即时通信 IM

常规集成（无 UI 库）

产品文档

腾讯云

文档目录

常规集成（无 UI 库）
快速导入到工程
集成 SDK（Android）

最近更新时间：2020-09-04 15:49:35

本文主要介绍如何快速地将腾讯云即时通信 IM SDK 集成到您的项目中，只要按照如下步骤进行配置，就可以完成 SDK 的集成工作。

开发环境要求

- JDK 1.6。
- Android 4.1（SDK API 16）及以上系统。

集成 SDK（aar）

您可以选择使用 Gradle 自动加载的方式，或者手动下载 aar 再将其导入到您当前的工程项目中。

方法一：自动加载（aar）

IM SDK 已经发布到 jcenter 库，您可以通过配置 gradle 自动下载更新。
只需要用 Android Studio 打开需要集成 SDK 的工程，然后通过如下三个步骤修改 app/build.gradle 文件，就可以完成 SDK 集成：

- **第一步：添加 SDK 依赖**

  找到 app 的 build.gradle，在 dependencies 中添加 IM SDK 的依赖。
  如果使用标准版 IM SDK，请添加如下依赖。

```
dependencies {
api 'com.tencent.imsdk:imsdk:版本号'
}
```

  如果使用精简版 IM SDK，请添加如下依赖。

```
dependencies {
api 'com.tencent.imsdk:imsdk-smart:版本号'
}
```

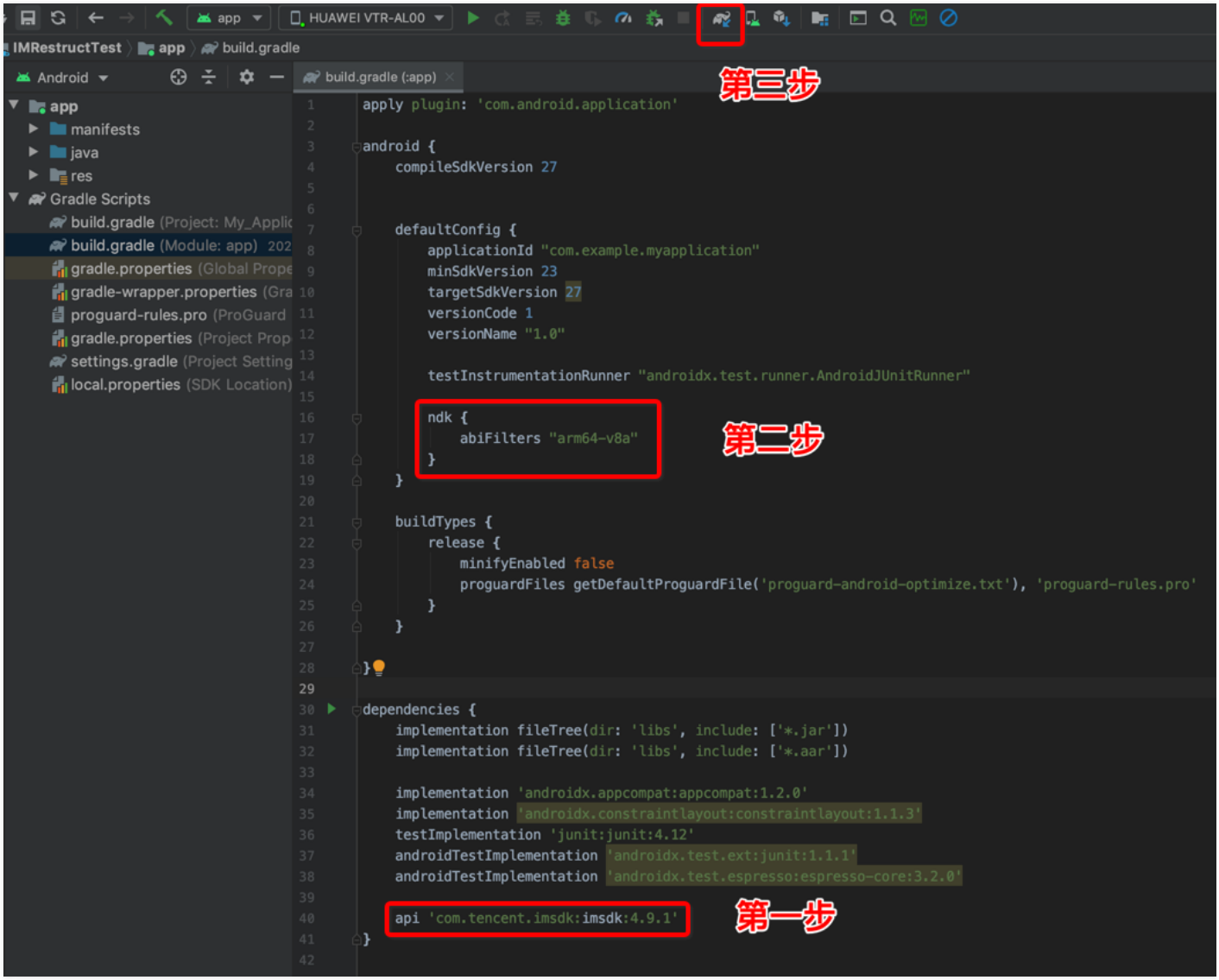
  说明：
  “版本号”应替换为 SDK 的实际版本号，建议使用 最新版本。
  以版本号是 4.9.1 为例：

```
dependencies {
api 'com.tencent.imsdk:imsdk:4.9.1'
}
```

- **第二步：指定 App 使用架构**
  在 defaultConfig 中，指定 App 使用的 CPU 架构（从 IM SDK 4.3.118 版本开始支持 armeabi-v7a，arm64-v8a，x86，x86_64）：

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

- **第三步：同步 SDK**
  单击 Sync 按钮，如果您的网络连接 jcenter 没有问题，SDK 就会自动下载集成到工程里。

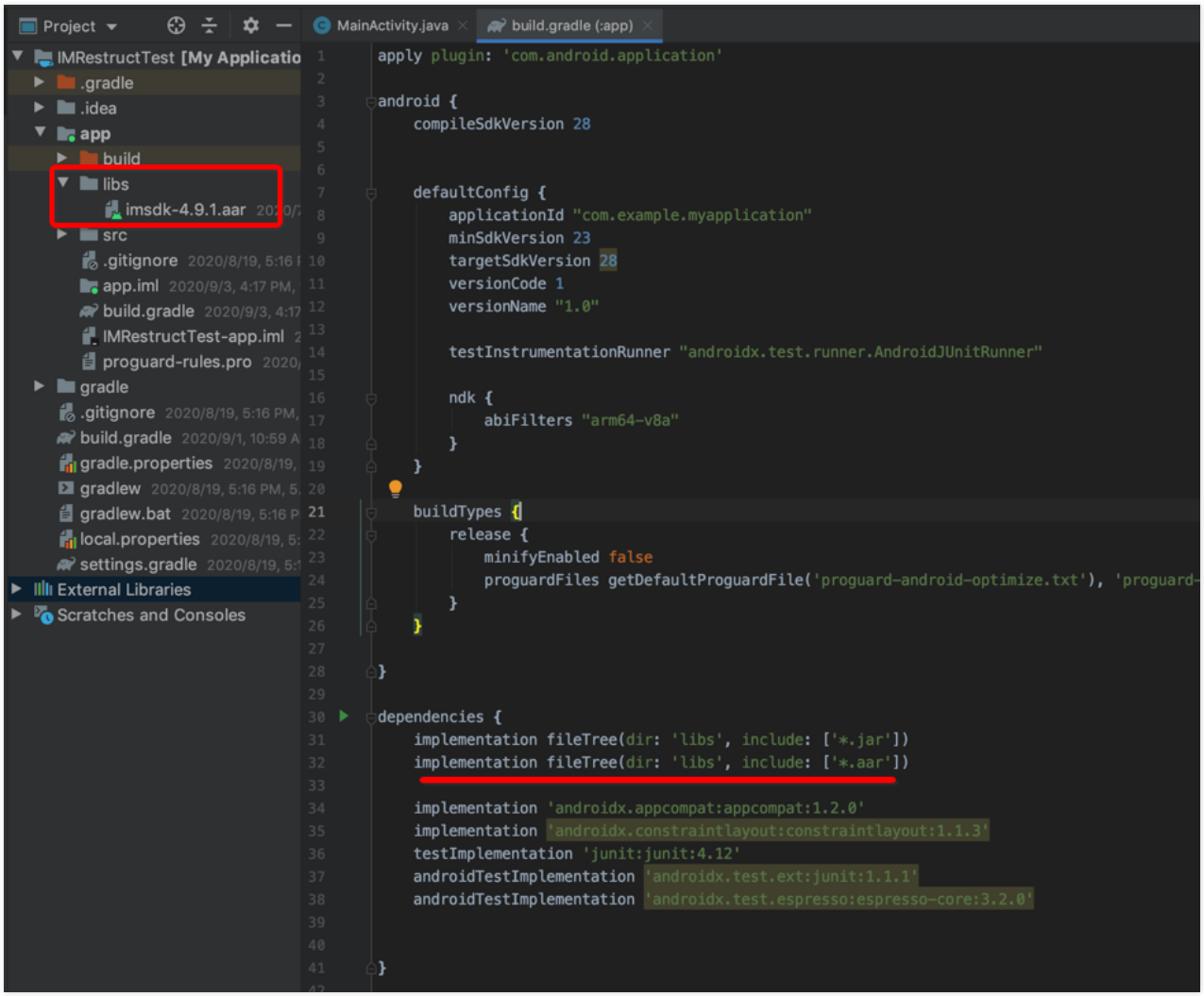


方法二：手动下载（aar）

如果您的网络连接 jcenter 有问题，也可以手动下载 SDK 集成到工程里：

- **第一步：下载 IM SDK**
  在 Github 上可以下载到最新版本的 IM SDK。

- **第二步：拷贝 IM SDK 到工程目录**
  将下载到的 aar 文件拷贝到 app 工程的 **/libs** 目录下：



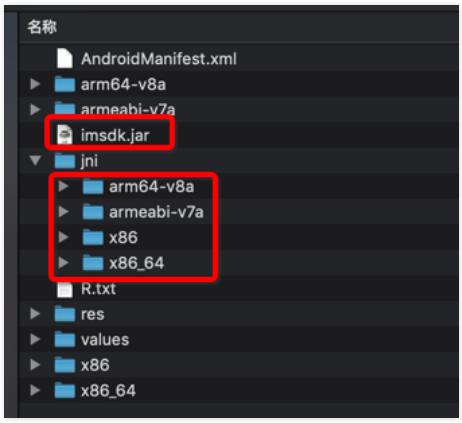
- **第三步：指定 App 使用架构并编译运行**
  在 app/build.gradle的defaultConfig 中，指定 App 使用的 CPU 架构（从 IM SDK 4.3.118 版本开始支持 armeabi-v7a，arm64-v8a，x86，x86_64）：

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

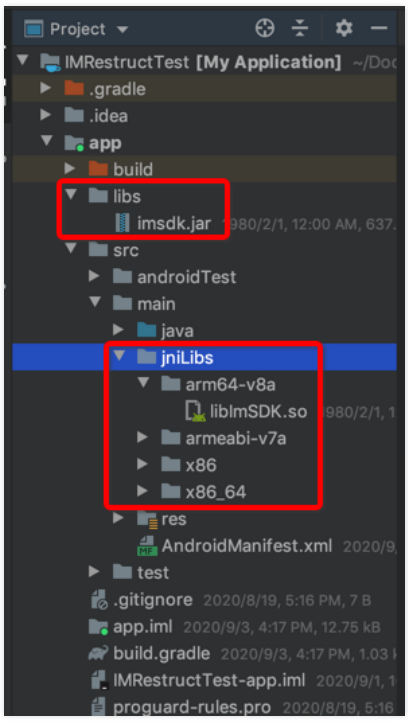
集成 SDK

如果您不想集成 aar 库，也可以通过导入 jar 和 so 库的方式集成 IM SDK：

- **第一步：下载解压 IM SDK**
  在 Github 上可以 下载 到最新版本的 aar 文件。解压后的目录里面主要包含 jar 文件和 so 文件夹，把其中的 **classes.jar** 重命名成 **imsdk.jar** 。



- **第二步：拷贝 SDK 文件到工程目录**
  将重命名后的 jar 文件和各个架构的 so 文件分别拷贝到 Android Studio 默认加载的目录下：



- **第三步：指定 App 使用架构并编译运行**
  在 app/build.gradle 的 defaultConfig 中，指定 App 使用的 CPU 架构（从 IM SDK 4.3.118 版本开始支持 armeabi-v7a，arm64-v8a，x86，x86_64）：

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

配置 App 权限

在 AndroidManifest.xml 中配置 App 的权限，IM SDK 需要以下权限：

```
<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" />
```

设置混淆规则

在 proguard-rules.pro 文件，将 IM SDK 相关类加入不混淆名单：

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

集成 SDK（iOS）

最近更新时间：2020-09-04 15:49:39

本文主要介绍如何快速地将腾讯云 IM SDK(iOS) 集成到您的项目中，只要按照如下步骤进行配置，就可以完成 SDK 的集成工作。

开发环境要求

- Xcode 9.0+。
- iOS 8.0 以上的 iPhone 或者 iPad 真机。
- 项目已配置有效的开发者签名。

集成 IM SDK

您可以选择使用 CocoaPods 自动加载的方式，或者先 下载 SDK，再将其导入到您当前的工程项目中。

CocoaPods 自动加载

1. 安装 CocoaPods

在终端窗口中输入如下命令（需要提前在 Mac 中安装 Ruby 环境）：

```
sudo gem install cocoapods
```

2. 创建 Podfile 文件

进入项目所在路径输入以下命令行，之后项目路径下会出现一个 Podfile 文件。

```
pod init
```

3. 编辑 Podfile 文件

如果使用标准版 SDK，请您按照如下方式设置：

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

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

如果使用精简版 SDK，请您按照如下方式设置：

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

target 'App' do
pod 'TXIMSDK_Smart_iOS'
end
```

4. 更新并安装 SDK

在终端窗口中输入如下命令以更新本地库文件，并安装 TXIMSDK：

```
pod install
```

或使用以下命令更新本地库版本：

```
pod update
```

pod 命令执行完后，会生成集成了 SDK 的 .xcworkspace 后缀的工程文件，双击打开即可。

说明：
若 pod 搜索失败，建议尝试更新 pod 的本地 repo 缓存。更新命令如下：

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

手动集成

1. 下载 SDK

从 Github 下载最新版本 SDK。

- ImSDK.framework 和 ImSDK_Smart.framework 是 IM SDK 的核心动态库文件。

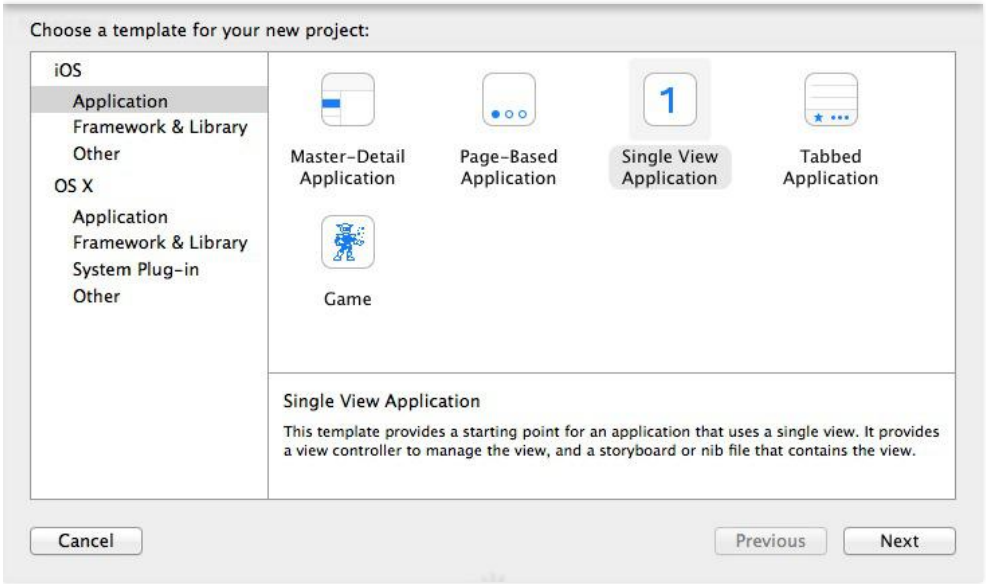
包名	介绍
ImSDK.framework	标准版 IM 功能包
ImSDK_Smart.framework	精简版 IM 功能包

- TXLiteAVSDK_UGC.framework 是腾讯云短视频（UGC）SDK，用于实现即时通信 IM 中的短视频收发能力，为可选组件。

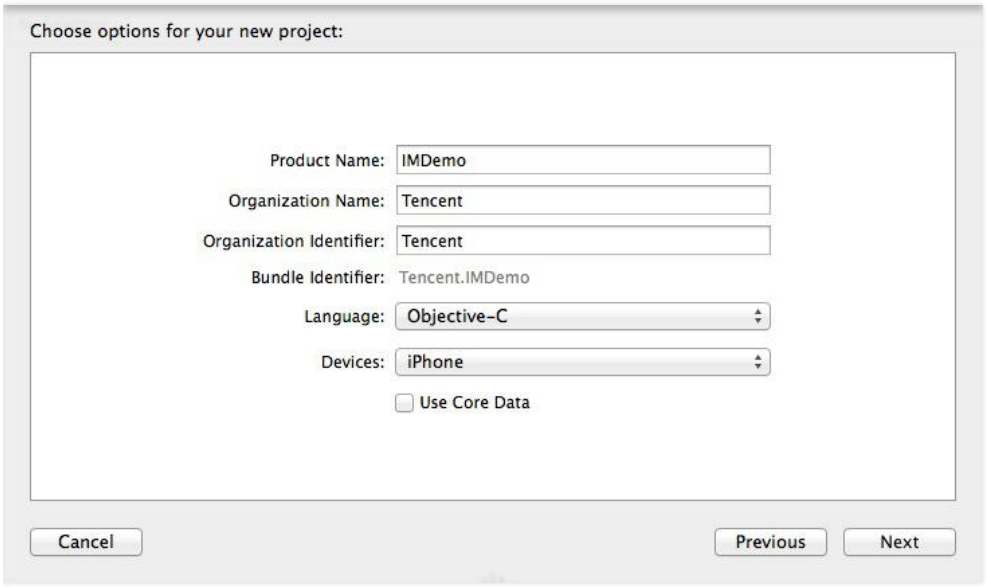
包名	介绍	功能
TXLiteAVSDK_UGC.framework	小视频录制、编辑能力扩展包	包含小视频录制功能、小视频编辑功能，详情请参阅 短视频 SDK 文档

2. 创建工程

创建一个新工程：

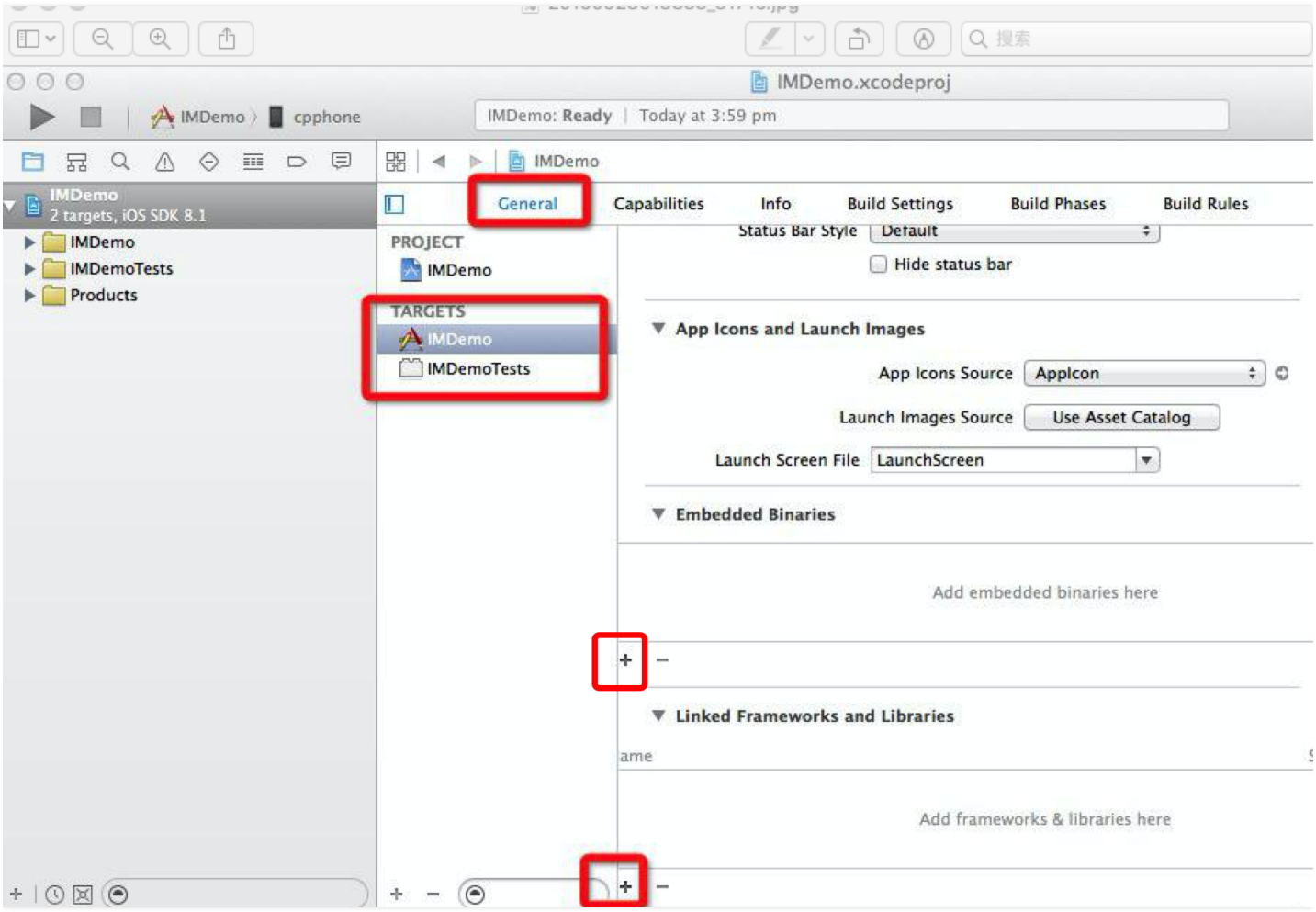


填入工程名（例如 IMDemo）：



3. 集成 IM SDK

添加依赖库：选中 IMDemo 的【Target】，在【General】面板中的 【Embedded Binaries】和【Linked Frameworks and Libraries】添加依赖库。若使用标准版 SDK，请选择 ImSDK.framework；若使用精简版 SDK，请选择 ImSDK_Smart.framework。



设置链接参数：在【Build Setting】-【Other Linker Flags】添加 `-ObjC`。

引用 IM SDK

项目代码中使用 SDK 有两种方式：

方式一

在 Xcode > Build Setting > Header Search Paths 设置 SDK 头文件的路径，然后在项目需要使用 SDK API 的文件里，引入具体的头文件。

- 如果使用标准版，请按照如下方式引用头文件：

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

- 如果使用精简版，请按照如下方式引用头文件：

```
#import "ImSDK_Smart.h"
```

方式二

在项目需要使用 SDK API 的文件里，引入具体的头文件。

- 如果使用标准版，请按照如下方式引用头文件：

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

- 如果使用精简版，请按照如下方式引用头文件：

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

集成 SDK（Mac）

最近更新时间：2020-06-29 15:49:31

本文主要介绍如何快速地将腾讯云 IM SDK（Mac）集成到您的项目中，只要按照如下步骤进行配置，就可以完成 SDK 的集成工作。

开发环境要求

- Xcode 9.0+。
- OS X 10.10+ 的 Mac 真机。
- 项目已配置有效的开发者签名。

集成 IM SDK

您可以选择使用 CocoaPods 自动加载的方式，或者先下载 SDK 再将其导入到您当前的工程项目中。

CocoaPods 自动加载

1. 安装 CocoaPods

在终端窗口中输入如下命令（需要提前在 Mac 中安装 Ruby 环境）：

```
sudo gem install cocoapods
```

2. 创建 Podfile 文件

进入项目所在路径输入以下命令行，之后项目路径下会出现一个 Podfile 文件。

```
pod init
```

3. 编辑 Podfile 文件

编辑 Podfile 文件，按如下方式设置：

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

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

4. 更新并安装 SDK

在终端窗口中输入如下命令以更新本地库文件，并安装 TXIMSDK_Mac：

```
pod install
```

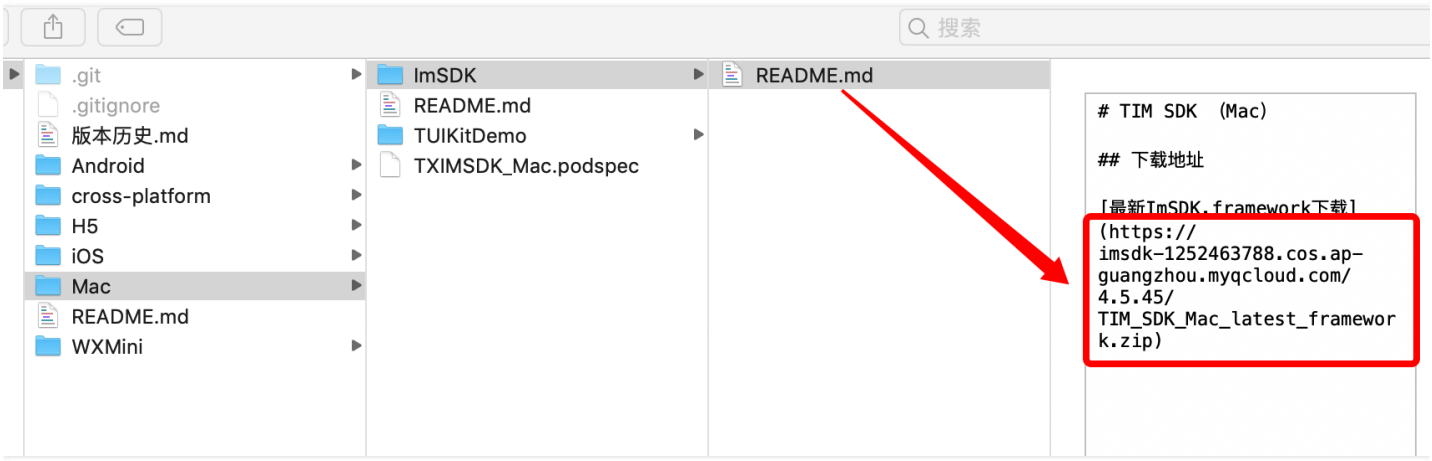
或使用以下命令更新本地库版本:

```
pod update
```

pod 命令执行完后，会生成集成了 SDK 的 .xcworkspace 后缀的工程文件，双击打开即可。

手动集成

1. 从 Github 获取 SDK 的下载地址：

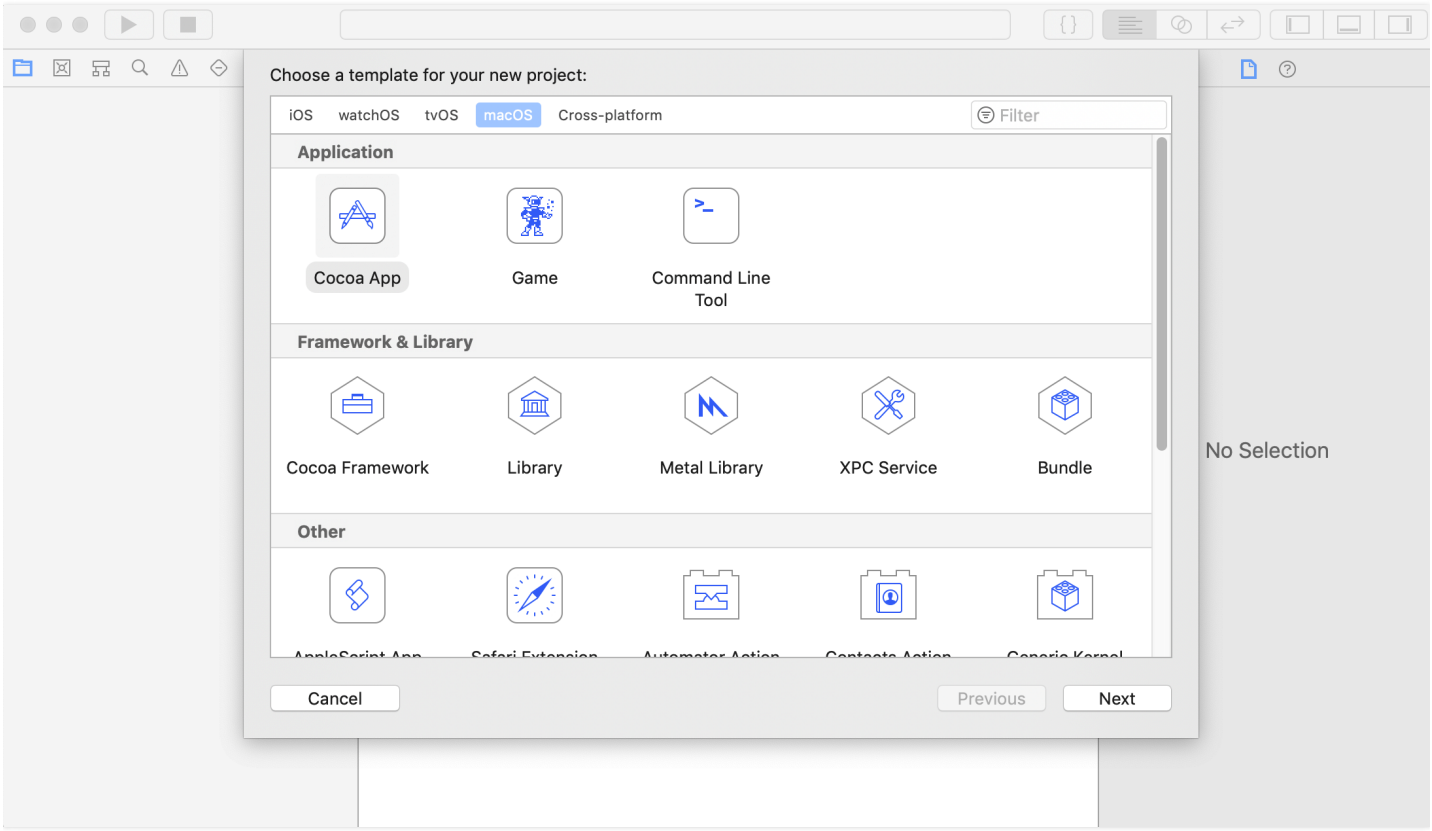


- ImSDKForMac.framework 为 IM SDK 的核心动态库文件。

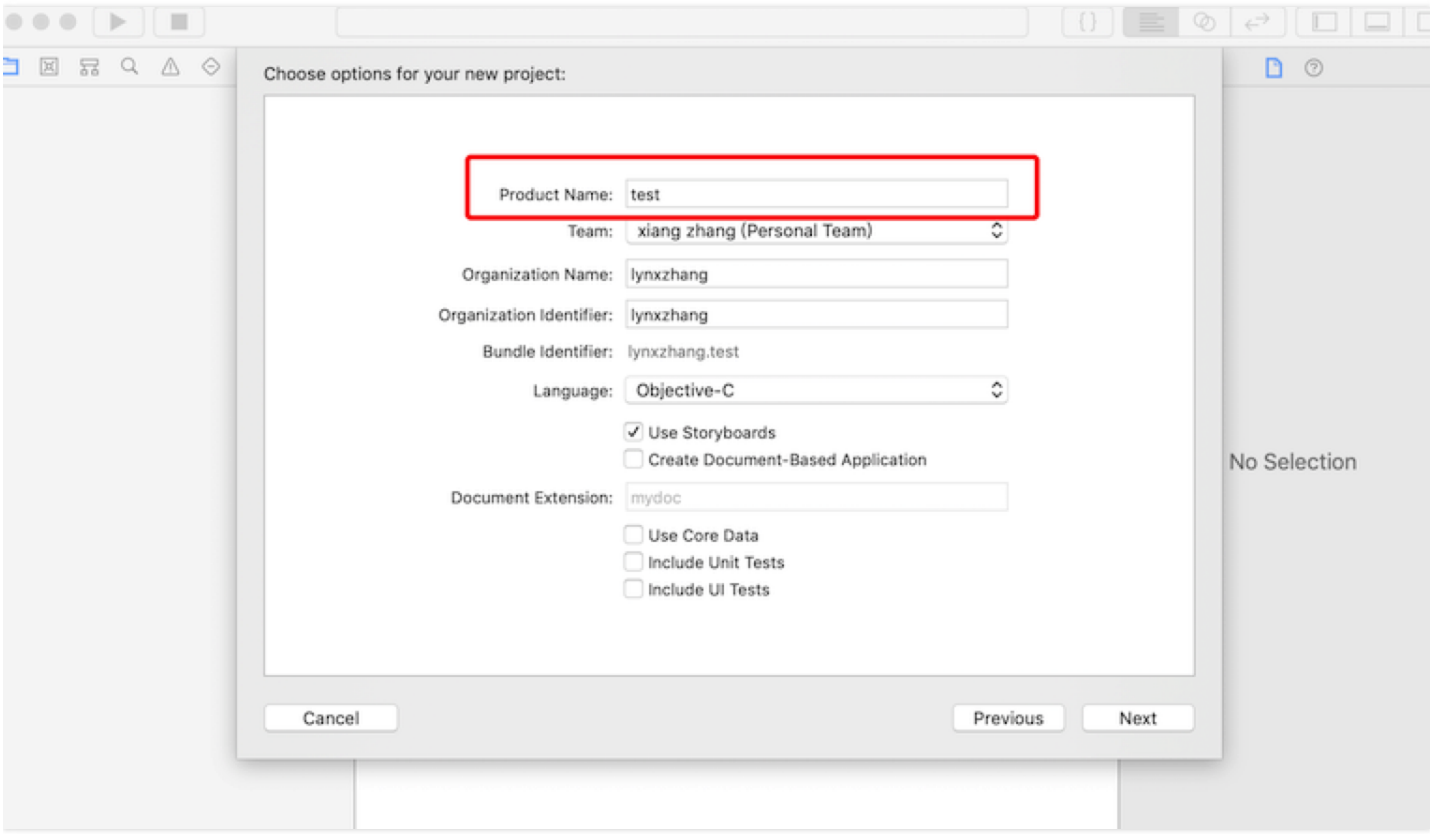
包名	介绍	ipa增量
ImSDKForMac.framework	即时通信 IM 功能包	1.4M

2. 创建工程

创建一个新的工程：

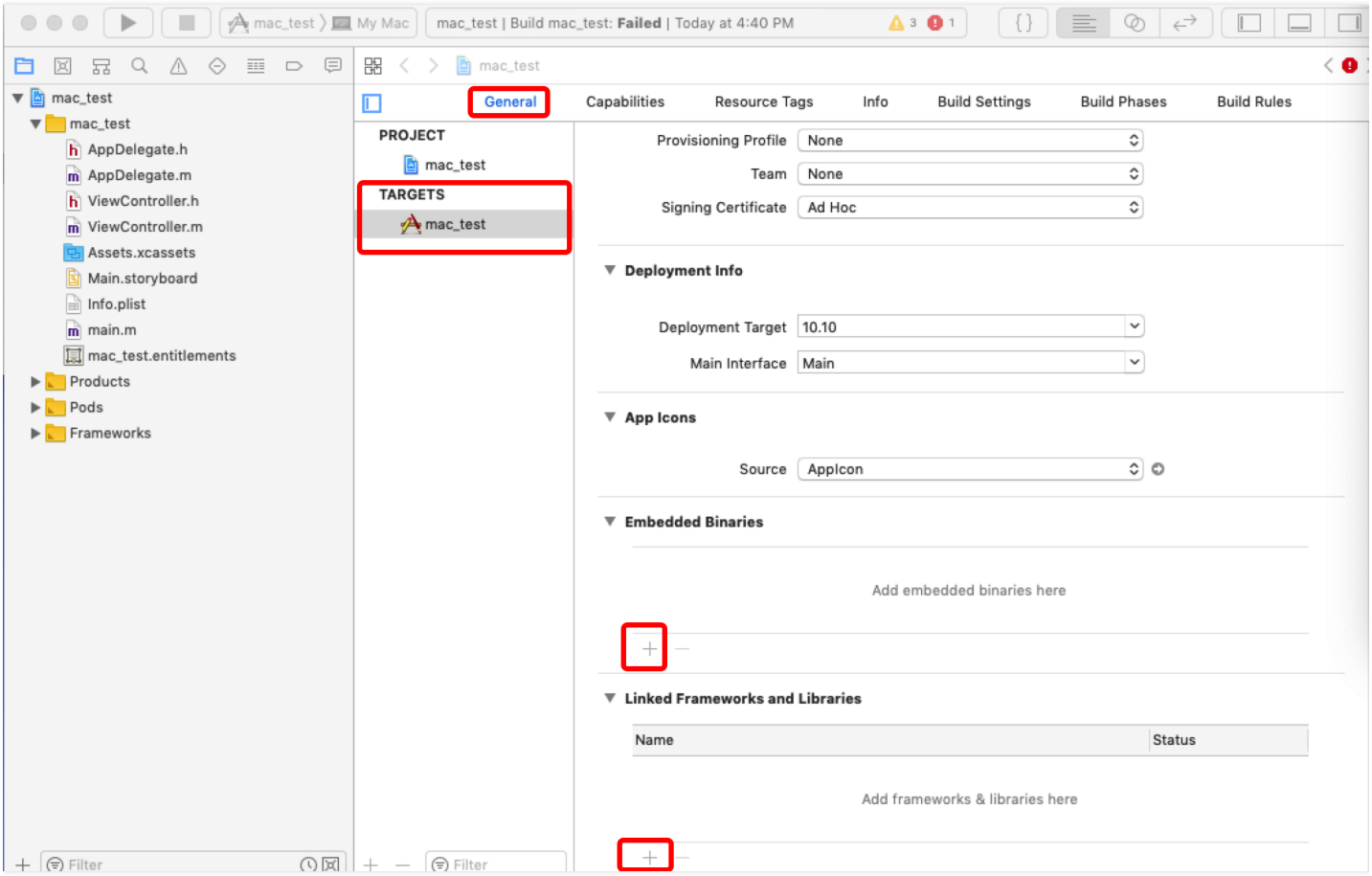


填入工程名：



2. 集成 IM SDK

添加依赖库：选中 Demo 的【Target】，在【General】面板中的 【Embedded Binaries】和【Linked Frameworks and Libraries】添加依赖库。



添加依赖库：

ImSDKForMac.framework

注意：
需要在【Build Setting】-【Other Linker Flags】添加 -ObjC 。

引用 IM SDK

项目代码中使用 SDK 有两种方式：

- 方式一： 在 Xcode -> Build Setting -> Herader Search Paths 设置 ImSDKForMac.framework/Headers 路径，在项目需要使用 SDK API 的文件里，直接引用头文件"ImSDK.h"。

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

- 方式二：在项目需要使用 SDK API 的文件里，引入具体的头文件 < ImSDKForMac/ImSDK.h >。

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

集成 SDK（Web & 小程序）

最近更新时间：2020-05-29 16:06:09

本文主要介绍如何快速地将腾讯云 IM SDK 集成到您的 Web 或者小程序项目中。

- 您可以通过 NPM 和 Script 方式将 IM SDK 集成到您的 Web 项目中，推荐使用 NPM 集成。
- 您可以通过 NPM 方式将 IM SDK 集成到您的小程序项目中。

以下视频将帮助您快速了解如何将 IM SDK 集成到您的 Web 或者小程序项目中：

点击查看视频

准备工作

- 已创建即时通信 IM 应用并获取 SDKAppID。
- 已获取密钥信息。

相关文档

- 一分钟跑通 Demo
- IM SDK（小程序）Demo 运行
- IM SDK（Web）Demo 运行

集成 SDK

NPM 集成（推荐）

在您的项目中使用 NPM 安装相应的 IM SDK 依赖。

Web 项目

```
// IM Web SDK
npm install tim-js-sdk --save
// 发送图片、文件等消息需要的 COS SDK
npm install cos-js-sdk-v5 --save
```

说明：
若同步依赖过程中出现问题，请切换 npm 源后再次重试。

```
// 切换 cnpm 源
npm config set registry http://r.cnpmjs.org/
```

在项目脚本里引入模块。

```
import TIM from 'tim-js-sdk';
import COS from "cos-js-sdk-v5";

let options = {
  SDKAppID: 0 // 接入时需要将0替换为您的即时通信 IM 应用的 SDKAppID
};
// 创建 SDK 实例，`TIM.create()`方法对于同一个 `SDKAppID` 只会返回同一份实例
let tim = TIM.create(options); // SDK 实例通常用 tim 表示

// 设置 SDK 日志输出级别，详细分级请参见 setLogLevel 接口的说明
tim.setLogLevel(0); // 普通级别，日志量较多，接入时建议使用
// tim.setLogLevel(1); // release 级别，SDK 输出关键信息，生产环境时建议使用
```

```
// 注册 COS SDK 插件
tim.registerPlugin({'cos-js-sdk': COS});
```

小程序项目

```
// IM 小程序 SDK
npm install tim-wx-sdk --save
// 发送图片、文件等消息需要的 COS SDK
npm install cos-wx-sdk-v5 --save
```

说明：
若同步依赖过程中出现问题，请切换 npm 源后再次重试。

```
// 切换 cnpm 源
npm config set registry http://r.cnpmjs.org/
```

在项目脚本里引入模块，并初始化。

```
import TIM from 'tim-wx-sdk';
import COS from "cos-wx-sdk-v5";

let options = {
  SDKAppID: 0 // 接入时需要将0替换为您的即时通信 IM 应用的 SDKAppID
};
// 创建 SDK 实例，`TIM.create()`方法对于同一个 `SDKAppID` 只会返回同一份实例
let tim = TIM.create(options); // SDK 实例通常用 tim 表示

// 设置 SDK 日志输出级别，详细分级请参见 setLogLevel 接口的说明
tim.setLogLevel(0); // 普通级别，日志量较多，接入时建议使用
// tim.setLogLevel(1); // release 级别，SDK 输出关键信息，生产环境时建议使用

// 注册 COS SDK 插件
tim.registerPlugin({'cos-wx-sdk': COS});
```

更详细的初始化流程和 API 使用介绍请参见 SDK 初始化。

Script 集成

在您的项目中使用 script 标签引入 SDK，并初始化。

```
<!-- tim-js.js 可以从 https://github.com/tencentyun/TIMSDK/tree/master/H5/sdk 获取 -->
<script src="./tim-js.js"></script>
<!-- cos-js-sdk-v5.min.js 可以从 https://github.com/tencentyun/cos-js-sdk-v5/tree/master/dist 获取 -->
<script src="./cos-js-sdk-v5.min.js"></script>
<script>
var options = {
SDKAppID: 0 // 接入时需要将0替换为您的即时通信 IM 应用的 SDKAppID
};
// 创建 SDK 实例，`TIM.create()`方法对于同一个 `SDKAppID` 只会返回同一份实例
var tim = TIM.create(options);
// 设置 SDK 日志输出级别，详细分级请参见 setLogLevel 接口的说明
tim.setLogLevel(0); // 普通级别，日志量较多，接入时建议使用
// tim.setLogLevel(1); // release 级别，SDK 输出关键信息，生产环境时建议使用

// 注册 COS SDK 插件
tim.registerPlugin({'cos-js-sdk': COS});

// 接下来可以通过 tim 进行事件绑定和构建 IM 应用
</script>
```

说明：
设置 SDK 日志输出级别，详细分级请参见 setLogLevel 接口的说明。

更详细的初始化流程和 API 使用介绍请参见 SDK 初始化。

相关资源

- SDK 更新日志
- SDK 接口文档
- 常见问题
- IM Web Demo
- 腾讯云 COS JS SDK 下载地址

常见问题

1. 小程序如果需要上线或者部署正式环境怎么办？

请在【微信公众平台】>【开发】>【开发设置】>【服务器域名】中进行域名配置：

将以下域名添加到 **request 合法域名**：

域名	说明	是否必须
https://webim.tim.qq.com	Web IM 业务域名	必须
https://yun.tim.qq.com	Web IM 业务域名	必须
https://events.tim.qq.com	Web IM 业务域名	必须
https://grouptalk.c2c.qq.com	Web IM 业务域名	必须
https://pingtas.qq.com	Web IM 统计域名	必须

将以下域名添加到 **uploadFile 合法域名**：

域名	说明	是否必须
https://cos.ap-shanghai.myqcloud.com	文件上传域名	必须

将以下域名添加到 **downloadFile 合法域名**：

域名	说明	是否必须
https://cos.ap-shanghai.myqcloud.com	文件下载域名	必须

集成 SDK（Windows）

最近更新时间：2019-11-07 17:29:25

本文介绍如何快速地将腾讯云的 IM SDK 集成到项目中，只要按照如下步骤进行操作，可以轻松完成 IM SDK 的集成工作。

开发环境要求

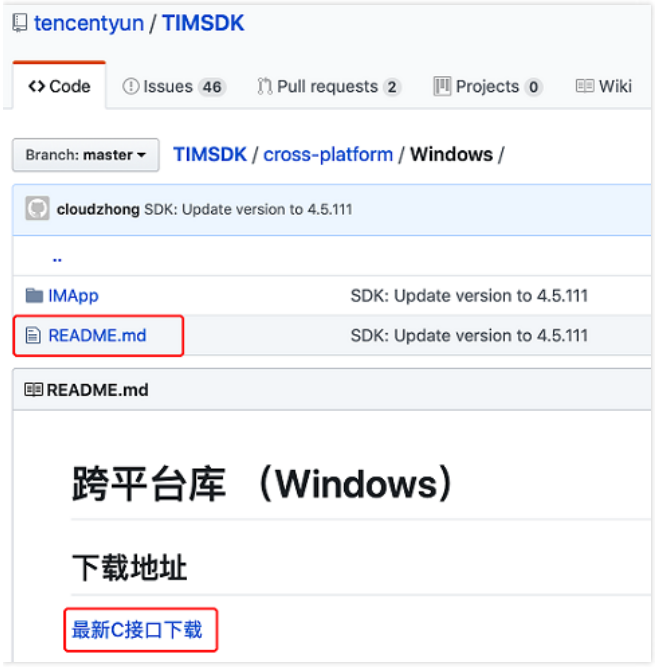
- 操作系统：最低要求是 Windows 7。
- 开发环境：最低版本要求是 Visual Studio 2010，推荐使用 Visual Studio 2015。

集成 IM SDK

下面通过创建一个简单的 MFC 项目，介绍如何在 Visual Studio 2015 工程中集成 SDK。

步骤1. 下载 IM SDK

在 Github 下载 Windows IM SDK，Windows IM SDK 的下载方式如下：

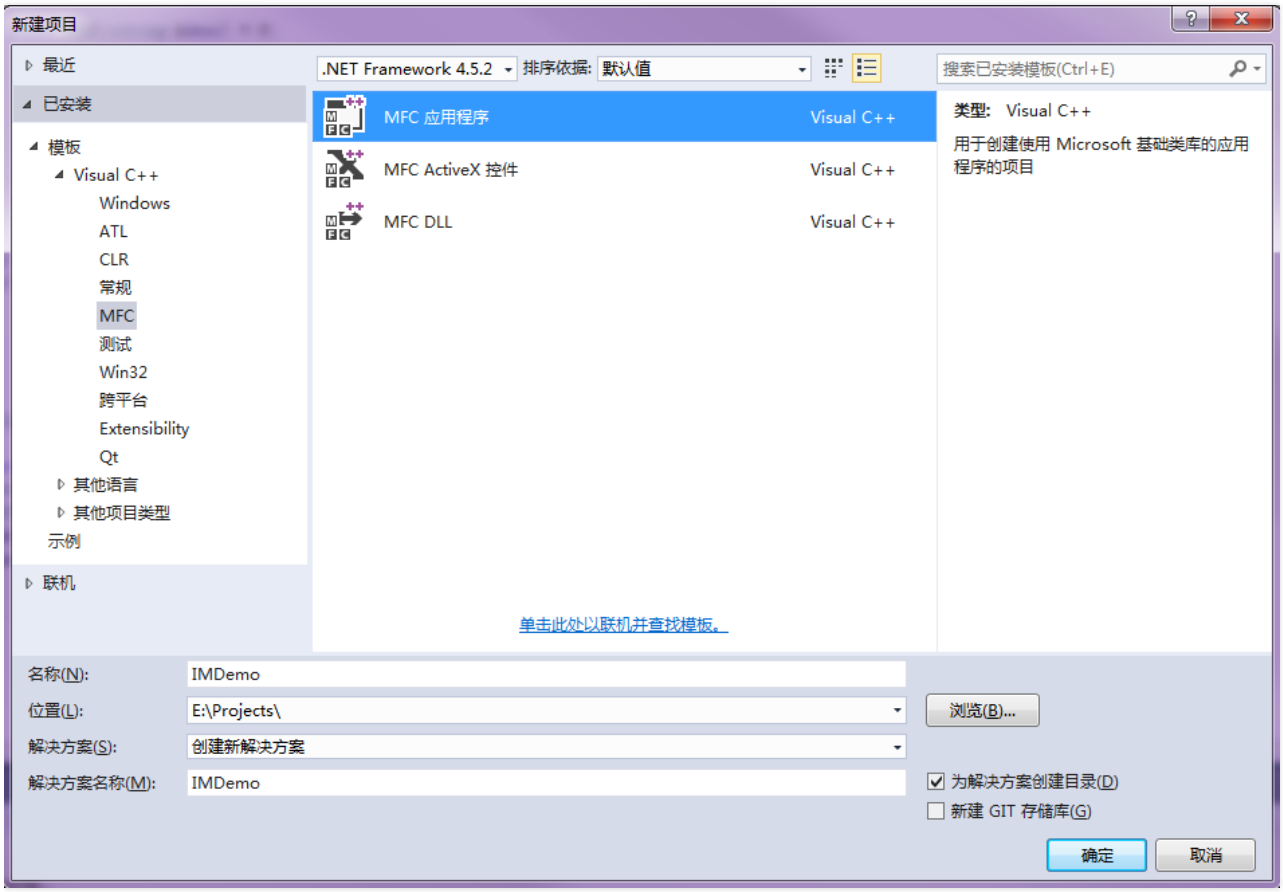


下载并解压打开 IM SDK 文件夹，包含以下几个部分：

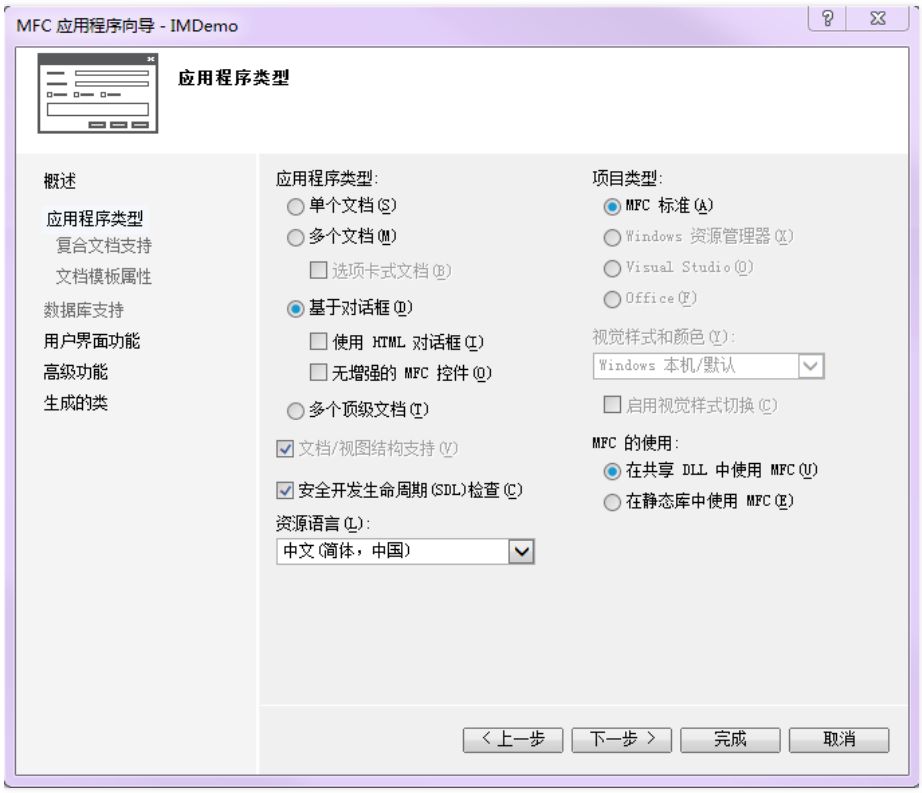
目录名	说明
includes	接口头文件
lib\Win32\Debug	**32位 Debug模式**，采用/MTd 编译生成的 .lib 静态库文件和 dll 动态库文件
lib\Win32\Release	**32位 Release模式**，采用/MT 编译生成的 .lib 静态库文件和 dll 动态库文件
lib\Win64\Debug	**64位 Debug模式**，采用/MTd 编译生成的 .lib 静态库文件和 dll 动态库文件
lib\Win64\Release	**64位 Release模式**，采用/MT 编译生成的 .lib 静态库文件和 dll 动态库文件

步骤2. 新建工程

打开 Visual Studio，新建一个名为 IMDemo 的 MFC 应用程序，如下图所示：

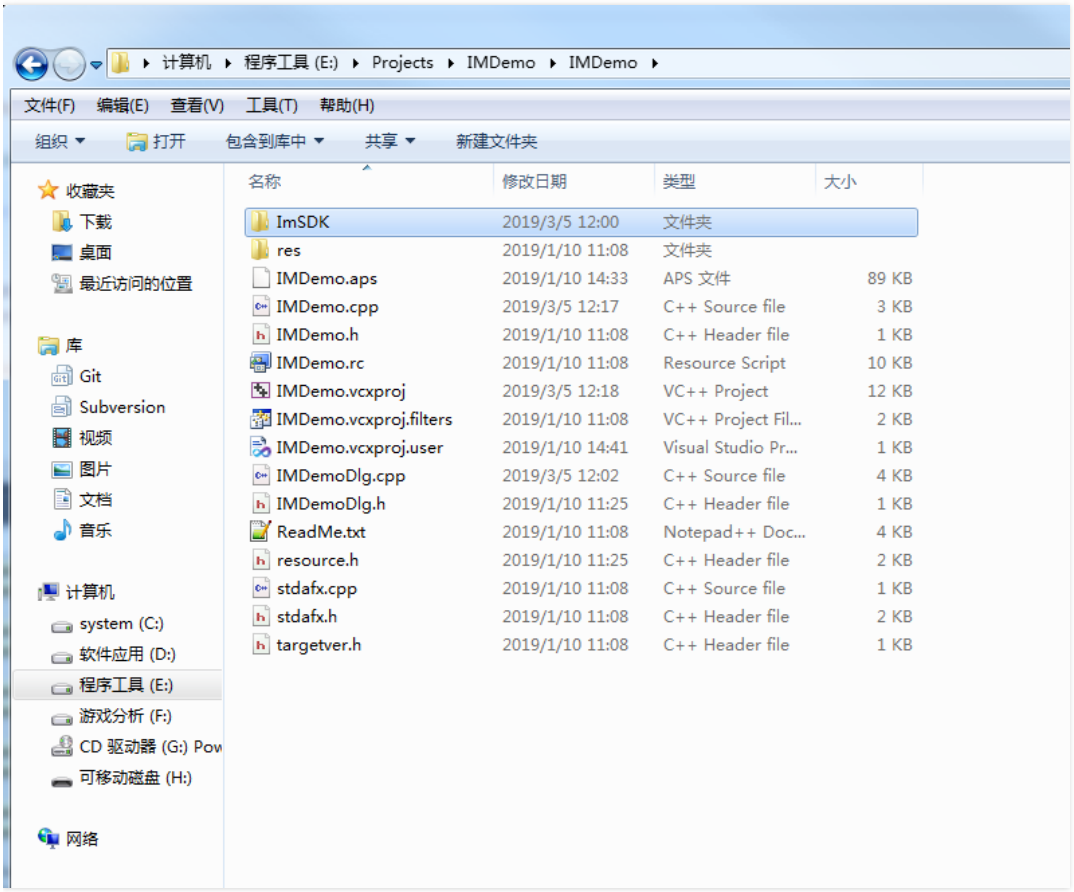


为了便于快速集成，在向导的【应用程序类型】页面，请选择比较简单的【基于对话框】类型，其他的向导配置，请选择默认的配置即可。如下图所示：



步骤3. 拷贝文件

将解压后的 IM SDK 文件夹拷贝到 IMDemo.vcxproj 所在目录下，如下图所示：

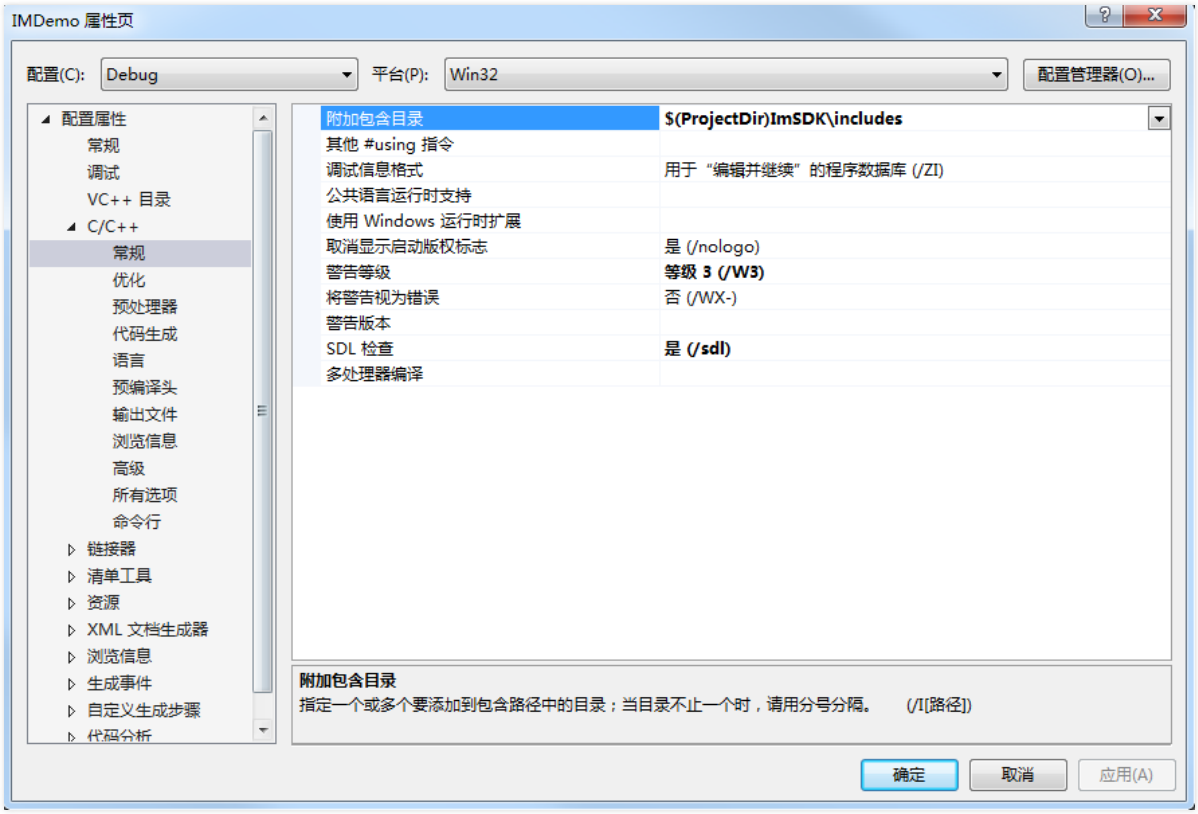


步骤4. 修改工程配置

IM SDK 提供了 **Debug** 和 **Release** 两种编译生成的静态库，针对这两种有些地方要专门配置。打开 IMDemo 属性页，在【解决方案资源管理器】>【IMDemo工程的右键菜单】>【属性】。

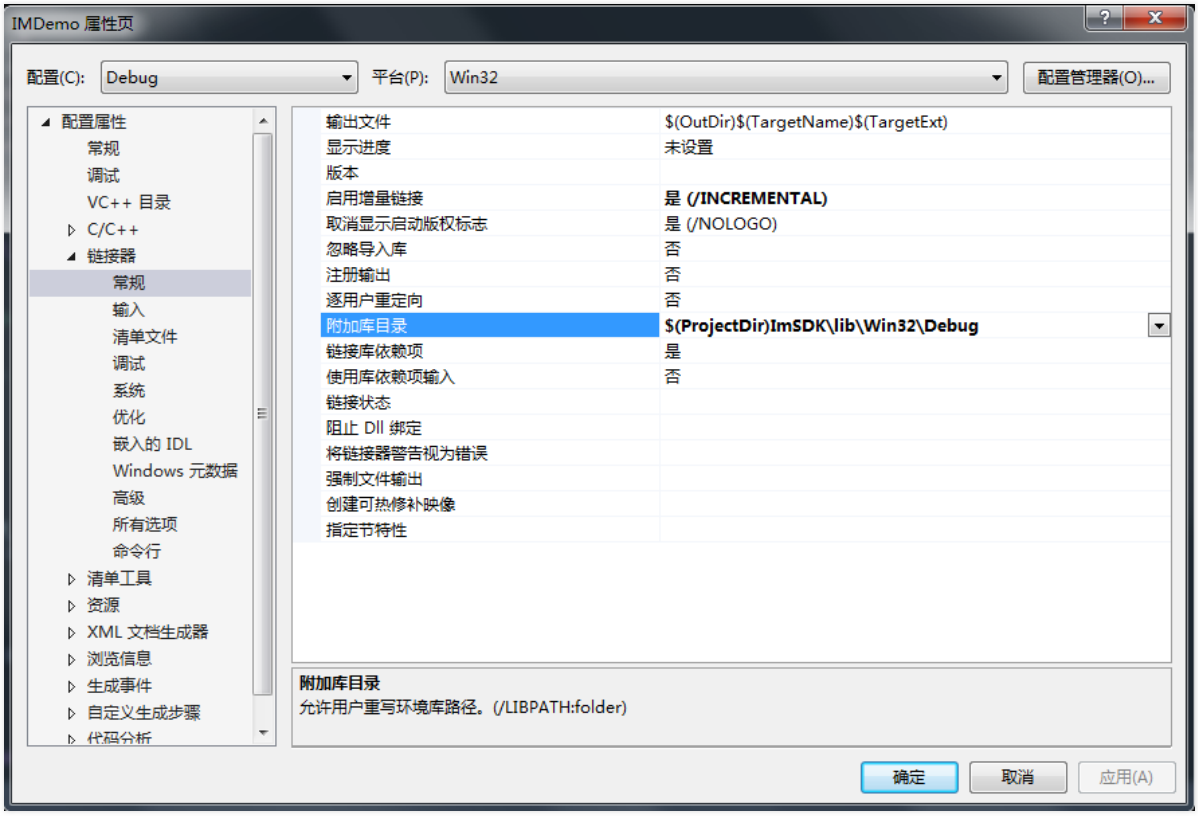
以**32位 Debug 模式**为例，请按照以下步骤进行配置：

1. 添加包含目录
   在 【C/C++】>【常规】>【附件包含目录】，添加 IM SDK 头文件目录 $(ProjectDir)ImSDK\includes，如下图所示：



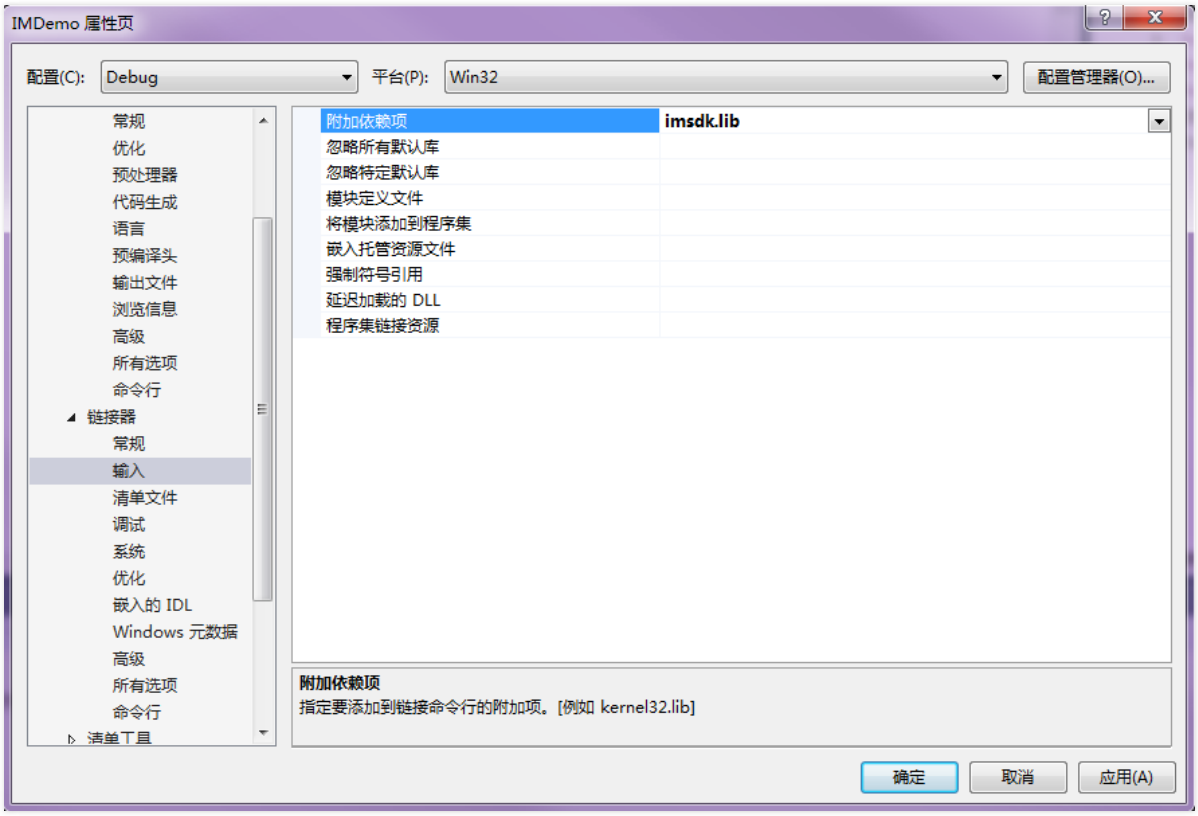
2. 添加库目录

在 【链接器】>【常规】>【附加库目录】，添加 IM SDK 库目录 $(ProjectDir)ImSDK\lib\Win32\Debug



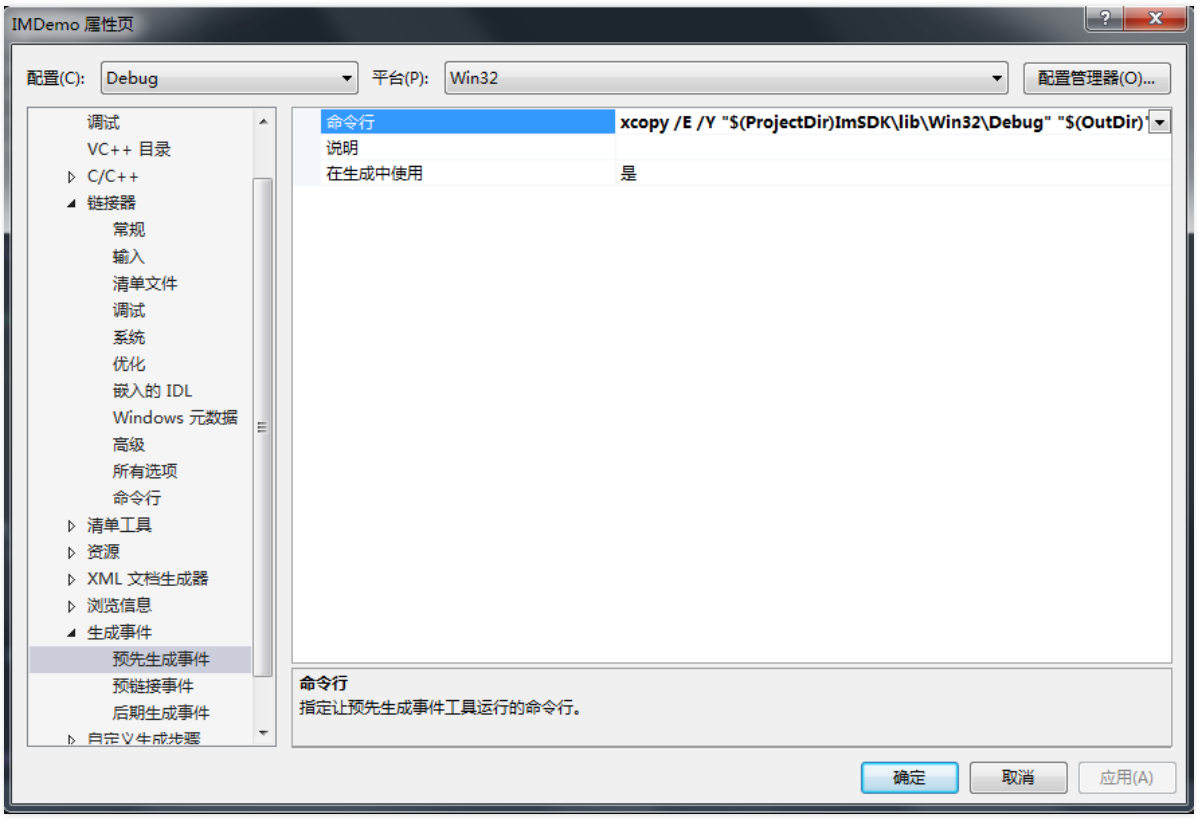
3. 添加库文件

在 【链接器】>【输入】>【附加依赖项】，添加 IM SDK 库文件 imsdk.lib ，如下图所示：



4. 拷贝 DLL 到执行目录

在【生成事件】>【预先生成事件】>【命令行】，输入 `xcopy /E /Y "$(ProjectDir)ImSDK\lib\Win32\Debug" "$(OutDir)"` ，拷贝 imsdk.dll 动态库文件到程序生成目录，如下图所示：

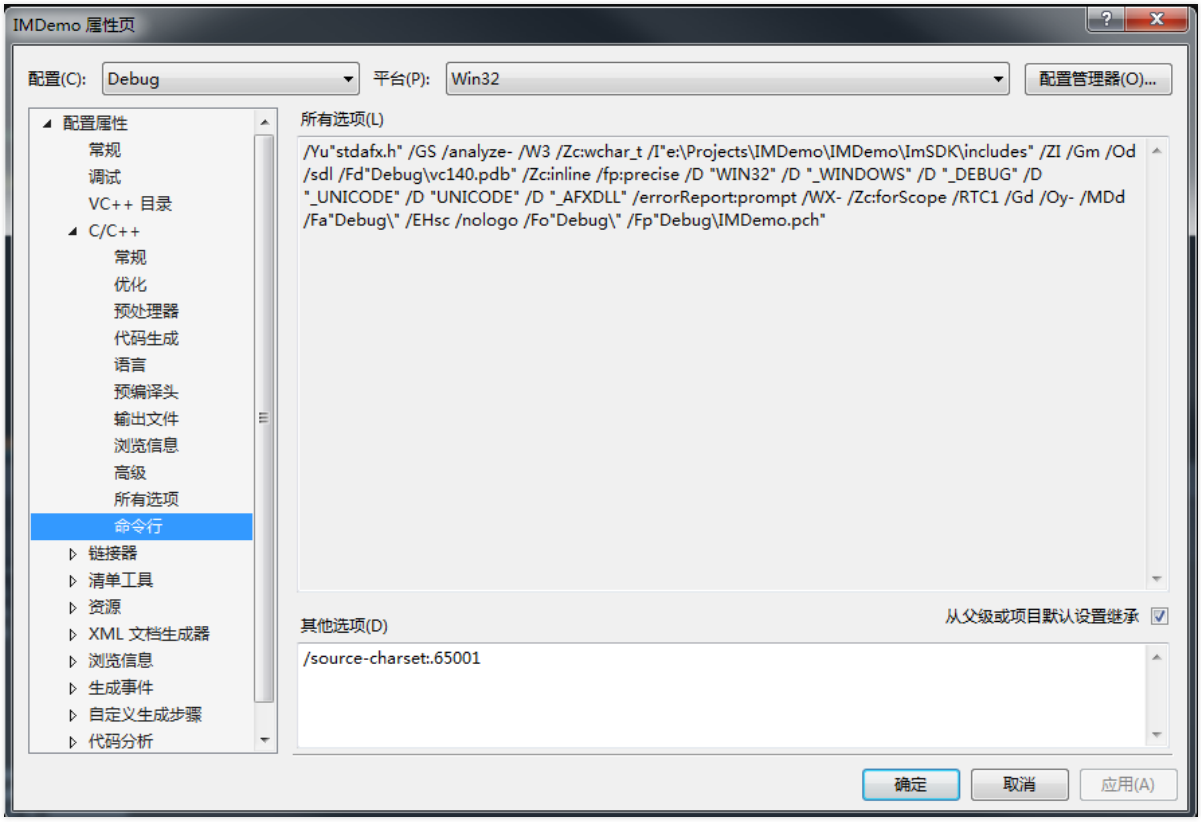


5. 指定源文件的编码格式

由于 IM SDK 的头文件采用 UTF-8 编码格式，部分编译器按默认系统编码格式编译源文件，可能导致编译无法通过，设置此参数可指定编译器按照 UTF-8 的编码格式编

译源文件。

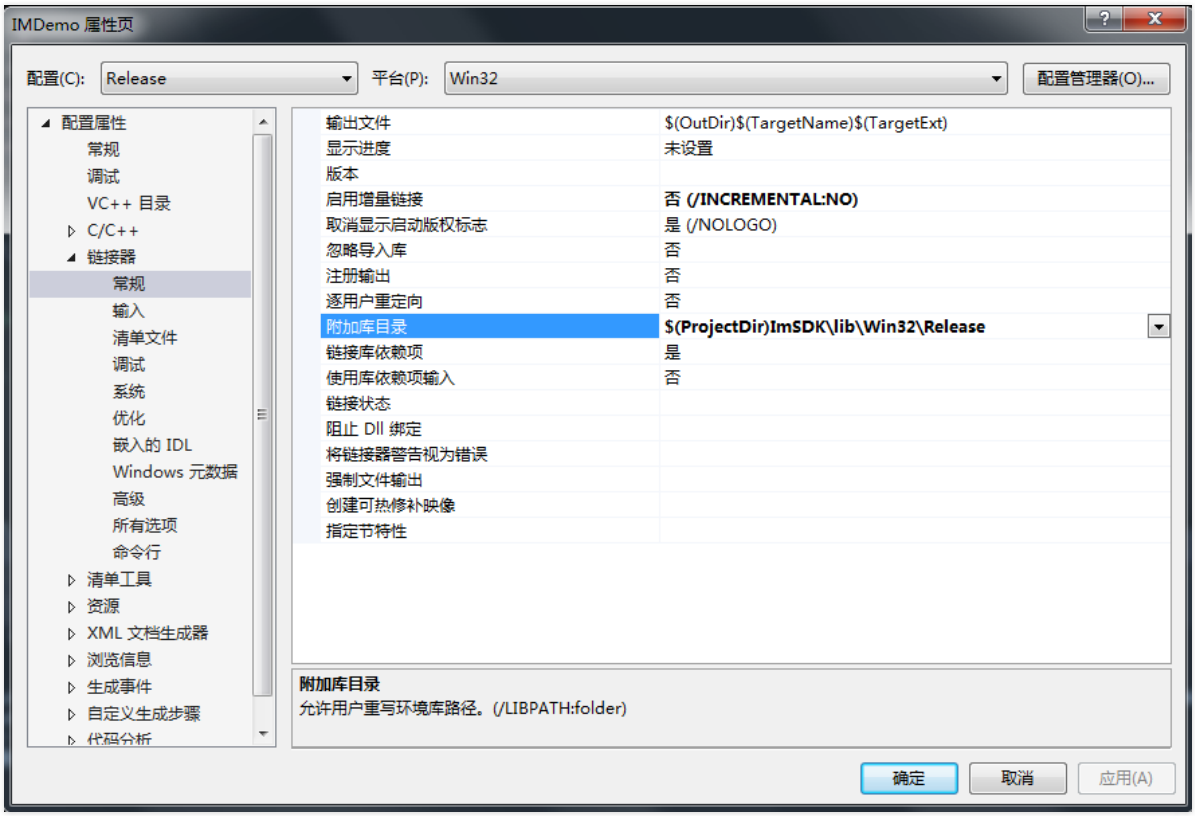
在 【C/C++】>【命令行】>【其他选项】，输入 /source-charset:.65001 ，如下图所示：



Release 模式具体设置如下：

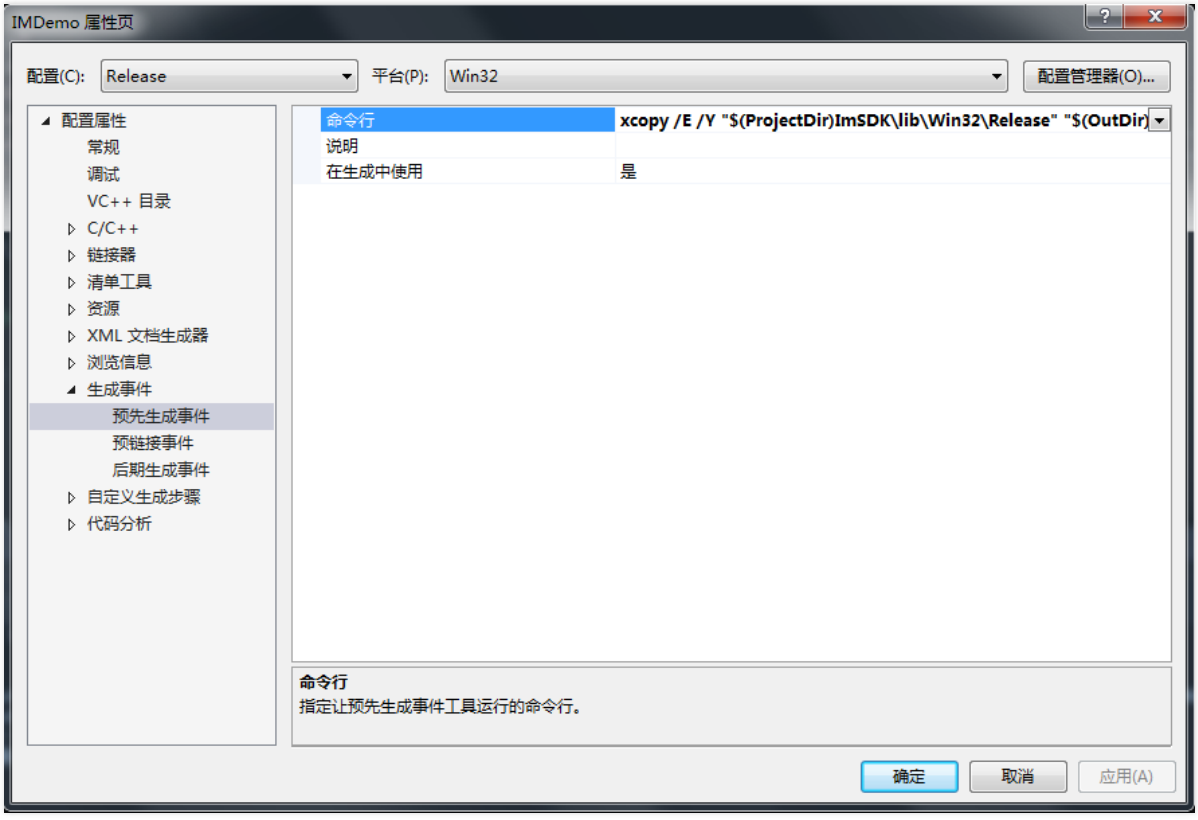
1. 添加库目录

   在 【链接器】>【常规】>【附加库目录】，添加 IM SDK 库目录 $(ProjectDir)ImSDK\lib\Win32\Release，如下图所示：



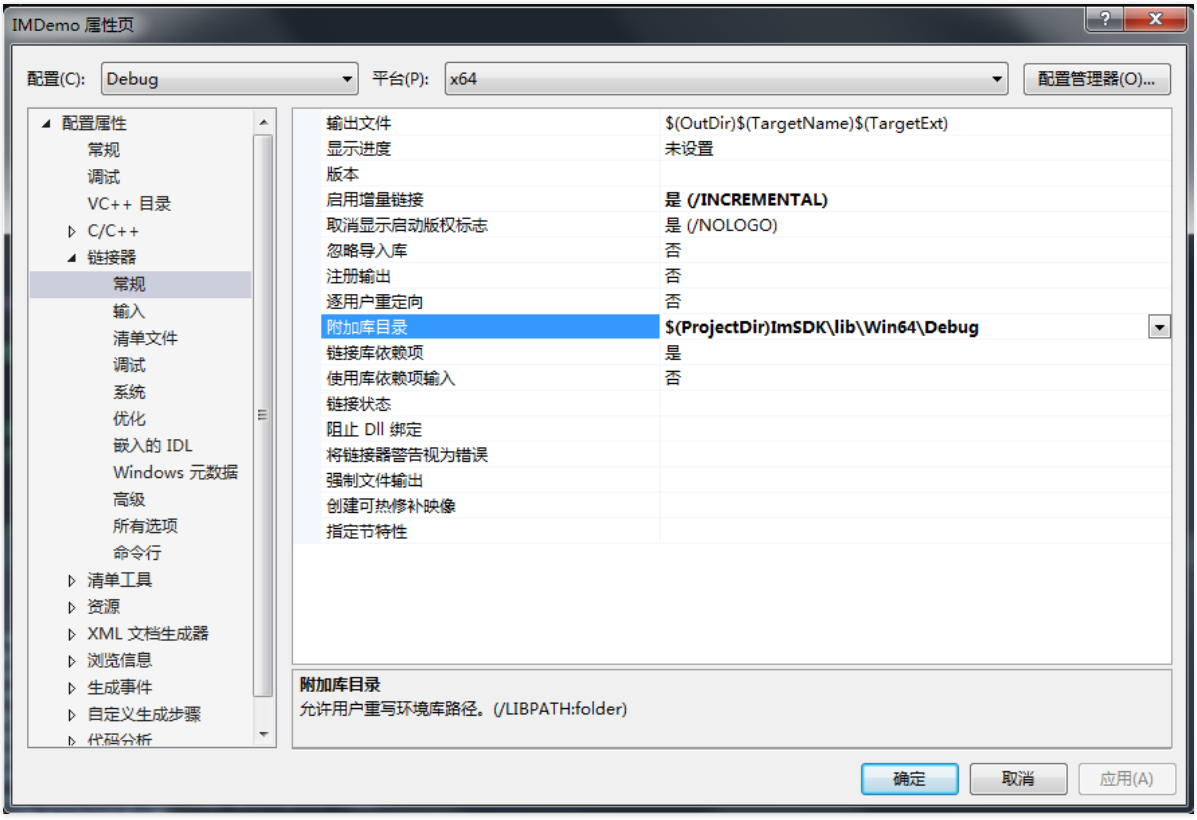
2. 拷贝 DLL 到执行目录

在【生成事件】>【预先生成事件】>【命令行】，输入 `xcopy /E /Y "$(ProjectDir)ImSDK\lib\Win32\Release" "$(OutDir)"` ，拷贝 imsdk.dll 动态库文件到程序生成目录，如下图所示：

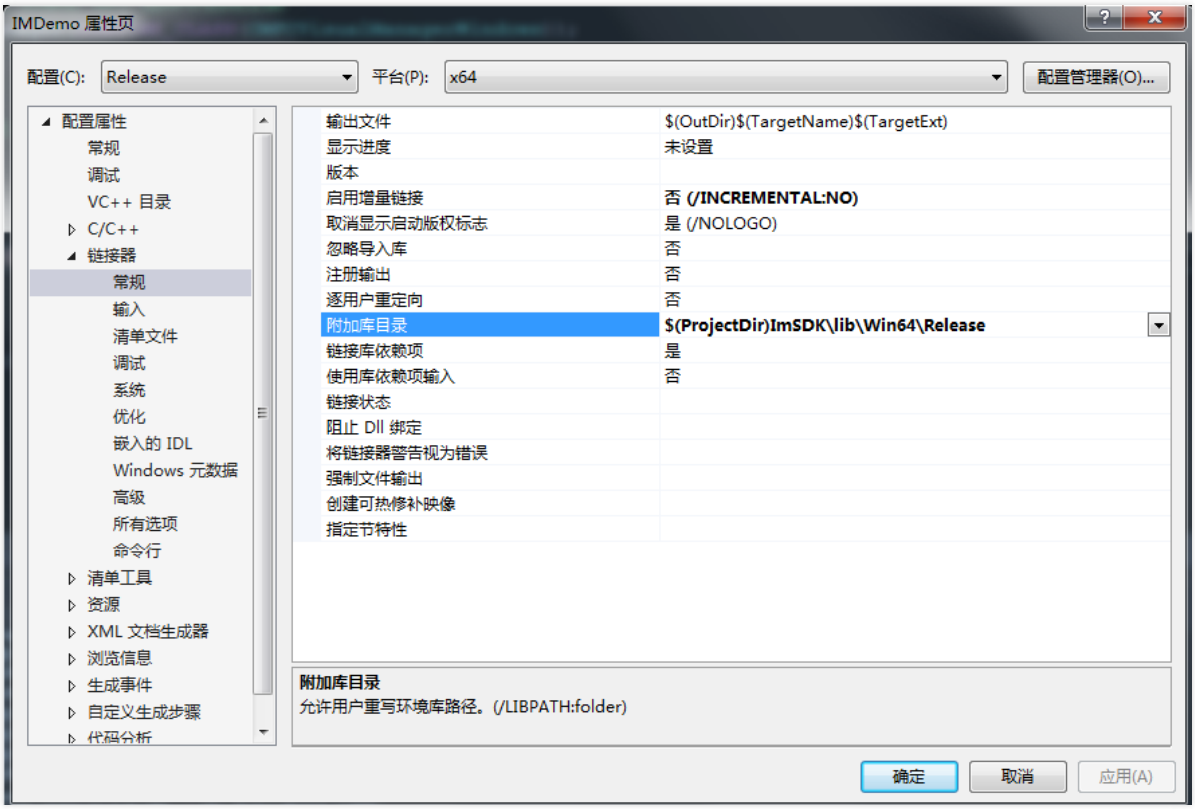


64位 Debug/Release 与 **32位** 的设置也大部分相同，不同在于 IM SDK 的库目录。具体如下

1. 添加库目录
   - **Debug 模式** 在 【链接器】>【常规】>【附加库目录】，添加 IM SDK 库目录 $(ProjectDir)ImSDK\lib\Win64\Debug，如下图所示：

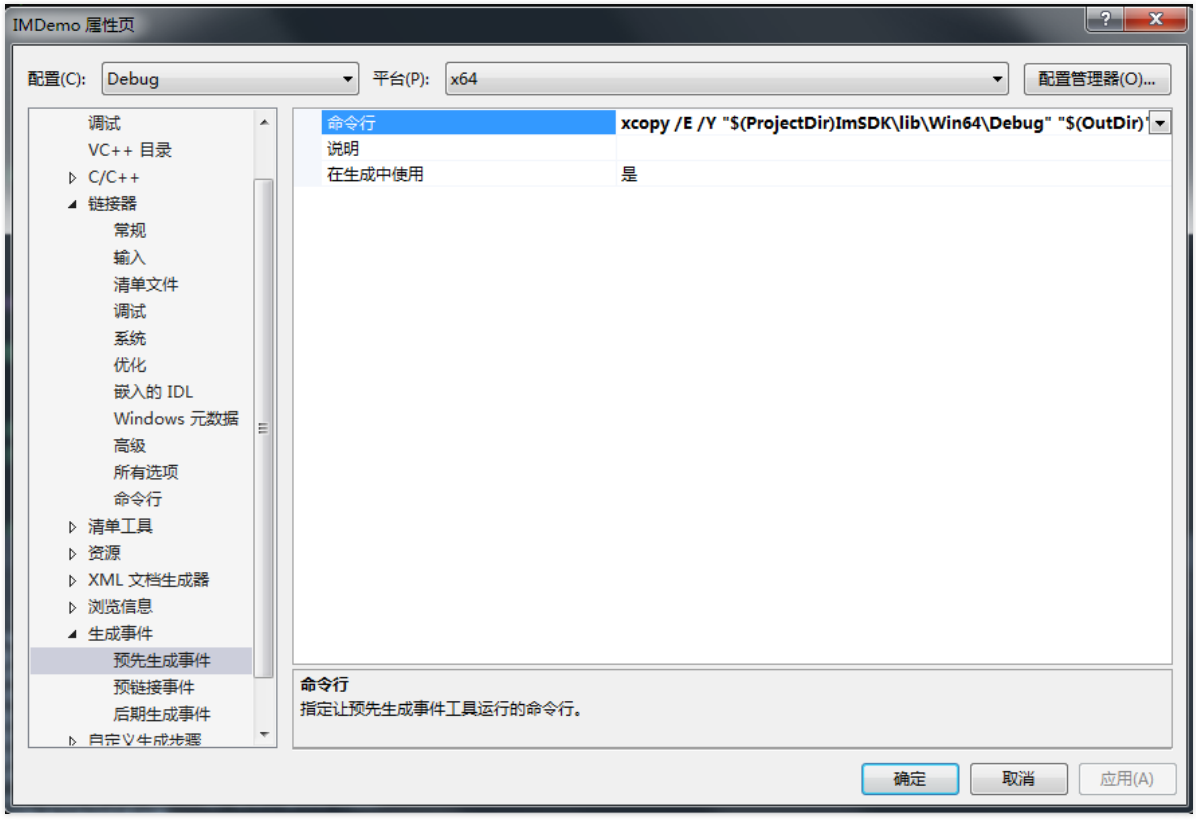


   - **Release 模式** 在 【链接器】>【常规】>【附加库目录】，添加 IM SDK 库目录 $(ProjectDir)ImSDK\lib\Win64\Release，如下图所示：

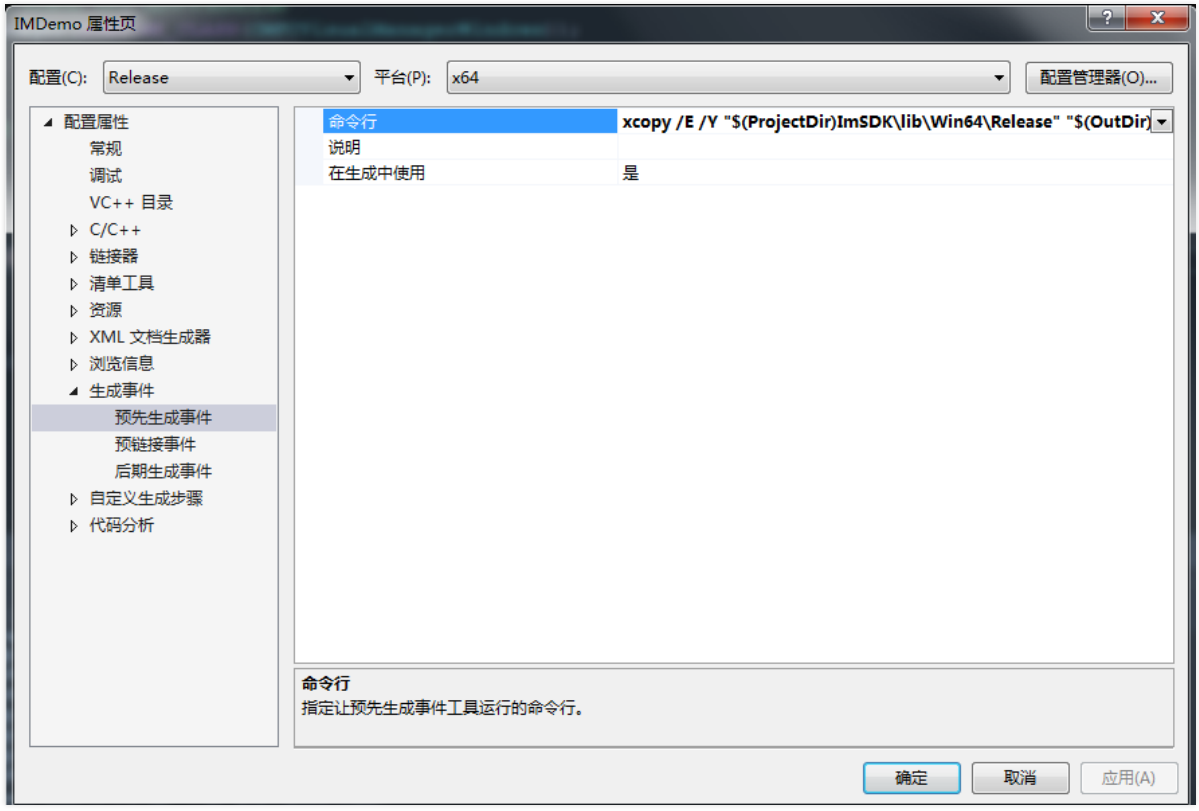


2. 拷贝 DLL 到执行目录

- **Debug 模式** 在【生成事件】>【预先生成事件】>【命令行】，输入 `xcopy /E /Y "$(ProjectDir)ImSDK\lib\Win64\Debug" "$(OutDir)"` ，拷贝 imsdk.dll 动态库文件到程序生成目录，如下图所示：



- **Release 模式** 在【生成事件】>【预先生成事件】>【命令行】，输入 `xcopy /E /Y "$(ProjectDir)ImSDK\lib\Win64\Release" "$(OutDir)"` ，拷贝 imsdk.dll 动态库文件到程序生成目录，如下图所示：



步骤5. 打印 IM SDK 版本号

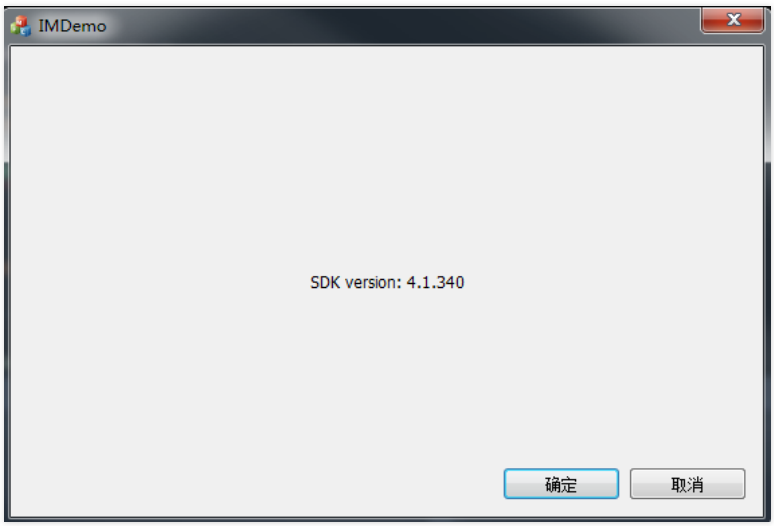
- 在 IMDemo.cpp 文件中，添加头文件包含：

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

- 在 CIMDemoDlg::OnInitDialog 函数中，添加下面的测试代码：

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

- 按键盘 F5 键运行，打印 IM SDK 的版本号，如下图所示：



常见问题

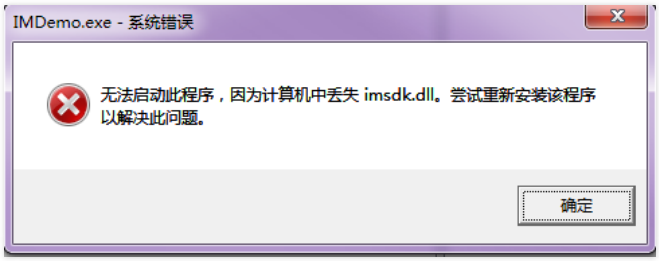
- 出现以下错误，请按照前面的工程配置，检查 IM SDK 头文件的目录是否正确添加：

```
fatal error C1083: 无法打开包括文件: “TIMCloud.h”: No such file or directory
```

- 出现以下错误，请按照前面的工程配置，检查 IM SDK 库目录和库文件是否正确添加：

```
LINK : fatal error LNK1104: 无法打开文件“imsdk.lib”

error LNK2019: 无法解析的外部符号 __imp__TIMGetSDKVersion，该符号在函数 "protected: virtual int __thiscall CIMDemoDlg::OnInitDialog(void)" (?OnInit
Dialog@CIMDemoDlg@@MAEHXZ) 中被引用
```

- 出现以下错误，请按照前面的工程配置，检查 IM SDK 的 DLL 是否拷贝到执行目录：



初始化与登录
初始化与登录（Android）

最近更新时间：2020-09-30 11:24:31

初始化

类 V2TIMManager 是 IM SDK 主核心类，负责 IM SDK 的初始化、登录、消息收发，建群退群等功能，是 IM SDK 的入口类。调用 initSDK 接口即可完成初始化：

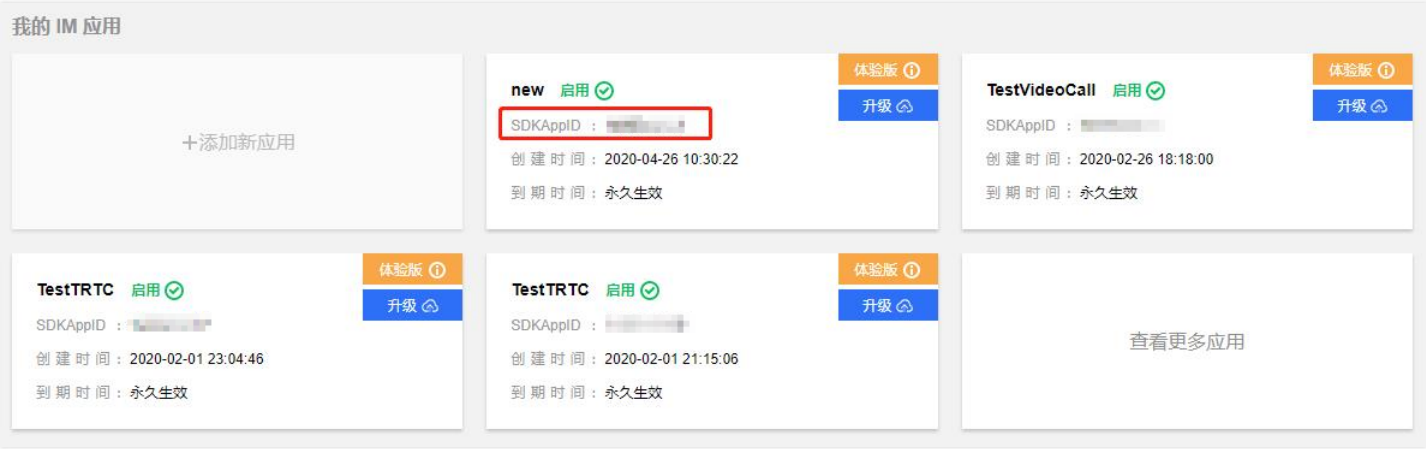
```
// 1. 从 IM 控制台获取应用 SDKAppID，详情请参考 SDKAppID。
// 2. 初始化 config 对象
V2TIMSDKConfig config = new V2TIMSDKConfig();
// 3. 指定 log 输出级别，详情请参考 SDKConfig。
config.setLogLevel(V2TIMSDKConfig.V2TIM_LOG_INFO);
// 4. 初始化 SDK 并设置 V2TIMSDKListener 的监听对象。
// initSDK 后 SDK 会自动连接网络，网络连接状态可以在 V2TIMSDKListener 回调里面监听。
V2TIMManager.getInstance().initSDK(context, sdkAppID, sdkConfig, new V2TIMSDKListener() {
// 5. 监听 V2TIMSDKListener 回调
@Override
public void onConnecting() {
// 正在连接到腾讯云服务器
}
@Override
public void onConnectSuccess() {
// 已经成功连接到腾讯云服务器
}
@Override
public void onConnectFailed(int code, String error) {
// 连接腾讯云服务器失败
}
});
```

初始化接口 initSDK(SDKAppID, SDKConfig, listener) 包含三个必填的参数，分别是 SDKAppID、SDKConfig 和事件监听器。

SDKAppID

SDKAppID 即应用 ID，它是腾讯云 IM 服务用于区分客户帐号的唯一标识。每一个独立的 App 都建议申请一个新的 SDKAppID，不同 SDKAppID 之间的消息天然隔离，不能互通。

您可以在 即时通信 IM 控制台 查看所有的 SDKAppID，单击【添加新应用】即可创建新的 SDKAppID。



SDKConfig

参数 V2TIMSDKConfig 用于对 SDK 进行初始化配置，常用于设置日志级别，即 setLogLevel 接口，日志级别如下表所示：

日志级别	LOG 输出量
V2TIM_LOG_NONE	不输出任何 log

日志级别	LOG 输出量
V2TIM_LOG_DEBUG	输出 DEBUG，INFO，WARNING，ERROR 级别的 log
V2TIM_LOG_INFO	输出 INFO，WARNING，ERROR 级别的 log
V2TIM_LOG_WARN	输出 WARNING，ERROR 级别的 log
V2TIM_LOG_ERROR	输出 ERROR 级别的 log

- IM SDK 的日志在4.8.50版本之前默认存储于 `/sdcard/tencenet/imsdklogs/应用包名` 目录下，4.8.50及之后的版本存储于 `/sdcard/Android/data/包名/files/log/tencent/imsdk` 目录下。
- 从 V4.7.1 开始，IM SDK 的日志开始采用微信团队的 xlog 模块进行输出，xlog 日志默认是压缩的，需要使用 Python 脚本进行解压。
  - 获取解压脚本：若使用 Python 2.7，则单击 Decode Log 27 获取解压脚本；若使用 Python 3.0，则单击 Decode Log 30 获取解压脚本。
  - 在 Windows 或者 Mac 控制台输入如下命令即可对 log 文件进行解压，解压后的文件以 xlog.log 结尾，可以直接使用文本编辑器打开。

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

事件监听器

V2TIMSDKListener 提供了网络状态以及用户信息变更的监听：

事件回调	事件描述	推荐操作
onConnecting	正在连接到腾讯云服务器	适合在 UI 上展示“正在连接”状态。
onConnectSuccess	已经成功连接到腾讯云服务器	-
onConnectFailed	连接腾讯云服务器失败	可以提示用户当前网络连接不可用。
onKickedOffline	当前用户被踢下线	此时可以 UI 提示用户“您已经在其他端登录了当前账号，是否重新登录？”
onUserSigExpired	登录票据已经过期	请使用新签发的 UserSig 进行登录。
onSelfInfoUpdated	当前用户的资料发生了更新	可以在 UI 上更新自己的头像和昵称。

注意：
若收到 `onUserSigExpired` 回调，说明您登录用的 UserSig 票据已经过期，请更新后重新登录。如果继续使用过期的 UserSig，会导致 SDK 登录死循环。

登录

调用 `V2TIMManager` 的 login(userID, userSig) 函数可以进行登录，只有在 SDK 登录成功后，才能使用 IM SDK 的各项能力。

- UserID： 建议只包含大小写英文字母（a-z、A-Z）、数字（0-9）、下划线（_）和连词符（-），长度最大不超过32字节。
- UserSig：IM SDK 登录票据，由您的业务服务器进行计算以保证安全，计算方法请参考 UserSig 后台 API。

登录时机

以下场景需调用登录：

- App 启动后首次使用 IM SDK 的能力时。
- IM SDK 抛出 `onUserSigExpired` 回调，即登录票据已过期时，需要使用新的 UserSig 进行登录。
- IM SDK 抛出 `onKickOffline` 回调，即当前用户被踢下线时，可以通过 UI 提示用户“您已经在其他端登录了当前账号，是否重新登录？” 如果用户选择“是”，就可以进行重新登录。

以下场景无需调用登录：

- 用户的网络断开并重新连接后，不需要调用 login 函数，SDK 会自动上线。
- 当一个登录过程在进行时，不需要进行重复登录。

多端登录

同样类型的两台手机不能同时登录一个帐号，例如两台苹果手机不能同时登录一个帐号。但是一台安卓手机和一台苹果手机会被认为是两端，可以同时登录。多端登录相关配置请参考 登录设置。

登出

登出操作相对简单，使用 logout 函数即可。

初始化与登录（iOS）

最近更新时间：2020-05-14 22:18:05

初始化

类 V2TIMManager 是 IM SDK 主核心类也是 IM SDK 的入口类，负责 IM SDK 的初始化、登录、消息收发，建群退群等功能。调用 initSDK 接口即可完成初始化：

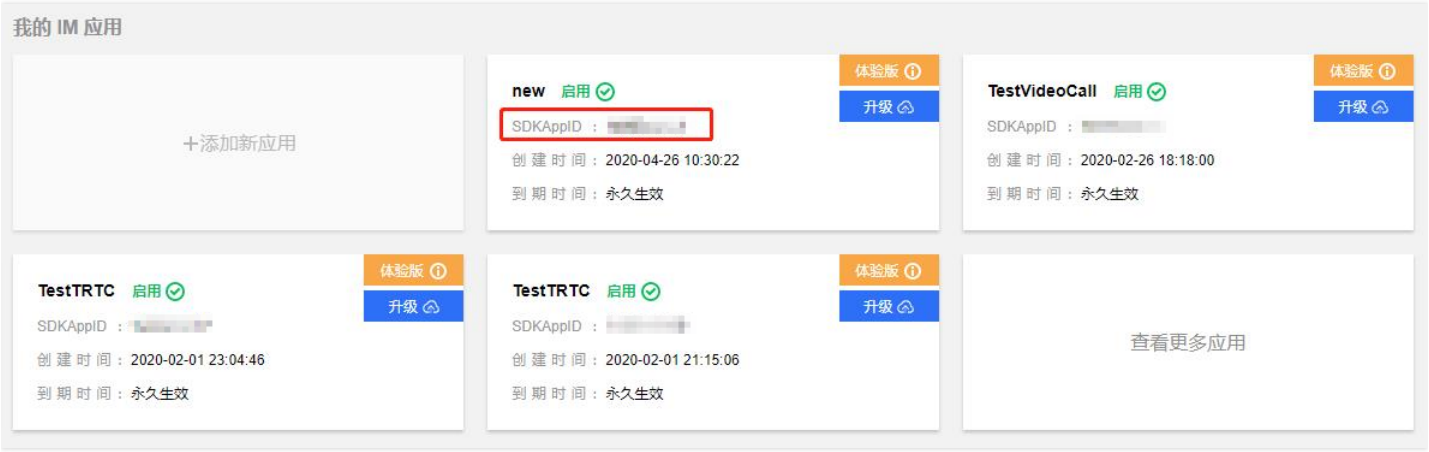
```
// 1. 从 IM 控制台获取应用 SDKAppID，详情请参考 SDKAppID。
// 2. 初始化 config 对象
V2TIMSDKConfig *config = [[V2TIMSDKConfig alloc] init];
// 3. 指定 log 输出级别，详情请参考 SDKConfig。
config.logLevel = V2TIM_LOG_INFO;
// 4. 初始化 SDK 并设置 V2TIMSDKListener 的监听对象。
// initSDK 后 SDK 会自动连接网络，网络连接状态可以在 V2TIMSDKListener 回调里面监听。
[[V2TIMManager sharedInstance] initSDK:1400000123 config:config listener:self];

