is offline.

The group message receiving option allows you to mute group messages:

- **No group message will be received**
  With the group message receiving option set to `V2TIM_GROUP_NOT_RECEIVE_MESSAGE`, no group message will be received, and the conversation list will not be updated.

- **Group messages will be received but the user will not be notified. A badge without the unread count will be displayed on the conversation list interface**

  Note :
  This mode requires the unread count feature and therefore it applies only to work groups (Work) and social networking groups (Public).

  With the group message receiving option set to `V2TIM_GROUP_RECEIVE_NOT_NOTIFY_MESSAGE`, when new group messages are received and the conversation list needs to update, get the unread count through unreadCount. Use recvOpt to verify that the group message receiving option is `V2TIM_GROUP_RECEIVE_NOT_NOTIFY_MESSAGE` and then display a badge without the unread count.

Group Attributes (Custom Group Fields)

New custom group fields, also called group attributes, are designed based on API 2.0. They have the following features:

1. You can CRUD group attributes in the client instead of console.
2. You can configure up to 16 group attributes. The size of each group attribute can be up to 4 KB, and the total size of all group attributes can be up to 16 KB.
3. Only live streaming groups (AVChatRoom) support group attributes.
4. The `initGroupAttributes`, `setGroupAttributes`, and `deleteGroupAttributes` APIs each can be called by the SDK for up to 10 times per 5 seconds, and the 8511 error code will be called back if the limit is exceeded. The APIs each can be called by the backend for up to 5 times per second, and the 10049 error code will be called back if the limit is exceeded.
5. The `getGroupAttributes` API can be called by the SDK for up to 20 times per 5 seconds.

With group attributes, you can manage the seats of audio chat rooms. When a user mics on, you can set a group attribute to manage the information of the user. When the user mics off, you can delete the group attribute. Other members can get the list of group attributes to display the seat list.

Initializing group attributes

Call the initGroupAttributes API to initialize group attributes. If the group already has group attributes, the existing group attributes will be cleared.

Setting group attributes

Call the setGroupAttributes API to set group attributes. If the group attributes to set do not exist, they will be automatically added.

Deleting group attributes

Call the deleteGroupAttributes API to delete specified group attributes. If the `keys` field is set to `nil`, all group attributes will be cleared.

Getting group attributes

Call the getGroupAttributes API to get specified group attributes. If the `keys` field is set to `nil`, all group attributes will be got.

Updating group attributes

If group attributes have any updates, all group attribute fields will be called back via the onGroupAttributeChanged API.

Group Member Management

Getting the group member list

Call getGroupMemberList to get the list of group members of a given group. The list contains profile information about individual members, such as user ID (`userID`), group name card (`nameCard`), profile photo (`faceUrl`), nickname (`nickName`), and time of joining group (`joinTime`).
As a group might have a large number of members (for example, 5,000+), this API supports two advanced attributes: `filter` and `nextSeq` .

Filters

When calling the getGroupMemberList API, you can specify `filter` to pull the information list of certain roles.

Filter	Description
V2TIM_GROUP_MEMBER_FILTER_ALL	Pull the information list of all group members
V2TIM_GROUP_MEMBER_FILTER_OWNER	Pull the information list of the group owner
V2TIM_GROUP_MEMBER_FILTER_ADMIN	Pull the information list of the group admin
V2TIM_GROUP_MEMBER_FILTER_COMMON	Pull the information list of all group members

```
// Sample code: pull the profile of the group owner using the filter parameter
[[V2TIMManager sharedInstance] getGroupMemberList:@"testGroup" filter:V2TIM_GROUP_MEMBER_FILTER_O
WNER
nextSeq:0 succ:^(uint64_t nextSeq, NSArray<V2TIMGroupMemberInfo *> *memberList) {
// Pulled successfully
} fail:^(int code, NSString *msg) {
// Messages fail to be pulled
}];
```

Pulling paginated results with nextSeq

In many cases, it makes more sense for the user interface to display the first page of the group member list instead of the complete list. More group members can be pulled when the user clicks **Next Page** or pull the list to refresh. For this scenario, you can apply the method of pulling paginated results.

The getGroupMemberList API returns a maximum of 50 members at a time. You can use the pagination flag `nextSeq` to pull the paginated group member list. In the first attempt to pull the group member list, enter 0 for `nextSeq` . When the first pull succeeds, the `getGroupMemberList` callback result V2TIMGroupMemberInfoResult contains the `nextSeq` field.

- If `nextSeq` is 0, the complete group member list has been pulled.
- If `nextSeq` is greater than 0, there remains group member information to be pulled. You can then decide whether to make another call to pull group member information based on the user's action on the UI. In the second pull, you need to pass the `nextSeq` in the `V2TIMGroupMemberInfoResult` returned from the previous pull as parameter to the getGroupMemberList API.

```
// Sample code: pull the paginated group member list using nextSeq
[[V2TIMManager sharedInstance] getGroupMemberList:@"testGroup" filter:V2TIM_GROUP_MEMBER_FILTER_A
LL nextSeq:0
succ:^(uint64_t nextSeq, NSArray<V2TIMGroupMemberInfo *> *memberList) {
// If nextSeq is greater than 0, make another pull
if (nextSeq > 0) {
[[V2TIMManager sharedInstance] getGroupMemberList:@"testGroup" filter:V2TIM_GROUP_MEMBER_FILTER_A
LL
nextSeq:nextSeq succ:^(uint64_t nextSeq, NSArray<V2TIMGroupMemberInfo *> *memberList) {
// The second pull succeeded
} fail:^(int code, NSString *msg) {
// The second pull failed
}];
}
// The first pull succeeded
} fail:^(int code, NSString *msg) {
```

```
// The first pull failed
}];
```

Getting group member profiles

Call getGroupMembersInfo to get the profile of group members in batches. You can pass in multiple `userID` at a time to improve network transmission efficiency.

Modifying group member profiles

The group owner or admin can call the setGroupMemberInfo API to modify information of group members, including group name card (`nameCard`), group member role (`role`), and muting duration (`muteUntil`).

Muting group members

The group owner or admin can mute a group member and set muting duration (in seconds) via muteGroupMember. Muting information is stored in the muteUntil field of the group member. After the group member is muted, all group members (including the muted member) receive the onMemberInfoChanged callback.
The group owner or admin can mute the entire group via the setGroupInfo API by setting allMuted to `true` . There is no time limit for muting a group. To unmute a group, set `allMuted` to `NO` .

Removing group members

The group owner or admin can call the kickGroupMember API to remove a group member. As a live streaming group (AVChatRoom) can have unlimited members, it does not support the API. You can use muteGroupMember to achieve the same effect instead.
After the member is removed, all group members (including the removed member) receive the onMemberKicked.

Changing group member roles

The group owner can call setGroupMemberRole to change the role of a member of social networking group (Public) or temporary meeting group (Meeting). Roles available for changing are ordinary members and group admins.

- After a member is set as group admin, all group members (including the new admin) receive the onGrantAdministrator callback.
- After the admin role is removed for a member, all group members (including the member with admin role removed) receive the onRevokeAdministrator callback.

Transferring a group

The group owner can call transferGroupOwner to transfer the ownership of the group to other group members.
After the group ownership is transferred, all group members receive the onGroupInfoChanged callback, where the type of `V2TIMGroupChangeInfo` is `V2TIM_GROUP_INFO_CHANGE_TYPE_OWNER` and the value is the UserID of the new group owner.

FAQs

1. Can a live streaming group (AVChatRoom) continue to receive messages after it was disconnected and then reconnected?

Yes, but since live streaming group (AVChatRoom) does not support storing message history in the cloud, it cannot pull the messages that were sent when it was disconnected.

2. Why doesn’t the group receive notifications when a user joins or quits the group?

Verify the group type:

- Temporary meeting group (Meeting) does not support member change notifications.
- Live streaming group (AVChatRoom) can receive up to 40 messages per second, therefore it prioritizes the receiving and sending of high-priority messages and discards messages with the lowest priority first once the frequency limit is exceeded.

3. Why does the unread count of temporary meeting group (Meeting) remain at 0?

Temporary meeting group (Meeting) and live streaming group (AVChatRoom) are designed for conference and live streaming scenarios respectively, and they do not support the unread count feature.

Group Management (Web & Mini Program)

Last updated : 2021-04-30 17:10:47

Group Overview

Instant Messaging (IM) supports multiple group types. For more information on their characteristics and limits, see Group System. A group is identified by a unique ID that enables different operations.

Group Management

Obtaining your group list

This API is used to obtain your group list when you need to render or refresh **My Groups**. For more information, see Group.

> ⚠ **Note :**
>
> The list of groups returned by this API does not include TIM.TYPES.GRP_AVCHATROOM (live streaming groups).

API

```
tim.getGroupList();
```

Request Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Required	Description
`groupProfileFilter`	`Array<String>`	`<optional>`	Group profile filter. The system specifies some profile fields to retrieve by default. can specify additional group profile fields retrieve. Valid values: TIM.TYPES.GRP_PROFILE_OWNER_ID: gro owner ID TIM.TYPES.GRP_PROFILE_CREATE_TIME: group creation time

			TIM.TYPES.GRP_PROFILE_LAST_INFO_TIM the last time the group profile was modif TIM.TYPES.GRP_PROFILE_MEMBER_NUM: number of group members TIM.TYPES.GRP_PROFILE_MAX_MEMBER_I the maximum number of group members TIM.TYPES.GRP_PROFILE_JOIN_OPTION: th options for joining the group TIM.TYPES.GRP_PROFILE_INTRODUCTION group introduction TIM.TYPES.GRP_PROFILE_NOTIFICATION: group announcement TIM.TYPES.GRP_PROFILE_MUTE_ALL_MBR the Mute All setting, supported by v2.6.2 later.

Response

This API returns a `Promise` object.

- The callback parameter of `then` is IMResponse. `IMResponse.data.groupList` contains the list of groups.
- The callback parameter of `catch` is IMError.

Example

- Group profile information that is retrieved by default:

```
// This API retrieves the following information by default: group type, group name, group prof
ile photo, and the time of the last message.
let promise = tim.getGroupList();
promise.then(function(imResponse) {
console.log(imResponse.data.groupList); // Group list.
}).catch(function(imError) {
console.warn('getGroupList error:', imError); // Information on the failure in obtaining the g
roup list.
});
```

- Other group profile information to retrieve:

```
// If the profile fields that are retrieved by default fail to meet your requirements, you can
retrieve additional profile fields by referring to the following code.
let promise = tim.getGroupList({
groupProfileFilter: [TIM.TYPES.GRP_PROFILE_OWNER_ID],
});
promise.then(function(imResponse) {
```

```
console.log(imResponse.data.groupList); // Group list.
}).catch(function(imError) {
console.warn('getGroupList error:', imError); // Information on the failure in obtaining the g
roup list.
});
```

Obtaining the detailed group profile

For more information, see Group.

API

```
tim.getGroupProfile(options);
```

Request Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Required	Description
`groupID`	`String`	-	Group ID.
`groupCustomFieldFilter`	`Array<String>`	`<optional>`	The filter for group custom fields. You can specify group custom fields to retrieve. For more information, see Custom Fields.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. You can obtain group profiles from `IMResponse.data.group` .
- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.getGroupProfile({
groupID: 'group1',
groupCustomFieldFilter: ['key1','key2']
});
promise.then(function(imResponse) {
console.log(imResponse.data.group);
}).catch(function(imError) {
console.warn('getGroupProfile error:', imError); // Information on the failure in obtaining the d
```

```
etailed group profile.
});
```

Creating a group

For more information, see Group.

> **⚠ Note :**
>
> After this API is used to create TIM.TYPES.GRP_AVCHATROOM (live streaming group), you must call joinGroup to join the group and enable the messaging process.

API

```
tim.createGroup(options);
```

Request Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Required	Default Value	De
`name`	`String`	-	-	Re len
`type`	`String`	`<optional>`	`TIM.TYPES.GRP_WORK`	Gr • (de • gr • me • liv
`groupID`	`String`	`<optional>`	-	Gr gr
`introduction`	`String`	`<optional>`	-	Gr 24
`notification`	`String`	`<optional>`	-	Gr by
`avatar`	`String`	`<optional>`	-	Th val

`maxMemberNum`	`Number`	`<optional>`	-	The The gro ten live
`joinOption`	`String`	`<optional>`	`TIM.TYPES.JOIN_OPTIONS_FREE_ACCESS`	The join **sp ter liv ** this **dis ** and set • ` use • TIM app • ` use Thi TIM TIM TIM car ten gro
`memberList`	`Array<Object>`	`<optional>`	-	Init nur live see bel
`groupCustomField`	`Array<Object>`	`<optional>`	-	Gro spe cus

`memberList` parameter description

Name	Type	Required	Description
`userID`	`String`	-	Required. The UserID of the group member.
`role`	`String`	`<optional>`	The role of the group member. The only available value is Admin, which means to add the user and set the user as an admin.
`memberCustomField`	`Array<Object>`	`<optional>`	Group member custom fields. No custom field is specified by default. To create a group member custom field, see Custom Fields.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. You can obtain group profiles from `IMResponse.data.group` .
- The callback parameter of `catch` is IMError.

Example

```
// Create a work group.
let promise = tim.createGroup({
type: TIM.TYPES.GRP_WORK,
name: 'WebSDK',
memberList: [{userID: 'user1'}, {userID: 'user2'}] // If memberList is specified, userID must als
o be specified.
});
promise.then(function(imResponse) { // Created successfully.
console.log(imResponse.data.group); // The profile of the created group.
}).catch(function(imError) {
console.warn('createGroup error:', imError); // Information on the failure in creating the group.
});
```

Disbanding a group

This API is used by a group owner to disband a group.

> ⚠ **Note :**
>
> The group owner cannot disband work groups.

API

```
tim.dismissGroup(groupID);
```

Request Parameters

Name	Type	Description
`groupID`	`String`	Group ID.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. You can obtain the ID of the disbanded group from `IMResponse.data.groupID` .
- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.dismissGroup('group1');
promise.then(function(imResponse) { // Disbanded successfully.
console.log(imResponse.data.groupID); // The ID of the disbanded group.
}).catch(function(imError) {
console.warn('dismissGroup error:', imError); // Information on the failure in disbanding the gro
up.
});
```

Updating a group profile

API

```
tim.updateGroupProfile(options);
```

Request Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Required	Default	De
`groupID`	`Object`	-	-	Gro
`name`	`Object`	`<optional>`	-	Gro byt

`avatar`	`Object`	`<optional>`	-	The val
`introduction`	`Object`	`<optional>`	-	Gro 24(
`notification`	`Object`	`<optional>`	-	Gro byt
`maxMemberNum`	`Number`	`<optional>`	-	The The
`muteAllMembers`	`Boolean`	-	-	The set fals
`joinOption`	`String`	`<optional>`	`TIM.TYPES.JOIN_OPTIONS_FREE_ACCESS`	The joir • use • TIM app • use !Th TIM TIM mo gro gro
`groupCustomField`	`Array<Object>`	`<optional>`	-	Gro inf par No cre Cu

`groupCustomField` parameter description

Name	Type	Description

`key`	`String`	The key of the custom field.
`value`	`String`	The value of the custom field.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. You can obtain the modified group profile from `IMResponse.data.group` .
- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.updateGroupProfile({
groupID: 'group1',
name: 'new name', // Modify the group name.
introduction: 'this is introduction.', // Modify the group notice.
// Starting from v2.6.0, group members can receive group prompts about custom group field modific
ations and obtain related content. For more information, see Message.payload.newGroupProfile.grou
pCustomField.
groupCustomField: [{ key: 'group_level', value: 'high'}] // Modify the group-level custom field.
});
promise.then(function(imResponse) {
console.log(imResponse.data.group) // The detailed group profile after modification.
}).catch(function(imError) {
console.warn('updateGroupProfile error:', imError); // Information on the failure in modifying th
e group profile.
});
```

Applying to join a group

This API is called when a user applies to join a group.

⚠ Note :

- Users cannot apply to join a work group and can only be added to the group through the addGroupMember method.
- TIM.TYPES.GRP_AVCHATROOM (livestreaming groups) support two ways to join a group:
- Users join the group in the normal manner after login. All APIs in the SDK can be called normally.
- Users join the group anonymously without login. Messages will be received, but any APIs that require authentication cannot be called.

- Only TIM.TYPES.GRP_AVCHATROOM (livestreaming groups) support anonymous group joining.
- Users can join only one livestreaming group at a time. **For example**, when a user joins live streaming group B when already in live streaming group A, the SDK quits group A first and then joins group B.

API

```
tim.joinGroup(options);
```

Request Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Required	Description
`groupID`	`String`	-	-
`applyMessage`	`String`	-	Remarks on the application.
`type`	`String`	`<optional>`	The type of the group the user wants to join. Valid values: • TIM.TYPES.GRP_PUBLIC: social networking group • TIM.TYPES.GRP_MEETING: temporary meeting group • TIM.TYPES.GRP_AVCHATROOM: livestreaming group

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data` contains the following property values:

Name	Description
status	The status of the group joining process. Valid values: ◦ TIM.TYPES.JOIN_STATUS_WAIT_APPROVAL: waiting for the admin’s approval ◦ TIM.TYPES.JOIN_STATUS_SUCCESS: joined successfully ◦ TIM.TYPES.JOIN_STATUS_ALREADY_IN_GROUP: already in the group
group	The group profile displayed when the user joins the group.

- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.joinGroup({ groupID: 'group1', type: TIM.TYPES.GRP_AVCHATROOM });
promise.then(function(imResponse) {
switch (imResponse.data.status) {
case TIM.TYPES.JOIN_STATUS_WAIT_APPROVAL: break; // Waiting for the admin's approval.
case TIM.TYPES.JOIN_STATUS_SUCCESS: // Joined the group successfully.
console.log(imResponse.data.group); // The profile of the group.
break;
case TIM.TYPES.JOIN_STATUS_ALREADY_IN_GROUP: //Already in the group
break;
default: break;
}
}).catch(function(imError){
console.warn('joinGroup error:', imError); // Information on the failure in joining the group.
});
```

Quitting a group

A group owner can only quit work groups. After the group owner quits, the work group has no group owner.

API

```
tim.quitGroup(groupID);
```

Request Parameters

Name	Type	Description
`groupID`	`String`	Group ID.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data.groupID` is the ID of the group that the user quits.
- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.quitGroup('group1');
promise.then(function(imResponse) {
console.log(imResponse.data.groupID); // The ID of the group that the user quits.
```

```
}).catch(function(imError){
console.warn('quitGroup error:', imError); // Information on the failure in quitting the group.
});
```

Searching for a group by group ID

This API is used to search for a group by group ID.

> Note: work groups (TIM.TYPES.GRP_WORK) can not be searched for.

API

```
tim.searchGroupByID(groupID);
```

Request Parameters

Name	Type	Description
`groupID`	`String`	Group ID.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data.group` is the group profile.
- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.searchGroupByID('group1');
promise.then(function(imResponse) {
const group = imResponse.data.group; // Group profile.
}).catch(function(imError) {
console.warn('searchGroupByID error:', imError); // Information on the failure in searching for the group.
});
```

Transferring a group

This API is used to transfer a group. Only the group owner has the permission to perform this operation.

Note: only the group owner can transfer the ownership of a group. Livestreaming groups (TIM.TYPES.GRP_AVCHATROOM) cannot be transferred.

API

```
tim.changeGroupOwner(options);
```

Request Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Description
`groupID`	`String`	The ID of the group to be transferred.
`newOwnerID`	`String`	The ID of the new group owner.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data.group` is the group profile.
- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.changeGroupOwner({
groupID: 'group1',
newOwnerID: 'user2'
});
promise.then(function(imResponse) { // Transferred successfully.
console.log(imResponse.data.group); // Group profile.
}).catch(function(imError) { // Failed to transfer the group.
console.warn('changeGroupOwner error:', imError); // Information on the failure in transferring t
he group.
});
```

Processing an application to join a group

This API is used to approve or reject an application to join a group.

> ⚠ **Note :**
>
> If a group has multiple administrators, all online administrators receive a group system notification when someone applies to join the group. If one of the administrators has approved or rejected the application, other administrators cannot change the result.

API

```
tim.handleGroupApplication(options);
```

Request Parameters

The `options` parameter is of the `Object` type. It contains the following property values:

Name	Type	Required	Description
`handleAction`	`String`	-	Processing result. Agree: the application is approved. Reject: the application is rejected.
`handleMessage`	`String`	`<optional>`	Remarks.
`message`	Message	-	The message instance of the **Group System Notification** for an application to join a group. This instance can be obtained in the following ways: • The callback function parameter of the event for new group system notifications • The messages for system sessions

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data.group` is the group profile.
- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.handleGroupApplication({
handleAction: 'Agree',
handleMessage: 'Welcome',
message: message // The message instance of the group system notification for an application to j
oin a group.
```

```
});
promise.then(function(imResponse) {
console.log(imResponse.data.group); // Group profile.
}).catch(function(imError){
console.warn('handleGroupApplication error:', imError); // Error information.
});
```

Setting the notification type for group messages

API

```
tim.setMessageRemindType(options);
```

Request Parameters

The `options` parameter is of the `Object` type. It contains the following property values:

Name	Type	Description
`groupID`	`String`	Group ID.
`messageRemindType`	`String`	The notification type of group messages. Valid values: • TIM.TYPES.MSG_REMIND_ACPT_AND_NOTE: the SDK receives a message and throws a message receiving event to notify the access side, which then sends a notification. • TIM.TYPES.MSG_REMIND_ACPT_NOT_NOTE: the SDK receives a message and throws a message receiving event to notify the access side, which then does not send any notifications. • TIM.TYPES.MSG_REMIND_DISCARD: the SDK rejects a message and does not throw any message receiving events.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data.group` is the group profile.
- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.setMessageRemindType({ groupID: 'group1', messageRemindType: TIM.TYPES.MSG_REMI
ND_DISCARD }); // Reject the message.
promise.then(function(imResponse) {
```

```
console.log(imResponse.data.group); // The group profile that is displayed after modification.
}).catch(function(imError) {
console.warn('setMessageRemindType error:', imError);
});
```

Group Member Management

Getting group member list

⚠ Note :

- Starting from v2.6.2, this API can be used to pull the muting stop timestamps of group members (muteUntil). Based on its value, the access side can find out whether the member is muted and the remaining muting time.
- In versions earlier than v2.6.2, this API can be used to get profile information including profile photo and nickname, which is sufficient to meet the rendering requirements of the group member list. To get detailed information such as member muting stop timestamp (muteUntil), use getGroupMemberProfile.
- This API is used to pull a paginated list of group members and not the complete list. To get the complete list of group members (memberNum), use getGroupProfile.

For more information, see GroupMember.

API

```
tim.getGroupMemberList(options);
```

Request Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Required	Default Value	Description
`groupID`	`String`	-	-	Group ID.
`count`	`Number`	`<optional>`	`15`	The number of members whose IDs are to be retrieved. The maximum value is 100, which avoids request failure caused by excessively large returned packets. If the IDs of more than 100 members are passed in, only the IDs of the first 100 members will be retrieved.

`offset`	`Number`	`<optional>`	`0`	Offset. The IDs are retrieved from 0 by default.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data.memberList` is the list of group members. For more information, see GroupMember.
- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.getGroupMemberList({ groupID: 'group1', count: 30, offset:0 }); // Pull 30 grou
p members starting from 0.
promise.then(function(imResponse) {
console.log(imResponse.data.memberList); // Group member list.
}).catch(function(imError) {
console.warn('getGroupMemberList error:', imError);
});
// Starting from v2.6.2, this API can be used to pull the muting stop timestamps of group member
s.
let promise = tim.getGroupMemberList({ groupID: 'group1', count: 30, offset:0 }); // Pull 30 grou
p members starting from 0.
promise.then(function(imResponse) {
console.log(imResponse.data.memberList); // Group member list.
for (let groupMember of imResponse.data.memberList) {
if (groupMember.muteUntil * 1000 > Date.now()) {
console.log(`${groupMember.userID} muted`);
} else {
console.log(`${groupMember.userID} not muted`);
}
}
}).catch(function(imError) {
console.warn('getGroupMemberProfile error:', imError);
});
```

Getting the profiles of group members

⚠ Note :

- This API requires SDK v2.2.0 or later.
- The maximum number of users in each query is 50. If the length of the array passed in is greater than 50, only the first 50 users will be queried and the rest will be discarded.

For more information, see GroupMember.

API

```
tim.getGroupMemberProfile(options);
```

Request Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Required	Description
`groupID`	`String`	-	Group ID.
`userIDList`	`Array.<String>`	-	IDs of group members whose profiles you want to query.
`memberCustomFieldFilter`	`Array.<String>`	`<optional>`	The filter for group member custom fields. If this field is not specified, all the group member custom fields are queried by default.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data.memberList` is the list of group members whose profiles are queried successfully. For more information, see GroupMember.
- The callback parameter of `catch` is IMError.

Adding group members

The rules for adding group members are as follows:

- TIM.TYPES.GRP_PRIVATE (work group): any group member can invite users to the group and acceptance by the invitee is not required.
- TIM.TYPES.GRP_PUBLIC (social networking group)/TIM.TYPES.GRP_CHATROOM (temporary meeting group): only the app admin can invite users to the group and acceptance by the invitee is not required.
- TIM.TYPES.GRP_AVCHATROOM (livestreaming group): no member (including the app admin) is allowed to invite any user to the group.

For more information, see GroupGroupMember and Differences in joining a group.

API

```
tim.addGroupMember(options);
```

Request Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Description
`groupID`	`String`	Group ID.
`userIDList`	`Array<String>`	An array of IDs of the group members to be added. Up to 300 group members can be added at a time.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data` contains the following property values:

Name	Type	Description
successUserIDList	Array<String>	List of userIDs that were added successfully
failureUserIDList	Array<String>	List of userIDs that failed to be added
existedUserIDList	Array<String>	List of userIDs that were already in the group
group	Group	Group profile that is displayed after the API is called

- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.addGroupMember({
groupID: 'group1',
userIDList: ['user1','user2','user3']
});
promise.then(function(imResponse) {
console.log(imResponse.data.successUserIDList); // The userIDList of group members that were added successfully.
console.log(imResponse.data.failureUserIDList); // The userIDList of group members that failed to be added.
```

```
console.log(imResponse.data.existedUserIDList); // The userIDList of group members that were already in the group.
console.log(imResponse.data.group); // The group profile that is displayed after the group members are added.
}).catch(function(imError) {
console.warn('addGroupMember error:', imError); // Error information.
});
```

Deleting group members

This API is used to delete group members. Only the group owner can delete group members.

API

```
tim.deleteGroupMember(options)
```

Request Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Description
`groupID`	`String`	Group ID.
`userIDList`	`Array<String>`	List of IDs of the group members that are to be deleted.
`reason`	`String`	Reason for deleting the one or more group members. This field is optional.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data.group` is the updated group profile.
- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.deleteGroupMember({groupID: 'group1', userIDList:['user1'], reason: 'You are deleted from the group because you have violated the group rules.'});
promise.then(function(imResponse) {
console.log(imResponse.data.group); // Group profile that is displayed after one or more group members are deleted.
console.log(imResponse.data.userIDList); // List of userIDs of the deleted group members.
}).catch(function(imError) {
```

```
console.warn('deleteGroupMember error:', imError); // Error information.
});
```

Muting or unmuting a group member

This API is used to set the muting duration for a group member. You can mute or unmute a group member. The muting and unmuting features are unavailable in work groups (TIM.TYPES.GRP_WORK).

Note :

Only the group owner and the group admin have the permissions for this operation.

- The group owner can mute and unmute the admin and ordinary group members.
- The admin can mute and unmute ordinary group members.

API

```
tim.setGroupMemberMuteTime(options)
```

Request Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Description
`groupID`	`String`	Group ID.
`userID`	`String`	Group member ID.
`muteTime`	`Number`	The muting duration in seconds. For example, if the muting duration is set to 1,000, the user is muted for 1,000 seconds immediately. If the muting duration is set to 0, the user is unmuted.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data.group` is the group profile displayed after modification.
- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.setGroupMemberMuteTime({
groupID: 'group1',
userID: 'user1',
muteTime: 600 // The user is muted for 10 minutes. If the value is set to 0, the user is unmuted.
});
promise.then(function(imResponse) {
console.log(imResponse.data.group); // The group profile that is displayed after modification.
console.log(imResponse.data.member); // The group member's profile that is displayed after modif
ication.
}).catch(function(imError) {
console.warn('setGroupMemberMuteTime error:', imError); // Information on the failure in muting t
he group member.
});
```

Setting or canceling the admin

This API is used to change the role of a group member. Only the group owner has the permission to perform this operation.

API

```
tim.setGroupMemberRole(options)
```

Request Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Description
`groupID`	`String`	Group ID.
`userID`	`String`	Group member ID
`role`	`String`	Valid values: `TIM.TYPES.GRP_MBR_ROLE_ADMIN` : group admin. `TIM.TYPES.GRP_MBR_ROLE_MEMBER` : ordinary group member.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data.group` is the group profile displayed after modification.
- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.setGroupMemberRole({
groupID: 'group1',
userID: 'user1',
role: TIM.TYPES.GRP_MBR_ROLE_ADMIN // Set user1 as the admin of group1.
});
promise.then(function(imResponse) {
console.log(imResponse.data.group); // The group profile that is displayed after modification.
console.log(imResponse.data.member); // The group member's profile that is displayed after modif
ication.
}).catch(function(imError) {
console.warn('setGroupMemberRole error:', imError); // Error information.
});
```

Modifying a group member's name card

This API is used to modify a group member's name card.

- The group owner can set the name cards of all members.
- The group admin can set its own name card and the name cards of ordinary group members.
- Ordinary group members can only set their own name cards.

API

```
tim.setGroupMemberNameCard(options)
```

Request Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Description
`groupID`	`String`	Group ID.
`userID`	`String<optional>`	The userID of the user whose name card is to be modified. The value is the user's own userID by default.
`nameCard`	`String`	-

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data.group` is the group profile displayed after modification.
- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.setGroupMemberNameCard({ groupID: 'group1', userID: 'user1', nameCard: 'Name card' });
promise.then(function(imResponse) {
console.log(imResponse.data.group); // The group profile that is displayed after modification.
console.log(imResponse.data.member); // The group member's profile that is displayed after modification.
}).catch(function(imError) {
console.warn('setGroupMemberNameCard error:', imError); // Information on the failure in setting a group member's name card.
});
```

Modifying a group member custom field

This API is used to modify a group member custom field.

> Ordinary group members can only modify their own custom fields.

API

```
tim.setGroupMemberCustomField(options)
```

Request Parameters

`options` is of the `Object` type. Its values are as follows:

Name	Type	Description
`groupID`	`String`	Group ID.
`userID`	`String<optional>`	The ID of the group member. If this field is not specified, the user's own group member custom field is modified by default.
`memberCustomField`	`Array<Object>`	The group member custom field to be modified.

The `memberCustomField` parameter contains the following property values:

Name	Type	Description
`key`	`String`	The key of the custom field.

`value`	`String<optional>`	The value of the custom field.

Response

This API returns a `Promise` object.

- The callback function parameter for `then` is IMResponse. `IMResponse.data.group` is the group profile displayed after modification.
- The callback parameter of `catch` is IMError.

Example

```
let promise = tim.setGroupMemberCustomField({ groupID: 'group1', memberCustomField: [{key: 'group_member_test', value: 'test'}]});
promise.then(function(imResponse) {
console.log(imResponse.data.group); // The group profile that is displayed after modification.
console.log(imResponse.data.member); // The group member's profile that is displayed after modification.
}).catch(function(imError) {
console.warn('setGroupMemberCustomField error:', imError); // Information on the failure in modifying the group member custom field.
});
```

Group Notification

This API is used to send a group notification in a group when a user is invited to the group or deleted from the group. The access side can determine whether to display the message to group members as needed or ignore all the messages.
IM provides multiple types of group notifications. For more information, see Message.GroupTipPayload.

Name	Type	Description
`operatorID`	`String`	The ID of the user who performs the operation.
`operationType`	`Number`	The type of the operation.
`userIDList`	`Array<String>`	List of relevant userIDs.
`newGroupProfile`	`Object`	The new group profile to which the group profile needs to be changed.

The parameters of a group notification are described above. The system sends a group notification to all group members at the appropriate time. For example, when a user quits or joins a group, the system sends the corresponding group notification to all group members.

Group System Notification

When a user applies to join a group, the group admin receives a system message about the application. The admin can accept or reject the application, and the IM SDK sends the result to the access side via a corresponding group system notification, which is then displayed to the user by the access side.
IM provides multiple types of system notifications. For more information, see Types, Constants, and Meanings of Group System Notifications.

```
let onGroupSystemNoticeReceived = function(event) {
  const type = event.data.type; // The type of the group system notification. For more information,
  const message = event.data.message; // The message instance of the group system notification. For
  console.log(message.payload); // The content of the message. This describes the payload of the gro
};
tim.on(TIM.EVENT.GROUP_SYSTEM_NOTICE_RECEIVED, onGroupSystemNoticeReceived);
```

Name	Type	Description
`operatorID`	`String`	The ID of the user who performs the operation.
`operationType`	`Number`	The type of the operation.
`groupProfile`	`Object`	The profile of the relevant group.
`handleMessage`	`Object`	Remarks on the processing. For example, if user1 enters remarks on an application to join group1 that requires approval, the admin of group1 will see this field in the group system notification.

`operationType` description

Name	Description	Recipient
1	A user applies to join the group.	The group admin and the group owner
2	Application to join the group is approved.	The applicant
3	Application to join group is rejected.	The applicant

4	A user is deleted from a group.	The user who is deleted
5	The group is deleted.	All group members
6	The group is created.	The creator
7	A user is invited to the group.	The invitee
8	A user quits the group.	The user who quits the group
9	The admin is modified.	The new admin
10	The admin is canceled.	The admin who is canceled
255	A custom notification is triggered.	All group members by default

The parameters of a group system notification are described above. The system sends a group system notification to all group members at the appropriate time. For example, when user1 is deleted from a group, the system sends the corresponding group system notification to user1.

Signaling
Signaling Management (Android)

Last updated : 2021-10-15 16:10:24

Overview

Signaling APIs are a set of invitation process control APIs based on IM messages. They can be used to implement a variety of real-time features, including:

- Audio-video chat rooms: mic on or mic off
- Chatting: audio/video calls as those in WeChat
- Education: control of the process for teachers to invite students to "raise hands"

Features

Signaling APIs support the following features:

One-to-one chat invitation

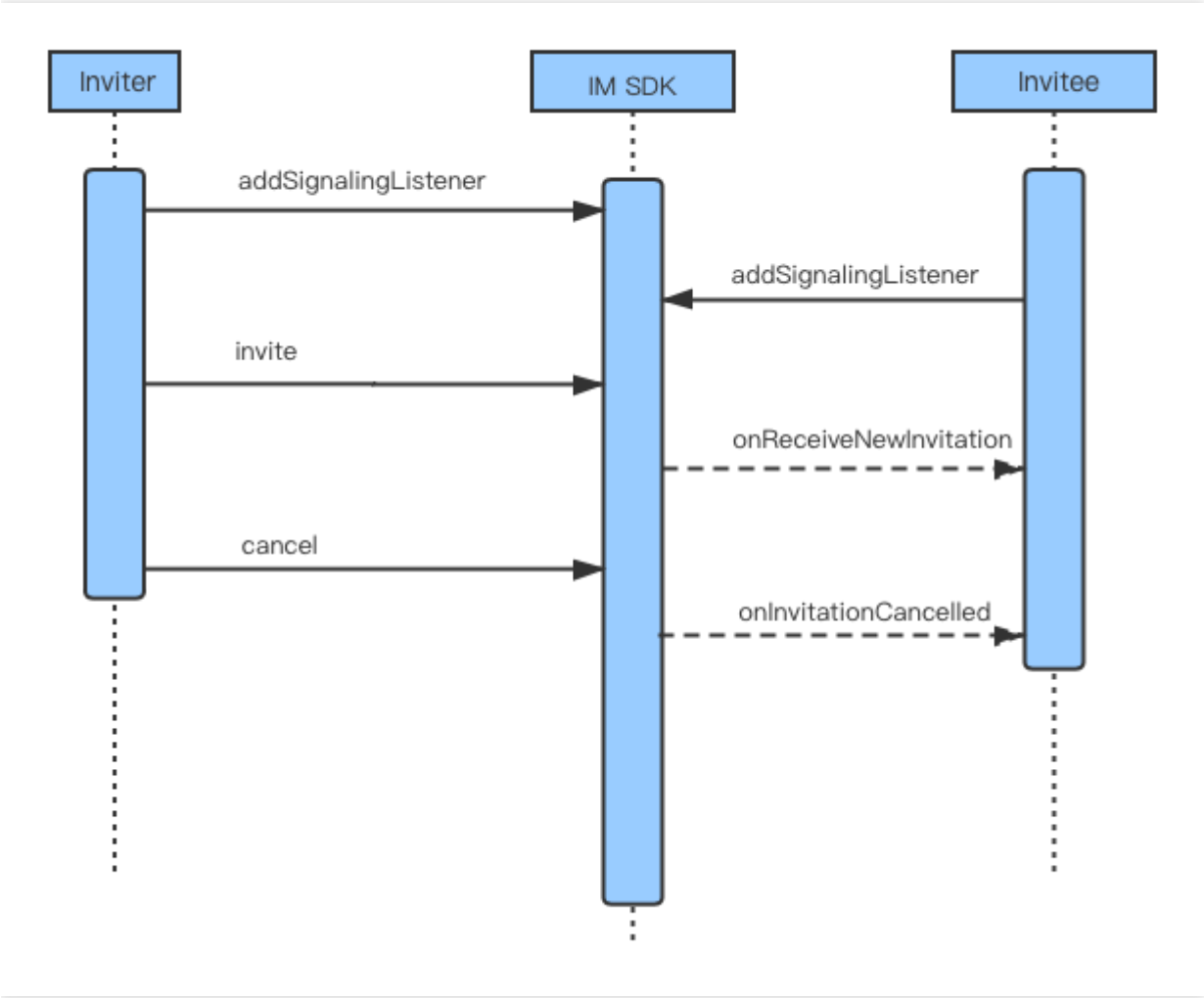
When making a one-to-one chat via the simple message sending or receiving API or rich media message sending or receiving API, you can use the invite signaling API to make an end-to-end call. When receiving the invitation notification onReceiveNewInvitation, the invitee can choose to accept the invitation, reject the invitation, or wait until the invitation times out.

Group chat invitation

First, you need to manage a group by using group management APIs and listen for the group's event callbacks via V2TIMGroupListener. Then members of the group can initiate a group call invitation via inviteInGroup within the group. When receiving the invitation onReceiveNewInvitation, an invitee can choose to accept the invitation, reject the invitation, or wait until the invitation times out.

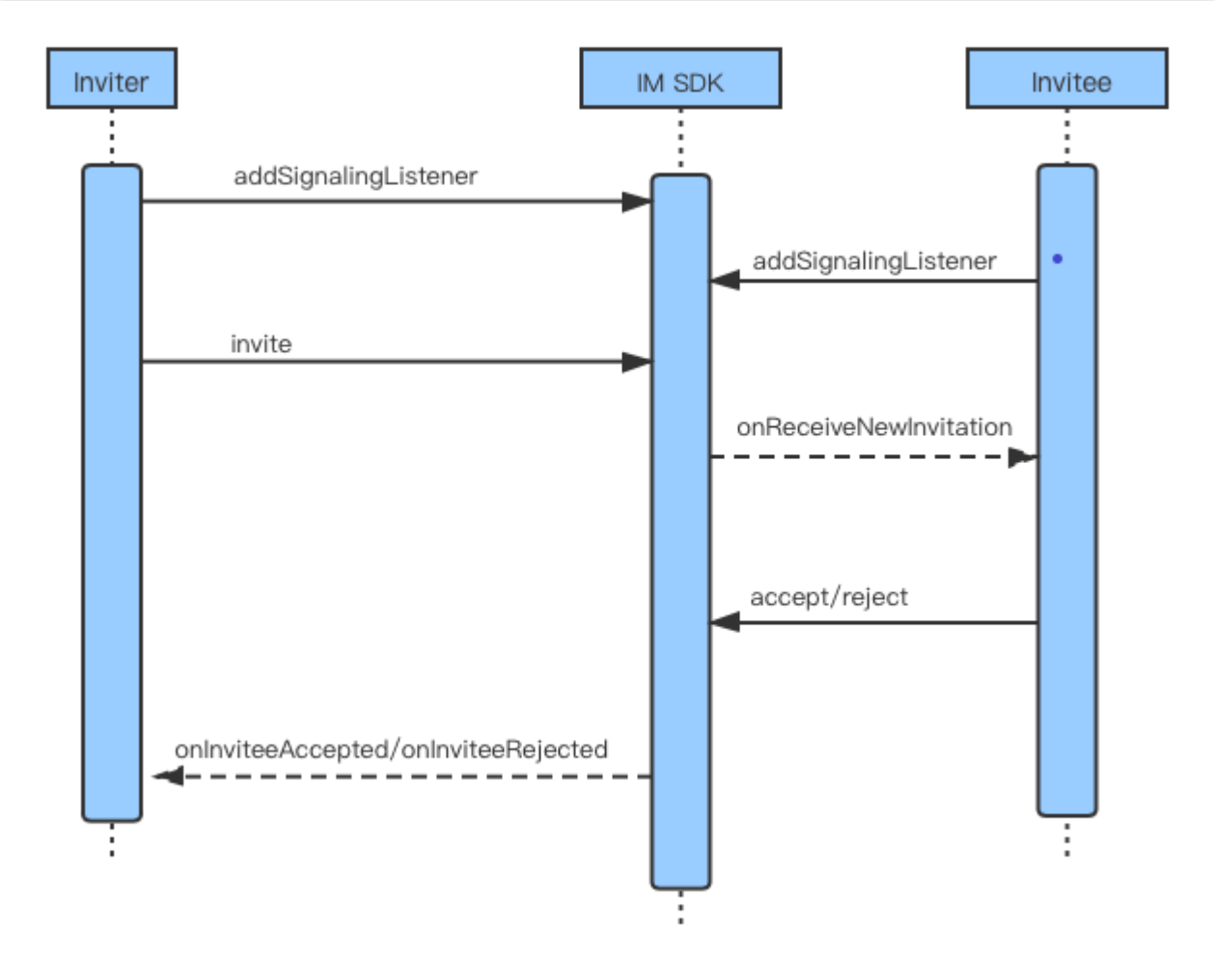
Canceling an invitation

Before an invitation times out and the invitee processes the invitation, the inviter can cancel the invitation via cancel. After the inviter cancels the invitation, an invitee will receive a cancellation notification onInvitationCancelled, and the invitation process ends.



Accepting an invitation

When receiving an invitation onReceiveNewInvitation, an invitee can accept the invitation via accept before the invitation times out and the inviter cancels the invitation. If the invitee accepts the invitation, the inviter will receive an invitation acceptance notification onInviteeAccepted. After the processing (including acceptance, rejection, and timeout) at the invitee side ends, the invitation process ends.

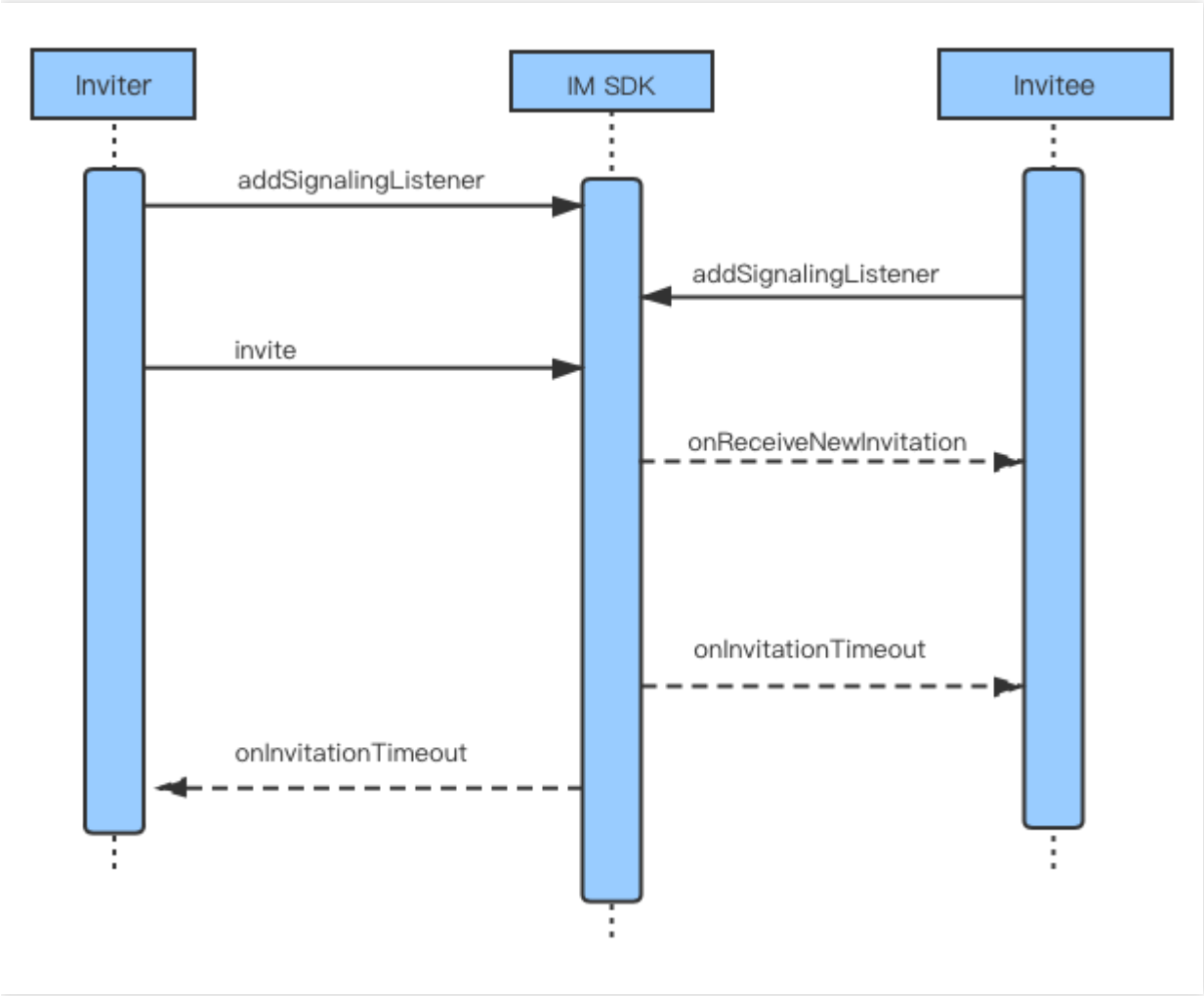


Rejecting an invitation

When receiving an invitation onReceiveNewInvitation, an invitee can reject the invitation via reject before the invitation times out and the inviter cancels the invitation. If the invitee rejects the invitation, the inviter will receive an invitation rejection notificationonInviteeRejected. After the processing (including acceptance, rejection, and timeout) at the invitee side ends, the invitation process ends.

Invitation timeout

If the timeout duration of the invitation API is greater than 0 and an invitee does not respond within the timeout duration, the invitation times out, and the inviter and invitee will receive a timeout notification onInvitationTimeout. After the processing (including acceptance, rejection, and timeout) at the invitee side ends, the invitation process ends. If the timeout duration of the invitation API is 0, there will be no timeout notification.



Use Cases

Audio/Video call

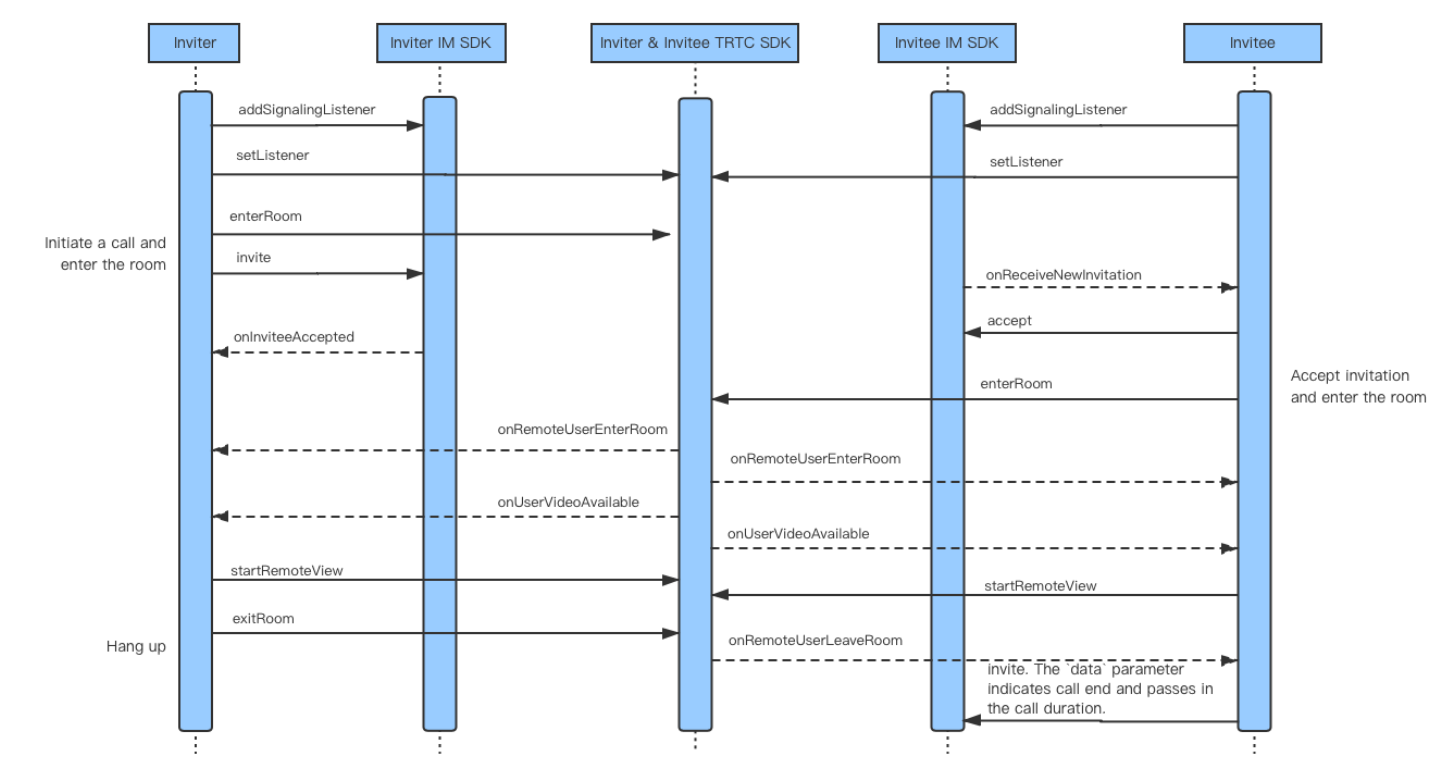
In the open-source project TUIKit Demo, we developed an audio/video call solution suitable for one-to-one and multi-person chatting based on TRTC. You can directly modify the TUIKit demo for adaptation. The following takes the one-to-one video call process as an example to introduce how signaling APIs work with the TRTC SDK.

One-to-one video call process:

1. The inviter enters the TRTC room based on the room ID generated at the service layer and calls the signaling invitation API invite to initiate an audio/video call request, including the room ID in the custom field of the invitation API.
2. The invitee receives the signaling invitation notification onReceiveNewInvitation and gets the room ID based on the custom data. The invitee's phone begins to ring.

3. The invitee processes the invitation notification:
   - Accepting the invitation: the invitee calls the accept signaling API and enters the TRTC room based on the room ID, and calls the `openCamera()` function to enable the local camera. The inviter and invitee receive the `onRemoteUserEnterRoom` callback from the TRTC SDK, and the systems of the two parties record the start time of the call.
   - Rejecting the invitation: the invitee calls the reject signaling API to end the call.
   - If the invitee is on the phone with someone else, the invitee calls the reject signaling API to reject the invitation and specify the rejection reason (busy local line) in the custom data.
4. After the invitee answers the call and the audio/video channel between the inviter and invitee is established, both the inviter and invitee will receive the `onUserVideoAvailable` event notification from the TRTC SDK, which indicates that they have received each other's video image. At this point, they can call the `startRemoteView` API of the TRTC SDK to display the remote video image, and the remote audio will be played back automatically by default.
5. After the call ends (either the inviter or invitee hangs up), the party who hangs up exits the TRTC room. The other party receives the `onRemoteUserLeaveRoom` callback from the TRTC SDK, and its system calculates the total duration of the call and initiates an invitation again, including the call end event and call duration in the custom data to facilitate UI display.

Flowchart



Education: teacher inviting students to "raise hands"

In this scenario, the teacher asks students to "raise hands" and then chooses one of the students to speak. The process is as follows:

1. The teacher calls the inviteInGroup API to invite students to "raise hands", specifying the "hand raising" operation in the custom field `data`. The students receive the onReceiveNewInvitation callback.
2. Based on the `inviteeList` and `data` fields in onReceiveNewInvitation, a student determines that he/she is one of the invitees and the operation is raising hands. Then the student calls the accept API to raise her/his hand.
3. If a student raises her/his hand, others can receive the onInviteeAccepted callback. The system determines that the `data` field is hand raising and displays the list of students who raise hands.
4. The teacher calls the inviteInGroup API to invite one of the students who raise their hands to speak. At this time, the system specifies the "speaking" operation in the custom field `data`. The students receive the onReceiveNewInvitation callback.
5. Based on the `inviteeList` and `data` fields in the onReceiveNewInvitation callback, a student determines that he/she is one of the invitees and the operation is speaking. Then the student calls the accept API to speak.
6. If a student speaks, others can receive the onInviteeAccepted callback, and their systems determine that the `data` field is speaking and display the list of students who speak.

Signaling Management (iOS)

Last updated : 2021-10-15 16:27:04

Overview

Signaling APIs are a set of invitation process control APIs based on IM messages. They can be used to implement a variety of real-time features, including:

- Audio-video chat rooms: mic on or mic off
- Chatting: audio/video calls as those in WeChat
- Education: control of the process for teachers to invite students to "raise hands"

Features

Signaling APIs support the following features:

One-to-one chat invitation

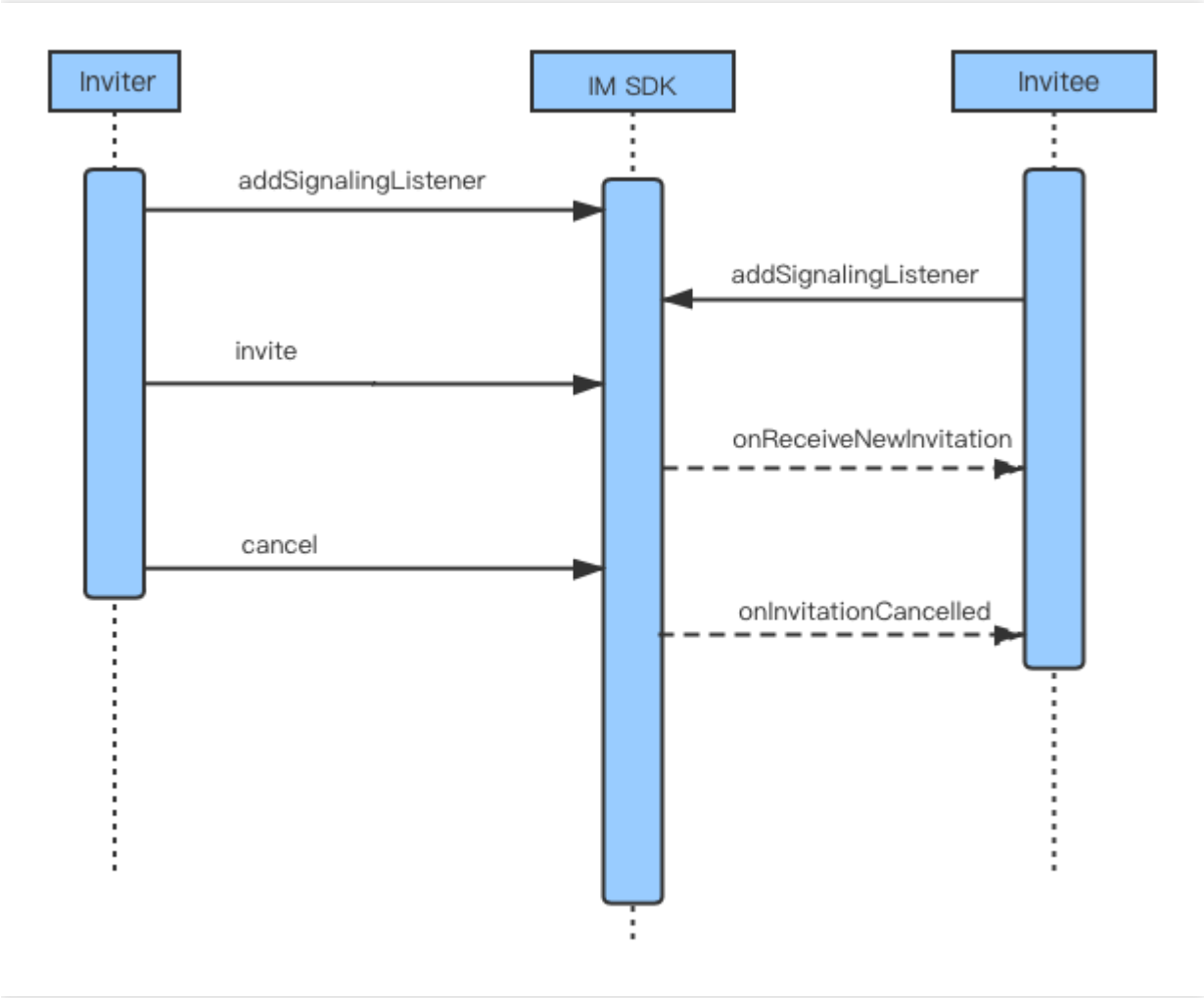
When making a one-to-one chat via the simple message sending or receiving API or rich media message sending or receiving API, you can use the invite signaling API to make an end-to-end call. When receiving the invitation notification onReceiveNewInvitation, the invitee can choose to accept the invitation, reject the invitation, or wait until the invitation times out.

Group chat invitation

First, you need to manage a group by using group management APIs and listen for the group's event callbacks via V2TIMGroupListener. Then members of the group can initiate a group call invitation via inviteInGroup within the group. When receiving the invitation onReceiveNewInvitation, an invitee can choose to accept the invitation, reject the invitation, or wait until the invitation times out.

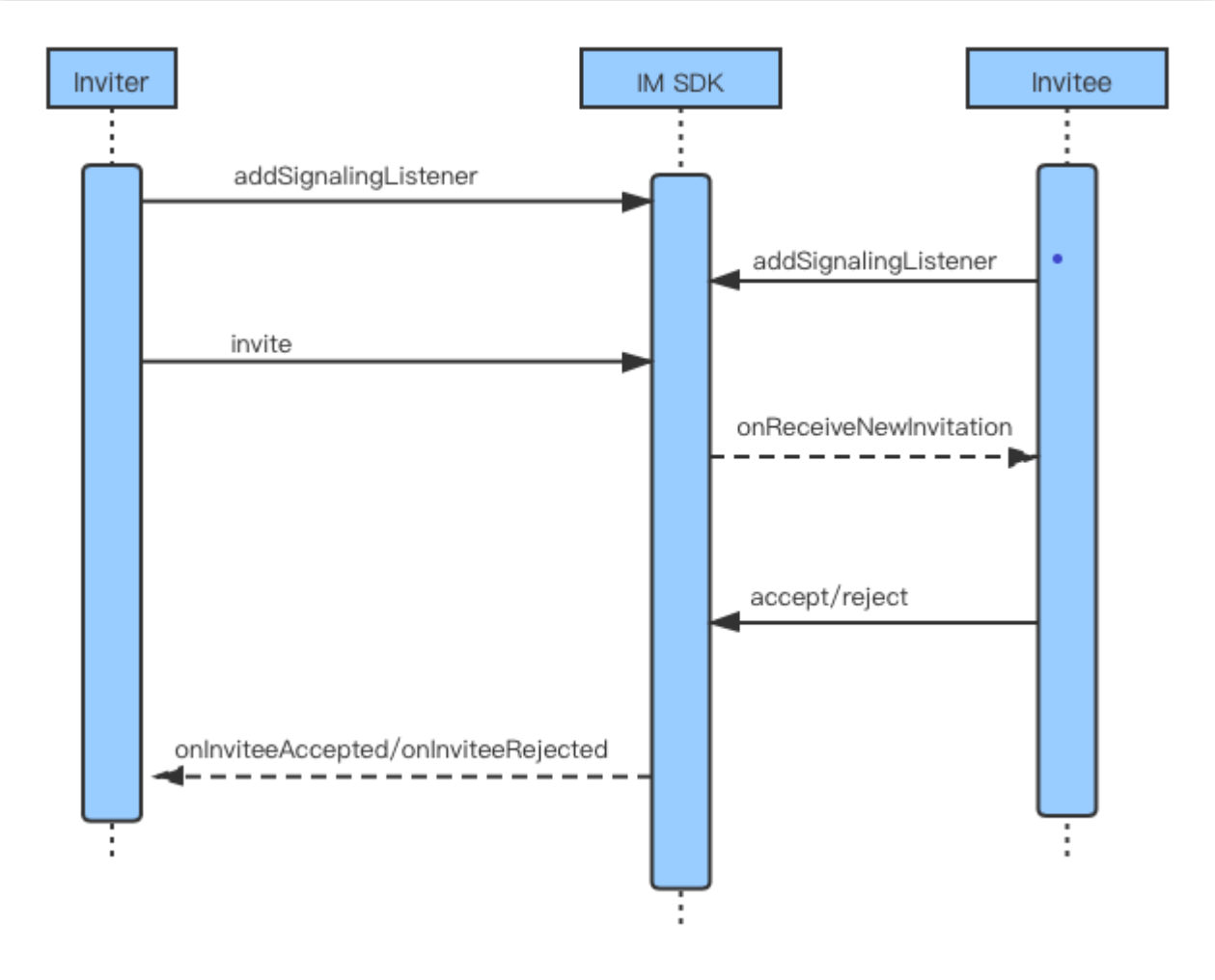
Canceling an invitation

Before an invitation times out and the invitee processes the invitation, the inviter can cancel the invitation via cancel. After the inviter cancels the invitation, an invitee will receive a cancellation notification onInvitationCancelled, and the invitation process ends.



Accepting an invitation

When receiving an invitation onReceiveNewInvitation, an invitee can accept the invitation via accept before the invitation times out and the inviter cancels the invitation. If the invitee accepts the invitation, the inviter will receive an invitation acceptance notification onInviteeAccepted. After the processing (including acceptance, rejection, and timeout) at the invitee side ends, the invitation process ends.

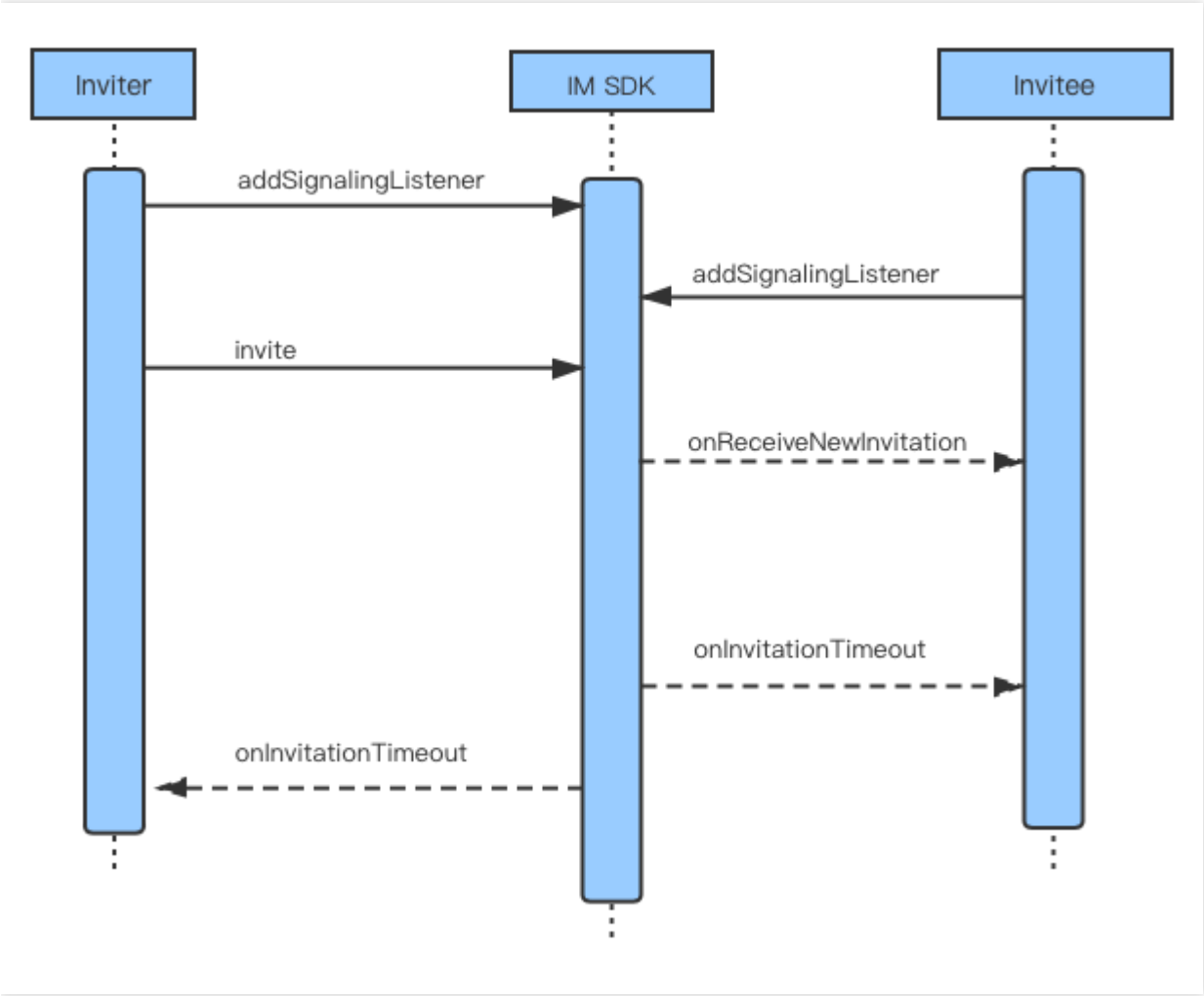


Rejecting an invitation

When receiving an invitation onReceiveNewInvitation, the invitee can reject the invitation via reject before the invitation times out and the inviter cancels the invitation. If the invitee rejects the invitation, the inviter will receive an invitation rejection notification onInviteeRejected. After the processing (including acceptance, rejection, and timeout) at the invitee side ends, the invitation process ends.

Invitation timeout

If the timeout duration of the invitation API is greater than 0 and an invitee does not respond within the timeout duration, the invitation times out, and the inviter and invitee will receive a timeout notification onInvitationTimeout. After the processing (including acceptance, rejection, and timeout) at the invitee side ends, the invitation process ends. If the timeout duration of the invitation API is 0, there will be no timeout notification.



Use Cases

Audio/Video call

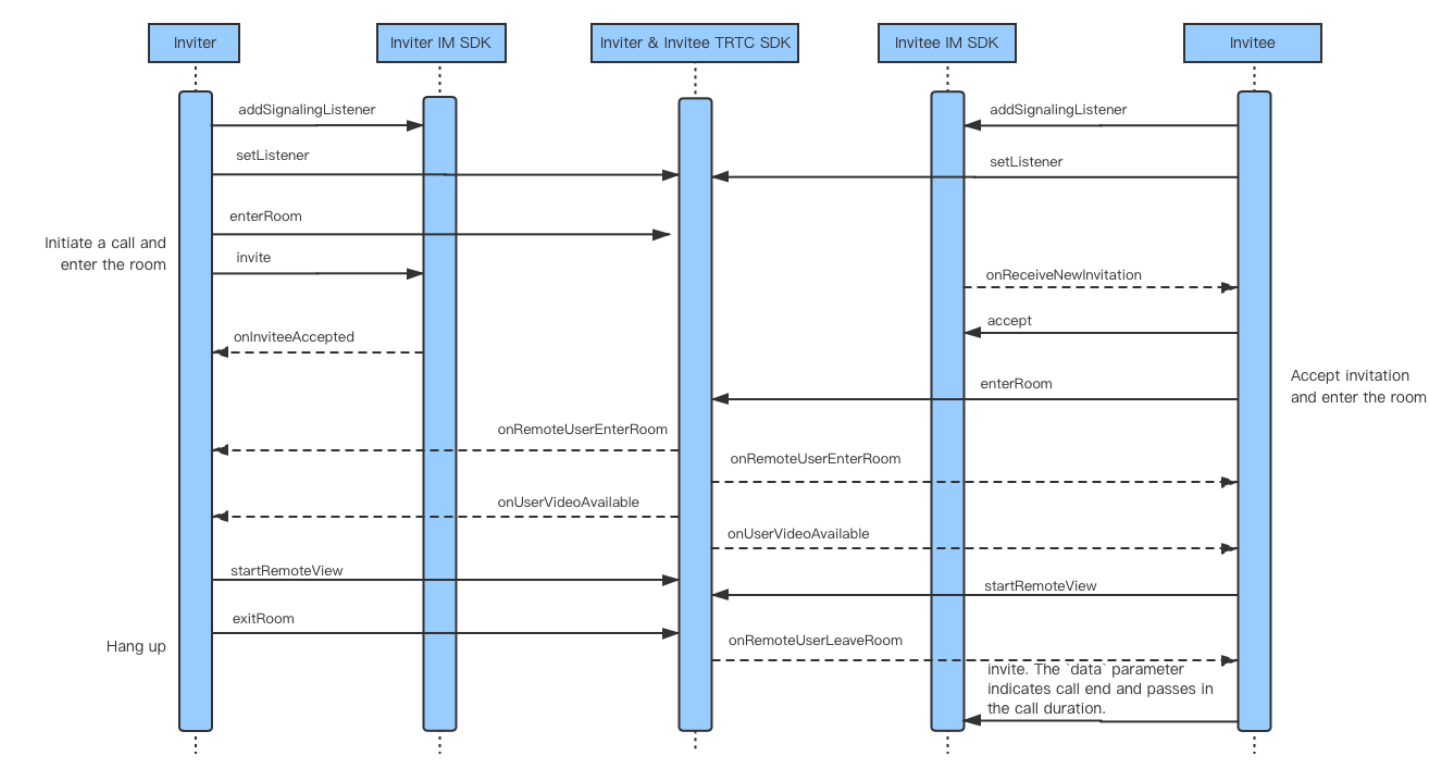
In the open-source project TUIKit Demo, we developed an audio/video call solution suitable for one-to-one and multi-person chatting based on TRTC. You can directly modify the TUIKit demo for adaptation. The following takes the one-to-one video call process as an example to introduce how signaling APIs work with the TRTC SDK.

One-to-one video call process:

1. The inviter enters the TRTC room based on the room ID generated at the service layer and calls the signaling invitation API invite to initiate an audio/video call request, including the room ID in the custom field of the invitation API.
2. The invitee receives the signaling invitation notification onReceiveNewInvitation and gets the room ID based on the custom data. The invitee's phone begins to ring.

3. The invitee processes the invitation notification:
   - Accepting the invitation: the invitee calls the accept signaling API and enters the TRTC room based on the room ID, and calls the `openCamera()` function to enable the local camera. The inviter and invitee receive the `onRemoteUserEnterRoom` callback from the TRTC SDK, and the systems of the two parties record the start time of the call.
   - Rejecting the invitation: the invitee calls the reject signaling API to end the call.
   - If the invitee is on the phone with someone else, the invitee calls the reject signaling API to reject the invitation and specify the rejection reason (busy local line) in the custom data.
4. After the invitee answers the call and the audio/video channel between the inviter and invitee is established, both the inviter and invitee will receive the `onUserVideoAvailable` event notification from the TRTC SDK, which indicates that they have received each other's video image. At this point, they can call the `startRemoteView` API of the TRTC SDK to display the remote video image, and the remote audio will be played back automatically by default.
5. After the call ends (either the inviter or invitee hangs up), the party who hangs up exits the TRTC room. The other party receives the `onRemoteUserLeaveRoom` callback from the TRTC SDK, and its system calculates the total duration of the call and initiates an invitation again, including the call end event and call duration in the custom data to facilitate UI display.

Flowchart



Education: teacher inviting students to "raise hands"

In this scenario, the teacher asks students to "raise hands" and then chooses one of the students to speak. The process is as follows:

1. The teacher calls the inviteInGroup API to invite students to "raise hands", specifying the "hand raising" operation in the custom field `data`. The students receive the onReceiveNewInvitation callback.
2. Based on the `inviteeList` and `data` fields in onReceiveNewInvitation, a student determines that he/she is one of the invitees and the operation is raising hands. Then the student calls the accept API to raise her/his hand.
3. If a student raises her/his hand, others can receive the onInviteeAccepted callback. The system determines that the `data` field is hand raising and displays the list of students who raise hands.
4. The teacher calls the inviteInGroup API to invite one of the students who raise their hands to speak. At this time, the system specifies the "speaking" operation in the custom field `data`. The students receive the onReceiveNewInvitation callback.
5. Based on the `inviteeList` and `data` fields in the onReceiveNewInvitation callback, a student determines that he/she is one of the invitees and the operation is speaking. Then the student calls the accept API to speak.
6. If a student speaks, others can receive the onInviteeAccepted callback, and their systems determine that the `data` field is speaking and display the list of students who speak.

User Profile and Relationship Chain
User Profile and Relationship Chain (Android)

Last updated : 2021-10-15 16:29:49

User Profile Management

Querying and modifying your own profile

Use getUsersInfo to **query your own profile**, where `userIDList` indicates your own UserID.
Use setSelfInfo to **modify your own profile**. You will receive the onSelfInfoUpdated callback after your profile is modified successfully.

Querying the profile of a non-friend

Use getUsersInfo to query the user profile of a non-friend, where `userIDList` indicates the UserID of the user to query.

Querying and modifying a friend's profile

Use getFriendsInfo to **query the profile of a specified friend**. Use `getRelation()` of `V2TIMFriendInfoResult` to obtain your relationship from the callback information:

- `V2TIMCheckFriendResult.V2TIM_FRIEND_RELATION_TYPE_NONE` : the user is not a friend.
- `V2TIMCheckFriendResult.V2TIM_FRIEND_RELATION_TYPE_BOTH_WAY` : the user is a two-way friend.
- `V2TIMCheckFriendResult.V2TIM_FRIEND_RELATION_TYPE_IN_MY_FRIEND_LIST` : the user is in your friend list.

Use setFriendInfo to **modify the information of a specified friend**, such as friend remarks.

Blocking Messages from a Specified User

- **Blocking a user**
  To block messages from a specified user, call the addToBlackList API to block the user.
  By default, the blocked user is unaware of the "blocked" status. An error code indicating blocking will not be returned after the user sends a message. If you want blocked users to receive the blocked message in this situation, see How to display an error message to a blocked user after the user sends a message.

- **Removing a user from the blocklist**
  Call deleteFromBlackList to remove a user from your blocklist and receive the user's messages again.

- **Obtaining the blocklist**
  Call getBlackList to view and manage the blocklist.

Friend Management

Determining whether a friend request is required

By default, the IM SDK does not check the relationship between two parties when sending one-to-one chat messages. This default setting is generally applied in customer service scenarios, where having to friend a customer service agent before chatting is inefficient.
If you want to demand users to be friends before they can chat, as in WeChat and QQ, log in to the IM console, choose **Feature Configuration** > **Login and Message** > **Relationship Check**, and enable "Check Relationship for One-to-One Chat Messages". With this feature enabled, users can only send messages to friends and will receive the 20009 error code from the SDK when sending a message to a non-friend user.

Friend list management

The IM SDK supports relationship chain logic. You can call getFriendList to obtain the friend list, call deleteFromFriendList to delete a friend from the friend list, or call addFriend to add a friend.

The process has the following varations depending on whether friend verification is required.

Friend request approval is not required

1. User A and user B call setFriendListener to set a relationship chain listener.
2. User B calls setSelfInfo and sets `info` to `V2TIM_FRIEND_ALLOW_ANY` through the setAllowType API.
3. User A becomes user B's friend simply by calling addFriend to send a friend request. If setAddType in the request parameter `V2TIMFriendAddApplication` is set to `V2TIMFriendInfo.V2TIM_FRIEND_TYPE_BOTH` (that is, setting as a two-way friend), both users A and B receive the onFriendListAdded callback.
   If this value is set to `V2TIMFriendInfo.V2TIM_FRIEND_TYPE_SINGLE` (that is, setting as a one-way friend), only user A receives the onFriendListAdded callback.

Friend request approval is required

1. User A and user B call setFriendListener to set a relationship chain listener.

2. User B calls setSelfInfo and sets `info` to `V2TIM_FRIEND_NEED_CONFIRM` through the setAllowType API.
3. User A calls addFriend to send a friend request to user B. getResultCode in the success callback parameter `V2TIMFriendOperationResult` returns the 30539 error code, indicating that user B's approval is required in this case. At this time, both users A and B receive the onFriendApplicationListAdded callback.
4. User B receives the onFriendApplicationListAdded callback. If getType in the parameter `V2TIMFriendApplication` is set to `V2TIMFriendApplication.V2TIM_FRIEND_APPLICATION_COME_IN`, user B can accept or reject the request.

- User B calls acceptFriendApplication to accept the friend request. If the acceptance type is `V2TIMFriendApplication.V2TIM_FRIEND_ACCEPT_AGREE` (that is, accepting as a one-way friend), user A receives the onFriendListAdded callback, indicating that a one-way relationship has been established. Meanwhile, user B receives the onFriendApplicationListDeleted callback, indicating that user B is now in user A's friend list, but user A is not in user B's friend list.
- If the acceptance type is `V2TIMFriendApplication.V2TIM_FRIEND_ACCEPT_AGREE_AND_ADD` (that is, accepting as a two-way friend), both users A and B receive the onFriendListAdded callback, indicating that they are now in each other's friend list.
- User B calls refuseFriendApplication to reject the friend request and both users receive the onFriendApplicationListDeleted callback.

Friend group management

To group friends into categories such as "classmates" and "coworkers", call the following APIs.

Feature	API
Create a friend group	createFriendGroup
Delete a friend group	deleteFriendGroup
Modify a friend group	renameFriendGroup
Obtain the list of friend groups	getFriendGroupList
Add friends to a friend group	addFriendsToFriendGroup
Delete friends from a friend group	deleteFriendsFromFriendGroup

FAQs

How can I disable messaging between two users who are not friends?

By default, the IM SDK does not prevent message sending and receiving between strangers. If you want messages to be sent or received only between friends, log in to the IM console, choose **Feature Configuration** > **Login and Messages** > **Relationship Check**, and enable **Check Relationship for One-to-One Chat Messages**. After this feature is enabled, you can send messages only to friends. When you try to send messages to strangers, the IM SDK returns the 20009 error code.

2. How do I enable error messages when blocked users try to send a message?

When a user is added to the blocklist, by default, the user does not know that he/she is in the blocklist. That is, after this user sends a message, the user is still prompted that the message was sent successfully, but in fact, the recipient will not receive the message. If you want a user in the blocklist to know that his/her message failed to be sent, log in to the IM console, choose **Feature Configuration** > **Login and Messages** > **Blocklist Check**, and disable **Show "Sent successfully" After Sending Messages**. After this feature is disabled, the IM SDK will return the 20007 error code when a user in the blocklist sends a message.

3. Why can't the SDK enhanced edition get the latest user profiles?

There are two types of user profile updates in the enhanced SDK: friend's profile and stranger's profile:

- Friend's profile: when the profile of a friend is updated, the backend will send a system notification to the SDK, so the friend's profile will be updated in real time.
- Stranger's profile: when the profile of a stranger is updated, the backend will not send any system notification because the stranger is not a friend of yours, so the stranger's profile will not be updated in real time. To avoid sending a network request to the backend every time the user profile is obtained, the SDK adds a caching logic, setting a 10-minute interval between pulls of the same user's profile from the backend.

User Profile and Relationship Chain (iOS)

Last updated : 2021-10-15 16:31:08

User Profile Management

Querying and modifying your own profile

Use getUsersInfo to **query your own profile**, where `userIDList` indicates your own UserID.
Use setSelfInfo to **modify your own profile**. You will receive the onSelfInfoUpdated callback after the profile is modified successfully.

Querying the profile of a non-friend

Use getUsersInfo to query the user profile of a non-friend, where `userIDList` indicates the UserID of the user to query.

Querying and modifying a friend’s profile

Use getFriendsInfo to **query the profile of a specified friend**. Use `getRelation()` of `V2TIMFriendInfoResult` to obtain your relationship from the callback information:

- `V2TIM_FRIEND_RELATION_TYPE_NONE` : the user is not a friend.
- `V2TIM_FRIEND_RELATION_TYPE_BOTH_WAY` : the user is a two-way friend.
- `V2TIM_FRIEND_RELATION_TYPE_IN_MY_FRIEND_LIST` : the user is in your friend list.

Use setFriendInfo to **modify the information of a specified friend**, such as friend remarks.

Blocking Messages from a Specified User

- **Blocking a user**
  To block messages from a specified user, call the addToBlackList API to block the user.
  By default, the blocked user is unaware of the "blocked" status. An error code indicating blocking will not be returned after the user sends a message. If you want blocked users to receive error message in this situation, see How to display an error message to a blocked user after the user sends a message.

- **Removing a user from the blocklist**
  Call deleteFromBlackList to remove a user from the blocklist and receive the user's messages again.

- **Obtaining the blocklist**
  Call getBlackList to view and manage the blocklist.

Friend Management

Determining whether a friend request is required

By default, the IM SDK does not check the relationship between two parties when sending one-to-one chat messages. This default setting is generally applied in customer service scenarios, where having to friend a customer service agent before chatting is inefficient.
If you want to demand users to be friends before they can chat, as in WeChat and QQ, log in to the IM console, choose **Feature Configuration** > **Login and Message** > **Relationship Check**, and enable "Check Relationship for One-to-One Chat Messages". With this feature enabled, users can only send messages to friends and will receive the 20009 error code from the SDK when sending a message to a non-friend user.

Friend list management

The IM SDK supports relationship chain logic. You can call getFriendList to obtain the friend list, call deleteFromFriendList to delete a friend from the friend list, or call addFriend to add a friend.

The process has the following varations depending on whether friend verification is required.

Friend request approval is not required

1. User A and user B call setFriendListener to set a relationship chain listener.
2. User B calls setSelfInfo and sets allowType to `V2TIM_FRIEND_ALLOW_ANY`.
3. User A becomes user B's friend simply by calling addFriend to send a friend request.

- If addType in the request parameter `V2TIMFriendAddApplication` is set to `V2TIM_FRIEND_TYPE_BOTH` (that is, setting as a two-way friend), both users A and B receive the onFriendListAdded callback;
- If this value is set to `V2TIM_FRIEND_TYPE_SINGLE` (that is, setting as a one-way friend), only user A receives the onFriendListAdded callback.

Friend request approval is required

1. User A and user B call setFriendListener to set a relationship chain listener.
2. User B calls setSelfInfo and sets allowType to `V2TIM_FRIEND_NEED_CONFIRM`.

3. User A calls addFriend to send a friend request to user B. `resultCode` in the success callback parameter `V2TIMFriendOperationResult` returns the 30539 error code, indicating that user B's approval is required in this case. At this time, both users A and B receive the onFriendApplicationListAdded callback.
4. User B receives the onFriendApplicationListAdded callback. If type in the parameter `V2TIMFriendApplication` is set to `V2TIM_FRIEND_APPLICATION_COME_IN`, user B can accept or reject the request.
   - User B calls acceptFriendApplication to accept the friend request. If the acceptance type is `V2TIM_FRIEND_ACCEPT_AGREE` (that is, accepting as a one-way friend), user A receives the onFriendListAdded callback, indicating that a one-way relationship has been established. Meanwhile, user B receives the onFriendApplicationListDeleted callback, indicating that user B is now in user A's friend list, but user A is not in user B's friend list.

- User B calls acceptFriendApplication to accept the friend request. If the acceptance type is `V2TIM_FRIEND_ACCEPT_AGREE_AND_ADD` (that is, accepting as a two-way friend), both users A and B receive the onFriendListAdded callback, indicating that they are now in each other's friend list.
  - User B calls refuseFriendApplication to reject the friend request and both users receive the onFriendApplicationListDeleted callback.

Friend group management

To group friends into categories such as "classmates" and "coworkers", call the following APIs.

Feature	API
Create a friend group	createFriendGroup
Delete a friend group	deleteFriendGroup
Modify a friend group	renameFriendGroup
Obtain the list of friend groups	getFriendGroupList
Add friends to a friend group	addFriendsToFriendGroup
Delete friends from a friend group	deleteFriendsFromFriendGroup

FAQs

How can I disable messaging between two users who are not friends?

By default, the IM SDK does not prevent message sending and receiving between strangers. If you want messages to be sent or received only between friends, log in to the IM console, choose **Feature Configuration** > **Login and Messages** > **Relationship Check**, and enable **Check Relationship for One-to-One Chat Messages**. After this feature is enabled, you can send messages only to friends. When you try to send messages to strangers, the IM SDK returns the 20009 error code.

2. How do I enable error messages when blocked users try to send a message?

When a user is added to the blocklist, by default, the user does not know that he/she is in the blocklist. That is, after this user sends a message, the user is still prompted that the message was sent successfully, but in fact, the recipient will not receive the message. If you want a user in the blocklist to know that his/her message failed to be sent, log in to the IM console, choose **Feature Configuration** > **Login and Messages** > **Blocklist Check**, and disable **Show "Sent successfully" After Sending Messages**. After this feature is disabled, the IM SDK will return the 20007 error code when a user in the blocklist sends a message.

3. Why can't the SDK enhanced edition get the latest user profiles?

There are two types of user profile updates in the enhanced SDK: friend's profile and stranger's profile:

- Friend's profile: when the profile of a friend is updated, the backend will send a system notification to the SDK, so the friend's profile will be updated in real time.
- Stranger's profile: when the profile of a stranger is updated, the backend will not send any system notification because the stranger is not a friend of yours, so the stranger's profile will not be updated in real time. To avoid sending a network request to the backend every time the user profile is obtained, the SDK adds a caching logic, setting a 10-minute interval between pulls of the same user's profile from the backend.

User Profile (Web & Mini Program)

Last updated : 2021-04-30 17:12:49

User Profile

Obtaining your personal profile

This API is used to obtain your personal profile. For more information on the properties, see Profile.

> ⚠ **Note :**
>
> Custom profile fields are supported since SDK v2.3.2. Before using this API, upgrade your SDK to v2.3.2 or later.

API name

```
tim.getMyProfile()
```

Returned values

This API returns a `Promise` object. The callback functions are as follows:

- The callback function parameter for `then` is IMResponse. You can obtain your personal profile from `IMResponse.data` .
- The callback function parameter for `catch` is IMError.

Sample

```
let promise = tim.getMyProfile();
promise.then(function(imResponse) {
console.log(imResponse.data); // Personal profile from the profile instance
}).catch(function(imError) {
console.warn('getMyProfile error:', imError); // Information on the failure in obtaining the pers
onal profile.
});
```

Obtaining other users' profiles

This API is used to obtain standard profile fields and Custom Profile Fields.

⚠ Note :

- Custom profile fields are supported since SDK v2.3.2. Before using this API, upgrade your SDK to v2.3.2 or later.
- If you have not configured any custom profile fields or you have configured custom profile fields but have not set the values, this API will not return custom profile information.
- You may pull profiles of up to 100 users to avoid failure caused by a large amount of data returned. If the length of an array passed in is greater than 100, only the first 100 users will be queried and the rest are discarded.

API name

```
tim.getUserProfile(options)
```

Request Parameters

The `options` parameter is of the `Object` type. It contains the following property values:

Name	Type	Description
`userIDList`	`Array<String>`	UserIDs. The value is of array type.

Returned values

This API returns a `Promise` object. The callback functions are as follows:

- The callback function parameter for `then` is IMResponse. You can obtain other users' profiles from `IMResponse.data` .
- The callback function parameter for `catch` is IMError.

Sample

```
let promise = tim.getUserProfile({
userIDList: ['user1', 'user2'] // Note: even if you retrieve only one user's profile, the value must be of the array type, for example, userIDList: ['user1'].
});
promise.then(function(imResponse) {
console.log(imResponse.data); // The array that stores other users' profiles - [Profile]
}).catch(function(imError) {
console.warn('getUserProfile error:', imError); // Information on the failure in obtaining other users' profiles.
});
```

Updating your personal profile

> ⚠ **Note :**
>
> Custom profile fields are supported since SDK v2.3.2. Before using this API, upgrade your SDK to v2.3.2 or later.

API name

```
tim.updateMyProfile(options)
```

Request Parameters

The `options` parameter is of the `Object` type. It contains the following property values:

Name	Type	Description
`nick`	`String`	Nickname.
`avatar`	`String`	URL of the avatar .
`gender`	`String`	Gender. Valid values: • TIM.TYPES.GENDER_UNKNOWN: unspecified • TIM.TYPES.GENDER_FEMALE: female • TIM.TYPES.GENDER_MALE: male
`selfSignature`	`String`	Personal signature.
`allowType`	`String`	Specifies whether a friending request sent to the user must be verified. • TIM.TYPES.ALLOW_TYPE_ALLOW_ANY: the friending request is automatically accepted without verification. • TIM.TYPES.ALLOW_TYPE_NEED_CONFIRM: the friending request must be verified. • TIM.TYPES.ALLOW_TYPE_DENY_ANY: the friending request is rejected.
`birthday`	`Number`	Birthday. It is recommended that the value is in the format of yyyymmdd, for example, 20000101.
`location`	`String`	Location. It is recommended that the app locally define a set of mappings between digits and location names, and the backend actually save four digits of the uint32_t type. • The first uint32_t indicates the country. • The second uint32_t indicates the province.

		• The third uint32_t indicates the city. • The fourth uint32_t indicates the county.
`language`	`Number`	Language.
`messageSettings`	`Number`	Message settings of the user. 0: receive messages. 1: do not receive messages.
`adminForbidType`	`String`	Specifies whether the admin forbids the user from friending other users. • TIM.TYPES.FORBID_TYPE_NONE: the friend can friend other users. This is the default value. • TIM.TYPES.FORBID_TYPE_SEND_OUT: the admin forbids the user from sending a friend request.
`level`	`Number`	Level. It is recommended that you divide the values into categories to save level information of multiple roles.
`role`	`Number`	Role. It is recommended that you divide the values into categories to save information of multiple roles.
`profileCustomField`	`Array<Object>`	Custom profile fields. The value is a set of key-value pair. You can use this field based on your business needs.

Returned values

This API returns a `Promise` object. The callback functions are as follows:

- The callback function parameter for `then` is IMResponse. You can obtain other users' new profiles from `IMResponse.data`.
- The callback function parameter for `catch` is IMError.

Sample

```
// Modify your personal standard profile.
let promise = tim.updateMyProfile({
nick: 'My profile',
avatar: 'http(s)://url/to/image.jpg',
gender: TIM.TYPES.GENDER_MALE,
selfSignature: 'My personal signature',
allowType: TIM.TYPES.ALLOW_TYPE_ALLOW_ANY
});
promise.then(function(imResponse) {
console.log(imResponse.data); // The profile is updated.
```

```
}).catch(function(imError) {
console.warn('updateMyProfile error:', imError); // Information on the failure in updating the pe
rsonal profile.
});

// Modify my personal custom profile.
// Custom profile fields must be configured on the IM console in advance. For more information, s
ee https://intl.cloud.tencent.com/document/product/1047/33520.
let promise = tim.updateMyProfile({
// Ensure that you have applied for the custom profile field Tag_Profile_Custom_Test1 in the IM c
onsole by clicking the desired app card and choosing **Feature Configuration** > **User Custom Fi
eld**.
// Note: even if only this one custom data field exists, the format of profileCustomField must be
of array type.
profileCustomField: [
{
key: 'Tag_Profile_Custom_Test1',
value: 'My custom profile 1'
}
]
});
promise.then(function(imResponse) {
console.log(imResponse.data); // The profile is updated.
}).catch(function(imError) {
console.warn('updateMyProfile error:', imError); // Information on the failure in updating the pe
rsonal profile.
});

// Modify my personal standard profile and custom profile.
let promise = tim.updateMyProfile({
nick: 'My profile',
// Ensure that you have applied for the custom profile fields Tag_Profile_Custom_Test1 and Tag_Pr
ofile_Custom_Test2 in the IM console by clicking the desired app card and choosing **Feature Conf
iguration** > **User Custom Field**.
profileCustomField: [
{
key: 'Tag_Profile_Custom_Test1',
value: 'My custom profile 1'
},
{
key: 'Tag_Profile_Custom_Test2',
value: 'My custom profile 2'
},
]
});
promise.then(function(imResponse) {
```

```
console.log(imResponse.data); // The profile is updated.
}).catch(function(imError) {
console.warn('updateMyProfile error:', imError); // Information on the failure in updating the pe
rsonal profile.
});
```

Blacklists

Obtaining my blacklist

API name

```
tim.getBlacklist()
```

Request Parameters

The `options` parameter is of the `Object` type. It contains the following property values:

Returned values

This API returns a `Promise` object. The callback functions are as follows:

- The callback function parameter for `then` is IMResponse. You can obtain the blacklist from `IMResponse.data` .
- The callback function parameter for `catch` is IMError.

Sample

```
let promise = tim.getBlacklist();
promise.then(function(imResponse) {
console.log(imResponse.data); // My blacklist. The value is an array that contains userIDs - [use
rID].
}).catch(function(imError) {
console.warn('getBlacklist error:', imError); // Information on the failure in obtaining the blac
klist.
});
```

Adding a user to the blacklist

This API is used to add a user to the blacklist. By adding a user to the blacklist, you can block all the messages sent by the user. Therefore, this API can be used to block the messages of a specified user.

- If user A and user B are friends, the two-way friend relationship is terminated when either A or B is blacklisted by the other user.
- If user A and user B are in a blacklist relationship, neither user A nor user B can send a friend request to the other user.
- If both user A and user B have blacklisted each other, user A and user B cannot set up a chat session.
- If user A has blacklisted user A but user B has not blacklisted user A, user A can send a message to user B, but user B cannot send a message to user A.

API name

```
tim.addToBlacklist(options)
```

Request Parameters

The `options` parameter is of the `Object` type. It contains the following property values:

Name	Type	Description
`userIDList`	`Array<String>`	All the userIDs to be added to the blacklist. The number of userIDs in a single request cannot exceed 1,000.

Returned values

This API returns a `Promise` object. The callback functions are as follows:

- The callback function parameter for `then` is IMResponse. You can obtain the blacklist from `IMResponse.data`.
- The callback function parameter for `catch` is IMError.

Sample

```
let promise = tim.addToBlacklist({userIDList: ['user1', 'user2']}); // Note: even if you add only one userID to the blacklist, the value must be of array type, for example, userIDList: ['user1'].
promise.then(function(imResponse) {
console.log(imResponse.data); // Information on the userIDs that are added to the blacklist. The value must be an array that contains userIDs - [userID].
}).catch(function(imError) {
console.warn('addToBlacklist error:', imError); // Information on the failure in adding userIDs to the blacklist.
});
```

Removing a user from the blacklist

This API is used to delete a user from the blacklist. After deleting the user from the blacklist, you can receive all the messages sent by the user.

API name

```
tim.removeFromBlacklist(options)
```

Request Parameters

The `options` parameter is of the `Object` type. It contains the following property values:

Name	Type	Description
`userIDList`	`Array<String>`	All the userIDs to be deleted from the blacklist. The number of userIDs in a single request cannot exceed 1000.

Returned values

This API returns a `Promise` object. The callback functions are as follows:

- The callback function parameter for `then` is IMResponse. You can obtain the userIDs that are deleted from the blacklist from `IMResponse.data` .
- The callback function parameter for `catch` is IMError.

Sample

```
let promise = tim.removeFromBlacklist({userIDList: ['user1', 'user2']}); // Note: even if you delete only one userID from the blacklist, the value must be of array type, for example, userIDList: ['user1'].
result.then(function(imResponse) {
console.log(imResponse.data); // All userIDs that are deleted from the blacklist. The value is an array that contains the userIDs - [userID].
}).catch(function(imError) {
console.warn('removeFromBlacklist error:', imError); // Information on the failure in deleting users from the blacklist.
});
```

Offline Push
Offline Push (Android)

Last updated : 2021-11-15 17:11:31

Overview

IM terminal users need to obtain the latest messages at any time. However, due to the limited performance and battery power of mobile devices, when the app is running in the background, IM recommends that you use the system-grade push channels provided by vendors for message notifications to avoid excessive resource consumption caused by maintaining a persistent connection. Compared with third-party push, system-grade push channels provide more stable system-grade persistent connections, enabling users to receive push messages at any time and greatly reducing resource consumption.

IM supports Mi Push, Huawei Push, Meizu Push, vivo Push, OPPO Push, and Google FCM Push. The vendor channels used by IM demos are provided and maintained by TPNS in a unified manner. You can integrate the TPNS service or the offline push service of the desired vendor for offline push feature.
Supported vendor channels are as below:

> Note :
> If you need to improve the push delivery rate or implement diversified push, we recommend that you install the SDK of TPNS to enjoy the complete push service. If you use IM and TPNS at the same time, you do not need to repeatedly integrate vendor channels.

Push Channel	System Requirements	Conditions
Mi Push	MIUI	To use Mi Push, add the dependency: implementation 'com.tencent.tpns:xiaomi:1.2.1.2-release'.
Huawei Push	EMUI	To use Huawei Push, add the dependencies: implementation 'com.tencent.tpns:huawei:1.2.1.2-release' and implementation 'com.huawei.hms:push:5.0.2.300'.

Google FCM Push	Android 4.1 and later versions	To use mobile phones installed with Google Play Services outside the Chinese mainland, add the dependency: implementation 'com.google.firebase:firebase-messaging:20.2.3'.
Meizu Push	Flyme	To use Meizu Push, add the dependency: implementation 'com.tencent.tpns:meizu:1.2.1.2-release'.
OPPO Push	ColorOS	Not all OPPO models and versions support OPPO Push. To use OPPO Push, add the dependency: implementation 'com.tencent.tpns:oppo:1.2.1.2-release'.
vivo Push	Funtouch OS	Not all vivo models and versions support vivo Push. To use vivo Push, add the dependency: implementation 'com.tencent.tpns:vivo:1.2.1.2-release'.

Here, “offline” means that the app is closed by the system or user without logging out. In such cases, if you want to receive IM SDK message reminders, you can integrate IM offline push.

Note :

- Users who have logged out or have been forced offline will not receive any message notifications.
- For Mi and Huawei vendors, if a ChannelID has been configured on the official website of the vendor developer, you need to configure the same ChannelID on the IM console. Otherwise, push may fail. If you do not configure the ChannelID, you will be limited by the frequency limit.

The process of implementing offline message push is as follows:

1. Register with the vendor, apply to enable the push service and create an app. Obtain information such as AppID, AppKey, and AppSecret.
2. Integrate the push SDK provided by the vendor with your project. Use the vendor’s console to test notification messages to ensure the SDK was integrated properly.
3. Log in to the IM console to upload the certificate and enter other required information. The IM server uses the certificate to generate a unique certificate ID.

4. Send your certificate ID and device information to IM server.

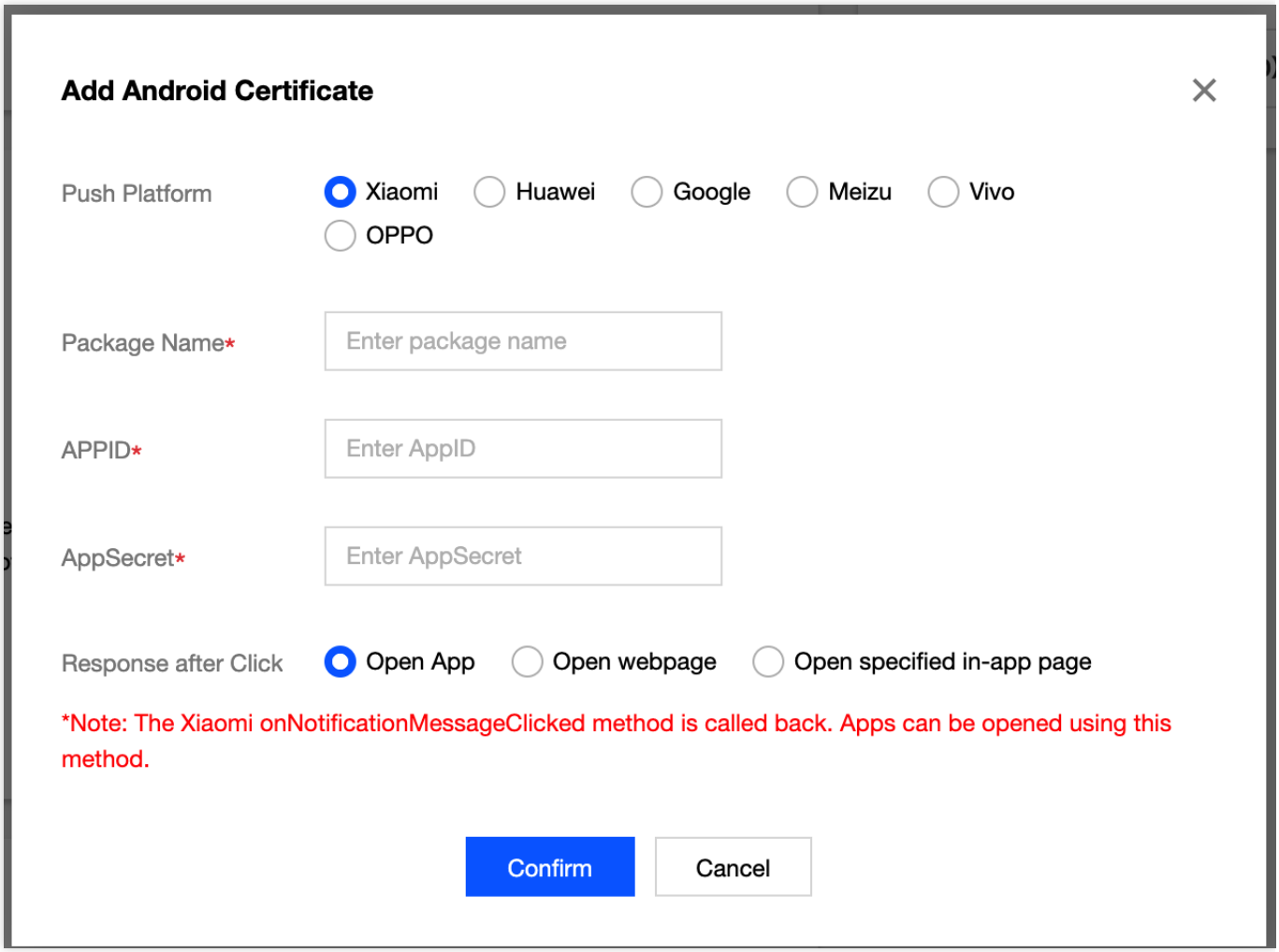
When the client app is killed by the system or user without IM logout, the IM server will push the messages sent by other accounts through vendor's channel.

Mi Push

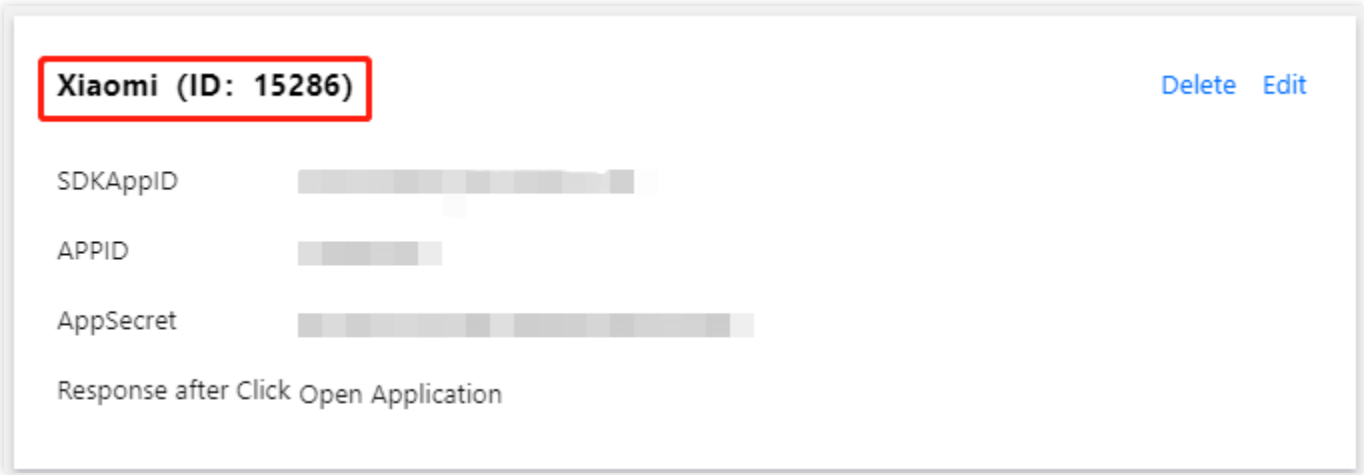
Configuring the push certificate

1. Access the Mi open platform website to register an account and pass the developer verification. Log in to the console of the Mi open platform, choose **App Service** > **Push Service**, and create a Mi Push service app. Take note of the `Primary package name`, `AppID`, and `AppSecret` information.

2. Log in to the IM console and click the target app card to go to the basic configuration page of the app. Click **Add Certificate** under **Android Platform Push Settings**. Use the information you obtained in step 1 to configure the following parameters:
   - **Push Platform**: choose **Mi**.
   - **SDKAppID**: the **primary package name** of the Mi Push service app.
   - **AppID**: enter the **AppID** you got from Mi Push.
   - **AppSecret**: enter the **AppSecret** you got from Mi Push.
   - **Response after Click**: the event to take place after the notification bar message is clicked. Valid values include **Open app**, **Open webpage**, and **Open specified in-app page**. For more information, see Configuring Click Event.
   - *Open app** or **Open specified in-app page** allows [custom content pass through(#xiaomi_custom).



Click **Confirm** to save the information. Take note of the `ID` of the certificate. Certificate information takes effect within 10 minutes after you save it.



Integrating the push SDK

1. Add the Mi dependency: implementation 'com.tencent.tpns:xiaomi:1.2.1.2-release'.
2. Refer to the Integration Guide for Mi Push, and use the Mi console to test notification messages to ensure that the SDK was integrated properly.
3. Call `MiPushClient.registerPush` to initialize the Mi Push service. After successful registration, you will receive the registration result in `onReceiveRegisterResult` of the custom `BroadcastReceiver`. `regId` is the unique identifier of the current app on the current device. After successful login to the IM SDK, you need to call setOfflinePushConfig to report the **certificate ID** and **regId** to the IM server.

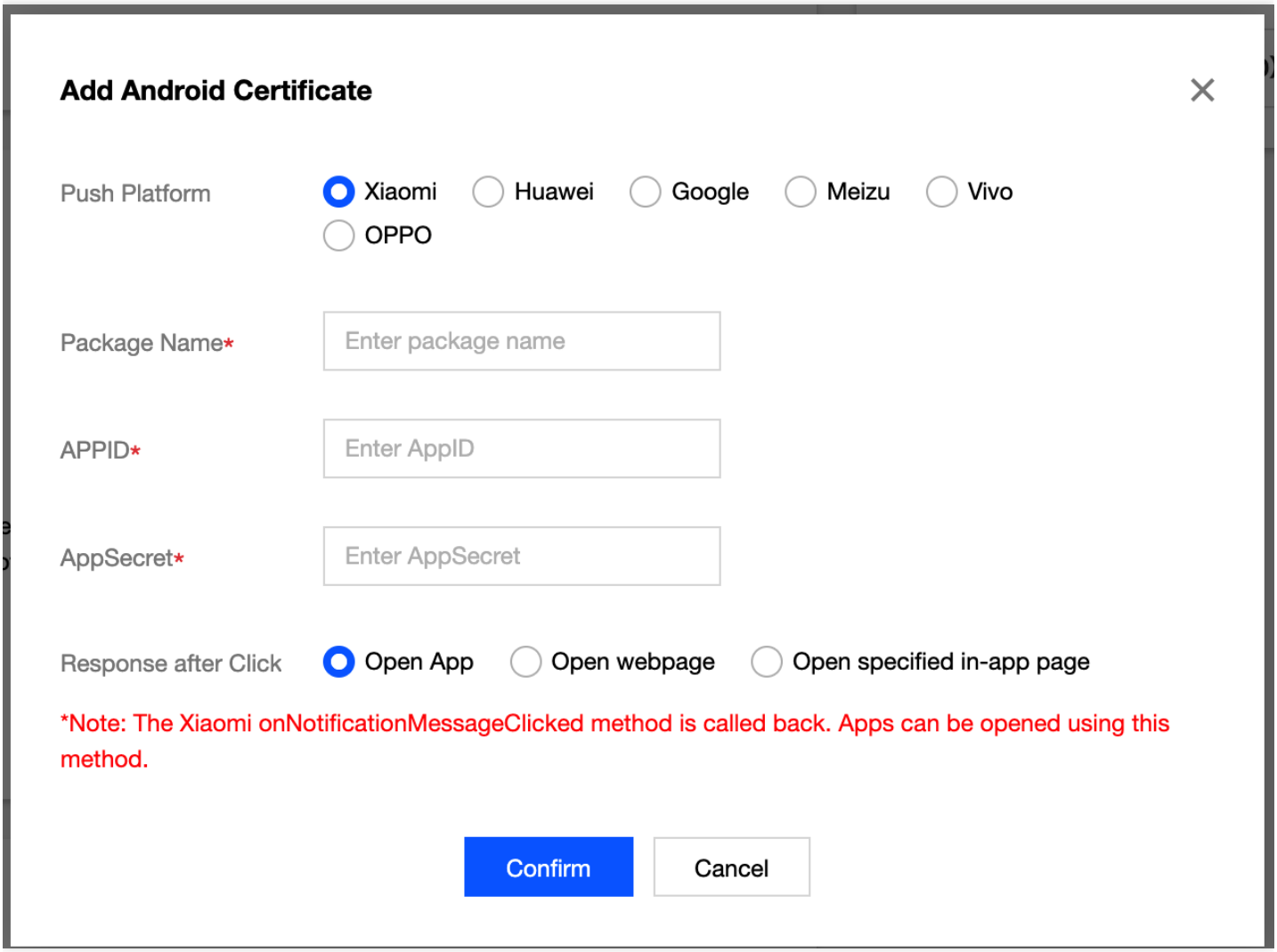
After the certificate ID and regId are successfully reported, the IM server sends messages via Mi Push notifications to the user when the app has been killed but the user has not logged out of IM.

Configuring click events

You can select one of the following events: **Open app**, **Open webpage**, or **Open specified in-app page**.

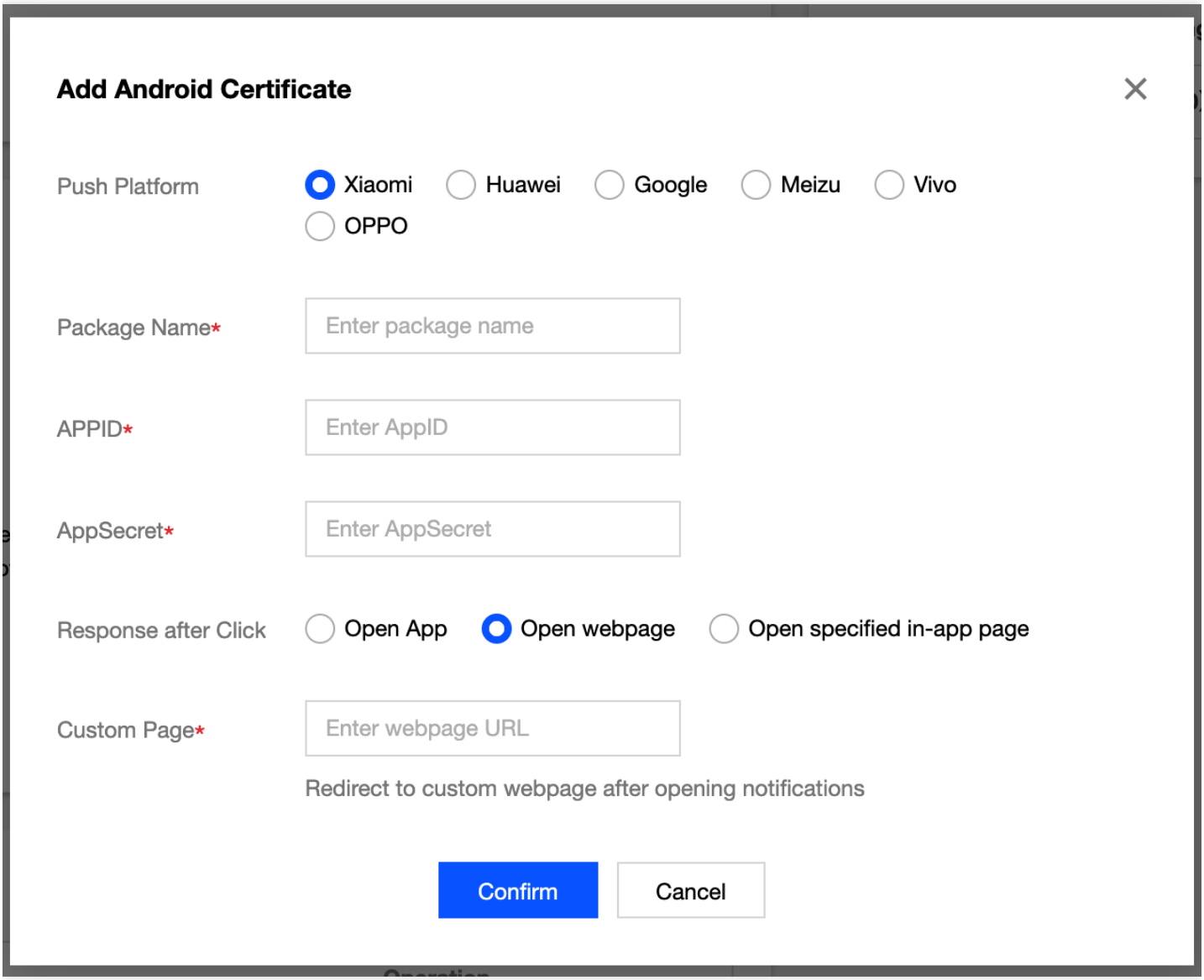
Open app

If you choose **Open app**, the `onNotificationMessageClicked` method of Mi will be called back, and the app itself can process app opening in this method.



Open webpage

You need to select **Open webpage** when adding a certificate and enter a URL that starts with either `http://` or `https://`, such as `https://cloud.tencent.com/document/product/269`.



Open specified in-app page

1. In manifest, configure the `intent-filter` of the Activity to be opened. See the sample code below. You can refer to AndroidManifest.xml of the demo:

```
<activity android:name="com.tencent.qcloud.tim.demo.main.MainActivity" android:launchmode="sin
gleTask" android:screenorientation="portrait" android:windowsoftinputmode="adjustResize|stateH
idden">
<intent-filter>
<action android:name="android.intent.action.VIEW">
<data android:host="com.tencent.qcloud" android:path="/detail" android:scheme="pushscheme">
</data></action></intent-filter>
</activity>
```

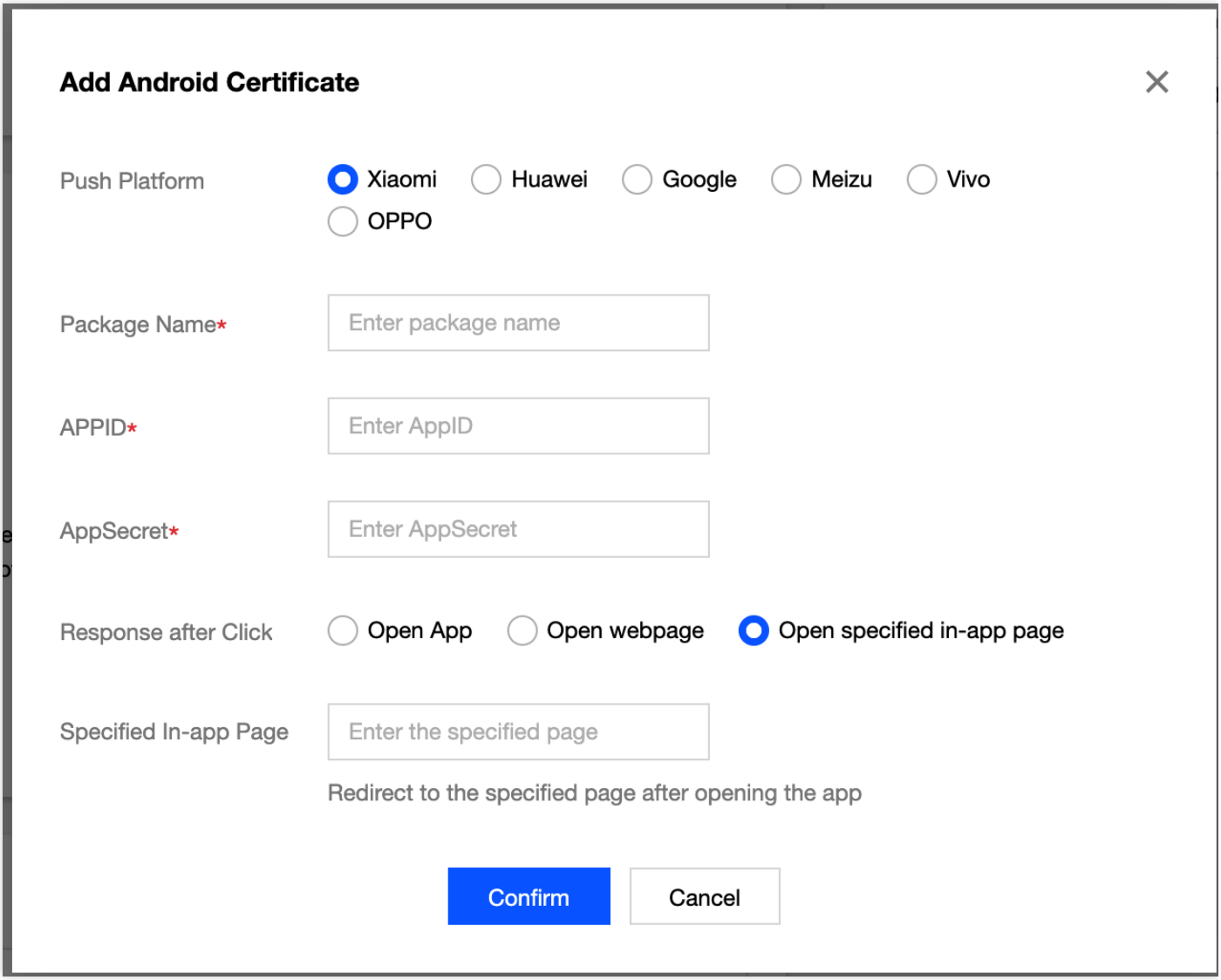
2. Obtain the intent URL, as shown below:

```
Intent intent = new Intent(this, MainActivity.class);
intent.setData(Uri.parse("pushscheme://com.tencent.qcloud.tim/detail"));
intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
String intentUri = intent.toUri(Intent.URI_INTENT_SCHEME);
Log.i(TAG, "intentUri = " + intentUri);
```

Print results:

```
intent://com.tencent.qcloud.tim/detail#Intent;scheme=pushscheme;launchFlags=0x4000000;componen
t=com.tencent.qcloud.tim.tuikit/com.tencent.qcloud.tim.demo.main.MainActivity;end
```

3. Select **Open specified in-app page** when adding a certificate and enter the result above.



Custom content pass through

Select **Open app** or **Open specified in-app page** in **Response after Click** when adding a certificate to support custom content pass through.

Step 1. Set custom content (sender)

Set the custom content for the notification bar message before sending the message.

- Below is a simple example on the Android platform. You can also refer to the corresponding logic in the `sendMessage()` method in the ChatProvider.java class in the TUIKit:

```
OfflineMessageContainerBean containerBean = new OfflineMessageContainerBean();
OfflineMessageBean entity = new OfflineMessageBean();
entity.content = message.getExtra().toString();
entity.sender = message.getFromUser();
entity.nickname = chatInfo.getChatName();
entity.faceUrl = TUIChatConfigs.getConfigs().getGeneralConfig().getUserFaceUrl();
containerBean.entity = entity;
V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
v2TIMOfflinePushInfo.setExt(new Gson().toJson(containerBean).getBytes());
// For OPPO, you must set the `ChannelID` to receive push messages. The `ChannelID` must be co
nsistent with that in the console.
v2TIMOfflinePushInfo.setAndroidOPPOChannelID("tuikit");
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, userID, null,
V2TIMMessage.V2TIM_PRIORITY_DEFAULT, false, v2TIMOfflinePushInfo, new V2TIMSendCallback<v2timm
essage>() {
@Override
public void onError(int code, String desc) {}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {}
@Override
public void onProgress(int progress) {}
});
```

- For information on configurations for the IM server, refer to the OfflinePushInfo Format Example.

Step 2. Set custom content (receiver)

- If you selected **Open app** in **Response after Click** when adding a certificate, clicking the notification bar message triggers the `onNotificationMessageClicked(Context context, MiPushMessage miPushMessage)` callback. The custom content can be obtained from `miPushMessage`. You can refer to the parsing implementation in XiaomiMsgReceiver.java.

```
Map extra = miPushMessage.getExtra();
String extContent = extra.get("ext");
```

- If you selected **Open specified in-app page** in **Response after Click** when adding a certificate, `MiPushMessage`, which is the object that encapsulates the message, is passed to the client through `Intent`. The client then obtains the custom content from `Activity`. You can refer to the implementation of the `parseOfflineMessage(Intent intent)` method in the OfflineMessageDispatcher.java class.

```
Bundle bundle = getIntent().getExtras();
MiPushMessage miPushMessage = (MiPushMessage)bundle.getSerializable(PushMessageHelper.KEY_MESS
AGE);
Map extra = miPushMessage.getExtra();
String extContent = extra.get("ext");
```

Huawei Push

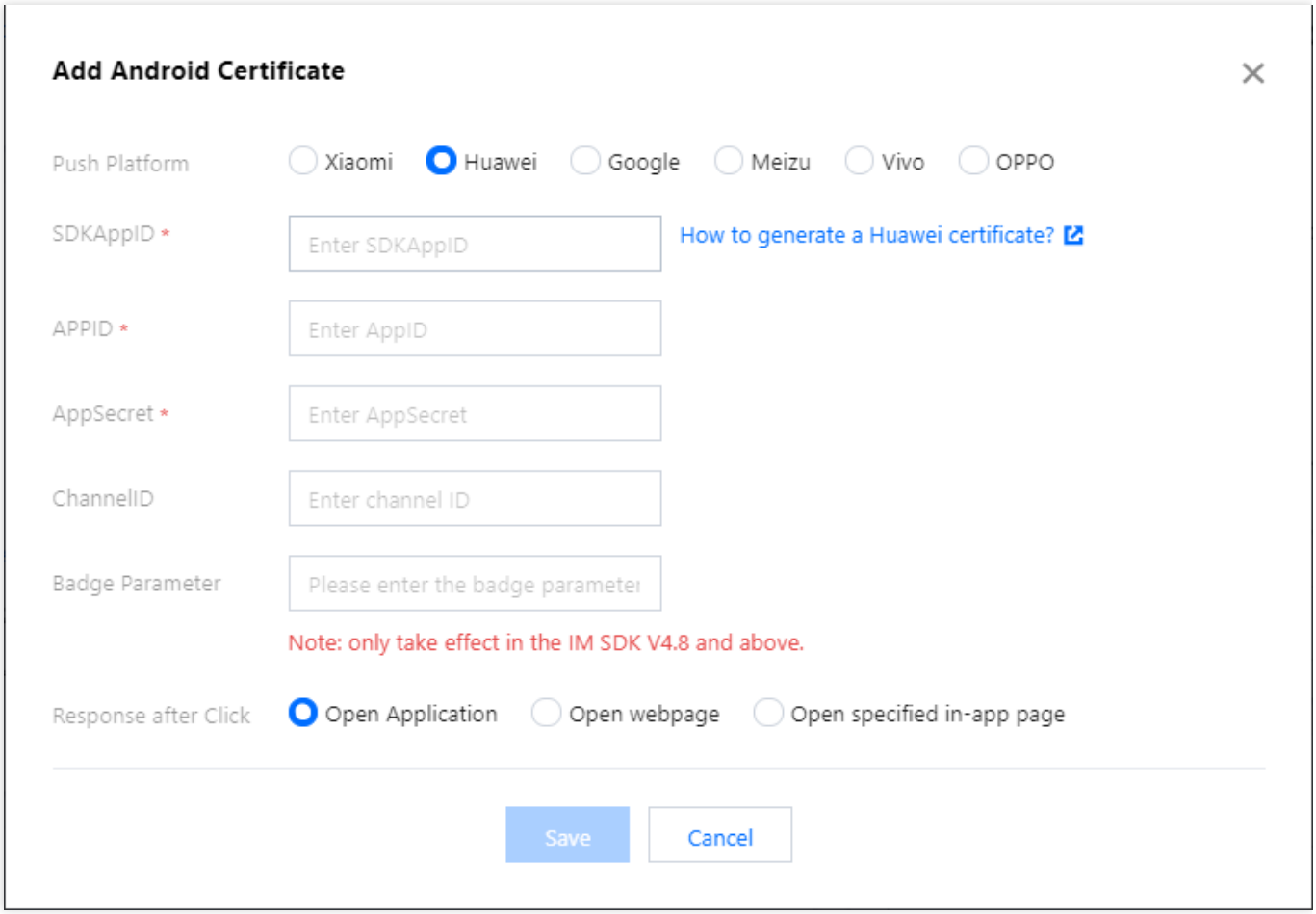
Configuring the push certificate

1. Access the official website of the Huawei Developers Alliance, register an account, and pass the developer verification. Log in to the console of the Huawei Developers Alliance, choose **App Service** > **Development Service** > **PUSH**, and create a Huawei push service app. Take note of the `Package name`, `APP ID`, and `APP SECRET`.

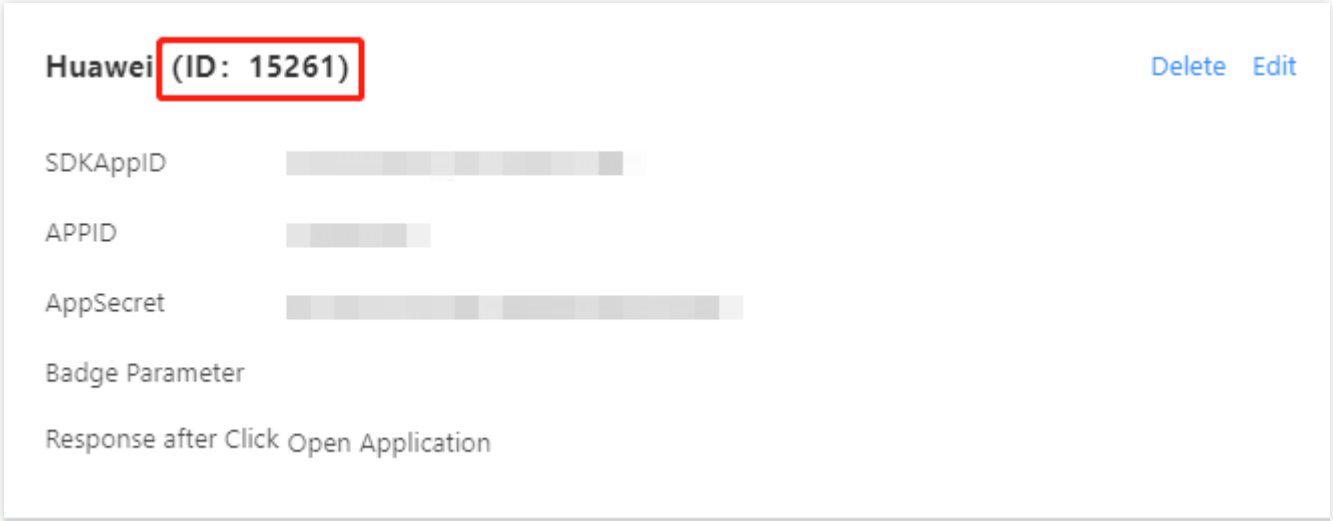
2. Log in to the IM console and click the target app card to go to the basic configuration page of the app. Click **Add Certificate** under **Android Platform Push Settings**. Use the information you obtained in step 1 to configure the following parameters:
   - **Push Platform**: select **Huawei**.
   - **SDKAppID**: the **package name** of the Huawei Push service app.
   - **AppID**: enter the **App ID** you got from Huawei Push.
   - **AppSecret**: enter the **APP SECRET** you got from Huawei Push.
   - **Badge Parameters**: enter the full `Activity` class name of the app entry, which will be used as the Huawei desktop app badge for display. For more information, see the description of desktop app badge in the Huawei Push service development document.
   - **Response after Click**: the event to take place after the notification bar message is clicked. Valid values include **Open app**, **Open webpage**, and **Open specified in-app page**. For more

information, refer to Configuring Click Event.

- *Open app** or **Open specified in-app page** allows [custom content pass through(#huawei_custom).



Click **Save** to save the information. Take note of the `ID` of the certificate. Certificate information takes effect within 10 minutes after you save it.



Integrating the push SDK

1. Add the Huawei dependencies: implementation 'com.tencent.tpns:huawei:1.2.1.2-release' and implementation 'com.huawei.hms:push:5.0.2.300'.
2. Refer to the Integration Guide for Huawei Push and use the Huawei console to test notification messages to ensure that the SDK was integrated properly.
3. Call the Huawei `HmsInstanceId.getToken` API to request the unique app identifier Push Token from the server. `Push Token` is the unique identifier of the current app on the current device. After successful login to the IM SDK, you need to call setOfflinePushConfig to report the **certificate ID** and **Push Token** to the IM server.

After the certificate ID and regId are successfully reported, the IM server sends messages via Huawei Push notifications to the user when the app has been killed but the user has not logged out of IM.

Configuring click events

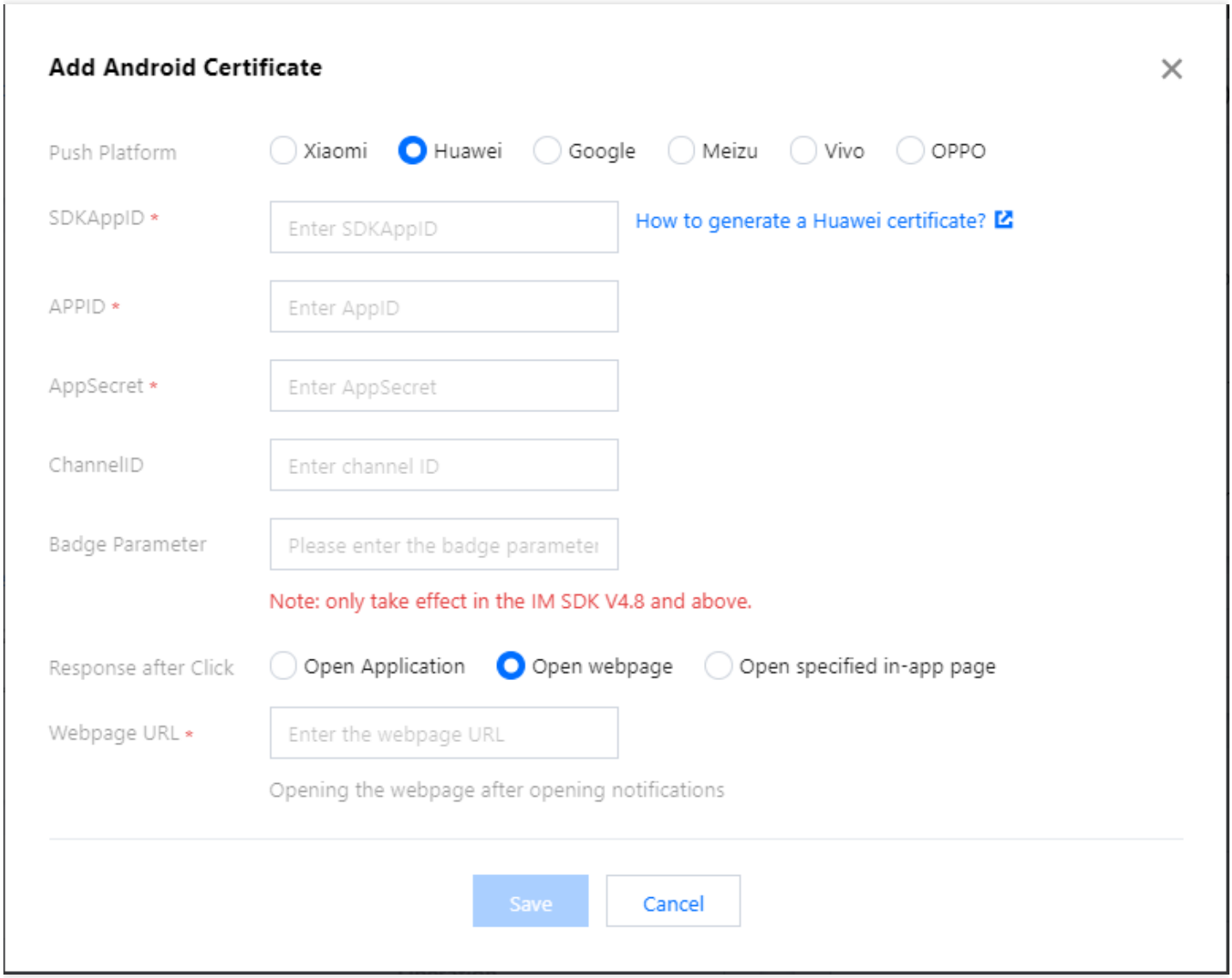
You can select one of the following events: **Open app**, **Open webpage**, or **Open specified in-app page**.

Open app

This is the default event, which opens the app once the notification bar message is clicked.

Open webpage

You need to select **Open webpage** when adding a certificate and enter a URL that starts with either `http://` or `https://`, such as `https://cloud.tencent.com/document/product/269`.



Open specified in-app page

1. In manifest, configure the `intent-filter` of the Activity to be opened. See the sample code below. You can refer to AndroidManifest.xml of the demo:

```
<activity android:name="com.tencent.qcloud.tim.demo.main.MainActivity" android:launchmode="sin
gleTask" android:screenorientation="portrait" android:windowsoftinputmode="adjustResize|stateH
idden">
<intent-filter>
<action android:name="android.intent.action.VIEW">
<data android:host="com.tencent.qcloud" android:path="/detail" android:scheme="pushscheme">
</data></action></intent-filter>
</activity>
```

2. Obtain the intent URL, as shown below:

```
Intent intent = new Intent(this, MainActivity.class);
intent.setData(Uri.parse("pushscheme://com.tencent.qcloud.tim/detail"));
intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
String intentUri = intent.toUri(Intent.URI_INTENT_SCHEME);
Log.i(TAG, "intentUri = " + intentUri);
```

Print results:

```
intent://com.tencent.qcloud.tim/detail#Intent;scheme=pushscheme;launchFlags=0x4000000;componen
t=com.tencent.qcloud.tim.tuikit/com.tencent.qcloud.tim.demo.main.MainActivity;end
```

3. Select **Open specified in-app page** when adding a certificate and enter the result above.

Custom content pass through

> Note :
> Due to the compatibility issues of Huawei Push, the pass-through content can only be received on some EUI10+ devices.

Step 1. Set custom content (sender)

Set the custom content for the notification bar message before sending the message.

- Below is a simple example on the Android platform. You can also refer to the corresponding logic in the `sendMessage()` method in the ChatProvider.java class in the TUIKit:

```
OfflineMessageContainerBean containerBean = new OfflineMessageContainerBean();
OfflineMessageBean entity = new OfflineMessageBean();
entity.content = message.getExtra().toString();
entity.sender = message.getFromUser();
entity.nickname = chatInfo.getChatName();
entity.faceUrl = TUIChatConfigs.getConfigs().getGeneralConfig().getUserFaceUrl();
containerBean.entity = entity;
V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
v2TIMOfflinePushInfo.setExt(new Gson().toJson(containerBean).getBytes());
// For OPPO, you must set the `ChannelID` to receive push messages. The `ChannelID` must be co
nsistent with that in the console.
v2TIMOfflinePushInfo.setAndroidOPPOChannelID("tuikit");
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, userID, null,
V2TIMMessage.V2TIM_PRIORITY_DEFAULT, false, v2TIMOfflinePushInfo, new V2TIMSendCallback<v2timm
essage>() {
@Override
public void onError(int code, String desc) {}
```

```
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {}
@Override
public void onProgress(int progress) {}
});
```

- For information on configurations for the IM server, refer to the OfflinePushInfo Format Example.

Step 2. Set custom content (receiver)

- If you selected **Open app** or **Open specified in-app page** in **Response after Click** when adding a certificate, the client can obtain the custom content from `Activity` when the notification bar message is clicked. You can refer to the parseOfflineMessage(Intent intent) implementation method in the OfflineMessageDispatcher.java class.

```
Bundle bundle = getIntent().getExtras();
String value = bundle.getString("ext");
```

OPPO Push

Configuring the push certificate

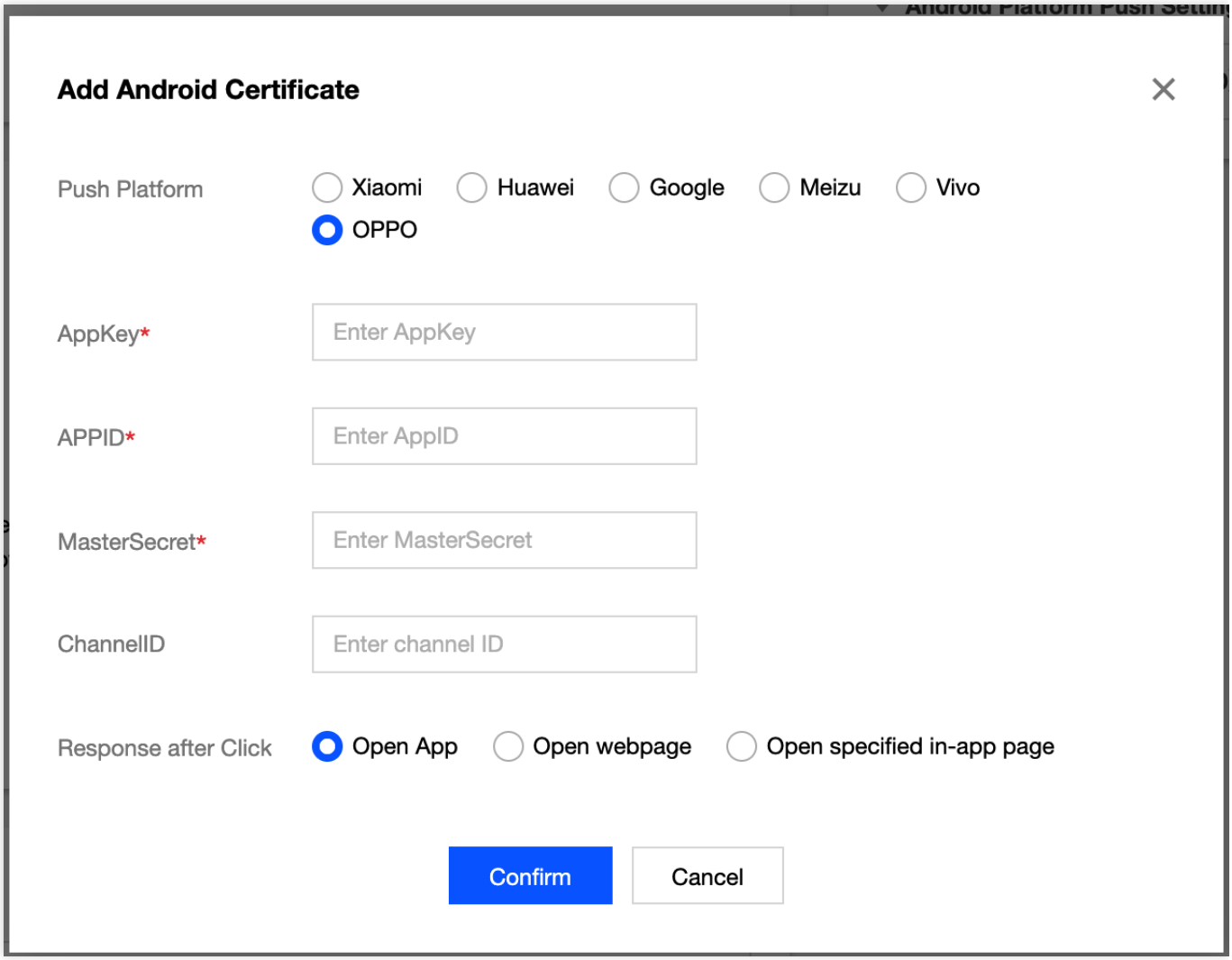
1. Refer to How to enable OPPO Push for instructions on how to enable OPPO Push. Go to OPPO push platform > **Configuration Management** > **App Configuration** to view detailed app information. Take note of `AppId` , `AppKey` , `AppSecret` , and `MasterSecret` .

2. The official OPPO documentation states that ChannelIDs are required for push messages on OPPO Android 8.0 and above. Therefore, create a ChannelID for your app. Below is a sample code that creates a ChannelID called `tuikit` :

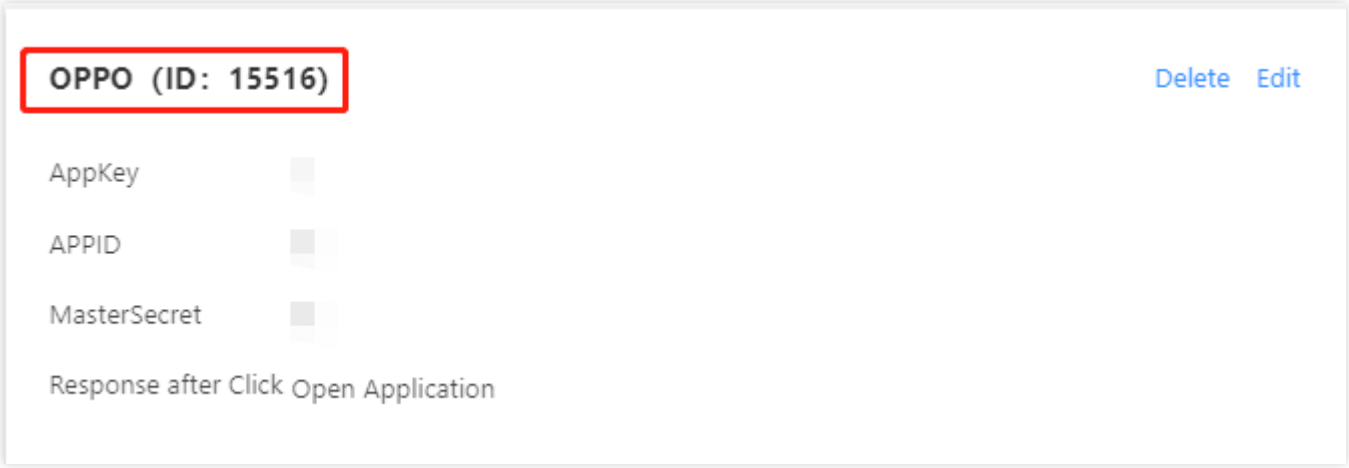
```
public void createNotificationChannel(Context context) {
// Create the NotificationChannel, but only on API 26+ because
// the NotificationChannel class is new and not in the support library
if (Build.VERSION.SDK_INT &gt;= Build.VERSION_CODES.O) {
CharSequence name = "oppotest";
String description = "this is opptest";
int importance = NotificationManager.IMPORTANCE_DEFAULT;
NotificationChannel channel = new NotificationChannel("tuikit", name, importance);
channel.setDescription(description);
// Register the channel with the system; you can't change the importance
// or other notification behaviors after this
```

```
NotificationManager notificationManager = context.getSystemService(NotificationManager.class);
notificationManager.createNotificationChannel(channel);
}
}
```

3. Log in to the IM console and click the target app card to go to the basic configuration page of the app. Click **Add Certificate** under **Android Platform Push Settings**. Use the information you obtained in step 1 to configure the following parameters:
   - **Push Platform**: select **OPPO**.
   - **AppKey**: enter the **AppKey** you got from OPPO PUSH.
   - **AppID**: enter the **AppID** you got from OPPO PUSH.
   - **MasterSecret**: enter the **MasterSecret** you got from OPPO PUSH.
   - **ChannelID**: enter the **ChannelID** created in Step 2.
   - **Response after Click**: the event to take place after the notification bar message is clicked. Valid values include **Open app**, **Open webpage**, and **Open specified in-app page**. For more information, refer to Configuring Click Event.
   - *Open app** or **Open specified in-app page** allows [custom content pass through(#oppo_custom).



Click **Confirm** to save the information. Take note of the `ID` of the certificate. Certificate information takes effect within 10 minutes after you save it.



Integrating the push SDK

1. Add the OPPO dependency: implementation 'com.tencent.tpns:oppo:1.2.1.2-release'.
2. Refer to the OPPO PUSH SDK API Documentation and use the OPPO console to test notification messages to ensure that the SDK was integrated properly.
3. Call `HeytapPushManager.register(…)` in the OPPO SDK to initialize the Opush service. After successful registration, you can obtain `regId` in the `onRegister` callback method of `ICallBackResultService`. `regId` is the unique identifier of the current app on the current device. After successful login to the IM SDK, you need to call setOfflinePushConfig to report the **certificate ID** and **regId** to the IM server.

After the certificate ID and regId are successfully sent, the IM server will push the notification to the client through OPPO PUSH when the app is killed by the system before the user logs out.

Configuring click events

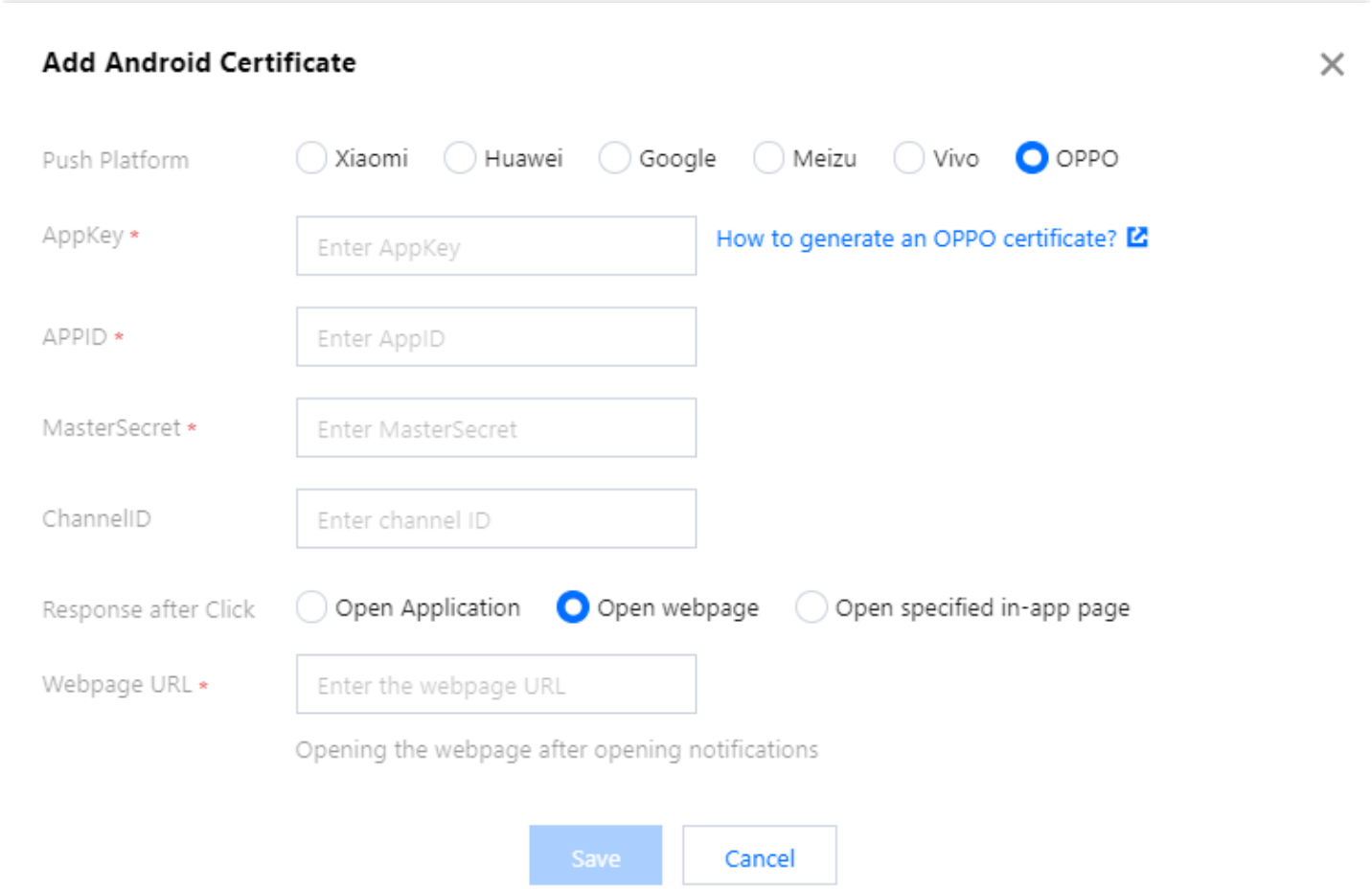
You can select one of the following events: **Open app**, **Open webpage**, or **Open specified in-app page**.

Open app

This is the default event, which opens the app once the notification bar message is clicked.

Open webpage

You need to select **Open webpage** when adding a certificate and enter a URL that starts with either `http` or `https`, such as `https://cloud.tencent.com/document/product/269`.



Open specified in-app page

These are the ways you can open a specific in-app interface:

Activity (recommended)
This is rather simple. Enter the whole name of an Activity, such as `com.tencent.qcloud.tim.demo.main.MainActivity` .

Intent action

1. In AndroidManifest, set the following configuration in the Activity to be opened and add category without data. You can refer to AndroidManifest.xml of the demo:

```
<intent-filter>
<action android:name="android.intent.action.VIEW">
<category android:name="android.intent.category.DEFAULT">
</category></action></intent-filter>
```

2. Enter `android.intent.action.VIEW` in the console.

Custom content pass through

Select **Open app** or **Open specified in-app page** in **Response after Click** when adding a certificate to support custom content pass through.

Step 1. Set custom content (sender)

Set the custom content for the notification bar message before sending the message.

- Below is a simple example on the Android platform. You can also refer to the corresponding logic in the `sendMessage()` method in the ChatProvider.java class in the TUIKit:

```
OfflineMessageContainerBean containerBean = new OfflineMessageContainerBean();
OfflineMessageBean entity = new OfflineMessageBean();
entity.content = message.getExtra().toString();
entity.sender = message.getFromUser();
entity.nickname = chatInfo.getChatName();
entity.faceUrl = TUIChatConfigs.getConfigs().getGeneralConfig().getUserFaceUrl();
containerBean.entity = entity;
V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
v2TIMOfflinePushInfo.setExt(new Gson().toJson(containerBean).getBytes());
// For OPPO, you must set the `ChannelID` to receive push messages. The `ChannelID` must be co
nsistent with that in the console.
v2TIMOfflinePushInfo.setAndroidOPPOChannelID("tuikit");
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, userID, null,
V2TIMMessage.V2TIM_PRIORITY_DEFAULT, false, v2TIMOfflinePushInfo, new V2TIMSendCallback<v2timm
essage>() {
@Override
public void onError(int code, String desc) {}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {}
@Override
public void onProgress(int progress) {}
});
```

- For information on configurations for the IM server, refer to the OfflinePushInfo Format Example.

Step 2. Set custom content (receiver)

When the notification bar message is clicked, the client can obtain the custom content from the launched `Activity`. You can refer to the `parseOfflineMessage(Intent intent)` implementation method in the OfflineMessageDispatcher.java class.

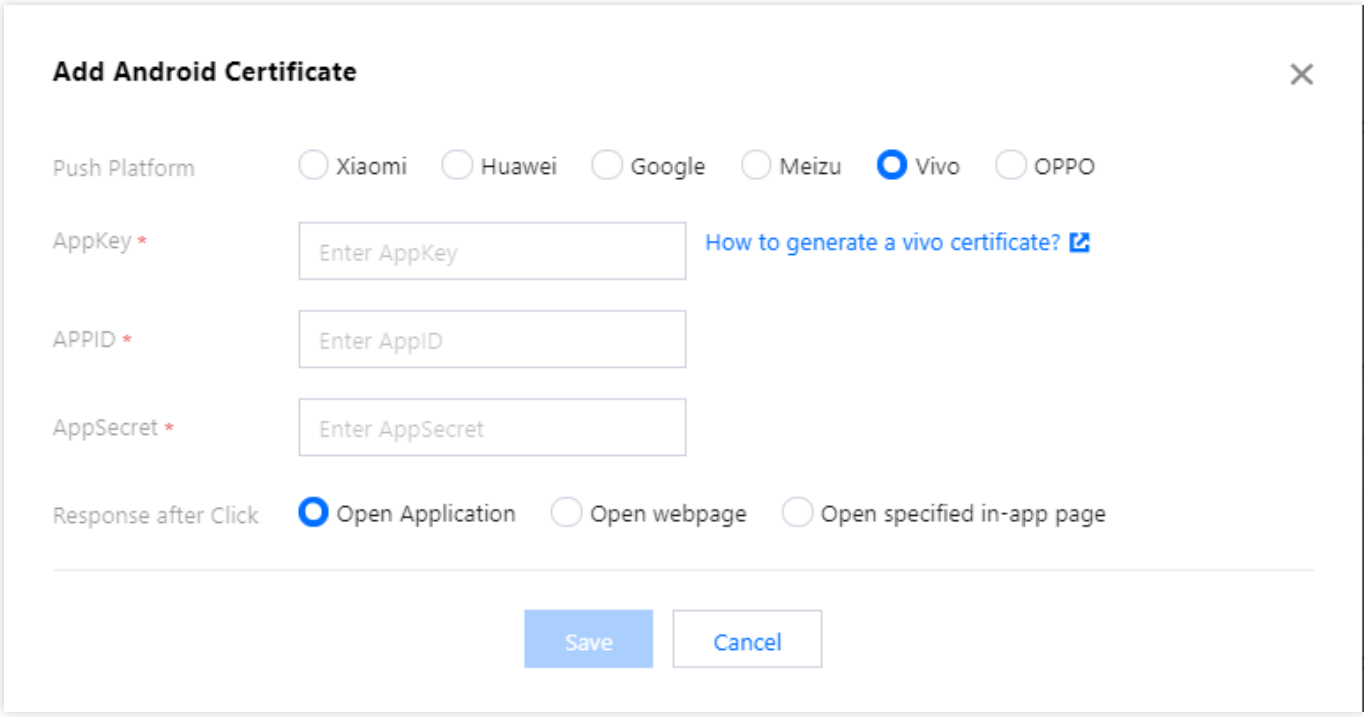
```
Bundle bundle = intent.getExtras();
Set<string> set = bundle.keySet();
if (set != null) {
for (String key : set) {
// `key` and `value` correspond to `extKey` and `ext content` set at the sender
String value = bundle.getString(key);
Log.i("oppo push custom data", "key = " + key + ":value = " + value);
}
}
```

vivo Push

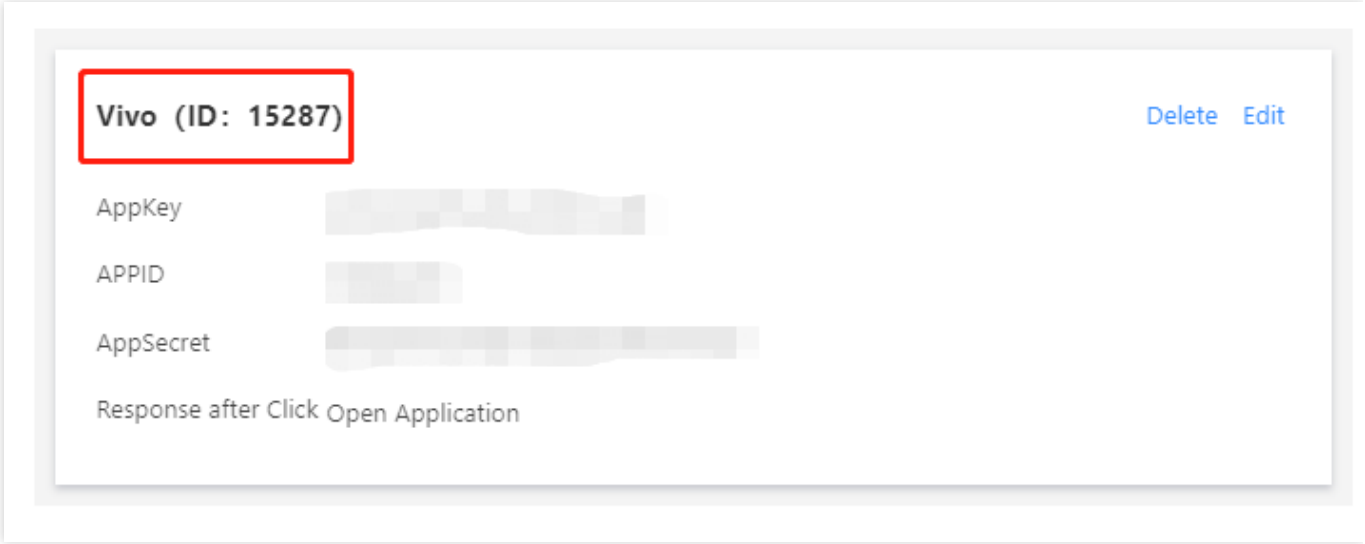
Configuring the push certificate

1. Visit the vivo open platform official website and register for an account. Complete developer verification. Log in to the console of the vivo open platform, choose **Message Push** > **Create** > **Test Push**, and create a vivo push service app. Take note of **APP ID**, **APP key**, and **APP secret**.

2. Log in to the IM console and click the target app card to go to the basic configuration page of the app. Click **Add Certificate** under **Android Platform Push Settings**. Use the information you obtained in step 1 to configure the following parameters:
   - **Push Platform**: select **vivo**.
   - **AppKey**: enter the **AppKey** you got from vivo Push.
   - **AppID**: enter the **AppID** you got from vivo Push.
   - **AppSecret**: enter the **APP secret** you got from vivo Push.
   - **Response after Click**: the event to take place after the notification bar message is clicked. Valid values include **Open app**, **Open webpage**, and **Open specified in-app page**. For more information, refer to Configuring Click Event.
   - *Open app** or **Open specified in-app page** allows [custom content pass through(#vivo_custom).



Click **Confirm** to save the information. Take note of the **ID** of the certificate. Certificate information takes effect 10 minutes after you save it.



Integrating the push SDK

1. Add vivo dependency: implementation 'com.tencent.tpns:vivo:1.2.1.2-release'.
2. Refer to the Integration Guide for vivo Push, and use the vivo console to test notification messages to ensure that the SDK was integrated properly.
3. Call `PushClient.getInstance(getApplicationContext()).initialize()` to initialize the vivo Push service and call `PushClient.getInstance(getApplicationContext()).turnOnPush()` to launch push. If

this succeeds, you will receive the `regId` in the `onReceiveRegId` of the custom `BroadcastReceiver`. `regId` is the unique identifier of the current app on the current device. After successful login to the IM SDK, you need to call setOfflinePushConfig to report the **certificate ID** and **regId** to the IM server.

After the certificate ID and regId are successfully reported, the IM server sends messages via vivo Push notifications to the user when the app has been killed but the user has not logged out of IM.

Configuring click events

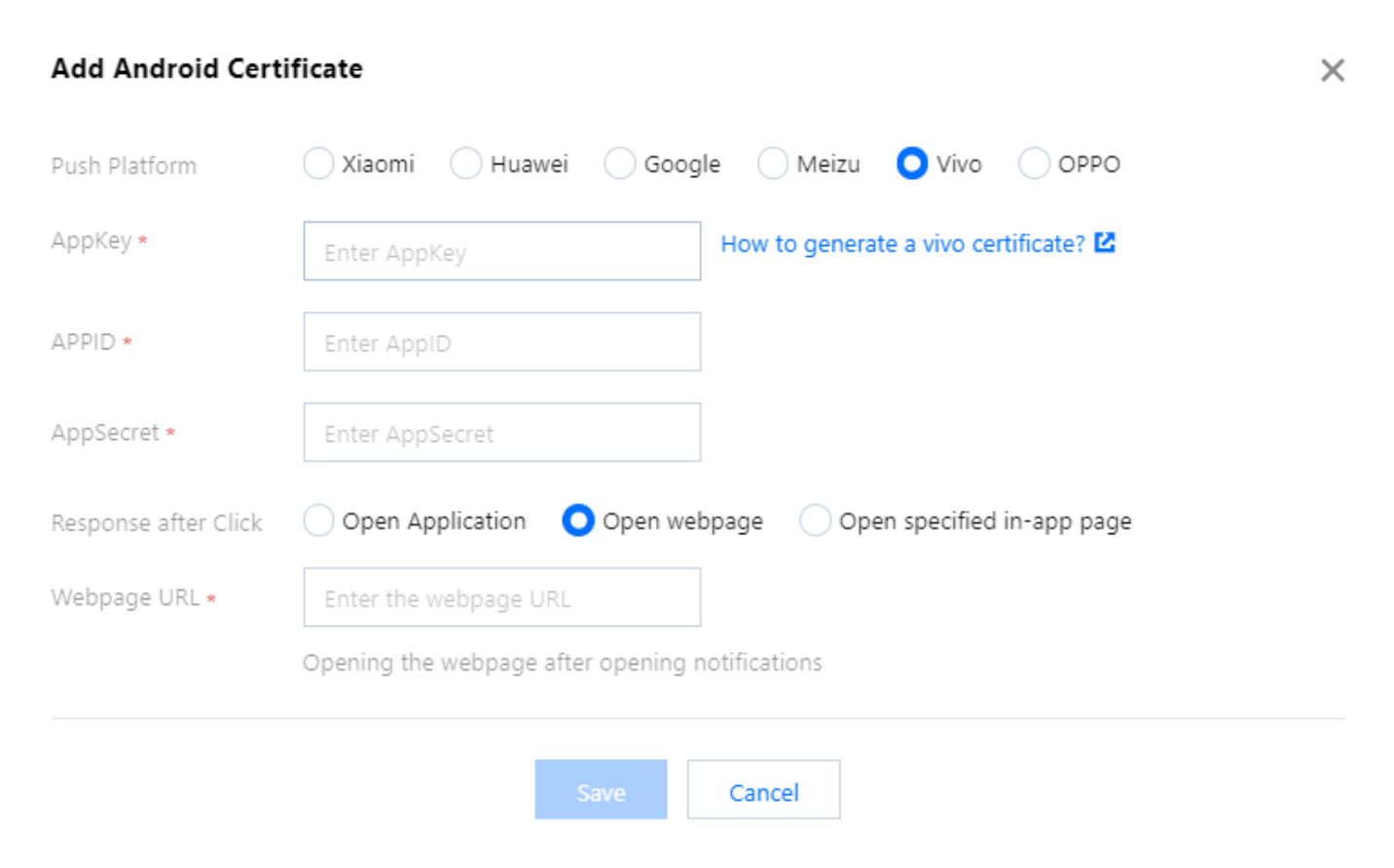
You can select one of the following events: **Open app**, **Open webpage**, or **Open specified in-app page**.

Open app

This is the default event, which opens the app once the notification bar message is clicked.

Open webpage

You need to select **Open webpage** when adding a certificate and enter a URL that starts with either `http://` or `https://`, such as `https://cloud.tencent.com/document/product/269`.



Open specified in-app page

1. In manifest, configure the `intent-filter` of the Activity to be opened. See the sample code below. You can refer to AndroidManifest.xml of the demo:

```
<activity android:name="com.tencent.qcloud.tim.demo.main.MainActivity" android:launchmode="sin
gleTask" android:screenorientation="portrait" android:windowsoftinputmode="adjustResize|stateH
idden">
<intent-filter>
<action android:name="android.intent.action.VIEW">
<data android:host="com.tencent.qcloud" android:path="/detail" android:scheme="pushscheme">
</data></action></intent-filter>
</activity>
```

2. Obtain the intent URL, as shown below:

```
Intent intent = new Intent(this, MainActivity.class);
intent.setData(Uri.parse("pushscheme://com.tencent.qcloud.tim/detail"));
intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
String intentUri = intent.toUri(Intent.URI_INTENT_SCHEME);
Log.i(TAG, "intentUri = " + intentUri);
```

Print results:

```
intent://com.tencent.qcloud.tim/detail#Intent;scheme=pushscheme;launchFlags=0x4000000;componen
t=com.tencent.qcloud.tim.tuikit/com.tencent.qcloud.tim.demo.main.MainActivity;end
```

3. Select **Open specified in-app page** when adding a certificate and enter the result above.

Custom content pass through

Select **Open app** or **Open specified in-app page** in **Response after Click** when adding a certificate to support custom content pass through.

Step 1. Set custom content (sender)

Set the custom content for the notification bar message before sending the message.

- Below is a simple example on the Android platform. You can also refer to the corresponding logic in the `sendMessage()` method in the ChatProvider.java class in the TUIKit:

```
OfflineMessageContainerBean containerBean = new OfflineMessageContainerBean();
OfflineMessageBean entity = new OfflineMessageBean();
entity.content = message.getExtra().toString();
entity.sender = message.getFromUser();
```

```
entity.nickname = chatInfo.getChatName();
entity.faceUrl = TUIChatConfigs.getConfigs().getGeneralConfig().getUserFaceUrl();
containerBean.entity = entity;
V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
v2TIMOfflinePushInfo.setExt(new Gson().toJson(containerBean).getBytes());
// For OPPO, you must set the `ChannelID` to receive push messages. The `ChannelID` must be co
nsistent with that in the console.
v2TIMOfflinePushInfo.setAndroidOPPOChannelID("tuikit");
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, userID, null,
V2TIMMessage.V2TIM_PRIORITY_DEFAULT, false, v2TIMOfflinePushInfo, new V2TIMSendCallback<v2timm
essage>() {
@Override
public void onError(int code, String desc) {}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {}
@Override
public void onProgress(int progress) {}
});
```

- For information on configurations for the IM server, refer to the OfflinePushInfo Format Example.

Step 2. Set custom content (receiver)

When the notification bar message is clicked, the `onNotificationMessageClicked(Context context, UPSNotificationMessage upsNotificationMessage)` callback of the vivo Push SDK is triggered. The custom content can be obtained from `upsNotificationMessage`. You can refer to the parsing implementation in VIVOPushMessageReceiverImpl.java.

```
Map<string, string=""> paramMap = upsNotificationMessage.getParams();
String extContent = paramMap.get("ext");
```

Meizu Push

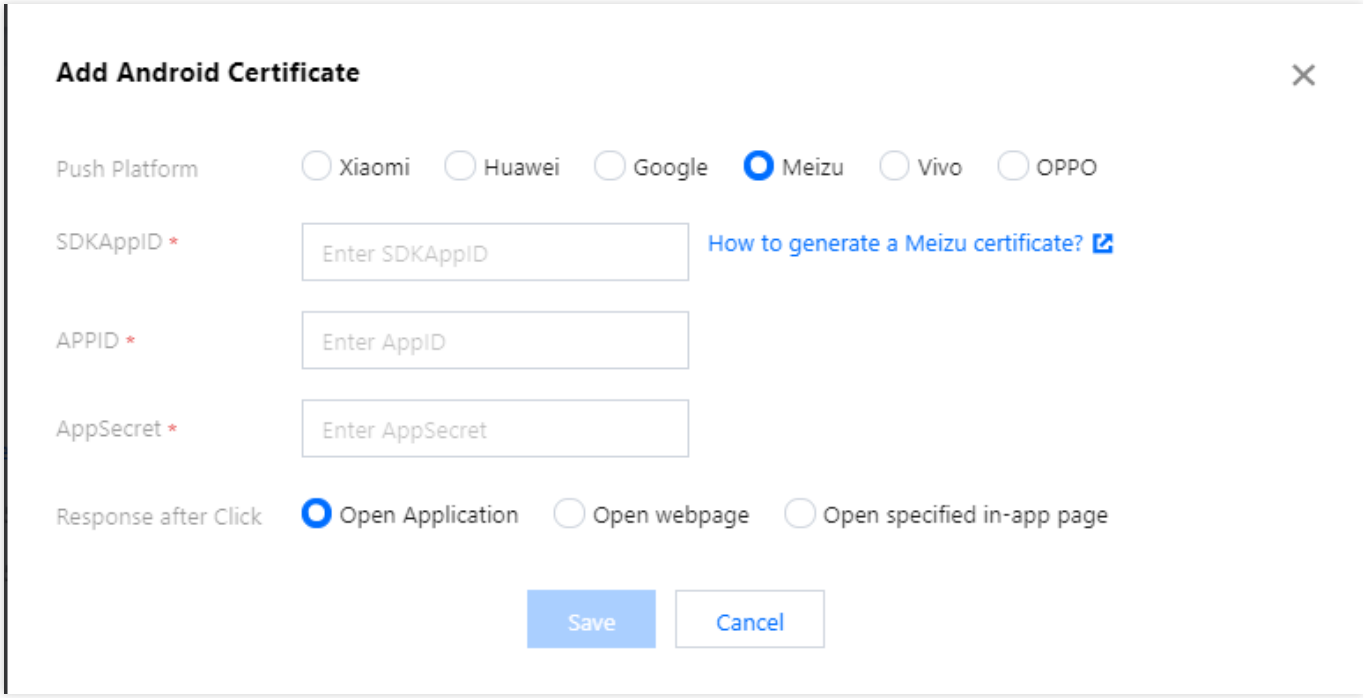
Configuring the push certificate

1. Access the Meizu open platform website to register an account and pass the developer verification. Log in to the Meizu console, choose **Development Service** > **Flyme Push** and create a Meizu push service app. Take note of the `app package name`, `App ID`, and `App Secret`.

2. Log in to the IM console and click the target app card to go to the basic configuration page of the app. Click **Add Certificate** under **Android Platform Push Settings**. Use the information you

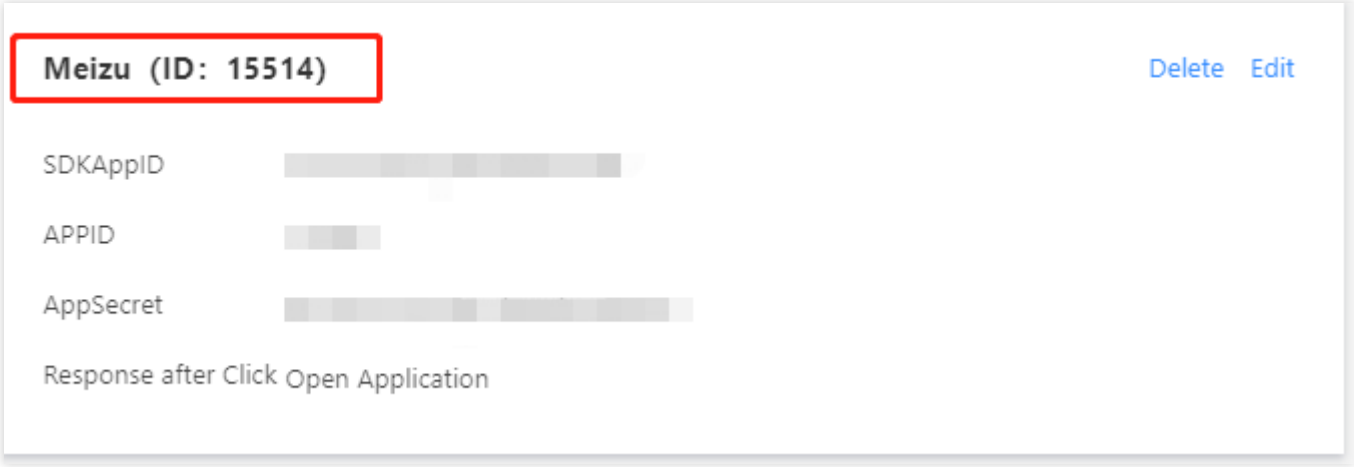
obtained in step 1 to configure the following parameters:

- **Push Platform**: choose **Meizu**.
- **SDKAppID**: enter the **app package name** of the Meizu push service app.
- **AppID**: enter the **App ID** of the Meizu push service app.
- **AppSecret**: enter the **App Secret** of the Meizu push service app.
- **Response after Click**: the event to take place after the notification bar message is clicked. Valid values include **Open app**, **Open webpage**, and **Open specified in-app page**. For more information, refer to Configuring Click Event.
- *Open app** or **Open specified in-app page** allows [custom content pass through(#meizu_custom).



Click **Confirm** to save the information. Take note of the `ID` of the certificate. Certificate

information takes effect within 10 minutes after you save it.



Integrating the push SDK

1. Add Meizu dependency: implementation 'com.tencent.tpns:meizu:1.2.1.2-release'.
2. Refer to Meizu Push Integration, and use the Meizu console to test notification messages to ensure that the SDK was integrated properly.
3. Call `PushManager.register` to initialize the Meizu Push service. After successful registration, you will receive the registration result in `onRegisterStatus` of the custom `BroadcastReceiver`. `registerStatus.getPushId()` is the unique identifier of the current app on the current device. After successful login to the IM SDK, you need to call setOfflinePushConfig to report the **certificate ID** and **PushId** to the IM server.

After the certificate ID and regId are successfully reported, the IM server sends messages via Meizu Push notifications to the user when the app has been killed but the user has not logged out of IM.

Configuring click events

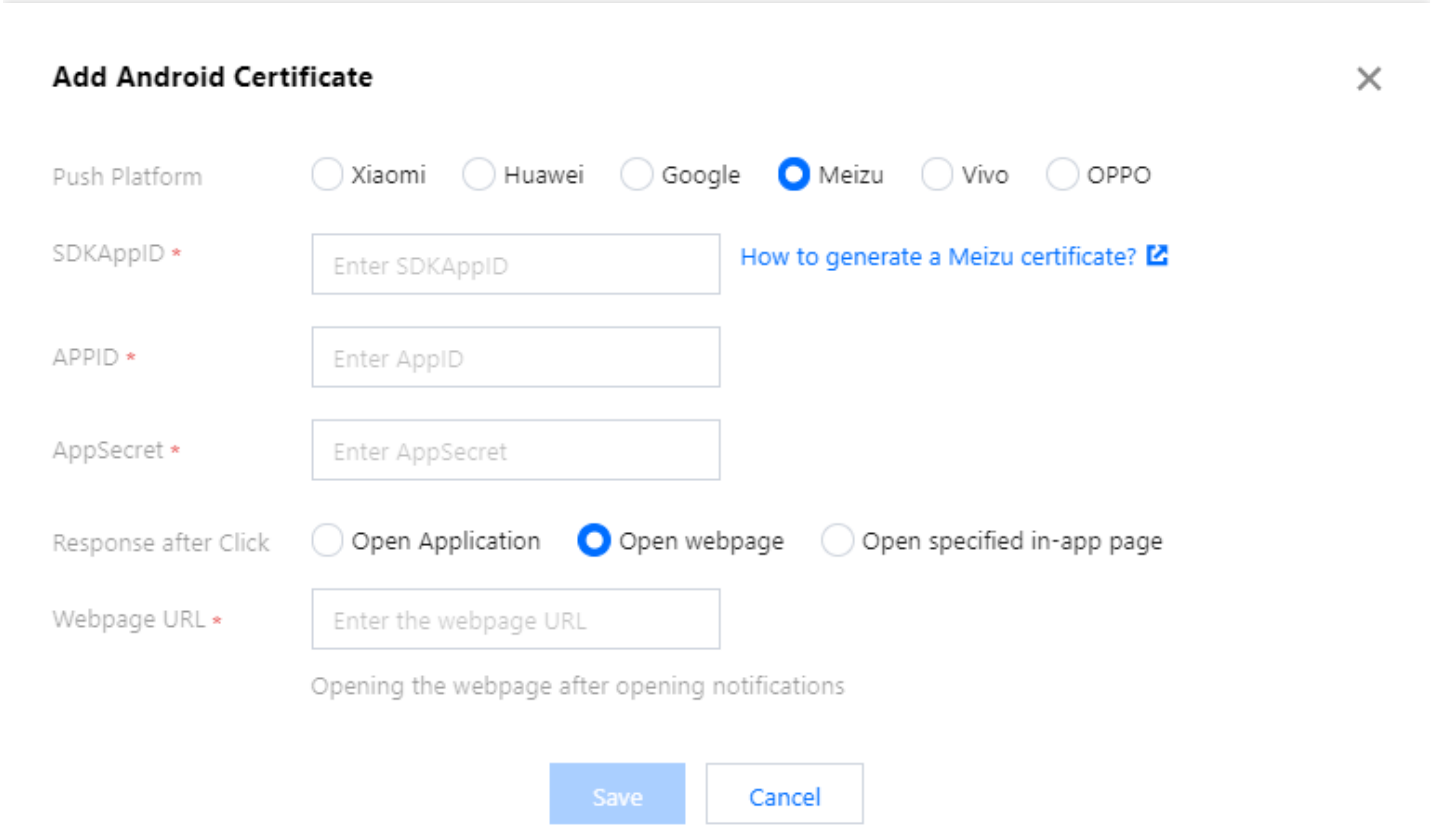
You can select one of the following events: **Open app**, **Open webpage**, or **Open specified in-app page**.

Open app

This is the default event, which opens the app once the notification bar message is clicked.

Open webpage

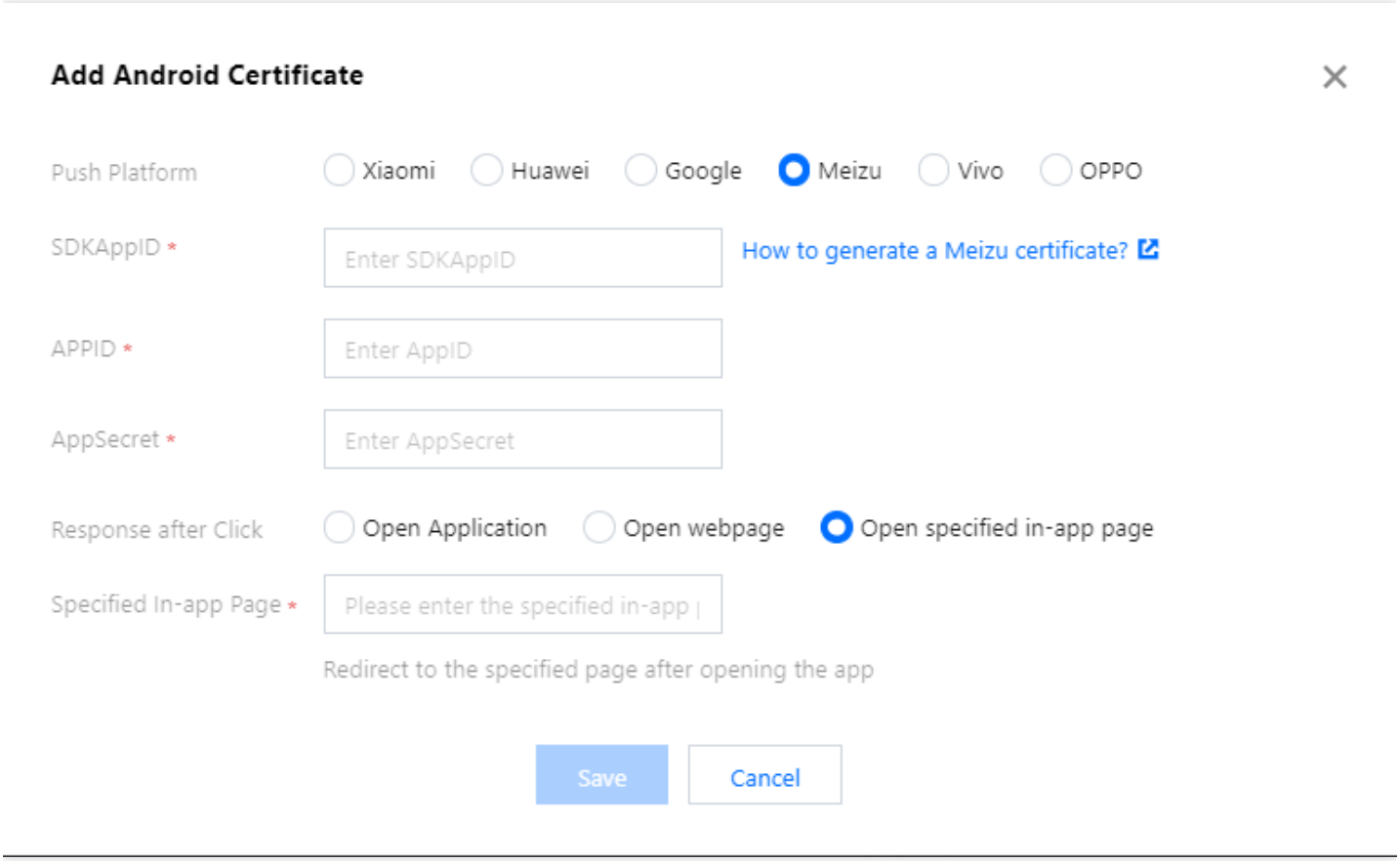
You need to select **Open webpage** when adding a certificate and enter a URL that starts with either `http://` or `https://`, such as `https://cloud.tencent.com/document/product/269`.



Open specified in-app page

When adding a certificate, you need to choose **Open specified in-app page** and enter the complete class name of the Activity to be opened, for example,

`com.tencent.qcloud.tim.demo.main.MainActivity` .



Custom content pass through

Select **Open app** or **Open specified in-app page** in **Response after Click** when adding a certificate to support custom content pass through.

Step 1. Set custom content (sender)

Set the custom content for the notification bar message before sending the message.

- Below is a simple example on the Android platform. You can also refer to the corresponding logic in the `sendMessage()` method in the ChatProvider.java class in the TUIKit:

```
OfflineMessageContainerBean containerBean = new OfflineMessageContainerBean();
OfflineMessageBean entity = new OfflineMessageBean();
entity.content = message.getExtra().toString();
entity.sender = message.getFromUser();
entity.nickname = chatInfo.getChatName();
entity.faceUrl = TUIChatConfigs.getConfigs().getGeneralConfig().getUserFaceUrl();
containerBean.entity = entity;
V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
v2TIMOfflinePushInfo.setExt(new Gson().toJson(containerBean).getBytes());
// For OPPO, you must set the `ChannelID` to receive push messages. The `ChannelID` must be co
```

```
nsistent with that in the console.
v2TIMOfflinePushInfo.setAndroidOPPOChannelID("tuikit");
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, userID, null,
V2TIMMessage.V2TIM_PRIORITY_DEFAULT, false, v2TIMOfflinePushInfo, new V2TIMSendCallback<v2timm
essage>() {
@Override
public void onError(int code, String desc) {}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {}
@Override
public void onProgress(int progress) {}
});
```

- For information on configurations for the IM server, refer to the OfflinePushInfo Format Example.

Step 2. Set custom content (receiver)

Clicking a notification bar message triggers a callback of `onNotificationClicked(Context context, MzPushMessage mzPushMessage)`, which is part of the Meizu Push SDK. The custom content can be obtained from the value of `mzPushMessage`.

```
String extContent = mzPushMessage.getSelfDefineContentString();
```

Alternatively, the client can obtain the custom content from the opened `Activity`. You can refer to the `parseOfflineMessage(Intent intent)` implementation method in the OfflineMessageDispatcher.java class.

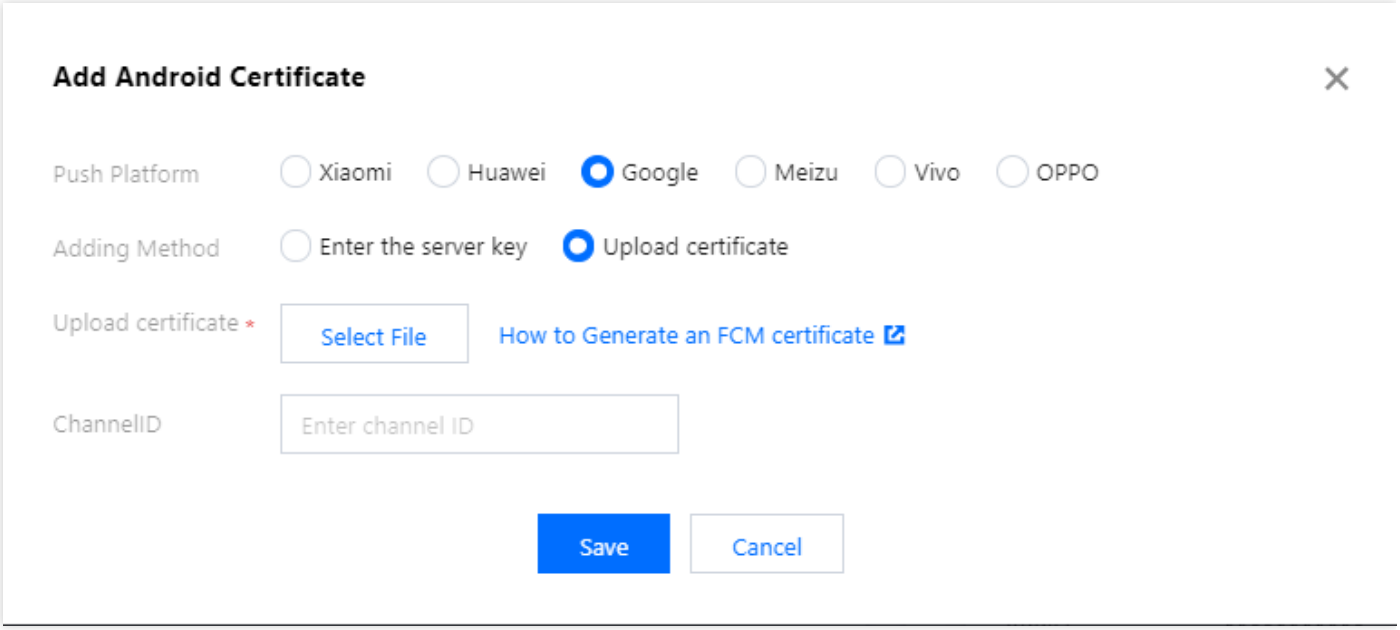
```
Bundle bundle = getIntent().getExtras();
String extContent = bundle.getString("ext");
```

Google FCM Push

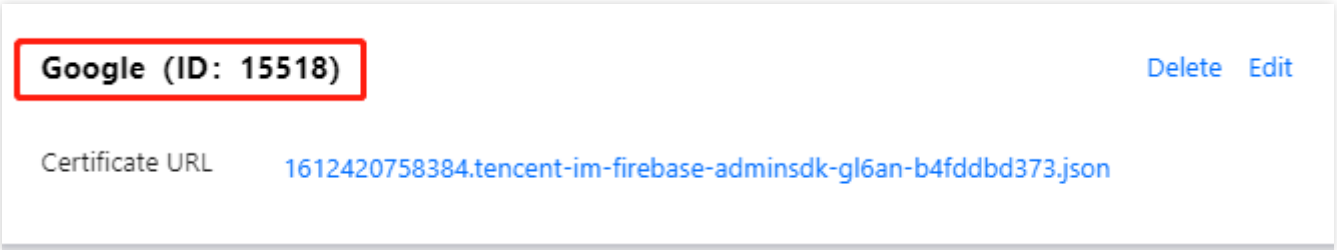
Integrating the SDK

1. Register with Firebase Cloud Messaging and create an app.
2. Log in to the Firebase console and click your app card to go to the app configuration page. Click ⚙ on the right side of **Project Overview**, choose **Project Settings** > **Service Account**, and click **Generate New Private Key** to generate a new private key file.
3. Log in to the Tencent Cloud IM console and click the target app card to go to the basic configuration page of the app. Click **Add Certificate** under **Android Platform Push Settings**.

Upload the private key file obtained in Step 2.



4. Click **Confirm** to save the information. Take note of the `ID` of the certificate. Certificate information takes effect within 10 minutes after you save it.



Integrating the push SDK

1. Add the FCM dependency: implementation 'com.google.firebase:firebase-messaging:20.2.3'.
2. Refer to Firebase Cloud Messaging to set up Firebase. Refer to the FCM Testing Guide to test notification messages to ensure that FCM was integrated properly.
3. After calling `FirebaseInstanceId.getInstance().getInstanceId()`, you can obtain the token in the callback. The token is the unique identifier of the current app. After successful login to the IM SDK, you need to call setOfflinePushConfig to report the **certificate ID** and **token** to the IM server.

After the certificate ID and regId are successfully reported, the IM server sends messages via FCM Push notifications to the user when the app has been killed but the user has not logged out of IM.

Custom content pass through

Step 1. Set custom content (sender)

Set the custom content for the notification bar message before sending the message.

- Below is a simple example on the Android platform. You can also refer to the corresponding logic in the `sendMessage()` method in the ChatProvider.java class in the TUIKit:

```
OfflineMessageContainerBean containerBean = new OfflineMessageContainerBean();
OfflineMessageBean entity = new OfflineMessageBean();
entity.content = message.getExtra().toString();
entity.sender = message.getFromUser();
entity.nickname = chatInfo.getChatName();
entity.faceUrl = TUIChatConfigs.getConfigs().getGeneralConfig().getUserFaceUrl();
containerBean.entity = entity;
V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
v2TIMOfflinePushInfo.setExt(new Gson().toJson(containerBean).getBytes());
// For OPPO, you must set the `ChannelID` to receive push messages. The `ChannelID` must be co
nsistent with that in the console.
v2TIMOfflinePushInfo.setAndroidOPPOChannelID("tuikit");
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, userID, null,
V2TIMMessage.V2TIM_PRIORITY_DEFAULT, false, v2TIMOfflinePushInfo, new V2TIMSendCallback<v2timm
essage>() {
@Override
public void onError(int code, String desc) {}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {}
@Override
public void onProgress(int progress) {}
});
```

- For information on configurations for the IM server, refer to the OfflinePushInfo Format Example.

Step 2. Set custom content (receiver)

When the notification bar message is clicked, the client can obtain the custom content from the corresponding `Activity`. You can refer to the `parseOfflineMessage(Intent intent)` implementation method in the OfflineMessageDispatcher.java class.

```
Bundle bundle = getIntent().getExtras();
String value = bundle.getString("ext");
```

Setting Custom iOS Push Alert Sound

When calling sendMessage to send messages, use the setIOSSound API in `V2TIMOfflinePushInfo` to set the sound for push notifications on iOS devices.

Setting Custom Display for Offline Push

When calling sendMessage to send messages, use setTitle and setDesc in `V2TIMOfflinePushInfo` to set the title and content of notification bar messages respectively.

FAQs

How to set a custom sound for push notifications on Android phones?

Currently, most vendors do not support setting a custom sound for push notifications, therefore it is not supported by the IM SDK.

Why do OPPO mobile phones fail to receive offline push messages?

This generally occurs for the following reasons:

- According to requirements on the official website of OPPO Push, ChannelID must be configured on OPPO mobile phones that run Android 8.0 or later versions. Otherwise, push messages cannot be displayed. For the configuration method, see OPPO Push configuration.
- The custom content in the message for pass-through offline push is not in the JSON format. As a result, OPPO mobile phones do not receive the push message.

Why doesn't offline push work for custom messages?

The offline push for custom messages is different from that for ordinary messages. As we cannot parse the content of custom messages, the push content cannot be determined. Therefore, by default, custom messages are not pushed offline. If you need offline push for custom messages, you need to set the desc field in offlinePushInfo during sendMessage, and the `desc` information will be displayed by default during push.

How do I disable the receiving of offline push messages?

To disable the receiving of offline push messages, set the `config` parameter of the setOfflinePushConfig API to `null`. This feature is supported from v5.6.1200.

Offline Push (iOS)

Last updated : 2021-10-15 16:42:58

[](id: configuring push)

Configuring Offline Push

If you want to receive APNs offline message notifications, follow these steps:

1. Apply for an APNs certificate.
2. Upload the certificate to the console.
3. The app requests deviceToken from Apple every time it logs in.
4. Call setAPNS to report the token to the IM backend.

When the app configured with APNs switches to the background or is killed by the user, the Tencent Cloud backend pushes offline messages to the device through Apple's APNs. For more information, see Apple Push Notification Service.

> Note :
> Users who have logged out normally or have been forced offline will not receive any message notifications.

Step 1: apply for an APNs certificate

For more information on how to apply for an APNs certificate, see Applying for an Apple Push Certificate.

Step 2: upload the certificate to the console

1. Log in to the IM console.
2. Click the target app card to go to its basic configuration page.
3. Click **Add Certificate** on the right side of **iOS Platform Push Settings**.
4. Choose the certificate type, upload an iOS certificate (p.12), set the certificate password, and click **OK**.

> Note :

- We recommend that the name of the certificate to be uploaded should be in all English letters (it must not contain special characters such as brackets).
- You need to set a password for the uploaded certificate. Without a password, push messages cannot be received.
- Certificates to be published on App Store need to be set to the Release environment. Otherwise, push cannot be received.
- The uploaded p12 certificate must be an authentic valid certificate that you have personally applied for.

5. After the push certificate information is generated, record the certificate ID.

Step 3: request DeviceToken from the Apple backend

To request DeviceToken from Apple's backend server, add the following code to your app.

```
// Request DeviceToken from the Apple backend
- (void)registNotification
{
if ([[[UIDevice currentDevice] systemVersion] floatValue] >= 8.0)
{
[[UIApplication sharedApplication] registerUserNotificationSettings:
[UIUserNotificationSettings settingsForTypes:
(UIUserNotificationTypeSound | UIUserNotificationTypeAlert | UIUserNotificationTypeBadge)
categories:nil]];
[[UIApplication sharedApplication] registerForRemoteNotifications];
}
else
{
[[UIApplication sharedApplication] registerForRemoteNotificationTypes:
(UIUserNotificationTypeBadge | UIUserNotificationTypeSound | UIUserNotificationTypeAlert)];
}
}
//The callback of AppDelegate returns deviceToken, which needs to be reported to the Tencent Clou
d backend after login.
-(void)application:(UIApplication *)app didRegisterForRemoteNotificationsWithDeviceToken:(NSData
*)deviceToken
{
// Note the deviceToken returned by Apple.
_deviceToken = deviceToken;
}
```

Step 4: upload the token to Tencent Cloud after IM SDK login

After logging in to the IM SDK, call setAPNS to upload the DeviceToken obtained in step 3 to the Tencent Cloud backend:

```
V2TIMAPNSConfig *confg = [[V2TIMAPNSConfig alloc] init];
// Push certificate ID, generated after the push certificate (p.12) is uploaded to the IM console backend
confg.businessID = businessID;
// The deviceToken requested by Apple’s backend
confg.token = deviceToken;
[[V2TIMManager sharedInstance] setAPNS:confg succ:^{
NSLog(@"-----> APNS set successfully");;
} fail:^(int code, NSString *msg) {
NSLog(@"-----> Failed to set APNS");
}];
```

Note :
`businessID` must be consistent with the certificate ID assigned by the console.

General push rules

For one-to-one messages, the APNs push rules are as follows. Note that the nickname is the sender's nickname. If the nickname is not set, only the content is displayed.

```
Nickname:content
```

For group messages, the APNs push rules are as follows. The priority order for displaying the name is: message sender's `group name card` > `group nickname`. If neither is available, no name is displayed.

```
Name (Group Name):content
```

Push rules for different types of messages

The APNs push content consists of the content of each `Elem` in the message body. The display of different `Elem` in offline messages is shown in the following table.

Parameter	Description
Text Elem	Directly display the content
Audio Elem	Display `[audio]`

Parameter	Description
File Elem	Display `[file]`
Image Elem	Display `[image]`
Custom Elem	Display the desc field set when message is sent. If `desc` is not set, it will not be pushed.

Communication among multiple apps

If you set `SDKAppID` to the same value for multiple apps, these apps communicate with each other. Different apps need to use different push certificates, and you need to apply for an APNs certificate for each app and complete [offline push configuration](#configuring push).

Setting Custom iOS Push Alert Sound

Set the `iOSSound` field of offlinePushInfo when calling sendMessage to send messages. `iOSSound` passes the name (with the extension) of the audio file which must be linked to the Xcode project.

Setting Custom Display for Offline Push

Set the `title` and `desc` fields of offlinePushInfo when calling sendMessage to send messages. Once set, the `title` content will be added to the default push content and `desc` will be displayed as the push content.

Setting Custom Click-to-Redirect Logic

Set the `ext` field of offlinePushInfo when calling sendMessage to send messages. When the user receives offline push and starts the app, the `ext` field can be obtained from the `AppDelegate -> didReceiveRemoteNotification` system callback, and the user is redirected to the UI as specified by `ext`.

The following example assumes that Denny sends a message to Vinson.

- Sender: Denny needs to set `ext` before sending a message.

```
// Denny sets `offlinePushInfo` and specifies `ext` before sending a message
V2TIMMessage *msg = [[V2TIMManager sharedInstance] createTextMessage:@"Text message"];
```

```
V2TIMOfflinePushInfo *info = [[V2TIMOfflinePushInfo alloc] init];
info.ext = @"jump to denny";
[[V2TIMManager sharedInstance] sendMessage:msg receiver:@"vinson" groupID:nil priority:V2TIM_PRIO
RITY_DEFAULT
onlineUserOnly:NO offlinePushInfo:info progress:^(uint32_t progress) {
} succ:^{
} fail:^(int code, NSString *msg) {
}];
```

- Recipient: although Vinson's app is not online, it can still receive an APNs offline message notification. When Vinson clicks this notification, the app is started.

```
// Vinson receives the following callback when the app is started.
- (void)application:(UIApplication *)application didReceiveRemoteNotification:(NSDictionary *)use
rInfo
fetchCompletionHandler:(void (^)(UIBackgroundFetchResult result))completionHandler {
// Resolve the extension field `desc`.
if ([userInfo[@"ext"] isEqualToString:@"jump to denny"]) {
// Go to the chat window with Denny.
}
}
```

FAQs

Why doesn't offline push work for common messages?

First, verify that the app and the certificate have the same runtime environment. Otherwise, offline pushes will not be received.
Then, verify that the app and the certificate run in the development environment. If that is the case, requesting `deviceToken` from Apple might fail. Please switch to the production environment to solve the problem.

Why doesn't offline push work for custom messages?

The offline push for custom messages is different from common messages. Since we are not able to resolve the content of custom messages and determine what to push, we simply do not push custom messages by default. If you want to enable offline push for custom messages, set the `desc` field of offlinePushInfo when calling sendMessage, then the content of `desc` will be used for display by offline push.

Local Search
Local Search (Android)

Last updated : 2021-10-15 16:46:33

Local search is supported starting from Enhanced Edition v5.4.666. To use local search, you need to purchase the Flagship Edition package. For operation details, please see Purchase Guide.

Feature Demonstration

The search API interface consists of three parts: the upper part is for friend search, the middle part is for group and group member search, and the lower part is for message search, where messages are classified by conversation.

Integration Guide

Scheme 1: Integrating TUIKit search source code

Step 1. Purchase the package

Purchase the Flagship Edition package by referring to Purchase Guide.

Step 2. Download source code

Download source code to integrate the TUIKit module. TUIKit supports local search starting from v5.4.666.

```
implementation project(':tuikit')
```

Step 3. Initialize TUIKit and log in

```
// Initialize TUIKit
TUIKitConfigs configs = TUIKit.getConfigs();
TUIKit.init(this, SDKAPPID, configs);
// Log in to TUIKit
TUIKit.login(userID, userSig, new IUIKitCallBack() {
@Override
public void onSuccess(Object data) {
// Login succeeded
}
```

```
@Override
public void onError(String module, final int code, final String desc) {
// Login failed
}
});
```

Step 4. Start the search interface

You only need to start `SearchMainActivity` .

Scheme 2: Integrate the IM SDK search API

Step 1. Purchase the package

Purchase the Flagship Edition package by referring to Purchase Guide.

Step 2. Integrate IM SDK Enhance Edition

The IM SDK supports local search starting from v5.4.666.

```
dependencies {
api 'com.tencent.imsdk:imsdk-plus:Version number`
}
```

Step 3. Call the local user profile search API

You can call the searchFriends API to search for local user profiles. The API supports the following search fields: userID, nickName, and remark.

Step 4. Call the local group and group member search APIs

You can call the searchGroups API to search for the profiles of local groups.
You can call the searchGroupMembers API to search for the profiles of local group members. There are two cases, depending on whether the value of `groupIDList` in V2TIMGroupMemberSearchParam is `null` :

- groupIDList == null: search the members in all groups, and the returned results will be classified by group ID.
- groupIDList != null: search the members in a specified group.

Step 5. Call the local message search API

You can enter keywords in the search box to call searchLocalMessages to search for local messages. There are two cases, depending on whether the value of conversationID in the V2TIMMessageSearchParam API is `null` :

- conversationID == null: search all conversations, and the returned results will be classified by conversation.
- conversationID != null: search a specified conversation.

Displaying recent active conversations

As shown in Figure 1, the bottom part of the search interface is the list of the last 3 conversations to which the messages found belong. The implementation method is as follows:

- In the V2TIMMessageSearchParam API, conversationID is set to `null`, indicating the messages of all conversations are searched; pageIndex is set to `0`, indicating data on page 0 of the conversations to which the messages found belong; pageSize indicates the number of recent conversations to be returned. Usually, 3 recent conversations are displayed on the UI.
- In the search callback result API V2TIMMessageSearchResult, totalCount indicates the total number of conversations to which the matched messages belong; messageSearchResultItems indicates the information of the recent conversations (conversation quantity specified by pageSize). In V2TIMMessageSearchResultItem, conversationID indicates the conversation ID, messageCount indicates the total number of messages found in the current conversation, and messageList indicates the list of messages found. `messageList` has two cases:
  - If the number of messages found is greater than 1, `messageList` is empty. You can display related chat records (quantity specified by `messageCount`) on the UI.
  - If the number of messages found is equal to 1, `messageList` is the matched message. You can display the message content on the UI and highlight the search keyword, such as **test** in the above figures.

Sample

```
List<String> keywordList = new ArrayList<>();
keywordList.add("test");
V2TIMMessageSearchParam v2TIMMessageSearchParam = new V2TIMMessageSearchParam();
// Setting `conversationID` to `null` is to search for messages in all conversations and the resu
lts will be classified by conversation
v2TIMMessageSearchParam.setConversationID(null);
v2TIMMessageSearchParam.setKeywordList(keywordList);
v2TIMMessageSearchParam.setPageSize(3);
v2TIMMessageSearchParam.setPageIndex(0);
V2TIMManager.getMessageManager().searchLocalMessages(v2TIMMessageSearchParam, new V2TIMValueCallb
ack<V2TIMMessageSearchResult>() {
@Override
public void onSuccess(V2TIMMessageSearchResult v2TIMMessageSearchResult) {
// Total number of matched conversations to which messages belong
int totalCount = v2TIMMessageSearchResult.getTotalCount();
// Last 3 messages classified by conversation
```

```
List<V2TIMMessageSearchResultItem> resultItemList = v2TIMMessageSearchResult.getMessageSearchResu
ltItems();
for (V2TIMMessageSearchResultItem resultItem : resultItemList) {
// Conversation ID
String conversationID = resultItem.getConversationID();
// Total number of messages matching the conversation
int totalMessageCount = resultItem.getMessageCount();
// List of messages. If `totalMessageCount` is greater than 1, the list is empty. If `totalMessag
eCount` is equal to 1, the list contains the current message.
List<V2TIMMessage> v2TIMMessageList = resultItem.getMessageList();
}
}
@Override
public void onError(int code, String desc) {}
```

Displaying the list of conversations to which the messages found belong

For example, you can click **More Chat History** in Figure 1 to redirect to the list of conversations to which the messages found belong, as shown in Figure 2. The search parameters and results are similar to those in the preceding scenario. To avoid memory ballooning, it is strongly recommended that you load the conversation list with pagination. For example, to display 10 conversations as the result, you can set the search parameter API V2TIMMessageSearchParam as follows:

- First call: Set `pageSize` to `10` and `pageIndex` to `0` , and call searchLocalMessages. Then you can get the total number of conversations from `totalCount` in the callback.
- Page quantity calculation: totalPage = (totalCount % pageSize == 0) ? (totalCount / pageSize) : (totalCount / pageSize + 1)
- Second call: You can specify `pageIndex` (`pageIndex` < `totalPage`) to return the subsequent page number.

Sample

```
......
// Calculate the total number of pages, given that 10 messages are displayed per page
int totalPage = (totalCount % 10 == 0) ? (totalCount / 10) : (totalCount / 10 + 1);
......
private void searchConversation(int index) {
if (index >= totalPage) {
return;
}
List<String> keywordList = new ArrayList<>();
keywordList.add("test");
V2TIMMessageSearchParam v2TIMMessageSearchParam = new V2TIMMessageSearchParam();
v2TIMMessageSearchParam.setConversationID(null);
v2TIMMessageSearchParam.setKeywordList(keywordList);
v2TIMMessageSearchParam.setPageSize(10);
```

```
v2TIMMessageSearchParam.setPageIndex(index);
V2TIMManager.getMessageManager().searchLocalMessages(v2TIMMessageSearchParam, new V2TIMValueCallb
ack<V2TIMMessageSearchResult>() {
@Override
public void onSuccess(V2TIMMessageSearchResult v2TIMMessageSearchResult) {
// Total number of matched conversations to which messages belong
int totalCount = v2TIMMessageSearchResult.getTotalCount();
// Calculate the total number of pages, given that 10 messages are displayed per page
int totalPage = (totalCount % 10 == 0) ? (totalCount / 10) : (totalCount / 10 + 1);
// Information of messages classified by conversation
List<V2TIMMessageSearchResultItem> resultItemList = v2TIMMessageSearchResult.getMessageSearchResu
ltItems();
for (V2TIMMessageSearchResultItem resultItem : resultItemList) {
// Conversation ID
String conversationID = resultItem.getConversationID();
// Total number of messages matching the conversation
int totalMessageCount = resultItem.getMessageCount();
// List of messages. If `totalMessageCount` is greater than 1, the list is empty. If `totalMessag
eCount` is equal to 1, the list contains the current message.
List<V2TIMMessage> v2TIMMessageList = resultItem.getMessageList();
}
@Override
public void onError(int code, String desc) {}
});
}
// Load the next page
public void loadMore() {
searchConversation(++pageIndex);
}
```

Searching for messages in a specified conversation

Figure 2 shows the effect of displaying the conversation list, while Figure 3 shows the effect of displaying the list of the messages found in a specified conversation. To avoid memory ballooning, it is strongly recommended that you load the message list with pagination. The implementation method is as follows:

- In the V2TIMMessageSearchParam API, set conversationID to the ID of the conversation to search, and set pageIndex and pageSize by referring to the settings in the preceding calculation mode.
- In the V2TIMMessageSearchResult API, totalCount indicates the total number of messages matched in the conversation, and messageSearchResultItems lists only the results in the conversation. In V2TIMMessageSearchResultItem, messageCount is the number of messages on the current page, and messageList is the list of messages on the current page.

Sample

```
......
// Calculate the total number of pages, given that 10 messages are displayed per page
int totalMessagePage = (totalMessageCount % 10 == 0) ? (totalMessageCount / 10) : (totalMessageCo
unt / 10 + 1);
......
private void searchMessage(int index) {
if (index >= totalMessagePage) {
return;
}
List<String> keywordList = new ArrayList<>();
keywordList.add("test");
V2TIMMessageSearchParam v2TIMMessageSearchParam = new V2TIMMessageSearchParam();
v2TIMMessageSearchParam.setConversationID(conversationID);
v2TIMMessageSearchParam.setKeywordList(keywordList);
v2TIMMessageSearchParam.setPageSize(10);
v2TIMMessageSearchParam.setPageIndex(index);
V2TIMManager.getMessageManager().searchLocalMessages(v2TIMMessageSearchParam, new V2TIMValueCallb
ack<V2TIMMessageSearchResult>() {
@Override
public void onSuccess(V2TIMMessageSearchResult v2TIMMessageSearchResult) {
// Total number of messages matching the conversation
int totalMessageCount = v2TIMMessageSearchResult.getTotalCount();
// Calculate the total number of pages, given that 10 messages are displayed per page
int totalMessagePage = (totalMessageCount % 10 == 0) ? (totalMessageCount / 10) : (totalMessageCo
unt / 10 + 1);
// Information of the messages on the conversation page
List<V2TIMMessageSearchResultItem> resultItemList = v2TIMMessageSearchResult.getMessageSearchResu
ltItems();
for (V2TIMMessageSearchResultItem resultItem : resultItemList) {
// Conversation ID
String conversationID = resultItem.getConversationID();
// Number of messages on the current page
int totalMessageCount = resultItem.getMessageCount();
// List of messages on the current page
List<V2TIMMessage> v2TIMMessageList = resultItem.getMessageList();
}
@Override
public void onError(int code, String desc) {
}
});
}
// Load the next page
public void loadMore() {
searchMessage(++pageIndex);
}
```

FAQs

1. How do I search for custom messages?

Use the createCustomMessage (byte[] data, String description, byte[] extension) API to create and send a search request. In the request, you need to specify the text to search in the `description` parameter. Custom messages created via the createCustomMessage (byte[] data) API cannot be searched because the binary data stream passed in by parameters is saved locally.
If you configure the offline push feature and the `description` parameter, custom messages will also be pushed offline, and the content specified in the `description` parameter will be displayed in the notification bar. If offline push is not needed, disable it using disablePush in V2TIMOfflinePushInfo in the sendMessage API. If you do not want to display the searched text in the push notification bar, set other push content using setDesc in V2TIMOfflinePushInfo.

2. How do I search for rich media messages?

Rich media messages include file, image, voice, and video messages.
For file messages, the screen usually displays the filename. Therefore, you can set the `fileName` parameter as the searched content when creating messages. If `fileName` is not set, the system gets the filename from `filePath` and saves it to the local storage and the server.
For image, voice, and video messages, the screen usually displays the thumbnail or duration. In this case, you can specify the message type to search by type, but cannot search by keywords.

Local Search (iOS)

Last updated : 2021-10-15 16:51:43

Local search is supported starting from Enhanced Edition v5.4.666. To use local search, you need to purchase the Flagship Edition package. For operation details, please see Purchase Guide.

Feature Demonstration

The search API interface consists of three parts: the upper part is for friend search, the middle part is for group and group member search, and the lower part is for message search, where messages are classified by conversation.
Download and install the app to experience it now.

Integration Guide

Scheme 1: Integrating TUIKit search source code

Step 1. Purchase the package

Purchase the Flagship Edition package by referring to Purchase Guide.

Step 2. Download source code

Download source code to integrate the TUIKit module. TUIKit supports local search starting from v5.4.666.

```
pod 'TXIMSDK_TUIKit_iOS', '~>'5.4.666'
```

Step 3. Initialize TUIKit and log in

```
// Initialize TUIKit
[[TUIKit sharedInstance] setupWithAppId:SDKAPPID];
// Log in to TUIKit
[[TUIKit sharedInstance] login:identifier userSig:sig succ:^{
// Login succeeded
} fail:^(int code, NSString *msg) {
// Login failed
}];
```

Step 4. Start the search interface

Enable `TUISearchViewController`.

Scheme 2: Integrate the IM SDK search API

Step 1. Purchase the package

Purchase the Flagship Edition package by referring to Purchase Guide.

Step 2. Integrate IM SDK Enhance Edition

The IM SDK supports local search starting from v5.4.666.

```
pod 'TXIMSDK_Plus_iOS', '~>'5.4.666'
```

Step 3. Call the local user profile search API

You can call the searchFriends API to search for local user profiles. The API supports the following search fields: userID, nickName, and remark.

Step 4. Call the local group and group member search APIs

You can call the searchGroups API to search for the profiles of local groups.
You can call the searchGroupMembers API to search for the profiles of local group members. There are two cases, depending on whether the value of `groupIDList` in V2TIMGroupMemberSearchParam is `nil`:

- groupIDList == nil: search the members in all groups, and the returned results will be classified by group ID.
- groupIDList != nil: search the members in a specified group.

Step 5. Call the local message search API

You can enter keywords in the search box to call searchLocalMessages to search for local messages. There are two cases, depending on whether the value of conversationID in the V2TIMMessageSearchParam API is `nil`:

- conversationID == nil: search all conversations, and the returned results will be classified by conversation.
- conversationID != nil: search a specified conversation.

Displaying recent active conversations

As shown in Figure 1, the bottom part of the search interface is the list of the last 3 conversations to which the messages found belong. The implementation method is as follows:

- In the V2TIMMessageSearchParam API, conversationID is set to `nil`, indicating that the messages of all conversations are searched; pageIndex is set to `0`, indicating data on page 0 of the conversations to which the messages found belong; pageSize indicates the number of recent conversations to be returned. Usually, 3 recent conversations are displayed on the UI.
- In the search callback result API V2TIMMessageSearchResult, totalCount indicates the total number of conversations to which the matched messages belong; messageSearchResultItems indicates the information of the recent conversations (conversation quantity specified by pageSize). In V2TIMMessageSearchResultItem, conversationID indicates the conversation ID, messageCount indicates the total number of messages found in the current conversation, and messageList indicates the list of messages found. `messageList` has two cases:
  - If the number of messages found is greater than 1, `messageList` is empty. You can display related chat records (quantity specified by `messageCount`) on the UI.
  - If the number of messages found is equal to 1, `messageList` is the matched message. You can display the message content on the UI and highlight the search keyword, such as **test** in the above figures.

Sample

```
V2TIMMessageSearchParam *param = [[V2TIMMessageSearchParam alloc] init];
param.keywordList = @[@"test"];
// Setting `conversationID` to `nil` is to search for messages in all conversations and the resul
ts will be classified by conversation
param.conversationID = nil;
param.pageIndex = 0;
param.pageSize = 3;
[V2TIMManager.sharedInstance searchLocalMessages:param succ:^(V2TIMMessageSearchResult *searchRes
ult) {
// Total number of matched conversations to which messages belong
NSInteger totalCount = searchResult.totalCount;
// Last 3 messages classified by conversation
NSArray<V2TIMMessageSearchResultItem *> *messageSearchResultItems = searchResult.messageSearchRes
ultItems;
for (V2TIMMessageSearchResultItem *searchItem in messageSearchResultItems) {
// Conversation ID
NSString *conversationID = searchItem.conversationID;
// Total number of messages matching the conversation
NSUInteger messageCount = searchItem.messageCount;
// Message list
NSArray<V2TIMMessage *> *messageList = searchItem.messageList ?: @[];
}
} fail:^(int code, NSString *desc) {
// fail
}];
```

Displaying the list of conversations to which the messages found belong

For example, you can click **More Chat History** in Figure 1 to redirect to the list of conversations to which the messages found belong, as shown in Figure 2. The search parameters and results are similar to those in the preceding scenario. To avoid memory ballooning, it is strongly recommended that you load the conversation list with pagination. For example, to display 10 conversations as the result, you can set the search parameter API V2TIMMessageSearchParam as follows:

- First call: Set `pageSize` to `10` and `pageIndex` to `0`, and call searchLocalMessages. Then you can get the total number of conversations from `totalCount` in the callback.
- Page quantity calculation: totalPage = (totalCount % pageSize == 0) ? (totalCount / pageSize) : (totalCount / pageSize + 1)
- Second call: You can specify `pageIndex` (`pageIndex` < `totalPage`) to return the subsequent page number.

Sample

```
......
// Calculate the total number of pages, given that 10 messages are displayed per page
NSInteger totalPage = (totalCount % 10 == 0) ? (totalCount / 10) : (totalCount / 10 + 1);
......
- (void)searchConversation:(NSUInteger)index {
if (index >= totalPage) {
return;
}
V2TIMMessageSearchParam *param = [[V2TIMMessageSearchParam alloc] init];
param.keywordList = @[@"test"];
param.conversationID = nil;
param.pageIndex = index;
param.pageSize = 10;
[V2TIMManager.sharedInstance searchLocalMessages:param succ:^(V2TIMMessageSearchResult *searchRes
ult) {
// Total number of matched conversations to which messages belong
NSUInteger totalCount = searchResult.totalCount;
// Calculate the total number of pages, given that 10 messages are displayed per page
NSUInteger totalPage = (totalCount % 10 == 0) ? (totalCount / 10) : (totalCount / 10 + 1);
// Information of messages classified by conversation
NSArray<V2TIMMessageSearchResultItem *> *messageSearchResultItems = searchResult.messageSearchRes
ultItems;
for (V2TIMMessageSearchResultItem *searchItem in messageSearchResultItems) {
// Conversation ID
NSString *conversationID = searchItem.conversationID;
// Total number of messages matching the conversation
NSUInteger totalMessageCount = searchItem.messageCount;
// List of messages. If `totalMessageCount` is greater than 1, the list is empty. If `totalMessag
```

```
eCount` is equal to 1, the list contains the current message.
NSArray<V2TIMMessage *> *messageList = searchItem.messageList ?: @[];
}
} fail:^(int code, NSString *desc) {
// fail
}];
}
// Load the next page
- (void)loadMore {
[self searchConversation:++pageIndex];
}
```

Searching for messages in a specified conversation

Figure 2 shows the effect of displaying the conversation list, while Figure 3 shows the effect of displaying the list of the messages found in a specified conversation. To avoid memory ballooning, it is strongly recommended that you load the message list with pagination. The implementation method is as follows:

- In the V2TIMMessageSearchParam API, set conversationID to the conversation ID, and set pageIndex and pageSize by referring to the settings in the preceding calculation mode.
- In the V2TIMMessageSearchResult API, totalCount indicates the total number of messages matched in the conversation, and messageSearchResultItems lists only the results in the conversation. In the V2TIMMessageSearchResultItem API, messageCount is the number of messages on the current page, and messageList is the list of messages on the current page.

Sample

```
......
// Calculate the total number of pages, given that 10 messages are displayed per page
NSInteger totalMessagePage = (totalMessageCount % 10 == 0) ? (totalMessageCount / 10) : (totalMes
sageCount / 10 + 1);
......
- (void)searchMessage:(NSUInteger)index {
if (index >= totalMessagePage) {
return;
}
V2TIMMessageSearchParam *param = [[V2TIMMessageSearchParam alloc] init];
param.keywordList = @[@"test"];
param.conversationID = conversationID;
param.pageIndex = index;
param.pageSize = 10;
[V2TIMManager.sharedInstance searchLocalMessages:param succ:^(V2TIMMessageSearchResult *searchRes
ult) {
// Total number of messages matching the conversation
NSUInteger totalMessageCount = searchResult.totalCount;
```