// 5. 监听 V2TIMSDKListener 回调
- (void)onConnecting {
// 正在连接到腾讯云服务器
}
- (void)onConnectSuccess {
// 已经成功连接到腾讯云服务器
}
- (void)onConnectFailed:(int)code err:(NSString*)err {
// 连接腾讯云服务器失败
}
```

初始化接口 initSDK(SDKAppID, SDKConfig, listener) 包含三个必填的参数，分别是 SDKAppID，Config 和事件监听器。

SDKAppID

SDKAppID 即应用 ID，它是腾讯云 IM 服务用于区分客户账号的唯一标识。每一个独立的 App 都建议申请一个新的 SDKAppID，不同 SDKAppID 之间的消息天然隔离，不能互通。

您可以在 即时通信 IM 控制台 查看所有的 SDKAppID，单击【添加新应用】即可创建新的 SDKAppID。



SDKConfig

参数 V2TIMSDKConfig 用于对 SDK 进行初始化配置，其中较常用的是对日志级别的设置，即 logLevel 参数，日志级别如下表所示：

日志级别	LOG 输出量
V2TIM_LOG_NONE	不输出任何 log
V2TIM_LOG_DEBUG	输出 DEBUG，INFO，WARNING，ERROR 级别的 log
V2TIM_LOG_INFO	输出 INFO，WARNING，ERROR 级别的 log
V2TIM_LOG_WARN	输出 WARNING，ERROR 级别的 log
V2TIM_LOG_ERROR	输出 ERROR 级别的 log

- IM SDK 的日志默认存储于 /Library/Caches/ 目录下。

- 从 V 4.7.1 开始，IM SDK 的日志开始采用微信团队的 xlog 模块进行输出，xlog 日志默认是压缩的，需要使用 python 脚本进行解压。
  - 获取解压脚本：若使用 Python 2.7，则单击 Decode Log 27 获取解压脚本；若使用 Python 3.0，则单击 Decode Log 30 获取解压脚本。
  - 在 Windows 或者 Mac 控制台输入如下命令即可对 log 文件进行解压，解压后的文件以 xlog.log 结尾，可以直接使用文本编辑器打开。

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

事件监听器

V2TIMSDKListener 提供了网络状态以及用户信息变更的监听：

事件回调	事件描述	推荐操作
onConnecting	正在连接到腾讯云服务器	适合在 UI 上展示“正在连接”状态。
onConnectSuccess	已经成功连接到腾讯云服务器	-
onConnectFailed	连接腾讯云服务器失败	可以提示用户当前网络连接不可用。
onKickedOffline	当前用户被踢下线	此时可以 UI 提示用户“您已经在其他端登录了当前账号，是否重新登录？”
onUserSigExpired	登录票据已经过期	请使用新签发的 UserSig 进行登录。
onSelfInfoUpdated	当前用户的资料发生了更新	可以在 UI 上更新自己的头像和昵称。

注意：
若收到 `onUserSigExpired` 回调，说明您登录用的 UserSig 票据已经过期，请更新后重新登录。如果继续使用过期的 UserSig，会导致 SDK 登录死循环。

登录

调用 `V2TIMManager` 的 login(userID, userSig) 函数可以进行登录，只有在 SDK 登录成功后，才能使用 IM SDK 的各项能力。

- UserID：建议只包含大小写英文字母、数字、下划线和连词符集中类型的字符，长度最大不超过32字节。
- UserSig：IM SDK 登录票据，由您的业务服务器进行计算以保证安全，计算方法请参考 UserSig 后台 API。

登录时机

以下场景需调用登录：

- App 启动后首次使用 IM SDK 的能力时。
- IM SDK 抛出 `onUserSigExpired` 回调时，即登录票据已过期时，需要使用新的 UserSig 进行登录。
- IM SDK 抛出 `onKickOffline` 回调时，即当前用户被踢下线时，可以通过 UI 提示用户“您已经在其他端登录了当前账号，是否重新登录？” 如果用户选择“是”，就可以进行重新登录。

以下场景无需调用登录：

- 用户的网络断开并重新连接后，不需要调用 login 函数，SDK 会自动上线。
- 当一个登录过程在进行时，不需要进行重复登录。

多端登录

同样类型的两台手机不能同时登录一个帐号，例如两台苹果手机不能同时登录一个帐号。但是一台安卓手机和一台苹果手机会被认为是两端，可以同时登录。多端登录相关配置请参考 登录设置。

登出

登出比较简单，使用 logout() 函数即可。

初始化与登录（Web & 小程序）

最近更新时间：2020-07-10 14:23:51

创建 SDK 实例

Web 项目

```
import TIM from 'tim-js-sdk';
// 发送图片、文件等消息需要的 COS SDK
import COS from "cos-js-sdk-v5";

let options = {
SDKAppID: 0 // 接入时需要将0替换为您的即时通信 IM 应用的 SDKAppID
};
// 创建 SDK 实例，TIM.create() 方法对于同一个 SDKAppID 只会返回同一份实例
let tim = TIM.create(options); // SDK 实例通常用 tim 表示

// 设置 SDK 日志输出级别，详细分级请参见 setLogLevel 接口的说明
tim.setLogLevel(0); // 普通级别，日志量较多，接入时建议使用
// tim.setLogLevel(1); // release 级别，SDK 输出关键信息，生产环境时建议使用

// 注册 COS SDK 插件
tim.registerPlugin({'cos-js-sdk': COS});
```

小程序项目

```
import TIM from 'tim-wx-sdk';
// 发送图片、文件等消息需要的 COS SDK
import COS from "cos-wx-sdk-v5";


let options = {
SDKAppID: 0 // 接入时需要将0替换为您的即时通信 IM 应用的 SDKAppID
};
// 创建 SDK 实例，TIM.create() 方法对于同一个 SDKAppID 只会返回同一份实例
let tim = TIM.create(options); // SDK 实例通常用 tim 表示

// 设置 SDK 日志输出级别，详细分级请参见 setLogLevel 接口的说明
tim.setLogLevel(0); // 普通级别，日志量较多，接入时建议使用
// tim.setLogLevel(1); // release 级别，SDK 输出关键信息，生产环境时建议使用

// 注册 COS SDK 插件
tim.registerPlugin({'cos-wx-sdk': COS});
```

设置日志级别

```
// 设置 SDK 日志输出级别，详细分级请参见 setLogLevel 接口的说明
tim.setLogLevel(0);
```

事件绑定

```
// 监听事件，例如：
tim.on(TIM.EVENT.SDK_READY, function(event) {
// 收到离线消息和会话列表同步完毕通知，接入侧可以调用 sendMessage 等需要鉴权的接口
// event.name - TIM.EVENT.SDK_READY
});

tim.on(TIM.EVENT.MESSAGE_RECEIVED, function(event) {
```

```
// 收到推送的单聊、群聊、群提示、群系统通知的新消息，可通过遍历 event.data 获取消息列表数据并渲染到页面
// event.name - TIM.EVENT.MESSAGE_RECEIVED
// event.data - 存储 Message 对象的数组 - [Message]
});

tim.on(TIM.EVENT.MESSAGE_REVOKED, function(event) {
// 收到消息被撤回的通知
// event.name - TIM.EVENT.MESSAGE_REVOKED
// event.data - 存储 Message 对象的数组 - [Message] - 每个 Message 对象的 isRevoked 属性值为 true
});

tim.on(TIM.EVENT.MESSAGE_READ_BY_PEER, function(event) {
// SDK 收到对端已读消息的通知，即已读回执。使用前需要将 SDK 版本升级至 v2.7.0 或以上。仅支持单聊会话。
// event.name - TIM.EVENT.MESSAGE_READ_BY_PEER
// event.data - event.data - 存储 Message 对象的数组 - [Message] - 每个 Message 对象的 isPeerRead 属性值为 true
});

tim.on(TIM.EVENT.CONVERSATION_LIST_UPDATED, function(event) {
// 收到会话列表更新通知，可通过遍历 event.data 获取会话列表数据并渲染到页面
// event.name - TIM.EVENT.CONVERSATION_LIST_UPDATED
// event.data - 存储 Conversation 对象的数组 - [Conversation]
});

tim.on(TIM.EVENT.GROUP_LIST_UPDATED, function(event) {
// 收到群组列表更新通知，可通过遍历 event.data 获取群组列表数据并渲染到页面
// event.name - TIM.EVENT.GROUP_LIST_UPDATED
// event.data - 存储 Group 对象的数组 - [Group]
});

tim.on(TIM.EVENT.PROFILE_UPDATED, function(event) {
// 收到自己或好友的资料变更通知
// event.name - TIM.EVENT.PROFILE_UPDATED
// event.data - 存储 Profile 对象的数组 - [Profile]
});

tim.on(TIM.EVENT.BLACKLIST_UPDATED, function(event) {
// 收到黑名单列表更新通知
// event.name - TIM.EVENT.BLACKLIST_UPDATED
// event.data - 存储 userID 的数组 - [userID]
});

tim.on(TIM.EVENT.ERROR, function(event) {
// 收到 SDK 发生错误通知，可以获取错误码和错误信息
// event.name - TIM.EVENT.ERROR
// event.data.code - 错误码
// event.data.message - 错误信息
});

tim.on(TIM.EVENT.SDK_NOT_READY, function(event) {
// 收到 SDK 进入 not ready 状态通知，此时 SDK 无法正常工作
// event.name - TIM.EVENT.SDK_NOT_READY
});

tim.on(TIM.EVENT.KICKED_OUT, function(event) {
// 收到被踢下线通知
// event.name - TIM.EVENT.KICKED_OUT
// event.data.type - 被踢下线的原因，例如:
// - TIM.TYPES.KICKED_OUT_MULT_ACCOUNT 多实例登录被踢
// - TIM.TYPES.KICKED_OUT_MULT_DEVICE 多终端登录被踢
// - TIM.TYPES.KICKED_OUT_USERSIG_EXPIRED 签名过期被踢（v2.4.0起支持）。
});

tim.on(TIM.EVENT.NET_STATE_CHANGE, function(event) {
// 网络状态发生改变（v2.5.0 起支持）。
// event.name - TIM.EVENT.NET_STATE_CHANGE
// event.data.state 当前网络状态，枚举值及说明如下：
// \- TIM.TYPES.NET_STATE_CONNECTED - 已接入网络
// \- TIM.TYPES.NET_STATE_CONNECTING - 连接中。很可能遇到网络抖动，SDK 在重试。接入侧可根据此状态提示“当前网络不稳定”或“连接中”
```

```
// \- TIM.TYPES.NET_STATE_DISCONNECTED - 未接入网络。接入侧可根据此状态提示“当前网络不可用”。SDK 仍会继续重试，若用户网络恢复，SDK 会自动同步消息
});

// 开始登录
tim.login({userID: 'your userID', userSig: 'your userSig'});
```

参数 options 为 Object 类型：

Name	Type	Description
options	Object	应用配置

options 包含的属性值：

Name	Type	Description
SDKAppID	Number	即时通信 IM 应用的 SDKAppID

更详细的初始化流程和 API 使用介绍请参见 SDK 初始化。

登录

用户登录 IM SDK 才能正常收发消息，登录需要用户提供 UserID、UserSig 等信息，具体含义请参见 登录鉴权。登录成功后，需要先等 SDK 处于 ready 状态才能调用 sendMessage 等需要鉴权的接口，您可以通过监听事件 TIM.EVENT.SDK_READY 获取 SDK 状态。

> 注意：
> 默认情况下，不支持多实例登录，即如果此帐号已在其他页面登录，若继续在当前页面登录成功，有可能会将其他页面踢下线。用户被踢下线时会触发事件 TIM.EVENT.KICKED_OUT ，用户可在监听到事件后做相应处理。多端登录监听示例如下：

```
let onKickedOut = function (event) {
console.log(event.data.type); // mutipleAccount(同一设备，同一帐号，多页面登录被踢)
};
tim.on(TIM.EVENT.KICKED_OUT, onKickedOut);
```

如需支持多实例登录（允许在多个网页中同时登录同一帐号），请登录 即时通信 IM 控制台，找到相应 SDKAppID，选择【应用配置】>【功能配置】>【登录与消息】>【Web端实例同时在线】配置实例个数。配置将在50分钟内生效。

接口名

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

请求参数

名称	类型	描述
userID	String	用户 ID。
userSig	String	用户登录即时通信 IM 的密码，其本质是对 UserID 等信息加密后得到的密文。 具体生成方法请参见 生成 UserSig。

返回值

该接口返回 Promise 对象。

示例

```
let promise = tim.login({userID: 'your userID', userSig: 'your userSig'});
promise.then(function(imResponse) {
console.log(imResponse.data); // 登录成功
if (imResponse.data.repeatLogin === true) {
// 标识账号已登录，本次登录操作为重复登录。v2.5.1 起支持
```

```
console.log(imResponse.data.errorInfo);
}
}).catch(function(imError) {
console.warn('login error:', imError); // 登录失败的相关信息
});
```

登出

登出即时通信 IM，通常在切换帐号的时候调用，清除登录态以及内存中的所有数据。

注意：

- 调用此接口的实例会发布 SDK_NOT_READY 事件，此时该实例下线，无法收、发消息。
- 如果您在即时通信 IM 控制台配置的"Web端实例同时在线个数"大于 1，且同一账号登录了 a1 和 a2 两个实例（含小程序端），当执行 a1.logout() 后， a1 会下线，无法收、发消息。而 a2 实例不会受影响。
- 多实例被踢：基于第 2 点，如果"Web端实例同时在线个数"配置为 2，且您的某一账号已经登录了 a1 ， a2 两个实例，当使用此账号成功登录第三个实例 a3 时， a1 或 a2 中的一个实例会被踢下线（通常是最先处在登录态的实例会触发），这种情况称之为**"多实例被踢"**。假设 a1 实例被踢下线， a1 实例内部会执行登出流程，然后抛出 KICKED_OUT 事件，接入侧可以监听此事件，并在触发时跳转到登录页。此时 a1 实例下线，而 a2 、 a3 实例可以正常运行。

接口名

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

请求参数

无

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse， IMResponse.data 为空对象。表示成功登出。
- catch 的回调函数参数为 IMError。

示例

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

消息收发
消息收发（Android）

最近更新时间：2020-10-09 14:26:13

消息的分类

腾讯云 IM 消息按照消息的发送目标可以分为：“单聊消息”（又称 “C2C 消息”）和“群聊消息” 两种：

消息分类	API 关键词	说明
单聊消息	C2CMessage	又称 C2C 消息，在发送时需要指定消息接收者的 UserID，只有接受者可以收到该消息。
群聊消息	GroupMessage	在发送时需要指定目标群组的 groupID，该群中的所有用户均能收到消息。

按照消息承载的内容可以分为：“文本消息”、“自定义（信令）消息”，“图片消息”、“视频消息”、“语音消息”、“文件消息”、“位置消息”、“群 Tips 消息”等几种类型。

消息分类	API 关键词	说明
文本消息	TextElem	即普通的文字消息，该类消息会经过即时通信 IM 的敏感词过滤，发送包含的敏感词消息时会报80001错误码。
自定义消息	CustomElem	即一段二进制 buffer，通常用于传输您应用中的自定义信令，内容不会经过敏感词过滤。
图片消息	ImageElem	SDK 会在发送原始图片的同时，自动生成两种不同尺寸的缩略图，三张图分别被称为原图、大图、微缩图。
视频消息	VideoElem	一条视频消息包含一个视频文件和一张配套的缩略图。
语音消息	SoundElem	支持语音是否播放红点展示。
文件消息	FileElem	文件消息最大支持100MB。
位置消息	LocationElem	地理位置消息由位置描述、经度（longitude ）和纬度（latitude）三个字段组成。
群 Tips 消息	GroupTipsElem	群 Tips 消息常被用于承载群中的系统性通知消息，例如有成员进出群组，群的描述信息被修改，群成员的资料发生变化等。

收发简单消息

V2TIMManager 中提供了一组简单的消息收发接口，虽只能用于文本消息和自定义（信令）消息的收发，但使用方法特别简单，只需要几分钟即可完成对接。

发送文本和信令消息（简化接口）

调用 sendC2CTextMessage 或者 sendGroupTextMessage 可以发送文本消息，其中文本消息会经过即时通信 IM 的敏感词过滤，包含的敏感词消息在发送时会报80001错误码。调用 sendC2CCustomMessage 或者 sendGroupCustomMessage 可以发送 C2C 自定义（信令）消息，自定义消息本质是一段二进制 buffer，通常用于传输您应用中的自定义信令，内容不会经过敏感词过滤。

接收文本和信令消息（简化接口）

通过 addSimpleMsgListener 可以监听简单的文本和信令消息，复杂的图片、视频、语音消息则需要通过 V2TIMMessageManager 中定义的 addAdvancedMsgListener 实现。

> 注意：
> addSimpleMsgListener 与 addAdvancedMsgListener 请勿混用，以免产生逻辑 BUG。

经典示例：直播群中收发弹幕消息

直播场景下，在直播群中收发弹幕消息是非常普遍的交互方式，其实现方式非常简单，通过简单消息接口即可满足：

1. 主播调用 createGroup 创建一个直播群（AVChatRoom），并在“正在直播”的房间列表中记录群组 ID。
2. 观众选择自己喜欢的主播，并调用 joinGroup 加入该主播创建的直播群。
3. 消息的发送方可以通过 sendGroupTextMessage 群发弹幕文本消息。
4. 消息的接收方可以通过 addSimpleMsgListener 注册简单消息监听器，并通过监听回调函数 onRecvGroupTextMessage 获取文本消息。

为直播间增加“点赞飘心”的功能，“点赞飘心”属于一条指令，操作步骤如下：

1. 定义一个的自定义消息类型，例如一个 JSON 字符串：`{"command": "favor", "value": 101}`。
2. 通过 sendGroupCustomMessage 接口进行消息的发送，并通过 onRecvGroupCustomMessage 进行接收。

收发富媒体消息

图片、视频、语音、文件、地理位置等类型的消息称为“富媒体消息”。相比于简单消息，富媒体消息的收发相对复杂：

- 在发送时，富媒体消息需要先用对应的 `create` 函数创建一个 V2TIMMessage 对象，再调用对应的 `send` 接口发送。
- 在接收时，富媒体消息要先判断 `elemType`，并根据 `elemType` 获取对应的 `Elem` 进行二次解析。

发送富媒体消息

本文以图片消息为例，介绍发送一条富媒体消息的过程：

1. 发送方调用 createImageMessage 创建一条图片消息，拿到消息对象 V2TIMMessage。
2. 发送方调用 sendMessage 接口将刚才创建的消息对象发送出去。

接收富媒体消息

1. 接收方调用 addAdvancedMsgListener 接口设置高级消息监听。
2. 接收方通过监听回调 onRecvNewMessage 获取图片消息 V2TIMMessage。
3. 接收方解析 V2TIMMessage 消息中的 `elemType` 属性，并根据其类型进行二次解析，获取消息内部 Elem 中的具体内容。

经典示例：收发图片消息

发送方创建一条图片消息并发送：

```
// 创建图片消息
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createImageMessage("/sdcard/test.png");
// 发送图片消息
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "toUserID", null, V2TIMMessage.V2TIM_PRIORITY_DEFAULT, false, null, new V2TIMSe
ndCallback<V2TIMMessage>() {
@Override
public void onError(int code, String desc) {
// 图片消息发送失败
}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {
// 图片消息发送成功
}
@Override
public void onProgress(int progress) {
// 图片上传进度（0-100）
}
});
```

接收方识别一条图片消息并将解析中包含的原图、大图和微缩图：

```
@Override
public void onRecvNewMessage(V2TIMMessage msg) {
int elemType = msg.getElemType();
if (elemType == V2TIMMessage.V2TIM_ELEM_TYPE_IMAGE) {
V2TIMImageElem v2TIMImageElem = msg.getImageElem();
// 一个图片消息会包含三种格式大小的图片，分别为原图、大图、微缩图(SDK内部自动生成大图和微缩图)
// 大图：是将原图等比压缩，压缩后宽、高中较小的一个等于720像素。
// 缩略图：是将原图等比压缩，压缩后宽、高中较小的一个等于198像素。
List<V2TIMImageElem.V2TIMImage> imageList = v2TIMImageElem.getImageList();
for (V2TIMImageElem.V2TIMImage v2TIMImage : imageList) {
String uuid = v2TIMImage.getUUID(); // 图片 ID
int imageType = v2TIMImage.getType(); // 图片类型
int size = v2TIMImage.getSize(); // 图片大小（字节）
int width = v2TIMImage.getWidth(); // 图片宽度
int height = v2TIMImage.getHeight(); // 图片高度
// 设置图片下载路径 imagePath，这里可以用 uuid 作为标识，避免重复下载
```

```
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) {
// 图片下载进度：已下载大小 v2ProgressInfo.getCurrentSize()；总文件大小 v2ProgressInfo.getTotalSize()
}
@Override
public void onError(int code, String desc) {
// 图片下载失败
}
@Override
public void onSuccess() {
// 图片下载完成
}
});
} else {
// 图片已存在
}
}
}
}
```

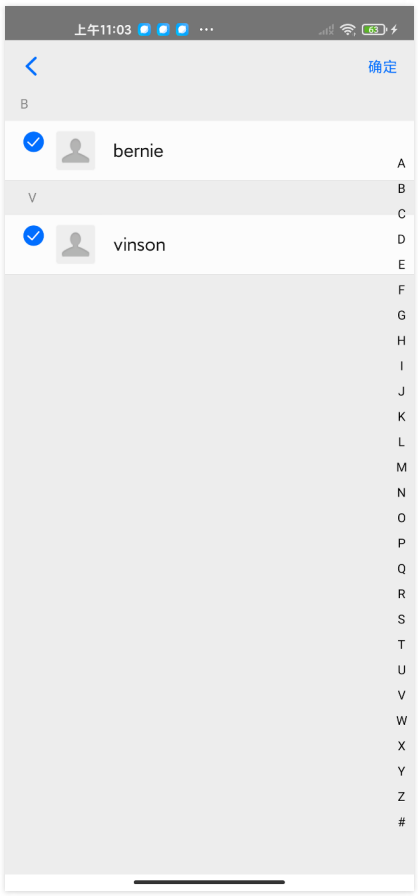
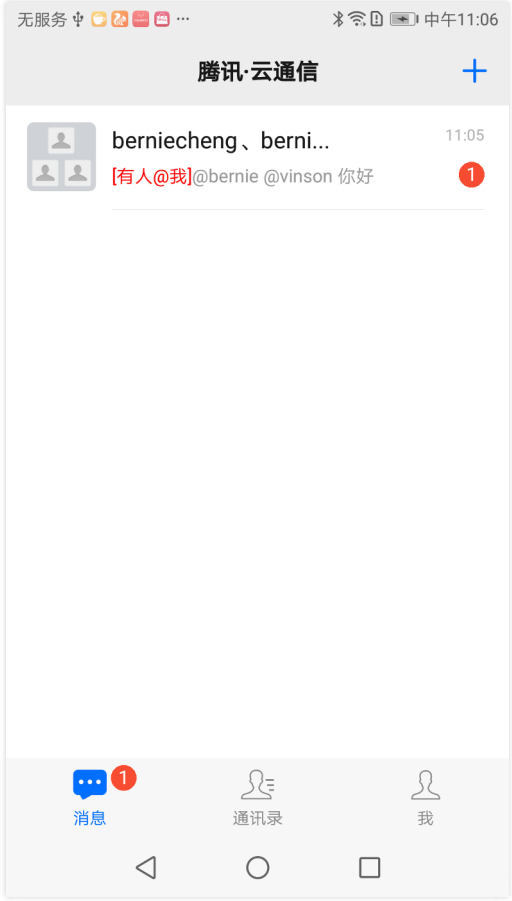
说明：
更多消息解析示例代码请参考 常见问题 > 5. 各类型消息应该如何解析。

收发群 @ 消息

群 @ 消息，发送方可以在输入栏监听 @ 字符输入，调用到群成员选择界面，选择完成后以 “@A @B @C.....” 形式显示在输入框，并可以继续编辑消息内容，完成消息发送。接收方会在会话界面的群聊天列表，重点显示 “有人@我” 或者 “@所有人” 标识，提醒用户有人在群里 @ 自己了。

说明：
目前仅支持文本 @ 消息。

监听 @ 字符选择群成员	编辑群 @ 消息发送	收到群 @ 消息

监听 @ 字符选择群成员	编辑群 @ 消息发送	收到群 @ 消息
		

发送群 @ 消息

1. 发送方监听聊天界面的文本输入框，启动群成员选择界面，选择完成后回传选择群成员的 ID 和昵称信息，ID 用来构建消息对象 V2TIMMessage，昵称用来在文本框显示。
2. 发送方调用 V2TIMMessageManager 的 createTextAtMessage 创建一条 @ 文本消息，拿到消息对象 V2TIMMessage。
3. 发送方调用 sendMessage 接口将刚才创建的 @ 消息对象发送出去。

接收群 @ 消息

1. 在加载和更新会话处，需要调用 V2TIMConversation 的 getGroupAtInfoList接口获取会话的 @ 数据列表 List < V2TIMGroupAtInfo >。
2. 通过列表中 V2TIMGroupAtInfo 对象的 getAtType 接口获取 @ 数据类型，并更新到当前会话的 @ 信息。

经典示例：收发群 @ 消息

- **发送群 @ 消息**：
  发送方创建一条群 @ 消息并发送：

```
// 获取群成员ID数据
List<String> atUserList = updateAtUserList(mTextInput.getMentionList(true));
// 创建群@消息
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createTextAtMessage(message, atUserList);
// 发送群@消息
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, null, "toGroupID", V2TIMMessage.V2TIM_PRIORITY_DEFAULT, false, null, new V2TI
MSendCallback<V2TIMMessage>() {
@Override
public void onError(int code, String desc) {
// 群@消息发送失败
}
@Override
```

```
public void onSuccess(V2TIMMessage v2TIMMessage) {
// 群@消息发送成功
}
@Override
public void onProgress(int progress) {

}
});
```

- **接收群 @ 消息**：
  加载和更新会话处，获取群 @ 数据列表：

```
boolean atMe = false;
boolean atAll = false;
//获取群@数据列表
List<V2TIMGroupAtInfo> atInfoList = conversation.getGroupAtInfoList();
if (atInfoList == null || atInfoList.isEmpty()){
return V2TIMGroupAtInfo.TIM_AT_UNKNOWN;
}
//获取@数据类型
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;
}
//更新@类型到当前会话
switch (atInfoType){
case V2TIMGroupAtInfo.TIM_AT_ME:
Log.d(TAG, "更新到当前会话显示[有人@我]");
break;
case V2TIMGroupAtInfo.TIM_AT_ALL:
Log.d(TAG, "更新到当前会话显示[@所有人]");
break;
case V2TIMGroupAtInfo.TIM_AT_ALL_AT_ME:
Log.d(TAG, "更新到当前会话显示[有人@我][@所有人]");
break;
default:
break;

}
```

设置离线推送（offlinePushInfo）

当接收方的 App 被 kill 时，IM SDK 无法通过正常的网络连接收取新消息。如需实现在此场景下接收方仍能感知到新消息，需要使用各个手机厂商提供的离线推送服务，更多详细请参见 Android 离线推送。

设置离线推送的标题和内容

您可以在发送消息时，通过 sendMessage 接口中的 **offlinePushInfo** 字段，设置离线推送的标题和内容。

```
// 创建一条文本消息发送给 groupA，并且自定义推送 Title、推送内容
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createTextMessage(content);
V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
// 设置通知栏标题
v2TIMOfflinePushInfo.setTitle("offline_title");
// 设置通知栏内容
v2TIMOfflinePushInfo.setDesc("offline_desc");
// 发送消息
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, null, "groupA", V2TIMMessage.V2TIM_PRIORITY_DEFAULT, false, v2TIMOfflinePushInfo, new V2TIMSendCallback<V2TIMMessage>() {
@Override
public void onError(int code, String desc) {
// 消息发送失败
}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {
// 消息发送成功
}
@Override
public void onProgress(int progress) {
}
});
```

点击推送消息跳转到对应的聊天窗口

如需实现该功能，发送消息时需设置离线推送对象 `offlinePushInfo` 的扩展字段 `ext`，收到消息的用户打开 App 时可以通过不同厂商提供的获取自定义内容的方式拿到这个扩展字段 `ext`，然后根据 `ext` 内容跳转到对应的聊天界面。

本文以 “denny 给 vinson 发送消息” 的场景为例。
发送方：denny 要在发送消息的时候设置推送扩展字段 ext：

```
// denny 在发送消息时设置 offlinePushInfo，并指定 ext 字段
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_PRIORITY_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) {}
});
```

接收方：vinson 的 App 虽然不在线，但可以接收到手机厂商（我们以 OPPO 手机为例）的离线推送，当 vinson 点击推送消息时会启动 App：

```
// vinson 启动 App 后在打开的 Activity 中获取自定义内容
Bundle bundle = intent.getExtras();
Set<String> set = bundle.keySet();
if (set != null) {
for (String key : set) {
// 其中 key 和 value 分别为发送端设置的 extKey 和 ext content
String value = bundle.getString(key);
if (value.equals("jump to denny")) {
// 跳转到和 denny 的聊天界面
...
}
```

```
}
}
```

设置消息为只能在线接收（onlineUserOnly）

某些场景下，您可能希望发出去的消息只被在线用户接收，即当接收者不在线时就不会感知到该消息。您只需在 sendMessage 时，将参数 `onlineUserOnly` 设置为 `true`，此时发送出去的消息跟普通消息相比，会有如下差异点：

- 不支持离线存储，即如果接收方不在线就无法收到。
- 不支持多端漫游，即如果接收方在一台终端设备上一旦接收过该消息，无论是否已读，都不会在另一台终端上再次收到。
- 不支持本地存储，即本地的云端的历史消息中均无法找回。

经典示例：实现“对方正在输入”功能

在 C2C 单聊场景下，您可以通过 sendMessage 接口发送 "自己正在输入" 的提示性消息，接收方收到该消息时可以在 UI 界面展示 "对方正在输入"，示例代码如下：

```
// 给 userA 发送 "正在输入" 的提示消息
JSONObject jsonObject = new JSONObject();
try {
jsonObject.put("command", "textInput");
} catch (JSONException e) {
e.printStackTrace();
}
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createCustomMessage(jsonObject.toString().getBytes());
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "userA", null, V2TIMMessage.V2TIM_PRIORITY_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) {}
});
```

撤回消息

发送方通过 revokeMessage 接口可以撤回一条已经发送成功的消息。默认情况下，发送者只能撤回2分钟以内的消息，您可以按需更改消息撤回时间限制，具体操作请参见消息撤回设置。

消息的撤回同时需要接收方 UI 代码的配合：当发送方撤回一条消息后，接收方会收到消息撤回通知 onRecvMessageRevoked，通知中包含了撤回消息的 `msgID`，您可以根据这个 `msgID` 判断 UI 层是哪一条消息撤回了，然后把对应的消息气泡切换成 "消息已被撤回" 状态。

发送方撤回一条消息

```
V2TIMManager.getMessageManager().revokeMessage(v2TIMMessage, new V2TIMCallback() {
@Override
public void onError(int code, String desc) {
// 撤回消息失败
}
@Override
public void onSuccess() {
// 撤回消息成功
}
});
```

接收方感知消息被撤回

1. 调用 addAdvancedMsgListener 设置高级消息监听。
2. 通过 onRecvMessageRevoked 接收消息撤回通知。

```
@Override
public void onRecvMessageRevoked(String msgID) {
```

```
// msgList 为当前聊天界面的消息列表
for (V2TIMMessage msg : msgList) {
if (msg.getMsgID().equals(msgID)) {
// msg 即为被撤回的消息，您需要修改 UI 上相应的消息气泡的状态
}
}
}
```

给消息增加已读回执

在 C2C 单聊场景下，当接收方通过 markC2CMessageAsRead 接口将来自某人的消息标记为已读时，消息的发送方将会收到“已读回执”，表示“xxx 已经读过我的消息了”。

> 注意：
> 目前仅 C2C 单聊消息支持已读回执，群聊场景暂不支持。虽然群聊消息也有对应的 markGroupMessageAsRead 接口，但群消息的发送者目前无法收到已读回执。

接收方标记消息已读

```
//将来自 haven 的消息均标记为已读
V2TIMManager.getMessageManager().markC2CMessageAsRead("haven", new V2TIMCallback() {
@Override
public void onError(int code, String desc) {
// 设置消息已读失败
}
@Override
public void onSuccess() {
// 设置消息已读成功
}
});
```

发送方感知消息已读

消息已读回执的事件通知位于高级消息监听器 V2TIMAdvancedMsgListener 中，如需支持感知消息已读，需要先通过 addAdvancedMsgListener 设置监听器，然后通过 onRecvC2CReadReceipt 回调即可感知接收方的已读确认。

```
@Override
public void onRecvC2CReadReceipt(List<V2TIMMessageReceipt> receiptList) {
// 由于接收方一次性可能会收到多个已读回执，所以这里采用了数组的回调形式
for (V2TIMMessageReceipt v2TIMMessageReceipt : receiptList) {
// 消息接收者 receiver
String userID = v2TIMMessageReceipt.getUserID();
// 已读回执时间，聊天窗口中时间戳小于或等于 timestamp 的消息都可以被认为已读
long timestamp = v2TIMMessageReceipt.getTimest
}
}
```

查看历史消息

您可以调用 getC2CHistoryMessageList 获取单聊历史消息，调用 getGroupHistoryMessageList 获取群聊历史消息。如果当前设备网络连接正常，SDK 会默认从服务器拉取历史消息；如果没有网络连接，SDK 会直接从本地数据库中读取历史消息。

分页拉取历史消息

SDK 支持分页拉取历史消息，一次分页拉取的消息数量不宜太大，否则会影响拉取速度，建议一次最多拉取20条。
本文以分页拉取名为 `groupA` 的群的历史消息，每次分页拉取20条为例，示例代码如下：

```
// 第一次拉取 lastMsg 传 null，表示从最新的消息开始拉取 20 条消息
V2TIMManager.getMessageManager().getGroupHistoryMessageList("groupA", 20, null, new V2TIMValueCallback<List<V2TIMMessage>>() {
@Override
public void onError(int code, String desc) {
// 拉取失败
```

```
}
@Override
public void onSuccess(List<V2TIMMessage> v2TIMMessages) {
// 分页拉取返回的消息默认是按照从新到旧排列
if (v2TIMMessages.size() > 0) {
// 获取下一次分页拉取的起始消息
V2TIMMessage lastMsg = v2TIMMessages.get(v2TIMMessages.size() - 1);
// 拉取剩下的20条消息
V2TIMManager.getMessageManager().getGroupHistoryMessageList("groupA", 20, lastMsg, new V2TIMValueCallback<List<V2TIMMessage>>() {
@Override
public void onError(int code, String desc) {
// 拉取消息失败
}

@Override
public void onSuccess(List<V2TIMMessage> v2TIMMessages) {
// 拉取消息结束
}
});
}
}
});
```

现实场景中的分页拉取，通常由用户的滑动操作触发的，用户每下拉一次消息列表就触发一次分页拉取。但原理上跟上述示例代码类似，都是以 `lastMsg` 作为分页的标记，以 `count` 控制每次拉取的消息条数。

历史消息的注意事项

- 历史消息存储时长如下：
  - 体验版：免费存储7天，不支持延长
  - 专业版：免费存储7天，支持延长
  - 旗舰版：免费存储30天，支持延长

  延长历史消息存储时长是增值服务，您可以登录 即时通信 IM 控制台 修改相关配置，具体计费说明请参见 增值服务资费。
- 只有会议群（Meeting）（对应老版本的 ChatRoom 群）才支持拉取到用户**入群之前**的历史消息。
- 直播群（AVChatRoom）中的消息均不支持本地存储和多终端漫游，因此对直播群调用 getGroupHistoryMessageList 接口是无效的。

删除消息

对于历史消息，您可以调用 deleteMessages 接口删除历史消息，消息删除后，无法再恢复。

设置消息权限

只允许好友间收发消息

SDK 默认不限制非好友之间收发消息。如果您希望仅允许好友之间发送单聊消息，您可以在 即时通信 IM 控制台 >【功能配置】>【登录与消息】>【好友关系检查】中开启"发送单聊消息检查关系链"。开启后，用户只能给好友发送消息，当用户给非好友发消息时，SDK 会报20009错误码。

屏蔽某人的消息

如果您想屏蔽某人的消息，请调用 addToBlackList 接口把该用户加入黑名单，即拉黑该用户。
当消息发送者被拉黑后，发送者默认不会感知到"被拉黑"的状态，即发送消息后仍展示发送成功（实际上此时接收方不会收到消息）。如果需要被拉黑的发送者收到消息发送失败的提示，请在 即时通信 IM 控制台 >【功能配置】>【登录与消息】>【黑名单检查】中关闭"发送消息后展示发送成功"，关闭后，被拉黑的发送者在发送消息时，SDK 会报20007错误码。

屏蔽某个群组的消息

如果您想屏蔽某个群组的消息，请调用 setReceiveMessageOpt 接口，设置群消息接收选项为 `V2TIM_GROUP_NOT_RECEIVE_MESSAGE` 状态。

敏感词过滤

SDK 发送的文本消息默认会经过即时通信 IM 的敏感词过滤，如果发送者在发送的文本消息中包含敏感词，SDK 会报 80001 错误码。

常见问题

1. 为什么会收到重复的消息？

请检查以下逻辑是否正确：

- 请检查 addSimpleMsgListener 与 addAdvancedMsgListener 是否混用。如果混用，当收到文本消息或自定义消息时，两个监听都会回调，会导致收到重复消息。
- 请检查同一个监听对象是否重复 `add` ，如果监听对象不再使用，请主动调用对应的 removeSimpleMsgListener 或 removeAdvancedMsgListener 接口移除多余的监听器。

2. App 卸载重装后已读回执为什么失效了？

在单聊场景下，接收方如果调用 markC2CMessageAsRead 设置消息已读，发送方收到的已读回执里面包含了对方已读的时间戳 `timestamp` ，SDK 内部会根据 `timestamp` 判断消息对方是否已读， `timestamp` 目前只在本地保存，程序卸载重装后会丢失。

3. 有多个 Elem 的消息应该如何解析？

出于降低消息复杂度的考虑，SDK API2.0 接口不再支持创建包含多个 Elem 的 Message 对象。如果您收到了来自老版本的包含多个 Elem 的 Message 对象，可以按照以下步骤解析：

1. 正常解析出第一个 `Elem` 对象。
2. 通过第一个 `Elem` 对象的 getNextElem 方法获取下一个 `Elem` 对象。如果下一个 `Elem` 对象存在，会返回 `Elem` 对象实例，如果不存在，会返回 `null` 。

```
@Override
public void onRecvNewMessage(V2TIMMessage msg) {
// 查看第一个 Elem
int elemType = msg.getElemType();
if (elemType == V2TIMMessage.V2TIM_ELEM_TYPE_TEXT) {
// 文本消息
V2TIMTextElem v2TIMTextElem = msg.getTextElem();
String text = v2TIMTextElem.getText();
// 查看 v2TIMTextElem 后面还有没有更多 elem
V2TIMElem elem = v2TIMTextElem.getNextElem();
while (elem != null) {
// 判断 elem 类型，以 V2TIMCustomElem 为例
if (elem instanceof V2TIMCustomElem) {
V2TIMCustomElem customElem = (V2TIMCustomElem) elem;
byte[] data = customElem.getData();
}
// 继续查看当前 elem 后面还有没更多 elem
elem = elem.getNextElem();
}
// elem 如果为 null，表示所有 elem 都已经解析完
```

```
}
}
```

4. 各种不同类型的消息应该如何解析？

解析消息相对复杂，我们提供了各种类型消息解析的 示例代码，您可以直接把相关代码拷贝到您的工程，然后根据实际需求进行二次开发。

消息收发（iOS）

最近更新时间：2020-09-29 16:55:06

消息的分类

腾讯云 IM 消息按照消息的发送目标可以分为：“单聊消息”（又称“C2C 消息”）和“群聊消息”两种：

消息分类	API 关键词	详细解释
单聊消息	C2CMessage	又称 C2C 消息，在发送时需要指定消息接收者的 UserID，只有接受者可以收到该消息。
群聊消息	GroupMessage	在发送时需要指定目标群组的 groupID，该群中的所有用户均能收到消息。

按照消息承载的内容可以分为：“文本消息”、“自定义（信令）消息”，“图片消息”、“视频消息”、“语音消息”、“文件消息”、“位置消息”、“群 Tips 消息”等几种类型。

消息分类	API 关键词	详细解释
文本消息	TextElem	即普通的文字消息，该类消息会经过腾讯云 IM 的敏感词过滤，包含的敏感词消息在发送时会报80001错误码。
自定义消息	CustomElem	即一段二进制 buffer，通常用于传输您应用中的自定义信令，内容不会经过敏感词过滤。
图片消息	ImageElem	SDK 会在发送原始图片的同时，自动生成两种不同尺寸的缩略图，三张图分别被称为原图、大图、微缩图。
视频消息	VideoElem	一条视频消息包含一个视频文件和一张配套的缩略图。
语音消息	SoundElem	支持语音是否播放红点展示。
文件消息	FileElem	文件消息最大支持100MB。
位置消息	LocationElem	地理位置消息由位置描述、经度（longitude ）和维度（latitude）三个字段组成。
群 Tips 消息	GroupTipsElem	群 Tips 消息常被用于承载群中的系统性通知消息，例如有成员进出群组，群的描述信息被修改，群成员的资料发生变化等。

收发简单消息

V2TIMManager.h 中提供了一组简单的消息收发接口，虽只能用于文本消息和自定义（信令）消息的收发，但使用方法特别简单，只需要几分钟即可完成对接。

发送文本和信令消息（简化接口）

调用 sendC2CTextMessage 或者 sendGroupTextMessage 可以发送文本消息，其中文本消息会经过即时通信 IM 的敏感词过滤，包含的敏感词消息在发送时会报80001错误码。
调用 sendC2CCustomMessage 或者 sendGroupCustomMessage 可以发送 C2C 自定义（信令）消息，自定义消息本质是一段二进制 buffer，通常用于传输您应用中的自定义信令，内容不会经过敏感词过滤。

接收文本和信令消息（简化接口）

通过 addSimpleMsgListener 可以监听简单的文本和信令消息，复杂的图片、视频、语音消息则需要通过 V2TIMManager + Message.h 中定义的 addAdvancedMsgListener 实现。

> 注意：
> addSimpleMsgListener 与 addAdvancedMsgListener 请勿混用，以免产生逻辑 BUG。

经典示例：直播群中收发弹幕消息

直播场景下，在直播群中收发弹幕消息是非常普遍的交互方式，其实现方式非常简单，通过简单消息接口即可满足：

1. 主播调用 createGroup 创建一个直播群（AVChatRoom），并在“正在直播”的房间列表中记录群组 ID。
2. 观众选择自己喜欢的主播，并调用 joinGroup 加入该主播创建的直播群。
3. 消息的发送方可以通过 sendGroupTextMessage 群发弹幕文本消息。
4. 消息的接收方可以通过 addSimpleMsgListener 注册简单消息监听器，并通过监听回调函数 onRecvGroupTextMessage 获取文本消息。

为直播间增加“点赞飘心”的功能，“点赞飘心”属于一条指令，操作步骤如下：

1. 定义一个的自定义消息类型，例如一个 JSON 字符串：`{"command": "favor", "value": 101}`。
2. 通过 sendGroupCustomMessage 接口进行消息的发送，并通过 onRecvGroupCustomMessage 进行接收。

收发富媒体消息

图片、视频、语音、文件、地理位置等类型的消息称为“富媒体消息”。相比于简单消息，富媒体消息的收发相对复杂：

- 在发送时，富媒体消息需要先用对应的 `create` 函数创建一个 V2TIMMessage 对象，再调用对应的 `send` 接口发送。
- 在接收时，富媒体消息需要先判断 `elemType` ，并根据 `elemType` 获取对应的 `Elem` 进行二次解析。

发送富媒体消息

本文以图片消息为例，介绍发送一条富媒体消息的过程：

1. 发送方调用 createImageMessage 创建一条图片消息，获取消息对象 V2TIMMessage。
2. 发送方调用 sendMessage 接口将刚才创建的消息对象发送出去。

接收富媒体消息

1. 接收方调用 addAdvancedMsgListener 接口设置高级消息监听。
2. 接收方通过监听回调 onRecvNewMessage 获取图片消息 V2TIMMessage。
3. 接收方解析 V2TIMMessage 消息中的 `elemType` 属性，并根据其类型进行二次解析，获取消息内部 Elem 中的具体内容。

经典示例：收发图片

发送方创建一条图片消息并发送：

```
// 获取本地图片路径
NSString *imagePath = [[NSBundle mainBundle] pathForResource:@"test" ofType:@"png"];
// 创建图片消息
V2TIMMessage *msg = [[V2TIMManager sharedInstance] createImageMessage:imagePath];
// 发送图片消息
[[V2TIMManager sharedInstance] sendMessage:msg receiver:@"userA" groupID:nil
priority:V2TIM_PRIORITY_DEFAULT
onlineUserOnly:NO offlinePushInfo:nil progress:^(uint32_t progress) {
// 图片上传进度（0 - 100）
} succ:^{
// 图片消息发送成功
} fail:^(int code, NSString *msg) {
// 图片消息发送失败
}];
```

接收方识别一条图片消息并将解析中包含的原图、大图和微缩图：

```
- (void)onRecvNewMessage:(V2TIMMessage *)msg {
if (msg.elemType == V2TIM_ELEM_TYPE_IMAGE) {
V2TIMImageElem *imageElem = msg.imageElem;
// 一个图片消息会包含三种格式大小的图片，分别为原图、大图、微缩图(SDK内部自动生成大图和微缩图)
// 大图：是将原图等比压缩，压缩后宽、高中较小的一个等于720像素。
// 缩略图：是将原图等比压缩，压缩后宽、高中较小的一个等于198像素。
NSArray<V2TIMImage *> *imageList = imageElem.imageList;
for (V2TIMImage *timImage in imageList) {
NSString *uuid = timImage.uuid; // 图片 ID
V2TIMImageType type = timImage.type; // 图片类型
int size = timImage.size; // 图片大小（字节）
int width = timImage.width; // 图片宽度
int height = timImage.height; // 图片高度
// 设置图片下载路径 imagePath，这里可以用 uuid 作为标识，避免重复下载
NSString *imagePath = [NSTemporaryDirectory() stringByAppendingPathComponent:
[NSString stringWithFormat: @"testImage%@",timImage.uuid]];
if (![[NSFileManager defaultManager] fileExistsAtPath:imagePath]) {
[timImage downloadImage:imagePath
progress:^(NSInteger curSize, NSInteger totalSize) {
NSLog(@"图片下载进度：curSize：%lu,totalSize:%lu",curSize,totalSize);
} succ:^{
NSLog(@"图片下载完成");
} fail:^(int code, NSString *msg) {
NSLog(@"图片下载失败：code：%d,msg:%@",code,msg);
}];
} else {
```

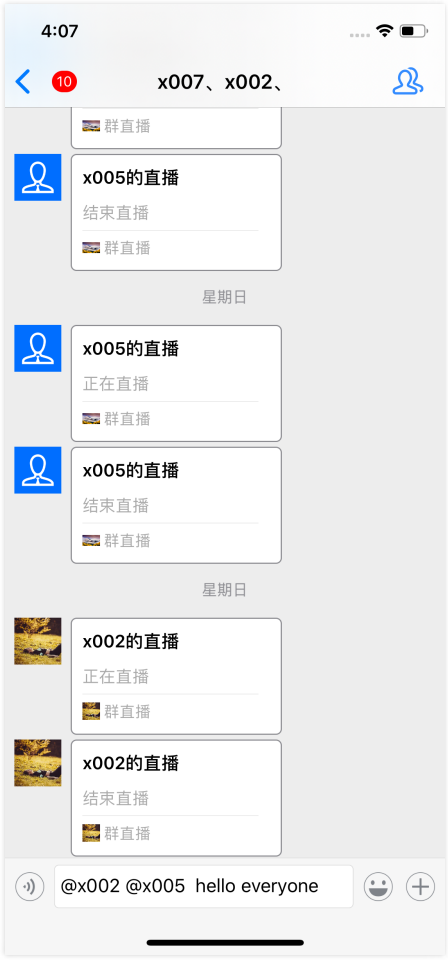
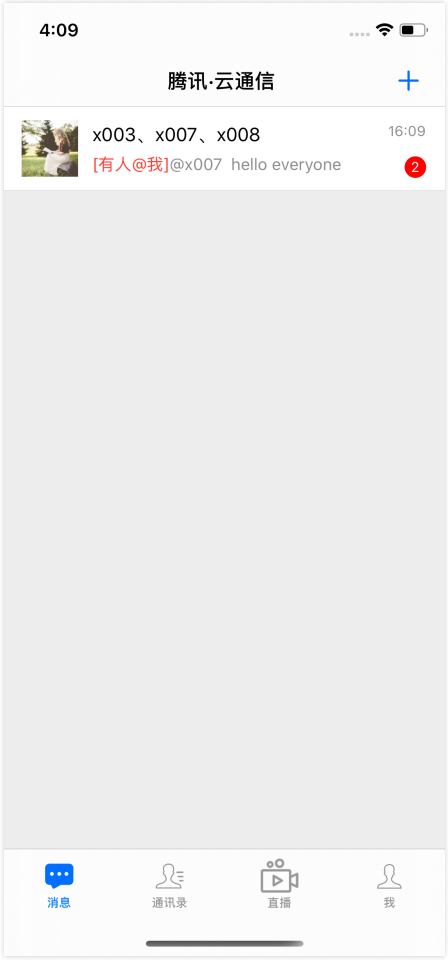
```
// 图片已存在
}
}
}
}
```

说明：
更多消息解析示例代码请参考 常见问题 > 5. 各类型消息应该如何解析。

收发群 @ 消息

群 @ 消息，发送方可以在输入栏监听 @ 字符输入，调用到群成员选择界面，选择完成后以 “@A @B @C.....” 形式显示在输入框，并可以继续编辑消息内容，完成消息发送；接收方会在会话界面的群聊天列表，重点显示 “有人@我” 或者 “@所有人” 标识，提醒用户有人在群里 @ 自己了。

说明：
目前仅支持文本 @ 消息。

监听 @ 字符选择群成员	编辑群 @ 消息发送	收到群 @ 消息
		

发送群 @ 消息

1. 发送方监听聊天界面的文本输入框，启动群成员选择界面，选择完成后回传选择群成员的 ID 和昵称信息，ID 用来构建消息对象 V2TIMMessage，昵称用来在文本框显示。

2. 发送方调用 V2TIMManager+Message 的 createTextAtMessage 创建一条 @ 文本消息，拿到消息对象 V2TIMMessage。
3. 发送方调用 sendMessage 接口将刚才创建的 @ 消息对象发送出去。

接收群 @ 消息

1. 在加载和更新会话处，需要调用 V2TIMConversation 的 groupAtInfolist 接口获取会话的 @ 数据列表 。
2. 通过列表中 V2TIMGroupAtInfo 对象的 atType 接口获取 @ 数据类型，并更新到当前会话的 @ 信息。

经典示例：收发群 @ 消息

- **发送群 @ 消息：**
  发送方创建一条群 @ 消息并发送。

```
// 获取@群成员的ID数据
TUITextMessageCellData *text = (TUITextMessageCellData *)data;
NSMutableArray<NSString *> *atUserList = text.atUserList;

// 创建群@消息
V2TIMMessage *atMsg = [[V2TIMManager sharedInstance] createTextAtMessage:text.content atUserList:atUserList];

// 发送群@消息
[[V2TIMManager sharedInstance] sendMessage:atMsg
receiver:nil
groupID:@"toGroupId"
priority:V2TIM_PRIORITY_DEFAULT
onlineUserOnly:NO
offlinePushInfo:nil
progress:nil
succ:^{
NSLog(@"群@消息发送成功");
}
fail:^(int code, NSString *desc) {
NSLog(@"群@消息发送失败");
}];
```

- **接收群 @ 消息：**
  在加载和更新会话处，获取群 @ 数据列表，解析当前的 @ 类型，根据 @ 类型显示对应的提示文本。

```
// 获取群@数据列表
NSArray<V2TIMGroupAtInfo *> *atInfoList = conversation.groupAtInfolist;

// 解析@类型（@我，@所有人，@我且@所有人）
BOOL atMe = NO; // 是否@我
BOOL atAll = NO; // 是否@所有人
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;
}
}

// 根据@类型，提示
if (atMe && !atAll) {
atTipsStr = @"[有人@我]";
```

```
}
if (!atMe && atAll) {
atTipsStr = @"[@所有人]";
}
if (atMe && atAll) {
atTipsStr = @"[有人@我][@所有人]";
}
```

设置 APNS 离线推送（offlinePushInfo）

当接收方的 App 被 kill 或者被切到后台时，IM SDK 无法通过正常的网络连接收取新消息。如需实现在此场景下接收方仍能感知到新消息，需要使用苹果提供的 APNs 服务，即“苹果离线推送”，更多详细请参见 开启 iOS 离线推送。

设置 APNS 离线推送的标题和声音

您可以在发送消息时，通过 sendMessage 接口中的 **offlinePushInfo** 字段，设置 APNS 离线推送的标题和声音。

```
// 创建一条图片消息发送给 groupA，并且自定义推送 Title、推送声音
NSString *imagePath = [[NSBundle mainBundle] pathForResource:@"test" ofType:@"png"];
// 创建图片消息
V2TIMMessage *msg = [[V2TIMManager sharedInstance] createImageMessage:imagePath];
V2TIMOfflinePushInfo *pushInfo = [[V2TIMOfflinePushInfo alloc] init];
// 自定义推送的标题和声音（01.caf 是一个示例文件，需要链接进 Xcode 工程，这里只需要填写带后缀的文件名）
pushInfo.title = @"自定义推送 Title 展示";
pushInfo.iOSSound = @"01.caf";
[[V2TIMManager sharedInstance] sendMessage:msg receiver:nil groupID:@"groupA" priority:V2TIM_PRIORITY_DEFAULT
onlineUserOnly:NO offlinePushInfo:pushInfo progress:^(uint32_t progress) {
} succ:^{
// 消息发送成功
} fail:^(int code, NSString *msg) {
// 消息发送失败
}];
```

点击推送消息跳转到对应的聊天窗口

如需实现该功能，发送消息时需设置离线推送对象 `offlinePushInfo` 的扩展字段 `ext`，收到消息的用户打开 App 时可以通过 `didReceiveRemoteNotification` 系统回调获取到扩展字段 `ext`，再根据 `ext` 内容跳转到对应的聊天界面。

本文以 “denny 给 vinson 发送消息” 的场景为例。

- 发送方：denny 需在发送消息时设置推送扩展字段 `ext`：

```
// denny在发送消息时设置 offlinePushInfo，并指定 ext 字段
V2TIMMessage *msg = [[V2TIMManager sharedInstance] createTextMessage:@"文本消息"];
V2TIMOfflinePushInfo *info = [[V2TIMOfflinePushInfo alloc] init];
info.ext = @"jump to denny";
[[V2TIMManager sharedInstance] sendMessage:msg receiver:@"vinson" groupID:nil priority:V2TIM_PRIORITY_DEFAULT
onlineUserOnly:NO offlinePushInfo:info progress:^(uint32_t progress) {
} succ:^{
} fail:^(int code, NSString *msg) {
}];
```

- 接收方：vinson 的 App 虽然不在线，但可以接收到 APNS 离线推送，当 vinson 点击推送消息时会启动 App：

```
// vinson 启动 App 后会收到以下回调
- (void)application:(UIApplication *)application didReceiveRemoteNotification:(NSDictionary *)userInfo
fetchCompletionHandler:(void (^)(UIBackgroundFetchResult result))completionHandler {
// 解析推送扩展字段 desc
if ([userInfo[@"ext"] isEqualToString:@"jump to denny"]) {
//跳转到和 denny 的聊天界面
}
}
```

设置消息为只能在线接收（onlineUserOnly）

某些场景下，您可能希望发出去的消息只被在线用户接收，即当接收者不在线时就不会感知到该消息。您只需在 sendMessage 时，将参数 onlineUserOnly 设置为 YES ，此时发送出去的消息与普通消息相比，会有如下差异点：

- 不支持离线存储，即如果接收方不在线就无法收到。
- 不支持多端漫游，即如果接收方在一台终端设备上一旦接收过该消息，无论是否已读，都不会在另一台终端上再次收到。
- 不支持本地存储，即本地的云端的历史消息中均无法找回。

经典示例：实现“对方正在输入”功能

在 C2C 单聊场景下，您可以通过 sendMessage 接口发送 "自己正在输入" 的提示性消息，接收方收到该消息时可以在 UI 界面展示 "对方正在输入"，示例代码如下：

```
// 给 userA 发送 "自己正在输入" 的提示消息
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 progress) {
} succ:^{
// 消息发送成功
} fail:^(int code, NSString *msg) {
// 消息发送失败
}];
```

撤回消息

发送方通过 revokeMessage 接口可以撤回一条已经发送成功的消息。默认情况下，发送者只能撤回2分钟以内的消息，您可以按需更改消息撤回时间限制，具体操作请参见 消息撤回设置。

消息的撤回同时需要接收方 UI 代码的配合：当发送方撤回一条消息后，接收方会收到消息撤回通知 onRecvMessageRevoked，通知中包含被撤回消息的 msgID ，您可以根据 msgID 判断 UI 层是哪一条消息被撤回了，然后把对应的消息气泡切换成 "消息已被撤回" 状态。

发送方撤回一条消息

```
[[V2TIMManager sharedInstance] revokeMessage:msg succ:^{
// 撤回消息成功
} fail:^(int code, NSString *msg) {
// 撤回消息失败
}];
```

接收方感知消息被撤回

1. 调用 addAdvancedMsgListener 设置高级消息监听。
2. 通过 onRecvMessageRevoked 接收消息撤回通知。

```
- (void)onRecvMessageRevoked:(NSString *)msgID {
// msgList 为当前聊天界面的消息列表
for(V2TIMMessage *msg in msgList){
if ([msg.msgID isEqualToString:msgID]) {
//msg 即为被撤回的消息，您需要修改 UI 上相应的消息气泡的状态
}
}
}
```

给消息增加已读回执

在 C2C 单聊场景下，当接收方通过 markC2CMessageAsRead 接口将来自某人的消息标记为已读时，消息的发送方将会收到“已读回执”，表示“xxx 已经读过我的消息了”。

注意：

目前仅 C2C 单聊消息支持已读回执，群聊场景暂不支持。虽然群聊消息也有对应的 markGroupMessageAsRead 接口，但群消息的发送者目前无法收到已读回执。

接收方标记消息已读

```
//将来自 haven 的消息均标记为已读
[[V2TIMManager sharedInstance] markC2CMessageAsRead:@"haven" succ:^{
} fail:^(int code, NSString *msg) {
}];
```

发送方感知消息已读

消息已读回执的事件通知位于高级消息监听器 V2TIMAdvancedMsgListener 中，如需支持感知消息已读，需要先通过 addAdvancedMsgListener 设置监听器，然后通过 onRecvC2CReadReceipt 回调即可感知接收方的已读确认。

```
- (void)onRecvC2CReadReceipt:(NSArray<V2TIMMessageReceipt *> *)receiptList {
// 接收方可能一次性会收到多个已读回执，因此这里采用数组的回调形式
for (V2TIMMessageReceipt *receipt in receiptList) {
// 消息接收者 receiver
NSString * receiver = receipt.userID;
// 已读回执时间，聊天窗口中时间戳小于或等于 timestamp 的消息都可以被认为已读
time_t timestamp = receipt.timestamp;
}
}
@end
```

查看历史消息

您可以调用 getC2CHistoryMessageList 获取单聊历史消息，调用 getGroupHistoryMessageList 获取群聊历史消息。如果当前设备网络连接正常，SDK 会默认从服务器拉取历史消息；如果没有网络连接，SDK 会直接从本地数据库中读取历史消息。

分页拉取历史消息

SDK 支持分页拉取历史消息，一次分页拉取的消息数量不宜太大，否则会影响拉取速度， 建议一次最多拉取20条。
本文以分页拉取名为 `groupA` 的群的历史消息，每次分页拉取 20 条为例，示例代码如下：

```
// 第一次拉取 lastMsg 传 nil，表示从最新的消息开始拉取 20 条消息
[[V2TIMManager sharedInstance] getGroupHistoryMessageList:@"groupA" count:20
lastMsg:nil succ:^(NSArray<V2TIMMessage *> *msgs) {
// 分页拉取返回的消息默认是按照从新到旧排列
if (msgs.count > 0) {
// 获取下一次分页拉取的起始消息
V2TIMMessage *lastMsg = msgs.lastObject;
// 拉取剩下的20条消息
[[V2TIMManager sharedInstance] getGroupHistoryMessageList:@"groupA" count:20
lastMsg:lastMsg succ:^(NSArray<V2TIMMessage *> *msgs) {
// 拉取消息结束
} fail:^(int code, NSString *msg) {
// 拉取消息失败
}];
}
} fail:^(int code, NSString *msg) {
// 拉取消息失败
}];
```

现实场景中的分页拉取，通常由用户的滑动操作触发的，用户每下拉一次消息列表就触发一次分页拉取。但原理上跟上述示例代码类似，都是以 `lastMsg` 作为分页的标记，以 `count` 控制每次拉取的消息条数。

历史消息的注意事项

- 历史消息存储时长如下：
  - 体验版：免费存储7天，不支持延长
  - 专业版：免费存储7天，支持延长
  - 旗舰版：免费存储30天，支持延长

延长历史消息存储时长是增值服务，您可以登录 即时通信 IM 控制台 修改相关配置，具体计费说明请参见 增值服务资费。

- 只有会议群（Meeting）（对应老版本的 ChatRoom 群）才支持拉取到用户**入群之前**的历史消息。
- 直播群（AVChatRoom）中的消息均不支持本地存储和多终端漫游，因此对直播群调用 getGroupHistoryMessageList 接口是无效的。

删除消息

对于历史消息，您可以调用 deleteMessages 接口删除历史消息，消息删除后，无法再恢复。

设置消息权限

只允许好友间收发消息

SDK 默认不限制非好友之间收发消息。如果您希望仅允许好友之间发送单聊消息，您可以在 即时通信 IM 控制台 >【功能配置】>【登录与消息】>【好友关系检查】中开启"发送单聊消息检查关系链"。开启后，用户只能给好友发送消息，当用户给非好友发消息时，SDK 会报20009错误码。

屏蔽某人的消息

如果您想屏蔽某人的消息，请调用 addToBlackList 接口把该用户加入黑名单，即拉黑该用户。
当消息发送者被拉黑后，发送者默认不会感知到"被拉黑"的状态，即发送消息后仍展示发送成功（实际上此时接收方不会收到消息）。如果需要被拉黑的发送者收到消息发送失败的提示，请在 即时通信 IM 控制台 >【功能配置】>【登录与消息】>【黑名单检查】中关闭"发送消息后展示发送成功"，关闭后，被拉黑的发送者在发送消息时，SDK 会报20007错误码。

屏蔽某个群组的消息

如果您想屏蔽某个群组的消息，请调用 setReceiveMessageOpt 接口，设置群消息接收选项为 `V2TIM_GROUP_NOT_RECEIVE_MESSAGE` 状态。

敏感词过滤

SDK 发送的文本消息默认会经过即时通信 IM 的敏感词过滤，如果发送者在发送的文本消息中包含敏感词，SDK 会报 80001 错误码。

常见问题

1. 为什么会收到重复的消息？

- 请检查 addSimpleMsgListener 与 addAdvancedMsgListener 是否混用。如果混用，当收到文本消息或自定义消息时，两个监听都会回调，会导致收到重复消息。
- 请检查同一个监听对象是否重复 `add`，如果监听对象不再使用，请主动调用对应的 removeSimpleMsgListener 或 removeAdvancedMsgListener 接口移除多余的监听器。

2. App 卸载重装后已读回执为什么失效了？

在单聊场景下，接收方如果调用 markC2CMessageAsRead 设置消息已读，发送方收到的已读回执里面包含了对方已读的时间戳 timestamp ，SDK 内部会根据 timestamp 判断消息对方是否已读， timestamp 目前只在本地保存，程序卸载重装后会丢失。

3. 有多个 Elem 的消息应该如何解析？

出于降低消息复杂度的考虑，SDK API 2.0 接口不再支持创建包含多个 Elem 的 Message 对象。如果您收到了来自老版本的包含多个 Elem 的 Message 对象，可以按照以下步骤解析：

1. 正常解析出第一个 Elem 对象。
2. 通过第一个 Elem 对象的 nextElem 方法获取下一个 Elem 对象。如果下一个 Elem 对象存在，会返回 Elem 对象实例，如果不存在，会返回 nil 。

```
- (void)onRecvNewMessage:(V2TIMMessage *)msg {
// 查看第一个 Elem
if (msg.elemType == V2TIM_ELEM_TYPE_TEXT) {
V2TIMTextElem *textElem = msg.textElem;
NSString *text = textElem.text;
NSLog(@"文本信息：%@", text);
// 查看 textElem 后面还有没更多 Elem
V2TIMElem *elem = textElem.nextElem;
while (elem != nil) {
// 判断 elem 类型
if ([elem isKindOfClass:[V2TIMCustomElem class]]) {
V2TIMCustomElem *customElem = (V2TIMCustomElem *)elem;
NSData *customData = customElem.data;
NSLog(@"自定义信息：%@",customData);
}
// 继续查看当前 elem 后面还有没更多 elem
elem = elem.nextElem;
}
// elem 如果为 nil，表示所有 elem 都已经解析完
}
}
```

4. 各种不同类型的消息应该如何解析？

解析消息相对复杂，我们提供了各种类型消息解析的 示例代码，您可以直接把相关代码拷贝到您的工程，然后根据实际需求进行二次开发。

消息收发（Web & 小程序）

最近更新时间：2020-10-20 14:31:44

本文将为您详细介绍消息收发（Web & 小程序），以下为视频介绍：

点击查看视频

发送消息

创建文本消息

创建文本消息的接口，此接口返回一个消息实例，可以在需要发送文本消息时调用 发送消息 接口发送消息实例。

接口名

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

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Attributes	Default	Description
to	String	-	-	消息接收方的 userID 或 groupID
conversationType	String	-	-	会话类型，取值 TIM.TYPES.CONV_C2C （端到端会话）或 TIM.TYPES.CONV_GROUP （群组会话）
priority	String	<optional>	TIM.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	-	消息内容的容器

payload 的描述如下表所示：

Name	Type	Description
text	String	消息文本内容

示例

```
// 发送文本消息，Web 端与小程序端相同
// 1. 创建消息实例，接口返回的实例可以上屏
let message = tim.createTextMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
// 消息优先级，用于群聊（v2.4.2起支持）。如果某个群的消息超过了频率限制，后台会优先下发高优先级的消息，详细请参考：https://cloud.tencent.com/document/product/269/3663#.E6.B6.88.E6.81.AF.E4.BC.98.E5.85.88.E7.BA.A7.E4.B8.8E.E9.A2.91.E7.8E.87.E6.8E.A7.E5.88.B6)
// 支持的枚举值：TIM.TYPES.MSG_PRIORITY_HIGH, TIM.TYPES.MSG_PRIORITY_NORMAL（默认）, TIM.TYPES.MSG_PRIORITY_LOW, TIM.TYPES.MSG_PRIORITY_LOWEST
// priority: TIM.TYPES.MSG_PRIORITY_NORMAL,
payload: {
text: 'Hello world!'
}
});
// 2. 发送消息
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// 发送成功
console.log(imResponse);
}).catch(function(imError) {
// 发送失败
console.warn('sendMessage error:', imError);
});
```

返回

消息实例 Message。

创建图片消息

创建图片消息的接口，此接口返回一个消息实例，可以在需要发送图片消息时调用 发送消息 接口发送消息实例。

> 注意：
> v2.3.1版本开始支持传入 File 对象，使用前需要将 SDK 升级至v2.3.1或以上。

接口

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

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Attributes	Default	Description
to	String	-	-	消息的接收方
conversationType	String	-	-	会话类型，取值 TIM.TYPES.CONV_C2C 或 TIM.TYPES.CONV_GROUP
priority	String	<optional>	TIM.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	-	消息内容的容器
onProgress	function	-	-	获取上传进度的回调函数

paylaod 的描述如下表所示：

Name	Type	Description
file	HTMLInputElement 或 Object	用于选择图片的 DOM 节点（Web）或者 File 对象（Web）或者微信小程序 wx.chooseImage 接口的 success 回调参数。SDK 会读取其中的数据并上传图片

Web 示例

```
// Web 端发送图片消息示例1 - 传入 DOM 节点
// 1. 创建消息实例，接口返回的实例可以上屏
let message = tim.createImageMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
// 消息优先级，用于群聊（v2.4.2起支持）。如果某个群的消息超过了频率限制，后台会优先下发高优先级的消息，详细请参考 消息优先级与频率控制
// 支持的枚举值：TIM.TYPES.MSG_PRIORITY_HIGH, TIM.TYPES.MSG_PRIORITY_NORMAL（默认）, TIM.TYPES.MSG_PRIORITY_LOW, 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. 发送消息
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// 发送成功
console.log(imResponse);
}).catch(function(imError) {
// 发送失败
console.warn('sendMessage error:', imError);
});


// Web 端发送图片消息示例2- 传入 File 对象
// 先在页面上添加一个 ID 为 "testPasteInput" 的消息输入框，例如 <input type="text" id="testPasteInput" placeholder="截图后粘贴到输入框中" 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];
// 图片消息发送成功后，file 指向的内容可能被浏览器清空，如果接入侧有额外的渲染需求，可以提前复制一份数据
fileCopy = file.slice();
}
if (typeof file === 'undefined') {
console.warn('file 是 undefined，请检查代码或浏览器兼容性！');
return;
}
// 1. 创建消息实例，接口返回的实例可以上屏
let message = tim.createImageMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
payload: {
file: file
},
onProgress: function(event) { console.log('file uploading:', event) }
});
// 2. 发送消息
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// 发送成功
console.log(imResponse);
}).catch(function(imError) {
// 发送失败
console.warn('sendMessage error:', imError);
});
});
```

小程序示例

```
// 小程序端发送图片
// 1. 选择图片
wx.chooseImage({
sourceType: ['album'], // 从相册选择
count: 1, // 只选一张，目前 SDK 不支持一次发送多张图片
success: function (res) {
// 2. 创建消息实例，接口返回的实例可以上屏
let message = tim.createImageMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
payload: { file: res },
onProgress: function(event) { console.log('file uploading:', event) }
});
// 3. 发送图片
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// 发送成功
console.log(imResponse);
}).catch(function(imError) {
// 发送失败
console.warn('sendMessage error:', imError);
});
}
})
```

返回

消息实例 Message。

创建音频消息

创建音频消息实例的接口，此接口返回一个消息实例，可以在需要发送音频消息时调用 发送消息 接口发送消息。 目前 createAudioMessage 只支持在微信小程序环境使用。

接口

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

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Attributes	Default	Description
to	String	-	-	消息的接收方
conversationType	String	-	-	会话类型，取值 TIM.TYPES.CONV_C2C 或 TIM.TYPES.CONV_GROUP
priority	String	<optional>	TIM.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	-	消息内容的容器

paylaod 的描述如下表所示：

Name	Type	Description
file	Object	录音后得到的文件信息

小程序示例

```
// 示例：使用微信官方的 RecorderManager 进行录音，参考 RecorderManager.start(Object object)
// 1. 获取全局唯一的录音管理器 RecorderManager
const recorderManager = wx.getRecorderManager();


// 录音部分参数
const recordOptions = {
duration: 60000, // 录音的时长，单位 ms，最大值 600000（10 分钟）
sampleRate: 44100, // 采样率
numberOfChannels: 1, // 录音通道数
encodeBitRate: 192000, // 编码码率
format: 'aac' // 音频格式，选择此格式创建的音频消息，可以在即时通信 IM 全平台（Android、iOS、微信小程序和 Web）互通
};

// 2.1 监听录音错误事件
recorderManager.onError(function(errMsg) {
console.warn('recorder error:', errMsg);
});
// 2.2 监听录音结束事件，录音结束后，调用 createAudioMessage 创建音频消息实例
recorderManager.onStop(function(res) {
console.log('recorder stop', res);

// 4. 创建消息实例，接口返回的实例可以上屏
const message = tim.createAudioMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
payload: {
file: res
}
});

// 5. 发送消息
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// 发送成功
console.log(imResponse);
}).catch(function(imError) {
// 发送失败
console.warn('sendMessage error:', imError);
});
```

```
});

// 3. 开始录音
recorderManager.start(recordOptions);
```

返回

消息实例 Message。

创建文件消息

创建文件消息的接口，此接口返回一个消息实例，可以在需要发送文件消息时调用 发送消息 接口发送消息实例。

> 注意：
>
> ! v2.3.1版本开始支持传入 File 对象，使用前需要将 SDK 升级至v2.3.1或以上。
> ! v2.4.0版本起，上传文件大小最大值调整为100MB。
> ! 微信小程序目前不支持选择文件的功能，故该接口暂不支持微信小程序端。

接口

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

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Attributes	Default	Description
to	String	-	-	消息接收方的 userID 或 groupID
conversationType	String	-	-	会话类型，取值 TIM.TYPES.CONV_C2C （端到端会话）或 TIM.TYPES.CONV_GROUP （群组会话）
priority	String	<optional>	TIM.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	-	消息内容的容器
onProgress	function	-	-	获取上传进度的回调函数

payload 的描述如下表所示：

Name	Type	Description
file	HTMLInputElement	用于选择文件的 DOM 节点（Web）或者 File 对象（Web），SDK 会读取其中的数据并上传文件

示例

```
// Web 端发送文件消息示例1 - 传入 DOM 节点
// 1. 创建文件消息实例，接口返回的实例可以上屏
let message = tim.createFileMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
// 消息优先级，用于群聊（v2.4.2起支持）。如果某个群的消息超过了频率限制，后台会优先下发高优先级的消息，详细请参考 消息优先级与频率控制
// 支持的枚举值：TIM.TYPES.MSG_PRIORITY_HIGH, TIM.TYPES.MSG_PRIORITY_NORMAL（默认）, TIM.TYPES.MSG_PRIORITY_LOW, 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. 发送消息
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
```

```
// 发送成功
console.log(imResponse);
}).catch(function(imError) {
// 发送失败
console.warn('sendMessage error:', imError);
});

// Web 端发送文件消息示例2- 传入 File 对象
// 先在页面上添加一个 ID 为 "testPasteInput" 的消息输入框，如 <input type="text" id="testPasteInput" placeholder="截图后粘贴到输入框中" 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];
// 图片消息发送成功后，file 指向的内容可能被浏览器清空，如果接入侧有额外的渲染需求，可以提前复制一份数据
fileCopy = file.slice();
}
if (typeof file === 'undefined') {
console.warn('file 是 undefined，请检查代码或浏览器兼容性！');
return;
}
// 1. 创建消息实例，接口返回的实例可以上屏
let message = tim.createFileMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
payload: {
file: file
},
onProgress: function(event) { console.log('file uploading:', event) }
});
// 2. 发送消息
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// 发送成功
console.log(imResponse);
}).catch(function(imError) {
// 发送失败
console.warn('sendMessage error:', imError);
});
});
```

返回

消息实例 Message。

创建自定义消息

创建自定义消息实例的接口，此接口返回一个消息实例，可以在需要发送自定义消息时调用 发送消息 接口发送消息实例。
当 SDK 提供的能力不能满足您的需求时，可以使用自定义消息进行个性化定制，例如投骰子功能。

接口

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

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Attributes	Default	Description
to	String	-	-	消息接收方的 userID 或 groupID
conversationType	String	-	-	会话类型，取值 TIM.TYPES.CONV_C2C （端到端会话） 或 TIM.TYPES.CONV_GROUP （群组会话）
priority	String	<optional>	TIM.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	-	消息内容的容器

payload 的描述如下表所示：

Name	Type	Description
data	String	自定义消息的数据字段
description	String	自定义消息的说明字段
extension	String	自定义消息的扩展字段

示例

```
// 示例：利用自定义消息实现投骰子功能
// 1. 定义随机函数
function random(min, max) {
return Math.floor(Math.random() * (max - min + 1) + min);
}
// 2. 创建消息实例，接口返回的实例可以上屏
let message = tim.createCustomMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
// 消息优先级，用于群聊（v2.4.2起支持）。如果某个群的消息超过了频率限制，后台会优先下发高优先级的消息，详细请参考 消息优先级与频率控制
// 支持的枚举值：TIM.TYPES.MSG_PRIORITY_HIGH, TIM.TYPES.MSG_PRIORITY_NORMAL（默认）, TIM.TYPES.MSG_PRIORITY_LOW, TIM.TYPES.MSG_PRIORITY_LOWEST
// priority: TIM.TYPES.MSG_PRIORITY_HIGH,
payload: {
data: 'dice', // 用于标识该消息是骰子类型消息
description: String(random(1,6)), // 获取骰子点数
extension: ''
}
});
// 3. 发送消息
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// 发送成功
console.log(imResponse);
}).catch(function(imError) {
// 发送失败
console.warn('sendMessage error:', imError);
});
```

返回

消息实例 Message。

创建视频消息

创建视频消息实例的接口，此接口返回一个消息实例，可以在需要发送视频消息时调用 发送消息 接口发送消息。

注意：

- 使用该接口前，需要将SDK版本升级至v2.2.0或以上。
- createVideoMessage 支持在微信小程序环境使用，从v2.6.0起，支持在 Web 环境使用。
- 微信小程序录制视频，或者从相册选择视频文件，没有返回视频缩略图信息。为了更好的体验，SDK 在创建视频消息时会设置默认的缩略图信息。如果接入侧不想展示默认的缩略图，可在渲染的时候忽略缩图相关信息，自主处理。
- 全平台互通视频消息，移动端请升级使用 最新的 TUIKit 或 SDK。

接口

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

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Attributes	Default	Description
to	String	-	-	消息的接收方
conversationType	String	-	-	会话类型，取值 TIM.TYPES.CONV_C2C 或 TIM.TYPES.CONV_GROUP
priority	String	<optional>	TIM.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	-	消息内容的容器

payload 的描述如下表所示：

Name	Type	Description
file	HTMLInputElement 、 File 或 Object	用于选择视频文件的 DOM 节点（Web）或者 File 对象（Web），或微信小程序录制或者从相册选择的视频文件。SDK 会读取其中的数据并上传

示例

```
// 小程序端发送视频消息示例 wx.chooseVideo
// 1. 调用小程序接口选择视频，接口详情请查阅
wx.chooseVideo({
sourceType: ['album', 'camera'], // 来源相册或者拍摄
maxDuration: 60, // 设置最长时间60s
camera: 'back', // 后置摄像头
success (res) {
// 2. 创建消息实例，接口返回的实例可以上屏
let message = tim.createVideoMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
payload: {
file: res
},
onProgress: function(event) { console.log('video uploading:', event) }
})
// 3. 发送消息
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// 发送成功
console.log(imResponse);
}).catch(function(imError) {
// 发送失败
console.warn('sendMessage error:', imError);
});
}
})

// web 端发送视频消息示例（v2.6.0起支持）：
// 1. 获取视频：传入 DOM 节点
// 2. 创建消息实例
const message = tim.createVideoMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
payload: {
file: document.getElementById('videoPicker') // 或者用event.target
},
onProgress: function(event) { console.log('file uploading:', event) }
});
// 3. 发送消息
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// 发送成功
console.log(imResponse);
}).catch(function(imError) {
// 发送失败
```

```
console.warn('sendMessage error:', imError);
});
```

返回

消息实例 Message。

创建表情消息

创建表情消息实例的接口，此接口返回一个消息实例，可以在需要发送表情消息时调用 发送消息 接口发送消息。

> 注意：
> 使用该接口前，需要将 SDK 版本升级至v2.3.1或以上。

接口

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

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Attributes	Default	Description
to	String	-	-	消息接收方的 userID 或 groupID
conversationType	String	-	-	会话类型，取值 TIM.TYPES.CONV_C2C （端到端会话）或 TIM.TYPES.CONV_GROUP （群组会话）
priority	String	<optional>	TIM.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	-	消息内容的容器

payload 的描述如下表所示：

Name	Type	Description
index	Number	表情索引，用户自定义
data	String	额外数据

示例

```
// 发送表情消息，Web 端与小程序端相同。
// 1. 创建消息实例，接口返回的实例可以上屏
let message = tim.createFaceMessage({
to: 'user1',
conversationType: TIM.TYPES.CONV_C2C,
// 消息优先级，用于群聊（v2.4.2起支持）。如果某个群的消息超过了频率限制，后台会优先下发高优先级的消息，详细请参考 消息优先级与频率控制
// 支持的枚举值：TIM.TYPES.MSG_PRIORITY_HIGH, TIM.TYPES.MSG_PRIORITY_NORMAL（默认），TIM.TYPES.MSG_PRIORITY_LOW, TIM.TYPES.MSG_PRIORITY_LOWEST
// priority: TIM.TYPES.MSG_PRIORITY_NORMAL,
payload: {
index: 1, // Number 表情索引，用户自定义
data: 'tt00' // String 额外数据
}
});
// 2. 发送消息
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
// 发送成功
console.log(imResponse);
}).catch(function(imError) {
// 发送失败
console.warn('sendMessage error:', imError);
});
```

返回

消息实例 Message。

发送消息

发送消息的接口，需先调用下列的创建消息实例的接口获取消息实例后，再调用该接口发送消息实例。

- 创建文本消息
- 创建图片消息
- 创建音频消息
- 创建视频消息
- 创建自定义消息
- 创建表情消息
- 创建文件消息

> 注意：
> 调用该接口发送消息实例，需要 SDK 处于 ready 状态，否则将无法发送消息实例。SDK 状态，可通过监听以下事件得到：
>
> - TIM.EVENT.SDK_READY：SDK 处于 ready 状态时触发。
> - TIM.EVENT.SDK_NOT_READY：SDK 处于 not ready 状态时触发。

接收推送的单聊、群聊、群提示、群系统通知的新消息，需监听事件 TIM.EVENT.MESSAGE_RECEIVED。
本实例发送的消息，不会触发事件 TIM.EVENT.MESSAGE_RECEIVED。同帐号从其他端（或通过 REST API）发送的消息，会触发事件 TIM.EVENT.MESSAGE_RECEIVED，离线推送仅适用于终端（Android 或 iOS），Web 和 微信小程序不支持。

接口

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

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	**Attributes**	Description
message	Message	-	消息实例
options	Object	optional	消息发送选项（消息内容的容器）

options 的描述如下表所示：

Name	Type	**Attributes**	Description
onlineUserOnly	Boolean	optional	v2.6.4起支持，消息是否仅发送给在线用户的标识，默认值为 false；设置为 true，则消息既不存漫游，也不会计入未读，也不会离线推送给接收方。适合用于发送广播通知等不重要的提示消息场景。在 AVChatRoom 发送消息不支持此选项
offlinePushInfo	Object	optional	v2.6.4起支持，离线推送 配置

offlinePushInfo 的描述如下表所示：

Name	Type	**Attributes**	Description
disablePush	Boolean	optional	true 关闭离线推送；false 开启离线推送（默认）
title	String	optional	离线推送标题，该字段为 iOS 和 Android 共用
description	String	optional	离线推送内容，该字段会覆盖消息实例的离线推送展示文本。若发送的是自定义消息，该 description 字段会覆盖 message.payload.description。如果 description 和 message.payload.description 字段都不填，接收方将收不到该自定义消息的离线推送
extension	String	optional	离线推送透传内容

Name	Type	Attributes	Description
ignoreIOSBadge	Boolean	optional	离线推送忽略 badge 计数（仅对 iOS 生效），如果设置为 true，在 iOS 接收端，这条消息不会使 App 的应用图标未读计数增加
androidOPPOChannelID	String	optional	离线推送设置 OPPO 手机 8.0 系统及以上的渠道 ID

示例

```
// 如果接收方不在线，则消息将存入漫游，且进行离线推送（在接收方 App 退后台或者进程被 kill 的情况下）。离线推送的标题和内容使用默认值。
// 离线推送的说明请参考 离线推送
tim.sendMessage(message);
// v2.6.4起支持消息发送选项
tim.sendMessage(message, {
onlineUserOnly: true// 如果接收方不在线，则消息不存入漫游，且不会进行离线推送
});
// v2.6.4起支持消息发送选项
tim.sendMessage(message, {
offlinePushInfo: {
disablePush: true // 如果接收方不在线，则消息将存入漫游，但不进行离线推送
}
});
// v2.6.4起支持消息发送选项
tim.sendMessage(message, {
// 如果接收方不在线，则消息将存入漫游，且进行离线推送（在接收方 App 退后台或者进程被 kill 的情况下）。接入侧可自定义离线推送的标题及内容
offlinePushInfo: {
title: '', // 离线推送标题
description: '', // 离线推送内容
androidOPPOChannelID: '' // 离线推送设置 OPPO 手机 8.0 系统及以上的渠道 ID
}
});
```

返回

Type：Promise

撤回消息

撤回单聊消息或者群聊消息。撤回成功后，消息对象的 isRevoked 属性值为 true。

注意：

- 使用该接口前，需要将 SDK 版本升级至v2.4.0或以上。
- 消息可撤回时间默认为2分钟。可通过 控制台 调整消息可撤回时间。
- 被撤回的消息，可以调用 getMessageList 接口从单聊或者群聊消息漫游中拉取到。接入侧需根据消息对象的 isRevoked 属性妥善处理被撤回消息的展示。例如，单聊会话内可展示为 "对方撤回了一条消息"，群聊会话内可展示为 "张三撤回了一条消息"。
- 可使用 REST API 撤回单聊消息 或 撤回群聊消息。

接口

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

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Description
message	Message	消息实例

示例

```
// 主动撤回消息
let promise = tim.revokeMessage(message);
```

```
promise.then(function(imResponse) {
// 消息撤回成功
}).catch(function(imError) {
// 消息撤回失败
console.warn('revokeMessage error:', imError);
});

tim.on(TIM.EVENT.MESSAGE_REVOKED, function(event) {
// 收到消息被撤回的通知。使用前需要将 SDK 版本升级至v2.4.0或以上。
// event.name - TIM.EVENT.MESSAGE_REVOKED
// event.data - 存储 Message 对象的数组 - [Message] - 每个 Message 对象的 isRevoked 属性值为 true
});

// 获取会话的消息列表时遇到被撤回的消息
let promise = tim.getMessageList({conversationID: 'C2Ctest', count: 15});
promise.then(function(imResponse) {
const messageList = imResponse.data.messageList; // 消息列表
messageList.forEach(function(message) {
if (message.isRevoked) {
// 处理被撤回的消息
} else {
// 处理普通消息
}
});
});
```

返回

Type： Promise

重发消息

重发消息的接口，当消息发送失败时，调用该接口进行重发。

接口

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

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Description
message	Message	消息实例

示例

```
// 重发消息
let promise = tim.resendMessage(message); // 传入需要重发的消息实例
promise.then(function(imResponse) {
// 重发成功
console.log(imResponse.data.message);
}).catch(function(imError) {
// 重发失败
console.warn('resendMessage error:', imError);
});
```

返回

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data.groupList 中获取群组列表。
- catch 的回调函数参数为 IMError。

接收消息

接收消息

请参考 接收消息事件。

接受消息的接口，接收消息需要通过事件监听实现：

示例

```
let onMessageReceived = function(event) {
// event.data - 存储 Message 对象的数组 - [Message]
};
tim.on(TIM.EVENT.MESSAGE_RECEIVED, onMessageReceived);
```

解析文本消息

- **简单版**
  如果您的文本消息只含有文字，则可以直接在 UI 上渲染出``xxxxxxx``文字。
- **含有 [呲牙] 内容需要解析为😀的文本**

```
const emojiMap = { // 根据[呲牙]可匹配的路径地址
'[微笑]': 'emoji_0.png',
'[呲牙]': 'emoji_1.png',
'[下雨]': 'emoji_2.png'
}

const emojiUrl = 'http://xxxxxxxx/emoji/' // 为<img src="https://main.qcloudimg.com/raw/6be88c30a4552b5eb93d8eec243b6593.png" style="margin:0;">图片的地址

function parseText (payload) {
let renderDom = []
// 文本消息
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]) { // 如果您需要渲染表情包，需要进行匹配您对应[呲牙]的表情包地址
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
}


// 最后的 renderDom 结构为[{name: 'text', text: 'XXX'}, {name: 'img', src: 'http://xxx'}.....]
// 渲染当前数组即可得到想要的 UI 结果，如：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[呲牙XXX]
```

解析系统消息

```
function parseGroupSystemNotice (payload) {
const groupName =
payload.groupProfile.groupName || payload.groupProfile.groupID
switch (payload.operationType) {
case 1:
return `${payload.operatorID} 申请加入群组：${groupName}`
case 2:
return `成功加入群组：${groupName}`
case 3:
return `申请加入群组：${groupName}被拒绝`
case 4:
return `被管理员${payload.operatorID}踢出群组：${groupName}`
case 5:
return `群：${groupName} 已被${payload.operatorID}解散`
case 6:
return `${payload.operatorID}创建群：${groupName}`
case 7:
return `${payload.operatorID}邀请你加群：${groupName}`
case 8:
return `你退出群组：${groupName}`
case 9:
return `你被${payload.operatorID}设置为群：${groupName}的管理员`
case 10:
return `你被${payload.operatorID}撤销群：${groupName}的管理员身份`
case 255:
return '自定义群系统通知'
}
}
```

解析群提示消息

```
function parseGroupTipContent (payload) {
switch (payload.operationType) {
case this.TIM.TYPES.GRP_TIP_MBR_JOIN:
return `群成员：${payload.userIDList.join(',')}，加入群组`
case this.TIM.TYPES.GRP_TIP_MBR_QUIT:
return `群成员：${payload.userIDList.join(',')}，退出群组`
case this.TIM.TYPES.GRP_TIP_MBR_KICKED_OUT:
return `群成员：${payload.userIDList.join(',')}，被${payload.operatorID}踢出群组`
case this.TIM.TYPES.GRP_TIP_MBR_SET_ADMIN:
return `群成员：${payload.userIDList.join(',')}，成为管理员`
case this.TIM.TYPES.GRP_TIP_MBR_CANCELED_ADMIN:
return `群成员：${payload.userIDList.join(',')}，被撤销管理员`
default:
return '[群提示消息]'
```

```
}
}
```

会话相关

获取某会话的消息列表

请参考 Conversation。

分页拉取指定会话的消息列表的接口，当用户进入会话首次渲染消息列表或者用户“下拉查看更多消息”时，需调用该接口。

接口

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

> 注意：
> 该接口可用于"拉取历史消息"。

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Attributes	Description
conversationID	String	<optional>	会话 ID。会话 ID 组成方式：C2C+userID（单聊）GROUP+groupID（群聊）@TIM#SYSTEM（系统通知会话）
nextReqMessageID	String	<optional>	用于分页续拉的消息 ID。第一次拉取时该字段可不填，每次调用该接口会返回该字段，续拉时将返回字段填入即可
count	Number	<optional>	需要拉取的消息数量，默认值和最大值为15，即一次拉取至多返回15条消息

示例

```
// 打开某个会话时，第一次拉取消息列表
let promise = tim.getMessageList({conversationID: 'C2Ctest', count: 15});
promise.then(function(imResponse) {
const messageList = imResponse.data.messageList; // 消息列表。
const nextReqMessageID = imResponse.data.nextReqMessageID; // 用于续拉，分页续拉时需传入该字段。
const isCompleted = imResponse.data.isCompleted; // 表示是否已经拉完所有消息。
});


// 打开某个会话时，第一次拉取消息列表
// 下拉查看更多消息
let promise = tim.getMessageList({conversationID: 'C2Ctest', nextReqMessageID, count: 15});
promise.then(function(imResponse) {
const messageList = imResponse.data.messageList; // 消息列表。
const nextReqMessageID = imResponse.data.nextReqMessageID; // 用于续拉，分页续拉时需传入该字段。
const isCompleted = imResponse.data.isCompleted; // 表示是否已经拉完所有消息。
});
```

返回

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data.groupList 中获取群组列表。
- catch 的回调函数参数为 IMError。

将会话设置为已读

将某会话下的未读消息状态设置为已读，置为已读的消息不会计入到未读统计，当打开会话或切换会话时调用该接口。如果在打开/切换会话时，不调用该接口，则对应的消息会一直是未读的状态。

接口

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

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Description
options	Object	消息内容的容器

payload 的描述如下表所示：

Name	Type	Description
conversationID	String	会话 ID。会话 ID 组成方式：C2C+userID（单聊）GROUP+groupID（群聊）@TIM#SYSTEM（系统通知会话）

示例

```
// 将某会话下所有未读消息已读上报
tim.setMessageRead({conversationID: 'C2Cexample'});
```

获取会话列表

获取会话列表的接口，该接口拉取最近的100条会话，当需要刷新会话列表时调用该接口。

注意：

- 该接口获取的会话列表中的资料是不完整的（仅包括头像、昵称等，能够满足会话列表的渲染需求），若要查询详细会话资料，请参考 getConversationProfile。
- 会话保存时长跟会话最后一条消息保存时间一致，消息默认保存7天，即会话默认保存7天。

接口

```
tim.getConversationList()
```

示例

```
// 拉取会话列表
let promise = tim.getConversationList();
promise.then(function(imResponse) {
const conversationList = imResponse.data.conversationList; // 会话列表，用该列表覆盖原有的会话列表
}).catch(function(imError) {
console.warn('getConversationList error:', imError); // 获取会话列表失败的相关信息
});
```

返回

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data.groupList 中获取群组列表。
- catch 的回调函数参数为 IMError。

获取会话资料

获取会话资料的接口，当单击会话列表中的某个会话时，调用该接口获取会话的详细信息。

接口

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

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Description
conversationID	String	会话 ID。会话 ID 组成方式：C2C+userID（单聊）GROUP+groupID（群聊）@TIM#SYSTEM（系统通知会话）

示例

```
let promise = tim.getConversationProfile(conversationID);
promise.then(function(imResponse) {
// 获取成功
console.log(imResponse.data.conversation); // 会话资料
}).catch(function(imError) {
console.warn('getConversationProfile error:', imError); // 获取会话资料失败的相关信息
});
```

返回

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data.groupList 中获取群组列表。
- catch 的回调函数参数为 IMError。

删除会话

根据会话 ID 删除会话的接口，该接口只删除会话，不删除消息。例如，删除与用户 A 的会话，下次再与用户 A 发起会话时，之前的聊天信息仍在。

接口

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

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Description
conversationID	String	会话 ID。会话 ID 组成方式：C2C+userID（单聊）GROUP+groupID（群聊）@TIM#SYSTEM（系统通知会话）

示例

```
let promise = tim.deleteConversation('C2CExample');
promise.then(function(imResponse) {
//删除成功。
const { conversationID } = imResponse.data;// 被删除的会话 ID。
}).catch(function(imError) {
console.warn('deleteConversation error:', imError); // 删除会话失败的相关信息
});
```

返回

该接口返回 Promise 对象：

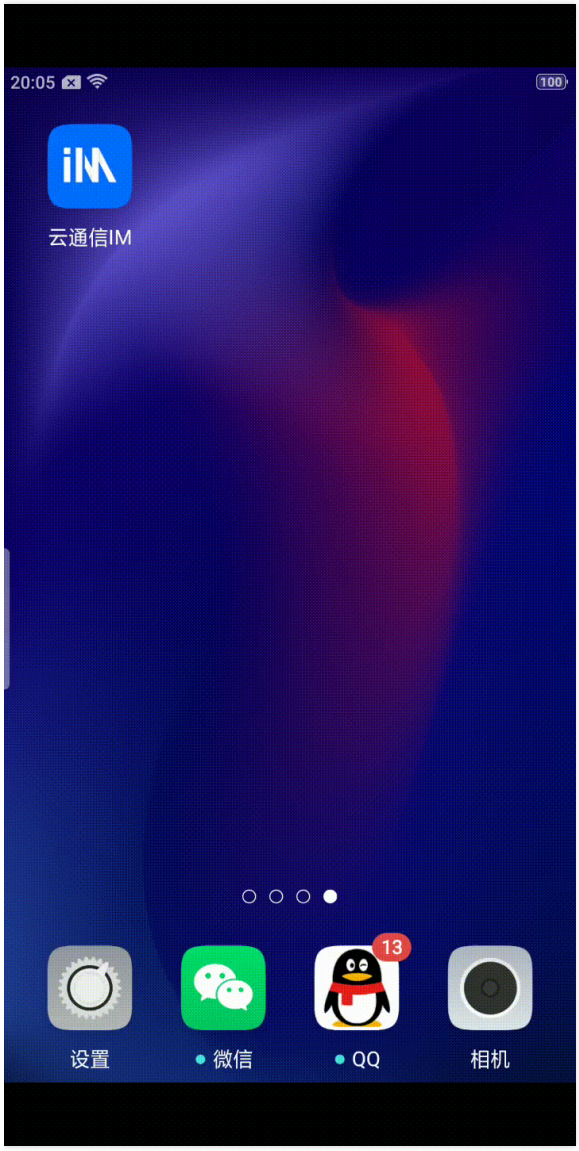
- then 的回调函数参数为 IMResponse，可在 IMResponse.data.groupList 中获取群组列表。
- catch 的回调函数参数为 IMError。

会话相关
会话（Android）

最近更新时间：2020-09-29 14:40:10

展示会话列表

用户在登录 App 后，可以像微信那样展示最近会话列表。整个过程分为**拉取会话列表**、**处理更新通知**和**更新 UI 内容（包括未读计数）**，本文主要介绍这些步骤的详细细节。



拉取会话列表

用户在登录后调用 getConversationList() 拉取本地会话列表做 UI 展示，会话列表是一个 V2TIMConversation 对象的列表，每一个对象都代表一个会话。

由于本地会话可能很多（例如超过500个），一次性全部加载完毕可能会耗时很久，导致界面展示比较慢。为了提升用户体验，`getConversationList()` 接口支持分页拉取能力：

1. 首次调用 `getConversationList()` 接口时，可以指定其参数 `nextSeq` 为0，表示从头开始拉取会话列表，并指定 `count` 为50，表示一次拉取50个会话对象。
2. IM SDK 按照从新到旧的顺序拉取会话列表，当首次拉取会话列表成功后，`getConversationList()` 的回调结果 V2TIMConversationResult 中会包含下次分页拉取的 `nextSeq` 字段以及会话拉取是否完成的 `isFinish` 字段：
   - 如果 `isFinished` 返回 `true`，表示所有会话已经拉取完成。

- 如果 isFinished 返回 false ，表示还有更多的会话可以拉取。此时并不意味着要立刻开始拉取“下一页”的会话列表。在常见的通信软件中，分页拉取通常由用户的滑动操作触发的，用户每下拉一次会话列表就触发一次分页拉取。

3. 当用户继续下拉会话列表时，如果还有没有拉取下来的会话列表，可以继续调用 getConversationList 接口，并传入新一轮的 nextSeq 和 count 参数（数值来自上一次拉取返回的 V2TIMConversationResult 对象）。
4. 重复执行 步骤3 直至 isFinished 返回 true 。

显示会话信息

获取到 V2TIMConversation 对象后，即可在 UI 上展示， V2TIMConversation 有如下关键字段常被用于构造会话列表：

字段名称	含义
getShowName ()	会话名称： • 如果是单聊，此接口会优先返回对方好友备注，若没有备注或者不是好友，则返回对方昵称，若昵称也没有，则返回对方的 UserID。 • 如果是群聊，会显示群的名称。
getFaceUrl ()	会话头像： • 如果是单聊，会显示对方的头像。 • 如果是群聊，会显示群头像。
getRecvOpt ()	消息接收选项，一般用于群会话，可以显示该群是否设置了“消息免打扰”模式。
getUnreadCount ()	用于显示未读计数，表示有多少条未读消息。
getLastMessage ()	最后一条消息，用于显示会话的消息摘要。

更新会话列表

IM SDK 会在登录成功后、用户上线后、以及断线重连后，自动更新会话列表。更新过程如下：

- 当有会话更新时，例如新收到一条消息，SDK 会通过 V2TIMConversationListener 中的 onConversationChanged 事件通知您。
- 当有会话新增时，SDK 会通过 V2TIMConversationListener 中的 onNewConversation 事件通知您。

> 注意：
> 为保证会话列表顺序符合最后一条消息的排序原则，您需要根据 getLastMessage 中的 getTimestamp 对数据源重新排序。

示例代码

示例代码将介绍如何拉取、展示和更新会话列表：

```
// 1. 设置会话监听
V2TIMManager.getConversationManager().setConversationListener(this);
// 2. 先拉取50个本地会话做 UI 展示，nextSeq 第一次拉取传0
V2TIMManager.getConversationManager().getConversationList(0, 50,
new V2TIMValueCallback<V2TIMConversationResult>() {
@Override
public void onError(int code, String desc) {
// 拉取会话列表失败
}
@Override
public void onSuccess(V2TIMConversationResult v2TIMConversationResult) {
// 拉取成功，更新 UI 会话列表
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) {
// 拉取成功，更新 UI 会话列表
updateConversation(v2TIMConversationResult.getConversationList(), false);
```

```
}
});
}
}
// 3.1 收到会话新增的回调
@Override
public void onNewConversation(List<V2TIMConversation> conversationList) {
updateConversation(conversationList, true);
}
// 3.2 收到会话更新的回调
@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);
// UI 会话列表存在该会话，则替换
if (uiConv.getConversationID().equals(conv.getConversationID())) {
uiConvList.set(j, conv);
isExit = true;
break;
}
}
// UI 会话列表没有该会话，则新增
if (!isExit) {
uiConvList.add(conv);
}
}
// 4. 按照会话 lastMessage 的 timestamp 对 UI 会话列表做排序并更新界面
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();
}
```

删除会话

调用 deleteConversation 接口可以删除某个会话，会话删除不支持多端同步，删除会话时默认删除本地和服务器历史消息，且无法恢复。

草稿箱

在发送消息时，可能会遇到消息尚未编辑完就要切换至其它聊天窗口的情况，这些未编辑完的消息可通过 setConversationDraft 接口保存，以便于回到聊天界面后调用 getDraftText 继续编辑内容。

注意：

- 草稿仅支持文本内容。

- 草稿仅在本地保存，不会存储到服务器，因此不能多端同步，程序卸载重装会失效。

常见问题

1. 最近会话列表的保存数量上限是多少？

本地存储的会话列表没有数量上限，云端存储的会话列表最大数量为100。
如果一个会话长时间没有信息变更，该会话在云端最多保存7天，如需放宽限制，请 联系我们。

2. 为什么换了一个手机登录相同帐号后拉取的会话列表不一致？

本地存储的会话和云端存储的会话并不总是一致的，如果用户不主动调用 deleteConversation 接口删除本地的会话，该会话就会一直存在。而云端存储的会话最大只会保存100条，且对于长时间没有信息变更的会话，云端最多保存7天，所以不同的终端本地显示的会话可能会不一样。

3. 为什么会拉取到重复的会话？

调用 `getConversationList` 接口拉取的会话可能已经通过 `onNewConversation` 回调接口添加到了 UI 会话列表的数据源中，因此为了避免重复添加同一个会话，您需要在 UI 会话列表数据源中根据 getConversationID 找到相同的会话并做替换。

4. IM SDK 支持会话置顶吗？

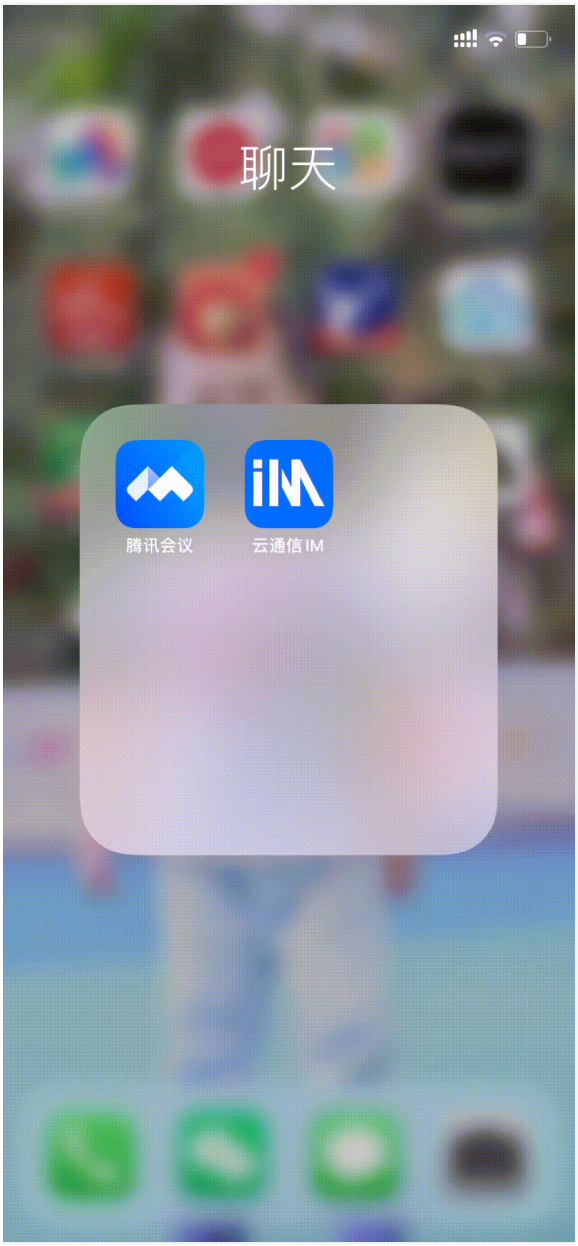
IM SDK 并不提供会话置顶功能，但是可以通过封装会话再重排序，可以参考 TUIKit 实现。置顶仅对本机生效，不会保存到云端上。

会话（iOS）

最近更新时间：2020-09-29 14:40:05

展示会话列表

用户在登录 App 后，可以像微信那样展示最近会话列表。整个过程分为**拉取会话列表**、**处理更新通知**和**更新 UI 内容（包括未读计数）**，本文主要介绍这些步骤的详细细节。



拉取会话列表

用户在登录后调用 getConversationList() 拉取本地会话列表做 UI 展示，会话列表是一个 V2TIMConversation 对象的列表，每一个对象都代表一个会话。

由于本地会话可能较多（例如超过500个），一次性全部加载完毕可能会耗时很久，导致界面展示比较慢。为了提升用户体验，`getConversationList()` 接口支持分页拉取能力：

1. 首次调用 `getConversationList()` 接口时，可以指定其参数 `nextSeq` 为0，表示从头开始拉取会话列表，并指定 `count` 为50，表示一次拉取50个会话对象。
2. IM SDK 按照从新到旧的顺序拉取会话列表，当首次拉取会话列表成功后，`getConversationList()` 的回调结果 V2TIMConversationResult 中会包含下次分页拉取的 `nextSeq` 字段以及会话拉取是否完成的 `isFinish` 字段：
   - 如果 `isFinished` 返回 `true`，表示所有会话已经拉取完成。

- 如果 isFinished 返回 false ，表示还有更多的会话可以拉取。此时并不意味着要立刻开始拉取“下一页”的会话列表。在常见的通信软件中，分页拉取通常由用户的滑动操作触发的，用户每下拉一次会话列表就触发一次分页拉取。

3. 当用户继续下拉会话列表时，如果还有更多的会话可以拉取，可以继续调用 getConversationList() 接口，并传入新一轮的 nextSeq 和 count 参数（数值来自上一次拉取返回的 V2TIMConversationResult 对象）。
4. 重复执行 步骤3 直至 isFinished 返回 true 。

显示会话信息

获取到 V2TIMConversation 对象后，即可在 UI 上展示， V2TIMConversation 有如下关键字段常被用于构造会话列表：

字段名称	含义
showName	会话名称： • 如果是单聊，此接口会优先返回对方好友备注，若没有备注或者不是好友，则返回对方昵称，若昵称也没有，则返回对方的 UserID。 • 如果是群聊，会显示群的名称。
faceUrl	会话头像： • 如果是单聊，会显示对方的头像。 • 如果是群聊，会显示群头像。
recvOpt	消息接收选项，一般用于群会话，可以显示该群是否设置了“消息免打扰”模式。
unreadCount	用于显示未读计数，表示有多少条未读消息。
lastMessage	最后一条消息，用于显示会话的消息摘要。

更新会话列表

IM SDK 会在登录成功后、用户上线后、以及断线重连后，自动更新会话列表。更新过程如下：

- 当有会话更新时，例如新收到一条消息，SDK 会通过 V2TIMConversationListener 中的 onConversationChanged 事件通知您。
- 当有会话新增时，SDK 会通过 V2TIMConversationListener 中的 onNewConversation 事件通知您。

> 注意：
> 为保证会话列表顺序符合最后一条消息的排序原则，您需要根据 lastMessage 中的 timestamp 对数据源重新排序。

示例代码

示例代码将介绍如何拉取、展示和更新会话列表：

```
// 1. 设置会话监听
[[V2TIMManager sharedInstance] setConversationListener:self];
// 2. 登录
[[V2TIMManager sharedInstance] login:@"yahaha" userSig:@"传入实际的 userSig" succ:^{
// 3. 先拉取50个本地会话做 UI 展示，nextSeq 第一次拉取传 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;
// 拉取成功，更新 UI 会话列表
[strongSelf updateConversation:list];
// 4. 如果会话还没拉取完，按需继续拉取，nextSeq 传上次拉取返回的 nextSeq
if(!isFinished) {
[[V2TIMManager sharedInstance] getConversationList:nextSeq count:50
succ:^(NSArray<V2TIMConversation *> *list, uint64_t nextSeq, BOOL isFinished) {
// 拉取成功，更新 UI 会话列表
[strongSelf updateConversation:list];
} fail:^(int code, NSString *msg) {
// 拉取会话列表失败
}];
}
} fail:^(int code, NSString *msg) {
// 拉取会话列表失败
```

```
}];
} fail:^(int code, NSString *msg) {
// 登录失败
}];

// 收到会话新增的回调
- (void)onNewConversation:(NSArray<V2TIMConversation*> *) conversationList {
[self updateConversation:conversationList];
}

// 收到会话更新的回调
- (void)onConversationChanged:(NSArray<V2TIMConversation*> *) conversationList {
[self updateConversation:conversationList];
}

// 更新 UI 会话列表
- (void)updateConversation:(NSArray *)convList
{
// 如果 UI 会话列表有更新的会话，就替换，如果没有，就新增
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];
// UI 会话列表有更新的会话，直接替换
if ([uiConv.conversationID isEqualToString:conv.conversationID]) {
[self.uiConvList replaceObjectAtIndex:j withObject:conv];
isExit = YES;
break;
}
}
// UI 会话列表没有更新的会话，直接新增
if (!isExit) {
[self.uiConvList addObject:conv];
}
}
// 重新按照会话 lastMessage 的 timestamp 对 UI 会话列表做排序
[self.uiConvList sortUsingComparator:^NSComparisonResult(V2TIMConversation *obj1, V2TIMConversation *obj2) {
return [obj2.lastMessage.timestamp compare:obj1.lastMessage.timestamp];
}];
}
```

删除会话

调用 deleteConversation 接口可以删除某个会话，会话删除不支持多端同步，删除会话时默认删除本地和服务器历史消息，且无法恢复。

草稿箱

在发送消息时，可能会遇到消息尚未编辑完就要切换至其它聊天窗口的情况，这些未编辑完的消息可通过 setConversationDraft 接口保存，以便于回到聊天界面后调用 draftText 继续编辑内容。

注意：

- 草稿仅支持文本内容。
- 草稿仅在本地保存，不会存储到服务器，因此不能多端同步，程序卸载重装会失效。

常见问题

1. 最近会话列表的保存数量上限是多少？

本地存储的会话列表没有数量上限，云端存储的会话列表最大数量为100。
如果一个会话长时间没有信息变更，该会话在云端最多保存7天，如需放宽限制，请 联系我们。

2. 为什么换了一个手机登录相同帐号后拉取的会话列表不一致？

本地存储的会话和云端存储的会话并不总是一致的，如果用户不主动调用 deleteConversation 接口删除本地的会话，该会话就会一直存在。而云端存储的会话最大只会保存100条，且对于长时间没有信息变更的会话，云端最多保存7天，所以不同的终端本地显示的会话可能会不一样。

3. 为什么会拉取到重复的会话？

调用 `getConversationList` 接口拉取的会话可能已经通过 `onNewConversation` 回调接口添加到了 UI 会话列表的数据源中，因此为了避免重复添加同一个会话，您需要在 UI 会话列表数据源中根据 getConversationID 找到相同的会话并做替换。

4. IM SDK 支持会话置顶吗？

IM SDK 并不提供会话置顶功能，但是可以通过封装会话再重排序，可以参考 TUIKit 实现。置顶仅对本机生效，不会保存到服务器上。

未读计数（Web & 小程序）

最近更新时间：2020-07-02 17:47:54

本文的未读消息是指用户没有进行已读上报的消息，而非对方是否已经阅读。如需显示正确的未读计数，需要开发者显式调用已读上报，告诉 IM SDK 某个会话的消息是否已读，例如，当用户进入聊天界面，可以设置整个会话的消息已读。

获取当前未读消息数量

每次使用 getConversationList() 时，会获得[Conversation，Conversation，.....]数组，每个 Conversation 都有当前会话的未读数目，用 unreadCount 表示。
所有会话的未读计数，由所有会话的 unreadCount 相加所得。

已读上报

当用户阅读某个会话的消息后，需要进行会话消息的已读上报，IM SDK 根据会话中最后一条阅读的消息，设置会话中之前所有消息为已读。建议在单击进行切换会话时进行消息的已读上报。

> 说明：
> 已读上报会改变会话的未读计数。v2.7.0 起，设置 C2C 会话消息已读，会向对端推送已读回执，请参考事件 TIM.EVENT.MESSAGE_READ_BY_PEER。

接口

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

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Description
conversationID	String	会话 ID

示例

```
// 将某会话下所有未读消息已读上报
let promise = tim.setMessageRead({conversationID: 'C2Cexample'});
promise.then(function(imResponse) {
// 已读上报成功，指定 ID 的会话的 unreadCount 属性值被置为0
}).catch(function(imError) {
// 已读上报失败
console.warn('setMessageRead error:', imError);
});
```

群组相关
群组管理（Android）

最近更新时间：2020-09-30 11:24:51

群类型介绍

即时通信 IM 群组分为以下类型：

- **好友工作群（Work）**：类似普通微信群，创建后仅支持已在群内的好友邀请加群，且无需被邀请方同意或群主审批。
- **陌生人社交群（Public）**：类似 QQ 群，创建后群主可以指定群管理员，用户搜索群 ID 发起加群申请后，需要群主或管理员审批通过才能入群。
- **临时会议群（Meeting）**：创建后可以随意进出，且支持查看入群前消息；适合用于音视频会议场景、在线教育场景等与实时音视频产品结合的场景。
- **直播群（AVChatRoom）**：创建后可以随意进出，没有群成员数量上限，但不支持历史消息存储；适合与直播产品结合，用于弹幕聊天场景。

每种群类型的功能特性及限制如下表所示：

功能项	好友工作群（Work）	陌生人社交群（Public）	临时会议群（Meeting）	直播群（AVChatRoom）
可用群成员角色	群主、普通成员	群主、管理员、普通成员	群主、管理员、普通成员	群主、普通成员
是否支持申请加群	不支持	支持，但需要群主或管理员审批	支持，且无需审批	支持，且无需审批
是否支持成员邀请他人加群	支持	不支持	不支持	不支持
是否支持群主退群	支持	不支持	不支持	不支持
群组资料修改权限	任意群成员均可修改	群主和管理员	群主和管理员	群主
“踢人”权限	群主可踢人	群主和管理员可踢人，但管理员仅支持踢普通群成员		不支持踢人，可用“禁言”功能达到类似效果
“禁言”权限	不支持禁言	群主和管理员可禁言，但管理员仅支持禁言普通群成员		群主可禁言
是否支持未读消息计数	支持	支持	不支持	不支持
是否支持查看入群前消息记录	不支持	不支持	支持	不支持
是否支持云端历史消息存储	• 体验版：7天 • 专业版 ：默认7天，最高支持 增值 延长至360天 • 旗舰版 ：默认30天，最高支持 增值 延长至360天			不支持
群组数量	• 体验版：最多同时存在100个，已解散的群组不计数 • 专业版或旗舰版：无上限			• 体验版：最多同时存在10个，已解散的群组不计数 • 专业版：最多同时存在50个，已解散的群组不计数; 支持 增值 扩展直播群创建数至无上限 • 旗舰版：无上限
群成员数量	• 体验版：20人/群 • 专业版 ：默认为200人/群，最高支持 增值 扩展至2000人/群 • 旗舰版 ：默认为2000人/群，最高支持 增值 扩展至6000人/群			群成员人数无上限

说明：
专业版或旗舰版 SDKAppID 下，所有群类型日净增群组数上限为1万个。免费峰值群组数为10万个/月，超出免费量将产生 套餐外超量费用。

群组管理

创建群组

简化版接口

调用 createGroup 接口，并指定需要的 groupType 、 groupID 和 groupName 参数，即可简单创建一个群组。

高级版接口

如果您想在创建群组的同时初始化群的信息，例如群简介、群头像、以及最初的几个群成员等，可以调用 V2TIMGroupManager 管理类中的 createGroup 接口实现，其中 V2TIMGroupManager 管理类可以通过 V2TIMManager.getGroupManager 获取。

```
// 示例代码：使用高级版 createGroup 创建一个工作群
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) {
// 创建失败
}
@Override
public void onSuccess(String groupID) {
// 创建成功
}
});
```

- 参数 groupType 是字符串类型，可以选择“Work”、“Public”、“Meeting”和“AVChatRoom”中的任何一个，各种不同类型之间的差异请参见 群类型介绍。
- 参数 groupID 用于指定群组 ID，它用于唯一标识一个群，请勿在同一个 SDKAppID 下创建相同 groupID 的群。如果您指定 groupID 为 null，系统会为您默认分配一个群 ID。
- 参数 groupName 用于指定群的描述信息，最长支持30个字节。

加入群组

不同类型的群，加群的方法不同， 下面根据加群流程从简单到复杂进行逐一介绍：

类型	好友工作群（Work）	陌生人社交群（Public）	临时会议群（Meeting）	直播群（AVChatRoom）
加群方法	必须由其他群成员邀请	用户申请，群主或管理员审批	用户可随意加入	用户可随意加入

场景一：用户可以自由进出群

临时会议群（Meeting）和直播群（AVChatRoom）主要用于满足成员进进出出的交互场景，例如在线会议，秀场直播等。因此，这两种类型群的入群流程最为简单。

用户调用 joinGroup 即可加入该群，加群成功后，全体群成员（包括加群者）都会收到 onMemberEnter 回调。

场景二：需被邀请才能进入群

好友工作群（Work）类似微信群和企业微信群，适用于工作交流，在交互设计上限制用户主动加入，只能由现有的群成员邀请才能加群。

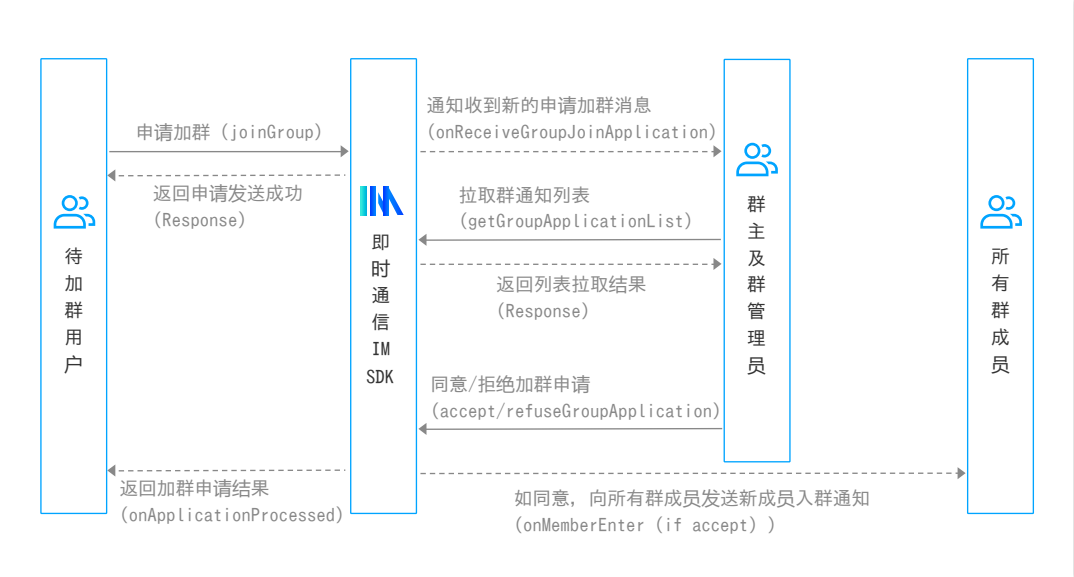
现有的群成员调用 inviteUserToGroup 邀请另一个用户入群，全体群成员（包括邀请者自己）会收到 onMemberInvited 回调。

场景三：需要审批才能进入群

陌生人社交群（Public）类似 QQ 中的各种兴趣群和部落区，任何人都可以申请入群，但需要经过群主或管理员审批才能真正入群。陌生人社交群默认需要群主或管理员进行审批才能加群的，但群主或管理员也可以通过 setGroupInfo 接口调整加群选项（ V2TIMGroupAddOpt ），可以设置为更严格的“禁止任何人加群”，也可以设置为更宽松的“放开审批流程”。

- V2TIM_GROUP_ADD_FORBID ：禁止任何人加群。
- V2TIM_GROUP_ADD_AUTH ：需要群主或管理员审批才能加入（默认值）。
- V2TIM_GROUP_ADD_ANY ：取消审批流程，任何用户都可以加入。

需要审批才能进入群的流程如下：



1. **申请者提出加群申请**
   申请者调用 joinGroup 申请加群。
2. **群主或管理员处理加群申请**
   群主或管理员在收到加群申请的回调 onReceiveJoinApplication 后调用接口 getGroupApplicationList 获取加群申请列表，然后通过 acceptGroupApplication 或者 refuseGroupApplication 来同意或者拒绝某一条加群请求。
3. **申请者收到处理结果**
   请求加群被同意或者拒绝后，请求者会收到 V2TIMGroupListener 中的 onApplicationProcessed 回调，其中 `isAgreeJoin` 为 `true` 表示同意加群，反之被拒绝。同意加群后，全员（包括请求者）收到 onMemberEnter 回调。

退出群组

调用 quitGroup 可以退出群组，退群者会收到 onQuitFromGroup 回调，群其他成员会收到 onMemberLeave 回调。

> 注意：
> 对于陌生人社交群（Public）、临时会议群（Meeting）和直播群（AVChatRoom），群主不可以退群的，群主只能 解散群组。

解散群组

调用 dismissGroup 可以解散群组，全员会收到 onGroupDismissed 回调。

> 注意：
> - 对于陌生人社交群（Public）、临时会议群（Meeting）和直播群（AVChatRoom），群主随时可以解散群。
> - 好友工作群（Work）的解散最为严格，即使群主也不能随意解散，只能由您的业务服务器调用 解散群组 REST API 解散。

获取已加入的群组

调用 getJoinedGroupList 可以获取已加入的好友工作群（Work）、陌生人社交群（Public）、临时会议群（Meeting）列表，但直播群（AVChatRoom）不包含在此列表中。

群资料和群设置

获取群资料

调用 getGroupsInfo 可以获取群资料，该接口支持批量获取。您可以一次传入多个 `groupID` 获取多个群的群资料。

修改群资料

调用 setGroupInfo 可以修改群资料。群资料被修改后，全员会收到 onGroupInfoChanged 回调。

> 注意：
>
> - 好友工作群（Work）所有群成员都可以修改群基础资料。
> - 陌生人社交群（Public）、临时会议群（Meeting）只有群主或管理员可以修改群基础资料。
> - 直播群（AVChatRoom）只有群主可以修改群基础资料。

```
// 示例代码：修改群资料
V2TIMGroupInfo v2TIMGroupInfo = new V2TIMGroupInfo();
v2TIMGroupInfo.setGroupID("需要修改的群 ID");
v2TIMGroupInfo.setFaceUrl("http://xxxx");
V2TIMManager.getGroupManager().setGroupInfo(v2TIMGroupInfo, new V2TIMCallback() {
@Override
public void onError(int code, String desc) {
// 失败
}
@Override
public void onSuccess() {
// 成功
}
});
```

设置群消息的接收选项

任何群成员都可以调用 setReceiveMessageOpt 接口修改群消息接收选项。群消息接收选项包括：

- V2TIMGroupInfo.V2TIM_GROUP_RECEIVE_MESSAGE：在线正常接收消息，离线时会有厂商的离线推送通知。
- V2TIMGroupInfo.V2TIM_GROUP_NOT_RECEIVE_MESSAGE：不会接收到群消息。
- V2TIMGroupInfo.V2TIM_GROUP_RECEIVE_NOT_NOTIFY_MESSAGE：在线正常接收消息，离线不会有推送通知。

根据群消息接收选择可以实现群消息免打扰：

- **完全不接收群内消息**
  群消息接收选项设置为 `V2TIMGroupInfo.V2TIM_GROUP_NOT_RECEIVE_MESSAGE` 后，群内的任何消息都收不到，会话列表也不会更新。
- **接收群内消息但不提醒，在会话列表界面显示小圆点，而不显示未读数**

> 说明：
> 此方式需使用未读计数功能，因此仅适用于好友工作群（Work）和陌生人社交群（Public）。

群消息接收选项设置为 `V2TIMGroupInfo.V2TIM_GROUP_RECEIVE_NOT_NOTIFY_MESSAGE` ，当群内收到新消息，会话列表需要更新时，可以通过会话中的 getUnreadCount 获取到消息未读数。根据 getRecvOpt 判断获取到的群消息接收选项为 `V2TIMGroupInfo.V2TIM_GROUP_RECEIVE_NOT_NOTIFY_MESSAGE` 时显示小红点而非消息未读数。

群属性（群自定义字段）

基于 API2.0 我们设计了全新的群自定义字段，我们称之为 "群属性"，其特性如下：

1. 不再需要控制台配置，客户端可以直接增删改查群属性。
2. 最多支持16个群属性，每个群属性的大小最大支持4k，所有群属性的大小最大支持16k。
3. 目前仅支持直播群（AVChatRoom）。
4. initGroupAttributes、setGroupAttributes、deleteGroupAttributes 接口合并计算， SDK 限制为5秒10次，超过后回调8511错误码；后台限制1秒5次，超过后返回10049错误码。
5. getGroupAttributes 接口 SDK 限制5秒20次。

基于群属性，我们可以做语聊房的麦位管理，当有人上麦的时候，可以设置一个群属性管理上麦人信息，当有人下麦的时候，可以删除对应群属性，其他成员可以通过获取群属性列表来展示麦位列表。

初始化群属性

调用 initGroupAttributes 接口可以初始化群属性，如果该群之前有群属性，会先清空原来的群属性。

设置群属性

调用 setGroupAttributes 接口可以设置群属性，如果设置的群属性不存在，会自动添加该群属性。

删除群属性

调用 deleteGroupAttributes 接口可以删除指定群属性，如果 `keys` 字段填 `null` ，则会清空所有的群属性。

获取群属性

调用 getGroupAttributes 接口可以获取指定群属性，如果 `keys` 字段填 `null` ，则会获取所有的群属性。

群属性更新

群属性有任何的更新变化，都会通过 onGroupAttributeChanged 回调出来所有的群属性字段。

群成员管理

获取群成员列表

调用 getGroupMemberList 可以获取某个群的群成员列表，该列表中包含了各个群成员的资料信息，例如用户ID（ `userID` ）、群名片（ `nameCard` ）、头像（ `faceUrl` ）、昵称（ `nickName` ）、进群时间（ `joinTime` ）等信息。
一个群中的成员人数可能很多（例如5000+），群成员列表的拉取接口支持过滤器（ `filter` ）和分页拉取（ `nextSeq` ）两个高级特性。

过滤器（filter）

在调用 getGroupMemberList 接口时，您可以指定 `filter` 确定是否仅拉取特定角色的信息列表。

过滤器	过滤类型
V2TIMGroupMemberFullInfo.V2TIM_GROUP_MEMBER_FILTER_ALL	拉取所有群成员的信息列表
V2TIMGroupMemberFullInfo.V2TIM_GROUP_MEMBER_FILTER_OWNER	仅拉取群主的信息列表
V2TIMGroupMemberFullInfo.V2TIM_GROUP_MEMBER_FILTER_ADMIN	仅拉取群管理员的信息列表
V2TIMGroupMemberFullInfo.V2TIM_GROUP_MEMBER_FILTER_COMMON	仅拉取普通群成员的信息列表

```
//示例代码：通过 filter 参数指定只拉取群主的资料
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) {
// 拉取失败
}

@Override
public void onSuccess(V2TIMGroupMemberInfoResult v2TIMGroupMemberInfoResult) {
// 拉取成功
}
});
```

分页拉取（nextSeq）

很多情况下，用户界面上并不需要展示全部的群成员信息，只需展示群成员列表的第一页即可，等用户单击“下一页”或在列表页下拉刷新时，才需要拉取更多的群成员。针对此类场景，您可以使用分页拉取。

在调用 getGroupMemberList 接口时，一次最多返回50个成员，您可以通过 `nextSeq` 参数分页拉取成员列表， `nextSeq` 参数为分页拉取标志，第一次拉取时请填0。当首次拉取群成员信息成功后， `getGroupMemberList` 的回调结果 V2TIMGroupMemberInfoResult 中会包含 getNextSeq() 接口。

- 如果 getNextSeq() 返回0，表示已经拉取全部的群成员列表。
- 如果 getNextSeq() 返回 >0 的数值，表示还有更多的群成员信息可以拉取。您可以根据用户在 UI 上的操作，选择是否进行第二次接口调用以拉取更多的群成员信息。当您进行第二次拉取时，需要将上一次拉取返回的 `V2TIMGroupMemberInfoResult` 中的 getNextSeq() 作为参数，传入 getGroupMemberList 接口。

```
//示例代码：通过 nextSeq 参数进行分页拉取
{
...
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) {
// 拉取失败
}

@Override
public void onSuccess(V2TIMGroupMemberInfoResult groupMemberInfoResult) {
if (groupMemberInfoResult.getNextSeq() != 0) {
// 继续分页拉取
getGroupMemberList(groupMemberInfoResult.getNextSeq());
...
} else {
// 拉取结束
}
}
});
}
```

获取群成员资料

调用 getGroupMembersInfo 可以获取群成员资料，该接口支持批量获取，您可以一次传入多个 `userID` 获取多个群成员的资料，从而提升网络传输效率。

修改群成员资料

群主或管理员可以调用 setGroupMemberInfo 接口修改群成员的群名片（ `nameCard` ）、 群成员角色（ `role` ）、禁言时间（ `muteUntil` ）以及自定义字段等与群成员相关的资料。

禁言

群主或管理员可以通过 muteGroupMember 禁言某一个群成员并设置禁言时间，禁言时间单位为秒，禁言信息存储于群成员的 muteUtil 属性字段中。群成员被禁言后，全员（包括被禁言的群成员）都会收到 onGroupMemberInfoChanged 事件回调。
群主或管理员也可以通过 setGroupInfo 接口对整个群进行禁言，将 allMuted 属性字段设置为 `true` 即可。全群禁言没有时间限制，需通过将群资料 `setAllMuted(false)` 解除禁言。

踢人

群主或管理员调用 kickGroupMember 接口可以实现踢人。由于直播群（AVChatRoom）对进群没有限制，因此直播群（AVChatRoom）没有支持踢人的接口，您可以使用 muteGroupMember 达到同样的目的。
成员被踢后，全员（包括被踢人）会收到 onMemberKicked 回调。

切换群成员角色

群主调用 setGroupMemberRole 可以对陌生人社交群（Public）或临时会议群（Meeting）中的群成员进行角色切换，可切换角色包括普通成员、管理员。

- 被设置为管理员后，全员（包括被设置的成员）会收到 onGrantAdministrator 回调。
- 被取消管理员后，全员（包括被设置的成员）会收到 onRevokeAdministrator 回调。

转让群主

群主可以调用 transferGroupOwner 把群主转让给其他群成员。
群主转让后，全员会收到 onGroupInfoChanged 回调，其中 `V2TIMGroupChangeInfo` 的 type 为 `V2TIMGroupChangeInfo.V2TIM_GROUP_INFO_CHANGE_TYPE_OWNER` ，value 值为新群主的 UserID。

常见问题

1. 直播群（AVChatRoom）中途掉线又连接上后，能否继续接收消息？

可以继续接收消息，但是直播群（AVChatRoom）中的消息不支持云端存储，因此无法拉取到掉线期间的消息。

2. 为什么群成员进群和退群收不到通知？

请确认群组类型：

- 临时会议群（Meeting）不支持群成员变更通知。
- 直播群（AVChatRoom）消息限制40条/秒，会优先保证高优先级消息的收发，超过限制后会优先丢弃低优先级的消息。

3. 为什么会议群（Meeting） 中的未读数一直为零?

临时会议群（Meeting）和直播群（AVChatRoom）分别配合会议和直播的音视频场景，因此这两类群组均不支持未读消息计数。

群组管理（iOS）

最近更新时间：2020-09-30 11:24:46

群类型介绍

即时通信 IM 群组分为以下类型：

- **好友工作群（Work）**：类似普通微信群，创建后仅支持已在群内的好友邀请加群，且无需被邀请方同意或群主审批。
- **陌生人社交群（Public）**：类似 QQ 群，创建后群主可以指定群管理员，用户搜索群 ID 发起加群申请后，需要群主或管理员审批通过才能入群。
- **临时会议群（Meeting）**：创建后可以随意进出，且支持查看入群前消息；适合用于音视频会议场景、在线教育场景等与实时音视频产品结合的场景。
- **直播群（AVChatRoom）**：创建后可以随意进出，没有群成员数量上限，但不支持历史消息存储；适合与直播产品结合，用于弹幕聊天场景。

每种群类型的功能特性及限制如下表所示：

功能项	好友工作群（Work）	陌生人社交群（Public）	临时会议群（Meeting）	直播群（AVChatRoom）
可用群成员角色	群主、普通成员	群主、管理员、普通成员	群主、管理员、普通成员	群主、普通成员
是否支持申请加群	不支持	支持，但需要群主或管理员审批	支持，且无需审批	支持，且无需审批
是否支持成员邀请他人加群	支持	不支持	不支持	不支持
是否支持群主退群	支持	不支持	不支持	不支持
群组资料修改权限	任意群成员均可修改	群主和管理员	群主和管理员	群主
“踢人”权限	群主可踢人	群主和管理员可踢人，但管理员仅支持踢普通群成员		不支持踢人，可用“禁言”功能达到类似效果
“禁言”权限	不支持禁言	群主和管理员可禁言，但管理员仅支持禁言普通群成员		群主可禁言
是否支持未读消息计数	支持	支持	不支持	不支持
是否支持查看入群前消息记录	不支持	不支持	支持	不支持
是否支持云端历史消息存储	• 体验版：7天 • 专业版 ：默认7天，最高支持 增值 延长至360天 • 旗舰版 ：默认30天，最高支持 增值 延长至360天			不支持
群组数量	• 体验版：最多同时存在100个，已解散的群组不计数 • 专业版或旗舰版：无上限			• 体验版：最多同时存在10个，已解散的群组不计数 • 专业版：最多同时存在50个，已解散的群组不计数; 支持 增值 扩展直播群创建数至无上限 • 旗舰版：无上限
群成员数量	• 体验版：20人/群 • 专业版 ：默认为200人/群，最高支持 增值 扩展至2000人/群 • 旗舰版 ：默认为2000人/群，最高支持 增值 扩展至6000人/群			群成员人数无上限

说明：
专业版或旗舰版 SDKAppID 下，所有群类型日净增群组数上限为1万个。免费峰值群组数为10万个/月，超出免费量将产生 套餐外超量费用。

群组管理

创建群组

简化版接口

调用 createGroup 接口，并指定需要的 groupType 、 groupID 和 groupName 参数，即可简单创建一个群组。

高级版接口

如果您想在创建群组的同时初始化群的信息，例如群简介、群头像、以及最初的几个群成员等，可以调用 V2TIMManager+Group.h 管理类中的 createGroup 接口实现。

```
// 示例代码：使用高级版 createGroup 创建一个工作群
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) {
// 创建群组成功
} fail:^(int code, NSString *msg) {
// 创建群组失败
}];
```

- 参数 groupType 是字符串类型，可以选择 “Work”、“Public”、“Meeting” 和 “AVChatRoom” 中的任何一个，各种不同类型之间的差异请参见 群类型介绍。
- 参数 groupID 用于指定群组 ID，它用于唯一标识一个群，请勿在同一个 SDKAppID 下创建相同 groupID 的群。如果您指定 groupID 为 nil，系统会为您默认分配一个群 ID。
- 参数 groupName 用于指定群的描述信息，最长支持30个字节。

加入群组

不同类型的群，加群的方法不同， 下面根据加群流程从简单到复杂进行逐一介绍：

类型	好友工作群（Work）	陌生人社交群（Public）	临时会议群（Meeting）	直播群（AVChatRoom）
加群方法	必须由其他群成员邀请	用户申请，群主或管理员审批	用户可随意加入	用户可随意加入

场景一：用户可以自由进出群

临时会议群（Meeting）和直播群（AVChatRoom）主要用于满足成员进进出出的交互场景，例如在线会议，秀场直播等。因此，这两种类型群的入群流程最为简单。

用户调用 joinGroup 即可加入该群，加群成功后，全体群成员（包括加群者）都会收到 onMemberEnter 回调。

场景二：需被邀请才能进入群

好友工作群（Work）类似微信群和企业微信群，适用于工作交流，在交互设计上限制用户主动加入，只能由现有的群成员邀请才能加群。

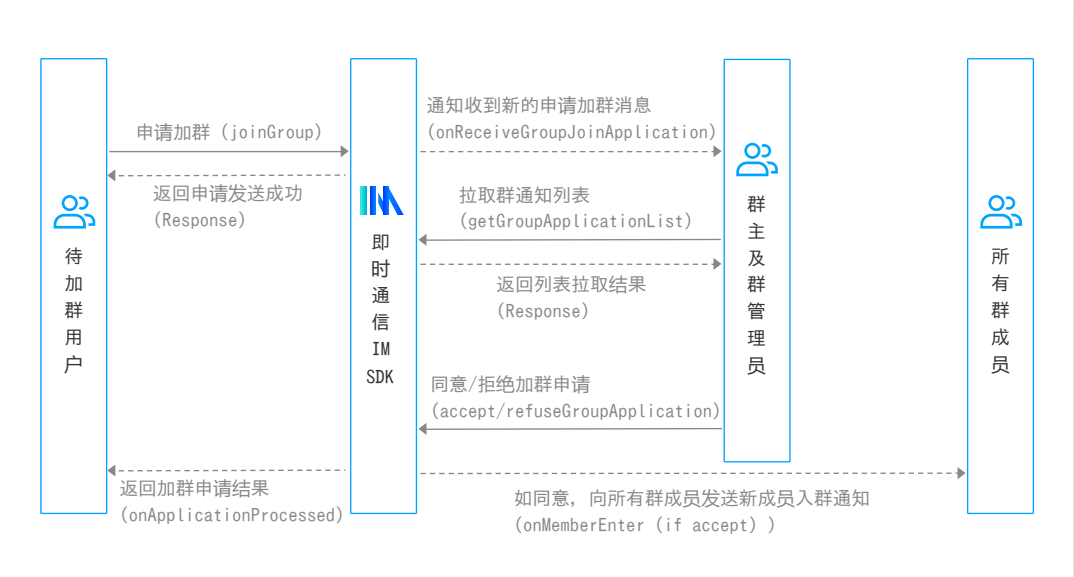
现有的群成员调用 inviteUserToGroup 邀请另一个用户入群，全体群成员（包括邀请者自己）会收到 onMemberInvited 回调。

场景三：需要审批才能进入群

陌生人社交群（Public）类似 QQ 中的各种兴趣群和部落区，任何人都可以申请入群，但需要经过群主或管理员审批才能真正入群。陌生人社交群默认需要群主或管理员进行审批才能加群的，但群主或管理员也可以通过 setGroupInfo 接口调整加群选项（ V2TIMGroupAddOpt ），可以设置为更严格的“禁止任何人加群”，也可以设置为更宽松的“放开审批流程”。

- V2TIM_GROUP_ADD_FORBID ：禁止任何人加群。
- V2TIM_GROUP_ADD_AUTH ：需要群主或管理员审批才能加入（默认值）。
- V2TIM_GROUP_ADD_ANY ：取消审批流程，任何用户都可以加入。

需要审批才能进入群的流程如下：



1. **申请者提出加群申请**
   申请者调用 joinGroup 申请加群。
2. **群主或管理员处理加群申请**
   群主或管理员在收到加群申请的回调 onReceiveJoinApplication 后调用接口 getGroupApplicationList 获取加群申请列表，然后通过 acceptGroupApplication 或者 refuseGroupApplication 来同意或者拒绝某一条加群请求。
3. **申请者收到处理结果**
   请求加群被同意或者拒绝后，请求者会收到 V2TIMGroupListener 中的 onApplicationProcessed 回调，其中 `isAgreeJoin` 为 `true` 表示同意加群，反之被拒绝。同意加群后，全员（包括请求者）收到 onMemberEnter 回调。

退出群组

调用 quitGroup 可以退出群组，退群者会收到 onQuitFromGroup 回调，群其他成员会收到 onMemberLeave 回调。

> 注意：
> 对于陌生人社交群（Public）、临时会议群（Meeting）和直播群（AVChatRoom），群主不可以退群的，群主只能 解散群组。

解散群组

调用 dismissGroup 可以解散群组，全员会收到 onGroupDismissed 回调。

> 注意：
> - 对于陌生人社交群（Public）、临时会议群（Meeting）和直播群（AVChatRoom），群主随时可以解散群。
> - 工作群（Work）的解散最为严格，即使群主也不能随意解散，只能由您的业务服务器调用 解散群组 REST API 解散。

获取已加入的群组

调用 getJoinedGroupList 可以获取已加入的好友工作群（Work）、陌生人社交群（Public）、临时会议群（Meeting）列表，但直播群（AVChatRoom）不包含在此列表中。

群资料和群设置

获取群资料

调用 getGroupsInfo 可以获取群资料，该接口支持批量获取。您可以一次传入多个 `groupID` 获取多个群的群资料。

修改群资料

调用 setGroupInfo 可以修改群资料。群资料被修改后，全员会收到 onGroupInfoChanged 回调。

> 注意：
>
> - 好友工作群（Work）所有群成员都可以修改群基础资料。
> - 陌生人社交群（Public）、临时会议群（Meeting）只有群主或管理员可以修改群基础资料。
> - 直播群（AVChatRoom）只有群主可以修改群基础资料。

```
// 示例代码：修改群资料
V2TIMGroupInfo *info = [[V2TIMGroupInfo alloc] init];
info.groupID = @"需要修改的群 ID";
info.faceURL = @"http://xxxx";
[[V2TIMManager sharedInstance] setGroupInfo:info succ:^{
// 群资料修改成功
} fail:^(int code, NSString *msg) {
// 群资料修改失败
}];
```

设置群消息的接收选项

任何群成员都可以调用 setReceiveMessageOpt 接口修改群消息接收选项。群消息接收选项包括：

- V2TIM_GROUP_RECEIVE_MESSAGE：在线正常接收消息，离线时会有 APNs 推送。
- V2TIM_GROUP_NOT_RECEIVE_MESSAGE：不会接收到群消息。
- V2TIM_GROUP_RECEIVE_NOT_NOTIFY_MESSAGE：在线正常接收消息，离线不会有推送通知。

根据群消息接收选择可以实现群消息免打扰：

- **完全不接收群内消息**
  群消息接收选项设置为 `V2TIM_GROUP_NOT_RECEIVE_MESSAGE` 后，群内的任何消息都收不到，会话列表也不会更新。
- **接收群内消息但不提醒，在会话列表界面显示小圆点，而不显示未读数**

> 说明：
> 此方式需使用未读计数功能，因此仅适用于好友工作群（Work）和陌生人社交群（Public）。

群消息接收选项设置为 `V2TIM_GROUP_RECEIVE_NOT_NOTIFY_MESSAGE` ，当群内收到新消息，会话列表需要更新时，可以通过会话中的 unreadCount 获取到消息未读数。根据 recvOpt 判断获取到的群消息接收选项为 `V2TIM_GROUP_RECEIVE_NOT_NOTIFY_MESSAGE` 时显示小红点而不是消息未读数。

群属性（群自定义字段）

基于 API2.0 我们设计了全新的群自定义字段，我们称之为 "群属性"，其特性如下：

1. 不再需要控制台配置，客户端可以直接增删改查群属性。
2. 最多支持16个群属性，每个群属性的大小最大支持4k，所有群属性的大小最大支持16k。
3. 目前仅支持直播群（AVChatRoom）。
4. initGroupAttributes、setGroupAttributes、deleteGroupAttributes 接口合并计算， SDK 限制为5秒10次，超过后回调8511错误码；后台限制1秒5次，超过后返回10049错误码。
5. getGroupAttributes 接口 SDK 限制5秒20次。

基于群属性，我们可以做语聊房的麦位管理，当有人上麦的时候，可以设置一个群属性管理上麦人信息，当有人下麦的时候，可以删除对应群属性，其他成员可以通过获取群属性列表来展示麦位列表。

初始化群属性

调用 initGroupAttributes 接口可以初始化群属性，如果该群之前有群属性，会先清空原来的群属性。

设置群属性

调用 setGroupAttributes 接口可以设置群属性，如果设置的群属性不存在，会自动添加该群属性。

删除群属性

调用 deleteGroupAttributes 接口可以删除指定群属性，如果 keys 字段填 nil ，则会清空所有的群属性。

获取群属性

调用 getGroupAttributes 接口可以获取指定群属性，如果 keys 字段填 nil ，则会获取所有的群属性。

群属性更新

群属性有任何的更新变化，都会通过 onGroupAttributeChanged 回调出来所有的群属性字段。

群成员管理

获取群成员列表

调用 getGroupMemberList 可以获取某个群的群成员列表，该列表中包含了各个群成员的资料信息，例如用户 ID（ userID ）、群名片（ nameCard ）、头像（ faceUrl ）、昵称（ nickName ）、进群时间（ joinTime ）等信息。
一个群中的成员人数可能很多（例如5000+），群成员列表的拉取接口支持过滤器（ filter ）和分页拉取（ nextSeq ）两个高级特性。

过滤器（filter）

在调用 getGroupMemberList 接口时，您可以指定 filter 确定是否仅拉取特定角色的信息列表。

过滤器	过滤类型
V2TIM_GROUP_MEMBER_FILTER_ALL	拉取所有群成员的信息列表
V2TIM_GROUP_MEMBER_FILTER_OWNER	仅拉取群主的信息列表
V2TIM_GROUP_MEMBER_FILTER_ADMIN	仅拉取群管理员的信息列表
V2TIM_GROUP_MEMBER_FILTER_COMMON	仅拉取普通群成员的信息列表

```
//示例代码：通过 filter 参数指定只拉取群主的资料
[[V2TIMManager sharedInstance] getGroupMemberList:@"testGroup" filter:V2TIM_GROUP_MEMBER_FILTER_OWNER
nextSeq:0 succ:^(uint64_t nextSeq, NSArray<V2TIMGroupMemberInfo *> *memberList) {
// 拉取成功
} fail:^(int code, NSString *msg) {
// 拉取失败
}];
```

分页拉取（nextSeq）

很多情况下，用户界面上并不需要展示全部的群成员信息，只需展示群成员列表的第一页即可，等用户单击“下一页”或在列表页下拉刷新时，才需要拉取更多的群成员。针对此类场景，您可以使用分页拉取。

在调用 getGroupMemberList 接口时，一次最多返回50个成员，您可以通过 nextSeq 参数分页拉取成员列表， nextSeq 参数为分页拉取标志，第一次拉取时请填0。当首次拉取群成员信息成功后， getGroupMemberList 的回调结果 V2TIMGroupMemberInfoResult 中会包含 nextSeq 字段：

- 如果 nextSeq 等于0，表示已经拉取到了全部的群成员列表。
- 如果 nextSeq 大于0，表示还有更多的群成员信息可以拉取。您可以根据用户在 UI 上的操作，选择是否进行第二次接口调用以拉取更多的群成员信息。当您进行第二次拉取时，需要将上一次拉取返回的 V2TIMGroupMemberInfoResult 中的 nextSeq 作为参数，传入 getGroupMemberList 接口。

```
//示例代码：通过 nextSeq 参数进行分页拉取
[[V2TIMManager sharedInstance] getGroupMemberList:@"testGroup" filter:V2TIM_GROUP_MEMBER_FILTER_ALL nextSeq:0
succ:^(uint64_t nextSeq, NSArray<V2TIMGroupMemberInfo *> *memberList) {
// nextSeq 如果大于 0 ，继续分页拉取
if (nextSeq > 0) {
[[V2TIMManager sharedInstance] getGroupMemberList:@"testGroup" filter:V2TIM_GROUP_MEMBER_FILTER_ALL
nextSeq:nextSeq succ:^(uint64_t nextSeq, NSArray<V2TIMGroupMemberInfo *> *memberList) {
// 第二次分页拉取成功
} fail:^(int code, NSString *msg) {
// 第二次分页拉取失败
}];
}
// 第一次分页拉取成功
```

```
} fail:^(int code, NSString *msg) {
// 第一次分页拉取失败
}];
```

获取群成员资料

调用 getGroupMembersInfo 可以获取群成员资料，该接口支持批量获取，您可以一次传入多个 userID 获取多个群成员的资料，从而提升网络传输效率。

修改群成员资料

群主或者管理员可以调用 setGroupMemberInfo 接口修改群成员的群名片（ nameCard ）、 群成员角色（ role ）、禁言时间（ muteUntil ）以及自定义字段等与群成员相关的资料。

禁言

群主或管理员可以通过 muteGroupMember 禁言某一个群成员并设置禁言时间，禁言时间单位为秒，禁言信息存储于群成员的 muteUntil 属性字段中。群成员被禁言后，全员（包括被禁言的群成员）都会收到 onGroupMemberInfoChanged 事件回调。
群主或管理员也可以通过 setGroupInfo 接口对整个群进行禁言，将 allMuted 属性字段设置为 true 即可。全群禁言没有时间限制，需将群资料 allMuted 设置为 NO 才能解除禁言。

踢人

群主或管理员调用 kickGroupMember 接口可以实现踢人。由于直播群（AVChatRoom）对进群没有限制，因此直播群（AVChatRoom）没有支持踢人的接口，您可以使用 muteGroupMember 达到同样的目的。
成员被踢后，全员（包括被踢人）会收到 onMemberKicked 回调。

切换群成员角色

群主调用 setGroupMemberRole 可以对陌生人社交群（Public）或临时会议群（Meeting）中的群成员进行角色切换，可切换角色包括普通成员、管理员。

- 被设置为管理员后，全员（包括被设置的成员）会收到 onGrantAdministrator 回调。
- 被取消管理员后，全员（包括被设置的成员）会收到 onRevokeAdministrator 回调。

转让群主

群主可以调用 transferGroupOwner 把群主转让给其他群成员。
群主转让后，全员会收到 onGroupInfoChanged 回调，其中 V2TIMGroupChangeInfo 的 type 为 V2TIM_GROUP_INFO_CHANGE_TYPE_OWNER ，value 值为新群主的 userID。

常见问题

1. 直播群（AVChatRoom）中途掉线又连接上后，能否继续接收消息？

可以继续接收消息，但是直播群（AVChatRoom）中的消息不支持云端存储，因此无法拉取到掉线期间的消息。

2. 为什么群成员进群和退群收不到通知？

请确认群组类型：

- 临时会议群（Meeting）不支持群成员变更通知。
- 直播群（AVChatRoom）消息限制40条/秒，会优先保证高优先级消息的收发，超过限制后会优先丢弃低优先级的消息。

3. 为什么会议群（Meeting） 中的未读数一直为零?

临时会议群（Meeting）和直播群（AVChatRoom）分别配合会议和直播的音视频场景，因此这两类群组均不支持未读消息计数。

群组管理（Web & 小程序）

最近更新时间：2020-08-20 17:46:04

群组综述

即时通信 IM 有多种群组类型，其特点以及限制因素可参考 群组系统。群组使用唯一 ID 标识，通过群组 ID 可以进行不同操作。

群组管理

获取加入的群组列表

需要渲染或刷新【我的群组列表】时，调用该接口获取群组列表，更多详情请参见 Group。

> 注意：
> 接口返回的群组列表，不包含 TIM.TYPES.GRP_AVCHATROOM（直播群）类型的群组

接口名

```
tim.getGroupList();
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	属性	描述
groupProfileFilter	Array<String>	<optional>	群资料过滤器。除默认拉取的群资料外，指定需要额外拉取的群资料，支持的值如下： TIM.TYPES.GRP_PROFILE_OWNER_ID：群主 ID TIM.TYPES.GRP_PROFILE_CREATE_TIME：群创建时间 TIM.TYPES.GRP_PROFILE_LAST_INFO_TIME：最后一次群资料变更时间 TIM.TYPES.GRP_PROFILE_MEMBER_NUM：群成员数量 TIM.TYPES.GRP_PROFILE_MAX_MEMBER_NUM：最大群成员数量 TIM.TYPES.GRP_PROFILE_JOIN_OPTION：申请加群选项 TIM.TYPES.GRP_PROFILE_INTRODUCTION：群介绍 TIM.TYPES.GRP_PROFILE_NOTIFICATION：群公告 TIM.TYPES.GRP_PROFILE_MUTE_ALL_MBRS (全体禁言设置) v2.6.2起支持

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data.groupList 中获取群组列表。
- catch 的回调函数参数为 IMError。

示例

- 默认拉取：

```
// 该接口默认只拉取这些资料：群类型、群名称、群头像以及最后一条消息的时间。
let promise = tim.getGroupList();
promise.then(function(imResponse) {
console.log(imResponse.data.groupList); // 群组列表
}).catch(function(imError) {
console.warn('getGroupList error:', imError); // 获取群组列表失败的相关信息
});
```

- 拉取其他资料：

```
// 若默认拉取的字段不满足需求，可以参考下述代码，拉取额外的资料字段。
let promise = tim.getGroupList({
groupProfileFilter: [TIM.TYPES.GRP_PROFILE_OWNER_ID],
});
```

```
promise.then(function(imResponse) {
console.log(imResponse.data.groupList); // 群组列表
}).catch(function(imError) {
console.warn('getGroupList error:', imError); // 获取群组列表失败的相关信息
});
```

获取群详细资料

更多详情请参见 Group。

接口名

```
tim.getGroupProfile(options);
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	属性	描述
groupID	String	-	群组 ID
groupCustomFieldFilter	Array<String>	<optional>	群组的自定义字段过滤器，指定需要获取的群组的自定义字段，详情请参见 自定义字段

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data.group 中获取群组资料。
- catch 的回调函数参数为 IMError。

示例

```
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); // 获取群详细资料失败的相关信息
});
```

创建群组

更多详情请参见 Group。

> 注意：
> 该接口创建 TIM.TYPES.GRP_AVCHATROOM（直播群）后，需调用 joinGroup 接口加入群组后，才能进行消息收发流程。

接口名

```
tim.createGroup(options);
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	属性	默认值	描述
name	String	-	-	必填，群组名称，最长30字节

名称	类型	属性	默认值	描述
type	String	<optional>	TIM.TYPES.GRP_WORK	群组类型，包括： • TIM.TYPES.GRP_WORK：好友工作群，默认 • TIM.TYPES.GRP_PUBLIC：陌生人社交群 • TIM.TYPES.GRP_MEETING：临时会议群 • TIM.TYPES.GRP_AVCHATROOM：直播群
groupID	String	<optional>	-	群组 ID。不填该字段时，会自动为群组创建一个唯一的群 ID
introduction	String	<optional>	-	群简介，最长240字节
notification	String	<optional>	-	群公告，最长300字节
avatar	String	<optional>	-	群头像 URL，最长100字节
maxMemberNum	Number	<optional>	-	最大群成员数量，缺省时的默认值：好友工作群是6000，陌生人社交群是6000，临时会议群是6000，直播群无限制
joinOption	String	<optional>	TIM.TYPES.JOIN_OPTIONS_FREE_ACCESS	申请加群处理方式。**创建好友工作群/临时会议群/直播群时不能填写该字段。**好友工作群该字段固定为：禁止申请加群，临时会议群和直播群该字段固定为：自由加入 • TIM.TYPES.JOIN_OPTIONS_FREE_ACCESS：自由加入 • TIM.TYPES.JOIN_OPTIONS_NEED_PERMISSION：需要验证 • TIM.TYPES.JOIN_OPTIONS_DISABLE_APPLY：禁止加群 创建 TIM.TYPES.GRP_WORK, TIM.TYPES.GRP_MEETING, TIM.TYPES.GRP_AVCHATROOM 类型的群组不能填写该字段。好友工作群禁止申请加群，临时会议群和直播群自由加入。
memberList	Array<Object>	<optional>	-	初始群成员列表，最多500个。创建直播群时不能添加成员。详情请参见下方 memberList 参数说明
groupCustomField	Array<Object>	<optional>	-	群组维度的自定义字段，默认没有自定义字段，如需开通请参见 群成员资料

memberList 参数说明

名称	类型	属性	描述
userID	String	-	必填，群成员的 UserID
role	String	<optional>	成员身份，可选值只有 Admin，表示添加该成员并设置为管理员
memberCustomField	Array<Object>	<optional>	群成员维度的自定义字段，默认没有自定义字段，如需开通请参见 自定义字段

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data.group 中获取群组资料。
- catch 的回调函数参数为 IMError。

示例

```
// 创建好友工作群
let promise = tim.createGroup({
type: TIM.TYPES.GRP_WORK,
```

```
name: 'WebSDK',
memberList: [{userID: 'user1'}, {userID: 'user2'}] // 如果填写了 memberList，则必须填写 userID
});
promise.then(function(imResponse) { // 创建成功
console.log(imResponse.data.group); // 创建的群的资料
}).catch(function(imError) {
console.warn('createGroup error:', imError); // 创建群组失败的相关信息
});
```

解散群组

群主可调用该接口解散群组。

> 注意：
> 群主不能解散好友工作群。

接口名

```
tim.dismissGroup(groupID);
```

请求参数

名称	类型	描述
groupID	String	群组 ID

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data.groupID 中获取被解散的群组 ID。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.dismissGroup('group1');
promise.then(function(imResponse) { // 解散成功
console.log(imResponse.data.groupID); // 被解散的群组 ID
}).catch(function(imError) {
console.warn('dismissGroup error:', imError); // 解散群组失败的相关信息
});
```

更新群组资料

接口名

```
tim.updateGroupProfile(options);
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	属性	Default	描述
groupID	Object	-	-	群组 ID
name	Object	<optional>	-	群名称，最长30字节
avatar	Object	<optional>	-	群头像 URL，最长100字节
introduction	Object	<optional>	-	群简介，最长240字节
notification	Object	<optional>	-	群公告，最长300字节
maxMemberNum	Number	<optional>	-	最大群成员数量，最大为6000

名称	类型	属性	Default	描述
muteAllMembers	Boolean	-	-	设置全体禁言，true 表示全体禁言，false 表示取消全体禁言，v2.6.2 起支持
joinOption	String	<optional>	TIM.TYPES.JOIN_OPTIONS_FREE_ACCESS	申请加群处理方式 • TIM.TYPES.JOIN_OPTIONS_FREE_ACCESS：自由加入 • TIM.TYPES.JOIN_OPTIONS_NEED_PERMISSION：需要验证 • TIM.TYPES.JOIN_OPTIONS_DISABLE_APPLY：禁止加群 !TIM.TYPES.GRP_WORK, TIM.TYPES.GRP_MEETING, TIM.TYPES.GRP_AVCHATROOM 类型群组的该属性不允许修改。好友工作群禁止申请加群，临时会议群和直播群自由加入。
groupCustomField	Array<Object>	<optional>	-	群自定义字段，详情请参见下方 groupCustomField 参数说明 默认没有自定义字段，如需开通请参见 自定义字段

groupCustomField 参数说明

名称	类型	描述
key	String	自定义字段的 Key
value	String	自定义字段的 Value

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data.group 中获取修改后的群组资料。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.updateGroupProfile({
groupID: 'group1',
name: 'new name', // 修改群名称
introduction: 'this is introduction.', // 修改群公告
// v2.6.0 起，群成员能收到群自定义字段变更的群提示消息，且能获取到相关的内容，详见 Message.payload.newGroupProfile.groupCustomField
groupCustomField: [{ key: 'group_level', value: 'high'}] // 修改群组维度自定义字段
});
promise.then(function(imResponse) {
console.log(imResponse.data.group) // 修改成功后的群组详细资料
}).catch(function(imError) {
console.warn('updateGroupProfile error:', imError); // 修改群组资料失败的相关信息
});
```

申请加群

申请加群的接口，申请加入某个群组时调用。

注意：

- 好友工作群不允许申请加群，只能通过 addGroupMember 方式添加。
- TIM.TYPES.GRP_AVCHATROOM（直播群）有两种加群方式：
  - 正常加群，即登录加群。此时 SDK 内的所有接口都能正常调用。
  - 匿名加群，即不登录加群。此时只能收消息，其他任何需要鉴权的接口都不能调用。

- 只有 TIM.TYPES.GRP_AVCHATROOM（直播群）支持匿名加群，其他类型的群组不支持。
- 同一用户同时只能加入一个直播群。【例如】用户已在直播群 A 中，再加入直播群 B，SDK 会先退出直播群 A，然后加入直播群 B。

接口名

```
tim.joinGroup(options);
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	属性	描述
groupID	String	-	-
applyMessage	String	-	附言
type	String	<optional>	待加入的群组的类型，加入直播群时该字段必填。可选值： • TIM.TYPES.GRP_PUBLIC：陌生人社交群 • TIM.TYPES.GRP_MEETING：临时会议群 • TIM.TYPES.GRP_AVCHATROOM：直播群

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，IMResponse.data 中包括的属性值如下表所示：

名称	含义
status	加群的状态。包括： ○ TIM.TYPES.JOIN_STATUS_WAIT_APPROVAL：等待管理员审核 ○ TIM.TYPES.JOIN_STATUS_SUCCESS：加群成功 ○ TIM.TYPES.JOIN_STATUS_ALREADY_IN_GROUP：已在群中
group	加群成功后的群组资料

- catch 的回调函数参数为 IMError。

示例

```
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; // 等待管理员同意
case TIM.TYPES.JOIN_STATUS_SUCCESS: // 加群成功
console.log(imResponse.data.group); // 加入的群组资料
break;
case TIM.TYPES.JOIN_STATUS_ALREADY_IN_GROUP: // 已经在群中
break;
default: break;
}
}).catch(function(imError){
console.warn('joinGroup error:', imError); // 申请加群失败的相关信息
});
```

退出群组

群主只能退出好友工作群，退出后该好友工作群无群主。

接口名

```
tim.quitGroup(groupID);
```

请求参数

名称	类型	描述
groupID	String	群组 ID

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，IMResponse.data.groupID 为退出的群组 ID。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.quitGroup('group1');
promise.then(function(imResponse) {
console.log(imResponse.data.groupID); // 退出成功的群 ID
}).catch(function(imError){
console.warn('quitGroup error:', imError); // 退出群组失败的相关信息
});
```

根据群 ID 搜索群组

通过 groupID 搜索群组。

> 注意：TIM.TYPES.GRP_WORK 类型的群组（好友工作群）不能被搜索。

接口名

```
tim.searchGroupByID(groupID);
```

请求参数

名称	类型	描述
groupID	String	群组 ID

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，IMResponse.data.group 为群组资料。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.searchGroupByID('group1');
promise.then(function(imResponse) {
const group = imResponse.data.group; // 群组信息
}).catch(function(imError) {
console.warn('searchGroupByID error:', imError); // 搜索群组失败的相关信息
});
```

转让群组

转让群组。只有群主有权限操作。

> 注意：只有群主拥有转让的权限。TIM.TYPES.GRP_AVCHATROOM（直播群）类型的群组不能转让。

接口名

```
tim.changeGroupOwner(options);
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	描述
groupID	String	待转让的群组 ID
newOwnerID	String	新群主的 ID

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，IMResponse.data.group 为群组资料。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.changeGroupOwner({
groupID: 'group1',
newOwnerID: 'user2'
});
promise.then(function(imResponse) { // 转让成功
console.log(imResponse.data.group); // 群组资料
}).catch(function(imError) { // 转让失败
console.warn('changeGroupOwner error:', imError); // 转让群组失败的相关信息
});
```

处理加群申请

处理申请加群（同意或拒绝）

> 注意：
> 如果一个群有多位管理员，当有人申请加群时，所有在线的管理员都会收到 申请加群的群系统通知。如果某位管理员处理了这个申请（同意或者拒绝），则其他管理员无法重复处理（即不能修改处理的结果）。

接口名

```
tim.handleGroupApplication(options);
```

请求参数

参数 options 为 Object 类型，包含的属性值如下：

名称	类型	属性	描述
handleAction	String	-	处理结果 Agree（同意）/ Reject（拒绝）
handleMessage	String	<optional>	附言
message	Message	-	申请加群的【群系统通知消息】的消息实例。该实例可通过以下方式获取： • 收到新的群系统通知事件 的回调参数中获取 • 系统类型会话的消息列表中获取

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，IMResponse.data.group 为群组资料。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.handleGroupApplication({
handleAction: 'Agree',
handleMessage: '欢迎欢迎',
message: message // 申请加群群系统通知的消息实例
});
promise.then(function(imResponse) {
console.log(imResponse.data.group); // 群组资料
}).catch(function(imError){
console.warn('handleGroupApplication error:', imError); // 错误信息
});
```

设置群消息提示类型

接口名

```
tim.setMessageRemindType(options);
```

请求参数

参数 options 为 Object 类型，包含的属性值如下：

名称	类型	描述
groupID	String	群组 ID
messageRemindType	String	群消息提示类型。详细如下： • TIM.TYPES.MSG_REMIND_ACPT_AND_NOTE：SDK 接收消息并抛出 收到消息事件 通知接入侧，接入侧做提示 • TIM.TYPES.MSG_REMIND_ACPT_NOT_NOTE：SDK 接收消息并抛出 收到消息事件 通知接入侧，接入侧不做提示 • TIM.TYPES.MSG_REMIND_DISCARD：SDK 拒收消息，不会抛出 收到新消息事件

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse， IMResponse.data.group 为群组资料。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.setMessageRemindType({ groupID: 'group1', messageRemindType: TIM.TYPES.MSG_REMIND_DISCARD }); // 拒收消息
promise.then(function(imResponse) {
console.log(imResponse.data.group); // 设置后的群资料
}).catch(function(imError) {
console.warn('setMessageRemindType error:', imError);
});
```

群成员管理

获取群成员列表

注意：

- 从v2.6.2版本开始，该接口支持拉取群成员禁言截止时间戳（muteUntil），接入侧可根据此值判断群成员是否被禁言，以及禁言的剩余时间。
- 低于v2.6.2版本时，该接口获取的群成员列表中的资料仅包括头像、昵称等，能够满足群成员列表的渲染需求。如需查询群成员禁言截止时间戳（muteUntil）等详细资料，请使用 getGroupMemberProfile。
- 该接口是分页拉取群成员，不能直接用于获取群的总人数。获取群的总人数（memberNum）请使用 getGroupProfile 。

更多详情请参见 GroupMember。

接口名

```
tim.getGroupMemberList(options);
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	属性	默认值	描述
groupID	String	-	-	群组的 ID
count	Number	<optional>	15	需要拉取的数量。最大值为100，避免回包过大导致请求失败。若传入超过100，则只拉取前100个
offset	Number	<optional>	0	偏移量，默认从0开始拉取

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse， IMResponse.data.memberList 为群成员列表，请参考 GroupMember。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.getGroupMemberList({ groupID: 'group1', count: 30, offset:0 }); // 从0开始拉取30个群成员
promise.then(function(imResponse) {
console.log(imResponse.data.memberList); // 群成员列表
}).catch(function(imError) {
console.warn('getGroupMemberList error:', imError);
});
// 从v2.6.2 起，该接口支持拉取群成员禁言截止时间戳。
let promise = tim.getGroupMemberList({ groupID: 'group1', count: 30, offset:0 }); // 从0开始拉取30个群成员
promise.then(function(imResponse) {
console.log(imResponse.data.memberList); // 群成员列表
for (let groupMember of imResponse.data.memberList) {
if (groupMember.muteUntil * 1000 > Date.now()) {
console.log(`${groupMember.userID} 禁言中`);
} else {
console.log(`${groupMember.userID} 未被禁言`);
}
}
}).catch(function(imError) {
console.warn('getGroupMemberProfile error:', imError);
});
```

获取群成员资料

注意：

- 使用该接口前，需要将 SDK 版本升级至v2.2.0或以上。
- 每次查询的用户数上限为50。如果传入的数组长度大于50，则只取前50个用户进行查询，其余丢弃。

更多详情请参见 GroupMember。

接口名

```
tim.getGroupMemberProfile(options);
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	属性	描述
groupID	String	-	群组的 ID

名称	类型	属性	描述
userIDList	Array.<String>	-	要查询的群成员用户 ID 列表
memberCustomFieldFilter	Array.<String>	<optional>	群成员自定义字段筛选。可选，若不填，则默认查询所有群成员自定义字段

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse， IMResponse.data.memberList 为查询成功的群成员列表，请参考 GroupMember。
- catch 的回调函数参数为 IMError。

添加群成员

详细规则如下：

- TIM.TYPES.GRP_WORK 好友工作群：任何群成员都可邀请他人加群，且无需被邀请人同意，直接将其拉入群组中。
- TIM.TYPES.GRP_PUBLIC 陌生人社交群/ TIM.TYPES.GRP_MEETING 临时会议群：只有 App 管理员可以邀请他人入群，且无需被邀请人同意，直接将其拉入群组中。
- TIM.TYPES.GRP_AVCHATROOM 直播群：不允许任何人邀请他人入群（包括 App 管理员）。

更多详情请参见 GroupGroupMember 和 加群方式差异。

接口名

```
tim.addGroupMember(options);
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	描述
groupID	String	群组 ID
userIDList	Array<String>	待添加的群成员 ID 数组。单次最多添加300个成员

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse， IMResponse.data 属性值如下表所示：

名称	类型	含义
successUserIDList	Array<String>	添加成功的 userID 列表
failureUserIDList	Array<String>	添加失败的 userID 列表
existedUserIDList	Array<String>	已在群中的 userID 列表
group	Group	接口调用后的群组资料

- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.addGroupMember({
groupID: 'group1',
userIDList: ['user1','user2','user3']
});
promise.then(function(imResponse) {
console.log(imResponse.data.successUserIDList); // 添加成功的群成员 userIDList
console.log(imResponse.data.failureUserIDList); // 添加失败的群成员 userIDList
console.log(imResponse.data.existedUserIDList); // 已在群中的群成员 userIDList
console.log(imResponse.data.group); // 添加后的群组信息
}).catch(function(imError) {
```

```
console.warn('addGroupMember error:', imError); // 错误信息
});
```

删除群成员

删除群成员。群主可移除群成员。

接口名

```
tim.deleteGroupMember(options)
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	描述
groupID	String	群组 ID
userIDList	Array<String>	待删除的群成员的 ID 列表
reason	String	踢人的原因，可选参数

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse， IMResponse.data.group 为更新后的群组资料。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.deleteGroupMember({groupID: 'group1', userIDList:['user1'], reason: '你违规了，我要踢你！'});
promise.then(function(imResponse) {
console.log(imResponse.data.group); // 删除后的群组信息
console.log(imResponse.data.userIDList); // 被删除的群成员的 userID 列表
}).catch(function(imError) {
console.warn('deleteGroupMember error:', imError); // 错误信息
});
```

禁言或取消禁言

设置群成员的禁言时间，可以禁言群成员，也可取消禁言。TIM.TYPES.GRP_WORK 类型的群组（即好友工作群）不能禁言。

> 说明：
> 只有群主和管理员拥有该操作权限：
>
> - 群主可以禁言/取消禁言管理员和普通群成员。
> - 管理员可以禁言/取消禁言普通群成员。

接口名

```
tim.setGroupMemberMuteTime(options)
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	描述
groupID	String	群组 ID
userID	String	群成员 ID
muteTime	Number	禁言时长，单位秒 例如，设置该值为1000，则表示即刻起禁言该用户1000秒，设置为0，则表示取消禁言

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，IMResponse.data.group 是修改后的群资料。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.setGroupMemberMuteTime({
groupID: 'group1',
userID: 'user1',
muteTime: 600 // 禁言10分钟；设为0，则表示取消禁言
});
promise.then(function(imResponse) {
console.log(imResponse.data.group); // 修改后的群资料
console.log(imResponse.data.member); // 修改后的群成员资料
}).catch(function(imError) {
console.warn('setGroupMemberMuteTime error:', imError); // 禁言失败的相关信息
});
```

设为管理员或撤销管理员

修改群成员角色，只有群主拥有操作权限。

接口名

```
tim.setGroupMemberRole(options)
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	描述
groupID	String	群组 ID
userID	String	群成员 ID
role	String	可选值：TIM.TYPES.GRP_MBR_ROLE_ADMIN（群管理员）或 TIM.TYPES.GRP_MBR_ROLE_MEMBER（群普通成员）

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，IMResponse.data.group 是修改后的群资料。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.setGroupMemberRole({
groupID: 'group1',
userID: 'user1',
role: TIM.TYPES.GRP_MBR_ROLE_ADMIN // 将群 ID: group1 中的用户：user1 设为管理员
});
promise.then(function(imResponse) {
console.log(imResponse.data.group); // 修改后的群资料
console.log(imResponse.data.member); // 修改后的群成员资料
}).catch(function(imError) {
console.warn('setGroupMemberRole error:', imError); // 错误信息
});
```

修改群名片

设置群成员名片。

- 群主：可设置所有群成员的名片。

- 管理员：可设置自身和其他普通群成员的群名片。
- 普通群成员：只能设置自身群名片。

接口名

```
tim.setGroupMemberNameCard(options)
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	描述
groupID	String	群组 ID
userID	String<optional>	可选，默认修改自身的群名片
nameCard	String	-

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse， IMResponse.data.group 是修改后的群资料。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.setGroupMemberNameCard({ groupID: 'group1', userID: 'user1', nameCard: '用户名片' });
promise.then(function(imResponse) {
console.log(imResponse.data.group); // 设置后的群资料
console.log(imResponse.data.member); // 修改后的群成员资料
}).catch(function(imError) {
console.warn('setGroupMemberNameCard error:', imError); // 设置群成员名片失败的相关信息
});
```

修改自定义字段

设置群成员自定义字段。

> 注意：
> 普通群成员只能设置自己的自定义字段。

接口名

```
tim.setGroupMemberCustomField(options)
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	描述
groupID	String	群组 ID
userID	String<optional>	群成员 ID，可选，不填则修改自己的群成员自定义字段
memberCustomField	Array<Object>	群成员自定义字段

memberCustomField 包含的属性值如下表所示：

名称	类型	描述
key	String	自定义字段的 Key

名称	类型	描述
value	String<optional>	自定义字段的 Value

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，IMResponse.data.group 是修改后的群资料。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.setGroupMemberCustomField({ groupID: 'group1', memberCustomField: [{key: 'group_member_test', value: 'test'}]});
promise.then(function(imResponse) {
console.log(imResponse.data.group); // 设置后的群资料
console.log(imResponse.data.member); // 修改后的群成员资料
}).catch(function(imError) {
console.warn('setGroupMemberCustomField error:', imError); // 设置群成员自定义字段失败的相关信息
});
```

群提示消息

当有用户被邀请加入群组或有用户被移出群组等事件发生时，群内会产生提示消息，接入侧可以根据需要展示给群组用户，或者忽略。
群提示消息有多种类型，详细描述请参见 Message.GroupTipPayload。

名称	类型	描述
operatorID	String	执行该操作的用户 ID
operationType	Number	操作类型
userIDList	Array<String>	相关的 userID 列表
newGroupProfile	Object	若是群资料变更，该字段存放变更的群资料

群提示消息的 content 结构。系统会在恰当的时机，向全体群成员发出群提示消息。例如：有群成员退群/进群，系统会给所有群成员发对应的群提示消息。

群系统通知

当有用户申请加群等事件发生时，管理员会收到申请加群等系统消息。管理员同意或拒绝加群申请，IM SDK 会将相应的消息通过群系统通知消息发送给接入侧，由接入侧展示给用户。
群系统通知消息有多种类型，详细描述请参见 群系统通知类型常量及含义。

```
let onGroupSystemNoticeReceived = function(event) {
  const type = event.data.type; // 群系统通知的类型，详情请参见 Message.GroupSystemNoticePayload
  const message = event.data.message; // 群系统通知的消息实例，详情请参见 Message
  console.log(message.payload); // 消息内容. 群系统通知 payload 结构描述
};
tim.on(TIM.EVENT.GROUP_SYSTEM_NOTICE_RECEIVED, onGroupSystemNoticeReceived);
```

名称	类型	描述
operatorID	String	执行该操作的用户 ID
operationType	Number	操作类型
groupProfile	Object	相关的群组资料
handleMessage	Object	处理的附言 例如，user1 申请加入进群需要验证的 group1 时，若 user1 填写了申请加群的附言，则 group1 的管理员会在相应群系统通知中看到该字段

operationType 描述

名称	描述	接收对象
1	有用户申请加群	群管理员/群主接收
2	申请加群被同意	申请加群的用户接收
3	申请加群被拒绝	申请加群的用户接收
4	被踢出群组	被踢出的用户接收
5	群组被解散	全体群成员接收
6	创建群组	创建者接收
7	邀请加群	被邀请者接收
8	退群	退群者接收
9	设置管理员	被设置方接收
10	取消管理员	被取消方接收
255	用户自定义通知	默认全员接收

群系统通知的 content 结构。系统会在恰当的时机，向特定用户发出群系统通知。例如，user1 被踢出群组，系统会给 user1 发送对应的群系统消息。

信令相关
信令管理（Android）

最近更新时间：2020-07-08 16:50:16

概述

信令接口是基于 IM 消息提供的一套邀请流程控制的接口，可以实现多种实时场景，例如：

- 直播聊天室中进行上麦、下麦管理。
- 聊天场景中实现类似微信中的音视频通话功能。
- 教育场景中老师邀请同学们举手、发言的流程控制。

功能

信令接口支持以下功能：

单聊邀请

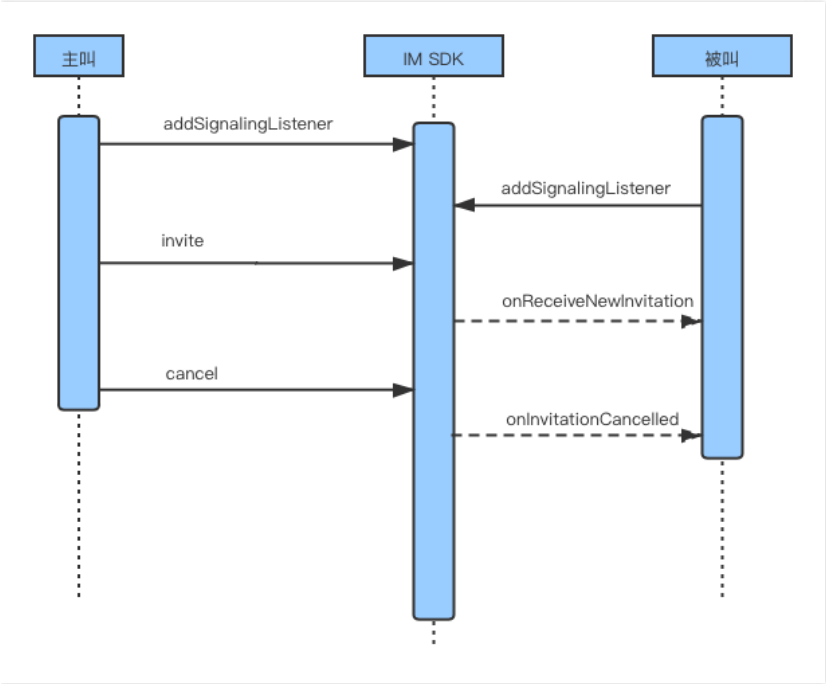
在使用 简单收发消息接口 或 富媒体消息接口 进行单聊的同时，可以使用 invite 信令接口进行点对点呼叫，对方收到邀请通知 onReceiveNewInvitation 后可以选择接受、拒绝或者等待超时。

群聊邀请

首先需通过 建群、加群、退群、解散群以及群资料 和 群成员 相关接口完成对群组的管理，并监听群内的相关事件回调 V2TIMGroupListener。然后群成员可以在群内发起群呼叫邀请 inviteInGroup，被邀请的群成员会收到邀请通知 onReceiveNewInvitation 后可以选择接受、拒绝或者等待超时。

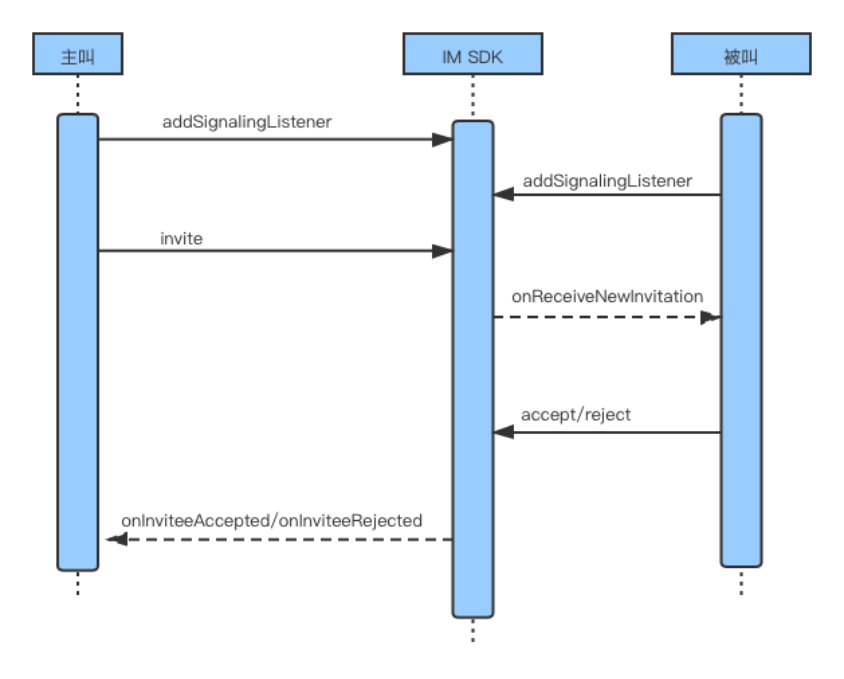
取消邀请

主叫可以在超时前且被叫未处理前取消邀请 cancel。被邀请者会收到取消通知 onInvitationCancelled，该邀请流程结束。



接受邀请

被叫收到邀请通知 onReceiveNewInvitation 后可以在超时前且主叫取消前接受邀请 accept，主叫会收到接受邀请通知 onInviteeAccepted，所有被叫处理完后（包括接受、拒绝、超时）该邀请流程结束。

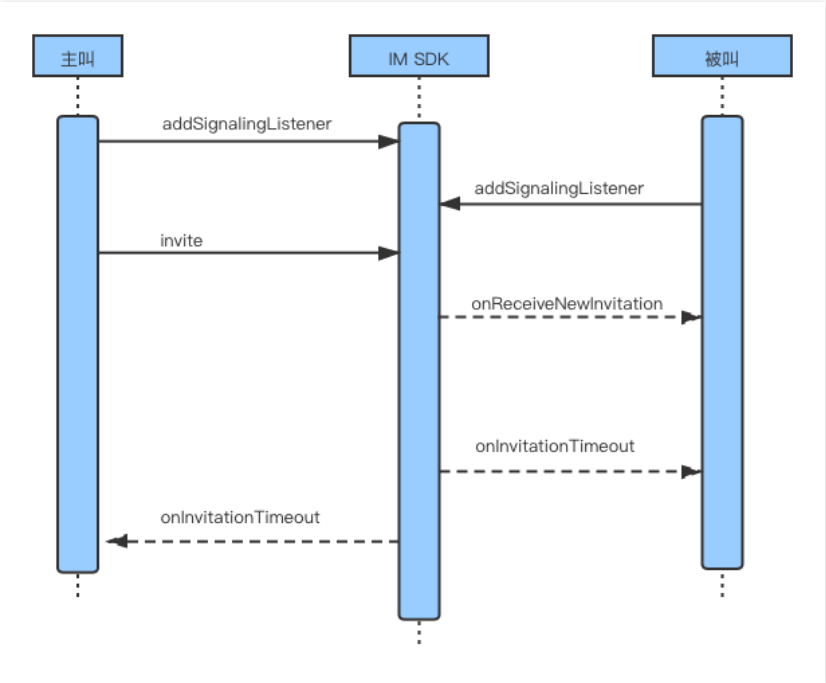


拒绝邀请

被叫收到邀请通知 onReceiveNewInvitation后可以在超时前且主叫取消前拒绝邀请 reject，主叫会收到拒绝邀请通知 onInviteeRejected，所有被叫处理完后（包括接受、拒绝、超时）该邀请流程结束。

邀请超时

若邀请接口的超时时间大于0，且被叫未在超时时间之内响应则邀请超时，主叫和被叫都会收到超时通知 onInvitationTimeout，所有被叫处理完后（包括接受、拒绝、超时）该邀请流程结束。若邀请接口的超时时间等于0，则不会有超时通知。



应用场景案例

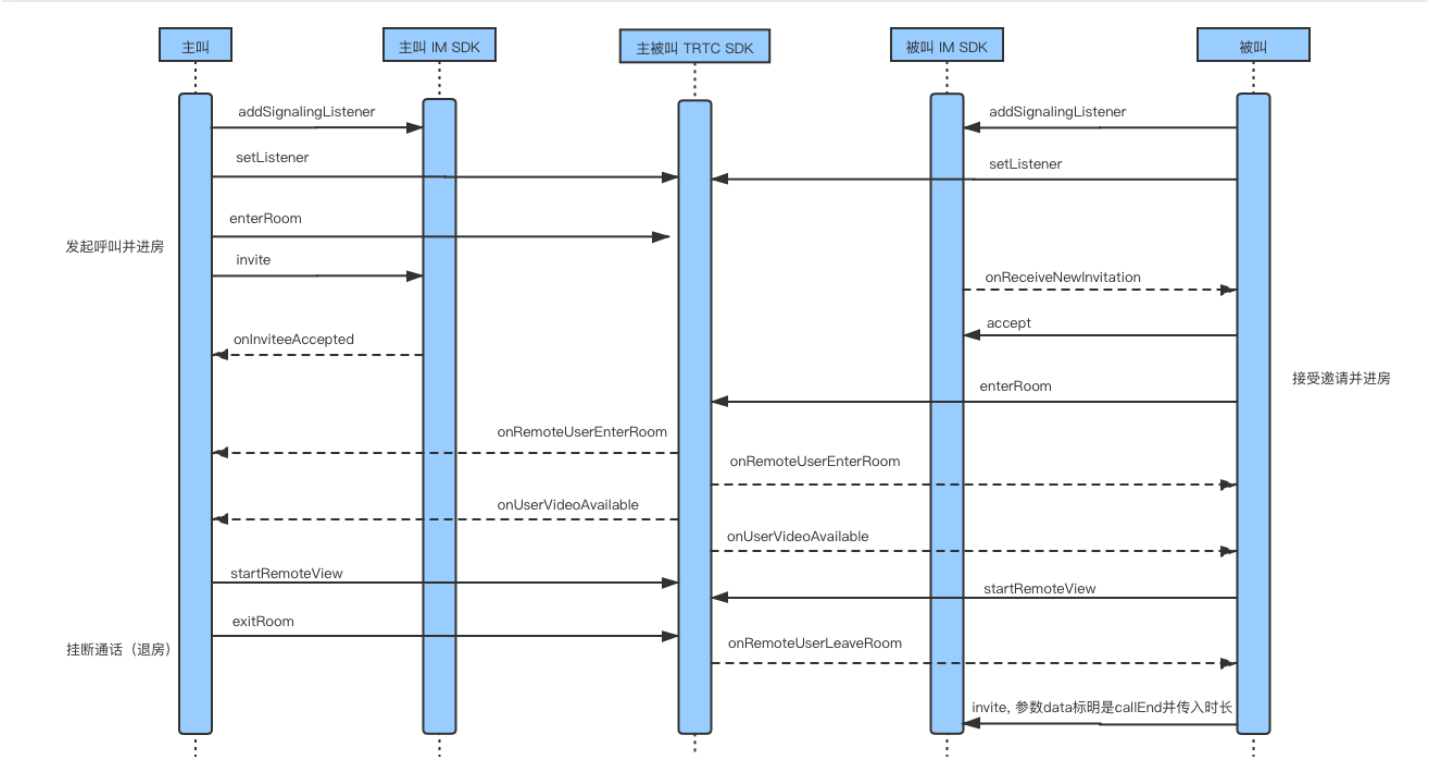
音视频通话

在开源项目 TUIKit Demo 中，我们基于 TRTC 组件 并对其稍作修改提供了一个适合聊天场景的1v1和多人音视频通话的方案，您可以直接基于我们提供的 Demo 进行修改适配。我们以1v1视频通话为例介绍下信令接口跟 TRTC SDK 的结合使用。

1v1视频通话的流程：

1. 主叫根据业务层生成的 roomID 进入该 TRTC 房间，同时调用信令邀请接口 invite 发起音视频通话请求，并把 roomID 放到邀请接口的自定义字段中。
2. 被叫收到信令邀请通知 onReceiveNewInvitation，并通过自定义数据拿到 roomID，界面开始响铃。
3. 被叫处理邀请通知：
   - 接受邀请需调用信令 accept 接口，并根据 roomID 进入到 TRTC 房间，并同时调用 openCamera() 函数打开自己本地的摄像头，双方收到 TRTC SDK 的 onRemoteUserEnterRoom 回调后记录本次通话的开始时间。
   - 拒绝邀请需调用信令 reject 接口结束本次通话。
   - 如果被叫正在跟其他人通话，则调用信令 reject 接口拒绝本次邀请，并在自定义数据中告诉对方是由于本地线路忙而拒绝。
4. 接听并当双方的音视频通道建立完成后，通话的双方都会接收到 TRTC SDK 的 onUserVideoAvailable 的事件通知，表示对方的视频画面已经拿到。此时双方用户均可以调用 TRTC SDK 接口 startRemoteView 展示远端的视频画面。远端的声音默认是自动播放的。
5. 通话结束即某一方挂断电话，该用户退出 TRTC 房间。对方收到 TRTC SDK 的 onRemoteUserLeaveRoom 回调后计算通话总时长并再次发起一次邀请，此邀请的自定义数据中标明是结束通话并附带通话时长，方便 UI 界面做展示。

时序图



教育场景中老师邀请学生举手发言

该场景为老师先让同学们举手，再从举手的同学中选一个同学进行发言。详细流程如下：

1. 老师调用 inviteInGroup 接口邀请同学们举手，自定义 data 中填入“举手操作”，同学们收到 onReceiveNewInvitation 回调。
2. 同学们根据 onReceiveNewInvitation 中的 inviteeList 和 data 字段判断被邀请者里有自己且是举手操作，那么调用 accept 接口举手。
3. 如果有学生举手，所有人都可以收到 onInviteeAccepted 回调，判断 data 中的字段为“举手操作”，展示举手学生列表。
4. 老师从举手成员列表中邀请某个同学进行发言，调用 inviteInGroup 接口，此时自定义 data 中填入“发言操作”，学生们都收到 onReceiveNewInvitation 回调。
5. 学生根据 onReceiveNewInvitation 回调中的 inviteeList 和 data 字段判断被邀请者里有自己且是发言操作，则调用 accept 接口发言。
6. 如果有学生发言，所有人都可以收到 onInviteeAccepted 回调，判断 data 中的字段为“发言操作”，展示发言成员列表。

信令管理（iOS）

最近更新时间：2020-07-08 16:50:23

概述

信令接口是基于 IM 消息提供的一套邀请流程控制的接口，可以实现多种实时场景，例如：

- 直播聊天室中进行上麦、下麦管理。
- 聊天场景中实现类似微信中的音视频通话功能。
- 教育场景中老师邀请同学们举手、发言的流程控制。

功能

信令接口支持以下功能：

单聊邀请

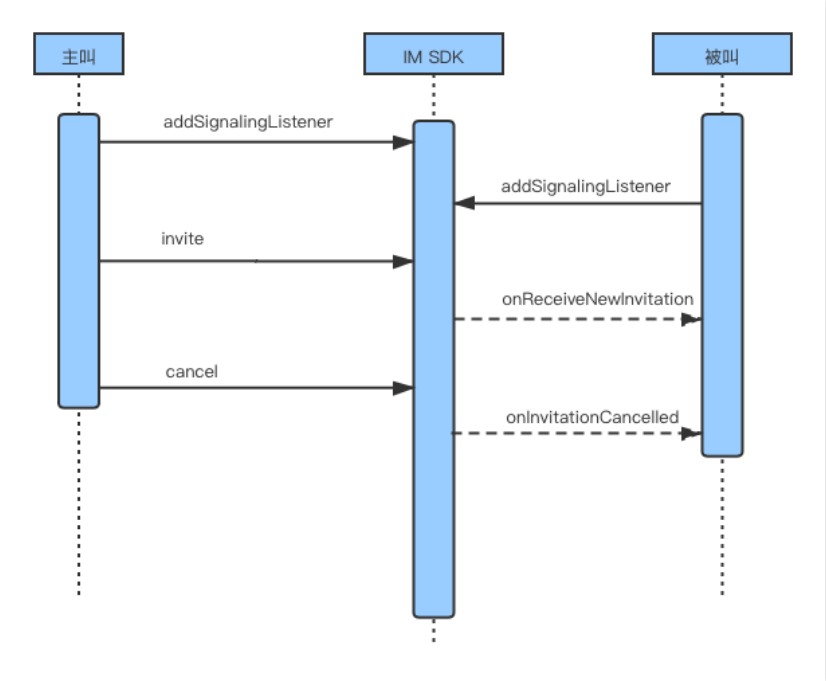
在使用 简单收发消息接口 或 富媒体消息接口 进行单聊的同时，可以使用 invite 信令接口进行点对点呼叫，对方收到邀请通知 onReceiveNewInvitation 后可以选择接受、拒绝或等待超时。

群聊邀请

首先需通过 建群、加群、退群、解散群以及群资料 和 群成员 相关接口完成对群组的管理，并监听群内的相关事件回调 V2TIMGroupListener。然后群成员可以在群内发起群呼叫邀请 inviteInGroup，被邀请的群成员会收到邀请通知 onReceiveNewInvitation 后可以选择接受、拒绝或等待超时。

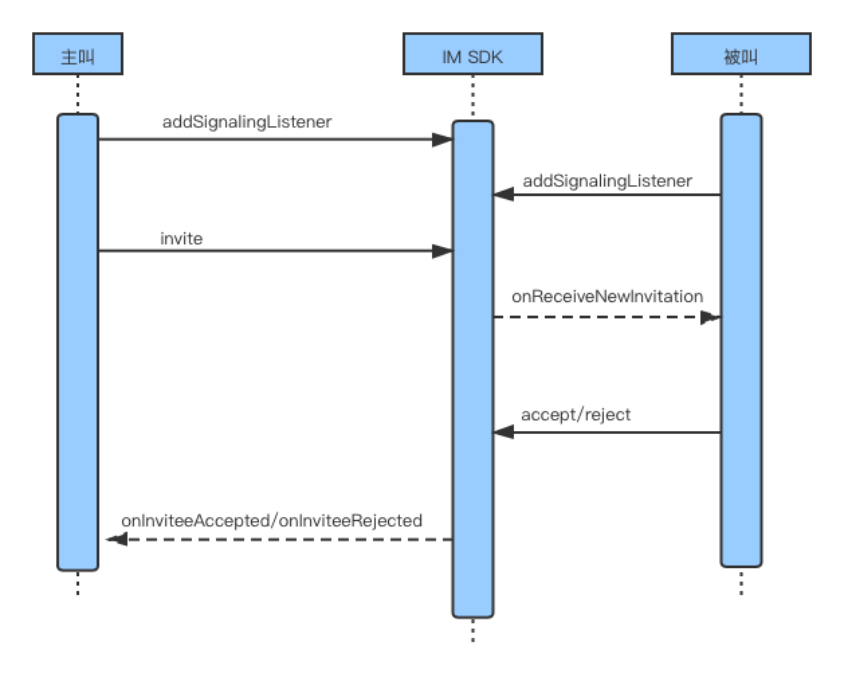
取消邀请

主叫可以在超时前且被叫未处理前取消邀请 cancel。被邀请者会收到取消通知 onInvitationCancelled，该邀请流程结束。



接受邀请

被叫收到邀请通知 onReceiveNewInvitation 后可以在超时前且主叫取消前接受邀请 accept，主叫会收到接受邀请通知 onInviteeAccepted，所有被叫处理完后（包括接受、拒绝、超时）该邀请流程结束。

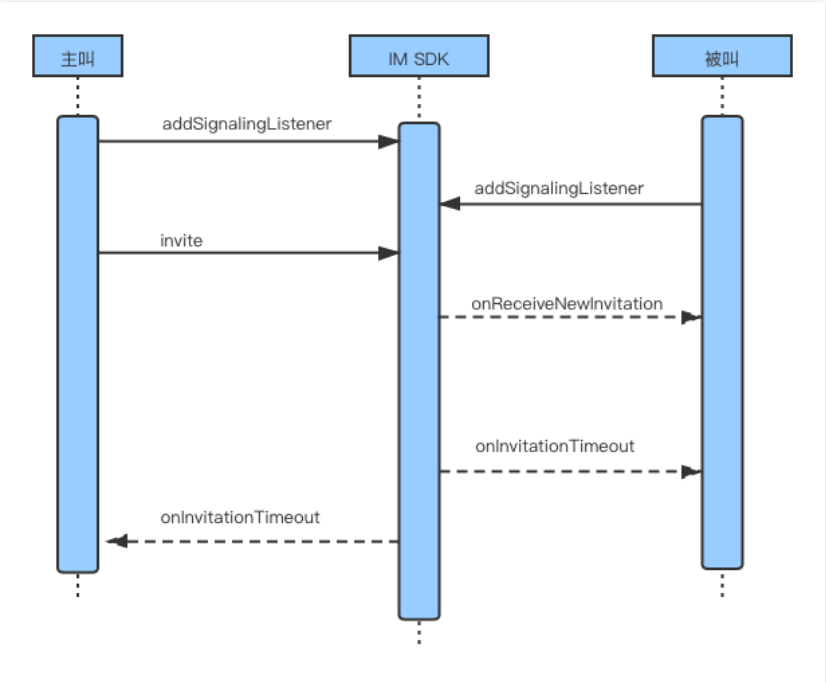


拒绝邀请

被叫收到邀请通知 onReceiveNewInvitation后可以在超时前且主叫取消前拒绝邀请 reject，主叫会收到拒绝邀请通知 onInviteeRejected，所有被叫处理完后（包括接受、拒绝、超时）该邀请流程结束。

邀请超时

若邀请接口的超时时间大于0，且被叫未在超时时间之内响应则邀请超时，主叫和被叫都会收到超时通知 onInvitationTimeout，所有被叫处理完后（包括接受、拒绝、超时）该邀请流程结束。若邀请接口的超时时间等于0，则不会有超时通知。



应用场景案例

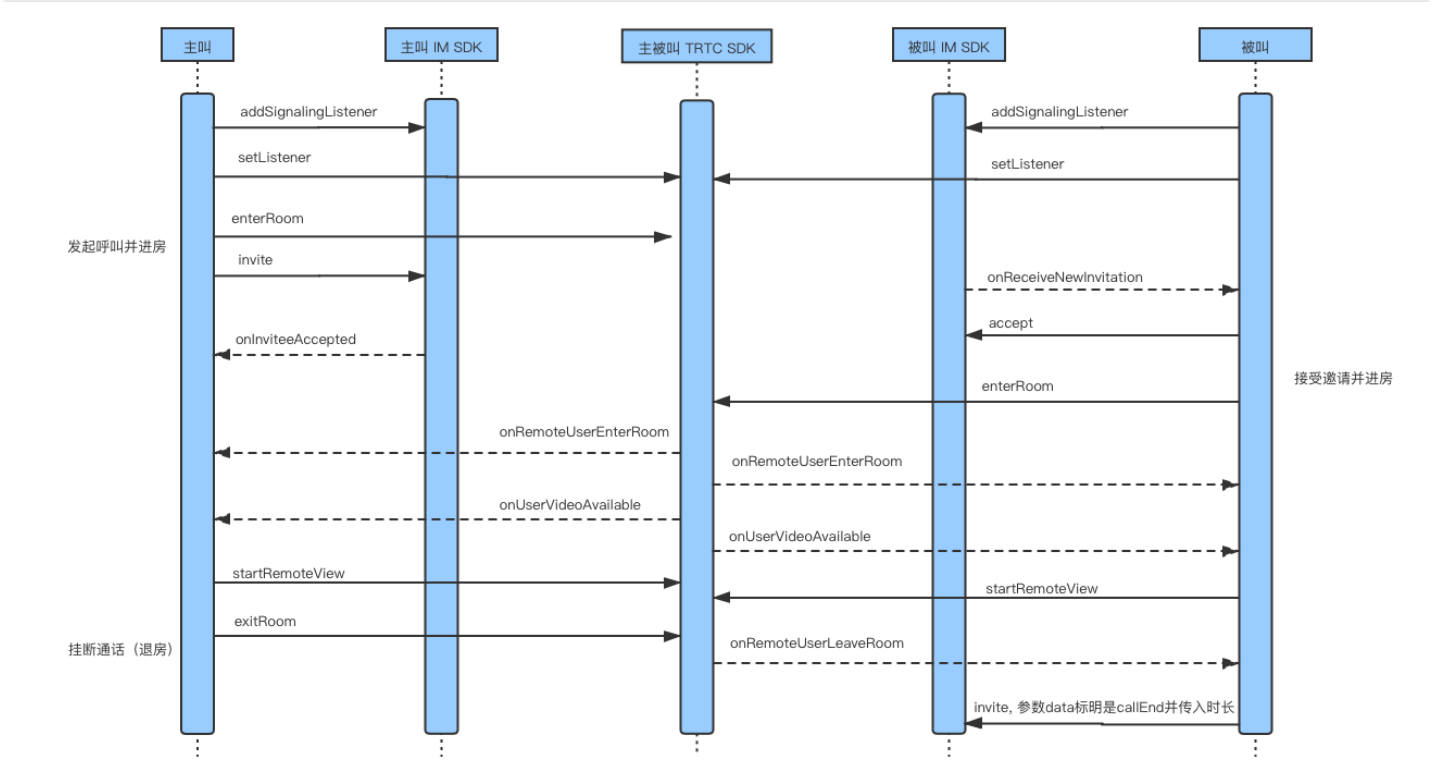
音视频通话

在开源项目 TUIKit Demo 中，我们基于 TRTC 组件 并对其稍作修改提供了一个适合聊天场景的1v1和多人音视频通话的方案，您可以直接基于我们提供的 Demo 进行修改适配。我们以1v1视频通话为例介绍下信令接口跟 TRTC SDK 的结合使用。

1v1视频通话的流程：

1. 主叫根据业务层生成的 roomID 进入该 TRTC 房间，同时调用信令邀请接口 invite 发起音视频通话请求，并把 roomID 放到邀请接口的自定义字段中。
2. 被叫收到信令邀请通知 onReceiveNewInvitation，并通过自定义数据拿到 roomID，界面开始响铃。
3. 被叫处理邀请通知：
   - 接受邀请需调用信令 accept 接口，并根据 roomID 进入到 TRTC 房间，并同时调用 openCamera() 函数打开自己本地的摄像头，双方收到 TRTC SDK 的 onRemoteUserEnterRoom 回调后记录本次通话的开始时间。
   - 拒绝邀请需调用信令 reject 接口结束本次通话。
   - 如果被叫正在跟其他人通话，则调用信令 reject 接口拒绝本次邀请，并在自定义数据中告诉对方是由于本地线路忙而拒绝。
4. 接听并当双方的音视频通道建立完成后，通话的双方都会接收到 TRTC SDK 的 onUserVideoAvailable 的事件通知，表示对方的视频画面已经拿到。此时双方用户均可以调用 TRTC SDK 接口 startRemoteView 展示远端的视频画面。远端的声音默认是自动播放的。
5. 通话结束即某一方挂断电话，该用户退出 TRTC 房间。对方收到 TRTC SDK 的 onRemoteUserLeaveRoom 回调后计算通话总时长并再次发起一次邀请，此邀请的自定义数据中标明是结束通话并附带通话时长，方便 UI 界面做展示。

时序图



教育场景中老师邀请学生举手发言

该场景为老师先让同学们举手，再从举手的同学中选一个同学进行发言。详细流程如下：

1. 老师调用 inviteInGroup 接口邀请同学们举手，自定义 data 中填入“举手操作”，同学们收到 onReceiveNewInvitation 回调。
2. 同学们根据 onReceiveNewInvitation 中的 inviteeList 和 data 字段判断被邀请者里有自己且是举手操作，那么调用 accept 接口举手。
3. 如果有学生举手，所有人都可以收到 onInviteeAccepted 回调，判断 data 中的字段为“举手操作”，展示举手学生列表。
4. 老师从举手成员列表中邀请某个同学进行发言，调用 inviteInGroup 接口，此时自定义 data 中填入“发言操作”，学生们都收到 onReceiveNewInvitation 回调。
5. 学生根据 onReceiveNewInvitation 回调中的 inviteeList 和 data 字段判断被邀请者里有自己且是发言操作，则调用 accept 接口发言。
6. 如果有学生发言，所有人都可以收到 onInviteeAccepted 回调，判断 data 中的字段为“发言操作”，展示发言成员列表。

用户资料与关系链
用户资料与关系链（Android）

最近更新时间：2020-05-29 09:01:41

用户资料管理

查询和修改自己的资料

查询自己的资料接口为 getUsersInfo，其中参数 `userIDList` 需填入自己的 UserID。
修改自己的资料接口为 setSelfInfo。修改自己的资料成功后，会收到 onSelfInfoUpdated 回调。

查询非好友用户资料

查询非好友资料接口同查询自己的资料 getUsersInfo，参数 `userIDList` 填入非好友的 UserID 即可。

查询和修改好友的资料

查询指定的好友资料接口为 getFriendsInfo，从回调信息中通过 `V2TIMFriendInfoResult` 的 `getRelation()` 可以得到该用户与自己的关系：

- `V2TIMFriendCheckResult.V2TIM_FRIEND_RELATION_TYPE_NONE` 表示不是好友。
- `V2TIMFriendCheckResult.V2TIM_FRIEND_RELATION_TYPE_BOTH_WAY` 表示互为好友。
- `V2TIMFriendCheckResult.V2TIM_FRIEND_RELATION_TYPE_IN_MY_FRIEND_LIST` 表示对方在我的好友列表中。
- `V2TIMFriendCheckResult.V2TIM_FRIEND_RELATION_TYPE_IN_OTHER_FRIEND_LIST` 表示我在对方的好友列表中。

修改指定的好友信息接口为 setFriendInfo ，可修改好友备注等资料。

屏蔽某人消息

- **拉黑某人**
  如需屏蔽某人的消息，请调用 addToBlackList 接口把该用户加入黑名单，即拉黑该用户。
  被拉黑的用户默认不会感知到"被拉黑"的状态，消息发送后不会返回已被对方拉黑的错误码。如果希望被拉黑的用户在发消息时返回已被对方拉黑的错误提醒，可以参考被拉黑的用户发消息怎么给错误提示。

- **解除拉黑**
  从黑名单中移除对方后可再次接收对方的消息，可调用 deleteFromBlackList。

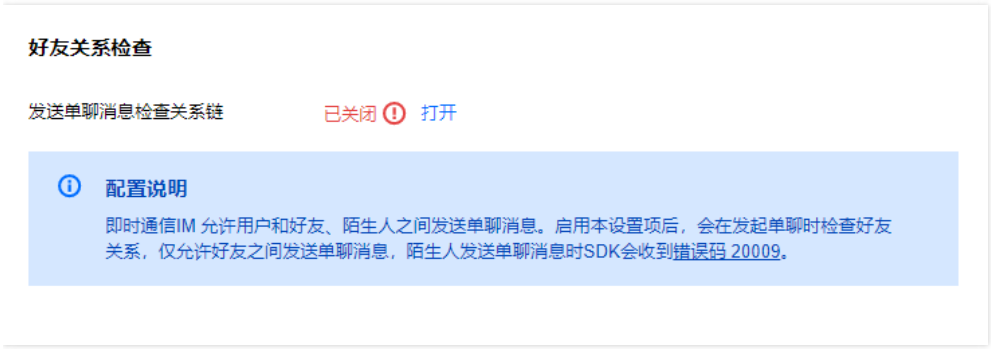
- **获取黑名单列表**
  您可以通过 getBlackList 查看已拉黑多少用户，并对黑名单人员进行管理。

好友管理

是否需要加好友

IM SDK 在发送单聊消息的时候，默认不检查好友关系。在客服场景中，如果用户需要先加客服为好友才能进行沟通非常不方便，因此该默认设置常用于在线客服等场景。
如需实现类似"微信"或者"QQ"中"先加好友，再发消息"的交互体验，您可以在 即时通信 IM 控制台 >【功能配置】>【登录与消息】>【好友关系检查】中开启"发送单聊消息

检查关系链"。开启后，用户只能给好友发送消息，当用户给非好友发消息时，SDK 会报20009错误码。



好友列表管理

IM SDK 支持好友关系链逻辑，您可以调用 getFriendList 接口获取好友列表，调用 deleteFromFriendList 接口删除好友关系，也可以调用 addFriend 接口添加好友。

根据对方用户资料中的加好友需要验证与否，可以分为两种处理流程：

第一种 加好友不需要验证

1. 用户 A 和 B 调用 setFriendListener 设置关系链监听。
2. 用户 B 调用 setSelfInfo 接口并将参数 info 通过接口 setAllowType 设置为加好友不需要验证 V2TIM_FRIEND_ALLOW_ANY 。
3. 用户 A 调用 addFriend 申请添加 B 为好友即可添加成功。如果申请参数 V2TIMFriendAddApplication 中 setAddType 设置为双向好友即 V2TIMFriendInfo.V2TIM_FRIEND_TYPE_BOTH ，则用户 A 和 B 都会收到 onFriendListAdded 回调；
   如果设置为单向好友即 V2TIMFriendInfo.V2TIM_FRIEND_TYPE_SINGLE ，则只有用户 A 收到 onFriendListAdded 回调。

第二种 加好友需要验证

1. 用户 A 和 B 调用 setFriendListener 设置关系链监听。
2. 用户 B 调用 setSelfInfo 接口并将参数 info 通过接口 setAllowType 设置为加好友需要验证 V2TIM_FRIEND_NEED_CONFIRM 。
3. 用户 A 调用 addFriend 申请添加 B 为好友，接口的成功回调参数中 V2TIMFriendOperationResult 中的 getResultCode 返回30539，表示需要等待用户 B 的验证，同时 A 和 B 都会收到 onFriendApplicationListAdded 的回调。
4. 用户 B 会收到 onFriendApplicationListAdded 的回调，当参数 V2TIMFriendApplication 中的 getType 为 V2TIMFriendApplication.V2TIM_FRIEND_APPLICATION_COME_IN 时，可以选择接受或者拒绝
   - 调用 acceptFriendApplication 接受好友请求，如果参数接受类型为 V2TIMFriendApplication.V2TIM_FRIEND_ACCEPT_AGREE 仅同意加单向好友时，A 会收到 onFriendListAdded 回调，说明单向加好友成功，B 会收到 onFriendApplicationListDeleted 回调，此时 B 成为 A 的好友，但 A 仍不是 B 的好友。
   - 如果参数接受类型为 V2TIMFriendApplication.V2TIM_FRIEND_ACCEPT_AGREE_AND_ADD 同意加双向好友时，双方都会收到 onFriendListAdded 回调，说明互相加好友成功。
   - 调用 refuseFriendApplication 拒绝好友请求，双方都会收到 onFriendApplicationListDeleted 回调。

好友分组管理

在某些场景下，您可能需要对好友进行分组，例如分为 "大学同学"、"公司同事" 等，您可以调用以下接口实现。

功能描述	接口指引
新建好友分组	createFriendGroup
删除好友分组	deleteFriendGroup
修改好友分组	renameFriendGroup
获取好友分组	getFriendGroupList
添加好友到一个分组	addFriendsToFriendGroup
从分组中删除某好友	deleteFriendsFromFriendGroup

常见使用问题

1. 非好友之间怎么禁止收发消息？

SDK 默认不限制非好友之间收发消息。如果您希望只允许好友之间收发消息，请在 即时通信 IM 控制台 >【功能配置】>【登录与消息】>【好友关系检查】中开启"发送单聊消息检查关系链"。开启之后，给陌生人发消息时，SDK 会报20009错误码。

2. 被拉黑的用户发消息怎么给错误提示？

当消息发送者被拉黑后，发送者默认不会感知到"被拉黑"的状态，即发送消息后仍展示发送成功（实际上此时接收方不会收到消息）。如果需要被拉黑的发送者收到消息发送失败的提示，请在 即时通信 IM 控制台 >【功能配置】>【登录与消息】>【黑名单检查】中关闭"发送消息后展示发送成功"，关闭后，被拉黑的发送者在发送消息时，SDK 会报20007错误码。

用户资料与关系链（iOS）

最近更新时间：2020-05-14 22:26:15

用户资料管理

查询和修改自己的资料

查询自己的资料接口为 getUsersInfo，其中参数 `userIDList` 需填入自己的 UserID。
修改自己的资料接口为 setSelfInfo。修改自己的资料成功后，会收到 onSelfInfoUpdated 回调。

查询非好友用户资料

查询非好友资料接口同查询自己的资料 getUsersInfo，参数 `userIDList` 填入非好友的 UserID 即可。

查询和修改好友资料

查询指定的好友资料接口为 getFriendsInfo，从回调信息中通过 `V2TIMFriendGetResult` 的 `relation` 字段可以得到该用户与自己的关系：

- `V2TIM_FRIEND_RELATION_TYPE_NONE` 表示不是好友。
- `V2TIM_FRIEND_RELATION_TYPE_BOTH_WAY` 表示互为好友。
- `V2TIM_FRIEND_RELATION_TYPE_IN_MY_FRIEND_LIST` 表示对方在我的好友列表中。

修改指定的好友信息接口为 setFriendInfo ，可修改好友备注等资料。

屏蔽某人消息

- **拉黑某人**
  如需屏蔽某人的消息，请调用 addToBlackList 接口把该用户加入黑名单，即拉黑该用户。
  被拉黑的用户默认不会感知到"被拉黑"的状态，消息发送后不会返回已被对方拉黑的错误码。如果希望被拉黑的用户在发消息时返回已被对方拉黑的错误提醒，可以参考被拉黑的用户发消息怎么给错误提示。

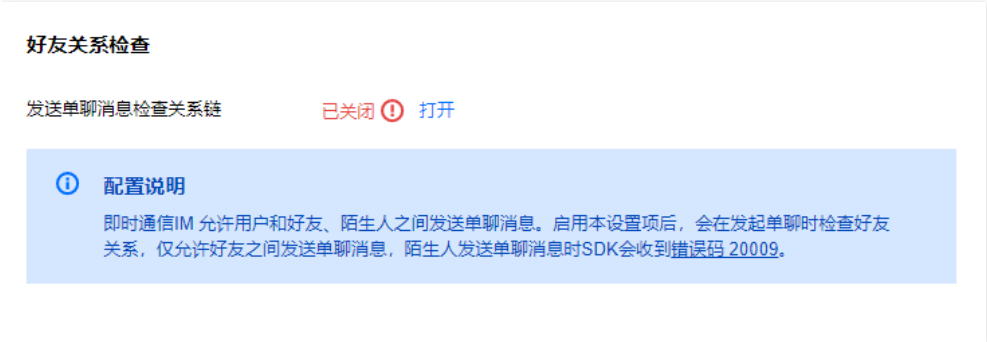
- **解除拉黑**
  从黑名单中移除对方后可再次接收对方的消息，可调用 deleteFromBlackList。

- **获取黑名单列表**
  您可以通过 getBlackList 查看已拉黑多少用户，并对黑名单人员进行管理。

好友管理

是否需要加好友

IM SDK 在发送单聊消息的时候，默认不检查好友关系。在客服场景中，如果用户需要先加客服为好友才能进行沟通非常不方便，因此该默认设置常用于在线客服等场景。
如需实现类似"微信"或者"QQ"中"先加好友，再发消息"的交互体验，您可以在 即时通信 IM 控制台 >【功能配置】>【登录与消息】>【好友关系检查】中开启"发送单聊消息检查关系链"。开启后，用户只能给好友发送消息，当用户给非好友发消息时，SDK 会报20009错误码。



好友列表管理

IM SDK 支持好友关系链逻辑，您可以调用 getFriendList 接口获取好友列表，调用 deleteFromFriendList 接口删除好友关系，也可以调用 addFriend 接口添加好友。

根据对方用户资料中的加好友需要验证与否，可以分为两种处理流程：

第一种：加好友不需要对方验证

1. 用户 A 和 B 调用 setFriendListener 设置关系链监听。
2. 用户 B 通过 setSelfInfo 函数里的 allowType 字段设置为加好友不需要验证 V2TIM_FRIEND_ALLOW_ANY 。
3. 用户 A 调用 addFriend 申请添加 B 为好友即可添加成功。
   - 如果申请参数 V2TIMFriendAddApplication 中 addType 设置为双向好友即 V2TIM_FRIEND_TYPE_BOTH ，则用户 A 和 B 都会收到 onFriendListAdded 回调；
   - 如果设置为单向好友即 V2TIM_FRIEND_TYPE_SINGLE ，则只有用户 A 收到 onFriendListAdded 回调。

第二种：加好友需要通过对方验证

1. 用户 A 和 B 调用 setFriendListener 设置关系链监听。
2. 用户 B 通过 setSelfInfo 函数里的 allowType 字段设置为加好友需要验证 V2TIM_FRIEND_NEED_CONFIRM 。
3. 用户 A 调用 addFriend 申请添加 B 为好友，接口的成功回调参数中 V2TIMFriendOperationResult 中的 resultCode 返回30539，表示需要等待用户 B 的验证，同时 A 和 B 都会收到 onFriendApplicationListAdded 的回调。
4. 用户 B 会收到 onFriendApplicationListAdded 的回调，当参数 V2TIMFriendApplication 中的 type 为 V2TIM_FRIEND_APPLICATION_COME_IN 时，可以选择接受或者拒绝：
   - 调用 acceptFriendApplication 接受好友请求，如果参数接受类型为 V2TIM_FRIEND_ACCEPT_AGREE 仅同意加单向好友时，A 会收到 onFriendListAdded 回调，说明单向加好友成功，B 会收到 onFriendApplicationListDeleted 回调，此时 B 成为 A 的好友，但 A 仍不是 B 的好友。
   - 调用 acceptFriendApplication 接受好友请求，如果参数接受类型为 V2TIM_FRIEND_ACCEPT_AGREE_AND_ADD 同意加双向好友时，A 和 B 都会收到 onFriendListAdded 回调，说明互相加好友成功。
   - 调用 refuseFriendApplication 拒绝好友请求，双方都会收到 onFriendApplicationListDeleted 回调。

好友分组管理

在某些场景下，您可能需要对好友进行分组，例如分为 "大学同学"、"公司同事" 等，您可以调用以下接口实现。

功能描述	接口指引
新建好友分组	createFriendGroup
删除好友分组	deleteFriendGroup
修改好友分组	renameFriendGroup
获取好友分组	getFriendGroupList
添加好友到一个分组	addFriendsToFriendGroup
从分组中删除某好友	deleteFriendsFromFriendGroup

常见使用问题

非好友之间怎么禁止收发消息？

SDK 默认不限制非好友之间收发消息。如果您希望只允许好友之间收发消息，请在 即时通信 IM 控制台 >【功能配置】>【登录与消息】>【好友关系检查】中开启"发送单聊消息检查关系链"。开启之后，给陌生人发消息时，SDK 会报20009错误码。

被拉黑的用户发消息怎么给错误提示？

当消息发送者被拉黑后，发送者默认不会感知到"被拉黑"的状态，即发送消息后仍展示发送成功（实际上此时接收方不会收到消息）。如果需要被拉黑的发送者收到消息发送失败的提示，请在 即时通信 IM 控制台 >【功能配置】>【登录与消息】>【黑名单检查】中关闭"发送消息后展示发送成功"，关闭后，被拉黑的发送者在发送消息时，SDK 会报20007错误码。

用户资料（Web & 小程序）

最近更新时间：2020-07-02 17:50:19

用户资料

获取我的个人资料

获取个人资料，更多详情请参见 Profile。

> 注意：
> v2.3.2版本开始支持自定义资料字段，使用前需要将 SDK 升级至v2.3.2或以上。

接口名