```
// Calculate the total number of pages, given that 10 messages are displayed per page
NSUInteger totalMessagePage = (totalMessageCount % 10 == 0) ? (totalMessageCount / 10) : (totalMe
ssageCount / 10 + 1);
// Information of the messages on the conversation page
NSArray<V2TIMMessageSearchResultItem *> *messageSearchResultItems = searchResult.messageSearchRes
ultItems;
for (V2TIMMessageSearchResultItem *searchItem in messageSearchResultItems) {
// Conversation ID
NSString *conversationID = searchItem.conversationID;
// Number of messages on the current page
NSUInteger totalMessageCount = searchItem.messageCount;
// List of messages on the current page
NSArray<V2TIMMessage *> *messageList = searchItem.messageList ?: @[];
}
} fail:^(int code, NSString *desc) {
// fail
}];
}
// Load the next page
- (void)loadMore {
[self searchMessage:++pageIndex];
}
```

FAQs

1. How do I search for custom messages?

Use the createCustomMessage:desc:extension API to create and send a search request. In the request, you need to specify the text to search in the `desc` parameter. Custom messages created via the createCustomMessage API cannot be searched because the binary data stream passed in by parameters is saved locally.
If you configure the offline push feature and the `desc` parameter, custom messages will also be pushed offline, and the content specified in the `desc` parameter will be displayed in the notification bar. If offline push is not needed, disable it using disablePush in V2TIMOfflinePushInfo in the sendMessage API. If you do not want to display the searched text in the push notification bar, set other push content using desc in V2TIMOfflinePushInfo.

2. How do I search for rich media messages?

Rich media messages include file, image, voice, and video messages.
For file messages, the screen usually displays the filename. Therefore, you can set the `fileName` parameter as the searched content when creating messages. If `fileName` is not set, the system gets the filename from `filePath` and saves it to the local storage and the server.

For image, voice, and video messages, the screen usually displays the thumbnail or duration. In this case, you can specify the message type to search by type, but cannot search by keywords.

Update Logs (Web & Mini Programs)

Last updated : 2021-08-17 18:05:37

2.12.1 @2021.7.20

New features

- Supports counting unread meeting groups.
- The TIM.EVENT.MESSAGE_MODIFIED event is added. When a third-party calls back a modified message, the SDK uses this event to notify the message sender of the message modification.

Bug fixing

- Fixed the issue where group roaming messages occasionally get lost when they are pulled.
- Fixed the `xx.toFixed is not a function` issue that may occur during uni-app integration.

2.12.0 @2021.7.5

New features

- The deleteMessage API for deleting messages is added.
- During conversation list synchronization, `lastMessage` can be set to a recalled message.
- getGroupMemberList supports pulling the group joining time `joinTime`.

Bug fixing

The `nick` value is incorrect in the notifications sent when a user is set or canceled as the admin.

2.11.2 @2021.6.16

New features

- Supports WebSocket. Find the WebSocket upgrade guide here.
- Allows uni-app to send image, video, and other file messages.

2.10.2 @2021.4.27

New features

- The custom field `cloudCustomData` can be set during message creation to meet diverse business needs.
- When createGroup or addGroupMember is called, if a single user exceeds the maximum number of groups a single user can join, use `overLimitUserIDList` to notify the access side.

Bug fixing

- After an audio-video group (AVChatRoom) was created in the console and a group owner was specified, messages sent by the RESTful API for Sending System Messages in a Group would be repeated on the group owner side after the group owner joined the group.
- Nickname missing occurred in createForwardMessage.
- Occasional errors occurred in downloadMergerMessage.

2.10.1 @2021.3.19

New features

- The createMergerMessage API for creating combined messages.
- The createForwardMessage API for creating forward messages.
- When an account logs in on multiple instances or clients, once conversation read is reported on one instance or client, the unread count of the conversation will be synchronously cleared on the web client.

Changes

The MTA statistics feature was deprecated.

Bug fixing

- Web: when an account logged in on multiple instances, the profile photo and nickname of the other party in a one-to-one conversation were incorrect.
- When you called back and called the RESTful API to recall messages frequently after sending messages, some of them were not recalled correctly.

2.9.3 @2021.2.3

Changes

If a user hasn't joined a group (not an audio-video group), calling quitGroup will return error code 2623, indicating that the user is not in the group.

Bug fixing

`avatar` (profile photo) or `nick` (nickname) was inconsistent in the one-to-one conversation message list.

2.9.2 @2021.1.26

New features

- Support for sending and receiving one-to-one messages with `avatar` (profile photo) and `nick` (nickname) displayed.

- Support for the Tencent Cloud IM upload plugin tim-upload-plugin. This plugin enables more secure file upload, supports web, WeChat, QQ, Baidu, Toutiao, and Alipay Mini Program platforms, and is merely 26 KB. For more information, see registerPlugin.

Bug fixing

- When a user joined an audio-video group anonymously after logging out, the error code 70402 was returned in the response packet during a long polling.
- The browser environment was misjudged during Taro 3.0+ integration.
- When the image type and size verification failed, there were errors in the returned data structure.

2.9.1 @2020.12.23

Bug fixing

A compilation error occurred when tim-wx-sdk.js was imported into the basic library 2.14.1 of WeChat Developer Tools.

2.9.0 @2020.12.15

New features

- The createTextAtMessage API allows users to specify @ a specific member or @ all members during a group chat.
- Message added the `namecard` attribute to display group members' group name cards (i.e., their nicknames in a group).

2.8.5 @2020.11.23

Changes

The logout API can be called when the SDK is not ready.

Bug fixing

- Errors occurred in SDK operations when read receipts and read notifications existed at the same time.
- Attempts to anonymously re-join an audio-video group after logout failed.
- The group list was cleared abnormally.

2.8.4 @2020.11.4

New features

- The WeChat, QQ, Baidu, Toutiao, and Alipay Mini Program platforms are supported (currently on the Baidu, Toutiao, and Alipay Mini Program platforms, image, video, or file messages, or other

messages that need to be uploaded to COS, cannot be sent).
- The third-party frameworks of MPX and uni-app are supported.

2.8.1 @2020.10.29

New features

Images in BMP format can be sent.

Changes

`unreadCount` and `lastMessage` of the conversation object are not updated when the sender sends an online message and the recipient receives the online message.

Bug fixing

The SDK could not enter the ready state due to problems synchronizing the list of recent contacts.

2.8.0 @2020.10.20

New features

- getGroupOnlineMemberCount was added to query the number of online users in an audio-video group.
- Image compression was integrated in the sending of image messages. The access side can choose to display the original image or thumbnail image based on business requirements. For more information, see ImagePayload.

Bug fixing

Compatibility issues when Taro 3.x integrates WebIM

Changes

Reduced the SDK size. The size of tim-js-sdk was reduced by 8.5%, and that of tim-wx-sdk was reduced by 15%.

2.7.8 @2020.9.24

New features

The TIM.create API added the `oversea` parameter. When this parameter is set to `true`, the SDK uses a domain name outside the Chinese mainland to avoid interference.

Bug fixing

- The return value for calling relevant APIs was `undefined` when the SDK was in the `not ready` state.

- Issues related to statistics

2.7.7 @2020.8.12

New features

The TIM.EVENT.SDK_RELOAD event was added.

Bug fixing

- Audio-video groups occasionally failed to pull messages in cases where the network was reconnected after a long disconnection or the Mini Program switched to the foreground after running in the background for a long time.
- The type and value of `imageFormat` of an image message were inconsistent with those of the actual image.
- The nicknames displayed in work groups and public groups were incorrect.

2.7.6 @2020.7.9

Bug fixing

Messages occasionally failed to be pulled if an audio-video group (AVChatRoom) was used for a long time.

2.7.5 @2020.7.2

Bug fixing

After the RESTful API for creating a work group was called to create a work group successfully and the group members were specified, messages from group members would fail to be sent.

2.7.2 @2020.6.30

Bug fixing

- Occasionally, when joinGroup was called, the SDK prompted "Already in the group" but in fact the user was not in the group. Consequently, the user could not send or receive messages.
- The count of messages sent in a temporary meeting group was incorrect.

2.7.0 @2020.6.8

New features

Provided support for one-to-one message read receipts (indicating whether the peer has read your messages). For more information, see the event TIM.EVENT.MESSAGE_READ_BY_PEER. In a message that has already been read by the peer, the value of `isPeerRead` is `true`.

Bug fixing

- After a user joined a chat room (ChatRoom), the newly created conversation did not display the last message.
- After login, a user who had not joined an audio-video group (AVChatRoom) could still send a message to the audio-video group (AVChatRoom).

2.6.6 @2020.5.27

Bug fixing

- In audio-video groups (AVChatRoom), messages were occasionally repeatedly displayed on the screen.
- An error was reported when getMessageList received an empty message.
- If login was called again after logout, error 70001 occasionally occurred when joinGroup was called.

2.6.4 @2020.5.8

New features

The sendMessage API added the sending option to support the sending of online messages (no offline or roaming messages; cannot be used for AVChatRoom or BChatRoom) and the configuration of offline push.

2.6.3 @2020.4.26

Bug fixing

- Message content was lost because the input `payload.data payload.extension` type of createCustomMessage is incorrect.
- Multiple messages contained in a response to a single request were disordered.
- The unread count could not be cleared occasionally after the read count is reported because the number of unread one-to-one conversions overflows.
- TIM.EVENT.ERROR `event.data.code` and `event.data.undefined` were undefined occasionally.

2.6.2 @2020.4.16

New features

- updateGroupProfile supports muting and unmuting all.
- getGroupMemberList can get the group member muting deadline timestamp muteUntil.

Bug fixing

The unread count could not be cleared when the latest group message was a group prompt.

2.6.1 @2020.4.8

Bug fixing

Files could not be uploaded occasionally when the uploaded COS signature was invalid and not updated in a timely manner.

2.6.0 @2020.3.30

New features

- The web client can create and send video messages createVideoMessage of up to 100 MB in size.
- The `nick` and `avatar` attributes are added in Message to display the nickname and profile photo address of the message sender in an audio-video group (AVChatRoom). The nickname and profile photo address need to be set in advance by calling updateMyProfile.
- Web: when an account logs in on multiple instances, the one-to-one message recall notification can be synchronized across these instances.
- After custom group fields are modified via updateGroupProfile, group members can receive a group notification and obtain the related content: Message.payload.newGroupProfile.groupCustomField

Changes

TIM.EVENT.GROUP_SYSTEM_NOTICE_RECEIVED is deprecated and replaced by MESSAGE_RECEIVED.

Bug fixing

Errors occurred occasionally when the getGroupList API was called.

2.5.2 @2020.3.13

Changes

When searchGroupByID fails, the log level is degraded to warning and the prompt text is modified.

Bug fixing

- Anonymous users or visitors failed to join TIM.TYPES.GRP_AVCHATROOM groups and had statistical problems.
- Other known issues

2.5.1 @2020.3.5

Changes

When login is successful, the key-value pair `repeatLogin: true` is added for the `imResponse.data` callback object to identify repeated login of a login account.

Bug fixing

The priority of messages received at the receiver side of an audio-video group is different from that set on the sender side.

2.5.0 @2020.2.28

New features

- The network status change event TIM.EVENT.NET_STATE_CHANGE is added, and the access side can make related prompts and guidance based on this event.
- Running in WeChat Mini Program plug-in environments is supported.

Changes

Error codes are reduced and optimized.

Bug fixing

- After an audio-video group was created in the console and a group owner was specified, messages sent by other group members will be repeated on the group owner side after the group owner joins the group.
- When groups were created and terminated in the console or using a RESTful API frequently, the SDK did not deliver the TIM.EVENT.GROUP_SYSTEM_NOTICE_RECEIVED event.
- getMessageList failed to get the group message list occasionally.

2.4.2 @2020.2.7

New features

Message priorities, enumerated values, and use cases can be set for group messages.

2.4.1 @2020.1.14

Changes

- Anonymous users or visitors can only join TIM.TYPES.GRP_AVCHATROOM groups.

Bug fixing

- Some online messages could not be pulled occasionally.
- After a system notification from an audio-video group was received, the TIM.EVENT.MESSAGE_RECEIVED event was not delivered.
- In some scenarios, the group message recall result was inaccurate.

- Other known issues

2.4.0 @2020.1.3

New features

- The revokeMessage API is added.
- The `isRevoked` attribute is added for Message. The attribute value `true` identifies recalled messages.
- The message recall event notification TIM.EVENT.MESSAGE_REVOKED is added.
- The following kicked-offline types are added to the kicked-offline event notification TIM.EVENT.KICKED_OUT: Kicked offline due to multi-client login and kicked offline due to UserSig expiration

Changes

- The maximum size of files uploaded through createFileMessage is increased from 20 MB to 100 MB.
- `msgMemberInfo` and `shutupTime` of group prompts will be deprecated. Use `memberList` and `muteTime` instead.

Bug fixing

- Listening events could not be canceled by calling the off API.
- The value and type of the `isRead` attribute in Message were incorrect.
- The error code and error message were incorrect when the video file in a sent video message exceeded the maximum size.
- The content of updated custom fields was incorrect occasionally.
- The JOIN_STATUS_ALREADY_IN_GROUP event occurred occasionally when a user logged in and joined an audio-video group.
- core-js caused potential performance issues.

2.3.2 @2019.12.18

Changes

getUserProfile and updateMyProfile support custom profile fields.

Bug fixing

Messages were lost in combined messages obtained using getMessageList.

2.3.1 @2019.12.13

New features

- createImageMessage and createFileMessage support passing in File objects.
- The createFaceMessage API is added to create emoji messages.
- The message notification efficiency for TIM.TYPES.GRP_AVCHATROOM groups is optimized to improve the user experience.

Changes

- When messages fail to be sent, the SDK returns the actual error codes and error messages.
- When logout is called, only the message channel of the current instance logs out.
- When a callback function passed in by the access side is encapsulated for security purposes and the logic of the callback function is incorrect, errors can be captured and located quickly.
- The SDK provides Chinese error information when IM server-side error codes are received.

Bug fixing

- Messages were lost occasionally when the WeChat Mini Program went to foreground after staying in the background for a long time.
- TIM.EVENT.CONVERSATION_LIST_UPDATED was triggered several times when a message was sent.
- The SDK reported errors when files, such as images were uploaded if registerPlugin was not called or incorrect parameters were entered.
- Long polling did not stop after a TIM.TYPES.GRP_AVCHATROOM group was disbanded.
- When "multi-instance" or "multi-client" login was enabled, other instances or clients failed to receive messages after a web instance was logged out.
- The SDK reported errors occasionally due to the structure of session lists that were pulled.

2.2.1 @2019.11.28

Changes

The logic for getting group roaming messages is optimized.

Bug fixing

- The SDK reported the 2901 error code after the group owner of an audio-video group modified the group profile.
- After the group admin processed apps for joining a group, processed apps can be received after refresh.

2.2.0 @2019.11.21

New features

- Mini Programs support creating and sending video messages createVideoMessage. Video messages can be synced across platforms. Update to the latest versions of the TUIKit and SDK.

- The getGroupMemberProfile API is added.
- Compatible with audio and file messages sent by Native IM v3.x.
- Location messages GeoPayload can be received.

Changes

Up to 100 groups can be written into the local storage. A list containing more than 100 groups is no longer fully written into the local storage.

Bug fixing

- Long polling of TIM.TYPES.GRP_AVCHATROOM groups continues after logout.
- The group contact cards in message instances of TIM.TYPES.GRP_AVCHATROOM groups did not have values.
- Errors were reported when Internet Explorer 10 was used.
- Users could not join groups anonymously.

2.1.4 @2019.11.7

Changes

- When the `Promise` status returned by an SDK API is `rejected`, the SDK no longer delivers a TIM.EVENT.ERROR event.
- Updates to a user's profile are immediately written to the local cache.

Bug fixing

- Code running failed after SDK integration when Angular zone.js modified prototype chains.
- After a group owner created and joined a TIM.TYPES.GRP_AVCHATROOM group, the group owner could not receive messages.
- Initialization failed when the group list was excessively large.

2.1.3 @2019.10.31

Changes

Combined messages (multiple message elements in one message) sent by RESTful APIs or the legacy IM are compatible. For more information, see Compatibility Guide.

Bug fixing

- The unread count was inaccurate.
- Messages were disordered because read messages were not reported.
- Empty image messages could be sent successfully but could not be rendered. The SDK does not support sending empty image messages.

- When an empty file message was sent, the message status was incorrect. The SDK does not support sending empty file messages.
- SDK code errors were reported occasionally when getGroupMemberList was called.

2.1.2 @2019.10.25

New features

getGroupList supports pulling group profile information, including the group owner ID and group member count.

Bug fixing

- SDK code errors were reported when a RESTful API is used to send custom group notifications in an audio-video chat room.
- The SDK did not send a request to pull historical messages when a user re-joined a left group and called the `getMessageList` API.
- SDK code errors were reported when upload failed.

2.1.1 @2019.10.18

New features

Mini Programs support sending audio messages. Audio messages can be synced across platforms. Update to the latest versions of the TUIKit and SDK.

Bug fixing

getMessageList could still pull historical messages in a quit group after rejoining.

2.1.0 @2019.10.16

New features

- Web and Mini Programs support receiving audio messages.
- Web and Mini Programs support receiving video messages.

Changes

- The getMessageList API can pull up to 15 messages at a time.
- TIM.TYPES.MSG_SOUND is deprecated and replaced by TIM.TYPES.MSG_AUDIO.

Bug fixing

- getMessageList could not pull messages in deleted group chats.
- Group system notifications did not contain group names.

- When a conversation was created after receiving a new message, the conversion did not have the profile of the message sender.

2.0.11 @2019.10.12

Bug fixing

Image messages failed to be sent under the React framework.

2.0.9 @2019.9.19

New features

The actual width and height of an image are detected before the image message is sent.

Changes

- The HTTPS protocol is used by default.
- TIM.EVENT.GROUP_SYSTEM_NOTICE_RECEIVED events are sent when new group system notifications are received.

Bug fixing

- Screen splash occurred when mini programs sent image messages.
- JPG or other images failed to be sent.

Update Log (Native)

Last updated : 2021-08-17 18:05:37

Latest Enhanced Version 5.5.892 @2021.07.14

SDK

- Added support for message search by multiple keywords combined with AND or OR.
- Added support for message search by a specified message sender account.
- Added support for historical message pulling by a specified time range.
- Added support for historical group message pulling by a specified sequence.
- Added notifications for message modifications by a third-party callback.
- Added the API for getting the maximum number of group members that can be added to a group.
- Added the `orderKey` field for sorting conversation objects to facilitate sorting conversations without the last message at the app layer.
- Optimized the audio-video group message receiving latency by making the backend complete account conversation in advance.
- Upgraded the network connection scheduling protocol to reduce the network connection time outside the Chinese mainland.
- Optimized the conversation list pulling logic.
- Optimized the group member pulling logic and enabled local cache.
- Fixed the issue where log callback was not triggered when the log level was lower than Debug.
- Fixed the issue where group member profiles obtained did not include friend remarks.
- Fixed the issue where the obtained list of groups the user has joined contained groups to be approved by the group owner.
- Fixed the stability issue reported online.

Latest Basic Version 5.1.62 @2021.05.20

SDK

- Fixed known issues.

5.4.666 @2021.06.03 - Enhanced Version

SDK

- Changed the name of Lite Edition SDK to Enhanced Edition SDK.
- Added support for message, group, and friend search.
- Added a parameter to specify whether to update the last message of the conversation during message sending.
- Added support for clearing the roaming messages of a conversation while retaining the conversation.
- Added support for concurrent multi-device login on the same platform.
- Reduced the time for network connection and login.
- Optimized the data reporting feature.
- Optimized the offline push logic to support disabling offline push globally.
- Optimized the offline push logic to allow setting the message classification field `classification` for vivo phone offline push.
- Fixed the occasional incorrectness of the unread message count of one-to-one conversations.
- Optimized the historical message pulling speed.
- Added support for adding emojis and locations to multi-element messages.
- Fixed the issue where, if an offline user changed the user's nickname in a group, the user's nickname in the corresponding conversation was not updated in a timely manner when the user logged in the next time.
- Fixed the issue where the 20005 error code was occasionally reported when read messages of one-to-one conversations were reported.

5.3.435 @2021.05.20 - Lite Edition

SDK

- Added the API for deleting conversation roaming messages.
- Fixed the issue where some Android phones could not receive network status change notifications over persistent connections.
- Optimized the logic for pulling user profiles to avoid requesting the backend every time when strangers request for user profiles.
- Fixed the issue where group profiles and historical messages could not be obtained when the groups were deleted but conversations were retained.
- Fixed the issue where conversations were out of order when you got them via the API for getting a conversation list.
- Fixed the issue where group conversations in Mute Notifications mode were filtered out when getting the total message unread count.
- Fixed the occasional crashes caused by iOS HTTP requests.

5.3.425 @2021.04.19 - Lite Edition

SDK

- Added support for pinning a conversion on top.
- Added support for setting the Mute Notifications option for one-to-one messages.
- Added support for sending messages that are excluded from the unread count.
- Added support for getting local conversation and message data when there is no network connection or your login fails.
- Added XCFramework (supporting Mac Catalyst) to the SDK for iOS.
- Added the API for getting the conversation unread count.
- Added the `birthday` field to personal profiles.
- Fixed the issue where, when group @ messages were recalled, the conversations of the @ target users still contained the group @ notification.
- Fixed the issue where, for some Android phones, the network would be disconnected and connected again after a successful initial network connection during persistent connections.
- Fixed the issue where users could not set custom fields when creating a group in the SDK for iOS.
- Fixed the issue where users with special accounts could not search for local messages via `findMessage`.

5.1.60 @2021.04.06 - Standard Edition

iOS

- Fixed the issue where the SDK may be rejected by the App Store for using IDFA related keywords.

5.2.212 @2021.04.06 - Lite Edition

iOS

- Fixed the issue where the SDK may be rejected by the App Store for using IDFA related keywords.

5.2.210 @2021.03.12 - Lite Edition

SDK

Common changes

- Added support for forwarding multiple messages as a combined single message.

- Optimized the logic of persistent connections, improving the quality of connections outside the Chinese mainland.
- Specified login error codes in a detailed way to distinguish whether the network is normal during login.
- Optimized the logic of COS upload, providing better experience of sending rich media messages.
- Added the advanced API for getting historical messages.
- Added the API for getting conversations in batches.
- Added the API for checking friend relationships in batches.
- Fixed the issue where two messages were generated in the local database after a message that failed to be sent was sent again.
- Fixed the issue where the muting time called back was incorrect when the group member profile was changed.
- Fixed the issue where the width of the image called back was incorrect when an image message was received.
- Fixed the issue where the console still printed logs after `logLevel` was set to `None`.
- Fixed the issue where the `add_source` field of adding friends was incorrect.
- Fixed the issue where sometimes the sending progress called back was negative when a video file greater than 24 MB was sent.

5.1.56 @2021.03.03 - Standard Edition

SDK

Common changes

- Optimized the logic of persistent connection, improving the quality of connections outside the Chinese mainland.
- Optimized data reporting and specified error codes related to network timeout in a detailed way.
- Fixed known stability issues.

iOS

- Fixed occasional failures of extracting logs in the iOS SDK.

Android

- Replaced the log component of the Android SDK to improve stability.

Windows

- Fixed the issue where the client thread might block the SDK logic thread when a new message callback was triggered in the Windows SDK.

5.1.138 @2021.02.05 - Lite Edition

SDK

Common changes

- Optimized logging.
- Optimized the policy of persistent connection, improving the quality of connections outside the Chinese mainland.
- Fixed the issue where sometimes the last message was incorrect when multiple one-to-one messages were sent or received in the same second.
- Fixed the issue where sometimes there was no callback for querying the conversation list.
- Fixed the issue where sometimes the sequence number of a one-to-one message was incorrect.

Android

- Fixed the issue where sometimes a negative upload progress was displayed when a video greater than 24 MB was sent.
- Fixed occasional crashes when messages were sent.

5.1.50 @2021.02.05 - Standard Edition

SDK

- V2 APIs added the `random` field for message objects.
- Added support for recalling the `lastMsg` message in a conversation.
- Fixed occasional exceptions in the status of the last message obtained via the `getMessage` API.
- Fixed the issue where messages were delayed when user profiles were frequently pulled after messages were received.
- Fixed the issue where deleting the account might cause the failure to pull the group member list.
- Fixed the issue where the message might not be found when `findMessage` was called after `insertLocalMessage`.
- Fixed the issue where a conversation update callback was triggered when a conversation was deleted.
- Fixed the issue of the Android SDK where the nicknames of historical group messages were not timely updated.

- Improved the database stability of the iOS SDK.

TUIKit and demo

- Fixed the issue of the Android TUIKit where a black screen was displayed when you tried to view the original images that were not downloaded.
- Fixed the internationalization issue of the iOS version.
- Fixed the issue of the iOS version where images were overwritten when multiple images were sent at a time.
- Fixed the issue of the iOS 14 operating system where there was no response when you clicked the "add" or "delete" button on the group details page.
- Fixed the issue of the iOS 14 operating system where the tab bar disappeared after you left a group conversation and went back to the message list.

5.1.21 @2021.01.15 - Standard Edition

SDK

Android

- Fixed the issue where custom messages with the extended field `extension` failed to be sent on the Android platform.

TUIKit and demo

iOS/Android

- Improved internationalization support by eliminating the issue where there were Chinese characters in the English version.

5.1.137 @2021.01.29 - Lite Edition

SDK

Common changes

- Fixed the issue where there was no callback for the login API occasionally when a user logged in to the same account repeatedly on multiple iOS devices or Android devices.

Android

- Fixed crashes that occurred occasionally when a low-end Android device tried to obtain the log path.

5.1.136 @2021.01.27 - Lite Edition

SDK

Common changes

- V2 APIs added an API for log callbacks.
- Fixed the issue where the UserID of the @ target user in the group @ message was empty.
- Fixed the issue where audio-video group messages occasionally could not be received.
- Fixed the occasional issue of incorrect login status in the case of frequent network reconnection.
- Fixed the issue where users occasionally failed to log in again after going offline and being kicked off.
- Fixed occasional crashes in DNS resolution.

5.1.132 @2021.01.22 - Lite Edition

SDK

Common changes

- Added support for overload protection in the network module.
- Fixed the issue where some sessions occasionally were lost when the standard edition was upgraded to the Lite Edition.
- Fixed the issue where the `onUserSigExpired` callback could not be received after the login information expired.
- Fixed the issue where a member received the `onMemberKicked` callback after being kicked out of a group and joining the group again.

5.1.131 @2021.01.19 - Lite Edition

SDK

Common changes

- Added the API for forwarding a single message.
- Optimized the logic of receiving audio-video group messages. When an audio-video group receives a message, the sender’s nickname and profile photo are no longer queried.

- Fixed the issue where there was no conversation update notification when the last message in a conversation was deleted.
- Fixed the issue where the unread messages count in one-to-one conversations occasionally was cleared when the one-to-one messages were synchronized after login.
- Fixed the issue where the last message in a conversation was not updated when the conversation list was synchronized after a user went offline and then online.

Android

- Fixed the issue where the settings of the custom message field `description` and personal profile fields `level` and `role` did not take effect.
- Fixed occasional crashes during deinitialization.

5.1.129 @2021.01.13 - Lite Edition

SDK

Common changes

- Fixed the issue where a conversation update callback was triggered when a user tried to get the conversation list and there was no conversation update.
- Fixed the issue where the last message in a conversation was not cleared when a user tried to delete all the messages in the conversation.

iOS

- Fixed the issue where the returned information was not `nil` when a non-signaling message was passed in using the `getSignallingInfo` method.

Android

- Fixed occasional crashes caused by JNI local reference table exceeding the limit.

5.1.20 @2021.01.08 - Standard Edition

SDK

Common changes

- V2 custom messages added the `desc` and `ext` fields.
- V2 user profile APIs added the `role` and `level` fields.

- Optimized V2 APIs. Whether your login is successful or not, you can get the data of the local conversation list and local historical messages.
- V2 added the `getHistoryMessageList` API to support getting cloud or local messages and getting messages sent before or after a specific time.
- Optimized the issue in getting the profile photos of one-to-one messages.
- Optimized the security and renewal of rich media message file upload.
- Fixed the issue where the local paths of sent rich media messages were empty.
- Fixed the issue where when a local message was inserted into a group, the previous message was displayed as the `lastMessage` of the conversation after you logged out and logged back in.
- Fixed the Elem out-of-order issue.
- Fixed the issue where the @ prompt still existed in the message list after the group @ message was recalled.
- Fixed the issue where system messages were pulled when you pulled the offline historical group messages after going online.
- Fixed the issue where two offline push notifications were received when only one signaling invitation for a voice call was sent.
- Fixed the issue where the settings of local "custom message data" became invalid when there were too many messages.
- Fixed the issue where the unread number did not decrease after an unread group message was recalled.
- Fixed other stability issues.

iOS and Mac

- Fixed receiver crashes that occurred when `array json` was passed for custom messages.
- Fixed crashes after calling `deleteConversation` and passing the wrong conversation ID.
- Fixed the issue where the last draft in the draft box could not be deleted.

TUIKit and demo

- Fixed the issue where the information of the conversation pinned to the top was not deleted after you deleted the friend or left the group on the iOS platform.
- Fixed the issue where after a user was set as the administrator, the console still showed that the user did not have the administrator permissions on the iOS platform.
- Fixed crashes that occurred when the thumbnail was empty on the iOS platform.
- Fixes the issue where there was an exception in the height of a recalled long-text message on the iOS platform.
- Fixed the issue where group muting tips were not displayed on the iOS platform.
- Optimized the time display of the conversation UI on the iOS platform.

- Fixed crashes that occurred when a user clicked **Back** after the creation of a live room entered the countdown process on the Android platform.
- Fixed the issue where the call interface did not disappear when a member refused to answer the call in a group chat on the Android platform.
- Fixed the issue where the small window was not closed when a viewer in the live room was kicked offline in the small window mode on the Android platform.
- Fixed occasional crashes that occurred when someone joined a group on Android devices.

5.1.125 @2021.01.08 - Lite Edition

SDK

Common changes

- V2 APIs added the `random` field for message objects.
- V2 APIs added the `description` and `extension` fields for custom messages.
- V2 APIs added the `role` and `level` fields for user profile objects.
- Fixed the database compatibility issue in the upgrade from versions below 4.8.1 to the Lite Edition.
- Fixed the issue where users occasionally received the callbacks of messages sent by themselves.
- Fixed the issue where there was no callback when you tried to get the list of groups that you joined when you hadn’t joined any group.
- Fixed the issue where there was no conversation update callback when setting group message receiving options.
- Fixed the issue where occasionally there was no end callback for conversion synchronization.
- Fixed occasional crashes during conversion synchronization.

5.1.123 @2020.12.31 - Lite Edition

SDK

Common changes

- Fixed the issue where the Android edition could not receive custom group system messages sent via the RESTful API.
- Optimized the method of generating the value of the `random` field for a message.
- Optimized log printing to facilitate troubleshooting.
- Fixed occasional crashes in the network module.

5.1.122 @2020.12.25 - Lite Edition

SDK

Common changes

- Fixed the issue where there might be no callback when setting conversation drafts.
- Fixed the issue where the message sender information was not completed when searching for messages via `findMessage` .
- Fixed the issue where it might fail to search for messages via `findMessage` after inserting local messages.
- Fixed the issue where conversation objects were not updated when setting group message receiving options.
- Fixed the issue where conversation change notifications were not sent when personal or group nicknames or profile photos were changed.
- Fixed the issue where the last message in a conversation was not updated when inserting local messages.
- Enabled on-cloud control over personal profile update cycle.

iOS

- Fixed occasional crashes caused by improper dictionary or array operations.

Android

- Fixed occasional crashes when deleting messages.

5.1.121 @2020.12.18 - Lite Edition

SDK

Common changes

- Optimized the group profile pull logic. For audio-video groups, users’ own group member information does not need to be pulled.
- Improved log printing and added the device type field.
- Fixed the issue where, when a message recall notification was received in a one-to-one conversation, the status of the last message in the conversation was not updated.
- Fixed the issue of excessive message delay during long polling in an audio-video group.
- Fixed the issue where, when a user logged in to the same account repeatedly and then joined the same audio-video group, the message long polling module did not update the message pull key.

iOS

- Fixed the issue where, when a JSON array was passed in for custom message fields on iOS, the signaling module on the receiving end crashed during parsing.

Android

- Fixed occasional crashes when setting conversation drafts.

5.1.118 @2020.12.11 - Lite Edition

SDK

Common changes

- Optimized the message deduplication logic and fixed the issue where repeated callbacks were triggered for the same message.
- Added an API for local insertion of one-to-one messages.
- Fixed the issue where the unread group message count did not decrease when unread group messages were deleted or recalled.
- Fixed the issue where messages that failed to be sent could not be deleted.
- Fixed the issue where the deletion failure callback was triggered when a user attempted to delete a conversation in a group that the user had left or a group that had been deleted.
- Fixed the issue where the setting failure callback was triggered when a user attempted to enable group message read reports for a group that the user had left or a group that had been deleted.

iOS

- Fixed the issue where setting the signature in personal profiles failed.

Android

- Fixed the issue where adding a friend to a blocklist occasionally led to crashes.
- Fixed the issue where no message ID was returned when a message was sent.

5.1.10 @2020.12.04 - Standard Edition

SDK

Common changes

- V2 APIs added support for custom group fields and multi-element messages.

- V2 APIs added an API for the local insertion of one-to-one messages.
- Mitigated the issue of message loss for ordinary groups and audio-video groups.
- Fixed the issue where messages that failed to be sent could not be deleted.
- Fixed the one-to-one conversation issue where, if the first message was sent online, the read receipt was not received.
- Fixed the issue where, after a recalled message was returned through the API for pulling historical messages, the message status was incorrect.
- Fixed the failure to return information of all friend lists when `null` was entered as the friend list name for the API for obtaining friend list information on iOS.
- Fixed known stability issues.

5.1.115 @2020.12.04 - Lite Edition

SDK

Common changes

- Optimized synchronization between the signaling timeout threshold and server time.
- Fixed occasional failures in establishing connections on a weak network.

iOS

- Completed API header files.

Android

- Fixed crashes by replacing Gson with JSON.

5.1.111 @2020.12.01 - Lite Edition

SDK

Common changes

- Improved log printing.
- Fixed known stability issues.

5.1.2 @2020.11.11 - Standard Edition

SDK

iOS and Mac

- iOS allows iPhones and iPads to be online at the same time.
- Mac supports the ARM64 architecture.

Android

- Fixed a stability issue in the Android edition.
- Substituted the standard TRTC dependency package.

5.1.110 @2020.11.26 - Lite Edition

SDK

Common changes

- Added all V2 APIs.
- Added the conversation feature.
- Added the relationship chain feature.
- Added the group @ feature.
- iOS allows iPhones and iPads to be online at the same time.
- Added support for multi-element message sending.
- Supplemented custom fields in group profiles.
- Fixed known stability issues.

5.1.1 @2020.11.05 - Standard Edition

SDK

iOS/Android

- Added an API to obtain the number of online users in an audio-video group (AVChatRoom).
- Added an API to query messages based on the unique ID.
- Added an API to obtain the server calibration timestamp.
- Optimized the login speed.
- Optimized the group profile pull logic.
- Fixed the issue where pulling local messages failed after users left a group.
- Fixed the issue where, after a successfully sent message was modified by a third-party callback, the message on the sender end was not promptly updated.

- Fixed the issue where, after configuration via the console, conversations of meeting groups still did not support unread counts.
- Fixed the issue where users in an audio-video group (AVChatRoom) occasionally failed to receive messages.
- Fixed some other occasional stability issues.

TUIKit and demo

iOS/Android

- Group members can input `@All` .
- TUIKit components added international support.
- Added support for selecting videos when sending image messages through the Android edition.
- Optimized the timeout logic for voice and video call requests.
- Updated Android offline push to be dependent on the TPNS package.
- Group live streaming added an opening animation.
- Group live streaming added support for a small livestreaming window.

5.0.108 @2020.11.02 - Lite Edition

SDK

Common changes

- Fixed a stability issue in the iOS edition.
- Fixed the occasional message callback failures in the Android edition.

5.0.10 @2020.10.15 - Standard Edition

SDK

iOS/Android

- Optimized signaling APIs to support the setting of `onlineUserOnly` for online messages and `offlinePushInfo` for offline push messages.
- Optimized the async callback for the API for obtaining a single conversation.
- Added an API for obtaining group types for conversations to facilitate display filtering of the conversation list.

TUIKit and demo

iOS/Android

- Added group livestreaming features, such as co-anchoring, gifts, beauty filter, and voice changing.
- Added live rooms that support co-anchoring, PK, likes, gifts, beauty filter, on-screen comments, following friends, and other features.
- Optimized the recognition of audio and video signaling.

5.0.106 @2020.09.21 - Lite Edition

SDK

Common changes

- Fixed known stability issues.

5.0.6 @2020.09.18 - Standard Edition

SDK

Common changes

- Added the group @ feature.
- Added the `deleteMessages` API for iOS and Android, which will simultaneously delete local and roaming messages.
- When deleting a conversation, the `deleteConversation` API also deletes local and roaming messages.
- API2.0 added APIs for setting and obtaining custom fields for user profiles, friend profiles, and group member profiles.
- Optimized image upload compatibility issues.
- Fixed the issue where after the group message receiving option was modified and then immediately obtained, the option remained unchanged.
- Fixed the issue where after a local C2C conversation was deleted, C2C system notifications updated the conversation but the message `elem` was empty.
- Fixed the issue where image upload failed when the userID contained Chinese characters.
- Fixed the issue where after an account with special characters successfully set the user nickname and entered the group to send a message, the nickname was still blank in the new message callback received by other group members.
- Fixed known crashes.

iOS

- Fixed the crash issue that occurred when message listening was removed.
- Fixed the issue where deleting a conversation peer account led to exceptions in obtaining the conversation.
- Mitigated the issue of initialization lag.

Android

- Optimized the processing for signaling sending timeout failure.
- Fixed the issue of invalid custom data for the signaling cancellation API.
- Fixed the issue where attempts to delete all attributes failed when `null` was passed in for the `keys` of the group attribute deletion API.
- Fixed the issue where signaling group calls could still be accepted or rejected after being accepted or rejected.
- Fixed the multi-element resolution issue for API 2.0.

Windows

- Fixed the known issue of memory leak.
- Optimized log upload.
- Fixed the issue where a user who simultaneously logged in to the same account from multiple PCs of the same model was not forced offline.
- Fixed the issue where received messages were out of order on a PC.

TUIKit and demo

iOS

- Added the group @ feature.
- Added new emoji packs.
- Updated the SDWebImage dependent library.
- Optimized UI display for applications to join a group.
- Optimized the text display of voice and video calls.

Android

- Added the group @ feature.
- Fixed the issue where the contacts displayed during group creation might be inconsistent with those actually selected.
- Fixed the issue where the display of custom messages might be out of order.
- Fixed occasional crashes of AVCallManager and TRTCAVCallImpl.
- Added new emoji packs.

5.0.102 @2020.09.04 - Lite Edition

SDK

Common changes

- Released the Android & iOS Lite-Edition SDK.
- Compared with the standard edition SDK, the Lite Edition SDK removed the friend and conversation capabilities and optimized some service logic to ensure higher execution efficiency and a smaller installation package size.

4.9.1 @2020.07.24 - Standard Edition

SDK

Common changes

- Optimized login outside the Chinese mainland.
- Fixed file upload failures in some regions outside the Chinese mainland.
- Fixed file upload failures for accounts containing the @ symbol.
- Fixed occasional errors with unread count of one-to-one messages.
- Fixed occasional exceptions in conversation `showName` display.
- Added an API for obtaining the download URL of file messages.

iOS

- Fixed the issue where there was no callback when users attempted to obtain one-to-one messages while network connection was not available.

Android

- Fixed occasional crashes of signaling parsing APIs.
- Fixed occasional crashes when obtaining offline push information.
- Fixed the issue of no callback when API 2.0 `getFriendApplicationList` carried no data, and fixed the issue of no callback when non-members were specified for `getGroupMembersInfo`.

Windows

- Added detailed group information when users obtain the list of groups joined.
- Fixed the failure to send small files.
- Fixed error 6002 reported by logs.

TUIKit and demo

iOS

- Added push of offline voice and video calls and enabled redirection to the call answering interface.
- Fixed failure to delete or recall custom messages.
- Optimized the interface.
- Migrated the voice and video code from Swift to Objective-C to substantially reduce third-party dependent libraries.
- Added support for TUIKit pod integration of two types of voice and video dependent libraries: LiteAV_TRTC and LiteAV_Professional.

Android

- Optimized the offline push of the demo and upgraded the push SDK version for each vendor.
- Added push of offline voice and video calls and enabled redirection to the call answering interface.

4.8.50 @2020.06.22 - Standard Edition

SDK

Common changes

- Fixed the API 2.0 issue where the `onMemberEnter` callback was not triggered when someone entered an audio-video group (AVChatRoom).
- Added the `groupID` parameter to the `onGroupInfoChanged` and `onMemberInfoChanged` callbacks of API 2.0.
- Fixed the issue where there was no conversation update callback after a one-to-one message was sent successfully.
- Fixed the issue where a user failed to receive messages after switching accounts and joining the same audio-video group (AVChatRoom).
- Fixed the occasional issue of incorrect callback sequence during unread message synchronization after login.
- Adding signaling APIs.
- Added the custom group attribute API for audio-video groups (AVChatRoom).
- Fixed known crashes.

Android

Changed the default log storage location to `/sdcard/Android/data/package name/files/log/tencent/imsdk` to be compatible with Android Q versions.

Windows

Fixed group member role issues during group creation.

TUIKit and demo

iOS

- TUIKit replaced API 2.0.
- Integrated TRTC to realize the voice and video call feature.
- Added the deep-color mode.

Android

- TUIKit replaced API 2.0.
- Integrated TRTC to realize the voice and video call feature.
- Supports AndroidX.

4.8.10 @2020.05.15

SDK

Common changes

- iOS and Android support IPv6.
- Audio-video groups (AVChatRoom) support dynamic updates of the group member list.
- Fixed xlog crashes.

iOS and Mac

- Fixed the failure of iOS to send big files.
- Fixed the exceptions that occurred when `getFriendRemark` was triggered to fetch the sender's friend remark in a V2TIMMessage message.

Android

- IM SDK supports AndroidX.
- Fixed the crashes of Android devices caused by network permission issues.

4.8.1 @2020.04.30

SDK

Common changes

- Launched brand-new API 2.0 for iOS & Android.
- Fixed conversation errors when users logged in to different accounts in certain scenarios.

4.7.10 @2020.04.23

SDK

Common changes

- Fixed login timeout in some network environments.
- Fixed inaccurate unread counts in some scenarios.

4.7.2 @2020.04.03

SDK

Common changes

Fixed a data error.

4.7.1 @2020.03.23

SDK

Common changes

- Optimized the local log size.
- Optimized the login time.
- Fixed the multi-terminal unread count synchronization issue.
- Added the `getFriendList` API.
- The iOS and Android SDKs enable you to set the message title and content to display on the offline push notifications bar of iOS and Android devices, respectively.

4.6.102 @2020.02.28

SDK

Common changes

- Fixed slow message pulling in some scenarios.
- Fixed the compatibility issue with sending 3.x version audio messages to later versions.
- Fixed the issue where the identifiers of some conversions in the obtained conversion list were null.
- Fixed known crashes.
- Fixed SOCKS5 proxy users' password verification issue.
- Optimized the pending group processing logic.
- Improved the file upload limit to 100 MB.
- Optimized COS upload.
- Fixed the issue where an exception was returned for obtaining the friend list if there was no friend.

4.6.56 @2020.01.08

SDK

Common changes

- Mitigated the issue where memory grew when user profiles were frequently pulled.
- Improved compatibility with special characters in user profiles.
- Fixed known crashes.
- Fixed occasional login failures when accounts are switched frequently.
- Fixed reconnection in the pressure test.

4.6.51 @2019.12.23

SDK

Common changes

- Improved network connection quality to quickly detect network quality changes.
- Optimized audio-video group message handling.

iOS and Mac

- Changed all IMSDK listeners from strong references to weak references of external objects.
- Added the `getSenderNickname` API for messages.

Android

- Fixed the issue where offline users are kicked off.
- Fixed exceptional upload progress callback on devices running earlier Android versions.

- Fixed memory leak during login.
- Added the `getSenderNickname` API for messages.

Windows

- Fixed the issue where messages failed to be sent to newly added friends.
- Improved modification and query of custom fields for group information and group member information.
- Improved callbacks for all APIs to ensure that callbacks will be called and that objects are transferred to JSON strings only when callbacks succeed and empty strings are returned when callbacks fail.

TUIKit and demo

Android

- Profile photos displayed in conversation lists can be set with rounded corners.
- Fixed the issue where account switching is exceptional when a conversation is pinned to the top.

4.6.1 @2019.11.13

SDK

Common changes

- Roaming messages can be recalled.
- Fixed the unread count error when a user was invited to join a group in silent mode through a RESTful API.
- Fixed occasional message sending exceptions due to poor network connection.
- Fixed incorrect logic for role filter conditions when group members are obtained.
- Fixed the issue where the SDK failed to get the group name the first time users sent a message in a group created by a RESTful API.
- Fixed the issue where `getUsersProfile` failed to get user information after caching was disabled.
- Fixed the issue where voice message files without a suffix could not be downloaded after they were received.

iOS and Mac

- Added OPPOChannelID settings to fix the issue where OPPO mobile phones running Android 8.0 or later failed to receive iOS push messages.
- Optimized annotations to `getGrouplist` return objects.

Android

- Offline pushed channelID on OPPO mobile phones running Android 8.0 or later can be configured in the console.
- The ext, sound, and desc fields of `TIMCustomElem` have been deprecated.

Windows

- Fixed the exceptional type field of group system messages.
- Fixed inconsistent group type and header file in the returned group information.
- Fixed the issue where specifying custom group fields failed during group creation.
- Added sender profile and offline push configuration to messages.

TUIKit and demo

iOS

- Added the video call feature.
- Added 3x3 grid display of group profile photos.
- Optimized the conversation list, contacts, and chat UIs.

Android

- Added a method to set whether to display read receipts.
- Added 3x3 grid display of group profile photos.
- Optimized the conversation list, contacts, and chat UIs.
- Fixed compatibility issues with the input method, UI, and file selection for some mobile phones.
- Fixed messy display of custom messages.
- Fixed slow contact loading in the stress test.
- Fixed the conflicts with other library resources.
- Fixed ineffective cache directory settings.

4.5.111 @2019.10.16

SDK

Common changes

- Fixed the paging issue of the API used to get the list of group members of a specified type.
- Added file format extension to the URL generated upon sending a file message.
- Added the notification callback after custom group fields are modified.

- Local user and group information can be obtained before login by calling the `initStorage` method.
- Fixed the memory leak issue.
- Fixed the issue with incorrect message status codes after sent messages are recalled.
- Fixed the issue with incorrect `getMessage` callback error codes.
- Fixed incorrect one-to-one chat unread count after an app is killed and restarted.

iOS and Mac

Fixed occasional login failures for sleeping Mac devices.

Android

- Fixed stability issues in some scenarios.
- Fixed the issue where OPPO mobile phones running Android 8.0 or later could not receive offline push notifications.
- Optimized the return types of the `getElementCount` API.

Windows

- Improved the network reconnection speed for cross platform libraries.
- Fixed the Windows public group management setting failure.
- Added JVM configuration to cross-platform libraries to facilitate passing jvm from an Android environment.

TUIKit and demo

iOS

- Added support for sending and receiving voice messages to and from web applications.
- Fixed the issue where TUIKit resource files could not be found when swift loading.
- Fixed the issue where a friend's alias could not be seen on the chat interface after it was modified.
- Fixed the issue where the conversation list did not refresh promptly after a conversation was pinned to the top.

Android

- Added support for sending and receiving voice messages to and from web applications.
- Added support for setting the input box style.
- Displayed a red dot on unread voice messages.
- Fixed the issue where video messages could not be played on x86 devices.
- Fixed conflicts between FileProvider and the integration side.

- Fixed the issue where audio permissions could not be identified on some mobile phone models.
- Fixed the issue where the profile photo cannot be loaded in specific conditions.
- Fixed occasional incomplete display of bubbles.

4.5.55 @2019.10.10

SDK

Common changes

- Fixed crashes when networks are switched multiple times.
- Improved network connection quality.
- Optimized annotations of some APIs.

Android

Optimized HTTP request restrictions on Android 9.0 or later.

iOS and Mac

Optimized pod integration.

4.5.45 @2019.09.18

SDK

Common changes

- Improved network connection quality.
- Fixed the exceptional unread count when new messages are received after a group chat is deleted.
- Fixed the issue where deleted conversations could still be obtained from the conversation update callback.
- Optimized the logic for pulling custom group/group member fields.

Android

Deprecated the `setOfflinePushListener` API and `TIMOfflinePushNotification` class in `TIMManager`.

TUIKit and demo

iOS

- Fixed the NSSting + Common.h class conflict issue.
- Fixed the incomplete group tip display issue.

Android

- Added read receipts.
- Compatible with typing display in earlier versions.
- Fixed the issue where resent messages failed to immediately appear at the bottom of the chat window.
- Fixed the issue where profile photos in a group chat failed to be displayed under specific conditions.
- Fixed the issue where multi-element group messages could not be displayed.
- Fixed crashes caused by specific messages.
- Fixed the group admin permission error.
- Fixed the issue where files sent by web applications could not be received.

4.5.15 @2019.08.30

SDK

Common changes

- Improved the speed of sending file messages for users outside the Chinese mainland.
- Fixed the issue where the message status fetched by `getLastMessage` was incorrect after a message was recalled. Fixed the issue where the callback is called multiple times after message listening was recalled.
- Fixed the issue where the backend failed to obtain the muting time after a member is muted, left the group, and joined the group again.
- Fixed the issue where the message time was ineffective during `savemsg` after the message time was proactively modified.
- Fixed the issue where no callback occurred occasionally upon login.
- Fixed the issue where `rand` and `timestamp` of a recalled group message were empty.
- Fixed the issue where UserSig in a callback expired when the user was logged out. Fixed the issue where reconnection continued when the user was logged out.

Android

- Added support for FCM push notifications on Android devices in the backend.
- Fixed the issue where an error was reported when null was passed for getting a specified friend list.

- Fixed checkEquals crashes in specified scenarios.

Windows

- Added the `unique_id` field to `MessageLocator`.
- Added support for 64-bit Windows.
- Added user profile APIs and relationship chain APIs to the cross-platform library.

TUIKit and demo

iOS

- Added support for sending custom messages.
- Added read receipts for one-to-one messages.
- Added a red dot to unplayed audio messages.

Android

- Fixed the demo memory leak issue in some scenarios.
- Fixed crashes in some scenarios.
- Fixed the incorrect custom message color issue.
- Fixed the incorrect or incomplete bubble display issue.
- Fixed the issue where conversation lists failed to display profile photos.
- Fixed the issue where the title bar color could not be changed by ConversationLayout.
- Added support for 64-bit ijkplayer.
- Added support for multi-element messages.

4.4.900 @2019.08.07

SDK

Common changes

- Fixed stability issues in some scenarios.
- Optimized the unread message count.
- Improved the latest conversation list loading speed after login.
- Added the log cleaning feature.
- Fixed message loss when synchronizing a large number of unread one-to-one messages.
- After a user leaves an audio-video group, system messages about members leaving the group will not be pushed to the user's device.
- Fixed the issue where group system messages occasionally failed to be delivered to users.

- Added the frequency limit logic to `onRefresh/onRefreshConversations`.
- Optimized exceptional `saveMessage` ordering.

iOS and Mac

- Changed the `getGroupInfo` callback parameter to `TIMGroupInfoResult` to fetch the error codes corresponding to each group.
- Optimized the display style of push notifications for 4.x versions to keep consistency with 2.x and 3.x versions.
- Fixed the issue where login accounts that contain Chinese characters failed to send images, files, and videos.

Android

- Fixed the issue where mobile phones running the 4.2.2 system version failed to load so.
- Fixed the issue where `getGroupInfo` returns an incorrect amount of data.
- Changed the `getGroupInfo` callback parameter to `TIMGroupDetailInfoResult` to fetch the error codes corresponding to each group.
- Used the `com.tencent.imsdk.TIMGroupReceiveMessageOpt` class in a unified manner.

Windows

Fixed the issue where the Windows configuration file path is garbled.

TUIKit and demo

iOS

- Modified the iOS demo UI, including the default profile photo and four feature icons (camera, video, album, and file) on the input interface.
- Added the profile card to "Me" and put personal information in the profile card.
- Added the feature to view the large image by tapping the profile photo.
- Modified the style of the small gray bar in group chats in the demo so that the member nickname becomes blue and tapping the nickname will redirect the member to the member's profile page.
- Optimized the logic for displaying nicknames in groups in the demo.
- Optimized the logic for displaying profile photos on the chat interface.
- Added tap feedback to all interfaces, allowing users to set and customize feedback in TUIKit.

Android

- Added `MotionEvent.ACTION_CANCEL` event handling for audio messages in chats.
- Added profile photo display in the conversation list, chat interface, detailed profile, and contacts.

- Added profile photo change in user profiles.
- Added Intent redirection to offline push functions.
- Added random profile photos for one-to-one chats and group chats.
- Added prompts for granting and revoking the group admin role for a group member.
- Added prompts for muting and unmuting group members.
- Fixed the issue where the text "You've recalled a message" was not displayed in tips after a message was recalled.
- Fixed the issue where the content of a recalled message was always displayed as the last message in the conversation list.
- Fixed the white screen issue on the chat interface after offline messages were received on Meizu mobile phones.
- Fixed the issue where the chat conversation pinned to the top did not update to the last message when new messages came in.
- Fixed Toast notifications when the username or password is empty.
- Fixed the issue where GroupTips messages transferred from the group owner were displayed abnormally in TUIKit.
- Fixed the Didn't find class "android.support.v4.content.FileProvider" error reported on some mobile phones.
- Optimized the logic for pinning a chat to the top to arrange chats in chronological order starting from the most recent.
- Fixed the issue where the soft keyboard and other layouts appeared in chats at the same time.
- Fixed the issue where the Group Chats, Blocklist, and New Contacts items were not displayed on the Contacts interface when a user is newly registered with no contacts.
- Fixed the issue where the video sound continued to play after a user taps the Back button on a mobile phone.
- Fixed the issue where the playing voice message did not stop and its sound was also recorded during voice message recording.
- Fixed the issue where videos sent by iOS devices failed to playback on some mobile phones.

4.4.716 @2019.07.16

iOS and Mac

- Organized and merged APIs.
- Added APIs to get the download URLs of file, video, and voice messages.
- Added the `disableStorage` API to disable all local storage.
- Fixed the issue where the conversation on the sender's device could still get lastMsg after an online message was sent.

- Removed the return value of `getSenderProfile`, and used callback instead.
- Changed the group function `modifyReciveMessageOpt` to `modifyReceiveMessageOpt`.
- Fixed the issue where video screenshots sent from a device running iOS 2.X or 3.X to a device running iOS 4.X could not be obtained.
- Fixed occasional crashes when data was reported upon exit.
- Optimized the login module (repeated login/frequent login/frequent account switching/automatic connection/offline user being kicked off).
- Fixed the issue where the unread count could not be cleared after a member left a group or a group was deleted.
- Fixed the issue where group deletion notifications could not be received occasionally.
- Fixed the issue where longer time was required to deliver messages when the app went to the foreground after staying in the background for a long time.
- Optimized the one-to-one chat unread count.
- Changed the input parameter `TIMLoginParam` of `autoLogin` to `userID`.
- Changed the input parameter `TIMLoginParam` of `initStorage` to `userID`.
- Removed multi-account login APIs: `newManager`, `getManager`, and `deleteManager`.
- Fixed occasional respondsToLocator crashes.
- Fixed occasional crashes caused by TIMGroupInfo > lastMsg calling related functions.
- TUIKit
  - Optimized the recent contact list update algorithm to reduce the refresh frequency.
  - Fixed blocklist memory leak.
  - Added message bubble and profile photo click event callbacks.
  - Fixed the issue where the latest profile photo was not displayed in recent contacts or the chat window.
  - Optimized document annotations.

Android

- Organized and merged APIs.
  - Added all APIs in `TIMManagerExt` to `TIMManager`.
  - Added all APIs in `TIMConversationExt` to `TIMConversation`.
  - Added all APIs in `TIMGroupManagerExt` to `TIMGroupManager`.
  - Added all APIs in `TIMMessageExt` to `TIMMessage`.
  - Added all APIs in `TIMUserConfigMsgExt` to `TIMUserConfig`.
  - Retained APIs in `TIMManagerExt`, `TIMMessageExt`, `TIMConversationExt`, `TIMGroupManagerExt`, and `TIMUserConfigMsgExt` classes provisionally for compatibility purposes, which will be deprecated in the future.
- Added options to add friends in one-way or two-way manner.
- Added the `disableStorage` API to disable all local storage.

- Added APIs to get the download URLs of file, video, and voice messages.
- Fixed the issue where `queryUserProfile` was null on some Android mobile phones.
- Fixed the issue where the conversation on the sender's device could still get lastMsg after an online message was sent.
- Removed the return value of `getSenderProfile`, and used callback instead.
- Fixed occasional crashes when data was reported upon exit.
- Optimized the login module (repeated login/frequent login/frequent account switching/automatic connection/offline user being kicked off).
- Fixed the issue where the unread count could not be cleared after a member left a group or a group was deleted.
- Fixed the issue where group deletion notifications could not be received occasionally.
- Fixed the issue where longer time was required to deliver messages when the app went to the foreground after staying in the background for a long time.
- Optimized the one-to-one chat unread count.
- TUIKit
  - Short video messages in chats can be played in landscape or portrait orientation.
  - Added support for Javadoc documentation.
  - Fixed the issue where downloading a video that was being sent failed.
  - Fixed the issue where the `onSuccess` callback of the `GroupChatManagerKit.getInstance().sendMessage` method could be triggered twice.
  - Fixed the issue with short audio messages on the chat interface. Audio messages should be at least 1 second long. For messages shorter than 1 second, "Message too short" is displayed.
  - Fixed the issue where a user could be invited to join a private group repeatedly.
  - Fixed the issue where remarks could not be empty.
  - Fixed the issue where the time displayed on the chat interface was incorrect when the system time of the device was incorrect.
  - Fixed the issue where voice messages sent locally could not be downloaded on another mobile phone from roaming messages.
  - Fixed the issue where the group owner failed to set the group name to null but a message stating that the setting was successful was displayed.

Windows

- Fixed the issues where various platforms sent Chinese characters when image, file, audio, and video messages contained Chinese paths.
- Fixed the issue where `TIMMsgReportReaded` was invalid.
- Fixed the issue where the received message and recalled message have different rand and seq.
- Fixed occasional crashes when data was reported upon exit.

- Optimized the login module (repeated login/frequent login/frequent account switching/automatic connection/offline user being kicked off).
- Fixed the issue where the unread count could not be cleared after a member left a group or a group was deleted.
- Fixed the issue where group deletion notifications could not be received occasionally.
- Fixed the issue where longer time was required to deliver messages when the app went to the foreground after staying in the background for a long time.

Patch 4.4.631 @2019.07.03

Android

Fixed offline push issues and crashes.

4.4.627 @2019.06.27

iOS and Mac

- Fixed the message sending timeout issue when no network connection was available.
- Fixed the issue where the message ID value was changed after the message was sent.
- Fixed the disordered message issue.
- Fixed the issue where messages were lost when chat room historical messages were pulled.
- Fixed the issue with incorrect system message types.
- Fixed the issue where the obtained original image size of an image message was 0.
- Fixed the issue where mobile phones failed to send messages after the system time was changed.
- Fixed the issue where reporting conversation read and getting the unread count failed in some cases.
- Fixed the issue where online messages that had been sent could be obtained through getLastMessage of the conversation.
- Fixed the issue where getting lastMsg status through the conversation was exceptional after the last message was recalled.
- Fixed the issue where recalled message content still existed in the conversation list of the peer.
- Fixed the issue where the sending status of image/voice/file messages was exceptional after network reconnection.
- Fixed the issue where login accounts that contained special characters could not send audio and images.
- Fixed the issue where the V4 version could not get the width and height of thumbnails sent by the V2 version.

- Fixed the issue where recent conversations failed to be pulled after `saveMessage` was created for a conversation.
- Fixed the issue where `getMessage` failed to get the `MemberChangeList` content of group tips.
- Fixed the issue when `getLoginStatus` failed to get the login status.
- Fixed the issue where applicants became group members after their requests to join the group were rejected.
- Fixed the issue where a log file existed under the root directory of the drive letter after a log path was set.
- Mac: fixed the issue where the callback failed to be received in case of force offline.
- TUIKit
  - Optimized the group management page logic.
  - Fixed the iOS 13 compatibility issue.
  - Fixed known issues.

Android

- Fixed the message sending timeout issue when no network connection was available.
- Fixed the issue where the message ID value was changed after the message was sent.
- Fixed the disordered message issue.
- Fixed the issue where messages were lost when chat room historical messages were pulled.
- Fixed the issue with incorrect system message types.
- Fixed the issue with exceptional progress value when files were downloaded.
- Fixed the issue where mobile phones failed to send messages after the system time was changed.
- Fixed the issue where the sending status of image/voice/file messages was exceptional after network reconnection.
- Fixed exceptional message sorting after a group was deleted or a user was muted.
- Fixed the issue where reporting conversation read and getting the unread count failed in some cases.
- Fixed the issue where recalled message content still existed in the conversation list of the peer.
- Fixed the issue where the status fetched by `getLastMessage` of the conversation was exceptional after the last message was recalled.
- Fixed the issue where sent online messages could be obtained through `getLastMessage` of the conversation.
- Fixed the issue where the obtained original image size of an image message was 0.
- Fixed the issue where the V4 version could not get the width and height of thumbnails sent by the V2 version.
- Fixed the issue where `getLoginUser()` could still get login users after they were forced offline.
- Fixed the issue where `getSenderProfile` returned blank information.
- Fixed the issue where `getOpUser` of `TIMGroupSystemElem` was empty.

- Fixed the issue where `getMessage` failed to get the `MemberChangeList` content of group tips.
- Fixed the issue where recent conversations failed to be pulled after `saveMessage` was created for a conversation.
- Fixed the issue where a log file existed under the root directory of the drive letter after a log path was set.
- Fixed known TUIKit issues.

Windows

- Fixed the message sending timeout issue when no network connection was available.
- Fixed the issue where the message ID value was changed after the message was sent.
- Fixed the disordered message issue.
- Fixed the issue where messages were lost when chat room historical messages were pulled.
- Fixed the issue with incorrect system message types.
- Fixed the issue where the iOS IM SDK module of the cross-platform library did not include the ARMv7-A architecture.
- Fixed the issue where empty messages were not supported by the `TIMMsgReportReaded` API of the cross-platform library.
- Fixed the issue where multiple IM instances could run on one cross-platform library device with the same account and would be kicked off.
- Added the JSON key for getting the unique ID of messages to cross-platform library messages.
- Fixed the issue where a log file existed under the root directory of the drive letter after a log path was set.
- Fixed the issue where `getMessage` failed to get the `MemberChangeList` content of group tips.
- Fixed the issue where getting lastMsg status through the conversation was exceptional after the last message was recalled.
- Fixed the issue where reporting conversation read and getting the unread count failed in some cases.

4.4.479 @2019.06.12

iOS

- Fixed the issue with message loss when offline messages were pulled.
- Fixed the login failure caused by changing SDKAppID.
- Fixed the issue where voice messages failed to play.
- Fixed crashes caused by recalling group messages.
- Fixed the 6002 error when getting friend lists and creating groups.

- Improved the message sending efficiency.
- Optimized the cache to mitigate UI lag.
- TUIKit
  - New UI design
  - New architecture design
  - Improved features such as contacts, group management, and relationship chain.
  - Fixed bugs.

Android

- Fixed the issue with message loss when offline messages were pulled.
- Fixed the login failure caused by changing SDKAppID.
- Fixed the issue where voice messages failed to play.
- Fixed crashes caused by recalling group messages.
- Fixed the 6002 error when getting friend lists and creating groups.
- Fixed Android device crashes caused by creating groups with too many members.
- Improved the message sending efficiency.
- Optimized the cache to mitigate UI lag.
- TUIKit
  - New UI design
  - New architecture design
  - Improved features such as contacts, group management, and relationship chain.
  - Fixed bugs.

Windows

- Fixed the issue with message loss when offline messages were pulled.
- Fixed the login failure caused by changing SDKAppID.
- Fixed the issue where voice messages failed to play.
- Fixed crashes caused by recalling group messages.
- Fixed the 6002 error when getting friend lists and creating groups.
- Optimized the cache to mitigate UI lag.
- Improved the message sending efficiency.

4.3.145 @2019.05.31

iOS

- Fixed the issue where the same message was received after switching to another account.

- Fixed crashes caused by getting one-to-one roaming messages after the ticket expired.
- Fixed the issue where new chat room members could not see the chat history.
- Fixed FindMsg crashes.
- Optimized group message synchronization.
- Fixed occasional getReciveMessageOpt errors.

Android

- Fixed the issue where the same message was received after switching to another account.
- Fixed crashes caused by getting one-to-one roaming messages after the ticket expired.
- Fixed the issue where new chat room members could not see the chat history.
- Fixed the issue where the same message listener was added repeatedly.
- Fixed FindMsg crashes.
- Optimized group message synchronization.

Windows

- Fixed the issue where the same message was received after switching to another account.
- Fixed crashes caused by getting one-to-one roaming messages after the ticket expired.
- Fixed the issue where new chat room members could not see the chat history.
- Optimized group message synchronization.

4.3.135 @2019.05.24

iOS

- Added the `checkFriends` API to verify friends.
- Added the `queryGroupInfo` API to get local data.
- Deprecated `getGroupPublicInfo` and replaced it with `getGroupInfo` .
- Fixed the issue where deleted messages could be seen in the message list.
- Fixed the issue where local messages could not be obtained before login.
- Fixed the pulling quantity and sorting issues of recent contacts.
- Fixed group message synchronization after network reconnection.
- Fixed the issue where identifying duplicates failed when a large number of messages were received in a short time.
- Fixed the issue where the same message might be received again after the app restarted.
- Fixed occasional errors in initialization and message synchronization.
- Fixed occasional errors caused when `lastMsg` of a conversation was deleted.
- Fixed the issue where `onRefreshConversation` was called back twice with identical data.

- Fixed the issue where users could not obtain the chat history of a chat room before the time they joined the chat room.
- Fixed the issue where `copyFrom` of `TIMMessage` failed to work.
- Fixed the issue where `TIMGroupEventListener` failed to receive callbacks.
- Fixed crashes reported online.
- Optimized connection requests during reconnection.
- Optimized the quality of first connections to different networks and access points outside the Chinese mainland.
- Improved the network reconnection speed when iOS devices switch to Wi-Fi networks.

Android

- Added the `checkFriends` API to verify friends.
- Added the `queryGroupInfo` API to get local data.
- Deprecated the `getGroupDetailInfo` and `getGroupPublicInfo` APIs and replaced them with the `getGroupInfo` API.
- Fixed the issue where deleted messages could be seen in the message list.
- Fixed `modifyGroupOwner` and `getGroupMembersByFilter` callback issues.
- Fixed the issue where local messages could not be obtained before login.
- Fixed the pulling quantity and sorting issues of recent contacts.
- Fixed group message synchronization after network reconnection.
- Fixed the issue where identifying duplicates failed when a large number of messages were received in a short time.
- Fixed the issue where the same message might be received again after the app restarted.
- Fixed occasional errors in initialization and message synchronization.
- Fixed occasional errors caused when `lastMsg` of a conversation was deleted.
- Fixed the issue where `onRefreshConversation` was called back twice with identical data.
- Fixed the issue where users could not obtain the chat history of a chat room before the time they joined the chat room.
- Fixed crashes reported online.
- Optimized connection requests during reconnection.
- Optimized the quality of first connections to different networks and access points outside the Chinese mainland.

Windows

- Added support for custom field data reporting.
- Added messages that disappear after being viewed.
- Added use cases for recalling messages.

- Fixed occasional failures in setting upload files.
- Fixed the issue where deleted messages could be seen in the message list.
- Fixed the pulling quantity and sorting issues of recent contacts.
- Fixed group message synchronization after network reconnection.
- Fixed the issue where identifying duplicates failed when a large number of messages were received in a short time.
- Fixed the issue where the same message might be received again after the app restarted.
- Fixed occasional errors caused when `lastMsg` of a conversation was deleted.
- Fixed occasional errors in initialization and message synchronization.
- The JSON string of a delivered message is returned in the callback indicating successful delivery.
- Replaced `TIMSetRecvNewMsgCallback` with `TIMAddRecvNewMsgCallback` and `TIMRemoveRecvNewMsgCallback` .
- Added SOCKS5 proxy configuration.
- Optimized connection requests during reconnection.
- Optimized the quality of first connections to different networks and access points outside the Chinese mainland.

4.3.118 @2019.05.10

iOS

- Added `querySelfProfile` and `queryUserProfile` to the `TIMFriendshipManager` class (reading local data).
- Fixed the issue where `getLoginUser` returned a login user exception.
- Fixed the issue where online reported user profiles failed to be obtained.
- Fixed the issue where some local fields became invalid after the app restarted.
- Fixed occasional errors when calling read reports after messages were deleted.
- Fixed the online reported IM group issue.
- Fixed the issue with conversation unread counts.
- Fixed the issue with online messages.
- Fixed the issue where messages failed to be re-sent occasionally.
- Fixed the issue where local ticket expiration caused repeated reconnection.
- Fixed crashes reported online.
- Optimized the server connection strategy.
- Optimized the network reconnection strategy.
- Optimized the server overload strategy.
- Optimized heartbeat to reduce unnecessary outbound packets.
- Added support for importing through CocoaPods for TUIKit.

- Added the Contacts interface for TUIKit.
- Added the Adding Friends interface for TUIKit.
- Added the Blocklist interface for TUIKit.
- Added the Search Friend interface for TUIKit.
- Added the New Friends interface for TUIKit.
- Optimized the friend's profile page for TUIKit: added the Remarks, Blocklist, and Delete Friend features.
- Optimized the user profile page for TUIKit: added support for modification of nicknames, personal signature, date of birth, gender, and location.
- Improve the group pinning feature for TUIKit.

Android

- Added `querySelfProfile` and `queryUserProfile` to the `TIMFriendshipManager` class (reading local data).
- Added the `addTime` field when getting a friend's profile.
- Added support for the x86 and x86_64 architecture.
- Fixed the issue where `getLoginUser` returned a login user exception.
- Fixed the issue where online reported user profiles failed to be obtained.
- Fixed the issue where some local fields became invalid after the app restarted.
- Fixed occasional errors when calling read reports after messages were deleted.
- Fixed the online reported IM group issue.
- Fixed the issue with conversation unread counts.
- Fixed the issue with online messages.
- Fixed the issue where messages failed to be re-sent occasionally.
- Fixed the issue where local ticket expiration caused repeated reconnection.
- Fixed crashes reported online.
- Optimized the server connection strategy.
- Optimized the network reconnection strategy.
- Optimized the server overload strategy.
- Optimized heartbeat to reduce unnecessary outbound packets.
- Added the "pin chat to top" feature to TUIKit.
- TUIKit: nickname and personal signature can be changed, and the nickname is displayed on the profile page.
- TUIKit: fixed the issue where emojis sent by iOS devices failed to be displayed on Android devices.
- TUIKit: fixed the unread message red dot issue.
- TUIKit: fixed the issue where a message appeared stating that UIs were abnormal after the plus sign was tapped on Meitu M8 mobile phones.

- TUIKit: fixed the issue where profile photos were scaled down after being set and did not fill the entire UI.
- TUIKit: fixed the login and auto login logic.
- TUIKit: fixed the ANR issue when the input content exceeds the maximum limit.
- TUIKit: fixed the issue where no response was received when images were selected from the photo album and the **OK** button on the preview screen was tapped.
- TUIKit: fixed the issue where the message deleting and recalling buttons were not displayed after image messages were tapped and held on the chat interface.
- TUIKit: optimized and fixed crashes reported online.

Windows

- Fixed the issue where `getLoginUser` returned a login user exception.
- Fixed the issue where online reported user profiles failed to be obtained.
- Fixed the issue where some local fields became invalid after the app restarted.
- Fixed occasional errors when calling read reports after messages were deleted.
- Fixed the online reported IM group issue.
- Fixed the issue with conversation unread counts.
- Fixed the issue with online messages.
- Fixed the issue where messages failed to be re-sent occasionally.
- Fixed the issue where local ticket expiration caused repeated reconnection.
- Fixed crashes reported online.
- Optimized the server connection strategy.
- Optimized the network reconnection strategy.
- Optimized the server overload strategy.
- Optimized heartbeat to reduce unnecessary outbound packets.

4.3.81 @2019.04.24

iOS

- Fixed crashes caused by adding message elements to drafts.
- Fixed the issue where some accounts failed to pull conversation lists after an app was removed and reinstalled.
- Fixed the issue where login failed when usersig expired in login state and the app was not restarted.
- Fixed the issue where messages could not be sent and the usersig expiration callback was not received when usersig expired in login state.

- Fixed the issue with getting group member counts.
- Fixed the request timeout issue (error code 6012).

Android

- Added:
  - Supplemented relationship chain features such as blocklist, friend list, and friend request handling of earlier version SDKs.
- Fixed:
  - Fixed the issue where an error was reported when the main process of the app was killed.
  - Fixed the issue with getting group member counts.
  - Fixed issues with setting and getting custom group fields and custom group member fields.
  - Fixed the issue where no `onError` callback was sent after getting group profile timed out.
  - Fixed the issue where some accounts failed to pull conversation lists after an app was removed and reinstalled.
  - Fixed the issue where login failed when usersig expired in login state and the app was not restarted.
  - Fixed the issue where messages could not be sent and the usersig expiration callback was not received when usersig expired in login state.
  - Fixed disordered messages.
  - Fixed the request timeout issue (error code 6012).
  - Updated relationship chain error codes.
  - TUIKit: fixed a critical bug with the `DateUtils` class (GitHub issue #75).
  - TUIKit: fixed a crash (GitHub issue #86).
  - TUIKit: fixed issues with using SDK without permissions.
  - TUIKit: fixed crashes after deleting conversation, deleting message, and long-pressing.
  - TUIKit: fixed the issue where `popupwindow` would not disappear.
  - TUIKit: fixed the issue with repeated messages.
  - TUIKit: fixed the issue with intercepting empty messages containing whitespace.
  - TUIKit: fixed the issue where unread counts did not update after conversations were deleted.
  - TUIKit: fixed the issue with the maximum number of characters in a message.
  - TUIKit: improved experience and fixed several Array Index Out of Bounds exceptions.

Windows

- Fixed some crashes.
- Fixed the request timeout issue (error code 6012).
- Fixed the issue where some accounts failed to pull conversation lists after an app was removed and reinstalled.

- Fixed the issue where login failed when usersig expired in login state and the app was not restarted.
- Fixed the issue where messages could not be sent and the usersig expiration callback was not received when usersig expired in login state.

4.2.52 @2019.04.17

iOS

- Added:
  - Supplemented relationship chain features such as blocklist, friend list, and friend request handling of earlier version SDKs.
- Fixed:
  - Optimized API annotations.
  - Fixed the issue with ineffective group custom fields and group member custom fields.
  - Fixed the issue where `TIMMessage` failed to get user profiles through `senderProfile`.
  - Fixed the issue with read receipt callback and status.
  - Fixed the issue where the last message did not call back when unread messages were synchronized.
  - Fixed the issue where group messages occasionally could not be received.
  - Fixed the issue where login response packets could not be decrypted.
  - Added support for IP connection and login information reporting.
  - Fixed the message seq error.

Android

- Added:
  - Supplemented relationship chain features such as blocklist, friend list, and friend request handling of earlier version SDKs.
- Fixed:
  - Fixed jni leak on Android.
  - Fixed incorrect group member roles.
  - Fixed recalling group message crashes after a member left the group and joined the group again.
  - Fixed the issue where emojis were not displayed in the TUIKit demo.
  - Fixed the issue where the second page would often contain repeated messages when group chat messages were received.
  - Fixed some crashes in the TUIKit demo.

- Fixed the issue where `TIMMessage` failed to get user profiles through `senderProfile`.
- Fixed the issue with read receipt callback and status.
- Fixed the issue where the last message did not call back when unread messages were synchronized.
- Fixed the issue where group messages occasionally could not be received.
- Fixed the issue where login response packets could not be decrypted.
- Added support for IP connection and login information reporting.
- Fixed the message seq error.

Windows

- Added:
  - Supplemented relationship chain features such as blocklist, friend list, and friend request handling of earlier version SDKs.
- Fixed:
  - Fixed the issue where `TIMMessage` failed to get user profiles through `senderProfile`.
  - Fixed the issue with read receipt callback and status.
  - Fixed the issue where the last message did not call back when unread messages were synchronized.
  - Fixed the issue where group messages occasionally could not be received.
  - Fixed the issue where login response packets could not be decrypted.
  - Added support for IP connection and login information reporting.
  - Fixed the message seq error.

4.2.28 @2019.04.08

iOS

- Optimized issues related to unread counts.
- Optimized message read status.
- Fixed disordered one-to-one messages sent by RESTful APIs.
- Fixed occasional repeated roaming messages fetched.
- Optimized the `uniqueId` empty implementation issue.

Android

- Added:
  Added the logic for adding, deleting, and querying friends.
- Fixed:

- Optimized issues related to unread counts.
- Optimized message read status.
- Fixed disordered one-to-one messages sent by RESTful APIs.
- Fixed occasional repeated roaming messages fetched.
- Optimized the `uniqueId` empty implementation issue.

Windows

- Optimized issues related to unread counts.
- Optimized message read status.
- Mitigated disordered one-to-one messages sent by RESTful APIs.
- Fixed occasional repeated roaming messages fetched.

4.2.10 @2019.03.29

iOS

- New features
  Added the logic for adding, deleting, and querying friends.
- Fixed:
  - Mitigated the timeout issue.
  - Optimized the auto login logic.
  - Fixed crashes.
  - Fixed occasional network connection exceptions.

Android

- Mitigated the timeout issue.
- Optimized the auto login logic.
- Mitigated the JNI leak issue.
- Fixed crashes.
- Fixed occasional network connection exceptions.

Windows

- Mitigated the timeout issue.
- Fixed crashes.
- Fixed occasional network connection exceptions.

4.2.9 @2019.03.27

iOS and Mac

- Fixed crashes in the IPv6 environment.
- Fixed the issue where setting profiles to int failed.

Android

Fixed the issue where setting profiles to int failed.

4.2.1 @2019.03.15

iOS

- Fixed the issue where clients did not receive relevant instructions after a group was deleted in the backend.
- Fixed the issue where calling `deleteConversationAndMessage()` failed.
- Fixed the issue where no messages were received after network reconnection (On the conversion interface, messages can be proactively pulled after network reconnection.)

Android

- Fixed incorrect group pending and processed requests returned.
- Fixed client crashes when the client went to the backend.
- Fixed the issue where no messages were received after network reconnection.
- Fixed occasional message sorting errors.
- Fixed the issue where messages occasionally failed to be sent.

Web

Web IM can play .amr recordings.

Windows

- Added the /source-charset:.65001 compilation option.
- Fixed crashes when the file system directly ran IMAPP.exe.
- Fixed various compilation errors and crashes.
- Removed X64 compilation (not supported at present).

4.0.13 @2019.03.13

Android

Fixed crashes caused by login after 3.x is upgraded to 4.x.

iOS

- pod can directly integrate the TUIKit.framework.
- Fixed crashes caused by login after 3.x is upgraded to 4.x.

Windows

- Added the IM demo with the duilib library as a UI component.
- Added usage instructions and integration guide.

IM SDK 4.0.12 2019-3-11

iOS

- TUIKit.framework supports bitcode 2.
- Fixed ineffective group muting.
- Fixed the feature for modifying a user's role in a group.

Android

- Fixed ineffective group muting.
- Fixed the feature for modifying a user's role in a group.
- Fixed the issue with modifying group message receiving options.
- Fixed the issue with ineffective offline push toggle.

IM SDK 4.0.10 2019-3-7

Fixed the message receiving error when an audio-video group had more than 100 members.

IM SDK 4.0.8 2019-3-6

Optimized the audio playback logic for TUIKit.

IM SDK 4.0.7 2019-3-1

- Fixed the compatibility issue with audio, file, and video messages between earlier and later versions.
- Fixed "-5 tls exchange failed" where login was successful after uninstalling and then reinstalling the app.

IM SDK 4.0.4 2019-2-28

- Fixed the issue where an incorrect error code was returned when a user logged in after userSig expired. The correct error code is 6206.
- Optimized the force offline logic.

IM SDK 4.0.3 2019-2-25

Fixed the issue with third-party offline push.

IM SDK 4.0.2 2019-2-20

Fixed the issue where bitcode packaging activation failed.

IM SDK 4.0.1 2019-2-20

Fixed the issue where -5 is returned after login.

iOS--IM SDK 4.0.0.1 2019-1-21

Added TUIKIt.

IM SDK 3.3.2 2018-7-5

- Automatic read reporting is disabled by default.
- Custom information types of profile relationship chains support integer.
- Fixed the issue where the group member count obtained from local storage was incorrect.

- Fixed the issue where the nickname carried in one-to-one chat messages was not updated in real time.

IM SDK 2.7.2 2018-7-5

- Automatic read reporting is disabled by default.
- Custom information types of profile relationship chains support integer.
- Added the message recalling feature.
- Fixed the issue where the nickname carried in one-to-one chat messages was not updated in real time.

Windows--IM SDK 2.5.8 2018-7-5

- Fixed login failures in some cases.
- Custom information types of profile relationship chains support integer.

IM SDK 3.3.0 2018-4-4

iOS

Added the level and role fields to `TIMUserProfile`.

Android

- Added support for offline push on Meizu mobile phones.
- Added standard level and role attributes to user profiles.
- Fixed the issue where UGC short video failed to be sent when a user logged in after logging out.

IM SDK 2.7.0 2018-4-4

iOS

- Added custom data parameters to the API for inviting users to join a group.

Android

- Added support for offline push on Meizu mobile phones.
- Added support for custom data for the API for inviting users to join a group.

Windows--IM SDK 2.5.7 2018-3-13

- Modified the login module to improve communication security.
- Improved the message delivery capability with poor network connection.
- Fixed occasional crashes when logs were printed.

iOS--IM SDK 2.6.0 2018-3-13

- Provided an API for deleting roaming messages.
- Provided an API for serializing and deserializing message objects.
- Fixed some known issues.

iOS--IM SDK 3.2.0 2018-3-13

- Fixed the issue where an error was reported when `getUserProfile` contained custom friend fields.
- Optimized the group unread count update strategy.
- Optimized the logic and strategy for local message storage.
- Fixed some crashes.

Android--IM SDK 3.2.0 2018-3-13

- Fixed the issue where UGC short videos failed to be sent.
- Fixed the issue with no callbacks for sent messages when the network connection is interrupted.
- Fixed the issue where muting all did not take effect.
- Optimized the logic and strategy for local message storage.
- Fixed some crashes.

Android--IM SDK 2.6.0 2018-3-13

- Provided an API for deleting roaming messages.
- Provided an API for serializing and deserializing message objects.
- Fixed some known issues.

IM SDK 3.1.2 2017-12-12

- Mitigated the network timeout issue on Android devices.
- Fixed the audio download error on Android devices.
- Fixed various crashes on Android devices.

IM SDK 2.5.7 2017-11-08

- Fixed SDK crashes when app processes were killed.
- Fixed the issue where offline messages were repeatedly pushed.
- Fixed the issue where internal accounts may be empty when `initStorage` and `login` are called at the same time.
- Optimized the network detection strategy.
- Fixed the error in getting friend lists.
- Fixed some crashes.

IM SDK 3.1.1 2017-8-16

- Optimized the regular log clearing mechanism.
- Fixed the issue where iOS QALSDK crashed upon initialization.
- Added the feature for muting all group members.
- iOS: fixed the multi-user login failure.
- Android: fixed crashes caused by getting group lists before login.

IM SDK 2.5.6 2017-7-14

- Fixed crashes during login and logout.
- Fixed crashes during push and recording.

IM SDK 3.1.0 2017-7-3

- Added IMUGCExt.framework and TXRTMPSDK.framework to provide short video recording and upload.
- Added the Recall Message feature.

IM SDK 2.5.5 2017-6-6

- Optimized the logic for internal response packets to reduce time consumption.
- Improved the log time granularity to millisecond.
- Fixed some crashes and message synchronization issues.

IM SDKV3 3.0.2 2017-5-22

- Fixed the issue where users cannot receive group messages in an audio-video group.
- Adjusted APIs.
  i. Deprecated `TIMFileElem` and the `setData` API in `TIMSoundElem`.
  ii. Corrected spelling of the `getConversionList` API in `TIMManagerExt` to `getConversationList`.

IM SDKV3 3.0.1 2017-5-15

Fixed the issue where some .so libraries were incompatible with devices running systems earlier than Android 5.0.

IM SDKV3 3.0 2017-5-8

- Regrouped IM SDK and IMCore into IM SDK, IMMessageExt, IMGroupExt, and IMFriendExt.
- Optimized the IM SDK initialization method to initSdk and setUserConfig.
- Names of IM SDK APIs and protocol callback methods start with lowercase letters.
- IM SDK features: basic login, receiving and sending messages, profile, and group features
- IMMessageExt features: full message features, including message pulling, local storage, and unread count
- IMGroupExt features: full group features, including group type management and group member management
- IMFriendExt features: full relationship chain features, including friend list and blocklist

IM SDK 2.5.4 2017-4-28

- Fixed the timer mechanism bug in the IM SDK.

IM SDK 2.5.3 2017-4-17

iOS

- `sendOnlineMessage` supports group messages, which will not be saved to local storage, stored offline, or included in the unread count.
- Added the `findMessages` method to get local messages by message ID.
- `TIMIOSOfflinePushConfig` provides the option for setting APNs push muting.
- Fixed the issue of excessive memory consumption when messages were received at high frequency.

Android

- Added the API for searching for messages. (For more information, see `findMessages` under `TIMConversation`.)
- `sendOnlineMessage` supports group messages, which will not be saved to local storage, stored offline, or included in the unread count.
- Added the configuration item that allows a device to receive APNs push notifications without playing a sound or vibration. (For more information, see TIMMessageOfflinePushSettings.IOSSettings.NO_SOUND_NO_VIBRATION.)
- Optimized networking to improve SDK robustness in poor network connection.

Windows

- Fixed issues that may cause crashes.

API changes:

- Changed how TIMMessageOfflinePushSettings.AndroidSettings and TIMMessageOfflinePushSettings.IOSSettings are constructed.
  For more information, see Offline Push

IM Android SDK 2.5.2 2017-3-1

- Fixed the issue where the return of outgoing packets occasionally timed out (return code 6205).

IM SDK 2.5.1 2017-2-16

- Limited the maximum size of log files to 50 MB.
- Fixed the bug where the online state was returned after a user logged out and the app went to the backend.

- iOS: updated the audio and file downloading strategy and supported HTTP and HTTPS download.
- Fixed the status mismatch bug after messages failed to be sent when the user was not logged in.

IM Web SDK 1.7 2016-12-20

- Added support for multi-instance force offline.
- Added support for simultaneous online of multiple instances.
- Added support for synchronization of read group messages.
- Added support for synchronization of read one-to-one messages.
- Optimized the demo directory structure and code.
- Added the recent contacts list.

IM SDK 2.5 2016-12-16

- Optimized the `TIMOfflinePushInfo` object structure.
- Fixed audio and file download failures in iOS 9.1.
- Optimized network operations.
- Fixed some bugs.

IM SDK 2.4.1 2016-11-24

- Fixed the bug where `TIMGroupAssistant` exceptionally pulls the group profile after entering an audio-video group.
- Fixed the bug where disabling console print failed.
- Fixed the issue where various listeners became invalid when logout is called before login after initialization.

IM SDK 2.4 2016-11-09

- Full compatible with the ATS mode.
- Message forwarding feature: The `copyFrom` API forwards image and file messages by copying images and files without downloading them.
- The number of members in an audio-video group is dynamically updated. `TIMGroupEventListener` returns the current number of group members.
- Message filtering can be customized for audio-video groups.

- `TIMOfflinePushInfo` attributes support push notification settings of Mi and Huawei mobile phones.
- Optimized the process of pulling group roaming messages.
- Optimized the processes of uploading and downloading audio, files, and short videos.
- Throwing `onNewMessage` when pulling the recent contacts list can be disallowed.

IM SDK 2.3 2016-9-13

- Added support for push notifications to multiple apps with one appid.
- Added `setOfflinePushToken` with callback to the Android version.
- Optimized the message deletion logic to automatically filter messages in the DELETED state when messages were pulled.
- iOS: moved database files from subdirectory Library/Caches/ to subdirectory Document/ to prevent them being cleared by the system.
- Multiple TIMMessageListeners can be added and deleted in iOS versions.
- Resident threads in iOS versions are named in a unified manner.
- The API for getting conversation lists automatically filters conversations with the message count set to 0.

IM Web SDK 1.6 2016-8-15

- Web broadcast message requirements
- Added friend system notifications.
- Added profile system notifications.

IM SDK 2.2 2016-8-10

- Added support for conversation drafts.
- Conversations can be marked whether to store messages to ensure more flexible message handling.
- Roaming messages can be traversed from old to new, which applies to scenarios where message recording is needed.
- Added ext and sounds of push notifications to messages, allowing setting push information for some messages.
- Added `stopQALService` to the Android SDK, which turns off QALService when exiting the app.
- Added support for network status monitoring and added error codes for network errors.

IM SDK 2.1 2016-7-15

- Added support for notification push to Mi and Huawei mobile phones.
- Added support for the read receipts feature, which is optional depending on product needs.
- Added support for typing reminder, which is optional depending on product needs.
- Added standard fields such as the gender, date of birth, address, and language to profile relationship chains.
- Notifications for joining and quitting a group contain the group member count.
- Fixed some SDK and demo bugs.

IM Web SDK 1.5 2016-7-13

- Merged broadcasting chat room SDK capabilities.
- Fixed issues with uploading images in Internet Explorer 8 and 9.
- Added a group member count field to tips for joining and leaving groups.
- Fixed some SDK and demo bugs.

IM SDK 2.0 2016-6-16

- The unread count can be synchronized between multiple online devices.
- Historical messages can be imported when an app is migrated to ensure smooth migration.
- Added the message notification status to group message attributes.
- Added support for flexible settings for message priorities.
- Push notifications can be filtered by attribute and tag.

IM Web SDK 1.4 2016-6-7

- Friends' message history can be pulled.
- Red packets and like messages can be sent.
- The API for creating groups supports custom group IDs and broadcasting chat rooms.
- Optimized SDK APIs and merged the login and initialization APIs.
- Optimized the demo directory structure and code.

IM SDK 1.9.3 2016-5-31

- Fixed resource destruction deadlocks when the winsdk process exited.

IM SDK 1.9.2 2016-5-27

- Added the ticket expiration callback.
- Added support for IPv6 (iOS).

IM SDK 1.9 2016-5-4

- Added support for groups with more than 10,000 members (no limit on the number of members, which is suitable for broadcasting scenarios).
- Reconstructed the IM demo for better experience and ease-of-use.
- Messages can be sent based on their priorities.
- Added storage and cache for group profiles and relationship chains.
- Added APIs to synchronize group profiles and relationship chains and change callbacks.
- Added support for getting friend profiles, including remarks and lists.
- Added support for setting default group profile and relationship chain fields to be pulled.
- Added support for disabling pulling recent contacts.
- Added synchronizing the last message to the conversation list.
- You can specify the group members whose group information, such as group name cards, is to be pulled.
- Added support for passing in file paths for voice and file messages (messages can be resent).
- Adapted to Android 6.0 dynamic permission management.

IM SDK 1.8.1 2016-4-13

- Android: optimized the auto-start process. (To modify configuration, see ReadMe.txt.)
- Added the API for sending online messages in one-to-one chats. (The messages will be received only when the receiver is online and will not be stored when the receiver is offline.)
- Added the API for batch sending messages.
- Optimized Android performance.

IM SDK 1.8 2016-3-23

- Android offline push

- Added the API to verify friend relationships.
- Added the relationship chain custom field API.
- Messages can be customized for local storage (for example, audio can be identified as read or unread).
- Added the API to compress images, meeting the need for image compression in detached communication scenarios.
- Customized messages' sound fields to specify APNs sounds.
- Optimized callback APIs for online status change.

IM SDK 1.7 2016-1-25

- Added support for limiting the message sending frequency in groups.
- Added support for group ownership transfer.
- Group message notification intensity can be customized.
- CS channels are established to remove the need for a persistent connection between the app and backend to reduce battery consumption.
- Added configuration items, including message and recent contacts roaming switch, storage duration, and multi-device online switch to improve operational efficiency.
- Downstream messages carry group member nicknames and contact cards to improve user experience and ease-of-use.
- Simplified the SDK to reduce the installation package size.

IM SDK 1.6 2015-12-25

- Short video messages are supported to meet growing needs for video messages and social communication.
- Added support for rule-based sorting of group members.
- Added support for relationship chain friend lists.
- Added support for filtering of sensitive words in group names, announcements, and introductions.
- Added support for group member contact cards to help users identify group members.
- Added support for the message notifications switch, allowing users to turn on or off message notifications for one-to-one chats and group chats.

IM SDK 1.5 2015-11-16

- Added support for asynchronous download of message records.

- Group messages can be deleted at the server side.
- Users can be searched by nickname.
- Groups can be searched by group name.
- Event callbacks can be configured in the console.
- User credentials of admin accounts can be downloaded.
- Optimized some demo and technical logic.

IM SDK 1.4 2015-10-16

- Multi-device login is supported.
- Messages from blocklisted users cannot be received.
- Deleted friend recommendations.
- APNs pushes nicknames.
- Added support for filtering sensitive words in group names.
- Added support for filtering sensitive words in group announcements.
- Demo supports the guest mode and third-party account login.

IM SDK 1.3 2015-09-10

- Users can log in as guests without usernames and passwords.
- Message roaming is supported. (Messages are stored for seven days by default.)
- Recent contacts roaming and deletion are supported.
- Real-time message synchronization through callbacks is supported.
- Friend recommendation is supported after the recommendation logic has been defined.
- Sending original images or thumbnails is supported for better user experience.
- Added support for push notifications (available only to online Android users).
- Added support for smooth migration.
- Added support for deleting local messages to protect users' privacy.

IM SDK 1.2 2015-08-18

- One-to-one chats on the web platform are supported.
- The maximum number of group members is increased to 10,000.
- Added support for filtering ads and sensitive words in messages.
- Added an API that provides message IDs to precisely locate messages.
- Added remarks to user profiles.

- Added support for viewing local messages when offline.

IM SDK 1.1 2015-07-13

- Windows C++ platform is supported.
- Public groups and chat rooms are supported.
- Added support for adding group introductions and announcements and added muting, message block, and group role setting.
- Added APIs for user profile and relationship chain operations, such as setting nicknames, adding friends, and setting a blocklist.
- Added support for file messages.
- Optimized image messages: image quality includes the original image, thumbnail, and large image. Changed upload and download APIs. Image URLs can be passed.
- Added log levels to the log callback API.
- Added the logic to execute forced logout on one device in the event of repeated logins.
- Added automatic crash reporting.
- Added support for self-owned account and third-party account integration in hosting mode.
- Added SMS authentication for user registration and login.
- Added support for ticket verification using public keys and private keys generated by Tencent.
- Added user and group management.

IM SDK 1.0 2015-05-11

- Added support for Android/iOS platforms.
- Added support for integrating Tencent account and third-party account logins.
- Added support for one-to-one chats and group chats (discussion groups).
- Added support for text, emoji, image, audio, location, and custom messages.
- APNs push notifications (token reporting, foreground and background switching event reporting)
- Messages can be stored locally.

Update Log (Unity)

Last updated : 2021-10-09 11:04:23

1.4.0 @2021.08.03

- Simplified the configuration process for the iOS client.
- Fixed the issue with the Android client where an error occurred during packaging under IL2CPP.

1.3.1 @2021.05.21

- Fixed known issues.

1.3.0 @2021.05.10

- Added the C# model to instantiate data returned by APIs.
- Added the usage of the C# model to `ExampleEntry.cs` .

1.2.0 @2021.04.28

- Added `sequenceID` to some message sending APIs to associate message requests and responses.

1.1.1 @2021.04.25

- Separated the method of dynamically fetching userSig.

1.1.0 @2021.04.15

- Added advanced message APIs.
- Added signaling message APIs.

1.0.1 @2021.04.01

- Initialized the project and implemented most APIs.

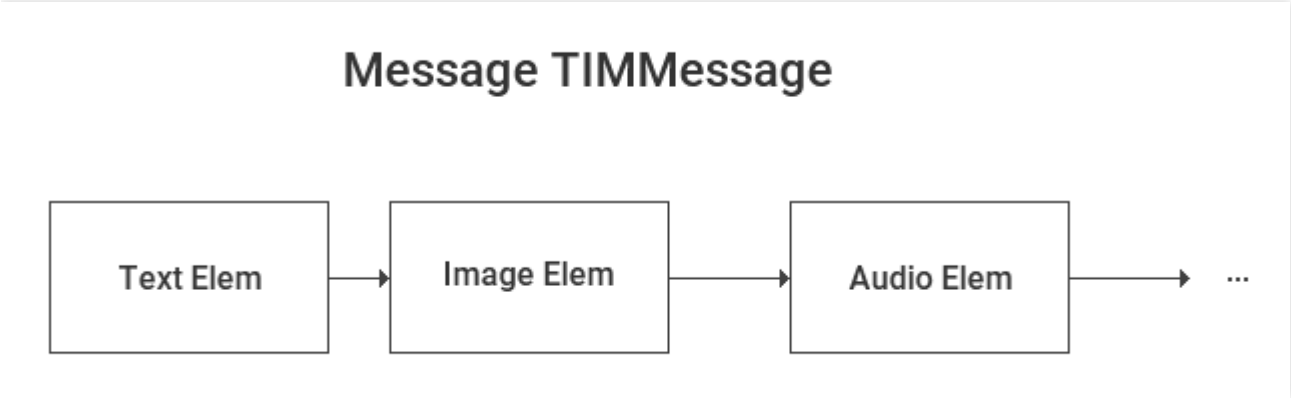
Legacy API Tutorials
Overview
Overview (Android)

Last updated : 2020-03-12 17:03:18

IM SDK Concepts

Conversation: there are two types of conversations in the IM SDK. One is the C2C conversation, which is a one-to-one chat between users with messages read and sent within the conversation. The other is the group conversation, which is a group chat where all group members can receive messages.

Message: a message in the IM SDK is the information to be sent to the recipient. A message includes properties such as whether it was read by yourself, whether is was sent successfully, the sender’s account, and the message generation time. A message consists of several Elems, which can be text, images, emojis, and others. A message can contain multiple Elems.



Group ID: a group ID uniquely identifies a group. It is generated by the backend and returned when a group is created.

Introduction to IM SDK Objects

IM SDK objects include the communication manager, conversation, message, and group management. The following table describes them in detail.

Object	Description	Features
TIMManager	Manager, which is responsible for basic operations of the IM SDK	Initialization, login, logout, and creating conversations
TIMConversation	Conversation, which is responsible for conversation-related operations	Sending messages, obtaining cached conversation messages, and obtaining the unread count
TIMMessage	Message	Support various message types including text and image messages
TIMGroupManager	Group manager	Creating groups, joining groups, and quitting groups
TIMFriendshipManager	Profile and relationship chain manager	Obtaining and modifying profiles and relationship chains

Calling Sequence

The IM SDK call APIs in the following sequence, and other methods will be called after login succeeds.

Step	Function	Description
Initialization	TIMSdkConfig	Configure basic IM SDK settings such as SDKAppID and the log level
	TIMManager : init	Initialize the IM SDK
	TIMManager : setUserConfig	Configure basic user settings
	TIMManager : addMessageListener	Set the message listener
Login	TIMManager : login	Log users in
Sending and receiving messages	TIMManager : getConversation	Obtain conversations
	TIMConversation : sendMessage	Send messages

Group management	TIMGroupManager	Managing groups
Logout	TIMManager : logout	Log users out

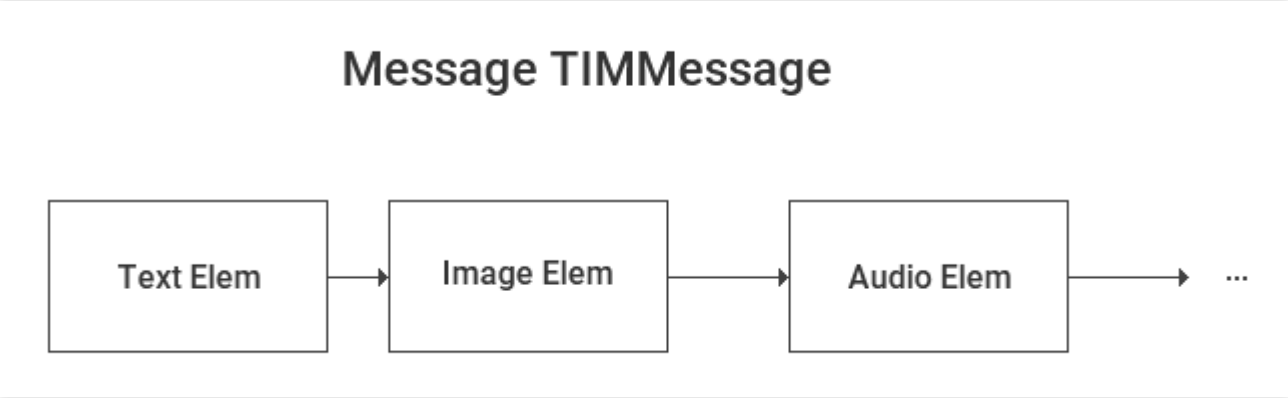
Overview (iOS)

Last updated : 2020-03-12 17:05:32

IM SDK Concepts

Conversation: there are two types of conversations in the IM SDK. One is the **C2C conversation**, which is a one-to-one chat between users with messages read and sent within the conversation. The other is the **group conversation**, which is a group chat where all group members can receive messages.

Message: a message in the IM SDK is the information to be sent to the recipient. A message includes properties such as whether it was read by yourself, whether it was sent successfully, the sender's account, and the message generation time. A message consists of several `Elems`, which can be text, images, emojis, and others. A message can contain multiple `Elems`.



Group ID: a group ID uniquely identifies a group. It is generated by the backend and returned when a group is created.

Introduction to IM SDK Objects

iOS IM SDK objects include the communication manager, conversation, message, and group management. The following table describes them in detail.

Object	Description	Features

Object	Description	Features
TIMManager	Manager	Responsible for basic IM SDK operations such as initialization, login, logout, and creating conversations
TIMConversation	Conversation	Responsible for conversation-related operations such as sending messages, obtaining cached conversation messages, and obtaining the unread count
TIMMessage	Message	Support various message types including text and image messages
TIMGroupManager	Group manager	Responsible for creating groups, adding and deleting members, and modifying group profiles
TIMFriendshipManager	Friend relationship chain manager	Responsible for adding and deleting friends and managing friend profiles

Calling Sequence

The IM SDK calls APIs in the following sequence, and other methods will be called after login succeeds.

Step	Function	Description
Initialization	TIMManager:initSdk	Configure SDK settings
Initialization	TIMManager:setUserConfig	Configure user settings
Login	TIMManager:login	Log users in
Receiving and sending messages	TIMManager:getConversation	Obtain conversations
Receiving and sending messages	TIMConversation:sendMessage	Send messages
Group management	TIMGroupManager	Manage groups
Relationship chain management	TIMFriendshipManager	Manage relationship chains

Step	Function	Description
Logout	TIMManager:logout	Log users out (optional for users)

Overview (Web & Mini Programs)

Last updated : 2020-08-27 16:55:18

TIM is the namespace of the IM Web SDK and provides the create() static method, the EVENT event constant, and the TYPES type constant.

IM SDK Concepts

Concept	Description
Message	A message indicates the information to be sent. It carries multiple properties, which specify whether you are the sender, the sender account, the message generation time, and others.
Conversation	There are two types of conversations: • Client to Client (C2C): a one-to-one chat with only two participants. • GROUP: a group chat with more than two participants.
Profile	The profile describes the basic information of a user, including the nickname, profile photo URL, personal signature, and gender.
Group	Group in IM SDK is a communication system for group chatting, including Work, Public, Meeting, and AVChatRoom.
GroupMember (group member)	GroupMember specifies the basic information of each group member, such as the ID, nickname, group role, and joining time.
Group notification	A group notification is generated when an event, such as the addition or deletion of a group member, occurs. You can configure whether to display group notifications to group members. For more information on group notification types, see Message.GroupTipPayload.
Group system message	For example, when a user applies for joining a group, the group admin will receive a system message. After the admin accepts or denies the application, the IM SDK returns the application result to the access terminal, and then the user can view the result on the access terminal. For more information on the types of group system messages, see Message.GroupSystemNoticePayload.
Message on-screen display	This is the process by which messages, including text and images, are displayed on the computer or mobile phone screen after the user clicks send.

Supported Platforms

The IM SDK supports IE 9+, Chrome, WeChat, Mobile QQ, QQ Browser, FireFox, Opera, and Safari.

Calling Sequence

The IM SDK calls APIs in the following sequence.

Operation	Value	Description
Create an SDK instance	TIM.create(options)	Creates an SDK instance (which is usually represented by tim) by using a TIM factory function.
Set the log level	tim.setLogLevel(level)	Sets the log level. Logs with lower levels will not be output.
Register the plugin	tim.registerPlugin(optoins)	Cloud Object Storage (COS) must be registered as the upload plugin of the IM SDK to upload images, files, and other media.
Listen for events	tim.on(event, handler)	Listens for events and handles data thrown by the SDK in the handler.
Login	tim.login(options)	Messages can be sent and received after login succeeds and the SDK is ready.
Create a text message	tim.createTextMessage(options)	Creates text messages. This API returns a message instance that can be immediately displayed on the screen by the access terminal.
Send a message	tim.sendMessage(message)	Sends message instances that have been created.
Obtain the conversation list	tim.getConversationList()	Obtains the conversation list. The access terminal can then process conversation list data and render the conversation list interface.
Obtain the group list	tim.getGroupList()	Obtains the group list. The access terminal can then process group list data and render the group list interface.

Obtain the blocklist	tim.getBlacklist()	Obtains the blocklist. The access terminal can then process blocklist data and render the blocklist interface.
Obtain the personal profile	tim.getMyProfile()	Obtains the user’s personal profile. The access terminal can then process personal profile data and render the personal profile interface.
Logout	tim.logout()	Logs users out.

Overview (Windows)

Last updated : 2021-09-23 09:28:11

This document describes the usage of some basic features of the Tencent Cloud IM SDK to give you a better understanding of the basic processes of Instant Messaging (IM).

Initialization

Before using the SDK to perform IM operations, you need to initialize the SDK.

Example:

```
int sdk_app_id = 12345678;
std::string json_init_cfg;
Json::Value json_value_dev;
json_value_dev[kTIMDeviceInfoDevId] = "12345678";
json_value_dev[kTIMDeviceInfoPlatform] = TIMPlatform::kTIMPlatform_Windows;
json_value_dev[kTIMDeviceInfoDevType] = "";
Json::Value json_value_init;
json_value_init[kTIMSdkConfigLogFilePath] = path;
json_value_init[kTIMSdkConfigConfigFilePath] = path;
json_value_init[kTIMSdkConfigAccountType] = "107";
json_value_init[kTIMSdkConfigDeviceInfo] = json_value_dev;
TIMInit(sdk_app_id, json_value_init.toStyledString().c_str());
```

You can obtain the SDKAppID after creating an app in the IM console. For more information on initialization operations, see Initialization.

Login/Logout

Login

- Users can normally send and receive messages only after they have logged in to the Tencent backend server. To log in to the Tencent backend server, a user needs to provide information including UserID and UserSig. For more information, see Login Authentication.
  Login is an asynchronous process, and the result returned by the callback function indicates whether the login was successful. Users can proceed to subsequent operations only after successful login. When login succeeds or fails, the system will trigger the corresponding callback.

Example:

```
const void* user_data = nullptr; // Return of the callback function
const char* id = "WIN01";
const char* user_sig = "WIN01UserSig";
TIMLogin(id, user_sig, [](int32_t code, const char* desc, const char* json_param, const void* use
r_data) {
if (code != ERR_SUCC) {
// Failed to log in
return;
}
// Logged in successfully

}, user_data);
```

Here, code indicates the error code and desc indicates error description. For more information, see Error Codes.

onForceOffline

If this user has been forced logout by another client, the login fails and returns the error code (`ERR_IMSDK_KICKED_BY_OTHERS: 6208`). If a user is forced logout, be sure to notify the user of this with a notification window such as an Alert window. For more information on forcible logout, see User State Changes.

onUserSigExpired

Every UserSig has an expiration time. When a UserSig expires, `login` returns error code `70001` . If you receive this error code, request a new UserSig from your business server. For more information, see User Ticket Expiration.

Logout

The logout operation needs to be called if the user wants to log out or switch to another user.

Example:

```
const void* user_data = nullptr; // Return of the callback function
TIMLogout([](int32_t code, const char* desc, const char* json_param, const void* user_data) {
if (code != ERR_SUCC) {
// Failed to log out
return;
}
```

```
// Logged out successfully

}, user_data);
```

When you need to switch to another account, `login` can be called again only after the `logout` callback succeeds or fails. Otherwise, `login` may fail.
For more information on login and logout operations, see Login and Logout.

Sending Messages

Obtaining conversations

A conversation refers to a conversation with a user or a group. To send or receive messages in a one-to-one or group conversation, you need to first obtain the conversation by specifying the conversation type (one-to-one chat or group chat) and the peer's identifier (the peer's account or group ID).

Example of obtaining the one-to-one conversation with a recipient whose UserID is Windows-02

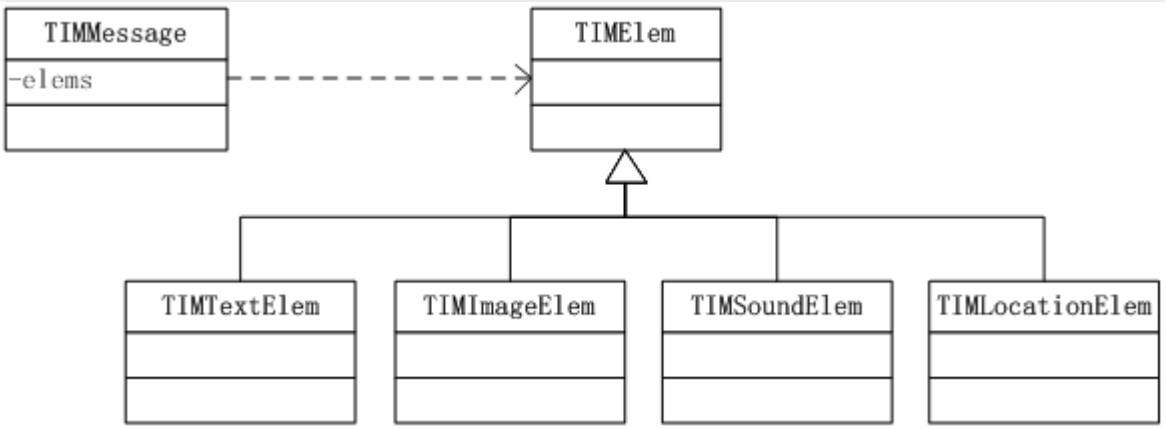
```
const void* user_data = nullptr; // Return of the callback function
const char* userid = "Windows-02";
int ret = TIMConvCreate(userid, kTIMConv_C2C, [](int32_t code, const char* desc, const char* json
_param, const void* user_data) {
// The callback returns details about the conversation.
}, user_data);
if (ret != TIM_SUCC) {
// Failed to call the TIMConvCreate API
}
```

Example of obtaining the conversation with the group ID of Windows-Group-01

```
const void* user_data = nullptr; // Return of the callback function
const char* groupid = "Windows-Group-01";
int ret = TIMConvCreate(groupid, kTIMConv_Group, [](int32_t code, const char* desc, const char* j
son_param, const void* user_data) {
// The callback returns details about the conversation.
}, user_data);
if (ret != TIM_SUCC) {
// Failed to call the TIMConvCreate API
}
```

Sending messages

In the IM SDK, messages are represented by `TIMMessage`. Each `TIMMessage` consists of multiple `TIMElem` objects. A `TIMElem` object can be text or an image, which means a message can contain multiple objects including the text, image, and other objects.



Example:

```
const void* user_data = nullptr; // Return of the callback function
Json::Value json_value_text;
json_value_text[kTIMElemType] = kTIMElem_Text;
json_value_text[kTIMTextElemContent] = "Message Send to Windows-02";
Json::Value json_value_msg;
json_value_msg[kTIMMsgElemArray].append(json_value_text);
const char* userid = "Windows-02";
int ret = TIMMsgSendNewMsg(userid, kTIMConv_C2C, json_value_msg.toStyledString().c_str(), [](int32_t code, const char* desc, const char* json_param, const void* user_data) {
if (code != ERR_SUCC) {
// Failed to send the message
return;
}
// Sent the message successfully

}, user_data);
if (ret != TIM_SUCC) {
// Failed to call the TIMMsgSendNewMsg API
}
```

Receiving Messages

In most cases, users need to be notified of new messages. For this purpose, you simply need to register the new message notification callback `TIMMessageListener`. When the user logs in, offline messages will be pulled. To avoid missing message notifications, register the listener callback for new messages before login.

Example of setting a message listener

```
// Sets a new message listener. When a new message arrives, a callback is triggered through this listener.
const void *user_data = nullptr;
TIMSetRecvNewMsgCallback([](const char* json_msg_array, const void* user_data) {
Json::Value json_value_msgs; // Parse the message
Json::Reader reader;
if (!reader.parse(json_msg_array, json_value_msgs)) {
printf("reader parse failure!%s", reader.getFormattedErrorMessages().c_str());
return;
}
for (Json::ArrayIndex i = 0; i < json_value_msgs.size(); i++) { // Traverse messages
Json::Value& json_value_msg = json_value_msgs[i];
Json::Value& elems = json_value_msg[kTIMMsgElemArray];
for (Json::ArrayIndex m = 0; m < elems.size(); m++) { // Traverse Elems
Json::Value& elem = elems[i];
uint32_t elem_type = elem[kTIMElemType].asUInt();
if (elem_type == TIMElemType::kTIMElem_Text) { // Text

}else if (elem_type == TIMElemType::kTIMElem_Sound) { // Audio

} else if (elem_type == TIMElemType::kTIMElem_File) { // File

} else if (elem_type == TIMElemType::kTIMElem_Image) { // Image

}
}
}

}, user_data);
```

For more information on receiving and sending messages, see Sending Messages and Receiving Messages.

Group Management

There are multiple group types in IM. For information on their characteristics and limits, see the Group System. A group is identified by a unique ID, which allows for different operations. Group-

related operations are implemented by `TIMGroupManager`, which can be used after login.

Type	Description
Private	This is suitable for more private chat scenarios, where the group profile is not made public and a user can join the group only by being invited by a group member. Private groups are similar to groups in WeChat.
Public	This is suitable for public chat scenarios. Public groups have a strict management and admission mechanism.
ChatRoom	Chatroom members can join and quit freely.
AVChatRoom	This is similar to ChatRoom, but supports an unlimited number of group members.
BChatRoom	This is suitable for scenarios where messages are pushed to all online users.

Creating a group

The following example shows how to create a public group named Windows-Group-Name and invite user Windows_002 to the group.

Example:

```
Json::Value json_group_member_array(Json::arrayValue);
// Initial group members
Json::Value json_group_member;
json_group_member[kTIMGroupMemberInfoIdentifier] = "Windows_002";
json_group_member[kTIMGroupMemberInfoMemberRole] = kTIMGroupMemberRoleFlag_Member;
json_group_member_array.append(json_group_member);
Json::Value json_value_createparam;
json_value_createparam[kTIMCreateGroupParamGroupId] = "Windows-Group-01";
json_value_createparam[kTIMCreateGroupParamGroupType] = kTIMGroup_Public;
json_value_createparam[kTIMCreateGroupParamGroupName] = "Windows-Group-Name";
json_value_createparam[kTIMCreateGroupParamGroupMemberArray] = json_group_member_array;
json_value_createparam[kTIMCreateGroupParamNotification] = "group notification";
json_value_createparam[kTIMCreateGroupParamIntroduction] = "group introduction";
json_value_createparam[kTIMCreateGroupParamFaceUrl] = "group face url";
json_value_createparam[kTIMCreateGroupParamMaxMemberCount] = 2000;
json_value_createparam[kTIMCreateGroupParamAddOption] = kTIMGroupAddOpt_Any;
const void* user_data = nullptr;
int ret = TIMGroupCreate(json_param.c_str(), [](int32_t code, const char* desc, const char* json_
params, const void* user_data) {
if (code != ERR_SUCC) {
// Failed to create the group
return;
```

```
}

// Created the group successfully and parsed the JSON string to obtain the GroupID

}, user_data))
```

For more information on group operations, see Group APIs.

Group messages

Group messages are similar to C2C (one-to-one chat) messages and require you to enter the group ID and group type `kTIMConv_Group` when the message is sent. For more information, see Sending Messages in SDK documentation.

Initialization
Initialization (Android)

Last updated : 2020-12-30 14:47:29

Getting Communication Manager

Every operation in the IM SDK starts with `TIMManager`. Therefore the first step is to get a `TIMManager` singleton. The prototype of `getInstance` retrieval of a communication manager instance is as follows.

Prototype:

```
public static TIMManager getInstance()
```

Example:

```
TIMManager.getInstance();
```

Configuring IM SDK for Initialization

Before initializing the IM SDK, you need to configure some IM SDK settings, including SDKAppID and log control. The corresponding configuration class is `TIMSdkConfig`.

Log event

The IM SDK prints logs internally. If callers have their own unified log collection methods, they can utilize the `setLogListener` API in `TIMSdkConfig` to set a log event callback which returns logs to the callers. However, the IM SDK will still print logs internally in this case. You can disable printing by setting the console not to print logs, or setting the log level.

Prototype:

```
/**
* Setting the current log callback listener. This must be performed before IM SDK initialization.
* @param logListener Log callback listener
*/
public TIMSdkConfig setLogListener(TIMLogListener logListener)
```

Example:

```
//Set log callback. This API returns a copy of logs output by the IM SDK.
//[NOTE]: level is defined in TIMManager, for example, TIMManager.ERROR. It is not equivalent to the definition in Android systems.
mTIMSdkConfig.setLogListener(new TIMLogListener() {
@Override
public void log(int level, String tag, String msg) {
//You can output the SDK logs to your own logging system through this callback.
}
});
```

Setting the log level

When permissions allow, the IM SDK writes logs to log files by default. You can control the file log output of the IM SDK by modifying the internal write log level of the IM SDK through `setLogLevel` in `TIMSdkConfig` .

⚠ Note :

- The API for setting the write log level **must be called before IM SDK initialization** for the setting to take effect.
- You can disable the file log output of the IM SDK by setting the log level to TIMLogLevel.OFF. We recommend that you leave it enabled to facilitate troubleshooting.

Prototype:

```
/**
* Setting the write log level. This must be called before IM SDK initialization for the setting to take effect.
* @param logLevel Log level
*/
public TIMSdkConfig setLogLevel(@NonNull TIMLogLevel logLevel)
```

Disabling console log printing

By default, the IM SDK prints logs to the console. If this produces too much disruption, you can disable console logs through `enableLogPrint` in `TIMSdkConfig` (file logs will still be printed, but you can disable this by setting the log level).

⚠ Note :

> The API for log settings **must be called before IM SDK initialization** for the settings to take effect.

Prototype:

```
/**
* Enabling or disabling printing logs to the console. This must be set before IM SDK initializati
on.
* @param logPrintEnabled true - Logs will be output to the console
*/
public TIMSdkConfig enableLogPrint(boolean logPrintEnabled)
```

Modifying the log path

For the unified management of logs, you can modify the default log storage path. Use `setLogPath` in `TIMSdkConfig` to set the storage path for logs.

> ⚠ **Note :**
>
> - The API for setting the log path **must be called before IM SDK initialization** for the settings to take effect.
> - The default IM SDK log storage path is: `/tencent/imsdklogs/(your app package name)/` on the SD card.

Prototype:

```
/**
* Setting the log path. This must be called before IM SDK initialization for the settings to take
effect.
* @param logPath Log path
*/
public TIMSdkConfig setLogPath(@NonNull String logPath)
```

Initializing the IM SDK

Before using the IM SDK for further operations, you need to initialize the IM SDK.

> ⚠ **Note :**

When there are **multiple processes**, initialize the IM SDK in only one process. Call `SessionWrapper.isMainProcess(Context context)` to determine the correct process.

Prototype:

```
/**
* Initializing the IM SDK
* @param context application context
* @param config IM SDK global configuration
* @return true - initialization successful, false - initialization failed
*/
public boolean init(@NonNull Context context, @NonNull TIMSdkConfig config)
```

Example:

```
//Initialize the IM SDK basic configuration
//Determine whether this is the main thread
if (SessionWrapper.isMainProcess(getApplicationContext())) {
TIMSdkConfig config = new TIMSdkConfig(sdkAppId)
.enableCrashReport(false) //API has been deprecated
.enableLogPrint(true)
.setLogLevel(TIMLogLevel.DEBUG)
.setLogPath(Environment.getExternalStorageDirectory().getPath() + "/justfortest/");

//Initialize the SDK
TIMManager.getInstance().init(getApplicationContext(), config);
}
```

User Configuration

After the IM SDK has been initialized and before logging in to the IM SDK, configure a user through TIMUserConfig. After configuration and **before login**, bind the user configuration to the current communication manager through `setUserConfig` of `TIMManager`.

Prototype:

```
/**
* Setting the user configuration for the current user before login
* @param userConfig User configuration
*/
public void setUserConfig(TIMUserConfig userConfig)
```

Example:

```
//Basic user configuration
TIMUserConfig userConfig = new TIMUserConfig()
//Set listener for user status change events
.setUserStatusListener(new TIMUserStatusListener() {
@Override
public void onForceOffline() {
//Kicked offline by another client
Log.i(tag, "onForceOffline");
}

@Override
public void onUserSigExpired() {
//User signature expired, you must refresh userSig and log in to the IM SDK again
Log.i(tag, "onUserSigExpired");
}
})
//Set listener for connection status events
.setConnectionListener(new TIMConnListener() {
@Override
public void onConnected() {
Log.i(tag, "onConnected");
}

@Override
public void onDisconnected(int code, String desc) {
Log.i(tag, "onDisconnected");
}

@Override
public void onWifiNeedAuth(String name) {
Log.i(tag, "onWifiNeedAuth");
}
})
//Set listener for group events
.setGroupEventListener(new TIMGroupEventListener() {
@Override
public void onGroupTipsEvent(TIMGroupTipsElem elem) {
Log.i(tag, "onGroupTipsEvent, type: " + elem.getTipsType());
}
})
//Set listener for conversation refresh
.setRefreshListener(new TIMRefreshListener() {
@Override
public void onRefresh() {
Log.i(tag, "onRefresh");
```

```
}

@Override
public void onRefreshConversation(List<TIMConversation> conversations) {
Log.i(tag, "onRefreshConversation, conversation size: " + conversations.size());
}
});

//Disable all local storage
userConfig.disableStorage();
//Enable read receipt
userConfig.enableReadReceipt(true);

//Bind user configuration to communication manager
TIMManager.getInstance().setUserConfig(userConfig);
```

Network event notifications

This is an optional setting. To allow users to detect whether the IM SDK is connected to the server, set this callback through `TIMUserConfig` . It notifies the user whether the link between the caller and communication backend is connected or disconnected. Additionally, if the network is disconnected, the IM SDK will reconnect to the network after the network recovers and automatically pull messages to notify the user. The user does not need to worry about the network status. This is for notification purposes only.

> ⚠ **Note :**
>
> Here, network events do not indicate the user's local network status, but the connection status between the IM SDK and IM Cloud Server. As long as the user is logged in, the **IM SDK will reconnect internally upon disconnection, and no intervention by the user is required**.

Prototype:

```
/**
* Setting connection listener
* @param listener Connection listener
*/
public TIMUserConfig setConnectionListener(TIMConnListener listener)
```

Example:

See the example in User Configuration.

User status changes

The IM SDK sends notifications for user status changes. You can listen to notifications for various changes by setting a listener for user status change notifications through `TIMUserConfig`. Currently, there are two kinds of notifications. For more information, please see Force offline notifications and Ticket expiration notifications.

Prototype:

```
/**
* Set the user status notification callback
* @param userStatusListener User status notification callback
*/
public TIMUserConfig setUserStatusListener(TIMUserStatusListener userStatusListener)
```

The listener for user status change notifications, `TIMUserStatusListener`, is defined as follows:

```
/**
* User status change notification listener
*/
public interface TIMUserStatusListener {

/**
* Force offline callback
*/
public void onForceOffline();

/**
* Expired ticket callback
*/
public void onUserSigExpired();
}
```

Example:

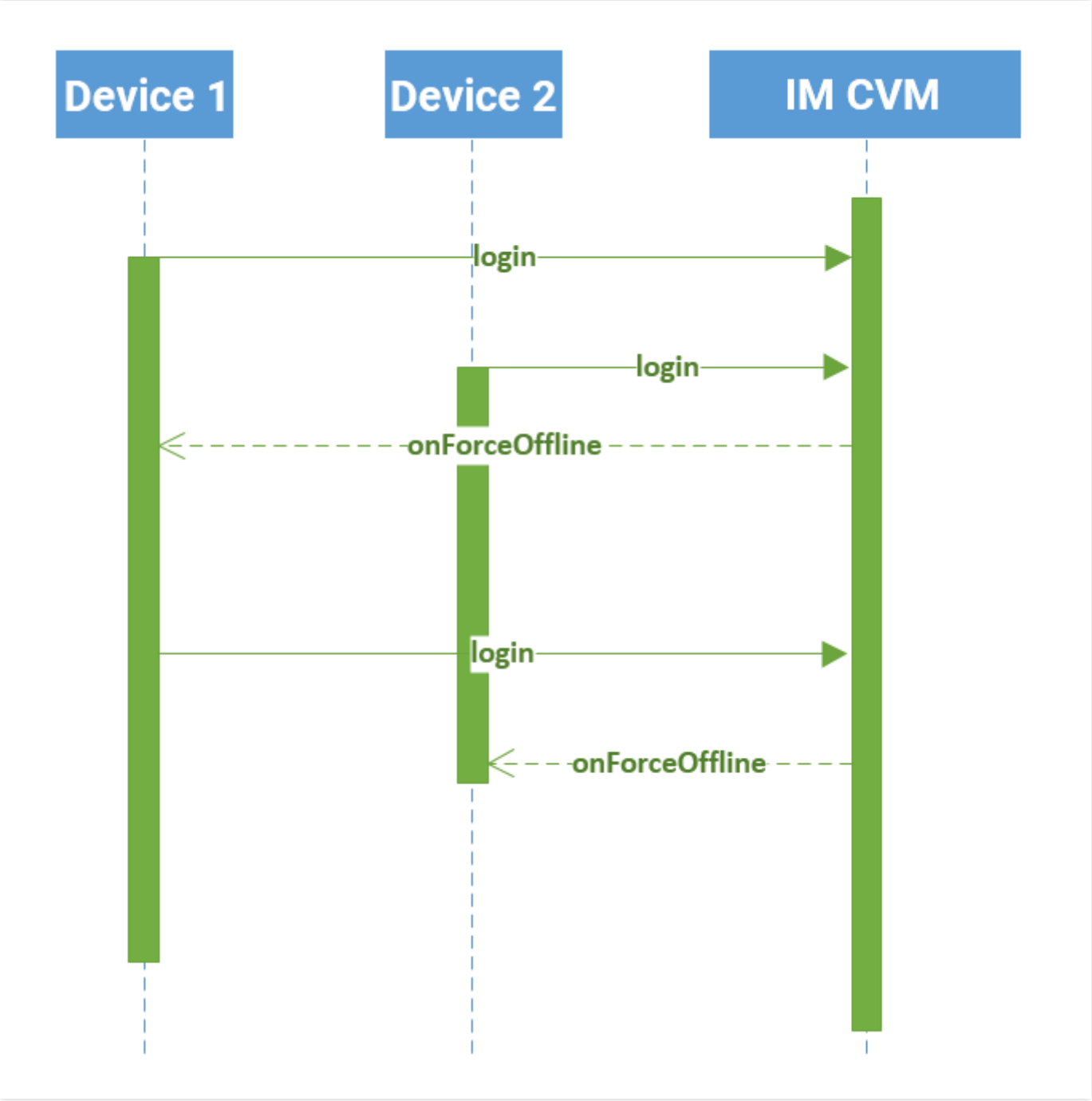
See the example in User Configuration.

Force offline notifications

The user will be forced to log out when logging in on another device. When this happens, the IM SDK sends a force offline notification. If a user status change notification listener has been set (see User status changes), the situation will be handled in the listener's callback method `onForceOffline`. Common practice is to prompt the user to log out or force the other party to log out.

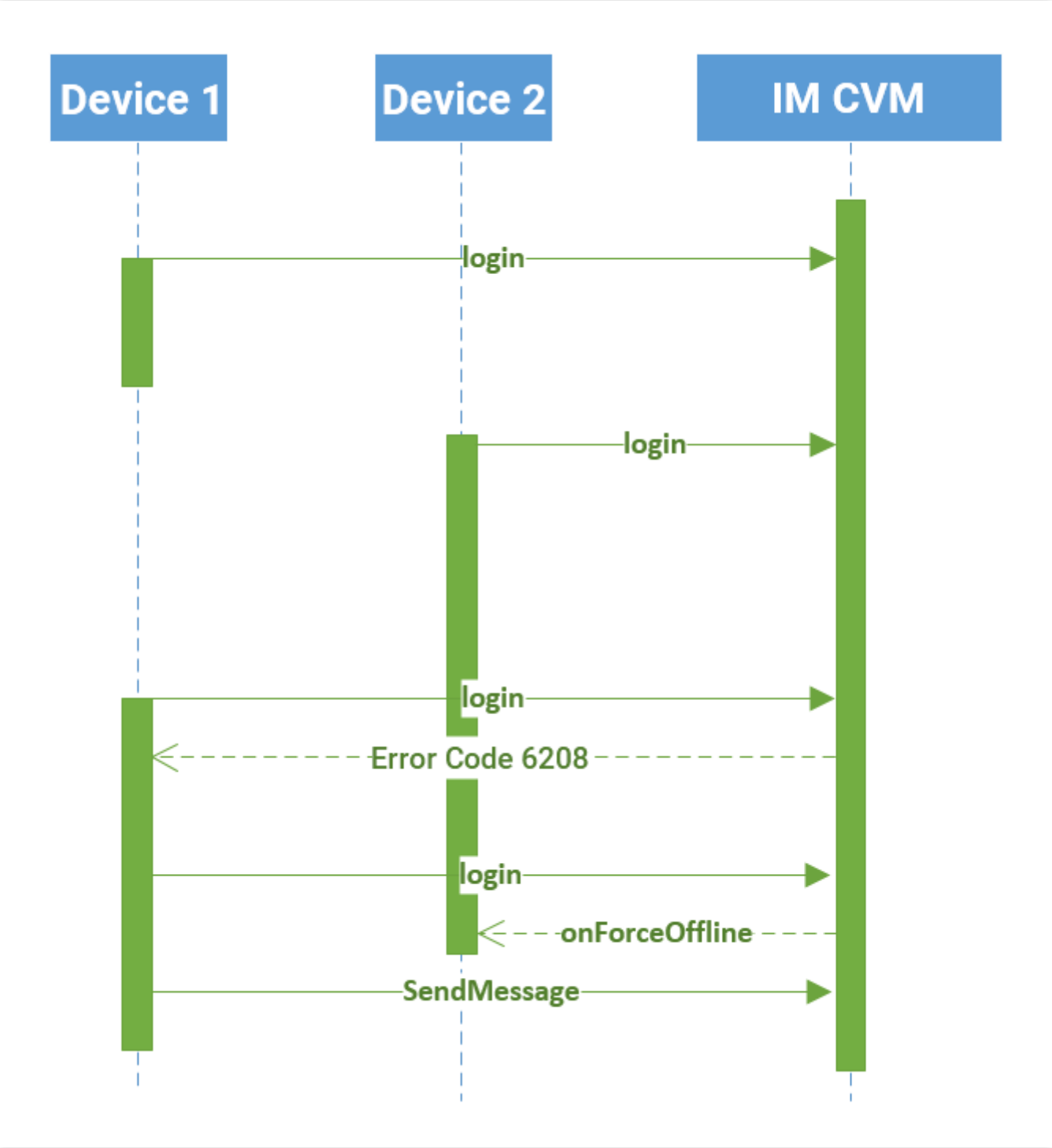
> ⚠ **Note :**
>
> If the user is logged out when offline, the subsequent login will fail and a strong alert (login error code ERR_IMSDK_KICKED_BY_OTHERS: 6208) is displayed to the user. Developers can also choose to ignore this error and let the user log in again.

The following diagram illustrates the force offline process in online scenarios. The user logs in on device 1, stays online, and then logs in on device 2. At this point, the user is logged out on device 1 and the `onForceOffline` callback is triggered. After receiving the callback on device 1, the user is prompted to call `login` to go back online and force device 2 to log out.



The following diagram illustrates the force offline process in offline scenarios. The user logs in on device 1 and kills the app process without calling `logout`. The user then logs in on device 2, but device 1 is unaware of this event because the app process has been killed. To explicitly alert the user and avoid imperceptible force offline, `ERR_IMSDK_KICKED_BY_OTHERS: 6208` is returned when the user tries to log in on device 1 again. This notifies the user of the force offline event and asks whether to kick the other party offline. To kick the other party offline, the user calls `login` again to force a login

and the logged-in instance on device 2 receives the `onForceOffline` callback.



Ticket expiration notifications

When the user logs in (see Login), a user ticket needs to be provided, which will expire after a certain period of time. If the user ticket has expired, the interaction between the SDK and the server fails and the SDK gives the user ticket expiration notification. If a user status change notification listener is set (see [User status change](#.E7.94.A8.E6.88.B7.E7.8A.B6.E6.80.81.E5.8F.98.E6.9B .B4)),

corresponding processing can be carried out in the listener's callback method `onUserSigExpired`. To continue interacting with the server, the user must change the ticket and log in again.

Disabling storage

By default, the IM SDK stores messages, profiles, conversations, and other information. If you do not need to store this information, disable storage through `TIMUserConfig` to improve processing performance.

> ⚠ **Note :**
>
> The API for disabling local storage **must be called before login**.

Prototype:

```
/**
* Disabling local storage
*/
public TIMUserConfig disableStorage()
```

Conversation refresh listener

By default, C2C offline messages and recent contacts will be obtained asynchronously and profile data will be synced after login. When synchronization is completed, the `onRefresh` callback in the conversation refresh listener `TIMRefreshListener` sends an interface refresh notification. Upon receiving this message, the user can refresh the interface (for example, refresh unread messages in the conversation list).

> ⚠ **Note :**
>
> If offline messages are not needed, you can just send online messages.

In the event of multi-device login, unread count synchronization notifications are delivered by the server. The IM SDK updates the unread count locally and then notifies the user to update conversations. The notification will initiate a callback through `onRefreshConversation` in `TIMRefreshListener`. Users who require multi-device synchronization can perform relevant synchronous processing in this API. Therefore, we recommend using `setRefreshListener` in `TIMUserConfig` to configure a conversation refresh listener.

Prototype:

```
/**
* Setting data refresh notification listener
* @param listener Data refresh notification listener
*/
public TIMUserConfig setRefreshListener(TIMRefreshListener listener)
```

Listener for message recall notifications

A message recall feature is available in IM SDK 3.1.0 and later versions. A message recall listener can be set through `setMessageRevokedListener` of `TIMUserConfig`.

Prototype:

```
/**
* Setting message recall notification listener
* @param listener Message recall notification listener
* @since 3.1.0
*/
public TIMUserConfig setMessageRevokedListener(@NonNull TIMMessageRevokedListener listener)
```

New Message Notifications

In most cases, users need to be notified of new messages. Therefore, register the new message notification callback `TIMMessageListener`. When the user logs in, C2C offline messages and recent contacts will be pulled. To ensure that users do not miss message notifications, we recommend registering new message notifications before login

> ⚠ **Note :**
>
> The IM SDK calls back all messages that are not stored locally to the upper-level app through registered message notifications.

The following is a message listener prototype. By default, all message listeners will be called back according to the order in which they were added until the `onNewMessages` callback returns true. Then, the next message listener is not called back.

Prototype:

```
/**
* Adding a message listener
* @param listener Message listener
```

```
* By default, all message listeners will be called back according to the order in which they were added.
* However, if the `onNewMessages` callback returns true, the next message listener will not be called back.
*/
public void addMessageListener(TIMMessageListener listener)
```

The following is a new message receipt callback:

```
/**
* New message receipt callback
* @param msgs New messages received
* @return Normally, if multiple listeners are registered, the IM SDK calls back all listeners in sequence. When the callback of a listener returns true, the subsequent listeners will not be called back.
*/
public boolean onNewMessages(List<TIMMessage> msgs)
```

A deleted listener will not be called. **The following is the prototype for message listener deletion:**

```
public void removeMessageListener(TIMMessageListener listener)
```

The content of messages that are called back is passed through the parameter `TIMMessage` . With `TIMMessage` , you can get detailed information about messages and conversations, such as message text, audio data, and images, please see Sending and Receiving Messages (Android).

Example:

```
//Set message listener. When new messages arrive, callback is initiated through this listener.
TIMManager.getInstance().addMessageListener(new TIMMessageListener() {//Message listener
@Override
public boolean onNewMessages(List<TIMMessage> msgs) {//New message received
//For the parsing of message content, see the message parsing description in the Receiving and Sending Messages documentation.
return true; //When true is returned, the callback chain stops and the next new message listener will not be called.
}
});
```

Initialization (iOS)

Last updated : 2021-01-25 17:25:01

Initializing Communication Manager

Every operation in the IM SDK starts with `TIMManager`. Therefore, the first step is to get a `TIMManager` singleton.

Prototype:

```
@interface TIMManager : NSObject
/**
* Get manager instance
*
* @return manager instance
*/
+(TIMManager*)sharedInstance;
@end
```

Example:

```
TIMManager * manager = [TIMManager sharedInstance];
```

Before using the SDK for further operations, you need to initialize the SDK.

Prototype:

```
@interface TIMManager : NSObject

/**
* Initialize the SDK
*
* @param config Configuration information, globally effective
*
* @return 0 Success
*/
- (int)initSdk:(TIMSdkConfig*)globalConfig;

/**
* Initialize the current manager, call after initSdk: and before login:
*
* @param config Configuration information, valid for the current TIMManager
*
```

```
 * @return 0 Success
 */
- (int)setUserConfig:(TIMUserConfig*)config;

@end

//Global configuration information
@interface TIMSdkConfig : NSObject

//User identifier app ID for connecting to the SDK (required)
@property(nonatomic,assign) int sdkAppId;

//Forbid the console from printing logs
@property(nonatomic,assign) BOOL disableLogPrint;

//Local write log file level. The default level is DEBUG.
@property(nonatomic,assign) TIMLogLevel logLevel;

//Log file path. The default path is used when this parameter is not set. The log path can be obt
ained through TIMManager -> getLogPath.
@property(nonatomic,strong) NSString * logPath;

//Log level that is called back to the logFunc function. The default level is DEBUG.
@property(nonatomic,assign) TIMLogLevel logFuncLevel;

//Log listener function
@property(nonatomic,copy) TIMLogFunc logFunc;

//Message database path. The default path is used when this parameter is not set.
@property(nonatomic,strong) NSString * dbPath;

//Network listener that listens for the successful and failed status of network connections
@property(nonatomic,strong) id<TIMConnListener> connListener;

@end

//User configuration information
@interface TIMUserConfig : NSObject

//Disable local storage
@property(nonatomic,assign) BOOL disableStorage;

//Whether or not to enable multi-client unread synchronization notification. This option modifies
the unread notification logic in the case of multi-client login. YES: only when one client calls
setReadMessage() to mark the message as read, the other client will not receive an unread notific
ation. NO: once the message is received by one client, the other client will not receive an unrea
d notification. In the same way, these unread messages will not be received after the app is unin
stalled and installed again.
```

```
@property(nonatomic,assign) BOOL disableAutoReport;

//Whether or not to enable read receipts. YES: after the recipient reads the message (setReadMess
age), the sender will receive the TIMMessageReceiptListener callback notification. NO: do not ena
ble read receipts. This is the default setting.
@property(nonatomic,assign) BOOL enableReadReceipt;

//Set the group profiles to be pulled by default. To pull custom fields, configure "Custom Field
s" and corresponding user operation permissions in IM Console -> Feature Configuration -> Group
Custom Fields. The configuration takes effect after 5 minutes.
@property(nonatomic,strong) TIMGroupInfoOption * groupInfoOpt;

//Set the group member profiles to be pulled by default. To pull custom fields, configure "Custo
m Fields" and corresponding user operation permissions in IM Console -> Feature Configuration ->
Group Member Custom Fields. The configuration takes effect after 5 minutes.
@property(nonatomic,strong) TIMGroupMemberInfoOption * groupMemberInfoOpt;

//Relationship chain parameter
@property(nonatomic,strong) TIMFriendProfileOption * friendProfileOpt;

//User login status listener that listens for force offline, network reconnection failure, and Us
erSig expiration notifications
@property(nonatomic,weak) id<TIMUserStatusListener> userStatusListener;

//Conversation refresh listener that listens for the refresh of conversations
@property(nonatomic,weak) id<TIMRefreshListener> refreshListener;

//Read receipts listener that listens for the read receipts of messages. The enableReadReceipt fi
eld must be set to YES.
@property(nonatomic,weak) id<TIMMessageReceiptListener> messageReceiptListener;

//Message update listener that listens for message status changes
@property(nonatomic,weak) id<TIMMessageUpdateListener> messageUpdateListener;

//Message recall listener that listens for message recall notifications in conversations
@property(nonatomic,weak) id<TIMMessageRevokeListener> messageRevokeListener;

//File upload progress listener that listens for the upload progress of audio, image, video, and
file messages which are uploaded to the server before being sent
@property(nonatomic,weak) id<TIMUploadProgressListener> uploadProgressListener;

//Group event notification listener
@property(nonatomic,weak) id<TIMGroupEventListener> groupEventListener;

//Listener for locally cached relationship chain data
@property(nonatomic,weak) id<TIMFriendshipListener> friendshipListener;

@end
```

New Message Notifications

In most cases, users need to be notified of new messages. Therefore, register the new message notification callback `TIMMessageListener`. When the user logs in, offline messages will be pulled. So that users do not miss message notifications, register new message notifications before login.

Prototype:

```
/**
* Callback for receiving new messages
*/
@protocol TIMMessageListener <NSObject>
@optional
/**
* New message callback notification
*
* @param msgs List of new messages, an array of TIMMessage types
*/
- (void)onNewMessage:(NSArray*) msgs;
@end

@interface TIMManager : NSObject

/**
* Add message callback (ignores repeated adding)
*
* @param listener Callback
*
* @return Successful
*/
- (int)addMessageListener:(id<TIMMessageListener>)listener;

@end
```

Callback message content is passed through `TIMMessage`. With `TIMMessage`, you can get detailed information about messages and the corresponding conversations, such as text, audio data, and images. The following example sets a message callback notification and prints new messages directly. For more information, see Parsing messages.

Example:

```
@interface TIMMessageListenerImpl : NSObject
- (void)onNewMessage:(TIMMessage*) msg;
```

```
@end
@implementation TIMMessageListenerImpl
- (void)onNewMessage:(NSArray*) msgs {
NSLog(@"NewMessages: %@", msgs);
}
@end
TIMMessageListenerImpl * impl = [[TIMMessageListenerImpl alloc] init];
[[TIMManager sharedInstance] addMessageListener:impl];
```

Network Event Notifications

This setting is optional. To allow users to detect whether the IM SDK is connected to the server, set this callback function. It notifies the user of the connection and disconnection events between the caller and the communication backend. If the network connection is interrupted, the IM SDK will reconnect to the network after the network recovers and automatically pull messages to notify the user. The user does not need to worry about the network status. This is for notification purposes only.

> **Note:**
> Here, network events do not indicate the user's local network status, but whether the IM SDK is connected to the IM cloud server. As long as the user is logged in, **IM SDK will reconnect internally, and no intervention by the user required.**

Prototype:

```
/**
* Connection notification callback
*/
@protocol TIMConnListener <NSObject>
@optional
/**
* Network connection successful
*/
- (void)onConnSucc;
/**
* Network connection failed
*
* @param code Error code
* @param err Error description
*/
- (void)onConnFailed:(int)code err:(NSString*)err;
```

```
/**
* Network disconnected. Users are notified of the disconnection but do not need to log in again.
Users return to the online status automatically once the network is reconnected.
*
* @param code Error code
* @param err Error description
*/
- (void)onDisconnect:(int)code err:(NSString*)err;
/**
* Connecting
*/
- (void)onConnecting;
@end
@interface TIMSdkConfig : NSObject
/**
* Network listener
*/
@property(nonatomic,retain) id<TIMConnListener> connListener;
@end
```

The following example listens for network events and outputs logs.

Example:

```
@interface TIMConnListenerImpl : NSObject
- (void)onConnSucc;
- (void)onConnFailed:(int)code err:(NSString*)err;
- (void)onDisconnect:(int)code err:(NSString*)err;
@end
@implementation TIMConnListenerImpl

- (void)onConnSucc {
NSLog(@"Connect Succ");
}
- (void)onConnFailed:(int)code err:(NSString*)err {
    // code Error code: see the Error Code Table for more information
NSLog(@"Connect Failed: code=%d, err=%@", code, err);
}
- (void)onDisconnect:(int)code err:(NSString*)err {
    // code Error code: see the Error Code Table for more information
NSLog(@"Disconnect: code=%d, err=%@", code, err);
}
@end
TIMConnListenerImpl * connListenerImpl = [[TIMConnListenerImpl alloc] init];
TIMSdkConfig * cfg = [[TIMSdkConfig alloc] init];
cfg.connListener = connListenerImpl;
[[TIMManager sharedInstance] initSdk:cfg];
```

Log Events

The IM SDK prints logs internally. If callers have their own unified log collection methods, they can set log event callbacks, which are called by the SDK to return logs to the callers. Callbacks can be called through **blocks** or the `protocol` **API**.
After callbacks are set, the IM SDK will still print logs internally. You can disable this by setting the console to not print logs, or setting the log level.

Prototype:

```
@interface TIMSdkConfig : NSObject
/**
* log listener function
*/
@property(nonatomic,copy) TIMLogFunc logFunc;
@end
```

The following example uses block to call back printing logs to the console.
Example:

```
TIMSdkConfig * cfg = [[TIMSdkConfig alloc] init];
cfg.logFunc = ^(NSString* content) {
NSLog(@"%@", content);
}];
```

User Status Changes

The SDK sends notifications for user status changes. You can listen to notifications for various changes by setting listeners for user status change notifications using the `userStatusListener` property in `TIMUserConfig` . Currently, there are three kinds of notifications. For more information, see User force offline notifications and User ticket expiration notifications. In this case, users need to log in again to use messages, groups, and friends features normally.

Prototype:

```
/**
* User online status notification
*/
@protocol TIMUserStatusListener <NSObject>
@optional
/**
```

```
* Force offline notification
*/
- (void)onForceOffline;
/**
* Reconnection failed
*/
- (void)onReConnFailed:(int)code err:(NSString*)err;
/**
* The userSig expired (Obtain a new userSig to log in)
*/
- (void)onUserSigExpired;
@end
@interface TIMUserConfig : NSObject
/**
* User login status listener
*/
@property(nonatomic,retain) id<TIMUserStatusListener> userStatusListener;
@end
```

The following example prints logs after receiving a force offline event callback. **Example:**

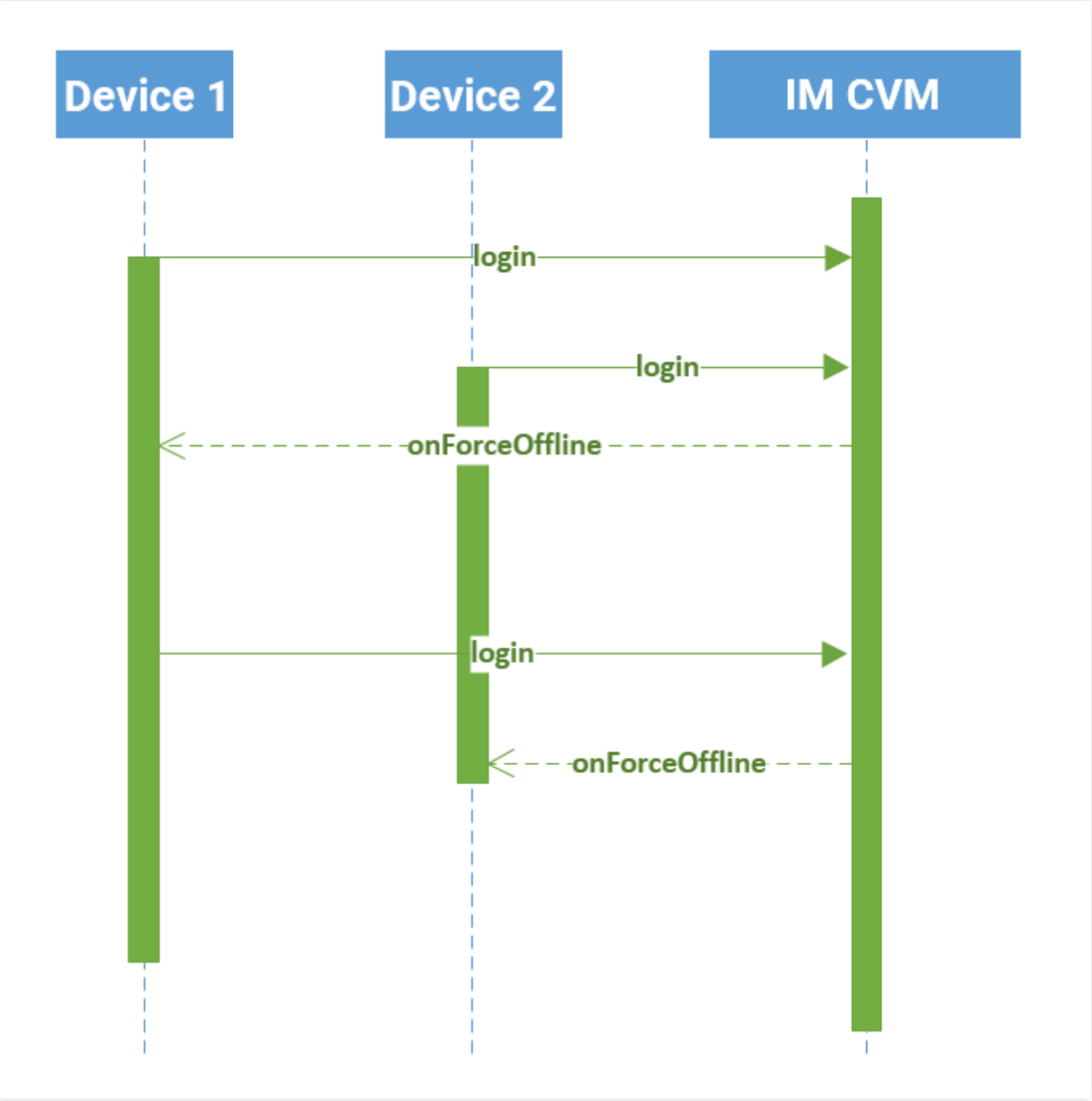
```
@interface TIMUserStatusListenerImpl : NSObject{
}
- (void)onForceOffline;
- (void)onUserSigExpired;
@end
@implementation TIMUserStatusListenerImpl
- (void)onForceOffline {
NSLog(@"force offline");
}
- (void)onUserSigExpired {
NSLog(@"userSig expired");
}
@end
TIMUserStatusListenerImpl * impl = [[TIMUserStatusListenerImpl alloc] init];
TIMUserConfig * cfg = [[TIMUserConfig alloc] init];
cfg.userStatusListener = impl;
```

Force offline notifications

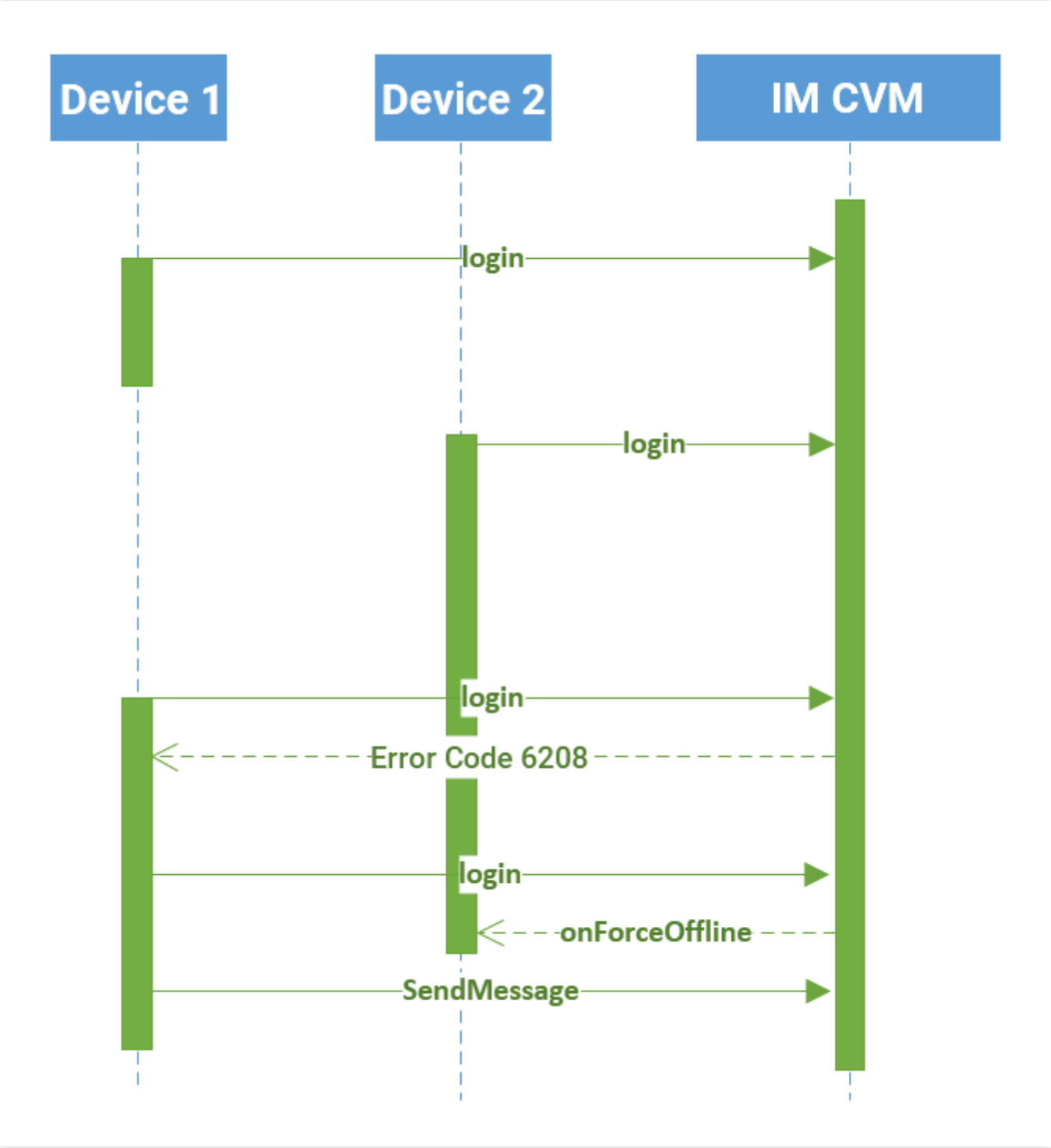
The user will be forced to log out when calling login to log in on another device. When this happens, the SDK sends a force offline notification. If a user status change notification listener has been set (see User Status Changes), the situation will be handled in the listener's callback method `onForceOffline` . Common practice is to prompt the user to log out or force the other party to log out by calling login again.

Note:
If the user is logged out when offline, the subsequent login by calling login will fail with a strong alert and a strong alert (login error code `ERR_IMSDK_KICKED_BY_OTHERS: 6208`) is displayed to the user. Developers can also choose to ignore this error and let the user log in again.

The following diagram illustrates the **force offline process in online scenarios**. The user logs in on device 1 by calling login, stays online, and then logs in on device 2 by calling login. At this point, the user is logged out on device 1 and receives the `onForceOffline` callback. After receiving the callback on device 1, the user is prompted to call `login` to go back online and force device 2 to log out.



The following diagram illustrates the **force offline process in offline scenarios**. The user logs in on device 1 by calling login and the process exits without calling `logout`. The user then logs in on device 2 by calling login, but device 1 is unaware of this event because the user is not online. To explicitly alert the user and avoid imperceptible force offline, `ERR_IMSDK_KICKED_BY_OTHERS: 6208` is returned when the user tries to log in on device 1 again, notifying the user of the force offline event and asking whether to kick the other party offline. To kick the other party offline, the user calls `login` again to force a login and the logged-in instance on device 2 receives the `onForceOffline` callback.



User ticket expiration notifications

When the user logs in (see Login), a user ticket needs to be provided, which will expire after a certain period of time. If the user ticket has expired, the interaction between the SDK and the server fails and the SDK gives the user ticket expiration notification. If a user status change notification listener is set (see User Status Changes), corresponding processing can be carried out in the listener's

callback method `onUserSigExpired`. To continue interacting with the server, the user must change the ticket and log in again.

Setting Log Level

You can modify the IM SDK internal log level by configuring `TIMSdkConfig`. You can disable IM SDK log output by setting the log level to TIMLogLevel.OFF. We recommend that you leave it enabled to facilitate troubleshooting.

Prototype:

```
@interface TIMSdkConfig : NSObject
/**
* Local write log file level. The default level is DEBUG.
*/
@property(nonatomic,assign) TIMLogLevel logLevel;
@end
```

Disabling Console Log Printing or Modifying the Log Path

By default, the IM SDK prints logs to the console. If this produces too much disruption during testing and debugging, you can disable console logs (file logs will still be printed, but you can disable this by setting the log level). If you do not modify the log path but only the level, use `getLogPath` to get the default path to pass to `initLogSettings`. The default log path is `Library/Caches/imsdk_YYYYMMDD.log` under the app directory.

Prototype:

```
@interface TIMSdkConfig : NSObject
/**
* Log file path. The default path is used if it is not set.
*/
@property(nonatomic,retain) NSString * logPath;
/**
* Forbid the console from printing logs
*/
@property(nonatomic,assign) BOOL disableLogPrint;
@end
```

Login
Login (Android)

Last updated : 2020-12-30 11:46:29

Login

Users can normally send and receive messages only after they have logged in to the Tencent backend server. To log in to the Tencent backend server, a user needs to provide information including `UserID` and `UserSig` . For more information, see Login Authentication.

⚠ Note :

- If the user is forced logout on another terminal, the login attempt fails, and the error code (`ERR_IMSDK_KICKED_BY_OTHERS: 6208`) is returned. In this case, developers must analyze the cause to the login error code `ERR_IMSDK_KICKED_BY_OTHERS` . For details on forcible logout, see User State Changes.
- If users have saved user tickets, these tickets may expire. If their user tickets expire, `login` returns the error code `70001` . In this case, developers can change the ticket based on the error code.

Login is an asynchronous process, and the result returned by the callback function indicates whether the login was successful. Users can proceed to subsequent operations only after successful login.

Prototype:

```
/** Login
* @param identifier User account
* @param userSig UserSig, which indicates the user account signature resulting from private key e
ncryption. For details, see the relevant document.
* @param callback Callback API
*/
public void login(@NonNull String identifier, @NonNull String userSig, @NonNull TIMCallBack callb
ack)
```

Example:

```
// identifier indicates the username, and userSig indicates the user login credential.
TIMManager.getInstance().login(identifier, userSig, new TIMCallBack() {
@Override
```

```
public void onError(int code, String desc) {
//"code" (error code) and "desc" (error description) can be used to locate the cause of the reque
st failure.
//For the list of error codes, see the error code table.
Log.d(tag, "login failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess() {
Log.d(tag, "login succ");
}
});
```

Logout

To log out or switch to another user, call the logout operation.

Prototype:

```
/**
* Logout
* @param callback Callback, which is null if it is not needed
*/
public void logout(@Nullable TIMCallBack callback)
```

Example:

```
//Logout
TIMManager.getInstance().logout(new TIMCallBack() {
@Override
public void onError(int code, String desc) {

//"code" (error code) and "desc" (error description) can be used to locate the cause of the reque
st failure.
//For the list of error codes, see the error code table.
Log.d(tag, "logout failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess() {
//Successful logout
}
});
```

Viewing Messages Without Network Connection

If the current network is abnormal or you want to view user messages without calling `login`, you can call the `initStorage` method of `TIMManager` to initialize storage. After that, you can obtain the conversation list and messages.

> ⚠ **Note :**
>
> - This method is for viewing historical messages only when the login attempt fails or no network connection is available. **To receive and send messages, you must call the login API** `login`.
> - If the login attempt succeeds, the IM SDK automatically initializes local storage, without the need to manually call this API.

Prototype:

```
/** Initialize local storage to load local conversations and messages without network connection.
* @param identifier User ID
* @param cb Callback
*/
public int initStorage(@NonNull String identifier, @NonNull TIMCallBack cb)
```

The following example shows how to initialize storage. If the initialization succeeds, the conversation list can be obtained.

Example:

```
//Initialize local storage
TIMManager.getInstance().initStorage(identifier, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "initStorage failed, code: " + code + "|descr: " + desc);
}

@Override
public void onSuccess() {
Log.i(tag, "initStorage succ");
}
});

//Obtain a conversation instance
TIMConversation conversation = TIMManager.getInstance().getConversation(TIMConversationType.C2C,
peer);
```

```
//Obtain local messages
conversation.getLocalMessage(5, null, new TIMValueCallBack<List<TIMMessage>>() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "get msgs failed, code: " + code + "|msg: " + desc);
}

@Override
public void onSuccess(List<TIMMessage> timMessages) {
Log.i(tag, "get msgs succ, size: " + timMessages.size());
}
});
```

Obtaining the Currently Logged-in User

The `getLoginUser` method of `TIMManager` can be used to obtain the current username and check whether the user has logged in.

Prototype:

```
public String getLoginUser()
```

> ⚠ **Note :**
>
> The returned value is the username of the currently logged-in user. Note that if the logged-in account is a self-owned account, the username is the same as the `identifier` that was passed in during login. If the user logged in with a third-party account, such as a WeChat or QQ account, an internally converted `identifier` will be generated after login. Subsequent operations, such as searching for friends and joining groups, require the converted `identifier` .

Login (iOS)

Last updated : 2021-01-25 17:58:32

Login

Users can normally send and receive messages only after they have logged in to the Tencent backend server. To log in to the Tencent backend server, a user needs to provide `UserID` and `UserSig` . If users have saved user tickets, these tickets may expire. If their user tickets expire, `login` returns the error code `6206` . In this case, developers can change the ticket based on the error code. Login is an asynchronous process, and the result returned by the callback function indicates whether the login was successful. Users can proceed to subsequent operations only after successful login. The `succ` and `fail` blocks are used for the callback upon successful or failed login, respectively.

⚠ Note :

- If the user is forced logout on another terminal, the login attempt fails, and the error code (`ERR_IMSDK_KICKED_BY_OTHERS: 6208`) is returned. In this case, developers must analyze the cause to the login error code `ERR_IMSDK_KICKED_BY_OTHERS` . For details on forcible logout, see User State Changes.
- After successful login, as long as users do not log out or are not forced logout and automatic network reconnection upon connectivity change is supported, developers need not be concerned. However, they should pay special attention to situations where users are forced logout. Therefore, the callback for a user state change must be registered, otherwise no notification can be received when forcible logout occurs.

Prototype:

```
/**
* Login information
*/
@interface TIMLoginParam : NSObject
/**
* Username
*/
@property(nonatomic,retain) NSString* identifier;
/**
* Authentication token
*/
```

```
@property(nonatomic,retain) NSString* userSig;

@end
```

```
/**
* Login
*
* @param param Login parameter
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Request succeeded
*/
- (int)login:(TIMLoginParam*)param succ:(TIMLoginSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
param	Login parameter. For details, see the description of the TIMLoginParam structure.
succ	Callback for successful login
fail	Callback for failed login

Example:

```
TIMLoginParam * login_param = [[TIMLoginParam alloc ]init];
// identifier Username
login_param.identifier = @"iOS_001";
//userSig User login credential
login_param.userSig = @"usersig";

[[TIMManager sharedInstance] login: login_param succ:^(){
NSLog(@"Login succ");
} fail:^(int code, NSString * err) {
NSLog(@"Login Failed: %d->%@", code, err);
}];
```

For the correct method of issuing UserSig, see Login Authentication.

Logout

To log out or switch to another user, call the logout operation.

Prototype:

```
@interface TIMManager : NSObject
/**
* Logout
*
* @param succ Success callback, which indicates that the logout succeeded
* @param fail Failure callback, in which case the error code and error information are returned
*
* @return 0 Logout package was sent successfully, and the callback is pending
*/
- (int)logout:(TIMLoginSucc)succ fail:(TIMFail)fail;
@end
```

Example:

> ⚠ **Note :**
>
> When you need to switch to another account, `login` can be called again only after the `logout` callback succeeds or fails. Otherwise, `login` may fail.

```
[[TIMManager sharedInstance] logout:^() {
NSLog(@"logout succ");
} fail:^(int code, NSString * err) {
NSLog(@"logout fail: code=%d err=%@", code, err);
}];
```

Viewing Messages Without Network Connection

If the current network is abnormal or you want to view user messages without calling `login`, you can call the `initStorage` method to initialize storage. After that, you can obtain the conversation list and messages.

Prototype:

```
@interface TIMManager : NSObject
/**
* Initializes storage. This is only for viewing historical messages. For operations such as receiving or sending messages, this function does not need to be called if login succeeded.
*
```

```
* @param userID Username
* @param succ Success callback. You can obtain the conversation list and messages when receiving
this callback.
* @param fail Failure callback
*
* @return 0 Request succeeded
*/
- (int)initStorage:(NSString*)userID succ:(TIMLoginSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
userID	Username
succ	Success callback, in which case you can obtain the conversation list and perform further login
fail	Failure callback

The following sample shows how to initialize storage. If the initialization succeeds, the conversation list can be obtained. **Example:**

```
TIMLoginParam * login_param = [[TIMLoginParam alloc ]init];
[[TIMManager sharedInstance] initStorage: @"iOS_001" succ:^(){
NSLog(@"Init Succ");
} fail:^(int code, NSString * err) {
NSLog(@"Init Failed: %d->%@", code, err);
}];
```

Obtaining the Currently Logged-in User

The `getLoginUser` method of `TIMManager` can be used to obtain the current username and check whether the user has logged in. The return value is the username of the currently logged-in user. Note that if the logged-in account is a self-owned account, the username is the same as the `UserID` that was passed in during login. If the user logged in with a third-party account, such as a WeChat or QQ account, an internally converted `UserID` will be generated after login. Subsequent operations, such as searching for friends and joining groups, require the converted `UserID` .

Prototype:

```
@interface TIMManager : NSObject

/**
* Obtain the currently logged-in user
*
* @return Return the logged-in user's identifier. If no user is logged in, 'nil' is returned.
*/
- (NSString*)getLoginUser;

@end
```

Synchronizing Offline Messages by the IM SDK

The IM SDK automatically synchronizes offline messages and recent contacts after startup. If offline messages are not needed, you can send messages by referring to sending online messages. By default, after login, the IM SDK asynchronously obtains offline messages and synchronizes profile data (if this feature is enabled, and you can see the section about relationship chain profiles for more information). After the synchronization is completed, the IM SDK notifies users of UI updates through the `onRefresh` callback. After receiving the notification, users can update the UI, for example, to view unread messages in the conversation list.

```
@interface TIMUserConfig : NSObject
/**
* Conversation refreshment listener
*/
@property(nonatomic,retain) id<TIMRefreshListener> refreshListener;
@end
```

Login (Web & Mini Program)

Last updated : 2020-06-01 18:51:44

Login

You can send and receive messages in the Instant Messaging (IM) console only after logging in to the IM SDK. To log in to the IM SDK, you need to provide information including the UserID and UserSig. For more information, see Login Authentication. After successful login, to call APIs that require authentication, such as sendMessage, you must wait until the SDK enters the ready state. You can obtain the status of the SDK by listening on events. For more information, see TIM.EVENT.SDK_READY.

> By default, multi-device login is not supported. If you use an account that has been logged in on another page to log in on the current page, the previous page may be forcibly logged out. When the previous page is forcibly logged out, the event `TIM.EVENT.KICKED_OUT` is triggered. You can proceed accordingly after detecting the event through listening. The following shows an example of listening on multi-device login:

```
let onKickedOut = function (event) {
console.log(event.data.type); // mutipleAccount (The same account that is used to log in on multi
ple pages on the same device is forcibly logged out.)
};
tim.on(TIM.EVENT.KICKED_OUT, onKickedOut);
```

To support multi-device login (which means that the same account can be used to concurrently log in on multiple pages), log in to the Instant Messaging Console, and then click the SDKAppID of the desired app in My IM Apps. Choose **Feature Configuration** -> **Login and Message**. On the Login and Message page, click **Edit** in the Login settings area. Set **Online Web Instances** as required. The configuration will take effect within 50 minutes.

API name

```
tim.login(options)
```

Request parameters

Name	Type	Description

Name	Type	Description
UserID	String	The ID of the user.
UserSig	String	The password with which the user logs in to the IM console. It is essentially the ciphertext generated by encrypting the information such as the UserID. For the detailed generation method, see Generating UserSig.

Returned values

This API returns a `Promise` object.

Sample

```
let promise = tim.login({userID: 'your userID', userSig: 'your userSig'});
promise.then(function(imResponse) {
console.log(imResponse.data); // Login succeeded.
}).catch(function(imError) {
console.warn('login error:', imError); // Information about login failure.
});
```

Logout

This API is usually called when you switch between accounts. It clears the login status of the current account and all the data in the memory.

- When calling this API, the instance publishes the SDK_NOT_READY event. In this case, the instance is automatically logged out and cannot receive or send messages.
- If the value of **Online Web Instances** configured in the Instant Messaging Console is greater than 1, and the same account has been used to log in to instances `a1` and `a2` (including Mini Program instances), after `a1.logout()` is executed, `a1` is automatically logged out and cannot receive or send messages, whereas `a2` is not affected.
- Multi-device login: if **Online Web Instances** is set to 2 and your account has been used to log in to instances `a1` and `a2`, when you use this account to log in to instance `a3`, either `a1` or `a2` will be forcibly logged out. In most cases, the instance that first enters the login state is forcibly logged out. Assuming that `a1` is forcibly logged out, a logout process is internally executed in `a1` and the KICKED_OUT event is triggered. The access side can listen on this event and redirect to the login page when the event is triggered. In this case,

instance `a1` is forcibly logged out, whereas instances `a2` and `a3` can continue to run properly.

API name

```
tim.logout();
```

Request parameters

None.

Returned values

This API returns a `Promise` object. The callback functions are as follows:

- The callback function parameter for `then` is IMResponse. `IMResponse.data` is a null object, indicating that logout succeeded.
- The callback function parameter for `catch` is IMError.

Sample

```
let promise = tim.logout();
promise.then(function(imResponse) {
console.log(imResponse.data); // Logout succeeded.
}).catch(function(imError) {
console.warn('logout error:', imError);
});
```

Group Management
Group Management (Android)

Last updated : 2021-02-05 15:34:45

Group Overview

Instant Messaging (IM) supports multiple group types. For more information on their characteristics and limits, see Group System. A group is identified by a unique ID that enables different operations.

Group Messages

Group messages and C2C (one-to-one) messages are the same except for the conversation type obtained through Conversation. For more information, see Sending Messages.

Group Management

Group-related operations are all implemented by `TIMGroupManager`. You must log in before performing such operations.

Getting a singleton prototype:

```
/** Getting an instance
* @return TIMGroupManager instance
*/
public static TIMGroupManager getInstance()
```

Creating a group

IM provides built-in group types including **private group (Private), public group (Public), chat room (ChatRoom), audio-video group (AVChatRoom), and broadcasting chat room (BChatRoom)**. For more information, see Group Types.

- **Audio-video groups (AVChatRoom):** support an unlimited number of members but do not support features such as adding members or querying the total number of members.
- You can create a group by calling the `createGroup` API in `TIMGroupManager`. During creation, you can specify the group profile (such as the group type, group name, group introduction, list of

members, and even the group ID). After the group is created, the group ID is returned, and you can use it to obtain Conversation for receiving and sending messages.

> ⚠ **Note :**
>
> You need to follow certain rules when defining group IDs. For more information, see Custom Group IDs.

Prototype:

```
/**
* Create a group
* @param param Creates the information set needed for the group. For more information, see {@see
CreateGroupParam}
* @param cb Callback. The group ID of the created group will be returned in a parameter of the On
Success function.
*/
public void createGroup(@NonNull CreateGroupParam param, @NonNull TIMValueCallBack<String> cb)
```

`TIMGroupManager.CreateGroupParam` **provides the following APIs:**

```
/**
* Create a constructor for group parameters
* @param type Group type, which currently supports private group (Private), public group (Public
),
* chat room (ChatRoom), audio-video group (AVChatRoom), and broadcasting chat room (BChatRoom).
* @param name Group name
*/
public CreateGroupParam(@NonNull String type, @NonNull String name)

/**
* Set the group ID of the group to be created
* @param groupId Group ID
*/
public CreateGroupParam setGroupId(String groupId)

/**
* Set the group notice of the group to be created
* @param notification Group notice
*/
public CreateGroupParam setNotification(String notification)

/**
* Set the group introduction of the group to be created
* @param introduction Group introduction
```

```
*/
public CreateGroupParam setIntroduction(String introduction)

/**
* Set the group profile photo URL for the group to be created
* @param url Group profile photo URL
*/
public CreateGroupParam setFaceUrl(String url)

/**
* Set the group joining option for the group to be created
* @param option Group joining option
*/
public CreateGroupParam setAddOption(TIMGroupAddOpt option)

/**
* Set the maximum number of members allowed for the group to be created
* @param maxMemberNum Maximum number of members
*/
public CreateGroupParam setMaxMemberNum(long maxMemberNum)

/**
* Set the custom information of the group to be created
* @param key Custom information key, with a maximum length of 16 bytes
* @param value Custom information value, with a maximum length of 512 bytes
*/
public CreateGroupParam setCustomInfo(String key, byte[] value)

/**
* Set the initial members of the group to be created
* @param infos Information list of initial members
*/
public CreateGroupParam setMembers(List<TIMGroupMemberInfo> infos)
```

Example:

```
//Create a public group without specifying the group ID
TIMGroupManager.CreateGroupParam param = new TIMGroupManager.CreateGroupParam("Public", "test_gro
up");
//Specify the group introduction
param.setIntroduction("hello world");
//Specify the group notice
param.setNotification("welcome to our group");

//Add group members
List<TIMGroupMemberInfo> infos = new ArrayList<TIMGroupMemberInfo>();
TIMGroupMemberInfo member = new TIMGroupMemberInfo("cat");
```

```
infos.add(member);
param.setMembers(infos);

//Set custom group fields. Before that, you need to configure the corresponding keys in the conso
le.
try {
param.setCustomInfo("GroupKey1", "wildcat".getBytes("utf-8"));
} catch (UnsupportedEncodingException e) {
e.printStackTrace();
}

//Create a group
TIMGroupManager.getInstance().createGroup(param, new TIMValueCallBack<String>() {
@Override
public void onError(int code, String desc) {
Log.d(tag, "create group failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess(String s) {
Log.d(tag, "create group succ, groupId:" + s);
}
});
```

Inviting users to a group

The `inviteGroupMember` API of `TIMGroupManager` can be used to invite users to a group.

Permission notes:

For more information, see Differences in Joining a Group.

Prototype:

```
/**
* Invite users to a group
* @param groupId Group ID
* @param memList List of IDs of the users to be added to the group
* @param cb Callback. The user accounts successfully added to the group are returned in a paramet
er of the OnSuccess function.
*/
public void inviteGroupMember(@NonNull String groupId, @NonNull List<String> memList,
@NonNull TIMValueCallBack<List<TIMGroupMemberResult>> cb)
```

`TIMGroupMemberResult` is defined as follows:

```
/**
* Get the operation result
* @return Operation result: 0: failed. 1: successful. 2: the user was already a group member when added, or the user was not a group member when deleted.
*/
public long getResult()

/**
* Get user accounts
* @return User accounts
*/
public String getUser()
```

Example:

```
//Create a list of the users to be added to the group
ArrayList list = new ArrayList();

String user = "";

//Add users
user = "sample_user_1";
list.add(user);
user = "sample_user_2";
list.add(user);
user = "sample_user_3";
list.add(user);

//Callback
TIMValueCallBack<List<TIMGroupMemberResult>> cb = new TIMValueCallBack<List<TIMGroupMemberResult>>() {
@Override
public void onError(int code, String desc) {
}

@Override
public void onSuccess(List<TIMGroupMemberResult> results) { //Group member operation results
for(TIMGroupMemberResult r : results) {
Log.d(tag, "result: " + r.getResult() //Operation result: 0: failed to add. 1: added successfully. 2: already a group member.
+ " user: " + r.getUser()); //User account
}
}
};

//Add the listed users to the group
```

```
TIMGroupManager.getInstance().inviteGroupMember(
groupId, //Group ID
list, //List of the users to be added to the group
cb); //Callback
```

Applying to join a group

The `applyJoinGroup` API of `TIMGroupManager` can be used to apply to join a group. This operation is valid only for public groups, chat rooms, and audio-video groups.

Permission notes:

For more information, see Differences in Joining a Group.

Prototype:

```
/**
* Join a group
* @param groupId Group ID
* @param reason Application reason (optional)
* @param cb Callback
*/
public void applyJoinGroup(@NonNull String groupId, String reason, @NonNull TIMCallBack cb)
```

In the following example, a user applies to join the group [@TGS#1JYSZEAEQ] for the reason of [some reason]. **Example:**

```
TIMGroupManager.getInstance().applyJoinGroup("@TGS#1JYSZEAEQ", "some reason", new TIMCallBack() {
@java.lang.Override
public void onError(int code, String desc) {
//The API returns "code" (error code) and "desc" (error description), which can be used to identi
fy request failure causes.
//For a list of error codes, see the Error Code Table
Log.e(tag, "applyJoinGroup err code = " + code + ", desc = " + desc);
}

@java.lang.Override
public void onSuccess() {
Log.i(tag, "applyJoinGroup success");
}
});
```

Quitting a group

Group members can quit their groups. The API for quitting a group is provided by `TIMGroupManager` .

Permission notes:

- **Private group:** all members can quit the group.
- **Public group, chat room, and audio-video group:** the group owner cannot quit the group.

For more information, see Differences in Member Management Capabilities.

Prototype:

```
/**
* Quit a group
* @param groupId Group ID
* @param cb Callback
*/
public void quitGroup(@NonNull String groupId, @NonNull TIMCallBack cb)
```

Example:

```
//Create callback
TIMCallBack cb = new TIMCallBack() {
@Override
public void onError(int code, String desc) {
//"code" (error code) and "desc" (error description) can be used to locate the cause of the reque
st failure
//For the meanings of error codes, see the Error Code Table
}

@Override
public void onSuccess() {
Log.e(tag, "quit group succ");
}
};

//Quit a group
TIMGroupManager.getInstance().quitGroup(
groupId, //Group ID
cb); //Callback
```

Deleting group members

The function parameter information for deleting group members is the same as that for joining a group. The API for deleting group members is provided by `TIMGroupManager.

Permission notes:

For more information, see Differences in Member Management Capabilities.

Prototype:

```
/**
* Deleting group members
* @param param Parameter for deleting group members
* @param cb Callback. The list of deleted group members is returned in a parameter of the OnSucce
ss function.
*/
public void deleteGroupMember(@NonNull DeleteMemberParam param,
@NonNull TIMValueCallBack<List<TIMGroupMemberResult>> cb)
```

`DeleteMemberParam` **is defined as follows:**

```
/**
* Construct parameters
* @param groupId Group ID
* @param members List of user IDs
*/
public DeleteMemberParam(@NonNull String groupId, @NonNull List<String> members)

/**
* Set the reason for deleting group members (optional)
* @param reason Reason for deletion
*/
public DeleteMemberParam setReason(@NonNull String reason)
```

Example:

```
//Create a list of users to be removed from the group
ArrayList list = new ArrayList();

String user = "";
//Add users
user = "sample_user_1";
list.add(user);
user = "sample_user_2";
list.add(user);
user = "sample_user_3";
list.add(user);

TIMGroupManager.DeleteMemberParam param = new TIMGroupManager.DeleteMemberParam(groupId, list);
param.setReason("some reason");

TIMGroupManager.getInstance().deleteGroupMember(param, new TIMValueCallBack<List<TIMGroupMemberRe
sult>>() {
@Override
```

```
public void onError(int code, String desc) {
Log.e(tag, "deleteGroupMember onErr. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess(List<TIMGroupMemberResult> results) { //Group member operation results
for(TIMGroupMemberResult r : results) {
Log.d(tag, "result: " + r.getResult() //Operation result: 0: failed to delete. 1: deleted success
fully. 2: not a group member.
+ " user: " + r.getUser()); //User account
}
}
});
```

Getting group member list

The `getGroupMembers` API can be used to get the group member list. By default, built-in fields and custom fields are pulled. For custom fields, you can configure the corresponding keys and permissions in **Feature Configuration** > **Custom Group Member Fields** in the IM console. The configuration will take effect in 5 minutes.

Permission notes:

- **An group type:** the member list can be obtained.
- **Audio-video group:** only a partial list of group members can be pulled, including the group owner, admin, and some members.

For more information, see Differences in Basic Group Capabilities.

Prototype:

```
/**
* Get the group member list
* @param groupId Group ID
* @param cb Callback. The group member list is returned in a parameter of the OnSuccess function.
*/
public void getGroupMembers(@NonNull String groupId, @NonNull TIMValueCallBack<List<TIMGroupMembe
rInfo>> cb)
```

Example:

```
//Create callback
TIMValueCallBack<List<TIMGroupMemberInfo>> cb = new TIMValueCallBack<List<TIMGroupMemberInfo>> ()
{
@Override
public void onError(int code, String desc) {
```

```
}

@Override
public void onSuccess(List<TIMGroupMemberInfo> infoList) {//The group member information is retur
ned in the parameter

for(TIMGroupMemberInfo info : infoList) {
Log.d(tag, "user: " + info.getUser() +
"join time: " + info.getJoinTime() +
"role: " + info.getRole());
}
}
};

//Get group member information
TIMGroupManager.getInstance().getGroupMembers(
groupId, //Group ID
cb); //Callback
```

Obtaining your group list

You can get the list of groups the current user has joined through `TIMGroupManager` . The returned information contains only part of the basic information. To get detailed group information, see Group Members Obtain Group Profiles.

Permission notes:

- **Private groups, public groups, and chat rooms:** you can use this API to get information about the public groups, chat rooms, and activated private groups that you have joined.
- **Audio-video groups and broadcasting chat rooms:** these two types of groups were not obtained through this API due to the differences in internal implementation.

Prototype:

```
/**
* Get the list of groups the user has joined
* @param cb Callback. Information about the groups that the current user has joined is returned i
n a parameter of the OnSuccess function.
*/
public void getGroupList(@NonNull TIMValueCallBack<List<TIMGroupBaseInfo>> cb)
```

`TIMGroupBaseInfo` provides the following method:

```
/**
* Get the group ID
```

```
* @return Group ID
*/
public String getGroupId()

/**
* Get the group name
* @return Group name
*/
public String getGroupName()

/**
* Get the group type
* @return Group type
*/
public String getGroupType()

/**
* Get the group profile photo URL
* @return Group profile photo URL
*/
public String getFaceUrl()

/**
* Get whether the current group has muted all members
* @return true - All members are muted
* @since 3.1.1
*/
public boolean isSilenceAll()
```

Example:

```
//Create callback
TIMValueCallBack<List<TIMGroupBaseInfo>> cb = new TIMValueCallBack<List<TIMGroupBaseInfo>>() {
@Override
public void onError(int code, String desc) {
//"code" (error code) and "desc" (error description) can be used to locate the cause of the reque
st failure
//For the meanings of error codes, see the Error Code Table
Log.e(tag, "get group list failed: " + code + " desc");
}

@Override
public void onSuccess(List<TIMGroupBaseInfo> timGroupInfos) {//Basic information about each group
is returned in the parameter
Log.d(tag, "get group list succ");

for(TIMGroupBaseInfo info : timGroupInfos) {
```

```
Log.d(tag, "group id: " + info.getGroupId() +
" group name: " + info.getGroupName() +
" group type: " + info.getGroupType());
}
}
};

//Get the list of groups the user has joined
TIMGroupManager.getInstance().getGroupList(cb);
```

Deleting groups

The API for deleting groups is provided by `TIMGroupManager`.

Permission notes:

For more information, see Differences in Basic Group Capabilities.

Prototype:

```
/**
* Delete a group
* @param groupId Group ID
* @param cb Callback
*/
public void deleteGroup(@NonNull String groupId, @NonNull TIMCallBack cb)
```

Example:

```
//Deleting a group
TIMGroupManager.getInstance().deleteGroup(groupId, new TIMCallBack() {
@Override
public void onError(int code, String desc) {

//"code" (error code) and "desc" (error description) can be used to locate the cause of the reque
st failure
//For a list of error codes, see the Error Code Table
Log.d(tag, "login failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess() {
//The group was deleted successfully
}
});
```

Transferring a group

The API for transferring a group is provided by `TIMGroupManager` .

Permission notes:

Only the **group owner** can transfer a group.

Prototype:

```
/**
* Change the group owner
* @param groupId Group ID
* @param identifier Identifier of the new group owner
* @param cb Callback
*/
public void modifyGroupOwner(@NonNull String groupId, @NonNull String identifier, @NonNull TIMCal
lBack cb)
```

Other APIs

The API for obtaining a specified type of member (admin, group owner, or ordinary member) is defined as follows:

```
/**
* You can obtain the group member list based on filter conditions (for example, by field or by pa
ge)
* @param groupId Group ID
* @param flags Profile pull flag. It can be a flag or combination bitmap, for example, {@see TIMG
roupManager#TIM_GET_GROUP_MEM_INFO_FLAG_NAME_CARD}.
* @param filter Role filter type. For more information, see {@see TIMGroupMemberRoleFilter}.
* @param custom List of custom keys to be obtained
* @param nextSeq Pulling-by-page flag. It is set to 0 when the information is pulled for the firs
t time. If the callback succeeds and the result is not 0, pagination is needed. The value of this
field is passed in for the next pull until the value becomes 0.
* @param cb Callback
*/
public void getGroupMembersByFilter(@NonNull String groupId, long flags, @NonNull TIMGroupMemberR
oleFilter filter,
List<String> custom, long nextSeq, TIMValueCallBack<TIMGroupMemberSucc> cb)
```

Getting Group Profiles

Getting group profiles

The `getGroupInfo` method of `TIMGroupManager` can be used to obtain group information from the server. The `queryGroupInfo` method can be used to obtain locally cached group information. Group members can pull group information. Non-members are not allowed to pull the information of private groups. For other group types, non-members can only pull public fields: `groupId¥groupName¥groupOwner¥groupType¥createTime¥memberNum¥maxMemberNum¥onlineMemberNum¥groupIntroduction¥groupFaceUrl¥addOption¥custom` .

Note:

By default, basic information and custom fields are pulled. For custom fields, you need to configure the corresponding keys and permissions in **Feature Configuration** > **Custom Group Fields** in the IM console. The configuration will take effect in 5 minutes.

Prototype:

```
/**
* Get group information from the server
* @param groupIdList List of group IDs of the groups for which you want to pull detailed information. A maximum of 50 group IDs can be listed at a time.
* @param cb Callback. The group information {@see TIMGroupDetailInfo} list is returned in a parameter of the OnSuccess function.
*/
public void getGroupInfo(@NonNull List<String> groupIdList,
@NonNull TIMValueCallBack<List<TIMGroupDetailInfo>> cb)
/**
* Get locally stored group information
* @param groupId ID of the group for which you want to pull detailed information
* @return Group information. If no group information is stored locally, ‘null’ is returned.
*/
public TIMGroupDetailInfo queryGroupInfo(@NonNull String groupId)
```

The `TIMGroupDetailInfo` API is defined as follows:

```
/**
* Get the group ID
* @return Group ID
*/
public String getGroupId()

/**
* Get the group name
* @return Group name
*/
public String getGroupName()

/**
```

```
* Get the group creator' s account
* @return Group creator' s account
*/
public String getGroupOwner()

/**
* Get the group creation time
* @return Group creation time
*/
public long getCreateTime()

/**
* Get the last time that the group information was modified
* @return Last time that the group information was modified
*/
public long getLastInfoTime()

/**
* Get the time of the latest group message
* @return Time of the latest group message
*/
public long getLastMsgTime()

/**
* Get the number of group members
* @return Number of group members
*/
public long getMemberNum()

/**
* Get the maximum number of group members allowed
* @return Maximum number of group members
*/
public long getMaxMemberNum()

/**
* Get the group introduction
* @return Group introduction
*/
public String getGroupIntroduction()

/**
* Get the group notice
* @return Group notice
*/
public String getGroupNotification()

/**
```

```
* Get the group profile photo URL
* @return Group profile photo URL
*/
public String getFaceUrl()

/**
* Get the group type
* @return Group type
*/
public String getGroupType()

/**
* Get the group joining option
* @return Group joining option
*/
public TIMGroupAddOpt getGroupAddOpt()

/**
* Get the last message in the group
* @return Last message in the group
*/
public TIMMessage getLastMsg()

/**
* Get the custom group field map
* @return Custom group field map
*/
public Map<String, byte[]> getCustom()

/**
* Get whether the group has muted all members
* @return true - All members are muted
* @since 3.1.1
*/
public boolean isSilenceAll()
```

Example:

```
//Create a list of IDs of the groups for which you want to get information
ArrayList<String> groupList = new ArrayList<String>();

//Create callback
TIMValueCallBack<List<TIMGroupDetailInfo>> cb = new TIMValueCallBack<List<TIMGroupDetailInfo>>()
{
@Override
public void onError(int code, String desc) {
//"code" (error code) and "desc" (error description) can be used to locate the cause of the reque
```

```
st failure
//For a list of error codes, see the Error Code Table
}

@Override
public void onSuccess(List<TIMGroupDetailInfo> infoList) { //The group information list is return
ed in the parameter
for(TIMGroupDetailInfo info : infoList) {
Log.d(tag, "groupId: " + info.getGroupId() //Group ID
+ " group name: " + info.getGroupName() //Group name
+ " group owner: " + info.getGroupOwner() //Group creator's account
+ " group create time: " + info.getCreateTime() //Group creation time
+ " group last info time: " + info.getLastInfoTime() //Last time the group information was modifi
ed
+ " group last msg time: " + info.getLastMsgTime() //Time of the last group message
+ " group member num: " + info.getMemberNum()); //Number of group members
}
}
};

//Add the group ID
String groupId = "TGID1EDABEAE0";
groupList.add(groupId);

//Get the server group information
TIMGroupManager.getInstance().getGroupInfo(
groupList, //List of group IDs for which you want to get information
cb); //Callback

//Get group information cached locally
TIMGroupDetailInfo timGroupDetailInfo = TIMGroupManager.getInstance().queryGroupInfo(groupId);
```

Getting your own profile in a group

You can obtain your own profile in a group when the list of groups you have joined is pulled. See Getting group list. If you want to get your profile in a single group, use `getSelfInfo` of `TIMGroupManager` as shown below. If the app needs to get a group list, we recommend that you get the profiles in groups when getting the group list instead of calling the following API.

Permission notes:

Audio-video group: you cannot get your own profile in the group.

Prototype:

```
/**
* Get your own information in a group
```

```
 * @param groupId Group ID
 * @param cb Callback. Your own profile is returned in a parameter of the onSuccess function.
 */
public void getSelfInfo(@NonNull String groupId, @NonNull TIMValueCallBack<TIMGroupSelfInfo> cb)
```

Getting a group member's profile

The API for getting a group member's profile is provided by `TIMGroupManager`. By default, the basic profile is pulled.

Permission notes:

Audio-video group: only the profiles of some members can be obtained, including the group owner, admin, and some group members.

Prototype:

```
/**
 * Get a specified group member's information in the group
 * @param groupId Specified group ID
 * @param identifiers Identifiers of specified group members. A maximum of 100 identifiers can be specified at a time.
 * @param cb Callback. The group member list is returned in a parameter of the OnSuccess function.
 */
public void getGroupMembersInfo(@NonNull String groupId, @NonNull List<String> identifiers,
@NonNull TIMValueCallBack<List<TIMGroupMemberInfo>> cb)
```

Modifying Group Profiles

The API for modifying group profiles is provided by `TIMGroupManager`. You can modify the group name, group introduction, group notice, and other information.

Prototype:

```
/**
 * Modify the basic group information
 * @param param Parameters
 * @param cb Callback
 */
public void modifyGroupInfo(@NonNull ModifyGroupInfoParam param, @NonNull TIMCallBack cb)
```

`TIMGroupManager.ModifyGroupInfoParam` is defined as follows:

```
/**
* Construct parameter instances
* @param groupId Group ID
*/
public ModifyGroupInfoParam(@NonNull String groupId)

/**
* Set the new group name
* @param groupName Group name
*/
public ModifyGroupInfoParam setGroupName(@NonNull String groupName)

/**
* Set the modified group notice
* @param notification Group notice
*/
public ModifyGroupInfoParam setNotification(@NonNull String notification)

/**
* Set the modified group introduction
* @param introduction Group introduction
*/
public ModifyGroupInfoParam setIntroduction(@NonNull String introduction)

/**
* Set the modified group profile photo URL
* @param faceUrl Group profile photo URL
*/
public ModifyGroupInfoParam setFaceUrl(@NonNull String faceUrl)

/**
* Set the group joining option
* @param addOpt Group joining option
*/
public ModifyGroupInfoParam setAddOption(@NonNull TIMGroupAddOpt addOpt)

/**
* Set the maximum number of group members
* @param maxMemberNum Maximum number of group members
*/
public ModifyGroupInfoParam setMaxMemberNum(long maxMemberNum)

/**
* Set whether group members are visible to external users
* @param visable Whether group members are visible to external users
*/
public ModifyGroupInfoParam setVisable(boolean visable)
```

```
/**
* Set custom group fields
* @param customInfos Custom group field dictionary
*/
public ModifyGroupInfoParam setCustomInfo(@NonNull Map<String, byte[]> customInfos)

/**
* Set whether to mute all group members
* @param silenceAll true: mute all group members. false: unmute all group members.
* @since 3.1.1
*/
public ModifyGroupInfoParam setSilenceAll(boolean silenceAll)
```

Changing the group name

Permission notes:

- **Public groups, chat rooms, and audio-video groups:** only the group owner or admin can change the group name.
- **Private groups:** any member can change the group name.

Example:

```
TIMGroupManager.ModifyGroupInfoParam param = new TIMGroupManager.ModifyGroupInfoParam(getGroupId
());
param.setGroupName("Great Team")
TIMGroupManager.getInstance().modifyGroupInfo(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "modify group info failed, code:" + code +"|desc:" + desc);
}

@Override
public void onSuccess() {
Log.e(tag, "modify group info succ");
}
});
```

Modifying the group introduction

Permission notes:

- **Public groups, chat rooms, and audio-video groups:** only the group owner or admin can modify the group introduction.

- **Private groups:** any member can modify the group introduction.

Example:

```
TIMGroupManager.ModifyGroupInfoParam param = new TIMGroupManager.ModifyGroupInfoParam(getGroupId
());
param.setIntroduction("this is a introduction");
TIMGroupManager.getInstance().modifyGroupInfo(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "modify group info failed, code:" + code +"|desc:" + desc);
}

@Override
public void onSuccess() {
Log.e(tag, "modify group info succ");
}
});
```

Modifying the group notice

Permission notes:

- **Public groups, chat rooms, and audio-video groups:** only the group owner or admin can modify the group notice.
- **Private groups:** any member can modify the group notice.

Example:

```
TIMGroupManager.ModifyGroupInfoParam param = new TIMGroupManager.ModifyGroupInfoParam(getGroupId
());
param.setNotification("this is a notification");
TIMGroupManager.getInstance().modifyGroupInfo(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "modify group info failed, code:" + code +"|desc:" + desc);
}

@Override
public void onSuccess() {
Log.e(tag, "modify group info succ");
}
});
```

Modifying the group profile photo

Permission notes:

- **Public groups, chat rooms, and audio-video groups:** only the group owner or admin can modify the group profile photo.
- **Private groups:** any member can modify the group profile photo.

Example:

```
TIMGroupManager.ModifyGroupInfoParam param = new TIMGroupManager.ModifyGroupInfoParam(getGroupId
());
param.setFaceUrl("http://faceurl");
TIMGroupManager.getInstance().modifyGroupInfo(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "modify group info failed, code:" + code +"|desc:" + desc);
}

@Override
public void onSuccess() {
Log.e(tag, "modify group info succ");
}
});
```

Modifying the group joining option

Permission notes:

- **Public groups, chat rooms, and audio-video groups:** only the group owner or admin can modify the group joining option.
- **Private groups:** users can only be invited to join the group and cannot apply to join the group.

Example:

```
TIMGroupManager.ModifyGroupInfoParam param = new TIMGroupManager.ModifyGroupInfoParam(getGroupId
());
param.setAddOption(TIMGroupAddOpt.TIM_GROUP_ADD_ANY);
TIMGroupManager.getInstance().modifyGroupInfo(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "modify group info failed, code:" + code +"|desc:" + desc);
}

@Override
```

```
public void onSuccess() {
Log.e(tag, "modify group info succ");
}
});
```

Modifying custom group fields

Permission notes:

- You need to configure relevant keys and permissions on the backend.

Example:

```
TIMGroupManager.ModifyGroupInfoParam param = new TIMGroupManager.ModifyGroupInfoParam(getGroupId
());
Map<String, byte[]> customInfo = new HashMap<String, byte[]>();
try {
customInfo.put("Test", "Test_value".getBytes("utf-8"));
param.setCustomInfo(customInfo);
} catch (UnsupportedEncodingException e) {
e.printStackTrace();
}

TIMGroupManager.getInstance().modifyGroupInfo(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "modify group info failed, code:" + code +"|desc:" + desc);
}

@Override
public void onSuccess() {
Log.e(tag, "modify group info succ");
}
});
```

Muting all members

Permission notes:

- **Only the group owner or admin** has the permission to mute all members.
- **All group types** support muting all members.

Example:

```
TIMGroupManager.ModifyGroupInfoParam param = new TIMGroupManager.ModifyGroupInfoParam(groupId);
param.setSilenceAll(true);
TIMGroupManager.getInstance().modifyGroupInfo(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "modify group info failed, code:" + code +"|desc:" + desc);
}

@Override
public void onSuccess() {
Log.e(tag, "modify group info succ");
}
});
```

Modifying Group Member Profiles

The API for modifying group member profiles is provided by `TIMGroupManager`. You can modify group member roles and group name cards and mute group members.

Prototype:

```
/**
* Modify group member profiles
* @param param Parameter for modifying group member profiles
* @param cb Callback
*/
public void modifyMemberInfo(@NonNull ModifyMemberInfoParam param, @NonNull TIMCallBack cb)
```

`TIMGroupManager.ModifyMemberInfoParam` is defined as follows:

```
/**
* Construct parameters for modifying group member profiles
* @param groupId ID of the group to which the group member belongs
* @param identifier User ID of the group member whose profile is to be modified
*/
public ModifyMemberInfoParam(@NonNull String groupId, @NonNull String identifier)

/**
* Modify a group member's name card
* @param nameCard Group name card
*/
public ModifyMemberInfoParam setNameCard(@NonNull String nameCard)
```

```
/**
* Modify the option for receiving group messages
* @param receiveMessageOpt The option for receiving group messages. See {@see TIMGroupReceiveMess
ageOpt}
*/
public ModifyMemberInfoParam setReceiveMessageOpt(@NonNull TIMGroupReceiveMessageOpt receiveMessa
geOpt)

/**
* Modify a group member's role (only the group owner and admins can modify roles)
* @param roleType The type of the role. Cannot be changed to group owner. See {@see TIMGroupMembe
rRoleType}
*/
public ModifyMemberInfoParam setRoleType(TIMGroupMemberRoleType roleType)

/**
* Set the muting duration for group members (only the group owner and admin can do this)
* @param silence Muting duration
*/
public ModifyMemberInfoParam setSilence(long silence)

/**
* Set custom group fields
* @param customInfo Custom group field dictionary
*/
public ModifyMemberInfoParam setCustomInfo(Map<String, byte[]> customInfo)
```

Modifying a user's role in the group

Permission notes:

- **Only the group owner or admin** can modify group member roles.
- The roles of members in an **audio-video group** cannot be modified.

Example:

```
TIMGroupManager.ModifyMemberInfoParam param = new TIMGroupManager.ModifyMemberInfoParam(groupId,
identifier);
param.setRoleType(TIMGroupMemberRoleType.Admin);

TIMGroupManager.getInstance().modifyMemberInfo(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "modifyMemberInfo failed, code:" + code + "|msg: " + desc);
}
```

```
@Override
public void onSuccess() {
Log.d(tag, "modifyMemberInfo succ");
}
});
```

Muting group members

You can mute group members and set the muting duration through `modifyMemberInfoParam.setSilence()` .

Permission notes:

- **Only the group owner or admin** can mute group members.

Example:

```
//Mute a member for 100s
TIMGroupManager.ModifyMemberInfoParam param = new TIMGroupManager.ModifyMemberInfoParam(groupId,
identifier);
param.setSilence(100);

TIMGroupManager.getInstance().modifyMemberInfo(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "modifyMemberInfo failed, code:" + code + "|msg: " + desc);
}

@Override
public void onSuccess() {
Log.d(tag, "modifyMemberInfo succ");
}
});
```

Modifying a group member's name card

Example:

```
TIMGroupManager.ModifyMemberInfoParam param = new TIMGroupManager.ModifyMemberInfoParam(groupId,
identifier);
param.setNameCard("cat");

TIMGroupManager.getInstance().modifyMemberInfo(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "modifyMemberInfo failed, code:" + code + "|msg: " + desc);
```

```
}

@Override
public void onSuccess() {
Log.d(tag, "modifyMemberInfo succ");
}
});
```

Modifying custom group member fields

Example:

```
TIMGroupManager.ModifyMemberInfoParam param = new TIMGroupManager.ModifyMemberInfoParam(groupId,
identifier);
Map<String, byte[]> customInfo = new HashMap<>();
try {
customInfo.put("Test", "Custom".getBytes("utf-8"));
param.setCustomInfo(customInfo);
} catch (UnsupportedEncodingException e) {
e.printStackTrace();
}

TIMGroupManager.getInstance().modifyMemberInfo(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "modifyMemberInfo failed, code:" + code + "|msg: " + desc);
}

@Override
public void onSuccess() {
Log.d(tag, "modifyMemberInfo succ");
}
});
```

Modifying the option for receiving group messages

Permission notes:

- **Public groups and private groups:** the default option is to receive and push group messages offline.
- **Chat rooms and audio-video groups:** the default option is to receive but not push group messages offline.

`TIMGroupReceiveMessageOpt` **is defined as follows:**

```
//Do not receive group messages, and the server will not forward them
TIMGroupReceiveMessageOpt.NotReceive

//Receive group messages, but offline messages will not be pushed if users are offline
TIMGroupReceiveMessageOpt.ReceiveNotNotify

//Receive group messages, and offline messages will be pushed if users are offline
TIMGroupReceiveMessageOpt.ReceiveAndNotify
```

Example:

```
TIMGroupManager.ModifyMemberInfoParam param = new TIMGroupManager.ModifyMemberInfoParam(groupId,
identifier);
param.setReceiveMessageOpt(TIMGroupReceiveMessageOpt.ReceiveAndNotify);

TIMGroupManager.getInstance().modifyMemberInfo(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "modifyMemberInfo failed, code:" + code + "|msg: " + desc);
}

@Override
public void onSuccess() {
Log.d(tag, "modifyMemberInfo succ");
}
});
```

Group Pending Requests

Group pending requests are requests that need to be approved, such as pending requests to join a group and requests to invite users to a group. Group pending requests are indicated by the `TIMGroupPendencyItem` class.

`TIMGroupPendencyItem` provides the following methods:

```
/**
* Get the group ID
* @return Group ID
*/
public String getGroupId()

/**
* Get the requester's identifier. For a request to join a group, it refers to the requester's i
```

```
dentifier. For a request to invite users to a group, it refers to the inviter's identifier.
* @return Requester's identifier
*/
public String getFromUser()

/**
* Get the identifier of the handler. The identifier is 0 if it is an "apply to join" request an
d the invitee if it is an "invited to join" request.
* @return Handler's identifier
*/
public String getToUser()

/**
* Get the time when the group pending request was added
* @return Time when the group pending request was added
*/
public long getAddTime()

/**
* Get the type of the group pending request
* @return The type of the group pending request. See TIMGroupPendencyGetType for details
*/
public TIMGroupPendencyGetType getPendencyType()

/**
* Get the processing status of the group pending request
* @return The processing status of the group pending request. See {@see TIMGroupPendencyHandledSt
atus}
*/
public TIMGroupPendencyHandledStatus getHandledStatus()

/**
* Get the processing type of the group pending request. Only valid when the processing status is
not {@see TIMGroupPendencyHandledStatus#NOT_HANDLED}.
* @return The processing type of the group pending request. See {@see TIMGroupPendencyOperationTy
pe}
*/
public TIMGroupPendencyOperationType getOperationType()

/**
* Get the additional information added by the requester
* @return Additional information added by the requester
*/
public String getRequestMsg()

/**
* Get the custom information added by the requester
* @return Custom information added by the requester
```

```
*/
private String getRequestUserData()

/**
* Get the additional information added by the handler. Only valid when the processing status is n
ot {@see TIMGroupPendencyHandledStatus#NOT_HANDLED}.
* @return Additional information added by the handler
*/
public String getHandledMsg()

/**
* Get the custom information added by the handler. Only valid when the processing status is not {
@see TIMGroupPendencyHandledStatus#NOT_HANDLED}.
* @return Custom information added by the handler
*/
private String getHandledUserData()

/**
* Approve the request. Currently, this is valid only for group joining application/invitation mes
sages.
*
* @param msg Reason for approval (optional)
* @param cb Callback
*/
public void accept(String msg, TIMCallBack cb)

/**
* Reject the request. Currently, this is valid only for group joining application/invitation mess
ages.
*
* @param msg Reason for approval (optional)
* @param cb Callback
*/
public void refuse(String msg, TIMCallBack cb)
```

Pulling the list of group pending requests

The `getGroupPendencyList` API provided by `TIMGroupManager` can be used to pull group pending requests. Even after being approved or rejected, pending requests can still be pulled, but in that case, a flag is carried to indicate that the requests have been processed.

Permission notes:

Only the approver has the permission to pull relevant information.

Example:

- When User A applies to join Group A, the group admin can pull the pending request. As User A does not have the approval permission, User A does not need to pull the pending request.
- If Admin A invites User A to Group A, User A can pull this pending request, because the pending request needs to be approved by User A.

Prototype:

```
/**
* Get the list of group pending requests by page
* @param param The parameter for getting a list of group pending requests. See {@see TIMGroupPend
encyGetParam}
* @param cb Callback. The parameter of onSuccess returns the list of group pending requests and m
etadata. See {@see TIMGroupPendencyMeta} and {@see TIMGroupPendencyItem}
*/
public void getGroupPendencyList(@NonNull TIMGroupPendencyGetParam param,
@NonNull TIMValueCallBack<TIMGroupPendencyListGetSucc> cb)
```

`TIMGroupPendencyGetParam` **is defined as follows:**

```
/**
* Set the page turning timestamp, which is only used to turn pages. Enter 0 for the first reques
t. Subsequently, enter the corresponding value based on the timestamp in {@see TIMGroupPendencyMe
ta} returned by the server.
* @param timestamp Page turning timestamp
*/
public void setTimestamp(long timestamp)

/**
* Set the number of requests per page (this is a suggested value which does not indicate completi
on; the server can return more or less requests)
* @param numPerPage Number of requests per page
*/
public void setNumPerPage(long numPerPage)
```

Example:

```
TIMGroupPendencyGetParam param = new TIMGroupPendencyGetParam();
param.setTimestamp(0);//Specify 0 for the first pull
param.setNumPerPage(10);

TIMGroupManager.getInstance().getGroupPendencyList(param, new TIMValueCallBack<TIMGroupPendencyLi
stGetSucc>() {
@Override
public void onError(int code, String desc) {
```

```
}

@Override
public void onSuccess(TIMGroupPendencyListGetSucc timGroupPendencyListGetSucc) {
//If the value of nextStartTimestamp in meta is not 0, you can save it first
// Enter the value into TIMGroupPendencyGetParam as the parameter for obtaining the next page of
data
TIMGroupPendencyMeta meta = timGroupPendencyListGetSucc.getPendencyMeta();
Log.d(tag, meta.getNextStartTimestamp()
+ "|" + meta.getReportedTimestamp() + "|" + meta.getUnReadCount());

List<TIMGroupPendencyItem> pendencyItems = timGroupPendencyListGetSucc.getPendencies();
for(TIMGroupPendencyItem item : pendencyItems){
//Perform an operation on group pending requests. For example, view, approve, or reject a pending
request.
}
}
});
```

Reporting that group pending requests are read

You can use the `reportGroupPendency` API of `TIMGroupManager` to report that a pending request and all other pending requests before it have been read. After reporting, you can still pull these pending requests and determine whether they have been read based on the read timestamp.

Prototype:

```
/**
* Report that group pending requests are read
* @param timestamp Read timestamp (in seconds). All group pending requests before this timestamp
will be set as read.
* @param cb Callback
*/
public void reportGroupPendency(long timestamp, @NonNull TIMCallBack cb)
```

Processing group pending requests

Through `getGroupPendencyList` , a group pending request list (`TIMGroupPendencyItem`) is obtained. Each element on the list can be processed through the `accept/refuse` API of the `TIMGroupPendencyItem` class as a group pending request. Pending requests that have been successfully processed cannot be processed again.

Prototype:

```
/**
* Approve the request. Currently, this is valid only for group joining application/invitation mes
sages.
*
* @param msg Reason for approval (optional)
* @param cb Callback
*/
public void accept(String msg, TIMCallBack cb)

/**
* Reject the request. Currently, this is valid only for group joining application/invitation mess
ages.
*
* @param msg Reason for approval (optional)
* @param cb Callback
*/
public void refuse(String msg, TIMCallBack cb)
```

Group Event Messages

When a user is invited to join a group or is removed from a group, a tip message is displayed in the group. The caller can choose to display the tip message to group users or ignore it. A tip message is identified by a special `Elem` and returned by the new message callback (see New message notification). To get group event messages, in addition to relying on new message notifications, you can also set group event listeners to listen to different events through `setGroupEventListener` of `TIMUserConfig` **before login** (see Initialization (Android)).

> ⚠ **Note :**
>
> The group event messages of chat rooms (ChatRoom) and audio-video groups (AVChatRoom) are not delivered by new message notifications. Therefore, you must register group event listeners to listen to different group events.

`TIMGroupTipsElem` provides the following methods:

```
//Get the group profile change information list. This is valid only when the value of tipsType is
TIMGroupTipsType.ModifyGroupInfo.
java.util.List<TIMGroupTipsElemGroupInfo> getGroupInfoList()

//Get the group name
java.lang.String getGroupName()
```

```
//Get the group member change information list. This is valid only when the value of tipsType is TIMGroupTipsType.ModifyMemberInfo.
java.util.List<TIMGroupTipsElemMemberInfo> getMemberInfoList()

//Get the operator
java.lang.String getOpUser()

//Get the group event notification type
TIMGroupTipsType getTipsType()

//Get the list of accounts to be operated on
java.util.List<java.lang.String> getUserList()
```

`TIMGroupTipsType` prototype:

```
//Cancel an admin
CancelAdmin

//Join a group
Join

//Remove from a group
Kick

//Modify group profiles
ModifyGroupInfo

//Modify member information
ModifyMemberInfo

//Quit a group
Quit

//Set an admin
SetAdmin
```

Group joining notification

Trigger: when a user joins a group (through application or invitation), the system sends a notification in the group. Developers can choose the display mode and can update the group member list. The message type is `TIMGroupTipsType.Join`.

`TIMGroupTipsElem` methods and return description:

Method	Return Description

getType	**TIMGroupTipsType.Join**
getOpUser	**Application to join a group: applicant Invitation to a group: inviter**
getGroupName	**Group name**
getUserList	**List of users to join the group**

Group departure notification

Trigger: when a user chooses to leave a group, the system sends a notification in the group. You can choose to update the group member list. The message type is `TIMGroupTipsType.Quit` .

`TIMGroupTipsElem` methods and return description:

Method	Return Description
getType	**TIMGroupTipsType.Quit**
getOpUser	**Identifier of the user who leaves the group**
getGroupName	**Group name**

Member removal notification

Trigger: when a user is removed from a group, the system sends a notification. You can update the group member list. The message type is `TIMGroupTipsType.Kick` .

`TIMGroupTipsElem` methods and return description:

Method	Return Description
getType	**TIMGroupTipsType.Kick**
getOpUser	**Identifier of the user who removes members from the group**
getGroupName	**Group name**
getUserList	**List of users removed from the group**

Admin setting/cancellation notifications

Trigger: when a user is set or canceled as an admin, the system sends a notification in the group. If the UI shows whether a user is an admin, you can update the admin flag.

The message types are `TIMGroupTipsType.SetAdmin` and `TIMGroupTipsType.CancelAdmin`.

`TIMGroupTipsElem` methods and return description:

Method	Return Description
getType	**Setting: TIMGroupTipsType.SetAdmin Cancellation: TIMGroupTipsType.CancelAdmin**
getOpUser	**Identifier of the user who performs the operation**
getGroupName	**Group name**
getUserList	**List of users who are set or canceled as admins**

Group profile change notification

Trigger: when the group profile changes, for example, when the group name or introduction changes, the system sends a notification. You can update relevant display fields or choose to display the message to users.

`TIMGroupTipsElem` methods and return description:

Method	Return Description
getType	**TIMGroupTipsType.ModifyGroupInfo**
getOpUser	**Identifier of the user who performs the operation**
getGroupName	**Group name**
getGroupInfoList	**Specific profile information of the group whose profile changes. It's the TIMGroupTipsElemGroupInfo structure list.**

`TIMGroupTipsElemGroupInfo` Prototype:

```
//Get the message content
java.lang.String getContent()

//Get the group profile change message type
TIMGroupTipsGroupInfoType getType()
```

`TIMGroupTipsGroupInfoType` Prototype:

```
//Modify the group profile photo URL
ModifyFaceUrl
```

```
//Modify the group introduction
ModifyIntroduction

//Change the group name
ModifyName

//Modify the group notice
ModifyNotification

//Change the group owner
ModifyOwner
```

Group member profile change notification

Trigger: when a group member's group member profile changes, including the role and whether the member is muted, the system sends a notification. You can update relevant displayed fields or choose to display the message to users.

⚠ Note :

- **The profile mentioned here includes only information related to the group, such as muting duration and member role change. Information related to the user, such as the user's nickname, is not included. For groups that have too many members, we recommend that you display the information in the message body instead of updating it in real time. For more information, see Message sender and related profile.**
- **If the user's profile is stored locally, determine whether the locally stored profile changes based on the message body information. If yes, update the profile after receiving a message from the user.**

`TIMGroupTipsElem` methods and return description:

Method	Return Description
getType	**TIMGroupTipsType.ModifyMemberInfo**
getOpUser	**Identifier of the user who performs the operation**
getGroupName	**Group name**
getMemberInfoList	**Specific profile information of the group member whose profile changes. It's the TIMGroupTipsElemMemberInfo structure list.**

`TIMGroupTipsElemMemberInfo` **prototype:**

```
//Get the identifier of the muted member
java.lang.String getIdentifier()

//Get the muting duration
long getShutupTime()
```

Group System Messages

When a group event occurs, for example, when a user applies to join a group, the admin will receive a corresponding group system message. Users can determine whether to accept or reject the request. Relevant messages are displayed to users through group system messages.

Group system message type definitions:

```
//Application to join the group is approved (received only by the applicant)
TIM_GROUP_SYSTEM_ADD_GROUP_ACCEPT_TYPE

//Application to join the group is rejected (received only by the applicant)
TIM_GROUP_SYSTEM_ADD_GROUP_REFUSE_TYPE

//Application is submitted for joining the group (received only by the admin)
TIM_GROUP_SYSTEM_ADD_GROUP_REQUEST_TYPE

//Admin status canceled (received only by the canceled admin)
TIM_GROUP_SYSTEM_CANCEL_ADMIN_TYPE

//Group created (received only by initial members)
TIM_GROUP_SYSTEM_CREATE_GROUP_TYPE

//Group deleted (received by all members)
TIM_GROUP_SYSTEM_DELETE_GROUP_TYPE

//Admin status set (received only by the set admin)
TIM_GROUP_SYSTEM_GRANT_ADMIN_TYPE

//Invited to join a group (received only by the invitee)
TIM_GROUP_SYSTEM_INVITED_TO_GROUP_TYPE

//Removed from the group by the admin (received only by the removed user)
TIM_GROUP_SYSTEM_KICK_OFF_FROM_GROUP_TYPE
```

```
//Group departure (received only by the user who leaves the group)
TIM_GROUP_SYSTEM_QUIT_GROUP_TYP

//Group repossessed (received by all members)
TIM_GROUP_SYSTEM_REVOKE_GROUP_TYPE

//Invited to join a group (received only by the invitee)
TIM_GROUP_SYSTEM_INVITE_TO_GROUP_REQUEST_TYPE

//Group invitation request approved (received only by the inviter)
TIM_GROUP_SYSTEM_INVITATION_ACCEPTED_TYPE

//Group invitation request rejected (received only by the inviter)
TIM_GROUP_SYSTEM_INVITATION_REFUSED_TYPE
```

`TIMGroupSystemElem` methods are defined as follows:

```
/**
* Operator platform information
* Values: iOS, Android, Windows, Mac, Web, RESTAPI, Unknown
* @return Operator platform information returned
*/
public String getPlatform()

/**
* Get the message sub-type
* @return Group system message sub-type
*/
public TIMGroupSystemElemType getSubtype()

/**
* Get the group ID of the message
* @return
*/
public String getGroupId()

/**
* Get the operator
* @return Operator' s identifier
*/
public String getOpUser()

/**
* Get the reason for the operation
* @return Reason for the operation
*/
public String getOpReason()
```

```
/**
* Get custom notification
* @return Custom notification
*/
public byte[] getUserData()

/**
* Get the operator' s profile
* @return Operator' s profile
*/
public TIMUserProfile getOpUserInfo()

/**
* Get the operator' s group member profile
* @return Operator' s group member profile
*/
public TIMGroupMemberInfo getOpGroupMemberInfo()
```

Group joining application request

Trigger: when a user applies to join a group, the group admin receives a group joining application request and can determine whether to approve the request. Message type: `TIM_GROUP_SYSTEM_ADD_GROUP_REQUEST_TYPE` .

TIMGroupSystemElem methods and return description:

Method	Return Description
getSubtype	**TIM_GROUP_SYSTEM_ADD_GROUP_REQUEST_TYPE**
getGroupId	**The ID of the group for which the application was sent**
getOpUser	**Applicant**
getOpReason	**Reason for application (optional)**

Group joining application approval/rejection notifications

Trigger: when an admin approves a group joining application, the applicant receives an application approval notification. When the admin rejects the application, the applicant receives an application rejection notification.

`TIMGroupSystemElem` methods and return description:

Method	Return Description

getSubtype	**Approval: TIM_GROUP_SYSTEM_ADD_GROUP_ACCEPT_TYPE Rejection: TIM_GROUP_SYSTEM_ADD_GROUP_REFUSE_TYPE**
getGroupId	**ID of the group for which the request is approved/rejected**
getOpUser	**Identifier of the admin who processed the request**
getOpReason	**Reason for approval or rejection (optional)**

Group invitation request

Trigger: when a user is invited to join a group (assuming the user has not yet joined the group and needs to process the invitation), the user receives an invitation request message.

> ⚠ **Note :**
>
> **During the creation of a group, initial members can join it directly without the need of an invitation.**

`TIMGroupSystemElem` **methods and return description:**

Method	Return Description
getSubtype	**TIM_GROUP_SYSTEM_INVITE_TO_GROUP_REQUEST_TYPE**
getGroupId	**ID of the group that the invitee is invited to join**
getOpUser	**Operator, i.e., the inviter**

Group invitation acceptance/rejection notifications

Trigger: when the invitee accepts a group invitation, the inviter receives a group invitation acceptance notification. When the invitee rejects the invitation, the inviter receives a group invitation rejection notification.

`TIMGroupSystemElem` **methods and return description:**

Method	Return Description
getSubtype	**Acceptance: TIM_GROUP_SYSTEM_INVITATION_ACCEPTED_TYPE Rejection: TIM_GROUP_SYSTEM_INVITATION_REFUSED_TYPE**
getGroupId	**ID of the group for which the request is approved/rejected**

getOpUser	**Identifier of the admin who processed the request**
getOpReason	**Reason for acceptance or rejection (optional)**

Group member removal notification

Trigger: when a user is removed from a group by a group admin, the user receives a corresponding notification.

`TIMGroupSystemElem` **methods and return description:**

Method	Return Description
getSubtype	**TIM_GROUP_SYSTEM_KICK_OFF_FROM_GROUP_TYPE**
getGroupId	**ID of the group from which the user is removed**
getOpUser	**Identifier of the admin who performed the operation**

Group deletion notification

Trigger: when a group is deleted, all group members receive a corresponding notification.

`TIMGroupSystemElem` **methods and return description:**

Method	Return Description
getSubtype	**TIM_GROUP_SYSTEM_DELETE_GROUP_TYPE**
getGroupId	**ID of the deleted group**
getOpUser	**Identifier of the admin who performed the operation**

Group creation notification

Trigger: when a group is created, the creator receives a creation notification message.

When the callback for calling the group creation method succeeds, a group is created. This message is mainly used for multi-client synchronization. If other clients are also logged in, this message can be used to time group list updates. You can choose to ignore this message on the current client.

`TIMGroupSystemElem` **methods and return description:**

Method	Return Description
getSubtype	**TIM_GROUP_SYSTEM_CREATE_GROUP_TYPE**

getGroupId	**ID of the created group**
getOpUser	**Creator, i.e., the user who performed the operation**

Group invitation notification

Trigger: when a user is invited to join a group (the user has been added to the group at this time), the user receives an invitation notification message.

> ⚠ **Note :**
>
> **During the creation of a group, initial members can join it directly without the need of an invitation.**

`TIMGroupSystemElem` **methods and return description:**

Method	Return Description
getSubtype	**TIM_GROUP_SYSTEM_INVITED_TO_GROUP_TYPE**
getGroupId	**ID of the group that the invitee is invited to join**
getOpUser	**Operator, i.e., the inviter**

Group departure notification

Trigger: when a user chooses to leave a group, only the user himself/herself receives a corresponding notification.

If a user calls `QuitGroup` **and the success callback is returned, then the user has quit the group successfully. This message is used for multi-client synchronization. When receiving this message, the other clients can update the group list, while the current client can choose to ignore it.**

`TIMGroupSystemElem` **methods and return description:**

Method	Return Description
getSubtype	**TIM_GROUP_SYSTEM_QUIT_GROUP_TYPE**
getGroupId	**ID of the group that the user quits**
getOpUser	**Operator, i.e., the user who performed the operation**

Admin setting/cancellation notifications

Trigger: when a user is set or canceled as a group admin, the user receives a corresponding notification.

`TIMGroupSystemElem` **methods and return description:**

Method	Return Description
getSubtype	**Admin role is canceled: TIM_GROUP_SYSTEM_GRANT_ADMIN_TYPE Admin role is granted: TIM_GROUP_SYSTEM_CANCEL_ADMIN_TYPE**
getGroupId	**ID of the group where this event occurred**
getOpUser	**Operator**

Group repossessing notification

Trigger: when a group is repossessed by the system, all group members receive a corresponding notification.

`TIMGroupSystemElem` **methods and return description:**

Method	Return Description
getSubtype	**TIM_GROUP_SYSTEM_REVOKE_GROUP_TYPE**
getGroupId	**ID of the repossessed group**

Group Management (iOS)

Last updated : 2021-01-26 14:24:37

Group Overview

Instant Messaging (IM) supports multiple group types. For more information on their characteristics and limits, see Group System. A group is identified by a unique ID that enables different operations.

Group Chat Messages

Group messages and C2C (one-to-one) messages are the same except for the conversation type obtained through 'Conversation'. For more information, see Sending Messages.

Group Management

Operations related to groups are performed after login through `TIMGroupManager`.

Prototype for getting a singleton:

```
@interface TIMGroupManager : NSObject
+ (TIMGroupManager*)sharedInstance;
@end
```

Creating built-in group types

Instant Messaging (IM) supports the following group types by default: private group (Private), public group (Public), chat room (ChatRoom), audio-video chat room (AVChatRoom), and broadcasting chat room (BChatRoom). For more information, please see Group Types. You can specify the group name and the list of users to add. After the group is created, the group ID is returned, which allows you to receive and send messages through `Conversation`.

Description of group creation:

Method	Description
CreatePrivateGroup	**Creates a private group**
CreatePublicGroup	**Creates a public group**
CreateChatRoomGroup	**Creates a chat room**
CreateAVChatRoomGroup	**Creates a live-streaming group. A live-streaming group supports an unlimited number of members but does not support features such as adding members or querying the total number of members.**

Prototype:

```
@interface TIMGroupManager : NSObject
/**
```

```
* Create a private group
*
* @param members Group members, NSString* array
* @param groupName Group name
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)createPrivateGroup:(NSArray*)members groupName:(NSString*)groupName succ:(TIMCreateGroupSu
cc)succ fail:(TIMFail)fail;
/**
* Create a public group
*
* @param members Group members, NSString* array
* @param groupName Group name
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)createPublicGroup:(NSArray*)members groupName:(NSString*)groupName succ:(TIMCreateGroupSuc
c)succ fail:(TIMFail)fail;
/**
* Create a chat room
*
* @param members Group members, NSString* array
* @param groupName Group name
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)createChatRoomGroup:(NSArray*)members groupName:(NSString*)groupName succ:(TIMCreateGroupS
ucc)succ fail:(TIMFail)fail;
/**
* Create an audio-video chat room (ultra-large groups are supported, see the wiki documentation)
*
* @param groupName Group name
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)createAVChatRoomGroup:(NSString*)groupName succ:(TIMCreateGroupSucc)succ fail:(TIMFail)fai
l;
@end
```

Parameter description:

Parameter	Description
members	**A NSString list of members to be added to the group. The creator is added by default. Public, Private, and ChatRoom groups support up to 6,000 members. AVChatRoom groups can support an unlimited number of members.**
groupName	**The group name is of NSString type and its length is up to 30 bytes**
groupId	**The group ID that you specified, NSString type**
succ	**Success callback that returns the group ID**
fail	**Failure callback**

The following example creates a private group and adds user “iOS_002” to the group.
Example:

Note :

- **There is no need to explicitly specify the creator as the creator is added to the group by default.**
- **The methods and parameters of public groups and chat rooms are identical but have different method names.**

```
NSMutableArray * members = [[NSMutableArray alloc] init];
// Add user iOS_002
[members addObject:@”iOS_002”];
[[TIMGroupManager sharedInstance] createPrivateGroup:members groupName:@”GroupName” succ:^(NSStri
ng * group) {
NSLog(@”create group succ, sid=%@”, group);
} fail:^(int code, NSString* err) {
NSLog(@”failed code: %d %@”, code, err);
}];
```

Creating group with specified properties

When creating a group, you can set default members and the group name as well as the group announcement and group introduction.

```
/**
* Create group parameters
*/
@interface TIMCreateGroupInfo : TIMCodingModel
/**
* The group ID. If nil, use the system's default ID.
*/
@property(nonatomic,retain) NSString* group;
/**
* Group name
*/
@property(nonatomic,retain) NSString* groupName;
/**
* Group type: Private, Public, ChatRoom, AVChatRoom
*/
@property(nonatomic,retain) NSString* groupType;
/**
* Whether to set the option for joining the group. Set to false for private groups.
*/
@property(nonatomic,assign) BOOL setAddOpt;
/**
* Option for joining group
*/
@property(nonatomic,assign) TIMGroupAddOpt addOpt;
/**
* The maximum number of members. If 0 is entered, then the system uses the default value.
*/
@property(nonatomic,assign) uint32_t maxMemberNum;
/**
* Group announcement
*/
@property(nonatomic,retain) NSString* notification;
/**
* Group introduction
*/
@property(nonatomic,retain) NSString* introduction;
/**
* Group profile photo
*/
@property(nonatomic,retain) NSString* faceURL;
/**
* Collection of custom fields. Key is of NSString* type and value is of NSData* type.
*/
@property(nonatomic,retain) NSDictionary* customInfo;
/**
* Create a list of members (TIMCreateGroupMemberInfo*)
*/
```

```
@property(nonatomic,retain) NSArray* membersInfo;
@end

@interface TIMGroupManager : NSObject
/**
* Create group
*
* @param groupInfo Group information
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)createGroup:(TIMCreateGroupInfo*)groupInfo succ:(TIMCreateGroupSucc)succ fail:(TIMFail)fai
l;
@end
```

Parameter description:

Parameter	Description
groupInfo	**Group ID, group name, group type, option for joining group, maximum number of members, group announcement, group introduction, and group profile photo**
succ	**Success callback**
fail	**Failure callback**

The following example creates a private group with specified properties and adds user "iOS_001" to the group. You do not need to explicitly specify the creator as the creator is added to the group by default. Example:

```
// Create group information
TIMCreateGroupInfo *groupInfo = [[TIMCreateGroupInfo alloc] init];
groupInfo.group = nil;
groupInfo.groupName = @"group_private";
groupInfo.groupType = @"Private";
groupInfo.addOpt = TIM_GROUP_ADD_FORBID;
groupInfo.maxMemberNum = 3;
groupInfo.notification = @"this is a notification";
groupInfo.introduction = @"this is a introduction";
groupInfo.faceURL = nil;
// Create group member information
TIMCreateGroupMemberInfo *memberInfo = [[TIMCreateGroupMemberInfo alloc] init];
memberInfo.member = @"iOS_001";
```

```
memberInfo.role = TIM_GROUP_MEMBER_ROLE_ADMIN;
// Add group member information
NSMutableArray *membersInfo = [[NSMutableArray alloc] init];
[membersInfo addObject:memberInfo];
groupInfo.membersInfo = membersInfo;
// Create a group with specified properties
[[TIMGroupManager sharedInstance] createGroup:groupInfo succ:^(NSString * group) {
NSLog(@"create group succ, sid=%@", group);
} fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

Creating a group with a custom group ID

By default, the IM server generates a unique ID when a group is created. If a custom group ID is needed, the user can specify an ID when the group is created. A custom group ID can also be obtained by creating a group with specified properties.

```
@interface TIMGroupManager : NSObject
/**
* Create group
*
* @param type Group type: Private, Public, ChatRoom, AVChatRoom
* @param groupId The custom group ID. The system automatically assigns a group ID if it is empty.
* @param groupName Group name
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)createGroup:(NSString*)type groupId:(NSString*)groupId groupName:(NSString*)groupName suc
c:(TIMCreateGroupSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
type	**Group type**
members	**List of initial group members**
groupName	**Group name**
groupId	**The custom group ID**
succ	**Success callback**

fail	**Failure callback**

Inviting users to a group

You can invite users to a group through `inviteGroupMember` of `TIMGroupManager`.

Permission description:

For more information, see Differences in group member operations.

Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Invite friends to a group
*
* @param group Group ID
* @param members The list of users who will be added to the group (NSString* array)
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)inviteGroupMember:(NSString*)group members:(NSArray*)members succ:(TIMGroupMemberSucc)succ
fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
group	**The group ID, which is of NSString type**
members	**NSString list of users who will be added to the group**
succ	**The success callback that returns the list of members who are added to the group successfully and the success status. It is a TIMGroupMemberResult array.**
fail	**Failure callback**

The following example invites user "iOS_002" to join the group with the group ID of "TGID1JYSZEAEQ" and returns the operation list and success status after the operation succeeds. `result.status` indicates whether the current user's operation was successful.
Example:

```
NSMutableArray * members = [[NSMutableArray alloc] init];
// Add user iOS_002
[members addObject:@"iOS_002"];
// @"TGID1JYSZEAEQ" is the group ID
[[TIMGroupManager sharedInstance] inviteGroupMember:@"TGID1JYSZEAEQ" members:members succ:^(NSArr
ay* arr) {
for (TIMGroupMemberResult * result in arr) {
NSLog(@"user %@ status %d", result.member, result.status);
}
} fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

`result.status` prototype:

```
/**
* Group operation result
*/
typedef NS_ENUM(NSInteger, TIMGroupMemberStatus) {
/**
Operation failed.
*/
TIM_GROUP_MEMBER_STATUS_FAIL = 0,
/**
* Successful operation
*/
TIM_GROUP_MEMBER_STATUS_SUCC = 1,
/**
* Operation is invalid. The user is already a group member when added to the group, or the user i
s not a group member when removed from the group
*/
TIM_GROUP_MEMBER_STATUS_INVALID = 2,
/**
* Pending processing. Wait for the invitee to process.
*/
TIM_GROUP_MEMBER_STATUS_PENDING = 3,
};
```

Applying to join a group

`joinGroup` of `TIMGroupManager` allows users to apply to join a group. This operation is valid only for public groups, chat rooms, and audio-video chat rooms.

Permission description:

For more information, see Differences in group member operations.

Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Apply to join group
*
* @param group The ID of the target group
* @param msg Application message
* @param succ Success callback (application succeeds, pending approval)
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)joinGroup:(NSString*)group msg:(NSString*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
group	**The group ID, which is of NSString type**
msg	**Reason for application**
succ	**Success callback**
fail	**Failure callback**

In the following example, a user applies to join the group "TGID1JYSZEAEQ" and the reason for application is "Apply Join Group". Example:

```
[[TIMGroupManager sharedInstance] joinGroup:@"TGID1JYSZEAEQ" msg:@"Apply Join Group" succ:^(){
NSLog(@"Join Succ");
}fail:^(int code, NSString * err) {
NSLog(@"code=%d, err=%@", code, err);
}];
```

Quitting a group

Group members can quit a group.

Permission description:

- **Private group: every member can quit the group.**

- **Public group, chat room, and live-streaming group: the group owner cannot quit the group.**

For more information, see Differences in group member operations.

Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Quit group
*
* @param group Group ID
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)quitGroup:(NSString*)group succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
group	**The group ID, which is of NSString type**
succ	**Success callback**
fail	**Failure callback**

In the following example, a user quits the group "TGID1JYSZEAEQ". Example:

```
// @"TGID1JYSZEAEQ" is the group ID
[[TIMGroupManager sharedInstance] quitGroup:@"TGID1JYSZEAEQ" succ:^() {
NSLog(@"succ");
} fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

Deleting group members

Group members can delete other members. The parameters of this function are the same as the parameters for joining a group.

Permission description:

For more information, see Differences in group member operations.

Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Delete group members
*
* @param group Group ID
* @param reason Reason for deleting
* @param members List of members to be deleted
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)deleteGroupMemberWithReason:(NSString*)group reason:(NSString*)reason members:(NSArray*)me
mbers succ:(TIMGroupMemberSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
group	**The group ID, which is of NSString type**
reason	**The reason, which is of NSString type**
members	**List of members who are operated on. It is an `NSString*` array.**
succ	**The success callback that returns the list of members who are added to the group successfully and the success status. It is a `TIMGroupMemberResult` array.**
fail	**Failure callback**

The following example deletes user "iOS_002" from group "TGID1JYSZEAEQ" and returns the operation list and operation status after the deletion succeeds. Example:

```
NSMutableArray * members = [[NSMutableArray alloc] init];
// Add user iOS_002
[members addObject:@"iOS_002"];
// @"TGID1JYSZEAEQ" is the group ID
[[TIMGroupManager sharedInstance] deleteGroupMemberWithReason:@"TGID1JYSZEAEQ" reason:@"broke gro
up rules" members:members succ:^(NSArray* arr) {
```

```
for (TIMGroupMemberResult * result in arr) {
NSLog(@"user %@ status %d", result.member, result.status);
}
} fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

Obtaining the group member list

Get a group member list through the `getGroupMembers` method.

Permission description:

- **Any type of group: the list of members can be obtained.**
- **Live-streaming group: only some of the members are pulled, including the group owner, admin, and some members.**

For more information, see Differences in group operations.

Prototype:

```
/**
* Returned values of group member operations
*/
@interface TIMGroupMemberInfo : TIMCodingModel
/**
* The member that is operated on
*/
@property(nonatomic,retain) NSString* member;
/**
* Group name card
*/
@property(nonatomic,retain) NSString* nameCard;
/**
* The time of joining the group
*/
@property(nonatomic,assign) time_t joinTime;
/**
* Member's role in the group
*/
@property(nonatomic,assign) TIMGroupMemberRole role;
/**
* Muting duration (the remaining seconds)
*/
@property(nonatomic,assign) uint32_t silentUntil;
/**
* Collection of custom fields. Key is of NSString* type and value is of NSData* type.
```

```
*/
@property(nonatomic,retain) NSDictionary* customInfo;
@end

@interface TIMGroupManager : NSObject
/**
* Get group member list
*
* @param group Group ID
* @param succ Success callback (TIMGroupMemberInfo list)
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)getGroupMembers:(NSString*)groupId succ:(TIMGroupMemberSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
group	**The group ID, which is of NSString* type**
succ	**Success callback that returns a TIMGroupMemberInfo* array**
fail	**Failure callback**

This example gets the member list of group "TGID1JYSZEAEQ". `list` is `TIMGroupMemberInfo*` data and stores member information. Example:

```
// @"TGID1JYSZEAEQ" is the group ID
[[TIMGroupManager sharedInstance] getGroupMembers:@"TGID1JYSZEAEQ" succ:^(NSArray* list) {
for (TIMGroupMemberInfo * info in list) {
NSLog(@"user=%@ joinTime=%lu role=%d", info.member, info.joinTime, info.role);
}
} fail:^(int code, NSString * err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

If a group has too many members, use the paging API. Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Get the list of members whose type you specify (the list can be pulled by field and in pages)
*
* @param group Group ID: NSString* list
```

```
* @param filter Group member role filter
* @param flags The flag of the profile to be pulled
* @param custom The list of custom keys (NSString*) to get
* @param nextSeq The flag for pulling in pages. Enter 0 for the first request. If the success cal
lback returns a value that is not 0, then the list will be pulled in pages. Pass this parameter t
o pull again until the value becomes 0.
* @param succ Success callback
* @param fail Failure callback
*/
- (int)getGroupMembers:(NSString*)group ByFilter:(TIMGroupMemberFilter)filter flags:(TIMGetGroupM
emInfoFlag)flags custom:(NSArray*)custom nextSeq:(uint64_t)nextSeq succ:(TIMGroupMemberSuccV2)suc
c fail:(TIMFail)fail;
@end
```

Obtaining your group list

Get the list of groups the current user has joined through `getGroupList`.

Permission description:

- **Get the list of groups you have joined. The returned `TIMGroupInfo` contains only `group`, `groupName`, and `groupType`.**
- **Only some of the live-streaming groups the user has joined can be obtained.**

Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Get group list
*
* @param succ Success callback. The NSArray list is TIMGroupInfo. The structure contains only gro
up¥groupName¥groupType¥faceUrl¥selfInfo.
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)getGroupList:(TIMGroupListSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
succ	**Success callback that returns the list of group IDs. It is a TIMGroupInfo array.**

fail	Failure callback

The following example gets a group list and prints group IDs, group types (Private, Public, ChatRoom), and group names. Example:

```
[[TIMGroupManager sharedInstance] getGroupList:^(NSArray * list) {
for (TIMGroupInfo * info in list) {
NSLog(@"group=%@ type=%@ name=%@", info.group, info.groupType, info.groupName);
}
} fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

Disbanding groups

Disband groups through `DeleteGroup` .

Permission description:

For more information, see Differences in group operations.

Prototype:

```
@interface TIMGroupManager : NSObject

/**
* Disband group
*
* @param group Group ID
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)deleteGroup:(NSString*)group succ:(TIMSucc)succ fail:(TIMFail)fail;

@end
```

Parameter description:

Parameter	Description
group	**Group ID**
succ	**Success callback that returns group ID list. It is an NSString array.**

fail	Failure callback

The following example disbands the group “TGID1JYSZEAEQ”. Example:

```
[[TIMGroupManager sharedInstance] deleteGroup:@”TGID1JYSZEAEQ” succ:^() {
NSLog(@”delete group succ”);
}fail:^(int code, NSString* err) {
NSLog(@”failed code: %d %@”, code, err);
}];
```

Transferring a group

You can transfer groups through `modifyGroupOwner` .

Permission description:

- **Only the group owner has the permission to transfer the group ownership.**
- **The ownership of live-streaming groups cannot be transferred.**

Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Transfer group to a new group owner
*
* @param group Group ID
* @param identifier The ID of the new group owner
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)modifyGroupOwner:(NSString*)group user:(NSString*)identifier succ:(TIMSucc)succ fail:(TIMF
ail)fail;
@end
```

Parameter description:

Parameter	Description
group	**Group ID**
user	**User ID**
succ	**Success callback**

fail	Failure callback

The following example transfers group “TGID1JYSZEAEQ” to user “iOS_001”. Example:

```
[[TIMGroupManager sharedInstance] modifyGroupOwner:@"TGID1JYSZEAEQ" user:@"iOS_001" succ:^() {
NSLog(@"set new owner succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

Muting all members

Set Mute All through `modifyGroupAllShutup` .

Permission description:

- **The group owner and admin can mute all members.**
- **All group types support muting all members.**

Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Modify Mute All property
*
* @param group Group ID
* @param shutup Whether to mute or not
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)modifyGroupAllShutup:(NSString*)group shutup:(BOOL)shutup succ:(TIMSucc)succ fail:(TIMFai
l)fail;
@end
```

Parameter description:

Parameter	Description
group	**Group ID**
shutup	**Whether to mute or not**
succ	**Success callback**

fail	Failure callback

This example sets Mute All for the group “TGID1JYSZEAEQ”. Get the Mute All property on the client through `getGroupList` and `getGroupInfo`. Example:

```
[[TIMGroupManager sharedInstance] modifyGroupAllShutup:@”TGID1JYSZEAEQ” shutup:YES succ:^() {
NSLog(@”set all shutup succ”);
}fail:^(int code, NSString* err) {
NSLog(@”failed code: %d %@”, code, err);
}];
```

Getting Group Profiles

Getting group profiles

You can use the `getGroupInfo` method of `TIMGroupManager` to get group profiles stored on the server and use the `queryGroupInfo` method to get group profiles cached locally. Group profiles are defined by `TIMGroupInfo`.

Permission description:

- **The group members of public and private groups can pull group profiles.**
- **For a public group, non-members can pull profile fields including group, groupName, owner, groupType, createTime, maxMemberNum, memberNum, introduction, faceURL, addOpt, onlineMemberNum, and customInfo. For a private group, non-members cannot pull any group profile information.**

Prototype:

```
/**
* Group profile information
*/
@interface TIMGroupInfo : TIMCodingModel
/**
* Group ID
*/
@property(nonatomic,retain) NSString* group;
/**
* Group name
*/
@property(nonatomic,retain) NSString* groupName;
/**
```

```
* Group creator/admin
*/
@property(nonatomic,retain) NSString * owner;
/**
* Group type: Private, Public, and ChatRoom
*/
@property(nonatomic,retain) NSString* groupType;
/**
* The time the group was created
*/
@property(nonatomic,assign) uint32_t createTime;
/**
* The time the group profile was last modified
*/
@property(nonatomic,assign) uint32_t lastInfoTime;
/**
* The time the last group message was sent
*/
@property(nonatomic,assign) uint32_t lastMsgTime;
/**
* The maximum number of members
*/
@property(nonatomic,assign) uint32_t maxMemberNum;
/**
* The number of group members
*/
@property(nonatomic,assign) uint32_t memberNum;
/**
* The option for joining group
*/
@property(nonatomic,assign) TIMGroupAddOpt addOpt;
/**
* Group announcement
*/
@property(nonatomic,retain) NSString* notification;
/**
* Group introduction
*/
@property(nonatomic,retain) NSString* introduction;
/**
* Group profile photo
*/
@property(nonatomic,retain) NSString* faceURL;
/**
* The last message
*/
@property(nonatomic,retain) TIMMessage* lastMsg;
/**
```

```
* The number of online members
*/
@property(nonatomic,assign) uint32_t onlineMemberNum;
/**
* Whether or not the group can be searched for
*/
@property(nonatomic,assign) TIMGroupSearchableType isSearchable;
/**
* Whether group members are visible
*/
@property(nonatomic,assign) TIMGroupMemberVisibleType isMemberVisible;
/**
* Whether all members are muted
*/
@property(nonatomic,assign) BOOL allShutup;
/**
* Your own information in the group
*/
@property(nonatomic,retain) TIMGroupSelfInfo* selfInfo;
/**
* Collection of custom fields. Key is of NSString* type and value is of NSData* type.
*/
@property(nonatomic,retain) NSDictionary* customInfo;
@end

@interface TIMGroupManager : NSObject
/**
* Get group profiles stored on the server
*
* @param groups List of group IDs
* @param succ Success callback with selfInfo excluded
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)getGroupInfo:(NSArray*)groups succ:(TIMGroupListSucc)succ fail:(TIMFail)fail;
/**
* Get group profiles stored locally
*
* @param group Group ID
*
* @return Group profile
*/
- (TIMGroupInfo *)queryGroupInfo:(NSString *)group;
@end
```

Parameter description:

Parameter	Description
groups	**List of groups for which profiles are to be obtained, NSString array**
succ	**Success callback that returns the list of group profiles. It is a TIMGroupInfo array.**
fail	**Failure callback**

The following example gets the detailed information of the group “TGID1JYSZEAEQ”.

Example:

```
NSMutableArray * groupList = [[NSMutableArray alloc] init];
[groupList addObject:@”TGID1JYSZEAEQ”];
[[TIMGroupManager sharedInstance] getGroupInfo:groupList succ:^(NSArray * groups) {
for (TIMGroupInfo * info in groups) {
NSLog(@”get group succ, infos=%@”, info);
}
} fail:^(int code, NSString* err) {
NSLog(@”failed code: %d %@”, code, err);
}];
```

Getting your own profile in a group

Permission description:

Live-streaming group: your own profile cannot be pulled.

Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Get your own member information in the group
*
* @param group Group ID
* @param succ Success callback that returns the information
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)getGroupSelfInfo:(NSString*)groupId succ:(TIMGroupSelfSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description

groupId	Group ID
succ	Success callback that returns your own profile in the group
fail	Failure callback

Getting the profiles of specified members in the group

Permission description:

- ***Live-streaming group: only the profiles of some members can be pulled, including the group owner, admin, and some members.**

Prototype:

```
@interface TIMGroupManager : NSObject
/**
*
* To get the profiles of specified members in the group, you need to set members. For information
on other limits, please see getGroupMembers.
*
* @param groupId Group ID
* @param members List of member IDs (NSString*)
* @param succ Success callback (TIMGroupMemberInfo list)
* @param fail Failure callback
*
* @return 0: successful. 1: failed.
*/
- (int)getGroupMembersInfo:(NSString*)groupId members:(NSArray<NSString *>*)members succ:(TIMGrou
pMemberSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
groupId	Group ID
members	List of member IDs
succ	Success callback that returns the list of group member profiles
fail	Failure callback

Modifying the Group Profile

Modifying the group name

Modify the group name through `modifyGroupName` .

Permission description:

- **Public group, chat room, and live-streaming group: only the group owner or admin can modify the group name.**
- **Private group: any member can modify the group name.**

Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Modify the group name
*
* @param group Group ID
* @param groupName The new group name
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)modifyGroupName:(NSString*)group groupName:(NSString*)groupName succ:(TIMSucc)succ fail:(T
IMFail)fail;
@end
```

Parameter description:

Parameter	Description
group	**Group ID**
groupName	**The modified group name**
succ	**Success callback**
fail	**Failure callback**

The following example changes the name of group “TGID1JYSZEAEQ” to “ModifyGroupName”. Example:

```
[[TIMGroupManager sharedInstance] modifyGroupName:@"TGID1JYSZEAEQ" groupName:@"ModifyGroupName" s
ucc:^() {
NSLog(@"modify group name succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

Modifying the group introduction

Modify the group introduction through `modifyGroupIntroduction` .

Permission description:

- **Public group, chat room, and live-streaming group: only the group owner or admin can modify the group introduction.**
- **Private group: any member can modify the group introduction.**

Prototype:

```
@interface TIMGroupManager : NSObject

/**
* Modify group introduction
*
* @param group Group ID
* @param introduction Group introduction
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)modifyGroupIntroduction:(NSString*)group introduction:(NSString*)introduction succ:(TIMSuc
c)succ fail:(TIMFail)fail;

@end
```

Parameter description:

Parameter	Description
group	**Group ID**
introduction	**Group introduction, the length cannot exceed 120 bytes**
succ	**Success callback**

fail	Failure callback

Example:

```
[[TIMGroupManager sharedInstance] modifyGroupIntroduction:@"TGID1JYSZEAEQ" introduction :@"this i
s one group" succ:^() {
NSLog(@"modify group introduction succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

Modifying the group announcement

Modify group announcement through `modifyGroupNotification` .

Permission description:

- **Public, ChatRoom, and BChatRoom groups: only the group owner or admin can modify the group announcement.**
- **Private group: any member can modify the group announcement.**

Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Modify group announcement
*
* @param group Group ID
* @param notification Group notification, the length cannot exceed 150 bytes
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)modifyGroupNotification:(NSString*)group notification:(NSString*)notification succ:(TIMSuc
c)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
group	**Group ID**
notification	**Group notification, the length cannot exceed 150 bytes**

succ	Success callback
fail	Failure callback

The following example changes the notification of group “TGID1JYSZEAEQ” to “test notification”. Example:

```
[[TIMGroupManager sharedInstance] modifyGroupNotification:@"TGID1JYSZEAEQ" notification:@"test notification" succ:^() {
NSLog(@"modify group notification succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

Modifying the group profile photo

You can modify the group profile photo through `modifyGroupFaceUrl`.

Permission description:

- **Public, ChatRoom, and BChatRoom groups: only the group owner or admin can modify the group profile photo.**
- **Private group: any member can modify the group profile photo.**

Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Modify group profile photo
*
* @param group Group ID
* @param url URL of the group profile photo (cannot exceed 100 bytes)
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)modifyGroupFaceUrl:(NSString*)group url:(NSString*)url succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description

group	Group ID
url	URL of the group profile photo (cannot exceed 100 bytes)
succ	Success callback
fail	Failure callback

Example:

```
[[TIMGroupManager sharedInstance] modifyGroupFaceUrl:@"TGID1JYSZEAEQ" notification:@"http://test/x.jpg" succ:^() {
NSLog(@"modify group face url succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

Modifying the option for joining the group

You can modify the option for joining a group through `modifyGroupAddOpt` .

Permission description:

- **Public group, chat room, and live-streaming group: only the group owner or admin can modify the option for joining the group.**
- **Private group: users can join the group only through invitation and not application.**

Prototype:

```
@interface TIMGroupManager : NSObject

/**
* Modify the option for joining group
*
* @param group Group ID
* @param opt Option for joining group, see TIMGroupAddOpt
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)modifyGroupAddOpt:(NSString*)group opt:(TIMGroupAddOpt)opt succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
group	**Group ID**
opt	**The option for joining the group, which can be set to one of the following: allow anyone to join, approval required, forbid anyone to join**
succ	**Success callback**
fail	**Failure callback**

The following example sets the option for joining the group "TGID1JYSZEAEQ" to "forbid anyone to join". Example:

```
[[TIMGroupManager sharedInstance] modifyGroupAddOpt:@"TGID1JYSZEAEQ" opt:TIM_GROUP_ADD_FORBID suc
c:^() {
NSLog(@"modify group opt succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

Modifying group custom fields

You can modify group custom fields through `modifyGroupCustomInfo`.

Permission description:

- **Relevant keys and permissions need to be configured at the backend.**

Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Modify a collection of group custom fields
*
* @param group Group ID
* @param customInfo Collection of custom fields. Key is of NSString* type and value is of NSData*
type.
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)modifyGroupCustomInfo:(NSString*)group customInfo:(NSDictionary*)customInfo succ:(TIMSucc)
```

```
succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
group	**Group ID**
customInfo	**Collection of custom fields. Key is of NSString* type and value is of NSData* type.**
succ	**Success callback**
fail	**Failure callback**

The following example sets the option for joining the group "TGID1JYSZEAEQ" to "forbid anyone to join". Example:

```
// Configure custom data
NSMutalbeDictionary *customInfo = [[NSMutableDictionary alloc] init];
NSString *key = @"custom key";
NSData *data = [NSData dataWithBytes:"custom value" length:13];
[customInfo setObject:data forKey:key];
[[TIMGroupManager sharedInstance] modifyGroupCustomInfo:@"TGID1JYSZEAEQ" customInfo:customInfo su
cc:^() {
NSLog(@"modify group customInfo succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

Modifying a member's role in the group

You can modify a member's role in the group through `modifyGroupMemberInfoSetRole`.

Permission description:

- **The group owner and admin can modify group members' roles.**
- **Live-streaming group does not support modifying group members' roles.**

Prototype:

```
@interface TIMGroupManager : NSObject
- (int)modifyGroupMemberInfoSetRole:(NSString*)group user:(NSString*)identifier role:(TIMGroupMem
```

```
berRole)role succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
group	**Group ID**
identifier	**The ID of the group member whose name card is to be modified**
role	**The modified role, which can not be the group owner**
succ	**Success callback**
fail	**Failure callback**

The following example sets user “iOS_001” as the admin of group “TGID1JYSZEAEQ”.

Example:

```
[[TIMGroupManager sharedInstance] modifyGroupMemberInfoSetRole:@"TGID1JYSZEAEQ" user:@"iOS_001" r
ole:TIM_GROUP_MEMBER_ADMIN succ:^() {
NSLog(@"modify group member role succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

Muting group members

You can mute group members and set the duration of muting through `modifyGroupMemberInfoSetSilence` .

Permission description:

The group owner and admin can mute group members.

Prototype:

```
@interface TIMGroupManager : NSObject
- (int)modifyGroupMemberInfoSetSilence:(NSString*)group user:(NSString*)identifier stime:(uint32_
t)stime succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description

group	**Group ID**
identifier	**The ID of the group member to be muted**
stime	**Muting duration in seconds**
succ	**Success callback**
fail	**Failure callback**

The following example mutes member “iOS_001” in group “TGID1JYSZEAEQ” for 120 seconds. Example:

```
[[TIMGroupManager sharedInstance] modifyGroupMemberInfoSetSilence:@”TGID1JYSZEAEQ” user:@”iOS_001” stime:120 succ:^() {
NSLog(@”modify group member silence succ”);
}fail:^(int code, NSString* err) {
NSLog(@”failed code: %d %@”, code, err);
}];
```

Modifying a group member’s name card

You can modify a member’s name card in the group through `modifyGroupMemberInfoSetNameCard` .

Prototype:

```
@interface TIMGroupManager : NSObject
- (int)modifyGroupMemberInfoSetNameCard:(NSString*)group user:(NSString*)identifier nameCard:(NSString*)nameCard succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
group	**Group ID**
identifier	**The ID of the group member whose name card is to be modified**
nameCard	**The name card to be modified**
succ	**Success callback**
fail	**Failure callback**

The following example sets the name card of user “iOS_001” in group “TGID1JYSZEAEQ” to “iOS_001_namecard”. Example:

```
[[TIMGroupManager sharedInstance] modifyGroupMemberInfoSetNameCard:@"TGID1JYSZEAEQ" user:@"iOS_001" nameCard:@"iOS_001_namecard" succ:^() {
NSLog(@"modify group member namecard succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

Modifying group member custom fields

Modify group member custom fields through `modifyGroupMemberInfoSetCustomInfo` .

Prototype:

```
@interface TIMGroupManager : NSObject
- (int)modifyGroupMemberInfoSetCustomInfo:(NSString*)group user:(NSString*)identifier customInfo:(NSDictionary<NSString*,NSData*> *)customInfo succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
group	**Group ID**
identifier	**The ID of the group member for whom you want to set a custom property**
customInfo	**Collection of custom fields. Key is of NSString* type and value is of NSData* type.**
succ	**Success callback**
fail	**Failure callback**

The following example sets a custom property for user “iOS_001” in group “TGID1JYSZEAEQ”. Example:

```
[[TIMGroupManager sharedInstance] modifyGroupMemberInfoSetCustomInfo:@"TGID1JYSZEAEQ" user:@"iOS_001" customInfo:customInfo succ:^() {
NSLog(@"modify group member customInfo succ");
}fail:^(int code, NSString* err) {
```

```
NSLog(@"failed code: %d %@", code, err);
}];
```

Modifying the option for receiving group messages

You can set the option for receiving group messages through `modifyReceiveMessageOpt`. By default, public and private groups receive and push group messages offline. Chat rooms and live-streaming groups receive but do not push group messages offline.

Prototype:

```
@interface TIMGroupManager : NSObject
- (int)modifyReceiveMessageOpt:(NSString*)group opt:(TIMGroupReceiveMessageOpt)opt succ:(TIMSucc)
succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
group	**Group ID**
opt	**The option for receiving messages**
succ	**Success callback**
fail	**Failure callback**

The following example sets the option for receiving messages in the group "TGID1JYSZEAEQ" to receive online messages and not receive offline push. Example:

```
[[TIMGroupManager sharedInstance] modifyReciveMessageOpt:@"TGID1JYSZEAEQ" opt:TIM_GROUP_RECEIVE_N
OT_NOTIFY_MESSAGE succ:^() {
NSLog(@"modify receive group message option succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

Group Pending Requests

Pulling group pending requests

You can pull group pending requests through `getPendencyFromServer`. Group pending requests are group-related operations that require approval, such as pending "apply to join" requests and pending "invited to join" requests. Even after a pending request is approved or rejected, the information can still be pulled through this API. In this case, the message pulled has a "processed" flag.

Permission description:

The approver has the permission to pull the information.

Note :

- **If user A applies to join group A, the group admin can obtain information related to this pending request. As user A does not have approval permission, user A does not need to pull information related to this pending request.**
- **If admin A invites user A to join group A, user A can obtain the information related to this pending request because this pending request needs to be approved by user A.**

Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Get the list of group pending requests
*
* @param option Pending parameter configuration
* @param succ Success callback that returns the list of pending requests
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)getPendencyFromServer:(TIMGroupPendencyOption*)option succ:(TIMGetGroupPendencyListSucc)su
cc fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
option	**Pending parameter configuration**
succ	**Success callback that returns the list of pending requests**
fail	**Failure callback**

option parameter description:

Parameter	Description
timestamp	**The start timestamp for pulling. If pending items are pulled from the latest one, enter 0 or leave it empty. In the event of paging, the callback returns the start timestamp of the next page.**
numPerPage	**The number of items pulled at a time. This parameter is used for paging.**

Callback prototype:

```
/**
* Getting the list of group pending requests succeeds
*
* @param meta Metadata of pending request
* @param pendencies Array of pending request list (TIMGroupPendencyItem)
*/
typedef void (^TIMGetGroupPendencyListSucc)(TIMGroupPendencyMeta * meta, NSArray * pendencies);
```

Callback parameter description:

Parameter	Description
meta	**Information returned by the pull operation, including paging information and pulling status**
pendencies	**Array of pending items that are pulled**

Property description:

Property	Description
nextStartTime	**The start timestamp for pulling the next page. If the value is 0, there are no more pages.**
readTimeSeq	**The read timestamp for determining whether a pending item has been read or not**
unReadCnt	**The number of unread items, not limited to the current page**

Properties associated with pending items:

```
/**
* Pending application
*/
@interface TIMGroupPendencyItem : TIMCodingModel
/**
* Group ID
*/
@property(nonatomic,retain) NSString* groupId;
/**
* The ID of the requester, who is the requester for an “apply to join” request and the inviter for an “invited to join” request.
*/
@property(nonatomic,retain) NSString* fromUser;
/**
* The ID of the handler. The ID is 0 for an “apply to join” request and the invitee for an “invited to join” request.
*/
@property(nonatomic,retain) NSString* toUser;
/**
* The time the pending request was added
*/
@property(nonatomic,assign) uint64_t addTime;
/**
* The type of the pending request
*/
@property(nonatomic,assign) TIMGroupPendencyGetType getType;
/**
* “Processed” flag
*/
@property(nonatomic,assign) TIMGroupPendencyHandleStatus handleStatus;
/**
* Processing result
*/
@property(nonatomic,assign) TIMGroupPendencyHandleResult handleResult;
/**
** Additional information of application or invitation
*/
@property(nonatomic,retain) NSString* requestMsg;
/**
* Processing information: approval or rejection information
*/
@property(nonatomic,retain) NSString* handledMsg;
/**
* Approve application
*
* @param msg Reason for approval (optional)
* @param succ Success callback
```

```
* @param fail Failure callback, which returns the error code and error description
*/
-(void) accept:(NSString*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
/**
* Reject application
*
* @param msg Reason for rejection (optional)
* @param succ Success callback
* @param fail Failure callback, which returns the error code and error description
*/
-(void) refuse:(NSString*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
/**
* Your own ID
*/
@property(nonatomic,strong) NSString* selfIdentifier;
@end
```

Property description:

Property	Description
groupId	**Group ID**
fromUser	**The ID of the pending request initiator**
toUser	**The ID of the pending request handler**
addTime	**The time the pending request was added**
getType	**Enumerate pending request types: apply to join, invited to join**
handleStatus	**Enumerates the pending request status: pending, processed by another user, processed by handler (for example, when user A applies to join the group and admin A approves the application, the status of this pending item pulled by admin B is “processed by another user”.)**
handleResult	**Enumerates the processing result: approve, reject**
requestMsg/handleMsg	**The message left during application or processing**

Example:

```
TIMGroupPendencyOption option = [[TIMGroupPendencyOption alloc] init];
option.timestamp = 0;
option.numPerPage = 10;
```

```
[[TIMGroupManager sharedInstance] getPendencyFromServer:option succ:^(TIMGroupPendencyMeta *meta,
NSArray *pendencies) {
NSLog(@"get pendencies succ");
} fail:^(int code, NSString *msg) {
NSLog(@"get pendencies failed: %d->%@", code, msg);
}];
```

Reporting that group pending requests are read

The IM SDK can report that the current pending request and all the pending requests before it have been read. After the report, you can still pull these pending requests and determine whether they are read through the read timestamp.

Prototype:

```
@interface TIMGroupManager : NSObject
/**
* Report that group pending requests are read
*
* @param timestamp Read report timestamp
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Successful
*/
- (int)pendencyReport:(uint64_t)timestamp succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
timestamp	**Read report timestamp. For a single pending message, the timestamp is included in its property.**
succ	**Success callback**
fail	**Failure callback**

Example:

```
[[TIMGroupManager sharedInstance] pendencyReport:timestamp succ:^{
NSLog(@"pendency report succ");
} fail:^(int code, NSString *msg) {
NSLog(@"pendency report failed: %d->%@", code, msg);
}];
```

Processing group pending requests

The IM SDK provides an API for handling group pending requests. The handler can select a single pending request and approve or reject it. Pending requests that have been successfully handled cannot be handled again.

Prototype:

```
@interface TIMGroupPendencyItem: NSObject
/**
* Approve application
*
* @param msg Reason for approval (optional)
* @param succ Success callback
* @param fail Failure callback, which returns the error code and error description
*/
- (void)accept:(NSString*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
/**
* Reject application
*
* @param msg Reason for rejection (optional)
* @param succ Success callback
* @param fail Failure callback, which returns the error code and error description
*/
- (void)refuse:(NSString*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Example:

```
TIMGroupPendencyItem *item = [pendencies firstObject];
[item accept:@"thanks for inviting" succ:^{
NSLog(@"accept succ");
} fail:^(int code, NSString *msg) {
NSLog(@"accept fail: %d->%@", code, msg);
}];
[item refuse:@"i dont want to join" succ:^{
NSLog(@"refuse succ");
} fail:^(int code, NSString *msg) {
NSLog(@"refuse fail: %d->%@", code, msg);
}];
```

Group Event Messages

When a user is invited to join a group or is removed from a group, a tips message is displayed in the group. The caller can decide whether to display and how to display the tips message (for example, ignore it or display it to users as needed). A tips message is identified by a special `Elem` and returned by the new message callback.

Message prototype:

```
/**
* Group tips type
*/
typedef NS_ENUM(NSInteger, TIM_GROUP_TIPS_TYPE){
/**
* Invited to join group (opUser & groupName & userList)
*/
TIM_GROUP_TIPS_TYPE_INVITE = 0x01,
/**
* Quit group (opUser & groupName & userList)
*/
TIM_GROUP_TIPS_TYPE_QUIT_GRP = 0x02,
/**
* Kicked out of group (opUser & groupName & userList)
*/
TIM_GROUP_TIPS_TYPE_KICKED = 0x03,
/**
* Admin is set (opUser & groupName & userList)
*/
TIM_GROUP_TIPS_TYPE_SET_ADMIN = 0x04,
/**
* Admin is cancelled (opUser & groupName & userList)
*/
TIM_GROUP_TIPS_TYPE_CANCEL_ADMIN = 0x05,
/**
* Group profile is modified (opUser & groupName & introduction & notification & faceUrl & owner)
*/
TIM_GROUP_TIPS_TYPE_INFO_CHANGE = 0x06,
/**
* Group member profile is modified (opUser & groupName & memberInfoList)
*/
TIM_GROUP_TIPS_TYPE_MEMBER_INFO_CHANGE = 0x07,
};
/**
* Group Tips
*/
@interface TIMGroupTipsElem : TIMElem
/**
* Group ID
*/
```

```
@property(nonatomic,strong) NSString * group;
/**
* Group tips type
*/
@property(nonatomic,assign) TIM_GROUP_TIPS_TYPE type;
/**
* The user name of the operator
*/
@property(nonatomic,strong) NSString * opUser;
/**
* List of users who are operated on, NSString* array
*/
@property(nonatomic,strong) NSArray * userList;
/**
* The modified group name, otherwise it is nil
*/
@property(nonatomic,strong) NSString * groupName;
/**
* Group information modifications: valid at TIM_GROUP_TIPS_TYPE_INFO_CHANGE, TIMGroupTipsElemGrou
pInfo structure list
*/
@property(nonatomic,strong) NSArray * groupChangeList;
/**
* Member changes: valid at TIM_GROUP_TIPS_TYPE_MEMBER_INFO_CHANGE, TIMGroupTipsElemMemberInfo str
ucture list
*/
@property(nonatomic,strong) NSArray * memberChangeList;
/**
* The user profile of the operator
*/
@property(nonatomic,strong) TIMUserProfile * opUserInfo;
/**
* The group member profile of the operator
*/
@property(nonatomic,strong) TIMGroupMemberInfo * opGroupMemberInfo;
/**
* Modify member' s user profile
*/
@property(nonatomic,strong) NSDictionary * changedUserInfo;
/**
* Modify member' s profile in the group
*/
@property(nonatomic,strong) NSDictionary * changedGroupMemberInfo;
/**
* Current number of group members: valid at TIM_GROUP_TIPS_TYPE_INVITE, TIM_GROUP_TIPS_TYPE_QUIT_
GRP,
* TIM_GROUP_TIPS_TYPE_KICKED
*/
```

```
@property(nonatomic,assign) uint32_t memberNum;
/**
* Operator’ s platform
* Valid values: iOS, Android, Windows, Mac, Web, RESTful API, Unknown
*/
@property(nonatomic,strong) NSString * platform;
@end
```

The following example registers a new message callback and prints event notifications for users joining and leaving the group. The usage of other event notifications is the same. Example:

```
@interface TIMMessageListenerImpl : NSObject
- (void)onNewMessage:(NSArray*) msgs;
@end
@implementation TIMMessageListenerImpl
- (void)onNewMessage:(NSArray*) msgs {
for (TIMMessage * msg in msgs) {
TIMConversation * conversation = [msg getConversation];

for (int i = 0; i < [msg elemCount]; i++) {
TIMElem * elem = [msg getElem:i];
if ([elem isKindOfClass:[TIMGroupTipsElem class]]) {
TIMGroupTipsElem * tips_elem = (TIMGroupTipsElem * )elem;
switch ([tips_elem type]) {
case TIM_GROUP_TIPS_TYPE_INVITE:
NSLog(@”invite %@ into group %@”, [tips_elem userList], [conversation getReceiver]);
break;
case TIM_GROUP_TIPS_TYPE_QUIT_GRP:
NSLog(@”%@ quit group %@”, [tips_elem userList], [conversation getReceiver]);
break;
default:
NSLog(@”ignore type”);
break;
}
}
}
}
}
@end
```

User joins the group

Trigger: when a user joins a group through application or invitation, the system sends a message in the group. The developer can choose a display style and update the group list. The type of the message received is `TIM_GROUP_TIPS_TYPE_INVITE` .

`TIMGroupTipsElem` **parameter description:**

Parameter	Description
type	**TIM_GROUP_TIPS_TYPE_INVITE**
opUser	**Apply to join: the applicant / Invited to join: the inviter**
groupName	**Group name**
userList	**The list of users who are added to the group**

User quits the group

Trigger: when a user quits a group, the system sends a notification in the group. You can choose to update the group member list. The type of the message received is `TIM_GROUP_TIPS_TYPE_QUIT_GRP` **.**

`TIMGroupTipsElem` **parameter description:**

Parameter	Description
type	**TIM_GROUP_TIPS_TYPE_QUIT_GRP**
opUser	**The identifier of the user who quits the group**
groupName	**Group name**

User is kicked out of the group

Trigger: when a user is kicked out of the group, the system sends a notification in the group. You can choose to update the group member list. The type of the message received is `TIM_GROUP_TIPS_TYPE_KICKED` **.**

`TIMGroupTipsElem` **parameter description:**

Parameter	Description
type	**TIM_GROUP_TIPS_TYPE_KICKED**
opUser	**The identifier of the user who is kicked out of the group**
groupName	**Group name**

Group admin is set/canceled

Trigger: when a user is set or canceled as an admin, the system sends a notification in the group. If the UI shows whether a user is an admin, you can update the admin flag. The message types are `TIM_GROUP_TIPS_TYPE_SET_ADMIN` and `TIM_GROUP_TIPS_TYPE_CANCEL_ADMIN`.

`TIMGroupTipsElem` parameter description:

Parameter	Description
type	**Set: TIM_GROUP_TIPS_TYPE_SET_ADMIN**
Cancelled	**TIM_GROUP_TIPS_TYPE_CANCEL_ADMIN**
opUser	**The identifier of the operator**
groupName	**Group name**
userList	**The list of users who are set or canceled as admins**

Group profile modifications

Trigger: when group profile information, such as the group name and group introduction, is modified, the system sends a message. You can update the related display fields, or selectively display the message to users.

`TIMGroupTipsElem` parameter description:

Parameter	Description
type	**TIM_GROUP_TIPS_TYPE_INFO_CHANGE**
opUser	**The identifier of the operator**
groupName	**Group name**
groupChangeInfo	**The modified group profile information. It is a TIMGroupTipsElemGroupInfo structure list.**

`TIMGroupTipsElemGroupInfo` prototype:

```
/**
* Group tips, the modified group information
*/
@interface TIMGroupTipsElemGroupInfo : NSObject
/**
* Type of modification
*/
```

```
@property(nonatomic, assign) TIM_GROUP_INFO_CHANGE_TYPE type;

/**
* Represents different meanings based on the modification type
*/
@property(nonatomic,strong) NSString * value;
@end
```

Parameter description:

Parameter	Description
type	**The type of modification**
value	**The modified value, which represents different meanings based on the modification type**

Group member profile modifications

Trigger: when a group member's profile related to the group is modified, such as when the member has been muted or the member's role in the group has changed, the system sends a message. You can update related display fields, or selectively display the message to users.

> ⚠ **Note :**
>
> - **The profile mentioned here includes only information related to the group, such as muting duration and member role change. Information related to the user such as the user's nickname is not included. For groups that have too many members, we recommend that you display the information in the message body instead of updating it in real time.**
> - **If the user profile is stored locally, you can determine whether a change has occurred according to the information in the message body and update the profile after receiving a message from this user.**

`TIMGroupTipsElem` **parameter description:**

Parameter	Description
type	**TIM_GROUP_TIPS_TYPE_MEMBER_INFO_CHANGE**
opUser	**The identifier of the operator**
groupName	**Group name**

memberInfoList	The modified group member profile information. It is a `TIMGroupTipsElemMemberInfo` structure list.

`TIMGroupTipsElemMemberInfo` **prototype:**

```
/**
* Group tips, modified member information
*/
@interface TIMGroupTipsElemMemberInfo : NSObject
/**
* The user whose profile is modified
*/
@property(nonatomic,retain) NSString * identifier;
/**
* Muting duration in seconds
*/
@property(nonatomic,assign) uint32_t shutupTime;
@end
```

Parameter description:

Parameter	Description
identifier	**The identifier of the user whose profile is modified**
shutupTime	**Muting duration**

Group event message listeners

The group event messages of chat rooms and live-streaming groups are obtained by registering listeners in TIMManager -> setUserConfig -> TIMUserConfig -> groupEventListener. The `Elem` of the message contains the number of group members.

```
/**
* Group event notification callback
*/
@protocol TIMGroupEventListener <NSObject>
@optional
/**
* Group tips callback
*
* @param elem Group tips message
*/
- (void)onGroupTipsEvent:(TIMGroupTipsElem*)elem {
// Group ID
```

```
NSString *groupID = elem.group;
// The operator
NSString *opUser = elem.opUser;
// The user operated on
NSArray *userList = elem.userList;
switch (elem.type) {
case TIM_GROUP_TIPS_TYPE_INVITE:
// userList joined the group. For a private group, the message can be displayed as "opUser invites userList to join the group".
// For other types of groups, the message can be displayed as "userList joined the group".
break;
case TIM_GROUP_TIPS_TYPE_QUIT_GRP:
// opUser quits the group
break;
case TIM_GROUP_TIPS_TYPE_KICKED:
// opUser kicks userList out of the group
break;
case TIM_GROUP_TIPS_TYPE_SET_ADMIN:
// opUser sets userList as admin
break;
case TIM_GROUP_TIPS_TYPE_CANCEL_ADMIN:
// opUser cancels userList as admin
break;
case TIM_GROUP_TIPS_TYPE_INFO_CHANGE:
// groupID group profile information is changed
break;
case TIM_GROUP_TIPS_TYPE_MEMBER_INFO_CHANGE:
// groupID member group profile information is changed
break;
default:
break;
}
}
@end
```

Group System Messages

When a user applies to join a group, the group admin receives a system message on the application. The admin can accept or reject the application, and the corresponding group system message will be displayed to the user.

Group system message types:

```
/**
* Group system message type
*/
typedef NS_ENUM(NSInteger, TIM_GROUP_SYSTEM_TYPE){
/**
* Request to join group (received only by the admin)
*/
TIM_GROUP_SYSTEM_ADD_GROUP_REQUEST_TYPE = 0x01,
/**
* Application to join group is approved (received only by the applicant)
*/
TIM_GROUP_SYSTEM_ADD_GROUP_ACCEPT_TYPE = 0x02,
/**
* Application to join group is rejected (received only by the applicant)
*/
TIM_GROUP_SYSTEM_ADD_GROUP_REFUSE_TYPE = 0x03,
/**
* Kicked out of the group by the admin (received by the user who is kicked out)
*/
TIM_GROUP_SYSTEM_KICK_OFF_FROM_GROUP_TYPE = 0x04,
/**
* Group disbanded (received by all members)
*/
TIM_GROUP_SYSTEM_DELETE_GROUP_TYPE = 0x05,
/**
* Group created (received only by the creator)
*/
TIM_GROUP_SYSTEM_CREATE_GROUP_TYPE = 0x06,
/**
* Invited to join group (received by the invitee)
*/
TIM_GROUP_SYSTEM_INVITED_TO_GROUP_TYPE = 0x07,
/**
* Quit group (received by the user who quits group)
*/
TIM_GROUP_SYSTEM_QUIT_GROUP_TYPE = 0x08,
/**
* Group admin set (received by the new group admin)
*/
TIM_GROUP_SYSTEM_GRANT_ADMIN_TYPE = 0x09,
/**
* Admin status canceled (received only by the canceled admin)
*/
TIM_GROUP_SYSTEM_CANCEL_ADMIN_TYPE = 0x0a,
/**
* Group revoked (received by all members)
*/
```

```
TIM_GROUP_SYSTEM_REVOKE_GROUP_TYPE = 0x0b,
/**
* Group invitation request (received by the invitee)
*/
TIM_GROUP_SYSTEM_INVITE_TO_GROUP_REQUEST_TYPE = 0x0c,
/**
* Group invitation request approved (received only by the inviter)
*/
TIM_GROUP_SYSTEM_INVITE_TO_GROUP_ACCEPT_TYPE = 0x0d,
/**
* Group invitation request rejected (received only by the inviter)
*/
TIM_GROUP_SYSTEM_INVITE_TO_GROUP_REFUSE_TYPE = 0x0e,
/**
* Custom notification (received by all members by default)
*/
TIM_GROUP_SYSTEM_CUSTOM_INFO = 0xff,
};
/**
* Group system message
*/
@interface TIMGroupSystemElem : TIMElem
/**
* Operation type
*/
@property(nonatomic,assign) TIM_GROUP_SYSTEM_TYPE type;
/**
* Group ID
*/
@property(nonatomic,strong) NSString * group;
/**
* Operator
*/
@property(nonatomic,strong) NSString * user;
/**
* The reason for operation
*/
@property(nonatomic,strong) NSString * msg;
/**
* Message identifier, irrelevant to the client
*/
@property(nonatomic,assign) uint64_t msgKey;
/**
* Message identifier, irrelevant to the client
*/
@property(nonatomic,strong) NSData * authKey;
/**
* Custom pass-through message body (valid when type == TIM_GROUP_SYSTEM_CUSTOM_INFO)
```

```
*/
@property(nonatomic,strong) NSData * userData;
/**
* The user profile of the operator
*/
@property(nonatomic,strong) TIMUserProfile * opUserInfo;
/**
* The group member profile of the operator
*/
@property(nonatomic,strong) TIMGroupMemberInfo * opGroupMemberInfo;
/**
* Operator’s platform
* Valid values: iOS, Android, Windows, Mac, Web, RESTful API, Unknown
*/
@property(nonatomic,strong) NSString * platform;
@end
```

Parameter description:

Parameter	Description
type	**Message type**
group	**Group ID**
user	**The operator**
msg	**The reason for the operation**
msgKey & authKey	**Message identifiers that are irrelevant to the client. They are read by the IM SDK in the event of accept and refuse**

The following example processes group system messages that are received. Applications to join the group are approved by default and messages notifying that the group was disbanded are printed. Other types of messages are parsed in the same way. Example:

```
@interface TIMMessageListenerImpl : NSObject
- (void)onNewMessage:(NSArray*) msgs;
@end
@implementation TIMMessageListenerImpl
- (void)onNewMessage:(NSArray*) msgs {
for (TIMMessage * msg in msgs) {
for (int i = 0; i < [msg elemCount]; i++) {
TIMElem * elem = [msg getElem:i];
if ([elem isKindOfClass:[TIMGroupSystemElem class]]) {
TIMGroupSystemElem * system_elem = (TIMGroupSystemElem * )elem;
```

```
switch ([system_elem type]) {
case TIM_GROUP_SYSTEM_ADD_GROUP_REQUEST_TYPE:
NSLog(@"user %@ request join group %@", [system_elem user], [system_elem group]);
break;
case TIM_GROUP_SYSTEM_DELETE_GROUP_TYPE:
NSLog(@"group %@ deleted by %@", [system_elem group], [system_elem user]);
break;
default:
NSLog(@"ignore type");
break;
}
}
}
}
}
@end
```

User applies to join a group

Trigger: when a user applies to join a group, the group admin receives a message about the user applying to join the group. You can display the message to the user for the user to decide whether to accept the application. The message type is `TIM_GROUP_SYSTEM_ADD_GROUP_REQUEST_TYPE`.

Parameter description:

Parameter	Description
type	**TIM_GROUP_SYSTEM_ADD_GROUP_REQUEST_TYPE**
group	**The ID of the group that the user applies to join**
user	**The applicant**
msg	**The reason for application (optional)**

Method description:

- **To accept the application, call the accept method.**
- **To reject the application, call the refuse method.**

"Apply to join" request is approved/rejected

Trigger: when an admin approves an application to join the group, the applicant receives an approval notification message, and when the admin rejects the application, the applicant receives a rejection notification message.

Parameter description:

Parameter	Description
type	**Approve: TIM_GROUP_SYSTEM_ADD_GROUP_ACCEPT_TYPE Reject: TIM_GROUP_SYSTEM_ADD_GROUP_REFUSE_TYPE**
group	**The ID of the group for which the request is approved/rejected**
user	**The identifier of the admin who processed the request**
msg	**Reason for approval or rejection (optional)**

"Invited to join" request

Trigger: when invited to join a group, the invitee receives an invitation message. If the invitee approves the invitation, the accept method is called. Otherwise, the refuse method is called. The message type is `TIM_GROUP_SYSTEM_INVITE_TO_GROUP_REQUEST_TYPE`.

Parameter description:

Parameter	Description
type	**TIM_GROUP_SYSTEM_INVITE_TO_GROUP_REQUEST_TYPE**
group	**The ID of the group that the user is invited to join**
user	**The inviter**

Method description:

- **To accept the application, call the accept method.**
- **To reject the application, call the refuse method.**

"Invited to join" request is approved/rejected

Trigger: when the invitee approves the invitation, the inviter receives an approval notification message, and when the invitee rejects the invitation, the inviter receives a rejection notification message.

Parameter description:

Parameter	Description
type	**Approve: TIM_GROUP_SYSTEM_INVITE_TO_GROUP_ACCEPT_TYPE Reject: TIM_GROUP_SYSTEM_INVITE_TO_GROUP_REFUSE_TYPE**

group	**The ID of the group for which the request is approved/rejected**
user	**The identifier of the user who processed the request**
msg	**Reason for approval or rejection (optional)**

Kicked out by admin

Trigger: when a user is kicked out of a group by the admin, the user receives a kicked out notification message.

Parameter description:

Parameter	Description
type	**TIM_GROUP_SYSTEM_KICK_OFF_FROM_GROUP_TYPE**
group	**The ID of the group from which the user is kicked out**
user	**The identifier of the admin who performed the action**

Group is disbanded

Trigger: when a group is disbanded, all members receive a message notifying them that the group is disbanded.

Parameter description:

Parameter	Description
type	**TIM_GROUP_SYSTEM_DELETE_GROUP_TYPE**
group	**The ID of the group that is disbanded**
user	**The identifier of the admin who performed the action**

Group is created

Trigger: when a group is created, the creator receives a creation notification message. If a user calls the method to create a group and the success callback is returned, then the group was created successfully. This message is used for multi-client synchronization. When receiving this message, the other clients can update the group list, while the current client can choose to ignore it.

Parameter description:

Parameter	Description
type	**TIM_GROUP_SYSTEM_CREATE_GROUP_TYPE**
group	**The ID of the group that is created**
user	**The creator, who is the user in this case**

Invited to join group

Trigger: when a user is invited to join a group, the user receives an invitation message. Initial members are added to the group without invitation when the group is created.

Parameter description:

Parameter	Description
type	**TIM_GROUP_SYSTEM_INVITED_TO_GROUP_TYPE**
group	**The ID of the group that the user is invited to join**
user	**The operator, who is the inviter in this case**

Method description:

- **To accept the application, call the accept method.**
- **To reject the application, call the refuse method.**

Quit group

Trigger: when a user quits a group, the user receives a quit notification message, and only the user who quit will receive the message. If the user calls `QuitGroup` and the success callback is returned, then the user has quit the group successfully. This message serves the purpose of multi-client synchronization. When receiving this message, the other clients can update the group list, while the current client can choose to ignore it.

Parameter description:

Parameter	Description
type	**TIM_GROUP_SYSTEM_QUIT_GROUP_TYPE**
group	**The ID of the group that the user quits**
user	**The operator, who is the user who quits the group in this case**

Admin is set/cancelled

Trigger: when a user is set or canceled as a group admin, the user receives a corresponding notification.

Parameter description:

Parameter	Description
type	**Admin role is cancelled: TIM_GROUP_SYSTEM_GRANT_ADMIN_TYPE Admin role is granted: TIM_GROUP_SYSTEM_CANCEL_ADMIN_TYPE**
group	**The ID of the group in which the event occurs**
user	**The operator**

Group is revoked

Trigger: when a group is revoked by the system, all members receive a message notifying that the group was revoked.

Parameter description:

Parameter	Description
type	**TIM_GROUP_SYSTEM_REVOKE_GROUP_TYPE**
group	**The ID of the group that is revoked**

Send and Receive Messages
Sending and Receiving Messages (Android)

Last updated : 2021-10-09 11:32:37

Sending Messages

Sending common messages

Obtain a conversation: conversations are classified into conversations with an individual user and conversations with a group. To send or receive messages in a conversation, you first need to obtain the conversation by specifying the conversation type (C2C conversation or group conversation) and the peer's identifier (the peer's account or group ID). To obtain a conversation, call `getConversation` of `TIMManager`.

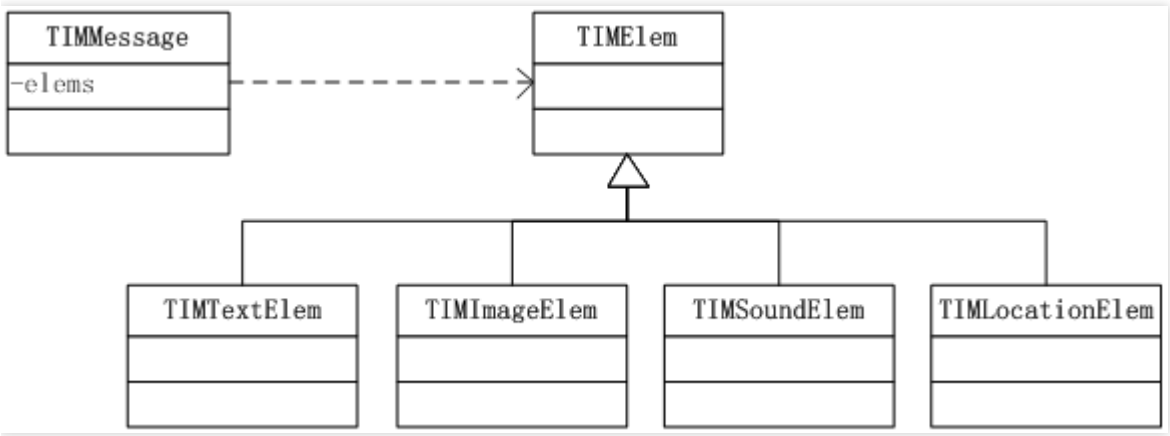
Prototype:

```
/**
* Obtain a conversation.
* @param type Conversation type
* @param peer Peer in the conversation. For a C2C conversation, use the peer's account identifie
r. For a group conversation, use the group ID.
* @return Conversation instance
*/
public TIMConversation getConversation(TIMConversationType type, String peer)
```

Example:

```
// Obtain a C2C conversation.
String peer = "sample_user_1"; // Obtain the conversation with user "sample_user_1".
conversation = TIMManager.getInstance().getConversation(
TIMConversationType.C2C, // Conversation type: C2C conversation
peer); // Peer's account // Peer's ID
// Obtain a group conversation.
String groupId = "TGID1EDABEAE0"; // Obtain the conversation with group "TGID1LTTZEAE0".
conversation = TIMManager.getInstance().getConversation(
TIMConversationType.Group, //Conversation type: group conversation
groupId); // Group ID
```

Send messages: after `TIMConversation` is obtained using `TIMManager`, you can send messages and obtain cached messages for the conversation. For more information about the interpretation of messages in the IM SDK, see Introduction to IM SDK Objects. In the IM SDK, a message is a `TIMMessage` object. A `TIMMessage` can contain multiple `TIMElem`, and each `TIMElem` can be a text or an image. That is, a message can contain multiple texts and images.



To send messages, use the `sendMessage` method of `TIMConversation`.

Prototype:

```
/**
* Send a message.
* @param msg Message
* @param callback Callback
*/
public void sendMessage(@NonNull TIMMessage msg, @NonNull TIMValueCallBack<TIMMessage> callback)
```

Sending text messages

A text message is defined by `TIMTextElem`. `The TIMTextElem` member methods are as follows:

```
// Obtain the text content.
java.lang.String getText()
// Set the text content. 'text' passes the text message to send.
void setText(java.lang.String text)
```

Example:

```
// Construct a message.
TIMMessage msg = new TIMMessage();
// Add the text content.
TIMTextElem elem = new TIMTextElem();
elem.setText("a new msg");
// Add 'elem' to the message.
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}
// Send a message.
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {// Callback for sending a messa
ge
@Override
public void onError(int code, String desc) {// Failed to send the message.
// 'code' (error code) and 'desc' (error description) can be used to locate the cause of the requ
est failure.
// For more information about the meaning of error codes, see the Error Code table.
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}
@Override
public void onSuccess(TIMMessage msg) {// Succeeded in sending the message.
Log.e(tag, "SendMsg ok");
}
});
```

Sending image messages

An image message is defined by `TIMImageElem`, which is a subclass of `TIMElem`. That is, the content of an image message includes one or more images. To send an image, add `TIMImageElem` to `TIMMessage` and send the image along with the message.

> **Note :**
> **`path` does not support file paths starting with `file://`. Therefore, the `file://` prefix must be removed.**

The `TIMImageElem` member methods are as follows:

```
/**
* Obtain the list of images contained in `elem`. It can be called when the IM SDK fetches `elem`.
* @return elem List of images contained in `elem`
*/
```

```
public ArrayList<TIMImage> getImageList()
/**
* Obtain the local file path of the original image. This method is valid only for message sender
s.
* @return Local file path
*/
public String getPath()
/**
* Set the file path of the original image to be sent.
* @param path File path of the original image
*/
public void setPath(String path)
/**
* Obtain the image quality level.
* @return Image quality level. 0: original image. 1: highly compressed image (small image). 2: hi
gh-resolution image (large image).
*/
public int getLevel()
/**
* Set the image quality level.
* @param level 0: original image. 1: highly compressed image (small image), default value. 2: hig
h-resolution image (large image).
*/
public void setLevel(int level)
/**
* Cancel image upload.
* @return Whether image upload was canceled
*/
public boolean cancelUploading()
/**
* Obtain the ID of the image upload task. The returned value of this API is valid after `sendMess
age` is called.
* @return ID of the image upload task
*/
public int getTaskId()
/**
* Obtain the image type.
* @return Image type
*/
public int getImageFormat()
```

To send an image, you only need to set `path`, which is the image path. After the image is sent successfully, you can call `getImageList` to obtain all image types. `TIMImage` stores the image type, size, width, and height. To download the binary data of the image, call `getImage`.

`The TIMImage` member methods are as follows:

```
/**
* Obtain an image.
* @param path Image storage path
* @param cb Callback
*/
public void getImage(@NonNull final String path, @NonNull final TIMCallBack cb)
/**
* Obtain the image type.
* @return Image type
*/
public TIMImageType getType()
/**
* Obtain the UUID.
* @return UUID, which can be used as the unique key for caching
*/
public String getUuid()
/**
* Obtain the image size.
* @return Image size.
*/
public long getSize()
/**
* Obtain the image height.
* @return Image height.
*/
public long getHeight()
/**
* Obtain the image width.
* @return Image width
*/
public long getWidth()
/**
* Obtain the image URL.
* @return Image URL
*/
public String getUrl()
```

Example:

```
// Construct a message.
TIMMessage msg = new TIMMessage();
// Add an image.
TIMImageElem elem = new TIMImageElem();
elem.setPath(Environment.getExternalStorageDirectory() + "/DCIM/Camera/1.jpg");
// Add `elem` to the message.
if(msg.addElement(elem) != 0) {
```

```
Log.d(tag, "addElement failed");
return;
}
// Send a message.
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {// Callback for sending a messa
ge
@Override
public void onError(int code, String desc) {// Failed to send the message.
// 'code' (error code) and 'desc' (error description) can be used to locate the cause of the requ
est failure.
// For more information about the list of error codes, see the Error Code table.
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}
@Override
public void onSuccess(TIMMessage msg) {// Succeeded in sending the message.
Log.e(tag, "SendMsg ok");
}
});
```

Sending emoji messages

An emoji message is defined by `TIMFaceElem`. The IM SDK does not provide emoji packages. Developers can use `index` to store the index entries of the emojis in their emoji packages. Alternatively, they can directly use `data` to store emoji binary data and the string `key`. Using either method, users can customize emojis. The IM SDK only passes them through.

`The TIMFaceElem` member methods are as follows:

```
/**
* Obtain an emoji index.
* @return Emoji index
*/
public int getIndex()
/**
* Set the emoji index.
* @param index Emoji index
*/
public void setIndex(int index)
/**
* Obtain custom emoji data.
* @return Custom emoji data
*/
public byte[] getData()
/**
* Set custom emoji data.
```

```
 * @param data Custom emoji data
 */
public void setData(byte[] data)
```

Example:

```
// Construct a message.
TIMMessage msg = new TIMMessage();
// Add an emoji.
TIMFaceElem elem = new TIMFaceElem();
elem.setData(sampleByteArray); // Custom byte[]
elem.setIndex(10); // Custom emoji index
// Add `elem` to the message.
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}
// Send a message.
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {// Callback for sending a messa
ge
@Override
public void onError(int code, String desc) {// Failed to send the message.
// 'code' (error code) and 'desc' (error description) can be used to locate the cause of the requ
est failure.
// For more information about the meaning of error codes, see the Error Code table.
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}
@Override
public void onSuccess(TIMMessage msg) {// Succeeded in sending the message.
Log.e(tag, "SendMsg ok");
}
});
```

Sending audio messages

An audio message is defined by `TIMSoundElem`, where `data` stores audio data. For audio data, you need to provide the audio length in seconds.

Note :

- **A message can contain only one audio `Elem`. If you attempt to add multiple audio `Elem` objects, the `AddElem` function returns Error 1 and the audio `Elem` objects cannot be added.**

- **Audio and file `Elem` objects are not always received in the order that they are added. We recommend that you determine and display `Elem` objects one by one. Moreover, audio and file `Elem` objects may not be sorted in the order that they are sent.**
- **`path` does not support file paths starting with `file://`. The `file://` prefix must be removed.**

The `TIMSoundElem` member methods are as follows:

```
/**
* Download and save an audio file to the specified path.
*
* @param path Specified storage path
* @param progressCb Download progress callback
* @param cb Callback
*/
public void getSoundToFile(@NonNull final String path, final TIMValueCallBack<ProgressInfo> progr
essCb, @NonNull final TIMCallBack cb)
/**
* Obtain the path of the audio file to be sent. This method is valid only for message senders.
* @return Audio file path
*/
public String getPath()
/**
* Set the path of the audio file to be uploaded. (If the file path is specified, the audio file i
n the specified file path will be uploaded first.)
* @param path Audio file path
*/
public void setPath(String path)
/**
* Obtain the UUID.
* @return uuid
*/
public String getUuid()
/**
* Obtain the length of binary data.
* @return Length of binary data
*/
public long getDataSize()
/**
* Obtain the audio length.
* @return Audio length
*/
public long getDuration()
/**
```

```
* Set the audio length.
* @param duration Audio length
*/
public void setDuration(long duration)
/**
* Obtain the ID of the audio upload task. The returned value of this API is valid after `sendMess
age` is called.
* @return ID of the audio file upload task
*/
public int getTaskId()
```

Example:

```
// Construct a message.
TIMMessage msg = new TIMMessage();
// Add an audio file.
TIMSoundElem elem = new TIMSoundElem();
elem.setPath(filePath); // Enter the audio file path.
elem.setDuration(20); // Enter the audio length.
// Add `elem` to the message.
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}
// Send a message.
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {// Callback for sending a messa
ge
@Override
public void onError(int code, String desc) {// Failed to send the message.
// 'code' (error code) and 'desc' (error description) can be used to locate the cause of the requ
est failure.
// For more information about the meaning of error codes, see the Error Code table.
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}
@Override
public void onSuccess(TIMMessage msg) {// Succeeded in sending the message.
Log.e(tag, "SendMsg ok");
}
});
```

Sending location messages

A location message is defined by `TIMLocationElem`, where `desc` stores the location description information, and `longitude` and `latitude` represent the location longitude and latitude, respectively.

The `TIMLocationElem` member methods are as follows:

```
/**
* Obtain the location description.
* @return Location description
*/
public String getDesc()
/**
* Set the location description.
* @param desc Location description
*/
public void setDesc(String desc)
/**
* Obtain the longitude.
* @return Longitude
*/
public double getLongitude()
/**
* Set the longitude.
* @param longitude Longitude
*/
public void setLongitude(double longitude)
/**
* Obtain the latitude.
* @return Latitude
*/
public double getLatitude()
/**
* Set the latitude.
* @param latitude Latitude
*/
public void setLatitude(double latitude)
```

Example:

```
// Construct a message.
TIMMessage msg = new TIMMessage();
// Add location information.
TIMLocationElem elem = new TIMLocationElem();
elem.setLatitude(113.93); // Set the latitude.
elem.setLongitude(22.54); // Set the longitude.
elem.setDesc("Tencent Building");
// Add 'elem' to the message.
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}
// Send a message.
```

```
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {// Callback for sending a messa
ge
@Override
public void onError(int code, String desc) {// Failed to send the message.
// 'code' (error code) and 'desc' (error description) can be used to locate the cause of the requ
est failure.
// For more information about the meaning of error codes, see the Error Code table.
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}
@Override
public void onSuccess(TIMMessage msg) {// Succeeded in sending the message.
Log.e(tag, "SendMsg ok");
}
});
```

Sending file messages

A file message is defined by `TIMFileElem`. You can also view additional information such as the file name.

Note :

- **Audio and file `Elem` objects are not always received in the order that they are added. We recommend that you determine and display `Elem` objects one by one. Moreover, audio and file `Elem` objects may not be sorted in the order that they are sent.**
- **`path` does not support file paths starting with `file://`. The `file://` prefix must be removed.**
- **The maximum file size is 28 MB.**

The `TIMFileElem` member methods are as follows:

```
/**
* Download and save a file to the specified path.
* @param path Specified storage path
* @param callback Callback
*/
public void getToFile(@NonNull final String path, @NonNull TIMCallBack callback)
/**
* Obtain the UUID.
* @return UUID, which can be used as the unique key for caching
*/
public String getUuid()
```

```
/**
* Obtain the file size.
* @return File size
*/
public long getFileSize()
/**
* Obtain the file name.
* @return File name
*/
public String getFileName()
/**
* Set the file name when sending the file.
* @param fileName File name
*/
public void setFileName(String fileName)
/**
* Obtain the path of the uploaded file. This method is valid only for message senders.
* @return File path
*/
public String getPath()
/**
* Set the path of the file to be uploaded. (If the file path is specified, the file in the specif
ied file path will be uploaded first.)
* @param path File path
*/
public void setPath(String path)
/**
* Obtain the ID of the file upload task. The returned value of this API is valid after `sendMessa
ge` is called.
* @return ID of the file upload task
*/
public int getTaskId()
```

Example:

```
// Construct a message.
TIMMessage msg = new TIMMessage();
// Add the file content.
TIMFileElem elem = new TIMFileElem();
elem.setPath(filePath); // Set the file path.
elem.setFileName("myfile.bin"); // Set the file name for displaying the message.
// Add `elem` to the message.
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}
// Send a message.
```

```
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {// Callback for sending a messa
ge
@Override
public void onError(int code, String desc) {// Failed to send the message.
// 'code' (error code) and 'desc' (error description) can be used to locate the cause of the requ
est failure.
// For more information about the meaning of error codes, see the Error Code table.
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}
@Override
public void onSuccess(TIMMessage msg) {// Succeeded in sending the message.
Log.e(tag, "SendMsg ok");
}
});
```

Sending custom messages

Developers can customize the message format and content when built-in message types cannot meet their special needs. The IM SDK only passes through custom messages. If iOS APNs push notifications are required, a push text description to be displayed needs to be provided. A custom message is defined by `TIMCustomElem`, where 'data' stores the binary data of the message, its format is defined by developers, and `desc` stores the description text. A message can contain multiple custom `Elem` objects, which can be mixed with other `Elem` objects. In offline push scenarios, the `desc` of each `Elem` are stacked and delivered.

The `TIMCustomElem` member methods are as follows:

```
/**
* Obtain the custom data.
* @return Custom data
*/
public byte[] getData()
/**
* Set the custom data.
* @param data Custom data
*/
public void setData(byte[] data)
/**
* Obtain custom description.
* @return Custom description
*/
public String getDesc()
/**
* Set the custom description.
```

```
* @param desc Custom description
*/
public void setDesc(String desc)
/**
* Obtain the `ext` field pushed by the backend.
* @return ext
*/
public byte[] getExt()
/**
* Set the `ext` field pushed by the backend.
* @param ext The `ext` field pushed by the backend
*/
public void setExt(byte[] ext)
/**
* Obtain the custom sound.
* @return Custom sound data
*/
public byte[] getSound()
/**
* Set custom sound data.
* @param data Custom sound data
*/
public void setSound(byte[] data)
```

The following example shows how to add an XML message, where the specific display is determined by the developer. Example:

```
// Construct a message.
TIMMessage msg = new TIMMessage();
// Custom XML message
String sampleXml = "<!--?xml version='1.0' encoding="utf-8"?-->testTitlethis is custom msgtest ms
g body";
// Add custom content to `TIMMessage`.
TIMCustomElem elem = new TIMCustomElem();
elem.setData(sampleXml.getBytes()); // Custom byte[]
elem.setDesc("this is one custom message"); // Custom description
// Add `elem` to the message.
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}
// Send a message.
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {// Callback for sending a messa
ge
@Override
public void onError(int code, String desc) {// Failed to send the message.
// 'code' (error code) and 'desc' (error description) can be used to locate the cause of the requ
```

```
est failure.
// For more information about the meaning of error codes, see the Error Code table.
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}
@Override
public void onSuccess(TIMMessage msg) {// Succeeded in sending the message.
Log.e(tag, "SendMsg ok");
}
});
```

Sending short video messages

A short video message is defined by `TIMVideoElem`, which is a subclass of `TIMElem`. These messages can contain video snapshots and content. To send a short video, add `TIMVideoElem` to `TIMMessage` and send the video along with the message.

`TIMVideoElem` prototype:

```
/**
* Obtain the ID of the short video upload task. The returned value of this API is valid after `se
ndMessage` is called.
*
* @return ID of the short video upload task
*/
public long getTaskId() {
return this.taskId;
}
/**
* Set short video information when sending the message.
*
* @param video Short video information. For more information, see {@link TIMVideo}.
*/
public void setVideo(TIMVideo video) {
this.video = video;
}
/**
* Obtain video information.
*
* @return Video information. For more information, see {@link TIMVideo}.
*/
public TIMVideo getVideoInfo() {
return this.video;
}
/**
* Set the video file path when sending the message.
*
* @param path Video file path
```

```
*/
public void setVideoPath(String path) {
this.videoPath = path;
}
/**
* Obtain the video file path.
*
* @return Video file path
*/
public String getVideoPath() {
return this.videoPath;
}
/**
* Set short video snapshot information when sending the message.
*
* @param snapshot Short video snapshot information. For more information, see {@link TIMSnapsho
t}.
*/
public void setSnapshot(TIMSnapshot snapshot) {
this.snapshot = snapshot;
}
/**
* Obtain video snapshot information.
*
* @return Video snapshot information. For more information, see {@link TIMSnapshot}.
*/
public TIMSnapshot getSnapshotInfo() {
return this.snapshot;
}
/**
* Set the short video snapshot file path when sending the message.
*
* @param path Short video snapshot file path
*/
public void setSnapshotPath(String path) {
this.snapshotPath = path;
}
/**
* Obtain the short video snapshot file path.
*
* @return Short video snapshot file path
*/
public String getSnapshotPath() {
return this.snapshotPath;
}
```

Parameter description:

Parameter	Description
taskId	**The upload task ID, which can be used to query the upload progress. This parameter has been deprecated. Therefore, use `TIMUploadProgressListener` instead to listen to the upload progress.**
videoPath	**The path of the local video to be sent.**
video	**The video information. Set the `type` and `duration` parameters when sending the message.**
snapshotPath	**The local snapshot path of the short video to be sent.**
snapshot	**The snapshot information. Set the `type` and `duration` parameters when sending the message.**

The following example shows how to send a short video message. Example:

```
// Construct a message.
TIMMessage msg = new TIMMessage();
// Construct a short video object.
TIMVideoElem ele = new TIMVideoElem();
TIMVideo video = new TIMVideo();
video.setDuaration(duration / 1000); // Set the video length.
video.setType("mp4"); // Set the video file type.
TIMSnapshot snapshot = new TIMSnapshot();
snapshot.setWidth(width); // Set the video snapshot width.
snapshot.setHeight(height); // Set the video snapshot height.
ele.setSnapshot(snapshot);
ele.setVideo(video);
ele.setSnapshotPath(imgPath);
ele.setVideoPath(videoPath);

// Add `elem` to the message.
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}
// Send a message.
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {// Callback for sending a messa
ge
@Override
public void onError(int code, String desc) {// Failed to send the message.
// 'code' (error code) and 'desc' (error description) can be used to locate the cause of the requ
est failure.
// For more information about the meaning of error codes, see the Error Code table.
```

```
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}
@Override
public void onSuccess(TIMMessage msg) {// Succeeded in sending the message.
Log.e(tag, "SendMsg ok");
}
});
```

'Elem' order

Currently, file and audio `Elem` objects might not be transferred in the order that they are added. Other `Elem` objects are transferred in the order that they are added. However, we recommend that you do not heavily rely on the `Elem` object sequence when processing elements. Instead, you should process `Elem` objects by type to prevent process crashes when exceptions occur.

Online messages

In some scenarios, you need to send online messages, which can only be received when a user is online. If the user is not online when the messages are sent, the user will not see them upon the next login. Online messages can be used for notifications. However, online messages will not be stored or included in the unread count. The API for sending online messages is similar to `sendMessage` .
If you don't want to receive offline pushes, you can call the `setOfflinePushSettings` and set the `TIMOfflinePushSettings` and `setEnabled(false)` parameters to disable push.

> **Note :**
> **In versions earlier than 2.5.3, online messages apply only to C2C conversations. In version 2.5.3 or later, online messages apply to group conversations, excluding audio-video chat rooms (AVChatRoom) and broadcasting chat rooms (BChatRoom).**

```
* Send an online message (the server does not save the message).
public void sendOnlineMessage(TIMMessage msg, TIMValueCallBack<TIMMessage> callback)
```

Forwarding messages

You can call `copyFrom` of `TIMMessage` to easily copy the content of another message to the current message and resend the message to other contacts.

Prototype:

```
/**
* Copy the message content to the current message, including `elem`, `priority`, `online`, and `offlinePushInfo`.
* @param srcMsg Source message
* @return true Copied successfully
*/
public boolean copyFrom(@NonNull TIMMessage srcMsg)
```

Receiving Messages

To be notified of new messages, you need to register the new message notification callback `TIMMessageListener`. If you have logged in to the IM console, the IM SDK uses the `onNewMessages` callback to send new messages. For more information about how to register the new message notification callback, see New Message Notification.

> **Note :**
> **Messages obtained using `onNewMessages` may not be unread messages. They can also be messages that have not been displayed locally. For example, when messages have been read on another client, messages of recent contacts can be pulled to obtain the latest messages in conversations. If these latest messages are not stored locally, they are sent using this method. After a user logs in, the IM SDK gets C2C offline messages. To avoid missing message notifications, the user needs to register new message notifications before login.**
> **The `onNewMessage` callback is also used to send group system messages, relationship chain changes, and friend profile changes.**

Parsing messages

After receiving a message, use `getElem` to obtain all `Elem` nodes in `TIMMessage`. The following is the prototype for traversing `Elem` nodes:

```
// Obtain message elements.
TIMElem getElement(int i)
// Obtain the number of elements.
int getElementCount()
```

Example:

```
TIMMessage msg = /* Message */
for(int i = 0; i < msg.getElementCount(); ++i) {
TIMElem elem = msg.getElement(i);
// Obtain the type of the current element.
TIMElemType elemType = elem.getType();
Log.d(tag, "elem type: " + elemType.name());
if (elemType == TIMElemType.Text) {
// Process text messages.
} else if (elemType == TIMElemType.Image) {
// Process image messages.
}//...Process more messages.
}
```

Receiving image messages

After receiving a message, use `getElem` to obtain all `Elem` nodes from `TIMMessage`. Nodes of the `TIMElemType.Image` type are image message nodes. To obtain all image specifications, including the original image, large image, and thumbnail, use `getImageList` of `TIMImageElem`. Each specification is stored in a `TIMImage` object.

```
/**
* Obtain the list of images contained in `Elem`. It can be called when the IM SDK fetches `Elem`.
* @return elem List of images contained in `elem`
*/
public ArrayList<TIMImage> getImageList()
```

`TIMImage`:

After receiving a message, use `imageList` to obtain all image specifications, which are `TIMImage` objects. After obtaining `TIMImage`, reserve a place based on the image size and use `getImage` to download images of different specifications for display.

> **Note:**
> **Developers need to cache the downloaded data. The IM SDK downloads data from the server each time it calls `getImage`. We recommend that you use the `uuid` of an image as the `key` to store images.**

Image specifications: each image has three specifications, including Original (original image), Large (large image), and Thumb (thumbnail).

- **Original image: the original image sent by a user, whose dimensions and size remain unchanged.**
- **Large image: an image obtained after the original image is proportionally compressed. The height or width of the compressed image, whichever is smaller, is equal to 720 pixels.**
- **Thumbnail: an image obtained after the original image is proportionally compressed. The height or width of the compressed image, whichever is smaller, is equal to 198 pixels.**

- **If the size of the original image is less than 198 pixels, the original size is retained for the three specifications, and no compression is needed.**
- **If the size of the original image falls between 198 and 720 pixels, the large image is the same as the original image, and no compression is needed.**
- **When an image is displayed on a mobile phone, we recommend that the thumbnail be displayed first. When a user taps the thumbnail, the large image is downloaded. When the user taps the large image, the original image is downloaded. Alternatively, developers can choose to skip the large image so that the original image is downloaded when the user taps the thumbnail.**
- **When an image is displayed on a tablet or PC, we recommend that the large image be displayed directly and the original image be downloaded when a user taps or clicks the large image due to the high resolution and availability of a Wi-Fi or wired network.**

Example:

```
// Traverse the element list of a message.
for(int i = 0; i < msg.getElementCount(); ++i) {
TIMElem elem = msg.getElement(i);
if (elem.getType() == TIMElemType.Image) {
// Image element
TIMImageElem e = (TIMImageElem) elem;
for(TIMImage image : e.getImageList()) {
// Obtain the image type, size, width, and height.
Log.d(tag, "image type: " + image.getType() +
" image size " + image.getSize() +
" image height " + image.getHeight() +
" image width " + image.getWidth());
image.getImage(path, new TIMCallBack() {
@Override
public void onError(int code, String desc) {// Failed to obtain the image.
```

```
// 'code' (error code) and 'desc' (error description) can be used to locate the cause of the requ
est failure.
// For more information about the meaning of error codes, see the Error Code table.
Log.d(tag, "getImage failed. code: " + code + " errmsg: " + desc);
}
@Override
public void onSuccess() {// Request succeeded, and the parameter is the image data.
//doSomething
Log.d(tag, "getImage success.");
}
});
}
}
}
```

Receiving audio messages

After receiving a message, use `getElem` to obtain all `Elem` nodes from `TIMMessage`. Nodes of the `TIMElemType.Sound` type are audio message nodes. When the message is received, reserve a place based on the audio length and use `getSoundToFile` to download audio resources. The `getSoundToFile` API downloads the data from the server. To cache or store the data, use `uuid` as the `key` to store the audio file externally. The IM SDK does not store resource files.

Prototype:

```
/**
* Download and save an audio file to the specified path.
*
* @param path Specified storage path
* @param progressCb Download progress callback
* @param cb Callback
*/
public void getSoundToFile(@NonNull final String path, final TIMValueCallBack<ProgressInfo> progr
essCb, @NonNull final TIMCallBack cb)
```

Audio message read status: you can use custom message fields to determine whether an audio message has been played. For example, the value 0 of `customInt` indicates that the audio message has not been played, and the value 1 indicates that the audio message has been played. When a user taps Play, `customInt` is set to 1. The following prototype shows how to set `customInt`, and the default value is 0.

Prototype:

```
public void setCustomInt(int value)
```

Receiving file messages

After receiving a message, use `getElem` to obtain all `Elem` nodes from `TIMMessage`. Nodes of the `TIMFileElem` type are file message nodes.

The `TIMFileElem` member methods are as follows:

```
// Download and save a file to the specified path.
void getToFile(String path, TIMCallBack callback)
// Obtain the file name.
java.lang.String getFileName()
// Obtain the file size.
long getFileSize()
// Obtain the UUID.
java.lang.String getUuid()
// Set the file name.
void setFileName(java.lang.String fileName)
```

You can choose to display only the file size and name of a received file message and use `getToFile` to download the file from the server each time. To cache or store the data, use `uuid` as the `key` to store the file externally. The IM SDK does not store resource files.

Prototype:

```
/**
* Download and save a file to the specified path.
* @param path Specified storage path
* @param callback Callback
*/
public void getToFile(@NonNull final String path, @NonNull TIMCallBack callback)
```

Receiving short video messages

After receiving a message, use `getElem` to obtain all `Elem` nodes from `TIMMessage`. Nodes of the `TIMVideoElem` type are short video message nodes. Use the `TIMVideo` and `TIMSnapshot` objects to obtain the video and snapshot content. After receiving `TIMVideoElem`, download the video file and snapshot file through the APIs defined in the `video` and `snapshot` properties. To cache or store the data, use `uuid` as the `key` to store the files externally. The IM SDK does not store resource files.

The `TIMVideo` member methods are as follows:

```
/**
* Obtain the video.
```

```
*
* @param path Video storage path
* @param cb Callback
* @deprecated
*/
getVideo(@NonNull final String path, @NonNull final TIMCallBack cb);
/**
* Obtain the video.
*
* @param path Video storage path
* @param progressCb Download progress callback
* @param cb Callback
*/
void getVideo(@NonNull final String path, final TIMValueCallBack<ProgressInfo> progressCb, @NonNu
ll final TIMCallBack cb)
/**
* Obtain the video size.
*
* @return Video size
*/
long getSize();
/**
* Obtain the UUID of the video file.
*
* @return UUID, which can be used as the unique key for caching
*/
String getUuid();
/**
* Obtain the video length.
*
* @return Video length
*/
long getDuaration();
/**
* Obtain the video file type.
*
* @return Video file type
*/
String getType();
```

The `TIMSnapshot` member methods are as follows:

```
/**
* Obtain the snapshot.
*
* @param path Snapshot storage path
* @param progressCb Download progress callback
```

```
     * @param cb Callback
     */
    void getImage(final String path, final TIMValueCallBack<ProgressInfo> progressCb, final TIMCallBa
    ck cb);
    /**
     * Obtain the snapshot.
     *
     * @param path Snapshot storage path
     * @param cb Callback
     * @deprecated
     */
    void getImage(final String path, final TIMCallBack cb);
    /**
     * Obtain the snapshot width.
     *
     * @return Snapshot width
     */
    long getWidth();
    /**
     * Obtain the snapshot height.
     *
     * @return Snapshot height
     */
    long getHeight();
    /**
     * Obtain the snapshot size.
     *
     * @return Snapshot size
     */
    long getSize();
    /**
     * Obtain the snapshot file type.
     *
     * @return Snapshot file type
     */
    String getType();
    /**
     * Obtain the UUID of the snapshot file.
     *
     * @return UUID, which can be used as the unique key for caching
     */
    String getUuid();
```

Parsing process for short video messages:

The new message receipt callback is used as an example. First, determine whether the message has `TIMVideoElem` based on the element type. If yes, the message is a short video message. In this case, run the following code to parse it.

```
TIMMessage timMsg = msg.getTIMMessage();
final TIMVideoElem videoEle = (TIMVideoElem) timMsg.getElement(0);
final TIMVideo video = videoEle.getVideoInfo();
final TIMSnapshot shotInfo = videoEle.getSnapshotInfo();
final String path = ” /xxx/ “ + videoEle.getSnapshotInfo().getUuid(); // Storage path of the rece
ived snapshot
final String videoPath = ” /xxx/ “ + video.getUuid(); // Storage path of the received video
videoEle.getSnapshotInfo().getImage(path, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, ”Failed to download the snapshot, code = ” + code + ”, errorinfo = ” + desc);
}
@Override
public void onSuccess() {
Log.d(tag, ”Succeeded in downloading the snapshot”);
}
});
video.getVideo(videoPath, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, ”Failed to download the short video, code = ” + code + ”, errorinfo = ” + desc);
}
@Override
public void onSuccess() {
Log.d(tag, ”Succeeded in downloading the short video”);
}
});
```

Message Properties

You can obtain message properties using the `TIMMessage` member methods.

Checking whether a message has been read

Use `isRead` of `TIMMessage` to check whether a message has been read, which is determined by the Unread Count on the app side. The prototype for checking whether a message has been read is as follows:

Prototype:

```
public boolean isRead()
```

Message status

To obtain the status of the current message, such as sending, sent successfully, failed to send, or deleted, use the `status` method of `TIMMessage`. For deleted messages, use the UI to determine the status and hide the messages accordingly.

```
// Sending
TIMMessageStatus.Sending
// Sent successfully
TIMMessageStatus.SendSucc
// Failed to send
TIMMessageStatus.SendFail
// Deleted
TIMMessageStatus.HasDeleted
// Recalled
TIMMessageStatus.HasRevoked
```

Checking whether a message was sent by oneself

To determine whether a message was sent by you yourself, use the `isSelf` method of `TIMMessage`. This method is available when the message is displayed on the interface. The following shows the prototype for determining whether a message was sent by yourself.

Prototype:

```
public boolean isSelf()
```

Message sender and related profile

To obtain the sender's ID, use the `getSender` method of `TIMMessage`.
For one-to-one chat messages, use the `getConversation` method of `TIMMessage` to obtain the corresponding conversation and use `getPeer` to obtain the recipient and the recipient's profile.
For group chat messages, use `getSenderProfile` and `getSenderGroupMemberProfile` to obtain the sender's profile and the profile of the group to which the sender belongs. To get custom fields, set the fields to be pulled before logging in to the IM SDK.

> **Note :**
> **This field obtains the user profile and writes it to the message body when the message is sent. If the user profile is updated, this field will not change unless new messages are generated.**
> **You can obtain profiles only from received group messages.**

```
/**
* Obtain the message sender.
* @return Message sender
*/
public String getSender()
/**
* Obtain the sender profile.
*
* In version 4.4.716, this information is returned through a callback.
*
* @param callBack Callback
*/
public void getSenderProfile( TIMValueCallBack < TIMUserProfile > callBack )
/**
* Obtain the sender's profile in the group, which is only available for received group messages
(may be empty if the sender is yourself).
*
* @return Sender's profile in the group. "null" indicates that no profile was obtained or the mes
sage is not a group message. Currently, only the user, nameCard, role, and customInfo fields can
be obtained. To obtain other fields, use `getGroupMembers` of `TIMGroupManager`.
*/
public TIMGroupMemberInfo getSenderGroupMemberProfile()
```

Message time

To obtain the message time, use the `timestamp` method of `TIMMessage`. This time is the server time, not the local time. When you create a message, this time is calibrated based on the server time and will be changed to the accurate server time after the message is successfully sent.

```
// Timestamp generated for the message by the server
public long timestamp()
```

Message ID

There are two types of message IDs. One is `msgId`, which is created when a message is generated. If `msgId` is used, messages may conflict with messages generated by other users, and therefore a time dimension needs to be added. Messages generated within 10 minutes can be distinguished by `msgId`. The other is `uniqueId`, which is generated after a message is sent successfully and is globally unique. Both types of message IDs must be checked in the same conversation.

```
// Obtain the message ID.
public String getMsgId()
// Obtain the uniqueId of the message.
public long getMsgUniqueId()
```

Custom message field

Developers can add custom fields to messages, such as the custom integer and custom binary data fields, and can customize different effects based on these two fields. For example, custom fields can be used to determine whether an audio message has been played. Note that these custom fields are only stored locally and not synchronized to the server. You will not obtain them after switching to another client.

```
// Set the custom integer, which is 0 by default.
public void setCustomInt(int value)
// Obtain the value of the custom integer.
public int getCustomInt()
// Set the custom data content, which is "". by default.
public void setCustomStr(String str)
// Obtain the value of the custom data content.
public String getCustomStr()
```

Message priority

Livestreaming scenarios involve the like and red packet features. Like messages have a lower priority than red packet messages. You can use `TIMCustomElem` to define the message content, and use different APIs to define the message priority when sending a message.

> **Note :**
> **Message priorities apply only to group messages.**

```
// Set the message priority.
public void setPriority(TIMMessagePriority priority)
// Obtain the message priority.
public TIMMessagePriority getPriority()
```

Read receipt

The IM SDK provides the read receipt feature for C2C messages. You can enable this feature using `enableReadReceipt` of `TIMUserConfig`. After this feature is enabled, the IM SDK will send read receipts to the message sender when sending message read reports.

To register a read receipt listener, use `setMessageReceiptListener` of `TIMUserConfig`. To check whether the current message has been read by the recipient, use `isPeerReaded` of `TIMMessage`.

Prototype:

```
/**
* Enable the read receipt feature. Then, read receipts will be sent to the message sender when me
ssage read reports are reported. This feature applies only to C2C conversations.
*/
public void enableReadReceipt()
/**
* Set the read receipt listener.
* @param receiptListener Read receipt listener
*/
public void setMessageReceiptListener(TIMMessageReceiptListener receiptListener)
/**
* Check whether the recipient has read the message. (This feature applies only to C2C messages.)
* @return true: read by the recipient. false: not read by the recipient.
*/
public boolean isPeerReaded()
```

Message sequence number

To obtain the sequence number of the current message, use `getSeq` of `TIMMessage`.

```
/**
* Obtain the sequence number of the current message.
* @return Sequence number of the current message
*/
public long getSeq()
```

Message random number

To obtain the random number of the current message, use `getRand` of `TIMMessage`.

```
/**
* Obtain the random number of the current message.
* @return Random number of the current message
*/
public long getRand()
```

Message query parameters

In the IM SDK, a message is identified by a `{seq, rand, timestamp, isSelf}` quad, which represents the query parameters of the message. To obtain query parameters of the current message, use `getMessageLocator` of `TIMMessage`.

```
/**
* Obtain the query parameters of the current message.
* @return Query parameters of the current message
*/
public TIMMessageLocator getMessageLocator()
```

Conversation Operations

Obtaining all conversations

Use `getConversationList` of `TIMManager` to obtain the current number of conversations and all local conversations.

> **Note :**
> **The SDK continuously updates the conversation list internally. The update will be sent back to the caller using `TIMRefreshListener.onRefresh`. Call `getConversationList` after `onRefresh` to update the conversation list.**

Prototype:

```
/**
* Obtain all conversations.
* @return Conversation list
*/
public List<TIMConversation> getConversationList()
```

Example:

```
List<TIMConversation> list = TIMManager.getInstance().getConversationList();
```

Recent contact roaming

By default, after the user logs in to the IM SDK, the IM SDK enables recent contact roaming and obtains the last message of each conversation.

Obtaining local messages in a conversation

The IM SDK stores messages locally. To obtain these messages, use `getLocalMessage` of `TIMConversation`. This is an asynchronous method, and a callback needs to be set to obtain message data. For a C2C conversation, offline messages will be obtained automatically after login. For a group conversation, when recent contacts roaming is enabled, only the last message is obtained after login, and roaming messages can be obtained using `getMessage`.

> **Note:**
> **For resource messages such as image and audio messages, the message body only contains descriptive information, and additional APIs are required to download data. For more information, see Parsing Messages. The actual data downloaded is not cached, and must be cached by the caller.**

Prototype:

```
/**
* Obtain only the local chat history.
* @param count Number of messages as of the last message
* @param lastMsg Last message that was obtained. "null" indicates the latest message.
* @param callback Callback that returns the list of obtained messages
*/
public void getLocalMessage(int count, TIMMessage lastMsg, @NonNull TIMValueCallBack<List<TIMMess
age>> callback)
```

Example:

```
// Obtain a conversation extension instance.
TIMConversation con = TIMManager.getInstance().getConversation(TIMConversationType.Group, groupI
d);
// Obtain all messages of this conversation.
con.getLocalMessage(10, // Obtain the last 10 messages in this conversation.
null, // The start message from which messages are obtained is not specified. In this case, the o
peration starts from the latest message.
new TIMValueCallBack<List<TIMMessage>>() {// Callback API
@Override
public void onError(int code, String desc) {// Failed to obtain the messages.
// "code" (error code) and "desc" (error description) can be used to locate the cause of the requ
```

```
est failure.
// For more information about the meaning of error codes, see the Error Code table.
Log.d(tag, "get message failed. code: " + code + " errmsg: " + desc);
}
@Override
public void onSuccess(List<TIMMessage> msgs) {// Succeeded in obtaining messages.
// Traverse the obtained messages.
for(TIMMessage msg : msgs) {
lastMsg = msg;
// The message timestamp can be obtained through `timestamp()`. `isSelf()` indicates whether the
message was sent by yourself.
Log.e(tag, "get msg: " + msg.timestamp() + " self: " + msg.isSelf() + " seq: " + msg.getSeq());
}
}
});
```

Obtaining roaming messages in a conversation

For group conversations, a user can obtain roaming messages after login. For C2C conversations, the user can obtain roaming messages after the roaming service is enabled. To obtain roaming messages, use `getMessage` of `TIMConversation`. If local messages are continuous, they are obtained directly, instead of over the network. If local messages are not continuous, missing messages need to be obtained over the network.

> **Note :**
> **For resource messages such as image and audio messages, the message body only contains descriptive information, and additional APIs are required to download data, which can participate in message parsing. The actual data downloaded is not cached, and must be cached by the caller.**

Prototype:

```
/**
* Obtain the chat history.
* @param count Number of messages as of the last message
* @param lastMsg Last message that was obtained
* @param callback Callback that returns the list of obtained messages
*/
public void getMessage(int count, TIMMessage lastMsg, @NonNull TIMValueCallBack< List<TIMMessage>
> callback)
```

Example:

```
// Obtain a conversation extension instance.
TIMConversation con = TIMManager.getInstance().getConversation(TIMConversationType.Group, groupI
d);
// Obtain all messages of this conversation.
con.getMessage(10, // Obtain the last 10 message in this conversation.
null, // The start message from which messages are obtained is not specified. In this case, the o
peration starts from the latest message.
new TIMValueCallBack<List<TIMMessage>>() {// Callback API
@Override
public void onError(int code, String desc) {// Failed to obtain the messages.
// "code" (error code) and "desc" (error description) can be used to locate the cause of the requ
est failure.
// For more information about the meaning of error codes, see the Error Code table.
Log.d(tag, "get message failed. code: " + code + " errmsg: " + desc);
}
@Override
public void onSuccess(List<TIMMessage> msgs) {// Succeeded in obtaining messages.
// Traverse the obtained messages.
for(TIMMessage msg : msgs) {
lastMsg = msg;
// The message timestamp can be obtained through `timestamp()`. `isSelf()` indicates whether the
message was sent by yourself.
Log.e(tag, "get msg: " + msg.timestamp() + " self: " + msg.isSelf() + " seq: " + msg.msg.seq());
}
}
});
```

Deleting conversations

While deleting a conversation, the IM SDK also deletes the local and roaming messages of this conversation, and the deleted conversation and messages cannot be recovered.

Prototype:

```
/**
* Delete a conversation stored locally and on the server, as well as all messages of this convers
ation that are stored locally and on the server.
*
* @param type Conversation type
* @param peer Peer in the conversation. For a C2C conversation, use the peer's account identifie
r. For a group conversation, use the group ID.
* @return true: deleted successfully. false: failed to delete.
*/
public boolean deleteConversation(TIMConversationType type, String peer)
```

The following example shows how to delete the C2C conversation with user1. Example:

```
TIMManager.getInstance().deleteConversation(TIMConversationType.C2C, "user1");
```

Synchronously obtaining the last message of a conversation

The UI displays the last messages from users in the recent contact list. The IM SDK provides the `getLastMsg` API in `TIMConverstion` to synchronously obtaining the last message of a conversation, allowing the user to obtain and display the last message. This feature requires a network connection. If recent contacts are disabled, the last message of a conversation cannot be obtained after login and before new messages are received. Messages obtained using this API include deleted messages, which need to be blocked by the app. To obtain multiple recent messages, use `getMessage` .

Prototype:

```
/**
* Obtain the last message from the cache.
* @return Last message. "null" is returned if the conversation is invalid.
*/
public TIMMessage getLastMsg()
/**
* Obtain the chat history.
* @param count Number of messages as of the last message
* @param lastMsg Last message that was obtained
* @param callback Callback that returns the list of obtained messages
*/
public void getMessage(int count, TIMMessage lastMsg, @NonNull TIMValueCallBack< List<TIMMessage>
> callback)
```

Setting conversation drafts

The IM SDK provides the conversation draft feature. Developers can call APIs of `TIMConversation` to perform draft-related operations.

> **Note :**
>
> - **Drafts are only valid locally. Users cannot see their drafts after switching devices or clearing the data.**
> - **Drafts are stored in a local database and can be obtained after users log in again.**

Prototype:

```
/**
* Set a draft.
* @param Draft content. If it is null, the draft is canceled.
*/
public void setDraft(TIMMessageDraft draft)
/**
* Obtain the draft.
* @return Draft content
*/
public TIMMessageDraft getDraft()
/**
* Check whether a draft is created for the current conversation.
* @return true: yes. false: no.
*/
public boolean hasDraft()
```

`TIMMessageDraft` is described as follows:

```
/**
* Obtain the list of message elements in a draft.
* @return List of message elements
*/
public List<TIMElem> getElems()
/**
* Set message elements in a draft.
* @param elem Message elements to be added to the draft
*/
public void addElem(TIMElem elem)
/**
* Obtain user-defined data in a draft.
* @return User-defined data
*/
public byte[] getUserDefinedData()
/**
* Set user-defined data in a draft.
* @param userDefinedData User-defined data
*/
public void setUserDefinedData(byte[] userDefinedData)
/**
* Obtain the edit time of a draft.
* @return Edit time of the draft
*/
public long getTimestamp()
```

Deleting messages of a conversation

The IM SDK allows you to delete the local and roaming messages of a conversation, and deleted messages cannot be recovered.

Prototype:

```
/**
* Delete the local and roaming messages of the current conversation.
*
* While deleting the local historical messages, this API also deletes the roaming messages stored on the server. After the IM SDK is uninstalled and then re-installed, these roaming messages cannot be pulled again. Note that:
* 1. Up to 30 messages can be deleted at a time.
* 2. The API can be called only one time per second.
* 3. If this account has been used to pull roaming messages on other devices, and the API is called to delete these messages, these messages still exist on those devices. In short, message deletion cannot be synchronized across multiple terminals.
*/
public void deleteMessages(List<TIMMessage> messages, TIMCallBack callback)
```

Searching for local messages

The IM SDK allows users to search for messages based on given parameters. Currently, only exact search is available, and fuzzy search is not supported. Developers can use the `findMessages` method of `TIMConversation` to search for messages.

```
/**
* Search for messages based on given parameters.
* @param locators Message search parameters
* @param cb Callback, which returns matching messages
*/
public void findMessages(@NonNull List<TIMMessageLocator> locators, TIMValueCallBack<List<TIMMessage>> cb)
```

`TIMMessageLocator` can be obtained using the `getMessageLocator` method of `TIMMessage`.

Prototype:

```
/**
* Obtain the locator of the current message.
* @return Locator of the current message
*/
public TIMMessageLocator getMessageLocator()
```

Recalling messages

Starting from version 3.1.0, the IM SDK provides an API to recall messages. To recall sent messages, call the `revokeMessage` API of `TIMConversation`.

> **Note :**
>
> - **The API applies only to C2C and group conversations, and not to onlineMessages, AVChatRooms, or BChatRooms.**
> - **By default, only messages that were sent within the last 2 minutes can be recalled.**

Prototype:

```
/**
* Recall a message. (This API applies only to C2C and group conversations, and not to onlineMessages, AVChatRooms, or BChatRooms.)
* @param msg Message to be recalled
* @param cb Callback
* @since 3.1.0
*/
public void revokeMessage(@NonNull TIMMessage msg, @NonNull TIMCallBack cb)
```

After a message is recalled, other members in the group or the peer in the C2C conversation will receive a message recall notification. In addition, the message recall notification listener `TIMMessageRevokeListener` notifies the upper-layer app. You can configure the message recall notification listener before login using `setMessageRevokedListener` of `TIMUserConfig`. For more information, see User Configuration.

Prototype:

```
/**
* Message recall notification listener
* @since 3.1.0
*/
public interface TIMMessageRevokedListener extends IMBaseListener {
/**
* Message recall notification
* @param locator Locator of the recalled message
*/
void onMessageRevoked(TIMMessageLocator locator);
}
```

After receiving a message recall notification, use the `checkEquals` method of `TIMMessage` to check whether the current message has been recalled by the sender and then refresh the UI when necessary.

Prototype:

```
/**
* Compare the current message with the message specified by the given locator to check whether th
ey are the same message.
* @param locator Message locator
* @return true: yes. false: no
* @since 3.1.0
*/
public boolean checkEquals(@NonNull TIMMessageLocator locator)
```

System Messages

In addition to C2C conversations and group conversations, system message is another conversation type (TIMConversationType). System messages are notifications that are sent by the system backend for various events. These messages cannot be sent by users. Currently, there are two types of system messages: relationship chain system messages and group system messages.

- **The system sends a relationship chain change message when a user adds you as a friend or deletes you from his or her friend list. The developer can then update the friend list. For more information, see System Notifications for Relationship Chain Changes.**
- **When the group profile is modified, for example, due to a change to the group name or group members, the system sends a group event message in the group. The developer can choose whether to display the message and, at the same time, refresh the group profile or group members. For more information, see Group Event Messages.**
- **When the group admin removes a member from the group or invites a user to join the group, the system sends a group system message to the user. For more information, see Group System Messages.**

Setting Backend Message Notification Bar Reminders

When the IM SDK is running in the background, it can continue to receive message notifications. If the program is running in the background, you can present new

messages to users in the form of system notification bar reminders. New messages can be displayed in the notification bar on the top of the screen, in the notification center, or on the lock screen. See the following example for the implementation method:

Example:

```
NotificationManager mNotificationManager = (NotificationManager) context.getSystemService(context
.NOTIFICATION_SERVICE);
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(context);
Intent notificationIntent = new Intent(context, MainActivity.class);
notificationIntent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP| Intent.FLAG_ACTIVITY_SINGLE_TOP);
PendingIntent intent = PendingIntent.getActivity(context, 0, notificationIntent, 0);
mBuilder.setContentTitle(senderStr)// Set the notification bar title.
.setContentText(contentStr)
.setContentIntent(intent) // Set the notification bar click intent.
.setNumber(++pushNum) // Set the number of notifications in a collection.
.setTicker(senderStr+":"+contentStr) // The notification appears in the notification bar for the
first time with a rising animation effect.
.setWhen(System.currentTimeMillis())// Generation time of the notification, which is displayed in
the notification information. It is normally the time obtained by the system.
.setDefaults(Notification.DEFAULT_ALL)// The simplest and most consistent way to add sound, flas
h, and vibration effects to notifications is to use the current default settings. The `defaults`
properties can be used in combination.
.setSmallIcon(R.drawable.ic_launcher);// Set the small icon for notifications.
Notification notify = mBuilder.build();
notify.flags |= Notification.FLAG_AUTO_CANCEL;
mNotificationManager.notify(pushId, notify);
```

Sending and Receiving Messages (iOS)

Last updated : 2021-10-09 11:17:45

Sending Messages

Sending common messages

Obtaining a conversation

Conversations are classified into conversations with an individual user and conversations with a group. To send or receive messages in a conversation, you need to first obtain the conversation by specifying the conversation type (C2C conversation or group conversation) and the peer's identifier (the peer's account or group ID). To obtain a conversation, call `getConversation` .

> **Note :**
> **If the conversation is not stored locally, calls to `TIMConversation` APIs will fail. We recommend that you operate on the `TIMConversation` object after receiving the `TIMUserConfig` > `TIMRefreshListener` callback.**

Prototype:

```
@interface TIMManager : NSObject
/**
* Obtain a conversation.
*
* @param type Conversation type. TIM_C2C indicates a one-to-one conversation, and TIM_GROUP indic
ates a group conversation.
* @param conversationId For a C2C conversation, use the peer's account identifier. For a group co
nversation, use the group ID.
*
* @return Conversation object
*/
- (TIMConversation*)getConversation:(TIMConversationType)type receiver:(NSString*)conversationId;
@end
```

Parameter description:

Parameter	Description

Parameter	Description
type	**The conversation type. For a one-to-one conversation, enter TIM_C2C. For a group conversation, enter TIM_GROUP.**
conversationId	**The conversation identifier. For a one-to-one conversation, the receiver is the peer's account identifier. For a group conversation, the receiver is the group ID.**

Obtain the one-to-one conversation in which the other participant's `identifier` is "iOS-001":

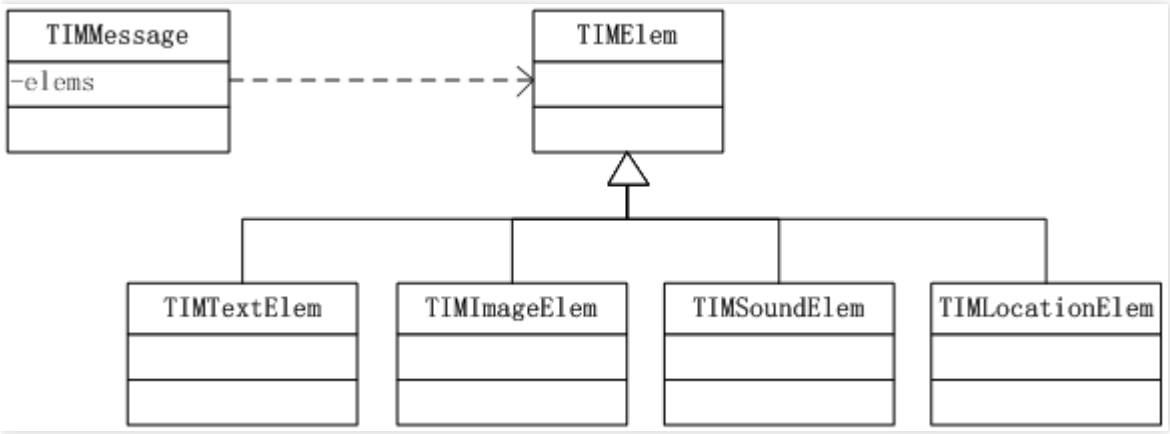
```
TIMConversation * c2c_conversation = [[TIMManager sharedInstance] getConversation:TIM_C2C receiver:@"iOS-001"];
```

Obtain the group chat with a group whose group ID is "TGID1JYSZEAEQ":

```
TIMConversation * grp_conversation = [[TIMManager sharedInstance] getConversation:TIM_GROUP receiver:@"TGID1JYSZEAEQ"];
```

Sending messages

After `TIMConversation` is obtained using `TIMManager`, you can send messages and obtain cached messages for the conversation. For more information about messages in the IM SDK, see Introduction to IM SDK Objects. In the IM SDK, a message is a `TIMMessage` object. A `TIMMessage` can contain multiple `TIMElem` units, which can be text or images. This means a message can contain multiple text segments and images. Use the `sendMessage` method of `TIMConversation` to send messages using blocks or the protocol callback.



Prototype:

```
@interface TIMConversation : NSObject
-(int) sendMessage: (TIMMessage*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
msg	**The message.**
succ	**The success callback.**
fail	**The failure callback.**

Sending text messages

A text message is defined by `TIMTextElem`.

```
@interface TIMTextElem : TIMElem {
NSString * text;
}
```

Example:

> **Note :**
>
> - `text` **passes the text message to send.**
> - **In the failure callback,** `code` **indicates the error code, and** `err` **indicates the error description. For more information, see Error Codes.**

```
TIMTextElem * text_elem = [[TIMTextElem alloc] init];
[text_elem setText:@"this is a text message"];
TIMMessage * msg = [[TIMMessage alloc] init];
[msg addElem:text_elem];
[conversation sendMessage:msg succ:^(){
NSLog(@"SendMsg Succ");
}fail:^(int code, NSString * err) {
NSLog(@"SendMsg Failed:%d->%@", code, err);
}];
```

Sending image messages

An image message is defined by `TIMImageElem`, which is a subclass of `TIMElem`. These messages can contain images. To send an image, add `TIMImageElem` to `TIMMessage` and send the image along with the message. When sending an image, you only need to set path, the image path. After the message is sent successfully, use `imageList` to obtain all image types. In addition, use `TIMUserConfig` > `TIMUploadProgressListener` to listen to the current upload progress.

`TIMImageElem` prototype:

```
/**
* Store the path of the image to be sent, which must be a local path. See the following example.
*/
@interface TIMImageElem : TIMElem
/**
* Path of the image to be sent
*/
@property(nonatomic,retain) NSString * path;
/**
* This parameter saves all specifications of the image when it is received. Therefore, you can sk
ip this parameter when sending a message.
*/
@property(nonatomic,retain) NSArray * imageList;
/**
* Upload task ID, which can be used to query the upload progress. This parameter has been depreca
ted. Therefore, use `TIMUploadProgressListener` instead to listen to the upload progress.
*/
@property(nonatomic,assign) uint32_t taskId DEPRECATED_ATTRIBUTE;
/**
* Image compression level. For more information, see `TIM_IMAGE_COMPRESS_TYPE`, which applies onl
y to the jpg format.
*/
@property(nonatomic,assign) TIM_IMAGE_COMPRESS_TYPE level;
/**
* Image format. For more information, see `TIM_IMAGE_FORMAT`.
*/
@property(nonatomic,assign) TIM_IMAGE_FORMAT format;
@end
```

Parameter description:

Parameter	Description
path	**Stores the path of the image to be sent, which must be a local path. For more information, see the example of sending an image.**

Parameter	Description
imageList	**Saves all specifications of the image when it is received. Therefore, you can skip this parameter when sending a message. For more information, see the section about receiving image messages.**
taskId	**Can be used to query the upload progress when sending images. This parameter has been deprecated. Therefore, use `TIMUploadProgressListener` instead to listen to the upload progress.**
level	**Images need to be compressed before being sent, and level represents the compression level. For more information, see `TIM_IMAGE_COMPRESS_TYPE` .**
format	**Image format. For more information, see `TIM_IMAGE_FORMAT.`**

The following example sends an image with the absolute path of `/xxx/imgPath.jpg` .
Example:

```
/**
* Obtain the conversation with user iOS-001.
*/
TIMConversation * c2c_conversation = [[TIMManager sharedInstance] getConversation:TIM_C2C receive
r:@"iOS-001"];
/**
* Construct a message.
*/
TIMMessage * msg = [[TIMMessage alloc] init];
/**
* Construct the image content.
*/
TIMImageElem * image_elem = [[TIMImageElem alloc] init];
image_elem.path = @"/xxx/imgPath.jpg";
/**
* Add the image content to the message container.
*/
[msg addElem:image_elem];
/**
* Send the message.
*/
[conversation sendMessage:msg succ:^(){ // Successful
NSLog(@"SendMsg Succ");
}fail:^(int code, NSString * err) { // Failed
NSLog(@"SendMsg Failed:%d->%@", code, err);
}];
```

Sending emoji messages

An emoji message is defined by `TIMFaceElem`. The IM SDK does not provide an emoji package. Developers can use `index` to store the indexes of the emojis in their emoji packages. Alternatively, they can directly use `data` to store emoji binary data and the string `key`. Using both these methods, users can customize emojis. The SDK only passes them through.

```
@interface TIMFaceElem : TIMElem
/**
* Emoji index, which can be customized by users
*/
@property(nonatomic, assign) int index;
/**
* Additional data, which can be customized by users
*/
@property(nonatomic,retain) NSData * data;
@end
```

Parameter description:

> **Note :**
> **You only need to pass either `index` or `data`, and the IM SDK simply passes them through.**

Parameter	Description
index	**Emoji index, which is customized by the developer**
data	**Emoji binary data, which is customized by the developer**

The following example sends an emoji with an index of 10. The developer must have an emoji package that contains the emoji with the index 10 at both sides. The emoji can also be identified by binary data using the `data` parameter. Example:

```
TIMFaceElem * face_elem = [[TIMFaceElem alloc] init];
[face_elem setIndex:10];
TIMMessage * msg = [[TIMMessage alloc] init];
[msg addElem:face_elem];
[conversation sendMessage:msg succ:^(){
NSLog(@"SendMsg Succ");
}fail:^(int code, NSString * err) {
```

```
NSLog(@"SendMsg Failed:%d->%@", code, err);
}];
```

Sending audio messages

An audio message is defined by `TIMSoundElem`, where `data` stores audio data. For audio data, you need to provide the audio length in seconds.

Note :

- **A message can contain only one audio `Elem`. If an attempt is made to add multiple audio `Elem` objects, the `AddElem` function returns error 1 and the audio `Elem` objects cannot be added.**
- **Audio and file `Elem` objects are not always received in the order that they are added. We recommend that you determine and display `Elem` objects one by one. Moreover, audio and file `Elem` objects may not be sorted in the order that they are sent.**

```
/**
* Audio message Elem
*/
@interface TIMSoundElem : TIMElem
/**
* Upload task ID, which can be used to query the upload progress. This parameter has been depreca
ted. Therefore, use `TIMUploadProgressListener` instead to listen to the upload progress.
*/
@property(nonatomic,assign) uint32_t taskId DEPRECATED_ATTRIBUTE;
/**
* The path of the audio file to be uploaded. Use `getSound` to obtain the data when receiving the
message.
*/
@property(nonatomic,strong) NSString * path;
/**
* Store audio data.
*/
@property(nonatomic,retain) NSData * data;
/**
* Internal ID of the audio message
*/
@property(nonatomic,strong) NSString * uuid;
/**
* Audio data size
```

```
*/
@property(nonatomic,assign) int dataSize;
/**
* Audio length in seconds, which should be set when the message is sent
*/
@property(nonatomic,assign) int second;
/**
* Obtain the download URL of the audio file.
*
* @param urlCallBack Callback for obtaining the URL
*/
-(void)getUrl:(void (^)(NSString * url))urlCallBack;
/**
* Save audio data to a file in the specified path.
*
* The `getSound` API downloads audio data from the server. To cache or store the data, use `uuid`
as the `key` to store the audio data externally. The IM SDK does not store resource files.
*
* @param path Audio storage path
* @param succ Success callback
* @param fail Failure callback, which returns the error code and error description
*/
- (void)getSound:(NSString*)path succ:(TIMSucc)succ fail:(TIMFail)fail;
/**
* Save audio data to a file in the specified path (with progress callback).
*
* The `getSound` API downloads audio data from the server. To cache or store the data, use `uuid`
as the `key` to store the audio data externally. The IM SDK does not store resource files.
*
* @param path Audio storage path
* @param progress Audio download progress
* @param succ Success callback
* @param fail Failure callback, which returns the error code and error description
*/
- (void)getSound:(NSString*)path progress:(TIMProgress)progress succ:(TIMSucc)succ fail:(TIMFail)
fail;
@end
```

Parameter description:

Parameter	Description
path	**The file path of the audio to be uploaded.**
uuid	**The unique identifier generated after the audio is uploaded. The user can save the file based on this identifier. The IM SDK does not save resource data internally.**

Parameter	Description
dataSize	**The audio data size.**
second	**The audio data length.**

Example:

```
TIMSoundElem * sound_elem = [[TIMSoundElem alloc] init];
[sound_elem setPath:@"./xxx.mp3"];
[sound_elem setSecond:10];
TIMMessage * msg = [[TIMMessage alloc] init];
[msg addElem:sound_elem];
[conversation sendMessage:msg succ:^(){
NSLog(@"SendMsg Succ");
}fail:^(int code, NSString * err) {
NSLog(@"SendMsg Failed:%d->%@", code, err);
}];
```

Sending location messages

A location message is defined by `TIMLocationElem`, where `desc` stores the description information of the location, and `longitude` and `latitude` specify the longitude and latitude of the location, respectively.

```
@interface TIMLocationElem : TIMElem
/**
* Description of the location, which is set when the message is sent
*/
@property(nonatomic,retain) NSString * desc;
/**
* Latitude, which is set when the message is sent
*/
@property(nonatomic,assign) double latitude;
/**
* Longitude, which is set when the message is sent
*/
@property(nonatomic,assign) double longitude;
@end
```

Example:

```
NSString *desc= @"Tencent Building";
TIMLocationElem * location_elem = [[TIMLocationElem alloc] init];
```

```
[location_elem setDesc:desc];
[location_elem setLatitude:113.93];
[location_elem setLongitude:22.54];
TIMMessage * msg = [[TIMMessage alloc] init];
[msg addElem:location_elem];
[conversation sendMessage:msg succ:^(){
NSLog(@"SendMsg Succ");
}fail:^(int code, NSString * err) {
NSLog(@"SendMsg Failed:%d->%@", code, err);
}];
```

Sending file messages

A file message is defined by `TIMFileElem`. You can also view additional information such as the file name.

> **Note :**
> **Audio and file `Elem` objects are not always received in the order that they are added. We recommend that you determine and display `Elem` objects one by one.**

```
/**
* File message Elem
*/
@interface TIMFileElem : TIMElem
/**
* Upload task ID, which can be used to query the upload progress. This parameter has been deprecated. Therefore, use `TIMUploadProgressListener` instead to listen to the upload progress.
*/
@property(nonatomic,assign) uint32_t taskId DEPRECATED_ATTRIBUTE;
/**
* Path of the file to be uploaded. (If the path is specified, the file in the specified path will be uploaded first.)
*/
@property(nonatomic,strong) NSString * path;
/**
* Internal ID of the file
*/
@property(nonatomic,strong) NSString * uuid;
/**
* File size
*/
@property(nonatomic,assign) int fileSize;
/**
```

```
* Display name of the file, which is set when the message is sent
*/
@property(nonatomic,strong) NSString * filename;
/**
* Obtain the download URL of the file.
*
* @param urlCallBack Callback for obtaining the URL
*/
-(void)getUrl:(void (^)(NSString * url))urlCallBack;
/**
* Save file data to the file in the specified path.
*
* The `getFile` API downloads file data from the server. To cache or store the data, use the `uui
d` as the `key` to store the file externally. The IM SDK does not store resource files.
*
* @param path File storage path
* @param succ Success callback, which returns data
* @param fail Failure callback, which returns the error code and error description
*/
- (void)getFile:(NSString*)path succ:(TIMSucc)succ fail:(TIMFail)fail;
/**
* Save file data to a file in the specified path (with progress callback).
*
* The `getFile` API downloads file data from the server. To cache or store the data, use the `uui
d` as the `key` to store the file externally. The IM SDK does not store resource files.
*
* @param path File storage path
* @param progress File download progress
* @param succ Success callback, which returns data
* @param fail Failure callback, which returns the error code and error description
*/
- (void)getFile:(NSString*)path progress:(TIMProgress)progress succ:(TIMSucc)succ fail:(TIMFail)f
ail;
@end
```

Parameter description:

Parameter	Description
path	**The file path.**
data	**The binary data of the file to be sent. You only need to set either `path` or `data`. We recommend that you set `path`.**
filename	**The file name. The IM SDK does not verify whether the file name is correct. It only passes the file name through.**

Example:

```
TIMFileElem * file_elem = [[TIMFileElem alloc] init];
[file_elem setPath:./xxx/a.txt];
[file_elem setFilename:@"a.txt"];
TIMMessage * msg = [[TIMMessage alloc] init];
[msg addElem:file_elem];
[conversation sendMessage:msg succ:^(){
NSLog(@"SendMsg Succ");
}fail:^(int code, NSString * err) {
NSLog(@"SendMsg Failed:%d->%@", code, err);
}];
```

Sending custom messages

Developers can customize the message format and content when built-in message types cannot meet their special needs. The IM SDK only passes through custom messages. If iOS APNs push notifications are required, a push text description to display needs to be provided. A custom message is defined by `TIMCustomElem`, where `data` stores the binary data of the message and developers define the data format. A message can contain multiple custom `Elem` objects, which can be mixed with other `Elem` objects. In offline push scenarios, `desc` of each `Elem` can be stacked and delivered.

```
/**
* Custom message type
*/
@interface TIMCustomElem : TIMElem
/**
* Custom message binary data
*/
@property(nonatomic,strong) NSData * data;
/**
* Custom message description, which is used for display during offline push. This parameter has b
een deprecated. Therefore, you need to use `offlinePushInfo` of `TIMMessage` to configure the inf
ormation.
*/
@property(nonatomic,strong) NSString * desc DEPRECATED_ATTRIBUTE;
/**
* Extension field information in offline push. This parameter has been deprecated. Therefore, you
need to use `offlinePushInfo` of `TIMMessage` to configure the information.
*/
@property(nonatomic,strong) NSString * ext DEPRECATED_ATTRIBUTE;
/**
* Sound field information in offline push. This parameter has been deprecated. Therefore, you nee
d to use `offlinePushInfo` of `TIMMessage` to configure the information.
```

```
*/
@property(nonatomic,strong) NSString * sound DEPRECATED_ATTRIBUTE;
@end
```

Parameter description:

Parameter	Description
data	**The binary data of the custom message.**

The following example adds an XML message, the display of which is determined by the developer.

Example:

```
// Custom XML message
NSString * xml = @"testTitlethis is custom msgtest msg body";
// Convert the message to NSData.
NSData *data = [xml dataUsingEncoding:NSUTF8StringEncoding];
TIMCustomElem * custom_elem = [[TIMCustomElem alloc] init];
[custom_elem setData:data];
TIMMessage * msg = [[TIMMessage alloc] init];
[msg addElem:custom_elem];
TIMConversation *conversation = [[TIMManager sharedInstance] getConversation:TIM_C2C receiver:@"y
ahaha"];
[conversation sendMessage:msg succ:^(){
NSLog(@"SendMsg Succ");
}fail:^(int code, NSString * err) {
NSLog(@"SendMsg Failed:%d->%@", code, err);
}];
```

Sending short video messages

A short video message is defined by `TIMVideoElem`, which is a subclass of `TIMElem`. These messages can contain video snapshots and content. To send a short video, add `TIMVideoElem` to `TIMMessage` and send the video along with the message.

`TIMVideoElem` prototype:

```
/**
* Short video message
*/
@interface TIMVideoElem : TIMElem
/**
* Upload task ID, which can be used to query the upload progress. This parameter has been depreca
```

```
ted. Therefore, use `TIMUploadProgressListener` instead to listen to the upload progress.
*/
@property(nonatomic,assign) uint32_t taskId DEPRECATED_ATTRIBUTE;
/**
* Video file path, which is set when the message is sent
*/
@property(nonatomic,strong) NSString * videoPath;
/**
* Video information, which is set when the message is sent
*/
@property(nonatomic,strong) TIMVideo * video;
/**
* Snapshot file path, which is set when the message is sent
*/
@property(nonatomic,strong) NSString * snapshotPath;
/**
* Video snapshot, which is set when the message is sent
*/
@property(nonatomic,strong) TIMSnapshot * snapshot;
@end
```

Parameter description:

Parameter	Description
taskId	**The upload task ID, which can be used to query the upload progress. This parameter has been deprecated. Therefore, use `TIMUploadProgressListener` instead to listen to the upload progress.**
videoPath	**The path of the local video to be sent.**
video	**The video information. Set the `type` and `duration` parameters when sending the message.**
snapshotPath	**The local snapshot path of the short video to be sent.**
snapshot	**The snapshot information. Set the `type` and `duration` parameters when sending the message.**

The following example shows how to send a short video message. Example:

```
/**
* Obtain the conversation with user iOS-001.
*/
TIMConversation * c2c_conversation = [[TIMManager sharedInstance] getConversation:TIM_C2C receive
r:@"iOS-001"];
/**
```

```
* Construct a message.
*/
TIMMessage * msg = [[TIMMessage alloc] init];
/**
* Construct the video content.
*/
TIMVideoElem * videoElem = [[TIMVideoElem alloc] init];
videoElem.videoPath = @"/xxx/videoPath.mp4";
videoElem.video = [[TIMVideo alloc] init];
videoElem.video.type = @"mp4";
videoElem.video.duration = 10;
videoElem.snapshotPath = @"/xxx/snapshotPath.jpg";
videoElem.snapshot = [[TIMSnapshot alloc] init];
videoElem.snapshot.type = @"jpg";
videoElem.snapshot.width = 100;
videoElem.snapshot.height = 200;
/**
* Add the short video content to the message container.
*/
[msg addElem:videoElem];
/**
* Send the message.
*/
[conversation sendMessage:msg succ:^(){ // Successful
NSLog(@"SendMsg Succ");
}fail:^(int code, NSString * err) { // Failed
NSLog(@"SendMsg Failed:%d->%@", code, err);
}];
```

'Elem' order

Currently, file and audio `Elem` objects might not be transferred in the order that they are added. Other `Elem` objects are transferred in the order that they are added. However, we recommend that you do not rely heavily on the `Elem` object sequence when processing elements. Instead, you should process `Elem` objects by type to prevent process crashes when exceptions occur.

Online messages

In some scenarios, you need to send online messages, which can only be received when a user is online. If the user is not online when the messages are sent, the user will not see them upon the next login. Online messages can be used for notifications. However, online messages will not be stored or included in the unread count. The API for sending online messages is similar to `sendMessage` .

If you don't want to receive offline pushes, you can call the `TIMOfflinePushInfo` and set the `TIMOfflinePushFlag` and `TIM_OFFLINE_PUSH_NO_PUSH` parameters to disable push.

> **Note :**
>
> - **In versions earlier than 2.5.3, online messages apply only to C2C conversations.**
> - **In version 2.5.3 or later, online messages apply to group conversations, excluding audio-video chat rooms (AVChatRoom) and broadcasting chat rooms (BChatRoom).**

```
@interface TIMConversation : NSObject
/**
* Send online message (the server does not save the message)
*
* @param msg Message body
* @param succ Success callback
* @param Failure callback
*
* @return 0 Success
*/
-(int) sendOnlineMessage: (TIMMessage*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Forwarding messages

In version 2.4.0 and later, you can use `copyFrom` of `TIMMessage` to copy the content of another message to the current message and resend it to other contacts.

Prototype:

```
/**
* Message
*/
@interface TIMMessage : NSObject
/**
* Copy properties in the message (Elem, priority, online, and offlinePushInfo).
*
* @param srcMsg Source message
*
* @return 0 Success
*/
- (int)copyFrom:(TIMMessage*)srcMsg;
@end
```

Receiving Messages

To be notified of new messages, register the new message notification callback `TIMMessageListener`. If you have logged in, the IM SDK will use the `onNewMessage` callback to send new messages. Callback message content is passed using the `TIMMessage` parameter. With `TIMMessage`, you can get detailed information about messages and the corresponding conversations, such as text, audio data, and images. For more information, see Parsing messages.

> **Note :**
> **Messages obtained using `onNewMessages` may not be unread messages. They can also be messages that have not been displayed locally. For example, when messages are already read on another client, messages of recent contacts can be pulled to obtain the last messages of conversations. If these last messages are not stored locally, they are sent using this method. After a user logs in, the IM SDK gets C2C offline messages. To avoid missing message notifications, the user needs to register new message notifications before login.**
> **The `onNewMessage` callback is also used to send group system messages, relationship chain changes, and friend profile changes.**

Prototype:

```
@protocol TIMMessageListener
@optional
/**
* New message notification
*
* @param msgs List of new messages, an array of TIMMessage types
*/
- (void)onNewMessage:(NSArray*) msgs;
@end
@interface TIMManager : NSObject
- (int)addMessageListener:(id<TIMMessageListener>)listener;
@end
```

Parameter description:

Parameter	Description
msgs	**The list of new messages. Note that multiple messages may be sent at the same time. Messages in the same conversation are sorted from old to new.**

The following example sets a message callback notification and prints new messages when they arrive. Example:

```
@interface TIMMessageListenerImpl : NSObject
- (void)onNewMessage:(NSArray*) msgs;
@end
@implementation TIMMessageListenerImpl
- (void)onNewMessage:(NSArray*) msgs {
NSLog(@"NewMessages: %@", msgs);
}
@end
TIMMessageListenerImpl * impl = [[TIMMessageListenerImpl alloc] init];
[[TIMManager sharedInstance] addMessageListener:impl];
```

Parsing messages

After receiving a message, use `getElem` to obtain all `Elem` nodes in `TIMMessage`.

Prototype for traversing `Elem`:

```
@interface TIMMessage : NSObject
-(int) elemCount;
-(TIMElem*) getElem:(int)index;
@end
```

Example:

```
TIMMessage * message = /* Message */
int cnt = [message elemCount];
for (int i = 0; i < cnt; i++) {
TIMElem * elem = [message getElem:i];
if ([elem isKindOfClass:[TIMTextElem class]]) {
TIMTextElem * text_elem = (TIMTextElem * )elem;
}
else if ([elem isKindOfClass:[TIMImageElem class]]) {
TIMImageElem * image_elem = (TIMImageElem * )elem;
}
}
```

Receiving image messages

After receiving a message, use `getElem` to obtain all `Elem` nodes from `TIMMessage`. Nodes of the `TIMImageElem` type are image message nodes. To obtain all image specifications for display, use `imageList`.