```
tim.getMyProfile();
```

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data 中获取个人信息。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.getMyProfile();
promise.then(function(imResponse) {
console.log(imResponse.data); // 个人资料 - Profile 实例
}).catch(function(imError) {
console.warn('getMyProfile error:', imError); // 获取个人资料失败的相关信息
});
```

获取其他用户资料

此接口会同时获取标配资料和 自定义资料。

> 注意：
> - v2.3.2版本开始支持自定义资料字段，使用前需要将 SDK 升级至v2.3.2或以上。
> - 如果您没有配置自定义资料字段，或者配置了自定义资料字段但未设置 value，此接口将不会返回自定义资料的内容。
> - 每次拉取的用户数不超过100，避免因回包数据量太大导致回包失败。如果传入的数组长度大于100，则只取前100个用户进行查询，其余丢弃。

接口名

```
tim.getUserProfile(options);
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	描述
userIDList	Array<String>	用户的帐号列表，类型为数组

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data 中获取用户资料数组。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.getUserProfile({
userIDList: ['user1', 'user2'] // 请注意：即使只拉取一个用户的资料，也需要用数组类型，例如：userIDList: ['user1']
});
promise.then(function(imResponse) {
console.log(imResponse.data); // 存储用户资料的数组 - [Profile]
}).catch(function(imError) {
console.warn('getUserProfile error:', imError); // 获取其他用户资料失败的相关信息
});
```

更新个人资料

> 注意：
> v2.3.2版本开始支持自定义资料字段，使用前需要将 SDK 升级至v2.3.2或以上。

接口名

```
tim.updateMyProfile(options);
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	描述
nick	String	昵称
avatar	String	头像地址
gender	String	性别 • TIM.TYPES.GENDER_UNKNOWN 表示未设置性别 • TIM.TYPES.GENDER_FEMALE 表示女 • TIM.TYPES.GENDER_MALE 表示男
selfSignature	String	个性签名
allowType	String	当被加人加好友时是否需要验证 • TIM.TYPES.ALLOW_TYPE_ALLOW_ANY 表示允许直接加为好友 • TIM.TYPES.ALLOW_TYPE_NEED_CONFIRM 表示需要验证 • TIM.TYPES.ALLOW_TYPE_DENY_ANY 表示拒绝
birthday	Number	生日，推荐用法：20000101
location	String	所在地，推荐用法：App 本地定义一套数字到地名的映射关系，后台实际保存的是4个 uint32_t 类型的数字： • 第一个 uint32_t 表示国家 • 第二个 uint32_t 用于表示省份 • 第三个 uint32_t 用于表示城市 • 第四个 uint32_t 用于表示区县
language	Number	语言
messageSettings	Number	消息设置，0表示接收消息，1表示不接收消息
adminForbidType	String	管理员禁止加好友标识： • TIM.TYPES.FORBID_TYPE_NONE 表示允许加好友，默认值 • TIM.TYPES.FORBID_TYPE_SEND_OUT 表示禁止该用户发起加好友请求
level	Number	等级，建议拆分以保存多种角色的等级信息
role	Number	角色，建议拆分以保存多种角色信息
profileCustomField	Array<Object>	自定义资料 键值对集合，可根据业务侧需要使用

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data 中获取用户的新资料。
- catch 的回调函数参数为 IMError。

示例

```
// 修改个人标配资料
let promise = tim.updateMyProfile({
nick: '我的昵称',
avatar: 'http(s)://url/to/image.jpg',
gender: TIM.TYPES.GENDER_MALE,
selfSignature: '我的个性签名',
allowType: TIM.TYPES.ALLOW_TYPE_ALLOW_ANY
});
promise.then(function(imResponse) {
console.log(imResponse.data); // 更新资料成功
}).catch(function(imError) {
console.warn('updateMyProfile error:', imError); // 更新资料失败的相关信息
});

// 修改个人自定义资料
// 自定义资料字段需要预先在控制台配置，详细请参考：https://cloud.tencent.com/document/product/269/1500#.E8.87.AA.E5.AE.9A.E4.B9.89.E8.B5.84.E6.96.99.E5.AD.97.E6.AE.B5
let promise = tim.updateMyProfile({
// 这里要求您已在即时通信 IM 控制台>【应用配置】>【功能配置】 申请了自定义资料字段 Tag_Profile_Custom_Test1
// 注意：即使只有一个自定义资料字段，profileCustomField 也需要用数组类型
profileCustomField: [
{
key: 'Tag_Profile_Custom_Test1',
value: '我的自定义资料1'
}
]
});
promise.then(function(imResponse) {
console.log(imResponse.data); // 更新资料成功
}).catch(function(imError) {
console.warn('updateMyProfile error:', imError); // 更新资料失败的相关信息
});

// 修改个人标配资料和自定义资料
let promise = tim.updateMyProfile({
nick: '我的昵称',
// 这里要求您已在即时通信 IM 控制台>【应用配置】>【功能配置】 申请了自定义资料字段 Tag_Profile_Custom_Test1 和 Tag_Profile_Custom_Test2
profileCustomField: [
{
key: 'Tag_Profile_Custom_Test1',
value: '我的自定义资料1'
},
{
key: 'Tag_Profile_Custom_Test2',
value: '我的自定义资料2'
},
]
});
promise.then(function(imResponse) {
console.log(imResponse.data); // 更新资料成功
}).catch(function(imError) {
console.warn('updateMyProfile error:', imError); // 更新资料失败的相关信息
});
```

黑名单

获取我的黑名单列表

接口名

```
tim.getBlacklist();
```

请求参数

参数 options 为 Object 类型，包含的属性值如下：

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data 中获取黑名单列表。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.getBlacklist();
promise.then(function(imResponse) {
console.log(imResponse.data); // 我的黑名单列表，结构为包含用户 userID 的数组 - [userID]
}).catch(function(imError) {
console.warn('getBlacklist error:', imError); // 获取黑名单列表失败的相关信息
});
```

添加用户到黑名单列表

添加用户到黑名单列表。将用户加入黑名单后可以屏蔽来自 TA 的所有消息，因此该接口可以实现“屏蔽该用户消息”的功能。

- 如果用户 A 与用户 B 之间存在好友关系，拉黑时会解除双向好友关系。
- 如果用户 A 与用户 B 之间存在黑名单关系，二者之间无法发起加好友请求。
- 如果用户 A 的黑名单中有用户 B 且用户 B 的黑名单中有用户 A，二者之间无法发起会话。
- 如果用户 A 的黑名单中有用户 B 但用户 B 的黑名单中没有用户 A，那么用户 A 可以给用户 B 发消息，用户 B 不能给用户 A 发消息。

接口名

```
tim.addToBlacklist(options);
```

请求参数

参数 options 为 Object 类型，包含的属性值如下：

名称	类型	描述
userIDList	Array<String>	待添加为黑名单的用户 userID 列表，单次请求的 userID 数不得超过1000

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data 中获取黑名单列表。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.addToBlacklist({userIDList: ['user1', 'user2']}); // 请注意：即使只添加一个用户帐号到黑名单，也需要用数组类型，例如：userIDList: ['user1']
promise.then(function(imResponse) {
console.log(imResponse.data); // 成功添加到黑名单的帐号信息，结构为包含用户 userID 的数组 - [userID]
}).catch(function(imError) {
console.warn('addToBlacklist error:', imError); // 添加用户到黑名单列表失败的相关信息
});
```

将用户从黑名单中移除

将用户从黑名单中移除。移除后，可以接收来自 TA 的所有消息。

接口名

```
tim.removeFromBlacklist(options);
```

请求参数

参数 options 为 Object 类型，包含的属性值如下表所示：

名称	类型	描述
userIDList	Array<String>	待从黑名单中移除的 userID 列表，单次请求的 userID 数不得超过1000

返回值

该接口返回 Promise 对象：

- then 的回调函数参数为 IMResponse，可在 IMResponse.data 中获取从黑名单中成功移除的帐号列表。
- catch 的回调函数参数为 IMError。

示例

```
let promise = tim.removeFromBlacklist({userIDList: ['user1', 'user2']}); // 请注意：即使只从黑名单中移除一个用户帐号，也需要用数组类型，例如：userIDList: ['user1']
result.then(function(imResponse) {
console.log(imResponse.data); // 从黑名单中成功移除的帐号列表，结构为包含用户 userID 的数组 - [userID]
}).catch(function(imError) {
console.warn('removeFromBlacklist error:', imError); // 将用户从黑名单中移除失败的相关信息
});
```

离线推送
离线推送（Android）

最近更新时间：2020-07-24 17:05:04

概述

即时通信 IM 的终端用户需要随时都能够得知最新的消息，而由于移动端设备的性能与电量有限，当 App 处于后台时，为了避免维持长连接而导致的过多资源消耗，即时通信 IM 推荐您使用各厂商提供的系统级推送通道来进行消息通知，系统级的推送通道相比第三方推送拥有更稳定的系统级长连接，可以做到随时接受推送消息，且资源消耗大幅降低。

即时通信 IM 目前已经支持了小米推送、华为推送、魅族推送、vivo 推送、OPPO 推送、Google FCM推送，具体如下：

推送通道	系统要求	条件说明
小米推送	MIUI	使用小米推送 MiPush_SDK_Client_3_7_6.jar
华为推送	EMUI	华为推送版本 com.huawei.hms:push:5.0.0.300
Google FCM 推送	Android 4.1 及以上	手机端需安装 Google Play Services 且在中国大陆地区以外使用。
魅族推送	Flyme	使用魅族推送 com.meizu.flyme.internet:push-internal:3.9.7
OPPO 推送	ColorOS	并非所有 OPPO 机型和版本都支持使用 OPPO 推送，SDK 版本 com.heytap.msp-push-2.1.0.aar
vivo 推送	FuntouchOS	并非所有 vivo 机型和版本都支持使用 vivo 推送，SDK 版本 vivo_pushsdk-v2.9.0.0.aar

这里的离线是指在没有退出登录的情况下，应用被系统或者用户关闭。在这种情况下，如果还想收到 IM SDK 的消息提醒，可以集成即时通信 IM 离线推送。

> 注意：
> 对于已经退出登录（主动登出或者被踢下线）的用户，不会收到任何消息通知。

实现离线消息推送的过程如下：

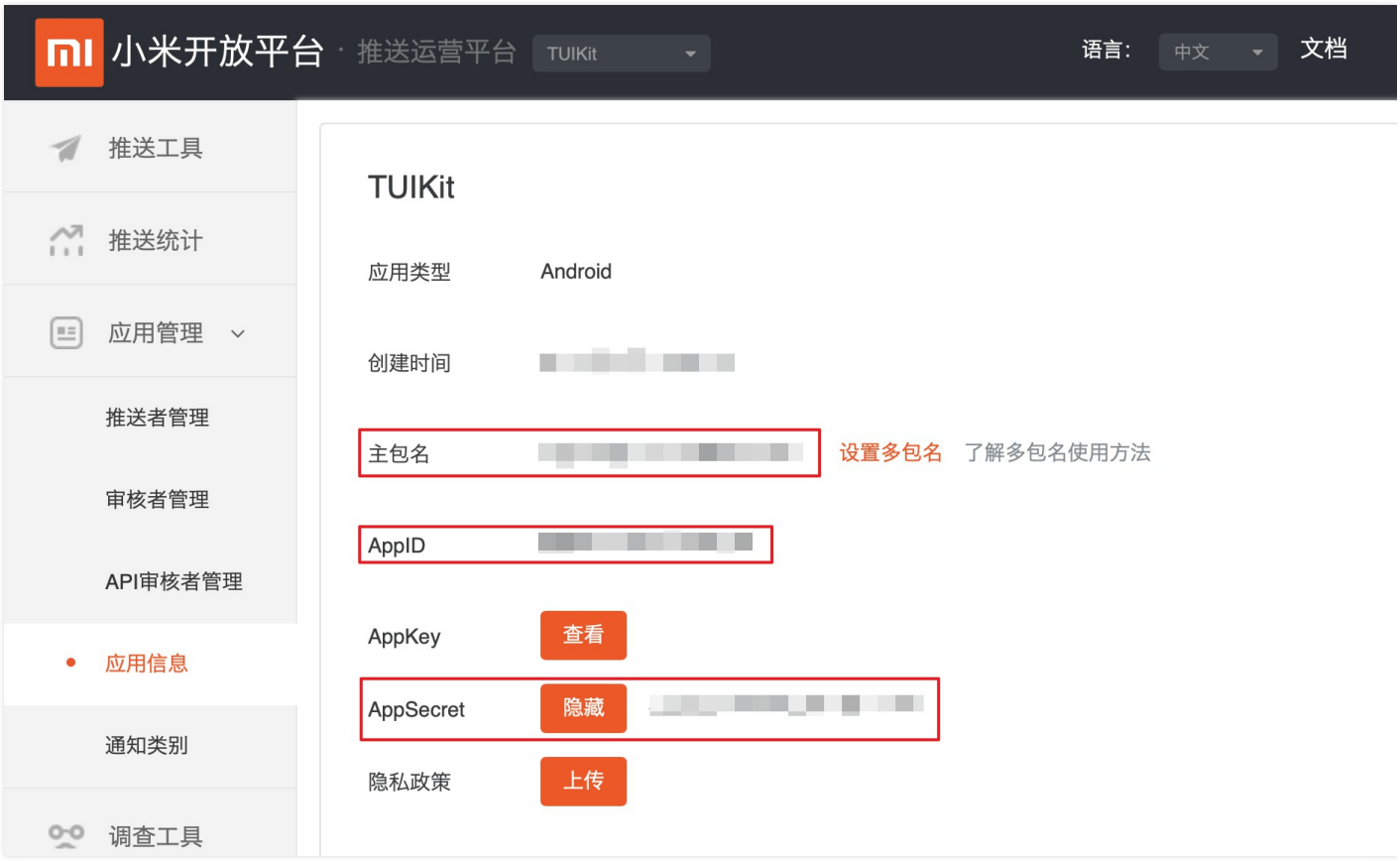
1. 开发者到厂商的平台注册账号，开通推送服务并创建应用，得到AppID、AppKey、AppSecret 等信息。
2. 将厂商提供的推送 SDK 集成到开发者的项目工程中，并在厂商控制台测试通知消息，确保成功集成。
3. 登录 即时通信 IM 控制台 填写推送证书及相关信息，即时通信 IM 服务端会为每个证书生成不同的证书 ID。
4. 集成即时通信 IM SDK 到项目后，将证书 ID、设备信息等上报至即时通信 IM 服务端。

当客户端 App 在即时通信 IM 没有退出登录的情况下被系统或者用户 kill 时，即时通信 IM 服务端会将其他账号发来的消息通过厂商的通道推送下去。

小米推送

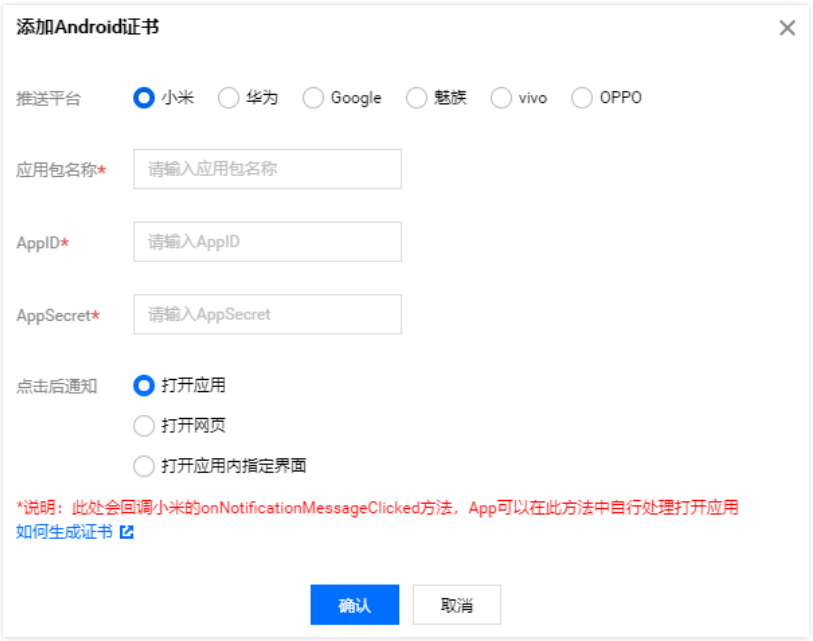
配置推送证书

1. 打开 小米开放平台官网 进行注册并通过开发者认证。登录小米开放平台的管理控制台，选择【应用服务】>【PUSH服务】，创建小米推送服务应用，记录 **主包名**、**AppID**、**AppSecret** 信息。

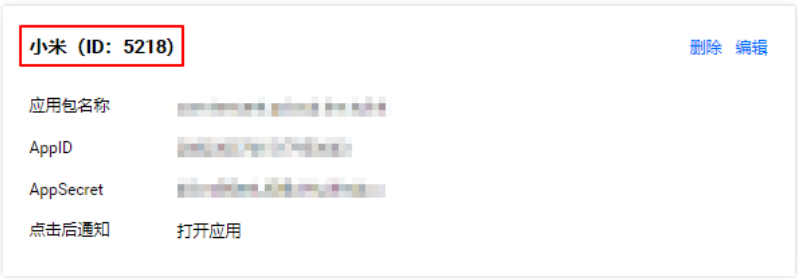


2. 登录腾讯云 即时通信 IM 控制台，单击目标应用卡片，进入应用的基础配置页面，单击【Android平台推送设置】区域的【添加证书】。根据 步骤1 中获取的信息设置以下参数：

- **推送平台**：选择**小米**
- **应用包名称**：填写小米推送服务应用的**主包名**
- **AppID**：填写小米推送服务应用的 **AppID**
- **AppSecret**：填写小米推送服务应用的 **AppSecret**
- **点击通知后**：选择点击通知栏消息后的响应操作， 支持**打开应用**、**打开网页**和**打开应用内指定界面**，更多详情请参见 配置点击通知栏消息事件
  当设置为【打开应用】或【打开应用内指定界面】操作时，支持 透传自定义内容。



单击【确认】保存信息，记录证书的 **ID** 。证书信息保存后10分钟内生效。



集成推送 SDK

1. 请参考小米推送集成指南 集成 SDK，并在小米控制台测试通知消息，确保已成功集成。
2. 通过调用 `MiPushClient.registerPush` 来对小米推送服务进行初始化，注册成功后您将在自定义的 `BroadcastReceiver` 的 `onReceiveRegisterResult` 中收到注册结果。其中 `regId` 为当前设备上当前 App 的唯一标识。当登录 IM SDK 成功后，需要调用 setOfflinePushConfig 将**证书 ID** 和 **regId** 上报到即时通信 IM 服务端。

成功上报证书 ID 及 regId 后，即时通信 IM 服务端会在该设备上的即时通信 IM 用户 logout 之前、App 被 kill 之后将消息通过小米推送通知到用户端。

配置点击通知栏消息事件

您可以选择点击通知栏消息后**打开应用**、**打开网页**或**打开应用内指定界面**。

打开应用

设置为点击通知栏消息打开应用时，会回调小米的 onNotificationMessageClicked 方法，App 可以在此方法中自行处理打开应用。

打开网页

您需要在 添加证书 时选择【打开网页】并输入以 `http://` 或 `https://` 开头的网址，例如 `https://cloud.tencent.com/document/product/269`。

打开应用内指定界面

1. 在 manifest 中配置需要打开的 Activity 的 `intent-filter`，示例代码如下，可以参考 Demo 的 AndroidManifest.xml：

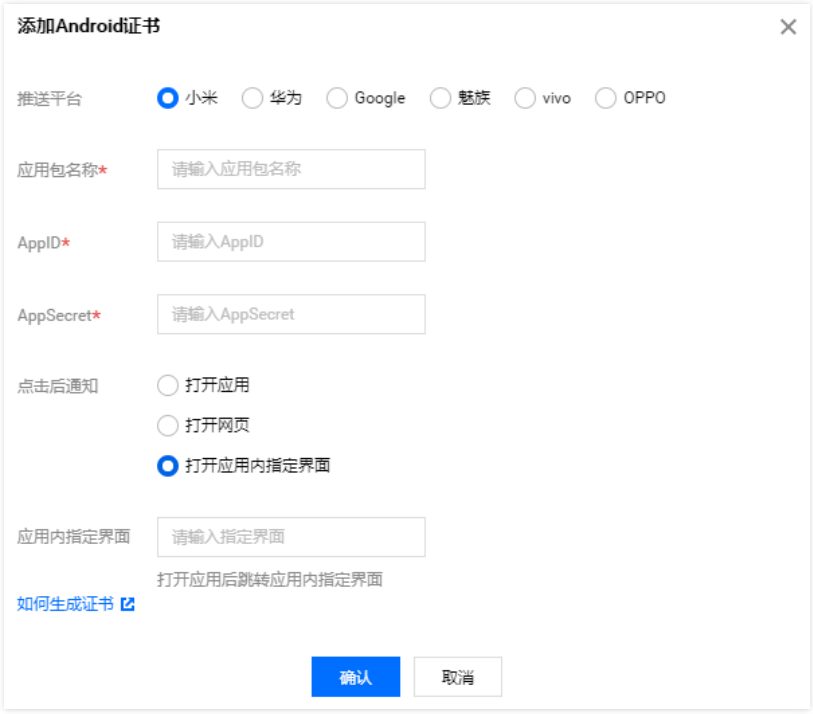
```
<activity
android:name="com.tencent.qcloud.tim.demo.chat.ChatActivity"
android:launchMode="singleTask"
android:screenOrientation="portrait"
android:windowSoftInputMode="adjustResize|stateHidden">
<intent-filter>
```

```
<action android:name="android.intent.action.VIEW" />
<data
android:host="com.tencent.qcloud.tim"
android:path="/detail"
android:scheme="pushscheme" />
</intent-filter>
</activity>
```

2. 获取 intent URL，方式如下：

```
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);

// 打印结果
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. 在 添加证书 时选择【打开应用内指定界面】并输入上述打印结果。



透传自定义内容

添加证书 时设置【点击通知后】为【打开应用】或【打开应用内指定界面】操作才支持透传自定义内容。

步骤1：发送端设置自定义内容

在发消息前设置每条消息的通知栏自定义内容。

- 下面是 Android 端简单示例，也可以参考 TUIKit 中的 ChatManagerKit.java 类的 sendMessage() 方法中对应的逻辑：

```
JSONObject jsonObject = new JSONObject();
try {
jsonObject.put("extKey", "ext content");
} catch (JSONException e) {
e.printStackTrace();
}
String extContent = jsonObject.toString();
```

```
V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
v2TIMOfflinePushInfo.setExt(extContent.getBytes());
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, userID, null, V2TIMMessage.V2TIM_PRIORITY_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) {}
});
```

- 服务端示例请参见 OfflinePushInfo 的格式示例

步骤2：接收端获取自定义内容

- 若 添加证书 时设置【点击通知后】的操作为【打开应用】，当点击通知栏的消息时，会触发小米推送 SDK 的 `onNotificationMessageClicked(Context context, MiPushMessage miPushMessage)` 回调，自定义内容可以从 `miPushMessage` 中获取，可以参考 XiaomiMsgReceiver.java 的解析实现。

```
Map extra = miPushMessage.getExtra();
String extContent = extra.get("ext");
```

- 若 添加证书 时设置【点击通知后】的操作为【打开应用内指定界面】，封装消息的 `MiPushMessage` 对象通过 `Intent` 传到客户端，客户端在相应的 `Activity` 中获取自定义内容，可以参考 OfflineMessageDispatcher.java 类的 parseOfflineMessage(Intent intent) 方法实现。

```
Bundle bundle = getIntent().getExtras();
MiPushMessage miPushMessage = (MiPushMessage)bundle.getSerializable(PushMessageHelper.KEY_MESSAGE);
Map extra = miPushMessage.getExtra();
String extContent = extra.get("ext");
```

华为推送

配置推送证书

1. 打开 华为开发者联盟官网 进行注册并通过开发者认证。进入管理中心，选择【应用服务】>【开发服务】>【PUSH】，创建华为推送服务应用。记录 **包名** 、 **APP ID** 、 **APP SECRET** 信息。

2. 登录腾讯云 即时通信 IM 控制台，单击目标应用卡片，进入应用的基础配置页面，单击【Android平台推送设置】区域的【添加证书】。根据 步骤1 中获取的信息设置以下参数：

- **推送平台**：选择**华为**
- **应用包名称**：填写华为推送服务应用的**包名**
- **AppID**：填写华为推送服务应用的 **APP ID**
- **AppSecret**：填写华为推送服务应用的 **APP SECRET**
- **角标参数**：填写应用入口完整 `Activity` 类名，用作华为桌面应用角标显示，请参考 华为桌面角标开发指导书
- **点击通知后**：选择点击通知栏消息后的响应操作， 支持**打开应用**、**打开网页**和**打开应用内指定界面**，更多详情请参见 配置点击通知栏消息事件
  当设置为【打开应用】或【打开应用内指定界面】操作时，支持 透传自定义内容。

单击【保存】保存信息，记录证书的 **ID** 。证书信息保存后10分钟内生效。

集成推送 SDK

1. 请参考华为推送集成指南集成 SDK，并在华为控制台测试通知消息，确保已成功集成。
2. 通过调用华为 `HmsInstanceId.getToken` 接口向服务端请求应用的唯一标识 Push Token， `Push Token` 为当前设备上当前 App 的唯一标识。当登录 IM SDK 成功后，需要调用 setOfflinePushConfig 将**证书 ID** 和 **Push Token** 上报到即时通信 IM 服务端。

成功上报证书 ID 及 regId 后，即时通信 IM 服务端会在该设备上的即时通信 IM 用户 logout 之前、App 被 kill 之后将消息通过小米推送通知到用户端。

配置点击通知栏消息事件

您可以选择点击通知栏消息后**打开应用**、**打开网页**或**打开应用内指定界面**。

打开应用

默认为点击通知栏消息打开应用。

打开网页

您需要在 添加证书 时选择【打开网页】并输入以 `http://` 或 `https://` 开头的网址，例如 `https://cloud.tencent.com/document/product/269` 。

打开应用内指定界面

1. 在 manifest 中配置需要打开的 Activity 的 `intent-filter` ，示例代码如下，可以参考 Demo 的 AndroidManifest.xml：

```
<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. 获取 intent URL，方式如下：

```
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);

// 打印结果
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. 在 添加证书 时选择【打开应用内指定界面】并输入上述打印结果。

透传自定义内容

> 注意：
> 由于华为推送的兼容性问题，透传内容只能在部分 EUI10+ 的设备上收到。

步骤1：发送端设置自定义内容
在发消息前设置每条消息的通知栏自定义内容。

- 下面是 Android 端简单示例，也可以参考 TUIKit 中 ChatManagerKit.java 类的 sendMessage() 方法中对应的逻辑：

```
JSONObject jsonObject = new JSONObject();
try {
jsonObject.put("extKey", "ext content");
} catch (JSONException e) {
e.printStackTrace();
}
String extContent = jsonObject.toString();

V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
v2TIMOfflinePushInfo.setExt(extContent.getBytes());
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, userID, null, V2TIMMessage.V2TIM_PRIORITY_DEFAULT, false, v2TIMOfflinePushIn
fo, new V2TIMSendCallback<V2TIMMessage>() {
@Override
public void onError(int code, String desc) {}
@Override
public void onSuccess(V2TIMMessage v2TIMMessage) {}
@Override
public void onProgress(int progress) {}
});
```

- 服务端示例请参见 OfflinePushInfo 的格式示例

步骤2：接收端获取自定义内容

- 若 添加证书 时设置【点击通知后】的操作为【打开应用】或【打开应用内指定界面】，当点击通知栏的消息时，客户端可以在相应的 `Activity` 中获取自定义内容，可以参考 OfflineMessageDispatcher.java 类的 parseOfflineMessage(Intent intent) 方法实现。

```
Bundle bundle = getIntent().getExtras();
String value = bundle.getString("ext");
```

OPPO 推送

配置推送证书

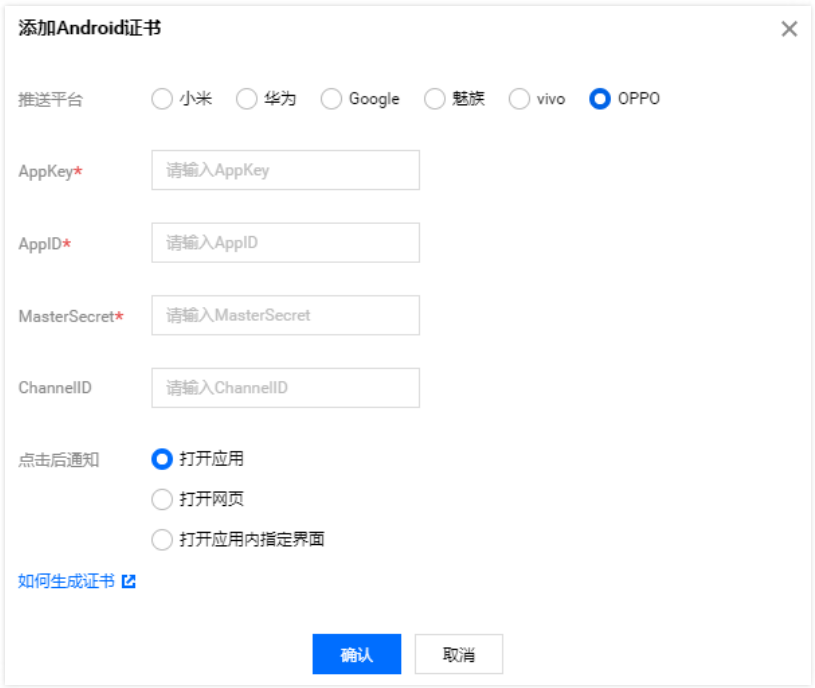
1. 打开 OPPO PUSH 服务开启指南 开通 PUSH 服务。在 OPPO 推送平台 >【配置管理】>【应用配置】页面，您可以查看详细的应用信息，记录 `AppId`、`AppKey`、`AppSecret` 和 `MasterSecret` 信息。

2. 按照 OPPO 官网要求，在 OPPO Android 8.0 及以上系统版本必须配置 ChannelID，否则推送消息无法展示。您需要先在 App 中创建对应的 ChannelID（例如 `tuikit`）：

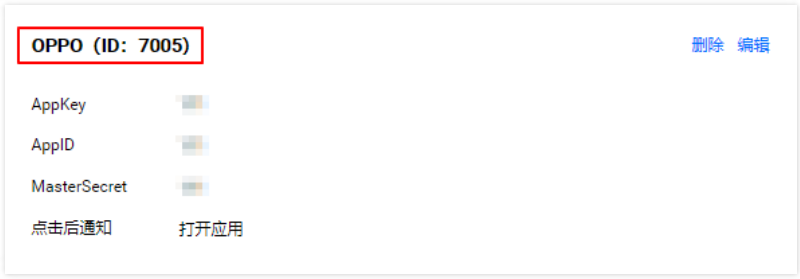
```
public void createNotificationChannel(Context context) {
// Create the NotificationChannel, but only on API 26+ because
// the NotificationChannel class is new and not in the support library
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
CharSequence name = "oppotest";
String description = "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);
}
}
```

3. 登录腾讯云 即时通信 IM 控制台，单击目标应用卡片，进入应用的基础配置页面，单击【Android平台推送设置】区域的【添加证书】。根据 步骤1 中获取的信息设置以下参数：
   - **推送平台**：选择**OPPO**
   - **AppKey**：填写 OPPO 推送服务应用的 **AppKey**
   - **AppID**：填写 OPPO 推送服务应用的 **AppId**
   - **MasterSecret**：填写 OPPO 推送服务应用的 **MasterSecret**
   - **ChannelID**：填写您在 App 中创建的 **ChannelID**
   - **点击通知后**：选择点击通知栏消息后的响应操作， 支持**打开应用**、**打开网页**和**打开应用内指定界面**，更多详情请参见 配置点击通知栏消息事件
     当设置为【打开应用】或【打开应用内指定界面】操作时，支持 透传自定义内容。



单击【确认】保存信息，记录证书的 **ID** 。证书信息保存后10分钟内生效。



集成推送 SDK

1. 请参考 OPPO PUSH SDK 接口文档 集成 SDK，并在 OPPO 控制台测试通知消息，确保已成功集成。
2. 通过调用 OPPO SDK 中的 PushManager.getInstance().register(...) 初始化 Opush 推送服务。
   注册成功后，您可以在 PushCallback 的 onRegister 回调方法中得到 regId ， regId 为当前设备上当前 App 的唯一标识。当登录 IM SDK 成功后，需要调用 setOfflinePushConfig 将**证书 ID** 和 **regId** 上报到即时通信 IM 服务端。

成功上报证书 ID 及 regId 后，即时通信 IM 服务端会在该设备上的即时通信 IM 用户 logout 之前、App 被 kill 之后将消息通过小米推送通知到用户端。

配置点击通知栏消息事件

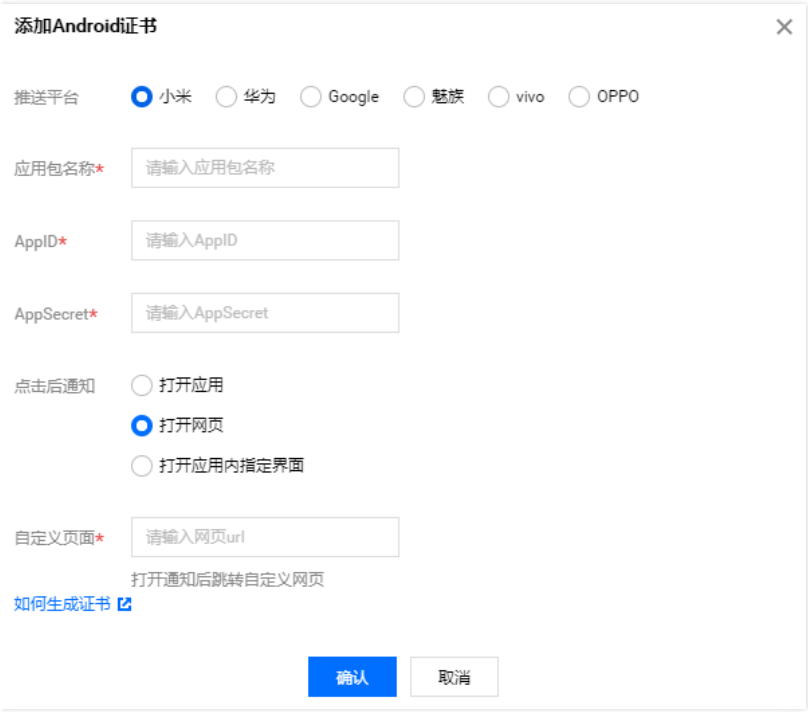
您可以选择点击通知栏消息后**打开应用**、**打开网页**或**打开应用内指定界面**。

打开应用

默认为点击通知栏消息打开应用。

打开网页

您需要在 添加证书 时选择【打开网页】并输入以 `http://` 或 `https://` 开头的网址，例如 `https://cloud.tencent.com/document/product/269`。



打开应用内指定界面

打开应用内指定界面有以下几种方式：

Activity（推荐）
该方式比较简单，填入打开的 Activity 的完整类名即可，例如 `com.tencent.qcloud.tim.demo.SplashActivity`

Intent action

1. 在 AndroidManifest 要打开的 Activity 中做如下配置，并且必须加上 category 且不能有 data 数据，可以参考 Demo 的 AndroidManifest.xml：

```
<intent-filter>
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

2. 在控制台上填入 `android.intent.action.VIEW`。

透传自定义内容

添加证书 时设置【点击通知后】为【打开应用】或【打开应用内指定界面】操作支持透传自定义内容。

步骤1：发送端设置自定义内容
在发消息前设置每条消息的通知栏自定义内容。

- 下面是 Android 端简单示例，也可以参考 TUIKit 中的 ChatManagerKit.java 类的 sendMessage() 方法中对应的逻辑：

```
JSONObject jsonObject = new JSONObject();
try {
jsonObject.put("extKey", "ext content");
```

```
} catch (JSONException e) {
e.printStackTrace();
}
String extContent = jsonObject.toString();

V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
v2TIMOfflinePushInfo.setExt(extContent.getBytes());
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, userID, null, V2TIMMessage.V2TIM_PRIORITY_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) {}
});
```

- 服务端示例请参见 OfflinePushInfo 的格式示例

步骤2：接收端获取自定义内容

当点击通知栏的消息时，客户端在启动的 `Activity` 中获取自定义内容，可以参考 OfflineMessageDispatcher.java 类的 parseOfflineMessage(Intent intent) 方法实现。

```
Bundle bundle = intent.getExtras();
Set<String> set = bundle.keySet();
if (set != null) {
for (String key : set) {
// 其中 key 和 value 分别为发送端设置的 extKey 和 ext content
String value = bundle.getString(key);
Log.i("oppo push custom data", "key = " + key + ":value = " + value);
}
}
```

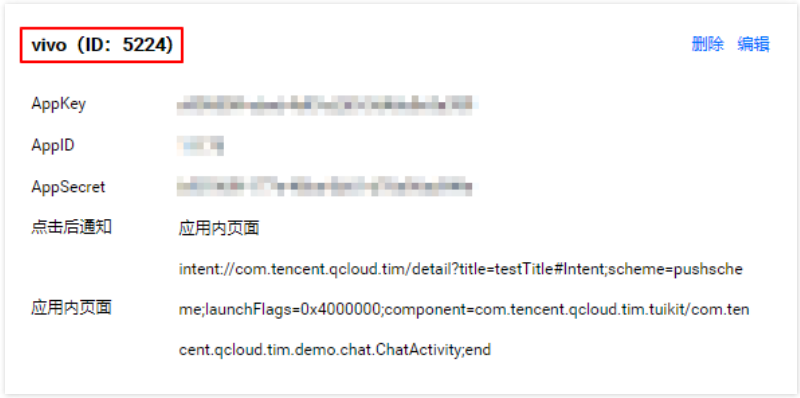
vivo 推送

配置推送证书

1. vivo 开放平台官网 进行注册并通过开发者认证。登录其开放平台的管理中心，选择【消息推送】>【创建】>【测试推送】，创建 vivo 推送服务应用。记录**APP ID**、**APP key**和**APP secret**信息。

2. 登录腾讯云 即时通信 IM 控制台，单击目标应用卡片，进入应用的基础配置页面，单击【Android平台推送设置】区域的【添加证书】。根据 步骤1 中获取的信息设置以下参数：
   - **推送平台**：选择 **vivo**
   - **AppKey**：填写 vivo 推送服务应用的 **APP key**
   - **AppID**：填写 vivo 推送服务应用的 **APP ID**
   - **AppSecret**：填写 vivo 推送服务应用的 **APP secret**
   - **点击通知后**：选择点击通知栏消息后的响应操作， 支持**打开应用**、**打开网页**和**打开应用内指定界面**，更多详情请参见 配置点击通知栏消息事件
     当设置为【打开应用】或【打开应用内指定界面】操作时，支持 透传自定义内容。

单击【确认】保存信息，记录证书的**ID**。证书信息保存后10分钟内生效。



集成推送 SDK

1. 请参考 vivo 推送集成指南 集成 SDK，并在 vivo 控制台测试通知消息，确保已成功集成。
2. 通过调用 `PushClient.getInstance(getApplicationContext()).initialize()` 来对 vivo 推送服务进行初始化，并调用 `PushClient.getInstance(getApplicationContext()).turnOnPush()` 启动推送，成功后您将在自定义的 `BroadcastReceiver` 的 `onReceiveRegId` 中收到 `regId`，`regId` 为当前设备上当前 App 的唯一标识。当登录 IM SDK 成功后，需要调用 setOfflinePushConfig 将**证书 ID** 和 **regId** 上报到即时通信 IM 服务端。

成功上报证书 ID 及 regId 后，即时通信 IM 服务端会在该设备上的即时通信 IM 用户 logout 之前、App 被 kill 之后将消息通过小米推送通知到用户端。

配置点击通知栏消息事件

您可以选择点击通知栏消息后**打开应用**、**打开网页**或**打开应用内指定界面**。

打开应用

默认为点击通知栏消息打开应用。

打开网页

您需要在 添加证书 时选择【打开网页】并输入以 http:// 或 https:// 开头的网址，例如 https://cloud.tencent.com/document/product/269 。

打开应用内指定界面

1. 在 manifest 中配置需要打开的 Activity 的 intent-filter ，示例代码如下，可以参考 Demo 的 AndroidManifest.xml：

```
<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. 获取 intent URL，方式如下：

```
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);

// 打印结果
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. 在 添加证书 时选择【打开应用内指定界面】并输入上述打印结果。

透传自定义内容

添加证书 时设置【点击通知后】为【打开应用】或【打开应用内指定界面】操作支持透传自定义内容。

步骤1：发送端设置自定义内容

在发消息前设置每条消息的通知栏自定义内容。

- 下面是 Android 端简单示例，也可以参考 TUIKit 中的 ChatManagerKit.java 类的 sendMessage() 方法中对应的逻辑：

```
JSONObject jsonObject = new JSONObject();
try {
jsonObject.put("extKey", "ext content");
} catch (JSONException e) {
e.printStackTrace();
}
String extContent = jsonObject.toString();

V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
v2TIMOfflinePushInfo.setExt(extContent.getBytes());
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, userID, null, V2TIMMessage.V2TIM_PRIORITY_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) {}
});
```

- 服务端示例请参见 OfflinePushInfo 的格式示例

步骤2：接收端获取自定义内容

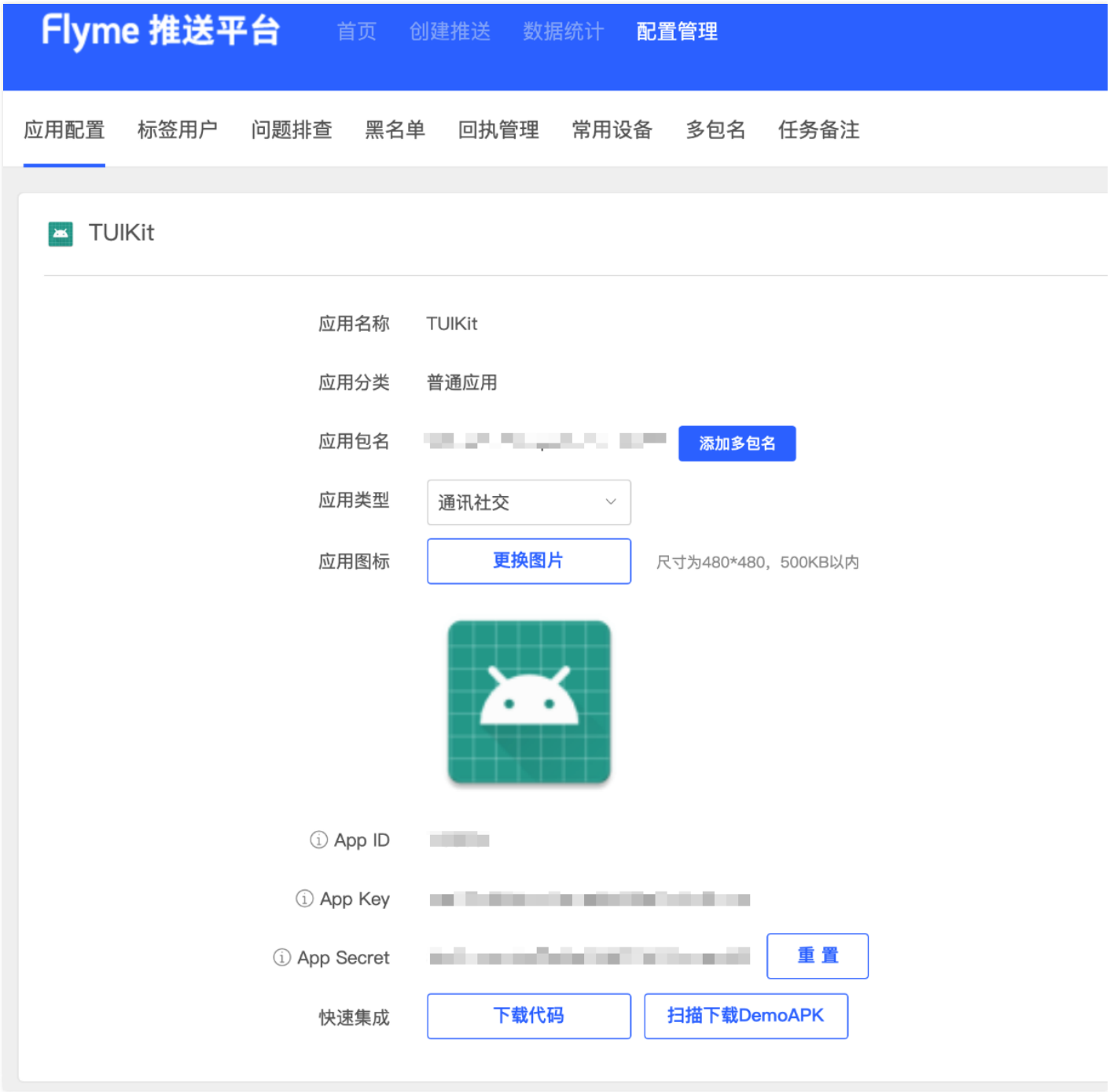
点击通知栏的消息时，会触发 vivo 推送 SDK 的 `onNotificationMessageClicked(Context context, UPSNotificationMessage upsNotificationMessage)` 回调，自定义内容可以从 `upsNotificationMessage` 中获取，可以参考 VIVOPushMessageReceiverImpl.java 的解析实现。

```
Map<String, String> paramMap = upsNotificationMessage.getParams();
String extContent = paramMap.get("ext");
```

魅族推送

配置推送证书

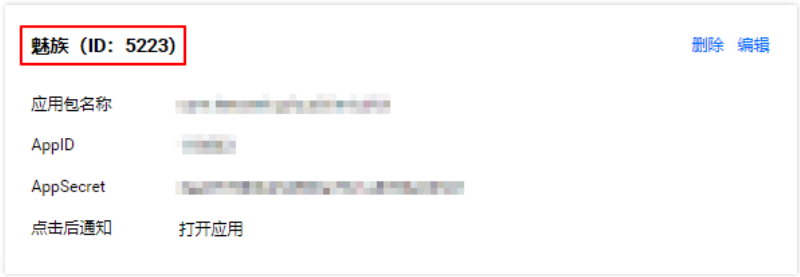
1. 打开 魅族开放平台官网 进行注册并通过开发者认证。登录其控制台，选择【开发服务】>【Flyme推送】，创建魅族推送服务应用。记录 **应用包名** 、 **App ID** 、 **App Secret** 信息。



2. 登录腾讯云 即时通信 IM 控制台，单击目标应用卡片，进入应用的基础配置页面，单击【Android平台推送设置】区域的【添加证书】。根据 步骤1 中获取的信息设置以下参数：

- **推送平台**：选择**魅族**
- **应用包名称**：填写魅族推送服务应用的**应用包名**
- **AppID**：填写魅族推送服务应用的 **App ID**
- **AppSecret**：填写魅族推送服务应用的 **App Secret**
- **点击通知后**：选择点击通知栏消息后的响应操作， 支持**打开应用**、**打开网页**和**打开应用内指定界面**，更多详情请参见 配置点击通知栏消息事件
  当设置为【打开应用】或【打开应用内指定界面】操作时，支持 透传自定义内容。

单击【确认】保存信息，记录证书的 **ID** 。证书信息保存后10分钟内生效。



集成推送 SDK

1. 请参考魅族推送接入 集成 SDK，并在其控制台测试通知消息，确保已成功集成。
2. 通过调用 PushManager.register 来对魅族推送服务进行初始化，注册成功后您将在自定义的 BroadcastReceiver 的 onRegisterStatus 中收到注册结果。其中 registerStatus.getPushId() 为当前设备上当前 App 的唯一标识。当登录 IM SDK 成功后，需要调用 setOfflinePushConfig 将**证书 ID** 和 **PushId** 上报到即时通信 IM 服务端。

成功上报证书 ID 及 regId 后，即时通信 IM 服务端会在该设备上的即时通信 IM 用户 logout 之前、App 被 kill 之后将消息通过小米推送通知到用户端。

配置点击通知栏消息事件

您可以选择点击通知栏消息后**打开应用**、**打开网页**或**打开应用内指定界面**。

打开应用

默认为点击通知栏消息打开应用。

打开网页

您需要在 添加证书 时选择【打开网页】并输入以 http:// 或 https:// 开头的网址，例如 https://cloud.tencent.com/document/product/269 。

打开应用内指定界面

您需要在 添加证书 时选择【打开应用内指定界面】并输入需要打开的 Activity 的完整类名，例如 com.tencent.qcloud.tim.demo.chat.ChatActivity 。

透传自定义内容

添加证书 时设置【点击通知后】为【打开应用】或【打开应用内指定界面】操作才支持透传自定义内容。

步骤1：发送端设置自定义内容

在发消息前设置每条消息的通知栏自定义内容。

- 下面是 Android 端简单示例，也可以参考 TUIKit 中的 ChatManagerKit.java 类的 sendMessage() 方法中对应的逻辑：

```
JSONObject jsonObject = new JSONObject();
try {
jsonObject.put("extKey", "ext content");
} catch (JSONException e) {
e.printStackTrace();
}
String extContent = jsonObject.toString();

V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
v2TIMOfflinePushInfo.setExt(extContent.getBytes());
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, userID, null, V2TIMMessage.V2TIM_PRIORITY_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) {}
});
```

- 服务端示例请参见 OfflinePushInfo 的格式示例

步骤2：接收端获取自定义内容

点击通知栏的消息时，会触发魅族推送 SDK 的 `onNotificationClicked(Context context, MzPushMessage mzPushMessage)` 回调 ，自定义内容可以从 `mzPushMessage` 中获取。

```
String extContent = mzPushMessage.getSelfDefineContentString();
```

另外，客户端也可以在打开的 `Activity` 中获取自定义内容，可以参考 OfflineMessageDispatcher.java 类的 parseOfflineMessage(Intent intent) 方法实现。

```
Bundle bundle = getIntent().getExtras();
String extContent = bundle.getString("ext");
```

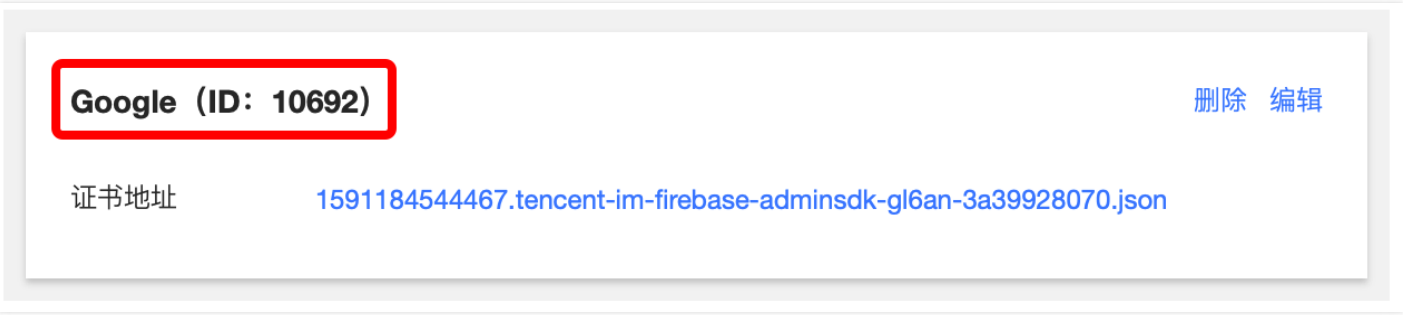
Google FCM 推送

集成 SDK

1. 打开 Firebase 云消息传递 注册账号并创建应用。
2. 登录 Firebase 控制台，单击您的应用卡片，进入应用配置页面。单击 Project Overview 右侧的  ，选择【项目设置】>【服务帐号】，单击【生成新的私钥】下载私钥文件。

3. 登录腾讯云 即时通信 IM 控制台，单击目标应用卡片，进入应用的基础配置页面，单击【Android平台推送设置】区域的【添加证书】。上传 步骤2 中获取的私钥文件。

4. 单击【确认】保存信息，记录证书的 **ID**。证书信息保存后10分钟内生效。



集成推送 SDK

1. 请参考 Firebase 云消息传递 设置 Firebase，集成 FCM SDK。参考 FCM 测试指引 测试通知消息，确保已成功集成 FCM。
2. 调用 `FirebaseInstanceId.getInstance().getInstanceId()` 后，在回调里获取当前 App 的唯一标识 token。当登录 IM SDK 成功后，需要调用 setOfflinePushConfig 将 **证书 ID** 和 **token** 上报到即时通信 IM 服务端。

成功上报证书 ID 及 regId 后，即时通信 IM 服务端会在该设备上的即时通信 IM 用户 logout 之前、App 被 kill 之后将消息通过 FCM 推送通知到用户端。

透传自定义内容

步骤1：发送端设置自定义内容
在发消息前设置每条消息的通知栏自定义内容。

- 下面是 Android 端简单示例，也可以参考 TUIKit 中 ChatManagerKit.java 类的 sendMessage() 方法中对应的逻辑：

```
JSONObject jsonObject = new JSONObject();
try {
jsonObject.put("extKey", "ext content");
} catch (JSONException e) {
e.printStackTrace();
}
String extContent = jsonObject.toString();

V2TIMOfflinePushInfo v2TIMOfflinePushInfo = new V2TIMOfflinePushInfo();
v2TIMOfflinePushInfo.setExt(extContent.getBytes());
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, userID, null, V2TIMMessage.V2TIM_PRIORITY_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) {}
});
```

- 服务端示例请参见 OfflinePushInfo 的格式示例

步骤2：接收端获取自定义内容

当点击通知栏的消息时，客户端在相应的 `Activity` 中获取自定义内容，可以参考 OfflineMessageDispatcher.java 类的 parseOfflineMessage(Intent intent) 方法实现。

```
Bundle bundle = getIntent().getExtras();
String value = bundle.getString("ext");
```

自定义 iOS 推送提示音

请在调用 sendMessage 发送消息的时候使用 `V2TIMOfflinePushInfo` 中的 setIOSSound 接口来设置 iOS 端的推送声音。

自定义离线推送展示

请在调用 sendMessage 发送消息的时候使用 `V2TIMOfflinePushInfo` 中的 setTitle 和 setDesc 接口来分别设置通知栏消息的标题和内容。

常见问题

Android 手机离线推送怎么自定义推送的声音？

目前大部分厂商都不支持离线推送声音的设置，因此 IM SDK 暂时不支持。

OPPO 手机为什么收不到离线推送？

OPPO 手机收不到推送一般有以下几种情况：

- 按照 OPPO 推送官网要求，在 Android 8.0 及以上系统版本的 OPPO 手机上必须配置 ChannelID，否则推送消息无法展示。配置方法可以参考 OPPO 推送配置。
- 在消息中 透传的离线推送的自定义内容 不是 JSON 格式，会导致 OPPO 手机收不到推送。

自定义消息为什么收不到离线推送？

自定义消息的离线推送和普通消息不太一样，自定义消息的内容我们无法解析，不能确定推送的内容，所以默认不推送，如果您有推送需求，需要您在 sendMessage 的时候设置 offlinePushInfo 的 desc 字段，推送的时候会默认展示 desc 信息。

离线推送（iOS）

最近更新时间：2020-08-31 16:55:50

配置离线推送

如想要接收 APNs 离线消息通知，需要遵从如下几个步骤：

1. 申请 APNs 证书。
2. 上传证书到 IM 控制台。
3. 在 App 每次登录时，向苹果获取 deviceToken。
4. 调用 setAPNS 接口将其上报到 IM 后台。

配置过 APNs 的 App ，当其切到后台或者被用户 Kill 之后，腾讯云就可以通过苹果的 APNs 后台对该设备进行离线消息推送，详细推送原理请参见 Apple Push Notification Service。

> 注意：
> 对于已经退出登录（主动登出或者被踢下线）的用户，不会收到任何消息通知。

步骤1：申请 APNs 证书

申请 APNs 证书的具体操作步骤请参见 Apple 推送证书申请。

步骤2：上传证书到控制台

1. 登录 即时通信 IM 控制台。
2. 单击目标应用卡片，进入应用的基础配置页面。
3. 单击【iOS平台推送设置】右侧的【添加证书】。
4. 选择证书类型，上传 iOS 证书（p.12），设置证书密码，单击【确认】。

> 注意：
> - 上传证书名最好使用全英文（尤其不能使用括号等特殊字符）。
> - 上传证书需要设置密码，无密码收不到推送。
> - 发布 App Store 的证书需要设置为生产环境，否则无法收到推送。
> - 上传的 p12 证书必须是自己申请的真实有效的证书。

5. 待推送证书信息生成后，记录证书的 ID。

步骤3：App 向苹果后台请求 DeviceToken

您可以在您的 App 中添加如下代码，用来向苹果的后台服务器获取 DeviceToken：

```
// 向苹果后台请求 DeviceToken
- (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)];
}
}
```

```
//在 AppDelegate 的回调中会返回 deviceToken，需要在登录后上报给腾讯云后台
-(void)application:(UIApplication *)app didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)deviceToken
{
//记录下 Apple 返回的 deviceToken
_deviceToken = deviceToken;
}
```

步骤4：登录 IM SDK 后上传 Token 到腾讯云

在 IM SDK 登录成功后，就可以调用 setAPNS 接口，将 步骤3 中获取的 DeviceToken 上传到腾讯云后台，实例代码如下：

```
V2TIMAPNSConfig *confg = [[V2TIMAPNSConfig alloc] init];
// 企业证书 ID，上传证书到 IM 控制台后生成
confg.businessID = businessID;
// 苹果后台请求的 deviceToken
confg.token = deviceToken;
[[V2TIMManager sharedInstance] setAPNS:confg succ:^{
NSLog(@"-----> 设置 APNS 成功");;
} fail:^(int code, NSString *msg) {
NSLog(@"-----> 设置 APNS 失败");
}];
```

注意：
businessID 需要与控制台分配的证书 ID 保持一致。

推送格式

推送格式示例如下图所示。

通用推送规则

对于单聊消息，APNs 推送规则如下，其中昵称是发送方用户昵称，如果未设置昵称，则只显示内容。

昵称:内容

对于群聊消息，APNs 推送规则如下，其中名称展示优先级为消息发送者的 群名片 > 群昵称 ，如果都没有，则不展示。

名称(群名):内容

不同类型消息推送规则

APNs 推送内容部分由消息体中各个 Elem 内容组成，不同 Elem 的离线消息展示效果如下表所示。

参数	说明
文本 Elem	直接显示内容
语音 Elem	显示 [语音]
文件 Elem	显示 [文件]
图片 Elem	显示 [图片]

参数	说明
自定义 Elem	显示发送消息时设置的 desc 的字段，如果 desc 不设置，则不进行推送

多 App 互通

如果将多个 App 中的 SDKAppID 设置为相同值，则可以实现多 App 互通。不同 App 需要使用不同的推送证书，您需要为每一个 App 申请 APNs 证书 并完成 离线推送配置。

自定义 iOS 推送提示音

请在调用 sendMessage 发送消息的时候设置 offlinePushInfo 的 iOSSound 字段， iOSSound 传语音文件名（带后缀），语音文件需要链接进 Xcode 工程。

自定义离线推送展示

请在调用 sendMessage 发送消息的时候设置 offlinePushInfo 的 title 和 desc 字段，其中 title 设置后，会在默认的推送内容上多展示 title 内容， desc 设置后，推送内容会变成 desc 内容。

自定义离线推送点击跳转逻辑

请在调用 sendMessage 发送消息的时候设置 offlinePushInfo 的 ext 字段，当用户收到离线推送启动 APP 的时候，可以在 AppDelegate -> didReceiveRemoteNotification 系统回调获取到 ext 字段，然后根据 ext 字段内容跳转到指定的 UI 界面。

本文以 “denny 给 vinson 发送消息” 的场景为例。

- 发送方：denny 需在发送消息时设置推送扩展字段 ext ：

```
// denny在发送消息时设置 offlinePushInfo，并指定 ext 字段
V2TIMMessage *msg = [[V2TIMManager sharedInstance] createTextMessage:@"文本消息"];
V2TIMOfflinePushInfo *info = [[V2TIMOfflinePushInfo alloc] init];
info.ext = @"jump to denny";
[[V2TIMManager sharedInstance] sendMessage:msg receiver:@"vinson" groupID:nil priority:V2TIM_PRIORITY_DEFAULT
onlineUserOnly:NO offlinePushInfo:info progress:^(uint32_t progress) {
} succ:^{
} fail:^(int code, NSString *msg) {
}];
```

- 接收方：vinson 的 App 虽然不在线，但可以接收到 APNS 离线推送，当 vinson 点击推送消息时会启动 App：

```
// vinson 启动 APP 后会收到以下回调
- (void)application:(UIApplication *)application didReceiveRemoteNotification:(NSDictionary *)userInfo
fetchCompletionHandler:(void (^)(UIBackgroundFetchResult result))completionHandler {
// 解析推送扩展字段 desc
if ([userInfo[@"ext"] isEqualToString:@"jump to denny"]) {
//跳转到和 denny 的聊天界面
}
}
```

常见问题

普通消息为什么收不到离线推送？

首先，请检查下 App 的运行环境和证书的环境是否一致，如果不一致，收不到离线推送。
其次，检查下 App 和证书的环境是否为开发环境，如果是开发环境，向苹果申请 deviceToken 可能会失败，生产环境暂时没有发现这个问题，请切换到生产环境测试。

自定义消息为什么收不到离线推送？

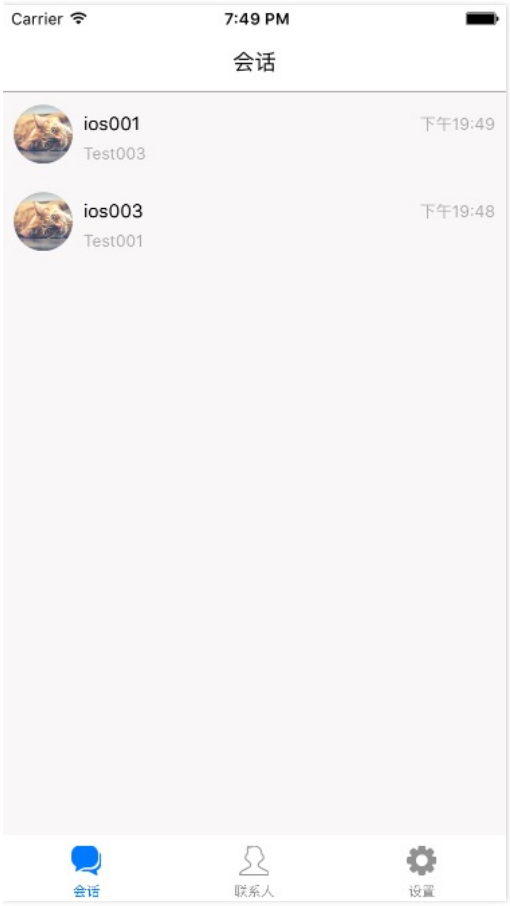
自定义消息的离线推送和普通消息不太一样，自定义消息的内容我们无法解析，不能确定推送的内容，所以默认不推送，如果您有推送需求，需要您在 sendMessage 的时候设置 offlinePushInfo 的 desc 字段，推送的时候会默认展示 desc 信息。

旧版 API 教程
概述
概述（Android）

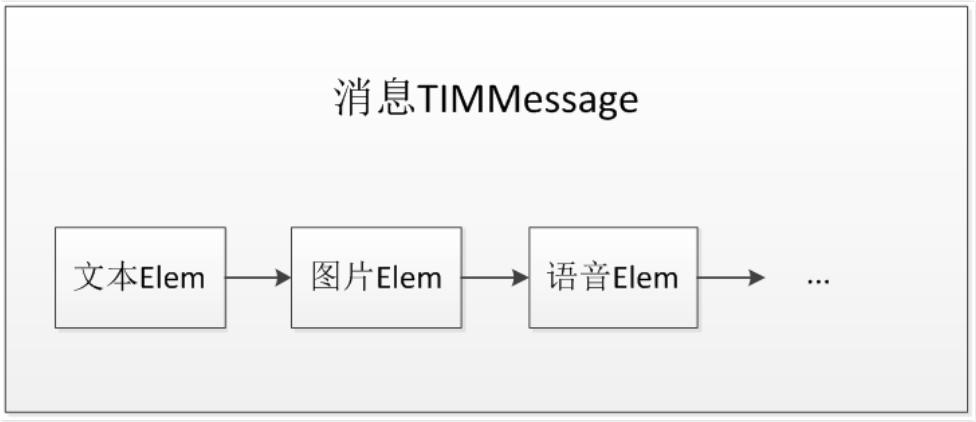
最近更新时间：2019-10-14 18:41:39

IM SDK 基本概念

会话：IM SDK 中会话（Conversation）分为两种，一种是 C2C 会话，表示单聊情况自己与对方建立的对话，读取消息和发送消息都是通过会话完成；另一种是群会话，表示群聊情况下，群内成员组成的会话，群会话内发送消息群成员都可接收到。如下图所示，一个会话表示与一个好友的对话。



消息：IM SDK 中消息(Message)表示要发送给对方的内容，消息包括若干属性，如是否自己已读，是否已经发送成功，发送人帐号，消息产生时间等；一条消息由若干 Elem 组合而成，每种 Elem 可以是文本、图片、表情等等，消息支持多种 Elem 组合发送。



群组 ID：群组 ID 唯一标识一个群，由后台生成，创建群组时返回。

IM SDK 对象简介

IM SDK 对象主要分为通讯管理器，会话、消息，群管理，具体的含义参见下表：

对象	介绍	功能
TIMManager	管理器类，负责 IM SDK 基本操作	初始化、登录、注销、创建会话等。
TIMConversation	会话，负责会话相关操作	如发送消息，获取会话消息缓存，获取未读计数等。
TIMMessage	消息	包括文本、图片等不同类型消息。
TIMGroupManager	群组管理器	负责创建群组、加群、退群等。
TIMFriendshipManager	资料和关系链管理器	负责资料获取、修改以及关系链等相关功能。

调用顺序介绍

IM SDK 调用 API 需要遵循以下顺序，其余辅助方法需要在登录成功后调用。

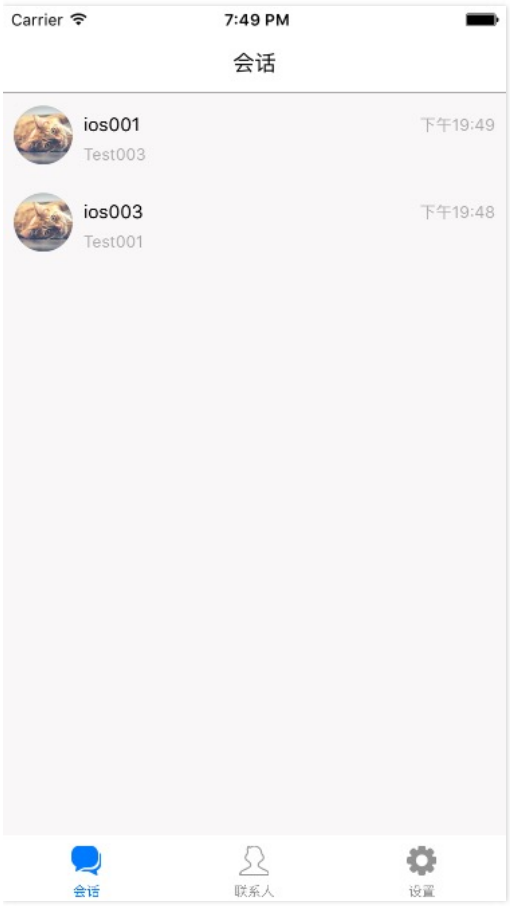
步骤	对应函数	说明
初始化	TIMSdkConfig	设置 IM SDK 基本配置，例如 SDKAppID、日志等级等
	TIMManager : init	初始化 IM SDK
	TIMManager : setUserConfig	设置用户基本配置
	TIMManager : addMessageListener	设置消息监听
登录	TIMManager : login	登录
消息收发	TIMManager : getConversation	获取会话
	TIMConversation : sendMessage	发送消息
群组管理	TIMGroupManager	群组管理
注销	TIMManager : logout	注销

概述（iOS）

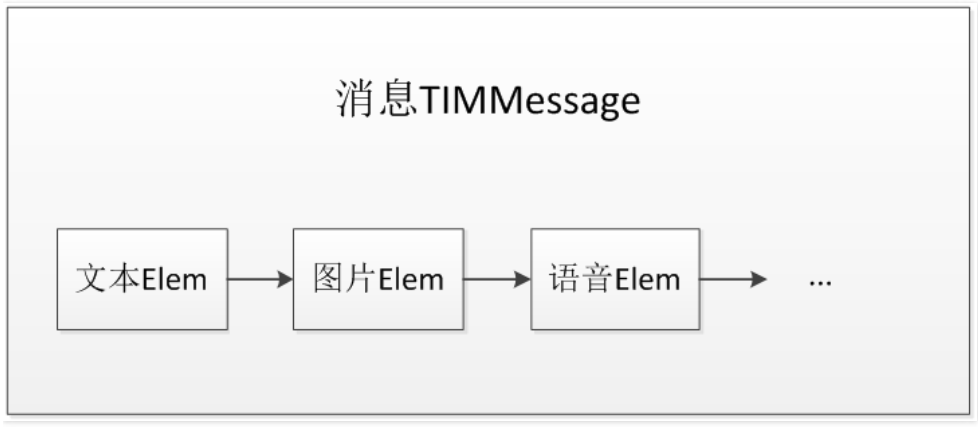
最近更新时间：2019-10-14 18:27:29

IM SDK 基本概念

会话：IM SDK 中会话（Conversation）分为两种，一种是 **C2C 会话**，表示单聊情况自己与对方建立的对话，读取消息和发送消息都是通过会话完成。另一种是**群会话**，表示群聊情况下，群内成员组成的会话，群会话内发送消息群成员都可接收到。如下图所示，一个会话表示与一个好友的对话。



消息：IM SDK 中消息（Message）表示要发送给对方的内容，消息包括若干属性，如是否自己已读，是否已经发送成功，发送人帐号，消息产生时间等。一条消息由若干 `Elem` 组合而成，每种 `Elem` 可以是文本、图片、表情等等，消息支持多种 `Elem` 组合发送。



群组 ID：群组 ID 唯一标识一个群，由后台生成，创建群组时返回。

IM SDK 对象简介

iOS IM SDK 对象主要分为通讯管理器、会话、消息、群管理，具体的含义参见下表。

对象	介绍	功能
TIMManager	管理器类	负责基本的 SDK 操作，包含初始化登录、注销、创建会话等
TIMConversation	会话	负责会话相关操作，包含发送消息、获取会话消息缓存、获取未读计数等
TIMMessage	消息	包含文本、图片等不同类型消息
TIMGroupManager	群管理器	负责创建群、增删成员、以及修改群资料等
TIMFriendshipManager	好友关系链管理器	负责添加、删除好友以及好友资料管理等

调用顺序介绍

IM SDK 调用 API 需要遵循以下顺序，其余辅助方法需要在登录成功后调用。

步骤	对应函数	说明
初始化	TIMManager:initSdk	设置 SDK 配置信息
初始化	TIMManager:setUserConfig	设置用户的配置信息
登录	TIMManager:login	登录
消息收发	TIMManager:getConversation	获取会话
消息收发	TIMConversation:sendMessage	发送消息
群组管理	TIMGroupManager	群组管理
关系链管理	TIMFriendshipManager	关系链管理
注销	TIMManager:logout	注销（用户可选）

概述（Web & 小程序）

最近更新时间：2020-06-24 10:49:18

TIM 是 IM Web SDK 的命名空间，提供了创建 SDK 实例的静态方法 create()，以及事件常量 EVENT，类型常量 TYPES

IM SDK 基本概念

基本概念	说明
Message（消息）	IM SDK 中 Message 表示要发送给对方的内容，消息包括若干属性，例如自己是否为发送者，发送人帐号以及消息产生时间等。
Conversation（会话）	IM SDK 中 Conversation 分为两种： • C2C（Client to Client）会话，表示单聊情况，自己与对方建立的对话。 • GROUP（群）会话，表示群聊情况下群内成员组成的会话。
Profile（资料）	IM SDK 中 Profile 描述个人的常用基本信息，例如昵称、头像地址、个性签名以及性别等。
Group（群组）	IM SDK 中 Group 表示一个支持多人聊天的通信系统，支持好友工作群（Work）、陌生人社交群（Public）、临时会议群（Meeting）和直播群（AVChatRoom）
GroupMember（群成员）	IM SDK 中 GroupMember 描述群内成员的常用基本信息，例如 ID、昵称、群内身份以及入群时间等。
群提示消息	当有用户被邀请加入群组或被移出群组等事件发生时，群内会产生提示消息，接入侧可以根据实际需求展示给群组用户或忽略。 群提示消息有多种类型，详细描述请参见 Message.GroupTipPayload。
群系统通知消息	当有用户申请加群等事件发生时，管理员会收到申请加群等系统消息。管理员同意或拒绝加群申请，IM SDK 会通过群系统通知消息将申请加群等相应消息发送给接入侧，由接入侧展示给用户。 群系统通知消息有多种类型，详细描述请参见 Message.GroupSystemNoticePayload。
消息上屏	用户单击发送后，事先输入的文字或选择的图片等信息显示在用户电脑屏幕或手机屏幕上的过程。

支持的平台

IM SDK 支持 IE 9+、Chrome、微信、手机 QQ、QQ 浏览器、FireFox、Opera 和 Safari。

调用顺序介绍

IM SDK 调用 API 需要遵循如下表所示顺序。

操作类型	值	含义
创建 SDK 实例	TIM.create(options)	通过 TIM 工厂函数创建 SDK 实例（通常用 tim 表示）。
设置日志级别	tim.setLogLevel(level)	设置日志级别，低于 level 的日志将不会输出。
注册插件	tim.registerPlugin(optoins)	注册上传图片、文件等需要用到 对象存储服务 COS 作为 IM SDK 的上传插件。
监听事件	tim.on(event, handler)	监听事件，在 handler 里处理 SDK 抛出来的数据。
登录	tim.login(options)	登录成功，SDK 状态为 ready 后，可收发消息。
创建文本消息	tim.createTextMessage(options)	创建文本消息。此接口返回一个消息实例，接入侧可将此消息做立即上屏处理。
发送消息	tim.sendMessage(message)	发送创建好的消息实例。
获取会话列表	tim.getConversationList()	获取会话列表，接入侧可处理会话列表数据，渲染会话列表界面。
获取群组列表	tim.getGroupList()	获取群组列表，接入侧可处理群组列表数据，渲染群组列表界面。
获取黑名单列表	tim.getBlacklist()	获取黑名单列表，接入侧可处理黑名单列表数据，渲染黑名单列表界面。
获取个人资料	tim.getMyProfile()	获取个人资料，接入侧可处理个人资料数据，渲染个人资料界面。

操作类型	值	含义
登出	tim.logout()	退出登录。

概述（Windows）

最近更新时间：2020-06-08 11:22:39

本文主要介绍腾讯云 IM SDK 的几个最基本功能的使用方法，阅读此文档有助于您对即时通信 IM 的基本使用流程有一个简单的认识。

初始化

在使用 SDK 进行即时通信 IM 操作之前，需要初始 SDK。

示例：

```
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());
```

SDKAppID 可以在即时通信 IM 控制台 创建应用后获取到。更多初始化操作请参考 初始化 文档。

登录/登出

登录

- 用户登录腾讯后台服务器后才能正常收发消息，登录需要用户提供 UserID、UserSig。详细请参阅 登录鉴权 文档。
- 登录为异步过程，通过回调函数返回是否成功，成功后方能进行后续操作。登录成功或者失败会主动调用提供的回调。

示例：

```
const void* user_data = nullptr; // 回调函数回传
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* user_data) {
if (code != ERR_SUCC) {
// 登入失败
return;
}
//登入成功

}, user_data);
```

> 注意：
> code 表示错误码，desc 表示错误描述，具体可参阅 错误码 文档。

onForceOffline

如果此用户在其他终端被踢，登录将会失败，返回错误码（ `ERR_IMSDK_KICKED_BY_OTHERS：6208` ），如果用户被踢了，请务必用 Alert 等提示窗提示用户，关于被踢的详细描述，参阅 用户状态变更。

onUserSigExpired

每一个 UserSig 都有一个过期时间，如果 UserSig 过期， `login` 将会返回 `70001` 错误码，如果您收到这个错误码，可以向您的业务服务器重新请求新的 UserSig，参阅 用户票据过期。

登出

如用户主动注销或需要进行用户的切换，则需要调用注销操作。

示例：

```
const void* user_data = nullptr; // 回调函数回传
TIMLogout([](int32_t code, const char* desc, const char* json_param, const void* user_data) {
if (code != ERR_SUCC) {
// 登出失败
return;
}
// 登出成功

}, user_data);
```

注意：
在需要切换帐号时，需要 Logout 回调成功或者失败后才能再次 Login ，否则 Login 可能会失败。
更多登录/登出操作请参考 登录登出 文档。

消息发送

会话获取

会话是指面向一个人或者一个群组的对话，通过与单个人或群组之间会话收发消息，发消息时首先需要先获取会话，获取会话需要指定会话类型（群组或者单聊），以及会话对方标志（对方帐号或者群号）。

获取对方 UserID 为 Windows-02 的单聊会话示例：

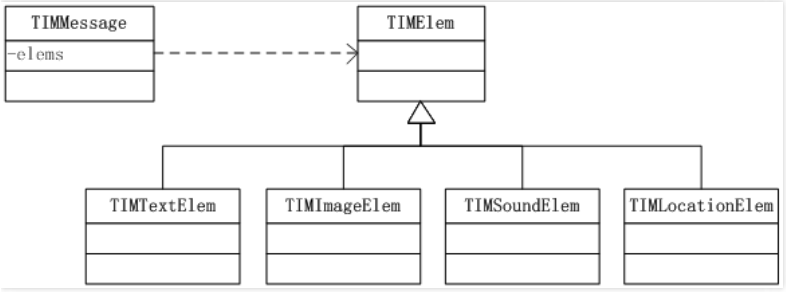
```
const void* user_data = nullptr; // 回调函数回传
const char* userid = "Windows-02";
int ret = TIMConvCreate(userid, kTIMConv_C2C, [](int32_t code, const char* desc, const char* json_param, const void* user_data) {
// 回调返回会话的具体信息
}, user_data);
if (ret != TIM_SUCC) {
// 调用 TIMConvCreate 接口失败
}
```

获取群组 ID 为 Windows-Group-01 的群聊会话示例：

```
const void* user_data = nullptr; // 回调函数回传
const char* groupid = "Windows-Group-01";
int ret = TIMConvCreate(groupid, kTIMConv_Group, [](int32_t code, const char* desc, const char* json_param, const void* user_data) {
// 回调返回会话的具体信息
}, user_data);
if (ret != TIM_SUCC) {
// 调用 TIMConvCreate 接口失败
}
```

消息发送

IM SDK 中的消息由 TIMMessage 表示，每个 TIMMessage 由多个 TIMElem 组成，每个 TIMElem 单元可以是文本，也可以是图片，也就是说每一条消息可包含多个文本、多张图片、以及其他类型的单元。



示例：

```
const void* user_data = nullptr; // 回调函数回传
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, c
onst void* user_data) {
if (code != ERR_SUCC) {
// 发送消息成功失败
return;
}
// 发送消息成功成功

}, user_data);
if (ret != TIM_SUCC) {
// 调用 TIMMsgSendNewMsg 接口失败
}
```

消息接收

在多数情况下，用户需要感知新消息的通知，这时只需注册新消息监听回调 `TIMSetRecvNewMsgCallback` ，在用户登录状态下，会拉取离线消息，为了不漏掉消息通知，需要在登录之前注册新消息监听回调。

设置消息监听示例：

```
// 设置新消息监听器，收到新消息时，通过此监听器回调
const void *user_data = nullptr;
TIMSetRecvNewMsgCallback([](const char* json_msg_array, const void* user_data) {
Json::Value json_value_msgs; // 解析消息
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++) { // 遍历Message
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++) { // 遍历Elem
Json::Value& elem = elems[i];

uint32_t elem_type = elem[kTIMElemType].asUInt();
if (elem_type == TIMElemType::kTIMElem_Text) { // 文本

}else if (elem_type == TIMElemType::kTIMElem_Sound) { // 声音

} else if (elem_type == TIMElemType::kTIMElem_File) { // 文件

} else if (elem_type == TIMElemType::kTIMElem_Image) { // 图片

}
}
}

}, user_data);
```

更多消息收发操作请参考 消息发送 和 消息接收。

群组管理

即时通信 IM 有多种群组类型，其特点以及限制因素可参考 群组类型介绍，群组使用唯一 ID 标识，通过群组 ID 可以进行不同操作，其中群组相关操作都由 `TIMGroupManager` 实现，需要用户登录成功后操作。

类型	说明
私有群（Private）	适用于较为私密的聊天场景，群组资料不公开，只能通过邀请的方式加入，类似于微信群。
公开群（Public）	适用于公开群组，具有较为严格的管理机制、准入机制，类似于 QQ 群。
聊天室（ChatRoom）	群成员可以随意进出。
直播聊天室（AVChatRoom）	与聊天室相似，但群成员人数无上限。
在线成员广播大群（BChatRoom）	适用于需要向全体在线用户推送消息的场景。

创建群组

以下示例创建一个叫 Windows-Group-Name 公开群组，并且把用户 Windows_002 拉入群组。
示例：

```
Json::Value json_group_member_array(Json::arrayValue);
// 初始群成员
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) {
// 创建群组失败
return;
}

// 创建群组成功 解析Json获取创建后的GroupID

}, user_data))
```

更多群组操作请参考 群组相关接口文档。

群组消息

群组消息与 C2C（单聊）消息相同，仅在发送时填写群组的 ID 和类型 `kTIMConv_Group` ，可参阅 SDK 文档 消息发送 部分。

初始化
初始化（Android）

最近更新时间：2020-05-19 18:31:04

获取通讯管理器

IM SDK 一切操作都是由通讯管理器 `TIMManager` 开始，IM SDK 操作第一步需要获取 `TIMManager` 单例。 `getInstance` 获取通讯管理器实例原型如下。

原型：

```
public static TIMManager getInstance()
```

示例：

```
TIMManager.getInstance();
```

初始化 IM SDK 配置

在初始化 IM SDK 之前，需要进行简单的 IM SDK 配置，包括 SDKAppID、日志控制等。对应的配置类为 `TIMSdkConfig` 。

日志事件

IM SDK 内部会进行打印日志，如果调用方有自己统一的日志收集方式，可以通过 `TIMSdkConfig` 中的 `setLogListener` 接口设置日志事件回调，把日志通过回调返给调用方，但 IM SDK 内部仍然会打印，如果需要禁掉，可以通过设置控制台不打印日志，或者设置日志级别。

原型：

```
/**
 * 设置当前日志回调监听器，必须在 IM SDK 初始化之前设置
 * @param logListener 日志回调监听器
 */
public TIMSdkConfig setLogListener(TIMLogListener logListener)
```

示例：

```
//设置日志回调，IM SDK 输出的日志将通过此接口回传一份副本
//[NOTE] 请注意 level 定义在 TIMManager 中，如 TIMManager.ERROR 等，并不同于 Android 系统定义
mTIMSdkConfig.setLogListener(new TIMLogListener() {
@Override
public void log(int level, String tag, String msg) {
//可以通过此回调将 sdk 的 log 输出到自己的日志系统中
}
});
```

设置日志级别

在权限允许的情况下，IM SDK 的日志默认会写到日志文件中。通过 `TIMSdkConfig` 中的 `setLogLevel` 接口修改 IM SDK 内部写日志级别可以控制 IM SDK 的文件日志输出。

注意：

- 设置写日志等级，**必须在 IM SDK 初始化之前调用**，在 IM SDK 初始化之后设置无效。
- 可以通过设置日志级别为 TIMLogLevel.OFF 来关闭 IM SDK 的文件日志输出，建议打开日志，方便排查问题。

原型：

```
/**
 * 设置写日志等级，必须在 IM SDK 初始化之前调用，在 IM SDK 初始化之后设置无效
```

```
 * @param logLevel 日志等级
 */
public TIMSdkConfig setLogLevel(@NonNull TIMLogLevel logLevel)
```

控制台不打印日志

默认 IM SDK 日志会打印到控制台，如果调试期间干扰太多，可选择通过 `TIMSdkConfig` 中的 `enableLogPrint` 关闭控制台日志（此时文件日志仍然会打印，可设置日志级别禁用）。

> 注意：
> 日志设置，**必须在 IM SDK 初始化之前调用**，在 IM SDK 初始化之后设置无效。

原型：

```
/**
 * 设置是否把日志输出到控制台，必须在 IM SDK 初始化之前设置
 * @param logPrintEnabled true - 日志将会输出到控制台
 */
public TIMSdkConfig enableLogPrint(boolean logPrintEnabled)
```

修改日志路径

为了方便统一管理日志，也可以修改默认的日志存储路径。通过 `TIMSdkConfig` 中的 `setLogPath` 接口可以设置日志文件存储路径。

> 注意：
> - 设置日志路径，**必须在 IM SDK 初始化之前调用**，在 IM SDK 初始化之后设置无效。
> - IM SDK 默认日志存储路径为：SD 卡下，`/tencent/imsdklogs/(your app package name)/`

原型：

```
/**
 * 设置日志路径，必须在 IM SDK 初始化之前调用，在 IM SDK 初始化之后设置无效
 * @param logPath 日志路径
 */
public TIMSdkConfig setLogPath(@NonNull String logPath)
```

初始化 IM SDK

在使用 IM SDK 进一步操作之前，需要初始化 IM SDK。

> 注意：
> 在存在**多进程**的情况下，请只在一个进程进行 IM SDK 初始化，调用接口 `SessionWrapper.isMainProcess(Context context)` 判断。

原型：

```
/**
 * 初始化 IM SDK
 * @param context application context
 * @param config IM SDK 全局配置
 * @return true - 初始化成功， false - 初始化失败
 */
public boolean init(@NonNull Context context, @NonNull TIMSdkConfig config)
```

示例：

```
//初始化 IM SDK 基本配置
//判断是否是在主线程
if (SessionWrapper.isMainProcess(getApplicationContext())) {
TIMSdkConfig config = new TIMSdkConfig(sdkAppId)
.enableCrashReport(false) //接口已废弃
.enableLogPrint(true)
.setLogLevel(TIMLogLevel.DEBUG)
.setLogPath(Environment.getExternalStorageDirectory().getPath() + "/justfortest/");

//初始化 SDK
TIMManager.getInstance().init(getApplicationContext(), config);
}
```

用户配置

在初始化 IM SDK 后，登录 IM SDK 之前，可以通过 TIMUserConfig 进行用户配置。配置完成后，**在登录前**，通过通讯管理器 `TIMManager` 的接口 `setUserConfig` 将用户配置与当前通讯管理器进行绑定。

原型：

```
/**
* 设置当前用户的用户配置，登录前设置
* @param userConfig 用户配置
*/
public void setUserConfig(TIMUserConfig userConfig)
```

示例：

```
//基本用户配置
TIMUserConfig userConfig = new TIMUserConfig()
//设置用户状态变更事件监听器
.setUserStatusListener(new TIMUserStatusListener() {
@Override
public void onForceOffline() {
//被其他终端踢下线
Log.i(tag, "onForceOffline");
}

@Override
public void onUserSigExpired() {
//用户签名过期了，需要刷新 userSig 重新登录 IM SDK
Log.i(tag, "onUserSigExpired");
}
})
//设置连接状态事件监听器
.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");
}
})
//设置群组事件监听器
.setGroupEventListener(new TIMGroupEventListener() {
@Override
```

```
public void onGroupTipsEvent(TIMGroupTipsElem elem) {
Log.i(tag, "onGroupTipsEvent, type: " + elem.getTipsType());
}
})
//设置会话刷新监听器
.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());
}
});

//禁用本地所有存储
userConfig.disableStorage();
//开启消息已读回执
userConfig.enableReadReceipt(true);

//将用户配置与通讯管理器进行绑定
TIMManager.getInstance().setUserConfig(userConfig);
```

网络事件通知

可选设置，如果要用户感知是否已经连接服务器，需要通过 `TIMUserConfig` 来设置此回调，用于通知调用者跟通讯后台链接的连接和断开事件，另外，如果断开网络，等网络恢复后会自动重连，自动拉取消息通知用户，用户无需关心网络状态，仅作通知之用。

> 注意：
> 这里的网络事件不表示用户本地网络状态，仅指明 IM SDK 是否与 IM 云 Server 连接状态。只要用户处于登录状态，**IM SDK 内部会进行断网重连，用户无需关心。**

原型：

```
/**
* 设置连接监听器
* @param listener 连接监听器
*/
public TIMUserConfig setConnectionListener(TIMConnListener listener)
```

示例：

请参考 用户配置 中的示例。

用户状态变更

用户状态变更的时候，IM SDK 会有相应的通知。通过 `TIMUserConfig` 设置用户状态变更通知监听器来对相应的通知进行监听。目前用户状态变更有两种通知，具体可参见 用户被踢下线通知 和 用户票据过期通知。

原型：

```
/**
* 设置用户状态通知回调
* @param userStatusListener 用户状态通知回调
*/
public TIMUserConfig setUserStatusListener(TIMUserStatusListener userStatusListener)
```

用户状态变更通知监听器 `TIMUserStatusListener` 的定义如下：

```
/**
* 用户状态变更通知监听器
*/
public interface TIMUserStatusListener {
```

```
/**
 * 被踢下线时回调
 */
public void onForceOffline();