**** `TIMImageElem` prototype:****

```
**
* Image message `Elem`
*/
@interface TIMImageElem : TIMElem
/**
* Path of the image to be sent
*/
@property(nonatomic,retain) NSString * path;
/**
* Save all specifications of the image, including the thumbnail, large image, and original image.
Each specification is saved in a `TIMImage` object.
*/
@property(nonatomic,retain) NSArray * imageList;
@end
```

Parameter description:

Parameter	Description
path	**You do not need to consider this parameter when receiving messages. It has a value of nil.**
imageList	**Saves all specifications of the image, including the thumbnail, large image, and original image. Each specification is saved in a `TIMImage` object.**

`TIMImage`:

After receiving a message, use `imageList` to obtain all image specifications, which are `TIMImage` objects. After obtaining `TIMImage`, reserve a place based on the image size and use `getImage` to download images of different specifications for display. Developers need to cache the downloaded data. The IM SDK downloads data from the server each time it calls `getImage`. We recommend that you use the `uuid` of an image as the `key` to store images.

Prototype:

```
@interface TIMImage : NSObject
/**
* Image ID, which is the internal identifier. It can be used as the key for external caching.
*/
@property(nonatomic,strong) NSString * uuid;
/**
* Image type
*/
@property(nonatomic,assign) TIM_IMAGE_TYPE type;
/**
* Image size
*/
@property(nonatomic,assign) int size;
/**
* Image width
*/
@property(nonatomic,assign) int width;
/**
* Image height
*/
@property(nonatomic,assign) int height;
/**
* Download URL
*/
@property(nonatomic, strong) NSString * url;
/**
* Obtain the image.
*
* Developers need to cache the downloaded data. The IM SDK downloads data from the server each ti
me it calls `getImage`. We recommend that you use the `uuid` of an image as the `key` to store im
ages.
*
* @param path Image storage path
* @param succ Success callback, which returns image data
* @param fail Failure callback, which returns the error code and error description
*/
- (void)getImage:(NSString*)path succ:(TIMSucc)succ fail:(TIMFail)fail;
/**
* Obtain the image (with progress callback).
*
* Developers need to cache the downloaded data. The IM SDK downloads data from the server each ti
me it calls `getImage`. We recommend that you use the `uuid` of an image as the `key` to store im
ages.
*
* @param path Image storage path
* @param progress Image download progress
* @param succ Success callback, which returns image data
```

```
 * @param fail Failure callback, which returns the error code and error description
 */
- (void)getImage:(NSString*)path progress:(TIMProgress)progress succ:(TIMSucc)succ fail:(TIMFail)
fail;
@end
```

Image specifications: each image has three specifications, including Original (original image), Large (large image), and Thumb (thumbnail).

- **Original image: the original image sent by a user, whose dimensions and size remain unchanged.**
- **Large image: an image obtained after the original image is proportionally compressed. The height or width of the compressed image, whichever is smaller, is equal to 720 pixels.**
- **Thumbnail: an image obtained after the original image is proportionally compressed. The height or width of the compressed image, whichever is smaller, is equal to 198 pixels.**

Note :

- **If the size of the original image is less than 198 pixels, the original size is retained for the three specifications, and no compression is needed.**
- **If the size of the original image falls between 198 and 720 pixels, the large image is the same as the original image, and no compression is needed.**
- **When an image is displayed on a mobile phone, we recommend that the thumbnail be displayed first. When a user taps the thumbnail, the large image is downloaded. When the user taps the large image, the original image is downloaded. Alternatively, developers can choose to skip the large image so that the original image is downloaded when the user taps the thumbnail.**
- **When an image is displayed on a tablet or PC, we recommend that the large image be displayed directly and the original image be downloaded when a user taps or clicks the large image due to the high resolution and availability of a Wi-Fi or wired network.**

The following example fetches 10 messages from a conversation, obtains image messages, and downloads the relevant data. Example:

```
//The following uses the new message callback as an example to explain the process of parsing ima
ge messages.
// Path for saving the received images
```

```
NSString * pic_path = @"/xxx/imgPath.jpg";
[conversation getMessage:10 last:nil succ:^(NSArray * msgList) { // Messages obtained successfull
y.
// Traverse all messages.
for (TIMMessage * msg in msgList) {
// Traverse all elements in a message.
for (int i = 0; i < msg.elemCount; ++i) {
TIMElem *elem = [msg getElem:i];
// Image element
if ([elem isKindOfClass:[TIMImageElem class]]) {
TIMImageElem * image_elem = (TIMImageElem * )elem;
// Traverse all image specifications (Thumbnail, Large, and Original).
NSArray * imgList = [image_elem imageList];
for (TIMImage * image in imgList) {
[image getImage:pic_path succ:^(){ // Received successfully.
NSLog(@"SUCC: pic store to %@", pic_path);
}fail:^(int code, NSString * err) { // Failed to receive.
NSLog(@"ERR: code=%d, err=%@", code, err);
}];
}
}
}
}
} fail:^(int code, NSString * err) { // Failed to receive messages.
NSLog(@"Get Message Failed:%d->%@", code, err);
}];
```

Receiving audio messages

After receiving a message, use `getElem` to obtain all `Elem` nodes from `TIMMessage`. Nodes of the `TIMElemType.Sound` type are audio message nodes. `path` indicates the audio message path specified when a message is created, and `path` is null when a message is received. When the message is received, reserve a place based on the audio length and use `getSound` to download audio resources. The `getSound` API downloads audio data from the server. To cache or store the data, use the `uuid` as the `key` to store the audio data externally. The IM SDK does not store resource files.

Prototype:

```
@interface TIMSoundElem : TIMElem
/**
* Upload task ID, which can be used to query the upload progress. This parameter has been depreca
ted. Therefore, use `TIMUploadProgressListener` instead to listen to the upload progress.
*/
@property(nonatomic,assign) uint32_t taskId DEPRECATED_ATTRIBUTE;
/**
```

```
* Set to audio data when sending the message, and use `getSound` to obtain data when receiving th
e message.
*/
@property(nonatomic,strong) NSString * path;
/**
* Internal ID of the audio message
*/
@property(nonatomic,strong) NSString * uuid;
/**
* Audio data size
*/
@property(nonatomic,assign) int dataSize;
/**
* Audio length in seconds, which should be set when the message is sent
*/
@property(nonatomic,assign) int second;
/**
* Obtain the download URL of the audio file.
*
* @param urlCallBack Callback for obtaining the URL
*/
-(void)getUrl:(void (^)(NSString * url))urlCallBack;
/**
* Save audio data to a file in the specified path.
*
* The `getSound` API downloads audio data from the server. To cache or store the data, use `uuid`
as the `key` to store the audio data externally. The IM SDK does not store resource files.
*
* @param path Audio storage path
* @param succ Success callback, which returns audio data
* @param fail Failure callback, which returns the error code and error description
*/
- (void)getSound:(NSString*)path succ:(TIMSucc)succ fail:(TIMFail)fail;
/**
* Save audio data to a file in the specified path (with progress callback).
*
* The `getSound` API downloads audio data from the server. To cache or store the data, use `uuid`
as the `key` to store the audio data externally. The IM SDK does not store resource files.
*
* @param path Audio storage path
* @param progress Audio download progress
* @param succ Success callback
* @param fail Failure callback, which returns the error code and error description
*/
- (void)getSound:(NSString*)path progress:(TIMProgress)progress succ:(TIMSucc)succ fail:(TIMFail)
fail;
@end
```

Other parameters:

Parameter	Description
path	**Set to audio data when sending the message, and use `getSound` to obtain data when receiving the message.**
uuid	**The unique identifier used to facilitate caching.**
dataSize	**The audio file size.**
second	**The audio length in seconds.**

Audio message read status: you can use custom message fields to determine whether an audio message has been played. For example, a `customInt` value of 0 indicates that the audio has not been played, and a value 1 indicates that the audio has been played. After a user taps play, set `customInt` to 1.

```
@interface TIMMessage : NSObject
/**
* Set the custom integer. The default value is 0.
*
* @param param Set parameter
*
* @return TRUE Set successfully.
*/
- (BOOL) setCustomInt:(int32_t) param;
/**
* Obtain `CustomInt`.
*
* @return CustomInt
*/
- (int32_t)customInt;
@end
```

Receiving file messages

After receiving a message, use `getElem` to obtain all `Elem` nodes from `TIMMessage`. Nodes of the `TIMFileElem` type are file message nodes. `path` is the file path entered when the message is created and is empty when GET is performed to send the message. You can choose to display only the file size and name when receiving the message and use `getFile` to download the file from the server each time. To cache or store the data, use the `uuid` as the `key` to store the file externally. The IM SDK does not store resource files.

Prototype:

```
@interface TIMFileElem : TIMElem
/**
* Upload task ID, which can be used to query the upload progress. This parameter has been deprecated. Therefore, use `TIMUploadProgressListener` instead to listen to the upload progress.
*/
@property(nonatomic,assign) uint32_t taskId DEPRECATED_ATTRIBUTE;
/**
* Path of the file to be uploaded. (If the path is specified, the file in the specified path will be uploaded first.)
*/
@property(nonatomic,strong) NSString * path;
/**
* Internal ID of the file
*/
@property(nonatomic,strong) NSString * uuid;
/**
* File size
*/
@property(nonatomic,assign) int fileSize;
/**
* Display name of the file, which is set when the message is sent
*/
@property(nonatomic,strong) NSString * filename;
/**
* Obtain the download URL of the file.
*
* @param urlCallBack Callback for obtaining the URL
*/
-(void)getUrl:(void (^)(NSString * url))urlCallBack;
/**
* Save file data to the file in the specified path.
*
* The `getFile` API downloads file data from the server. To cache or store the data, use the `uuid` as the `key` to store the file externally. The IM SDK does not store resource files.
*
* @param path File storage path
* @param succ Success callback, which returns data
* @param fail Failure callback, which returns the error code and error description
*/
- (void)getFile:(NSString*)path succ:(TIMSucc)succ fail:(TIMFail)fail;
/**
* Save file data to a file in the specified path (with progress callback).
*
* The `getFile` API downloads file data from the server. To cache or store the data, use the `uuid` as the `key` to store the file externally. The IM SDK does not store resource files.
```

```
*
* @param path File storage path
* @param progress File download progress
* @param succ Success callback, which returns data
* @param fail Failure callback, which returns the error code and error description
*/
- (void)getFile:(NSString*)path progress:(TIMProgress)progress succ:(TIMSucc)succ fail:(TIMFail)f
ail;
@end
```

Parameter description:

Parameter	Description
path	**The path of the file to be uploaded.**
uuid	**The unique identifier used to facilitate caching.**
fileSize	**The file size.**
filename	**The display name of the file.**

Receiving short video messages

After receiving a message, use `getElem` to obtain all `Elem` nodes from `TIMMessage`. Nodes of the `TIMVideoElem` type are short video message nodes. Use the `TIMVideo` and `TIMSnapshot` objects to obtain the video and snapshot content. After receiving `TIMVideoElem`, download the video file and snapshot file through the APIs defined in the `video` and `snapshot` properties. To cache or store the data, use the `uuid` as the `key` to store the files externally. The IM SDK does not store resource files.

Prototype:

```
@interface TIMVideo : NSObject
/**
* Internal ID of the video message, which does not need to be set
*/
@property(nonatomic,strong) NSString * uuid;
/**
* Video file type, which is set when the message is sent
*/
@property(nonatomic,strong) NSString * type;
/**
* Video size, which does not need to be set
*/
@property(nonatomic,assign) int size;
/**
```

```
* Video length, which is set when the message is sent
*/
@property(nonatomic,assign) int duration;
/**
* Obtain the download URL of the video.
*
* @param urlCallBack Callback for obtaining the URL
*/
-(void)getUrl:(void (^)(NSString * url))urlCallBack;
/**
* Obtain the video.
*
* The `getVideo` API downloads file data from the server. To cache or store the data, use the `uu
id` as the `key` to store the file externally. The IM SDK does not store resource files.
*
* @param path Video storage path
* @param succ Success callback
* @param fail Failure callback, which returns the error code and error description
*/
- (void)getVideo:(NSString*)path succ:(TIMSucc)succ fail:(TIMFail)fail;
/**
* Obtain the video (with progress callback).
*
* The `getVideo` API downloads file data from the server. To cache or store the data, use the `uu
id` as the `key` to store the file externally. The IM SDK does not store resource files.
*
* @param path Video storage path
* @param progress Video download progress
* @param succ Success callback
* @param fail Failure callback, which returns the error code and error description
*/
- (void)getVideo:(NSString*)path progress:(TIMProgress)progress succ:(TIMSucc)succ fail:(TIMFail)
fail;
@end
@interface TIMSnapshot : NSObject
/**
* Image ID, which does not need to be set
*/
@property(nonatomic,strong) NSString * uuid;
/**
* Snapshot file type, which is set when the message is sent
*/
@property(nonatomic,strong) NSString * type;
/**
* Image size, which does not need to be set
*/
@property(nonatomic,assign) int size;
/**
```

```
* Image width, which is set when the message is sent
*/
@property(nonatomic,assign) int width;
/**
* Image height, which is set when the message is sent
*/
@property(nonatomic,assign) int height;
/**
* Obtain the download URL of the snapshot.
*
* @param urlCallBack Callback for obtaining the URL
*/
-(void)getUrl:(void (^)(NSString * url))urlCallBack;
/**
* Obtain the image.
*
* The `getImage` API downloads file data from the server. To cache or store the data, use the `uu
id` as the `key` to store the file externally. The IM SDK does not store resource files.
*
* @param path Image storage path
* @param succ Success callback, which returns image data
* @param fail Failure callback, which returns the error code and error description
*/
- (void)getImage:(NSString*)path succ:(TIMSucc)succ fail:(TIMFail)fail;
/**
* Obtain the image (with progress callback).
*
* The `getImage` API downloads file data from the server. To cache or store the data, use the `uu
id` as the `key` to store the file externally. The IM SDK does not store resource files.
*
* @param path Image storage path
* @param progress Image download progress
* @param succ Success callback, which returns image data
* @param fail Failure callback, which returns the error code and error description
*/
- (void)getImage:(NSString*)path progress:(TIMProgress)progress succ:(TIMSucc)succ fail:(TIMFail)
fail;
@end
// The following uses the new message callback as an example to explain the process of parsing sh
ort video messages.
// Path for saving the received video and snapshot
NSString * video_path = @"/xxx/video.mp4";
NSString * snapshot_path = @"/xxx/snapshot.jpg";
[conversation getMessage:10 last:nil succ:^(NSArray * msgList) { // Messages obtained successfull
y.
// Traverse all messages.
for (TIMMessage * msg in msgList) {
// Traverse all elements in a message.
```

```
for (int i = 0; i < msg.elemCount; ++i) {
TIMElem *elem = [msg getElem:i];
if ([elem isKindOfClass:[TIMVideoElem class]]) {
TIMVideoElem * video_elem = (TIMVideoElem * )elem;
[video_elem.video getVideo:video_path succ:^() {
NSLog(@"Video file downloaded successfully");
}  fail:^(int code, NSString * err) {
NSLog(@"Failed to download the video file:%@ %d", err, code);
}];
[video_elem.snapshot getImage:snapshot_path succ:^() {
NSLog(@"Snapshot downloaded successfully");
} fail:^(int code, NSString * err) {
NSLog(@"Failed to download the video snapshot:%@ %d", err, code);
}];
}
}
} fail:^(int code, NSString * err) { // Failed to receive messages.
NSLog(@"Get Message Failed:%d->%@", code, err);
}];
```

Message Properties

Checking whether a message has been read

Use `isRead` to check whether a message has been read, which is determined by the Unread Count on the app side.

```
@interface TIMMessage : NSObject
/**
* Check whether a message has been read.
*
* @return TRUE: read. FALSE: unread.
*/
-(BOOL) isReaded;
@end
```

Message status

To obtain the status of the current message, such as sending, sent successfully, failed to send, or deleted, use `status`. For deleted messages, use the UI to determine the status and hide the messages accordingly.

```
/**
* Message status
*/
typedef NS_ENUM(NSInteger, TIMMessageStatus){
/**
* Sending
*/
TIM_MSG_STATUS_SENDING = 1,
/**
* Sent successfully
*/
TIM_MSG_STATUS_SEND_SUCC = 2,
/**
* Failed to send
*/
TIM_MSG_STATUS_SEND_FAIL = 3,
/**
* Deleted
*/
TIM_MSG_STATUS_HAS_DELETED = 4,
/**
* Imported to local storage
*/
TIM_MSG_STATUS_LOCAL_STORED = 5,
/**
* Recalled
*/
TIM_MSG_STATUS_LOCAL_REVOKED = 6,
};
@interface TIMMessage : NSObject
/**
* Message status
*
* @return TIMMessageStatus Message status
*/
-(TIMMessageStatus) status;
@end
```

Checking whether a message was sent by oneself

To determine whether a message was sent by you yourself, use `isSelf`. This method is available when the message is displayed on the interface.

```
@interface TIMMessage : NSObject
/**
* Check whether the user is the sender.
```

```
*
* @return TRUE: message sender. FALSE: message recipient.
*/
-(BOOL) isSelf;
@end
```

Message sender and related profile

For group messages, use the `sender` method of `TIMMessage` to obtain the sender, or use the `GetSenderProfile` and `GetSenderGroupMemberProfile` methods to obtain the sender profile and the profile of the sender's group. In versions earlier than 1.9, you can only obtain the sender profile from an online message received after `onNewMessage` is used. In version 1.9 and later, you can obtain the sender profile from any message received after `getMessage` is used (but not from local messages received before the version update). For a C2C conversation, use `getConversation` of `TIMMessage` to obtain the conversation and use `getReceiver` to obtain information on the peer in the conversation.

> **Note :**
> **This field obtains the user profile and writes it to the message body when the message is sent. If the user profile is updated, this field will not change unless new messages are generated.**

```
@interface TIMMessage : NSObject
/**
* Obtain the sender.
*
* @return Sender identifier
*/
-(NSString *) sender;
/**
* Obtain the sender profile.
*
* If the sender profile is stored locally, this profile is returned synchronously in the `profile
CallBack`. Otherwise, the SDK obtains the sender profile from the server and returns it asynchron
ously in the `profileCallBack`.
*
* @param profileCallBack Sender profile callback
*
*/
- (void)getSenderProfile:(ProfileCallBack)profileCallBack;
/**
```

```
* Obtain the sender profile in the group, which may be empty if the sender is the user.
*
* @return Sender's profile in the group. "nil" indicates that no profile is obtained or the messa
ge is not a group message. Currently, only the `member` field can be obtained. To obtain other fi
elds, use `TIMGroupManager+Ext.h` > `getGroupMembers`.
*/
-(TIMGroupMemberInfo *) GetSenderGroupMemberProfile;
@end
```

Message time

To obtain the message time, use `timestamp`. This time is the server time, not the local time. When you create a message, this time is calibrated based on the server time and will be changed to the accurate server time after the message is successfully sent.

```
@interface TIMMessage : NSObject
/**
* Timestamp of the current message
*
* @return Timestamp
*/
-(NSDate*) timestamp;
@end
```

Message ID

There are two types of message IDs. One is `msgId`, which is created when a message is generated. If `msgId` is used, messages may conflict with messages generated by other users, and therefore a time dimension needs to be added. Messages generated within 10 minutes can be distinguished by `msgId`. The other is `uniqueId`, which is generated after a message is sent successfully and is globally unique. Both types of message IDs must be checked in the same conversation.

```
@interface TIMMessage : NSObject
/**
* Message ID
*/
-(NSString *) msgId;
/**
* Obtain the uniqueId of the message.
*
* @return uniqueId
*/
```

```
- (uint64_t) uniqueId;
@end
```

Custom message field

Developers can add custom fields to messages, such as the custom integer and custom binary data fields, and can customize different effects based on these two fields. For example, custom fields can be used to determine whether an audio message has been played. Note that these custom fields are only stored locally and not synchronized to the server. You will not obtain them after switching to another client.

```
@interface TIMMessage : NSObject
/**
* Set the custom integer, which is 0 by default.
*
* @param param Set parameter
*
* @return TRUE Set successfully.
*/
- (BOOL) setCustomInt:(int32_t) param;
/**
* Set the custom data content, which is "". by default.
*
* @param data Set parameter
*
* @return TRUE Set successfully.
*/
- (BOOL) setCustomData:(NSData*) data;
/**
* Obtain CustomInt.
*
* @return CustomInt
*/
- (int32_t) customInt;
/**
* Obtain CustomData.
*
* @return CustomData
*/
- (NSData*) customData;
@end
```

Message priority

Livestreaming scenarios involve the like and red packet features. Like messages have a lower priority than red packet messages. You can use `TIMCustomElem` to define the message content, and define the message priority when sending a message.

> **Note :**
> **Message priorities apply only to group messages.**

```
@interface TIMMessage : NSObject
/**
* Set the message priority.
*
* @param priority Priority
*
* @return TRUE Set successfully.
*/
- (BOOL) setPriority:(TIMMessagePriority)priority;
/**
* Obtain the message priority.
*
* @return Priority
*/
- (TIMMessagePriority) getPriority;
@end
```

Read receipt

After the read receipt feature is enabled for C2C conversations, read messages are synchronized to the client when the recipient calls `setReadMessage` .

Enable the read receipt feature:

```
@interface TIMUserConfig : NSObject
/**
* Enable the read receipt feature. Then, read receipts will be sent to the message sender when me
ssage read reports are reported. This feature applies only to C2C conversations.
*/
-(void) enableReadReceipt;
/**
* Message read receipt listener
*/
@property(nonatomic,weak) id<TIMMessageReceiptListener> messageReceiptListener;
@end
```

Prototype:

```
@interface TIMMessage : NSObject
/**
* Check whether the recipient has read the message. (This feature applies only to C2C messages.)
*
* @return TRUE: read. FALSE: unread.
*/
-(BOOL) isPeerReaded;
@end
```

Conversation Operations

Obtaining all conversations

```
@interface TIMManager : NSObject
/**
* Obtain the conversation (TIMConversation*) list.
*
* @return Conversation list
*/
-(NSArray*) getConversationList;
@end
```

Note :
The SDK continuously updates the conversation list internally. The update will be sent back to the caller using `TIMRefreshListener.onRefresh`. Call `getConversationList` after `onRefresh` to update the conversation list.

Example:

```
NSArray * conversations = [[TIMManager sharedInstance] getConversationList];
NSLog(@"current session list : %@", [conversations description])
```

Obtaining local messages in a conversation

The IM SDK stores messages locally. To obtain these messages, use `getLocalMessage` of `TIMConversation`. This is an asynchronous method, and a callback needs to be set to obtain message data. For a C2C conversation, offline messages will be obtained automatically

after login. For a group conversation, when recent contacts roaming is enabled, only the last message is obtained after login, and roaming messages can be obtained using `getMessage` . For resource messages such as image and audio messages, the message body only contains descriptive information, and additional APIs are required to download data. For more information, see Parsing Messages. The actual data downloaded is not cached, and must be cached by the caller.

Prototype:

```
@interface TIMConversation : NSObject
/**
* Obtain messages in a local conversation.
*
* @param count Number of messages
* @param last The last message
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Operation successful
*/
-(int) getLocalMessage: (int)count last:(TIMMessage*)last succ:(TIMGetMsgSucc)succ fail:(TIMFail)
fail;
@end
```

Parameter description:

Parameter	Description
count	**Specifies the number of messages to obtain.**
last	**Specifies the last message obtained. If `last` passes `nil` , read from the latest message.**
succ	**The success callback.**
fail	**The failure callback.**

Example:

```
[conversation getLocalMessage:10 last:nil succ:^(NSArray * msgList) {
for (TIMMessage * msg in msgList) {
if ([msg isKindOfClass:[TIMMessage class]]) {
NSLog(@"GetOneMessage:%@", msg);
}
}
```

```
}fail:^(int code, NSString * err) {
NSLog(@"Get Message Failed:%d->%@", code, err);
}];
```

Obtaining roaming messages in a conversation

For group conversations, a user can obtain roaming messages after login. For C2C conversations, the user can obtain roaming messages after the roaming service is enabled. To obtain roaming messages, use `getMessage` of the IM SDK. If local messages are continuous, they are obtained directly, instead of over the network. If local messages are not continuous, missing messages need to be obtained over the network. For resource messages such as image and audio messages, the message body only contains descriptive information, and additional APIs are required to download data, which can participate in message parsing. The actual data downloaded is not cached, and must be cached by the caller.

Prototype:

```
@interface TIMConversation : NSObject
/**
* Obtain messages in a conversation.
*
* @param count Number of messages
* @param last The last message
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Operation successful
*/
-(int) getMessage: (int)count last:(TIMMessage*)last succ:(TIMGetMsgSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
count	**Specifies the number of messages to obtain.**
last	**Specifies the last message obtained. If `last` passes `nil`, read from the latest message.**
succ	**The success callback.**
fail	**The failure callback.**

Example:

```
[conversation getMessage:10 last:nil succ:^(NSArray * msgList) {
for (TIMMessage * msg in msgList) {
if ([msg isKindOfClass:[TIMMessage class]]) {
NSLog(@"GetOneMessage:%@", msg);
}
}
}fail:^(int code, NSString * err) {
NSLog(@"Get Message Failed:%d->%@", code, err);
}];
```

Deleting conversations

While deleting a conversation, the IM SDK also deletes the local and roaming messages of this conversation, and the deleted conversation and messages cannot be recovered.

Prototype:

```
@protocol TIMManager : NSObject
/**
*
* When a conversation is deleted, the roaming messages of this conversation are also deleted from the local storage and backend.
*
* @param type Conversation type. For more information, see the definition of `TIMConversationType` in `TIMComm.h`.
* @param conversationId Conversation ID
* C2C: userID of the peer
* GROUP: groupId of the group
* SYSTEM: @""
*
* @return YES: deleted successfully. NO: failed to delete.
*/
- (BOOL)deleteConversation:(TIMConversationType)type receiver:(NSString*)conversationId;
@end
```

Parameter description:

Parameter	Description
type	**The conversation type. For a C2C conversation, enter TIM_C2C. For a group conversation, enter TIM_GROUP.**
conversationId	**The conversation ID. For a C2C conversation, `receiver` is the user ID of the peer. For a group conversation, `receiver` is the group ID.**

The following example deletes the C2C conversation with a friend named "iOS_002".
Example:

```
[[TIMManager sharedInstance] deleteConversation:TIM_C2C receiver:@"iOS_002"];
```

Synchronously obtaining the last message of a conversation

The UI displays the last messages from users in the recent contact list. In versions later than 1.9, the IM SDK provides the `getLastMsg` API to allow users to obtain and display the last messages. This feature requires a network connection. Messages obtained using this API contain deleted messages, which need to be blocked by the app. To obtain multiple recent messages, use `getMessage` .

Prototype:

```
@interface TIMConversation : NSObject
/**
* Obtain the last message from the cache.
* @return Last message (TIMMessage*)
*/
- (TIMMessage*)getLastMsg;
/**
* Obtain roaming messages of a conversation.
* @param count Number of messages
* @param last The last message. If the value of `last` is `nil`, read from the latest message.
* @param succ Success callback
* @param fail Failure callback
* @return 0: this operation is successful. 1: this operation failed.
*/
- (int)getMessage:(int)count last:(TIMMessage*)last succ:(TIMGetMsgSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
count	**The number of messages to obtain. The maximum number is 20.**

Setting conversation drafts

The UI displays the user's drafts on the recent contact list. In version 2.2 or later, an API is added to allow users to set and obtain drafts. Draft information is stored in a local database and can be obtained after login.

Prototype:

```
@interface TIMMessageDraft : NSObject
/**
* Set the custom data.
*
* @param userData Custom data
*
* @return 0 Success
*/
-(int) setUserData:(NSData*)userData;
/**
* Obtain the custom data.
*
* @return Custom data
*/
-(NSData*) getUserData;
/**
* Add an `Elem`.
*
* @param elem Structure of `Elem`
*
* @return 0 Success
* 1 Adding Elem is not allowed (more than two file or audio `Elem` objects).
* 2 Unknown `Elem`
*/
-(int) addElem:(TIMElem*)elem;
/**
* Obtain `Elem` with the corresponding index.
*
* @param index Index
*
* @return The corresponding `Elem`
*/
-(TIMElem*) getElem:(int)index;
/**
* Obtain the number of `Elem` objects.
*
* @return elem Number of `Elem` objects
*/
-(int) elemCount;
/**
* Message generated based on the draft
*
* @return Message
*/
-(TIMMessage*) transformToMessage;
/**
* Timestamp of the current message
```

```
*
* @return Timestamp
*/
-(NSDate*) timestamp;
@end
@interface TIMConversation : NSObject
/**
* Set the conversation draft.
*
* @param draft Draft content
*
* @return 0 Success
*/
-(int) setDraft:(TIMMessageDraft*)draft;
/**
* Obtain the conversation draft.
*
* @return Draft content. `nil` is returned if no draft is available.
*/
-(TIMMessageDraft*) getDraft;
@end
```

Parameter description:

Parameter	Description
draft	**The draft to be set. To clear a conversation draft, pass `nil`.**

Deleting messages of a conversation

The IM SDK allows you to delete the local and roaming messages of a conversation, and deleted messages cannot be recovered.

Prototype:

```
@interface TIMConversation : NSObject
/**
* Delete the local and roaming messages of the current conversation.
*
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Operation successful
*/
- (int)deleteMessages:(NSArray<TIMMessage *>*)msgList succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
msgList	**The list of messages to be deleted.**
succ	**The success callback.**
fail	**The failure callback.**

Obtaining the message corresponding to a specified local ID

IM SDK 2.5.3 provides an API to obtain the message corresponding to a specified local ID.

Prototype:

```
/**
* Message
*/
@interface TIMMessage : NSObject
/**
* Obtain the message locator.
*
* @return locator
*/
- (TIMMessageLocator*) locator;
@end
@interface TIMConversation : NSObject
/**
* Obtain messages in a conversation.
*
* @param locators Message locator (TIMMessageLocator) array
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Operation successful
*/
-(int) findMessages:(NSArray*)locators succ:(TIMGetMsgSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
locators	**The list of message locators (TIMMessageLocator).**
succ	**The success callback, which returns a list of messages.**

Parameter	Description
fail	**The failure callback.**

Recalling messages

Starting from version 3.1.0, the IM SDK provides an API to recall messages. To recall sent messages, call the `revokeMessage` API of `TIMConversation`.

> **Note :**
>
> - **The API applies only to C2C and group conversations, and not to onlineMessages, AVChatRooms, or BChatRooms.**
> - **By default, only messages that were sent within the last 2 minutes can be recalled.**

Prototype:

```
/**
* Recall a message. (This API applies only to C2C and group conversations, and not to onlineMessages, AVChatRooms, or BChatRooms.)
* @param msg Message to be recalled
* @param succ Success callback
* @param fail Failure callback
*
* @return 0: this operation is successful. 1: this operation failed.
*/
- (int)revokeMessage:(TIMMessage*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
```

After a message is recalled, other members in the group or the peer in the C2C conversation will receive a message recall notification. In addition, the message recall notification listener `TIMMessageRevokeListener` notifies the upper-layer app. You can configure the message recall notification listener before login using `messageRevokeListener` of `TIMUserConfig`. For more information, see User Configuration.

Prototype:

```
@protocol TIMMessageRevokeListener <NSObject>
@optional
/**
* Message recall notification
*
```

```
 * @param locator Locator of the recalled message
 */
- (void)onRevokeMessage:(TIMMessageLocator*)locator;
@end
```

After receiving a message recall notification, use the `respondsToLocator` method of `TIMMessage` to check whether the current message has been recalled by the sender and then refresh the UI when necessary.

Prototype:

```
/**
* Check whether the message corresponds to the locator.
*
* @param locator Message locator
*
* @return YES: it is the correct message. NO: it is not the correct message.
*/
- (BOOL)respondsToLocator:(TIMMessageLocator*)locator;
```

System Messages

In addition to C2C conversations and group conversations, system message is another conversation type (TIMConversationType). System messages are notifications that are sent by the system backend for various events. These messages cannot be sent by users. Currently, there are two types of system messages: relationship chain system messages and group system messages.

- **Relationship chain change messages: the system sends a relationship chain change message when a user adds you as a friend or deletes you from his or her friend list. The developer can then update the friend list. For more information, see System Notifications for Relationship Chain Changes (iOS%20SDK).**

- **Group event messages: when the group profile is modified, for example, due to a change to the group name or group members, the system sends a group event message in the group. The developer can choose whether to display the message, and at the same time refresh the group profile or group members. For more information, see Group management - group event messages.**

- **Group system messages: when the group admin removes a member from the group or invites a user to join the group, the system sends a group system message to the user. For more information, see Group management - group system messages.**

Unread Count
Unread Message Counting (Android)

Last updated : 2021-03-23 10:25:39

Unread Messages

Here, unread messages are messages that have not been reported as read by users. This does not indicate whether the recipient has actually read the messages. To use this feature, you need to enable read receipts. For more information, see the Read receipt section in Sending and Receiving Messages (Android). The `isRead` of `TIMMessage` identifies whether a message is read or not. To display the correct unread count, developers need to explicitly call read reports to notify the IM SDK whether the messages have been read. For example, you can mark all messages as read when the user enters the chat UI.

> ⚠ **Note :**
>
> For chat rooms, the server does not save the unread count. The unread count becomes zero after login is completed and being synchronized with the server.

Prototype:

```
/**
* Check whether the message is read.
* @return Return the result of read or unread.
*/
public boolean isRead()
```

Getting the Current Unread Count

You can get the unread count of the current conversation via the `getUnReadMessageNum` method of `TIMConversation` .

> ⚠ **Note :**

For chat rooms, the server does not save the unread count. The unread count becomes zero after login is completed and being synchronized with the server.

Prototype:

```
/**
* Get the unread count.
* @return Return the unread count.
*/
public long getUnreadMessageNum()
```

Example:

```
// Get the conversation extension instance.
TIMConversation con = TIMManager.getInstance().getConversation(TIMConversationType.C2C, peer);
// Get the conversation unread count.
long num = con.getUnreadMessageNum();
Log.d(tag, "unread msg num: " + num);
```

Read Reporting

When a user reads a message in a conversation and a read report is sent, and IM SDK sets all messages before the last read message as read. The API for read reporting is `setReadMessage` in `TIMConversation`.

Prototype:

```
/**
* Mark all messages before this message as read.
* @param msg The last read message. Passing `null` marks all messages in the conversation as read
.
* @param callback Callback
*/
public void setReadMessage(TIMMessage msg, TIMCallBack callback)
```

Examples:

```
// Report read messages in a one-to-one conversation.
String peer = "sample_user_1"; // Get the conversation with the user "sample_user_1".
TIMConversation conversation = TIMManager.getInstance().getConversation(
TIMConversationType.C2C, // Conversation type: one-to-one chat
peer); // Account of the other party of the conversation
```

```
// Mark all messages in this conversation as read.
conversation.setReadMessage(null, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "setReadMessage failed, code: " + code + "|desc: " + desc);
}

@Override
public void onSuccess() {
Log.d(tag, "setReadMessage succ");
}
});

// Report read messages in a group conversation.
String groupId = "TGID1EDABEAE0"; // Get the conversation of the group "TGID1LTTZEAE0".
conversation = TIMManager.getInstance().getConversation(
TIMConversationType.Group, //Conversation type: group chat
groupId); // Group ID
// Mark the message represented by `lastMsg` and all previous messages as read.
conversation.setReadMessage(lastMsg, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "setReadMessage failed, code: " + code + "|desc: " + desc);
}

@Override
public void onSuccess() {
Log.d(tag, "setReadMessage succ");
}
});
```

Note :

The methods for setting messages in one-to-one and group chats as read are the same. The only difference is the conversation type.

Synchronizing Read Reports Across Multiple Devices

In the case of multi-device login, unread count synchronization notifications are sent by the server. The IM SDK updates the unread count locally and notifies the user to update conversations. Notifications trigger callbacks through the `onRefreshConversation` API in `TIMRefreshListener`. Users who require multi-device synchronization can perform relevant

synchronization through this API. For more information, see the Conversation refresh listener section in Initialization (Android).

Prototype:

```
/**
* Refresh some conversations, including synchronizing multi-device read reports.
* @param conversations List of conversations to be refreshed
*/
public void onRefreshConversation(List<TIMConversation> conversations);
```

Disabling Auto Reporting

For better performance, the IM SDK pulls unread messages to local storage, and the server deletes the unread messages by default. Unread messages that are previously pulled on another device are invisible to devices you switch to later, even if the read messages haven't been manually reported. For a single device, the unread count is correct. If you want unread messages to appear on multiple devices, you can disable auto reporting using the `disableAutoReport` method in `TIMUserConfig`. IM does not perform read reporting for users after auto reporting is disabled.

⚠ Note :

- **After auto reporting is disabled, developers need to explicitly call `setReadMessage` to perform read reporting.**
- **This must be set before login for the setting to take effect.**

Prototype:

```
/**
* Configure whether to enable auto read reporting, which is enabled by default, and set this before login.
* @param disableAutoReport `true`: disabled; `false`: enabled.
*/
public TIMUserConfig disableAutoReport(boolean disableAutoReport)
```

Unread Count (iOS)

Last updated : 2020-07-10 11:57:18

Unread Messages

Unread messages are messages that have not been read by users. This does not indicate whether the recipient has read the message. The `isReaded` method of `TIMMessage` identifies whether or not the messages are read. To show the correct unread count, developers need to explicitly call read reports to tell the app whether a message is read. For example, you can set the configuration so that all messages are read when the user enters the chat interface. For chat rooms, the server does not save the unread count. The unread count becomes zero after login and the unread count is synced with the server.

```
@interface TIMMessage : NSObject
/**
* Read or unread
*
* @return TRUE: read. FALSE: unread.
*/
- (BOOL)isReaded;
@end
```

Getting the Current Unread Count

The unread count of the current conversation can be obtained through the `getUnReadMessageNum` method of `TIMConversation`. For chat rooms, the server does not save the unread count. The unread count becomes zero after login and the unread count is synced with the server.

Prototype:

```
@interface TIMConversation : NSObject
- (int)getUnReadMessageNum;
@end
```

Example:

```
TIMConversation * conversation = [[TIMManager sharedInstance] getConversation:TIM_C2C receiver:@
"iOS_002"];
[conversation getUnReadMessageNum];
```

Read Reports

When the user reads messages in a conversation, a read report is sent, and the IM SDK sets all messages before the last read message as read.

Prototype:

```
@interface TIMConversation : NSObject
/**
* Configure read messages.
*
* @param readed The most recent read message in the conversation. Nil indicates reporting the lat
est messages.
*
* @param succ The success callback.
* @param fail The failure callback.
*
* @return 0 Successful
*/
- (int)setReadMessage:(TIMMessage*)readed succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Parameter description:

Parameter	Description
readed	**The last read message in the conversation. The IM SDK marks messages earlier than the last read message as read.**
succ	**The success callback**
fail	**The failure callback**

The following example sets all messages in a C2C (one-to-one) conversation as read.

Example:

```
TIMConversation * conversation = [[TIMManager sharedInstance] getConversation:TIM_C2C receiver:@
"iOS_002"];
[conversation setReadMessage:nil succ:nil fail:nil];
```

Disabling Auto Reports

In the event of single-client login, the default setting is sufficient. For better performance, the IM SDK pulls unread messages to local storage, and the server deletes the unread messages by default. Unread messages pulled on a previous client are invisible to clients switched to later. For a single client, the unread count is correct. If you want unread messages to appear on multiple clients, you can disable auto reports before initializing TIMManager. IM does not perform read reports for users. After auto reports is disabled, developers need to explicitly call `setReadMessage`.

```
@interface TIMUserConfig: NSObject
/**
* Disable auto reports (effective for loaded message extension package).
*/
@property(nonatomic,assign) BOOL disableAutoReport;
@end
```

Multi-client Read Synchronization

This feature is introduced on 2.0 and later versions. In the event of multi-client login, unread count synchronization notifications are delivered by the server. The IM SDK updates the unread count locally and tells the user to update conversations. This feature must be set before TIMManager login.

Prototype:

```
@protocol TIMRefreshListener <NSObject>
@optional
/**
* Refresh some conversations, including multi-client read reports synchronization.
*
* @param conversations The Conversation (TIMConversation*) list.
*/
- (void)onRefreshConversations:(NSArray*)conversations;
@end
@interface TIMUserConfig : NSObject
/**
** The conversation refresh listener for unread count and read synchronization (effective for loa
ded message extension package).
```

```
*/
@property(nonatomic,retain) id<TIMRefreshListener> refreshListener;
@end
```

Friend and User Profile
User Profiles and Relationship Chains (Android)

Last updated : 2020-12-30 11:37:12

IM provides the user profile hosting feature. App developers can use simple APIs to store the user profile. In addition, to facilitate the customization of different users' profiles, IM also provides custom fields for user profiles.

User Profiles

Obtaining your own profile

- **You can use the `getSelfProfile` method of `TIMFriendshipManager` to obtain your own profile that is stored on the server.**
- **You can use the `querySelfProfile` method of `TIMFriendshipManager` to obtain your own profile that is stored locally.**

Prototype:

```
/**
* Obtain your own profile that is stored on the server
* @param cb Callback. Parameters in the OnSuccess function return the corresponding {@see TIMUser
Profile} for users.
*/
public void getSelfProfile(final @NonNull TIMValueCallBack<TIMUserProfile> cb)

/**
* Obtains your own profile that is stored locally. If no profile is stored locally, null is retur
ned.
*
* @return TIMUserProfile
*/
public TIMUserProfile querySelfProfile()
```

`TIMUserProfile` provides the following APIs:

```
/**
* Obtain the user's identifier
```



```
     * @return User's identifier
     */
    public String getIdentifier()

    /**
     * Obtain the user's nickname
     * @return User's nickname
     */
    public String getNickName()

    /**
     * Obtain the user's profile photo URL
     * @return User's profile photo URL
     */
    public String getFaceUrl()

    /**
     * Obtain the user's personal signature
     * @return User's personal signature
     */
    public String getSelfSignature()

    /**
     * Obtain the user's friend request approval mode
     * @return User's friend request approval mode. See the constant in TIMFriendAllowType for more i
    nformation.
     */
    public String getAllowType()

    /**
     * Obtain the user's custom information
     * @return Custom information map
     */
    public Map<String, byte[]> getCustomInfo()

    /**
     * Obtain the user's custom information
     * @return Custom information map
     */
    public Map<String, Long> getCustomInfoUint()

    /**
     * Obtains the user's gender. See the constant definition in TIMFriendGenderType for more informa
    tion.
     * @return User's gender
     */
    public int getGender()
```

```
/**
* Obtain the user's birthday
* @return Birthday
*/
public long getBirthday()

/**
* Obtain the language
* @return Language
*/
public long getLanguage()

/**
* Obtain the location
* @return Location
*/
public String getLocation()
```

Example:

```
//Obtain your own profile that is stored on the server
TIMFriendshipManager.getInstance().getSelfProfile(new TIMValueCallBack<TIMUserProfile>(){
@Override
public void onError(int code, String desc){
//"code" (error code) and "desc" (error description) can be used to locate the cause of the reque
st failure.
//For the error code list, see the error code table.
Log.e(tag, "getSelfProfile failed: " + code + " desc");
}

@Override
public void onSuccess(TIMUserProfile result){
Log.e(tag, "getSelfProfile succ");
Log.e(tag, "identifier: " + result.getIdentifier() + " nickName: " + result.getNickName()
+ " allow: " + result.getAllowType());
}
});

//Obtain your own profile that is stored locally
TIMUserProfile selfProfile = TIMFriendshipManager.getInstance().querySelfProfile();
```

Obtaining a specified user's profile

You can use the `getUsersProfile` method of `TIMFriendshipManager` to obtain a friend's profile. This method can obtain the profile from either sources, the cache or the backend:

- **If `forceUpdate = true`, data is forcibly pulled from the backend, and the returned data will be cached.**
- **If `forceUpdate = false`, the system first searches for the data locally. If no local data can be found, it requests the data from the backend.**
  We recommend that data is forcibly pulled only for profile display to reduce the wait time.

You can use the `queryUserProfile` method of `TIMFriendshipManager` to obtain a friend's profile from the local cache based on the returned value. If none can be found, `null` is returned.

Prototype:

```
/**
* Obtain a specified friend's profile (excluding remarks and friend groups)
* @param users List of the identifiers of users whose profiles are to be obtained
* @param forceUpdate Forcibly pull data from the backend
* @param cb Callback. The parameters in the OnSuccess function return the corresponding user's
{@see TIMUserProfile} list.
*/
public void getUsersProfile(@NonNull List<String> users, boolean forceUpdate, @NonNull TIMValueCa
llBack<List<TIMUserProfile>> cb)

/**
* Obtains the local friend's profile (excluding remarks and friend groups). If none can be foun
d, 'null' is returned.
* @param identifier
* @return TIMUserProfile
*/
public TIMUserProfile queryUserProfile(String identifier)
```

Example:

```
//List of users whose profiles are to be obtained
List<String> users = new ArrayList<String>();
users.add("sample_user_1");
users.add("sample_user_2");

//Obtains user profiles
TIMFriendshipManager.getInstance().getUsersProfile(users, true, new TIMValueCallBack<List<TIMUser
Profile>>(){
@Override
public void onError(int code, String desc){
//"code" (error code) and "desc" (error description) can be used to locate the cause of the reque
st failure.
```

```
//For the error code list, see the error code table.
Log.e(tag, "getUsersProfile failed: " + code + " desc");
}

@Override
public void onSuccess(List<TIMUserProfile> result){
Log.e(tag, "getUsersProfile succ");
for(TIMUserProfile res : result){
Log.e(tag, "identifier: " + res.getIdentifier() + " nickName: " + res.getNickName());
}
}
});

//Obtain the locally cached user profile
TIMUserProfile userProfile = TIMFriendshipManager.getInstance().queryUserProfile("sample_user_1")
;
```

The caching time of the `getUsersProfile` API can be set through the `setExpiredSeconds` API of `TIMFriendProfileOption`. The default caching time is one day.

```
TIMUserConfig config = new TIMUserConfig();
TIMFriendProfileOption timFriendProfileOption = new TIMFriendProfileOption();
timFriendProfileOption.setExpiredSeconds(60 * 60); // 1 hour
config.setTIMFriendProfileOption(timFriendProfileOption);
TIMManager.getInstance().setUserConfig(config);
```

Modifying your own profile

You can use the `modifySelfProfile` method of `TIMFriendshipManager` to modify your own profile (such as your nickname, profile photo, and friend request approval mode).

Prototype:

```
/**
* Modify your own profile
* @param profileMap Fields to be modified are stored in hashMap, and the key values are the const
ants defined in TIMFriendshipManager:
* TIMFriendshipManager.TIM_PROFILE_TYPE_KEY_XXX
* @param cb Callback
*/
public void modifySelfProfile(@NonNull HashMap<String, Object> profileMap, @NonNull TIMCallBack c
b)
```

Through `profileMap`, you can set multiple fields at a time. For example, the code for simultaneously setting the nickname and gender is as follows:

```
HashMap<String, Object> profileMap = new HashMap<>();
profileMap.put(TIMUserProfile.TIM_PROFILE_TYPE_KEY_NICK, "My nickname");
profileMap.put(TIMUserProfile.TIM_PROFILE_TYPE_KEY_GENDER, TIMFriendGenderType.GENDER_MALE);
profileMap.put(TIMUserProfile.TIM_PROFILE_TYPE_KEY_BIRTHDAY, 20190419);
TIMFriendshipManager.getInstance().modifySelfProfile(profileMap, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "modifySelfProfile failed: " + code + " desc" + desc);
}

@Override
public void onSuccess() {
Log.e(tag, "modifySelfProfile success");
}
});
```

Setting a key value that does not exist may lead to failure. Some common key values are defined in `TIMUserProfile`:

Key	Value	Description
`TIM_PROFILE_TYPE_KEY_NICK`	**String**	**Nickname**
`TIM_PROFILE_TYPE_KEY_FACEURL`	**String**	**Profile photo**
`TIM_PROFILE_TYPE_KEY_ALLOWTYPE`	**String**	**Friend request**
`TIM_PROFILE_TYPE_KEY_GENDER`	**int**	**Gender**
`TIM_PROFILE_TYPE_KEY_BIRTHDAY`	**int**	**Birthday**
`TIM_PROFILE_TYPE_KEY_LOCATION`	**String**	**Location**
`TIM_PROFILE_TYPE_KEY_LANGUAGE`	**int**	**Language**
`TIM_PROFILE_TYPE_KEY_LEVEL`	**int**	**Level**
`TIM_PROFILE_TYPE_KEY_ROLE`	**int**	**Role**
`TIM_PROFILE_TYPE_KEY_SELFSIGNATURE`	**String**	**Signature**
`TIM_PROFILE_TYPE_KEY_CUSTOM_PREFIX`	**String, int**	**Prefix of the custom field**

For custom fields, you need to add prefixes. For example, to set the `Blood` custom field of the integer type on the backend, use the following code:

```
HashMap<String, Object> profileMap = new HashMap<>();
profileMap.put(TIMUserProfile.TIM_PROFILE_TYPE_KEY_CUSTOM_PREFIX + "Blood", 1);
TIMFriendshipManager.getInstance().modifySelfProfile(profileMap, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "modifySelfProfile failed: " + code + " desc" + desc);
}

@Override
public void onSuccess() {
Log.e(tag, "modifySelfProfile success");
}
});
```

Friend Relationships

Obtaining all friends

You can use the `getFriendList` method of `TIMFriendshipManager` to obtain the list of all friends.

```
/**
* Obtain the friend list
* @param cb Callback for the TIMFriend list
*/
public void getFriendList(@NonNull TIMValueCallBack<List<TIMFriend>> cb)
```

The obtained friend list is returned, and friend objects are stored in `TIMFriend`. `TIMFriend` is defined as follows:

```
/**
* Obtain the user's identifier
* @return User's identifier
*/
public String getIdentifier()

/**
* Obtain the friend's remarks
* @return Friend's remarks
*/
public String getRemark()

/**
```

```
* Obtain the reason of the friend request
* @return Reason of the friend request
*/
public String getAddWording()

/**
* Obtain the source of the friend request
* @return Source of the friend request
*/
public String getAddSource()

/**
* Obtain the friend group name
* @return List of friend group names
*/
public List<String> getGroupNames()

/**
* Obtains the friend's custom information. The key value is based on the character string config
ured on the backend, excluding the TIM_FRIEND_PROFILE_TYPE_KEY_CUSTOM_PREFIX prefix.
* @return Custom information map
*/
public Map<String, byte[]> getCustomInfo()

/**
* Obtains the friend's custom information of the uint type. The key value is based on the charac
ter string configured on the backend, excluding the
* TIM_FRIEND_PROFILE_TYPE_KEY_CUSTOM_PREFIX prefix.
* @return Custom information map
*/
public Map<String, byte[]> getCustomInfoUint()

/**
* Obtain the friend's profile
* @return User profile
*/
public TIMUserProfile getTimUserProfile()()
```

Sample code

```
TIMFriendshipManager.getInstance().getFriendList(new TIMValueCallBack<List<TIMFriend>>() {
@Override
public void onError(int code, String desc) {
QLog.e(TAG, "getFriendList err code = " + code);
}

@Override
```

```
public void onSuccess(List<TIMFriend> timFriends) {
StringBuilder stringBuilder = new StringBuilder();
for (TIMFriend timFriend : timFriends){
stringBuilder.append(timFriend.toString());
}
QLog.i(TAG, "getFriendList success result = " + stringBuilder.toString());
}
});
```

Modifying friends

The `modifyFriend` method can be called to modify friends. Similar to modifying your own profile, multiple fields can be updated at a time.

```
/**
* Modify a friend's profile
* @param identifier Friend's identifier
* @param profileMap Fields to be modified. See TIM_FRIEND_PROFILE_TYPE_KEY_XXX in TIMFriend for m
ore information.
* @param cb Callback
*/
public void modifyFriend(@NonNull String identifier, @NonNull HashMap<String, Object> profileMap,
@NonNull TIMCallBack cb)
```

Setting a key value that does not exist may lead to failure. The backend defines some common key values.

Key	Value	Description
TIM_FRIEND_PROFILE_TYPE_KEY_REMARK	**String**	**Remarks**
TIM_FRIEND_PROFILE_TYPE_KEY_GROUP	**List< String >**	**Friend group**
TIM_FRIEND_PROFILE_TYPE_KEY_CUSTOM_PREFIX	**String, int**	**Prefix of the custom field**

Example: set remarks for the friend [Android_002] to [002 remark]

```
String identifier = "Android_002";
HashMap<String, Object> hashMap = new HashMap<>();
hashMap.put(TIMFriend.TIM_FRIEND_PROFILE_TYPE_KEY_REMARK, "002 remark");
TIMFriendshipManager.getInstance().modifyFriend(identifier, hashMap, new TIMCallBack() {
@Override
public void onError(int i, String s) {
```

```
Log.e(TAG, "modifyFriend err code = " + i + ", desc = " + s);
}

@Override
public void onSuccess() {
Log.i(TAG, "modifyFriend success");
}
});
```

Note :

To modify a friend's custom information, you must first configure the relationship chain custom fields on the server.

Adding friends

You can use the `addFriend` method of `TIMFriendshipManager` to add friends.

```
/**
* Add a friend
* @param timFriendRequest Friend request
* @param cb Callback
*/
public void addFriend(@NonNull TIMFriendRequest timFriendRequest, @NonNull TIMValueCallBack<TIMFr
iendResult> cb)
```

To add a friend, the request parameter needs to be passed in. Its parameter type is defined as follows:

```
/**
* User's identifier
*/
private String identifier = "";

/**
* User's remarks (a maximum of 96 bytes)
*/
private String remark = "";

/**
* Request description (a maximum of 120 bytes)
*/
private String addWording = "";

/**
```

```
* Friend addition source
* The source cannot exceed 8 bytes and must be prefixed with "AddSource_Type_".
*/
private String addSource = "";

/**
* Group name
*/
private String friendGroup = "";
```

The success callback returns the `TIMFriendResult` result for the operating user. Developers can notify the user accordingly. The return code for adding friends is as follows:

```
public class TIMFriendStatus {
/**
* Operation succeeded
*/
public static final int TIM_FRIEND_STATUS_SUCC = 0;

/**
* The request parameter is incorrect. Check whether the request is correct based on the error des
cription.
*/
public static final int TIM_FRIEND_PARAM_INVALID = 30001;

/**
* Valid for adding friends and responding to friends: your number of friends has reached the limi
t of the system.
*/
public static final int TIM_ADD_FRIEND_STATUS_SELF_FRIEND_FULL = 30010;

/**
* Valid for adding friends and responding to friends: the peer's number of friends has reached t
he limit of the system.
*/
public static final int TIM_ADD_FRIEND_STATUS_THEIR_FRIEND_FULL = 30014;

/**
* Valid for adding friends: the peer is already in your blacklist.
*/
public static final int TIM_ADD_FRIEND_STATUS_IN_SELF_BLACK_LIST = 30515;

/**
* Valid for adding friends: the peer has forbidden friend requests.
*/
public static final int TIM_ADD_FRIEND_STATUS_FRIEND_SIDE_FORBID_ADD = 30516;
```

```
/**
* Valid for adding friends: the peer has blacklisted you.
*/
public static final int TIM_ADD_FRIEND_STATUS_IN_OTHER_SIDE_BLACK_LIST = 30525;
/**
* Valid for adding friends: the friend request is pending approval.
*/
public static final int TIM_ADD_FRIEND_STATUS_PENDING = 30539;
};
```

Sample code

```
TIMFriendRequest timFriendRequest = new TIMFriendRequest("test_id");
timFriendRequest.setAddWording("it's me!");
timFriendRequest.setAddSource("android");
TIMFriendshipManager.getInstance().addFriend(timFriendRequest, new TIMValueCallBack<TIMFriendResu
lt>() {
@Override
public void onError(int i, String s) {
QLog.e(TAG, "addFriend err code = " + i + ", desc = " + s);
}

@Override
public void onSuccess(TIMFriendResult timFriendResult) {
QLog.i(TAG, "addFriend success result = " + timFriendResult.toString());
}
});
```

Deleting friends

You can use the `deleteFriends` method of `TIMFriendshipManager` to delete friends in batches.

```
/**
* Delete friends
* @param identifiers Friend list
* @param delFriendType Deletion type
* @param cb Callback
*/
public void deleteFriends(@NonNull List<String> identifiers, @NonNull int delFriendType, @NonNull
TIMValueCallBack<List<TIMFriendResult>> cb)
```

The success callback returns the `TIMFriendResult` result for the operating user. Developers can notify the user accordingly. The error codes for deleting friends are as follows:

```
public class TIMFriendStatus {
/**
```

```
* Operation succeeded
*/
public static final int TIM_FRIEND_STATUS_SUCC = 0;
/**
* Valid for deleting friends: the friend that you want to delete is not your friend.
*/
public static final int TIM_DEL_FRIEND_STATUS_NO_FRIEND = 31704;
};
```

Sample code

```
List<String> identifiers = new ArrayList<>();
identifiers.add("test_id");
TIMFriendshipManager.getInstance().deleteFriends(identifiers, TIMDelFriendType.TIM_FRIEND_DEL_SIN
GLE, new TIMValueCallBack<List<TIMFriendResult>>() {
@Override
public void onError(int i, String s) {
QLog.e(TAG, "deleteFriends err code = " + i + ", desc = " + s);
}

@Override
public void onSuccess(List<TIMFriendResult> timUserProfiles) {
QLog.i(TAG, "deleteFriends success");
}
});
```

Approving or rejecting friend requests

You can use the `doResponse` method of `TIMFriendshipManager` to approve or reject friend requests.

```
/**
* Handle a friend request
* @param response Request parameter, including the friend ID, prior remarks, and response type
* @param cb
*/
public void doResponse(TIMFriendResponse response, @NonNull TIMValueCallBack<TIMFriendResult> cb)
```

The `response` parameter is defined as follows:

```
public class TIMFriendResponse {
/**
* Approve the friend request (establish a one-way friendship)
*/
public static final int TIM_FRIEND_RESPONSE_AGREE = 0;
```

```
/**
* Approve the friend request and add the friend (establish a two-way friendship)
*/
public static final int TIM_FRIEND_RESPONSE_AGREE_AND_ADD = 1;

/**
* Reject the friend request
*/
public static final int TIM_FRIEND_RESPONSE_REJECT = 2;

/**
* Response type
*/
private int responseType = TIM_FRIEND_RESPONSE_AGREE;

/**
* Friend ID for response
*/
private String identifier = ""; // Friend ID for response

/**
* Friend remarks (optional). You can add remarks if you want to add the user as a friend. The max
imum length of the remarks is 96 bytes.
*/
private String remark = "";

.......The get and set methods are omitted here.
}
```

The success callback returns the `TIMFriendResult` result for the operating user. The error codes for handling friend requests are as follows:

```
public class TIMFriendStatus {
/**
* Operation succeeded
*/
public static final int TIM_FRIEND_STATUS_SUCC = 0;

/**
* Valid for adding friends and responding to friends: your number of friends has reached the limi
t of the system.
*/
public static final int TIM_ADD_FRIEND_STATUS_SELF_FRIEND_FULL = 30010;

/**
* Valid for adding friends and responding to friends: the user's number of friends has reached t
he limit of the system.
```

```
*/
public static final int TIM_ADD_FRIEND_STATUS_THEIR_FRIEND_FULL = 30014;

/**
* Valid for responding to friend requests: the peer has not initiated a friend request.
*/
public static final int TIM_RESPONSE_FRIEND_STATUS_NO_REQ = 30614;
};
```

Verifying friend relationships

You can use the `checkFriends` method of `TIMFriendshipManager` to verify friend relationships.

```
/**
* Verify friends
* @param checkInfo Friend verification parameter
* @param cb Callback
*/
public void checkFriends(@NonNull TIMFriendCheckInfo checkInfo, @NonNull TIMValueCallBack<List<TI
MCheckFriendResult>> cb)
```

The `checkInfo` parameter is defined as follows:

```
public class TIMFriendCheckInfo {
private List<String> users = new ArrayList<>();
private int checkType = TIMFriendCheckType.TIM_FRIEND_CHECK_TYPE_UNIDIRECTION;

/**
* Set the ID of the friend to be checked
*
* @param users
*/
public void setUsers(List<String> users);

/**
* Sets the type of the relationship to be checked. See the constant defined in TIMFriendCheckType
for more information.
*
* @param type
*/
public void setCheckType(int type);
}
```

The `TIMFriendCheckType` parameter is defined as follows:

```
public class TIMFriendCheckType {
/**
* One-way friendship
*/
public static final int TIM_FRIEND_CHECK_TYPE_UNIDIRECTION = 1;

/**
* Two-way friendship
*/
public static final int TIM_FRIEND_CHECK_TYPE_BIDIRECTION = 2;
}
```

The success callback returns the `TIMCheckFriendResult` list for the operating user. This parameter is defined as follows:

```
public class TIMCheckFriendResult {
private String identifier = "";
private int resultCode;
private String resultInfo = "";
private int resultType;

/**
* Obtain the friend ID
*
* @return Friend ID
*/
public String getIdentifier();

/**
* Obtain the return code
*
* @return Return code
*/
public int getResultCode();

/**
* Obtain the returned result description
*
* @return Result description
*/
public String getResultInfo();

/**
* Obtains the friend relationship type to be checked. See the constant defined in TIMFriendRelati
onType for more information.
*
```

```
 * @return Friend relationship type
 */
public int getResultType();
}
```

The `TIMFriendRelationType` parameter is defined as follows:

```
public class TIMFriendRelationType {
/**
* You are not friends.
*/
public static final int TIM_FRIEND_RELATION_TYPE_NONE = 0;

/**
* The peer is in my friend list.
*/
public static final int TIM_FRIEND_RELATION_TYPE_MY_UNI = 1;

/**
* I am in the peer's friend list.
*/
public static final int TIM_FRIEND_RELATION_TYPE_OTHER_UNI = 2;

/**
* You are friends for each other.
*/
public static final int TIM_FRIEND_RELATION_TYPE_BOTH_WAY = 3;

}
```

Pending Friend Requests

Obtaining the pending request list

When another user uses the `addFriend` method to request to add you as a friend, a pending record is added on the backend. When you request to add another user as a friend, a pending record is also added on the backend. You can use the `getPendencyList` method to obtain the pending request list.

```
/**
* Obtain the pending request list
*
* @param timFriendPendencyRequest
* @param cb
```

```
*/
public void getPendencyList(TIMFriendPendencyRequest timFriendPendencyRequest, @NonNull TIMValueC
allBack<TIMFriendPendencyResponse> cb)
```

As the backend may store multiple pending friend requests that go beyond the display scope of the interface, the API allows you to page up or down to browse all requests. In this case, the `timFriendPendencyRequest` parameter needs to be passed in. The definition of the parameter is as follows:

```
/**
* Sequence number of the pending request list. We recommend that you save the sequence number and
pending request list on the client. Enter the sequence number returned by the server when sending
the request. If the sequence number is the latest on the server, no data is returned.
*
* @param seq Sequence number
*/
public void setSeq(long seq)

/**
* Paging timestamp, which is only used for paging up or down. If the server returns 0, it indicat
es that no more data is available. Enter 0 for the first request.
* Note that if the sequence number returned by the server is different from the entered sequence
number, use the original seq request on the client for paging up or down. The local seq is not up
dated until the data request is completed.
*
* @param timestamp Paging timestamp
*/
public void setTimestamp(long timestamp)

/**
* Number of requests per page, which is valid for requests
*
* @param numPerPage Number of requests per page
*/
public void setNumPerPage(int numPerPage)

/**
* Fetch type of pending requests. See the constant defined in TIMPendencyType for more informatio
n.
*
* @param timPendencyType Fetch type of pending requests
*/
public void setTimPendencyGetType(int timPendencyType)
```

After the operation succeeds, the callback returns the paging information and pending records `TIMFriendPendencyResponse` .

```
/**
* Obtain the sequence number of the pending request list for the current request
* @return Sequence number
*/
public long getSeq()

/**
* Obtain the paging timestamp for the current request
*
* @return Timestamp
*/
public long getTimestamp()

/**
* Obtain the number of unread pending requests
*
* @return Number of unread pending requests
*/
public long getUnreadCnt()

/**
* Obtain the pending information list
*
* @return Information list
*/
public List<TIMFriendPendencyItem> getItems()
```

`TIMFriendPendencyItem` is defined as follows:

```
/**
* Obtain the user's ID
*
* @return id
*/
public String getIdentifier()

/**
* Obtain the addition time
*
* @return Time
*/
public long getAddTime()

/**
* Obtain the source
*
```

```
* @return Source
*/
public String getAddSource()

/**
* Obtain friend request remarks
*
* @return Friend request remarks
*/
public String getAddWording()

/**
* Obtain the friend's nickname
*
* @return Nickname
*/
public String getNickname()

/**
* Obtains the pending request type. See the constant defined in TIMPendencyType for more information.
*
* @return Pending request type
*/
public int getType()
```

`TIMPendencyType` **is defined as follows:**

```
public class TIMPendencyType {
/**
* Pending requests from other users
*/
public static final int TIM_PENDENCY_COME_IN = 1;

/**
* Pending requests to other users
*/
public static final int TIM_PENDENCY_SEND_OUT = 2;

/**
* Pending requests from other users and pending requests to other users, which are valid only during fetching
*/
public static final int TIM_PENDENCY_BOTH = 3;

}
```

Deleting pending requests

```
/**
* Delete pending requests
*
* @param pendencyType Pending request type. See TIMPendencyType for more information. Only pendin
g requests of the TIM_PENDENCY_COME_IN and TIM_PENDENCY_SEND_OUT types can be deleted.
* @param users ID list of pending users to be deleted
* @param cb Callback
*/
public void deletePendency(int pendencyType, List<String> users, @NonNull TIMValueCallBack<List<T
IMFriendResult>> cb)
```

Reporting that a pending record is read

When a user fetches pending records, they can be set as read on the backend.

```
/**
* Report that a pending record is read
*
* @param timestamp Read timestamp. All messages prior to this timestamp are set to read.
* @param cb Callback
*/
public void pendencyReport(long timestamp, @NonNull TIMCallBack cb)
```

After reporting, the returned count of unread records is changed when `getPendencyList` is called again.

Blacklist

Adding users to the blacklist

You can blacklist any user. If the user is your friend, after you blacklist the user, your friend relationship with the user is canceled and you cannot no longer receive any messages from the user.

```
/**
* Add a user to the blacklist
*
* @param users User list
* @param cb Callback
*/
public void addBlackList(List<String> users, @NonNull TIMValueCallBack<List<TIMFriendResult>> cb)
```

Deleting a user from the blacklist

```
/**
* Delete a user from the blacklist
*
* @param users User list
* @param cb Callback
*/
public void deleteBlackList(List<String> users, @NonNull TIMValueCallBack<List<TIMFriendResult>>
cb)
```

Obtaining the blacklist

```
/**
* Obtain the blacklist
*
* @param cb Callback
*/
public void getBlackList(@NonNull TIMValueCallBack<List<TIMFriend>> cb)
```

Friend List

Creating a friend list

When creating a list, you can select users to be added to the list. A user can be added to multiple lists.

```
/**
* Create a friend group
*
* @param groupNames List of friend group names. The friend group to be created must be a new grou
p that does not already exist.
* @param identifiers Friends to be added to the friend group
* @param cb Callback
*/
public void createFriendGroup(List<String> groupNames, List<String> identifiers, @NonNull TIMValu
eCallBack<List<TIMFriendResult>> cb)
```

Deleting a friend group

```
/**
* Delete a friend group
```

```
*
* @param groupNames Name list of friend groups to be deleted
* @param cb Callback
*/
public void deleteFriendGroup(List<String> groupNames, @NonNull TIMCallBack cb)
```

Adding friends to a friend group

```
/**
* Add friends to a friend group
*
* @param groupName Friend group name
* @param identifiers List of friends to be added to the friend group
* @param cb Callback
*/
public void addFriendsToFriendGroup(String groupName, List<String> identifiers, @NonNull TIMValue
CallBack<List<TIMFriendResult>> cb)
```

Deleting friends from a friend group

```
/**
* Delete friends from a friend group
*
* @param groupName Friend group name
* @param identifiers Friends to be deleted from the friend group
* @param cb Callback
*/
public void deleteFriendsFromFriendGroup(String groupName, List<String> identifiers, @NonNull TIM
ValueCallBack<List<TIMFriendResult>> cb)
```

Renaming a friend group

```
/**
* Rename a friend group
*
* @param oldName Original name of the friend group
* @param newName New name of the friend group
* @param cb Callback
*/
public void renameFriendGroup(String oldName, String newName, @NonNull TIMCallBack cb)
```

Obtaining a friend group

```
/**
* Obtains a specified friend group. If 'null' is passed in, all friend groups are obtained.
* @param groupNames Name list of friend groups to be obtained
* @param cb Callback
*/
public void getFriendGroups(List<String> groupNames, @NonNull TIMValueCallBack<List<TIMFriendGroup>> cb)
```

System Notifications for Relationship Chain Changes

In `TIMMessage`, `Elem` = `TIMSNSSystemElem` indicates a system notification for a relationship chain change.

```
/**
* Message elements synchronized through backend push after relevant operations on relationship chains are completed
*
*/
public class TIMSNSSystemElem extends TIMElem {
private int subType = 0;

// subType corresponds to TIMSNSSystemType.TIM_SNS_SYSTEM_ADD_FRIEND.
private List<String> requestAddFriendUserList = new ArrayList<>();

// subType corresponds to TIMSNSSystemType.TIM_SNS_SYSTEM_DEL_FRIEND.
private List<String> delRequestAddFriendUserList = new ArrayList<>();

// subType corresponds to TIMSNSSystemType.TIM_SNS_SYSTEM_ADD_BLACKLIST.
private List<String> addBlacklistUserList = new ArrayList<>();

// subType corresponds to TIMSNSSystemType.TIM_SNS_SYSTEM_DEL_BLACKLIST.
private List<String> delBlacklistUserList = new ArrayList<>();

// subType corresponds to TIMSNSSystemType.TIM_SNS_SYSTEM_ADD_FRIEND_REQ.
private List<TIMFriendPendencyInfo> friendAddPendencyList = new ArrayList<>();

// subType corresponds to TIMSNSSystemType.TIM_SNS_SYSTEM_DEL_FRIEND_REQ.
private List<String> delFriendAddPendencyList = new ArrayList<>();

// subType corresponds to TIMSNSSystemType.TIM_SNS_SYSTEM_SNS_PROFILE_CHANGE.
private List<TIMSNSChangeInfo> changeInfoList = new ArrayList<>();

public TIMSNSSystemElem() { type = TIMElemType.SNSTips; }
public int getSubType();
```

```
public List<String> getRequestAddFriendUserList();
public List<String> getDelRequestAddFriendUserList();
public List<String> getAddBlacklistUserList();
public List<String> getDelBlacklistUserList();
public List<TIMFriendPendencyInfo> getFriendAddPendencyList();
public List<String> getDelFriendAddPendencyList();
public List<TIMSNSChangeInfo> getChangeInfoList();
}


/**
* System notification type for the relationship chain change
*/
public class TIMSNSSystemType {
/**
* Friend addition message
*/
public static final int TIM_SNS_SYSTEM_ADD_FRIEND = 0x01;

/**
* Friend deletion message
*/
public static final int TIM_SNS_SYSTEM_DEL_FRIEND = 0x02;

/**
* Add friend requests
*/
public static final int TIM_SNS_SYSTEM_ADD_FRIEND_REQ = 0x03;

/**
* Delete pending requests
*/
public static final int TIM_SNS_SYSTEM_DEL_FRIEND_REQ = 0x04;

/**
* Add users to the blacklist
*/
public static final int TIM_SNS_SYSTEM_ADD_BLACKLIST = 0x05;

/**
* Delete users from the blacklist
*/
public static final int TIM_SNS_SYSTEM_DEL_BLACKLIST = 0x06;

/**
* Report that a pending record is read
*/
public static final int TIM_SNS_SYSTEM_PENDENCY_REPORT = 0x07;
```

```
/**
* Relationship chain information is changed
*/
public static final int TIM_SNS_SYSTEM_SNS_PROFILE_CHANGE = 0x08;
};

/**
* Details of the relationship chain change
*
*/
public class TIMSNSChangeInfo {
/**
* User ID for the profile change
*/
private String updateUser = "";

/**
* Profile change information
*/
private Map<String, Object> itemMap = new HashMap<>();

public String getUpdateUser() {
return updateUser;
}

public Map<String, Object> getItemMap() {
return itemMap;
}
}
```

System notifications for adding friends

When two users become friends, both of them receive a system message indicating that they have been added as friends.

Triggering time:

When your relationship chain changes by adding a friend, you will receive a message (if the friend is already a one-way friend, the party whose relationship chain does not change will not receive the message.)

Parameter description:

Parameter	Description
subType	**TIM_SNS_SYSTEM_ADD_FRIEND**

requestAddFriendUserList	**List of users who become friends**

System notifications for deleting friends

When two users unfriend each other, they both receive a system message indicating that their friendship is canceled:

Triggering time:

When your relationship chain changes by deleting a friend, you will receive a message (if the deleted friend is a one-way friend, the party whose relationship chain does not change will not receive the message.)

Parameter description:

Parameter	Description
subType	**TIM_SNS_SYSTEM_DEL_FRIEND**
delRequestAddFriendUserList	**List of deleted friends**

System notifications for friend requests

When you send a friend request to a user and the user requires approval, both of you will receive a friend request system notification.

Triggering time:

When you send a friend request to a user and the user requires approval, both of you will receive a system notification for friend requests. The user can choose to approve or reject the request, whereas you cannot perform any operations. This is only for the purpose of information synchronization.

Parameter description:

Parameter	Description
subType	**TIM_SNS_SYSTEM_ADD_FRIEND_REQ**
friendAddPendencyList	**List of pending friend requests**

`TIMFriendPendencyInfo` **parameter description:**

Parameter	Description
fromUser	**Friend addition operator**

addSource	**Friend addition source**
fromUserNickName	**Nickname of the friend addition operator**
addWording	**Postscript to the friend request**

Notifications for deleting pending requests

Triggering time:

After your friend request to a user is approved or rejected, you will receive a message indicating that the pending request was deleted.

Parameter description:

Parameter	Description
subType	**TIM_SNS_SYSTEM_DEL_FRIEND_REQ**
delFriendAddPendencyList	**List of approved or rejected friend requests**

System Notifications for User Profile Changes

In `TIMMessage`, `Elem` = `TIMProfileSystemElem` indicates a system message for a user profile change.

```
/**
* Message element synchronized by backend push after the modification of your own and the frien
d’s profiles
*/
public class TIMProfileSystemElem extends TIMElem {
private int subType; //Profile modification type: TIMProfileSystemType
private String fromUser; //Profile modification source (the modifier)
private Map<String, Object> itemMap; //User’s profile

public int getSubType();
public String getFromUser();
public Map<String, Object> getItemMap();
}

/**
* System notification type for the user profile change
*/
public class TIMProfileSystemType {
```

```
/**
* Invalid value
*/
public static final int INVALID = 0;

/**
* Friend profile change
*/
public static final int TIM_PROFILE_SYSTEM_FRIEND_PROFILE_CHANGE = 1;
}
```

When your profile or a friend's profile is modified, you will receive a system notification message indicating that the user profile is modified. For example, if a friend changes his or her profile photo, then in `TIMProfileSystemElem`, `key` in `itemMap` will be `Tag_Profile_IM_Image`, and `value` will be the `url` of the profile photo. The constant value of `key` is defined in `TIMUserProfile`.

User Profiles and Relationship Chains (iOS)

Last updated : 2020-12-30 10:12:23

Instant Messaging (IM) provides the relationship chain and user profile hosting features. App developers can use simple APIs to store relationship chains and user profiles. In addition, to allow users conveniently customize profiles, IM also provides custom fields for user profiles and relationship chains. These fields must be configured in the console in advance. For more information, see User Custom Fields. The APIs described in this document are valid for both independent accounts and hosted accounts.

User Profile

Obtaining your own profile

You can use the `getSelfProfile` method of `TIMFriendshipManager` to obtain your own profile.

```
/**
* Obtaining your own profile
*
* @param succ Success callback. TIMUserProfile is returned.
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)getSelfProfile:(TIMGetProfileSucc)succ fail:(TIMFail)fail;
```

If the profile is obtained successfully, the succ callback returns the obtained `TIMUserProfile` object. `TIMUserProfile` is defined as follows:

```
/**
* User's profile
*/
@interface TIMUserProfile : TIMCodingModel

/**
* User' s identifier
*/
@property(nonatomic,strong) NSString* identifier;

/**
```

```
* User’s nickname
*/
@property(nonatomic,strong) NSString* nickname;

/**
* Friend request approval mode
*/
@property(nonatomic,assign) TIMFriendAllowType allowType;

/**
* User’s profile photo
*/
@property(nonatomic,strong) NSString* faceURL;

/**
* User’s signature
*/
@property(nonatomic,strong) NSData* selfSignature;

/**
* User’s gender
*/
@property(nonatomic,assign) TIMGender gender;

/**
* User’s birthday
*/
@property(nonatomic,assign) uint32_t birthday;

/**
* User’s location
*/
@property(nonatomic,strong) NSData* location;

/**
* User’s language
*/
@property(nonatomic,assign) uint32_t language;

/**
* Level
*/
@property(nonatomic,assign) uint32_t level;

/**
* Role
*/
@property(nonatomic,assign) uint32_t role;
```

```
/**
* A set of custom fields. Key is of the NSString type, and value is of the NSData type or NSNumbe
r type.
* (The key value is determined based on the character string configured on the backend.)
*/
@property(nonatomic,strong) NSDictionary* customInfo;

@end
```

Obtaining the profile of a specified user

You can use the `getUsersProfile` method of `TIMFriendshipManager` to obtain the profile of a specified user. This method supports obtaining the profile from two sources: the cache and the backend. If forceUpdate = YES, data is fetched forcibly from the backend, and the returned data is cached. If forceUpdate = NO, the system first searches for the data locally. If no data is found locally, it requests data from the backend. We recommend that data be forcibly fetched only for profile display, thus reducing the wait time.

```
/**
* Obtain the profile of a specified friend
*
* @param users User ID
* @prarm forceUpdate Forcibly fetches data from the backend
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)getUsersProfile:(NSArray<NSString *> *)users forceUpdate:(BOOL)forceUpdate succ:(TIMGetPro
fileArraySucc)succ fail:(TIMFail)fail;
@end
```

Example: obtain the profiles of user [iOS_002] and user [iOS_003]

```
NSMutableArray * arr = [[NSMutableArray alloc] init];
[arr addObject:@"iOS_002"];
[arr addObject:@"iOS_003"];
[[TIMFriendshipManager sharedInstance] getUsersProfile:arr forceUpdate:NO succ:^(NSArray * arr) {
for (TIMUserProfile * profile in arr) {
NSLog(@"user=%@", profile);
}
}fail:^(int code, NSString * err) {
NSLog(@"GetFriendsProfile fail: code=%d err=%@", code, err);
}];
```

In this example, as forceUpdate is set to No, the system first searches for the two users' profiles in the cache, thus reducing the wait time. The cache time is specified by TIMFriendProfileOption, and is 1 day by default.

```
/**
* Profiles and relationship chains
*/
@interface TIMFriendProfileOption : NSObject

/**
* Maximum cache time for relationship chains
* The default cache time is 1 day. If the time used for obtaining profiles and relationship chain
s exceeds the cache time, the system automatically sends a request to the server.
*/
@property NSInteger expiredSeconds;

@end
```

The method for configuring the expiry time is `-[TIMManager setUserConfig:]`. Sample code:

```
TIMUserConfig *config = ...;
TIMFriendProfileOption *option = [TIMFriendProfileOption new];
option.expiredSeconds = 60*60; // 1 hour
config.friendProfileOpt = option;
[[TIMManager sharedInstance] setUserConfig:config];
```

Modifying your own profile

You can use the `modifySelfProfile` method to modify your own profile.

```
@interface TIMFriendshipManager : NSObject
/**
* Set your own profile
*
* @param values Attributes to be updated. Multiple fields can be updated at a time.
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)modifySelfProfile:(NSDictionary<NSString *, id> *)values succ:(TIMSucc)succ fail:(TIMFail)
fail;
@end
```

Through the values dictionary, you can set multiple fields at a time. For example, the code for setting the nickname is as follows:

```
[[TIMFriendshipManager sharedInstance] modifySelfProfile:@{TIMProfileTypeKey_Nick:@"My nickname"}
succ:nil fail:nil];
```

Setting a non-existing key value may lead to a failure. The backend defines some common key values.

Key	Value	Description
TIMProfileTypeKey_Nick	**NSString**	**Nickname**
TIMProfileTypeKey_FaceUrl	**NSString**	**Profile photo**
TIMProfileTypeKey_AllowType	**NSNumber**	**Friend request**
TIMProfileTypeKey_Gender	**NSNumber**	**Gender**
TIMProfileTypeKey_Birthday	**NSNumber**	**Birthday**
TIMProfileTypeKey_Location	**NSString**	**Location**
TIMProfileTypeKey_Language	**NSNumber**	**Language**
TIMProfileTypeKey_Level	**NSNumber**	**Level**
TIMProfileTypeKey_Role	**NSNumber**	**Role**
TIMProfileTypeKey_SelfSignature	**NSString**	**Signature**
TIMProfileTypeKey_Custom_Prefix	**NSString, NSData, or NSNumber**	**Custom field prefix**

For custom fields, you need to add our prefixes. For example, if you want to set a custom field `Blood` of the integer type on the backend, the code is as follows:

```
NSString *key = [TIMProfileTypeKey_Custom_Prefix stringByAppendingString:@"Blood"];
[[TIMFriendshipManager sharedInstance] modifySelfProfile:@{key:@1} succ:nil fail:nil];
```

Note :

If the custom field value is set to an NSString object, the backend converts it into a UTF8 object and saves it in the database. Because some migrated user profiles may

not use the UTF8 format, the backend always returns the NSData type of the profile for profile requests.

Friend Relationships

Obtaining all friends

You can use the `getFriendList` method of `TIMFriendshipManager` to obtain the list of all friends.

```
@interface TIMFriendshipManager : NSObject
/**
* Obtain the friend list
*
* @param succ Success callback. The friend (TIMFriend) list was returned.
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
-(int)getFriendList:(TIMFriendArraySucc)succ fail:(TIMFail)fail;
@end
```

If the friend list is obtained successfully, the succ callback returns the friend list, and friend objects are stored in `TIMFriend`. `TIMFriend` is defined as follows:

```
@interface TIMFriend : TIMCodingModel

/**
* Friend identifier
*/
@property(nonatomic,strong) NSString *identifier;

/**
* Friend remark
*/
@property(nonatomic,strong) NSString *remark;

/**
* Friend list name NSString* list
*/
@property(nonatomic,strong) NSArray *groups;

/**
* Request reason
```

```
*/
@property(nonatomic,strong) NSString * addWording;

/**
* Request source
*/
@property(nonatomic,strong) NSString * addSource;

/**
* Creation time
*/
@property(nonatomic,assign) uint64_t addTime;

/**
* A set of custom fields. `key` is of the NSString type, and `value` is of the NSData type or NSN
umber type. The key value is determined based on the character string configured on the backend.
*/
@property(nonatomic,strong) NSDictionary* customInfo;

/**
* Friend profile
*/
@property(nonatomic,strong) TIMUserProfile *profile;

@end
```

Sample code

```
[[TIMFriendshipManager sharedInstance] getFriendList:^(NSArray<TIMFriend *> *friends) {
NSMutableString *msg = [NSMutableString new];
[msg appendString:@"Friend list: "];
for (TIMFriend *friend in friends) {
[msg appendFormat:@"[%@,%@,%d,%@,%@,%@]", friend.identifier, friend.remark, friend.addTime, frien
d.addSource, friend.addWording, friend.groups];
}
self.msgLabel.text = msg;
} fail:^(int code, NSString *msg) {
self.msgLabel.text = [NSString stringWithFormat:@"Failed: %d, %@", code, msg];
}];
```

Modifying a friend

You can use the `modifyFriend` method to modify a friend's profile, which is similar to the method for modifying your own profile. This method adopts the NSDictionary mode for modification, and multiple fields can be updated at a time.

```
@interface TIMFriendshipManager : NSObject
/**
* Modify a friend
*
* @param identifier Friend's identifier
* @param values Attributes to be updated. Multiple fields can be updated at a time. For more info
rmation, see TIMFriendTypeKey_XXX of TIMFriendshipDefine.h.
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)modifyFriend:(NSString *)identifier values:(NSDictionary<NSString *, id> *)values succ:(TI
MSucc)succ fail:(TIMFail)fail;
@end
```

Setting a non-existing key value may lead to a failure. The backend defines some common key values.

Key	Value	Description
TIMFriendTypeKey_Remark	**NSString**	**Remarks**
TIMFriendTypeKey_Group	**NSArray**	**Friend list**
TIMFriendTypeKey_Custom_Prefix	**NSNumber, NSData**	**Custom field prefix**

Example: set the remark of friend [iOS_002] to [002 remark]

```
[[TIMFriendshipManager sharedInstance] modifyFriend:@"iOS_002" values:@{ TIMFriendTypeKey_Remark:
@"002 remark"} succ:^{
self.msgLabel.text = @"OK";
} fail:^(int code, NSString *msg) {
self.msgLabel.text = [NSString stringWithFormat:@"Failed: %d, %@", code, msg];
}];
```

To modify a friend's custom information, you need to first configure the relationship chain custom fields on the server.

Adding a friend

You can use the `addFriend` method of `TIMFriendshipManager` to add a friend.

```
@interface TIMFriendshipManager : NSObject

/**
* Add a friend
*
* @param request Friend request
* @param succ Success callback (TIMFriendResult)
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)addFriend:(TIMFriendRequest *)request succ:(TIMFriendResultSucc)succ fail:(TIMFail)fail;

@end
```

The `request` parameter must be passed in to add a friend. Its parameter type is defined as follows:

```
/**
* Friend request
*/
@interface TIMFriendRequest : TIMCodingModel

/**
* User's identifier
*/
@property(nonatomic,strong) NSString* identifier;

/**
* User's remarks (a maximum of 96 bytes)
*/
@property(nonatomic,strong) NSString* remark;

/**
* Request description (a maximum of 120 bytes)
*/
@property(nonatomic,strong) NSString* addWording;

/**
* Source for adding a friend
* The source cannot exceed 8 bytes and must have the "AddSource_Type_" prefix.
*/
@property(nonatomic,strong) NSString* addSource;

/**
* Group name
```

```
*/
@property(nonatomic,strong) NSString* group;

@end
```

When the callback is successful, the `TIMFriendResult` result is returned for the operating user. Developers can notify the user accordingly. The return code for adding a friend is as follows:

```
typedef NS_ENUM(NSInteger, TIMFriendStatus) {
/**
* Successful operation
*/
TIM_FRIEND_STATUS_SUCC = 0,
/**
* Valid when adding a friend: the user that you want to add as a friend is in your blocklist.
*/
TIM_ADD_FRIEND_STATUS_IN_SELF_BLACK_LIST = 30515,
/**
* Valid when adding a friend: the user that you want to add as a friend has forbidden friend requ
ests.
*/
TIM_ADD_FRIEND_STATUS_FRIEND_SIDE_FORBID_ADD = 30516,
/**
* Valid when adding a friend and responding to a friend: your number of friends has reached the l
imit set by the system.
*/
TIM_ADD_FRIEND_STATUS_SELF_FRIEND_FULL = 30010,
/**
* Valid when adding a friend: the user that you want to add as a friend has added you to the bloc
klist.
*/
TIM_ADD_FRIEND_STATUS_IN_OTHER_SIDE_BLACK_LIST = 30525,
/**
* Valid when adding a friend: the friend request is pending approval.
*/
TIM_ADD_FRIEND_STATUS_PENDING = 30539,
};
```

Sample code

```
TIMFriendRequest *q = [TIMFriendRequest new];
q.identifier = @"abc"; // Add abc as a friend.
q.addWording = @"Please approve my request";
q.addSource = @"AddSource_Type_iOS";
q.remark = @"You are abc";
```

```
[[TIMFriendshipManager sharedInstance] addFriend:q succ:^(TIMFriendResult *result) {
if (result.result_code == 0)
self.msgLabel.text = @"Friend added successfully";
else
self.msgLabel.text = [NSString stringWithFormat:@"Exception: %ld, %@", (long)result.result_code,
result.result_info];
} fail:^(int code, NSString *msg) {
self.msgLabel.text = [NSString stringWithFormat:@"Failed: %d, %@", code, msg];
}];
```

Deleting friends

You can use the `deleteFriends` method of `TIMFriendshipManager` to delete friends in batches.

```
@interface TIMFriendshipManager : NSObject
/**
* Delete friends
*
* @param user Identifiers of friends to be deleted
* @param delType Deletion type (one-way friend or two-way friend)
* @param succ Success callback ([TIMFriendResult])
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)delFriend:(NSString *)user delType:(TIMDelFriendType)delType succ:(TIMHandleFriendArraySuc
c)succ fail:(TIMFail)fail;
@end
```

The success callback returns the `TIMFriendResult` result for the operating user. Developers can notify the user accordingly. The error codes for deleting friends are as follows:

```
typedef NS_ENUM(NSInteger, TIMFriendStatus) {
/**
* Successful operation
*/
TIM_FRIEND_STATUS_SUCC = 0,
/**
* Valid when deleting friends: the user that you want to delete is not your friend.
*/
TIM_DEL_FRIEND_STATUS_NO_FRIEND = 31704,
};
```

Sample code

```
NSMutableArray * del_users = [[NSMutableArray alloc] init];
// Delete the friend iOS_002.
[del_users addObject:@"iOS_002"];
// TIM_FRIEND_DEL_BOTH indicates deleting two-way friends.
[[TIMFriendshipManager sharedInstance] deleteFriends:del_users delType:TIM_FRIEND_DEL_BOTH succ:^
(NSArray<TIMFriendResult *> *results) {
for (TIMFriendResult * res in results) {
if (res.result_code != TIM_FRIEND_STATUS_SUCC) {
NSLog(@"deleteFriends failed: user=%@ result_code=%d", res.identifier, res.result_code);
}
else {
NSLog(@"deleteFriends succ: user=%@ result_code=%d", res.identifier, res.result_code);
}
}
} fail:^(int code, NSString * err) {
NSLog(@"deleteFriends failed: code=%d err=%@", code, err);
}];
```

Approving or rejecting a friend request

You can use the `doResponse` method of `TIMFriendshipManager` to approve or reject a friend request.

```
@interface TIMFriendshipManager : NSObject
/**
* Respond to a friend request
*
* @param response Response to the request
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)doResponse:(TIMFriendResponse *)response succ:(TIMFriendResultSucc)succ fail:(TIMFail)fai
l;
@end
```

The `response` parameter is defined as follows:

```
typedef NS_ENUM(NSInteger, TIMFriendResponseType) {
/**
* Approve the friend request (establish a one-way friendship)
*/
TIM_FRIEND_RESPONSE_AGREE = 0,
/**
```

```
* Approve the friend request and add the user as a friend (establish a two-way friendship)
*/
TIM_FRIEND_RESPONSE_AGREE_AND_ADD = 1,
/**
* Reject the friend request
*/
TIM_FRIEND_RESPONSE_REJECT = 2,
};
/**
* Respond to the friend request
*/
@interface TIMFriendResponse : NSObject

/**
* Response type
*/
@property(nonatomic,assign) TIMFriendResponseType responseType;

/**
* User's identifier
*/
@property(nonatomic,strong) NSString* identifier;

/**
* Friend remark (optional, if you want to add the user as your friend). The maximum length of the
remark is 96 bytes.
*/
@property(nonatomic,strong) NSString* remark;

@end
```

The success callback returns the `TIMFriendResult` result for the operating user. The error codes for handling friend requests are as follows:

```
typedef NS_ENUM(NSInteger, TIMFriendStatus) {
/**
* Successful operation
*/
TIM_FRIEND_STATUS_SUCC = 0,
/**
* Valid when responding to friend requests: the user has not requested to add you as a friend.
*/
TIM_RESPONSE_FRIEND_STATUS_NO_REQ = 30614,
/**
* Valid when adding a friend and responding to a friend: your number of friends has reached the l
imit set by the system.
*/
```

```
TIM_ADD_FRIEND_STATUS_SELF_FRIEND_FULL = 30010,
/**
* Valid when adding a friend and responding to a friend: the user's number of friends has reache
d the limit set by the system.
*/
TIM_ADD_FRIEND_STATUS_THEIR_FRIEND_FULL = 30014,
};
```

Verifying friend relationships

You can use the `checkFriends` method of `TIMFriendshipManager` to verify friend relationships.

```
/**
* Check your friend relationship with a specified user
*
* @param checkInfo Friend check information
* @param succ Success callback. The check result was returned.
* @param fail Failure callback
*
* @return 0 Sent successfully
*/
- (int)checkFriends:(TIMFriendCheckInfo *)checkInfo succ:(TIMCheckFriendResultArraySucc)succ fai
l:(TIMFail)fail;
```

The `checkInfo` parameter is defined as follows:

```
/**
* Friend relationship check
*/
@interface TIMFriendCheckInfo : NSObject
/**
* ID list of the users to be checked (NSString*)
*/
@property(nonatomic,strong) NSArray* users;

/**
* Check type
*/
@property(nonatomic,assign) TIMFriendCheckType checkType;

@end
```

The `TIMFriendCheckType` parameter is defined as follows:

```
/**
* Friend check type
```

```
*/
typedef NS_ENUM(NSInteger,TIMFriendCheckType) {
/**
* One-way friend
*/
TIM_FRIEND_CHECK_TYPE_UNIDIRECTION = 0x1,
/**
* Two-way friend
*/
TIM_FRIEND_CHECK_TYPE_BIDIRECTION = 0x2,
};
```

The success callback returns the `TIMCheckFriendResult` list for the operating user. The parameter is defined as follows:

```
@interface TIMCheckFriendResult : NSObject
/**
* User ID
*/
@property NSString* identifier;

/**
* Return code
*/
@property NSInteger result_code;

/**
* Return message
*/
@property NSString *result_info;

/**
* Check result
*/
@property(nonatomic,assign) TIMFriendRelationType resultType;

@end
```

The `TIMFriendRelationType` parameter is defined as follows:

```
/**
* Friend relationship type
*/
typedef NS_ENUM(NSInteger,TIMFriendRelationType) {
/**
* Non-friend
*/
```

```
TIM_FRIEND_RELATION_TYPE_NONE = 0x0,
/**
* The user is in my friend list
*/
TIM_FRIEND_RELATION_TYPE_MY_UNI = 0x1,
/**
* I am in the user’s friend list
*/
TIM_FRIEND_RELATION_TYPE_OTHER_UNI = 0x2,
/**
* Two-way friend
*/
TIM_FRIEND_RELATION_TYPE_BOTHWAY = 0x3,
};
```

Pending Friend Requests

Obtaining the pending request list

When another user uses the `addFriend` method to send a friend request to you, a pending record will be added on the backend. When you send a friend request to another user, a pending record will also be added on the backend. The `getPendencyList` method can be used to obtain the pending request list.

```
@interface TIMFriendshipManager : NSObject

/**
* Obtain the pending list
*
* @param pendencyRequest Request information. For more information, see TIMFriendPendencyRequest.
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)getPendencyList:(TIMFriendPendencyRequest *)pendencyRequest succ:(TIMGetFriendPendencyList
Succ)succ fail:(TIMFail)fail;
@end
```

The backend may store multiple pending friend requests that cannot be fully displayed on the screen. Therefore, this API allows you to page up or down the pending list. In this case, the `pendencyRequest` parameter must be passed in. This parameter is defined is as follows:

```
/**
* Pending request information
*/
@interface TIMFriendPendencyRequest : TIMCodingModel

/**
* `seq`, which is the sequence number of the pending request list
* We recommend that you save `seq` and the pending request list on the client. You need to enter `seq` returned by the server when sending a request.
* If `seq` is the latest one on the server, no data is returned.
*/
@property(nonatomic,assign) uint64_t seq;

/**
* Paging timestamp, only used for paging up or down. If the server returns 0, there is no more data. Enter 0 for the first request.
* Note that, if `seq` returned by the server is different from the entered `seq`, use the original `seq` on the client for paging up or down. The local `seq` is not updated until the data request is completed.
*/
@property(nonatomic,assign) uint64_t timestamp;

/**
* Amount of data per page, that is, the maximum amount of data returned per request
*/
@property(nonatomic,assign) uint64_t numPerPage;

/**
* Pending request fetching type
*/
@property(nonatomic,assign) TIMPendencyGetType type;

@end
```

After the operation succeeds, the `succ` callback returns the paging information and pending records.

```
/**
* Pending information returned
*/
@interface TIMFriendPendencyResponse : TIMCodingModel

/**
* Sequence number of the pending request list for the current request
*/
@property(nonatomic,assign) uint64_t seq;
```

```
/**
* Paging timestamp for the current request
*/
@property(nonatomic,assign) uint64_t timestamp;

/**
* Number of unread pending requests
*/
@property(nonatomic,assign) uint64_t unreadCnt;

@end
```

```
/**
* Pending request
*/
@interface TIMFriendPendencyItem : TIMCodingModel

/**
* User ID
*/
@property(nonatomic,strong) NSString* identifier;
/**
* Add time
*/
@property(nonatomic,assign) uint64_t addTime;
/**
* Source
*/
@property(nonatomic,strong) NSString* addSource;
/**
* Remarks of the friend request
*/
@property(nonatomic,strong) NSString* addWording;

/**
* Nickname of the added friend
*/
@property(nonatomic,strong) NSString* nickname;

/**
* Type of the pending request
*/
@property(nonatomic,assign) TIMPendencyGetType type;

@end
```

Deleting a pending record

```
@interface TIMFriendshipManager : NSObject
/**
* Delete a pending record
*
* @param type Type of the friend request
* @param identifiers Pending list to be deleted
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)deletePendency:(TIMPendencyGetType)type users:(NSArray *)identifiers succ:(TIMSucc)succ fa
il:(TIMFail)fail;
@end
```

Marking a pending record as read

When users fetch pending records, the pending records can be marked as read on the backend.

```
@interface TIMFriendshipManager : NSObject
/**
* Mark a pending record as read
*
* @param timestamp Read timestamp. All messages prior to this timestamp will be set to the read s
tate.
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)pendencyReport:(uint64_t)timestamp succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

After a pending record is marked as read, the unread count returned is changed the next time `getPendencyList` is called.

Blocklist

Adding a user to the blocklist

You can block any user. If the user is your friend, after you block the user, your friend relationship with the user is terminated and you cannot receive messages from this user.

```
@interface TIMFriendshipManager : NSObject
/**
* Add a user to the blocklist
*
* @param identifiers User list
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)addBlackList:(NSArray *)identifiers succ:(TIMFriendResultArraySucc)succ fail:(TIMFail)fai
l;
@end
```

Deleting a user from the blocklist

```
@interface TIMFriendshipManager : NSObject
/**
* Delete a user from the blocklist
*
* @param identifiers User list
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)deleteBlackList:(NSArray *)identifiers succ:(TIMFriendResultArraySucc)succ fail:(TIMFail)f
ail;
@end
```

Obtaining the blocklist

```
@interface TIMFriendshipManager : NSObject
/**
* Obtain the blocklist
*
* @param succ Success callback. The NSString* list is returned.
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
```

```
- (int)getBlackList:(TIMFriendArraySucc)succ fail:(TIMFail)fail;
@end
```

Friend List

Creating a friend list

While creating a friend list, you can select the users to be added to the list. A user can be added to multiple friend lists.

```
/**
* Create a friend list
*
* @param groupNames Name of the friend list. It must be a new friend list that currently does not exist.
* @param identifiers Friends to be added to the friend list
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)createFriendGroup:(NSArray *)groupNames users:(NSArray *)identifiers succ:(TIMFriendResultArraySucc)succ fail:(TIMFail)fail;
@end
```

Deleting a friend list

```
@interface TIMFriendshipManager : NSObject
/**
* Delete a friend list
*
* @param groupNames Name of the friend list to be deleted
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)deleteFriendGroup:(NSArray *)groupNames succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Adding a friend to a friend list

```
@interface TIMFriendshipManager : NSObject
/**
* Add a friend to a friend list
*
* @param groupName Friend list name
* @param identifiers Friend to be added to the friend list
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)addFriendsToFriendGroup:(NSString *)groupName users:(NSArray *)identifiers succ:(TIMFriend
ResultArraySucc)succ fail:(TIMFail)fail;
@end
```

Deleting a friend from a friend list

```
@interface TIMFriendshipManager : NSObject
/**
* Delete a friend from a friend list
*
* @param groupName Friend list name
* @param identifiers Friend to be deleted from the friend list
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)delFriendsFromFriendGroup:(NSString *)groupName users:(NSArray *)identifiers succ:(TIMFrie
ndResultArraySucc)succ fail:(TIMFail)fail;
@end
```

Renaming a friend list

```
@interface TIMFriendshipManager : NSObject
/**
* Modify the name of a friend list
*
* @param oldName Original name of the friend list
* @param newName New name of the friend list
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
```

```
- (int)renameFriendGroup:(NSString*)oldName newName:(NSString*)newName succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Obtaining a specified friend list

```
@interface TIMFriendshipManager : NSObject
/**
* Obtain a specified friend list
*
* @param groupNames Name of the friend list to be obtained. If `nil` is passed in, all friend lists are obtained.
* @param succ Success callback. The TIMFriendGroup* list is returned.
* @param fail Failure callback
*
* @return 0 The request was sent successfully.
*/
- (int)getFriendGroups:(NSArray *)groupNames succ:(TIMFriendGroupArraySucc)succ fail:(TIMFail)fail;
@end
```

System Notification for Relationship Chain Change

In `TIMMessage`, `Elem` = `TIMSNSSystemElem` indicates a system notification for a relationship chain change.

Prototype:

```
typedef NS_ENUM(NSInteger, TIM_SNS_SYSTEM_TYPE){
/**
* Friend adding message
*/
TIM_SNS_SYSTEM_ADD_FRIEND = 0x01,
/**
* Friend deleting message
*/
TIM_SNS_SYSTEM_DEL_FRIEND = 0x02,
/**
* Friend adding request
*/
TIM_SNS_SYSTEM_ADD_FRIEND_REQ = 0x03,
/**
* Pending request deleting request
*/
```

```
TIM_SNS_SYSTEM_DEL_FRIEND_REQ = 0x04,
};
/**
* Details of the relationship chain change
*/
@interface TIMSNSChangeInfo : NSObject
/**
* User's identifier
*/
@property(nonatomic,retain) NSString * identifier;
/**
* Valid when requesting to add a friend. It is used to add the cause.
*/
@property(nonatomic,retain) NSString * wording;
/**
* Entered to add the source when sending a friend request
*/
@property(nonatomic,retain) NSString * source;
@end
/**
* Relationship chain change message
*/
@interface TIMSNSSystemElem : TIMElem
/**
* Operation type
*/
@property(nonatomic,assign) TIM_SNS_SYSTEM_TYPE type;
/**
* Operated user list: TIMSNSChangeInfo list
*/
@property(nonatomic,retain) NSArray * users;

@end
```

Member description

Member	Description
type	**The type of modification.**
users	**The list of changed users.**

In this example, when a user adds another user as a friend or deletes a friend, a log is printed. When a user requests to become the friend of another user, the request reason is printed. Example:

```
@interface TIMMessageListenerImpl : NSObject
- (void)onNewMessage:(NSArray*) msgs;
@end
@implementation TIMMessageListenerImpl
- (void)onNewMessage:(NSArray*) msgs {
for (TIMMessage * msg in msgs) {
for (int i = 0; i < [msg elemCount]; i++) {
TIMElem * elem = [msg getElem:i];
if ([elem isKindOfClass:[TIMSNSSystemElem class]]) {
TIMSNSSystemElem * system_elem = (TIMSNSSystemElem * )elem;
switch ([system_elem type]) {
case TIM_SNS_SYSTEM_ADD_FRIEND:
for (TIMSNSChangeInfo * info in [system_elem users]) {
NSLog(@"user %@ become friends", [info identifier]);
}
break;
case TIM_SNS_SYSTEM_DEL_FRIEND:
for (TIMSNSChangeInfo * info in [system_elem users]) {
NSLog(@"user %@ delete friends", [info identifier]);
}
break;
case TIM_SNS_SYSTEM_ADD_FRIEND_REQ:
for (TIMSNSChangeInfo * info in [system_elem users]) {
NSLog(@"user %@ request friends: reason=%@", [info identifier], [info wording]);
}
break;
default:
NSLog(@"ignore type");
break;
}
}
}
}
}
@end
TIMMessageListenerImpl * impl = [[TIMMessageListenerImpl alloc] init];
[[TIMManager sharedInstance] setMessageListener:impl];
[[TIMManager sharedInstance] initSdk];
TIMLoginParam * login_param = [[TIMLoginParam alloc ]init];
login_param.accountType = @"107";
login_param.identifier = @"iOS_001";
login_param.userSig = @"";
login_param.appidAt3rd = @"123456";
login_param.sdkAppId = 123456;
[[TIMManager sharedInstance] login: login_param succ:^(){
NSLog(@"login succ");
} fail:^(int code, NSString * err) {
```

```
NSLog(@"login failed: %d->%@", code, err);
}];
```

System notification for adding a friend

When two users become friends, they both receive a system notification indicating that they are mutually added as friends.

Triggering time:

When you add a friend and your relationship chain changes accordingly, you will receive a message. If you are already a one-way friend of this added friend, the party whose relationship chain does not change will not receive a message.

Parameter description:

Parameter	Description
type	**TIM_SNS_SYSTEM_ADD_FRIEND**
users	**The list of users who become friends.**

`TIMSNSChangeInfo` **parameter description:**

Member	Description
identifier	**The user's identifier.**

System notification for deleting a friend

When two users terminate the friend relationship with each other, they both receive a system notification indicating that a friend has been deleted.

Triggering time:

When you delete a friend and your relationship chain changes accordingly, you will receive a message. If the deleted friend is a one-way friend, the party whose relationship chain does not change will not receive the message.

Parameter description:

| Parameter | Description |
type | TIM_SNS_SYSTEM_DEL_FRIEND
users | The list of deleted users.

`TIMSNSChangeInfo` **parameter description:**

Member	Description
identifier	**The user's identifier.**

System notification for a friend request

When you send a friend request to a user and the user requires verification, you and the user will receive a friend request system notification.

Triggering time: when you send a friend request to a user and the user requires verification, you and the user will receive a system notification for the friend request. The user can choose to approve or reject the request, whereas you cannot perform any operation. The system notification is only used for information synchronization.

Parameter description:

Parameter	Description
type	**TIM_SNS_SYSTEM_ADD_FRIEND_REQ**
users	**The list of friend requests.**

`TIMSNSChangeInfo` **parameter description:**

Parameter	Description
identifier	**The user's identifier.**
wording	**The request reason.**
source	**The request source.**

Notification for deleting a pending request

Triggering time: After your request to add a user as your friend is approved or rejected, you will receive a message indicating that the pending request was deleted.

System Notification for User Profile Change

In `TIMMessage` **,** `Elem` **=** `TIMProfileSystemElem` **indicates a system message for a user profile change.**

```
/**
* Message element pushed by the backend after your own profile or a friend's profile is changed
*/
@interface TIMProfileSystemElem : TIMElem
/**
* Change type
*/
@property(nonatomic,assign) TIM_PROFILE_SYSTEM_TYPE type;
/**
* User whose profile is changed
*/
@property(nonatomic,strong) NSString * fromUser;
/**
* Nickname of the changed profile (not yet implemented)
*/
@property(nonatomic,strong) NSString * nickName;
@end
/**
* Profile change
*/
typedef NS_ENUM(NSInteger, TIM_PROFILE_SYSTEM_TYPE){
/**
Change of a friend's profile
*/
TIM_PROFILE_SYSTEM_FRIEND_PROFILE_CHANGE = 0x01,
};
```

When your profile or a friend's profile is changed, you will receive a system notification indicating that a user profile is changed.

Offline Push
Offline Push (Android)
Offline Push Configuration

Last updated : 2021-09-01 17:43:30

Overview

IM terminal users need to obtain the latest messages at all times. However, due to the limited performance and battery power of mobile devices, when the app is running in the background, IM recommends that you use the system-grade push channels provided by vendors for message notifications to avoid excessive resource consumption caused by maintaining a persistent connection. Compared with third-party push, system-grade push channels provide more stable system-grade persistent connections, enabling users to receive push messages at any time and greatly reducing resource consumption.

Currently, IM supports APNs, MI push, Huawei push, Meizu push, Vivo push, OPPO push, and the push services of other vendors. The details are as follows:

Push Channel	System Requirements	Conditions
APNs	**iOS**	**iOS system push channel, the only iOS push channel**
MI push	**MIUI**	**Use MI push MiPush_SDK_Client_3_6_12.jar.**
Huawei push	**EMUI**	**Huawei mobile service versions of 20401300 and later, SDK version push: 2.6.3.301**
Google FCM push	**Android 4.1 and later versions**	**The mobile phone needs to install Google Play Services and be used outside the Chinese mainland.**
Meizu push	**Flyme**	**Use Meizu push push-internal:3.6.+.**
OPPO push	**ColorOS**	**Not all OPPO models and versions support OPPO push.**

		SDK version mcssdk-2.0.2.jar
Vivo push	**FuntouchOS**	**Not all Vivo models and versions support Vivo push. SDK version: vivo_pushsdk_v2.3.1.jar.**

Here, "offline" means that the app is closed by the system or user without logging out. In such cases, if you want to receive IM SDK message reminders, you can integrate IM offline push.

Note :

- **Users who have logged out normally or have been forced offline will not receive any message notifications.**
- **Currently, offline push notifications are only supported for ordinary chat messages, not for system messages.**

Basic Configuration of IM SDK Offline Push

Setting global offline push configuration

The IM SDK provides a feature for setting a global offline push configuration, allowing users to set whether to enable offline push and the alert sound when receiving offline push messages. The setting method is `setOfflinePushSettings` provided by `TIMManager` .

Note :

- **Calls to this method take effect only after successful login.**
- **Currently, only APNs custom alert sounds are supported, and the audio file needs to be a built-in audio file in the app.**

Prototype:

```
/**
* Initializing offline push configuration. The settings take effect only after login.
* @param settings Offline push configuration information
*/
```

```
public void setOfflinePushSettings(TIMOfflinePushSettings settings)
/**
* Obtaining offline push configuration from the server. It can be obtained only after login.
* @param cb Callback. The offline push configuration is returned in the onSuccess parameter.
*/
public void getOfflinePushSettings(final TIMValueCallBack<TIMOfflinePushSettings> cb)
```

Parameter description:

Parameter	Description
settings	**Offline push configuration**

TIMOfflinePushSettings description:

```
/**
* Obtaining whether the feature is enabled
* @return true: enabled. false: not enabled.
*/
public boolean isEnabled()
/**
* Setting whether to enable offline push
* @param enabled Whether to enable offline push
*/
public void setEnabled(boolean enabled)
/**
* Obtaining the alert sound for receiving offline push for C2C messages
* @return URI of the audio file. If it has not been set, 'null' is returned.
*/
public Uri getC2cMsgRemindSound()
/**
* Setting the alert sound for receiving offline push for C2C messages
* @param c2cMsgRemindSound URI of the audio file. To restore the default sound, enter 'null'.
*/
public void setC2cMsgRemindSound(Uri c2cMsgRemindSound)
/**
* Obtaining the alert sound for receiving offline push for group messages
* @return URI of the audio file. If it has not been set, 'null' is returned.
*/
public Uri getGroupMsgRemindSound()
/**
* Setting the alert sound for receiving offline push for group messages
* @param groupMsgRemindSound URI of the audio file. To restore the default sound, enter 'null'.
*/
public void setGroupMsgRemindSound(Uri groupMsgRemindSound)
```

Example:

```
TIMOfflinePushSettings settings = new TIMOfflinePushSettings();
//Enable offline push
settings.setEnabled(true);
//Set the alert sound for receiving offline C2C messages. In this example, the audio file is stor
ed in the res/raw folder.
settings.setC2cMsgRemindSound(Uri.parse("android.resource://" + getPackageName() + "/" + R.raw.du
dulu));
//Set the alert sound for receiving offline group messages. In this example, the audio file is st
ored in the res/raw folder.
settings.setGroupMsgRemindSound(Uri.parse("android.resource://" + getPackageName() + "/" + R.raw.
dudulu));
TIMManager.getInstance().setOfflinePushSettings(settings);
```

Setting offline push for a single message

The IM SDK provides a feature to set offline push configurations for specific messages. For any specified message, developers can set whether to enable offline push, the alert sound when receiving offline push messages, the offline push message description, and extension fields.

Note :

- **The offline push configuration for an individual message has the highest priority. That is, if a global offline push configuration and single-message offline push configuration are both set, the latter will prevail.**
- **Currently, only APNs custom alert sounds are supported, and the audio file needs to be a built-in audio file in the app.**

Prototype:

```
/**
* Setting the configuration for the time when the target user receives an offline push notificati
on for the current message (optional, set when sending the message)
* @param settings Offline push configuration
*/
public void setOfflinePushSettings(TIMMessageOfflinePushSettings settings)
/**
* Obtaining offline push configuration for the current message
* @return Offline push configuration. If the sender has not set it, 'null' is returned.
```

```
*/
public TIMMessageOfflinePushSettings getOfflinePushSettings()
```

`TIMMessageOfflinePushSettings` :

```
/**
* Display title for offline push on both iOS and Android platforms. If you want to set the displa
y title for the two platforms separately, set IOSSettings > title and AndroidSettings > title.
*
* @param title Notification bar title
* @return
*/
public TIMMessageOfflinePushSettings setTitle(String title)
/**
* Display text for offline push on both iOS and Android platforms. If you want to set the display
text for the two platforms separately, set IOSSettings > desc and AndroidSettings > desc.
* @param descr Text content
*/
public TIMMessageOfflinePushSettings setDescr(String descr)
/**
* Obtaining the offline push display content of the current message
* @return Text content
*/
public String getDescr()
/**
* Setting the extension field of the current message (optional, set when sending the message)
* @param ext Extension field content
*/
public TIMMessageOfflinePushSettings setExt(byte[] ext)
/**
* Obtaining the extension field of the current message
* @return Extension field content. If it has not been set, 'null' is returned.
*/
public byte[] getExt()
/**
* Setting whether the current message allows offline push. By default, it is allowed (optional, s
et when sending the message)
* @param enabled true: allow offline push. false: do not allow offline push.
*/
public TIMMessageOfflinePushSettings setEnabled(boolean enabled)
/**
* Obtaining whether push is allowed for the current message
* @return Whether push is allowed: true: allowed. false: not allowed.
*/
public boolean isEnabled()
/**
* Obtaining the offline push configuration for the current message on Android devices
```

```
 * @return Offline push configuration on Android devices
 */
public AndroidSettings getAndroidSettings()
/**
 * Setting the offline push configuration for the current message on Android devices (optional, se
t when sending the message)
 * @param androidSettings Offline push configuration for the current message on Android devices
 */
public TIMMessageOfflinePushSettings setAndroidSettings(AndroidSettings androidSettings)
/**
 * Obtaining the offline push configuration for the current message on iOS devices
 * @return Offline push configuration on iOS devices
 */
public IOSSettings getIosSettings()
/**
 * Setting the offline push configuration for the current message on iOS devices (optional, set wh
en sending the message)
 * @param iosSettings Offline push configuration for the current message on iOS devices
 */
public TIMMessageOfflinePushSettings setIosSettings(IOSSettings iosSettings)
```

TIMMessageOfflinePushSettings.AndroidSettings:

```
/**
 * Obtaining the notification title
 * @return Notification title
 */
public String getTitle()
/**
 * Setting the title to display for offline push
 * @param title Notification title
 */
public AndroidSettings setTitle(String title)
/**
 * Setting the custom text to display for offline push
 *
 * @param desc Display content of notification
 */
public AndroidSettings setDesc(String desc)
/**
 * Obtaining the offline push alert sound URI of the current message on Android devices
 * @return Sound URI. If it has not been set, 'null' is returned.
 */
public Uri getSound()
/**
 * Setting the offline push alert sound for the current message on Android devices (optional, set
when sending the message)
```

```
* @param sound Sound URI. Only the built-in audio resource files of apps are supported.
*/
public AndroidSettings setSound(Uri sound)
/**
* Obtaining the notification mode of the current message
* @return Notification mode
*/
public NotifyMode getNotifyMode()
/**
* Setting the notification mode of the current message when the receiver receives offline push (optional, will be deprecated soon)
*
* @param mode Notification mode. The default notification mode is ordinary notification bar message mode.
*/
public AndroidSettings setNotifyMode(NotifyMode mode)
```

TIMMessageOfflinePushSettings.NotifyMode:

```
/**
* Ordinary notification bar message mode. When an offline message is delivered, click the notification bar message to directly launch the app. It will not trigger callback for the app.
*/
NotifyMode.Normal
```

TIMMessageOfflinePushSettings.IOSSettings:

```
/**
* Setting the title to display for offline push
*
* @param title Notification title
*/
public IOSSettings setTitle(String title)
/**
* Setting the custom text to display for offline push
*
* @param desc
*/
public IOSSettings setDesc(String desc)
/**
* Obtaining the offline push alert sound of the current message on iOS devices
*
* @return Path of the audio file. If it has not been set, 'null' is returned.
*/
public String getSound()
/**
* Setting the offline push alert sound for the current message on iOS devices (optional, set when
```

```
sending the message)
*
* @param sound Path of the audio file. If it is set to {@see IOSSettings#NO_SOUND_NO_VIBRATION},
it indicates no alert sound and no vibration.
*/
public void setSound(String sound)
/**
* Obtaining whether badge count is enabled for the current message
*
* @return true Indicates that badge count is enabled for the current message.
*/
public boolean isBadgeEnabled()
/**
* Setting whether badge count is enabled for the current message. By default, it is enabled (opti
onal, set when sending the message).
*
* @param badgeEnabled Whether to enable badge count
*/
public IOSSettings setBadgeEnabled(boolean badgeEnabled)
```

Example:

```
// Construct a message
TIMMessage msg = new TIMMessage();
// Add text content
TIMTextElem elem = new TIMTextElem();
elem.setText("a new msg from " + selfId);
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}
// Set offline push configuration for the current message
TIMMessageOfflinePushSettings settings = new TIMMessageOfflinePushSettings();
settings.setEnabled(true);
// Set the title and content of notification bar messages on iOS and Android platforms. If you wa
nt the two platforms to display different titles and content in the notification bars, set them i
n AndroidSettings and IOSSettings respectively.
settings.setTitle("I'm title");
settings.setDescr("I'm description");
// Set offline push extension information
JSONObject object = new JSONObject();
try{
object.put("level", 15);
object.put("task", "TASK15");
settings.setExt(object.toString().getBytes("utf-8"));
} catch (JSONException e) {
e.printStackTrace();
```

```
} catch (UnsupportedEncodingException e) {
e.printStackTrace();
}
// Set offline configuration for receiving messages on Android devices
TIMMessageOfflinePushSettings.AndroidSettings androidSettings = new TIMMessageOfflinePushSetting
s.AndroidSettings();
// Construction mode prior to IM SDK 2.5.3
// TIMMessageOfflinePushSettings.AndroidSettings androidSettings = settings.new AndroidSettings
();
// Set the title and content of notification bar messages on Android platform
// androidSettings.setTitle("I'm title for android");
// androidSettings.setDesc("I'm desc for android");
// Set the alert sound for receiving messages on Android devices. The audio file needs to be put
in the raw folder.
androidSettings.setSound(Uri.parse("android.resource://" + getPackageName() + "/" +R.raw.hualal
a));
settings.setAndroidSettings(androidSettings);
//Set offline configuration for receiving messages on iOS devices.
TIMMessageOfflinePushSettings.IOSSettings iosSettings = new TIMMessageOfflinePushSettings.IOSSett
ings();
//Construction mode prior to IM SDK 2.5.3
//TIMMessageOfflinePushSettings.IOSSettings iosSettings = settings.new IOSSettings();
// Set the title and content of notification bar messages on iOS platform
// iosSettings.setTitle("I'm title for iOS");
// iosSettings.setDesc("I'm desc for iOS");
// Enable the badge count
iosSettings.setBadgeEnabled(true);
// Set no alert sound and no vibration for receiving messages on iOS devices (new feature introdu
ced by IM SDK 2.5.3)
//iosSettings.setSound(TIMMessageOfflinePushSettings.IOSSettings.NO_SOUND_NO_VIBRATION);
// Set the alert sound for receiving offline messages on iOS devices
iosSettings.setSound("/path/to/sound/file");
msg.setOfflinePushSettings(settings);
// Obtain a one-to-one conversation
TIMConversation conversation = TIMManager.getInstance().getConversation(
TIMConversationType.C2C, // Conversation type: one-to-one chat
peer); // Conversation peer's account
// Send the message
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {// Callback for sending a messa
ge
@Override
public void onError(int code, String desc) {// Failed to send the message
// "code" (error code) and "desc" (error description) can be used to locate the cause of the requ
est failure
// For a list of error codes, see the Error Code Table
Log.e(tag, "send message failed. code: " + code + " errmsg: " + desc);
}
@Override
```

```
public void onSuccess(TIMMessage msg) {//Message sent successfully
Log.d(tag, "SendMsg ok! peer:" + peer );
}
});
```

Offline Push (Mi)

Last updated : 2021-03-01 11:01:11

Offline Push Process

The process of implementing offline message push is as follows:

1. **Register with the vendor and complete the developer verification process. Apply to enable the push service.**
2. **Create a push service and bind app information to obtain the push certificate, password, key, and other data.**
3. **Log in to the IM console to upload the certificate and enter other required information. The IM server uses the certificate to generate a unique certificate ID.**
4. **Integrate the push messaging SDK provided by the vendor with your project and configure it according to the vendor's instructions.**
5. **Send your certificate ID and device information to IM server.**
6. **When the client app is killed by the system or user without IM logout, the IM server will remind the user via message push.**

Configuring Offline Push

MIUI is a highly customized Android system with very strict auto-start permission management. By default, no third-party apps are in the system's auto-start allowlist. As a result, background third-party apps are likely to be killed by the system. Therefore, you are advised to integrate Mi Push on Mi devices. Mi Push is a system service of MIUI, which has a high delivery rate for pushes. IM only supports Mi Push notification bar messages.

⚠ **Note :**

- **This document was written with reference to the official Mi Push documentation. In case of any change, the latest information can be found in the official Mi Push documentation.**
- **If you do not plan to implement a Mi device-specific offline push solution, skip this section.**

Step 1. Apply for a Mi Push certificate

1. Visit the Mi open platform website, register an account, and complete developer verification.

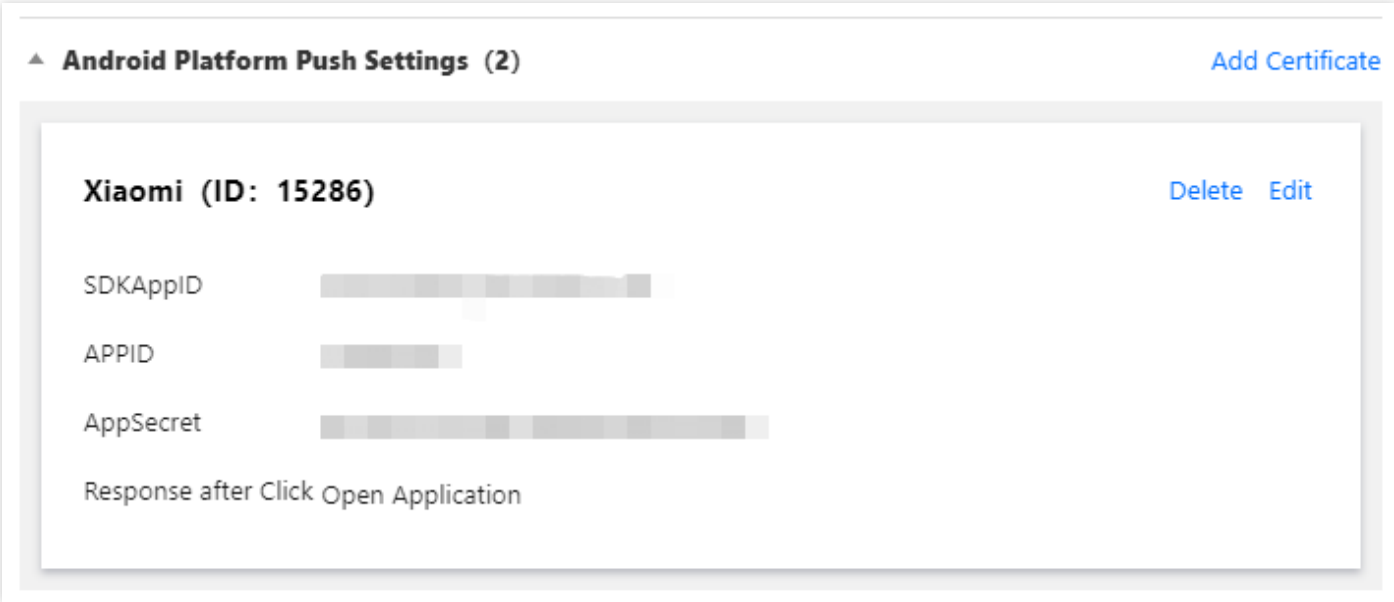
> **Note :**
>
> The verification process takes about two days. Please read the Mi Push Service Activation Guide in advance to avoid any effect on your connection progress.

2. Log in to the console of the Mi open platform, choose App Service -> Push Service, and create a Mi Push service app.
   Once the app is created, you can view detailed app information under the app details.
3. Record the `primary package name`, `AppID`, `AppSecret` information.

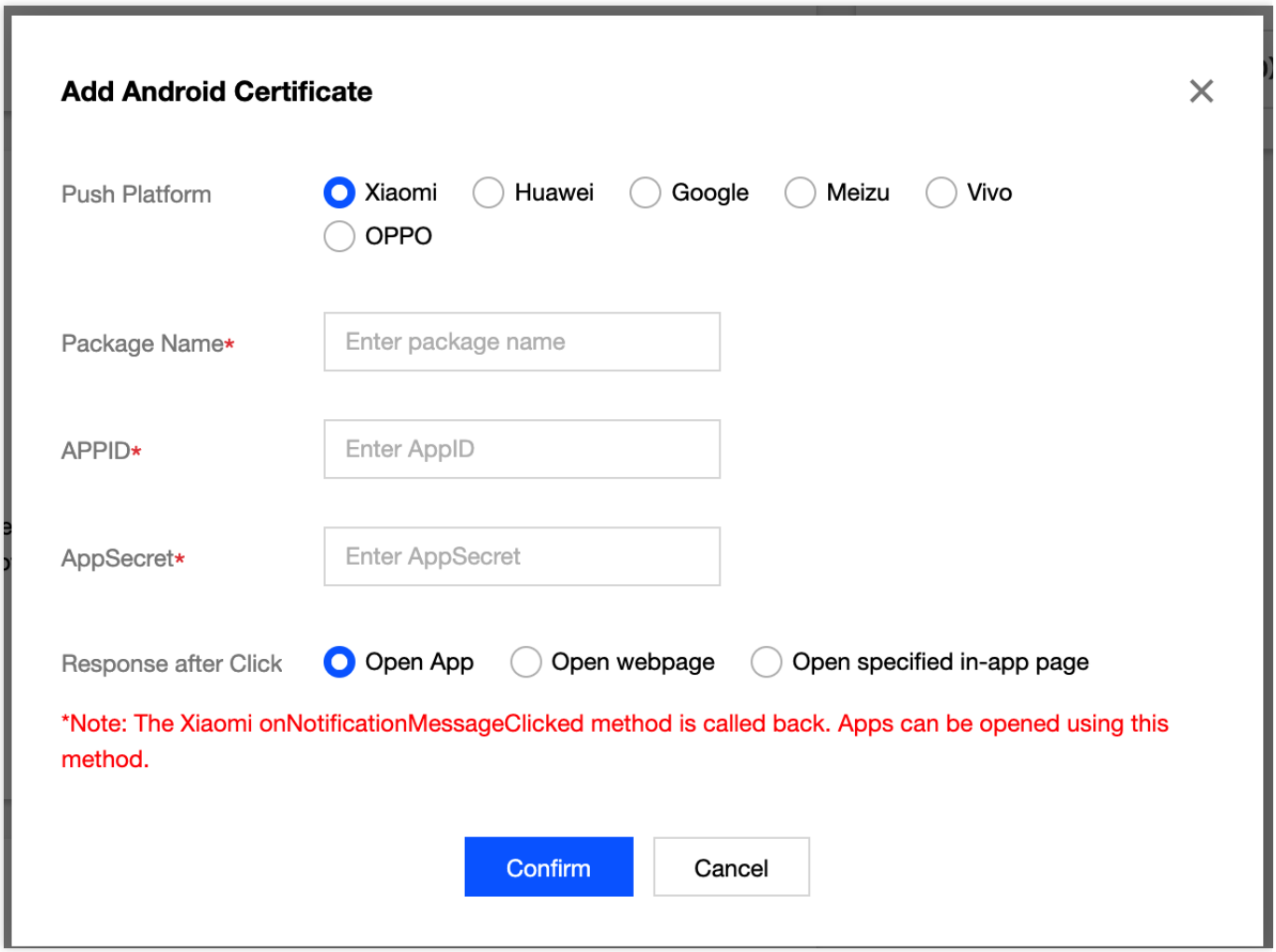
Step 2. Generate a certificate ID

1. Log in to the IM console and click the desired app to go to the configuration page of the app.
2. Click Add Certificate under Android Platform Push Settings.

> **Note :**
>
> If you already have a certificate and only want to change its information, you can click Edit in Android Platform Push Settings to modify and update the certificate.

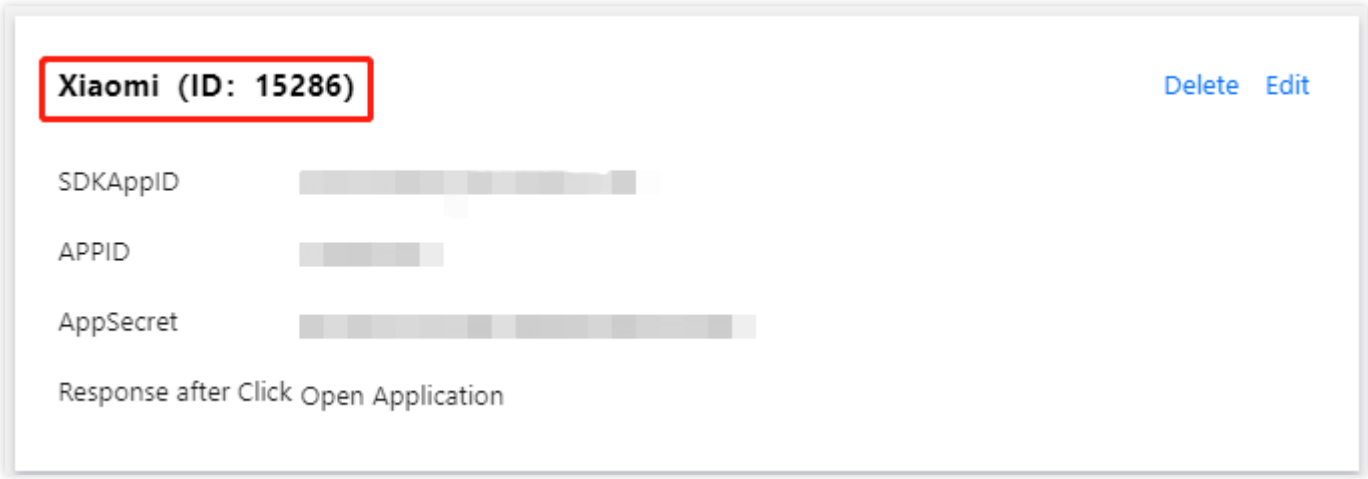


3. **Use the information you obtained in Step 1 to configure the following parameters:**
   - **Push Platform: choose Mi.**
   - **Package name: enter the primary package name of the Mi Push service app.**
   - **AppID: enter the AppID of the Mi Push service app.**
   - **AppSecret: enter the AppSecret of the Mi Push service app.**
   - **Response after Click: the event to take place after the notification bar message is clicked. Valid values include Open app, Open webpage, and Open specified in-app page. For more information, refer to Configuring Click Event.**
     Open app or Open specified in-app page allows custom content pass through.



4. **Click Confirm to save the information. Certificate information takes effect 10 minutes after you save it.**

5. Record the certificate ID once it is generated.



Step 3. Integrate the push SDK

Note :

- The default title of IM push notifications is `a new message`.
- Before reading this section, make sure that you have integrated and tested the IM SDK.
- You can find a sample for Mi Push implementation in our demo. Note that the features of Mi Push may be adjusted during Mi Push version updates. If you find any inconsistencies with the content of this section, please refer to the official Mi Push documentation and notify us of the difference so that we can make the necessary modifications in time.

Step 3.1. Download the Mi Push SDK and reference it in your project

1. Visit the Mi Push operations platform to download the Mi Push SDK.
2. Decompress the Mi Push SDK to obtain the `MiPush_SDK_client_**.jar` library files.
3. Copy these library files to `libs` under your project folder and reference them in your project.

Step 3.2. Configure the `AndroidManifest.xml` file

Add the necessary permissions for Mi Push:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
```

```
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.GET_TASKS" />
<uses-permission android:name="android.permission.VIBRATE"/>

<!--Change `com.tencent.qcloud.tim.tuikit` to your app package name.-->
<permission
android:name="com.tencent.qcloud.tim.tuikit.permission.MIPUSH_RECEIVE"
android:protectionLevel="signature" />
<uses-permission android:name="com.tencent.qcloud.tim.tuikit.permission.MIPUSH_RECEIVE" />
<!--Change `com.tencent.qcloud.tim.tuikit` to your app package name.-->
```

Configure the services and receivers required by the Mi Push service:

```
<service
android:enabled="true"
android:process=":pushservice"
android:name="com.xiaomi.push.service.XMPushService" />
<service
android:name="com.xiaomi.push.service.XMJobService"
android:enabled="true"
android:exported="false"
android:permission="android.permission.BIND_JOB_SERVICE"
android:process=":pushservice" /> <!--Note: this service must be added for 3.0.1 and later versio
ns.-->
<service
android:name="com.xiaomi.mipush.sdk.PushMessageHandler"
android:enabled="true"
android:exported="true" />

<service
android:name="com.xiaomi.mipush.sdk.MessageHandleService"
android:enabled="true" /> <!--Note: this service must be added for 2.2.5 and later versions.-->

<receiver
android:name="com.xiaomi.push.service.receivers.NetworkStatusReceiver"
android:exported="true" >
<intent-filter>
<action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
<category android:name="android.intent.category.DEFAULT" />
</intent-filter>
</receiver>

<receiver
android:name="com.xiaomi.push.service.receivers.PingReceiver"
android:exported="false"
android:process=":pushservice">
<intent-filter>
```

```
<action android:name="com.xiaomi.push.PING_TIMER" />
</intent-filter>
</receiver>
```

Step 3.3. Customize a `BroadcastReceiver` class

In order to receive messages, you need to customize a `BroadcastReceiver` class which inherits `PushMessageReceiver` and implements the `onReceiveRegisterResult` method. Also, register the `BroadcastReceiver` in `AndroidManifest.xml` .

The following is the sample code from the demo:

```
public class XiaomiMsgReceiver extends PushMessageReceiver {
private static final String TAG = "XiaomiMsgReceiver";
private String mRegId;

@Override
public void onReceiveRegisterResult(Context context, MiPushCommandMessage miPushCommandMessage) {
Log.d(TAG, "onReceiveRegisterResult is called. " + miPushCommandMessage.toString());
String command = miPushCommandMessage.getCommand();
List<String> arguments = miPushCommandMessage.getCommandArguments();
String cmdArg1 = ((arguments != null && arguments.size() > 0) ? arguments.get(0) : null);

Log.d(TAG, "cmd: " + command + " | arg: " + cmdArg1
+ " | result: " + miPushCommandMessage.getResultCode() + " | reason: " + miPushCommandMessage.get
Reason());

if (MiPushClient.COMMAND_REGISTER.equals(command)) {
if (miPushCommandMessage.getResultCode() == ErrorCode.SUCCESS) {
mRegId = cmdArg1;
}
}

Log.d(TAG, "regId: " + mRegId);
ThirdPushTokenMgr.getInstance().setThirdPushToken(mRegId); // Pass in `regId` here, which is requ
ired for subsequent push information reporting.
ThirdPushTokenMgr.getInstance().setPushTokenToTIM();
}

}
```

Register the custom `BroadcastReceiver` in `AndroidManifest.xml` :

```
<!--Change `com.tencent.qcloud.uipojo.thirdpush.XiaomiMsgReceiver` to the complete class name in
your app.-->
<receiver
```

```
android:name="com.tencent.qcloud.uipojo.thirdpush.XiaomiMsgReceiver"
android:exported="true">
<intent-filter>
<action android:name="com.xiaomi.mipush.RECEIVE_MESSAGE" />
</intent-filter>
<intent-filter>
<action android:name="com.xiaomi.mipush.MESSAGE_ARRIVED" />
</intent-filter>
<intent-filter>
<action android:name="com.xiaomi.mipush.ERROR" />
</intent-filter>
</receiver>
```

Step 3.4. Register Mi Push in your app

To enable Mi offline push, you need to register the push service with the Mi server, and initialize the Mi Push service by calling `MiPushClient.registerPush`. `MiPushClient.registerPush` can be called anywhere. To improve the registration success rate, Mi recommends calling in `onCreate` of `Application`.
After successful registration, you will receive the registration result in `onReceiveRegisterResult` of the `BroadcastReceiver` customized in Step 3.3. The `regId` in it is the unique identifier of the current app on the current device. Please record the `regId` information.

The following is the sample code from the demo:

```
public class DemoApplication extends Application {

private static DemoApplication instance;

@Override
public void onCreate() {
super.onCreate();
// Determine whether this is the main thread.
if (SessionWrapper.isMainProcess(getApplicationContext())) {
/**
* Initialization function of TUIKit
*
* @param context app context, usually corresponds to `ApplicationContext`.
* @param sdkAppID The `SDKAppID` assigned to you when registering the app in Tencent Cloud
* @param configs Relevant configuration items of TUIKit. Usually, you can use the default configu
ration. For special configuration, refer to the API Documentation.
*/
long current = System.currentTimeMillis();
TUIKit.init(this, Constants.SDKAPPID, BaseUIKitConfigs.getDefaultConfigs());
```

```
System.out.println(">>>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));
// Add custom initialization configuration.
customConfig();
System.out.println(">>>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));

if(IMFunc.isBrandXiaoMi()){
// Mi offline push
MiPushClient.registerPush(this, Constants.XM_PUSH_APPID, Constants.XM_PUSH_APPKEY);
}
if(IMFunc.isBrandHuawei()){
// Huawei offline push
HMSAgent.init(this);
}
if(MzSystemUtils.isBrandMeizu(this)){
// Meizu offline push
PushManager.register(this, Constants.MZ_PUSH_APPID, Constants.MZ_PUSH_APPKEY);
}
if(IMFunc.isBrandVivo()){
// vivo offline push
PushClient.getInstance(getApplicationContext()).initialize();
}
}
instance = this;
}
}
```

Step 4. Report the push information to the IM server

If you want to use IM to send notifications to a user through Mi Push, you need to use `setOfflinePushToken`, part of `TIMManager`, to send your certificate ID, generated by the IM console, and regId, generated by the Mi Push server, to the IM server, after the user has successfully logged in.

> ⚠ **Note :**
>
> **IM is only able to bind the user to the appropriate device and send notifications through Mi Push after the correct certificate ID and regId are sent.**

The following is the sample code from the demo:

- **Define certificate ID as a constant:**

```
/**
* Define some constant information in `Constants.java` first.
```

```
*/
/****** Mi offline push parameters start ******/
// Use your certificate ID in the Mi Push certificate information on the IM console.
public static final long XM_PUSH_BUZID = 6666;
// APPID and APPKEY assigned by the Mi open platform
public static final String XM_PUSH_APPID = "1234512345123451234";
public static final String XM_PUSH_APPKEY = "1234512345123";
/****** Mi offline push parameters end ******/
```

- **Report certificate ID and regId:**

```
/**
* Report the push certificate ID and device information in `ThirdPushTokenMgr.java`.
*/
public class ThirdPushTokenMgr {
private static final String TAG = "ThirdPushTokenMgr";
private String mThirdPushToken;

public static ThirdPushTokenMgr getInstance () {
return ThirdPushTokenHolder.instance;
}

private static class ThirdPushTokenHolder {
private static final ThirdPushTokenMgr instance = new ThirdPushTokenMgr();
}

public void setThirdPushToken(String mThirdPushToken) {
this.mThirdPushToken = mThirdPushToken; // The regId value is passed here. Describe it in accorda
nce with the above-mentioned custom `BroadcastReciever` class documentation.
}

public void setPushTokenToTIM(){
String token = ThirdPushTokenMgr.getInstance().getThirdPushToken();
if(TextUtils.isEmpty(token)){
QLog.i(TAG, "setPushTokenToTIM third token is empty");
return;
}
TIMOfflinePushToken param = null;
if(IMFunc.isBrandXiaoMi()){ // Select different push services for different vendors.
param = new TIMOfflinePushToken(Constants.XM_PUSH_BUZID, token);
}else if(IMFunc.isBrandHuawei()){
param = new TIMOfflinePushToken(Constants.HW_PUSH_BUZID, token);
}else if(IMFunc.isBrandMeizu()){
param = new TIMOfflinePushToken(Constants.MZ_PUSH_BUZID, token);
}else if(IMFunc.isBrandOppo()){
param = new TIMOfflinePushToken(Constants.OPPO_PUSH_BUZID, token);
}else if(IMFunc.isBrandVivo()){
```

```
param = new TIMOfflinePushToken(Constants.VIVO_PUSH_BUZID, token);
}else{
return;
}
TIMManager.getInstance().setOfflinePushToken(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.d(TAG, "setOfflinePushToken err code = " + code);
}

@Override
public void onSuccess() {
Log.d(TAG, "setOfflinePushToken success");
mIsTokenSet = true;
}
});
}
}
```

Step 5. Push messages offline

After the certificate ID and regId are successfully reported, the IM server sends messages via Mi Push notifications to the user when the app has been killed but the user has not logged out of IM.

Note :

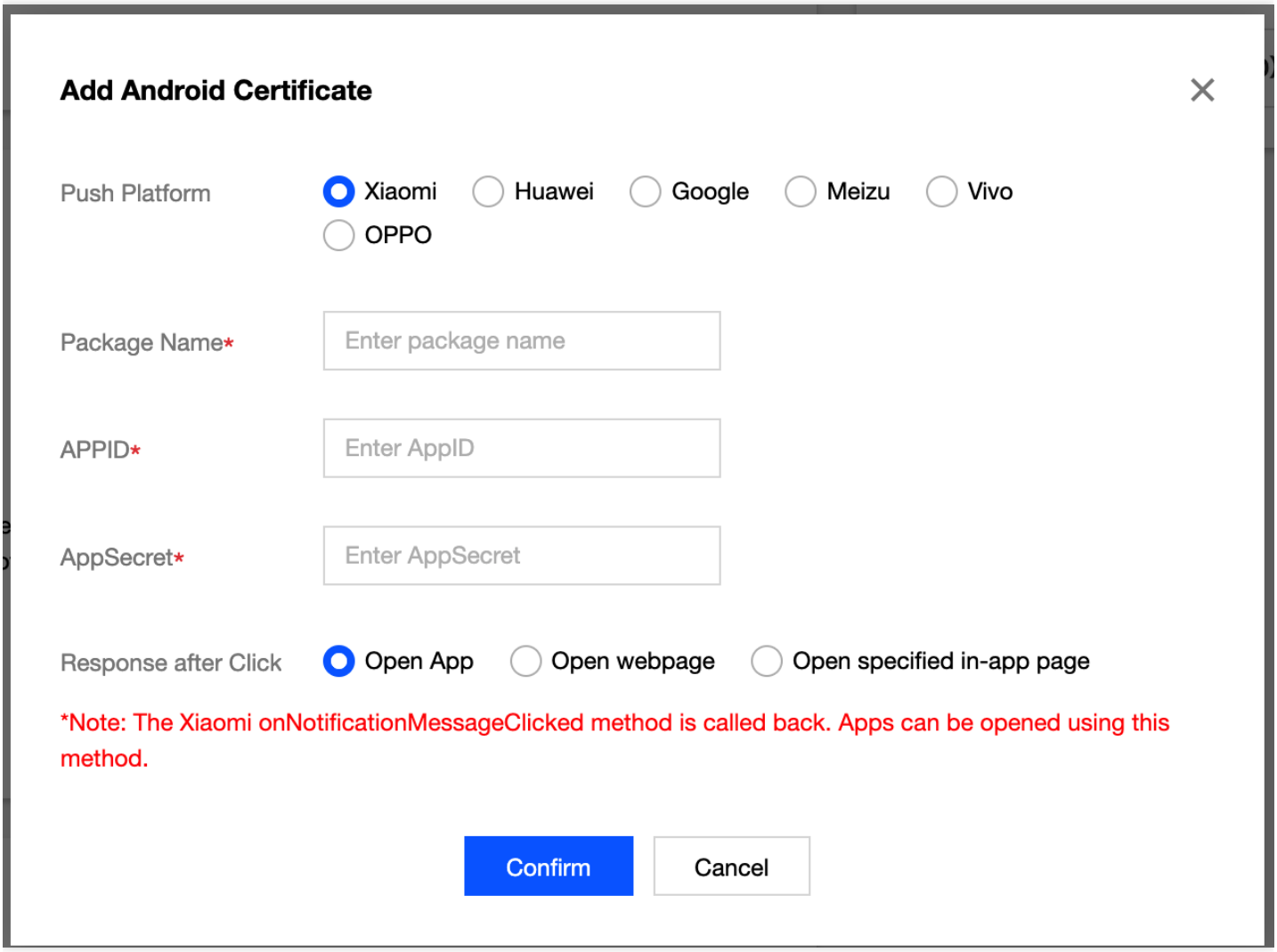
- **Mi Push does not guarantee 100% deliverability.**
- **Messages pushed via Mi Push may be delayed. Usually, this is related to the timing of app killing. In some cases, it is related to the Mi Push service.**
- **If the IM user has logged out or been forced offline by the IM server (for example, due to login on another device), the device cannot receive push messages.**

Configuring Click Events

You can select one of the following events: Open app, Open webpage, or Open specified in-app page.

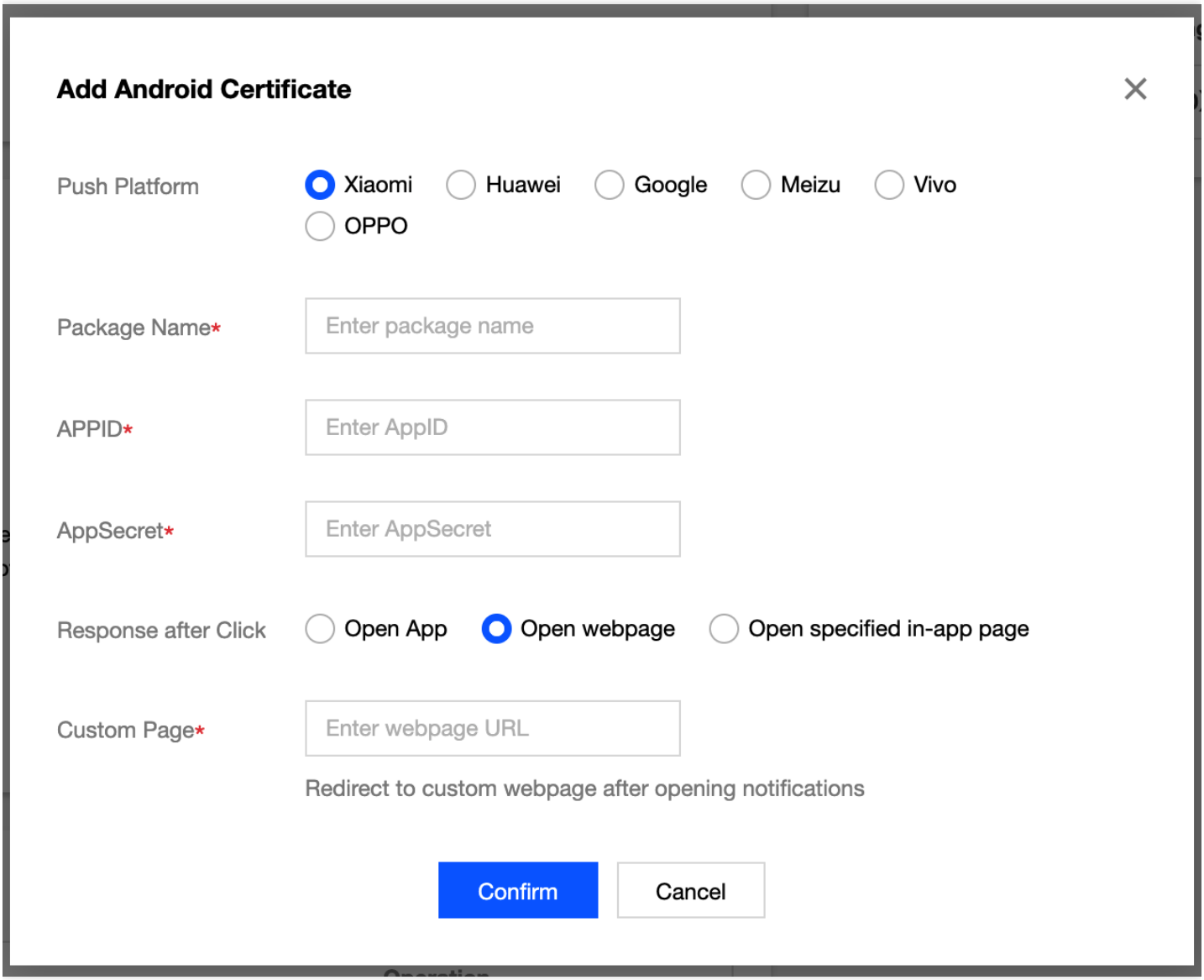
Open app

If you choose Open app, the `onNotificationMessageClicked` method of Mi Push will be called back, and the app itself can process app opening in this method.



Open webpage

You need to select Open webpage when adding a certificate and enter a URL that starts with either `http://` or `https://`, such as `https://cloud.tencent.com/document/product/269`.



Open specified in-app page

1. **Open manifest in a text editor and configure the `intent-filter` of the Activity you want to open as shown:**

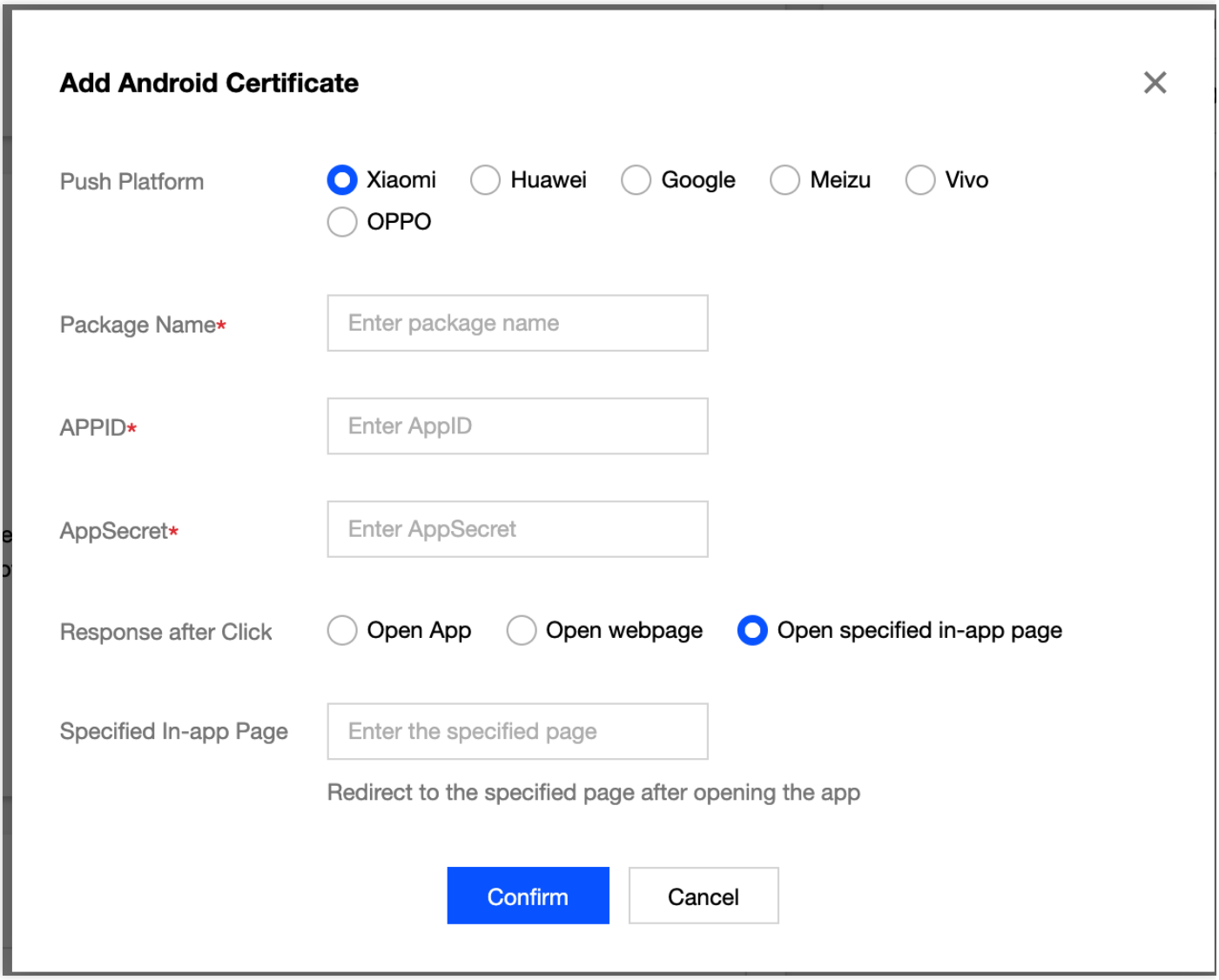
```
<activity
android:name="com.tencent.qcloud.tim.demo.chat.ChatActivity"
android:launchMode="singleTask"
android:screenOrientation="portrait"
android:windowSoftInputMode="adjustResize|stateHidden">

<intent-filter>
<action android:name="android.intent.action.VIEW" />
<data
```

```
android:host="com.tencent.qcloud.tim"
android:path="/detail"
android:scheme="pushscheme" />
</intent-filter>

</activity>
```

2. **Obtain the intent URL, as shown below:**

```
Intent intent = new Intent(this, ChatActivity.class);
intent.setData(Uri.parse("pushscheme://com.tencent.qcloud.tim/detail"));
intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
String intentUri = intent.toUri(Intent.URI_INTENT_SCHEME);
Log.i(TAG, "intentUri = " + intentUri);

// Print results.
intent://com.tencent.qcloud.tim/detail#Intent;scheme=pushscheme;launchFlags=0x4000000;componen
t=com.tencent.qcloud.tim.tuikit/com.tencent.qcloud.tim.demo.chat.ChatActivity;end
```

3. **Select Open specified in-app page when adding a certificate and enter the result above.**



Custom Content Pass Through

Select Open app or Open specified in-app page in Response after Click when adding a certificate to support custom content pass through.

Step 1. Set custom content (sender)

Set the custom content for the notification bar message before sending the message.

- **Sample on Android:**

```
String extContent = "ext content";
TIMMessageOfflinePushSettings settings = new TIMMessageOfflinePushSettings();
settings.setExt(extContent.getBytes());
```

```
timMessage.setOfflinePushSettings(settings);
mConversation.sendMessage(false, timMessage, callback);
```

- **For information on configurations for the IM server, refer to the OfflinePushInfo Format Example.**

Step 2. Set custom content (receiver)

- **If you selected Open app in Response after Click when adding a certificate, clicking the notification bar message triggers the `onNotificationMessageClicked(Context, MiPushMessage miPushMessage)` callback of Mi Push SDK. The custom content can be obtained from `miPushMessage`.**

```
Map extra = miPushMessage.getExtra();
String extContent = extra.get("ext");
```

- **If you selected Open specified in-app page in Response after Click when adding a certificate, `MiPushMessage`, which is the object that encapsulates the message, is passed to the client through `Intent`. The client then obtains the custom content from `Activity`.**

```
Bundle bundle = getIntent().getExtras();
MiPushMessage miPushMessage = (MiPushMessage)bundle.getSerializable(PushMessageHelper.KEY_MESSAGE);
Map extra = miPushMessage.getExtra();
String extContent = extra.get("ext");
```

FAQs

If the app uses obfuscation, how can I prevent exceptions in the Mi offline push feature?

If your app uses obfuscation, to prevent exceptions in the Mi offline push feature, you need to keep the custom `BroadcastReceiver` and add obfuscation rules by referring to the following:

Note :

The following code is a sample provided by Mi. Please modify it as needed before using it.

```
# Change `com.tencent.qcloud.tim.demo.thirdpush.XiaomiMsgReceiver` to the complete class name defined in your app.
-keep com.tencent.qcloud.tim.demo.thirdpush.XiaomiMsgReceiver {*;}
# If the Android v23 is used for compilation, adding this can prevent failed compilation caused by a false warning.
-dontwarn com.xiaomi.push.**
```

Can I set a custom notification sound?

Mi Push does not support custom notification sounds.

I cannot receive push messages. What should I do?

1. **No push message is 100% successful, even those from the vendor. If one or two push messages was not notified during a slew of rapid push messages, it is usually due to rate limiting measures put in place by the vendor.**
2. **Make sure the correct Mi Push certificate information is properly configured in the IM console.**
3. **Confirm that your project's Mi Push SDK integration configuration is correct and that you have obtained the regId.**
4. **Confirm that you have reported the correct push information to the IM server.**
5. **Manually kill the app on your device, send a few messages, and confirm whether you receive notifications within one minute.**

Offline Push (Huawei)

Last updated : 2021-01-26 16:39:06

Offline Push Process

The process of implementing offline message push is as follows:

1. **Register with the vendor and complete the developer verification process. Apply to enable the push service.**
2. **Create a push service and bind app information to obtain the push certificate, password, key, and other data.**
3. **Log in to the IM Console to upload the certificate and enter other required information. The IM server uses the certificate to generate a unique certificate ID.**
4. **Integrate the push messaging SDK provided by the vendor with your project and configure it according to the vendor's instructions.**
5. **Send your certificate ID and device information to IM server.**
6. **When the client App is killed by the system or user without IM logout, the IM server will remind the user via message push.**

Configuring Offline Push

Huawei EMUI is a highly customized Android system with strict backend policies. By default, third-party apps do not have auto-start permissions. As apps running in the background are often forcibly killed by the system, we recommend that the Huawei push service be integrated on Huawei devices. The Huawei push service is part of the Huawei Mobile Service (HMS) and a system-grade service of EMUI. Its delivery rate is higher than those of third-party push services. Currently, IM only supports the notification bar messages of Huawei push.

> ⚠ **Note :**
>
> - **This guide was prepared with direct reference to the official documentation of Huawei push. If Huawei push is changed, please refer to the official website of Huawei push.**
> - **If you do not plan to implement a Huawei-specific offline push solution, skip this section.**

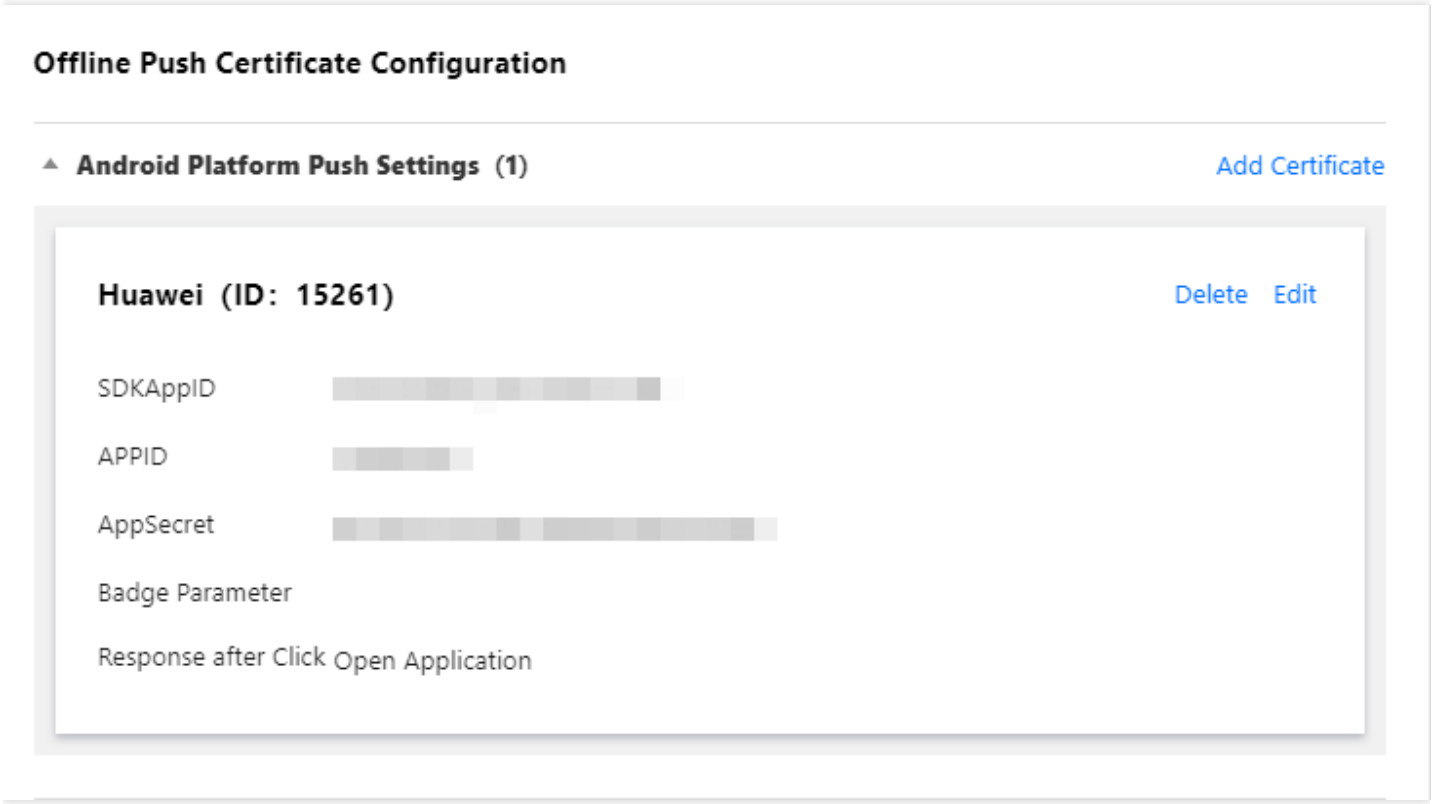
Step 1: Apply for a Huawei push certificate

1. Access the official website of the Huawei Developers Alliance, register an account, and pass the developer verification.
2. Log in to the console of the Huawei Developers Alliance, choose App Service -> Development Service -> PUSH, and create a Huawei push service app.
   When applying for the Huawei push service, you need to provide a maximum of five SHA256 fingerprints for the app signature certificates. After the Huawei push service app is created, you can view detailed app information on the app details page.
3. Record the `Package name`, `APP ID`, and `APP Secret` information.

Step 2: Generate a certificate ID

1. Log in to the IM Console and click the desired app. The app configuration page appears.
2. Click Add a certificate under Android push configuration.

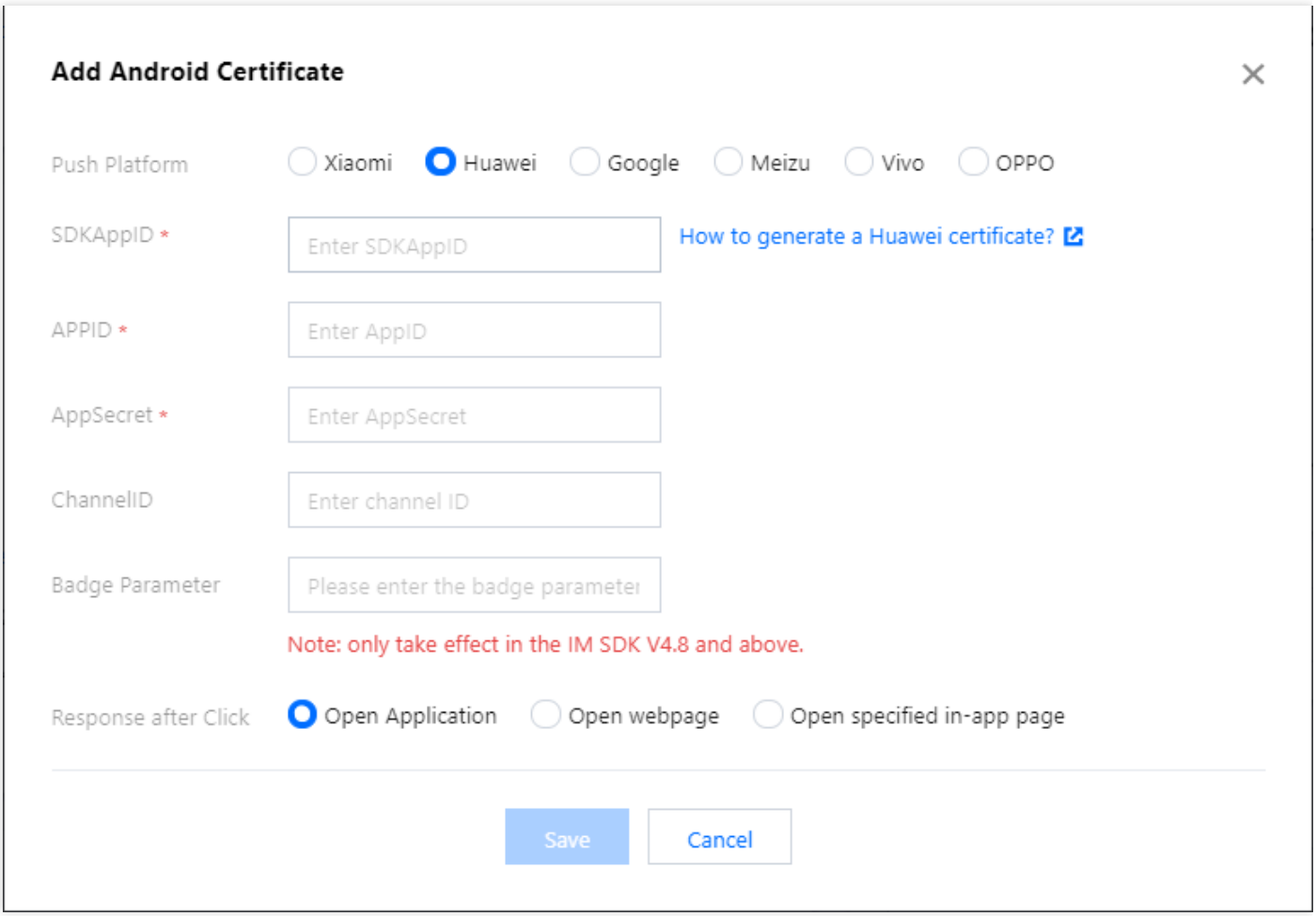
> **Note :**
>
> If you already have a certificate and only want to change its information, you can click Edit in Android push configuration to modify and update the certificate.



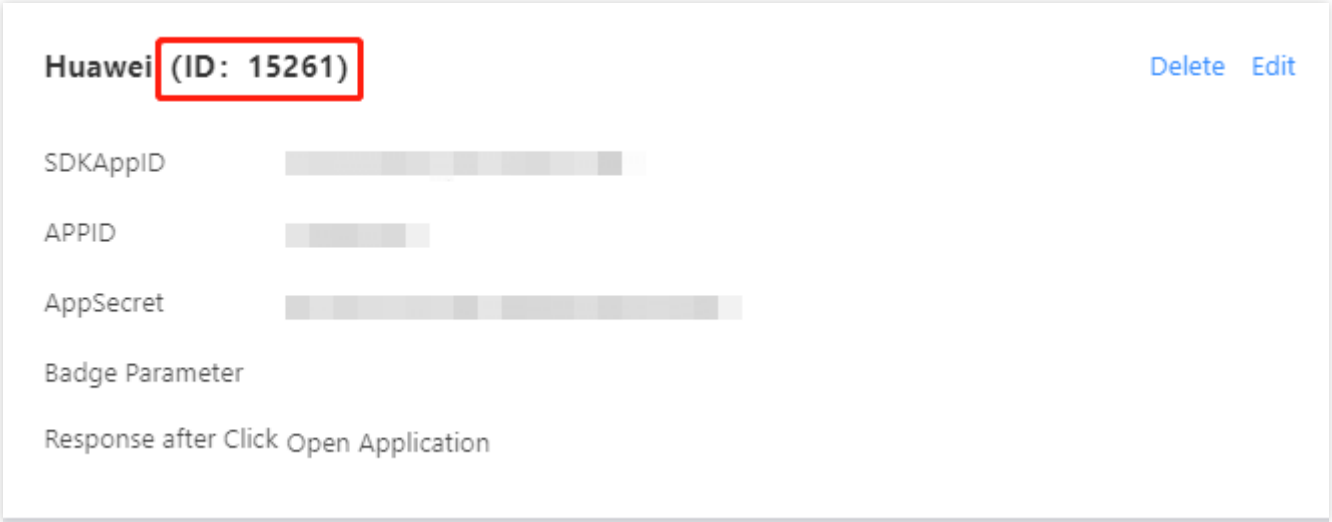
3. Use the information you obtained in Step 1 to configure the following parameters:

- **Push platform: select Huawei.**
- **Package name: the name of the Huawei Push service app.**
- **AppID: enter the App ID you got from Huawei Push.**
- **AppSecret: enter the APP SECRET you got from Huawei Push.**
- **Badge Parameter: enter the complete class name of `Activity` for the app entry as the application badge on Huawei Desktop. For more information, see Interface Description for Badging on Huawei Desktop**
- **Click event: the event to take place after the notification bar message is clicked. Valid values include Open App, Open URL, and Open specific App interface. For more information, refer to Configuring the Notification Bar Message Click Event.**

Open App or Open specific App interface allows custom content pass through.



4. **Click OK to save the information. Certificate information takes effect 10 minutes after you save it.**
5. **Record the Certificate ID once it is generated.**



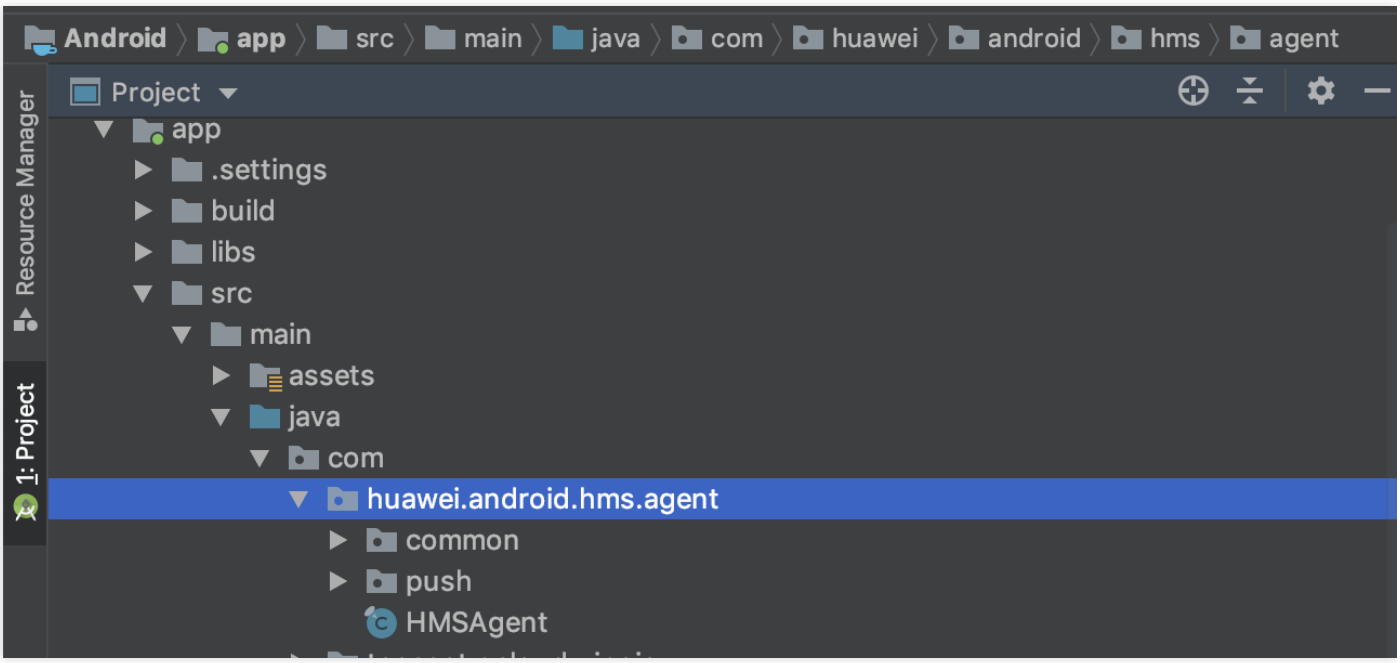
Step 3: Integrate push SDK

Note :

- The default title of IM push notifications is `a new message`.
- Before reading this section, make sure that you have integrated and tested the IM SDK.
- You can find a sample for Huawei push implementation in our demo. Note that the features of Huawei push may be adjusted during Huawei push version updates. If you find any inconsistencies with the content of this section, please refer to the official website of Huawei push and notify us of the difference so that we can make the necessary modifications in time.

Step 3.1: Download the Huawei Push SDK and reference it in your project

1. Visit the official website of Huawei push to download the HMS Agent.
2. Decompress the HMS Agent.
3. Copy the files under `hmsagents¥src¥main¥java` to your project's src\main\java directory.



4. Use Gradle to integrate the Huawei push SDK. Add the following code in the build.gradle of your project:

```
allprojects {
repositories {
jcenter()
maven {url 'http://developer.huawei.com/repo/'}
```

```
}
}
```

5. Add the following information in build.gradle of the sub-project:

```
dependencies {
// Huawei Push SDK. Replace 2.6.3.301 with the actual version number.
implementation 'com.huawei.android.hms:push:2.6.3.301'
// If an error occurs indicating that com.huawei.hms.api does not exist, this line of code als
o needs to be added. Note that the version number must be the same.
// implementation 'com.huawei.android.hms:base:2.6.3.301'
}
```

Step 3.2: Modify AndroidManifest.xml

1. Add necessary permissions for Huawei Push:

```
<!--HMS-SDK guides the upgrade of the HMS feature. Accessing the OTA server requires network perm
issions-->
<uses-permission android:name="android.permission.INTERNET" />
<!--HMS-SDK guides the upgrade of the HMS feature. Saving the downloaded upgrade package requires
the SD card write permissions-->
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<!--Check the network status-->
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<!--Check the Wi-Fi status-->
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<!--Obtain the user's mobile phone IMEI to uniquely mark the user-->
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>

<!--If the Android version is 8.0 with targetSdkVersion>=26 for application compilation configura
tion, you must add the following permissions -->
<uses-permission android:name="android.permission.REQUEST_INSTALL_PACKAGES" />

<!--Change com.tencent.qcloud.tim.tuikit to your App package name.-->
<permission
android:name="com.tencent.qcloud.tim.tuikit.permission.PROCESS_PUSH_MSG"
android:protectionLevel="signatureOrSystem"/>
<uses-permission android:name="com.tencent.qcloud.tim.tuikit.permission.PROCESS_PUSH_MSG" />
<!--Change com.tencent.qcloud.tim.tuikit to your App package name.-->
```

2. Add the following content under application. For a detailed description, see the official website of Huawei push.

```
<meta-data
android:name="com.huawei.hms.client.appid"
android:value="appid=1234567890"/> <!--Here, change the appid to your Huawei push App ID-->
<provider
android:name="com.huawei.hms.update.provider.UpdateProvider"
android:authorities="com.tencent.qcloud.tim.tuikit.hms.update.provider"
android:exported="false"
android:grantUriPermissions="true"/>
<provider
android:name="com.huawei.updatesdk.fileprovider.UpdateSdkFileProvider"
android:authorities="com.tencent.qcloud.tim.tuikit.updateSdk.fileProvider"
android:exported="false"
android:grantUriPermissions="true">
</provider>
<activity
android:name="com.huawei.android.hms.agent.common.HMSAgentActivity"
android:configChanges="orientation|locale|screenSize|layoutDirection|fontScale"
android:excludeFromRecents="true"
android:exported="false"
android:hardwareAccelerated="true"
android:theme="@android:style/Theme.Translucent" >
<meta-data
android:name="hwc-theme"
android:value="androidhwext:style/Theme.Emui.Translucent" />
</activity>
<activity
android:name="com.huawei.hms.activity.BridgeActivity"
android:configChanges="orientation|locale|screenSize|layoutDirection|fontScale"
android:excludeFromRecents="true"
android:exported="false"
android:hardwareAccelerated="true"
android:theme="@android:style/Theme.Translucent" >
<meta-data
android:name="hwc-theme"
android:value="androidhwext:style/Theme.Emui.Translucent" />
</activity>

<service
android:name="com.huawei.hms.support.api.push.service.HmsMsgService"
android:enabled="true"
android:exported="true"
android:process=":pushservice">
<intent-filter>
<action android:name="com.huawei.push.msg.NOTIFY_MSG" />
<action android:name="com.huawei.push.msg.PASSBY_MSG" />
</intent-filter>
</service>
```

Step 3.3: Define a BroadcastReceiver class

To receive messages, you need to customize a BroadcastReceiver inherited from the `PushReceiver` class, implement the `onToken` method in it, and register this receiver to AndroidManifest.xml.

The following is sample code from the demo:

```
public class HUAWEIPushReceiver extends PushReceiver {
private static final String TAG = "HUAWEIPushReceiver";

@Override
public void onToken(Context context, String token, Bundle extras) {
Log.i(TAG, "onToken:" + token);
ThirdPushTokenMgr.getInstance().setThirdPushToken(token); // The token is passed in here. It need
s to be used for subsequent reporting of push information.
ThirdPushTokenMgr.getInstance().setPushTokenToTIM();
}
}
```

Register the custom BroadcastReceiver to AndroidManifest.xml:

```
<!--Here, change com.tencent.qcloud.tim.demo.thirdpush.HUAWEIPushReceiver to the complete class n
ame in your App-->
<receiver android:name="com.tencent.qcloud.tim.demo.thirdpush.HUAWEIPushReceiver"
android:permission="com.tencent.qcloud.tim.tuikit.permission.PROCESS_PUSH_MSG">
<intent-filter>
<!-- Required; used for receiving the token -->
<action android:name="com.huawei.android.push.intent.REGISTRATION" />
<!-- Required; used for receiving pass-through messages -->
<action android:name="com.huawei.android.push.intent.RECEIVE" />
<!-- Required; used for receiving notification bar message click events. Developers do not need t
o handle the event. Only registration is required.-->
<action android:name="com.huawei.intent.action.PUSH_DELAY_NOTIFY"/>
</intent-filter>
</receiver>
```

Step 3.4: Register Huawei Push in your App

If you choose to enable the Huawei push service, you need to call `HMSAgent.init` in `onCreate` of the Application to initialize the Huawei push service.

After successful registration, you will receive the registration result in `onToken` of the BroadcastReceiver customized in Step 3.3. The `token` in it is the unique identifier of the current App on the current device. Please record the `token` information.

The following is sample code from the demo:

```
public class DemoApplication extends Application {

private static PojoApplication instance;

@Override
public void onCreate() {
super.onCreate();
// Determine whether this is the main thread
if (SessionWrapper.isMainProcess(getApplicationContext())) {
/**
* TUIKit initialization function
*
* @param context App context, usually corresponds to ApplicationContext
* @param sdkAppID The SDKAppID assigned to you when registering the App in Tencent Cloud
* @param configs Relevant configuration items of TUIKit. Usually, you can use the default configu
ration. For special configuration, refer to the API Documentation.
*/
long current = System.currentTimeMillis();
TUIKit.init(this, Constants.SDKAPPID, BaseUIKitConfigs.getDefaultConfigs());
System.out.println(">>>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));
// Add custom initialization configuration
customConfig();
System.out.println(">>>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));

if(IMFunc.isBrandXiaoMi()){
// Xiaomi offline push
MiPushClient.registerPush(this, Constants.XM_PUSH_APPID, Constants.XM_PUSH_APPKEY);
}
if(IMFunc.isBrandHuawei()){
// Huawei offline push
HMSAgent.init(this);
}
if(MzSystemUtils.isBrandMeizu(this)){
// Meizu offline push
PushManager.register(this, Constants.MZ_PUSH_APPID, Constants.MZ_PUSH_APPKEY);
}
if(IMFunc.isBrandVivo()){
// vivo offline push
PushClient.getInstance(getApplicationContext()).initialize();
}
}
instance = this;
}
}
```

Obtaining the token from the main interface:

```
if (IMFunc.isBrandHuawei()) {
// Huawei offline push
HMSAgent.connect(this, new ConnectHandler() {
@Override
public void onConnect(int rst) {
QLog.i(TAG, "huawei push HMS connect end:" + rst);
}
});
getHuaWeiPushToken();
}
```

Step 4: Report the push information to the IM server

If you need to use Huawei push to push IM message notifications, then after successful user login, you must use the `setOfflinePushToken` method of `TIMManager` to report the certificate ID generated and hosted by the IM console and token returned by the Huawei push service to the IM server.

> ⚠ **Note :**
>
> **After the token and certificate ID are correctly reported, IM service binds users with the corresponding device information. This enables the use of the Huawei push service to push notifications.**

The following is sample code from the demo:

- **Define Certificate ID as a constant:**

```
/**
* We first define some constant information in Constants.java.
*/
/****** Huawei offline push parameters start ******/
// Use your certificate ID in the Huawei push certificate information on the IM console
public static final long HW_PUSH_BUZID = 6666;
// APPID assigned by the Huawei Developers Alliance
public static final String HW_PUSH_APPID = "1234567890"; // See the checklist.
/****** Huawei offline push parameters end ******/
```

- **Report the push certificate ID and token:**

```
/**
* Report the push certificate ID and device information in ThirdPushTokenMgr.java
*/
public class ThirdPushTokenMgr {
private static final String TAG = "ThirdPushTokenMgr";
private String mThirdPushToken;

public static ThirdPushTokenMgr getInstance () {
return ThirdPushTokenHolder.instance;
}

private static class ThirdPushTokenHolder {
private static final ThirdPushTokenMgr instance = new ThirdPushTokenMgr();
}

public void setThirdPushToken(String mThirdPushToken) {
this.mThirdPushToken = mThirdPushToken; // token value is specified here. Describe it in accor
dance with the above-mentioned custom BroadcastReciever class documentation.
}

public void setPushTokenToTIM(){
String token = ThirdPushTokenMgr.getInstance().getThirdPushToken();
if(TextUtils.isEmpty(token)){
QLog.i(TAG, "setPushTokenToTIM third token is empty");
mIsTokenSet = false;
return;
}

TIMOfflinePushToken param = null;
if(IMFunc.isBrandXiaoMi()){ // Select different push services for different vendors.
param = new TIMOfflinePushToken(Constants.XM_PUSH_BUZID, token);
}else if(IMFunc.isBrandHuawei()){
param = new TIMOfflinePushToken(Constants.HW_PUSH_BUZID, token);
}else if(IMFunc.isBrandMeizu()){
param = new TIMOfflinePushToken(Constants.MZ_PUSH_BUZID, token);
}else if(IMFunc.isBrandOppo()){
param = new TIMOfflinePushToken(Constants.OPPO_PUSH_BUZID, token);
}else if(IMFunc.isBrandVivo()){
param = new TIMOfflinePushToken(Constants.VIVO_PUSH_BUZID, token);
}else{
return;
}
TIMManager.getInstance().setOfflinePushToken(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.d(TAG, "setOfflinePushToken err code = " + code);
}
```

```
@Override
public void onSuccess() {
Log.d(TAG, "setOfflinePushToken success");
mIsTokenSet = true;
}
});
}
}
```

Step 5: Offline push

After the certificate ID and token are successfully reported, the IM server sends messages via Huawei push notifications to the user when the App has been killed but the user has not logged out of IM.

Note :

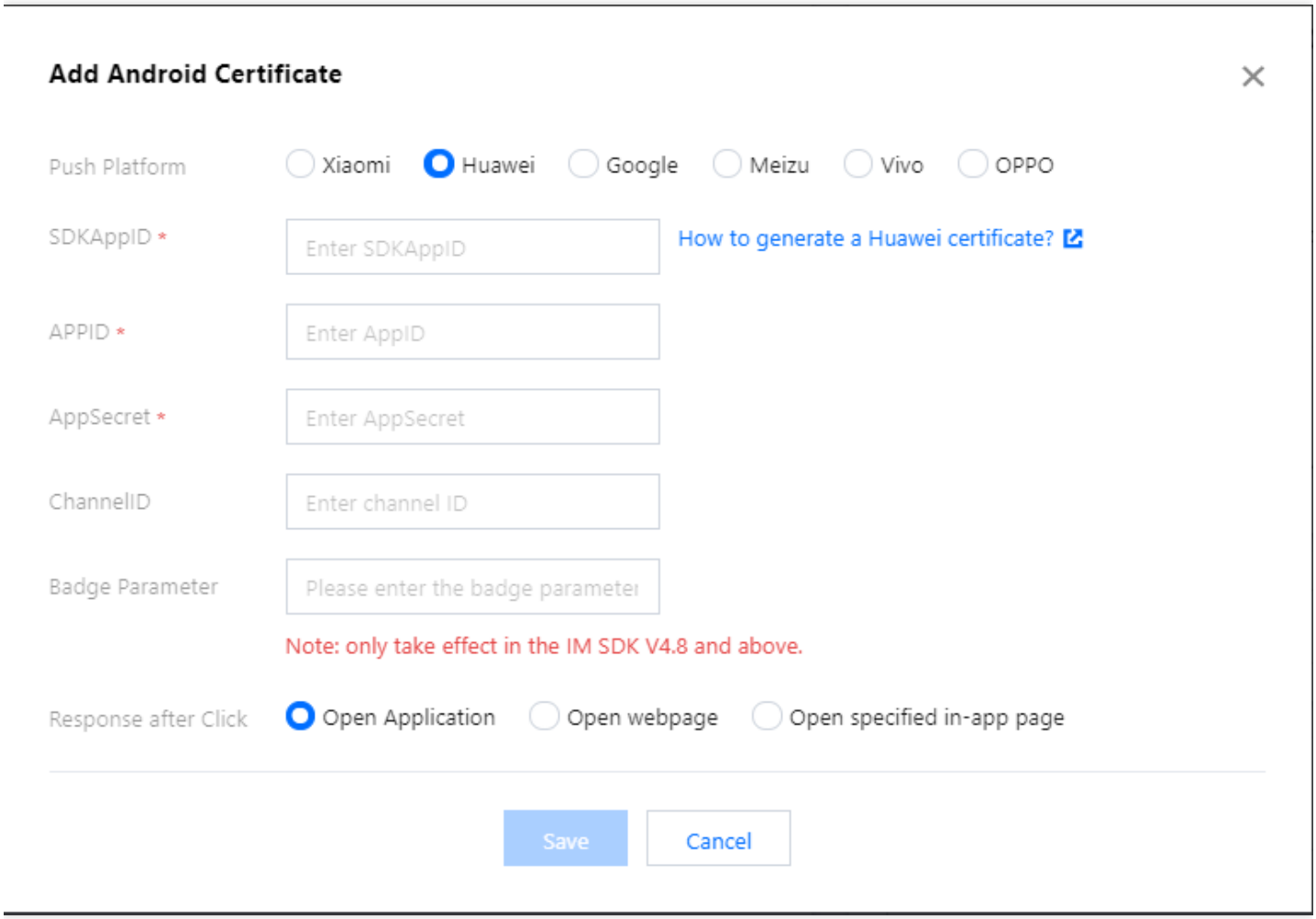
- **Huawei push is not 100% successful in reaching the target users.**
- **Huawei push may be delayed. Usually, this is related to the timing of App killing. In some cases, it is related to the Huawei push service.**
- **If the IM user has logged out or been forced offline by the IM server (for example, due to login on another device), the device cannot receive push messages.**

Configuring the Notification Bar Message Click Event

You can select one of the following events: Open App, Open URL, or Open specific App interface.

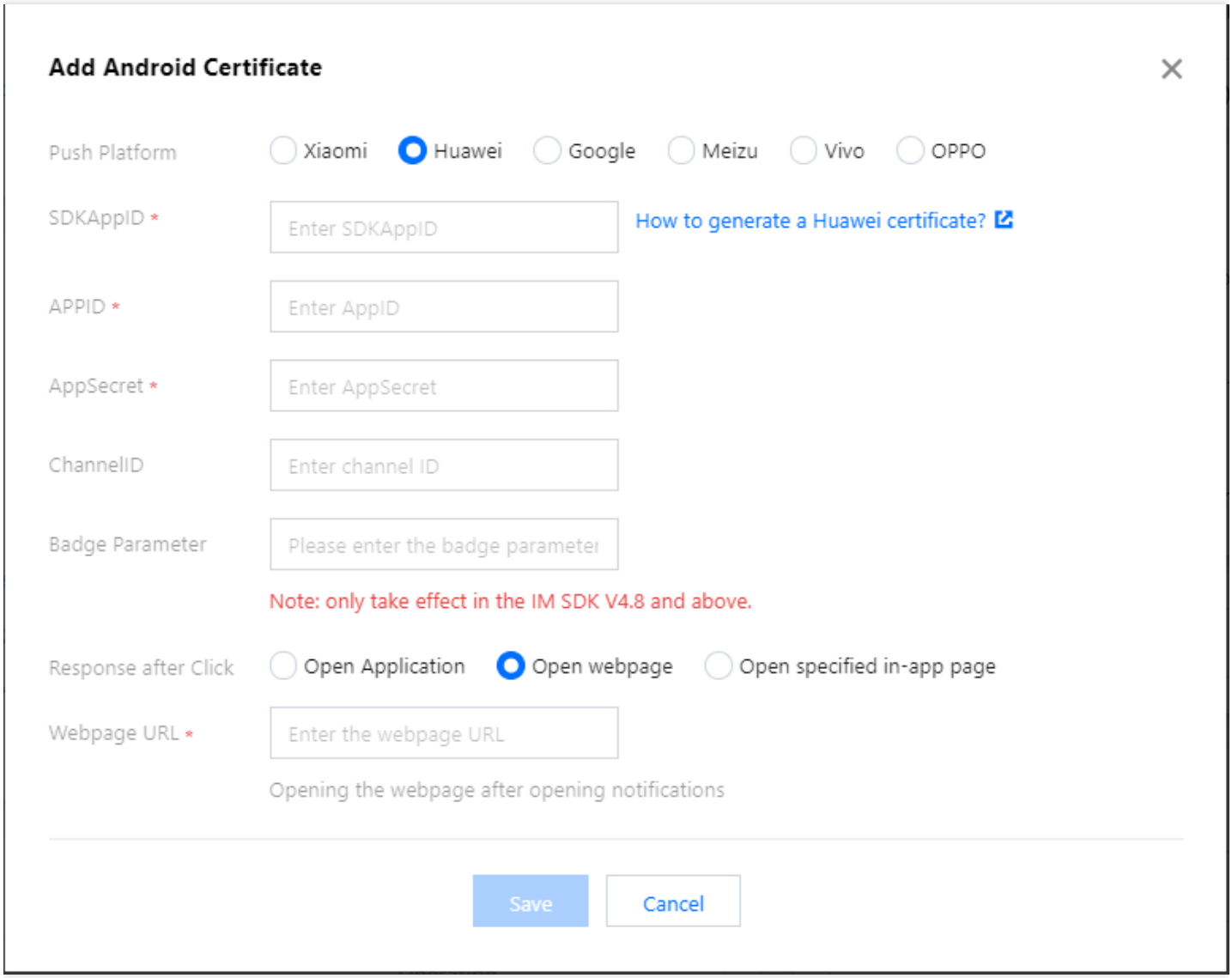
Open App

This is the default event, which opens the App once the notification bar message is clicked.



Open URL

You need to select Open URL in Step 2: Add a certificate and enter a URL that starts with either `http://` or `https://`, such as `https://cloud.tencent.com/document/product/269`.



Open specific App interface

1. **In manifest, configure the `intent-filter` of the Activity to be opened. The sample code is as follows:**

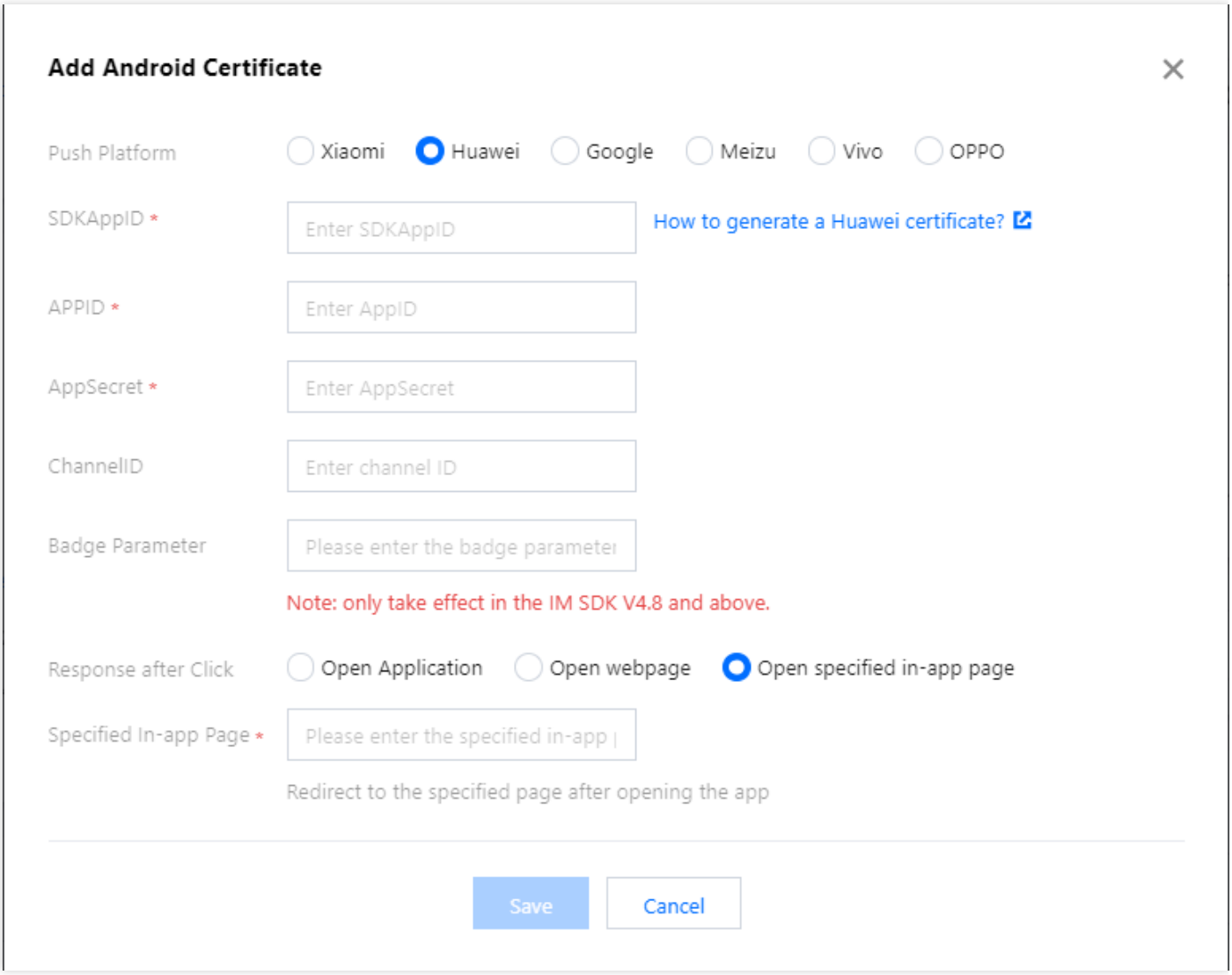
```
<activity
android:name="com.tencent.qcloud.tim.demo.chat.ChatActivity"
android:launchMode="singleTask"
android:screenOrientation="portrait"
android:windowSoftInputMode="adjustResize|stateHidden">

<intent-filter>
<action android:name="android.intent.action.VIEW" />
<data
android:host="com.tencent.qcloud.tim"
```

```
android:path="/detail"
android:scheme="pushscheme" />
</intent-filter>

</activity>
```

2. **Obtain the intent URL, as shown below:**

```
Intent intent = new Intent(this, ChatActivity.class);
intent.setData(Uri.parse("pushscheme://com.tencent.qcloud.tim/detail"));
intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
String intentUri = intent.toUri(Intent.URI_INTENT_SCHEME);
Log.i(TAG, "intentUri = " + intentUri);

// Print results
intent://com.tencent.qcloud.tim/detail#Intent;scheme=pushscheme;launchFlags=0x4000000;componen
t=com.tencent.qcloud.tim.tuikit/com.tencent.qcloud.tim.demo.chat.ChatActivity;end
```

3. **Select Open specific App interface in Step 2: Add a certificate and enter the result above.**



Custom Content Pass Through

Select Open App or Open specific App interface when configuring Click event in Step 2: Add a certificate to support custom content pass through.

Step 1: Custom content configuration (Sender)

Set the custom content for the notification bar message before sending the message.

- **Sample on Android:**

```
String extContent = "ext content";
TIMMessageOfflinePushSettings settings = new TIMMessageOfflinePushSettings();
settings.setExt(extContent.getBytes());
```

```
timMessage.setOfflinePushSettings(settings);
mConversation.sendMessage(false, timMessage, callback);
```

- **For information on configurations for the IM server, refer to the OfflinePushInfo Format Example.**

Step 2: Custom content configuration (receiver)

The client will obtain the custom content from the corresponding `Activity` once the notification bar message is clicked.

```
Bundle bundle = getIntent().getExtras();
String extContent = bundle.get("ext");
```

FAQs

If the App uses obfuscation, how can I prevent exceptions in the Huawei offline push feature?

If your App uses obfuscation, to prevent exceptions in the Huawei offline push feature, you need to keep the custom BroadcastReceiver and add obfuscation rules by referring to the following:

Note :

The following code is an official sample from Huawei. Please modify it according to your actual situation before use.

```
-ignorewarning
-keepattributes *Annotation*
-keepattributes Exceptions
-keepattributes InnerClasses
-keepattributes Signature
-keepattributes SourceFile,LineNumberTable
-keep class com.hianalytics.android.**{*;}
-keep class com.huawei.updatesdk.**{*;}
-keep class com.huawei.hms.**{*;}
-keep class com.huawei.android.hms.agent.**{*;}
# Change com.tencent.qcloud.tim.demo.thirdpush.HUAWEIPushReceiver to the complete class name defi
ned in your App.
-keep com.tencent.qcloud.tim.demo.thirdpush.HUAWEIPushReceiver {*;}
```

Can I set a custom notification sound?

Huawei does not support custom notification sounds.

I cannot receive push messages. What should I do?

1. **No push service is 100% successful in reaching target users, and vendor push is no exception. Therefore, if one or two push messages fail to reach users during a fast, continuous push process, it is usually due to the restrictions of vendor push frequency control.**
2. **According to the push process, confirm whether the Huawei push certificate information is correctly configured on the IM console.**
3. **Confirm that your project's Huawei push SDK integration configuration is correct and that you have obtained the token.**
4. **Confirm that you have reported push information to the IM server correctly.**
5. **Manually kill the App on your device, send a few messages, and confirm whether you receive notifications within one minute.**

Offline Push (Google FCM)

Last updated : 2020-12-29 17:40:20

> **Note :**
>
> To use FCM offline push, you need to install Google Play Services on your mobile phone and use it outside Mainland China.

Offline Push Process

The process of implementing offline message push is as follows:

1. **Register with the vendor and complete the developer verification process. Apply to enable the push service.**
2. **Create a push service and bind app information to obtain the push certificate, password, key, and other data.**
3. **Log in to the IM Console to upload the certificate and enter other required information. The IM server uses the certificate to generate a unique certificate ID.**
4. **Integrate the push messaging SDK provided by the vendor with your project and configure it according to the vendor's instructions.**
5. **Send your certificate ID and device information to IM server.**
6. **When the client App is killed by the system or user without IM logout, the IM server will remind the user via message push.**

Procedure

Step 1: set Firebase and FCM SDK

> **Note :**
>
> The website in this step is the official website of Firebase, which is accessible only outside Mainland China.

1. **Refer to Firebase Cloud Message Transfer to set Firebase and integrate the FCM SDK. After launching the app, obtain the device registration token.**

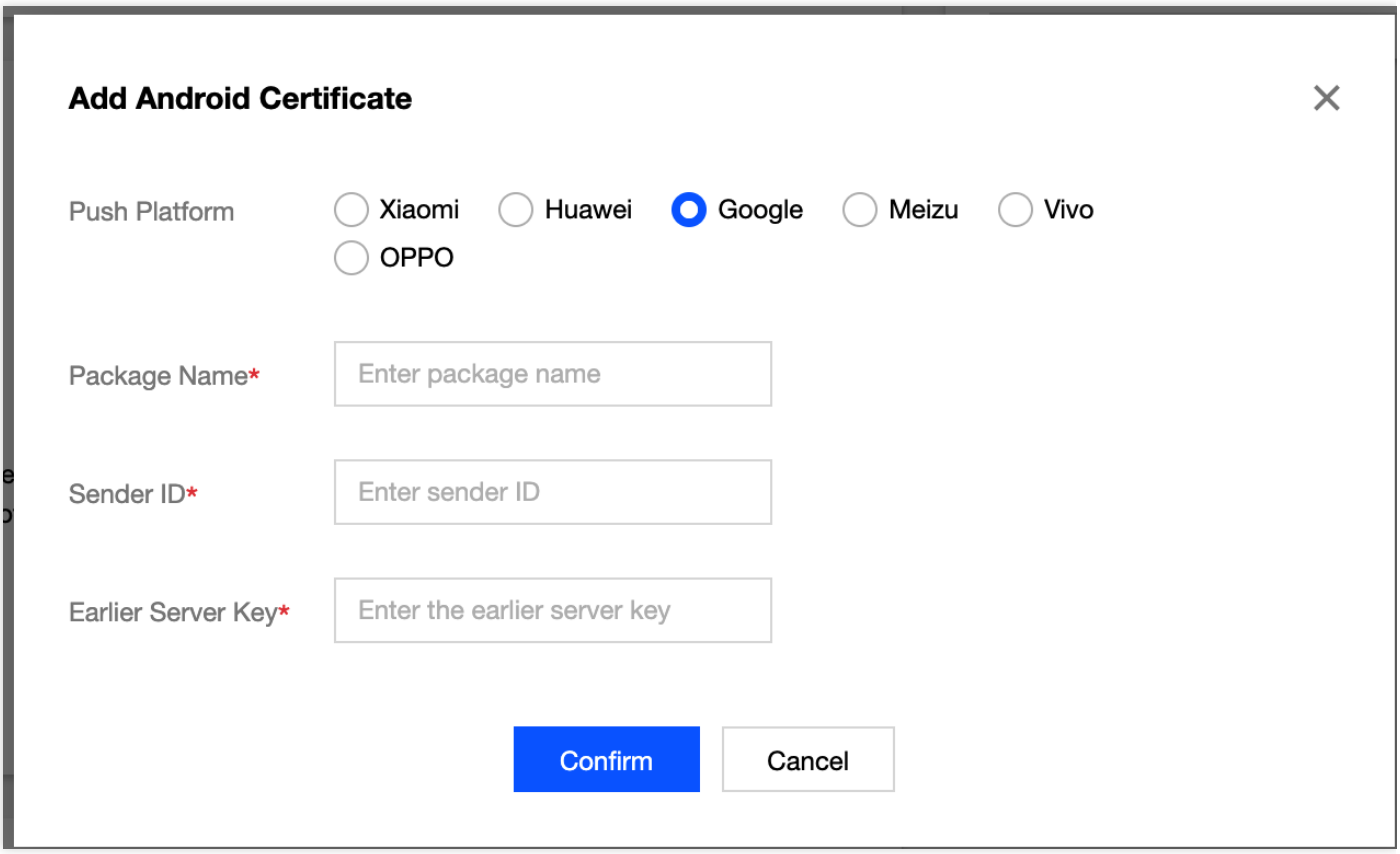
2. Refer to FCM Test Guide to test notification messages and make sure that FCM has been integrated successfully.
3. Log in to the Firebase console and click your app card to enter the app configuration page.
4. Click on the right side of Project Overview, and choose Project Settings -> Service Account.
5. Click Generate New Private Key to download the private key.

Step 2: generate a certificate ID

1. Log in to the IM Console and click the desired app. The app configuration page appears.
2. Click Add Certificate under Android Platform Push Settings.

> **Note :**
>
> You already have a certificate and only need to change its information, click Edit.

3. Upload the private key you obtained in step 1.



4. Click OK to save the information. Certificate information takes effect 10 minutes after you save it.
5. Record the Certificate ID once it is generated.

Step 3: report the push information to the IM server

After users successfully log in, use the `setOfflinePushToken` method of `TIMManager` to report the Certificate ID, generated and hosted by the IM console, and token, generated by the client after FCM integration, to the IM server.

> ⚠ **Note :**
>
> After the token and certificate ID are correctly reported, the IM service can bind users with the corresponding device information, thus enabling the use of FCM to push notifications.

The following is sample code from the demo:

Define the certificate ID constant:

```
/****** FCM offline push parameter start ******/
// Use your certificate ID in the FCM push certificate information on the IM console
public static final long GOOGLE_FCM_PUSH_BUZID = 6768;
/****** FCM offline push parameter end ******/
```

Reporting the push certificate ID and token:

```
/**
* Report the push certificate ID and device information in ThirdPushTokenMgr.java
*/
public class ThirdPushTokenMgr {
private static final String TAG = "ThirdPushTokenMgr";
private String mThirdPushToken;

public static ThirdPushTokenMgr getInstance () {
return ThirdPushTokenHolder.instance;
}

private static class ThirdPushTokenHolder {
private static final ThirdPushTokenMgr instance = new ThirdPushTokenMgr();
}

public String getThirdPushToken() {
return mThirdPushToken;
}

public void setThirdPushToken(String mThirdPushToken) {
this.mThirdPushToken = mThirdPushToken; // Token value specified here
}

public void setPushTokenToTIM(){
String token = ThirdPushTokenMgr.getInstance().getThirdPushToken();
if(TextUtils.isEmpty(token)){
QLog.i(TAG, "setPushTokenToTIM third token is empty");
mIsTokenSet = false;
return;
}
TIMOfflinePushToken param = new TIMOfflinePushToken(Constants.GOOGLE_FCM_PUSH_BUZID, token);
TIMManager.getInstance().setOfflinePushToken(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.d(TAG, "setOfflinePushToken err code = " + code);
}

@Override
public void onSuccess() {
```

```
Log.d(TAG, "setOfflinePushToken success");
mIsTokenSet = true;
}
});
}
}
```

Step 4: offline push

After the certificate ID and token are successfully reported, the IM server sends messages via FCM push notifications to the user before IM user logout on the device and even if the app is killed.

Note :

- **FCM push is not 100% successful in reaching target users.**
- **FCM push may be delayed. Usually, this is related to the timing of app killing. In some cases, it is related to the FCM push service.**
- **If the IM user has logged out or been forced offline by the IM server (for example, due to login on another device), the device cannot receive pushed messages.**

Custom Content Pass Through

Step 1: custom content configuration (Sender)

Set the custom content for the notification bar message before sending the message.

- **Android sample:**

```
String extContent = "ext content";

TIMMessageOfflinePushSettings settings = new TIMMessageOfflinePushSettings();
settings.setExt(extContent.getBytes());
timMessage.setOfflinePushSettings(settings);
mConversation.sendMessage(false, timMessage, callback);
```

- **For information on configurations for the IM server, refer to the OfflinePushInfo Format Example.**

Step 2: custom content configuration (receiver)

The client will obtain the custom content from the corresponding `Activity` once the notification bar message is clicked.

```
Bundle bundle = getIntent().getExtras();
String value = bundle.getString("ext");
```

FAQs

Can I set a custom notification sound?

Currently, FCM push does not support custom alert sounds.

I cannot receive pushed messages. What should I do?

1. **No push service is 100% successful in reaching target users, and FCM is no exception. Therefore, if 1 or 2 pushed messages fail to reach users during a fast, continuous push process, it is usually due to the restrictions of FCM's push frequency control.**
2. **According to the push process, confirm whether the FCM push certificate information is correctly configured on the IM console.**
3. **Confirm that your FCM project has been configured correctly and has obtained a token normally.**
4. **Confirm that you have reported push information to the IM server correctly.**
5. **Manually kill the App on your device, send a few messages, and confirm whether you receive notifications within 1 minute.**

Offline Push (Meizu)

Last updated : 2020-09-25 14:39:05

Process Description

The process of implementing offline message push is as follows:

1. **A developer registers an account on a vendor's platform, and after passing developer verification, the developer applies for the push service.**
2. **Create the push service, bind it with an app, and obtain information such as the push certificate, password, and key.**
3. **Log in to IM Console and specify the push certificate and relevant information. The IM server generates a different certificate ID for each certificate.**
4. **Integrate the push SDK provided by the vendor with the developer's project and configure it based on the vendor's requirements.**
5. **After integrating the IM SDK with the project, report the certificate ID, device information, and other information to the IM server.**
6. **When the client app is killed by the system or user without IM logout, the IM server sends notifications to the user through message push.**

Directions

Flyme is a highly customized Android system, with very strict management of the auto-start permissions of third-party apps. By default, third-party apps are not included in the auto-start allowlist of the system. As apps running in the background are often killed by the system, we recommend that Meizu push be integrated on Meizu devices. Meizu push is a system-grade service of Flyme, with a high push delivery rate. Currently, IM only supports the notification bar messages of Meizu push.

⚠ **Note :**

- **This document was prepared with direct reference to the official documentation of Meizu. If Meizu push is updated, refer to Meizu push documentation on the official website.**
- **This document was prepared based on the Flyme push access guide. It is intended for the Flyme system only and is not a unified push platform for Meizu (but the integration for different vendors).**

- If you do not need to implement special offline push adaptation for Meizu devices, ignore this section.

Step 1: Apply for a Meizu push certificate

1. Access the Meizu open platform website to register an account and pass developer verification.

> **Note :**
>
> The verification process takes about 3 days. Be sure to read the Meizu Push Service Activation Guide to facilitate access to the service.

2. Log in to the console of the Meizu open platform, choose Service > Integrate Push Service > Push Backend, and create a Meizu push service app.
   After the Meizu push service app is created, you can view detailed app information on the app details page.
3. Record the `App package name`, `App ID`, and `App Secret` items.

Step 2: Host the certificate to IM

1. Log in to Tencent Cloud IM Console and click the target app card to enter the basic configuration page of the app.

2. Click Add Certificate in the Android Platform Push Settings area.

> **Note :**
>
> If you already have a certificate and only want to modify its information, you can click Edit in the Android Platform Push Settings area to modify and update the certificate.

3. Set the following parameters based on the information obtained in Step 1:
   - Push Platform: select Meizu.
   - App Package Name: enter the App package name of the Meizu push service app.
   - AppID: enter the App ID of the Meizu push service app.
   - AppSecret: enter the App Secret of the Meizu push service app.
   - After Clicking Notification: select the response operation when users click notification bar messages. Available options are Open App, Open Web Page, and

Open Specified Interface in App. For more details, see Configuring the Notification Bar Message Click Event.

4. **Click OK to save the settings. The certificate information will take effect within 10 minutes after being saved.**

5. **Record the `ID` of the certificate after the push certificate information is generated.**

Step 3: Integrate the push SDK

Note :

- **The default notification title for IM push messages is `a new message` .**
- **Before reading this section, ensure that you have correctly integrated and used the IM SDK.**
- **You can find a sample for Meizu push implementation in our demo. Note that the features of Meizu push may be adjusted during Meizu push version updates. If you find any inconsistencies with the content of this section, refer to Meizu push documentation on the official website and notify us of the difference so that we can make the necessary modifications.**

Step 3.1: Download the Meizu push SDK and add references

Access the Meizu push operation platform and download the aar package of the Meizu Flyme push SDK or use jcenter integration.

```
dependencies {
// MEIZU push sdk
compile 'com.meizu.flyme.internet:push-internal:3.6.+@aar'
}
```

Step 3.2: Configure the AndroidManifest.xml file

Add the permissions required for Meizu push:

```
<!-- ********Start of Meizu push permission settings******** -->
<!-- The following settings are compatible with versions earlier than Flyme5.0 and are required f
or Meizu's internal pushSDK integration; otherwise, messages cannot be received-->
<uses-permission android:name="com.meizu.flyme.push.permission.RECEIVE"></uses-permission>
<permission
android:name="com.tencent.qcloud.tim.tuikit.push.permission.MESSAGE"
android:protectionLevel="signature"/>
```

```
<uses-permission android:name="com.tencent.qcloud.tim.tuikit.push.permission.MESSAGE"></uses-perm
ission>

<!-- The following settings are compatible with Flyme3.0 -->
<uses-permission android:name="com.meizu.c2dm.permission.RECEIVE" />
<permission
android:name="com.tencent.qcloud.tim.tuikit.permission.C2D_MESSAGE"
android:protectionLevel="signature"></permission>
<uses-permission android:name="com.tencent.qcloud.tim.tuikit.permission.C2D_MESSAGE"/>
<!-- ********End of Meizu push permission settings******** -->
<!--Here, replace com.tencent.qcloud.tim.tuikit with the package name of your app-->
```

Step 3.3: Customize a BroadcastReceiver class

To receive messages, you need to customize a BroadcastReceiver that is inherited from the `MzPushMessageReceiver` class, implement the `onRegisterStatus` method in it, and register this receiver to AndroidManifest.xml.

Sample code in the demo:

```
public class MEIZUPushReceiver extends MzPushMessageReceiver {
private static final String TAG = "MEIZUPushReceiver";

@Override
public void onRegisterStatus(Context context, RegisterStatus registerStatus) {
QLog.i(TAG, "onRegisterStatus token = " + registerStatus.getPushId());
ThirdPushTokenMgr.getInstance().setThirdPushToken(registerStatus.getPushId());
ThirdPushTokenMgr.getInstance().setPushTokenToTIM();
}

}
```

Register the custom BroadcastReceiver to AndroidManifest.xml:

```
<!--Here, change com.tencent.qcloud.tim.demo.thirdpush.MEIZUPushReceiver to the complete class na
me in your app -->
<!-- ********Start of Meizu push settings******** -->
<receiver android:name="com.tencent.qcloud.tim.demo.thirdpush.MEIZUPushReceiver">
<intent-filter>
<!-- Receive push messages -->
<action android:name="com.meizu.flyme.push.intent.MESSAGE" />
<!-- Receive register messages -->
<action android:name="com.meizu.flyme.push.intent.REGISTER.FEEDBACK" />
<!-- Receive unregister messages -->
<action android:name="com.meizu.flyme.push.intent.UNREGISTER.FEEDBACK"/>
<!-- The following settings are compatible with earlier Flyme3 push service configurations -->
```

```
<action android:name="com.meizu.c2dm.intent.REGISTRATION" />
<action android:name="com.meizu.c2dm.intent.RECEIVE" />
<category android:name="com.tencent.qcloud.tim.demo.thirdpush"></category>
</intent-filter>
</receiver>
<!-- ********End of Meizu push settings******** -->
```

Step 3.4: Register the Meizu push service in the app

If you choose to enable Meizu offline push, you need to register the Meizu push service with the Meizu server by calling `PushManager.register` to initialize the Meizu push service. `PushManager.register` can be called anywhere. To enhance the registration success rate, we recommend that you call it in `onCreate` of Application.
After successful registration, you will receive the registration result in `onRegisterStatus` of the BroadcastReceiver customized in Step 3.3. `registerStatus.getPushId()` indicates the unique identifier of the current app on the current device, and the `PushId` information needs to be recorded.

Sample code in the demo:

```
public class DemoApplication extends Application {

private static PojoApplication instance;

@Override
public void onCreate() {
super.onCreate();
// Determine whether the current thread is the main thread
if (SessionWrapper.isMainProcess(getApplicationContext())) {
/**
* TUIKit initialization function
*
* @param context App context, which usually corresponds to ApplicationContext
* @param sdkAppID SDKAppID assigned to the app that you registered in Tencent Cloud
* @param configs Relevant configuration items of TUIKit. Usually, you can use the default configu
ration. For special configurations, refer to "API Documentation".
*/
long current = System.currentTimeMillis();
TUIKit.init(this, Constants.SDKAPPID, BaseUIKitConfigs.getDefaultConfigs());
System.out.println(">>>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));
// Add custom initial configuration
customConfig();
System.out.println(">>>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));

if(IMFunc.isBrandXiaoMi()){
```

```
// MI offline push
MiPushClient.registerPush(this, Constants.XM_PUSH_APPID, Constants.XM_PUSH_APPKEY);
}
if(IMFunc.isBrandHuawei()){
// Huawei offline push
HMSAgent.init(this);
}
if(MzSystemUtils.isBrandMeizu(this)){
// Meizu offline push
PushManager.register(this, Constants.MZ_PUSH_APPID, Constants.MZ_PUSH_APPKEY);
}
if(IMFunc.isBrandVivo()){
// vivo offline push
PushClient.getInstance(getApplicationContext()).initialize();
}
}
instance = this;
}
}
```

Step 4: Report the push information to the IM server

If you need to use Meizu push to push IM message notifications, after successful user login, you must use the `setOfflinePushToken` method of `TIMManager` to report the certificate ID generated and hosted by the IM console and PushID returned by the Meizu push service, to the IM server.

> ⚠ **Note :**
>
> **After the PushId and certificate ID are correctly reported, IM service binds users with the corresponding device information. This enables the use of the Meizu push service to push notifications.**

Sample code in the demo:

- **Define the certificate ID constant:**

```
/**
* First, define some constant information in Constants.java
*/
/****** Start of Meizu offline push parameters ******/
// Use your certificate ID in the Meizu push certificate information in the IM console
public static final long MZ_PUSH_BUZID = 6666;
// APPID and APPKEY assigned by the Meizu open platform
```

```
public static final String MZ_PUSH_APPID = "1234512345123451234";
public static final String MZ_PUSH_APPKEY = "1234512345123";
/****** End of Meizu offline push parameters ******/
```

- **Report the push certificate ID and PushId:**

```
/**
* Report the push certificate ID and device information in ThirdPushTokenMgr.java
*/
public class ThirdPushTokenMgr {
private static final String TAG = "ThirdPushTokenMgr";

private String mThirdPushToken;

private boolean mIsTokenSet = false;
private boolean mIsLogin = false;

public static ThirdPushTokenMgr getInstance () {
return ThirdPushTokenHolder.instance;
}

private static class ThirdPushTokenHolder {
private static final ThirdPushTokenMgr instance = new ThirdPushTokenMgr();
}

public void setIsLogin(boolean isLogin){
mIsLogin = isLogin;
}

public String getThirdPushToken() {
return mThirdPushToken;
}

public void setThirdPushToken(String mThirdPushToken) {
this.mThirdPushToken = mThirdPushToken; // Here, the PushId value is specified. Describe it ba
sed on the aforementioned custom BroadcastReciever class documentation.
}

public void setPushTokenToTIM(){
if(mIsTokenSet){
QLog.i(TAG, "setPushTokenToTIM mIsTokenSet true, ignore");
return;
}
String token = ThirdPushTokenMgr.getInstance().getThirdPushToken();
if(TextUtils.isEmpty(token)){
QLog.i(TAG, "setPushTokenToTIM third token is empty");
```

```
mIsTokenSet = false;
return;
}
if( !mIsLogin ){
QLog.i(TAG, "setPushTokenToTIM not login, ignore");
return;
}
TIMOfflinePushToken param = null;
if(IMFunc.isBrandXiaoMi()){ // Identify the vendor brand and choose different push services fo
r different vendors
param = new TIMOfflinePushToken(Constants.XM_PUSH_BUZID, token);
}else if(IMFunc.isBrandHuawei()){
param = new TIMOfflinePushToken(Constants.HW_PUSH_BUZID, token);
}else if(IMFunc.isBrandMeizu()){
param = new TIMOfflinePushToken(Constants.MZ_PUSH_BUZID, token);
}else if(IMFunc.isBrandOppo()){
param = new TIMOfflinePushToken(Constants.OPPO_PUSH_BUZID, token);
}else if(IMFunc.isBrandVivo()){
param = new TIMOfflinePushToken(Constants.VIVO_PUSH_BUZID, token);
}else{
return;
}
TIMManager.getInstance().setOfflinePushToken(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.d(TAG, "setOfflinePushToken err code = " + code);
}

@Override
public void onSuccess() {
Log.d(TAG, "setOfflinePushToken success");
mIsTokenSet = true;
}
});
}
}
```

Step 5: Offline push

After the certificate ID and PushID are successfully reported, the IM server sends messages through Meizu push notifications to the user when the app has been killed but the user has not logged out of IM.

Note :

- **Meizu push does not guarantee 100% success in reaching target users.**

- **Meizu push may be delayed. Usually, this is related to the timing of app killing. In some cases, it is related to the Meizu push service.**
- **If the IM user has logged out or was forced logout by the IM server (for example, due to login on another terminal), the device will not receive push messages.**

Configuring the Notification Bar Message Click Event

You can choose to Open App, Open Web Page, or Open Specified Interface in App to follow the notification bar message click event.

Opening the app

The default option is Open App.

Opening webpages

When adding a certificate, you need to select Open Web Page and enter a website URL starting with `http://` or `https://`, for example, `https://intl.cloud.tencent.com/document/product/269`.

Opening a specified UI in the app

When adding a certificate, you need to select Open Specified Interface in App and enter the complete class name of Activity to be opened, for example, `com.tencent.qcloud.tim.demo.chat.ChatActivity`.

FAQs

If the app uses obfuscation, how can I prevent exceptions when using the Meizu offline push feature?

If your app uses obfuscation, to prevent exceptions when using the Meizu offline push feature, you need to keep the custom BroadcastReceiver and add obfuscation rules by referring to the following:

Note :

The following code is an official sample from Meizu. Please modify it based on your actual situation before use.