/**
 * 票据过期时回调
 */
public void onUserSigExpired();
}
```

示例：

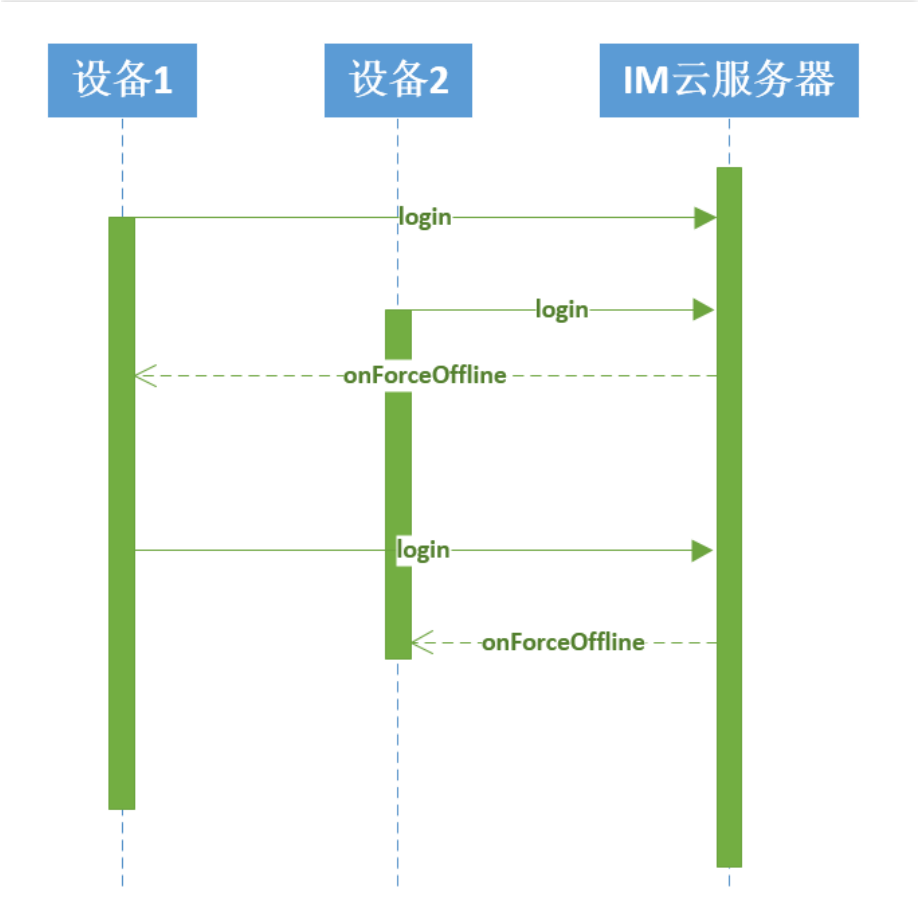
请参考 用户配置 中的示例。

用户被踢下线通知

用户如果在其他终端登录，会被踢下线，这时会收到用户被踢下线的通知。如果设置了用户状态变更通知监听器（参见 用户状态变更），则可以在监听器的回调方法 onForceOffline 中进行相应的处理，出现这种情况常规的做法是提示用户进行操作（退出，或者再次把对方踢下线）。

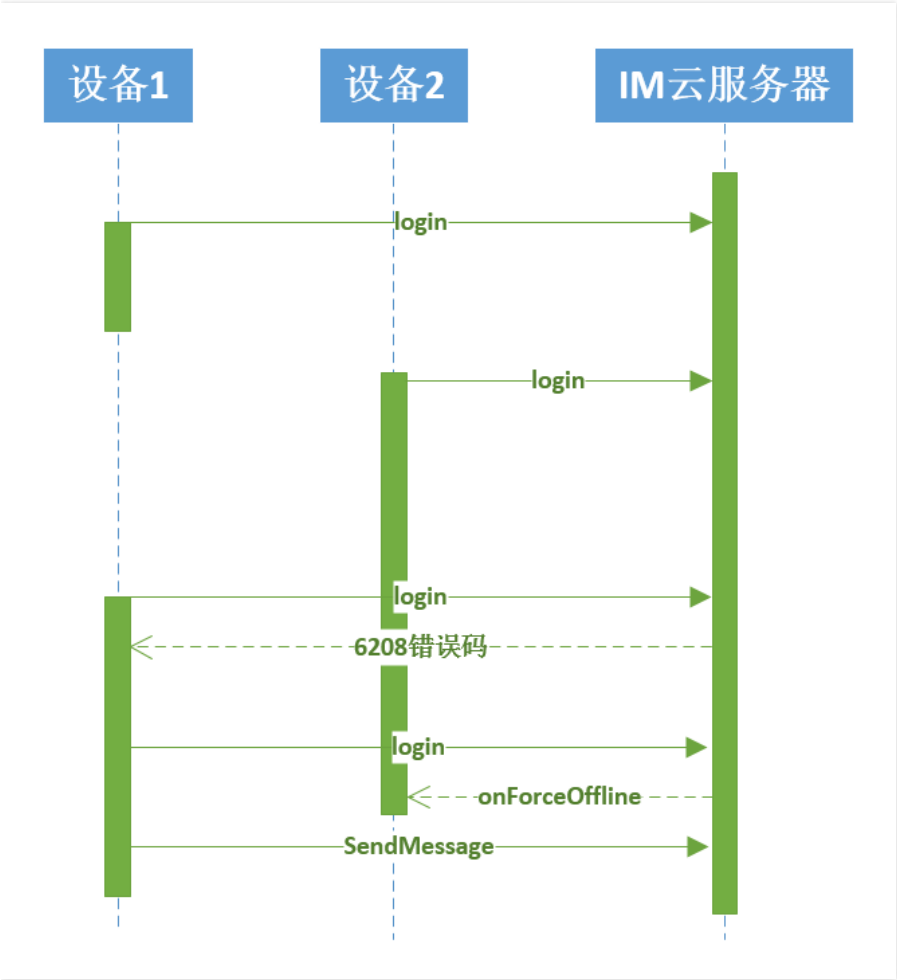
> 注意：
> 用户如果在离线状态下被踢，下次登录将会失败，可以给用户一个非常强的提醒（登录错误码 ERR_IMSDK_KICKED_BY_OTHERS：6208），开发者也可以选择忽略这次错误，再次登录即可。

用户在线情况下的互踢情况如下图所示。用户在设备 1 登录，保持在线状态下，该用户又在设备 2 登录，这时用户会在设备 1 上强制下线，收到 onForceOffline 回调。用户在设备 1 上收到回调后，提示用户，可继续调用 login 上线，强制设备 2 下线。这里是在线情况下互踢过程。



用户离线状态互踢如下图所示。用户在设备 1 登录，没有进行 logout 情况下杀掉应用进程。该用户在设备 2 登录，此时由于应用进程已不在了，设备 1 无法感知此事件，为了显式提醒用户，避免无感知的互踢，用户重新在设备 1 登录时，会返回（ ERR_IMSDK_KICKED_BY_OTHERS：6208 ）错误码，表明之前被踢，是否需要把对方踢下

线。如果需要，则再次调用 `login` 强制上线，设备2的登录的实例将会收到 `onForceOffline` 回调。



用户票据过期通知

在用户登录（参见 登录）的时候，需要提供一个用户票据，而这个用户票据在生成的时候是有一个有效使用期限的。在正常使用过程中，如果超过了用户票据的使用期限时，SDK 与服务器的交互会因为票据验证失败而操作失败，这个时候 SDK 会给出用户票据过期的通知。如果设置了用户状态变更通知监听器（参见 用户状态变更），则可以在监听器的回调方法 `onUserSigExpired` 中进行相应的处理，出现这种情况，如果仍需要继续与服务器进行交互，则需要更换票据后重新登录。

禁用存储

默认情况 IM SDK 会进行消息、资料、会话等存储，如无需存储，可选择通过 `TIMUserConfig` 关闭存储来提升处理性能。

> 注意：
> 禁用本地存储，**需要在登录之前调用。**

原型：

```
/**
* 禁用本地存储
*/
public TIMUserConfig disableStorage()
```

会话刷新监听

默认登录后会异步获取 C2C 离线消息、最近联系人以及同步资料数据（如果有开启 IM SDK 存储，可参见 关系链资料存储 及 群资料存储），同步完成后会通过会话刷新监听器 `TIMRefreshListener` 中的 `onRefresh` 回调通知更新界面，用户得到这个消息时，可以刷新界面，例如会话列表的未读等。

> 注意：
> 如果不需要离线消息，可以在发消息时 发送在线消息。

在多终端情况下，未读消息计数由 Server 下发同步通知，IM SDK 在本地更新未读计数后，通知用户更新会话。通知会通过 `TIMRefreshListener` 中的 `onRefreshConversation` 接口来进行回调，对于关注多终端同步的用户，可以在这个接口中进行相关的同步处理。所以建议在登录之前，通过 `TIMUserConfig` 中的 `setRefreshListener` 接口来设置会话刷新监听。

原型：

```
/**
 * 设置数据刷新通知监听器
 * @param listener 数据刷新通知监听器
 */
public TIMUserConfig setRefreshListener(TIMRefreshListener listener)
```

消息撤回通知监听

IM SDK 3.1.0 开始提供了消息撤回功能。通过 `TIMUserConfig` 的 `setMessageRevokedListener` 可以设置消息撤回通知监听器。

原型：

```
/**
 * 设置消息撤回通知监听器
 * @param listener 消息撤回通知监听器
 * @since 3.1.0
 */
public TIMUserConfig setMessageRevokedListener(@NonNull TIMMessageRevokedListener listener)
```

新消息通知

在多数情况下，用户需要感知新消息的通知，这时只需注册新消息通知回调 `TIMMessageListener` ，在用户登录的时候，会拉取 C2C 离线消息和最近联系人，为了不漏掉消息通知，建议在登录之前注册新消息通知。

> 注意：
> 只要是本地没有的消息，IM SDK 都会通过注册的消息通知回调给上层应用。

以下为添加一个消息监听器的原型。默认情况下所有消息监听器都将按添加顺序被回调一次。除非用户在 `onNewMessages` 回调中返回 true，此时将不再继续回调下一个消息监听器。

原型：

```
/**
 * 添加一个消息监听器
 * @param listener 消息监听器
 * 默认情况下所有消息监听器都将按添加顺序被回调一次
 * 除非用户在 onNewMessages 回调中返回 true，此时将不再继续回调下一个消息监听器
 */
public void addMessageListener(TIMMessageListener listener)
```

以下为收到新消息回调：

```
/**
 * 收到新消息回调
 * @param msgs 收到的新消息
 * @return 正常情况下，如果注册了多个listener, IM SDK会顺序回调到所有的listener。当碰到listener的回调返回true的时候，将终止继续回调后续的listener。
 */
public boolean onNewMessages(List<TIMMessage> msgs)
```

消息监听器被删除后，将不再被调用。**以下为删除一个消息监听器的原型：**

```
public void removeMessageListener(TIMMessageListener listener)
```

回调消息内容通过参数 `TIMMessage` 传递，通过 `TIMMessage` 可以获取消息和相关会话的详细信息，如消息文本，语音数据，图片等等，可参阅 消息解析 部分。

示例：

```
//设置消息监听器，收到新消息时，通过此监听器回调
TIMManager.getInstance().addMessageListener(new TIMMessageListener() {//消息监听器
@Override
public boolean onNewMessages(List<TIMMessage> msgs) {//收到新消息
//消息的内容解析请参考消息收发文档中的消息解析说明
return true; //返回true将终止回调链，不再调用下一个新消息监听器
}
});
```

初始化（iOS）

最近更新时间：2019-12-03 17:12:56

通讯管理器初始化

IM SDK 一切操作都是由通讯管理器 `TIMManager` 开始，SDK 操作第一步需要获取 `TIMManager` 单例。

原型：

```
@interface TIMManager : NSObject
/**
* 获取管理器实例
*
* @return 管理器实例
*/
+(TIMManager*)sharedInstance;
@end
```

示例：

```
TIMManager * manager = [TIMManager sharedInstance];
```

在使用 SDK 进一步操作之前，需要初始化 SDK。

原型:

```
@interface TIMManager : NSObject

/**
* 初始化 SDK
*
* @param config 配置信息，全局有效
*
* @return 0 成功
*/
- (int)initSdk:(TIMSdkConfig*)globalConfig;

/**
* 初始化当前 manager，在 initSdk:后调用，login:前调用
*
* @param config 配置信息，对当前 TIMManager 有效
*
* @return 0 成功
*/
- (int)setUserConfig:(TIMUserConfig*)config;

@end

//全局配置信息
@interface TIMSdkConfig : NSObject

//用户标识接入 SDK 的应用 ID，必填
@property(nonatomic,assign) int sdkAppId;

//禁止在控制台打印 log
@property(nonatomic,assign) BOOL disableLogPrint;

//本地写 log 文件的等级，默认 DEBUG 等级
@property(nonatomic,assign) TIMLogLevel logLevel;

//log 文件路径，不设置时为默认路径，可以通过 TIMManager -> getLogPath 获取 log 路径
@property(nonatomic,strong) NSString * logPath;

//回调给 logFunc 函数的 log 等级，默认 DEBUG 等级
@property(nonatomic,assign) TIMLogLevel logFuncLevel;
```

```
//log 监听函数
@property(nonatomic,copy) TIMLogFunc logFunc;

//消息数据库路径，不设置时为默认路径
@property(nonatomic,strong) NSString * dbPath;

//网络监听器，监听网络连接成功失败的状态
@property(nonatomic,strong) id<TIMConnListener> connListener;

@end

//用户配置信息
@interface TIMUserConfig : NSObject

//禁用本地存储
@property(nonatomic,assign) BOOL disableStorage;

//是否开启多终端同步未读提醒，这个选项主要影响多终端登录时的未读消息提醒逻辑。YES：只有当一个终端调用 setReadMessage() 将消息标记为已读，另一个终端再登录时才不会收到未读提醒；NO：消息一旦被一个终端接收，另一个终端都不会再次提醒。同理，卸载 App 再安装也无法再次收到这些未读消息。
@property(nonatomic,assign) BOOL disableAutoReport;

//是否开启被阅回执。YES：接收者查看消息（setReadMessage）后，消息的发送者会收到 TIMMessageReceiptListener 的回调提醒；NO: 不开启被阅回执，默认不开启。
@property(nonatomic,assign) BOOL enableReadReceipt;

//设置默认拉取的群组资料，如果想要拉取自定义字段，要通过即时通信 IM 控制台 > 功能配置 > 群维度自定义字段配置对应的 "自定义字段" 和用户操作权限，控制台配置之后5分钟后才会生效。
@property(nonatomic,strong) TIMGroupInfoOption * groupInfoOpt;

//设置默认拉取的群成员资料，如果想要拉取自定义字段，要通过即时通信 IM 控制台 > 功能配置 > 群成员维度自定义字段配置对应的 "自定义字段" 和用户操作权限，控制台配置之后5分钟后才会生效。
@property(nonatomic,strong) TIMGroupMemberInfoOption * groupMemberInfoOpt;

//关系链参数
@property(nonatomic,strong) TIMFriendProfileOption * friendProfileOpt;

//用户登录状态监听器,用于监听用户被踢，断网重连失败，UserSig 过期的通知
@property(nonatomic,weak) id<TIMUserStatusListener> userStatusListener;

//会话刷新监听器，用于监听会话的刷新
@property(nonatomic,weak) id<TIMRefreshListener> refreshListener;

//消息已读回执监听器，用于监听消息已读回执，enableReadReceipt 字段需要设置为 YES
@property(nonatomic,weak) id<TIMMessageReceiptListener> messageReceiptListener;

//消息修改监听器，用于监听消息状态的变化
@property(nonatomic,weak) id<TIMMessageUpdateListener> messageUpdateListener;

//消息撤回监听器，用于监听会话中的消息撤回通知
@property(nonatomic,weak) id<TIMMessageRevokeListener> messageRevokeListener;

//文件上传进度监听器，发送语音，图片，视频，文件消息的时候需要先上传对应文件到服务器，这里可以监听上传进度
@property(nonatomic,weak) id<TIMUploadProgressListener> uploadProgressListener;

//群组事件通知监听器
@property(nonatomic,weak) id<TIMGroupEventListener> groupEventListener;

//关系链数据本地缓存监听器
@property(nonatomic,weak) id<TIMFriendshipListener> friendshipListener;

@end
```

新消息通知

在多数情况下，用户需要感知新消息的通知，这时只需注册新消息通知回调 `TIMMessageListener` ，在用户登录状态下，会拉取离线消息，为了不漏掉消息通知，需要在登录之前注册新消息通知。

原型：

```
/**
 * 新消息接收回调
 */
@protocol TIMMessageListener <NSObject>
@optional
/**
 * 新消息回调通知
 *
 * @param msgs 新消息列表，TIMMessage 类型数组
 */
- (void)onNewMessage:(NSArray*) msgs;
@end

@interface TIMManager : NSObject

/**
 * 添加消息回调（重复添加无效）
 *
 * @param listener 回调
 *
 * @return 成功
 */
- (int)addMessageListener:(id<TIMMessageListener>)listener;

@end
```

回调消息内容通过参数 `TIMMessage` 传递，通过 `TIMMessage` 可以获取消息和相关会话的详细信息，如消息文本，语音数据，图片等。以下示例中设置消息回调通知，并且在有新消息时直接打印消息。详细可参阅 消息解析 部分。

示例：

```
@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];
```

网络事件通知

可选设置，如果要用户感知是否已经连接服务器，需要设置此回调，用于通知调用者跟通讯后台链接的连接和断开事件。另外，如果断开网络，等网络恢复后会自动重连，自动拉取消息通知用户，用户无需关心网络状态，仅作通知之用。

> **注意：**
> 这里的网络事件不表示用户本地网络状态，仅指明 SDK 是否与即时通信 IM 云 Server 连接状态。只要用户处于登录状态，**IM SDK 内部会进行断网重连，用户无需关心。**

原型：

```
/**
 * 连接通知回调
 */
@protocol TIMConnListener <NSObject>
```

```
@optional
/**
* 网络连接成功
*/
- (void)onConnSucc;
/**
* 网络连接失败
*
* @param code 错误码
* @param err 错误描述
*/
- (void)onConnFailed:(int)code err:(NSString*)err;
/**
* 网络连接断开（断线只是通知用户，不需要重新登录，重连以后会自动上线）
*
* @param code 错误码
* @param err 错误描述
*/
- (void)onDisconnect:(int)code err:(NSString*)err;
/**
* 连接中
*/
- (void)onConnecting;
@end
@interface TIMSdkConfig : NSObject
/**
* 网络监听器
*/
@property(nonatomic,retain) id<TIMConnListener> connListener;
@end
```

以下示例监听网络事件，并输出日志。

示例：

```
@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 错误码：具体参见错误码表
NSLog(@"Connect Failed: code=%d, err=%@", code, err);
}
- (void)onDisconnect:(int)code err:(NSString*)err {
    // code 错误码：具体参见错误码表
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];
```

日志事件

IM SDK 内部会打印日志，如果调用方有自己统一的日志收集方式，可以设置日志回调事件，SDK 会通过回调将日志返给调用方。回调通过两种形式，一种**使用闭包进行回调**，另一种**使用 protocol 接口回调。**

设置回调后 IM SDK 内部仍然会打印日志，如果需要禁用内部打印，可以设置控制台不打印日志，或者设置日志级别。

原型：

```
@interface TIMSdkConfig : NSObject
/**
* log监听函数
*/
@property(nonatomic,copy) TIMLogFunc logFunc;
@end
```

以下示例中通过闭包回调打印日志到控制台。

示例：

```
TIMSdkConfig * cfg = [[TIMSdkConfig alloc] init];
cfg.logFunc = ^(NSString* content) {
NSLog(@"%@", content);
}];
```

用户状态变更

用户状态变更的时候，SDK 会有相应的通知。通过设置 `TIMUserConfig` 中的 `userStatusListener` 属性可以设置用户状态变更通知监听器来对相应的通知进行监听。目前用户状态变更有三种通知，具体可参见 用户被踢下线通知 和 用户票据过期通知。这种情况下需要重新登录帐号后才能正常使用消息、群组和好友功能。

原型：

```
/**
* 用户在线状态通知
*/
@protocol TIMUserStatusListener <NSObject>
@optional
/**
* 踢下线通知
*/
- (void)onForceOffline;
/**
* 断线重连失败
*/
- (void)onReConnFailed:(int)code err:(NSString*)err;
/**
* 用户登录的 userSig 过期（用户需要重新获取 userSig 后登录）
*/
- (void)onUserSigExpired;
@end
@interface TIMUserConfig : NSObject
/**
* 用户登录状态监听器
*/
@property(nonatomic,retain) id<TIMUserStatusListener> userStatusListener;
@end
```

以下示例中当用户被踢下线时，收到回调后打印日志。**示例：**

```
@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];
```

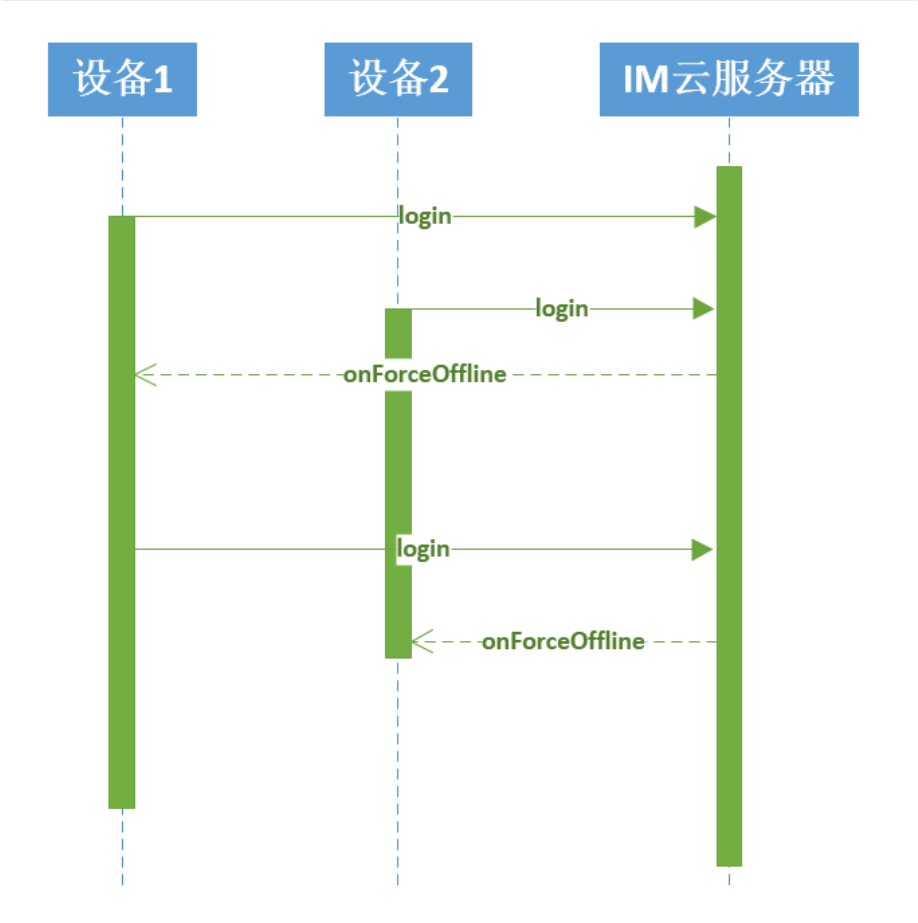
```
TIMUserConfig * cfg = [[TIMUserConfig alloc] init];
cfg.userStatusListener = impl;
```

用户被踢下线通知

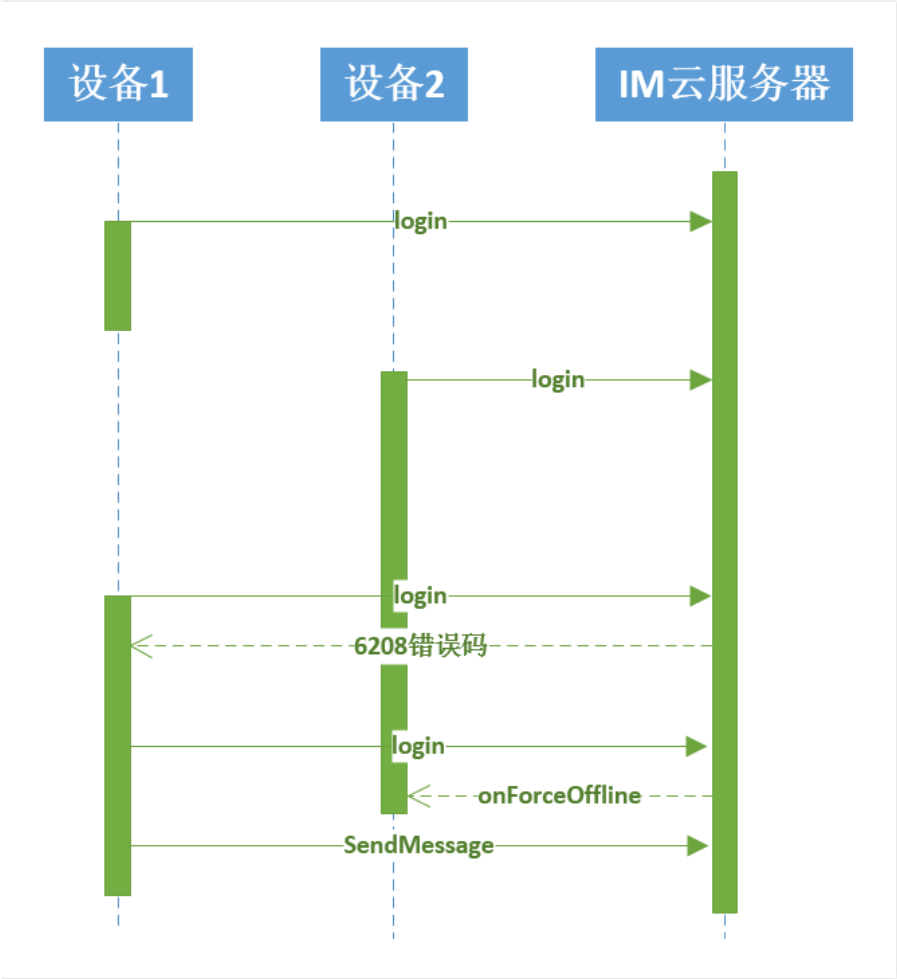
用户如果在其他终端调用 login 登录，会被踢下线，这时会收到用户被踢下线的通知。如果设置了用户状态变更通知监听器（参见 用户状态变更），则可以在监听器的回调方法 `onForceOffline` 中进行相应的处理，出现这种情况常规的做法是提示用户进行操作（退出，或者再次调用 login 把对方踢下线）。

> **注意：**
> 用户如果在离线状态下被踢，下次调用 login 登录将会失败，可以给用户一个非常强的提醒（登录错误码 `ERR_IMSDK_KICKED_BY_OTHERS：6208` ），开发者也可以选择忽略这次错误，再次登录即可。

用户在线情况下的互踢：如下图所示，用户在设备 1 调用 login 登录，保持在线状态下，该用户又在设备 2 调用 login 登录，这时用户会在设备 1 上强制下线，收到 `onForceOffline` 回调。用户在设备 1 上收到回调后，提示用户，可继续调用 `login` 上线，强制设备 2 下线。



用户离线状态互踢：如下图所示，用户在设备 1 调用 login 登录，没有进行 `logout` 情况下进程退出（此时可接收 iOS 远程推送消息）。该用户在设备 2 调用 login 登录，此时由于设备 1 用户不在线，无法感知此事件，为了显式提醒用户，避免无感知的互踢，用户在设备 1 重新调用 login 登录时，会返回 `ERR_IMSDK_KICKED_BY_OTHERS：6208` 错误码，表明之前被踢，是否需要把对方踢下线。如果需要，则再次调用 `login` 强制上线，设备 2 的登录的实例将会收到 `onForceOffline` 回调。



用户票据过期通知

在用户登录（参见 登录）的时候，需要提供一个用户票据，而这个用户票据在生成的时候是有一个有效使用期限的。在正常使用过程中，如果超过了用户票据的使用期限时，SDK 与服务器的交互会因为票据验证失败而操作失败，这个时候 SDK 会给出用户票据过期的通知。如果设置了用户状态变更通知监听器（参见 用户状态变更），则可以在监听器的回调方法 `onUserSigExpired` 中进行相应的处理，出现这种情况，如果仍需要继续与服务器进行交互，则需要更换票据后重新登录。

设置日志级别

IM SDK 内部日志级别可通过配置 `TIMSdkConfig` 进行修改，控制 IM SDK 的日志输出。可以通过设置日志级别为 `TIM_LOG_NONE` 来关闭 IM SDK 的日志输出，提升性能，建议在开发期间打开日志，方便排查问题。

原型：

```
@interface TIMSdkConfig : NSObject
/**
* 本地写 log 文件的等级，默认 DEBUG 等级
*/
@property(nonatomic,assign) TIMLogLevel logLevel;
@end
```

控制台不打印日志或修改日志路径

默认 IM SDK 日志会打印到控制台，如果调试期间干扰太多，可选择关闭控制台日志（此时文件日志仍然会打印，可设置日志级别禁用），另外也可以修改默认的存储路径，方便管理。如果不修改日志路径，仅修改级别，可使用 `getLogPath` 获取默认路径传入 `initLogSettings`。日志默认路径在 App 目录下：`Library/Caches/imsdk_YYYYMMDD.log`。

原型：

```
@interface TIMSdkConfig : NSObject
/**
* log 文件路径，不设置时为默认路径
*/
@property(nonatomic,retain) NSString * logPath;
/**
* 禁止在控制台打印 log
*/
@property(nonatomic,assign) BOOL disableLogPrint;
@end
```

登录
登录（Android）

最近更新时间：2019-11-19 11:41:50

登录

用户登录腾讯后台服务器后才能正常收发消息，登录需要用户提供 `UserID`、`UserSig` 等信息，具体含义可参阅 登录鉴权 。

> 注意：
>
> - 如果此用户在其他终端被踢，登录将会失败，返回错误码（ `ERR_IMSDK_KICKED_BY_OTHERS：6208` ）。开发者必须进行登录错误码 `ERR_IMSDK_KICKED_BY_OTHERS` 的判断。关于被踢的详细描述，请参考 用户状态变更。
> - 如果用户保存用户票据，可能会存在过期的情况，如果用户票据过期，`login` 将会返回 `70001` 错误码，开发者可根据错误码进行票据更换。

登录为异步过程，通过回调函数返回是否成功，成功后方能进行后续操作。

原型：

```
/** 登录
* @param identifier 用户帐号
* @param userSig userSig，用户帐号签名，由私钥加密获得，具体请参考文档
* @param callback 回调接口
*/
public void login(@NonNull String identifier, @NonNull String userSig, @NonNull TIMCallBack callback)
```

示例：

```
// identifier 为用户名，userSig 为用户登录凭证
TIMManager.getInstance().login(identifier, userSig, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 列表请参见错误码表
Log.d(tag, "login failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess() {
Log.d(tag, "login succ");
}
});
```

登出

如用户主动注销或需要进行用户的切换，则需要调用注销操作。

原型：

```
/**
* 注销
* @param callback 回调，不需要可以填 null
*/
public void logout(@Nullable TIMCallBack callback)
```

示例：

```
//登出
TIMManager.getInstance().logout(new TIMCallBack() {
```

```
@Override
public void onError(int code, String desc) {

//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 列表请参见错误码表
Log.d(tag, "logout failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess() {
//登出成功
}
});
```

无网络情况下查看消息

如用当前网络异常，或者想在不调用 `login` 的时候查看用户消息，可调用 `TIMManager` 中的 `initStorage` 方法初始化存储，完成后可获取会话列表和消息。

注意：

- 这个方法仅供登录失败或者没有网络的情况下查看历史消息使用，**如需要收发消息，请务必调用登录接口 `login`**。
- 如果登录成功，IM SDK 会自动初始化本地存储，无需手动调用这个接口。

原型：

```
/** 初始化本地存储，可以在无网络情况下加载本地会话和消息
* @param identifier 用户 ID
* @param cb 回调
*/
public int initStorage(@NonNull String identifier, @NonNull TIMCallBack cb)
```

以下示例中初始化存储，成功后可获取会话列表。

示例：

```
//初始化本地存储
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");
}
});

//获取会话实例
TIMConversation conversation = TIMManager.getInstance().getConversation(TIMConversationType.C2C, peer);
//获取本地消息
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());
}
});
```

获取当前登录用户

通过 `TIMManager` 成员方法 `getLoginUser` 可以获取当前用户名，也可以通过这个方法判断是否已经登录。

原型：

```
public String getLoginUser()
```

注意：
返回值为当前登录的用户名，如果是自有帐号登录，用户名与登录所传入的 `identifier` 相同，如果是第三方帐号，例如微信登录，QQ 登录等，登录后会有内部转换过的 `identifier` ，后续搜索好友，入群等，都需要使用转换后的 `identifier` 操作。

登录（iOS）

最近更新时间：2020-07-01 12:05:36

登录

用户登录腾讯后台服务器后才能正常收发消息，登录需要用户提供 `UserID` 、 `UserSig` 。如果用户保存用户票据，可能会存在过期的情况，如果用户票据过期， `login` 将会返回 `6206` 错误码，开发者可根据错误码进行票据更换。登录为异步过程，通过回调函数返回是否成功，成功后方能进行后续操作。登录成功或者失败后使用闭包 `succ` 和 `fail` 进行回调。

注意：

- 如果此用户在其他终端被踢，登录将会失败，返回错误码（ `ERR_IMSDK_KICKED_BY_OTHERS：6208` ）。开发者必须进行登录错误码 `ERR_IMSDK_KICKED_BY_OTHERS` 的判断。关于被踢的详细描述，参见 用户状态变更。
- 只要登录成功以后，用户没有主动登出或者被踢，网络变更会自动重连，无需开发者关心。不过特别需要注意被踢操作，需要注册 用户状态变更回调，否则被踢时得不到通知。

原型：

```
/**
* 登录信息
*/
@interface TIMLoginParam : NSObject
/**
* 用户名
*/
@property(nonatomic,retain) NSString* identifier;
/**
* 鉴权 Token
*/
@property(nonatomic,retain) NSString* userSig;

@end
```

```
/**
* 登录
*
* @param param 登录参数
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 请求成功
*/
- (int)login:(TIMLoginParam*)param succ:(TIMLoginSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
param	登录参数，详细信息参见 TIMLoginParam 结构说明
succ	登录成功回调
fail	登录失败回调

示例：

```
TIMLoginParam * login_param = [[TIMLoginParam alloc ]init];
// identifier 为用户名
login_param.identifier = @"iOS_001";
//userSig 为用户登录凭证
```

```
login_param.userSig = @"usersig";
[[TIMManager sharedInstance] login: login_param succ:^(){
NSLog(@"Login succ");
} fail:^(int code, NSString * err) {
NSLog(@"Login Failed: %d->%@", code, err);
}];
```

UserSig 正确的签发方式请参考 登录鉴权。

登出

如用户主动注销或需要进行用户的切换，则需要调用注销操作。

原型：

```
@interface TIMManager : NSObject
/**
* 登出
*
* @param succ 成功回调，登出成功
* @param fail 失败回调，返回错误码和错误信息
*
* @return 0 发送登出包成功，等待回调
*/
- (int)logout:(TIMLoginSucc)succ fail:(TIMFail)fail;
@end
```

示例：

> 注意：
> 在需要切换帐号时，需要 `logout` 回调成功或者失败后才能再次 `login`，否则 `login` 可能会失败。

```
[[TIMManager sharedInstance] logout:^() {
NSLog(@"logout succ");
} fail:^(int code, NSString * err) {
NSLog(@"logout fail: code=%d err=%@", code, err);
}];
```

无网络情况下查看消息

如用当前网络异常，或者想在不调用 `login` 的时候查看用户消息，可调用 `initStorage` 方法初始化存储，完成后可获取会话列表和消息。

原型：

```
@interface TIMManager : NSObject
/**
* 初始化存储，仅查看历史消息时使用，如果要收发消息等操作，如 login 成功，不需要调用此函数
*
* @param userID 用户名
* @param succ 成功回调，收到回调时，可以获取会话列表和消息
* @param fail 失败回调
*
* @return 0 请求成功
*/
- (int)initStorage:(NSString*)userID succ:(TIMLoginSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明

参数	说明
userID	用户名
succ	成功回调，成功后可获取会话列表，以及进一步登录
fail	失败回调

以下示例中初始化存储，成功后可获取会话列表。**示例：**

```
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);
}];
```

获取当前登录用户

通过 `TIMManager` 成员方法 `getLoginUser` 可以获取当前用户名，也可以通过这个方法判断是否已经登录。返回值为当前登录的用户名，需要注意的是，如果是自有帐号登录，用户名与登录所传入的 `UserID` 相同，如果是第三方帐号，如微信登录，QQ 登录等，登录后会有内部转换过的 `UserID` ，后续搜索好友，入群等，都需要使用转换后的 `UserID` 操作。

原型：

```
@interface TIMManager : NSObject

/**
* 获取当前登录的用户
*
* @return 如果登录返回用户的 identifier，如果未登录返回 nil
*/
- (NSString*)getLoginUser;

@end
```

IM SDK 同步离线消息

IM SDK 启动后会同步离线消息和最近联系人。如果不需要离线消息，可以在发消息时使用：发送在线消息。默认登录后会异步获取离线消息以及同步资料数据（如果有开启，可参见关系链资料章节），同步完成后会通过 `onRefresh` 回调通知更新界面，用户得到这个消息时，可以刷新界面，例如会话列表的未读等。

```
@interface TIMUserConfig : NSObject
/**
* 会话刷新监听器（未读计数、已读同步）
*/
@property(nonatomic,retain) id<TIMRefreshListener> refreshListener;
@end
```

群组管理
群组管理（Android）

最近更新时间：2020-06-08 12:08:56

群组综述

即时通信 IM 有多种群组类型，其特点以及限制因素请参见 群组系统。群组使用唯一 ID 标识，通过群组 ID 可以进行不同操作。

群组消息

群组消息与 C2C 消息相同，仅在获取 Conversation 时的会话类型不同，请参见 消息发送。

群组管理

群组相关操作都由 `TIMGroupManager` 实现，需要用户登录成功后操作。

获取单例原型：

```
/** 获取实例
 * @return TIMGroupManager 实例
 */
public static TIMGroupManager getInstance()
```

创建群组

即时通信 IM 中内置了**私有群（Private）、公开群（Public）、 聊天室（ChatRoom）、音视频聊天室（AVChatRoom）和在线成员广播大群（BChatRoom）**这几种群组类型，详情请参见 群组类型介绍。

- **音视频聊天室（AVChatRoom）：**也叫直播大群，此类型群对于加入人数不做限制，但是有一些能力上的限制，如不能拉人进去，不能查询总人数等。
- 可通过 `TIMGroupManager` 中的接口 `createGroup` 接口创建群组，创建时可指定一些群资料（例如群组类型、群组名称、群简介、加入的用户列表等，甚至可以指定群 ID），创建成功后返回群组 ID，可通过群组 ID 获取 Conversation 收发消息等。

> 注意：
> 自定义群组 ID 的时候，需要遵循一定的规则，具体请参考 自定义群组 ID。

原型：

```
/**
 * 创建群组
 * @param param 创建群组需要的信息集, 详见{@see CreateGroupParam}
 * @param cb 回调，OnSuccess 函数的参数中将返回创建成功的群组 ID
 */
public void createGroup(@NonNull CreateGroupParam param, @NonNull TIMValueCallBack<String> cb)
```

`TIMGroupManager.CreateGroupParam` 提供的接口如下：

```
/**
 * 创建群组参数类的构造函数
 * @param type 群类型, 目前支持的群类型：私有群（Private）、公开群（Public）、
 * 聊天室（ChatRoom）、音视频聊天室（AVChatRoom）和在线成员广播大群（BChatRoom）
 * @param name 群名称
 */
public CreateGroupParam(@NonNull String type, @NonNull String name)

/**
 * 设置要创建的群的群 ID
 * @param groupId 群 ID
```

```
*/
public CreateGroupParam setGroupId(String groupId)

/**
* 设置要创建的群的群公告
* @param notification 群公告
*/
public CreateGroupParam setNotification(String notification)

/**
* 设置要创建的群的群简介
* @param introduction 群简介
*/
public CreateGroupParam setIntroduction(String introduction)

/**
* 设置要创建的群的群头像 URL
* @param url 群头像 URL
*/
public CreateGroupParam setFaceUrl(String url)

/**
* 设置要创建的群的加群选项
* @param option 加群选项
*/
public CreateGroupParam setAddOption(TIMGroupAddOpt option)

/**
* 设置要创建的群允许的最大成员数
* @param maxMemberNum 最大成员数
*/
public CreateGroupParam setMaxMemberNum(long maxMemberNum)

/**
* 设置要创建的群的自定义信息
* @param key 自定义信息 key, 最长 16 字节
* @param value 自定义信息 value，最长 512 字节
*/
public CreateGroupParam setCustomInfo(String key, byte[] value)

/**
* 设置要创建的群的初始成员
* @param infos 初始成员的信息列表
*/
public CreateGroupParam setMembers(List<TIMGroupMemberInfo> infos)
```

示例：

```
//创建公开群，且不自定义群 ID
TIMGroupManager.CreateGroupParam param = new TIMGroupManager.CreateGroupParam("Public", "test_group");
//指定群简介
param.setIntroduction("hello world");
//指定群公告
param.setNotification("welcome to our group");

//添加群成员
List<TIMGroupMemberInfo> infos = new ArrayList<TIMGroupMemberInfo>();
TIMGroupMemberInfo member = new TIMGroupMemberInfo("cat");
infos.add(member);
param.setMembers(infos);

//设置群自定义字段，需要先到控制台配置相应的 key
try {
param.setCustomInfo("GroupKey1", "wildcat".getBytes("utf-8"));
} catch (UnsupportedEncodingException e) {
e.printStackTrace();
}
```

```
//创建群组
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);
}
});
```

邀请用户入群

`TIMGroupManager` 的接口 `inviteGroupMember` 可以拉（邀请）用户进入群组。

权限说明：

详情请参见 加群方式差异。

原型：

```
/**
* 邀请加入群组
* @param groupId 群组 ID
* @param memList 待加入群组的用户 ID 列表
* @param cb 回调，OnSuccess 函数的参数中返回成功加入群组的用户帐号
*/
public void inviteGroupMember(@NonNull String groupId, @NonNull List<String> memList,
@NonNull TIMValueCallBack<List<TIMGroupMemberResult>> cb)
```

TIMGroupMemberResult 接口定义如下：

```
/**
* 获取操作结果
* @return 操作结果 ： 0：失败；1：成功；2：添加成员时，该成员已经在群组中 或 删除成员时，该成员不在群组中
*/
public long getResult()

/**
* 获取用户帐号
* @return 用户帐号
*/
public String getUser()
```

示例：

```
//创建待加入群组的用户列表
ArrayList list = new ArrayList();

String user = "";

//添加用户
user = "sample_user_1";
list.add(user);
user = "sample_user_2";
list.add(user);
user = "sample_user_3";
list.add(user);

//回调
TIMValueCallBack<List<TIMGroupMemberResult>> cb = new TIMValueCallBack<List<TIMGroupMemberResult>>() {
@Override
public void onError(int code, String desc) {
}

@Override
```

```
public void onSuccess(List<TIMGroupMemberResult> results) { //群组成员操作结果
for(TIMGroupMemberResult r : results) {
Log.d(tag, "result: " + r.getResult() //操作结果: 0:添加失败；1：添加成功；2：原本是群成员
+ " user: " + r.getUser()); //用户帐号
}
}
};

//将 list 中的用户加入群组
TIMGroupManager.getInstance().inviteGroupMember(
groupId, //群组 ID
list, //待加入群组的用户列表
cb); //回调
```

申请加入群组

`TIMGroupManager` 的接口 `applyJoinGroup` 可以主动申请进入群组，此操作只对公开群、聊天室和音视频聊天室。

权限说明：

详情请参见 加群方式差异。

原型：

```
/**
* 加入群组
* @param groupId 群组 ID
* @param reason 申请理由（选填）
* @param cb 回调
*/
public void applyJoinGroup(@NonNull String groupId, String reason, @NonNull TIMCallBack cb)
```

以下示例中用户申请加入群组『@TGS#1JYSZEAEQ』，申请理由为『some reason』。**示例：**

```
TIMGroupManager.getInstance().applyJoinGroup("@TGS#1JYSZEAEQ", "some reason", new TIMCallBack() {
@java.lang.Override
public void onError(int code, String desc) {
//接口返回了错误码 code 和错误描述 desc，可用于原因
//错误码 code 列表请参见错误码表
Log.e(tag, "applyJoinGroup err code = " + code + ", desc = " + desc);
}

@java.lang.Override
public void onSuccess() {
Log.i(tag, "applyJoinGroup success");
}
});
```

退出群组

群组成员可以主动退出群组。退出群组的接口由 `TIMGroupManager` 提供。

权限说明：

- **私有群：**全员可退出群组。
- **公开群、聊天室和直播大群：**群主不能退出。

详情请参见 成员管理能力差异。

原型：

```
/**
* 退出群组
* @param groupId 群组 ID
* @param cb 回调
*/
public void quitGroup(@NonNull String groupId, @NonNull TIMCallBack cb)
```

示例：

```
//创建回调
TIMCallBack cb = new TIMCallBack() {
@Override
public void onError(int code, String desc) {
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 含义请参见错误码表
}

@Override
public void onSuccess() {
Log.e(tag, "quit group succ");
}
};

//退出群组
TIMGroupManager.getInstance().quitGroup(
groupId, //群组 ID
cb); //回调
```

删除群组成员

函数参数信息与加入群组相同，删除群组成员的接口由 `TIMGroupManager` 提供。

权限说明：
详情请参见 成员管理能力差异。

原型：

```
/**
* 删除群组成员
* @param param 删除群成员参数
* @param cb 回调，OnSuccess 函数的参数中返回成功删除的群成员列表
*/
public void deleteGroupMember(@NonNull DeleteMemberParam param,
@NonNull TIMValueCallBack<List<TIMGroupMemberResult>> cb)
```

`DeleteMemberParam` 接口定义如下：

```
/**
* 构造参数
* @param groupId 群 ID
* @param members 用户 ID 列表
*/
public DeleteMemberParam(@NonNull String groupId, @NonNull List<String> members)

/**
* 设置删除群成员的原因（选填）
* @param reason 删除原因
*/
public DeleteMemberParam setReason(@NonNull String reason)
```

示例：

```
//创建待踢出群组的用户列表
ArrayList list = new ArrayList();

String user = "";
//添加用户
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<TIMGroupMemberResult>>() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "deleteGroupMember onErr. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess(List<TIMGroupMemberResult> results) { //群组成员操作结果
for(TIMGroupMemberResult r : results) {
Log.d(tag, "result: " + r.getResult() //操作结果: 0：删除失败；1：删除成功；2：不是群组成员
+ " user: " + r.getUser()); //用户帐号
}
}
});
```

获取群成员列表

获取群成员列表的接口为 `getGroupMembers` ，默认拉取内置字段以及自定义字段，自定义字段要通过 即时通信 IM 控制台 >【功能配置】> 【群成员维度自定义字段】配置相关的 key 和权限，5分钟后生效。

权限说明：

- **任何群组类型：**可以获取成员列表。
- **直播大群：**只能拉取部分成员列表，包括群主、管理员和部分成员。

详情请参见 群组基础能力操作差异。

原型：

```
/**
* 获取群组成员列表
* @param groupId 群组 ID
* @param cb 回调，OnSuccess 函数的参数中返回群组成员列表
*/
public void getGroupMembers(@NonNull String groupId, @NonNull TIMValueCallBack<List<TIMGroupMemberInfo>> cb)
```

示例：

```
//创建回调
TIMValueCallBack<List<TIMGroupMemberInfo>> cb = new TIMValueCallBack<List<TIMGroupMemberInfo>> () {
@Override
public void onError(int code, String desc) {
}

@Override
public void onSuccess(List<TIMGroupMemberInfo> infoList) {//参数返回群组成员信息

for(TIMGroupMemberInfo info : infoList) {
Log.d(tag, "user: " + info.getUser() +
"join time: " + info.getJoinTime() +
"role: " + info.getRole());
}
}
};

//获取群组成员信息
TIMGroupManager.getInstance().getGroupMembers(
groupId, //群组 ID
cb); //回调
```

获取加入的群组列表

获取当前用户加入的所有群组的接口由 `TIMGroupManager` 提供。此接口可以获取自己所加入的群列表，返回的信息只包含部分基本信息，详细群组信息可以根据 群成员获取群组资料 进行获取。

权限说明：

- **私有群、公开群和聊天室：**支持使用本接口获取用户加入的公开群、聊天室、已激活的私有群信息。
- **音视频聊天室和在线成员广播大群：**因为内部实现的差异，获取用户加入的群组时不会获取到这两种类型的群组。

原型：

```
/**
 * 获取已加入的群组列表
 * @param cb 回调，OnSuccess 函数的参数中返回已加入的群组信息
 */
public void getGroupList(@NonNull TIMValueCallBack<List<TIMGroupBaseInfo>> cb)
```

TIMGroupBaseInfo 提供的方法如下：

```
/**
 * 获取群组 ID
 * @return 群组 ID
 */
public String getGroupId()

/**
 * 获取群组名称
 * @return 群组名称
 */
public String getGroupName()

/**
 * 获取群组类型
 * @return 群组类型
 */
public String getGroupType()

/**
 * 获取群头像 URL
 * @return 群头像 URL
 */
public String getFaceUrl()

/**
 * 获取当前群组是否设置了全员禁言
 * @return true - 设置了全员禁言
 * @since 3.1.1
 */
public boolean isSilenceAll()
```

示例：

```
//创建回调
TIMValueCallBack<List<TIMGroupBaseInfo>> cb = new TIMValueCallBack<List<TIMGroupBaseInfo>>() {
@Override
public void onError(int code, String desc) {
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 含义请参见错误码表
Log.e(tag, "get group list failed: " + code + " desc");
}

@Override
public void onSuccess(List<TIMGroupBaseInfo> timGroupInfos) {//参数返回各群组基本信息
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());
}
}
```

```
};

//获取已加入的群组列表
TIMGroupManager.getInstance().getGroupList(cb);
```

解散群组

解散群组的接口由 TIMGroupManager 提供。

权限说明：

详情请参见 群组基础能力操作差异。

原型：

```
/**
* 删除群组
* @param groupId 群组 ID
* @param cb 回调
*/
public void deleteGroup(@NonNull String groupId, @NonNull TIMCallBack cb)
```

示例：

```
//解散群组
TIMGroupManager.getInstance().deleteGroup(groupId, new TIMCallBack() {
@Override
public void onError(int code, String desc) {

//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 列表请参见错误码表
Log.d(tag, "login failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess() {
//解散群组成功
}
});
```

转让群组

转让群组的接口由 TIMGroupManager 提供。

权限说明：

只有**群主**可以进行群转让操作。

原型：

```
/**
* 群主变更
* @param groupId 群组 ID
* @param identifier 新群主的 identifier
* @param cb 回调
*/
public void modifyGroupOwner(@NonNull String groupId, @NonNull String identifier, @NonNull TIMCallBack cb)
```

其他接口

获取指定类型成员（可按照管理员、群主、普通成员拉取）接口定义如下：

```
/**
* 根据过滤条件获取群成员列表(支持按字段拉取，分页)
* @param groupId 群组 ID
* @param flags 拉取资料标志， 如{@see TIMGroupManager#TIM_GET_GROUP_MEM_INFO_FLAG_NAME_CARD}等标志的或组合位图
* @param filter 角色过滤类型，详见{@see TIMGroupMemberRoleFilter}
* @param custom 要获取的自定义 key 列表
* @param nextSeq 分页拉取标志，第一次拉取填0，回调成功如果不为零，需要分页，传入再次拉取，直至为0
```

```
 * @param cb 回调
 */
public void getGroupMembersByFilter(@NonNull String groupId, long flags, @NonNull TIMGroupMemberRoleFilter filter,
List<String> custom, long nextSeq, TIMValueCallBack<TIMGroupMemberSucc> cb)
```

获取群组资料

获取群组资料

TIMGroupManager 提供的 getGroupInfo 方法可以获取服务器存储的群组资料， queryGroupInfo 方法可以获取本地缓存的群组资料。群成员可以拉取到群组信息。非群成员无权限拉取私有群的信息，其他群类型仅可以拉取公开字段， groupId\groupName\groupOwner\groupType\createTime\memberNum\maxMemberNum\onlineMemberNum\groupIntroduction\groupFaceUrl\addOption\custom 。

说明：

默认拉取基本资料以及自定义字段，自定义字段要通过 即时通信 IM 控制台 >【功能配置】> 【群维度自定义字段】配置相关的 key 和权限，5分钟后生效。

原型：

```
/**
 * 获取服务器存储的群组信息
 * @param groupIdList 需要拉取详细信息的群组 ID 列表，一次最多 50 个
 * @param cb 回调，OnSuccess 函数的参数中返回群组信息{@see TIMGroupDetailInfo}列表
 */
public void getGroupInfo(@NonNull List<String> groupIdList,
@NonNull TIMValueCallBack<List<TIMGroupDetailInfo>> cb)
/**
 * 获取本地存储的群组信息
 * @param groupId 需要拉取详细信息的群组 ID
 * @return 群组信息，本地没有返回 null
 */
public TIMGroupDetailInfo queryGroupInfo(@NonNull String groupId)
```

TIMGroupDetailInfo 的接口定义如下：

```
/**
 * 获取群组 ID
 * @return 群组 ID
 */
public String getGroupId()

/**
 * 获取群组名称
 * @return 群组名称
 */
public String getGroupName()

/**
 * 获取群组创建者帐号
 * @return 群组创建者帐号
 */
public String getGroupOwner()

/**
 * 获取群组创建时间
 * @return 群组创建时间
 */
public long getCreateTime()

/**
 * 获取群组信息最后修改时间
 * @return 群组信息最后修改时间
 */
public long getLastInfoTime()
```

```
/**
 * 获取最新群组消息时间
 * @return 最新群组消息时间
 */
public long getLastMsgTime()

/**
 * 获取群组成员数量
 * @return 群组成员数量
 */
public long getMemberNum()

/**
 * 获取允许的最大群成员数
 * @return 最大群成员数
 */
public long getMaxMemberNum()

/**
 * 获取群简介内容
 * @return 群简介内容
 */
public String getGroupIntroduction()

/**
 * 获取群公告内容
 * @return 群公告内容
 */
public String getGroupNotification()

/**
 * 获取群头像 URL
 * @return 群头像 URL
 */
public String getFaceUrl()

/**
 * 获取群类型
 * @return 群类型
 */
public String getGroupType()

/**
 * 获取加群选项
 * @return 加群选项
 */
public TIMGroupAddOpt getGroupAddOpt()

/**
 * 获取群组内最新一条消息
 * @return 群组内最新一条消息
 */
public TIMMessage getLastMsg()

/**
 * 获取群组自定义字段 map
 * @return 群组自定义字段 map
 */
public Map<String, byte[]> getCustom()

/**
 * 获取此群组是否被设置了全员禁言
 * @return true - 群组被设置了全员禁言
 * @since 3.1.1
 */
public boolean isSilenceAll()
```

示例：

```
//创建待获取信息的群组 ID 列表
ArrayList<String> groupList = new ArrayList<String>();

//创建回调
TIMValueCallBack<List<TIMGroupDetailInfo>> cb = new TIMValueCallBack<List<TIMGroupDetailInfo>>() {
@Override
public void onError(int code, String desc) {
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 列表请参见错误码表
}

@Override
public void onSuccess(List<TIMGroupDetailInfo> infoList) { //参数中返回群组信息列表
for(TIMGroupDetailInfo info : infoList) {
Log.d(tag, "groupId: " + info.getGroupId() //群组 ID
+ " group name: " + info.getGroupName() //群组名称
+ " group owner: " + info.getGroupOwner() //群组创建者帐号
+ " group create time: " + info.getCreateTime() //群组创建时间
+ " group last info time: " + info.getLastInfoTime() //群组信息最后修改时间
+ " group last msg time: " + info.getLastMsgTime() //最新群组消息时间
+ " group member num: " + info.getMemberNum()); //群组成员数量
}
}
};

//添加群组 ID
String groupId = "TGID1EDABEAEO";
groupList.add(groupId);

//获取服务器群组信息
TIMGroupManager.getInstance().getGroupInfo(
groupList, //需要获取信息的群组 ID 列表
cb); //回调

//获取本地缓存的群组信息
TIMGroupDetailInfo timGroupDetailInfo = TIMGroupManager.getInstance().queryGroupInfo(groupId);
```

获取本人在群里的资料

如果需要获取本人在所在群内的资料，可以在通过 获取群组列表 拉取加入的群列表时得到。另外，如果需要单独获取某个群组，可使用以下 `TIMGroupManager` 提供的 `getSelfInfo` 获取。如果应用需要获取群组列表，建议在获取群组列表的时候获取个人在所在群内的资料，没有必要调用以下接口单独获取。

权限说明：

直播大群：无法获取本人在群内的资料。

原型：

```
/**
* 获取自己在群组中的信息
* @param groupId 群组 ID
* @param cb 回调，在 OnSuccess 函数的参数中返回自身信息
*/
public void getSelfInfo(@NonNull String groupId, @NonNull TIMValueCallBack<TIMGroupSelfInfo> cb)
```

获取群内某个人的资料

获取群成员资料的接口由 `TIMGroupManager` 提供，默认拉取基本资料。

权限说明：

直播大群只能获得部分成员的资料，包括群主、管理员和部分群成员。

原型：

```
/**
* 获取指定的群成员的群内信息
* @param groupId 指定群 ID
```

```
 * @param identifiers 指定群成员 identifier，一次最多 100 个
 * @param cb 回调，OnSuccess函数的参数中返回群组成员列表
 */
public void getGroupMembersInfo(@NonNull String groupId, @NonNull List<String> identifiers,
@NonNull TIMValueCallBack<List<TIMGroupMemberInfo>> cb)
```

修改群资料

修改群资料的接口由 `TIMGroupManager` 提供，可以对群名称、群简介、群公告等资料进行修改。

原型：

```
/**
 * 修改群组基本信息
 * @param param 参数类
 * @param cb 回调
 */
public void modifyGroupInfo(@NonNull ModifyGroupInfoParam param, @NonNull TIMCallBack cb)
```

`TIMGroupManager.ModifyGroupInfoParam` 接口定义如下：

```
/**
 * 构造参数实例
 * @param groupId 群 ID
 */
public ModifyGroupInfoParam(@NonNull String groupId)

/**
 * 设置修改后的群名称
 * @param groupName 群名称
 */
public ModifyGroupInfoParam setGroupName(@NonNull String groupName)

/**
 * 设置修改后的群公告
 * @param notification 群公告
 */
public ModifyGroupInfoParam setNotification(@NonNull String notification)

/**
 * 设置修改后的群简介
 * @param introduction 群简介
 */
public ModifyGroupInfoParam setIntroduction(@NonNull String introduction)

/**
 * 设置修改后的群头像 URL
 * @param faceUrl 群头像 URL
 */
public ModifyGroupInfoParam setFaceUrl(@NonNull String faceUrl)

/**
 * 设置加群选项
 * @param addOpt 加群选项
 */
public ModifyGroupInfoParam setAddOption(@NonNull TIMGroupAddOpt addOpt)

/**
 * 设置最大群成员数
 * @param maxMemberNum 最大群成员数
 */
public ModifyGroupInfoParam setMaxMemberNum(long maxMemberNum)

/**
 * 设置设置群组成员是否对外可见
```

```
 * @param visable 群组成员是否对外可见
 */
public ModifyGroupInfoParam setVisable(boolean visable)

/**
 * 设置群组自定义字段
 * @param customInfos 群组自定义字段字典
 */
public ModifyGroupInfoParam setCustomInfo(@NonNull Map<String, byte[]> customInfos)

/**
 * 设置群组全员禁言
 * @param silenceAll true - 设置全员禁言， false - 解除全员禁言
 * @since 3.1.1
 */
public ModifyGroupInfoParam setSilenceAll(boolean silenceAll)
```

修改群名

权限说明：

- **公开群、聊天室和直播大群：**只有群主或者管理员可以修改群名。
- **私有群：**任何人可修改群名。

示例：

```
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");
}
});
```

修改群简介

权限说明：

- **公开群、聊天室和直播大群：**只有群主或者管理员可以修改群简介。
- **私有群：**任何人可修改群简介。

示例：

```
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");
}
});
```

修改群公告

权限说明：

- **公开群、聊天室和直播大群：**只有群主或者管理员可以修改群公告。
- **私有群：**任何人可修改群公告。

示例：

```
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");
}
});
```

修改群头像

权限说明：

- **公开群、聊天室和直播大群：**只有群主或者管理员可以修改群头像。
- **私有群：**任何人可修改群头像。

示例：

```
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");
}
});
```

修改加群选项

权限说明：

- **公开群、聊天室和直播大群：**只有群主或者管理员可以修改加群选项。
- **私有群：**只能通过邀请加入群组，不能主动申请加入某个群组。

示例：

```
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");
}
});
```

修改群维度自定义字段

权限说明：

- 需要后台配置相关的 key 和权限。

示例：

```
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");
}
});
```

全员禁言

权限说明：

- **只有群主和管理员**才有权限进行全员禁言的操作。
- **所有群组类型**都支持全员禁言的操作。

示例：

```
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");
}
});
```

修改群成员资料

修改群成员资料的接口由 `TIMGroupManager` 提供，可以对修改群成员的身份、群名片、对群成员禁言等。

原型：

```
/**
* 修改群成员资料
* @param param 修改群成员资料参数
* @param cb 回调
*/
public void modifyMemberInfo(@NonNull ModifyMemberInfoParam param, @NonNull TIMCallBack cb)
```

`TIMGroupManager.ModifyMemberInfoParam` **接口定义如下：**

```
/**
 * 构造修改群成员资料参数
 * @param groupId 群成员所在群的群 ID
 * @param identifier 要修改的群成员的用户 ID
 */
public ModifyMemberInfoParam(@NonNull String groupId, @NonNull String identifier)

/**
 * 修改群成员群名片
 * @param nameCard 群名片
 */
public ModifyMemberInfoParam setNameCard(@NonNull String nameCard)

/**
 * 修改群消息接收选项
 * @param receiveMessageOpt 群消息接收选项，详见{@see TIMGroupReceiveMessageOpt}
 */
public ModifyMemberInfoParam setReceiveMessageOpt(@NonNull TIMGroupReceiveMessageOpt receiveMessageOpt)

/**
 * 修改群成员角色身份（只有群主和管理员可以修改）
 * @param roleType 身份类型。不能修改为群主类型，详见{@see TIMGroupMemberRoleType}
 */
public ModifyMemberInfoParam setRoleType(TIMGroupMemberRoleType roleType)

/**
 * 设置群成员的禁言时间（只有群主和管理员可以设置）
 * @param silence 禁言时间
 */
public ModifyMemberInfoParam setSilence(long silence)

/**
 * 设置群组自定义字段
 * @param customInfo 群组自定义字段字典
 */
public ModifyMemberInfoParam setCustomInfo(Map<String, byte[]> customInfo)
```

修改用户群内身份

权限说明：

- **只有群主或者管理员**可以进行对群成员的身份进行修改。
- **直播大群**不支持修改用户群内身份。

示例：

```
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");
}
});
```

对群成员进行禁言

通过 `modifyMemberInfoParam.setSilence()` 可以对群成员进行禁言并设置禁言时长。

权限说明：

- **只有群主或者管理员**可以进行对群成员进行禁言。

示例：

```
//禁言 100 秒
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");
}
});
```

修改群名片

示例：

```
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");
}
});
```

修改群成员维度自定义字段

示例：

```
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");
}
});
```

修改群消息接收选项

权限说明：

- **公开群和私有群：**默认消息接收方式为接收并离线推送群消息。
- **聊天室和音视频聊天室：**默认为接收但不离线推送群消息。

`TIMGroupReceiveMessageOpt` **接口定义如下：**

```
//不接收群消息， 服务器不会进行转发
TIMGroupReceiveMessageOpt.NotReceive

//接收群消息，但若离线情况下则不会推送离线消息
TIMGroupReceiveMessageOpt.ReceiveNotNotify

//接收群消息，若离线情况下会推送离线消息
TIMGroupReceiveMessageOpt.ReceiveAndNotify
```

示例：

```
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");
}
});
```

群组未决信息

群组未决信息泛指所有需要审批的群相关的操作。例如：加群待审批，拉人入群待审批等等。 群组未决信息由类 `TIMGroupPendencyItem` 表示。

`TIMGroupPendencyItem` **的成员方法如下：**

```
/**
* 获取群组 ID
* @return 群组 ID
*/
public String getGroupId()

/**
* 获取请求者 identifier，请求加群:请求者，邀请加群:邀请人
* @return 请求者identifier
*/
public String getFromUser()

/**
* 获取处理者 identifier, 请求加群:0，邀请加群:被邀请人
* @return 处理者 identifier
*/
public String getToUser()

/**
* 获取群未决添加的时间
* @return 群未决添加的时间
*/
public long getAddTime()
```

```
/**
 * 获取群未决请求类型
 * @return 群未决请求类型，详见 TIMGroupPendencyGetType
 */
public TIMGroupPendencyGetType getPendencyType()

/**
 * 获取群未决处理状态
 * @return 群未决处理状态，详见{@see TIMGroupPendencyHandledStatus}
 */
public TIMGroupPendencyHandledStatus getHandledStatus()

/**
 * 获取群未决处理操作类型，只有处理状态不为{@see TIMGroupPendencyHandledStatus#NOT_HANDLED}的时候有效
 * @return 群未决处理操作类型，详见{@see TIMGroupPendencyOperationType}
 */
public TIMGroupPendencyOperationType getOperationType()

/**
 * 获取请求者添加的附加信息
 * @return 请求者添加的附加信息
 */
public String getRequestMsg()

/**
 * 获取请求者添加的自定义信息
 * @return 请求者添加的自定义信息
 */
private String getRequestUserData()

/**
 * 获取处理者添加的附加信息，只有处理状态不为{@see TIMGroupPendencyHandledStatus#NOT_HANDLED}的时候有效
 * @return 处理者添加的附加信息
 */
public String getHandledMsg()

/**
 * 获取处理者添加的自定义信息，只有处理状态不为{@see TIMGroupPendencyHandledStatus#NOT_HANDLED}的时候有效
 * @return 处理者添加的自定义信息
 */
private String getHandledUserData()

/**
 * 同意申请，目前只对申请/邀请加群消息生效
 *
 * @param msg 同意理由，选填
 * @param cb 回调
 */
public void accept(String msg, TIMCallBack cb)

/**
 * 拒绝申请，目前只对申请/邀请加群消息生效
 *
 * @param msg 同意理由，选填
 * @param cb 回调
 */
public void refuse(String msg, TIMCallBack cb)
```

拉取群未决列表

通过 `TIMGroupManager` 提供的 `getGroupPendencyList` 接口可拉取群未决相关信息。即便审核通过或者拒绝后，该条信息也可通过此接口拉回，拉回的信息中有已决标志。

权限说明：

只有审批人有权限拉取相关信息。

例如：

- UserA 申请加入群 GroupA，则群管理员可获取此未决相关信息，UserA 因为没有审批权限，不需要获取未决信息。
- 如果 AdminA 拉 UserA 进去 GroupA，则 UserA 可以拉取此未决相关信息，因为该未决信息待 UserA 审批。

原型：

```
/**
 * 分页获取群未决请求列表
 * @param param 获取群未决请求列表参数类，详见{@see TIMGroupPendencyGetParam}
 * @param cb 回调，在 onSuccess 的参数中返回群未决的列表及元数据，详见{@see TIMGroupPendencyMeta} 及{@see TIMGroupPendencyItem}
 */
public void getGroupPendencyList(@NonNull TIMGroupPendencyGetParam param,
@NonNull TIMValueCallBack<TIMGroupPendencyListGetSucc> cb)
```

`TIMGroupPendencyGetParam` 的接口定义如下：

```
/**
 * 设置翻页时间戳，只用来翻页，第一次请求填 0，后边根据 server 返回的{@see TIMGroupPendencyMeta}中的时间戳进行填写
 * @param timestamp 翻页时间戳
 */
public void setTimestamp(long timestamp)

/**
 * 设置每页的数量（建议值，server 可根据需要返回或多或少，不能作为完成与否的标志）
 * @param numPerPage 每页的数量
 */
public void setNumPerPage(long numPerPage)
```

示例：

```
TIMGroupPendencyGetParam param = new TIMGroupPendencyGetParam();
param.setTimestamp(0);//首次获取填 0
param.setNumPerPage(10);

TIMGroupManager.getInstance().getGroupPendencyList(param, new TIMValueCallBack<TIMGroupPendencyListGetSucc>() {
@Override
public void onError(int code, String desc) {

}

@Override
public void onSuccess(TIMGroupPendencyListGetSucc timGroupPendencyListGetSucc) {
//meta中的nextStartTimestamp如果不为 0,可以先保存起来，
// 作为获取下一页数据的参数设置到 TIMGroupPendencyGetParam 中
TIMGroupPendencyMeta meta = timGroupPendencyListGetSucc.getPendencyMeta();
Log.d(tag, meta.getNextStartTimestamp()
+ "|" + meta.getReportedTimestamp() + "|" + meta.getUnReadCount());

List<TIMGroupPendencyItem> pendencyItems = timGroupPendencyListGetSucc.getPendencies();
for(TIMGroupPendencyItem item : pendencyItems){
//对群未决进行相应操作，例如查看/通过/拒绝等
}
}
});
```

上报群未决已读

对于未决信息，通过 `TIMGroupManager` 提供的 `reportGroupPendency` 可对其和之前的所有未决信息上报已读。上报已读后，仍然可以拉取到这些未决信息，但可通过对已读时戳的判断判定未决信息是否已读。

原型：

```
/**
 * 群未决请求已读上报
```

```
 * @param timestamp 已读时间戳(单位秒)，此时间戳以前的群未决请求都将置为已读
 * @param cb 回调
 */
public void reportGroupPendency(long timestamp, @NonNull TIMCallBack cb)
```

处理群未决信息

通过 `getGroupPendencyList` 获取到一个群未决请求（ `TIMGroupPendencyItem` ）的列表，对于列表中的每一个元素，都可以通过 `TIMGroupPendencyItem` 类中的 `accept/refuse` 接口来对群未决进行审批。已处理成功过的未决信息不能再次处理。

原型：

```
/**
 * 同意申请，目前只对申请/邀请加群消息生效
 *
 * @param msg 同意理由，选填
 * @param cb 回调
 */
public void accept(String msg, TIMCallBack cb)

/**
 * 拒绝申请，目前只对申请/邀请加群消息生效
 *
 * @param msg 同意理由，选填
 * @param cb 回调
 */
public void refuse(String msg, TIMCallBack cb)
```

群事件消息

当有用户被邀请加入群组，或者有用户被移出群组时，群内会产生提示消息，调用方可以根据需要展示给群组用户，或者忽略。提示消息使用一个特殊的 `Elem` 标识，通过新消息回调返回消息（请参见 新消息通知 ）。除了可以从新消息通知中获取群事件消息外，还可以在**登录前**通过 `TIMUserConfig` 中的 `setGroupEventListener` 接口设置群事件监听器来统一监听相应的事件（请参见 初始化（Android） ）。

> 注意：
> 聊天室（ChatRoom）和音视频聊天室（AVChatRoom）的群组事件消息不会通过新消息通知下发，只能通过注册群事件监听器对相应群事件进行监听。

如下图中，展示一条修改群名的事件消息。

`TIMGroupTipsElem` 成员方法：

```
//获取群资料变更信息列表，仅当 tipsType 值为 TIMGroupTipsType.ModifyGroupInfo 时有效
java.util.List<TIMGroupTipsElemGroupInfo> getGroupInfoList()
```

```
//获取群组名称
java.lang.String getGroupName()

//获取群成员变更信息列表，仅当 tipsType 值为 TIMGroupTipsType.ModifyMemberInfo 时有效
java.util.List<TIMGroupTipsElemMemberInfo> getMemberInfoList()

//获取操作者
java.lang.String getOpUser()

//获取群组事件通知类型
TIMGroupTipsType getTipsType()

//获取被操作的帐号列表
java.util.List<java.lang.String> getUserList()
```

TIMGroupTipsType 原型：

```
//取消管理员
CancelAdmin

//加入群组
Join

//被踢出群组
Kick

//修改群资料
ModifyGroupInfo

//修改成员信息
ModifyMemberInfo

//主动退出群组
Quit

//设置管理员
SetAdmin
```

用户加入群组通知

触发时机：当有用户加入群组时（包括申请入群和被邀请入群），群组内会由系统发出通知，开发者可选择展示样式。可以更新群成员列表。收到的消息 type 为 `TIMGroupTipsType.Join`。

TIMGroupTipsElem 成员方法返回说明：

方法	返回说明
getType	TIMGroupTipsType.Join
getOpUser	申请入群：申请人 邀请入群：邀请人
getGroupName	群名
getUserList	入群的用户列表

用户退出群组

触发时机：当有用户主动退群时，群组内会由系统发出通知。可以选择更新群成员列表。收到的消息 type 为 `TIMGroupTipsType.Quit`。

TIMGroupTipsElem 成员方法返回说明：

方法	返回说明
getType	TIMGroupTipsType.Quit
getOpUser	退出用户 identifier

方法	返回说明
getGroupName	群名

用户被踢出群组

触发时机：当有用户被踢出群组时，群组内会由系统发出通知。可以更新群成员列表。收到的消息 type 为 `TIMGroupTipsType.Kick`。

`TIMGroupTipsElem` 成员方法返回说明：

方法	返回说明
getType	TIMGroupTipsType.Kick
getOpUser	踢人的 identifier
getGroupName	群名
getUserList	被踢用户列表

被设置/取消管理员

触发时机：当有用户被设置为管理员或者被取消管理员身份时，群组内会由系统发出通知。如果界面有显示是否管理员，此时可更新管理员标识。收到的消息 type 为 `TIMGroupTipsType.SetAdmin` 和 `TIMGroupTipsType.CancelAdmin`。

`TIMGroupTipsElem` 成员方法返回说明：

方法	返回说明
getType	设置：TIMGroupTipsType.SetAdmin 取消：TIMGroupTipsType.CancelAdmin
getOpUser	操作用户 identifier
getGroupName	群名
getUserList	被设置/取消管理员身份的用户列表

群资料变更

触发时机：当群资料变更，如群名、群简介等，会有系统消息发出，可更新相关展示字段，或者选择性把消息展示给用户。

`TIMGroupTipsElem` 成员方法返回说明：

方法	返回说明
getType	TIMGroupTipsType.ModifyGroupInfo
getOpUser	操作用户 identifier
getGroupName	群名
getGroupInfoList	群变更的具体资料信息，为 TIMGroupTipsElemGroupInfo 结构体列表

`TIMGroupTipsElemGroupInfo` 原型：

```
//获取消息内容
java.lang.String getContent()

//获取群资料变更消息类型
TIMGroupTipsGroupInfoType getType()
```

`TIMGroupTipsGroupInfoType` 原型：

```
//修改群头像URL
ModifyFaceUrl

//修改群简介
ModifyIntroduction
```

```
//修改群名称
ModifyName

//修改群公告
ModifyNotification

//修改群主
ModifyOwner
```

群成员资料变更

触发时机：当群成员的群相关资料变更时，包括群内用户被禁言、群内成员角色变更，会有系统消息发出，可更新相关字段展示，或者选择性把消息展示给用户。

注意：

- **这里的资料仅包括群相关资料，例如禁言时间、成员角色变更等，不包括用户昵称等本身资料**，对于群内人数可能过多，不建议实时更新，建议的做法是直接显示消息体内的资料，参考：消息发送者以及相关资料。
- 如果本地有保存用户资料，可根据消息体内资料判断是否有变更，在收到此用户一条消息后更新资料。

TIMGroupTipsElem 成员方法返回说明：

方法	返回说明
getType	TIMGroupTipsType.ModifyMemberInfo
getOpUser	操作用户 identifier
getGroupName	群名
getMemberInfoList	变更的群成员的具体资料信息，为 TIMGroupTipsElemMemberInfo 结构体列表

TIMGroupTipsElemMemberInfo 原型：

```
//获取被禁言群成员的 identifier
java.lang.String getIdentifier()

//获取被禁言时间
long getShutupTime()
```

群系统消息

当有用户申请加群等事件发生时，管理员会收到邀请加群系统消息，用户可根据情况接受请求或者拒绝，相应的消息通过群系统消息展示给用户。

群系统消息类型定义：

```
//申请加群被同意（只有申请人能够收到）
TIM_GROUP_SYSTEM_ADD_GROUP_ACCEPT_TYPE

//申请加群被拒绝（只有申请人能够收到）
TIM_GROUP_SYSTEM_ADD_GROUP_REFUSE_TYPE

//申请加群请求（只有管理员会收到）
TIM_GROUP_SYSTEM_ADD_GROUP_REQUEST_TYPE

//取消管理员(被取消者接收)
TIM_GROUP_SYSTEM_CANCEL_ADMIN_TYPE

//创建群消息（初始成员能够收到）
TIM_GROUP_SYSTEM_CREATE_GROUP_TYPE

//群被解散（全员能够收到）
TIM_GROUP_SYSTEM_DELETE_GROUP_TYPE
```

```
//设置管理员(被设置者接收)
TIM_GROUP_SYSTEM_GRANT_ADMIN_TYPE

//邀请加群（被邀请者能够收到）
TIM_GROUP_SYSTEM_INVITED_TO_GROUP_TYPE

//被管理员踢出群（只有被踢的人能够收到）
TIM_GROUP_SYSTEM_KICK_OFF_FROM_GROUP_TYPE

//主动退群（主动退群者能够收到）
TIM_GROUP_SYSTEM_QUIT_GROUP_TYP

//群已被回收(全员接收)
TIM_GROUP_SYSTEM_REVOKE_GROUP_TYPE

//邀请入群请求（被邀请者接收）
TIM_GROUP_SYSTEM_INVITE_TO_GROUP_REQUEST_TYPE

//邀请加群被同意(只有发出邀请者会接收到)
TIM_GROUP_SYSTEM_INVITATION_ACCEPTED_TYPE

//邀请加群被拒绝(只有发出邀请者会接收到)
TIM_GROUP_SYSTEM_INVITATION_REFUSED_TYPE
```

TIMGroupSystemElem 成员方法定义如下：

```
/**
 * 操作方平台信息
 * 取值： iOS Android Windows Mac Web RESTAPI Unknown
 * @return 返回操作方平台信息
 */
public String getPlatform()

/**
 * 获取消息子类型
 * @return 群系统消息子类型
 */
public TIMGroupSystemElemType getSubtype()

/**
 * 获取消息群 ID
 * @return
 */
public String getGroupId()

/**
 * 获取操作人
 * @return 操作人的 identifier
 */
public String getOpUser()

/**
 * 获取操作理由
 * @return 操作理由
 */
public String getOpReason()

/**
 * 获取自定义通知
 * @return 自定义通知
 */
public byte[] getUserData()

/**
 * 获取操作者个人资料
 * @return 操作者个人资料
 */
```

```
public TIMUserProfile getOpUserInfo()

/**
 * 获取操作者群内资料
 * @return 操作者群内资料
 */
public TIMGroupMemberInfo getOpGroupMemberInfo()
```

申请加群消息

触发时机：当有用户申请加群时，群管理员会收到申请加群消息，可展示给用户，由用户决定是否同意对方加群。消息类型为：`TIM_GROUP_SYSTEM_ADD_GROUP_REQUEST_TYPE`。

TIMGroupSystemElem 成员方法返回说明：

方法	返回说明
getSubtype	TIM_GROUP_SYSTEM_ADD_GROUP_REQUEST_TYPE
getGroupId	群组 ID，表示是哪个群的申请
getOpUser	申请人
getOpReason	申请理由（可选）

申请加群同意/拒绝消息

触发时机：当管理员同意加群请求时，申请人会收到同意入群的消息，当管理员拒绝时，收到拒绝入群的消息。

`TIMGroupSystemElem` 成员方法返回说明：

方法	返回说明
getSubtype	同意：TIM_GROUP_SYSTEM_ADD_GROUP_ACCEPT_TYPE 拒绝：TIM_GROUP_SYSTEM_ADD_GROUP_REFUSE_TYPE
getGroupId	群组 ID，表示是哪个群通过/拒绝了
getOpUser	处理请求的管理员 identifier
getOpReason	同意或者拒绝理由（可选）

邀请入群请求消息

触发时机：当用户被邀请加入群组（**此时用户还没有加入到群组，需要用户审批**）时，该用户会收到邀请消息。

> 注意：
> 创建群组时初始成员无需邀请即可入群。

`TIMGroupSystemElem` 成员方法返回说明：

方法	返回说明
getSubtype	TIM_GROUP_SYSTEM_INVITE_TO_GROUP_REQUEST_TYPE
getGroupId	群组 ID，邀请进入哪个群
getOpUser	操作人，表示哪个用户的邀请

邀请入群同意/拒绝消息

触发时机：当被邀请者同意入群请求时，邀请者会收到同意入群的消息。当被邀请者拒绝时，邀请者收到拒绝入群的消息。

`TIMGroupSystemElem` 成员方法返回说明：

方法	返回说明

方法	返回说明
getSubtype	同意：TIM_GROUP_SYSTEM_INVITATION_ACCEPTED_TYPE 拒绝：TIM_GROUP_SYSTEM_INVITATION_REFUSED_TYPE
getGroupId	群组 ID，表示是哪个群通过/拒绝了
getOpUser	处理请求的管理员 identifier
getOpReason	同意或者拒绝理由（可选）

被管理员踢出群组

触发时机：当用户被管理员踢出群组时，申请人会收到被踢出群的消息。

`TIMGroupSystemElem` 成员方法返回说明：

方法	返回说明
getSubtype	TIM_GROUP_SYSTEM_KICK_OFF_FROM_GROUP_TYPE
getGroupId	群组 ID，表示在哪个群里被踢了
getOpUser	操作管理员 identifier

群被解散

触发时机：当群被解散时，全员会收到解散群消息。

`TIMGroupSystemElem` 成员方法返回说明：

方法	返回说明
getSubtype	TIM_GROUP_SYSTEM_DELETE_GROUP_TYPE
getGroupId	群组 ID，表示哪个群被解散了
getOpUser	操作管理员 identifier

创建群消息

触发时机：当群创建时，创建者会收到创建群消息。

当调用创建群方法成功回调后，即表示创建成功，此消息主要为多终端同步，如果有在其他终端登录，作为更新群列表的时机，本终端可以选择忽略。

`TIMGroupSystemElem` 成员方法返回说明：

方法	返回说明
getSubtype	TIM_GROUP_SYSTEM_CREATE_GROUP_TYPE
getGroupId	群组 ID，表示创建的群 ID
getOpUser	创建者，这里也就是用户自己

邀请入群消息

触发时机：当用户被邀请加入到群组（**此时用户已经加入到群组**）时，该用户会收到邀请消息。

> 注意：
> 创建群组时初始成员无需邀请即可入群。

`TIMGroupSystemElem` 成员方法返回说明：

方法	返回说明
getSubtype	TIM_GROUP_SYSTEM_INVITED_TO_GROUP_TYPE

方法	返回说明
getGroupId	群组 ID，邀请进入哪个群
getOpUser	操作人，表示哪个用户的邀请

主动退群

触发时机：当用户主动退出群组时，该用户会收到退群消息，只有退群的用户自己可以收到。

当用户调用 `QuitGroup` 时成功回调返回，表示已退出成功，此消息主要为了多终端同步，其他终端收到该消息时可以更新群列表，本终端可以选择忽略。

`TIMGroupSystemElem` 成员方法返回说明：

方法	返回说明
getSubtype	TIM_GROUP_SYSTEM_QUIT_GROUP_TYPE
getGroupId	群组 ID，表示退出的哪个群
getOpUser	操作人，这里即为用户自己

设置/取消管理员

触发时机：当用户被设置为管理员时，可收到被设置管理员的消息通知，当用户被取消管理员时，可收到取消通知，可提示用户。

`TIMGroupSystemElem` 成员方法返回说明：

方法	返回说明
getSubtype	取消管理员身份：TIM_GROUP_SYSTEM_GRANT_ADMIN_TYPE 授予管理员身份：TIM_GROUP_SYSTEM_CANCEL_ADMIN_TYPE
getGroupId	群组 ID，表示哪个群的事件
getOpUser	操作人

群被回收

触发时机：当群组被系统回收时，全员可收到群组被回收消息。

`TIMGroupSystemElem` 成员方法返回说明：

方法	返回说明
getSubtype	TIM_GROUP_SYSTEM_REVOKE_GROUP_TYPE
getGroupId	群组 ID，表示哪个群被回收了

群组管理（iOS）

最近更新时间：2020-06-08 12:08:38

群组综述

即时通信 IM 有多种群组类型，其特点以及限制因素可参考 群组系统。群组使用唯一 ID 标识，通过群组 ID 可以进行不同操作。

群组消息

群组消息与 C2C（单聊）消息相同，仅在获取 `Conversation` 时的会话类型不同，请参见 消息发送 。

群组管理

群组相关操作都由 `TIMGroupManager` 实现，需要用户登录成功后操作。

获取单例原型：

```
@interface TIMGroupManager : NSObject
+ (TIMGroupManager*)sharedInstance;
@end
```

创建内置类型群组

即时通信 IM 中内置了**私有群（Private）、公开群（Public）、 聊天室（ChatRoom）、音视频聊天室（AVChatRoom）和在线成员广播大群（BChatRoom）**群组类型，详情请参见 群组类型介绍。创建时可指定群组名称以及要加入的用户列表，创建成功后返回群组 ID，可通过群组 ID 获取 `Conversation` 收发消息等。

创建群组说明：

方法	说明
CreatePrivateGroup	创建私有群
CreatePublicGroup	创建公开群
CreateChatRoomGroup	创建聊天室
CreateAVChatRoomGroup	创建直播大群，此类型群可以加入人数不做限制，但是有一些能力上的限制，如不能拉人，不能查询总人数等

原型：

```
@interface TIMGroupManager : NSObject
/**
* 创建私有群
*
* @param members 群成员，NSString*数组
* @param groupName 群名
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)createPrivateGroup:(NSArray*)members groupName:(NSString*)groupName succ:(TIMCreateGroupSucc)succ fail:(TIMFail)fail;
/**
* 创建公开群
*
* @param members 群成员，NSString*数组
* @param groupName 群名
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
```

```
*/
- (int)createPublicGroup:(NSArray*)members groupName:(NSString*)groupName succ:(TIMCreateGroupSucc)succ fail:(TIMFail)fail;
/**
* 创建聊天室
*
* @param members 群成员，NSString*数组
* @param groupName 群名
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)createChatRoomGroup:(NSArray*)members groupName:(NSString*)groupName succ:(TIMCreateGroupSucc)succ fail:(TIMFail)fail;
/**
* 创建音视频聊天室（可支持超大群，详情可参考wiki文档）
*
* @param groupName 群名
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)createAVChatRoomGroup:(NSString*)groupName succ:(TIMCreateGroupSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
members	NSString 列表，指定加入群组的成员，创建者默认加入，无需指定（公开群、聊天室、私有群内最多6000人，直播大群没有限制）
groupName	NSString 类型，指定群组名称（最长 30 字节）
groupId	NSString 类型，指定群组 ID
succ	成功回调，返回群组 ID
fail	失败回调

以下示例创建一个私有群组，并且把用户『iOS_002』拉入群组。 **示例：**

说明：

- 创建者默认加入群组，无需显式指定。
- 公开群和聊天室调用方式和参数相同，仅方法名不同。

```
NSMutableArray * members = [[NSMutableArray alloc] init];
// 添加一个用户 iOS_002
[members addObject:@"iOS_002"];
[[TIMGroupManager sharedInstance] createPrivateGroup:members groupName:@"GroupName" succ:^(NSString * group) {
NSLog(@"create group succ, sid=%@", group);
} fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

创建指定属性群组

在创建群组时，除了设置默认的成员以及群名外，还可以设置如群公告、群简介等字段。

```
/**
* 创建群参数
*/
@interface TIMCreateGroupInfo : TIMCodingModel
/**
* 群组 ID,nil 则使用系统默认 ID
```

```
*/
@property(nonatomic,retain) NSString* group;
/**
* 群名
*/
@property(nonatomic,retain) NSString* groupName;
/**
* 群类型：Private,Public,ChatRoom,AVChatRoom
*/
@property(nonatomic,retain) NSString* groupType;
/**
* 是否设置入群选项，Private 类型群组请设置为 false
*/
@property(nonatomic,assign) BOOL setAddOpt;
/**
* 入群选项
*/
@property(nonatomic,assign) TIMGroupAddOpt addOpt;
/**
* 最大成员数，填 0 则系统使用默认值
*/
@property(nonatomic,assign) uint32_t maxMemberNum;
/**
* 群公告
*/
@property(nonatomic,retain) NSString* notification;
/**
* 群简介
*/
@property(nonatomic,retain) NSString* introduction;
/**
* 群头像
*/
@property(nonatomic,retain) NSString* faceURL;
/**
* 自定义字段集合,key 是 NSString* 类型，value 是 NSData* 类型
*/
@property(nonatomic,retain) NSDictionary* customInfo;
/**
* 创建成员（TIMCreateGroupMemberInfo*）列表
*/
@property(nonatomic,retain) NSArray* membersInfo;
@end

@interface TIMGroupManager : NSObject
/**
* 创建群组
*
* @param groupInfo 群组信息
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)createGroup:(TIMCreateGroupInfo*)groupInfo succ:(TIMCreateGroupSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
groupInfo	可设置群 ID、群名、群类型、入群选项、最大成员数、群公告、群简介、群头像等
succ	成功回调
fail	失败回调

以下示例创建一个指定属性的私有群，并把用户 『iOS_001』加入群组，创建者默认加入群，无需显示指定。**示例：**

```
// 创建群组信息
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;
// 创建群成员信息
TIMCreateGroupMemberInfo *memberInfo = [[TIMCreateGroupMemberInfo alloc] init];
memberInfo.member = @"iOS_001";
memberInfo.role = TIM_GROUP_MEMBER_ROLE_ADMIN;
// 添加群成员信息
NSMutableArray *membersInfo = [[NSMutableArray alloc] init];
[membersInfo addObject:memberInfo];
groupInfo.membersInfo = membersInfo;
// 创建指定属性群组
[[TIMGroupManager sharedInstance] createGroup:groupInfo succ:^(NSString * group) {
NSLog(@"create group succ, sid=%@", group);
} fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

自定义群组 ID 创建群组

默认创建群组时，通讯云 IM 服务器会生成一个唯一的 ID，以便后续操作，另外，如果用户需要自定义群组 ID，在创建时可指定 ID，通过 创建指定属性群组 也可以实现自定义群组 ID 的功能。

```
@interface TIMGroupManager : NSObject
/**
* 创建群组
*
* @param type 群类型,Private,Public,ChatRoom,AVChatRoom
* @param groupId 自定义群组 ID，为空时系统自动分配
* @param groupName 群组名称
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)createGroup:(NSString*)type groupId:(NSString*)groupId groupName:(NSString*)groupName succ:(TIMCreateGroupSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
type	群组类型
members	初始成员列表
groupName	群组名称
groupId	自定义群组 ID
succ	成功回调
fail	失败回调

邀请用户入群

`TIMGroupManager` 的接口 `inviteGroupMember` 可以邀请用户进入群组。

权限说明：

详情请参见 加群方式差异。

原型：

```
@interface TIMGroupManager : NSObject
/**
* 邀请好友入群
*
* @param group 群组 ID
* @param members 要加入的成员列表（NSString* 类型数组）
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)inviteGroupMember:(NSString*)group members:(NSArray*)members succ:(TIMGroupMemberSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	NSString 类型，群组 ID
members	NSString 列表，加入群组用户列表
succ	成功回调，TIMGroupMemberResult 数组，返回成功加入群组的用户列表以及成功状态
fail	失败回调

以下示例中邀请好友『iOS_002』加入群组 ID『TGID1JYSZEAEQ』，成功后返回操作列表以及成功状态，其中 `result.status` 表示当前用户操作是否成功。 **示例：**

```
NSMutableArray * members = [[NSMutableArray alloc] init];
// 添加一个用户 iOS_002
[members addObject:@"iOS_002"];
// @"TGID1JYSZEAEQ" 为群组 ID
[[TIMGroupManager sharedInstance] inviteGroupMember:@"TGID1JYSZEAEQ" 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);
}];
```

`result.status` 原型：

```
/**
* 群组操作结果
*/
typedef NS_ENUM(NSInteger, TIMGroupMemberStatus) {
/**
* 操作失败
*/
TIM_GROUP_MEMBER_STATUS_FAIL = 0,
/**
* 操作成功
*/
TIM_GROUP_MEMBER_STATUS_SUCC = 1,
/**
* 无效操作，加群时已经是群成员，移除群组时不在群内
*/
TIM_GROUP_MEMBER_STATUS_INVALID = 2,
/**
* 等待处理，邀请入群时等待对方处理
*/
TIM_GROUP_MEMBER_STATUS_PENDING = 3,
};
```

申请加入群组

`TIMGroupManager` 的接口 `joinGroup` 可以主动申请进入群组。此操作只对公开群、聊天室和音视频聊天室。

权限说明：

详情请参见 加群方式差异。

原型：

```
@interface TIMGroupManager : NSObject
/**
* 申请加群
*
* @param group 申请加入的群组 ID
* @param msg 申请消息
* @param succ 成功回调（申请成功等待审批）
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)joinGroup:(NSString*)group msg:(NSString*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	NSString 类型，群组 ID
msg	申请理由
succ	成功回调
fail	失败回调

以下示例中用户申请加入群组『TGID1JYSZEAEQ』，申请理由为『Apply Join Group』。**示例：**

```
[[TIMGroupManager sharedInstance] joinGroup:@"TGID1JYSZEAEQ" msg:@"Apply Join Group" succ:^(){
NSLog(@"Join Succ");
}fail:^(int code, NSString * err) {
NSLog(@"code=%d, err=%@", code, err);
}];
```

退出群组

群组成员可以主动退出群组。

权限说明：

- **私有群：**全员可退出群组。
- **公开群、聊天室、直播大群：**群主不能退出。

详情请参见 成员管理能力差异。

原型：

```
@interface TIMGroupManager : NSObject
/**
* 主动退出群组
*
* @param group 群组 ID
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)quitGroup:(NSString*)group succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	NSString 类型，群组 ID
succ	成功回调
fail	失败回调

以下示例中主动退出群组 『TGID1JYSZEAEQ』。 **示例：**

```
// @"TGID1JYSZEAEQ" 为群组 ID
[[TIMGroupManager sharedInstance] quitGroup:@"TGID1JYSZEAEQ" succ:^() {
NSLog(@"succ");
} fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

删除群组成员

群组成员也可以删除其他成员，函数参数信息与加入群组相同。

权限说明：
详情请参见 成员管理能力差异。

原型：

```
@interface TIMGroupManager : NSObject
/**
* 删除群成员
*
* @param group 群组 ID
* @param reason 删除原因
* @param members 要删除的成员列表
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)deleteGroupMemberWithReason:(NSString*)group reason:(NSString*)reason members:(NSArray*)members succ:(TIMGroupMemberSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	NSString 类型，群组 ID
reason	NSString 类型，原因
members	`NSString*` 数组，被操作的用户列表
succ	成功回调，`TIMGroupMemberResult` 数组，返回成功加入群组的用户列表已经成功状态
fail	失败回调

以下示例中把好友『iOS_002』从群组『TGID1JYSZEAEQ』中删除，执行成功后返回操作列表以及操作状态。 **示例：**

```
NSMutableArray * members = [[NSMutableArray alloc] init];
// 添加一个用户 iOS_002
[members addObject:@"iOS_002"];
// @"TGID1JYSZEAEQ" 为群组 ID
[[TIMGroupManager sharedInstance] deleteGroupMemberWithReason:@"TGID1JYSZEAEQ" reason:@"违反群规则" 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);
}];
```

获取群成员列表

`getGroupMembers` 方法可获取群内成员列表。

权限说明：

- **任何群组类型：**都可以获取成员列表。
- **直播大群：**只能拉取部分成员（包括群主、管理员和部分成员）。

详情请参见 群组基础能力操作差异。

原型：

```
/**
* 成员操作返回值
*/
@interface TIMGroupMemberInfo : TIMCodingModel
/**
* 被操作成员
*/
@property(nonatomic,retain) NSString* member;
/**
* 群名片
*/
@property(nonatomic,retain) NSString* nameCard;
/**
* 加入群组时间
*/
@property(nonatomic,assign) time_t joinTime;
/**
* 成员类型
*/
@property(nonatomic,assign) TIMGroupMemberRole role;
/**
* 禁言时间（剩余秒数）
*/
@property(nonatomic,assign) uint32_t silentUntil;
/**
* 自定义字段集合,key 是 NSString*类型,value 是 NSData*类型
*/
@property(nonatomic,retain) NSDictionary* customInfo;
@end

@interface TIMGroupManager : NSObject
/**
* 获取群成员列表
*
* @param group 群组 ID
* @param succ 成功回调(TIMGroupMemberInfo列表)
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)getGroupMembers:(NSString*)groupId succ:(TIMGroupMemberSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	NSString* 类型，群组 ID

参数	说明
succ	成功回调（返回 TIMGroupMemberInfo* 数组）
fail	失败回调

以下示例中获取群『TGID1JYSZEAEQ』的成员列表，`list` 为 `TIMGroupMemberInfo*` 数据，存储成员的相关信息。 **示例：**

```
// @"TGID1JYSZEAEQ" 为群组 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);
}];
```

如果群组人数过多，建议使用**分页接口。原型：**

```
@interface TIMGroupManager : NSObject
/**
* 获取指定类型的成员列表（支持按字段拉取，分页）
*
* @param group 群组 ID：（NSString*）列表
* @param filter 群成员角色过滤方式
* @param flags 拉取资料标志
* @param custom 要获取的自定义 key（NSString*）列表
* @param nextSeq 分页拉取标志，第一次拉取填0，回调成功如果不为零，需要分页，传入再次拉取，直至为0
* @param succ 成功回调
* @param fail 失败回调
*/
- (int)getGroupMembers:(NSString*)group ByFilter:(TIMGroupMemberFilter)filter flags:(TIMGetGroupMemInfoFlag)flags custom:(NSArray*)custom nextSe
q:(uint64_t)nextSeq succ:(TIMGroupMemberSuccV2)succ fail:(TIMFail)fail;
@end
```

获取加入的群组列表

通过 `getGroupList` 可以获取当前用户加入的所有群组。

权限说明：

- 可以获取自己所加入的群列表，返回的 `TIMGroupInfo` 只包含 `group`、`groupName`、`groupType` 信息。
- 只能获得加入的部分直播大群的列表。

原型：

```
@interface TIMGroupManager : NSObject
/**
* 获取群列表
*
* @param succ 成功回调，NSArray 列表为 TIMGroupInfo，结构体只包含 group\groupName\groupType\faceUrl\selfInfo 信息
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)getGroupList:(TIMGroupListSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
succ	成功回调，返回群组 ID 列表，TIMGroupInfo 数组
fail	失败回调

以下示例中获取群组列表，并打印群组 ID，群类型（Private、Public、ChatRoom）以及群名。**示例：**

```
[[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);
}];
```

解散群组

通过 `DeleteGroup` 可以解散群组。

权限说明：

详情请参见 群组基础能力操作差异。

原型：

```
@interface TIMGroupManager : NSObject

/**
* 解散群组
*
* @param group 群组 ID
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)deleteGroup:(NSString*)group succ:(TIMSucc)succ fail:(TIMFail)fail;

@end
```

参数说明：

参数	说明
group	群组 ID
succ	成功回调，返回群组 ID 列表，NSString 数组
fail	失败回调

以下示例中解散群组『TGID1JYSZEAEQ』。**示例：**

```
[[TIMGroupManager sharedInstance] deleteGroup:@"TGID1JYSZEAEQ" succ:^() {
NSLog(@"delete group succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

转让群组

通过 `modifyGroupOwner` 可以转让群组。

权限说明：

- 只有**群主**才有权限进行群转让操作。
- **直播大群**不能进行群转让操作。

原型：

```
@interface TIMGroupManager : NSObject
/**
* 转让群给新群主
```

```
 *
 * @param group 群组 ID
 * @param identifier 新的群主 ID
 * @param succ 成功回调
 * @param fail 失败回调
 *
 * @return 0 成功
 */
- (int)modifyGroupOwner:(NSString*)group user:(NSString*)identifier succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	群组 ID
user	用户 ID
succ	成功回调
fail	失败回调

以下示例中转让群组『TGID1JYSZEAEQ』给用户『iOS_001』。 **示例：**

```
[[TIMGroupManager sharedInstance] modifyGroupOwner:@"TGID1JYSZEAEQ" user:@"iOS_001" succ:^() {
NSLog(@"set new owner succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

全员禁言

通过 `modifyGroupAllShutup` 可以设置群组全员禁言。

权限说明：

- **群主、管理员：**有权限进行全员禁言的操作。
- **所有群组类型：**都支持全员禁言的操作。

原型：

```
@interface TIMGroupManager : NSObject
/**
 * 修改群组全员禁言属性
 *
 * @param group 群组 ID
 * @param shutup 是否禁言
 * @param succ 成功回调
 * @param fail 失败回调
 *
 * @return 0 成功
 */
- (int)modifyGroupAllShutup:(NSString*)group shutup:(BOOL)shutup succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	群组 ID
shutup	是否设置为禁言
succ	成功回调
fail	失败回调

示例中设置群组『TGID1JYSZEAEQ』为全员禁言的状态。客户端可以通过 getGroupList 和 getGroupInfo 接口获取当前群组全员禁言的属性。**示例：**

```
[[TIMGroupManager sharedInstance] modifyGroupAllShutup:@"TGID1JYSZEAEQ" shutup:YES succ:^() {
NSLog(@"set all shutup succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

获取群资料

获取群组资料

通过 TIMGroupManager 的方法 getGroupInfo 方法可以获取服务器存储的群组资料， queryGroupInfo 方法可以获取本地存储的群组资料，群资料信息由 TIMGroupInfo 定义。

权限说明：

- 无论是公开群还是私有群，群成员均可以拉到群组资料。
- 如果是公开群，非群组成员可以拉到 group、groupName、owner、groupType、createTime、maxMemberNum、memberNum、introduction、faceURL、addOpt、onlineMemberNum、customInfo 这些资料字段。如果是私有群，非群组成员拉取不到群组资料。

原型：

```
/**
* 群资料信息
*/
@interface TIMGroupInfo : TIMCodingModel
/**
* 群组 ID
*/
@property(nonatomic,retain) NSString* group;
/**
* 群名
*/
@property(nonatomic,retain) NSString* groupName;
/**
* 群创建人/管理员
*/
@property(nonatomic,retain) NSString * owner;
/**
* 群类型：Private、Public 和 ChatRoom
*/
@property(nonatomic,retain) NSString* groupType;
/**
* 群创建时间
*/
@property(nonatomic,assign) uint32_t createTime;
/**
* 最近一次群资料修改时间
*/
@property(nonatomic,assign) uint32_t lastInfoTime;
/**
* 最近一次发消息时间
*/
@property(nonatomic,assign) uint32_t lastMsgTime;
/**
* 最大成员数
*/
@property(nonatomic,assign) uint32_t maxMemberNum;
/**
* 群成员数量
*/
@property(nonatomic,assign) uint32_t memberNum;
/**
* 入群类型
```

```
 */
@property(nonatomic,assign) TIMGroupAddOpt addOpt;
/**
 * 群公告
 */
@property(nonatomic,retain) NSString* notification;
/**
 * 群简介
 */
@property(nonatomic,retain) NSString* introduction;
/**
 * 群头像
 */
@property(nonatomic,retain) NSString* faceURL;
/**
 * 最后一条消息
 */
@property(nonatomic,retain) TIMMessage* lastMsg;
/**
 * 在线成员数量
 */
@property(nonatomic,assign) uint32_t onlineMemberNum;
/**
 * 群组是否被搜索类型
 */
@property(nonatomic,assign) TIMGroupSearchableType isSearchable;
/**
 * 群组成员可见类型
 */
@property(nonatomic,assign) TIMGroupMemberVisibleType isMemberVisible;
/**
 * 是否全员禁言
 */
@property(nonatomic,assign) BOOL allShutup;
/**
 * 群组中的本人信息
 */
@property(nonatomic,retain) TIMGroupSelfInfo* selfInfo;
/**
 * 自定义字段集合，key 是 NSString* 类型，value 是 NSData* 类型
 */
@property(nonatomic,retain) NSDictionary* customInfo;
@end

@interface TIMGroupManager : NSObject
/**
 * 获取服务器存储的群资料
 *
 * @param groups 群组 ID 列表
 * @param succ 成功回调，不包含 selfInfo 信息
 * @param fail 失败回调
 *
 * @return 0 成功
 */
- (int)getGroupInfo:(NSArray*)groups succ:(TIMGroupListSucc)succ fail:(TIMFail)fail;
/**
 * 获取本地存储的群资料
 *
 * @param group 群组 ID
 *
 * @return 群组资料
 */
- (TIMGroupInfo *)queryGroupInfo:(NSString *)group;
@end
```

参数说明：

参数	说明

参数	说明
groups	NSString 数组，需要获取资料的群组列表
succ	成功回调，返回群组资料列表，TIMGroupInfo 数组
fail	失败回调

以下示例中获取群组『TGID1JYSZEAEQ』的详细信息。 **示例：**

```
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);
}];
```

获取本人在群里的资料

权限说明：

直播大群：不能拉取到本人资料。

原型：

```
@interface TIMGroupManager : NSObject
/**
* 获取本人在群组内的成员信息
*
* @param group 群组 ID
* @param succ 成功回调，返回信息
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)getGroupSelfInfo:(NSString*)groupId succ:(TIMGroupSelfSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
groupId	群组 ID
succ	成功回调，返回用户本人在群内的资料
fail	失败回调

获取指定成员在群里的资料

权限说明：

直播大群：只能获得部分成员的资料（包括群主、管理员和部分群成员）。

原型：

```
@interface TIMGroupManager : NSObject
/**
*
* 获取群组指定成员的信息，需要设置群成员 members，其他限制参考 getGroupMembers
*
* @param groupId 群组 ID
* @param members 成员 ID（NSString*）列表
* @param succ 成功回调 (TIMGroupMemberInfo 列表)
* @param fail 失败回调
```

```
 *
 * @return 0：成功；1：失败
 */
- (int)getGroupMembersInfo:(NSString*)groupId members:(NSArray<NSString *>*)members succ:(TIMGroupMemberSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
groupId	群组 ID
members	成员 ID 列表
succ	成功回调，返回群成员资料列表
fail	失败回调

修改群资料

修改群名

通过 `modifyGroupName` 可以修改群组名称。

权限说明：

- **公开群、聊天室和直播大群：**只有群主或者管理员可以修改群名。
- **私有群：**任何人可修改群名。

原型：

```
@interface TIMGroupManager : NSObject
/**
 * 修改群名
 *
 * @param group 群组 ID
 * @param groupName 新群名
 * @param succ 成功回调
 * @param fail 失败回调
 *
 * @return 0 成功
 */
- (int)modifyGroupName:(NSString*)group groupName:(NSString*)groupName succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	群组 ID
groupName	修改后的群名
succ	成功回调
fail	失败回调

以下示例修改群『TGID1JYSZEAEQ』的名字为『ModifyGroupName』。**示例：**

```
[[TIMGroupManager sharedInstance] modifyGroupName:@"TGID1JYSZEAEQ" groupName:@"ModifyGroupName" succ:^() {
NSLog(@"modify group name succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

修改群简介

通过 `modifyGroupIntroduction` 可以修改群组简介。

权限说明：

- **公开群、聊天室、直播大群：**只有群主或者管理员可以修改群简介。
- **私有群：**任何人可修改群简介。

原型：

```
@interface TIMGroupManager : NSObject

/**
* 修改群简介
*
* @param group 群组 ID
* @param introduction 群简介
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)modifyGroupIntroduction:(NSString*)group introduction:(NSString*)introduction succ:(TIMSucc)succ fail:(TIMFail)fail;

@end
```

参数说明：

参数	说明
group	群组 ID
introduction	群简介，简介最长120字节
succ	成功回调
fail	失败回调

示例：

```
[[TIMGroupManager sharedInstance] modifyGroupIntroduction:@"TGID1JYSZEAEQ" introduction :@"this is one group" succ:^() {
NSLog(@"modify group introduction succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

修改群公告

通过 `modifyGroupNotification` 可以修改群组公告。

权限说明：

- **公开群、聊天室、直播大群：**只有群主或者管理员可以修改群公告。
- **私有群：**任何人可修改群公告。

原型：

```
@interface TIMGroupManager : NSObject
/**
* 修改群公告
*
* @param group 群组 ID
* @param notification 群公告（最长150字节）
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
*/
```

```
- (int)modifyGroupNotification:(NSString*)group notification:(NSString*)notification succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	群组 ID
notification	群公告，群公告最长150字节
succ	成功回调
fail	失败回调

以下示例修改群『TGID1JYSZEAEQ』的公告为『test notification』。 **示例：**

```
[[TIMGroupManager sharedInstance] modifyGroupNotification:@"TGID1JYSZEAEQ" notification:@"test notification" succ:^() {
NSLog(@"modify group notification succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

修改群头像

通过 `modifyGroupFaceUrl` 可以修改群头像。

权限说明：

- **公开群、聊天室、直播大群：**只有群主或者管理员可以修改群头像。
- **私有群：**任何人可修改群头像。

原型：

```
@interface TIMGroupManager : NSObject
/**
* 修改群头像
*
* @param group 群组 ID
* @param url 群头像地址（最长100字节）
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)modifyGroupFaceUrl:(NSString*)group url:(NSString*)url succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	群组 ID
url	群头像地址（最长100字节）
succ	成功回调
fail	失败回调

示例：

```
[[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);
}];
```

修改加群选项

通过 `modifyGroupAddOpt` 可以修改加群选项。

权限说明：

- **公开群、聊天室、直播大群：**只有群主或者管理员可以修改加群选项。
- **私有群：**只能通过邀请加入群组，不能主动申请加入某个群组。

原型：

```
@interface TIMGroupManager : NSObject

/**
 * 修改加群选项
 *
 * @param group 群组 ID
 * @param opt 加群选项，详见 TIMGroupAddOpt
 * @param succ 成功回调
 * @param fail 失败回调
 *
 * @return 0 成功
 */
- (int)modifyGroupAddOpt:(NSString*)group opt:(TIMGroupAddOpt)opt succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	群组 ID
opt	加群选项，可设置为允许任何人加入、需要审核、禁止任何人加入
succ	成功回调
fail	失败回调

以下示例修改群『TGID1JYSZEAEQ』为禁止任何人加入。 **示例：**

```
[[TIMGroupManager sharedInstance] modifyGroupAddOpt:@"TGID1JYSZEAEQ" opt:TIM_GROUP_ADD_FORBID succ:^() {
NSLog(@"modify group opt succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

修改群维度自定义字段

通过 `modifyGroupCustomInfo` 可以对群维度自定义字段进行修改。

权限说明：

- 通过后台配置相关的 key 和权限。

原型：

```
@interface TIMGroupManager : NSObject
/**
 * 修改群自定义字段集合
 *
 * @param group 群组 ID
 * @param customInfo 自定义字段集合，key 是 NSString* 类型，value 是 NSData* 类型
 * @param succ 成功回调
 * @param fail 失败回调
 *
 * @return 0 成功
 */
```

```
- (int)modifyGroupCustomInfo:(NSString*)group customInfo:(NSDictionary*)customInfo succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	群组 ID
customInfo	自定义字段集合，key 是 NSString* 类型，value 是 NSData* 类型
succ	成功回调
fail	失败回调

以下示例修改群『TGID1JYSZEAEQ』为禁止任何人加入。**示例：**

```
// 设置自定义数据
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 succ:^() {
NSLog(@"modify group customInfo succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

修改用户群内身份

通过 `modifyGroupMemberInfoSetRole` 可以对群成员的身份进行修改。

权限说明：

- **群主、管理员：**可以进行对群成员的身份进行修改。
- **直播大群：**不支持修改用户群内身份。

原型：

```
@interface TIMGroupManager : NSObject
- (int)modifyGroupMemberInfoSetRole:(NSString*)group user:(NSString*)identifier role:(TIMGroupMemberRole)role succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	群组 ID
identifier	要修改的群成员的 ID
role	修改后的身份类型，不能修改为群主类型
succ	成功回调
fail	失败回调

以下示例设置群『TGID1JYSZEAEQ』的成员『iOS_001』为管理员。 **示例：**

```
[[TIMGroupManager sharedInstance] modifyGroupMemberInfoSetRole:@"TGID1JYSZEAEQ" user:@"iOS_001" role:TIM_GROUP_MEMBER_ADMIN succ:^() {
NSLog(@"modify group member role succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

对群成员进行禁言

通过 modifyGroupMemberInfoSetSilence 可以对群成员进行禁言并设置禁言时长。

权限说明：

群主、管理员：可以进行对群成员进行禁言。

原型：

```
@interface TIMGroupManager : NSObject
- (int)modifyGroupMemberInfoSetSilence:(NSString*)group user:(NSString*)identifier stime:(uint32_t)stime succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	群组 ID
identifier	要禁言的群成员的 ID
stime	禁言时间，单位秒
succ	成功回调
fail	失败回调

以下示例设置群『TGID1JYSZEAEQ』的成员『iOS_001』禁言120秒。 **示例：**

```
[[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);
}];
```

修改群名片

通过 modifyGroupMemberInfoSetNameCard 可以对群成员资料的群名片进行修改。

*原型： *

```
@interface TIMGroupManager : NSObject
- (int)modifyGroupMemberInfoSetNameCard:(NSString*)group user:(NSString*)identifier nameCard:(NSString*)nameCard succ:(TIMSucc)succ fail:(TIMFail)
fail;
@end
```

参数说明：

参数	说明
group	群组 ID
identifier	要修改的群成员的 ID
nameCard	要设置的群名片
succ	成功回调
fail	失败回调

以下示例设置群『TGID1JYSZEAEQ』的成员『iOS_001』群名片为『iOS_001_namecard』。 **示例：**

```
[[TIMGroupManager sharedInstance] modifyGroupMemberInfoSetNameCard:@"TGID1JYSZEAEQ" user:@"iOS_001" nameCard:@"iOS_001_namecard" suc
c:^() {
NSLog(@"modify group member namecard succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

修改群成员维度自定义字段

通过 `modifyGroupMemberInfoSetCustomInfo` 可以对群成员维度自定义字段进行修改。

原型：

```
@interface TIMGroupManager : NSObject
- (int)modifyGroupMemberInfoSetCustomInfo:(NSString*)group user:(NSString*)identifier customInfo:(NSDictionary<NSString*,NSData*> *)customInfo succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	群组 ID
identifier	要设置自定义属性的群成员的 ID
customInfo	自定义字段集合，key 是 NSString* 类型，value 是 NSData* 类型
succ	成功回调
fail	失败回调

以下示例设置群『TGID1JYSZEAEQ』的成员『iOS_001』的自定义属性。**示例：**

```
[[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);
}];
```

修改接收群消息选项

通过 `modifyReceiveMessageOpt` 可以设置群消息的接收选项。默认情况下，公开群和私有群是接收并离线推送群消息，聊天室和直播大群是接收但不离线推送群消息。

原型：

```
@interface TIMGroupManager : NSObject
- (int)modifyReceiveMessageOpt:(NSString*)group opt:(TIMGroupReceiveMessageOpt)opt succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
group	群组 ID
opt	接收消息选项
succ	成功回调
fail	失败回调

以下示例设置群『TGID1JYSZEAEQ』的接收消息选项为接收在线消息，不接收离线推送。**示例：**

```
[[TIMGroupManager sharedInstance] modifyReciveMessageOpt:@"TGID1JYSZEAEQ" opt:TIM_GROUP_RECEIVE_NOT_NOTIFY_MESSAGE succ:^() {
NSLog(@"modify receive group message option succ");
}fail:^(int code, NSString* err) {
NSLog(@"failed code: %d %@", code, err);
}];
```

群组未决信息

拉取群未决相关信息

通过 getPendencyFromServer 接口可拉取群未决相关信息。此处的群未决消息泛指所有需要审批的群相关的操作（例如：加群待审批，拉人入群待审批等等）。即便审核通过或者拒绝后，该条信息也可通过此接口拉回，拉回的信息中有已决标志。

权限说明：

审批人：有权限拉取相关信息。

说明：

- 如果 UserA 申请加入群 GroupA，则群管理员可获取此未决相关信息，UserA 因为没有审批权限，不需要拉取未决信息。
- 如果 AdminA 拉 UserA 进去 GroupA，则 UserA 可以拉取此未决相关信息，因为该未决信息待 UserA 审批。

原型：

```
@interface TIMGroupManager : NSObject
/**
* 获取群组未决列表
*
* @param option 未决参数配置
* @param succ 成功回调，返回未决列表
* @param fail 失败回调
*
* @return 0 成功
*/
- (int)getPendencyFromServer:(TIMGroupPendencyOption*)option succ:(TIMGetGroupPendencyListSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
option	未决参数配置
succ	成功回调，返回未决列表
fail	失败回调

option参数说明：

参数	说明
timestamp	拉取的开始时戳。若从最新的未决条目开始拉取，则填0或不填。若分页，则回调中返回下一个分页的拉取起始时戳
numPerPage	一次拉取的最多条目数，用于分页

回调原型：

```
/**
* 获取群组未决请求列表成功
*
* @param meta 未决请求元信息
* @param pendencies 未决请求列表（TIMGroupPendencyItem）数组
*/
typedef void (^TIMGetGroupPendencyListSucc)(TIMGroupPendencyMeta *meta, NSArray *pendencies);
```

回调参数说明：

参数	说明
meta	拉取操作返回的相关信息，包含分页信息和拉取状态等
pendencies	拉取的未决条目数组

属性说明：

属性	说明
nextStartTime	拉取下一个分页的起始时戳，为0时表示没有后面的分页了
readTimeSeq	已读时戳，用来判定未决条目是否已读
unReadCnt	所有未读条目个数，不限制于本次分页中

未决条目相关属性：

```
/**
* 未决申请
*/
@interface TIMGroupPendencyItem : TIMCodingModel
/**
* 相关群组 ID
*/
@property(nonatomic,retain) NSString* groupId;
/**
* 请求者 ID，请求加群:请求者，邀请加群:邀请人
*/
@property(nonatomic,retain) NSString* fromUser;
/**
* 判决者 ID，请求加群：0，邀请加群：被邀请人
*/
@property(nonatomic,retain) NSString* toUser;
/**
* 未决添加时间
*/
@property(nonatomic,assign) uint64_t addTime;
/**
* 未决请求类型
*/
@property(nonatomic,assign) TIMGroupPendencyGetType getType;
/**
* 已决标志
*/
@property(nonatomic,assign) TIMGroupPendencyHandleStatus handleStatus;
/**
* 已决结果
*/
@property(nonatomic,assign) TIMGroupPendencyHandleResult handleResult;
/**
* 申请或邀请附加信息
*/
@property(nonatomic,retain) NSString* requestMsg;
/**
* 审批信息：同意或拒绝信息
*/
@property(nonatomic,retain) NSString* handledMsg;
/**
* 同意申请
*
* @param msg 同意理由，选填
* @param succ 成功回调
* @param fail 失败回调，返回错误码和错误描述
*/
-(void) accept:(NSString*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
/**
* 拒绝申请
*
* @param msg 拒绝理由，选填
* @param succ 成功回调
* @param fail 失败回调，返回错误码和错误描述
*/
-(void) refuse:(NSString*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
/**
```

```
 * 用户自己的id
 */
@property(nonatomic,strong) NSString* selfIdentifier;
@end
```

*属性说明： *

属性	说明
groupId	群 ID
fromUser	未决发起者 ID
toUser	未决审批者 ID
addTime	添加未决时间
getType	枚举未决条目类型： 请求加群、邀请加群
handleStatus	枚举未决条目状态：未决、他人已决、操作者已决（例如：UserA 申请加入 Group，AdminA 审批通过。则 AdminB 拉取的此未决条目的类型为，他人已决。）
handleResult	枚举审批结果：同意、拒绝
requestMsg/handleMsg	申请、审批时的留言信息

示例：

```
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);
}];
```

上报群未决已读

对于未决信息，IM SDK 可对其和之前的所有未决信息上报已读。上报已读后，仍然可以拉取到这些未决信息，但可通过对已读时戳的判断判定未决信息是否已读。

原型：

```
@interface TIMGroupManager : NSObject
/**
 * 群未决已读上报
 *
 * @param timestamp 上报已读时间戳
 * @param succ 成功回调
 * @param fail 失败回调
 *
 * @return 0 成功
 */
- (int)pendencyReport:(uint64_t)timestamp succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
timestamp	上报已读时戳。对于单条未决信息，时戳包含在其属性里。
succ	成功回调
fail	失败回调

示例：

```
[[TIMGroupManager sharedInstance] pendencyReport:timestamp succ:^{
NSLog(@"pendency report succ");
} fail:^(int code, NSString *msg) {
NSLog(@"pendency report failed: %d->%@", code, msg);
}];
```

处理群未决信息

对于群的未决信息，IM SDK 增加了处理接口。审批人可以选择对单条信息进行同意或者拒绝。已处理成功过的未决信息不能再次处理。

原型：

```
@interface TIMGroupPendencyItem: NSObject
/**
* 同意申请
*
* @param msg 同意理由，选填
* @param succ 成功回调
* @param fail 失败回调，返回错误码和错误描述
*/
- (void)accept:(NSString*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
/**
* 拒绝申请
*
* @param msg 拒绝理由，选填
* @param succ 成功回调
* @param fail 失败回调，返回错误码和错误描述
*/
- (void)refuse:(NSString*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

示例：

```
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);
}];
```

群事件消息

当有用户被邀请加入群组，或者有用户被移出群组时，群内会产生有提示消息，调用方可选择是否予以展示，以及如何展示（例如：忽略或者根据需要展示给用户）。 提示消息使用一个特殊的 `Elem` 标识，通过新消息回调返回消息，参见 新消息通知。如下图中，展示一条修改群名的事件消息。

消息原型：

```
/**
* 群 Tips 类型
*/
typedef NS_ENUM(NSInteger, TIM_GROUP_TIPS_TYPE){
/**
* 邀请加入群 (opUser & groupName & userList)
*/
TIM_GROUP_TIPS_TYPE_INVITE = 0x01,
/**
* 退出群 (opUser & groupName & userList)
*/
TIM_GROUP_TIPS_TYPE_QUIT_GRP = 0x02,
/**
* 踢出群 (opUser & groupName & userList)
*/
TIM_GROUP_TIPS_TYPE_KICKED = 0x03,
/**
* 设置管理员 (opUser & groupName & userList)
*/
TIM_GROUP_TIPS_TYPE_SET_ADMIN = 0x04,
/**
* 取消管理员 (opUser & groupName & userList)
*/
TIM_GROUP_TIPS_TYPE_CANCEL_ADMIN = 0x05,
/**
* 群资料变更 (opUser & groupName & introduction & notification & faceUrl & owner)
*/
TIM_GROUP_TIPS_TYPE_INFO_CHANGE = 0x06,
/**
* 群成员资料变更 (opUser & groupName & memberInfoList)
*/
TIM_GROUP_TIPS_TYPE_MEMBER_INFO_CHANGE = 0x07,
};
/**
* 群 Tips
*/
@interface TIMGroupTipsElem : TIMElem
/**
* 群组 ID
*/
@property(nonatomic,strong) NSString * group;
/**
* 群 Tips 类型
*/
@property(nonatomic,assign) TIM_GROUP_TIPS_TYPE type;
```

```
/**
 * 操作人用户名
 */
@property(nonatomic,strong) NSString * opUser;
/**
 * 被操作人列表 NSString* 数组
 */
@property(nonatomic,strong) NSArray * userList;
/**
 * 在群名变更时表示变更后的群名，否则为 nil
 */
@property(nonatomic,strong) NSString * groupName;
/**
 * 群信息变更： TIM_GROUP_TIPS_TYPE_INFO_CHANGE 时有效，为 TIMGroupTipsElemGroupInfo 结构体列表
 */
@property(nonatomic,strong) NSArray * groupChangeList;
/**
 * 成员变更： TIM_GROUP_TIPS_TYPE_MEMBER_INFO_CHANGE 时有效，为 TIMGroupTipsElemMemberInfo 结构体列表
 */
@property(nonatomic,strong) NSArray * memberChangeList;
/**
 * 操作者用户资料
 */
@property(nonatomic,strong) TIMUserProfile * opUserInfo;
/**
 * 操作者群成员资料
 */
@property(nonatomic,strong) TIMGroupMemberInfo * opGroupMemberInfo;
/**
 * 变更成员资料
 */
@property(nonatomic,strong) NSDictionary * changedUserInfo;
/**
 * 变更成员群内资料
 */
@property(nonatomic,strong) NSDictionary * changedGroupMemberInfo;
/**
 * 当前群人数： TIM_GROUP_TIPS_TYPE_INVITE、TIM_GROUP_TIPS_TYPE_QUIT_GRP、
 * TIM_GROUP_TIPS_TYPE_KICKED时有效
 */
@property(nonatomic,assign) uint32_t memberNum;
/**
 * 操作方平台信息
 * 取值： iOS Android Windows Mac Web RESTAPI Unknown
 */
@property(nonatomic,strong) NSString * platform;
@end
```

以下示例中注册新消息回调，打印用户进入群组和离开群组的事件通知，其他事件通知用法相同。 **示例：**

```
@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
```

用户加入群组

触发时机：当有用户加入群组时（包括申请入群和被邀请入群），群组内会由系统发出通知，开发者可选择展示样式。可以更新群成员列表。收到的消息 type 为 `TIM_GROUP_TIPS_TYPE_INVITE`。

`TIMGroupTipsElem` 参数说明：

参数	说明
type	TIM_GROUP_TIPS_TYPE_INVITE
opUser	申请入群：申请人 / 邀请入群：邀请人
groupName	群名
userList	入群的用户列表

用户退出群组

触发时机：当有用户主动退群时，群组内会由系统发出通知。可以选择更新群成员列表。 收到的消息 type 为 `TIM_GROUP_TIPS_TYPE_QUIT_GRP`。

`TIMGroupTipsElem` 参数说明：

参数	说明
type	TIM_GROUP_TIPS_TYPE_QUIT_GRP
opUser	退出用户 identifier
groupName	群名

用户被踢出群组

触发时机：当有用户被踢时，群组内会由系统发出通知。可以选择更新群成员列表。 收到的消息 type 为 `TIM_GROUP_TIPS_TYPE_KICKED`。

`TIMGroupTipsElem` 参数说明：

参数	说明
type	TIM_GROUP_TIPS_TYPE_KICKED
opUser	被踢用户 identifier
groupName	群名

被设置/取消管理员

触发时机：当有用户被设置为管理员或者被取消管理员身份时，群组内会由系统发出通知。如果界面有显示是否管理员，此时可更新管理员标识。 收到的消息 type 为 `TIM_GROUP_TIPS_TYPE_SET_ADMIN` 和 `TIM_GROUP_TIPS_TYPE_CANCEL_ADMIN`。

* *`TIMGroupTipsElem` 参数说明：* *

参数	说明
type	设置：TIM_GROUP_TIPS_TYPE_SET_ADMIN
取消	TIM_GROUP_TIPS_TYPE_CANCEL_ADMIN
opUser	操作用户 identifier

参数	说明
groupName	群名
userList	被设置/取消管理员身份的用户列表

群资料变更

触发时机：当群资料变更（如群名、群简介等），会有系统消息发出，此时可以更新相关展示字段，选择性把消息展示给用户。

TIMGroupTipsElem 参数说明：

参数	说明
type	TIM_GROUP_TIPS_TYPE_INFO_CHANGE
opUser	操作用户 identifier
groupName	群名
groupChangeInfo	群变更的具体资料信息，为 TIMGroupTipsElemGroupInfo 结构体列表

TIMGroupTipsElemGroupInfo 原型：

```
/**
* 群 tips，群变更信息
*/
@interface TIMGroupTipsElemGroupInfo : NSObject
/**
* 变更类型
*/
@property(nonatomic, assign) TIM_GROUP_INFO_CHANGE_TYPE type;

/**
* 根据变更类型表示不同含义
*/
@property(nonatomic,strong) NSString * value;
@end
```

参数说明：

参数	说明
type	变更类型
value	变更后的值，根据变更类型表示不同含义

群成员资料变更

触发时机：当群成员的群相关资料变更时，包括群内用户被禁言、群内成员角色变更，会有系统消息发出，可更新相关字段展示，或者选择性把消息展示给用户。

注意：

- 这里的资料仅跟群相关资料，例如禁言时间、成员角色变更等，不包括用户昵称等本身资料，对于群内人数可能过多，不建议实时更新，建议的做法是直接显示消息体内的资料，参考：消息发送者以及相关资料
- 如果本地有保存用户资料，可根据消息体内资料判断是否有变更，在收到此用户一条消息后更新资料。

TIMGroupTipsElem 参数说明：

参数	说明
type	TIM_GROUP_TIPS_TYPE_MEMBER_INFO_CHANGE
opUser	操作用户 identifier
groupName	群名

参数	说明
memberInfoList	变更的群成员的具体资料信息，为 TIMGroupTipsElemMemberInfo 结构体列表

TIMGroupTipsElemMemberInfo 原型：

```
/**
* 群 tips，成员变更信息
*/
@interface TIMGroupTipsElemMemberInfo : NSObject
/**
* 变更用户
*/
@property(nonatomic,retain) NSString * identifier;
/**
* 禁言时间（秒，还剩多少秒可以发言）
*/
@property(nonatomic,assign) uint32_t shutupTime;
@end
```

参数说明：

参数	说明
identifier	变更的用户 identifier
shutupTime	被禁言的时间

群事件消息监听器

聊天室和直播大群的群事件消息需要通过注册监听器获得（设置监听位置在 TIMManager > setUserConfig > TIMUserConfig > groupEventListener），消息 Elem 中包含群的成员数。

```
/**
* 群事件通知回调
*/
@protocol TIMGroupEventListener <NSObject>
@optional
/**
* 群 tips 回调
*
* @param elem 群 tips 消息
*/
- (void)onGroupTipsEvent:(TIMGroupTipsElem*)elem {
// 群组 ID
NSString *groupID = elem.group;
// 操作者
NSString *opUser = elem.opUser;
// 被操作者
NSArray *userList = elem.userList;
switch (elem.type) {
case TIM_GROUP_TIPS_TYPE_INVITE:
// userList 加入群组，如果是私有群（Private），可以展示为 "opUser 邀请 userList 入群"。
// 如果是其他群组类型，可以展示为 "userList 加入群组"
break;
case TIM_GROUP_TIPS_TYPE_QUIT_GRP:
// opUser 退出群组
break;
case TIM_GROUP_TIPS_TYPE_KICKED:
// opUser 把 userList 踢出群组
break;
case TIM_GROUP_TIPS_TYPE_SET_ADMIN:
// opUser 设置 userList 为管理员
break;
case TIM_GROUP_TIPS_TYPE_CANCEL_ADMIN:
// opUser 取消 userList 管理员身份
```

```
break;
case TIM_GROUP_TIPS_TYPE_INFO_CHANGE:
// groupID 群信息发生了变化
break;
case TIM_GROUP_TIPS_TYPE_MEMBER_INFO_CHANGE:
// groupID 群成员群信息发生了变化
break;
default:
break;
}
}
@end
```

群系统消息

当有用户申请加群等事件发生时，管理员会收到邀请加群系统消息，用户可根据情况接受请求或者拒绝，相应的消息通过群系统消息展示给用户。

群系统消息类型定义：

```
/**
* 群系统消息类型
*/
typedef NS_ENUM(NSInteger, TIM_GROUP_SYSTEM_TYPE){
/**
* 申请加群请求（只有管理员会收到）
*/
TIM_GROUP_SYSTEM_ADD_GROUP_REQUEST_TYPE = 0x01,
/**
* 申请加群被同意（只有申请人能够收到）
*/
TIM_GROUP_SYSTEM_ADD_GROUP_ACCEPT_TYPE = 0x02,
/**
* 申请加群被拒绝（只有申请人能够收到）
*/
TIM_GROUP_SYSTEM_ADD_GROUP_REFUSE_TYPE = 0x03,
/**
* 被管理员踢出群（只有被踢的人能够收到）
*/
TIM_GROUP_SYSTEM_KICK_OFF_FROM_GROUP_TYPE = 0x04,
/**
* 群被解散（全员能够收到）
*/
TIM_GROUP_SYSTEM_DELETE_GROUP_TYPE = 0x05,
/**
* 创建群消息（创建者能够收到）
*/
TIM_GROUP_SYSTEM_CREATE_GROUP_TYPE = 0x06,
/**
* 邀请加群（被邀请者能够收到）
*/
TIM_GROUP_SYSTEM_INVITED_TO_GROUP_TYPE = 0x07,
/**
* 主动退群（主动退群者能够收到）
*/
TIM_GROUP_SYSTEM_QUIT_GROUP_TYPE = 0x08,
/**
* 设置管理员(被设置者接收)
*/
TIM_GROUP_SYSTEM_GRANT_ADMIN_TYPE = 0x09,
/**
* 取消管理员(被取消者接收)
*/
TIM_GROUP_SYSTEM_CANCEL_ADMIN_TYPE = 0x0a,
/**
* 群已被回收(全员接收)
*/
```

```
TIM_GROUP_SYSTEM_REVOKE_GROUP_TYPE = 0x0b,
/**
* 邀请入群请求(被邀请者接收)
*/
TIM_GROUP_SYSTEM_INVITE_TO_GROUP_REQUEST_TYPE = 0x0c,
/**
* 邀请加群被同意(只有发出邀请者会接收到)
*/
TIM_GROUP_SYSTEM_INVITE_TO_GROUP_ACCEPT_TYPE = 0x0d,
/**
* 邀请加群被拒绝(只有发出邀请者会接收到)
*/
TIM_GROUP_SYSTEM_INVITE_TO_GROUP_REFUSE_TYPE = 0x0e,
/**
* 用户自定义通知(默认全员接收)
*/
TIM_GROUP_SYSTEM_CUSTOM_INFO = 0xff,
};
/**
* 群系统消息
*/
@interface TIMGroupSystemElem : TIMElem
/**
* 操作类型
*/
@property(nonatomic,assign) TIM_GROUP_SYSTEM_TYPE type;
/**
* 群组 ID
*/
@property(nonatomic,strong) NSString * group;
/**
* 操作人
*/
@property(nonatomic,strong) NSString * user;
/**
* 操作理由
*/
@property(nonatomic,strong) NSString * msg;
/**
* 消息标识，客户端无需关心
*/
@property(nonatomic,assign) uint64_t msgKey;
/**
* 消息标识，客户端无需关心
*/
@property(nonatomic,strong) NSData * authKey;
/**
* 用户自定义透传消息体（type = TIM_GROUP_SYSTEM_CUSTOM_INFO 时有效）
*/
@property(nonatomic,strong) NSData * userData;
/**
* 操作人资料
*/
@property(nonatomic,strong) TIMUserProfile * opUserInfo;
/**
* 操作人群成员资料
*/
@property(nonatomic,strong) TIMGroupMemberInfo * opGroupMemberInfo;
/**
* 操作方平台信息
* 取值： iOS、Android、Windows、Mac、Web、RESTAPI、Unknown
*/
@property(nonatomic,strong) NSString * platform;
@end
```

参数说明：

参数	说明

参数	说明
type	消息类型
group	群组 ID
user	操作人
msg	操作理由
msgKey & authKey	消息的标识，客户端无需关心，调用 accept 和 refuse 时由 IM SDK 读取

以下示例中处理收到群系统消息，如果是入群申请则默认同意，如果是群解散通知则打印信息。其他类型消息解析方式相同。 *示例：*

```
@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
```

申请加群消息

触发时机：当有用户申请加群时，群管理员会收到申请加群消息，可展示给用户，由用户决定是否同意对方加群。 消息类型为 `TIM_GROUP_SYSTEM_ADD_GROUP_REQUEST_TYPE` 。

参数说明：

参数	说明
type	TIM_GROUP_SYSTEM_ADD_GROUP_REQUEST_TYPE
group	群组 ID，表示是哪个群的申请
user	申请人
msg	申请理由（可选）

方法说明：

- 同意申请人入群，可调用 accept 方法
- 不同意申请人入群，可调用 refuse 方法。

申请加群同意/拒绝消息

触发时机：当管理员同意加群请求时，申请人会收到同意入群的消息，当管理员拒绝时，收到拒绝入群的消息。

参数说明：

参数	说明
type	同意：TIM_GROUP_SYSTEM_ADD_GROUP_ACCEPT_TYPE 拒绝：TIM_GROUP_SYSTEM_ADD_GROUP_REFUSE_TYPE
group	群组 ID，表示是哪个群通过/拒绝了
user	处理请求的管理员 identifier
msg	同意或者拒绝理由（可选）

邀请入群消息

触发时机：当有用户被邀请群时，该用户会收到邀请入群消息，可展示给用户，由用户决定是否同意入群，如果同意，调用 accept 方法，拒绝调用 refuse 方法。 消息类型为 `TIM_GROUP_SYSTEM_INVITE_TO_GROUP_REQUEST_TYPE`

参数说明：

参数	说明
type	TIM_GROUP_SYSTEM_INVITE_TO_GROUP_REQUEST_TYPE
group	群组 ID，表示是哪个群的邀请
user	邀请人

方法说明：

- 同意申请人入群，可调用 accept 方法。
- 不同意申请人入群，可调用 refuse 方法。

邀请入群同意/拒绝消息

触发时机：当被邀请者同意入群请求时，邀请者会收到同意入群的消息，当被邀请者拒绝时，邀请者会收到拒绝入群的消息。

参数说明：

参数	说明
type	同意：TIM_GROUP_SYSTEM_INVITE_TO_GROUP_ACCEPT_TYPE 拒绝：TIM_GROUP_SYSTEM_INVITE_TO_GROUP_REFUSE_TYPE
group	群组 ID，表示是对哪个群通过/拒绝了
user	处理请求的用户 identifier
msg	同意或者拒绝理由（可选）

被管理员踢出群组

触发时机：当用户被管理员踢出群组时，申请人会收到被踢出群的消息。

参数说明：

参数	说明
type	TIM_GROUP_SYSTEM_KICK_OFF_FROM_GROUP_TYPE
group	群组 ID，表示在哪个群里被踢了
user	操作管理员 identifier

群被解散

触发时机：当群被解散时，全员会收到解散群消息。

参数说明：

参数	说明

参数	说明
type	TIM_GROUP_SYSTEM_DELETE_GROUP_TYPE
group	群组 ID，表示哪个群被解散了
user	操作管理员 identifier

创建群消息

触发时机：当群创建时，创建者会收到创建群消息。当调用创建群方法成功回调后，即表示创建成功，此消息主要为多终端同步，如果有在其他终端登录，作为更新群列表的时机，本终端可以选择忽略。

参数说明：

参数	说明
type	TIM_GROUP_SYSTEM_CREATE_GROUP_TYPE
group	群组 ID，表示创建的群 ID
user	创建者，这里也就是用户自己

邀请加群

触发时机：当用户被邀请加入群组时，该用户会收到邀请消息，**创建群组时初始成员无需邀请即可入群。 **

参数说明：

参数	说明
type	TIM_GROUP_SYSTEM_INVITED_TO_GROUP_TYPE
group	群组 ID，邀请进入哪个群
user	操作人，表示哪个用户的邀请

方法说明：

- 同意申请人入群，可调用 accept 方法。
- 不同意申请人入群，可调用 refuse 方法。

主动退群

触发时机：当用户主动退出群组时，该用户会收到退群消息，只有退群的用户自己可以收到。当用户调用 `QuitGroup` 时成功回调返回，表示已退出成功，此消息主要为了多终端同步，其他终端可以作为更新群列表的时机，本终端可以选择忽略。

参数说明：

参数	说明
type	TIM_GROUP_SYSTEM_QUIT_GROUP_TYPE
group	群组 ID，表示退出的哪个群
user	操作人，这里即为用户自己

设置/取消管理员

触发时机：当用户被设置为管理员时，可收到被设置管理员的消息通知，当用户被取消管理员时，可收到取消通知，可提示用户。

参数说明：

参数	说明
type	取消管理员身份：TIM_GROUP_SYSTEM_GRANT_ADMIN_TYPE 授予管理员身份：TIM_GROUP_SYSTEM_CANCEL_ADMIN_TYPE
group	群组 ID，表示哪个群的事件

参数	说明
user	操作人

群被回收

触发时机：当群组被系统回收时，全员可收到群组被回收消息。

参数说明：

参数	说明
type	TIM_GROUP_SYSTEM_REVOKE_GROUP_TYPE
group	群组 ID，表示哪个群被回收了

消息收发
消息收发（Android）

最近更新时间：2020-09-29 14:25:04

以下视频将帮助您快速了解 Android SDK 的消息收发相关功能：

点击查看视频

消息发送

通用消息发送

会话获取：会话是指面向一个人或者一个群组的对话，通过与单个人或群组之间会话收发消息，发消息时首先需要先获取会话，获取会话需要指定会话类型（群组 & 单聊），以及会话对方标志（对方帐号或者群号）。获取会话由 `TIMManager` 中的 `getConversation` 实现。

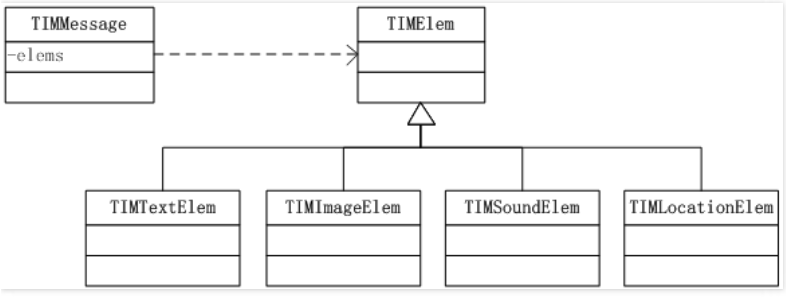
原型：

```
/**
 * 获取会话
 * @param type 会话类型
 * @param peer 参与会话的对方, C2C 会话为对方帐号 identifier, 群组会话为群组 ID
 * @return 会话实例
 */
public TIMConversation getConversation(TIMConversationType type, String peer)
```

示例：

```
//获取单聊会话
String peer = "sample_user_1"; //获取与用户 "sample_user_1" 的会话
conversation = TIMManager.getInstance().getConversation(
TIMConversationType.C2C, //会话类型：单聊
peer); //会话对方用户帐号//对方ID

//获取群聊会话
String groupId = "TGID1EDABEAEO"; //获取与群组 "TGID1LTTZEAEO" 的会话
conversation = TIMManager.getInstance().getConversation(
TIMConversationType.Group, //会话类型：群组
groupId); //群组 ID
```

消息发送：通过 `TIMManager` 获取会话 `TIMConversation` 后，可发送消息和获取会话缓存消息。IM SDK 中消息的解释可参阅 IM SDK 对象简介。IM SDK 中的消息由 `TIMMessage` 表达，一个 `TIMMessage` 由多个 `TIMElem` 组成，每个 `TIMElem` 可以是文本和图片，也就是说每一条消息可包含多个文本和多张图片。



发消息通过 `TIMConversation` 的方法 `sendMessage` 实现。

原型：

```
/**
 * 发送消息
 * @param msg 消息
 * @param callback 回调
```

```
*/
public void sendMessage(@NonNull TIMMessage msg, @NonNull TIMValueCallBack<TIMMessage> callback)
```

文本消息发送

文本消息由 `TIMTextElem` 定义。 **`TIMTextElem` 成员方法如下：**

```
//获取文本内容
java.lang.String getText()

//设置文本内容，text 传递需要发送的文本消息
void setText(java.lang.String text)
```

示例：

```
//构造一条消息
TIMMessage msg = new TIMMessage();

//添加文本内容
TIMTextElem elem = new TIMTextElem();
elem.setText("a new msg");

//将elem添加到消息
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}

//发送消息
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {//发送消息回调
@Override
public void onError(int code, String desc) {//发送消息失败
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 含义请参见错误码表
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess(TIMMessage msg) {//发送消息成功
Log.e(tag, "SendMsg ok");
}
});
```

图片消息发送

图片消息由 `TIMImageElem` 定义。它是 `TIMElem` 的一个子类，也就是说图片也是消息的一种内容。 发送图片的过程，就是将 `TIMImageElem` 加入到 `TIMMessage` 中，然后随消息一起发送出去。

> 注意：
> `path` 不支持 `file://` 开头的文件路径，需要去掉 `file://` 前缀。

`TIMImageElem` 成员方法如下：

```
/**
* 从 IM SDK 取出 elem 时可以调用，获取 elem 包含的图片列表
* @return elem 包含的图片列表
*/
public ArrayList<TIMImage> getImageList()

/**
* 获取原图本地文件路径，只对消息发送方有效
* @return 本地文件路径
*/
public String getPath()
```

```
/**
 * 发送消息时，设置待发送的原图文件路径
 * @param path 原图文件路径
 */
public void setPath(String path)

/**
 * 获取图片质量级别
 * @return 图片质量级别，0: 原图发送 1: 高压缩率图发送(图片较小) 2:高清图发送(图片较大)
 */
public int getLevel()

/**
 * 设置图片质量级别
 * @param level 0: 原图发送 1: 高压缩率图发送(图片较小，默认值) 2:高清图发送(图片较大)
 */
public void setLevel(int level)

/**
 * 取消图片上传
 * @return 取消图片上传是否成功
 */
public boolean cancelUploading()

/**
 * 获取图片上传任务 ID, 调用 sendMessage 后此接口的返回值有效
 * @return 图片上传任务 ID
 */
public int getTaskId()

/**
 * 获取图片类型
 * @return 图片类型
 */
public int getImageFormat()
```

发送图片时，只需要设置图片路径 `path` 。发送成功后可通过 `getImageList` 获取所有图片类型。 `TIMImage` 存储了图片列表的类型，大小，宽高信息，如需要图片二进制数据，需通过 `getImage` 接口下载。

`TIMImage` 成员方法如下：

```
/**
 * 获取图片
 * @param path 图片保存路径
 * @param cb 回调
 */
public void getImage(@NonNull final String path, @NonNull final TIMCallBack cb)

/**
 * 获取图片类型
 * @return 图片类型
 */
public TIMImageType getType()

/**
 * 获取 uuid
 * @return uuid，可作为唯一标示用于缓存的 key
 */
public String getUuid()

/**
 * 获取图片大小
 * @return 图片大小
 */
public long getSize()

/**
```

```
 * 获取图片高度
 * @return 图片高度
 */
public long getHeight()

/**
 * 获取图片宽度
 * @return 图片宽度
 */
public long getWidth()

/**
 * 获取图片 url
 * @return 图片 url
 */
public String getUrl()
```

示例：

```
//构造一条消息
TIMMessage msg = new TIMMessage();

//添加图片
TIMImageElem elem = new TIMImageElem();
elem.setPath(Environment.getExternalStorageDirectory() + "/DCIM/Camera/1.jpg");

//将 elem 添加到消息
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}

//发送消息
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {//发送消息回调
@Override
public void onError(int code, String desc) {//发送消息失败
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 列表请参见错误码表
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess(TIMMessage msg) {//发送消息成功
Log.e(tag, "SendMsg ok");
}
});
```

表情消息发送

表情消息由 `TIMFaceElem` 定义，IM SDK 并不提供表情包，如果开发者有表情包，可使用 `index` 存储表情在表情包中的索引，由用户自定义，或者直接使用data存储表情二进制信息以及字符串 key，都由用户自定义，IM SDK 内部只做透传。

`TIMFaceElem` 成员方法如下：

```
/**
 * 获取表情索引
 * @return 表情索引
 */
public int getIndex()

/**
 * 设置表情索引
 * @param index 表情索引
 */
public void setIndex(int index)

/**
```

```
 * 获取表情自定义数据
 * @return 表情自定义数据
 */
public byte[] getData()

/**
 * 设置表情自定义数据
 * @param data 表情自定义数据
 */
public void setData(byte[] data)
```

示例：

```
//构造一条消息
TIMMessage msg = new TIMMessage();

//添加表情
TIMFaceElem elem = new TIMFaceElem();
elem.setData(sampleByteArray); //自定义 byte[]
elem.setIndex(10); //自定义表情索引

//将 elem 添加到消息
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}

//发送消息
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {//发送消息回调
@Override
public void onError(int code, String desc) {//发送消息失败
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 含义请参见错误码表
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess(TIMMessage msg) {//发送消息成功
Log.e(tag, "SendMsg ok");
}
});
```

语音消息发送

语音消息由 `TIMSoundElem` 定义，其中 `data` 存储语音数据，语音数据需要提供时长信息，以秒为单位。

注意：

- 一条消息只能有一个语音 `Elem` ，添加多条语音 `Elem` 时， `AddElem` 函数返回错误 1，添加不生效。
- 语音和文件 `Elem` 不一定会按照添加时的顺序获取，建议逐个判断 `Elem` 类型展示，而且语音和文件 `Elem` 也不保证按照发送的 `Elem` 顺序排序。
- `path` 不支持 `file://` 开头的文件路径，需要去掉 `file://` 前缀。

`TIMSoundElem` 成员方法如下：

```
/**
 * 下载语音文件到指定的保存路径
 *
 * @param path 指定保存路径
 * @param progressCb 下载进度回调
 * @param cb 回调
 */
public void getSoundToFile(@NonNull final String path, final TIMValueCallBack<ProgressInfo> progressCb, @NonNull final TIMCallBack cb)

/**
 * 获取需要发送的语音文件的路径，只对发送方有效
```

```
 * @return 语音文件路径
 */
public String getPath()

/**
 * 设置需要发送的语音文件的路径（上传时，如果设置了文件路径，优先上传路径所指定的语音文件）
 * @param path 语音文件路径
 */
public void setPath(String path)

/**
 * 获取 uuid
 * @return uuid
 */
public String getUuid()

/**
 * 获取二进制数据长度
 * @return 二进制数据长度
 */
public long getDataSize()

/**
 * 获取语音时长
 * @return 语音时长
 */
public long getDuration()

/**
 * 设置语音时长
 * @param duration 语音时长
 */
public void setDuration(long duration)

/**
 * 获取语音上传任务 ID, 调用 sendMessage 后此接口的返回值有效
 * @return 语音文件上传任务ID
 */
public int getTaskId()
```

示例：

```
//构造一条消息
TIMMessage msg = new TIMMessage();

//添加语音
TIMSoundElem elem = new TIMSoundElem();
elem.setPath(filePath); //填写语音文件路径
elem.setDuration(20); //填写语音时长

//将 elem 添加到消息
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}
//发送消息
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {//发送消息回调
@Override
public void onError(int code, String desc) {//发送消息失败
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 含义请参见错误码表
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess(TIMMessage msg) {//发送消息成功
Log.e(tag, "SendMsg ok");
```

```
}
});
```

地理位置消息发送

地理位置消息由 `TIMLocationElem` 定义，其中 `desc` 存储位置的描述信息， `longitude` 、 `latitude` 分别表示位置的经度和纬度。

`TIMLocationElem` 成员方法如下：

```
/**
* 获取位置描述
* @return 位置描述
*/
public String getDesc()

/**
* 设置位置描述
* @param desc 位置描述
*/
public void setDesc(String desc)

/**
* 获取经度
* @return 经度
*/
public double getLongitude()

/**
* 设置经度
* @param longitude 经度
*/
public void setLongitude(double longitude)

/**
* 获取纬度
* @return 纬度
*/
public double getLatitude()

/**
* 设置纬度
* @param latitude 纬度
*/
public void setLatitude(double latitude)
```

示例：

```
//构造一条消息
TIMMessage msg = new TIMMessage();

//添加位置信息
TIMLocationElem elem = new TIMLocationElem();
elem.setLatitude(113.93); //设置纬度
elem.setLongitude(22.54); //设置经度
elem.setDesc("腾讯大厦");

//将elem添加到消息
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}
//发送消息
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {//发送消息回调
@Override
public void onError(int code, String desc) {//发送消息失败
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 含义请参见错误码表
```

```
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}
@Override
public void onSuccess(TIMMessage msg) {//发送消息成功
Log.e(tag, "SendMsg ok");
}
});
```

小文件消息发送

文件消息由 `TIMFileElem` 定义，另外还可以提供额外的显示文件名信息。

注意：

- 语音和文件 `Elem` 不一定会按照添加时的顺序获取，建议逐个判断 `Elem` 类型展示，而且语音和文件 `Elem` 也不保证按照发送的 `Elem` 顺序排序。
- `path` 不支持 `file://` 开头的文件路径，需要去掉 `file://` 前缀。
- 文件大小限制28MB。

`TIMFileElem` 成员方法如下：

```
/**
* 下载文件到指定的保存路径
* @param path 指定保存路径
* @param callback 回调
*/
public void getToFile(@NonNull final String path, @NonNull TIMCallBack callback)

/**
* 获取uuid
* @return uuid，可作为唯一标示用于缓存的 key
*/
public String getUuid()

/**
* 获取文件大小
* @return 文件大小
*/
public long getFileSize()

/**
* 获取显示文件名
* @return 文件名
*/
public String getFileName()

/**
* 设置显示文件名，在发送文件时进行设置
* @param fileName 文件名
*/
public void setFileName(String fileName)

/**
* 获取上传文件所在路径，只对发送方有效
* @return 文件路径
*/
public String getPath()

/**
* 设置上传文件所在路径（上传时，如果设置了文件路径，优先上传路径所指定的文件）
* @param path 文件路径
*/
public void setPath(String path)

/**
* 获取文件上传任务 ID, 调用 sendMessage 后此接口的返回值有效
```

```
 * @return 文件上传任务 ID
 */
public int getTaskId()
```

示例：

```
//构造一条消息
TIMMessage msg = new TIMMessage();

//添加文件内容
TIMFileElem elem = new TIMFileElem();
elem.setPath(filePath); //设置文件路径
elem.setFileName("myfile.bin"); //设置消息展示用的文件名称

//将 elem 添加到消息
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}
//发送消息
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {//发送消息回调
@Override
public void onError(int code, String desc) {//发送消息失败
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 含义请参见错误码表
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess(TIMMessage msg) {//发送消息成功
Log.e(tag, "SendMsg ok");
}
});
```

自定义消息发送

自定义消息是指当内置的消息类型无法满足特殊需求，开发者可以自定义消息格式，内容全部由开发者定义，IM SDK 只负责透传。另外如果需要 iOS APNs 推送，还需要提供一段推送文本描述，方便展示。自定义消息由 `TIMCustomElem` 定义，其中 `data` 存储消息的二进制数据，其数据格式由开发者定义，`desc` 存储描述文本。一条消息内可以有多个自定义 `Elem`，并且可以跟其他 `Elem` 混合排列，离线 Push 时叠加每个 `Elem` 的 `desc` 描述信息进行下发。

`TIMCustomElem` 成员方法如下：

```
/**
* 获取自定义数据
* @return 自定义数据
*/
public byte[] getData()

/**
* 设置自定义数据
* @param data 自定义数据
*/
public void setData(byte[] data)

/**
* 获取自定义描述
* @return 自定义描述
*/
public String getDesc()

/**
* 设置自定义描述
* @param desc 自定义描述
*/
public void setDesc(String desc)

/**
```

```
 * 获取后台推送对应的 ext 字段
 * @return ext
 */
public byte[] getExt()

/**
 * 设置后台推送对应的 ext 字段
 * @param ext 后台推送对应的 ext 字段
 */
public void setExt(byte[] ext)

/**
 * 获取自定义声音
 * @return 自定义声音的数据
 */
public byte[] getSound()

/**
 * 设置自定义声音的数据
 * @param data 自定义声音的数据
 */
public void setSound(byte[] data)
```

示例中拼接一段 XML 消息，具体展示由开发者决定。**示例：**

```
//构造一条消息
TIMMessage msg = new TIMMessage();

// xml 协议的自定义消息
String sampleXml = "<!--?xml version='1.0' encoding="utf-8"?-->testTitlethis is custom msgtest msg body";

//向 TIMMessage 中添加自定义内容
TIMCustomElem elem = new TIMCustomElem();
elem.setData(sampleXml.getBytes()); //自定义 byte[]
elem.setDesc("this is one custom message"); //自定义描述信息

//将 elem 添加到消息
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}
//发送消息
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {//发送消息回调
@Override
public void onError(int code, String desc) {//发送消息失败
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 含义请参见错误码表
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess(TIMMessage msg) {//发送消息成功
Log.e(tag, "SendMsg ok");
}
});
```

短视频消息发送

短视频消息由 `TIMVideoElem` 定义。它是 `TIMElem` 的一个子类，也就是说视频截图和视频内容也是消息的一种内容。发送短视频的过程，就是将 `TIMVideoElem` 加入到 `TIMMessage` 中，然后随消息一起发送出去。

`TIMVideoElem` 原型：

```
/**
 * 获取微视频上传任务 ID, 调用 sendMessage 后此接口的返回值有效
 *
 * @return 微视频上传任务 ID
```

```
*/
public long getTaskId() {
return this.taskId;
}

/**
* 设置微视频信息，在发送消息时进行设置
*
* @param video 微视频信息，详见{@link TIMVideo}
*/
public void setVideo(TIMVideo video) {
this.video = video;
}

/**
* 获取视频信息
*
* @return 视频信息，详见{@link TIMVideo}
*/
public TIMVideo getVideoInfo() {
return this.video;
}

/**
* 设置视频文件路径，在发送消息时进行设置
*
* @param path 视频文件路径
*/
public void setVideoPath(String path) {
this.videoPath = path;
}

/**
* 获取视频文件路径
*
* @return 视频文件路径
*/
public String getVideoPath() {
return this.videoPath;
}

/**
* 设置微视频截图信息，在发送消息时进行设置
*
* @param snapshot 微视频截图信息，详见{@link TIMSnapshot}
*/
public void setSnapshot(TIMSnapshot snapshot) {
this.snapshot = snapshot;
}

/**
* 获取视频截图信息
*
* @return 视频截图信息，详见{@link TIMSnapshot}
*/
public TIMSnapshot getSnapshotInfo() {
return this.snapshot;
}

/**
* 设置微视频截图文件路径，在发送消息时进行设置
*
* @param path 微视频截图文件路径
*/
public void setSnapshotPath(String path) {
this.snapshotPath = path;
}
```

```
/**
 * 获取微视频截图文件路径
 *
 * @return 微视频截图文件路径
 */
public String getSnapshotPath() {
return this.snapshotPath;
}
```

参数说明：

参数	说明
taskId	上传时任务 ID，可用来查询上传进度（已废弃，请在 TIMUploadProgressListener 监听上传进度）
videoPath	发送短视频时，本地视频文件的路径
video	视频信息，发送消息时设置 type、duration 参数
snapshotPath	发送短视频时，本地截图文件的路径
snapshot	截图信息，发送消息时设置 type、width、height 参数

以下示例中发送了一个短视频消息。**示例：**

```
//构造一条消息
TIMMessage msg = new TIMMessage();

//构造一个短视频对象
TIMVideoElem ele = new TIMVideoElem();

TIMVideo video = new TIMVideo();
video.setDuaration(duration / 1000); //设置视频时长
video.setType("mp4"); // 设置视频文件类型

TIMSnapshot snapshot = new TIMSnapshot();
snapshot.setWidth(width); // 设置视频快照图宽度
snapshot.setHeight(height); // 设置视频快照图高度

ele.setSnapshot(snapshot);
ele.setVideo(video);
ele.setSnapshotPath(imgPath);
ele.setVideoPath(videoPath);


//将 elem 添加到消息
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}

//发送消息
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {//发送消息回调
@Override
public void onError(int code, String desc) {//发送消息失败
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 含义请参见错误码表
Log.d(tag, "send message failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess(TIMMessage msg) {//发送消息成功
Log.e(tag, "SendMsg ok");
}
});
```

Elem 顺序

目前文件和语音 Elem 不一定会按照添加顺序传输，其他 Elem 按照顺序，不过建议不要过于依赖 Elem 顺序进行处理，应该逐个按照 Elem 类型处理，防止异常情况下进程 Crash。

在线消息

对于某些场景，需要发送在线消息，即用户在线时收到消息，如果用户不在线，下次登录也不会看到消息，可用于通知类消息，这种消息不会进行存储，也不会计入未读计数。发送接口与 sendMessage 类似。

> 注意：
> 2.5.3版本以前只针对单聊消息有效。2.5.3版本以后对群组消息有效(暂不支持 AVChatRoom 和 BChatRoom 类型)

```
//发送在线消息（服务器不保存消息）
public void sendOnlineMessage(TIMMessage msg, TIMValueCallBack<TIMMessage> callback)
```

消息转发

在 TIMMessage 中提供了 copyFrom 接口，可以方便地拷贝其他消息的内容到当前消息，然后将消息重新发送给其他人。

原型：

```
/**
* 复制消息内容到当前消息（Elem, priority, online, offlinePushInfo 等）
* @param srcMsg 源消息
* @return true 复制成功
*/
public boolean copyFrom(@NonNull TIMMessage srcMsg)
```

接收消息

用户需要感知新消息的通知时，只需注册新消息通知回调 TIMMessageListener ，如果用户是登录状态，IM SDK 收到新消息会通过回调中的 onNewMessages 抛出。注册方法请参考 新消息通知。

> 注意：
> 通过 onNewMessages 抛出的消息不一定是未读的消息，只是本地曾经没有过的消息（例如在另外一个终端已读，拉取最近联系人消息时可以获取会话最后一条消息，如果本地没有，会通过此方法抛出）。在用户登录之后，IM SDK 会拉取 C2C 离线消息，为了不漏掉消息通知，需要在登录之前注册新消息通知。
> 群系统消息、关系链变化、好友资料变更也会通过该回调 onNewMessages 抛出。

消息解析

收到消息后，可用过 getElem 从 TIMMessage 中获取所有的 Elem 节点。**遍历 Elem 原型如下：**

```
//获取消息元素
TIMElem getElement(int i)

//获取元素个数
int getElementCount()
```

示例：

```
TIMMessage msg = /* 消息 */

for(int i = 0; i < msg.getElementCount(); ++i) {
TIMElem elem = msg.getElement(i);

//获取当前元素的类型
TIMElemType elemType = elem.getType();
Log.d(tag, "elem type: " + elemType.name());
if (elemType == TIMElemType.Text) {
//处理文本消息
```

```
} else if (elemType == TIMElemType.Image) {
//处理图片消息
}//...处理更多消息
}
```

接收图片消息

接收方收到消息后，可通过 `getElem` 从 `TIMMessage` 中获取所有的 `Elem` 节点，其中类型为 `TIMElemType.Image` 的是图片消息节点。然后通过 `TIMImageElem` 中的 `getImageList` 获取该图片的所有规格，目前最多包含三种规格：原图、大图、缩略图， 每种规格保存在一个 `TIMImage` 对象中。

```
/**
* 从 IM SDK 取出 Elem 时可以调用，获取 Elem 包含的图片列表
* @return elem 包含的图片列表
*/
public ArrayList<TIMImage> getImageList()
```

TIMImage说明：

获取到消息时通过imageList得到所有的图片规格，为 `TIMImage` 数据，得到 `TIMImage` 后可通过图片大小进行占位，通过接口 `getImage` 下载不同规格的图片进行展示。

> 注意：
> 下载的数据需要由开发者缓存，IM SDK 每次调用 `getImage` 都会从服务端重新下载数据。建议通过图片的 `uuid` 作为 `key` 进行图片文件的存储。

图片规格说明：每幅图片有三种规格，分别是 Original(原图)、Large(大图)、Thumb(缩略图)。

- **原图：**指用户发送的原始图片，尺寸和大小都保持不变。
- **大图：**是将原图等比压缩，压缩后宽、高中较小的一个等于 720 像素。
- **缩略图：**是将原图等比压缩，压缩后宽、高中较小的一个等于 198 像素。

> - 如果原图尺寸就小于 198 像素，则三种规格都保持原始尺寸，不需压缩。
> - 如果原图尺寸在 198~720 之间，则大图和原图一样，不需压缩。
> - 在手机上展示图片时，建议优先展示缩略图，用户单击缩略图时再下载大图，单击大图时再下载原图。当然开发者也可以选择跳过大图，单击缩略图时直接下载原图。
> - 在 Pad 或 PC 上展示图片时，由于分辨率较大，且基本都是 Wi-Fi 或有线网络，建议直接显示大图，用户单击大图时再下载原图。

示例：

```
//遍历一条消息的元素列表
for(int i = 0; i < msg.getElementCount(); ++i) {
TIMElem elem = msg.getElement(i);
if (elem.getType() == TIMElemType.Image) {
//图片元素
TIMImageElem e = (TIMImageElem) elem;
for(TIMImage image : e.getImageList()) {

//获取图片类型, 大小, 宽高
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) {//获取图片失败
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 含义请参见错误码表
Log.d(tag, "getImage failed. code: " + code + " errmsg: " + desc);
}

@Override
```

```
public void onSuccess() {//成功，参数为图片数据
//doSomething
Log.d(tag, "getImage success.");
}
});
}
}
}
```

接收语音消息

收到消息后，可用过 getElem 从 TIMMessage 中获取所有的 Elem 节点，其中类型为 TIMElemType.Sound 的为语音消息节点。获取到消息时可通过时长占位，通过接口 getSoundToFile 下载语音资源， getSoundToFile 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。

原型：

```
/**
* 下载语音文件到指定的保存路径
*
* @param path 指定保存路径
* @param progressCb 下载进度回调
* @param cb 回调
*/
public void getSoundToFile(@NonNull final String path, final TIMValueCallBack<ProgressInfo> progressCb, @NonNull final TIMCallBack cb)
```

语音消息已读状态：语音是否已经播放，可使用 消息自定义字段 实现，例如 customInt 的值 0 表示未播放，1 表示播放，当用户单击播放后可设置 customInt 的值为 1。以下为设置自定义整数， 默认为 0。

原型：

```
public void setCustomInt(int value)
```

接收小文件消息

收到消息后，可用过 getElem 从 TIMMessage 中获取所有的 Elem 节点，其中 TIMFileElem 为文件消息节点。

TIMFileElem 成员方法如下：

```
//下载文件到指定的保存路径
void getToFile(String path, TIMCallBack callback)

//获取文件名
java.lang.String getFileName()

//获取文件大小
long getFileSize()

//获取uuid
java.lang.String getUuid()

//设置文件名
void setFileName(java.lang.String fileName)
```

获取到消息时可只展示文件大小和显示名，通过接口 getToFile 下载文件资源。 getToFile 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。

原型：

```
/**
* 下载文件到指定的保存路径
* @param path 指定保存路径
* @param callback 回调
*/
public void getToFile(@NonNull final String path, @NonNull TIMCallBack callback)
```

接收短视频消息

收到消息后，可通过 getElem 从 TIMMessage 中获取所有的 Elem 节点，其中 TIMVideoElem 为文件消息节点，通过 TIMVideo 和 TIMSnapshot 对象获取视频和截图内容。接收到 TIMVideoElem 后，通过 video 属性和 snapshot 属性中定义的接口下载视频文件和截图文件。如需缓存或存储，开发者可根据 UUID 作为 key 进行外部存储，IM SDK 并不会存储资源文件。

TIMVideo 成员方法如下：

```
/**
 * 获取视频
 *
 * @param path 视频保存路径
 * @param cb 回调
 * @deprecated
 */
getVideo(@NonNull final String path, @NonNull final TIMCallBack cb);

/**
 * 获取视频
 *
 * @param path 视频保存路径
 * @param progressCb 下载进度回调
 * @param cb 回调
 */
void getVideo(@NonNull final String path, final TIMValueCallBack<ProgressInfo> progressCb, @NonNull final TIMCallBack cb)

/**
 * 获取视频文件大小
 *
 * @return 返回视频文件大小
 */
long getSize();

/**
 * 获取视频文件 UUID
 *
 * @return uuid，可作为唯一标示用于缓存的 key
 */
String getUuid();

/**
 * 获取视频时长
 *
 * @return 返回视频时长
 */
long getDuaration();

/**
 * 获取视频文件类型
 *
 * @return 返回视频文件类型
 */
String getType();
```

TIMSnapshot 成员方法如下：

```
/**
 * 获取截图
 *
 * @param path 保存截图的路径
 * @param progressCb 下载进度回调
 * @param cb 回调
 */
void getImage(final String path, final TIMValueCallBack<ProgressInfo> progressCb, final TIMCallBack cb);

/**
 * 获取截图
```

```
*
* @param path 保存截图的路径
* @param cb 回调
* @deprecated
*/
void getImage(final String path, final TIMCallBack cb);

/**
* 获取截图宽度
*
* @return 截图宽度
*/
long getWidth();

/**
* 获取截图高度
*
* @return 截图高度
*/
long getHeight();

/**
* 获取截图文件大小
*
* @return 返回截图文件大小
*/
long getSize();

/**
* 获取截图文件类型
*
* @return 返回视频文件类型
*/
String getType();

/**
* 获取截图文件uuid
*
* @return uuid，可作为唯一标示用于缓存的key
*/
String getUuid();
```

短视频消息的解析过程：

以收到新消息回调为例，需要先通过 element 的 type 判断是否为 TIMVideoElem，若是则表示该消息为短视频消息，需执行以下代码进行解析。

```
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(); //接收到的快照图片保存的路径
final String videoPath = " /xxx/ " + video.getUuid(); //接收到的视频保存的路径
videoEle.getSnapshotInfo().getImage(path, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "下载快照图片失败，code = " + code + ", errorinfo = " + desc);
}

@Override
public void onSuccess() {
Log.d(tag, "下载快照图片成功");
}
});

video.getVideo(videoPath, new TIMCallBack() {
@Override
public void onError(int code, String desc) {
Log.e(tag, "下载短视频失败，code = " + code + ", errorinfo = " + desc);
```

```
}

@Override
public void onSuccess() {
Log.d(tag, "下载短视频成功");
}
});
```

消息属性

可通过 TIMMessage 的成员方法获取消息属性。

消息是否已读

通过 TIMMessage 的方法 isRead 可以获取消息是否已读。这里已读与否取决于 App 侧进行的 已读上报。消息是否已读的原型如下。

原型：

```
public boolean isRead()
```

消息状态

通过 TIMMessage 的方法 status 可以获取当前消息的状态，如发送中、发送成功、发送失败和删除，对于删除的消息，需要 UI 判断状态并隐藏。

```
//发送中
TIMMessageStatus.Sending

//发送成功
TIMMessageStatus.SendSucc

//发送失败
TIMMessageStatus.SendFail

//删除
TIMMessageStatus.HasDeleted

//消息被撤回
TIMMessageStatus.HasRevoked
```

是否自己发出的消息

通过 TIMMessage 的方法 isSelf 可以判断消息是否是自己发出的消息，界面显示时可用。消息是否是自己发出的原型如下。

原型：

```
public boolean isSelf()
```

消息发送者及其相关资料

可以通过 TIMMessage 的方法 getSender 获取发送用户的 ID。
对于单聊消息，可以通过 TIMMessage 的 getConversation 获取到对应会话，会话的 getPeer 即可得到正在聊天的用户及其相关资料。
对于群消息，可以通过 getSenderProfile 和 getSenderGroupMemberProfile 获取发送者的资料和所在群的资料。如需拉取自定义字段，需在登录 IM SDK 之前 设置拉取字段 。

> 注意：
> 此字段是消息发送时获取用户资料写入消息体，如后续用户资料更新，此字段不会相应变更，只有产生的新消息中才会带最新的昵称。
> 只有接收到的群消息才能获取到相应的资料。

```
/**
* 获取消息发送方
* @return 消息发送方 user
```

```
 */
public String getSender()

/**
 * 获取发送者资料
 *
 * 4.4.716 版本统一通过回调返回
 *
 * @param callBack 回调
 */
public void getSenderProfile( TIMValueCallBack < TIMUserProfile > callBack )

/**
 * 获取发送者群内资料，只有接收到的群消息才能获取到资料（发送者为自己时可能为空）
 *
 * @return 发送者群内资料，null 表示没有获取到资料或者不是群消息，目前仅能获取字段：user、nameCard、role、customInfo，其他的字段获取建议通过 TIMGr
oupManager -> getGroupMembers 获取
 */
public TIMGroupMemberInfo getSenderGroupMemberProfile()
```

消息时间

通过 TIMMessage 的方法 timestamp 可以得到消息时间，**该时间是 Server 时间，而非本地时间。**在创建消息时，此时间为根据 Server 时间校准过的时间，发送成功后会改为准确的 Server 时间。

```
//消息在服务端生成的时间戳
public long timestamp()
```

消息 ID

消息 ID 也有两种，一种是当消息生成时，就已经固定（ msgId ），这种方式可能跟其他用户产生的消息冲突，需要再加一个时间维度，可以认为 10 分钟以内的消息可以使用 msgId 区分。另外一种，当消息发送成功以后才能固定下来（ uniqueId ），这种方式能保证全局唯一。这两种方式都需要在同一个会话内判断。

```
//获取消息 ID
public String getMsgId()

//获取消息uniqueId
public long getMsgUniqueId()
```

消息自定义字段

开发者可以对消息增加自定义字段，如自定义整数、自定义二进制数据，可以根据这两个字段做出各种不同效果，例如语音消息是否已经播放等等。另外需要注意，此自定义字段仅存储于本地，不会同步到 Server，更换终端获取不到。

```
//设置自定义整数， 默认为 0
public void setCustomInt(int value)

//获取自定义整数值
public int getCustomInt()

//设置自定义数据内容，默认为""
public void setCustomStr(String str)

//获取自定义数据内容的值
public String getCustomStr()
```

消息优先级

对于直播场景，会有点赞和发红包功能，点赞相对优先级较低，红包消息优先级较高，具体消息内容可以使用 TIMCustomElem 进行定义，发送消息时，可使用不同接口定义消息优先级。

> 注意：
> 只针对群聊消息有效。

```
//设置消息优先级
public void setPriority(TIMMessagePriority priority)

//获取消息优先级
public TIMMessagePriority getPriority()
```

已读回执

IM SDK 提供**针对于 C2C 消息**的已读回执功能。通过 `TIMUserConfig` 中的 `enableReadReceipt` 接口可以启用消息已读回执功能。启用已读回执功能后，在进行 消息已读上报 时会发送已读回执给对方。

通过 `TIMUserConfig` 的接口 `setMessageReceiptListener` 可以注册已读回执监听器。通过 `TIMMessage` 中的 `isPeerReaded` 可以查询当前消息对方是否已读。

原型：

```
/**
* 启用已读回执，启用后在已读上报时会发送回执给对方，只对单聊会话有效
*/
public void enableReadReceipt()

/**
* 设置已读回执监听器
* @param receiptListener 已读回执监听器
*/
public void setMessageReceiptListener(TIMMessageReceiptListener receiptListener)

/**
* 获取对方是否已读（仅对 C2C 消息有效）
* @return true - 对方已读， false - 对方未读
*/
public boolean isPeerReaded()
```

消息序列号

通过 `TIMMessage` 中的接口 `getSeq` 可以获取到当前消息的序列号。

```
/**
* 获取当前消息的序列号
* @return 当前消息的序列号
*/
public long getSeq()
```

消息随机码

通过 `TIMMessage` 中的接口 `getRand` 可以获取到当前消息的随机码。

```
/**
* 获取当前消息的随机码
* @return 当前消息的随机码
*/
public long getRand()
```

消息查找参数

IM SDK 中的消息需要通过 {seq, rand, timestamp, isSelf} 四元组来唯一确定一条具体的消息，我们把这个四元组称为消息的查找参数。通过 `TIMMessage` 中的 `getMessageLocator` 接口可以从当前消息中获取到当前消息的查找参数。

```
/**
* 获取当前消息的查找参数
* @return 当前消息的查找参数
*/
public TIMMessageLocator getMessageLocator()
```

会话操作

获取所有会话

通过 `TIMManager` 的 `getConversationList` 获取当前会话数量，从而得到所有本地会话。

> 注意：
> SDK 会在内部不断更新会话列表，每次更新后都会通过 `TIMRefreshListener.onRefresh` 回调，请**在 `onRefresh` 之后再调用 `getConversationList`** 更新会话列表。

原型：

```
/**
 * 获取所有会话
 * @return 会话列表
 */
public List<TIMConversation> getConversationList()
```

示例：

```
List<TIMConversation> list = TIMManager.getInstance().getConversationList();
```

最近联系人漫游

IM SDK 登录以后默认会获取最近联系人漫游，同时每个会话会获取到最近的一条消息。

获取会话本地消息

IM SDK 会在本地进行消息存储，可通过 `TIMConversation` 方法的 `getLocalMessage` 获取，此方法为异步方法，需要通过设置回调得到消息数据，**对于单聊，登录后会自动获取离线消息，对于群聊，开启最近联系人漫游的情况下，登录后只能获取最近一条消息，可通过 `getMessage` 获取漫游消息。**

> 注意：
> 对于图片、语音等资源类消息，消息体只会包含描述信息，需要通过额外的接口下载数据，可参与消息解析部分，下载后的真实数据不会缓存，需要调用方进行缓存。

原型：

```
/**
 * 仅获取本地聊天记录
 * @param count 从最后一条消息往前的消息条数
 * @param lastMsg 已获取的最后一条消息, 为 null 时表示最新一条消息
 * @param callback 回调, 参数中返回获取的消息列表
 */
public void getLocalMessage(int count, TIMMessage lastMsg, @NonNull TIMValueCallBack<List<TIMMessage>> callback)
```

示例：

```
//获取会话扩展实例
TIMConversation con = TIMManager.getInstance().getConversation(TIMConversationType.Group, groupId);

//获取此会话的消息
con.getLocalMessage(10, //获取此会话最近的 10 条消息
null, //不指定从哪条消息开始获取 - 等同于从最新的消息开始往前
new TIMValueCallBack<List<TIMMessage>>() {//回调接口
@Override
public void onError(int code, String desc) {//获取消息失败
//接口返回了错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 含义请参见错误码表
Log.d(tag, "get message failed. code: " + code + " errmsg: " + desc);
}

@Override
```

```
public void onSuccess(List<TIMMessage> msgs) {//获取消息成功
//遍历取得的消息
for(TIMMessage msg : msgs) {
lastMsg = msg;
//可以通过 timestamp()获得消息的时间戳, isSelf()是否为自己发送的消息
Log.e(tag, "get msg: " + msg.timestamp() + " self: " + msg.isSelf() + " seq: " + msg.getSeq());
}
}
});
```

获取会话漫游消息

对于群组，登录后可以获取漫游消息，对于C2C，开通漫游服务后可以获取漫游消息，通过 `TIMConversation` 的 `getMessage` 接口可以获取漫游消息，如果本地消息全部都是连续的，则不会通过网络获取，如果本地消息不连续，会通过网络获取断层消息。

> 注意：
> 对于图片、语音等资源类消息，消息体只会包含描述信息，需要通过额外的接口下载数据，可参与消息解析部分，下载后的真实数据不会缓存，需要调用方进行缓存。

原型：

```
/**
* 获取聊天记录
* @param count 从最后一条消息往前的消息数
* @param lastMsg 已取得的最后一条消息
* @param callback 回调, 参数中返回获取的消息列表
*/
public void getMessage(int count, TIMMessage lastMsg, @NonNull TIMValueCallBack< List<TIMMessage> > callback)
```

示例：

```
//获取会话扩展实例
TIMConversation con = TIMManager.getInstance().getConversation(TIMConversationType.Group, groupId);

//获取此会话的消息
con.getMessage(10, //获取此会话最近的 10 条消息
null, //不指定从哪条消息开始获取 - 等同于从最新的消息开始往前
new TIMValueCallBack<List<TIMMessage>>() {//回调接口
@Override
public void onError(int code, String desc) {//获取消息失败
//接口返回了错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 含义请参见错误码表
Log.d(tag, "get message failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess(List<TIMMessage> msgs) {//获取消息成功
//遍历取得的消息
for(TIMMessage msg : msgs) {
lastMsg = msg;
//可以通过 timestamp()获得消息的时间戳, isSelf()是否为自己发送的消息
Log.e(tag, "get msg: " + msg.timestamp() + " self: " + msg.isSelf() + " seq: " + msg.msg.seq());

}
}
});
```

删除会话

删除会话的同时 IM SDK 会删除该会话的本地和漫游消息，会话和消息删除后，无法再恢复。

原型：

```
/**
* 删除本地和服务器上保存的单个会话，以及该会话中的本地和服务器的所有消息
```

```
*
* @param type 会话类型
* @param peer 参与会话的对方, C2C 会话为对方帐号 identifier, 群组会话为群组 ID
* @return true 成功 false 失败
*/
public boolean deleteConversation(TIMConversationType type, String peer)
```

以下示例中删除了与用户 user1 的 C2C 会话。**示例：**

```
TIMManager.getInstance().deleteConversation(TIMConversationType.C2C, "user1");
```

同步获取会话最后的消息

UI 展示最近联系人列表时，时常会展示用户的最后一条消息，IM SDK 在 `TIMConverstion` 中提供了同步获取会话最近消息的接口 `getLastMsg` ，用户可以通过此接口方便获取最后一条消息进行展示。**目前没有网络无法获取，另外如果禁用了最近联系人，登录后在有新消息过来之前无法获取。**此接口获取并不会过滤删除状态消息，需要 App 层进行屏蔽。获取最近的多条消息，可以通过 `getMessage` 来获取。

原型：

```
/**
* 从 cache 中获取最后一条消息
* @return 最后一条消息。会话非法时，返回 null
*/
public TIMMessage getLastMsg()

/**
* 获取聊天记录
* @param count 从最后一条消息往前的消息数
* @param lastMsg 已取得的最后一条消息
* @param callback 回调, 参数中返回获取的消息列表
*/
public void getMessage(int count, TIMMessage lastMsg, @NonNull TIMValueCallBack< List<TIMMessage> > callback)
```

设置会话草稿

IM SDK 提供了会话草稿功能，开发者可以通过 `TIMConversation` 中的相关接口进行草稿操作。

注意：

- 草稿只能本地有效，更换终端或者清除数据后将看不到草稿。
- 草稿信息会存本地数据库，重新登录后依然可以获取。

原型：

```
/**
* 设置草稿
* @param draft 草稿内容, 为 null 则表示取消草稿
*/
public void setDraft(TIMMessageDraft draft)

/**
* 获取草稿
* @return 返回草稿内容
*/
public TIMMessageDraft getDraft()

/**
* 当前会话是否存在草稿
* @return true - 存在，false - 不存在
*/
public boolean hasDraft()
```

`TIMMessageDraft` 说明如下：

```
/**
 * 获取草稿中的消息元素列表
 * @return 消息元素列表
 */
public List<TIMElem> getElems()

/**
 * 设置草稿中的消息元素
 * @param elem 要添加到草稿中的消息元素
 */
public void addElem(TIMElem elem)

/**
 * 获取草稿中用户自定义数据
 * @return 用户自定义数据
 */
public byte[] getUserDefinedData()

/**
 * 设置草稿中用户自定义数据
 * @param userDefinedData 用户自定义数据
 */
public void setUserDefinedData(byte[] userDefinedData)

/**
 * 获取草稿的编辑时间
 * @return 草稿编辑时间
 */
public long getTimestamp()
```

删除会话消息

IM SDK 支持删除会话的本地及漫游消息，消息删除后，无法再恢复。

原型：

```
/**
 * 删除当前会话的本地及漫游消息
 *
 * 该接口会删除本地历史的同时也会把漫游消息即保存在服务器上的消息也删除，卸载重装后无法再拉取到。需要注意的是：
 * 1. 一次最多只能删除 30 条消息。
 * 2. 一秒钟最多只能调用一次。
 * 3. 如果该账号在其他设备上拉取过这些消息，那么调用该接口删除后，这些消息仍然会保存在那些设备上，即删除消息不支持多端同步。
 */
public void deleteMessages(List<TIMMessage> messages, TIMCallBack callback)
```

查找本地消息

IM SDK 提供了根据提供参数查找相应消息的功能，只能精准查找，暂时不支持模糊查找。开发者可以通过调用 `TIMConversation` 中的 `findMessages` 方法进行消息查找。

```
/**
 * 根据提供的参数查找相应消息
 * @param locators 消息查找参数
 * @param cb 回调，返回查找到的消息
 */
public void findMessages(@NonNull List<TIMMessageLocator> locators, TIMValueCallBack<List<TIMMessage>> cb)
```

其中参数中的 `TIMMessageLocator` 可以通过 `TIMMessage` 中的 `getMessageLocator` 方法来获取。

原型：

```
/**
 * 获取当前消息的消息定位符
 * @return 当前消息的消息定位符
```

```
 */
public TIMMessageLocator getMessageLocator()
```

撤回消息

IM SDK 在 3.1.0 版本开始提供撤回消息的接口。可以通过调用 `TIMConversation` 的 `revokeMessage` 接口来撤回自己发送的消息。

> 注意：
>
> - 仅 C2C 和 GROUP 会话有效、onlineMessage 无效、AVChatRoom 和 BChatRoom 无效。
> - 默认只能撤回 2 分钟内的消息。

原型：

```
/**
 * 消息撤回（仅 C2C 和 GROUP 会话有效，其中 onlineMessage、AVChatRoom 和 BChatRoom 无效）
 * @param msg 需要撤回的消息
 * @param cb 回调
 * @since 3.1.0
 */
public void revokeMessage(@NonNull TIMMessage msg, @NonNull TIMCallBack cb)
```

成功撤回消息后，群组内其他用户和 C2C 会话对端用户会收到一条消息撤回通知，并通过消息撤回通知监听器 `TIMMessageRevokeListener` 通知到上层应用。消息撤回通知监听器可以在登录前，通过 `TIMUserConfig` 的 `setMessageRevokedListener` 来进行配置。具体可以参考 用户配置。

原型：

```
/**
 * 消息被撤回通知监听器
 * @since 3.1.0
 */
public interface TIMMessageRevokedListener extends IMBaseListener {
/**
 * 消息撤回通知
 * @param locator 被撤回的消息的消息定位符
 */
void onMessageRevoked(TIMMessageLocator locator);
}
```

收到一条消息撤回通知后，通过 `TIMMessage` 中的 `checkEquals` 方法判断当前消息是否是被对方撤回了，然后根据需要对 UI 进行刷新。

原型：

```
/**
 * 比较当前消息与提供的消息定位符表示的消息是否是同一条消息
 * @param locator 消息定位符
 * @return true - 表示是同一条消息； false - 表示不是同一条消息
 * @since 3.1.0
 */
public boolean checkEquals(@NonNull TIMMessageLocator locator)
```

系统消息

会话类型（TIMConversationType）除了 C2C 单聊和 Group 群聊以外，还有一种系统消息，系统消息不能由用户主动发送，是系统后台在相应的事件发生时产生的通知消息。系统消息目前分为两种，一种是关系链系统消息，一种是群系统消息。

- 关系链变更系统消息，当有用户加自己为好友，或者有用户删除自己好友的情况下，系统会发出变更通知，开发者可更新好友列表。相关细节可参阅 关系链变更系统通知。
- 当群资料变更，如群名变更或者群内成员变更，在群里会有系统发出一条群事件消息，开发者可在收到消息时可选择是否展示给用户，同时可刷新群资料或者群成员。详细内容可参阅 群事件消息。
- 当被管理员踢出群组，被邀请加入群组等事件发生时，系统会给用户发出群系统消息，相关细节可参阅 群系统消息。

设置后台消息通知栏提醒

IM SDK 后台在线时可以持续接收消息通知，如果此时程序在后台运行，可以以系统通知栏提醒的形式给用户呈现新的消息。新消息可以显示在顶部通知栏，通知中心或锁屏上。具体的实现方式可参考下面的示例：

示例：

```
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)//设置通知栏标题
.setContentText(contentStr)
.setContentIntent(intent) //设置通知栏单击意图
.setNumber(++pushNum) //设置通知集合的数量
.setTicker(senderStr+":"+contentStr) //通知首次出现在通知栏，带上升动画效果的
.setWhen(System.currentTimeMillis())//通知产生的时间，会在通知信息里显示，一般是系统获取到的时间
.setDefaults(Notification.DEFAULT_ALL)//向通知添加声音、闪灯和振动效果的最简单、最一致的方式是使用当前的用户默认设置，使用 defaults 属性，可以组合
.setSmallIcon(R.drawable.ic_launcher);//设置通知小 ICON
Notification notify = mBuilder.build();
notify.flags |= Notification.FLAG_AUTO_CANCEL;
mNotificationManager.notify(pushId, notify);
```

消息收发（iOS）

最近更新时间：2020-09-29 14:25:09

消息发送

通用消息发送

会话获取

会话是指面向一个人或者一个群组的对话，通过与单个人或群组之间会话收发消息，发消息时首先需要先获取会话，获取会话需要指定会话类型（群组或者单聊），以及会话对方标志（对方帐号或者群号）。获取会话由 `getConversation` 实现。

> 注意：
> 如果本地没有这个会话，调用会话 TIMConversation 的 API 会失败。建议在收到 TIMUserConfig > TIMRefreshListener 回调后再去操作 TIMConversation 对象。

原型：

```
@interface TIMManager : NSObject
/**
* 获取会话
*
* @param type 会话类型，TIM_C2C 表示单聊 TIM_GROUP 表示群聊
* @param conversationId C2C 为对方帐号identifier， GROUP 为群组 ID
*
* @return 会话对象
*/
- (TIMConversation*)getConversation:(TIMConversationType)type receiver:(NSString*)conversationId;
@end
```

参数说明：

参数	说明
type	会话类型，如果是单聊，填写 TIM_C2C，如果是群聊，填写 TIM_GROUP
conversationId	会话标识，单聊情况下，receiver 为对方帐号 identifier，群聊情况下，receiver 为群组 ID

获取对方 identifie r 为『iOS-001』的单聊会话：

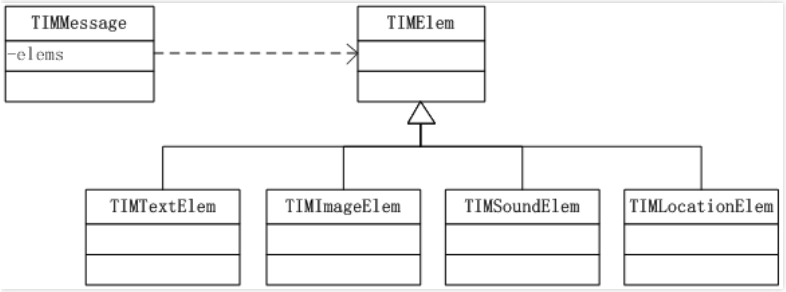
```
TIMConversation * c2c_conversation = [[TIMManager sharedInstance] getConversation:TIM_C2C receiver:@"iOS-001"];
```

获取群组 ID 为『TGID1JYSZEAEQ』的群聊会话示例：

```
TIMConversation * grp_conversation = [[TIMManager sharedInstance] getConversation:TIM_GROUP receiver:@"TGID1JYSZEAEQ"];
```

消息发送

通过 `TIMManager` 获取会话 `TIMConversation` 后，可发送消息和获取会话缓存消息。IM SDK 中消息的解释可参阅 IM SDK 基本概念。IM SDK 中的消息由 `TIMMessage` 表达，一个 `TIMMessage` 由多个 `TIMElem` 组成，每个 `TIMElem` 可以是文本和图片，也就是说每一条消息可包含多个文本和多张图片。发消息通过 `TIMConversation` 的成员 `sendMessage` 实现，有两种方式实现，一种使用闭包，另一种调用方实现 `protocol` 回调。



原型：

```
@interface TIMConversation : NSObject
-(int) sendMessage: (TIMMessage*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
msg	消息
succ	成功回调
fail	失败回调

文本消息发送

文本消息由 `TIMTextElem` 定义。

```
@interface TIMTextElem : TIMElem {
NSString * text;
}
```

示例：

> 说明：
>
> - text 传递需要发送的文本消息。
> - 失败回调中，code 表示错误码，具体可参阅 错误码，err 表示错误描述。

```
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);
}];
```

图片消息发送

图片消息由 `TIMImageElem` 定义。它是 `TIMElem` 的一个子类，也就是说图片也是消息的一种内容。 发送图片的过程，就是将 `TIMImageElem` 加入到 `TIMMessage` 中，然后随消息一起发送出去。发送图片时，只需要设置图片路径 `path` 。发送成功后可通过 `imageList` 获取所有图片类型。另外通过 `TIMUserConfig -> TIMUploadProgressListener` 监听当前上传进度。

`TIMImageElem` 原型：

```
/**
* 存储要发送的图片路径，必须是本地路径，可参考下面示例
*/
@interface TIMImageElem : TIMElem
/**
* 要发送的图片路径
*/
@property(nonatomic,retain) NSString * path;
/**
* 发送时不用关注，接收时保存生成的图片所有规格
*/
@property(nonatomic,retain) NSArray * imageList;
/**
* 上传时任务Id，可用来查询上传进度（已废弃，请在 TIMUploadProgressListener 监听上传进度）
*/
@property(nonatomic,assign) uint32_t taskId DEPRECATED_ATTRIBUTE;

/**
* 图片压缩等级，详见 TIM_IMAGE_COMPRESS_TYPE（仅对 jpg 格式有效）
*/
@property(nonatomic,assign) TIM_IMAGE_COMPRESS_TYPE level;

/**
* 图片格式，详见 TIM_IMAGE_FORMAT
*/
@property(nonatomic,assign) TIM_IMAGE_FORMAT format;
@end
```

参数说明：

参数	说明
path	存储要发送的图片路径，必须是本地路径，可参考图片发送示例
imageList	发送时不用关注，接收时保存生成的图片所有规格，可以参阅图片消息接收部分
taskId	发送图片时用来查询上传进度（已废弃，请在 TIMUploadProgressListener 监听上传进度）
level	发送图片前对图片进行压缩，level 表示压缩等级，详见 TIM_IMAGE_COMPRESS_TYPE 定义
format	图片格式，详见 TIM_IMAGE_FORMAT

以下示例中发送了一张绝对路径是 `/xxx/imgPath.jpg` 的图片。**示例：**

```
/**
* 获取聊天会话, 以同用户 iOS-001 的单聊为例
*/
TIMConversation * c2c_conversation = [[TIMManager sharedInstance] getConversation:TIM_C2C receiver:@"iOS-001"];
/**
* 构造一条消息
*/
TIMMessage * msg = [[TIMMessage alloc] init];
/**
* 构造图片内容
*/
TIMImageElem * image_elem = [[TIMImageElem alloc] init];
image_elem.path = @"/xxx/imgPath.jpg";
/**
* 将图片内容添加到消息容器中
*/
[msg addElem:image_elem];
/**
* 发送消息
*/
[conversation sendMessage:msg succ:^(){ //成功
NSLog(@"SendMsg Succ");
}fail:^(int code, NSString * err) { //失败
```

```
NSLog(@"SendMsg Failed:%d->%@", code, err);
}];
```

表情消息发送

表情消息由 `TIMFaceElem` 定义，IM SDK 并不提供表情包，如果开发者有表情包，可使用 `index` 存储表情在表情包中的索引，由用户自定义，或者直接使用 `data` 存储表情二进制信息以及字符串 `key` ，都由用户自定义，SDK 内部只做透传。

```
@interface TIMFaceElem : TIMElem
/**
* 表情索引，用户自定义
*/
@property(nonatomic, assign) int index;
/**
* 额外数据，用户自定义
*/
@property(nonatomic,retain) NSData * data;
@end
```

参数说明：

> 说明：
> index 和 data 只需要传入一个即可，IM SDK 只是透传这两个数据。

参数	说明
index	表情索引标号，由开发者定义
data	表情二进制数据，由开发者定义

以下示例中发送了索引为10的表情，具体10标识哪种表情，需要开发者在两端都持有一份表情包，索引到编号为10的表情，也可以通过 data 通过二进制数据来标识。**示例：**

```
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);
}];
```

语音消息发送

语音消息由 `TIMSoundElem` 定义，其中 `data` 存储语音数据，语音数据需要提供时长信息，以秒为单位。

> 注意：
> - 一条消息只能有一个语音 `Elem` ，添加多条语音 `Elem` 时， `AddElem` 函数返回错误1，添加不生效。
> - 语音和文件 `Elem` 不一定会按照添加时的顺序获取，建议逐个判断 `Elem` 类型展示，而且语音和文件 `Elem` 也不保证按照发送的 `Elem` 顺序排序。

```
/**
* 语音消息 Elem
*/
@interface TIMSoundElem : TIMElem
/**
* 上传时任务 ID，可用来查询上传进度（已废弃，请在 TIMUploadProgressListener 监听上传进度）
```

```
 */
@property(nonatomic,assign) uint32_t taskId DEPRECATED_ATTRIBUTE;
/**
 * 上传时，语音文件的路径，接收时使用 getSound 获得数据
 */
@property(nonatomic,strong) NSString * path;
/**
 * 存储语音数据
 */
@property(nonatomic,retain) NSData * data;
/**
 * 语音消息内部 ID
 */
@property(nonatomic,strong) NSString * uuid;
/**
 * 语音数据大小
 */
@property(nonatomic,assign) int dataSize;
/**
 * 语音长度（秒），发送消息时设置
 */
@property(nonatomic,assign) int second;

/**
 * 获取语音的 URL 下载地址
 *
 * @param urlCallBack 获取 URL 地址回调
 */
-(void)getUrl:(void (^)(NSString * url))urlCallBack;

/**
 * 获取语音数据到指定路径的文件中
 *
 * getSound 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。
 *
 * @param path 语音保存路径
 * @param succ 成功回调
 * @param fail 失败回调，返回错误码和错误描述
 */
- (void)getSound:(NSString*)path succ:(TIMSucc)succ fail:(TIMFail)fail;

/**
 * 获取语音数据到指定路径的文件中（有进度回调）
 *
 * getSound 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。
 *
 * @param path 语音保存路径
 * @param progress 语音下载进度
 * @param succ 成功回调
 * @param fail 失败回调，返回错误码和错误描述
 */
- (void)getSound:(NSString*)path progress:(TIMProgress)progress succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
path	上传语音的文件路径
uuid	上传成功以后会生成唯一的标识，用户可以根据此标识保存文件，IM SDK 内部不会保存资源数据
dataSize	语音数据大小
second	语音长度

示例：

```
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);
}];
```

地理位置消息发送

地理位置消息由 `TIMLocationElem` 定义，其中 `desc` 存储位置的描述信息， `longitude` 、 `latitude` 分别表示位置的经度和纬度。

```
@interface TIMLocationElem : TIMElem
/**
* 地理位置描述信息，发送消息时设置
*/
@property(nonatomic,retain) NSString * desc;
/**
* 纬度，发送消息时设置
*/
@property(nonatomic,assign) double latitude;
/**
* 经度，发送消息时设置
*/
@property(nonatomic,assign) double longitude;
@end
```

示例：

```
NSString *desc= @"腾讯大厦";
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);
}];
```

小文件消息发送

文件消息由 `TIMFileElem` 定义，另外还可以提供额外的显示文件名信息。

> 注意：
> 语音和文件 `Elem` 不一定会按照添加时的顺序获取，建议逐个判断 `Elem` 类型展示。

```
/**
* 文件消息 Elem
*/
@interface TIMFileElem : TIMElem
/**
* 上传时任务 ID，可用来查询上传进度（已废弃，请在 TIMUploadProgressListener 监听上传进度）
*/
@property(nonatomic,assign) uint32_t taskId DEPRECATED_ATTRIBUTE;
/**
* 上传时，文件的路径（设置 path 时，优先上传文件）
*/
```

```
@property(nonatomic,strong) NSString * path;
/**
* 文件内部 ID
*/
@property(nonatomic,strong) NSString * uuid;
/**
* 文件大小
*/
@property(nonatomic,assign) int fileSize;
/**
* 文件显示名，发消息时设置
*/
@property(nonatomic,strong) NSString * filename;

/**
* 获取文件的 URL 下载地址
*
* @param urlCallBack 获取 URL 地址回调
*/
-(void)getUrl:(void (^)(NSString * url))urlCallBack;

/**
* 获取文件数据到指定路径的文件中
*
* getFile 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。
*
* @param path 文件保存路径
* @param succ 成功回调，返回数据
* @param fail 失败回调，返回错误码和错误描述
*/
- (void)getFile:(NSString*)path succ:(TIMSucc)succ fail:(TIMFail)fail;

/**
* 获取文件数据到指定路径的文件中（有进度回调）
*
* getFile 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。
*
* @param path 文件保存路径
* @param progress 文件下载进度
* @param succ 成功回调，返回数据
* @param fail 失败回调，返回错误码和错误描述
*/
- (void)getFile:(NSString*)path progress:(TIMProgress)progress succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
path	文件路径
data	要发送的文件二进制数据。如设置 path，可不用设置 data，二者只需要设置一个字段即可，推荐使用 path
filename	文件名，IM SDK 不校验是否正确，只透传

示例：

```
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);
}];
```

自定义消息发送

自定义消息是指当内置的消息类型无法满足特殊需求，开发者可以自定义消息格式，内容全部由开发者定义，IM SDK 只负责透传。另外如果需要 iOS APNs 推送，还需要提供一段推送文本描述，方便展示。自定义消息由 `TIMCustomElem` 定义，其中 `data` 存储消息的二进制数据，其数据格式由开发者定义。一条消息内可以有多个自定义 `Elem` ，并且可以跟其他 `Elem` 混合排列，离线 `Push` 时叠加每个 `Elem` 的 `desc` 描述信息进行下发。

```
/**
 * 自定义消息类型
 */
@interface TIMCustomElem : TIMElem
/**
 * 自定义消息二进制数据
 */
@property(nonatomic,strong) NSData * data;
/**
 * 自定义消息描述信息，做离线Push时文本展示（已废弃，请使用 TIMMessage 中 offlinePushInfo 进行配置）
 */
@property(nonatomic,strong) NSString * desc DEPRECATED_ATTRIBUTE;
/**
 * 离线Push时扩展字段信息（已废弃，请使用 TIMMessage 中 offlinePushInfo 进行配置）
 */
@property(nonatomic,strong) NSString * ext DEPRECATED_ATTRIBUTE;
/**
 * 离线Push时声音字段信息（已废弃，请使用 TIMMessage 中 offlinePushInfo 进行配置）
 */
@property(nonatomic,strong) NSString * sound DEPRECATED_ATTRIBUTE;
@end
```

参数说明：

参数	说明
data	自定义消息二进制数据

以下示例中拼接一段 XML 消息，具体展示由开发者决定。

示例：

```
// XML 协议的自定义消息
NSString * xml = @"testTitlethis is custom msgtest msg body";
// 转换为 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:@"yahaha"];
[conversation sendMessage:msg succ:^(){
NSLog(@"SendMsg Succ");
}fail:^(int code, NSString * err) {
NSLog(@"SendMsg Failed:%d->%@", code, err);
}];
```

短视频消息发送

短视频消息由 `TIMVideoElem` 定义。它是 `TIMElem` 的一个子类，也就是说视频截图和视频内容也是消息的一种内容。发送短视频的过程，就是将 `TIMVideoElem` 加入到 `TIMMessage` 中，然后随消息一起发送出去。

`TIMVideoElem` 原型：

```
/**
 * 微视频消息
 */
@interface TIMVideoElem : TIMElem
/**
 * 上传时任务Id，可用来查询上传进度（已废弃，请在 TIMUploadProgressListener 监听上传进度）
 */
```

```
@property(nonatomic,assign) uint32_t taskId DEPRECATED_ATTRIBUTE;

/**
* 视频文件路径，发送消息时设置
*/
@property(nonatomic,strong) NSString * videoPath;

/**
* 视频信息，发送消息时设置
*/
@property(nonatomic,strong) TIMVideo * video;

/**
* 截图文件路径，发送消息时设置
*/
@property(nonatomic,strong) NSString * snapshotPath;

/**
* 视频截图，发送消息时设置
*/
@property(nonatomic,strong) TIMSnapshot * snapshot;
@end
```

参数说明：

参数	说明
taskId	上传时任务 ID，可用来查询上传进度（已废弃，请在 TIMUploadProgressListener 监听上传进度）
videoPath	发送短视频时，本地视频文件的路径
video	视频信息，发送消息时设置 type、duration 参数
snapshotPath	发送短视频时，本地截图文件的路径
snapshot	截图信息，发送消息时设置 type、width、height 参数

以下示例中发送了一个短视频消息。**示例：**

```
/**
* 获取聊天会话, 以同用户 iOS-001 的单聊为例
*/
TIMConversation * c2c_conversation = [[TIMManager sharedInstance] getConversation:TIM_C2C receiver:@"iOS-001"];
/**
* 构造一条消息
*/
TIMMessage * msg = [[TIMMessage alloc] init];
/**
* 构短视频内容
*/
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;
/**
* 将短视频内容添加到消息容器中
*/
[msg addElem:videoElem];
/**
* 发送消息
*/
[conversation sendMessage:msg succ:^(){ //成功
```

```
NSLog(@"SendMsg Succ");
}fail:^(int code, NSString * err) { //失败
NSLog(@"SendMsg Failed:%d->%@", code, err);
}];
```

Elem 顺序

目前文件和语音 Elem 不一定会按照添加顺序传输，其他 Elem 按照顺序，不过建议不要过于依赖 Elem 顺序进行处理，应该逐个按照 Elem 类型处理，防止异常情况下进程 Crash。

在线消息

对于某些场景，需要发送在线消息，即用户在线时收到消息，如果用户不在线，下次登录也不会看到消息，可用于通知类消息，这种消息不会进行存储，也不会计入未读计数。发送接口与 `sendMessage` 类似。

> 注意：
>
> - 2.5.3版本以前只针对单聊消息有效。
> - 2.5.3版本以后对群组消息有效（暂不支持 AVChatRoom 和 BChatRoom 类型）

```
@interface TIMConversation : NSObject
/**
* 发送在线消息（服务器不保存消息）
*
* @param msg 消息体
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
*/
-(int) sendOnlineMessage: (TIMMessage*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

消息转发

在2.4.0及以上版本，在 `TIMMessage` 中提供了 `copyFrom` 接口，可以方便地拷贝其他消息的内容到当前消息，然后将消息重新发送给其他人。

原型：

```
/**
* 消息
*/
@interface TIMMessage : NSObject
/**
* 拷贝消息中的属性（ELem、priority、online、offlinePushInfo）
*
* @param srcMsg 源消息
*
* @return 0 成功
*/
- (int)copyFrom:(TIMMessage*)srcMsg;
@end
```

接收消息

用户需要感知新消息的通知时，只需注册新消息通知回调 `TIMMessageListener`，如果用户是登录状态，IM SDK 收到新消息会通过回调中的 `onNewMessage` 抛出。回调消息内容通过参数 `TIMMessage` 传递，通过 `TIMMessage` 可以获取消息和相关会话的详细信息（例如消息文本，语音数据，图片等等），详情请参见 消息解析。

> 注意：
> 通过 `onNewMessage` 抛出的消息不一定是未读的消息，只是本地曾经没有过的消息（例如在另外一个终端已读，拉取最近联系人消息时可以获取会话最后一条消息，如果本地没有，会通过此方法抛出）。在用户登录之后，IM SDK 会拉取离线消息，为了不漏掉消息通知，需要在登录之前注册新消息通知。

群系统消息、关系链变化、好友资料变更也会通过该回调 `onNewMessage` 抛出。

原型：

```
@protocol TIMMessageListener
@optional
/**
* 新消息通知
*
* @param msgs 新消息列表，TIMMessage 类型数组
*/
- (void)onNewMessage:(NSArray*) msgs;
@end

@interface TIMManager : NSObject
- (int)addMessageListener:(id<TIMMessageListener>)listener;
@end
```

参数说明：

参数	说明
msgs	新消息列表，注意这里可能同时会有多条消息抛出，相同会话的消息由老到新排序

以下示例中设置消息回调通知，并且在有新消息时直接打印消息。**示例：**

```
@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];
```

消息解析

收到消息后，可通过 `getElem` 从 `TIMMessage` 中获取所有的 `Elem` 节点。

遍历 `Elem` 原型：

```
@interface TIMMessage : NSObject
-(int) elemCount;
-(TIMElem*) getElem:(int)index;
@end
```

示例：

```
TIMMessage * message = /* 消息 */
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;
}
}
```

接收图片消息

接收方收到消息后，可通过 getElem 从 TIMMessage 中获取所有的 Elem 节点，其中类型为 TIMImageElem 的是图片消息节点。然后通过 imageList 获取该图片的所有规格用来展示。

** TIMImageElem 类原型：**

```
**
* 图片消息 Elem
*/
@interface TIMImageElem : TIMElem
/**
* 要发送的图片路径
*/
@property(nonatomic,retain) NSString * path;
/**
* 保存本图片的所有规格，目前最多包含三种规格: 缩略图、大图、原图， 每种规格保存在一个 TIMImage 对象中
*/
@property(nonatomic,retain) NSArray * imageList;
@end
```

参数说明：

参数	说明
path	收消息时不用关注，为 nil
imageList	保存本图片的所有规格，目前最多包含三种规格：缩略图、大图、原图， 每种规格保存在一个 TIMImage 对象中

TIMImage 说明：

获取到消息时通过 imageList 得到所有的图片规格，为 TIMImage 数据，得到 TIMImage 后可通过图片大小进行占位，通过接口 getImage 下载不同规格的图片进行展示。**下载的数据需要由开发者缓存，IM SDK 每次调用 getImage 都会从服务端重新下载数据。建议通过图片的 uuid 作为 key 进行图片文件的存储。**

原型：

```
@interface TIMImage : NSObject
/**
* 图片 ID，内部标识，可用于外部缓存 key
*/
@property(nonatomic,strong) NSString * uuid;
/**
* 图片类型
*/
@property(nonatomic,assign) TIM_IMAGE_TYPE type;
/**
* 图片大小
*/
@property(nonatomic,assign) int size;
/**
* 图片宽度
*/
@property(nonatomic,assign) int width;
/**
* 图片高度
*/
@property(nonatomic,assign) int height;
/**
* 下载 URL
*/
@property(nonatomic, strong) NSString * url;

/**
* 获取图片
*
* 下载的数据需要由开发者缓存，IM SDK 每次调用 getImage 都会从服务端重新下载数据。建议通过图片的 uuid 作为 key 进行图片文件的存储。
*
* @param path 图片保存路径
* @param succ 成功回调，返回图片数据
* @param fail 失败回调，返回错误码和错误描述
```

```
 */
- (void)getImage:(NSString*)path succ:(TIMSucc)succ fail:(TIMFail)fail;

/**
 * 获取图片（有进度回调）
 *
 * 下载的数据需要由开发者缓存，IM SDK 每次调用 getImage 都会从服务端重新下载数据。建议通过图片的 uuid 作为 key 进行图片文件的存储。
 *
 * @param path 图片保存路径
 * @param progress 图片下载进度
 * @param succ 成功回调，返回图片数据
 * @param fail 失败回调，返回错误码和错误描述
 */
- (void)getImage:(NSString*)path progress:(TIMProgress)progress succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

图片规格说明：每幅图片有三种规格，分别是 Original（原图）、Large（大图）、Thumb（缩略图）。

- **原图：**指用户发送的原始图片，尺寸和大小都保持不变。
- **大图：**是将原图等比压缩，压缩后宽、高中较小的一个等于720像素。
- **缩略图：**是将原图等比压缩，压缩后宽、高中较小的一个等于198像素。

说明：

- 如果原图尺寸就小于198像素，则三种规格都保持原始尺寸，不需压缩。
- 如果原图尺寸在198 - 720之间，则大图和原图一样，不需压缩。
- 在手机上展示图片时，建议优先展示缩略图，用户单击缩略图时再下载大图，单击大图时再下载原图。当然开发者也可以选择跳过大图，单击缩略图时直接下载原图。
- 在 Pad 或 PC 上展示图片时，由于分辨率较大，且基本都是 Wi-Fi 或有线网络，建议直接显示大图，用户单击大图时再下载原图。

以下示例从会话中取出 10 条消息，获取图片消息并下载相应数据。**示例：**

```
//以收到新消息回调为例，介绍下图片消息的解析过程
//接收到的图片保存的路径
NSString * pic_path = @"/xxx/imgPath.jpg";
[conversation getMessage:10 last:nil succ:^(NSArray * msgList) { //获取消息成功
//遍历所有的消息
for (TIMMessage * msg in msgList) {
//遍历一条消息的所有元素
for (int i = 0; i < msg.elemCount; ++i) {
TIMElem *elem = [msg getElem:i];
//图片元素
if ([elem isKindOfClass:[TIMImageElem class]]) {
TIMImageElem * image_elem = (TIMImageElem * )elem;

//遍历所有图片规格(缩略图、大图、原图)
NSArray * imgList = [image_elem imageList];
for (TIMImage * image in imgList) {
[image getImage:pic_path succ:^(){ //接收成功
NSLog(@"SUCC: pic store to %@", pic_path);
}fail:^(int code, NSString * err) { //接收失败
NSLog(@"ERR: code=%d, err=%@", code, err);
}];
}
}
}
}
} fail:^(int code, NSString * err) { //获取消息失败
NSLog(@"Get Message Failed:%d->%@", code, err);
}];
```