```
# Change com.tencent.qcloud.tim.demo.thirdpush.MEIZUPushReceiver to the complete class name defined in your app.
-keep com.tencent.qcloud.tim.demo.thirdpush.MEIZUPushReceiver {*;}
```

Can I customize push notification sounds?

Currently, Meizu push does not support custom notification sounds.

How can I identify the cause to failures to receive push messages?

1. **No push service guarantees 100% success in reaching target users and zero vendor push exceptions. Therefore, if one or two push messages fail to reach users during a fast and continuous push process, it is usually due to the restrictions of vendor push frequency control.**
2. **According to the push process, confirm whether the Meizu push certificate information is correctly configured in IM Console.**
3. **Confirm that your project's Meizu push SDK integration configuration is correct and that you have obtained the PushId.**
4. **Confirm that you have reported push information to the IM server correctly.**
5. **Manually kill the app on your device, send several messages, and check whether you can receive notifications within one minute.**

Offline Push (vivo)

Last updated : 2021-01-26 17:14:37

Process

The following is the process of offline message push:

1. **Register with the vendor and complete the developer verification process. Apply to enable the push service.**
2. **Create a push service and bind app information to obtain the push certificate, password, key, and other data.**
3. **Log in to the IM Console to upload the certificate and enter other required information. The IM server uses the certificate to generate a unique certificate ID.**
4. **Integrate the push messaging SDK provided by the vendor with your project and configure it according to the vendor's instructions.**
5. **Send your certificate ID and device information to IM server.**
6. **If the user did not log out of IM but the client was terminated by the system or user, the IM server will send push messages as notifications.**

Procedure

vivo mobile phones use a highly customized Android system, with very strict management of the auto-start permissions of third-party apps. By default, third-party apps are not placed in the auto-start allowlist of the system. As apps running in the background are often killed by the system, we recommend that vivo push be integrated on vivo devices. vivo push is a system-grade service for vivo devices, with a high delivery rate. Currently, IM only supports the notification bar messages of vivo push.

⚠ **Note :**

- **This guide was prepared with direct reference to the official documentation of vivo push. If vivo push is changed, please refer to the vivo push documentation on the official website.**
- **If you do not plan to implement a vivo-specific offline push solution, skip this section.**

Step 1: Apply for a vivo Push certificate

1. Visit the vivo open platform official website and register for an account. Complete developer verification.

> **Note :**
>
> The verification process takes about 3 days. Be sure to read the vivo push service description beforehand to facilitate access to the service.

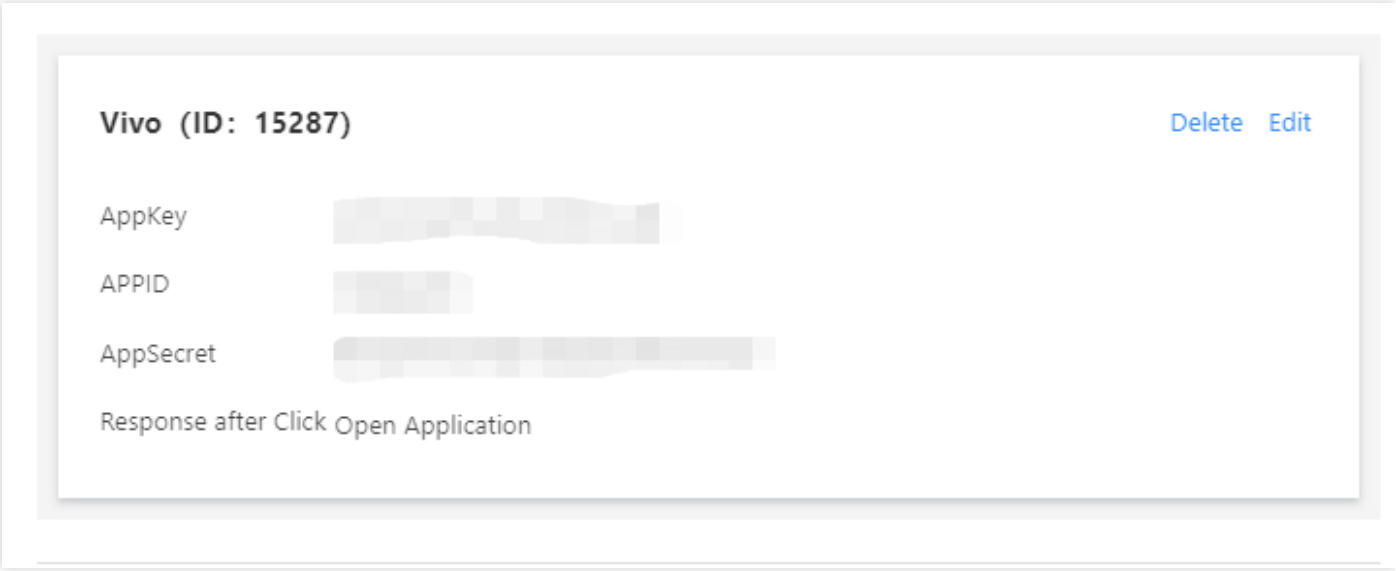
2. Log in to the console of the vivo open platform, choose Message Push -> Create -> Test Push, and create a vivo push service app.
   Once the app is created, you can view detailed app information under App details.
3. Record the following: `APP ID`, `APP key`, and `APP secret`.

Step 2: Generate a Certificate ID

1. Log in to the IM Console and click the desired app. The app configuration page appears.

2. Click Add a certificate under Android push configuration.

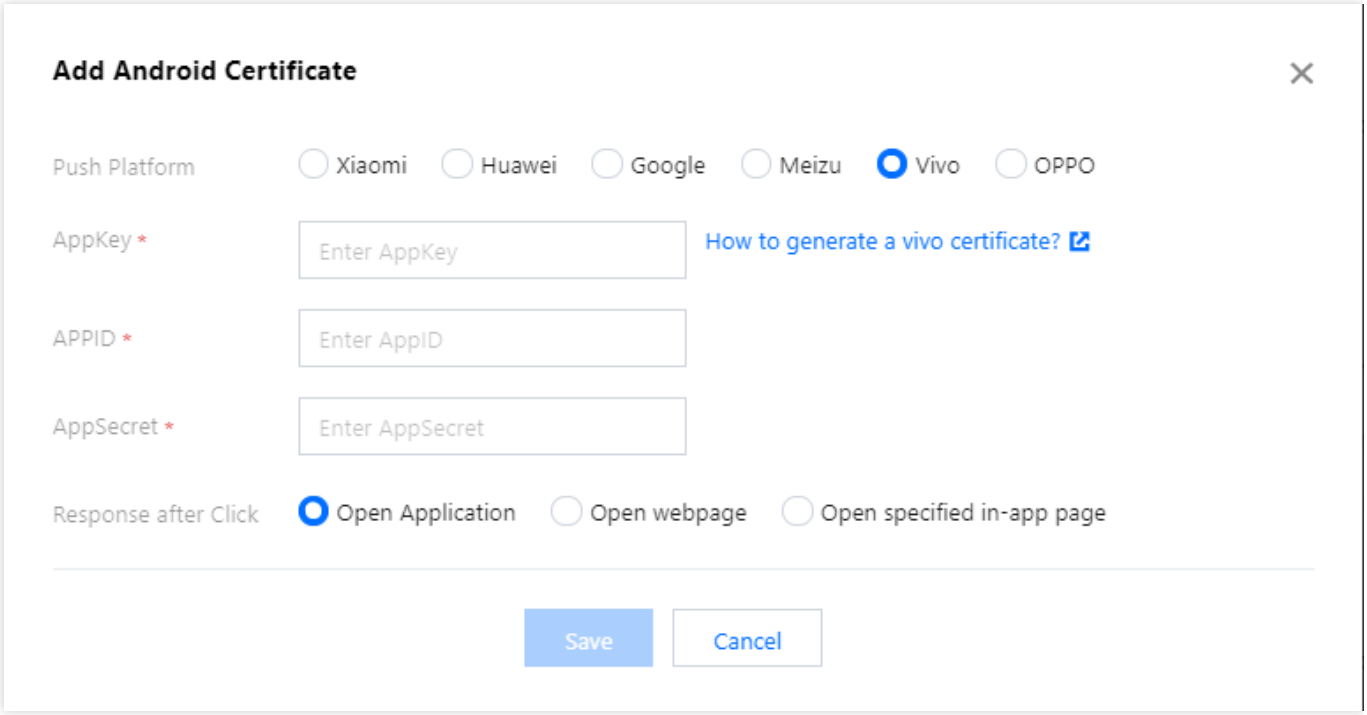
> **Note :**
>
> If you already have a certificate and only want to change its information, you can click Edit in the corresponding certificate area to modify and update the certificate.



3. **Use the information you obtained in Step 1 to configure the following parameters:**

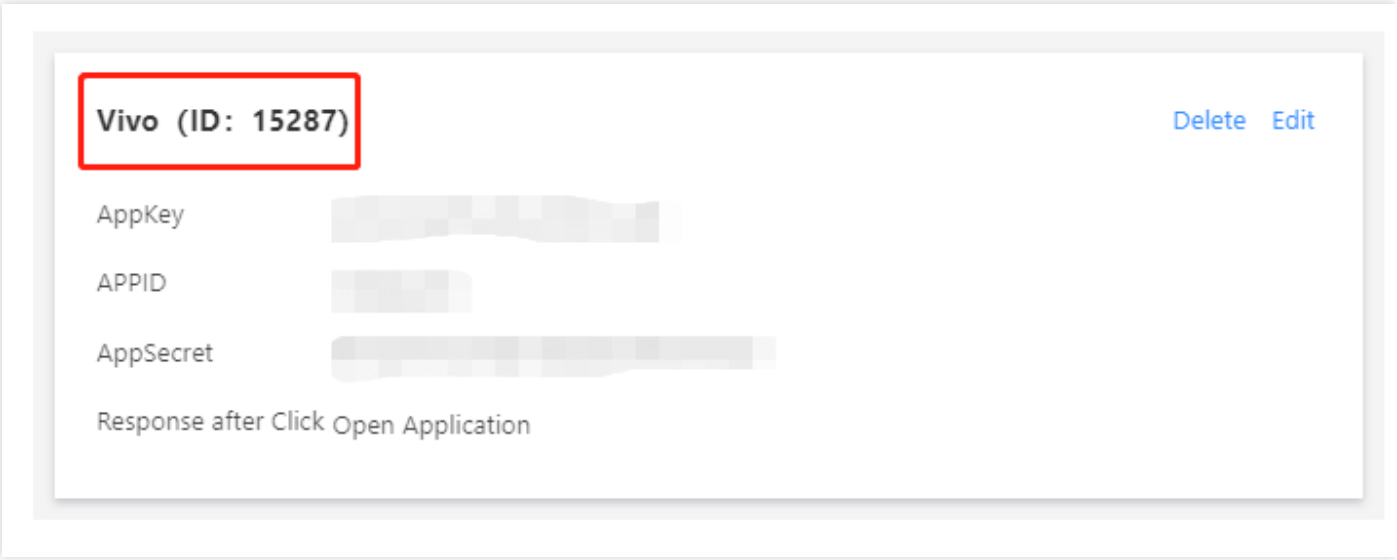
   - **Push platform: select vivo.**
   - **AppKey: enter the AppKey you got from vivo Push.**
   - **AppID: enter the AppID you got from vivo Push.**
   - **AppSecret: enter the APP secret you got from vivo Push.**
   - **Click event: the event to take place after the notification bar message is clicked. Valid values include Open app, Open URL, and Open specific app interface. For more information, refer to Configuring Click Event.**

Open app or Open specific app interface allows custom content pass through.



4. **Click OK to save the information. Certificate information takes effect 10 minutes after you save it.**

5. **Record the Certificate ID once it is generated.**



Step 3: Integrate push SDK

Note :

- **The default title of IM push notifications is `a new message`.**
- **Before reading this section, make sure that you have integrated and tested the IM SDK.**
- **You can find a sample for implementation of vivo push in our demo. Note that the features of vivo push may be adjusted during vivo push version updates. If you find any inconsistencies with the content of this section, please refer to the vivo push documentation on the official website and notify us of the difference so that we can make the necessary modifications.**

Step 3.1: Download the vivo Push SDK and reference it in your project

1. **Use vivo Push platform and download the SDK.**
2. **Decompress the SDK package and find the library file named `vivo_pushsdk_xxx.jar`.**
3. **Copy `vivo_pushsdk_xxx.jar` to the library folder (`libs`) of your project and add a reference to it in your project.**

Step 3.2: Modify AndroidManifest.xml

Open AndroidManifest.xml in a text editor and add the following:

```
<!-- ********vivo Push configuration start******** -->
<service
android:name="com.vivo.push.sdk.service.CommandClientService"
android:exported="true" />
<activity
android:name="com.vivo.push.sdk.LinkProxyClientActivity"
android:exported="false"
android:screenOrientation="portrait"
android:theme="@android:style/Theme.Translucent.NoTitleBar" />
<meta-data
android:name="com.vivo.push.api_key"
android:value="a90685ff-ebad-4df3-a265-3d4bb8e3a389" />
<meta-data
android:name="com.vivo.push.app_id"
android:value="11178" />
<!-- ********vivo Push configuration end******** -->
<!--com.vivo.push.app_id and com.vivo.push.api_key are generated by the vivo Push open platform--
>
```

Step 3.3: Define a BroadcastReceiver class

In order to receive messages, you need to define a BroadcastReceiver class which inherits `OpenClientPushMessageReceiver` and implements the `onReceiveRegId` and `onNotificationMessageClicked` methods. Also, register the BroadcastReceiver in AndroidManifest.xml.

The following is sample code from the demo:

```
public class VIVOPushMessageReceiverImpl extends OpenClientPushMessageReceiver {
private static final String TAG = "VIVOPushMessageReceiver";
@Override
public void onNotificationMessageClicked(Context context, UPSNotificationMessage upsNotificationM
essage) {
Log.i(TAG, "onNotificationMessageClicked");
}

@Override
public void onReceiveRegId(Context context, String regId) {
// Use this method as a callback if vivo regId changes. According to official vivo documentation,
to obtain regId, you need to call PushClient.getInstance(getApplicationContext()).getRegId() in t
he enable push callback. Also see LoginActivity.
Log.i(TAG, "onReceiveRegId = " + regId);
}
}
```

Register the custom BroadcastReceiver to AndroidManifest.xml:

```
<!--Change com.tencent.qcloud.tim.demo.thirdpush.VIVOPushMessageReceiverImpl to the full class na
me in your app-->
<!-- Deceleration of message receiver-->
<receiver android:name="com.tencent.qcloud.tim.demo.thirdpush.VIVOPushMessageReceiverImpl">
<intent-filter>
<!-- Receive push messages -->
<action android:name="com.vivo.pushclient.action.RECEIVE" />
</intent-filter>
</receiver>
```

Step 3.4: Register vivo Push in your app

To use vivo offline push, you need to register your push service with vivo's server. To do this, use `PushClient.getInstance(getApplicationContext()).initialize()` to initialize your push service. `PushClient.getInstance(getApplicationContext()).initialize()` can be called anywhere in your code. However, to improve the registration success rate, the vivo official documentation suggests that you call it in your app's `onCreate` .

After the push service is registered, obtain the results in the main interface of your app. `regId` is the unique identifier of the app on the current device. Record it for later use.

The following is sample code from the demo:

```
public class DemoApplication extends Application {

private static PojoApplication instance;

@Override
public void onCreate() {
super.onCreate();
// Determines whether this is the main thread
if (SessionWrapper.isMainProcess(getApplicationContext())) {
/**
* Initializes TUIKit
*
* @param context App context, usually corresponds to ApplicationContext
* @param sdkAppID, the SDKAppID assigned to you when registering the app in Tencent Cloud
* @param configs, TUIKit configuration options. The default values are suitable in most cases. Se
e the API documentation if you want to customize them.
*/
long current = System.currentTimeMillis();
TUIKit.init(this, Constants.SDKAPPID, BaseUIKitConfigs.getDefaultConfigs());
System.out.println(">>>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));
// Add custom initialization configuration
customConfig();
System.out.println(">>>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));

if(IMFunc.isBrandXiaoMi()){
// Xiaomi offline push
MiPushClient.registerPush(this, Constants.XM_PUSH_APPID, Constants.XM_PUSH_APPKEY);
}
if(IMFunc.isBrandHuawei()){
// Huawei offline push
HMSAgent.init(this);
}
if(MzSystemUtils.isBrandMeizu(this)){
// Meizu offline push
PushManager.register(this, Constants.MZ_PUSH_APPID, Constants.MZ_PUSH_APPKEY);
}
if(IMFunc.isBrandVivo()){
// vivo offline push
PushClient.getInstance(getApplicationContext()).initialize();
}
}
instance = this;
```

```
}
}
```

Open vivo Push in the main interface

```
if (IMFunc.isBrandVivo()) {
    // vivo offline push
    PushClient.getInstance(getApplicationContext()).turnOnPush(new IPushActionListener() {
        @Override
        public void onStateChanged(int state) {
            if (state == 0) {
                String regId = PushClient.getInstance(getApplicationContext()).getRegId();
                QLog.i(TAG, "vivopush open vivo push success regId = " + regId);
                ThirdPushTokenMgr.getInstance().setThirdPushToken(regId);
                ThirdPushTokenMgr.getInstance().setPushTokenToTIM();
            } else {
                // According to the vivo documentation, state = 101 means this particular vivo devic
                QLog.i(TAG, "vivopush open vivo push fail state = " + state);
            }
        }
    });
```

Step 4: Report the push information to the IM server

If you need to use vivo push to push IM message notifications, then after successful user login, you must use the `setOfflinePushToken` method of `TIMManager` to report the certificate ID generated and hosted by the IM console and regId returned by the vivo push service to the IM server.

> ⚠ **Note :**
>
> **After the regId and certificate ID are correctly reported, IM service binds users with the corresponding device information. This enables the use of the vivo push service to push notifications.**

The following is sample code from the demo:

- **Define Certificate ID as a constant:**

```
/**
* We first define some constant information in Constants.java.
*/
/****** vivo offline push parameters start ******/
// Certificate ID generated after uploading a third-party push certificate in the Tencent Clou
```

```
d console
public static final long VIVO_PUSH_BUZID = 6666;
// APPID and APPKEY assigned by the vivo open platform
public static final String VIVO_PUSH_APPID = "1234512345123451234"; // See the checklist
public static final String VIVO_PUSH_APPKEY = "12345abcde"; // See the checklist
/****** vivo offline push parameters end ******/
```

- **Report Certificate ID and regId:**

```
/**
* Report Certificate ID and regId to IM in ThirdPushTokenMgr.java
*/
public class ThirdPushTokenMgr {
private static final String TAG = "ThirdPushTokenMgr";

private String mThirdPushToken;

public static ThirdPushTokenMgr getInstance () {
return ThirdPushTokenHolder.instance;
}

private static class ThirdPushTokenHolder {
private static final ThirdPushTokenMgr instance = new ThirdPushTokenMgr();
}

public void setThirdPushToken(String mThirdPushToken) {
this.mThirdPushToken = mThirdPushToken; // The regId value is passed here Describe it in accor
dance with the above-mentioned custom BroadcastReciever class documentation.
}

public void setPushTokenToTIM(){
if(mIsTokenSet){
QLog.i(TAG, "setPushTokenToTIM mIsTokenSet true, ignore");
return;
}
String token = ThirdPushTokenMgr.getInstance().getThirdPushToken();
if(TextUtils.isEmpty(token)){
QLog.i(TAG, "setPushTokenToTIM third token is empty");
mIsTokenSet = false;
return;
}
if( !mIsLogin ){
QLog.i(TAG, "setPushTokenToTIM not login, ignore");
return;
}
TIMOfflinePushToken param = null;
```

```
if(IMFunc.isBrandXiaoMi()){ // Select different push services for different vendors.
param = new TIMOfflinePushToken(Constants.XM_PUSH_BUZID, token);
}else if(IMFunc.isBrandHuawei()){
param = new TIMOfflinePushToken(Constants.HW_PUSH_BUZID, token);
}else if(IMFunc.isBrandMeizu()){
param = new TIMOfflinePushToken(Constants.MZ_PUSH_BUZID, token);
}else if(IMFunc.isBrandOppo()){
param = new TIMOfflinePushToken(Constants.OPPO_PUSH_BUZID, token);
}else if(IMFunc.isBrandVivo()){
param = new TIMOfflinePushToken(Constants.VIVO_PUSH_BUZID, token);
}else{
return;
}
TIMManager.getInstance().setOfflinePushToken(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.d(TAG, "setOfflinePushToken err code = " + code);
}

@Override
public void onSuccess() {
Log.d(TAG, "setOfflinePushToken success");
mIsTokenSet = true;
}
});
}
}
```

Step 5: Offline push

After the certificate ID and regId are successfully reported, the IM server sends messages via vivo push notifications to the user when the app has been killed but the user has not logged out of IM.

Note :

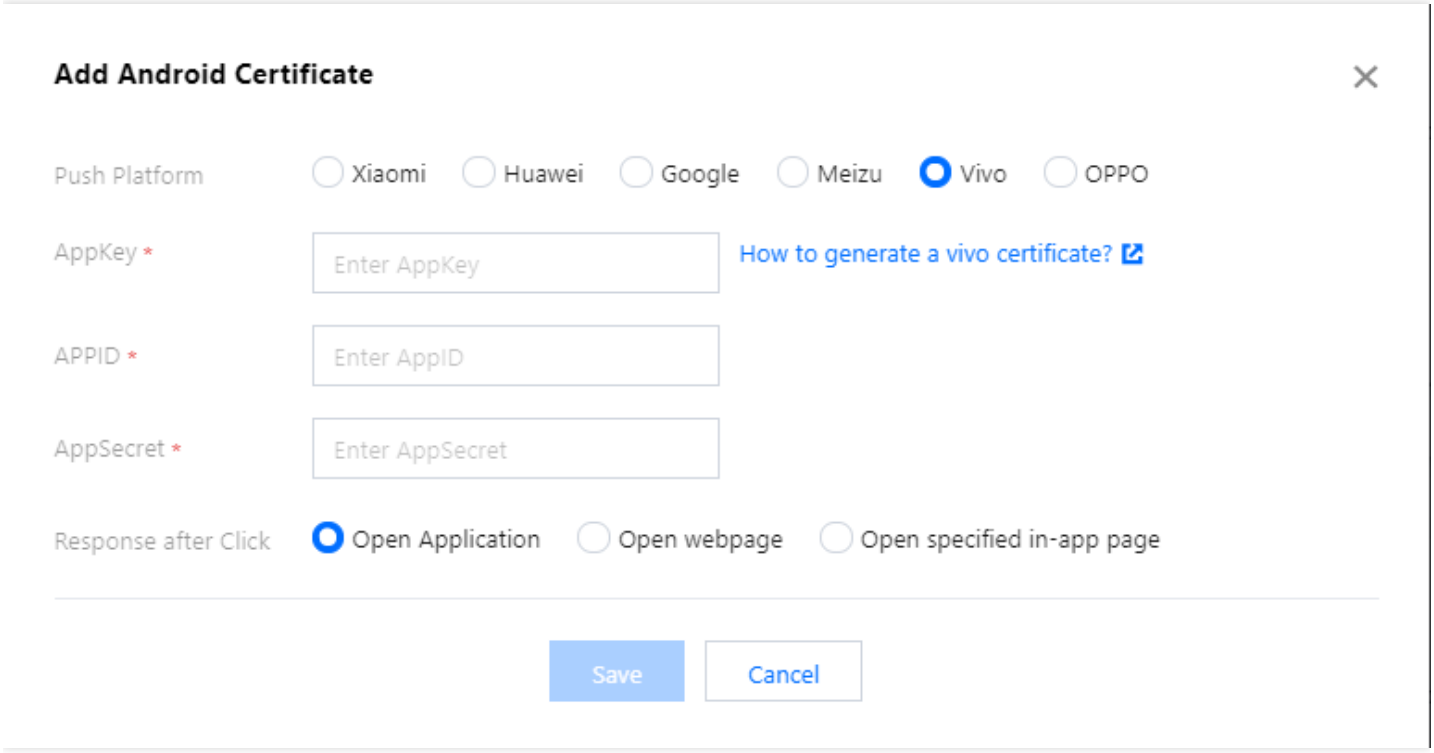
- **Not all vivo devices support vivo Push. For more information, see vivo Push FAQ.**
- **vivo push is not 100% successful in reaching the target users.**
- **vivo push may be delayed. Usually, this is related to the timing of app killing. In some cases, it is related to the vivo push service.**
- **If the user logs out, or is logged out by IM (such as when the user logs in on another device), the device will no longer receive push messages.**

Configuring Click Events

You can select one of the following events: Open app, Open URL, or Open specific app interface.

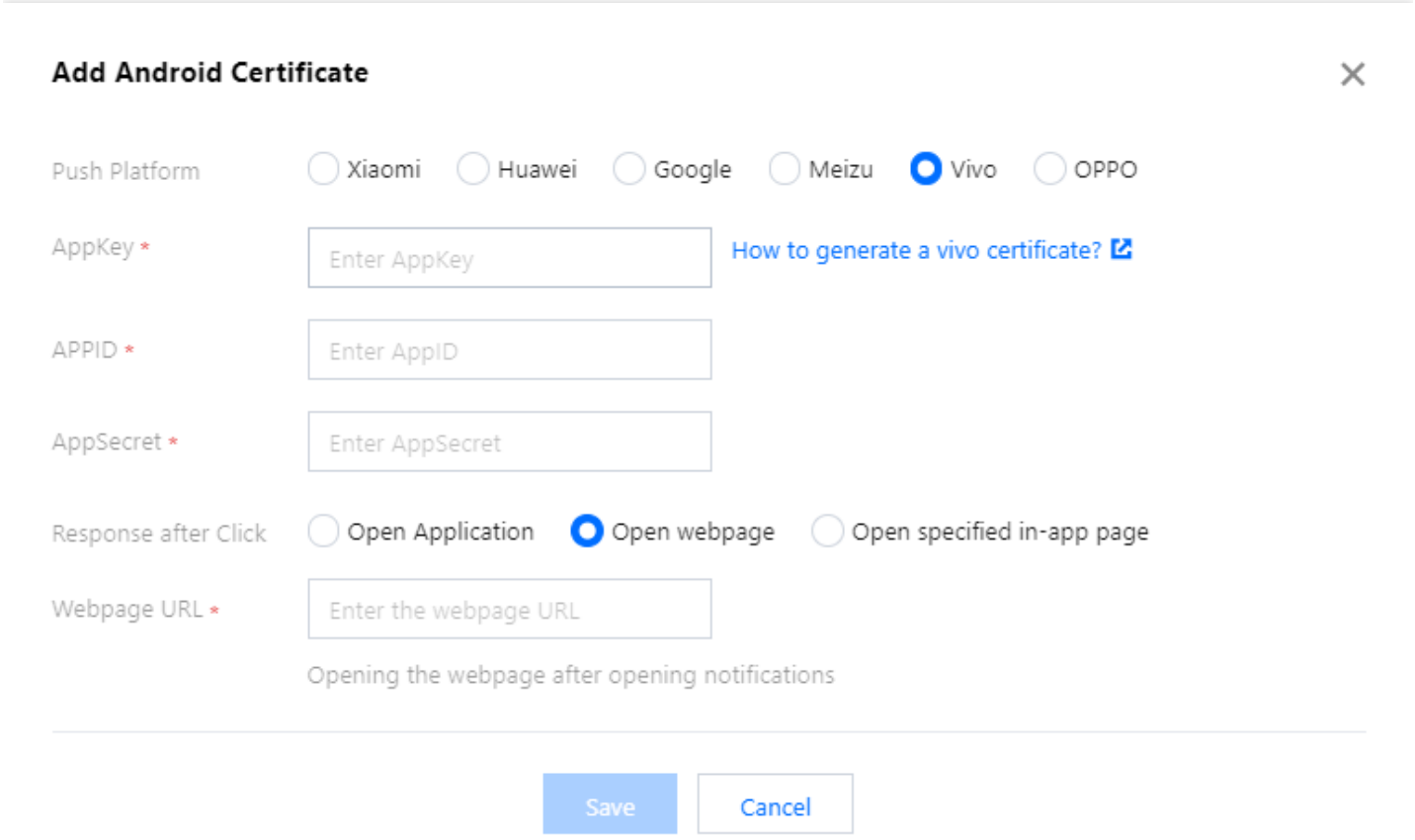
Open app

This is the default event, which opens the app once the notification bar message is clicked.



Open URL

You need to select Open URL in Step 2 and enter a URL that starts with either `http` or `https`, such as `https://cloud.tencent.com/document/product/269`.



Open specific app interface

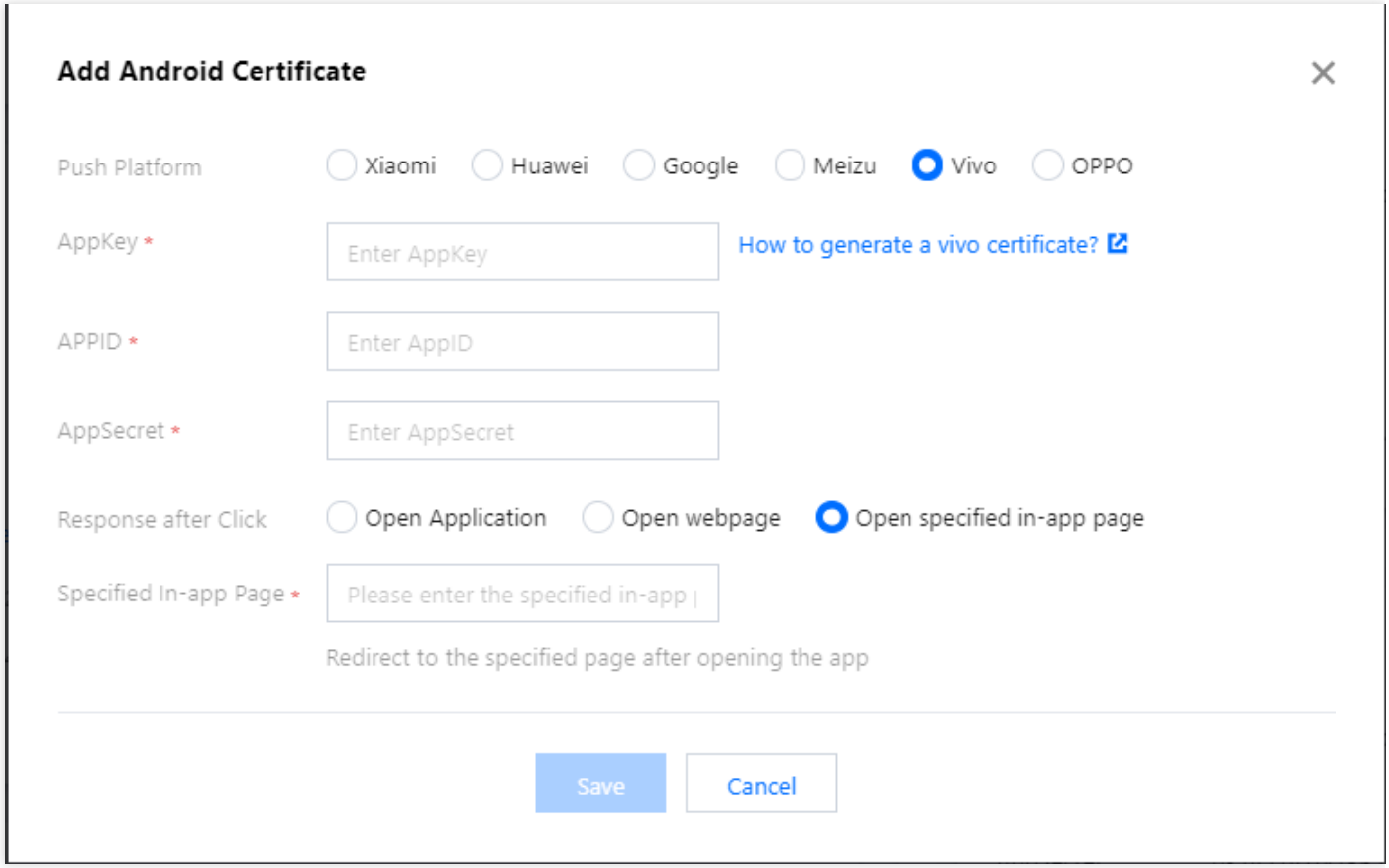
1. **Open manifest in a text editor and configure the `intent-filter` of the Activity you want to open as shown:**

```
<activity
android:name="com.tencent.qcloud.tim.demo.chat.ChatActivity"
android:launchMode="singleTask"
android:screenOrientation="portrait"
android:windowSoftInputMode="adjustResize|stateHidden">

<intent-filter>
<action android:name="android.intent.action.VIEW" />
<data
android:host="com.tencent.qcloud.tim"
android:path="/detail"
android:scheme="pushscheme" />
</intent-filter>

</activity>
```

2. Obtain the intent URL, as shown below:

```
Intent intent = new Intent(this, ChatActivity.class);
intent.setData(Uri.parse("pushscheme://com.tencent.qcloud.tim/detail"));
intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
String intentUri = intent.toUri(Intent.URI_INTENT_SCHEME);
Log.i(TAG, "intentUri = " + intentUri);

// Print results
intent://com.tencent.qcloud.tim/detail#Intent;scheme=pushscheme;launchFlags=0x4000000;component=com.tencent.qcloud.tim.tuikit/com.tencent.qcloud.tim.demo.chat.ChatActivity;end
```

3. Select Open specific app interface in Step 2 and enter the result above.



Custom Content Pass Through

Select Open app or Open specific app interface when configuring Click event in Step 2 to use custom content pass through.

Step 1: Custom content configuration (Sender)

Set the custom content for the notification bar message before sending the message.

- **Android sample:**

```
String extContent = "ext content";
TIMMessageOfflinePushSettings settings = new TIMMessageOfflinePushSettings();
settings.setExt(extContent.getBytes());
timMessage.setOfflinePushSettings(settings);
mConversation.sendMessage(false, timMessage, callback);
```

- **For information on configurations for the IM server, refer to the OfflinePushInfo Format Example.**

Step 2: Custom content configuration (receiver)

Clicking a notification bar message triggers a callback of `onNotificationMessageClicked(Context, UPSNotificationMessage upsNotificationMessage)` , which is part of the vivo Push SDK. The custom content can be obtained from the value of `upsNotificationMessage` .

```
Map<String, String> paramMap = upsNotificationMessage.getParams();
String extContent = paramMap.get("ext");
```

FAQ

If the app uses obfuscation, how can I prevent exceptions in the vivo offline push feature?

If your app uses obfuscation, to prevent exceptions in the vivo offline push feature, you need to keep the custom BroadcastReceiver and add obfuscation rules by referring to the following:

> **Note :**
>
> **The following code is an official sample from vivo. Please modify it according to your actual situation before use.**

```
# Change com.tencent.qcloud.tim.demo.thirdpush.VIVOPushMessageReceiverImpl to the complete class name defined in your app.
# vivo Push
-dontwarn com.vivo.push.**
-keep class com.vivo.push.**{*; }
-keep class com.vivo.vms.**{*; }
-keep class com.tencent.qcloud.tim.demo.thirdpush.VIVOPushMessageReceiverImpl{*;}
```

Can I set a custom notification sound?

vivo does not support custom notification sounds.

I cannot receive push messages. What should I do?

1. **No push service is 100% successful in reaching target users, and vendor push is no exception. Therefore, if one or two push messages fail to reach users during a fast, continuous push process, it is usually due to the restrictions of vendor push frequency control.**
2. **Make sure the correct push certificate information from vivo is properly configured in the IM Console.**
3. **Confirm that your project's vivo push SDK integration configuration is correct and that you have obtained the regId.**
4. **Confirm that you have reported push information to the IM server correctly.**
5. **Manually kill the app on your device, send a few messages, and confirm whether you receive notifications within one minute.**

Offline Push (OPPO)

Last updated : 2020-12-29 17:23:34

Process

The following is the process of offline message push:

1. **Register with the vendor and complete the developer verification process. Apply to enable the push service.**
2. **Create a push service and bind app information to obtain the push certificate, password, key, and other data.**
3. **Log in to the IM Console to upload the certificate and enter other required information. The IM server uses the certificate to generate a unique certificate ID.**
4. **Integrate the push messaging SDK provided by the vendor with your project and configure it according to the vendor's instructions.**
5. **Send your certificate ID and device information to IM server.**
6. **If the user did not log out of IM but the client was terminated by the system or user, the IM server will send push messages as notifications.**

Procedure

OPPO mobile phones use a highly customized Android system, with very strict management of the auto-start permissions of third-party apps. By default, third-party apps are not placed in the auto-start allowlist of the system. As apps running in the background are often killed by the system, we recommend that OPPO push be integrated on OPPO devices. OPPO push is a system-grade service for OPPO devices, with a high delivery rate. Currently, IM only supports the notification bar messages of OPPO push.

⚠ **Note :**

- **This guide was prepared with direct reference to the official documentation of OPPO push. If OPPO push is changed, please refer to the OPPO push documentation on the official website.**
- **If you do not plan to implement an OPPO-specific offline push solution, skip this section.**

Step 1: Apply for an OPPO PUSH certificate

1. Refer to How to enable OPPO PUSH for instructions on how to enable OPPO PUSH.
2. Navigate to OPPO PUSH > Configuration Management > Application Management for detailed app information.
3. Record the following: `AppId`, `AppKey`, `AppSecret`, and `MasterSecret`.

Step 2: Create a ChannelID

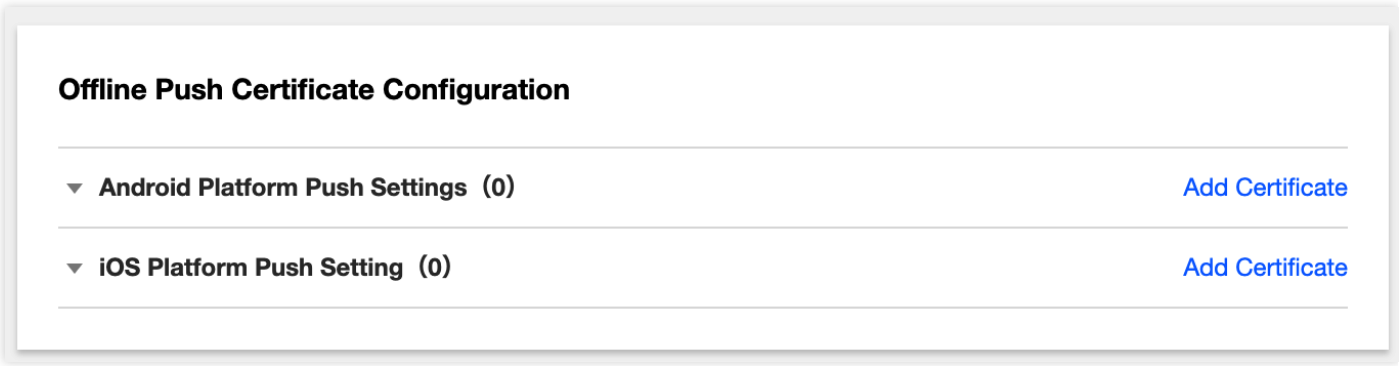
The official OPPO documentation states that ChannelIDs are required for push messages on OPPO Android 8.0 and above. Therefore, create a ChannelID for your app. Below is a sample code that creates a ChannelID called `tuikit`:

```
public void createNotificationChannel(Context context) {
// Create the NotificationChannel, but only on API 26+ because
// the NotificationChannel class is new and not in the support library
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
CharSequence name = "oppotest";
String description = "this is opptest";
int importance = NotificationManager.IMPORTANCE_DEFAULT;
NotificationChannel channel = new NotificationChannel("tuikit", name, importance);
channel.setDescription(description);
// Register the channel with the system; you can't change the importance
// or other notification behaviors after this
NotificationManager notificationManager = context.getSystemService(NotificationManager.class);
notificationManager.createNotificationChannel(channel);
}
}
```

Step 3: Generate a Certificate ID

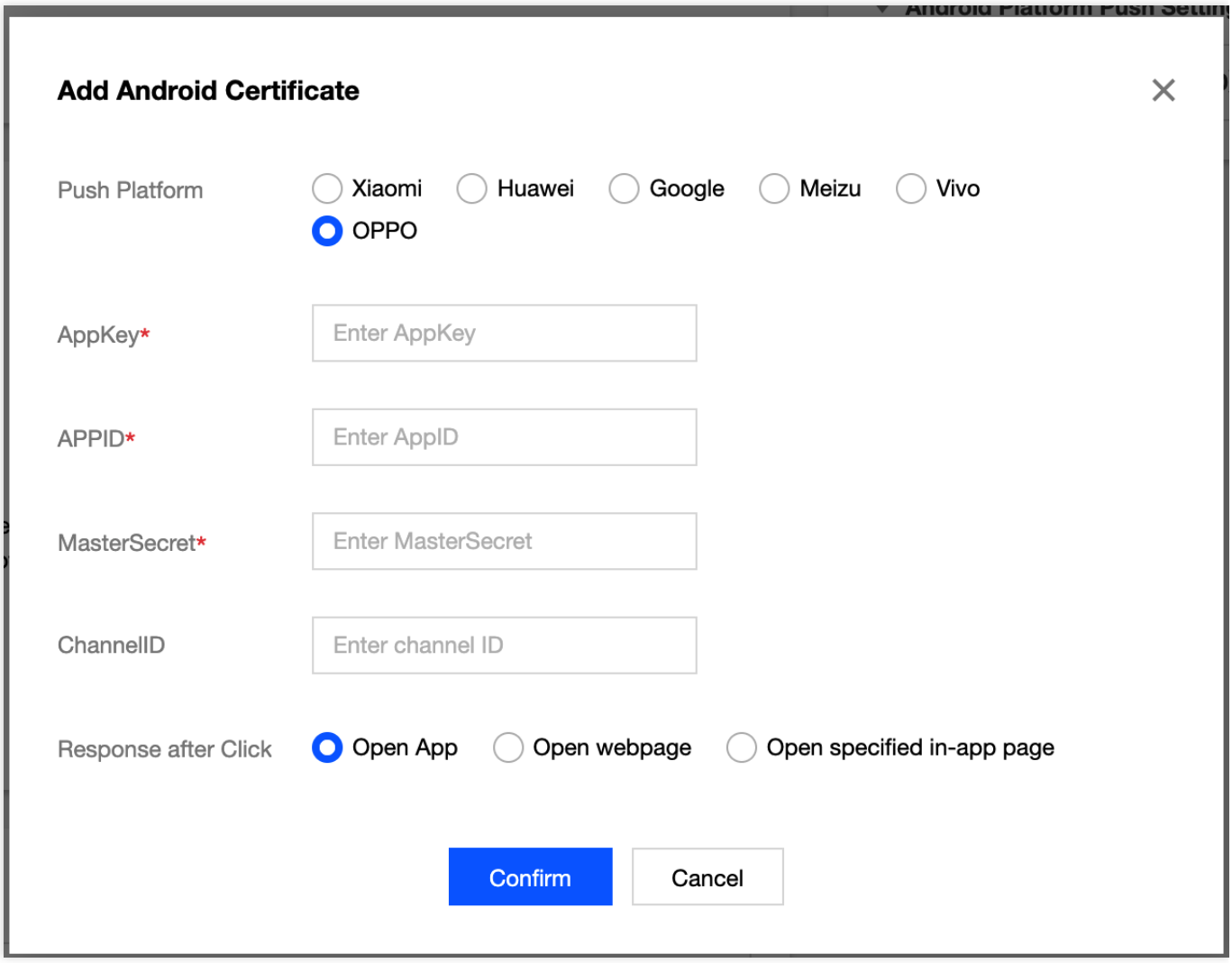
1. Log in to the IM Console and click the desired app. The app configuration page appears.
2. Click Add a certificate under Android push configuration.

> **Note :**
>
> If you already have a certificate and only need to change its information, click Edit.



3. **Use the information you obtained in Step 1 and Step 2 to configure the following parameters:**
   - **Push platform: select OPPO.**
   - **AppKey: enter the AppKey you got from OPPO PUSH.**
   - **AppID: enter the AppID you got from OPPO PUSH.**
   - **MasterSecret: enter the MasterSecret you got from OPPO PUSH.**
   - **ChannelID: enter the ChannelID generated in Step 2.**
   - **Click event: the event to take place after the notification bar message is clicked. Valid values include Open app, Open URL, and Open specific app interface. For more information, see Configuring Click Event.**
     Open app or Open specific app interface allows [custom content pass

through(#Trans).



4. Click OK to save the information. Certificate information takes effect 10 minutes after you save it.
5. Record the Certificate ID once it is generated.

Step 4: Integrate push SDK

1. Follow the instructions in the OPPO PUSH SDK API documentation to integrate the SDK. Use the OPPO console to test notification messages to ensure the SDK was integrated properly.
2. Use `PushManager.getInstance().register(…)` , which is part of the OPPO PUSH SDK, to initialize the Opush service.
   After the call is successfully registered, use `onRegister` , which is a `PushCallback` method, to obtain `regId` .
3. Record your `regId` .

Step 5: Report the push information to the IM server

If you need to use OPPO push to push IM message notification, then after successful user login, you must use the `setOfflinePushToken` method of `TIMManager` to report the certificate ID generated and hosted by the IM console and regId returned by the OPPO push service to the IM server.

> ⚠ **Note :**
>
> **After the regId and certificate ID are correctly reported, the IM service can bind users with the corresponding device information. This enables the use of the OPPO push service to push notifications.**

The following is a sample code defining Certificate ID as a constant:

```
/****** OPPO offline push parameter start ******/
// Certificate ID generated after uploading a third-party push certificate in the Tencent Cloud c
onsole
public static final long OPPO_PUSH_BUZID = 7005;
/****** OPPO offline push parameter start end ******/
```

The following is a sample code that reports Certificate ID and regId to the IM server:

```
/**
* Report Certificate ID and regId to IM in ThirdPushTokenMgr.java
*/
public class ThirdPushTokenMgr {
private static final String TAG = "ThirdPushTokenMgr";
private String mThirdPushToken;

public static ThirdPushTokenMgr getInstance () {
return ThirdPushTokenHolder.instance;
}
private static class ThirdPushTokenHolder {
private static final ThirdPushTokenMgr instance = new ThirdPushTokenMgr();
}

public void setThirdPushToken(String mThirdPushToken) {
this.mThirdPushToken = mThirdPushToken; // The regId value is passed here Describe it in accordan
ce with the above-mentioned custom BroadcastReciever class documentation.
}
public void setPushTokenToTIM(){
String token = ThirdPushTokenMgr.getInstance().getThirdPushToken();
if(TextUtils.isEmpty(token)){
QLog.i(TAG, "setPushTokenToTIM third token is empty");
```

```
mIsTokenSet = false;
return;
}
TIMOfflinePushToken param = null;
if(IMFunc.isBrandXiaoMi()){ // Select different push services for different vendors.
param = new TIMOfflinePushToken(Constants.XM_PUSH_BUZID, token);
}else if(IMFunc.isBrandHuawei()){
param = new TIMOfflinePushToken(Constants.HW_PUSH_BUZID, token);
}else if(IMFunc.isBrandMeizu()){
param = new TIMOfflinePushToken(Constants.MZ_PUSH_BUZID, token);
}else if(IMFunc.isBrandOppo()){
param = new TIMOfflinePushToken(Constants.OPPO_PUSH_BUZID, token);
}else if(IMFunc.isBrandVivo()){
param = new TIMOfflinePushToken(Constants.VIVO_PUSH_BUZID, token);
}else{
return;
}
TIMManager.getInstance().setOfflinePushToken(param, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.d(TAG, "setOfflinePushToken err code = " + code);
}
@Override
public void onSuccess() {
Log.d(TAG, "setOfflinePushToken success");
mIsTokenSet = true;
}
});
}
}
```

Step 6: Offline push

After the Certificate ID and regId are successfully sent, the IM server will push the notification to the client through OPPO PUSH when the app is killed by the system before the user logs out.

Note :

- **For a list of frequently asked questions, refer to OPPO PUSH FAQ.**
- **If the user logs out, or is logged out by IM (such as when the user logs in on another device), the device will no longer receive push messages.**

Configuring Click Events

You can select one of the following events: Open app, Open URL, or Open specific app interface.

Open app

This is the default event, which opens the app once the notification bar message is clicked.

Open URL

You need to select Open URL in Step 2 and enter a URL that starts with either `http` or `https`, such as `https://cloud.tencent.com/document/product/269`.

Open specific app interface

These are the ways you can open a specific app interface:

Activity (recommended)
This is rather simple. Enter the whole name of an Activity, such as `com.tencent.qcloud.tim.demo.SplashActivity`

Intent action

1. **Open AndroidManifest with a text editor and configure the Activity, as follows. You must add category but no data.**

```
<intent-filter>
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

2. **Enter `android.intent.action.VIEW` in the console.**

Custom Content Pass Through

Step 1: Custom content configuration (Sender)

Set the custom content for the notification bar message before sending the message.

> ⚠ **Note :**
>
> **OPPO requires custom data to be in JSON format.**

- **Android sample:**

```
JSONObject jsonObject = new JSONObject();
try {
jsonObject.put("extKey", "ext content");
} catch (JSONException e) {
e.printStackTrace();
}
String extContent = jsonObject.toString();

TIMMessageOfflinePushSettings settings = new TIMMessageOfflinePushSettings();
settings.setExt(extContent.getBytes());
timMessage.setOfflinePushSettings(settings);
mConversation.sendMessage(false, timMessage, callback);
```

- **For information on configurations for the IM server, refer to the OfflinePushInfo Format Example.**

Step 2: Custom content configuration (receiver)

On the console, after you set Open app or Open specific app interface as the click event for the push message and configure an Intent action or Activity, the client will be able to obtain the custom content from the corresponding `Activity` once the notification bar message is clicked.

```
Bundle bundle = intent.getExtras();
Set<String> set = bundle.keySet();
if (set != null) {
for (String key : set) {
// key and value correspond to extKey and ext content set in Step 1.
String value = bundle.getString(key);
Log.i("oppo push custom data", "key = " + key + ":value = " + value);
}
}
```

FAQ

Can I set a custom notification sound?

OPPO does not support custom notification sounds.

I cannot receive push messages. What should I do?

1. **No push service is 100% successful in reaching target users, and vendor push is no exception. Therefore, if one or two push messages fail to reach users during a fast, continuous push process, it is usually due to the restrictions of vendor push frequency control.**
2. **Make sure the correct push certificate information from OPPO is properly configured in the IM Console.**
3. **Confirm that your project's OPPO push SDK integration configuration is correct and that you have obtained the regId.**
4. **Confirm that you have reported push information to the IM server correctly.**
5. **Manually kill the app on your device, send a few messages, and confirm whether you receive notifications within one minute.**

Offline Push (iOS)
Obtaining Apple Push Notification Service Certificates

Last updated : 2021-02-25 11:51:11

Generating a CSR File

Follow these steps to generate a Certificate Signing Request (CSR).

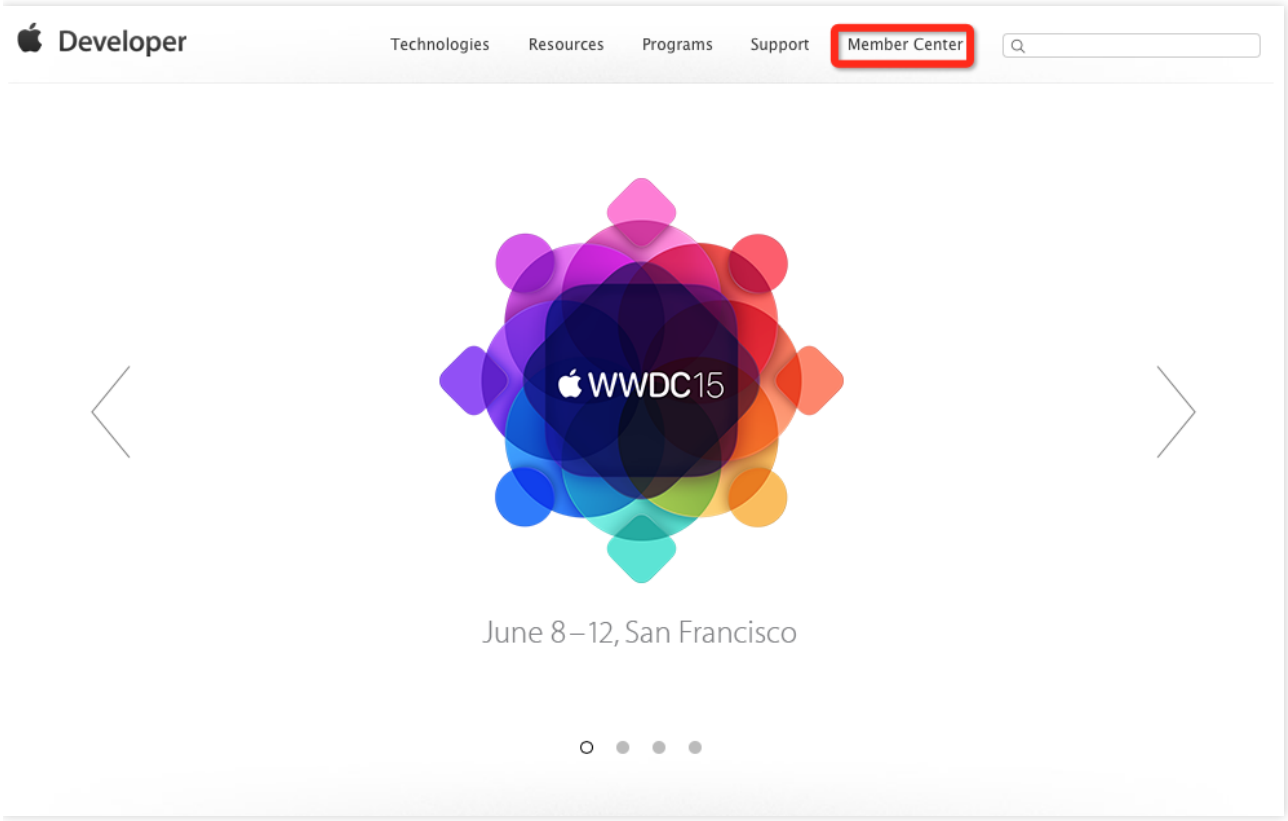
Specify the email address (of the paid account for applying for an AppID) and common name (the name of your computer by default, which does not need to be changed). Then, select "Save to disk".

Click "Continue".

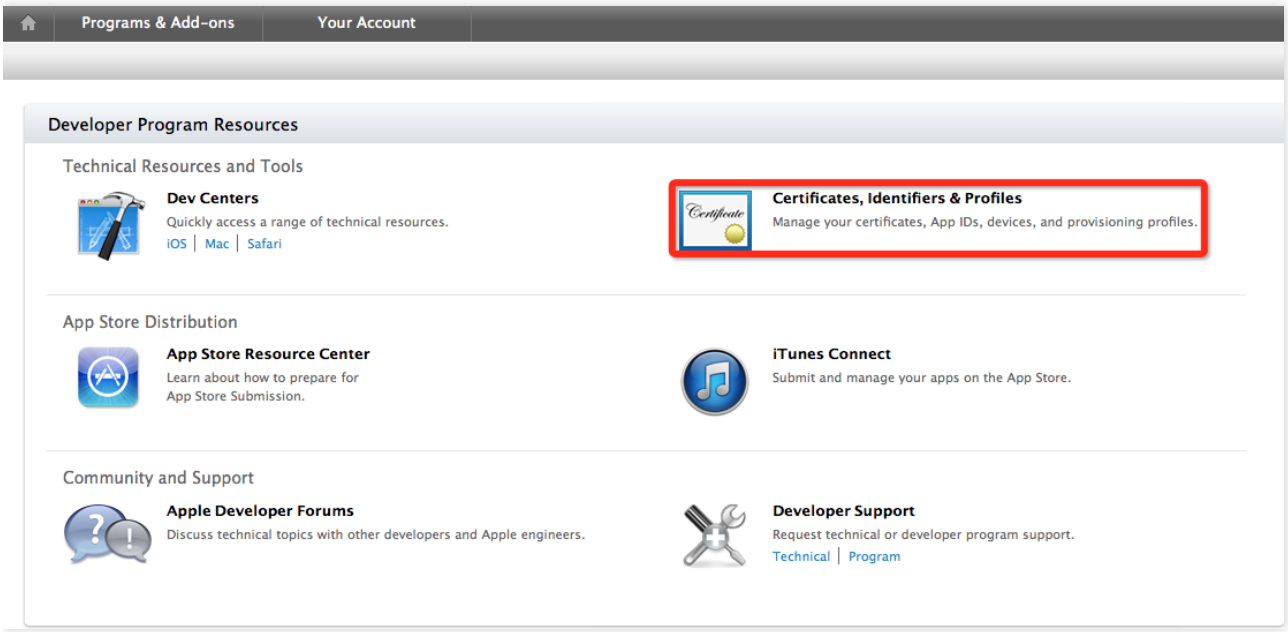
The CSR file TXIMDemoAPS.certSigningRequest is created locally.

Creating an App ID

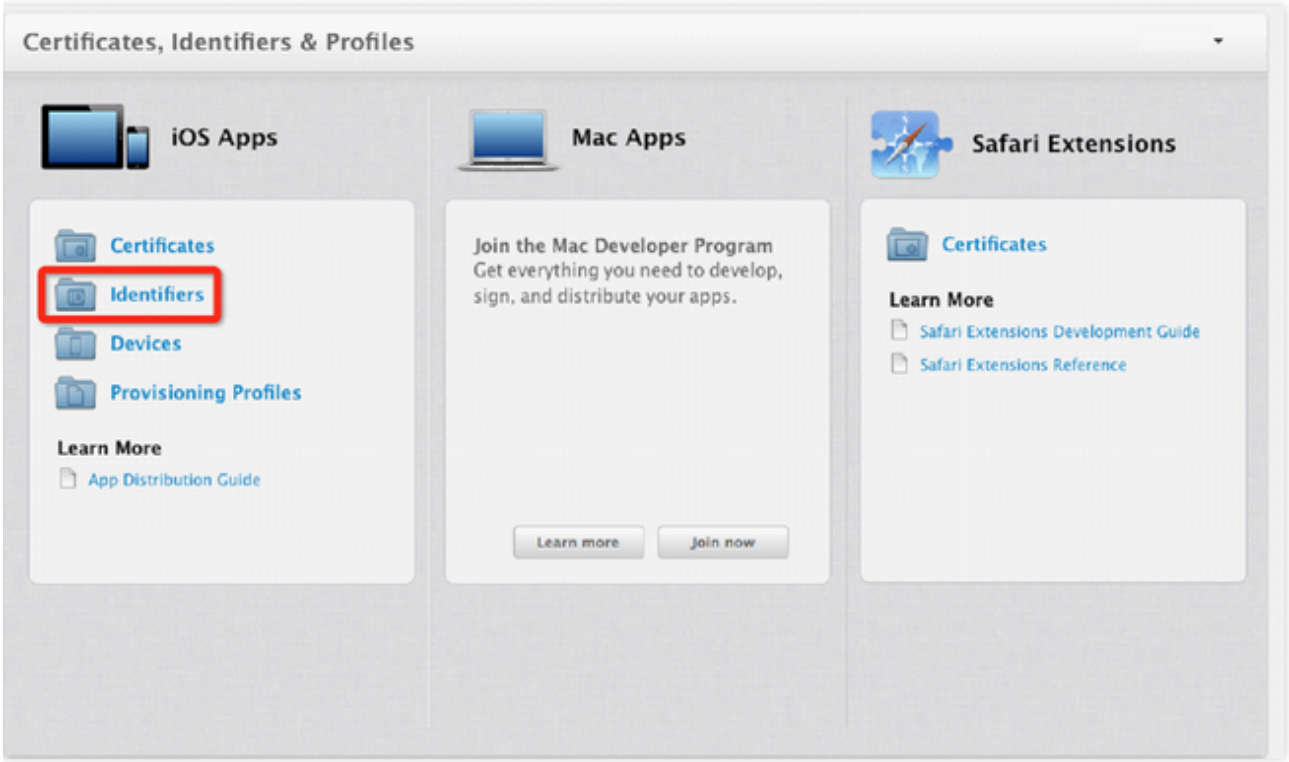
Log in to developer.apple.com and click "Member Center".



On the page that appears, select "Certificates, Identifiers & Profiles".



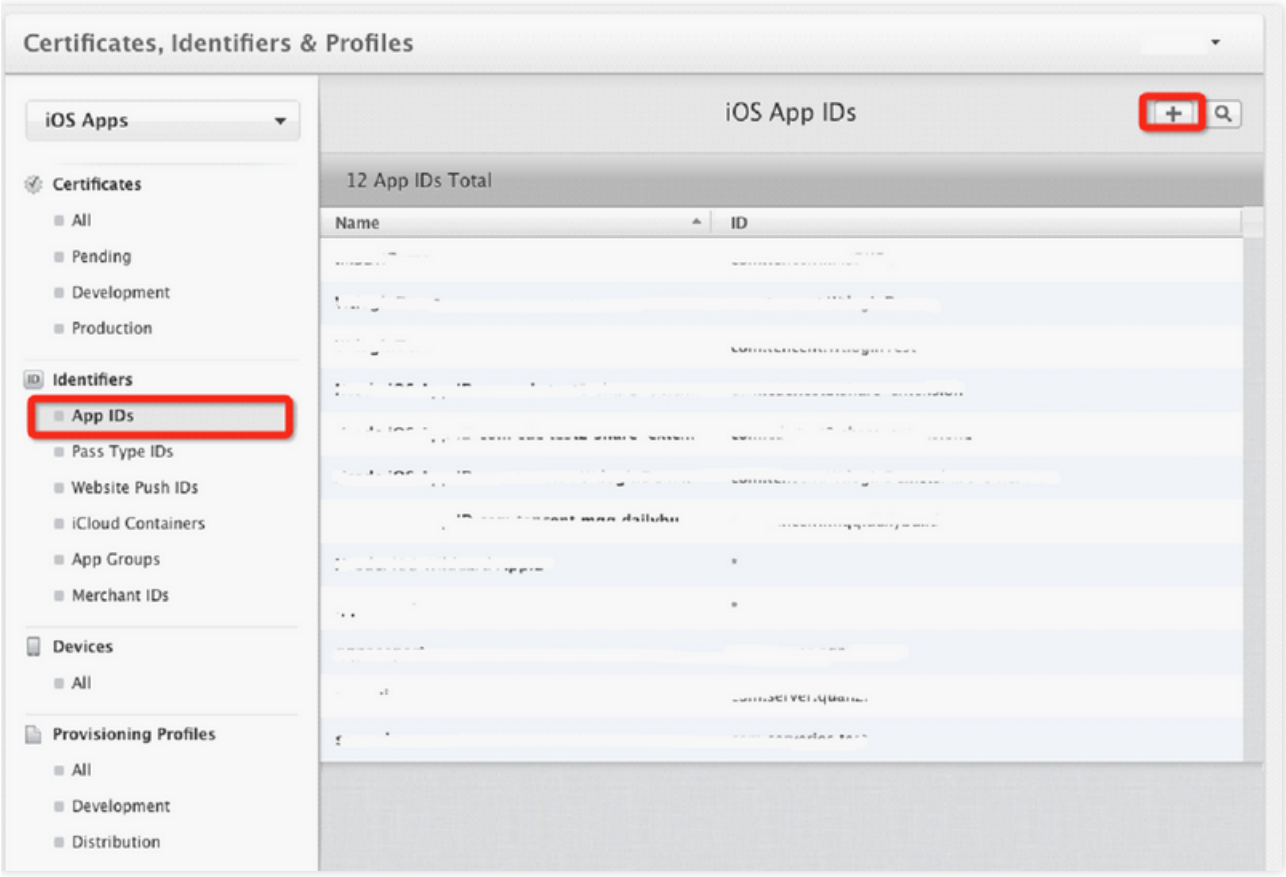
Select "Identifiers" to go to the identifier management page.



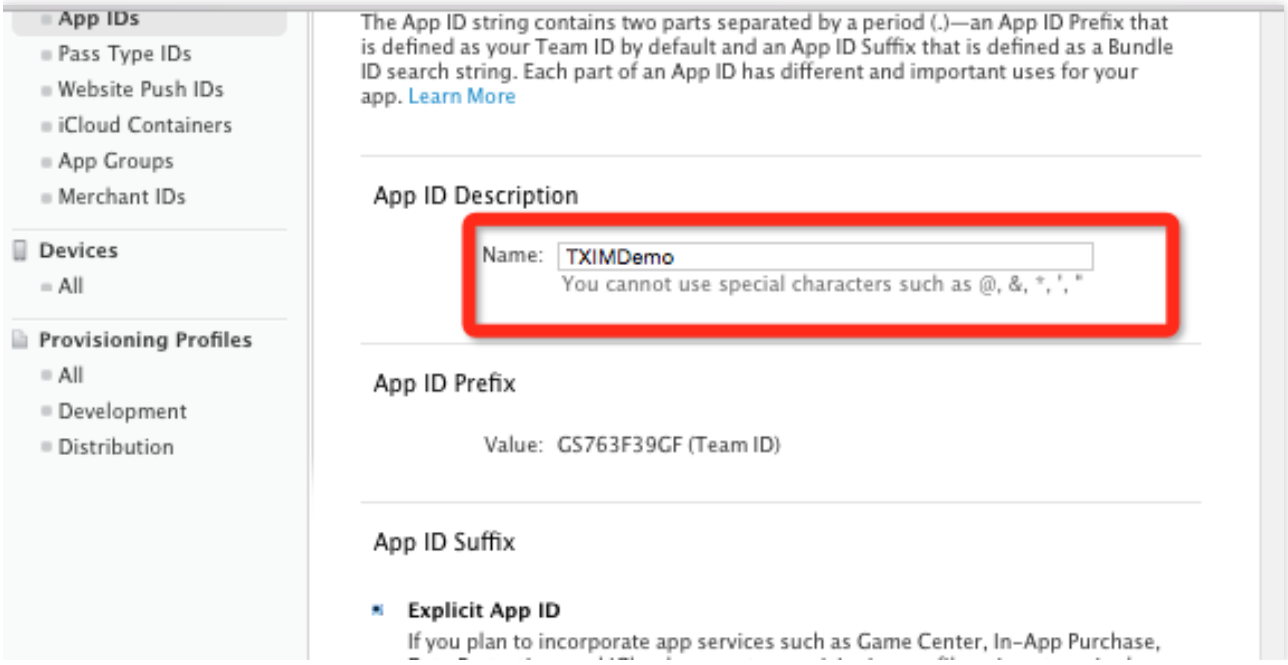
Generate an app ID as follows:

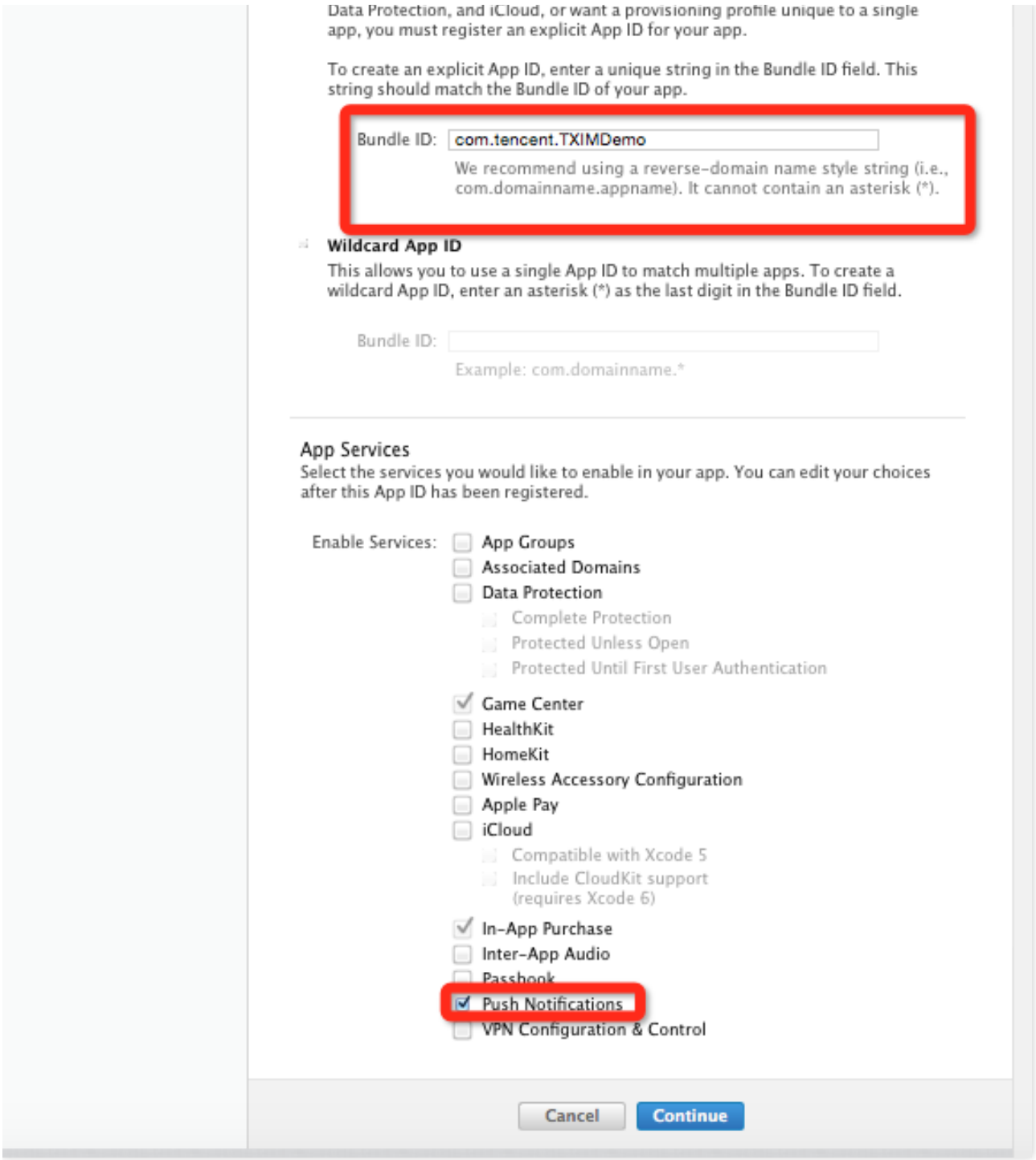
Click "App IDs" under "Identifiers", and a list of app IDs appears on the right. Skip to step

3 if you have already configured your app. Otherwise, click "+" to add an app ID.

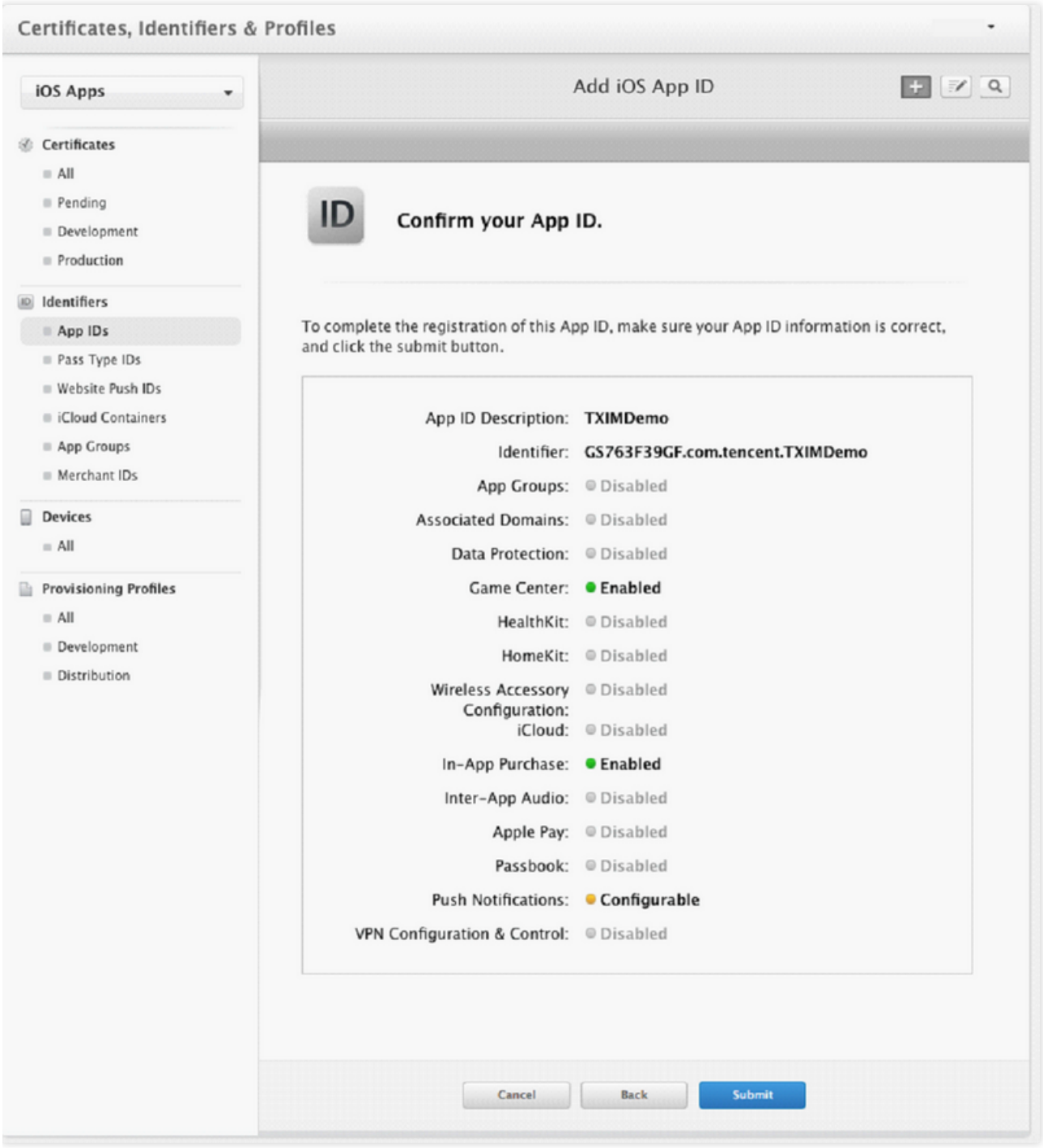


For "App ID Description", you can enter your project name. For "Bundle ID", which can be found under the "General" tab of your project, it is usually in com.youcompany.youprojname format. Select the checkbox for "Push Notifications" and click "Continue".



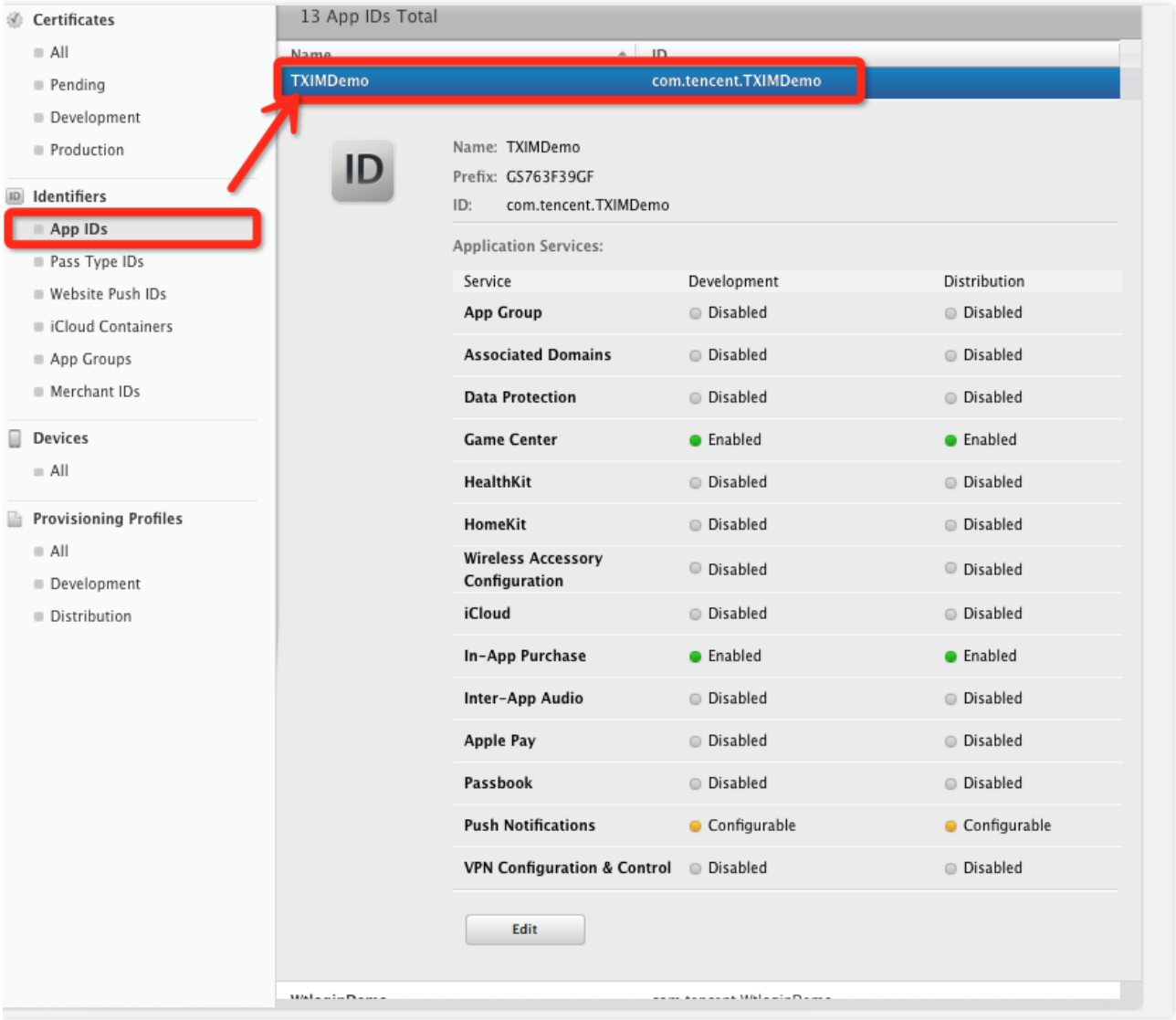


Click "Submit".



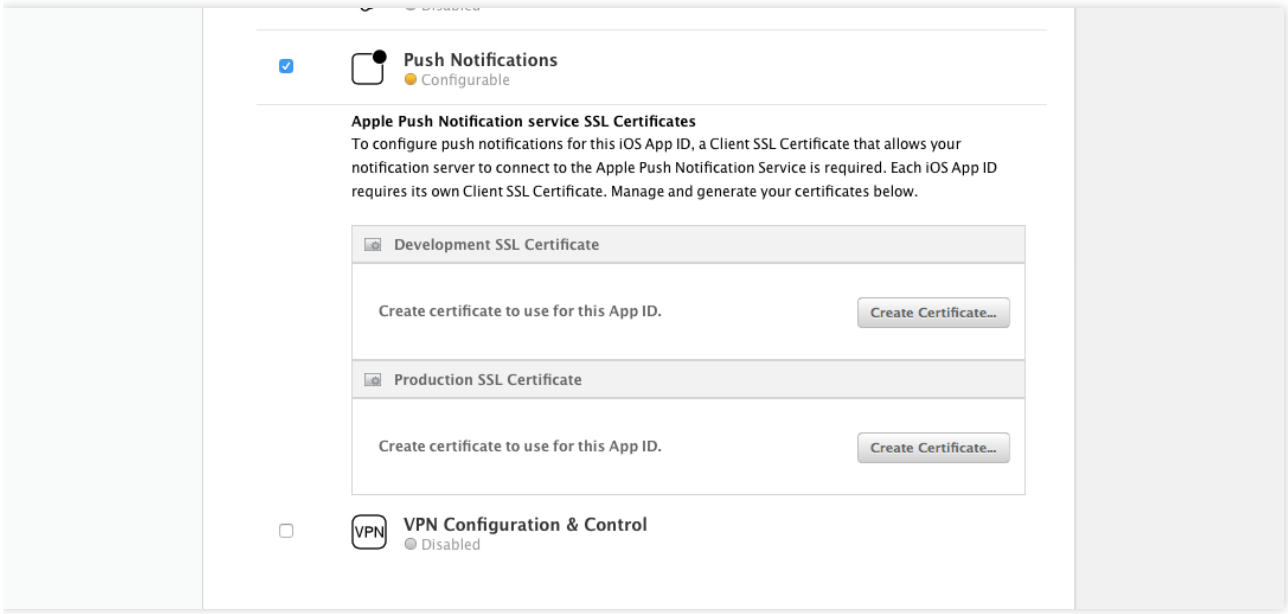
Creating an APS Certificate for the App

Return to "App IDs", select the app that needs push notifications, and then click "Edit".

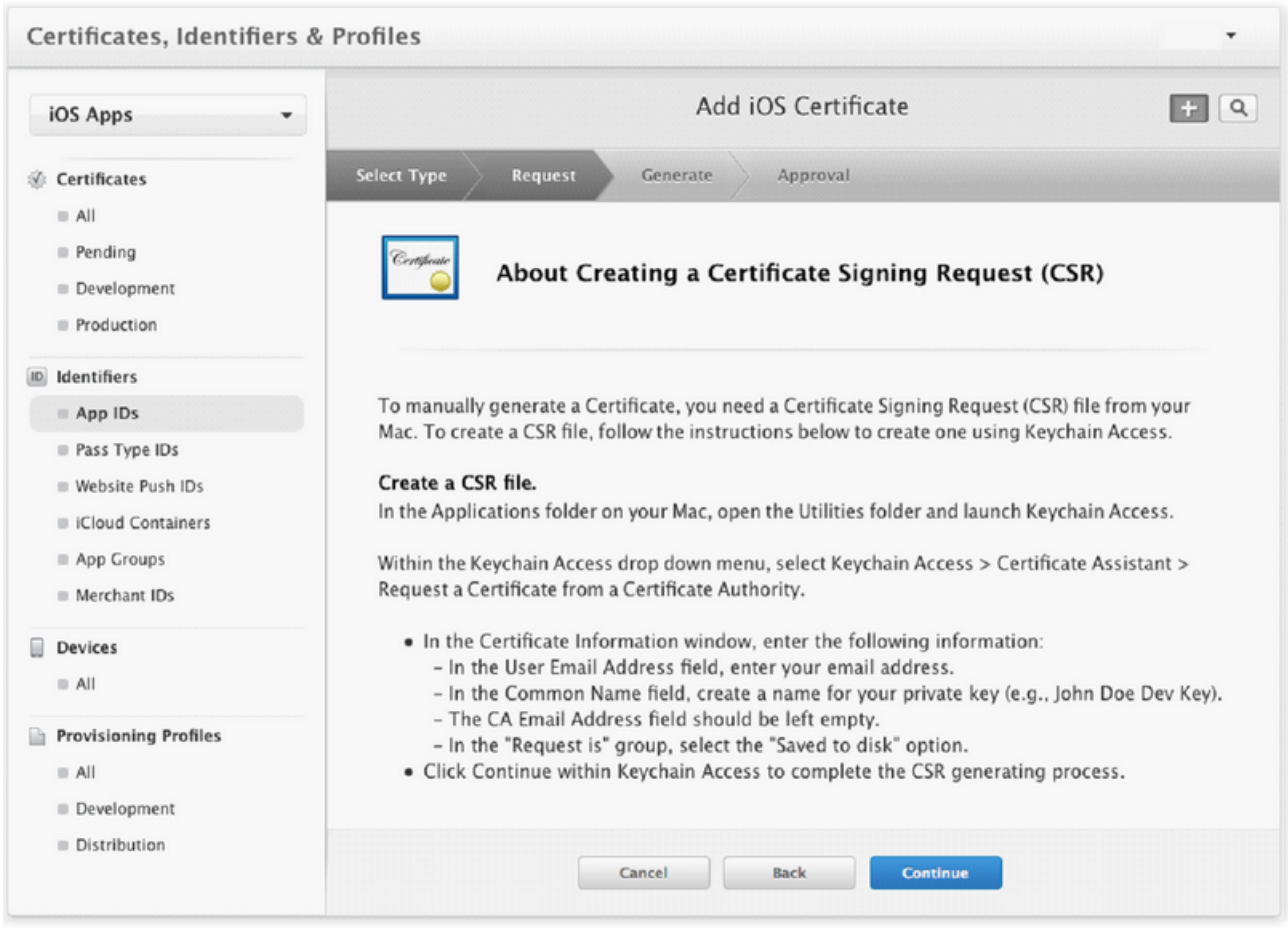


Scroll down to "Push Notifications" and click "Create Certificate..." to create a push certificate. Development certificates and production certificates need to be created

separately, which means you need to go through the same process twice.



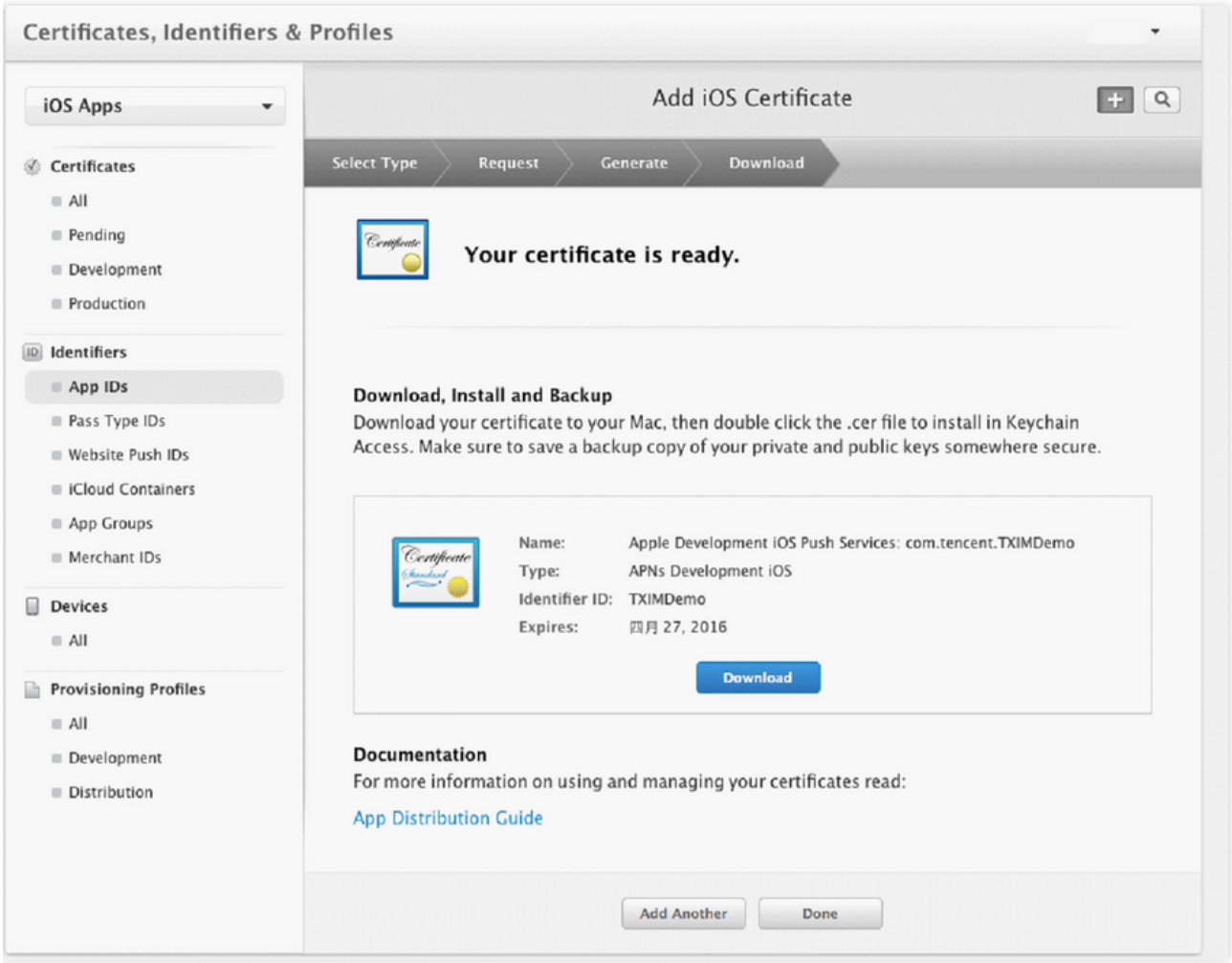
Click "Continue".



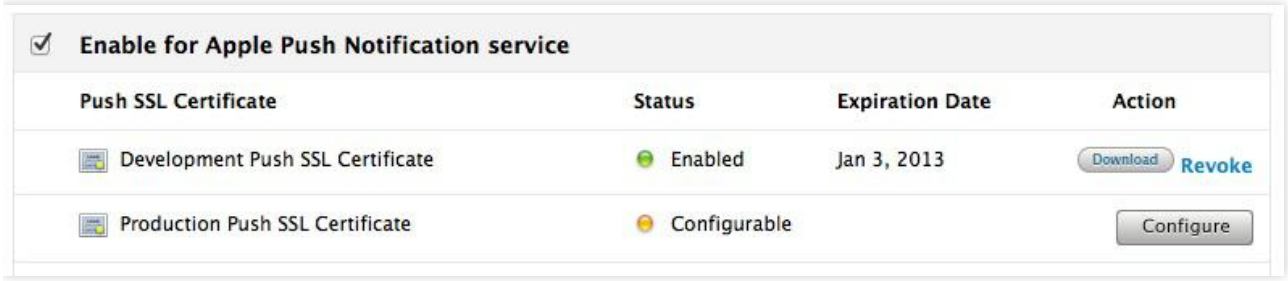
Upload the CSR file "xxx.certSigningRequest" created in section 1 (TXIMDemoAPS.certSigningRequest in this example) and click "Generate".

Now, you have finished creating the APS certificates. Click "Download" to save them locally (the development certificate is aps_development.cer, and the production

certificate is aps.cer.)



Click "Done". The push state of this environment becomes "Enabled".



Note: the "Apple Push Notification service" column for some app IDs is grayed out and the "Configure" button is unavailable. This is because APNS does not support app IDs that contain wildcards.

Generating a Push Certificate

Import the certificate
Double-click the downloaded files in the previous section (aps_development.cer and aps.ce) to install them on your computer. In "Keychain Access", you can find the imported certificates.

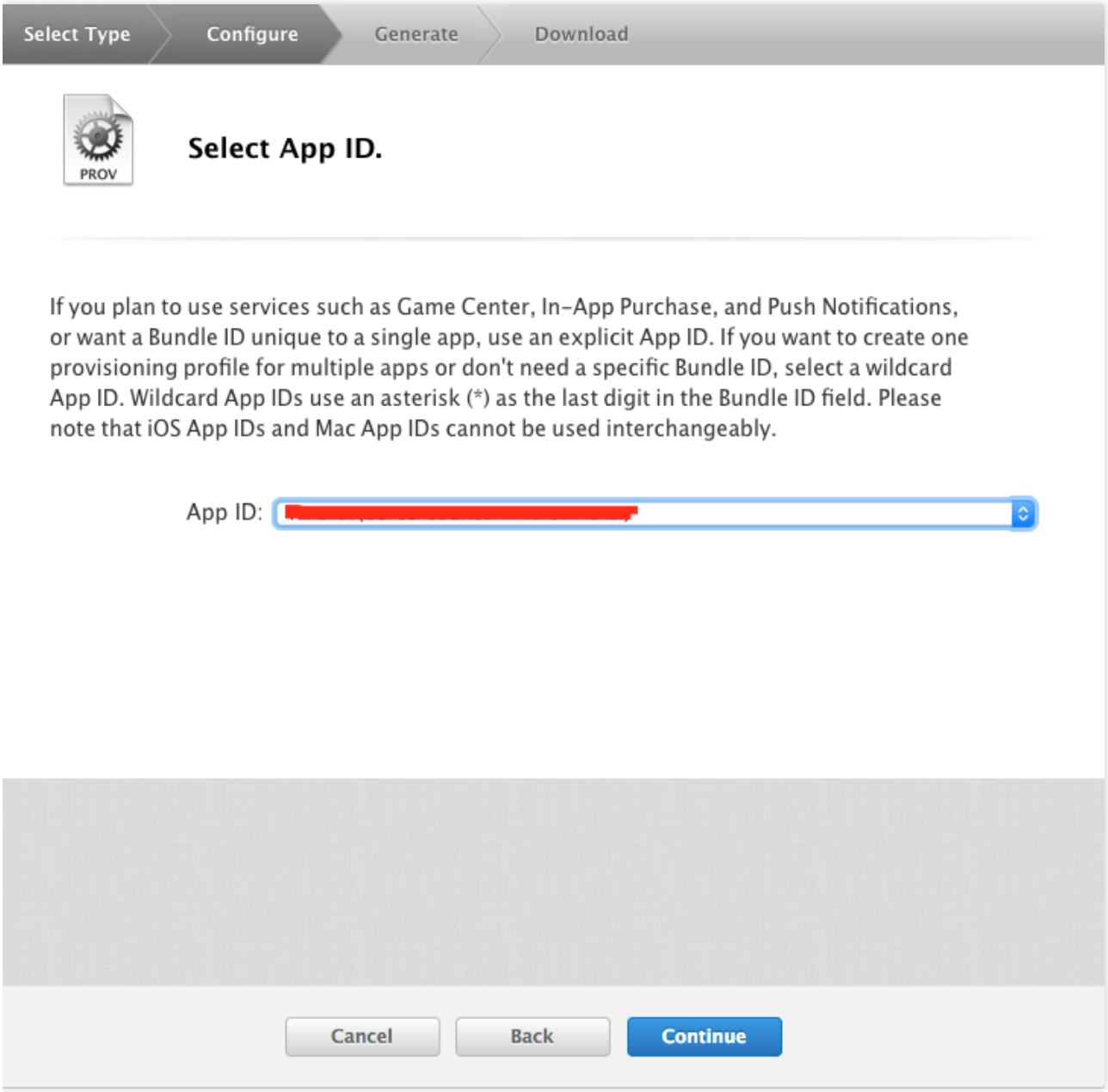
Right-click the certificate and export it as a .p12 file. For example, save the certificate as TXIMDemoAPS.p12.

Note: development certificates are valid only for development in debug mode. Always use distribution certificates for production release.

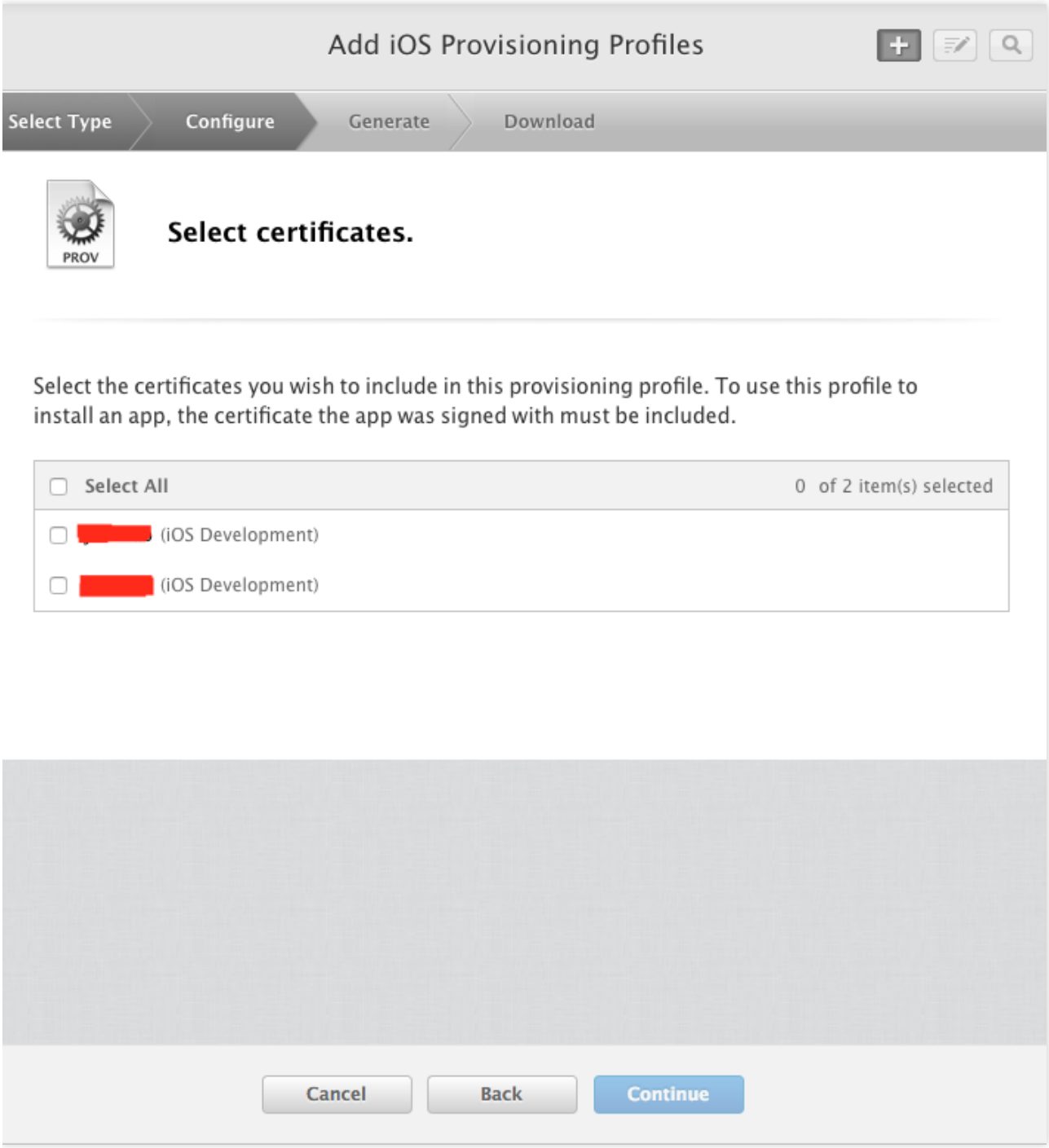
Generating a Provisioning Profile (PP)

This section describes how to create a development provisioning profile. You can create a distribution provisioning profile by following the same process. First, click "Continue".

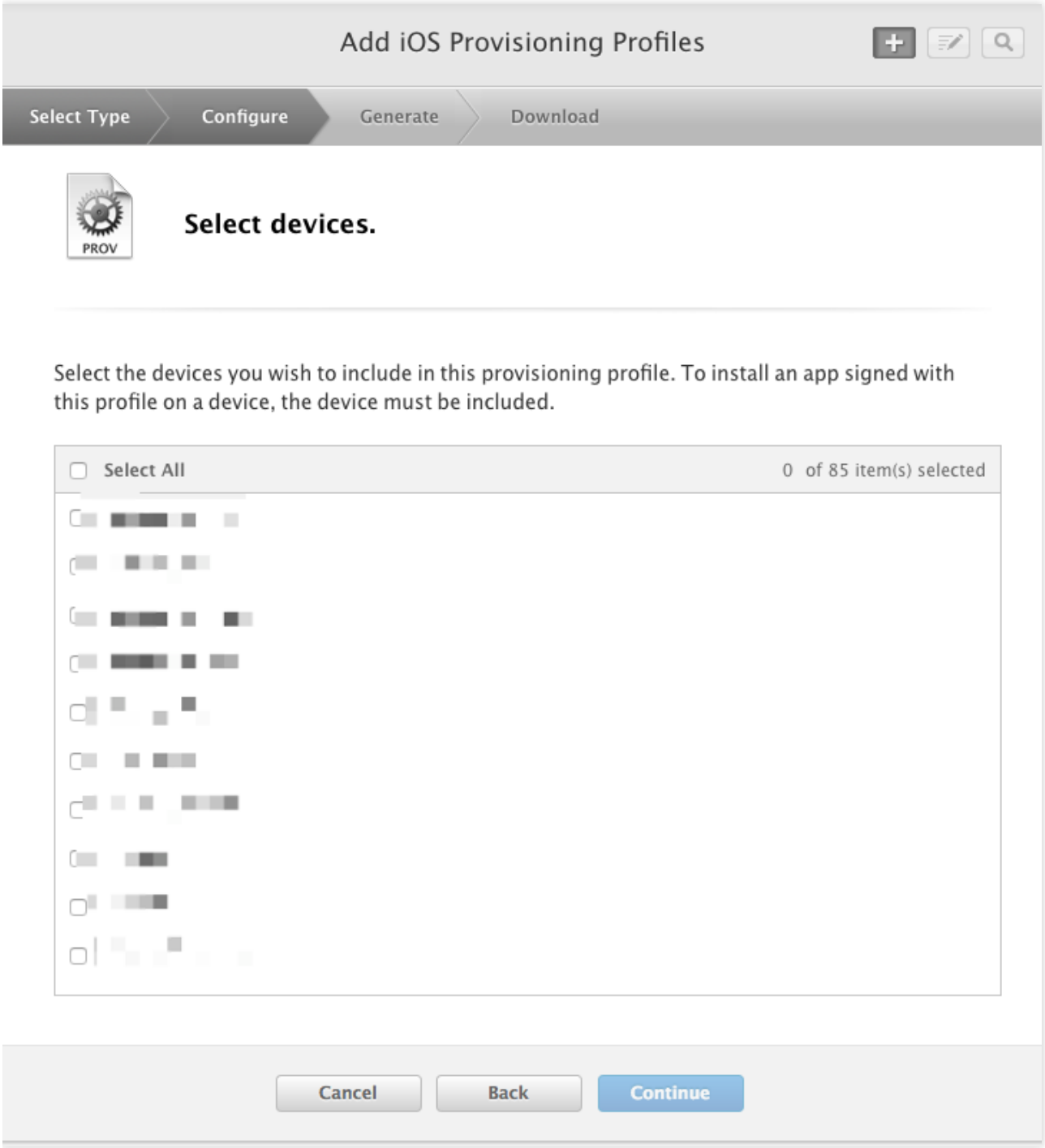
Select the App ID for which the push certificate was created in Step 3.3 and click "Continue".



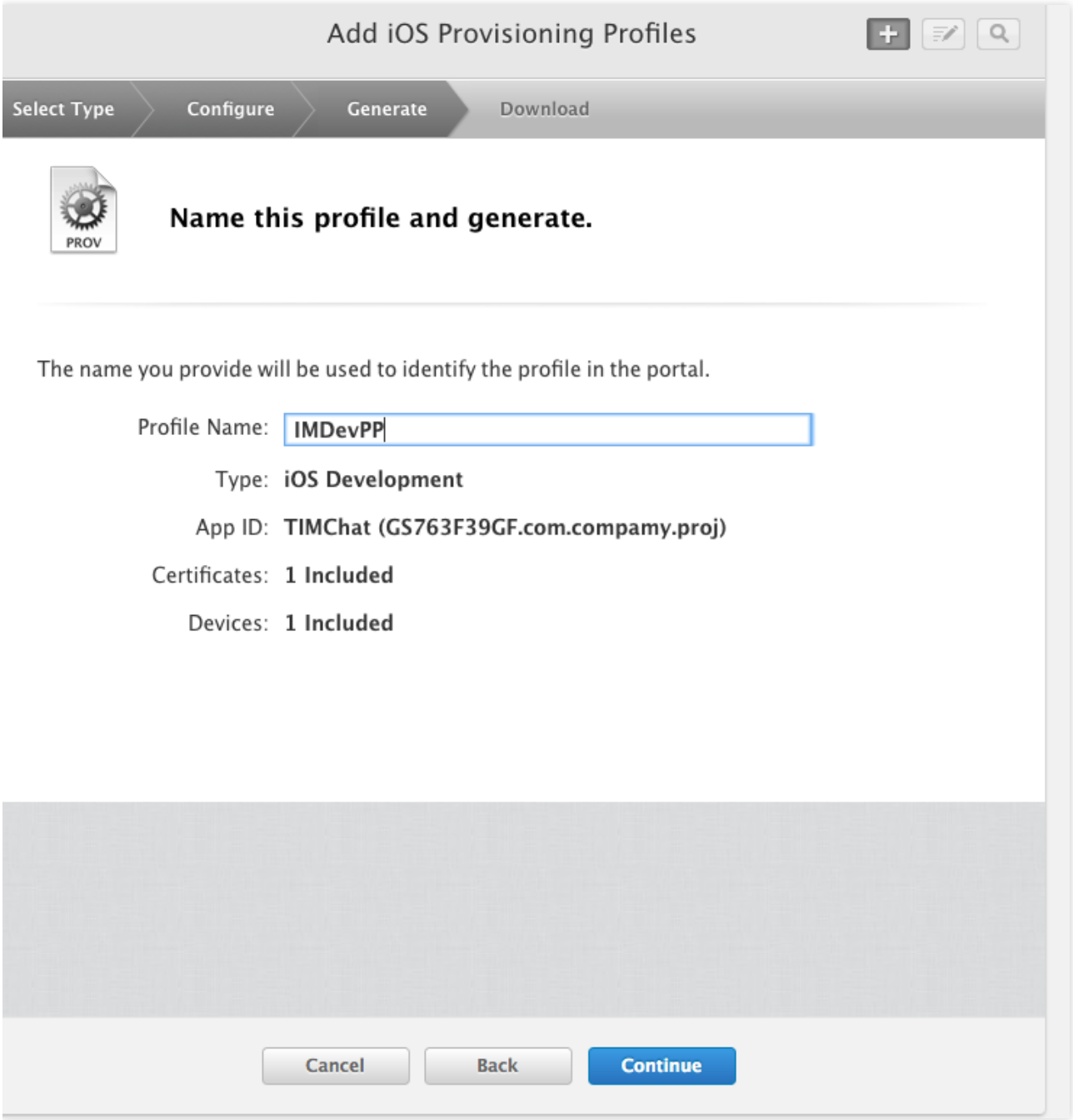
Select the development certificate generated in Step 3.3 (or the distribution certificate in Step 3.3 when creating a distribution provisioning profile) and click "Continue".



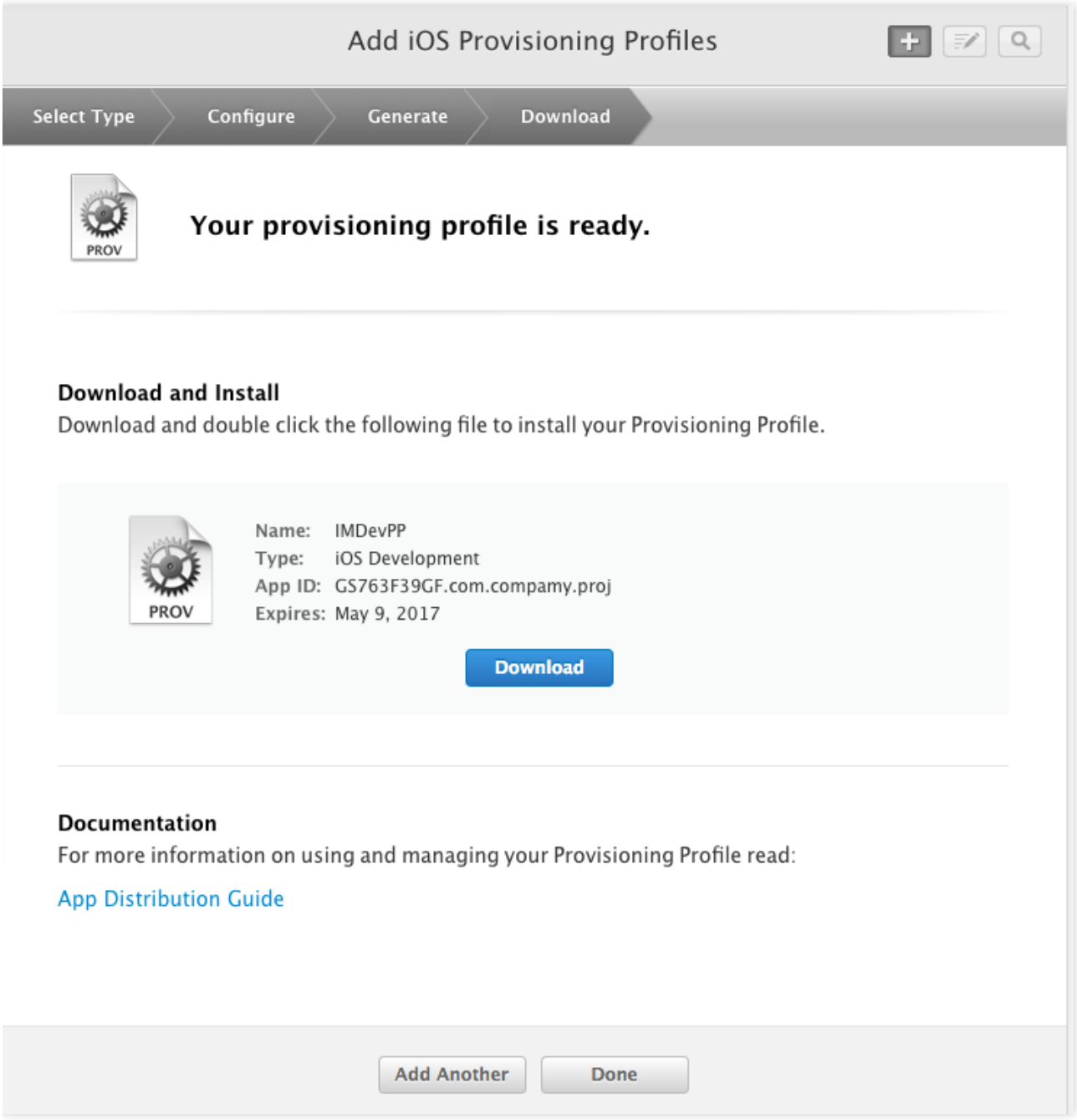
Select the devices to be included into the testing of the app (distribution certificates do not require this step) and click "Continue".



Enter the name of the PP, which is IMDevPP in this example.

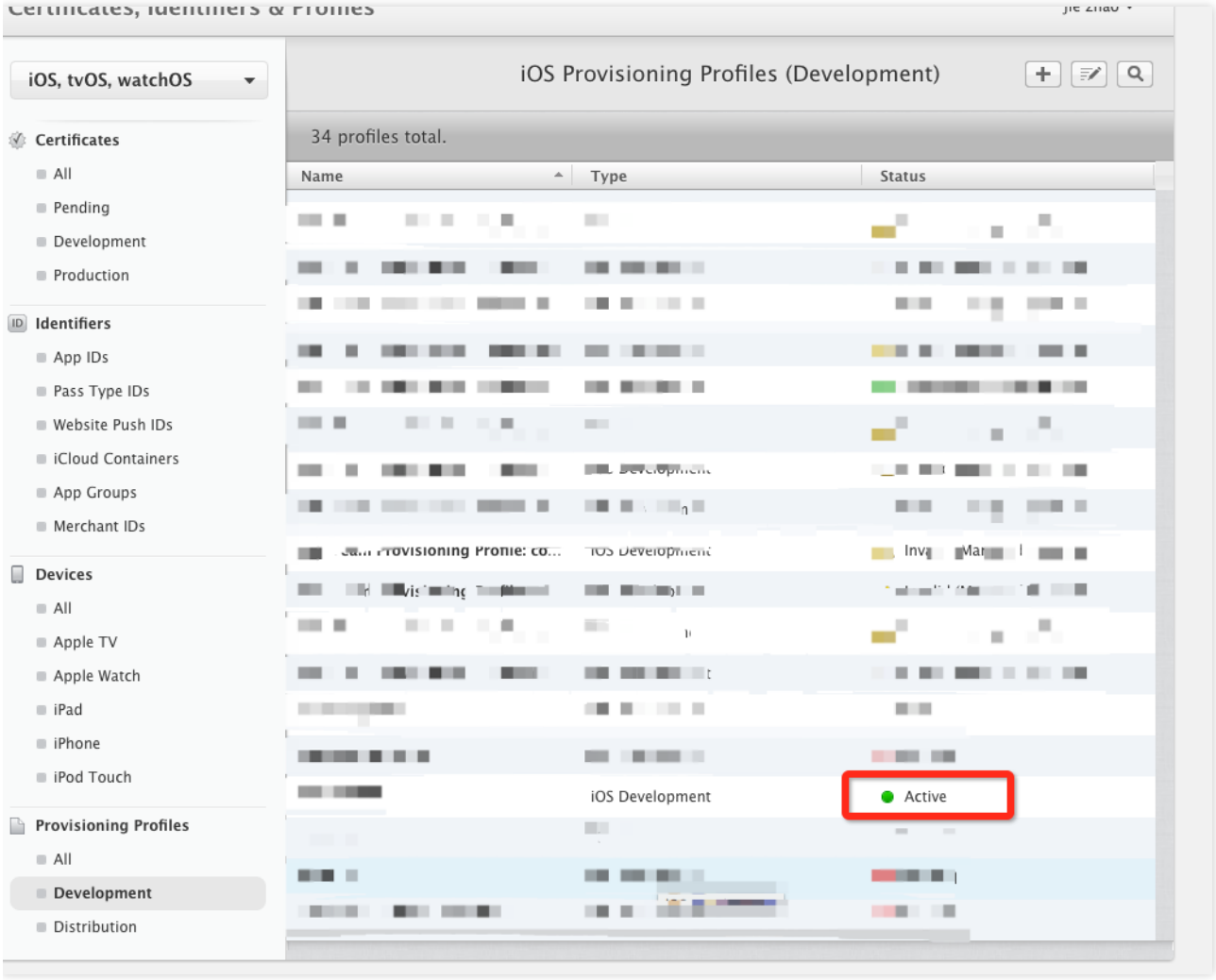


The PP is generated successfully.



Note: in all the preceding steps, no certificates, except the generated p12 certificate, need to be downloaded and installed locally.

Check the generated PP.
Verify that the state of the PP is "Active".



Click the PP to go to the details page and verify that its state is "Active".

Configuration in Xcode

The latest version of Xcode does not require the manual configuration of certificates and provisioning profiles. Instead, you only need to select the correct team in "General" and click "Fix Issue". This is why you do not need to download and install the generated certificates locally, as mentioned previously.

Offline Push (iOS)

Last updated : 2021-01-12 19:09:32

[](id: configuring push)

Configuring Offline Push

To receive APNs offline message notifications, you need to submit a Push certificate in the Tencent Cloud console. Then, during each login, the client obtains and reports the Token through an API. The APNs push feature is only used to notify users. If the app runs in the foreground, the new message obtained by the `onNewMessage` callback takes precedence, and the message obtained by `didReceiveRemoteNotification` can be ignored. For more information on how the push service works, see Apple Push Notification Service.

Applying for an APNs certificate

For more information on how to apply for an APNs certificate, see Applying for an Apple Push Certificate.

Uploading a certificate to the console

1. **Log in to the IM console.**
2. **Click the target app card to go to its basic configuration page.**
3. **Click Add Certificate on the right side of iOS Platform Push Settings.**
4. **Choose the certificate type, upload an iOS certificate (p.12), set the certificate password, and click OK.**

> ⚠ **Note :**
>
> - **We recommend that the name of the certificate to be uploaded should be in all English letters (it must not contain special characters such as brackets).**
> - **You need to set a password for the uploaded certificate. Without a password, push messages cannot be received.**
> - **Certificates to be published on App Store need to be set to the Release environment. Otherwise, push cannot be received.**
> - **The uploaded p12 certificate must be an authentic valid certificate that you have personally applied for.**

5. **After the push certificate information is generated, record the certificate ID.**

Implementing APNs push on clients

To implement APNs push on clients, perform the following steps:

Requesting DeviceToken from the Apple backend

```
- (void)registNotification
{
if ([[[UIDevice currentDevice] systemVersion] floatValue] >= 8.0)
{
[[UIApplication sharedApplication] registerUserNotificationSettings:[UIUserNotificationSettings s
ettingsForTypes:(UIUserNotificationTypeSound | UIUserNotificationTypeAlert | UIUserNotificationTy
peBadge) categories:nil]];
[[UIApplication sharedApplication] registerForRemoteNotifications];
}
else
{
[[UIApplication sharedApplication] registerForRemoteNotificationTypes: (UIUserNotificationTypeBad
ge | UIUserNotificationTypeSound | UIUserNotificationTypeAlert)];
}
}
/**
* The callback of AppDelegate returns deviceToken, which needs to be reported to the Tencent Clou
d backend after login.
/**
-(void)application:(UIApplication *)app didRegisterForRemoteNotificationsWithDeviceToken:(NSData
*)deviceToken
{
// Note the deviceToken returned by Apple.
_deviceToken = deviceToken;
}
```

Uploading Token to Tencent Cloud after IM SDK login

> ⚠ **Note :**
>
> **busiID must be consistent with the certificate ID assigned by the console.**

```
__weak typeof(self) ws = self;
// If the TUIKit is used here, set the Token in the TUIKit login callback. If it is not used, set
the Token in the TIMManager login callback.
[[TUIKit sharedInstance] loginKit:identifier userSig:userSig succ:^{
TIMTokenParam *param = [[TIMTokenParam alloc] init];
/* Users need to register a developer certificate with Apple, download and generate the certifica
te (p12 file) in their developer accounts, and upload the generated p12 file to the Tencent certi
```

```
ficate console. The console will automatically generate a certificate ID and pass it to the busiI
D parameter.*/
#if kAppStoreVersion
// App Store version
#if DEBUG
param.busiId = 2383;
#else
param.busiId = 2382;
#endif
#else
// Enterprise certificate ID
param.busiId = 2516;
#endif
[param setToken:ws.deviceToken];
[[TIMManager sharedInstance] setToken:param succ:^{
NSLog(@"-----> Uploading token succeeded ");
} fail:^(int code, NSString *msg) {
NSLog(@"-----> Uploading token failed ");
}];
} fail:^(int code, NSString *msg) {
NSLog(@"Login failed!");
}];
}
```

Reporting a background switch event when the app goes to the background

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
__block UIBackgroundTaskIdentifier bgTaskID;
bgTaskID = [application beginBackgroundTaskWithExpirationHandler:^ {
// End the background_task regardless of whether it has been completed
[application endBackgroundTask: bgTaskID];
bgTaskID = UIBackgroundTaskInvalid;
}];
// Obtain the unread count
int unReadCount = 0;
NSArray *convs = [[TIMManager sharedInstance] getConversationList];
for (TIMConversation *conv in convs) {
if([conv getType] == TIM_SYSTEM){
continue;
}
unReadCount += [conv getUnReadMessageNum];
}
[UIApplication sharedApplication].applicationIconBadgeNumber = unReadCount;

//doBackground
TIMBackgroundParam *param = [[TIMBackgroundParam alloc] init];
```

```
[param setC2cUnread:unReadCount];
[[TIMManager sharedInstance] doBackground:param succ:^() {
NSLog(@"doBackgroud Succ");
} fail:^(int code, NSString * err) {
NSLog(@"Fail: %d->%@", code, err);
}];
}
```

Reporting a foreground switch event when the app goes to the foreground

```
- (void)applicationDidBecomeActive:(UIApplication *)application {
[[TIMManager sharedInstance] doForeground:^() {
NSLog(@"doForegroud Succ");
} fail:^(int code, NSString * err) {
NSLog(@"Fail: %d->%@", code, err);
}];
}
```

Push Format

An example of the push format is as shown below.

General push rules

For one-to-one messages, the APNs push rules are as follows. Note that the nickname is the sender's nickname. If the nickname is not set, only the content is displayed.

```
Nickname: content
```

For group messages, the APNs push rules are as follows. The name is either the group name card or the sender's nickname. The priority is: `group name card` > `nickname` .

```
Name (group name): content
```

Push rules for different types of messages

The APNs push content consists of the content of each `Elem` in the message body. The display of different `Elem` in offline messages is shown in the following table.

Parameter	Description
Text Elem	**Directly display the content**
Voice Elem	**Display `[voice]`**
File Elem	**Display `[file]`**
Image Elem	**Display `[image]`**
Custom Elem	**Display the content of the `desc` field. If it is empty, the message will not be pushed offline.**

Communication among multiple apps

If you set `SDKAppID` to the same value for multiple apps, these apps communicate with each other. Different apps need to use different push certificates, and you need to apply for an APNs certificate for each app and complete [offline push configuration] (#configuring push).

Push Alert Sound

Setting custom push alert sounds

IM SDK provides an API for setting user sounds. You can use the API to customize the alert sound for one-to-one messages and the alert sound for group messages. You can also set the specific users to receive push messages.

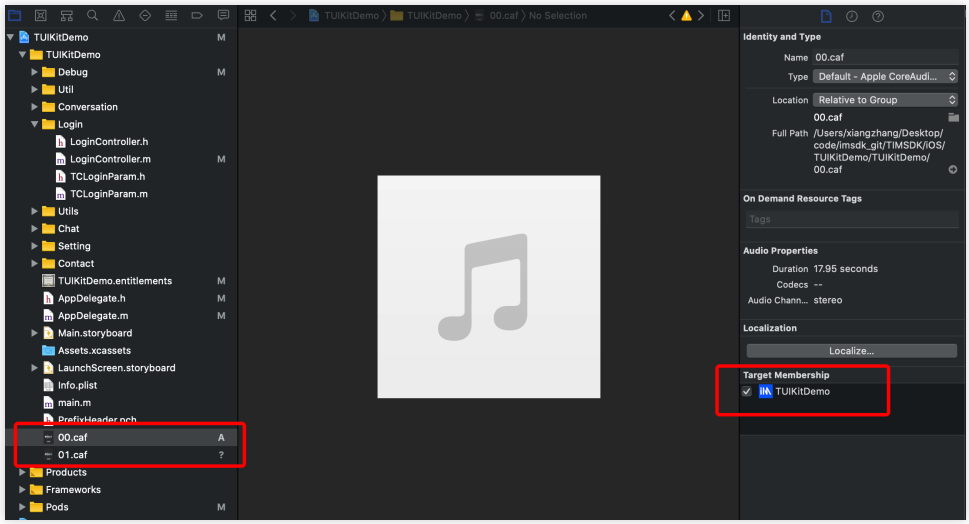
```
/**
* APNs configuration
*/
@interface TIMAPNSConfig : NSObject
/**
* Whether push is enabled. 0: not set. 1: enabled. 2: disabled.
*/
@property(nonatomic,assign) uint32_t openPush;
/**
* C2C message sound. If you do not want to set it, pass nil
*/
@property(nonatomic,retain) NSString * c2cSound;
/**
* Group message sound. If you do not want to set it, pass nil
*/
@property(nonatomic,retain) NSString * groupSound;
@end
@interface TIMManager : NSObject
/**
* Setting APNs configuration
*
* @param config APNs configuration
* @param succ Success callback
* @param fail Failure callback
*
* @return 0 Success
*/
-(int) setAPNS:(TIMAPNSConfig*)config succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

Parameters

Parameter	Description
config	**openPush: whether push is enabled. 0: not set. 1: enabled. 2: disabled. c2cSound: alert sound for one-to-one messages, which needs to be set to a file name (including the suffix) groupSound: alert sound for group messages, which needs to be set to a file name (including the suffix)**
succ	**Success callback**
fail	**Failure callback**

Directions

1. Integrate the audio file into your project.



2. After successful login, call the `setToken` API to set token and busiID information.

3. Call the `setAPNS` API to set the audio file information.

> **Note :**
>
> You only need to set the file name (including the suffix) of the audio file.

```
    [[TIMManager sharedInstance] login:param succ:^{
        if (ws.deviceToken) {
            TIMTokenParam *param = [[TIMTokenParam alloc] init];
            /* 用户自己到苹果注册开发者证书，在开发者帐号中下载并生成证书(p12
                文件)，将生成的 p12 文件传到腾讯证书管理控制台，控制台会自动生成一个证书
                ID，将证书 ID 传入一下 busiId 参数中。*/
            //企业证书 ID
            param.busiId = sdkBusiId;
            [param setToken:ws.deviceToken];
            [[TIMManager sharedInstance] setToken:param succ:^{
                NSLog(@"-----> 上传 token 成功 ");
                //推送声音的自定义化设置
                TIMAPNSConfig *config = [[TIMAPNSConfig alloc] init];
                config.openPush = 0;
                config.c2cSound = @"00.caf";
                config.groupSound = @"01.caf";
                [[TIMManager sharedInstance] setAPNS:config succ:^{
                    NSLog(@"-----> 设置 APNS 成功");
                } fail:^(int code, NSString *msg) {
                    NSLog(@"-----> 设置 APNS 失败");
                }];
            } fail:^(int code, NSString *msg) {
                NSLog(@"-----> 上传 token 失败 ");
            }];
        }
        ws.window.rootViewController = [self getMainController];
    } fail:^(int code, NSString *msg) {
```

Obtaining the push message alert sound

You can use the `getAPNSConfig` API to obtain the push message alert sound. This API synchronizes data from the server for each request and does not cache data locally.

```
@interface TIMManager : NSObject
/**
* Obtaining the APNs configuration
*
* @param succ Success callback, which returns the configuration information
* @param fail Failure callback
*
* @return 0 Success
*/
-(int) getAPNSConfig:(TIMAPNSConfigSucc)succ fail:(TIMFail)fail;
@end
```

Parameter descriptions:

Parameter	Description
succ	**Success callback, which returns the TIMAPNSConfig structure**
fail	**Failure callback**

Customizing Offline Message Attributes

You can set `TIMOfflinePushInfo` in each message to determine the displayed text, extension field, alert sound, and whether to enable push. During message push, the original default attributes will be replaced by the custom attributes that you have set. For example, if you enter `kIOSOfflinePushNoSound` in the `sound` attribute, the push received by the recipient will be forcibly muted.

```
/**
Entering `kIOSOfflinePushNoSound` in the `sound` field indicates that no sound will be played whe
n the push message is received.
*/
extern NSString * const kIOSOfflinePushNoSound;
@interface TIMAndroidOfflinePushConfig : NSObject
/**
* Offline push display tag
*/
@property(nonatomic,retain) NSString * title;
/**
* Sound field information during offline push on Android
*/
@property(nonatomic,retain) NSString * sound;
/**
* Notification mode for offline push
*/
@property(nonatomic,assign) TIMAndroidOfflinePushNotifyMode notifyMode;
@end
@interface TIMIOSOfflinePushConfig : NSObject
/**
* Sound field information during offline push
*/
@property(nonatomic,retain) NSString * sound;
/**
* Ignore badge count
*/
@property(nonatomic,assign) BOOL ignoreBadge;
@end
@interface TIMOfflinePushInfo : NSObject
/**
* Custom message description information to be displayed in text during offline push
*/
@property(nonatomic,retain) NSString * desc;
/**
* Extension field information during offline push
*/
```

```
@property(nonatomic,retain) NSString * ext;
/**
* Push rule flag
*/
@property(nonatomic,assign) TIMOfflinePushFlag pushFlag;
/**
* iOS offline push configuration
*/
@property(nonatomic,retain) TIMIOSOfflinePushConfig * iosConfig;
/**
* Android offline push configuration
*/
@property(nonatomic,retain) TIMAndroidOfflinePushConfig * androidConfig;
@end
```