接收语音消息

收到消息后，可用过 getElem 从 TIMMessage 中获取所有的 Elem 节点，其中 TIMSoundElem 为语音消息节点。其中 path 为创建消息时填写的语音信息，接收消息时为空。获取到消息时可通过时长占位，通过接口 getSound 下载语音资源， getSound 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。

原型：

```
@interface TIMSoundElem : TIMElem
/**
* 上传时任务 ID，可用来查询上传进度（已废弃，请在 TIMUploadProgressListener 监听上传进度）
*/
@property(nonatomic,assign) uint32_t taskId DEPRECATED_ATTRIBUTE;
/**
* 发送时设置为语音数据，接收时使用 getSound 获得数据
*/
@property(nonatomic,strong) NSString * path;
/**
* 语音消息内部 ID
*/
@property(nonatomic,strong) NSString * uuid;
/**
* 语音数据大小
*/
@property(nonatomic,assign) int dataSize;
/**
* 语音长度（秒），发送消息时设置
*/
@property(nonatomic,assign) int second;

/**
* 获取语音的 URL 下载地址
*
* @param urlCallBack 获取 URL 地址回调
*/
-(void)getUrl:(void (^)(NSString * url))urlCallBack;

/**
* 获取语音数据到指定路径的文件中
*
* getSound 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。
*
* @param path 语音保存路径
* @param succ 成功回调，返回语音数据
* @param fail 失败回调，返回错误码和错误描述
*/
- (void)getSound:(NSString*)path succ:(TIMSucc)succ fail:(TIMFail)fail;

/**
* 获取语音数据到指定路径的文件中（有进度回调）
*
* getSound 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。
*
* @param path 语音保存路径
* @param progress 语音下载进度
* @param succ 成功回调
* @param fail 失败回调，返回错误码和错误描述
*/
- (void)getSound:(NSString*)path progress:(TIMProgress)progress succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

其他参数说明：

参数	说明
path	发送时设置为语音数据，接收时使用 getSound 获得数据
uuid	唯一标识，方便用户缓存
dataSize	语音文件大小

参数	说明
second	语音时长，以秒为单位

语音消息已读状态：语音是否已经播放，可使用 消息自定义字段 实现，如 customInt 的值0表示未播放，1表示播放，当用户单击播放后可设置 customInt 的值为1。

```
@interface TIMMessage : NSObject
/**
* 设置自定义整数，默认为0
*
* @param param 设置参数
*
* @return TRUE 设置成功
*/
- (BOOL) setCustomInt:(int32_t) param;

/**
* 获取 CustomInt
*
* @return CustomInt
*/
- (int32_t)customInt;

@end
```

接收小文件消息

收到消息后，可用过 getElem 从 TIMMessage 中获取所有的 Elem 节点，其中 TIMFileElem 为文件消息节点。其中 path 为创建消息时填写的文件路径，GET 消息时为空。获取到消息时可只展示文件大小和显示名，通过接口 getFile 下载文件资源。 getFile 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。

原型：

```
@interface TIMFileElem : TIMElem
/**
* 上传时任务Id，可用来查询上传进度（已废弃，请在 TIMUploadProgressListener 监听上传进度）
*/
@property(nonatomic,assign) uint32_t taskId DEPRECATED_ATTRIBUTE;
/**
* 上传时，文件的路径（设置 path 时，优先上传文件）
*/
@property(nonatomic,strong) NSString * path;
/**
* 文件内部ID
*/
@property(nonatomic,strong) NSString * uuid;
/**
* 文件大小
*/
@property(nonatomic,assign) int fileSize;
/**
* 文件显示名，发消息时设置
*/
@property(nonatomic,strong) NSString * filename;

/**
* 获取文件的 URL 下载地址
*
* @param urlCallBack 获取 URL 地址回调
*/
-(void)getUrl:(void (^)(NSString * url))urlCallBack;

/**
* 获取文件数据到指定路径的文件中
*
* getFile 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。
```

```
 *
 * @param path 文件保存路径
 * @param succ 成功回调，返回数据
 * @param fail 失败回调，返回错误码和错误描述
 */
- (void)getFile:(NSString*)path succ:(TIMSucc)succ fail:(TIMFail)fail;

/**
 * 获取文件数据到指定路径的文件中（有进度回调）
 *
 * getFile 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。
 *
 * @param path 文件保存路径
 * @param progress 文件下载进度
 * @param succ 成功回调，返回数据
 * @param fail 失败回调，返回错误码和错误描述
 */
- (void)getFile:(NSString*)path progress:(TIMProgress)progress succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
path	上传时，文件的路径
uuid	唯一 ID，方便用户进行缓存
fileSize	文件大小
filename	文件显示名

接收短视频消息

收到消息后，可通过 getElem 从 TIMMessage 中获取所有的 Elem 节点，其中 TIMVideoElem 为文件消息节点，通过 TIMVideo 和 TIMSnapshot 对象获取视频和截图内容。接收到 TIMVideoElem 后，通过 video 属性和 snapshot 属性中定义的接口下载视频文件和截图文件。如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。

原型：

```
@interface TIMVideo : NSObject
/**
 * 视频消息内部 ID，不用设置
 */
@property(nonatomic,strong) NSString * uuid;
/**
 * 视频文件类型，发送消息时设置
 */
@property(nonatomic,strong) NSString * type;
/**
 * 视频大小，不用设置
 */
@property(nonatomic,assign) int size;
/**
 * 视频时长，发送消息时设置
 */
@property(nonatomic,assign) int duration;

/**
 * 获取视频的 URL 下载地址
 *
 * @param urlCallBack 获取 URL 地址回调
 */
-(void)getUrl:(void (^)(NSString * url))urlCallBack;

/**
 * 获取视频
 *
 * getVideo 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。
```

```
 *
 * @param path 视频保存路径
 * @param succ 成功回调
 * @param fail 失败回调，返回错误码和错误描述
 */
- (void)getVideo:(NSString*)path succ:(TIMSucc)succ fail:(TIMFail)fail;

/**
 * 获取视频（有进度回调）
 *
 * getVideo 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。
 *
 * @param path 视频保存路径
 * @param progress 视频下载进度
 * @param succ 成功回调
 * @param fail 失败回调，返回错误码和错误描述
 */
- (void)getVideo:(NSString*)path progress:(TIMProgress)progress succ:(TIMSucc)succ fail:(TIMFail)fail;

@end

@interface TIMSnapshot : NSObject
/**
 * 图片ID，不用设置
 */
@property(nonatomic,strong) NSString * uuid;
/**
 * 截图文件类型，发送消息时设置
 */
@property(nonatomic,strong) NSString * type;
/**
 * 图片大小，不用设置
 */
@property(nonatomic,assign) int size;
/**
 * 图片宽度，发送消息时设置
 */
@property(nonatomic,assign) int width;
/**
 * 图片高度，发送消息时设置
 */
@property(nonatomic,assign) int height;

/**
 * 获取截图的 URL 下载地址
 *
 * @param urlCallBack 获取 URL 地址回调
 */
-(void)getUrl:(void (^)(NSString * url))urlCallBack;

/**
 * 获取图片
 *
 * getImage 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。
 *
 * @param path 图片保存路径
 * @param succ 成功回调，返回图片数据
 * @param fail 失败回调，返回错误码和错误描述
 */
- (void)getImage:(NSString*)path succ:(TIMSucc)succ fail:(TIMFail)fail;

/**
 * 获取图片（有进度回调）
 *
 * getImage 接口每次都会从服务端下载，如需缓存或者存储，开发者可根据 uuid 作为 key 进行外部存储，IM SDK 并不会存储资源文件。
 *
 * @param path 图片保存路径
 * @param progress 图片下载进度
```

```
 * @param succ 成功回调，返回图片数据
 * @param fail 失败回调，返回错误码和错误描述
 */
- (void)getImage:(NSString*)path progress:(TIMProgress)progress succ:(TIMSucc)succ fail:(TIMFail)fail;
@end

//以收到新消息回调为例，介绍下短视频消息的解析过程
//接收到的视频和截图保存的路径
NSString * video_path = @"/xxx/video.mp4";
NSString * snapshot_path = @"/xxx/snapshot.jpg";
[conversation getMessage:10 last:nil succ:^(NSArray * msgList) { //获取消息成功
//遍历所有的消息
for (TIMMessage * msg in msgList) {
//遍历一条消息的所有元素
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(@"下载视频文件成功");
 }  fail:^(int code, NSString * err) {
NSLog(@"下载视频文件失败:%@ %d", err, code);
}];
[video_elem.snapshot getImage:snapshot_path succ:^() {
NSLog(@"下载截图成功");
} fail:^(int code, NSString * err) {
NSLog(@"下载截图失败:%@ %d", err, code);
}];
}
}
} fail:^(int code, NSString * err) { //获取消息失败
NSLog(@"Get Message Failed:%d->%@", code, err);
}];
```

消息属性

消息是否已读

通过消息属性 `isReaded` 是否消息已读。这里已读与否取决于 App 侧进行的 已读上报。

```
@interface TIMMessage : NSObject
/**
* 是否已读
*
* @return TRUE 已读 FALSE 未读
*/
-(BOOL) isReaded;
@end
```

消息状态

通过消息属性 `status` 可以获取消息的当前状态(如，发送中、发送成功、发送失败和删除)，对于删除的消息，需要 UI 判断状态并隐藏。

```
/**
* 消息状态
*/
typedef NS_ENUM(NSInteger, TIMMessageStatus){
/**
* 消息发送中
*/
TIM_MSG_STATUS_SENDING = 1,
/**
* 消息发送成功
*/
TIM_MSG_STATUS_SEND_SUCC = 2,
```

```
/**
* 消息发送失败
*/
TIM_MSG_STATUS_SEND_FAIL = 3,
/**
* 消息被删除
*/
TIM_MSG_STATUS_HAS_DELETED = 4,
/**
* 导入到本地的消息
*/
TIM_MSG_STATUS_LOCAL_STORED = 5,
/**
* 被撤销的消息
*/
TIM_MSG_STATUS_LOCAL_REVOKED = 6,
};

@interface TIMMessage : NSObject
/**
* 消息状态
*
* @return TIMMessageStatus 消息状态
*/
-(TIMMessageStatus) status;
@end
```

是否是自己发出的消息

通过消息属性 `isSelf` 可以判断消息是否是自己发出的消息，界面显示时可用。

```
@interface TIMMessage : NSObject
/**
* 是否发送方
*
* @return TRUE 表示是发送消息 FALSE 表示是接收消息
*/
-(BOOL) isSelf;
@end
```

消息发送者以及相关资料

对于群消息，可以通过 `TIMMessage` 的方法 `sender` 得到发送用户，另外也可以通过方法 `GetSenderProfile` 和 `GetSenderGroupMemberProfile` 获取用户自己的资料和所在群的资料。1.9版本之前，只有在线消息 `onNewMessage` 抛出的消息可以获取到用户资料，1.9版本以后，通过 `getMessage` 得到的消息也可以拿到资料（更新版本之前已经收到本地的消息无法获取到）。对于单聊消息，通过通过 `TIMMessage` 的 `getConversation` 获取到对应会话，会话的 `getReceiver` 可以得到正在聊天的用户。

> 注意：
> 此字段是消息发送时获取用户资料写入消息体，如后续用户资料更新，此字段不会相应变更，只有产生的新消息中才会带最新的昵称。

```
@interface TIMMessage : NSObject
/**
* 获取发送方
*
* @return 发送方标识
*/
-(NSString *) sender;
/**
* 获取发送者资料
*
* 如果本地有发送者资料，会在 profileCallBack 回调里面立即同步返回发送者资料，如果本地没有发送者资料，SDK 内部会先向服务器拉取发送者资料，并在 profile CallBack 回调里面异步返回发送者资料。
*
```

```
 * @param profileCallBack 发送者资料回调
 *
 */
- (void)getSenderProfile:(ProfileCallBack)profileCallBack;
/**
 * 获取发送者群内资料（发送者为本人时可能为空）
 *
 * @return 发送者群内资料，nil 表示没有获取资料或者不是群消息，目前仅能获取字段：member ，其他的字段获取建议通过 TIMGroupManager+Ext.h -> getGroupMembers 获取
 */
-(TIMGroupMemberInfo *) GetSenderGroupMemberProfile;
@end
```

消息时间

通过消息属性 `timestamp` 可以得到消息时间，该时间是 Server 时间，而非本地时间。在创建消息时，此时间为根据 Server 时间校准过的时间，发送成功后会改为准确的 Server 时间。

```
@interface TIMMessage : NSObject
/**
 * 当前消息的时间戳
 *
 * @return 时间戳
 */
-(NSDate*) timestamp;
@end
```

消息 ID

消息 ID 分为两种，一种是当消息生成时，就已经固定（ `msgId` ），这种方式可能跟其他用户产生的消息冲突，需要再加一个时间维度，可以认为10分钟以内的消息可以使用 `msgId` 区分。另外一种，当消息发送成功以后才能固定下来（ `uniqueId` ），这种方式能保证全局唯一。这两种方式都需要在同一个会话内判断。

```
@interface TIMMessage : NSObject
/**
 * 消息 ID
 */
-(NSString *) msgId;
/**
 * 获取消息uniqueId
 *
 * @return uniqueId
 */
- (uint64_t) uniqueId;
@end
```

消息自定义字段

开发者可以对消息增加自定义字段，如自定义整数、自定义二进制数据，可以根据这两个字段做出各种不通效果，例如语音消息是否已经播放等等。另外需要注意，此自定义字段仅存储于本地，不会同步到 Server，更换终端获取不到。

```
@interface TIMMessage : NSObject
/**
* 设置自定义整数，默认为 0
*
* @param param 设置参数
*
* @return TRUE 设置成功
*/
- (BOOL) setCustomInt:(int32_t) param;
/**
* 设置自定义数据，默认为""
*
* @param data 设置参数
*
* @return TRUE 设置成功
*/
```

```
- (BOOL) setCustomData:(NSData*) data;
/**
* 获取CustomInt
*
* @return CustomInt
*/
- (int32_t) customInt;
/**
* 获取CustomData
*
* @return CustomData
*/
- (NSData*) customData;
@end
```

消息优先级

对于直播场景，会有点赞和发红包功能，点赞相对优先级较低，红包消息优先级较高，具体消息内容可以使用 `TIMCustomElem` 进行定义，发送消息时，可设置消息优先级。

> 注意：
> 只针对群聊消息有效。

```
@interface TIMMessage : NSObject
/**
* 设置消息的优先级
*
* @param priority 优先级
*
* @return TRUE 设置成功
*/
- (BOOL) setPriority:(TIMMessagePriority)priority;
/**
* 获取消息的优先级
*
* @return 优先级
*/
- (TIMMessagePriority) getPriority;
@end
```

已读回执

对于单聊消息，用户开启已读回执功能后，对方调用 `setReadMessage` 时会同步已读信息到本客户端。

开启已读回执功能：

```
@interface TIMUserConfig : NSObject
/**
* 启用已读回执，启用后在已读上报时会给对方发送回执，只对单聊回话有效
*/
-(void) enableReadReceipt;
/**
* 消息已读回执监听器
*/
@property(nonatomic,weak) id<TIMMessageReceiptListener> messageReceiptListener;
@end
```

原型：

```
@interface TIMMessage : NSObject

/**
* 对方是否已读（仅 C2C 消息有效）
*
```

```
 * @return TRUE 已读 FALSE 未读
 */
-(BOOL) isPeerReaded;
@end
```

会话操作

获取所有会话

```
@interface TIMManager : NSObject

/**
 * 获取会话（TIMConversation*）列表
 *
 * @return 会话列表
 */
-(NSArray*) getConversationList;
@end
```

注意：
SDK 会在内部不断更新会话列表，每次更新后都会通过 TIMRefreshListener.onRefresh 回调，请**在 onRefresh 之后再调用 getConversationList** 更新会话列表。

示例：

```
NSArray * conversations = [[TIMManager sharedInstance] getConversationList];
NSLog(@"current session list : %@", [conversations description])
```

获取会话本地消息

IM SDK 会在本地进行消息存储，可通过 TIMConversation 方法的 getLocalMessage 获取，此方法为异步方法，需要通过设置回调得到消息数据，对于单聊，登录后可以获取离线消息，对于群聊，开启最近联系人漫游的情况下，登录后只能获取最近一条消息，可通过 getMessage 获取漫游消息。对于图片、语音等资源类消息，消息体只会包含描述信息，需要通过额外的接口下载数据，可参阅 消息解析，下载后的真实数据不会缓存，需要调用方进行缓存。

原型：

```
@interface TIMConversation : NSObject
/**
 * 获取本地会话消息
 *
 * @param count 获取数量
 * @param last 上次最后一条消息
 * @param succ 成功时回调
 * @param fail 失败时回调
 *
 * @return 0 本次操作成功
 */
-(int) getLocalMessage: (int)count last:(TIMMessage*)last succ:(TIMGetMsgSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
count	指定获取消息的数量
last	指定上次获取的最后一条消息，如果 last 传 nil，从最新的消息开始读取
succ	成功回调
fail	失败回调

示例：

```
[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);
}];
```

获取会话漫游消息

对于群组，登录后可以获取漫游消息，对于 C2C，开通漫游服务后可以获取漫游消息，通过 IM SDK 的 `getMessage` 接口可以获取漫游消息，如果本地消息全部都是连续的，则不会通过网络获取，如果本地消息不连续，会通过网络获取断层消息。对于图片、语音等资源类消息，消息体只会包含描述信息，需要通过额外的接口下载数据，可参与消息解析部分，下载后的真实数据不会缓存，需要调用方进行缓存。

原型：

```
@interface TIMConversation : NSObject
/**
* 获取会话消息
*
* @param count 获取数量
* @param last 上次最后一条消息
* @param succ 成功时回调
* @param fail 失败时回调
*
* @return 0 本次操作成功
*/
-(int) getMessage: (int)count last:(TIMMessage*)last succ:(TIMGetMsgSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
count	指定获取消息的数量
last	指定上次获取的最后一条消息，如果 last 传 nil，从最新的消息开始读取
succ	成功回调
fail	失败回调

示例：

```
[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);
}];
```

删除会话

删除会话的同时 IM SDK 会删除该会话的本地和漫游消息，会话和消息删除后，无法再恢复。

原型：

```
@protocol TIMManager : NSObject
/**
*
```

```
* 删除会话的同时会把会话的漫游消息从本地和后台都删除。
*
* @param type 会话类型，详情请参考 TIMComm.h 里面的 TIMConversationType 定义
* @param conversationId 会话 Id
* 单聊类型（C2C）：为对方 userID；
* 群组类型（GROUP）：为群组 groupId；
* 系统类型（SYSTEM）：为 @""
*
* @return YES:删除成功；NO:删除失败
*/
- (BOOL)deleteConversation:(TIMConversationType)type receiver:(NSString*)conversationId;
@end
```

参数说明：

参数	说明
type	会话类型，如果是单聊，填写 TIM_C2C，如果是群聊，填写 TIM_GROUP
conversationId	会话标识，单聊情况下，receiver 为对方用户 identifier，群聊情况下，receiver 为群组 ID

示例中删除好友『iOS_002』的 C2C 会话。**示例：**

```
[[TIMManager sharedInstance] deleteConversation:TIM_C2C receiver:@"iOS_002"];
```

同步获取会话最后的消息

UI 展示最近联系人列表时，时常会展示用户的最后一条消息，在1.9以后版本增加了同步获取接口 `getLastMsg` ，用户可以通过此接口方便获取最后一条消息进行展示。**目前没有网络无法获取。**此接口获取并不会过滤删除状态消息，需要 App 层进行屏蔽。获取最近的多条消息，可以通过 `getMessage` 来获取。

原型：

```
@interface TIMConversation : NSObject
/**
* 从 Cache 中获取最后一条消息
* @return 最后一条消息（TIMMessage*）
*/
- (TIMMessage*)getLastMsg;

/**
* 获取会话漫游消息
* @param count 获取数量
* @param last 上次最后一条消息，如果 last 为 nil，从最新的消息开始读取
* @param succ 成功时回调
* @param fail 失败时回调
* @return 0：本次操作成功；1：本次操作失败
*/
- (int)getMessage:(int)count last:(TIMMessage*)last succ:(TIMGetMsgSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
count	需要获取的消息数，注意这里最多为 20

设置会话草稿

UI 展示最近联系人列表时，时常会展示用户的草稿内容，在2.2以后版本增加了设置和获取草稿的接口，用户可以通过此接口设置会话的草稿信息。**草稿信息会存本地数据库，重新登录后依然可以获取。**

原型：

```
@interface TIMMessageDraft : NSObject
/**
* 设置自定义数据
*
```

```
 * @param userData 自定义数据
 *
 * @return 0 成功
 */
-(int) setUserData:(NSData*)userData;
/**
 * 获取自定义数据
 *
 * @return 自定义数据
 */
-(NSData*) getUserData;
/**
 * 增加 Elem
 *
 * @param elem elem 结构
 *
 * @return 0 表示成功
 * 1 禁止添加 Elem（文件或语音多于两个 Elem）
 * 2 未知 Elem
 */
-(int) addElem:(TIMElem*)elem;
/**
 * 获取对应索引的 Elem
 *
 * @param index 对应索引
 *
 * @return 返回对应 Elem
 */
-(TIMElem*) getElem:(int)index;
/**
 * 获取 Elem 数量
 *
 * @return elem 数量
 */
-(int) elemCount;
/**
 * 草稿生成对应的消息
 *
 * @return 消息
 */
-(TIMMessage*) transformToMessage;
/**
 * 当前消息的时间戳
 *
 * @return 时间戳
 */
-(NSDate*) timestamp;
@end

@interface TIMConversation : NSObject
/**
 * 设置会话草稿
 *
 * @param draft 草稿内容
 *
 * @return 0 成功
 */
-(int) setDraft:(TIMMessageDraft*)draft;
/**
 * 获取会话草稿
 *
 * @return 草稿内容，没有草稿返回nil
 */
-(TIMMessageDraft*) getDraft;
@end
```

参数说明：

参数	说明
draft	需要设置的草稿 ，需要清空会话草稿时传入 nil

删除会话消息

IM SDK 支持删除会话的本地及漫游消息，消息删除后，无法再恢复。

原型：

```
@interface TIMConversation : NSObject
/**
* 删除当前会话的本地及漫游消息
*
* @param succ 成功时回调
* @param fail 失败时回调
*
* @return 0 本次操作成功
*/
- (int)deleteMessages:(NSArray<TIMMessage *>*)msgList succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
msgList	需要删除的消息列表
succ	成功回调
fail	失败回调

获取本地指定 ID 的消息

IM SDK 2.5.3 版本提供获取本地指定 ID 消息的接口。

原型：

```
/**
* 消息
*/
@interface TIMMessage : NSObject
/**
* 获取消息定位符
*
* @return locator
*/
- (TIMMessageLocator*) locator;
@end
@interface TIMConversation : NSObject
/**
* 获取会话消息
*
* @param locators 消息定位符 ( TIMMessageLocator ) 数组
* @param succ 成功时回调
* @param fail 失败时回调
*
* @return 0 本次操作成功
*/
-(int) findMessages:(NSArray*)locators succ:(TIMGetMsgSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
locators	消息定位符 TIMMessageLocator 列表

参数	说明
succ	成功回调，返回消息列表
fail	失败回调

撤回消息

IM SDK 在 3.1.0 版本开始提供撤回消息的接口。可以通过调用 `TIMConversation` 的 `revokeMessage` 接口来撤回自己发送的消息。

注意：

- 仅 C2C 和 GROUP 会话有效、onlineMessage 无效、AVChatRoom 和 BChatRoom 无效。
- 默认只能撤回 2 分钟内的消息。

原型：

```
/**
* 消息撤回（仅 C2C 和 GROUP 会话有效，其中 onlineMessage、AVChatRoom 和 BChatRoom 无效）
* @param msg 被撤回的消息
* @param succ 成功时回调
* @param fail 失败时回调
*
* @return 0：本次操作成功；1：本次操作失败
*/
- (int)revokeMessage:(TIMMessage*)msg succ:(TIMSucc)succ fail:(TIMFail)fail;
```

成功撤回消息后，群组内其他用户和 C2C 会话对端用户会收到一条消息撤回通知，并通过消息撤回通知监听器 `TIMMessageRevokeListener` 通知到上层应用。消息撤回通知监听器可以在登录前，通过 `TIMUserConfig` 的 `messageRevokeListener` 来进行配置。具体可以参考 用户配置。

原型：

```
@protocol TIMMessageRevokeListener <NSObject>
@optional
/**
* 消息撤回通知
*
* @param locator 被撤回消息的标识
*/
- (void)onRevokeMessage:(TIMMessageLocator*)locator;

@end
```

收到一条消息撤回通知后，通过 `TIMMessage` 中的 `respondsToLocator` 方法判断当前消息是否是被对方撤回了，然后根据需要对 UI 进行刷新。

原型：

```
/**
* 是否为 locator 对应的消息
*
* @param locator 消息定位符
*
* @return YES：是对应的消息；NO：不是对应的消息
*/
- (BOOL)respondsToLocator:(TIMMessageLocator*)locator;
```

系统消息

会话类型（TIMConversationType）除了 C2C 单聊和 Group 群聊以外，还有一种系统消息，系统消息不能由用户主动发送，是系统后台在相应的事件发生时产生的通知消息。系统消息目前分为两种，一种是关系链系统消息，一种是群系统消息。

- **关系链变更系统消息：**当有用户加自己为好友，或者有用户删除自己好友的情况下，系统会发出变更通知，开发者可更新好友列表。相关细节可参阅 关系链变更系统通知 部分。
- **群事件消息：**当群资料变更，如群名变更或者群内成员变更，在群里会有系统发出一条群事件消息，开发者可在收到消息时可选择是否展示给用户，同时可刷新群资料或者群成员。详细内容可参阅：群组管理-群事件消息。
- **群系统消息：**当被管理员踢出群组，被邀请加入群组等事件发生时，系统会给用户发出群系统消息，相关细节可参阅 群组管理-群系统消息。

未读消息计数
未读消息计数（Android）

最近更新时间：2020-06-29 15:51:37

未读消息

这里的未读消息是指用户没有进行已读上报的消息（而非对方是否已经阅读，这种情况需要开启已读回执才能实现，请参考 已读回执）。`TIMMessage` 方法 `isRead` 标识消息是否已读，要想显示正确的未读计数，需要开发者显式调用已读上报，告诉 IM SDK 某条消息是否已读，例如，当用户进入聊天界面，可以设置整个会话的消息已读。

> 注意：
> 对于聊天室，Server 不保存未读计数，每次登录后跟 Server 同步未读计数后将会清零。

原型：

```
/**
 * 消息是否已读
 * @return 是否已读
 */
public boolean isRead()
```

获取当前未读消息数量

可通过 `TIMConversation` 的 `getUnReadMessageNum` 方法获取当前会话中未读消息的数量。

> 注意：
> 对于聊天室，Server 不保存未读计数，每次登录后跟 Server 同步未读计数后将会清零。

原型：

```
/**
 * 获取未读消息数
 * @return 未读消息计数
 */
public long getUnreadMessageNum()
```

示例：

```
//获取会话扩展实例
TIMConversation con = TIMManager.getInstance().getConversation(TIMConversationType.C2C, peer);
//获取会话未读数
long num = con.getUnreadMessageNum();
Log.d(tag, "unread msg num: " + num);
```

已读上报

当用户阅读某个会话的数据后，需要进行会话消息的已读上报，IM SDK 根据会话中最后一条阅读的消息，设置会话中之前所有消息为已读。已读上报接口为 `TIMConversation` 中的 `setReadMessage`。

原型：

```
/**
 * 将此消息之前的所有消息标记为已读
 * @param msg 最后一条已读的消息, 传 null 表示将当前会话的所有消息标记为已读
```

```
 * @param callback 回调
 */
public void setReadMessage(TIMMessage msg, TIMCallBack callback)
```

示例：

```
//对单聊会话已读上报
String peer = "sample_user_1"; //获取与用户 "sample_user_1" 的会话
TIMConversation conversation = TIMManager.getInstance().getConversation(
TIMConversationType.C2C, //会话类型：单聊
peer); //会话对方用户帐号
//将此会话的所有消息标记为已读
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");
}
});

//对群聊会话已读上报
String groupId = "TGID1EDABEAEO"; //获取与群组 "TGID1LTTZEAEO" 的会话
conversation = TIMManager.getInstance().getConversation(
TIMConversationType.Group, //会话类型：群组
groupId); //群组 Id
//将此会话的 lastMsg 代表的消息及这个消息之前的所有消息标记为已读
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");
}
});
```

说明：
单聊和群聊设置已读用法相同，区别在于会话类型。

多终端已读上报同步

在多终端情况下，未读消息计数由 Server 下发同步通知，IM SDK 在本地更新未读计数后，通知用户更新会话。通知会通过 `TIMRefreshListener` 中的 `onRefreshConversation` 接口来进行回调，对于关注多终端同步的用户，可以在这个接口中进行相关的同步处理。请参考 会话刷新 。

原型：

```
/**
 * 部分会话刷新（包括多终端已读上报同步）
 * @param conversations 需要刷新的会话列表
 */
public void onRefreshConversation(List<TIMConversation> conversations);
```

禁用自动上报

出于性能考虑，未读消息由 IM SDK 拉回到本地，Server 默认会删除未读消息，切换终端以后无法看到之前终端已经拉回的未读消息，即使没有主动进行已读上报。如果仅在一个终端，未读计数没有问题，如果需要多终端情况下仍然会有未读，可以通过 `TIMUserConfig` 中的 `disableAutoReport` 方法禁用自动上报，禁用后 IM 通讯云不会代替用户已读上报。

注意：

- 一旦禁用自动上报，需要开发者显式调用 `setReadMessage` 进行已读上报。
- 需要在登录前设置，登录后设置无效。

原型：

```
/**
* 设置是否开启自动已读上报功能（默认开启），登录前设置
* @param disableAutoReport true - 关闭， false - 开启
*/
public TIMUserConfig disableAutoReport(boolean disableAutoReport)
```

未读消息计数（iOS）

最近更新时间：2020-05-08 17:00:56

未读消息

未读消息是指用户没有读过的消息（而非对方是否已经阅读），`TIMMessage` 方法 `isReaded` 标识消息是否已读，要想显示正确的未读计数，需要开发者显式调用已读上报，告诉 App 某条消息是否已读。例如，当用户进入聊天界面，可以设置整个会话的消息已读。对于聊天室，Server 不保存未读计数，每次登录后跟 Server 同步未读计数后将会清零。

```
@interface TIMMessage : NSObject
/**
* 是否已读
*
* @return TRUE 已读 FALSE 未读
*/
- (BOOL)isReaded;
@end
```

获取当前未读消息数量

可通过 `TIMConversation` 的 `getUnReadMessageNum` 方法获取当前会话中未读消息的数量。对于聊天室，Server 不保存未读计数，每次登录后跟 Server 同步未读计数后将会清零。

原型：

```
@interface TIMConversation : NSObject
- (int)getUnReadMessageNum;
@end
```

示例：

```
TIMConversation * conversation = [[TIMManager sharedInstance] getConversation:TIM_C2C receiver:@"iOS_002"];
[conversation getUnReadMessageNum];
```

已读上报

当用户阅读某个会话的数据后，需要进行会话消息的已读上报，IM SDK 根据会话中最后一条阅读的消息，设置会话中之前所有消息为已读。

原型：

```
@interface TIMConversation : NSObject
/**
* 设置已读消息
*
* @param readed 会话内最近一条已读的消息，nil 表示上报最新消息
*
* @param succ 成功时回调
* @param fail 失败时回调
*
* @return 0 表示成功
*/
- (int)setReadMessage:(TIMMessage*)readed succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明：

参数	说明
readed	为当前会话中最后一条读过的消息，IM SDK 会把比 readed 时间更早的消息标记为已读消息

参数	说明
succ	成功回调
fail	失败回调

以下示例设置 C2C（单聊）会话内的所有消息为已读。
示例：

```
TIMConversation * conversation = [[TIMManager sharedInstance] getConversation:TIM_C2C receiver:@"iOS_002"];
[conversation setReadMessage:nil succ:nil fail:nil];
```

禁用自动上报

在单终端情况下，默认设置可以满足需求，出于性能考虑，未读消息由 IM SDK 拉回到本地，Server 默认会删除未读消息，切换终端以后无法看到之前终端已经拉回的未读消息，如果仅在一个终端，未读计数没有问题。如果需要多终端情况下仍然会有未读，可以在 **TIMManager 初始化之前**禁用自动上报，即时通信 IM 不会代替用户已读上报，**一旦禁用自动上报，需要开发者显式调用 `setReadMessage`。**

```
@interface TIMUserConfig: NSObject
/**
* 禁止自动上报（加载消息扩展包有效）
*/
@property(nonatomic,assign) BOOL disableAutoReport;
@end
```

多终端已读同步

在2.0以上版本中引入的功能，在多终端情况下，未读消息计数由 Server 下发同步通知，IM SDK 在本地更新未读计数后，通知用户更新会话。此功能需要在 **TIMManager 登录之前**设置。

原型：

```
@protocol TIMRefreshListener <NSObject>
@optional
/**
* 刷新部分会话（包括多终端已读上报同步）
*
* @param conversations 会话（TIMConversation*）列表
*/
- (void)onRefreshConversations:(NSArray*)conversations;
@end
@interface TIMUserConfig : NSObject
/**
* 会话刷新监听器（未读计数、已读同步）（加载消息扩展包有效）
*/
@property(nonatomic,retain) id<TIMRefreshListener> refreshListener;
@end
```

好友与用户资料
用户资料与关系链（Android）

最近更新时间：2020-01-06 11:24:26

即时通信 IM 提供了**用户资料托管**，App 开发者使用简单的接口就可实现用户资料存储功能。另外，为了方便不同用户定制化资料，也提供用户资料的自定义字段。

用户资料

获取自己的资料

- 可通过 `TIMFriendshipManager` 的 `getSelfProfile` 方法获取服务器保存的用户自己的资料。
- 可通过 `TIMFriendshipManager` 的 `querySelfProfile` 方法获取本地保存的用户自己的资料。

原型：

```
/**
 * 获取服务器保存的自己的资料
 * @param cb 回调，OnSuccess 函数的参数中返回包含相应自己的{@see TIMUserProfile}
 */
public void getSelfProfile(final @NonNull TIMValueCallBack<TIMUserProfile> cb)

/**
 * 获取本地保存的自己的资料，没有则返回 null
 *
 * @return TIMUserProfile
 */
public TIMUserProfile querySelfProfile()
```

`TIMUserProfile` 提供的接口如下：

```
/**
 * 获取用户的 identifier
 * @return 用户的 identifier
 */
public String getIdentifier()

/**
 * 获取用户的昵称
 * @return 用户的昵称
 */
public String getNickName()

/**
 * 获取用户头像 URL
 * @return 用户头像 URL
 */
public String getFaceUrl()

/**
 * 获取用户个人签名
 * @return 用户个人签名
 */
public String getSelfSignature()

/**
 * 获取用户加好友的选项
 * @return 用户好友选项，见 TIMFriendAllowType 中常量
 */
public String getAllowType()

/**
 * 获取用户自定义信息
 * @return 自定义信息 Map
```

```
 */
public Map<String, byte[]> getCustomInfo()

/**
 * 获取用户自定义信息
 * @return 自定义信息 Map
 */
public Map<String, Long> getCustomInfoUint()

/**
 * 获取用户性别类型， 见TIMFriendGenderType中的常量定义
 * @return 用户性别类型
 */
public int getGender()

/**
 * 获取用户生日信息
 * @return 生日信息
 */
public long getBirthday()

/**
 * 获取语言
 * @return 语言
 */
public long getLanguage()

/**
 * 获取位置信息
 * @return 位置信息
 */
public String getLocation()
```

示例：

```
//获取服务器保存的自己的资料
TIMFriendshipManager.getInstance().getSelfProfile(new TIMValueCallBack<TIMUserProfile>(){
@Override
public void onError(int code, String desc){
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 列表请参见错误码表
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());
}
});

//获取本地保存的自己的资料
TIMUserProfile selfProfile = TIMFriendshipManager.getInstance().querySelfProfile();
```

获取指定用户的资料

可通过 `TIMFriendshipManager` 的 `getUsersProfile` 方法获取好友的资料。该方法支持从缓存和后台两种方式获取：

- 当 `forceUpdate = true` 时，会强制从后台拉取数据，并把返回的数据缓存下来。
- 当 `forceUpdate = false` 时，则先在本地查找，如果本地没有数据则再向后台请求数据。
  建议只在显示资料时强制拉取，以减少等待时间。

可通过 `TIMFriendshipManager` 的 `queryUserProfile` 方法通过返回值获取本地缓存的好友资料，没有则返回 `null` 。

原型：

```
/**
 * 获取指定好友资料（不包括：备注，好友分组）
 * @param users 要获取资料的用户 identifier 列表
 * @param forceUpdate 强制从后台拉取
 * @param cb 回调，OnSuccess 函数的参数中返回包含相应用户的{@see TIMUserProfile}列表
 */
public void getUsersProfile(@NonNull List<String> users, boolean forceUpdate, @NonNull TIMValueCallBack<List<TIMUserProfile>> cb)

/**
 * 获取本地好友资料（不包括：备注，好友分组），没有则返回 null
 * @param identifier
 * @return TIMUserProfile
 */
public TIMUserProfile queryUserProfile(String identifier)
```

示例：

```
//待获取用户资料的用户列表
List<String> users = new ArrayList<String>();
users.add("sample_user_1");
users.add("sample_user_2");

//获取用户资料
TIMFriendshipManager.getInstance().getUsersProfile(users, true, new TIMValueCallBack<List<TIMUserProfile>>(){
@Override
public void onError(int code, String desc){
//错误码 code 和错误描述 desc，可用于定位请求失败原因
//错误码 code 列表请参见错误码表
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());
}
}
});

//获取本地缓存的用户资料
TIMUserProfile userProfile = TIMFriendshipManager.getInstance().queryUserProfile("sample_user_1");
```

`getUsersProfile` 接口缓存的时间可通过 `TIMFriendProfileOption` 的 `setExpiredSeconds` 接口设置，默认缓存时间一天。

```
TIMUserConfig config = new TIMUserConfig();
TIMFriendProfileOption timFriendProfileOption = new TIMFriendProfileOption();
timFriendProfileOption.setExpiredSeconds(60 * 60); // 1小时
config.setTIMFriendProfileOption(timFriendProfileOption);
TIMManager.getInstance().setUserConfig(config);
```

修改自己的资料

通过 `TIMFriendshipManager` 的 `modifySelfProfile` 方法可以对自己的资料（如昵称、头像、添加好友选项等）进行修改。

原型：

```
/**
 * 修改自己的资料信息
 * @param profileMap 需要修改的字段放在hashMap中, key值取TIMFriendshipManager中定义的常量:
 * TIMFriendshipManager.TIM_PROFILE_TYPE_KEY_XXX
 * @param cb 回调
 */
public void modifySelfProfile(@NonNull HashMap<String, Object> profileMap, @NonNull TIMCallBack cb)
```

通过 `profileMap` 可以一次设置多个字段，例如同时设置昵称和性别的代码如下：

```
HashMap<String, Object> profileMap = new HashMap<>();
profileMap.put(TIMUserProfile.TIM_PROFILE_TYPE_KEY_NICK, "我的昵称");
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");
}
});
```

设置不存在的键值可能会导致失败，在 `TIMUserProfile` 中定义了一些常用的键值：

Key	Value	说明
`TIM_PROFILE_TYPE_KEY_NICK`	String	昵称
`TIM_PROFILE_TYPE_KEY_FACEURL`	String	头像
`TIM_PROFILE_TYPE_KEY_ALLOWTYPE`	String	好友申请
`TIM_PROFILE_TYPE_KEY_GENDER`	int	性别
`TIM_PROFILE_TYPE_KEY_BIRTHDAY`	int	生日
`TIM_PROFILE_TYPE_KEY_LOCATION`	String	位置
`TIM_PROFILE_TYPE_KEY_LANGUAGE`	int	语言
`TIM_PROFILE_TYPE_KEY_LEVEL`	int	等级
`TIM_PROFILE_TYPE_KEY_ROLE`	int	角色
`TIM_PROFILE_TYPE_KEY_SELFSIGNATURE`	String	签名
`TIM_PROFILE_TYPE_KEY_CUSTOM_PREFIX`	String, int	自定义字段前缀

自定义字段需要您加上我们的前缀。例如后台有一个自定义字段 `Blood` ，类型为整数，设置代码如下：

```
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");
}
});
```

好友关系

获取所有好友

可通过 `TIMFriendshipManager` 的 `getFriendList` 方法获取所有好友列表。

```
/**
* 获取好友列表
```

```
 * @param cb 回调 TIMFriend 列表
 */
public void getFriendList(@NonNull TIMValueCallBack<List<TIMFriend>> cb)
```

获取成功后返回好友列表，好友对象用 `TIMFriend` 存储，`TIMFriend` 的定义如下:

```
/**
 * 获取用户的 identifier
 * @return 用户的 identifier
 */
public String getIdentifier()

/**
 * 获取好友备注
 * @return 好友备注
 */
public String getRemark()

/**
 * 获取申请加好友的理由
 * @return 申请理由
 */
public String getAddWording()

/**
 * 获取申请加好友的来源
 * @return 申请来源
 */
public String getAddSource()

/**
 * 获取分组名称
 * @return 分组名称列表
 */
public List<String> getGroupNames()

/**
 * 获取好友自定义信息，key 值按照后台配置的字符串传入，不包括 TIM_FRIEND_PROFILE_TYPE_KEY_CUSTOM_PREFIX 前缀
 * @return 自定义信息 Map
 */
public Map<String, byte[]> getCustomInfo()

/**
 * 获取 uint 类型的好友自定义信息，key 值按照后台配置的字符串传入，不包括
 * TIM_FRIEND_PROFILE_TYPE_KEY_CUSTOM_PREFIX 前缀
 * @return 自定义信息 Map
 */
public Map<String, byte[]> getCustomInfoUint()

/**
 * 获取好友资料
 * @return 用户资料
 */
public TIMUserProfile getTimUserProfile()()
```

示例代码

```
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());
}
});
```

修改好友

修改好友调用 `modifyFriend` 方法进行。与修改自己资料方法类似，可一次更新多个字段。

```
/**
* 修改好友资料
* @param identifier 好友标识
* @param profileMap 修改的字段，见TIMFriend中的TIM_FRIEND_PROFILE_TYPE_KEY_XXX
* @param cb 回调
*/
public void modifyFriend(@NonNull String identifier, @NonNull HashMap<String, Object> profileMap, @NonNull TIMCallBack cb)
```

设置不存在的键值可能会导致失败，后台定义了一些常用的键值

Key	Value	说明
TIM_FRIEND_PROFILE_TYPE_KEY_REMARK	String	备注
TIM_FRIEND_PROFILE_TYPE_KEY_GROUP	List< String >	分组
TIM_FRIEND_PROFILE_TYPE_KEY_CUSTOM_PREFIX	String、int	自定义字段前缀

示例：设置好友『Android_002』的备注为『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");
}
});
```

> 说明：
> 修改好友自定义资料，需先通过 Server 配置关系链自定义字段，才能修改成功。

添加好友

通过 `TIMFriendshipManager` 的 `addFriend` 方法可以添加好友。

```
/**
* 添加好友
* @param timFriendRequest 添加请求
* @param cb 回调
*/
public void addFriend(@NonNull TIMFriendRequest timFriendRequest, @NonNull TIMValueCallBack<TIMFriendResult> cb)
```

加好友需要传入 request 参数，该参数类型定义如下：

```
/**
* 用户 identifier
*/
```

```
private String identifier = "";

/**
* 用户备注（备注最大96字节）
*/
private String remark = "";

/**
* 请求说明（最大120字节）
*/
private String addWording = "";

/**
* 添加来源
* 来源不能超过8个字节，并且需要添加“AddSource_Type_”前缀
*/
private String addSource = "";

/**
* 分组名
*/
private String friendGroup = "";
```

成功回调会返回操作用户的 `TIMFriendResult` 结果数据，开发者可根据对应情况提示用户。添加好友的返回码如下：

```
public class TIMFriendStatus {
/**
* 操作成功
*/
public static final int TIM_FRIEND_STATUS_SUCC = 0;

/**
* 请求参数错误，请根据错误描述检查请求是否正确
*/
public static final int TIM_FRIEND_PARAM_INVALID = 30001;

/**
* 加好友、响应好友时有效：自己的好友数已达系统上限
*/
public static final int TIM_ADD_FRIEND_STATUS_SELF_FRIEND_FULL = 30010;

/**
* 加好友、响应好友时有效：对方的好友数已达系统上限
*/
public static final int TIM_ADD_FRIEND_STATUS_THEIR_FRIEND_FULL = 30014;

/**
* 加好友时有效：被加好友在自己的黑名单中
*/
public static final int TIM_ADD_FRIEND_STATUS_IN_SELF_BLACK_LIST = 30515;

/**
* 加好友时有效：被加好友设置为禁止加好友
*/
public static final int TIM_ADD_FRIEND_STATUS_FRIEND_SIDE_FORBID_ADD = 30516;

/**
* 加好友时有效：已被被添加好友设置为黑名单
*/
public static final int TIM_ADD_FRIEND_STATUS_IN_OTHER_SIDE_BLACK_LIST = 30525;
/**
* 加好友时有效：等待好友审核同意
*/
public static final int TIM_ADD_FRIEND_STATUS_PENDING = 30539;
};
```

示例代码

```
TIMFriendRequest timFriendRequest = new TIMFriendRequest("test_id");
timFriendRequest.setAddWording("it's me!");
timFriendRequest.setAddSource("android");
TIMFriendshipManager.getInstance().addFriend(timFriendRequest, new TIMValueCallBack<TIMFriendResult>() {
@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());
}
});
```

删除好友

可通过 `TIMFriendshipManager` 的 `deleteFriends` 方法批量删除好友。

```
/**
* 删除好友
* @param identifiers 好友列表
* @param delFriendType 删除类型
* @param cb 回调
*/
public void deleteFriends(@NonNull List<String> identifiers, @NonNull int delFriendType, @NonNull TIMValueCallBack<List<TIMFriendResult>> cb)
```

成功回调会返回操作用户的 `TIMFriendResult` 结果数据，开发者可根据情况提示用户。 删除好友的错误码如下：

```
public class TIMFriendStatus {
/**
* 操作成功
*/
public static final int TIM_FRIEND_STATUS_SUCC = 0;
/**
* 删除好友时有效：删除好友时对方不是好友
*/
public static final int TIM_DEL_FRIEND_STATUS_NO_FRIEND = 31704;
};
```

示例代码

```
List<String> identifiers = new ArrayList<>();
identifiers.add("test_id");
TIMFriendshipManager.getInstance().deleteFriends(identifiers, TIMDelFriendType.TIM_FRIEND_DEL_SINGLE, 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");
}
});
```

同意/拒绝好友申请

可通过 `TIMFriendshipManager` 的 `doResponse` 方法同意/拒绝好友申请

```
/**
* 处理好友请求
* @param response 请求参数，包含好友 ID，预备注，回应类型
* @param cb
```

```
 */
public void doResponse(TIMFriendResponse response, @NonNull TIMValueCallBack<TIMFriendResult> cb)
```

参数 `response` 定义如下：

```
public class TIMFriendResponse {
/**
* 同意加好友（建立单向好友）
*/
public static final int TIM_FRIEND_RESPONSE_AGREE = 0;

/**
* 同意加好友并加对方为好友（建立双向好友）
*/
public static final int TIM_FRIEND_RESPONSE_AGREE_AND_ADD = 1;

/**
* 拒绝对方好友请求
*/
public static final int TIM_FRIEND_RESPONSE_REJECT = 2;

/**
* 响应类型
*/
private int responseType = TIM_FRIEND_RESPONSE_AGREE;

/**
* 响应的好友 ID
*/
private String identifier = ""; // 回应好友的 ID

/**
* 备注好友（可选，如果要加对方为好友）。备注最大96字节
*/
private String remark = "";

.....省略 get set 方法
}
```

成功回调会返回操作用户的 `TIMFriendResult` 结果数据，处理用户请求的错误码如下。

```
public class TIMFriendStatus {
/**
* 操作成功
*/
public static final int TIM_FRIEND_STATUS_SUCC = 0;

/**
* 加好友、响应好友时有效：自己的好友数已达系统上限
*/
public static final int TIM_ADD_FRIEND_STATUS_SELF_FRIEND_FULL = 30010;

/**
* 加好友、响应好友时有效：对方的好友数已达系统上限
*/
public static final int TIM_ADD_FRIEND_STATUS_THEIR_FRIEND_FULL = 30014;

/**
* 响应好友申请时有效：对方没有申请过好友
*/
public static final int TIM_RESPONSE_FRIEND_STATUS_NO_REQ = 30614;
};
```

校验好友关系

可通过 `TIMFriendshipManager` 的 `checkFriends` 方法校验好友关系。

```
/**
 * 校验好友
 * @param checkInfo 校验好友参数
 * @param cb 回调
 */
public void checkFriends(@NonNull TIMFriendCheckInfo checkInfo, @NonNull TIMValueCallBack<List<TIMCheckFriendResult>> cb)
```

参数 checkInfo 定义如下：

```
public class TIMFriendCheckInfo {
private List<String> users = new ArrayList<>();
private int checkType = TIMFriendCheckType.TIM_FRIEND_CHECK_TYPE_UNIDIRECTION;

/**
 * 设置需要检查的好友 ID
 *
 * @param users
 */
public void setUsers(List<String> users);

/**
 * 设置需要检查的关系类型，见 TIMFriendCheckType 定义的常量
 *
 * @param type
 */
public void setCheckType(int type);
}
```

参数 TIMFriendCheckType 定义如下：

```
public class TIMFriendCheckType {
/**
 * 单向好友
 */
public static final int TIM_FRIEND_CHECK_TYPE_UNIDIRECTION = 1;

/**
 * 互为好友
 */
public static final int TIM_FRIEND_CHECK_TYPE_BIDIRECTION = 2;
}
```

成功回调会返回操作用户的 TIMCheckFriendResult 列表数据，定义如下。

```
public class TIMCheckFriendResult {
private String identifier = "";
private int resultCode;
private String resultInfo = "";
private int resultType;

/**
 * 获取好友 ID
 *
 * @return 好友 ID
 */
public String getIdentifier();

/**
 * 获取返回码
 *
 * @return 返回码
 */
public int getResultCode();

/**
 * 获取返回结果描述
 *
```

```
 * @return 结果描述
 */
public String getResultInfo();

/**
 * 获取检查好友类型，常量见 TIMFriendRelationType 中定义
 *
 * @return 好友关系类型
 */
public int getResultType();
}
```

参数 `TIMFriendRelationType` 定义如下：

```
public class TIMFriendRelationType {
/**
 * 不是好友
 */
public static final int TIM_FRIEND_RELATION_TYPE_NONE = 0;

/**
 * 对方在我的好友列表中
 */
public static final int TIM_FRIEND_RELATION_TYPE_MY_UNI = 1;

/**
 * 我在对方的好友列表中
 */
public static final int TIM_FRIEND_RELATION_TYPE_OTHER_UNI = 2;

/**
 * 互为好友
 */
public static final int TIM_FRIEND_RELATION_TYPE_BOTH_WAY = 3;

}
```

好友未决

获取未决列表

其它用户通过 `addFriend` 方法添加自己为好友，此时会在后台增加一条未决记录。当自己向其它用户请求好友时，后台也会记录一条未决信息。可通过 `getPendencyList` 方法获取未决列表

```
/**
 * 获取未决列表
 *
 * @param timFriendPendencyRequest
 * @param cb
 */
public void getPendencyList(TIMFriendPendencyRequest timFriendPendencyRequest, @NonNull TIMValueCallBack<TIMFriendPendencyResponse> cb)
```

由于后台可能存储多条好未决，超出界面显示范围，所以此接口支持翻页操作。需要传入参数 `timFriendPendencyRequest` 定义如下：

```
/**
 * 未决列表序列号。建议客户端保存 seq 和未决列表，请求时填入 server 返回的 seq。如果 seq 是 server 最新的，则不返回数据
 *
 * @param seq 序列号
 */
public void setSeq(long seq)

/**
 * 翻页时间戳，只用来翻页，server 返回0时表示没有更多数据，第一次请求填0
 * 特别注意的是，如果 server 返回的 seq 跟填入的 seq 不同，翻页过程中，需要使用客户端原始 seq 请求，直到数据请求完毕，才能更新本地 seq
 *
```

```
 * @param timestamp 翻页时间戳
 */
public void setTimestamp(long timestamp)

/**
 * 每页的数量，请求时有效
 *
 * @param numPerPage 每页的数量
 */
public void setNumPerPage(int numPerPage)

/**
 * 未决请求拉取类型，见 TIMPendencyType 中的常量定义
 *
 * @param timPendencyType 未决请求拉取类型
 */
public void setTimPendencyGetType(int timPendencyType)
```

操作成功后，回调返回分页信息和未决记录 `TIMFriendPendencyResponse`

```
/**
 * 获取本次请求的未决列表序列号
 * @return 序列号
 */
public long getSeq()

/**
 * 获取本次请求的翻页时间戳
 *
 * @return 时间戳
 */
public long getTimestamp()

/**
 * 获取未决请求未读数量
 *
 * @return 未读数量
 */
public long getUnreadCnt()

/**
 * 获取未决信息列表
 *
 * @return 信息列表
 */
public List<TIMFriendPendencyItem> getItems()
```

未决请求 `TIMFriendPendencyItem` 定义如下：

```
/**
 * 获取用户 ID
 *
 * @return id
 */
public String getIdentifier()

/**
 * 获取增加时间
 *
 * @return 时间
 */
public long getAddTime()

/**
 * 获取来源
 *
 * @return 来源
```

```
 */
public String getAddSource()

/**
 * 获取好友附言
 *
 * @return 附言
 */
public String getAddWording()

/**
 * 获取好友昵称
 *
 * @return 昵称
 */
public String getNickname()

/**
 * 获取未决请求类型，见 TIMPendencyType 常量定义
 *
 * @return 未决请求类型
 */
public int getType()
```

未决类型 `TIMPendencyType` 定义如下：

```
public class TIMPendencyType {
/**
 * 别人发给我的未决请求
 */
public static final int TIM_PENDENCY_COME_IN = 1;

/**
 * 我发给别人的未决请求
 */
public static final int TIM_PENDENCY_SEND_OUT = 2;

/**
 * 别人发给我的以及我发给别人的所有未决请求，仅在拉取时有效。
 */
public static final int TIM_PENDENCY_BOTH = 3;

}
```

未决删除

```
/**
 * 未决删除
 *
 * @param pendencyType 未决类型，见 TIMPendencyType, 删除只支持 TIM_PENDENCY_COME_IN 和 TIM_PENDENCY_SEND_OUT
 * @param users 要删除的未决用户 ID 列表
 * @param cb 回调
 */
public void deletePendency(int pendencyType, List<String> users, @NonNull TIMValueCallBack<List<TIMFriendResult>> cb)
```

未决已读上报

当用户拉取到未决记录，可以将本次拉取的未决在后台标记为已读。

```
/**
 * 未决已读上报
 *
 * @param timestamp 已读时间戳，此时间戳以前的消息都将置为已读
 * @param cb 回调
 */
public void pendencyReport(long timestamp, @NonNull TIMCallBack cb)
```

上报后，下次调用 getPendencyList 返回的未读计数将会改变。

黑名单

添加用户到黑名单

可以把任意用户拉黑，如果此前是好友关系，拉黑后自动解除好友，拉黑后对方发消息无法收到。

```
/**
 * 添加用户到黑名单
 *
 * @param users 用户列表
 * @param cb 回调
 */
public void addBlackList(List<String> users, @NonNull TIMValueCallBack<List<TIMFriendResult>> cb)
```

把用户从黑名单删除

```
/**
 * 把用户从黑名单中删除
 *
 * @param users 用户列表
 * @param cb 回调
 */
public void deleteBlackList(List<String> users, @NonNull TIMValueCallBack<List<TIMFriendResult>> cb)
```

获取黑名单列表

```
/**
 * 获取黑名单列表
 *
 * @param cb 回调
 */
public void getBlackList(@NonNull TIMValueCallBack<List<TIMFriend>> cb)
```

好友分组

创建好友分组

创建分组时，可以同时指定添加的用户。同一用户可以添加到多个分组。

```
/**
 * 新建好友分组
 *
 * @param groupNames 分组名称列表,必须是当前不存在的分组
 * @param identifiers 要添加到分组中的好友
 * @param cb 回调
 */
public void createFriendGroup(List<String> groupNames, List<String> identifiers, @NonNull TIMValueCallBack<List<TIMFriendResult>> cb)
```

删除好友分组

```
/**
 * 删除好友分组
 *
 * @param groupNames 要删除的好友分组名称列表
 * @param cb 回调
 */
public void deleteFriendGroup(List<String> groupNames, @NonNull TIMCallBack cb)
```

添加好友到某分组

```
/**
 * 添加好友到某分组
 *
 * @param groupName 好友分组名称
 * @param identifiers 要添加到分组中的好友列表
 * @param cb 回调
 */
public void addFriendsToFriendGroup(String groupName, List<String> identifiers, @NonNull TIMValueCallBack<List<TIMFriendResult>> cb)
```

从某分组删除好友

```
/**
 * 从某分组删除好友
 *
 * @param groupName 好友分组名称
 * @param identifiers 要移除分组的好友列表
 * @param cb 回调
 */
public void deleteFriendsFromFriendGroup(String groupName, List<String> identifiers, @NonNull TIMValueCallBack<List<TIMFriendResult>> cb)
```

重命名好友分组

```
/**
 * 重命名好友分组
 *
 * @param oldName 原来的分组名称
 * @param newName 新的分组名称
 * @param cb 回调
 */
public void renameFriendGroup(String oldName, String newName, @NonNull TIMCallBack cb)
```

获取好友分组

```
/**
 * 获取指定的好友分组，传入 null 获得所有分组信息
 * @param groupNames 要获取信息的好友分组名称列表
 * @param cb 回调
 */
public void getFriendGroups(List<String> groupNames, @NonNull TIMValueCallBack<List<TIMFriendGroup>> cb)
```

关系链变更系统通知

`TIMMessage` 中 `Elem` 类型 `TIMSNSSystemElem` 为关系链变更系统消息。

```
/**
 * 关系链相关操作后，后台 push 同步下来的消息元素
 *
 */
public class TIMSNSSystemElem extends TIMElem {
private int subType = 0;

// subType 对应 TIMSNSSystemType.TIM_SNS_SYSTEM_ADD_FRIEND
private List<String> requestAddFriendUserList = new ArrayList<>();

// subType 对应 TIMSNSSystemType.TIM_SNS_SYSTEM_DEL_FRIEND
private List<String> delRequestAddFriendUserList = new ArrayList<>();

// subType 对应 TIMSNSSystemType.TIM_SNS_SYSTEM_ADD_BLACKLIST
private List<String> addBlacklistUserList = new ArrayList<>();

// subType 对应 TIMSNSSystemType.TIM_SNS_SYSTEM_DEL_BLACKLIST
private List<String> delBlacklistUserList = new ArrayList<>();
```

```
// subType 对应 TIMSNSSystemType.TIM_SNS_SYSTEM_ADD_FRIEND_REQ
private List<TIMFriendPendencyInfo> friendAddPendencyList = new ArrayList<>();

// subType 对应 TIMSNSSystemType.TIM_SNS_SYSTEM_DEL_FRIEND_REQ
private List<String> delFriendAddPendencyList = new ArrayList<>();

// subType 对应 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();
}


/**
* 关系链变更系统通知类型
*/
public class TIMSNSSystemType {
/**
* 增加好友消息
*/
public static final int TIM_SNS_SYSTEM_ADD_FRIEND = 0x01;

/**
* 删除好友消息
*/
public static final int TIM_SNS_SYSTEM_DEL_FRIEND = 0x02;

/**
* 增加好友申请
*/
public static final int TIM_SNS_SYSTEM_ADD_FRIEND_REQ = 0x03;

/**
* 删除未决申请
*/
public static final int TIM_SNS_SYSTEM_DEL_FRIEND_REQ = 0x04;

/**
* 黑名单添加
*/
public static final int TIM_SNS_SYSTEM_ADD_BLACKLIST = 0x05;

/**
* 黑名单删除
*/
public static final int TIM_SNS_SYSTEM_DEL_BLACKLIST = 0x06;

/**
* 未决已读上报
*/
public static final int TIM_SNS_SYSTEM_PENDENCY_REPORT = 0x07;

/**
* 关系链资料变更
*/
public static final int TIM_SNS_SYSTEM_SNS_PROFILE_CHANGE = 0x08;
};

/**
```

```
 * 关系链变更详细信息
 *
 */
public class TIMSNSChangeInfo {
/**
 * 变更资料的用户 ID
 */
private String updateUser = "";

/**
 * 变更资料信息
 */
private Map<String, Object> itemMap = new HashMap<>();

public String getUpdateUser() {
return updateUser;
}

public Map<String, Object> getItemMap() {
return itemMap;
}
}
```

添加好友系统通知

当两个用户成为好友时，两个用户均可收到添加好友系统消息。

触发时机：

当自己的关系链变更，增加好友时，收到消息（如果已经是单向好友，关系链没有变更的一方不会收到）。

参数说明：

参数	说明
subType	TIM_SNS_SYSTEM_ADD_FRIEND
requestAddFriendUserList	成为好友的用户列表

删除好友系统通知

当两个用户解除好友关系时，会收到删除好友系统消息：

触发时机：

当自己的关系链变更，删除好友时，收到消息（如果删除的是单向好友，关系链没有变更的一方不会收到）。

参数说明：

参数	说明
subType	TIM_SNS_SYSTEM_DEL_FRIEND
delRequestAddFriendUserList	删除好友的用户列表

好友申请系统通知

当申请好友时对方需要验证，自己和对方会收到好友申请系统通知。

触发时机：

当申请好友时对方需要验证，自己和对方会收到好友申请系统通知，对方可选择同意或者拒绝，自己不能操作，只做信息同步之用。

参数说明：

参数	说明
subType	TIM_SNS_SYSTEM_ADD_FRIEND_REQ
friendAddPendencyList	申请的好友信息列表

TIMFriendPendencyInfo 参数说明：

参数	说明
fromUser	添加好友操作者
addSource	添加好友的来源
fromUserNickName	添加好友的操作者的昵称
addWording	添加好友附言

删除未决请求通知

触发时机：

当申请对方为好友，申请审核通过或者被拒后，自己会收到删除未决请求消息。

参数说明：

参数	说明
subType	TIM_SNS_SYSTEM_DEL_FRIEND_REQ
delFriendAddPendencyList	被通过或者被拒绝的好友列表

用户资料变更系统通知

TIMMessage 中 Elem 类型 TIMProfileSystemElem 为用户资料变更系统消息。

```
/**
* 自身和好友资料修改，后台 push 下来的消息元素
*/
public class TIMProfileSystemElem extends TIMElem {
private int subType; //修改资料的类型 TIMProfileSystemType
private String fromUser; //修改资料的来源（谁修改了）
private Map<String, Object> itemMap; //用户的资料

public int getSubType();
public String getFromUser();
public Map<String, Object> getItemMap();
}

/**
* 用户资料变更系统通知类型
*/
public class TIMProfileSystemType {
/**
* 无效值
*/
public static final int INVALID = 0;

/**
* 好友资料变更
*/
public static final int TIM_PROFILE_SYSTEM_FRIEND_PROFILE_CHANGE = 1;
}
```

当自己的资料或者好友的资料变更时，会收到用户资料变更系统消息。例如好友修改了头像，那么 TIMProfileSystemElem 中的 itemMap 的 key 为 Tag_Profile_IM_Image ， value 值为头像的 url 地址，其中 key 常量值定义在 TIMUserProfile 中。

用户资料与关系链（iOS）

最近更新时间：2020-05-22 14:59:58

即时通信 IM 提供了关系链和用户资料托管，App 开发者使用简单的接口就可实现关系链和用户资料存储功能，另外，为了方便不同用户定制化资料，也提供用户资料和用户关系链的自定义字段（用户自定义字段需要先在控制台配置，详情请参考 用户自定义字段）。本节所有的接口不论对独立帐号体系还是托管帐号体系都有效。

用户资料

获取自己的资料

可通过 `TIMFriendshipManager` 的 `getSelfProfile` 方法获取用户自己的资料。

```
/**
* 获取自己的资料
*
* @param succ 成功回调，返回 TIMUserProfile
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)getSelfProfile:(TIMGetProfileSucc)succ fail:(TIMFail)fail;
```

如果获取成功，succ 回调会返回获取到的 `TIMUserProfile` 对象。 `TIMUserProfile` 的定义如下：

```
/**
* 用户资料
*/
@interface TIMUserProfile : TIMCodingModel

/**
* 用户 identifier
*/
@property(nonatomic,strong) NSString* identifier;

/**
* 用户昵称
*/
@property(nonatomic,strong) NSString* nickname;

/**
* 好友验证方式
*/
@property(nonatomic,assign) TIMFriendAllowType allowType;

/**
* 用户头像
*/
@property(nonatomic,strong) NSString* faceURL;

/**
* 用户签名
*/
@property(nonatomic,strong) NSData* selfSignature;

/**
* 用户性别
*/
@property(nonatomic,assign) TIMGender gender;

/**
* 用户生日
*/
@property(nonatomic,assign) uint32_t birthday;
```

```
/**
* 用户区域
*/
@property(nonatomic,strong) NSData* location;

/**
* 用户语言
*/
@property(nonatomic,assign) uint32_t language;

/**
* 等级
*/
@property(nonatomic,assign) uint32_t level;

/**
* 角色
*/
@property(nonatomic,assign) uint32_t role;

/**
* 自定义字段集合,key 是 NSString 类型,value 是 NSData 类型或者 NSNumber 类型
* (key 值按照后台配置的字符串传入)
*/
@property(nonatomic,strong) NSDictionary* customInfo;

@end
```

获取指定用户的资料

可通过 TIMFriendshipManager 的 getUsersProfile 方法获取指定用户的资料。该方法支持从缓存和后台两种方式获取，当 forceUpdate = YES 时，会强制从后台拉取数据，并把返回的数据缓存下来；当 forceUpdate = NO 时，则先在本地查找，如果没有再向后台请求数据。建议只在显示资料的时候强制拉取，以减少等待时间。

```
/**
* 获取指定好友资料
*
* @param users 用户 ID
* @prarm forceUpdate 强制从后台拉取
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)getUsersProfile:(NSArray<NSString *> *)users forceUpdate:(BOOL)forceUpdate succ:(TIMGetProfileArraySucc)succ fail:(TIMFail)fail;
@end
```

示例：获取『iOS_002』和『iOS_003』两个用户的资料

```
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);
}];
```

该示例关闭了强制后台拉取，优先从缓存中查找用户资料，可减少等待时间。缓存的时间可通过 TIMFriendProfileOption 设置，默认缓存时间一天。

```
/**
* 资料与关系链
*/
@interface TIMFriendProfileOption : NSObject

/**
```

```
 * 关系链最大缓存时间
 * 默认缓存一天；获取资料和关系链超过缓存时间，将自动向服务器发起请求
 */
@property NSInteger expiredSeconds;

@end
```

设置的过期时间方法是 `-[TIMManager setUserConfig:]` ，示例代码：

```
TIMUserConfig *config = ...;
TIMFriendProfileOption *option = [TIMFriendProfileOption new];
option.expiredSeconds = 60*60; //1小时
config.friendProfileOpt = option;
[[TIMManager sharedInstance] setUserConfig:config];
```

修改自己的资料

修改自己的资料通过 `modifySelfProfile` 方法完成

```
@interface TIMFriendshipManager : NSObject
/**
 * 设置自己的资料
 *
 * @param values 需要更新的属性，可一次更新多个字段
 * @param succ 成功回调
 * @param fail 失败回调
 *
 * @return 0 发送请求成功
 */
- (int)modifySelfProfile:(NSDictionary<NSString *, id> *)values succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

通过 values 字典，可以一次设置多个字段。举例来说，设置昵称的代码如下：

```
[[TIMFriendshipManager sharedInstance] modifySelfProfile:@{TIMProfileTypeKey_Nick:@"我的昵称"} succ:nil fail:nil];
```

设置不存在的键值可能会导致失败，后台定义了一些常用的键值

Key	Value	说明
TIMProfileTypeKey_Nick	NSString	昵称
TIMProfileTypeKey_FaceUrl	NSString	头像
TIMProfileTypeKey_AllowType	NSNumber	好友申请
TIMProfileTypeKey_Gender	NSNumber	性别
TIMProfileTypeKey_Birthday	NSNumber	生日
TIMProfileTypeKey_Location	NSString	位置
TIMProfileTypeKey_Language	NSNumber	语言
TIMProfileTypeKey_Level	NSNumber	等级
TIMProfileTypeKey_Role	NSNumber	角色
TIMProfileTypeKey_SelfSignature	NSString	签名
TIMProfileTypeKey_Custom_Prefix	NSString、NSData 或 NSNumber	自定义字段前缀

自定义字段需要您加上我们的前缀。例如后台有一个自定义字段 `Blood` ，类型为整数，设置代码是

```
NSString *key = [TIMProfileTypeKey_Custom_Prefix stringByAppendingString:@"Blood"];
[[TIMFriendshipManager sharedInstance] modifySelfProfile:@{key:@1} succ:nil fail:nil];
```

说明：
当设置自定义字段值是 NSString 对象时，后台会将其转为 UTF8 保存在数据库中。由于部分用户迁移资料时可能不是 UTF8 类型，所以在获取资料时，统一返回 NSData 类型。

好友关系

获取所有好友

可通过 TIMFriendshipManager 的 getFriendList 方法获取所有好友列表

```
@interface TIMFriendshipManager : NSObject
/**
* 获取好友列表
*
* @param succ 成功回调，返回好友(TIMFriend)列表
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
-(int)getFriendList:(TIMFriendArraySucc)succ fail:(TIMFail)fail;
@end
```

如果获取成功，succ 回调返回好友列表。好友对象用 TIMFriend 存储， TIMFriend 的定义如下

```
@interface TIMFriend : TIMCodingModel

/**
* 好友 identifier
*/
@property(nonatomic,strong) NSString *identifier;

/**
* 好友备注
*/
@property(nonatomic,strong) NSString *remark;

/**
* 分组名称 NSString* 列表
*/
@property(nonatomic,strong) NSArray *groups;

/**
* 申请理由
*/
@property(nonatomic,strong) NSString * addWording;

/**
* 申请来源
*/
@property(nonatomic,strong) NSString * addSource;

/**
* 添加时间
*/
@property(nonatomic,assign) uint64_t addTime;

/**
* 自定义字段集合,key 是 NSString 类型,value 是 NSData 类型或者 NSNumber 类型(key 值按照后台配置的字符串传入)
*/
@property(nonatomic,strong) NSDictionary* customInfo;

/**
* 好友资料
*/
@property(nonatomic,strong) TIMUserProfile *profile;
```

```
@end
```

示例代码

```
[[TIMFriendshipManager sharedInstance] getFriendList:^(NSArray<TIMFriend *> *friends) {
NSMutableString *msg = [NSMutableString new];
[msg appendString:@"好友列表: "];
for (TIMFriend *friend in friends) {
[msg appendFormat:@"[%@,%@,%d,%@,%@,%@]", friend.identifier, friend.remark, friend.addTime, friend.addSource, friend.addWording, friend.groups];
}
self.msgLabel.text = msg;
} fail:^(int code, NSString *msg) {
self.msgLabel.text = [NSString stringWithFormat:@"失败：%d, %@", code, msg];
}];
```

修改好友

修改好友调用 `modifyFriend` 方法进行。与修改自己资料方法类似，该方法通过传入字典方式修改，可一次更新多个字段。

```
@interface TIMFriendshipManager : NSObject
/**
* 修改好友
*
* @param identifier 好友的 identifier
* @param values 需要更新的属性，可一次更新多个字段. 参见 TIMFriendshipDefine.h 的 TIMFriendTypeKey_XXX
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)modifyFriend:(NSString *)identifier values:(NSDictionary<NSString *, id> *)values succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

设置不存在的键值可能会导致失败，后台定义了一些常用的键值

Key	Value	说明
TIMFriendTypeKey_Remark	NSString	备注
TIMFriendTypeKey_Group	NSArray	分组
TIMFriendTypeKey_Custom_Prefix	NSNumber、NSData	自定义字段前缀

示例：设置好友『iOS_002』的备注为『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:@"失败：%d, %@", code, msg];
}];
```

> 修改好友自定义资料，需先通过 Server 配置关系链自定义字段，才能修改成功。

添加好友

通过 `TIMFriendshipManager` 的 `addFriend` 方法可以添加好友。

```
@interface TIMFriendshipManager : NSObject

/**
* 添加好友
*
* @param request 添加好友请求
```

```
 * @param succ 成功回调(TIMFriendResult)
 * @param fail 失败回调
 *
 * @return 0 发送请求成功
 */
- (int)addFriend:(TIMFriendRequest *)request succ:(TIMFriendResultSucc)succ fail:(TIMFail)fail;

@end
```

加好友需要传入 request 参数，该参数类型定义如下：

```
/**
 * 加好友请求
 */
@interface TIMFriendRequest : TIMCodingModel

/**
 * 用户 identifier
 */
@property(nonatomic,strong) NSString* identifier;

/**
 * 用户备注（备注最大96字节）
 */
@property(nonatomic,strong) NSString* remark;

/**
 * 请求说明（最大120字节）
 */
@property(nonatomic,strong) NSString* addWording;

/**
 * 添加来源
 * 来源不能超过8个字节，并且需要添加“AddSource_Type_”前缀
 */
@property(nonatomic,strong) NSString* addSource;

/**
 * 分组名
 */
@property(nonatomic,strong) NSString* group;

@end
```

成功回调会返回操作用户的 `TIMFriendResult` 结果数据，开发者可根据对应情况提示用户。添加好友的返回码如下。

```
typedef NS_ENUM(NSInteger, TIMFriendStatus) {
/**
 * 操作成功
 */
TIM_FRIEND_STATUS_SUCC = 0,
/**
 * 加好友时有效：被加好友在自己的黑名单中
 */
TIM_ADD_FRIEND_STATUS_IN_SELF_BLACK_LIST = 30515,
/**
 * 加好友时有效：被加好友设置为禁止加好友
 */
TIM_ADD_FRIEND_STATUS_FRIEND_SIDE_FORBID_ADD = 30516,
/**
 * 加好友、响应好友时有效：自己的好友数已达系统上限
 */
TIM_ADD_FRIEND_STATUS_SELF_FRIEND_FULL = 30010,
/**
 * 加好友时有效：已被被添加好友设置为黑名单
 */
TIM_ADD_FRIEND_STATUS_IN_OTHER_SIDE_BLACK_LIST = 30525,
```

```
/**
* 加好友时有效：等待好友审核同意
*/
TIM_ADD_FRIEND_STATUS_PENDING = 30539,
};
```

示例代码

```
TIMFriendRequest *q = [TIMFriendRequest new];
q.identifier = @"abc"; // 加好友 abc
q.addWording = @"求通过";
q.addSource = @"AddSource_Type_iOS";
q.remark = @"你是abc";
[[TIMFriendshipManager sharedInstance] addFriend:q succ:^(TIMFriendResult *result) {
if (result.result_code == 0)
self.msgLabel.text = @"添加成功";
else
self.msgLabel.text = [NSString stringWithFormat:@"异常：%ld, %@", (long)result.result_code, result.result_info];
} fail:^(int code, NSString *msg) {
self.msgLabel.text = [NSString stringWithFormat:@"失败：%d, %@", code, msg];
}];
```

删除好友

可通过 `TIMFriendshipManager` 的 `deleteFriends` 方法批量删除好友。

```
@interface TIMFriendshipManager : NSObject
/**
* 删除好友
*
* @param user 要删除的好友的 identifier
* @param delType 删除类型（单向好友、双向好友）
* @param succ 成功回调([TIMFriendResult])
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)delFriend:(NSString *)user delType:(TIMDelFriendType)delType succ:(TIMHandleFriendArraySucc)succ fail:(TIMFail)fail;
@end
```

成功回调会返回操作用户的 `TIMFriendResult` 结果数据，开发者可根据情况提示用户。 删除好友的错误码如下。

```
typedef NS_ENUM(NSInteger, TIMFriendStatus) {
/**
* 操作成功
*/
TIM_FRIEND_STATUS_SUCC = 0,
/**
* 删除好友时有效：删除好友时对方不是好友
*/
TIM_DEL_FRIEND_STATUS_NO_FRIEND = 31704,
};
```

示例代码

```
NSMutableArray * del_users = [[NSMutableArray alloc] init];
// 删除好友 iOS_002
[del_users addObject:@"iOS_002"];
// TIM_FRIEND_DEL_BOTH 指定删除双向好友
[[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);
}];
```

同意/拒绝 好友申请

可通过 `TIMFriendshipManager` 的 `doResponse` 方法同意/拒绝好友申请

```
@interface TIMFriendshipManager : NSObject
/**
* 响应对方好友邀请
*
* @param response 响应请求
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)doResponse:(TIMFriendResponse *)response succ:(TIMFriendResultSucc)succ fail:(TIMFail)fail;
@end
```

参数 response 的定义如下：

```
typedef NS_ENUM(NSInteger, TIMFriendResponseType) {
/**
* 同意加好友（建立单向好友）
*/
TIM_FRIEND_RESPONSE_AGREE = 0,
/**
* 同意加好友并加对方为好友（建立双向好友）
*/
TIM_FRIEND_RESPONSE_AGREE_AND_ADD = 1,
/**
* 拒绝对方好友请求
*/
TIM_FRIEND_RESPONSE_REJECT = 2,
};
/**
* 响应好友请求
*/
@interface TIMFriendResponse : NSObject

/**
* 响应类型
*/
@property(nonatomic,assign) TIMFriendResponseType responseType;

/**
* 用户 identifier
*/
@property(nonatomic,strong) NSString* identifier;

/**
* 备注好友（可选，如果要加对方为好友）。备注最大96字节
*/
@property(nonatomic,strong) NSString* remark;

@end
```

成功回调会返回操作用户的 `TIMFriendResult` 结果数据，处理用户请求的错误码如下。

```
typedef NS_ENUM(NSInteger, TIMFriendStatus) {
/**
* 操作成功
*/
TIM_FRIEND_STATUS_SUCC = 0,
```

```
/**
* 响应好友申请时有效：对方没有申请过好友
*/
TIM_RESPONSE_FRIEND_STATUS_NO_REQ = 30614,
/**
* 加好友、响应好友时有效：自己的好友数已达系统上限
*/
TIM_ADD_FRIEND_STATUS_SELF_FRIEND_FULL = 30010,
/**
* 加好友、响应好友时有效：对方的好友数已达系统上限
*/
TIM_ADD_FRIEND_STATUS_THEIR_FRIEND_FULL = 30014,
};
```

校验好友关系

可通过 `TIMFriendshipManager` 的 `checkFriends` 方法校验好友关系。

```
/**
* 检查指定用户的好友关系
*
* @param checkInfo 好友检查信息
* @param succ 成功回调，返回检查结果
* @param fail 失败回调
*
* @return 0 发送成功
*/
- (int)checkFriends:(TIMFriendCheckInfo *)checkInfo succ:(TIMCheckFriendResultArraySucc)succ fail:(TIMFail)fail;
```

参数 `checkInfo` 定义如下：

```
/**
* 好友关系检查
*/
@interface TIMFriendCheckInfo : NSObject
/**
* 检查用户的 ID 列表（NSString*）
*/
@property(nonatomic,strong) NSArray* users;

/**
* 检查类型
*/
@property(nonatomic,assign) TIMFriendCheckType checkType;

@end
```

参数 `TIMFriendCheckType` 定义如下：

```
/**
* 好友检查类型
*/
typedef NS_ENUM(NSInteger,TIMFriendCheckType) {
/**
* 单向好友
*/
TIM_FRIEND_CHECK_TYPE_UNIDIRECTION = 0x1,
/**
* 互为好友
*/
TIM_FRIEND_CHECK_TYPE_BIDIRECTION = 0x2,
};
```

成功回调会返回操作用户的 `TIMCheckFriendResult` 列表数据，定义如下。

```
@interface TIMCheckFriendResult : NSObject
/**
```

```
 * 用户 ID
 */
@property NSString* identifier;

/**
 * 返回码
 */
@property NSInteger result_code;

/**
 * 返回信息
 */
@property NSString *result_info;

/**
 * 检查结果
 */
@property(nonatomic,assign) TIMFriendRelationType resultType;

@end
```

参数 `TIMFriendRelationType` 定义如下：

```
/**
 * 好友关系类型
 */
typedef NS_ENUM(NSInteger,TIMFriendRelationType) {
/**
 * 不是好友
 */
TIM_FRIEND_RELATION_TYPE_NONE = 0x0,
/**
 * 对方在我的好友列表中
 */
TIM_FRIEND_RELATION_TYPE_MY_UNI = 0x1,
/**
 * 我在对方的好友列表中
 */
TIM_FRIEND_RELATION_TYPE_OTHER_UNI = 0x2,
/**
 * 互为好友
 */
TIM_FRIEND_RELATION_TYPE_BOTHWAY = 0x3,
};
```

好友未决

获取未决列表

其它用户通过 addFriend 方法添加自己为好友，此时会在后台增加一条未决记录。当自己向其它用户请求好友时，后台也会记录一条未决信息。可通过 getPendencyList 方法获取未决列表

```
@interface TIMFriendshipManager : NSObject

/**
 * 获取未决列表
 *
 * @param pendencyRequest 请求信息，详细参考 TIMFriendPendencyRequest
 * @param succ 成功回调
 * @param fail 失败回调
 *
 * @return 0 发送请求成功
 */
- (int)getPendencyList:(TIMFriendPendencyRequest *)pendencyRequest succ:(TIMGetFriendPendencyListSucc)succ fail:(TIMFail)fail;
@end
```

由于后台可能存储多条好未决，超出界面显示范围，所以此接口支持翻页操作。需要传入参数 pendencyRequest 定义如下

```
/**
* 未决请求信息
*/
@interface TIMFriendPendencyRequest : TIMCodingModel

/**
* 序列号，未决列表序列号
* 建议客户端保存 seq 和未决列表，请求时填入 server 返回的 seq
* 如果 seq 是 server 最新的，则不返回数据
*/
@property(nonatomic,assign) uint64_t seq;

/**
* 翻页时间戳，只用来翻页，server 返回0时表示没有更多数据，第一次请求填0
* 特别注意的是，如果 server 返回的 seq 跟填入 seq 不同，翻页过程中，需要使用客户端原始 seq 请求，直到数据请求完毕，才能更新本地 seq
*/
@property(nonatomic,assign) uint64_t timestamp;

/**
* 每页的数量，即本次请求最多返回都个数据
*/
@property(nonatomic,assign) uint64_t numPerPage;

/**
* 未决请求拉取类型
*/
@property(nonatomic,assign) TIMPendencyGetType type;

@end
```

操作成功后，succ 回调返回分页信息和未决记录

```
/**
* 未决返回信息
*/
@interface TIMFriendPendencyResponse : TIMCodingModel

/**
* 本次请求的未决列表序列号
*/
@property(nonatomic,assign) uint64_t seq;

/**
* 本次请求的翻页时间戳
*/
@property(nonatomic,assign) uint64_t timestamp;

/**
* 未决请求未读数量
*/
@property(nonatomic,assign) uint64_t unreadCnt;

@end

/**
* 未决请求
*/
@interface TIMFriendPendencyItem : TIMCodingModel

/**
* 用户标识
*/
@property(nonatomic,strong) NSString* identifier;
/**
* 增加时间
```

```
*/
@property(nonatomic,assign) uint64_t addTime;
/**
* 来源
*/
@property(nonatomic,strong) NSString* addSource;
/**
* 加好友附言
*/
@property(nonatomic,strong) NSString* addWording;

/**
* 加好友昵称
*/
@property(nonatomic,strong) NSString* nickname;

/**
* 未决请求类型
*/
@property(nonatomic,assign) TIMPendencyGetType type;

@end
```

未决删除

```
@interface TIMFriendshipManager : NSObject
/**
* 未决删除
*
* @param type 未决好友类型
* @param identifiers 要删除的未决列表
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)deletePendency:(TIMPendencyGetType)type users:(NSArray *)identifiers succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

未决已读上报

当用户拉取到未决记录，可以将本次拉取的未决在后台标记为已读。

```
@interface TIMFriendshipManager : NSObject
/**
* 未决已读上报
*
* @param timestamp 已读时间戳，此时间戳以前的消息都将置为已读
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)pendencyReport:(uint64_t)timestamp succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

上报后，下次调用 `getPendencyList` 返回的未读计数将会改变。

黑名单

添加用户到黑名单

可以把任意用户拉黑，如果此前是好友关系，拉黑后自动解除好友，拉黑后对方发消息无法收到。

```
@interface TIMFriendshipManager : NSObject
/**
* 添加用户到黑名单
*
* @param identifiers 用户列表
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)addBlackList:(NSArray *)identifiers succ:(TIMFriendResultArraySucc)succ fail:(TIMFail)fail;
@end
```

把用户从黑名单删除

```
@interface TIMFriendshipManager : NSObject
/**
* 把用户从黑名单中删除
*
* @param identifiers 用户列表
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)deleteBlackList:(NSArray *)identifiers succ:(TIMFriendResultArraySucc)succ fail:(TIMFail)fail;
@end
```

获取黑名单列表

```
@interface TIMFriendshipManager : NSObject
/**
* 获取黑名单列表
*
* @param succ 成功回调，返回 NSString* 列表
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)getBlackList:(TIMFriendArraySucc)succ fail:(TIMFail)fail;
@end
```

好友分组

创建好友分组

创建分组时，可以同时指定添加的用户。同一用户可以添加到多个分组。

```
/**
* 新建好友分组
*
* @param groupNames 分组名称列表,必须是当前不存在的分组
* @param identifiers 要添加到分组中的好友
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)createFriendGroup:(NSArray *)groupNames users:(NSArray *)identifiers succ:(TIMFriendResultArraySucc)succ fail:(TIMFail)fail;
@end
```

删除好友分组

```
@interface TIMFriendshipManager : NSObject
/**
* 删除好友分组
*
* @param groupNames 要删除的好友分组名称列表
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)deleteFriendGroup:(NSArray *)groupNames succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

添加好友到某分组

```
@interface TIMFriendshipManager : NSObject
/**
* 添加好友到一个好友分组
*
* @param groupName 好友分组名称
* @param identifiers 要添加到分组中的好友
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)addFriendsToFriendGroup:(NSString *)groupName users:(NSArray *)identifiers succ:(TIMFriendResultArraySucc)succ fail:(TIMFail)fail;
@end
```

从某分组删除好友

```
@interface TIMFriendshipManager : NSObject
/**
* 从好友分组中删除好友
*
* @param groupName 好友分组名称
* @param identifiers 要移出分组的好友
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)delFriendsFromFriendGroup:(NSString *)groupName users:(NSArray *)identifiers succ:(TIMFriendResultArraySucc)succ fail:(TIMFail)fail;
@end
```

重命名好友分组

```
@interface TIMFriendshipManager : NSObject
/**
* 修改好友分组的名称
*
* @param oldName 原来的分组名称
* @param newName 新的分组名称
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)renameFriendGroup:(NSString*)oldName newName:(NSString*)newName succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

获取指定的好友分组

```
@interface TIMFriendshipManager : NSObject
/**
* 获取指定的好友分组信息
*
* @param groupNames 要获取信息的好友分组名称列表,传入 nil 获得所有分组信息
* @param succ 成功回调，返回 TIMFriendGroup* 列表
* @param fail 失败回调
*
* @return 0 发送请求成功
*/
- (int)getFriendGroups:(NSArray *)groupNames succ:(TIMFriendGroupArraySucc)succ fail:(TIMFail)fail;
@end
```

关系链变更系统通知

`TIMMessage` 中 `Elem` 类型 `TIMSNSSystemElem` 为关系链变更系统消息。

原型：

```
typedef NS_ENUM(NSInteger, TIM_SNS_SYSTEM_TYPE){
/**
* 增加好友消息
*/
TIM_SNS_SYSTEM_ADD_FRIEND = 0x01,
/**
* 删除好友消息
*/
TIM_SNS_SYSTEM_DEL_FRIEND = 0x02,
/**
* 增加好友申请
*/
TIM_SNS_SYSTEM_ADD_FRIEND_REQ = 0x03,
/**
* 删除未决申请
*/
TIM_SNS_SYSTEM_DEL_FRIEND_REQ = 0x04,
};
/**
* 关系链变更详细信息
*/
@interface TIMSNSChangeInfo : NSObject
/**
* 用户 identifier
*/
@property(nonatomic,retain) NSString * identifier;
/**
* 申请添加时有效，添加理由
*/
@property(nonatomic,retain) NSString * wording;
/**
* 申请时填写，添加来源
*/
@property(nonatomic,retain) NSString * source;
@end
/**
* 关系链变更消息
*/
@interface TIMSNSSystemElem : TIMElem
/**
* 操作类型
*/
@property(nonatomic,assign) TIM_SNS_SYSTEM_TYPE type;
/**
* 被操作用户列表：TIMSNSChangeInfo 列表
*/
```

```
@property(nonatomic,retain) NSArray * users;

@end
```

成员说明：

成员	说明
type	变更类型
users	变更的用户列表

示例中，当成为好友或者解除好友关系时，打印日志，当有用户申请成为好友时，打印申请理由。 **示例：**

```
@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);
}];
```

添加好友系统通知

当两个用户成为好友时，两个用户均可收到添加好友系统消息。

触发时机：

当自己的关系链变更，增加好友时，收到消息（如果已经是单向好友，关系链没有变更的一方不会收到）。

参数说明：

参数	说明
type	TIM_SNS_SYSTEM_ADD_FRIEND
users	成为好友的用户列表

`TIMSNSChangeInfo` 参数说明：

成员	说明
identifier	用户 identifier

删除好友系统通知

当两个用户解除好友关系时，会收到删除好友系统消息：

触发时机：

当自己的关系链变更，删除好友时，收到消息（如果删除的是单向好友，关系链没有变更的一方不会收到）。

参数说明：

参数 | 说明
type | TIM_SNS_SYSTEM_DEL_FRIEND
users | 删除好友的用户列表

`TIMSNSChangeInfo` 参数说明：

成员	说明
identifier	用户 identifier

好友申请系统通知

当申请好友时对方需要验证，自己和对方会收到好友申请系统通知。

触发时机：当申请好友时对方需要验证，自己和对方会收到好友申请系统通知，对方可选择同意或者拒绝，自己不能操作，只做信息同步之用。

参数说明：

参数	说明
type	TIM_SNS_SYSTEM_ADD_FRIEND_REQ
users	申请的好友列表

`TIMSNSChangeInfo` 参数说明：

参数	说明
identifier	用户 identifier
wording	申请理由
source	申请来源

删除未决请求通知

触发时机：当申请对方为好友，申请审核通过或者被拒后，自己会收到删除未决请求消息。

用户资料变更系统通知

`TIMMessage` 中 `Elem` 类型 `TIMProfileSystemElem` 为用户资料变更系统消息。

```
/**
* 自身和好友资料修改，后台 push 下来的消息元素
*/
@interface TIMProfileSystemElem : TIMElem
/**
* 变更类型
*/
@property(nonatomic,assign) TIM_PROFILE_SYSTEM_TYPE type;
/**
* 资料变更的用户
*/
@property(nonatomic,strong) NSString * fromUser;
/**
* 资料变更的昵称（暂未实现）
*/
@property(nonatomic,strong) NSString * nickName;
@end
/**
* 资料变更
*/
typedef NS_ENUM(NSInteger, TIM_PROFILE_SYSTEM_TYPE){
/**
好友资料变更
*/
TIM_PROFILE_SYSTEM_FRIEND_PROFILE_CHANGE = 0x01,
};
```

当自己的资料或者好友的资料变更时，会收到用户资料变更系统消息。

离线推送
离线推送（Android）
离线推送基本配置

最近更新时间：2020-03-24 14:35:06

概述

即时通信 IM 的终端用户需要随时都能够得知最新的消息，而由于移动端设备的性能与电量有限，当 App 处于后台时，为了避免维持长连接而导致的过多资源消耗，即时通信 IM 推荐您使用各厂商提供的系统级推送通道来进行消息通知，系统级的推送通道相比第三方推送拥有更稳定的系统级长连接，可以做到随时接受推送消息，且资源消耗大幅降低。

即时通信 IM 目前已经支持了 APNs、小米推送、华为推送、魅族推送、vivo 推送、OPPO 推送等厂商推送，具体如下：

推送通道	系统要求	条件说明
APNs	iOS	iOS 系统推送通道，也是唯一的 iOS 推送通道
小米推送	MIUI	使用小米推送 MiPush_SDK_Client_3_6_12.jar
华为推送	EMUI	华为移动服务版本 20401300 以上，SDK 版本 push:2.6.3.301
Google FCM 推送	Android 4.1 及以上	手机端需安装 Google Play Services 且在中国大陆地区以外使用。
魅族推送	Flyme	使用魅族推送 push-internal:3.6.+
OPPO 推送	ColorOS	并非所有 OPPO 机型和版本都支持使用 OPPO 推送，SDK 版本 mcssdk-2.0.2.jar
vivo 推送	FuntouchOS	并非所有 vivo 机型和版本都支持使用 vivo 推送，SDK 版本 vivo_pushsdk_v2.3.1.jar

这里的离线是指在没有退出登录的情况下，应用被系统或者用户关闭。在这种情况下，如果还想收到 IM SDK 的消息提醒，可以集成即时通信 IM 离线推送。

注意：

- 对于已经退出登录（主动登出或者被踢下线）的用户，不会收到任何消息通知。
- 目前，离线推送只提供 普通聊天消息 进行消息提醒，暂不提供对 系统消息 的消息提醒。

IM SDK 离线推送基本配置

设置全局离线推送配置

IM SDK 提供了设置全局离线推送配置的功能，可以设置是否开启离线推送、收到离线推送时的提示声音等。这个设置方法是由 `TIMManager` 提供的 `setOfflinePushSettings`。

注意：

- 必须在登录成功后调用才生效。
- 目前仅支持 APNs 自定义提示音，声音文件需应用内置。

原型：

```
/**
* 初始化离线推送配置，需登录后设置才生效
* @param settings 离线推送配置信息
*/
public void setOfflinePushSettings(TIMOfflinePushSettings settings)

/**
```

```
 * 从服务器获取离线推送配置，需登录后才能获取
 * @param cb 回调，在 onSuccess 的参数中返回离线推送配置
 */
public void getOfflinePushSettings(final TIMValueCallBack<TIMOfflinePushSettings> cb)
```

参数说明：

参数	说明
settings	离线推送配置

TIMOfflinePushSettings 说明：

```
/**
 * 获取是否开启
 * @return true 表示开启， false 表示不开启
 */
public boolean isEnabled()

/**
 * 设置是否开启离线推送
 * @param enabled 是否开启离线推送
 */
public void setEnabled(boolean enabled)

/**
 * 获取收到 c2c 消息的离线推送时的提醒声音
 * @return 声音文件的 URI，没有设置时返回 null
 */
public Uri getC2cMsgRemindSound()

/**
 * 设置收到 c2c 消息的离线推送时的提醒声音
 * @param c2cMsgRemindSound 声音文件的 URI，恢复系统默认声音填 null
 */
public void setC2cMsgRemindSound(Uri c2cMsgRemindSound)

/**
 * 获取收到群消息的离线推送时的提醒声音
 * @return 声音文件的 URI，没有设置时返回 null
 */
public Uri getGroupMsgRemindSound()

/**
 * 设置收到群消息的离线推送时的提醒声音
 * @param groupMsgRemindSound 声音文件的 URI， 恢复系统默认声音填 null
 */
public void setGroupMsgRemindSound(Uri groupMsgRemindSound)
```

示例：

```
TIMOfflinePushSettings settings = new TIMOfflinePushSettings();
//开启离线推送
settings.setEnabled(true);
//设置收到 C2C 离线消息时的提示声音，以把声音文件放在 res/raw 文件夹下为例
settings.setC2cMsgRemindSound(Uri.parse("android.resource://" + getPackageName() + "/" + R.raw.dudulu));
//设置收到群离线消息时的提示声音，以把声音文件放在 res/raw 文件夹下为例
settings.setGroupMsgRemindSound(Uri.parse("android.resource://" + getPackageName() + "/" + R.raw.dudulu));

TIMManager.getInstance().setOfflinePushSettings(settings);
```

针对单条消息设置离线推送

IM SDK 提供针对单独每一条消息进行离线推送配置的功能。开发者可以针对指定的某一条消息设置是否开启离线推送、收到离线推送后提醒声音、离线推送消息描述及扩展字段等。

注意：

- 针对单条消息设置的离线推送配置优先级是最高的，也就是在同时设置了全局离线推送配置及单条消息离线推送配置的情况下，将以单条消息离线推送配置为准。
- 目前仅支持 APNs 自定义提示音，声音文件需应用内置。

原型：

```
/**
 * 设置当前消息在对方收到离线推送的时候的配置（可选，发送消息时设置）
 * @param settings 离线推送配置
 */
public void setOfflinePushSettings(TIMMessageOfflinePushSettings settings)

/**
 * 获取当前消息的离线推送配置
 * @return 离线推送配置，如果发送方没有设置的，返回 null
 */
public TIMMessageOfflinePushSettings getOfflinePushSettings()
```

TIMMessageOfflinePushSettings ：

```
/**
 * 离线 Push 展示标题，针对 iOS 和 Android 平台都生效，如果您需要分平台单独设置，请设置 IOSSettings -> title 和 AndroidSettings -> title
 *
 * @param title 通知栏标题
 * @return
 */
public TIMMessageOfflinePushSettings setTitle(String title)

/**
 * 离线 Push 展示文本，针对 iOS 和 Android 平台都生效，如果您需要分平台单独设置，请设置 IOSSettings -> desc 和 AndroidSettings -> desc
 * @param descr 正文内容
 */
public TIMMessageOfflinePushSettings setDescr(String descr)

/**
 * 获取当前消息的离线推送展示正文内容
 * @return 正文内容
 */
public String getDescr()

/**
 * 设置当前消息的扩展字段（可选，发送消息的时候设置）
 * @param ext 扩展字段内容
 */
public TIMMessageOfflinePushSettings setExt(byte[] ext)

/**
 * 获取当前消息的扩展字段
 * @return 扩展字段内容，没有设置返回 null
 */
public byte[] getExt()

/**
 * 设置当前消息是否允许离线推送，默认允许推送（可选，发送消息时设置）
 * @param enabled true 表示允许离线推送， false 表示不允许离线推送
 */
public TIMMessageOfflinePushSettings setEnabled(boolean enabled)

/**
 * 获取当前消息是否允许推送
 * @return 是否允许推送标识， true 表示允许推送， false 表示不允许推送
 */
public boolean isEnabled()
```

```
/**
 * 获取当前消息在 Android 设备上的离线推送配置
 * @return Android 设备上的离线推送配置
 */
public AndroidSettings getAndroidSettings()

/**
 * 设置当前消息在 Android 设备上的离线推送配置（可选，发送消息时设置）
 * @param androidSettings 当前消息在 Android 设备上的离线推送配置
 */
public TIMMessageOfflinePushSettings setAndroidSettings(AndroidSettings androidSettings)

/**
 * 获取当前消息在 iOS 设备上的离线推送配置
 * @return iOS 设备上的离线推送配置
 */
public IOSSettings getIosSettings()

/**
 * 设置当前消息在 iOS 设备上的离线推送配置（可选，发送消息时设置）
 * @param iosSettings 当前消息在 iOS 设备上的离线推送配置
 */
public TIMMessageOfflinePushSettings setIosSettings(IOSSettings iosSettings)
```

TIMMessageOfflinePushSettings.AndroidSettings：

```
/**
 * 获取通知标题
 * @return 通知标题
 */
public String getTitle()

/**
 * 设置离线 Push 展示标题

 * @param title 通知标题
 */
public AndroidSettings setTitle(String title)

/**
 * 设置离线 Push 展示自定义文本
 *
 * @param desc 通知显示内容
 */
public AndroidSettings setDesc(String desc)

/**
 * 获取当前消息在 Android 设备上的离线推送提示声音 URI
 * @return 声音 URI，没有设置则返回 null
 */
public Uri getSound()

/**
 * 设置当前消息在 Android 设备上的离线推送提示声音（可选，发送消息时设置）
 * @param sound 声音 URI，仅支持应用内部的声音资源文件
 */
public AndroidSettings setSound(Uri sound)

/**
 * 获取当前消息的通知模式
 * @return 通知模式
 */
public NotifyMode getNotifyMode()

/**
 * 设置当前消息在对方收到离线推送时候的通知模式（待废弃，可以不设置）。
 *
 * @param mode 通知模式，默认为普通通知栏消息模式
```

```
 */
public AndroidSettings setNotifyMode(NotifyMode mode)
```

TIMMessageOfflinePushSettings.NotifyMode：

```
/**
 * 普通通知栏消息模式，离线消息下发后，单击通知栏消息直接启动应用，不会给应用进行回调
 */
NotifyMode.Normal
```

TIMMessageOfflinePushSettings.IOSSettings：

```
/**
 * 设置离线 Push 展示标题
 *
 * @param title 通知标题
 */
public IOSSettings setTitle(String title)

/**
 * 设置离线 Push 展示自定义文本
 *
 * @param desc
 */
public IOSSettings setDesc(String desc)

/**
 * 获取当前消息在 iOS 设备上的离线推送提示声音
 *
 * @return 声音文件路径，没有设置则返回 null
 */
public String getSound()

/**
 * 设置当前消息在 iOS 设备上的离线推送提示声音（可选，发送消息时设置）
 *
 * @param sound 声音文件路径，当设置为{@see IOSSettings#NO_SOUND_NO_VIBRATION}时表示无提示音无振动
 */
public void setSound(String sound)

/**
 * 获取当前消息是否开启 Badge 计数
 *
 * @return true 表示当前消息开启 Badge 计数
 */
public boolean isBadgeEnabled()

/**
 * 设置当前消息是否开启 Badge 计数，默认开启（可选，发送消息时设置）
 *
 * @param badgeEnabled 否开启 Badge 计数
 */
public IOSSettings setBadgeEnabled(boolean badgeEnabled)
```

示例：

```
// 构造一条消息
TIMMessage msg = new TIMMessage();

// 添加文本内容
TIMTextElem elem = new TIMTextElem();
elem.setText("a new msg from " + selfId);
if(msg.addElement(elem) != 0) {
Log.d(tag, "addElement failed");
return;
}
```

```
// 设置当前消息的离线推送配置
TIMMessageOfflinePushSettings settings = new TIMMessageOfflinePushSettings();
settings.setEnabled(true);
// 设置 iOS 和 Android 通知栏消息的标题和内容。如果想要两个平台通知栏展示的标题和内容不同，可以通过 AndroidSettings 和 IOSSettings 分别设置。
settings.setTitle("I'm title");
settings.setDescr("I'm description");
// 设置离线推送扩展信息
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();
}

// 设置在 Android 设备上收到消息时的离线配置
TIMMessageOfflinePushSettings.AndroidSettings androidSettings = new TIMMessageOfflinePushSettings.AndroidSettings();
// IM SDK 2.5.3之前的构造方式
// TIMMessageOfflinePushSettings.AndroidSettings androidSettings = settings.new AndroidSettings();
// 设置 Android 通知栏消息的标题和内容
// androidSettings.setTitle("I'm title for android");
// androidSettings.setDesc("I'm desc for android");
// 设置 Android 设备收到消息时的提示音，声音文件需要放置到 raw 文件夹
androidSettings.setSound(Uri.parse("android.resource://" + getPackageName() + "/" +R.raw.hualala));
settings.setAndroidSettings(androidSettings);

//设置在 iOS 设备上收到消息时的离线配置
TIMMessageOfflinePushSettings.IOSSettings iosSettings = new TIMMessageOfflinePushSettings.IOSSettings();
//IM SDK 2.5.3之前的构造方式
//TIMMessageOfflinePushSettings.IOSSettings iosSettings = settings.new IOSSettings();
// 设置 iOS 通知栏消息的标题和内容
// iosSettings.setTitle("I'm title for iOS");
// iosSettings.setDesc("I'm desc for iOS");

// 开启 Badge 计数
iosSettings.setBadgeEnabled(true);
// 设置 iOS 收到消息时没有提示音且不振动（IM SDK 2.5.3新增特性）
//iosSettings.setSound(TIMMessageOfflinePushSettings.IOSSettings.NO_SOUND_NO_VIBRATION);
// 设置 iOS 设备收到离线消息时的提示音
iosSettings.setSound("/path/to/sound/file");

msg.setOfflinePushSettings(settings);

// 获取一个单聊会话
TIMConversation conversation = TIMManager.getInstance().getConversation(
TIMConversationType.C2C, // 会话类型：单聊
peer); // 会话对方用户帐号

// 发送消息
conversation.sendMessage(msg, new TIMValueCallBack<TIMMessage>() {// 发送消息回调
@Override
public void onError(int code, String desc) {// 发送消息失败
// 错误码 code 和错误描述 desc，可用于定位请求失败原因
// 错误码 code 列表请参见错误码表
Log.e(tag, "send message failed. code: " + code + " errmsg: " + desc);
}

@Override
public void onSuccess(TIMMessage msg) {//发送消息成功
Log.d(tag, "SendMsg ok! peer:" + peer );
}
});
```

离线推送（小米）

最近更新时间：2020-03-20 09:29:59

离线推送流程

实现离线消息推送的过程如下：

1. 开发者到厂商的平台注册账号，并通过开发者认证后，申请开通推送服务。
2. 创建推送服务，并绑定应用信息，获取推送证书、密码、密钥等信息。
3. 登录 即时通信 IM 控制台 填写推送证书及相关信息，即时通信 IM 服务端会为每个证书生成不同的证书 ID。
4. 将厂商提供的推送 SDK 集成到开发者的项目工程中，并按各厂商的要求进行配置。
5. 集成即时通信 IM SDK 到项目后，将证书 ID、设备信息等上报至即时通信 IM 服务端。
6. 当客户端 App 在即时通信 IM 没有退出登录的情况下，被系统或者用户 kill 时，即时通信 IM 服务端将通过消息推送进行提醒。

配置离线推送

MIUI 为深度定制 Android 系统，对于第三方 App 自启动权限管理很严格，默认情况下第三方 App 都不会在系统的自启动白名单内，App 在后台时容易被系统 kill，因此推荐在小米设备上集成小米推送 MiPush，MiPush 是 MIUI 的系统级服务，推送到达率较高。目前，**即时通信 IM 仅支持小米推送的通知栏消息。**

> 注意：
>
> - 此指引文档是直接参考小米官方文档所写，若小米推送有变动，请以 小米推送官网文档 为准。
> - 如果不需要对小米设备做专门的离线推送适配，可以忽略此章节。

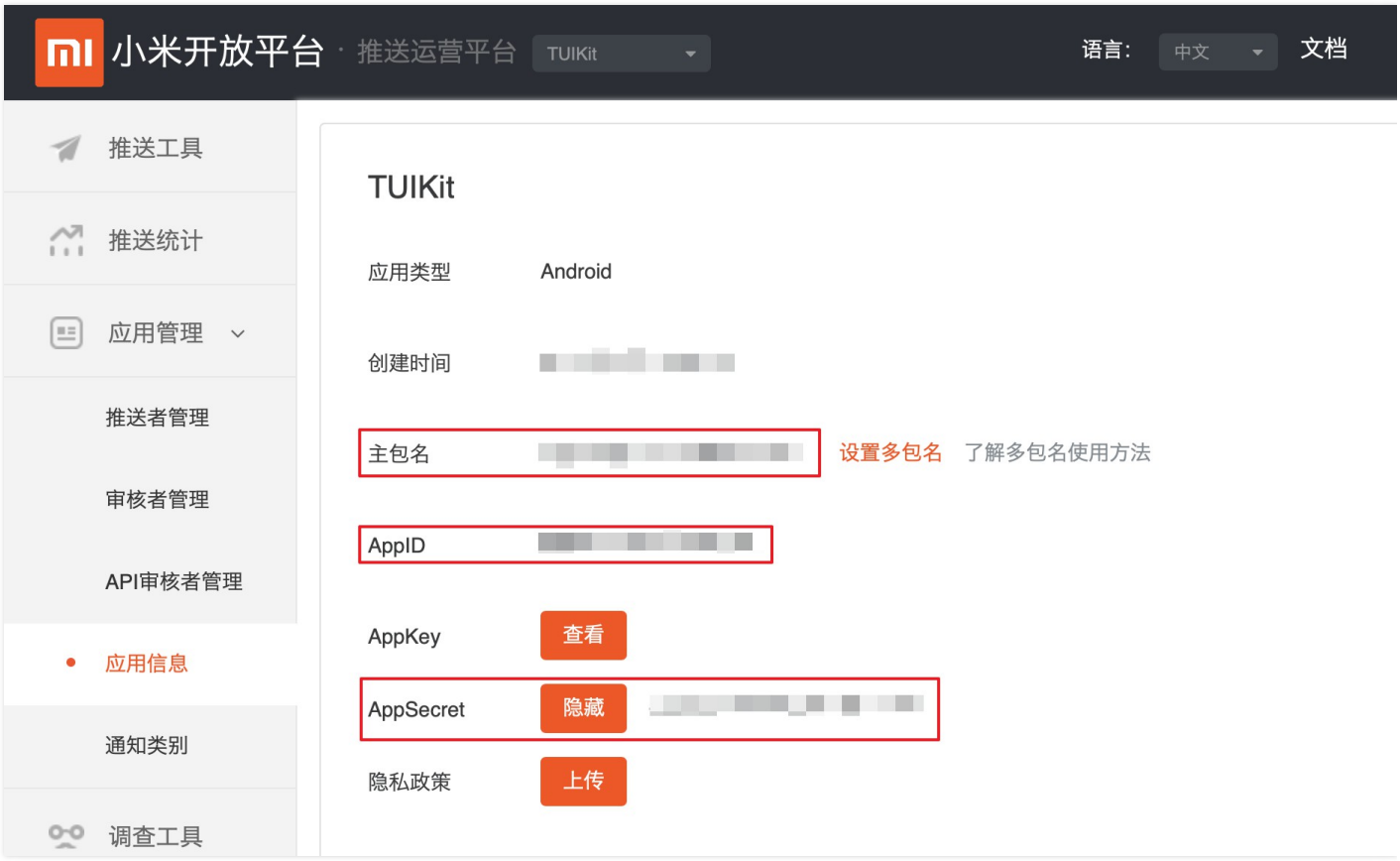
步骤1：申请小米推送证书

1. 打开 小米开放平台官网 进行注册并通过开发者认证。

> 说明：
> 认证过程大约需要2天左右，请务必提前阅读 小米推送服务启用指南，以免影响您的接入进度。

2. 登录小米开放平台的管理控制台，选择【应用服务】>【PUSH服务】，创建小米推送服务应用。
   小米推送服务应用创建完成后，在应用详情中，您可以查看详细的应用信息。

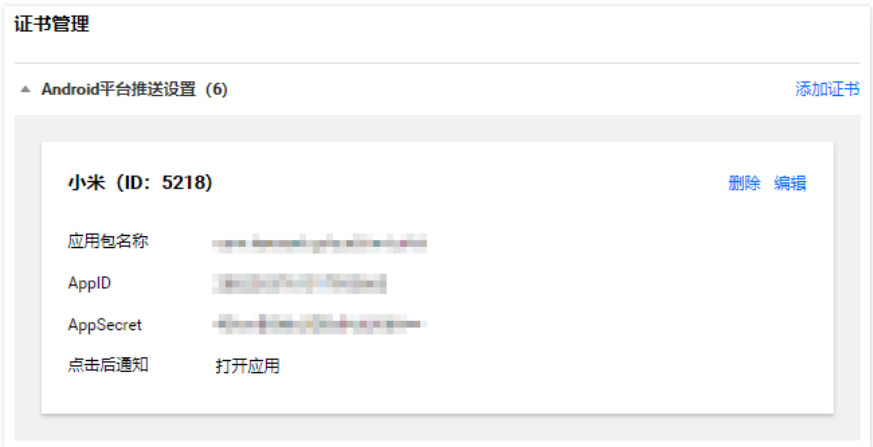
3. 记录 **主包名** 、 **AppID** 、 **AppSecret** 信息。



步骤2：托管证书信息到即时通信 IM

1. 登录腾讯云 即时通信 IM 控制台，单击目标应用卡片，进入应用的基础配置页面。
2. 单击【Android平台推送设置】区域的【添加证书】。

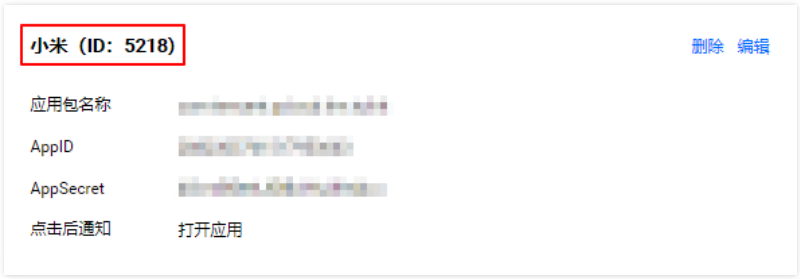
> 说明：
> 如果您原来已有证书只需变更信息，可以单击【Android平台推送设置】区域的【编辑】进行修改更新。



3. 根据 步骤1 中获取的信息设置以下参数：
   - **推送平台**：选择**小米**
   - **应用包名称**：填写小米推送服务应用的**主包名**
   - **AppID**：填写小米推送服务应用的 **AppID**

- **AppSecret**：填写小米推送服务应用的 **AppSecret**
- **点击通知后**：选择点击通知栏消息后的响应操作， 支持**打开应用**、**打开网页**和**打开应用内指定界面**，更多详情请参见 配置点击通知栏消息事件
  当设置为【打开应用】或【打开应用内指定界面】操作时，支持 透传自定义内容。

4. 单击【确认】保存信息，证书信息保存后10分钟内生效。
5. 待推送证书信息生成后，记录证书的 **ID**。



步骤3：集成推送 SDK

说明：

- 即时通信 IM 默认推送的通知标题为 a new message。
- 阅读此小节前，请确保您已经正常集成并使用即时通信 IM SDK。
- 您可以在我们的 demo 里找到小米推送的实现示例，请注意：小米推送版本更新时有可能会有功能调整，若您发现本节内容存在差异，烦请您及时查阅 小米推送官网文档，并将文档信息差异反馈给我们，我们会及时跟进修改。

步骤3.1：下载小米推送 SDK 并添加引用

1. 访问 小米推送运营平台 下载小米推送 SDK。
2. 解压小米推送 SDK，获取 MiPush_SDK_client_**.jar 库文件。
3. 将 MiPush_SDK_client_**.jar 库文件添加到您项目的 libs 目录下，并且在项目中添加引用。

步骤3.2：配置 AndroidManifest.xml 文件

添加小米推送必须的权限：

```
<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"/>

<!--这里的 com.tencent.qcloud.tim.tuikit 改成您的 App 的包名-->
<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" />
<!--这里的 com.tencent.qcloud.tim.tuikit 改成您的 App 的包名-->
```

配置小米推送服务需要的 service 和 receiver：

```
<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" /> <!--注：此 service 必须在3.0.1版本以后（包括3.0.1版本）加入-->
<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" /> <!--注：此 service 必须在2.2.5版本以后（包括2.2.5版本）加入-->

<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>
```

步骤3.3：自定义一个 BroadcastReceiver 类

为了接收消息，您需要自定义一个继承自 `PushMessageReceiver` 类的 BroadcastReceiver，并实现其中的 `onReceiveRegisterResult` 方法，然后将此 receiver 注册到 AndroidManifest.xml 中。

以下为 Demo 中的示例代码：

```
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.getReason());

if (MiPushClient.COMMAND_REGISTER.equals(command)) {
if (miPushCommandMessage.getResultCode() == ErrorCode.SUCCESS) {
mRegId = cmdArg1;
}
}

Log.d(TAG, "regId: " + mRegId);
ThirdPushTokenMgr.getInstance().setThirdPushToken(mRegId); // regId 在此处传入，后续推送信息上报时需要使用
ThirdPushTokenMgr.getInstance().setPushTokenToTIM();
}

}
```

将自定义的 BroadcastReceiver 注册到 AndroidManifest.xml：

```
<!--这里的 com.tencent.qcloud.uipojo.thirdpush.XiaomiMsgReceiver 修改成您 App 中的完整类名-->
<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>
```

步骤3.4：在 App 中注册小米推送服务

如果您选择启用小米离线推送，需要向小米服务器注册推送服务，通过调用 MiPushClient.registerPush 来对小米推送服务进行初始化 MiPushClient.registerPush 可在任意地方调用，为了提高注册成功率，小米官方建议在 Application 的 onCreate 中调用。
注册成功后，您将在 步骤3.3 自定义的 BroadcastReceiver 的 onReceiveRegisterResult 中收到注册结果。其中 regId 为当前设备上当前 App 的唯一标识，请记录 regId 信息。

以下为 Demo 中的示例代码：

```
public class DemoApplication extends Application {

private static DemoApplication instance;

@Override
public void onCreate() {
super.onCreate();
//判断是否是在主线程
if (SessionWrapper.isMainProcess(getApplicationContext())) {
/**
* TUIKit的初始化函数
*
* @param context 应用的上下文，一般为对应应用的 ApplicationContext
* @param sdkAppID 您在腾讯云注册应用时分配的 SDKAppID
* @param configs TUIKit 的相关配置项，一般使用默认即可，需特殊配置参考 API 文档
*/
long current = System.currentTimeMillis();
TUIKit.init(this, Constants.SDKAPPID, BaseUIKitConfigs.getDefaultConfigs());
System.out.println(">>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));
//添加自定初始化配置
customConfig();
System.out.println(">>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));
```

```
if(IMFunc.isBrandXiaoMi()){
// 小米离线推送
MiPushClient.registerPush(this, Constants.XM_PUSH_APPID, Constants.XM_PUSH_APPKEY);
}
if(IMFunc.isBrandHuawei()){
// 华为离线推送
HMSAgent.init(this);
}
if(MzSystemUtils.isBrandMeizu(this)){
// 魅族离线推送
PushManager.register(this, Constants.MZ_PUSH_APPID, Constants.MZ_PUSH_APPKEY);
}
if(IMFunc.isBrandVivo()){
// vivo 离线推送
PushClient.getInstance(getApplicationContext()).initialize();
}
}
instance = this;
}
}
```

步骤4：上报推送信息至即时通信 IM 服务端

若您需要通过小米推送进行即时通信 IM 消息的推送通知，必须在**用户登录成功后**通过 TIMManager 中的 setOfflinePushToken 方法将您托管到即时通信 IM 控制台生成的**证书 ID** 及小米推送服务端生成的 **regId** 上报到即时通信 IM 服务端。

> 注意：
> 正确上报 regId 与证书 ID 后，即时通信 IM 服务才能将用户与对应的设备信息绑定，从而使用小米推送服务进行推送通知。

以下为 Demo 中的示例代码：

- 定义证书 ID 常量：

```
/**
* 我们先定义一些常量信息在 Constants.java
*/
/****** 小米离线推送参数 start ******/
// 使用您在即时通信 IM 控制台上小米推送证书信息里的证书 ID
public static final long XM_PUSH_BUZID = 6666;
// 小米开放平台分配的应用 APPID 及 APPKEY
public static final String XM_PUSH_APPID = "1234512345123451234";
public static final String XM_PUSH_APPKEY = "1234512345123";
/****** 小米离线推送参数 end ******/
```

- 上报推送的证书 ID 及 regId：

```
/**
* 在 ThirdPushTokenMgr.java 中对推送的证书 ID 及设备信息进行上报操作
*/
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; // regId 在此处传值，结合上文自定义 BroadcastReciever 类文档说明
```

```
}

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()){ // 判断厂商品牌，根据不同厂商选择不同的推送服务
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;
}
});
}
}
```

步骤5：离线推送

成功上报证书 ID 及 regId 后，即时通信 IM 服务端会在该设备上的即时通信 IM 用户 logout 之前、App 被 kill 之后将消息通过小米推送通知到用户端。

说明：

- 小米推送并非100%必达。
- 小米推送可能会有一定延时，通常与 App 被 kill 的时机有关，部分情况下与小米推送服务有关。
- 若即时通信 IM 用户已经 logout 或被即时通信 IM 服务端主动下线（例如在其他端登录被踢等情况），则该设备上不会再收到消息推送。

配置点击通知栏消息事件

您可以选择点击通知栏消息后**打开应用**、**打开网页**或**打开应用内指定界面**。

打开应用

设置为点击通知栏消息打开应用时，会回调小米的 onNotificationMessageClicked 方法，App 可以在此方法中自行处理打开应用。

打开网页

您需要在 添加证书 时选择【打开网页】并输入以 http:// 或 https:// 开头的网址，例如 https://cloud.tencent.com/document/product/269 。

打开应用内指定界面

1. 在 manifest 中配置需要打开的 Activity 的 intent-filter ，示例代码如下：

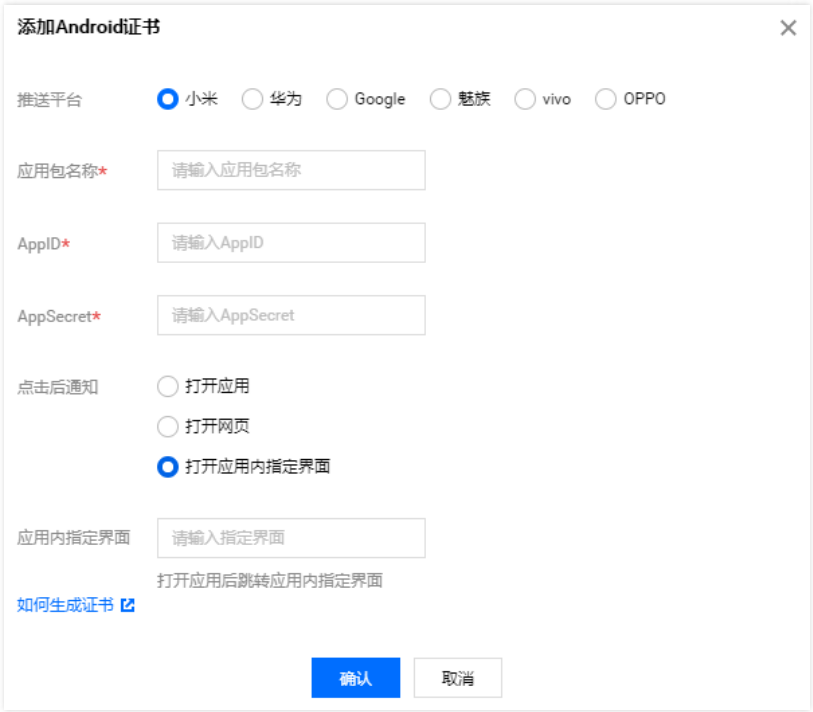
```
<activity
android:name="com.tencent.qcloud.tim.demo.chat.ChatActivity"
android:launchMode="singleTask"
android:screenOrientation="portrait"
android:windowSoftInputMode="adjustResize|stateHidden">
```

```
<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. 获取 intent URL，方式如下：

```
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);

// 打印结果
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. 在 添加证书 时选择【打开应用内指定界面】并输入上述打印结果。



透传自定义内容

添加证书 时设置【点击通知后】为【打开应用】或【打开应用内指定界面】操作才支持透传自定义内容。

步骤1：发送端设置自定义内容

在发消息前设置每条消息的通知栏自定义内容。

- Android 端示例：

```
String extContent = "ext content";
TIMMessageOfflinePushSettings settings = new TIMMessageOfflinePushSettings();
```

```
settings.setExt(extContent.getBytes());
timMessage.setOfflinePushSettings(settings);
mConversation.sendMessage(false, timMessage, callback);
```

- 服务端示例请参见 OfflinePushInfo 的格式示例

步骤2：接收端获取自定义内容

- 若 添加证书 时设置【点击通知后】的操作为【打开应用】，当点击通知栏的消息时，会触发小米推送 SDK 的 `onNotificationMessageClicked(Context context, MiPushMessage miPushMessage)` 回调，自定义内容可以从 `miPushMessage` 中获取。

```
Map extra = miPushMessage.getExtra();
String extContent = extra.get("ext");
```

- 若 添加证书 时设置【点击通知后】的操作为【打开应用内指定界面】，封装消息的 `MiPushMessage` 对象通过 `Intent` 传到客户端，客户端在相应的 `Activity` 中获取自定义内容。

```
```
Bundle bundle = getIntent().getExtras();
MiPushMessage miPushMessage = (MiPushMessage)bundle.getSerializable(PushMessageHelper.KEY_MESSAGE);
Map extra = miPushMessage.getExtra();
String extContent = extra.get("ext");
```
```

常见问题

如果应用使用了混淆，如何防止小米离线推送功能异常？

如果您的应用使用了混淆，为了防止小米离线推送功能异常，您需要 keep 自定义的 BroadcastReceiver，参考添加以下混淆规则：

> 说明：
> 以下代码为小米官方示例，请根据实际情况修改后再使用。

```
# 请将 com.tencent.qcloud.tim.demo.thirdpush.XiaomiMsgReceiver 改成您 App 中定义的完整类名
-keep com.tencent.qcloud.tim.demo.thirdpush.XiaomiMsgReceiver {*;}
# 如果编译使用的 Android 版本是23，添加这个可以防止一个误报的 warning 导致无法成功编译
-dontwarn com.xiaomi.push.**
```

能否自定义配置推送提示音？

目前小米推送不支持自定义的提示音。

收不到推送时，如何排查问题？

1. 任何推送都不是100%必达，厂商推送也不例外。因此，若在快速、连续的推送过程中偶现一两条推送未通知提醒，通常是由厂商推送频控的限制引起。
2. 按照推送的流程，确认小米推送证书信息是否正确配置在 即时通信 IM 控制台 中。
3. 确认您的项目 集成小米推送 SDK 的配置正确，并正常获取到了 regId。
4. 确认您已将正确的 推送信息上报 至即时通信 IM 服务端。
5. 在设备中手动 kill App，发送若干条消息，确认是否能在一分钟内接收到通知。

离线推送（华为）

最近更新时间：2020-06-22 16:04:12

离线推送流程

实现离线消息推送的过程如下：

1. 开发者到厂商的平台注册账号，并通过开发者认证后，申请开通推送服务。
2. 创建推送服务，并绑定应用信息，获取推送证书、密码、密钥等信息。
3. 登录 即时通信 IM 控制台 填写推送证书及相关信息，即时通信 IM 服务端会为每个证书生成不同的证书 ID。
4. 将厂商提供的推送 SDK 集成到开发者的项目工程中，并按各厂商的要求进行配置。
5. 集成即时通信 IM SDK 到项目后，将证书 ID、设备信息等上报至即时通信 IM 服务端。
6. 当客户端 App 在即时通信 IM 没有退出登录的情况下，被系统或者用户 kill 时，即时通信 IM 服务端将通过消息推送进行提醒。

配置离线推送

华为 EMUI 是一款深度定制的 Android 系统，后台策略严格，默认情况下第三方 App 不具有自启动权限，App 在后台时很容易被系统强制 kill，因此推荐在华为设备上集成华为推送服务，华为推送服务是华为消息服务（HMS）的一部分，在 EMUI 中属于系统级服务，推送到达率相比第三方更高。目前，**即时通信 IM 仅支持华为推送的通知栏消息。**

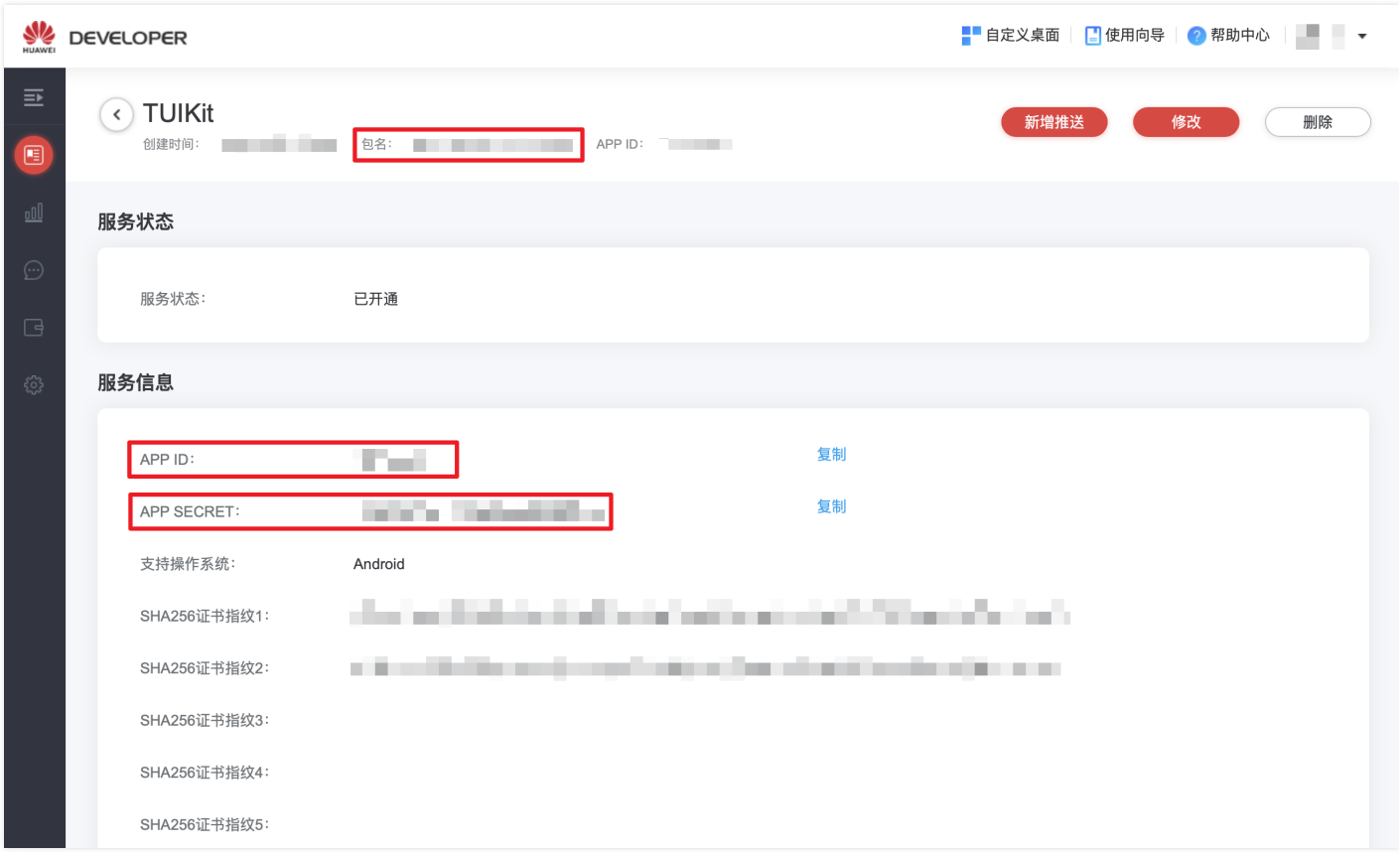
注意：

- 此指引文档是直接参考华为推送官方文档所写，若华为推送有变动，请以 华为推送官网 为准。
- 如果不需要对华为设备做专门的离线推送适配，可以忽略此章节。

步骤1：申请华为推送证书

1. 打开 华为开发者联盟官网 进行注册并通过开发者认证。
2. 登录华为开发者联盟的管理控制台，选择【应用服务】>【开发服务】>【PUSH】，创建华为推送服务应用。
   华为推送在申请 PUSH 服务时，需要您提供应用签名证书的 SHA256 指纹，最多允许添加5个。 华为推送服务应用创建完成后，在应用详情中，您可以查看详细的应用信息。

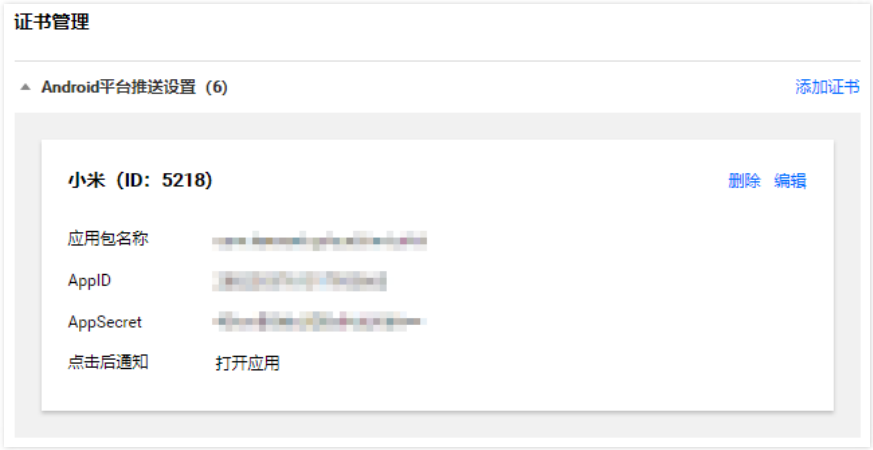
3. 记录**包名**、**APP ID**、**APP Secret**信息。



步骤2：托管证书信息到即时通信 IM

1. 登录腾讯云 即时通信 IM 控制台，单击目标应用卡片，进入应用的基础配置页面。
2. 单击【Android平台推送设置】区域的【添加证书】。

> 说明：
> 如果您原来已有证书只需变更信息，可以单击【Android平台推送设置】区域的【编辑】进行修改更新。



3. 根据 步骤1 中获取的信息设置以下参数：
   - **推送平台**：选择**华为**
   - **应用包名称**：填写华为推送服务应用的**包名**
   - **AppID**：填写华为推送服务应用的 **APP ID**

- **AppSecret**：填写华为推送服务应用的 **APP SECRET**
- **角标参数**：填写应用入口完整 Activity 类名，用作华为桌面应用角标显示，请参考 华为桌面角标开发指导书
- **点击通知后**：选择点击通知栏消息后的响应操作，支持**打开应用**、**打开网页**和**打开应用内指定界面**，更多详情请参见 配置点击通知栏消息事件
  当设置为【打开应用】或【打开应用内指定界面】操作时，支持 透传自定义内容。

4. 单击【保存】保存信息，证书信息保存后10分钟内生效。
5. 待推送证书信息生成后，记录证书的 **ID**。

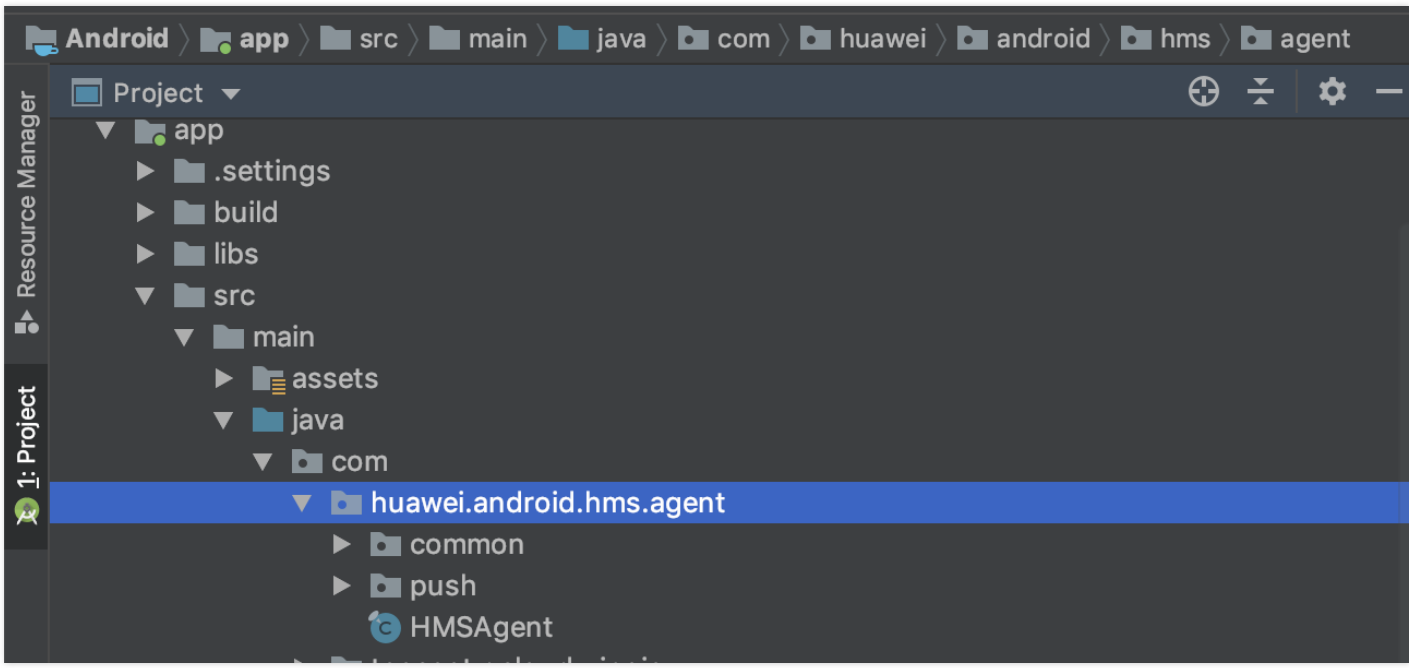
步骤3：集成推送 SDK

说明：

- 即时通信 IM 默认推送的通知标题为 a new message 。
- 阅读此小节前，请确保您已经正常集成并使用即时通信 IM SDK。
- 您可以在我们的 demo 里找到华为推送的实现示例，请注意：华为推送版本更新时有可能会有功能调整，若您发现本节内容存在差异，烦请您及时查阅 华为推送官网，并将文档信息差异反馈给我们，我们会及时跟进修改。

步骤3.1：下载华为推送 SDK 并添加引用

1. 访问 华为推送官网 下载**HMS Agent 套件**。
2. 解压 HMS Agent 套件。
3. 将 hmsagents\src\main\java 文件夹内的文件拷贝到您项目的 src\main\java 目录中。



4. 通过 Gradle 进行华为推送 SDK 集成，在您项目的 build.gradle 里添加以下代码：

```
allprojects {
repositories {
jcenter()
maven {url 'http://developer.huawei.com/repo/'}
}
}
```

5. 在子项目的 build.gradle 里添加以下信息：

```
dependencies {
// 华为推送 SDK，2.6.3.301可替换成实际所需要的版本
implementation 'com.huawei.android.hms:push:2.6.3.301'
// 如果碰到报错：com.huawei.hms.api 不存在，这条也需要加上，注意版本号必须相同
// implementation 'com.huawei.android.hms:base:2.6.3.301'
}
```

步骤3.2：配置 AndroidManifest.xml 文件

1. 添加华为推送的必要权限：

```
<!--HMS-SDK 引导升级 HMS 功能，访问 OTA 服务器需要网络权限-->
<uses-permission android:name="android.permission.INTERNET" />
<!--HMS-SDK 引导升级 HMS 功能，保存下载的升级包需要 SD 卡写权限-->
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<!--检测网络状态-->
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<!--检测 Wi-Fi 状态-->
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
```

```
<!--为了获取用户手机的 IMEI，用来唯一的标识用户。-->
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>

<!--如果是Android 8.0，应用编译配置的 targetSdkVersion>=26，请务必添加以下权限 -->
<uses-permission android:name="android.permission.REQUEST_INSTALL_PACKAGES" />

<!--这里的 com.tencent.qcloud.tim.tuikit 改成您的 App 的包名-->
<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" />
<!--这里的 com.tencent.qcloud.tim.tuikit 改成您的 App 的包名-->
```

2. 在 application 下增加以下内容，具体说明请参见 华为推送官网。

```
<meta-data
android:name="com.huawei.hms.client.appid"
android:value="appid=1234567890"/> <!--这里的 appid 值修改为您的华为推送 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>
```

步骤3.3：自定义一个 BroadcastReceiver 类

为了接收消息，您需要自定义一个继承自 `PushReceiver` 类的 BroadcastReceiver，我们只需要实现其中的 `onToken` 方法，然后将此 receiver 注册到 AndroidManifest.xml 中。

以下为 Demo 中的示例代码：

```
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); // token 在此处传入，后续推送信息上报时需要使用
ThirdPushTokenMgr.getInstance().setPushTokenToTIM();
}
}
```

将自定义的 BroadcastReceiver 注册到 AndroidManifest.xml：

```
<!--这里的 com.tencent.qcloud.tim.demo.thirdpush.HUAWEIPushReceiver 修改成您 App 中的完整类名-->
<receiver android:name="com.tencent.qcloud.tim.demo.thirdpush.HUAWEIPushReceiver"
android:permission="com.tencent.qcloud.tim.tuikit.permission.PROCESS_PUSH_MSG">
<intent-filter>
<!-- 必须,用于接收 token -->
<action android:name="com.huawei.android.push.intent.REGISTRATION" />
<!-- 必须, 用于接收透传消息 -->
<action android:name="com.huawei.android.push.intent.RECEIVE" />
<!-- 必须, 用于接收通知栏消息点击事件 此事件不需要开发者处理，只需注册就可以-->
<action android:name="com.huawei.intent.action.PUSH_DELAY_NOTIFY"/>
</intent-filter>
</receiver>
```

步骤3.4：在 App 中注册华为推送服务

如果您选择启用华为推送服务，需要在 Application 的 `onCreate` 中调用 `HMSAgent.init` 来对华为推送服务进行初始化。

注册成功后，您将在 步骤3.3 自定义的 BroadcastReceiver 的 `onToken` 中收到注册结果。其中 `token` 为当前设备上当前 App 的唯一标识，请记录 `token` 信息。

以下为 Demo 中的示例代码：

```
public class DemoApplication extends Application {

private static PojoApplication instance;

@Override
public void onCreate() {
super.onCreate();
// 判断是否是在主线程
if (SessionWrapper.isMainProcess(getApplicationContext())) {
/**
* TUIKit 的初始化函数
*
* @param context 应用的上下文，一般为对应应用的 ApplicationContext
* @param sdkAppID 您在腾讯云注册应用时分配的 SDKAppID
* @param configs TUIKit 的相关配置项，一般使用默认即可，需特殊配置参考 API 文档
*/
long current = System.currentTimeMillis();
TUIKit.init(this, Constants.SDKAPPID, BaseUIKitConfigs.getDefaultConfigs());
System.out.println(">>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));
// 添加自定初始化配置
customConfig();
System.out.println(">>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));

if(IMFunc.isBrandXiaoMi()){
// 小米离线推送
MiPushClient.registerPush(this, Constants.XM_PUSH_APPID, Constants.XM_PUSH_APPKEY);
}
if(IMFunc.isBrandHuawei()){
// 华为离线推送
HMSAgent.init(this);
}
if(MzSystemUtils.isBrandMeizu(this)){
```

```
// 魅族离线推送
PushManager.register(this, Constants.MZ_PUSH_APPID, Constants.MZ_PUSH_APPKEY);
}
if(IMFunc.isBrandVivo()){
// vivo 离线推送
PushClient.getInstance(getApplicationContext()).initialize();
}
}
instance = this;
}
}
```

在主界面中获取 token：

```
if (IMFunc.isBrandHuawei()) {
// 华为离线推送
HMSAgent.connect(this, new ConnectHandler() {
@Override
public void onConnect(int rst) {
QLog.i(TAG, "huawei push HMS connect end:" + rst);
}
});
getHuaWeiPushToken();
}
```

步骤4：上报推送信息至即时通信 IM 服务端

若您需要通过华为推送进行即时通信 IM 消息的推送通知，必须在**用户登录成功后**通过 TIMManager 中的 setOfflinePushToken 方法将您托管到即时通信 IM 控制台生成的**证书 ID** 及华为推送服务返回的 **token** 上报到即时通信 IM 服务端。

> 注意：
> 正确上报 token 与证书 ID 后，即时通信 IM 服务才能将用户与对应的设备信息绑定，从而使用华为推送服务进行推送通知。

以下为 Demo 中的示例代码：

- 定义证书 ID 常量

```
/**
* 我们先定义一些常量信息在 Constants.java
*/
/****** 华为离线推送参数 start ******/
// 使用您在即时通信 IM 控制台上华为推送证书信息里的证书 ID
public static final long HW_PUSH_BUZID = 6666;
// 华为开发者联盟给应用分配的应用 APPID
public static final String HW_PUSH_APPID = "1234567890"; // 见清单文件
/****** 华为离线推送参数 end ******/
```

- 上报推送的证书 ID 及 token

```
/**
* 在 ThirdPushTokenMgr.java 中对推送的证书 ID 及设备信息进行上报操作
*/
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 在此处传值，结合上文自定义 BroadcastReciever 类文档说明
}

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()){ // 判断厂商品牌，根据不同厂商选择不同的推送服务
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;
}
});
}
}
```

步骤5：离线推送

成功上报证书 ID 及 token 后，即时通信 IM 服务端会在该设备上的即时通信 IM 用户 logout 之前、App 被 kill 之后，将消息通过华为推送通知到用户端。

说明：

- 华为推送并非100%必达。
- 华为推送可能会有一定延时，通常与 App 被 kill 的时机有关，部分情况下与华为推送服务有关。
- 若即时通信 IM 用户已经 logout 或被即时通信 IM 服务端主动下线（例如在其他端登录被踢等情况），则该设备上不会再收到消息推送。

配置点击通知栏消息事件

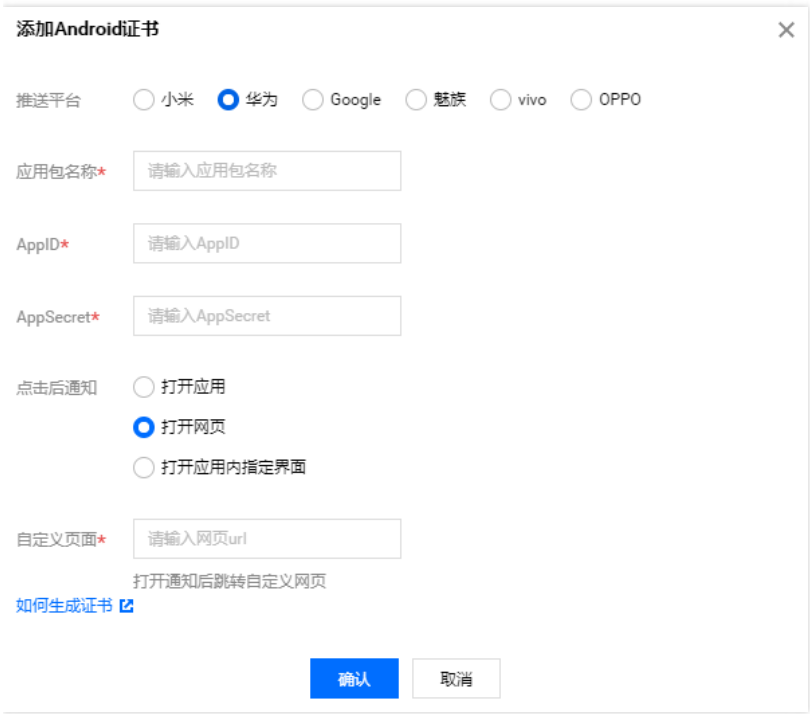
您可以选择点击通知栏消息后**打开应用**、**打开网页**或**打开应用内指定界面**。

打开应用

默认为点击通知栏消息打开应用。

打开网页

您需要在 添加证书 时选择【打开网页】并输入以 http:// 或 https:// 开头的网址，例如 https://cloud.tencent.com/document/product/269 。



打开应用内指定界面

1. 在 manifest 中配置需要打开的 Activity 的 intent-filter ，示例代码如下：

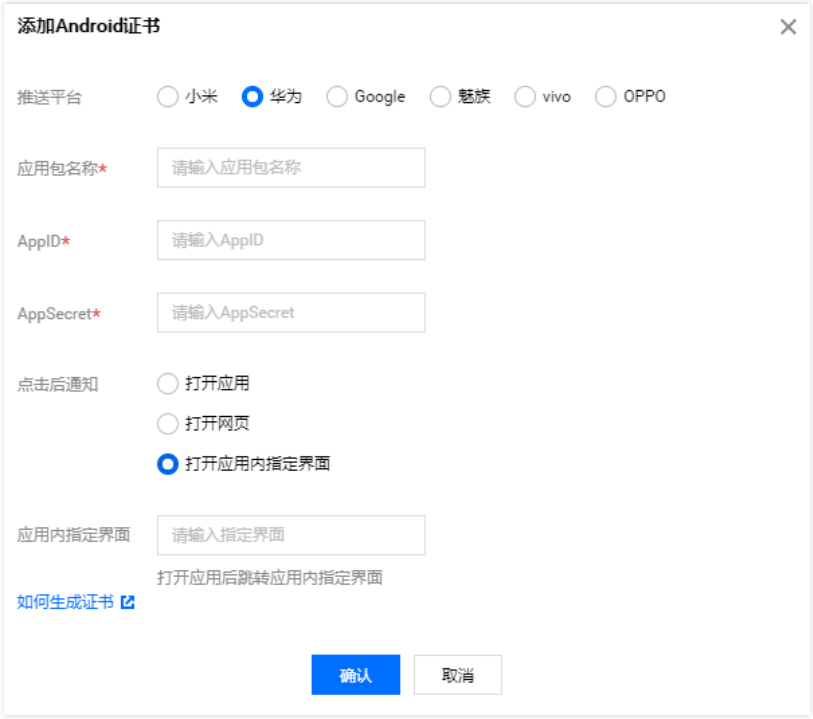
```
<activity
android:name="com.tencent.qcloud.tim.demo.chat.ChatActivity"
android:launchMode="singleTask"
android:screenOrientation="portrait"
android:windowSoftInputMode="adjustResize|stateHidden">

<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. 获取 intent URL，方式如下：

```
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);

// 打印结果
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. 在 添加证书 时选择【打开应用内指定界面】并输入上述打印结果。



透传自定义内容

添加证书 时设置【点击通知后】为【打开应用】或【打开应用内指定界面】操作才支持透传自定义内容。

步骤1：发送端设置自定义内容

在发消息前设置每条消息的通知栏自定义内容。

- Android 端示例：

```
String extContent = "ext content";
TIMMessageOfflinePushSettings settings = new TIMMessageOfflinePushSettings();
settings.setExt(extContent.getBytes());
```

```
timMessage.setOfflinePushSettings(settings);
mConversation.sendMessage(false, timMessage, callback);
```

- 服务端示例请参见 OfflinePushInfo 的格式示例。

步骤2：接收端获取自定义内容

点击通知栏的消息时，客户端在相应的 `Activity` 中获取自定义内容。

```
Bundle bundle = getIntent().getExtras();
String extContent = bundle.get("ext");
```

常见问题

如果应用使用了混淆，如何防止华为离线推送功能异常？

如果您的应用使用了混淆，为了防止华为离线推送功能异常，您需要 keep 自定义的 BroadcastReceiver，参考添加以下混淆规则：

> 说明：
> 以下代码为华为官方示例，请根据实际情况修改后再使用。

```
-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.**{*;}
# 请将 com.tencent.qcloud.tim.demo.thirdpush.HUAWEIPushReceiver 改成您 App 中定义的完整类名
-keep com.tencent.qcloud.tim.demo.thirdpush.HUAWEIPushReceiver {*;}
```

能否自定义配置推送提示音？

目前华为推送不支持自定义的提示音。

收不到推送时，如何排查问题？

1. 任何推送都不是100%必达，厂商推送也不例外。因此，若在快速、连续的推送过程中偶现一两条推送未通知提醒，通常是由厂商推送频控的限制引起。
2. 按照推送的流程，确认华为推送证书信息是否正确配置在 即时通信 IM 控制台 中。
3. 确认您的项目 集成华为推送 SDK 的配置正确，并正常获取到了 token。
4. 确认您已将正确的 推送信息上报 至即时通信 IM 服务端。
5. 在设备中手动 kill App，发送若干条消息，确认是否能在一分钟内接收到通知。

离线推送（Google FCM）

最近更新时间：2020-06-05 09:18:45

> 注意：
> 使用 FCM 离线推送需要手机端安装 Google Play Services 且在中国大陆地区以外使用。

离线推送流程

实现离线消息推送的过程如下：

1. 开发者到厂商的平台注册账号，并通过开发者认证后，申请开通推送服务。
2. 创建推送服务，并绑定应用信息，获取推送证书、密码、密钥等信息。
3. 登录 即时通信 IM 控制台 填写推送证书及相关信息，即时通信 IM 服务端会为每个证书生成不同的证书 ID。
4. 将厂商提供的推送 SDK 集成到开发者的项目工程中，并按各厂商的要求进行配置。
5. 集成即时通信 IM SDK 到项目后，将证书 ID、设备信息等上报至即时通信 IM 服务端。
6. 当客户端 App 在即时通信 IM 没有退出登录的情况下，被系统或者用户 kill 时，即时通信 IM 服务端将通过消息推送进行提醒。

配置离线推送

步骤1：设置 Firebase 和 FCM SDK

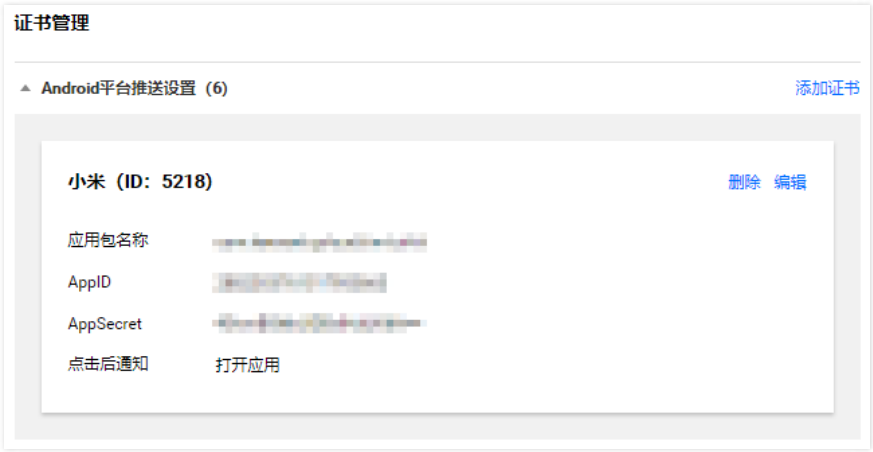
> 说明：
> 本步骤中的网址为 Firebase 官方网址，需要在中国大陆地区以外才能访问。

1. 请参考 Firebase 云消息传递 设置 Firebase，集成 FCM SDK，启动应用后获取设备注册令牌 **token**。
2. 请参考 FCM 测试指引 测试通知消息，确保已成功集成 FCM。
3. 登录 Firebase 控制台，单击您的应用卡片，进入应用配置页面。
4. 单击 Project Overview 右侧的  ，选择【项目设置】>【服务帐号】。
5. 单击【生成新的私钥】下载私钥文件。

步骤2：托管证书信息到即时通信 IM

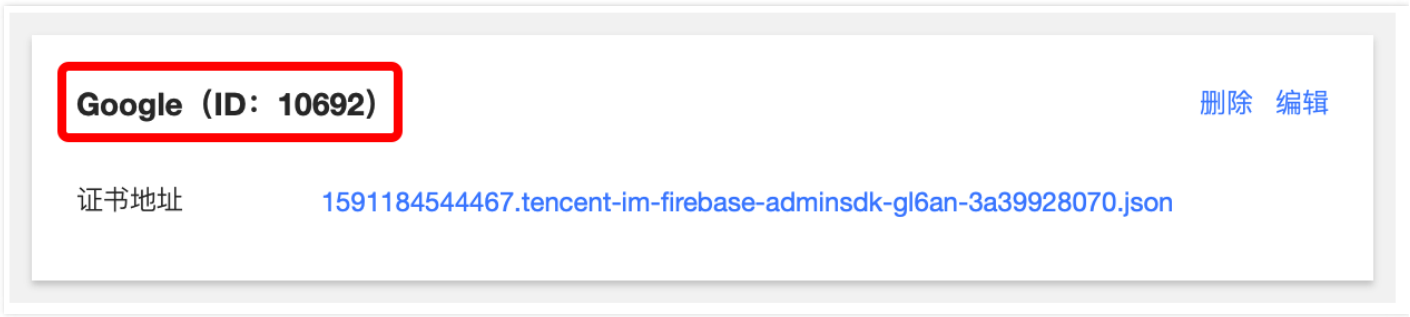
1. 登录腾讯云 即时通信 IM 控制台，单击目标应用卡片，进入应用的基础配置页面。
2. 单击【Android平台推送设置】区域的【添加证书】。

> 说明：
> 如果您原来已有证书只需变更信息，可以单击对应证书区域的【编辑】进行修改更新。



3. 上传 步骤1 中获取的私钥文件：

4. 单击【确认】保存信息，证书信息保存后10分钟内生效。
5. 待推送证书信息生成后，记录证书的 **ID** 。



步骤3：上报推送信息至即时通信 IM 服务端

在**用户登录成功后**通过 TIMManager 中的 setOfflinePushToken 方法将您托管到即时通信 IM 控制台生成的**证书 ID** 及集成 FCM 后在客户端生成的 **token** 上报到即时通信 IM 服务端。

注意：
正确上报 token 与证书 ID 后，即时通信 IM 服务才能将用户与对应的设备信息绑定，从而使用 FCM 进行推送通知。

以下为 Demo 中的示例代码：

定义证书 ID 常量：

```
/****** FCM 离线推送参数 start ******/
// 使用您在即时通信 IM 控制台上 FCM 推送证书信息里的证书 ID
public static final long GOOGLE_FCM_PUSH_BUZID = 6768;
/****** FCM 离线推送参数 end ******/
```

上报推送的证书 ID 及 token：

```
/**
* 在 ThirdPushTokenMgr.java 中对推送的证书 ID 及设备信息进行上报操作
*/
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 在此处传值
}

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;
}
});
}
}
```

步骤4：离线推送

成功上报证书 ID 及 token 后，即时通信 IM 服务端会在该设备上的即时通信 IM 用户 logout 之前、App 被 kill 之后将消息通过 FCM 推送通知到用户端。

说明：

- FCM 推送并非100%必达。
- FCM 推送可能会有一定延时，通常与 App 被 kill 的时机有关，部分情况下与 FCM 推送服务有关。
- 若即时通信 IM 用户已经 logout 或被即时通信 IM 服务端主动下线（例如在其他端登录被踢等情况），则该设备上不会再收到消息推送。

透传自定义内容

步骤1：发送端设置自定义内容

在发消息前设置每条消息的通知栏自定义内容。

- Android 端示例如下：

```
String extContent = "ext content";

TIMMessageOfflinePushSettings settings = new TIMMessageOfflinePushSettings();
settings.setExt(extContent.getBytes());
timMessage.setOfflinePushSettings(settings);
mConversation.sendMessage(false, timMessage, callback);
```

- 服务端示例请参见 OfflinePushInfo 的格式示例。

步骤2：接收端获取自定义内容

当点击通知栏的消息时，客户端在相应的 `Activity` 中获取自定义内容。

```
Bundle bundle = getIntent().getExtras();
String value = bundle.getString("ext");
```

常见问题

能否自定义配置推送提示音？

目前 FCM 推送不支持自定义的提示音。

收不到推送时，如何排查问题？

1. 任何推送都不是100%必达，FCM 推送也不例外。因此，若在快速、连续的推送过程中偶现一两条推送未通知提醒，通常是由 FCM 推送频控的限制引起。
2. 按照推送的流程，确认 FCM 推送证书信息是否正确配置在 即时通信 IM 控制台中。
3. 确认您的 FCM 项目配置正确，并已正常获取 token。
4. 确认您已将正确的 推送信息上报 至即时通信 IM 服务端。
5. 在设备中手动 kill App，发送若干条消息，确认是否能在一分钟内接收到通知。

离线推送（魅族）

最近更新时间：2020-09-23 10:56:25

离线推送流程

实现离线消息推送的过程如下：

1. 开发者到厂商的平台注册账号，并通过开发者认证后，申请开通推送服务。
2. 创建推送服务，并绑定应用信息，获取推送证书、密码、密钥等信息。
3. 登录 即时通信 IM 控制台 填写推送证书及相关信息，即时通信 IM 服务端会为每个证书生成不同的证书 ID。
4. 将厂商提供的推送 SDK 集成到开发者的项目工程中，并按各厂商的要求进行配置。
5. 集成即时通信 IM SDK 到项目后，将证书 ID、设备信息等上报至即时通信 IM 服务端。
6. 当客户端 App 在即时通信 IM 没有退出登录的情况下，被系统或者用户 kill 时，即时通信 IM 服务端将通过消息推送进行提醒。

配置离线推送

Flyme 为深度定制 Android 系统，对于第三方 App 自启动权限管理很严格，默认情况下第三方 App 都不会在系统的自启动白名单内，App 在后台时容易被系统 kill，因此推荐在魅族设备上集成魅族推送，魅族推送是 Flyme 的系统级服务，推送到达率较高。目前，**即时通信 IM 仅支持魅族推送的通知栏消息。**

> 注意：
>
> - 此指引文档是直接参考魅族官方文档所写，若魅族推送有变动，请以魅族推送官网文档为准。
> - 此指引是根据 Flyme 推送接入指南所写，仅针对 Flyme 系统，并非魅族统一推送平台（各个厂家的整合）。
> - 如果不需要对魅族设备做专门的离线推送适配，可以忽略此章节。

步骤1：申请魅族推送证书

1. 打开 魅族开放平台官网 进行注册并通过开发者认证。

> 说明：
> 认证过程大约需要3天左右，请务必提前阅读魅族推送服务启用指南，以免影响您的接入进度。

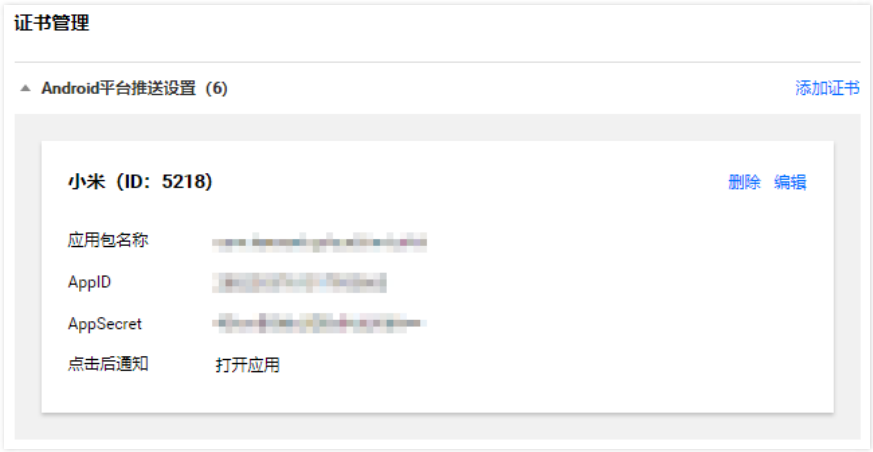
2. 登录魅族开放平台的管理控制台，选择【服务】>【集成推送服务】>【推送后台】，创建魅族推送服务应用。
   魅族推送服务应用创建完成后，在应用详情中，您可以查看详细的应用信息。

3. 记录 **应用包名** 、 **App ID** 、 **App Secret** 信息。

步骤2：托管证书信息到即时通信 IM

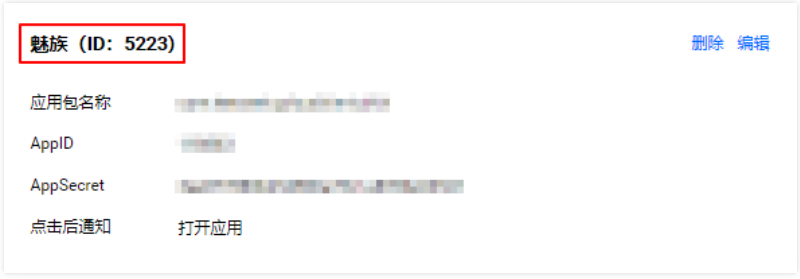
1. 登录腾讯云 即时通信 IM 控制台，单击目标应用卡片，进入应用的基础配置页面。
2. 单击【Android平台推送设置】区域的【添加证书】。

> 说明：
> 如果您原来已有证书只需变更信息，可以单击【Android平台推送设置】区域的【编辑】进行修改更新。



3. 根据 步骤1 中获取的信息设置以下参数：
   - **推送平台**：选择**魅族**
   - **应用包名称**：填写魅族推送服务应用的**应用包名**
   - **AppID**：填写魅族推送服务应用的 **App ID**
   - **AppSecret**：填写魅族推送服务应用的 **App Secret**
   - **点击通知后**：选择点击通知栏消息后的响应操作， 支持**打开应用**、**打开网页**和**打开应用内指定界面**，更多详情请参见 配置点击通知栏消息事件
     当设置为【打开应用】或【打开应用内指定界面】操作时，支持 透传自定义内容。

4. 单击【确认】保存信息，证书信息保存后10分钟内生效。
5. 待推送证书信息生成后，记录证书的 **ID**。



步骤3：集成推送 SDK

说明：

- 即时通信 IM 默认推送的通知标题为 `a new message`。
- 阅读此小节前，请确保您已经正常集成并使用即时通信 IM SDK。
- 您可以在我们的 demo 里找到魅族推送的实现示例，请注意：魅族推送版本更新时有可能会有功能调整，若您发现本节内容存在差异，烦请您及时查阅 魅族推送官网文档，并将文档信息差异反馈给我们，我们会及时跟进修改。

步骤3.1：下载魅族推送 SDK 并添加引用

访问 魅族推送运营平台 下载魅族 Flyme 推送 SDK aar 包或者使用 jcenter 集成。

```
dependencies {
// MEIZU push sdk
compile 'com.meizu.flyme.internet:push-internal:3.6.+@aar'
}
```

步骤3.2：配置 AndroidManifest.xml 文件

添加魅族推送必须的权限：

```
<!-- ********魅族推送权限设置 start******** -->
<!-- 兼容 flyme5.0 以下版本，魅族内部集成 pushSDK 必填，不然无法收到消息-->
<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-permission>

<!-- 兼容 flyme3.0 配置权限-->
<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"/>
<!-- ********魅族推送权限设置 end******** -->
<!--这里的 com.tencent.qcloud.tim.tuikit 改成您的 App 的包名-->
```

步骤3.3：自定义一个 BroadcastReceiver 类

为了接收消息，您需要自定义一个继承自 `MzPushMessageReceiver` 类的 BroadcastReceiver，并实现其中的 `onRegisterStatus` 方法，然后将此 receiver 注册到 AndroidManifest.xml 中。

以下为 Demo 中的示例代码：

```
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();
}

}
```

将自定义的 BroadcastReceiver 注册到 AndroidManifest.xml：

```
<!--这里的 com.tencent.qcloud.tim.demo.thirdpush.MEIZUPushReceiver 修改成您 App 中的完整类名-->
<!-- ********魅族推送设置 start******** -->
<receiver android:name="com.tencent.qcloud.tim.demo.thirdpush.MEIZUPushReceiver">
<intent-filter>
<!-- 接收 push 消息 -->
<action android:name="com.meizu.flyme.push.intent.MESSAGE" />
<!-- 接收 register 消息 -->
<action android:name="com.meizu.flyme.push.intent.REGISTER.FEEDBACK" />
```

```
<!-- 接收 unregister 消息-->
<action android:name="com.meizu.flyme.push.intent.UNREGISTER.FEEDBACK"/>
<!-- 兼容低版本 Flyme3 推送服务配置 -->
<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>
<!-- ********魅族推送设置 end******** -->
```

步骤3.4：在 App 中注册魅族推送服务

如果您选择启用魅族离线推送，需要向魅族服务器注册推送服务，通过调用 PushManager.register 来对魅族推送服务进行初始化。 PushManager.register 可在任意地方调用，为了提高注册成功率，魅族官方建议在 Application 的 onCreate 中调用。
注册成功后，您将在 步骤3.3 自定义的 BroadcastReceiver 的 onRegisterStatus 中收到注册结果。其中 registerStatus.getPushId() 为当前设备上当前 App 的唯一标识，请记录 PushId 信息。

以下为 Demo 中的示例代码：

```
public class DemoApplication extends Application {

private static PojoApplication instance;

@Override
public void onCreate() {
super.onCreate();
// 判断是否是在主线程
if (SessionWrapper.isMainProcess(getApplicationContext())) {
/**
* TUIKit 的初始化函数
*
* @param context 应用的上下文，一般为对应应用的 ApplicationContext
* @param sdkAppID 您在腾讯云注册应用时分配的 SDKAppID
* @param configs TUIKit 的相关配置项，一般使用默认即可，需特殊配置参考 API 文档
*/
long current = System.currentTimeMillis();
TUIKit.init(this, Constants.SDKAPPID, BaseUIKitConfigs.getDefaultConfigs());
System.out.println(">>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));
// 添加自定初始化配置
customConfig();
System.out.println(">>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));

if(IMFunc.isBrandXiaoMi()){
// 小米离线推送
MiPushClient.registerPush(this, Constants.XM_PUSH_APPID, Constants.XM_PUSH_APPKEY);
}
if(IMFunc.isBrandHuawei()){
// 华为离线推送
HMSAgent.init(this);
}
if(MzSystemUtils.isBrandMeizu(this)){
// 魅族离线推送
PushManager.register(this, Constants.MZ_PUSH_APPID, Constants.MZ_PUSH_APPKEY);
}
if(IMFunc.isBrandVivo()){
// vivo 离线推送
PushClient.getInstance(getApplicationContext()).initialize();
}
}
instance = this;
}
}
```

步骤4：上报推送信息至即时通信 IM 服务端

若您需要通过魅族推送进行即时通信 IM 消息的推送通知，必须在**用户登录成功后**通过 TIMManager 中的 setOfflinePushToken 方法将您托管到即时通信 IM 控制台生成的 **证书 ID** 及魅族推送服务端生成的 **PushId** 上报到即时通信 IM 服务端。

注意：
正确上报 PushId 与证书 ID 后，即时通信 IM 服务才能将用户与对应的设备信息绑定，从而使用魅族推送服务进行推送通知。

以下为 Demo 中的示例代码：

- 定义证书 ID 常量：

```
/**
* 我们先定义一些常量信息在 Constants.java
*/
/****** 魅族离线推送参数 start ******/
// 使用您在即时通信 IM 控制台上魅族推送证书信息里的证书 ID
public static final long MZ_PUSH_BUZID = 6666;
// 魅族开放平台分配的应用 APPID 及 APPKEY
public static final String MZ_PUSH_APPID = "1234512345123451234";
public static final String MZ_PUSH_APPKEY = "1234512345123";
/****** 魅族离线推送参数 end ******/
```

- 上报推送的证书 ID 及 PushId：

```
/**
* 在 ThirdPushTokenMgr.java 中对推送的证书 ID 及设备信息进行上报操作
*/
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; // PushId 在此处传值，结合上文自定义 BroadcastReciever 类文档说明
}

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()){ // 判断厂商品牌，根据不同厂商选择不同的推送服务
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;
}
});
}
}
```

步骤5：离线推送

成功上报证书 ID 及 PushId 后，即时通信 IM 服务端会在该设备上的即时通信 IM 用户 logout 之前、APP 被 kill 之后，将消息通过魅族推送通知到用户端。

说明：

- 魅族推送并非100%必达。
- 魅族推送可能会有一定延时，通常与 App 被 kill 的时机有关，部分情况下与魅族推送服务有关。
- 若即时通信 IM 用户已经 logout 或被即时通信 IM 服务端主动下线（例如在其他端登录被踢等情况），则该设备上不会再收到消息推送。

配置点击通知栏消息事件

您可以选择点击通知栏消息后**打开应用**、**打开网页**或**打开应用内指定界面**。

打开应用

默认为点击通知栏消息打开应用。

打开网页

您需要在 添加证书 时选择【打开网页】并输入以 http:// 或 https:// 开头的网址，例如 https://cloud.tencent.com/document/product/269 。

打开应用内指定界面

您需要在 添加证书 时选择【打开应用内指定界面】并输入需要打开的 Activity 的完整类名，例如 com.tencent.qcloud.tim.demo.chat.ChatActivity 。

透传自定义内容

添加证书 时设置【点击通知后】为【打开应用】或【打开应用内指定界面】操作才支持透传自定义内容。

步骤1：发送端设置自定义内容

在发消息前设置每条消息的通知栏自定义内容。

- Android 端示例如下：

```
String extContent = "ext content";
TIMMessageOfflinePushSettings settings = new TIMMessageOfflinePushSettings();
settings.setExt(extContent.getBytes());
timMessage.setOfflinePushSettings(settings);
mConversation.sendMessage(false, timMessage, callback);
```

- 服务端示例请参见 OfflinePushInfo 的格式示例。

步骤2：接收端获取自定义内容

点击通知栏的消息时，会触发魅族推送 SDK 的 `onNotificationClicked(Context context, MzPushMessage mzPushMessage)` 回调 ，自定义内容可以从 `mzPushMessage` 中获取。

```
String extContent = mzPushMessage.getSelfDefineContentString();
```

另外，客户端也可以在打开的 `Activity` 中获取自定义内容。

```
Bundle bundle = getIntent().getExtras();
String extContent = bundle.getString("ext");
```

常见问题

如果应用使用了混淆，如何防止魅族离线推送功能异常？

如果您的应用使用了混淆，为了防止魅族离线推送功能异常，您需要 keep 自定义的 BroadcastReceiver，参考添加以下混淆规则：

> 说明：
> 以下代码为魅族官方示例，请根据实际情况修改后再使用。

```
# 请将 com.tencent.qcloud.tim.demo.thirdpush.MEIZUPushReceiver 改成您 App 中定义的完整类名
-keep com.tencent.qcloud.tim.demo.thirdpush.MEIZUPushReceiver {*;}
```

能否自定义配置推送提示音？

目前魅族推送不支持自定义的提示音。

收不到推送时，如何排查问题？

1. 任何推送都不是100%必达，厂商推送也不例外。因此，若在快速、连续的推送过程中偶现一两条推送未通知提醒，通常是由厂商推送频控的限制引起。
2. 按照推送的流程，确认魅族推送证书信息是否正确配置在 即时通信 IM 控制台中。
3. 确认您的项目 集成魅族推送 SDK 的配置正确，并正常获取到了 PushId。
4. 确认您已将正确的 推送信息上报 至即时通信 IM 服务端。
5. 在设备中手动 kill App，发送若干条消息，确认是否能在一分钟内接收到通知。

离线推送（vivo）

最近更新时间：2020-03-20 09:32:09

离线推送流程

实现离线消息推送的过程如下：

1. 开发者到厂商的平台注册账号，并通过开发者认证后，申请开通推送服务。
2. 创建推送服务，并绑定应用信息，获取推送证书、密码、密钥等信息。
3. 登录 即时通信 IM 控制台 填写推送证书及相关信息，即时通信 IM 服务端会为每个证书生成不同的证书 ID。
4. 将厂商提供的推送 SDK 集成到开发者的项目工程中，并按各厂商的要求进行配置。
5. 集成即时通信 IM SDK 到项目后，将证书 ID、设备信息等上报至即时通信 IM 服务端。
6. 当客户端 App 在即时通信 IM 没有退出登录的情况下，被系统或者用户 kill 时，即时通信 IM 服务端将通过消息推送进行提醒。

配置离线推送

vivo 手机使用深度定制 Android 系统，对于第三方 App 自启动权限管理很严格，默认情况下第三方 App 都不会在系统的自启动白名单内，App 在后台时容易被系统 kill，因此推荐在 vivo 设备上集成 vivo 推送，vivo 推送 是 vivo 设备的系统级服务，推送到达率较高。目前，**即时通信 IM 仅支持 vivo 推送的通知栏消息。**

> 注意：
>
> - 此指引文档是直接参考 vivo 推送官方文档所写，若 vivo 推送有变动，请以 vivo 推送官网文档 为准。
> - 如果不需要对 vivo 设备做专门的离线推送适配，可以忽略此章节。

步骤1：申请 vivo 推送证书

1. 打开 vivo 开放平台官网 进行注册并通过开发者认证。

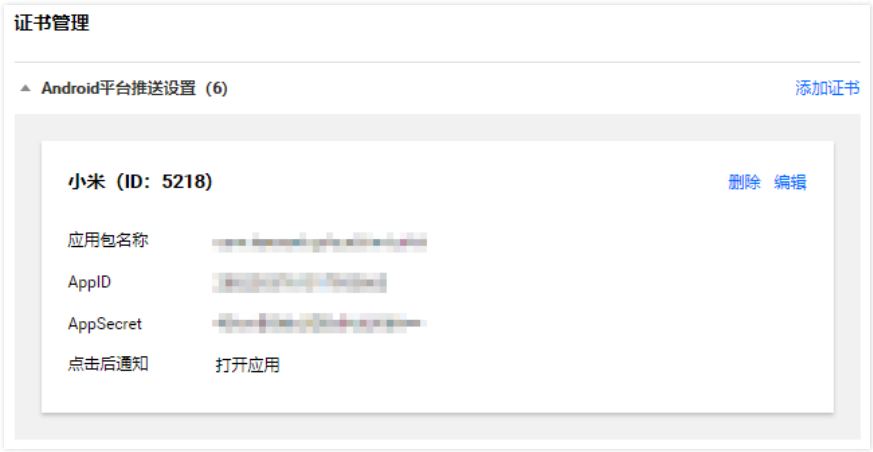
> 说明：
> 认证过程大约需要3天左右，请务必提前阅读 vivo 推送服务说明，以免影响您的接入进度。

2. 登录 vivo 开放平台的管理中心，选择【消息推送】>【创建】>【测试推送】，创建 vivo 推送服务应用。
   vivo 推送服务应用创建完成后，在应用详情中，您可以查看详细的应用信息。
3. 记录 **APP ID** 、 **APP key** 和 **APP secret** 信息。

步骤2：托管证书信息到即时通信 IM

1. 登录腾讯云 即时通信 IM 控制台，单击目标应用卡片，进入应用的基础配置页面。
2. 单击【Android平台推送设置】区域的【添加证书】。

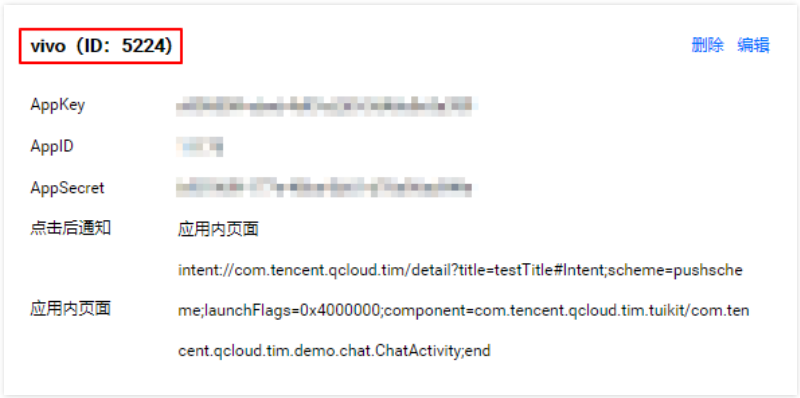
> 说明：
> 如果您原来已有证书只需变更信息，可以单击对应证书区域的【编辑】进行修改更新。



3. 根据 步骤1 中获取的信息设置以下参数：
   - **推送平台**：选择 **vivo**
   - **AppKey**：填写 vivo 推送服务应用的 **APP key**
   - **AppID**：填写 vivo 推送服务应用的 **APP ID**
   - **AppSecret**：填写 vivo 推送服务应用的 **APP secret**
   - **点击通知后**：选择点击通知栏消息后的响应操作，支持**打开应用**、**打开网页**和**打开应用内指定界面**，更多详情请参见 配置点击通知栏消息事件
     当设置为【打开应用】或【打开应用内指定界面】操作时，支持 透传自定义内容。

4. 单击【确认】保存信息，证书信息保存后10分钟内生效。

5. 待推送证书信息生成后，记录证书的 **ID**。



步骤3：集成推送 SDK

说明：

- 即时通信 IM 默认推送的通知标题为 `a new message`。
- 阅读此小节前，请确保您已经正常集成并使用即时通信 IM SDK。
- 您可以在我们的 demo 里找到 vivo 推送的实现示例，请注意： vivo 推送版本更新时有可能会有功能调整，若您发现本节内容存在差异，烦请您及时查阅 vivo 推送官网文档，并将文档信息差异反馈给我们，我们会及时跟进修改。

步骤3.1：下载 vivo 推送 SDK 并添加引用

1. 访问 vivo 推送运营平台 下载 vivo 推送 SDK。
2. 解压 vivo 推送 SDK，获取 `vivo_pushsdk_xxx.jar` 库文件。
3. 将 `vivo_pushsdk_xxx.jar` 库文件添加到您项目的 `libs` 目录下，并且在项目中添加引用。

步骤3.2：配置 AndroidManifest.xml 文件

添加 vivo 推送服务需要的配置：

```
<!-- ********vivo 推送设置 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 推送设置 end******** -->
<!--这里的 com.vivo.push.app_id ，com.vivo.push.api_key 由 vivo 开放平台生成 -->
```

步骤3.3：自定义一个 BroadcastReceiver 类

为了接收消息，您需要自定义一个继承自 `OpenClientPushMessageReceiver` 类的 BroadcastReceiver，并实现其中的 `onReceiveRegId` 和 `onNotificationMessageClicked` 方法，然后将此 receiver 注册到 AndroidManifest.xml 中。

以下为 Demo 中的示例代码：

```
public class VIVOPushMessageReceiverImpl extends OpenClientPushMessageReceiver {
private static final String TAG = "VIVOPushMessageReceiver";
@Override
```

```
public void onNotificationMessageClicked(Context context, UPSNotificationMessage upsNotificationMessage) {
Log.i(TAG, "onNotificationMessageClicked");
}

@Override
public void onReceiveRegId(Context context, String regId) {
// vivo regId 有变化会走这个回调。根据官网文档，获取 regId 需要在开启推送的回调里面调用PushClient.getInstance(getApplicationContext()).getRegId();参考 L
oginActivity
Log.i(TAG, "onReceiveRegId = " + regId);
}
}
```

将自定义的 BroadcastReceiver 注册到 AndroidManifest.xml：

```
<!--这里的 com.tencent.qcloud.tim.demo.thirdpush.VIVOPushMessageReceiverImpl 修改成您 App 中的完整类名-->
<!-- push 应用定义消息 receiver 声明 -->
<receiver android:name="com.tencent.qcloud.tim.demo.thirdpush.VIVOPushMessageReceiverImpl">
<intent-filter>
<!-- 接收 push 消息 -->
<action android:name="com.vivo.pushclient.action.RECEIVE" />
</intent-filter>
</receiver>
```

步骤3.4：在 App 中注册 vivo 推送服务

如果您选择启用 vivo 离线推送，需要向 vivo 服务器注册推送服务，通过调用 PushClient.getInstance(getApplicationContext()).initialize() 来对 vivo 推送服务进行初始化。 PushClient.getInstance(getApplicationContext()).initialize() 可在任意地方调用，为了提高注册成功率，vivo 官方建议在 Application 的 onCreate 中调用。

注册成功后，需要在您的 App 主界面中获取注册结果。其中 regId 为当前设备上当前 App 的唯一标识，请记录 regId 信息。

以下为 Demo 中的示例代码：

```
public class DemoApplication extends Application {

private static PojoApplication instance;

@Override
public void onCreate() {
super.onCreate();
// 判断是否是在主线程
if (SessionWrapper.isMainProcess(getApplicationContext())) {
/**
* TUIKit 的初始化函数
*
* @param context 应用的上下文，一般为对应应用的 ApplicationContext
* @param sdkAppID 您在腾讯云注册应用时分配的 SDKAppID
* @param configs TUIKit 的相关配置项，一般使用默认即可，需特殊配置参考 API 文档
*/
long current = System.currentTimeMillis();
TUIKit.init(this, Constants.SDKAPPID, BaseUIKitConfigs.getDefaultConfigs());
System.out.println(">>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));
// 添加自定初始化配置
customConfig();
System.out.println(">>>>>>>>>>>>>>>>>>>"+(System.currentTimeMillis()-current));

if(IMFunc.isBrandXiaoMi()){
// 小米离线推送
MiPushClient.registerPush(this, Constants.XM_PUSH_APPID, Constants.XM_PUSH_APPKEY);
}
if(IMFunc.isBrandHuawei()){
// 华为离线推送
HMSAgent.init(this);
}
if(MzSystemUtils.isBrandMeizu(this)){
// 魅族离线推送
PushManager.register(this, Constants.MZ_PUSH_APPID, Constants.MZ_PUSH_APPKEY);
}
if(IMFunc.isBrandVivo()){
```

```
// vivo 离线推送
PushClient.getInstance(getApplicationContext()).initialize();
}
}
instance = this;
}
}
```

在主界面中打开 vivo push 服务：

```
if (IMFunc.isBrandVivo()) {
    // vivo 离线推送
    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 {
                // 根据 vivo 推送文档说明，state = 101表示该 vivo 机型或者版本不支持 vivo 推送，详情请参考 vivo 推送常见问题汇总
                QLog.i(TAG, "vivopush open vivo push fail state = " + state);
            }
        }
    });
```

步骤4：上报推送信息至即时通信 IM 服务端

若您需要通过 vivo 推送进行即时通信 IM 消息的推送通知，必须在**用户登录成功后**通过 TIMManager 中的 setOfflinePushToken 方法将您托管到即时通信 IM 控制台生成的**证书 ID** 及 vivo 推送服务返回的 **regId** 上报到即时通信 IM 服务端。

> 注意：
> 正确上报 regId 与证书 ID 后，即时通信 IM 服务才能将用户与对应的设备信息绑定，从而使用 vivo 推送服务进行推送通知。

以下为 Demo 中的示例代码：

- 定义证书 ID 常量：

```
/**
* 我们先定义一些常量信息在 Constants.java
*/
/****** vivo 离线推送参数 start ******/
// 在腾讯云控制台上传第三方推送证书后分配的证书 ID
public static final long VIVO_PUSH_BUZID = 6666;
// vivo 开放平台分配的应用 APPID 及 APPKEY
public static final String VIVO_PUSH_APPID = "12345123451234512345"; // 见清单文件
public static final String VIVO_PUSH_APPKEY = "12345abcde"; // 见清单文件
/****** vivo 离线推送参数 end ******/
```

- 上报推送的证书 ID 及 regId：

```
/**
* 在 ThirdPushTokenMgr.java 中对推送的证书 ID 及设备信息进行上报操作
*/
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; // regId 在此处传值，结合上文自定义 BroadcastReciever 类文档说明
}

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()){ // 判断厂商品牌，根据不同厂商选择不同的推送服务
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;
}
});
}
}
```

步骤5：离线推送

成功上报证书 ID 及 regId 后，即时通信 IM 服务端会在该设备上的即时通信 IM 用户 logout 之前、App 被 kill 之后，将消息通过 vivo 推送通知到用户端。

说明：

- vivo 推送只支持部分 vivo 手机，详情请参见 vivo 推送常见问题汇总。
- vivo 推送并非100%必达。
- vivo 推送可能会有一定延时，通常与 App 被 kill 的时机有关，部分情况下与 vivo 推送服务有关。
- 若即时通信 IM 用户已经 logout 或被即时通信 IM 服务端主动下线（例如在其他端登录被踢等情况），则该设备上不会再收到消息推送。

配置点击通知栏消息事件

您可以选择点击通知栏消息后**打开应用**、**打开网页**或**打开应用内指定界面**。

打开应用

默认为点击通知栏消息打开应用。

打开网页

您需要在 添加证书 时选择【打开网页】并输入以 `http://` 或 `https://` 开头的网址，例如 `https://cloud.tencent.com/document/product/269` 。

打开应用内指定界面

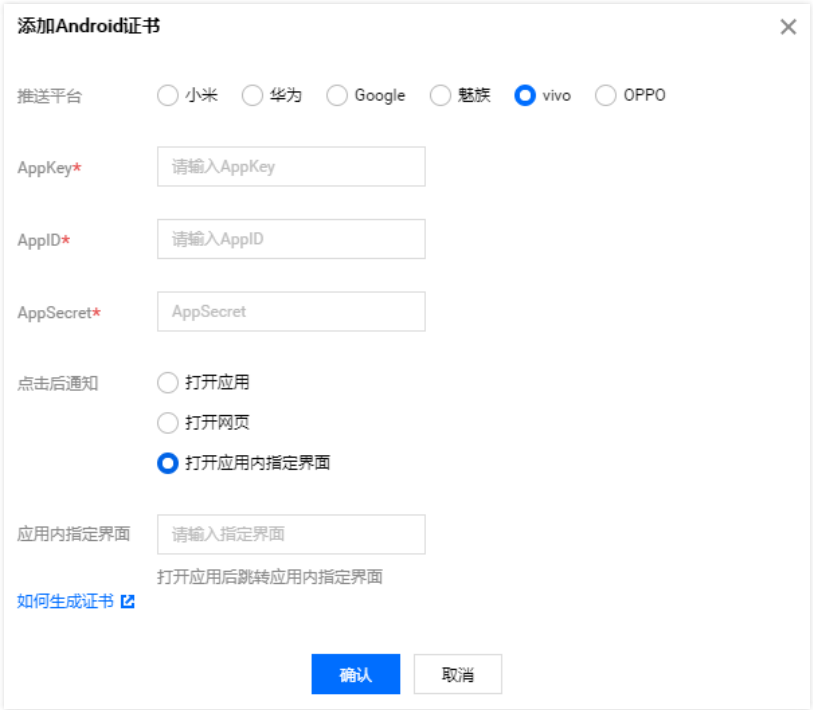
1. 在 manifest 中配置需要打开的 Activity 的 `intent-filter` ，示例代码如下：

```
<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. 获取 intent URL，方式如下：

```
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);

// 打印结果
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. 在 添加证书 时选择【打开应用内指定界面】并输入上述打印结果。



透传自定义内容

添加证书 时设置【点击通知后】为【打开应用】或【打开应用内指定界面】操作才支持透传自定义内容。

步骤1：发送端设置自定义内容

在发消息前设置每条消息的通知栏自定义内容。

- Android 端示例如下：

```
String extContent = "ext content";
TIMMessageOfflinePushSettings settings = new TIMMessageOfflinePushSettings();
settings.setExt(extContent.getBytes());
timMessage.setOfflinePushSettings(settings);
mConversation.sendMessage(false, timMessage, callback);
```

- 服务端示例请参见 OfflinePushInfo 的格式示例。

步骤2：接收端获取自定义内容

点击通知栏的消息时，会触发 vivo 推送 SDK 的 `onNotificationMessageClicked(Context context, UPSNotificationMessage upsNotificationMessage)` 回调，自定义内容可以从 `upsNotificationMessage` 中获取。

```
Map<String, String> paramMap = upsNotificationMessage.getParams();
String extContent = paramMap.get("ext");
```

常见问题

如果应用使用了混淆，如何防止 vivo 离线推送功能异常？

如果您的应用使用了混淆，为了防止 vivo 离线推送功能异常，您需要 keep 自定义的 BroadcastReceiver，参考添加以下混淆规则：

> 说明：
> 以下代码为 vivo 官方示例，请根据实际情况修改后再使用。

```
# 请将 com.tencent.qcloud.tim.demo.thirdpush.VIVOPushMessageReceiverImpl 改成您 App 中定义的完整类名
# vivo 推送
-dontwarn com.vivo.push.**
-keep class com.vivo.push.**{*; }
-keep class com.vivo.vms.**{*; }
-keep class com.tencent.qcloud.tim.demo.thirdpush.VIVOPushMessageReceiverImpl{*;}
```

能否自定义配置推送提示音？

目前 vivo 推送不支持自定义的提示音。

收不到推送时，如何排查问题？

1. 任何推送都不是100%必达，厂商推送也不例外。因此，若在快速、连续的推送过程中偶现一两条推送未通知提醒，通常是由厂商推送频控的限制引起。
2. 按照推送的流程，确认 vivo 推送证书信息是否正确配置在 即时通信 IM 控制台 中。
3. 确认您的项目 集成 vivo 推送 SDK 的配置正确，并正常获取到了 regId。
4. 确认您已将正确的 推送信息上报 至即时通信 IM 服务端。
5. 在设备中手动 kill App，发送若干条消息，确认是否能在一分钟内接收到通知。

离线推送（OPPO）

最近更新时间：2020-03-20 09:50:24

流程说明

实现离线消息推送的过程如下：

1. 开发者到厂商的平台注册账号，并通过开发者认证后，申请开通推送服务。
2. 创建推送服务，并绑定应用信息，获取推送证书、密码、密钥等信息。
3. 登录 即时通信 IM 控制台 填写推送证书及相关信息，即时通信 IM 服务端会为每个证书生成不同的证书 ID。
4. 将厂商提供的推送 SDK 集成到开发者的项目工程中，并按各厂商的要求进行配置。
5. 集成即时通信 IM SDK 到项目后，将证书 ID、设备信息等上报至即时通信 IM 服务端。
6. 当客户端 App 在即时通信 IM 没有退出登录的情况下，被系统或者用户 kill 时，即时通信 IM 服务端将通过消息推送进行提醒。

操作步骤

OPPO 手机使用深度定制 Android 系统，对于第三方 App 自启动权限管理很严格，默认情况下第三方 App 都不会在系统的自启动白名单内，App 在后台时容易被系统 kill，因此推荐在 OPPO 设备上集成 OPPO 推送，OPPO 推送是 OPPO 设备的系统级服务，推送到达率较高。目前，**即时通信 IM 仅支持 OPPO 推送的通知栏消息。**

注意：

- 此指引文档是直接参考 OPPO 推送官方文档所写，若 OPPO 推送有变动，请以 OPPO 推送官网文档 为准。
- 如果不需要对 OPPO 设备做专门的离线推送适配，可以忽略此章节。

步骤1：申请 OPPO 推送证书

1. 请参考 OPPO PUSH 服务开启指南 开通 PUSH 服务。
2. 在 OPPO 推送平台 >【配置管理】>【应用配置】页面，您可以查看详细的应用信息。
3. 记录 `AppId`、`AppKey`、`AppSecret` 和 `MasterSecret` 信息。

步骤2：创建 ChannelID

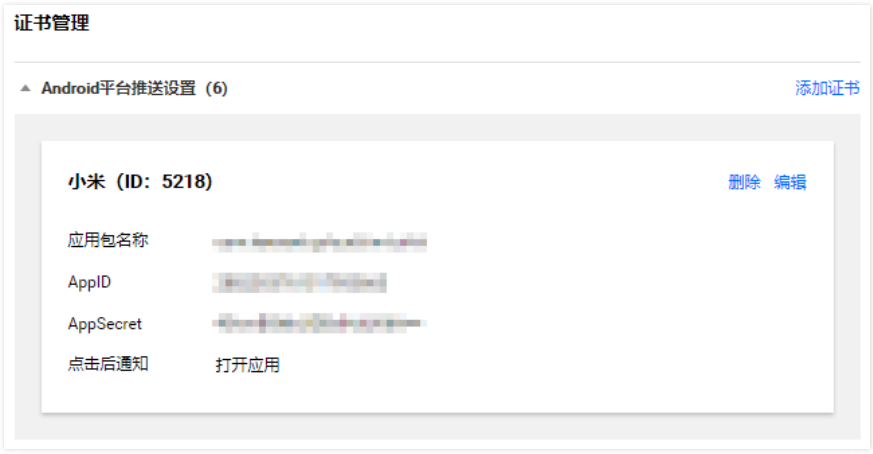
按照 OPPO 官网要求，在 OPPO Android 8.0 及以上系统版本必须配置 ChannelID，否则推送消息无法展示。您需要先在 App 中创建对应的 ChannelID（例如 `tuikit`）：

```
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);
}
}
```

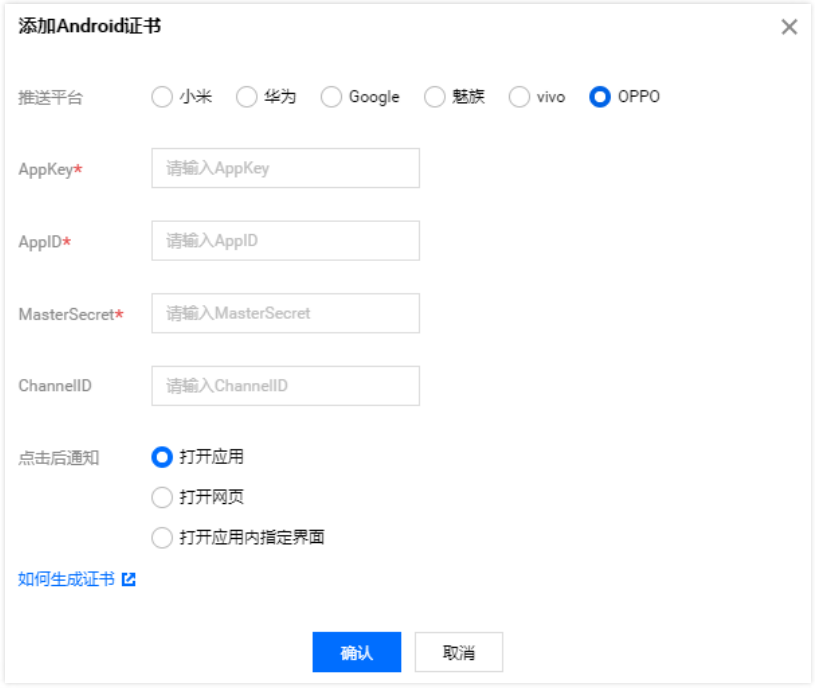
步骤3：托管证书信息到即时通信 IM

1. 登录腾讯云 即时通信 IM 控制台，单击目标应用卡片，进入应用的基础配置页面。
2. 单击【Android平台推送设置】区域的【添加证书】。

说明：
如果您原来已有证书只需变更信息，可以单击【Android平台推送设置】区域的【编辑】进行修改更新。

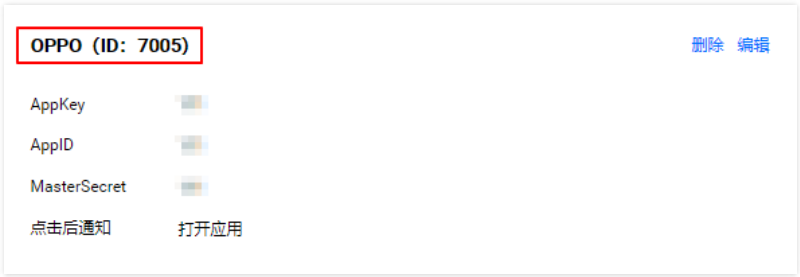


3. 根据 步骤1 和 步骤2 中获取的信息设置以下参数：
   - **推送平台**：选择 **OPPO**
   - **AppKey**：填写 OPPO 推送服务应用的 **AppKey**
   - **AppID**：填写 OPPO 推送服务应用的 **AppId**
   - **MasterSecret**：填写 OPPO 推送服务应用的 **MasterSecret**
   - **ChannelID**：填写您在 App 中创建的 **ChannelID**
   - **点击后续操作**：选择点击通知栏消息后的响应操作，支持**打开应用**、**打开网页**和**打开应用内指定界面**，更多详情请参见 配置点击通知栏消息事件
     当设置为【打开应用】或【打开应用内指定界面】操作时，支持 透传自定义内容。



4. 单击【确认】保存信息，证书信息保存后10分钟内生效。

5. 待推送证书信息生成后，记录证书的 **ID** 。



步骤4：集成推送 SDK

1. 请参考 OPPO PUSH SDK 接口文档 集成 SDK，并在 OPPO 控制台测试通知消息，确保已成功集成。
2. 通过调用 OPPO SDK 中的 PushManager.getInstance().register(...) 初始化 Opush 推送服务。
   注册成功后，您可以在 PushCallback 的 onRegister 回调方法中得到 regId 。
3. 记录 regId 信息。

步骤5：上报推送信息至即时通信 IM 服务端

若您需要通过 OPPO 推送进行即时通信 IM 消息的推送通知，必须在**用户登录成功后**通过 TIMManager 中的 setOfflinePushToken 方法将您托管到即时通信 IM 控制台生成的**证书 ID** 及 OPPO 推送服务返回的 **regId** 上报到即时通信 IM 服务端。

> 注意：
> 正确上报 regId 与证书 ID 后，即时通信 IM 服务才能将用户与对应的设备信息绑定，从而使用 OPPO 推送服务进行推送通知。

定义证书 ID 常量示例代码：

```
/****** OPPO 离线推送参数 start ******/
// 在腾讯云控制台上传第三方推送证书后分配的证书 ID
public static final long OPPO_PUSH_BUZID = 7005;
/****** OPPO 离线推送参数 end ******/
```

上报推送的证书 ID 及 regId 示例代码：

```
/**
* 在 ThirdPushTokenMgr.java 中对推送的证书 ID 及设备信息进行上报操作
*/
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; // regId 在此处传值，结合上文自定义 BroadcastReciever 类文档说明
}
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()){ // 判断厂商品牌，根据不同厂商选择不同的推送服务
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;
}
});
}
}
```

步骤6：离线推送

成功上报证书 ID 及 regId 后，即时通信 IM 服务端会在该设备上的即时通信 IM 用户 logout 之前、App 被 kill 之后将消息通过 OPPO 推送通知到用户端。

> 说明：
>
> - OPPO 推送的常见问题请参见 OPPO PUSH FAQ。
> - 若即时通信 IM 用户已经 logout 或被即时通信 IM 服务端主动下线（例如在其他端登录被踢等情况），则该设备上不会再收到消息推送。

配置点击通知栏消息事件

您可以选择点击通知栏消息后**打开应用**、**打开网页**或**打开应用内指定界面**。

打开应用

默认为点击通知栏消息打开应用。

打开网页

您需要在 添加证书 时选择【打开网页】并输入以 `http://` 或 `https://` 开头的网址，例如 `https://cloud.tencent.com/document/product/269` 。

打开应用内指定界面

打开应用内指定界面有以下几种方式：

Activity（推荐）
该方式比较简单，填入打开的 Activity 的完整类名即可，例如 `com.tencent.qcloud.tim.demo.SplashActivity`

Intent action

1. 在 AndroidManifest 要打开的 Activity 中做如下配置，并且必须加上 category 且不能有 data 数据。

```
<intent-filter>
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

2. 在控制台上填入 `android.intent.action.VIEW` 。

透传自定义内容

步骤1：发送端设置自定义内容

在发消息前设置每条消息的通知栏自定义内容。

> 注意：
> OPPO 要求自定义的数据必须是 json 格式。

- Android 端示例如下：

```
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);
```

- 服务端示例请参见 OfflinePushInfo 的格式示例。

步骤2：接收端获取自定义内容

在控制台选择设置点击通知 打开应用 、 打开应用内指定界面 的 Intent action 选项或者 Activity 选项后，当点击通知栏的消息时，客户端在相应的 `Activity` 中获取自定义内容。

```
Bundle bundle = intent.getExtras();
Set<String> set = bundle.keySet();
if (set != null) {
for (String key : set) {
// 其中 key 和 value 分别为发送端设置的 extKey 和 ext content
String value = bundle.getString(key);
Log.i("oppo push custom data", "key = " + key + ":value = " + value);
}
}
```

常见问题

能否自定义配置推送提示音？

目前 OPPO 推送不支持自定义的提示音。

收不到推送时，如何排查问题？

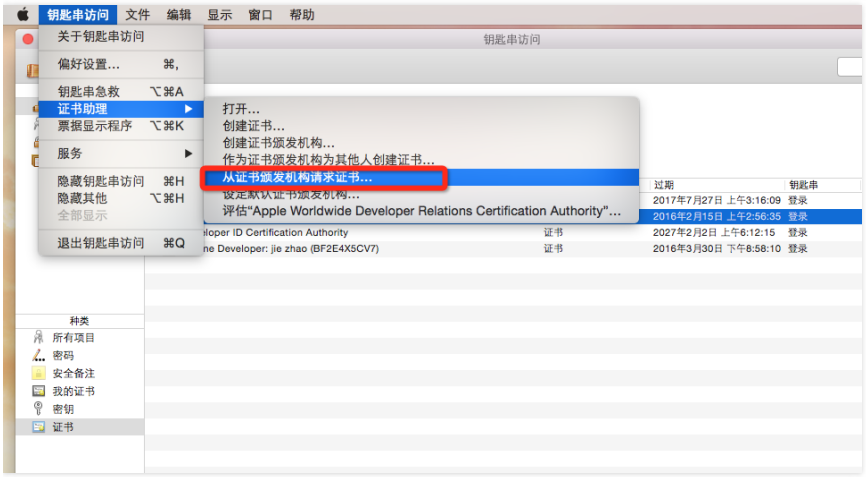
1. 任何推送都不是100%必达，厂商推送也不例外。因此，若在快速、连续的推送过程中偶现一两条推送未通知提醒，通常是由厂商推送频控的限制引起。
2. 按照推送的流程，确认 OPPO 推送证书信息是否正确配置在 即时通信 IM 控制台 中。
3. 确认您的项目 集成 OPPO 推送 SDK 的配置正确，并正常获取到了 regId。
4. 确认您已将正确的 推送信息上报 至即时通信 IM 服务端。
5. 在设备中手动 kill App，发送若干条消息，确认是否能在一分钟内接收到通知。

离线推送（iOS）
Apple 推送证书申请

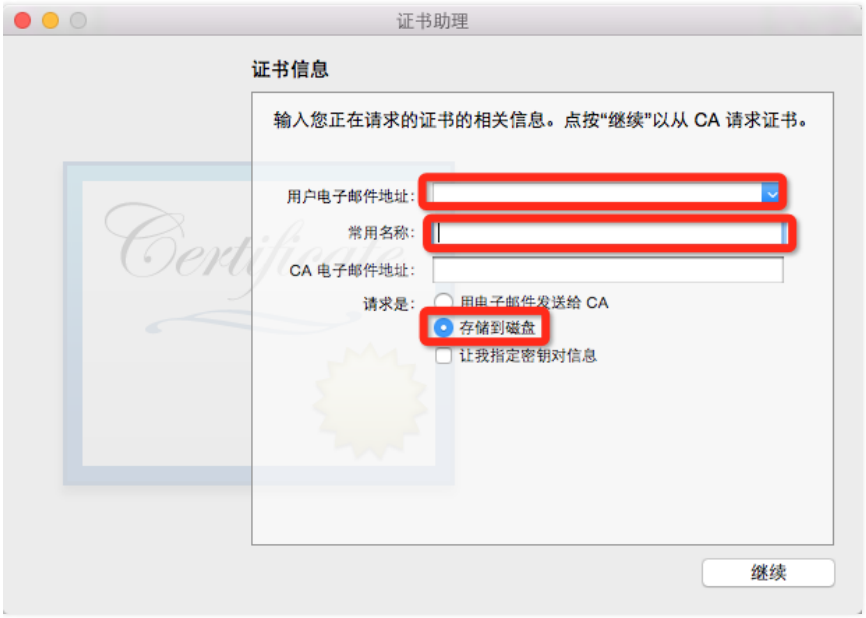
最近更新时间：2019-10-14 18:20:38

生成 CSR 文件

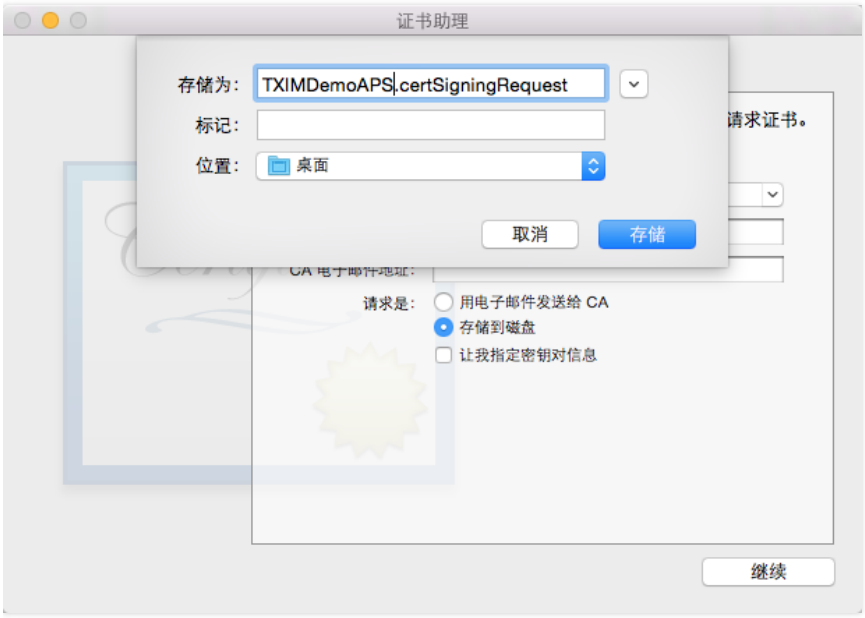
生成 Certificate Signing Request(CSR)：



填写您的邮箱（这个邮箱是申请 AppID 的付费帐号）和常用名称（一般默认是计算机名，不用更改），并选择保存到硬盘：



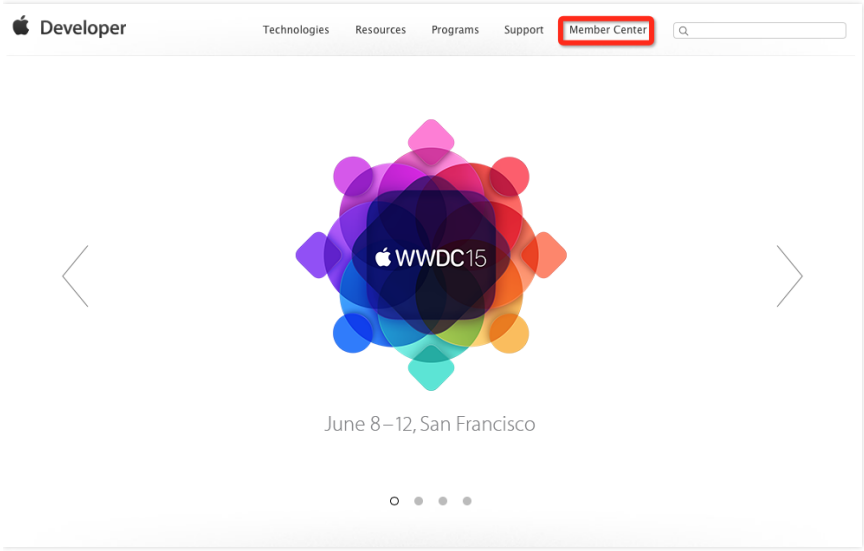
单击继续：



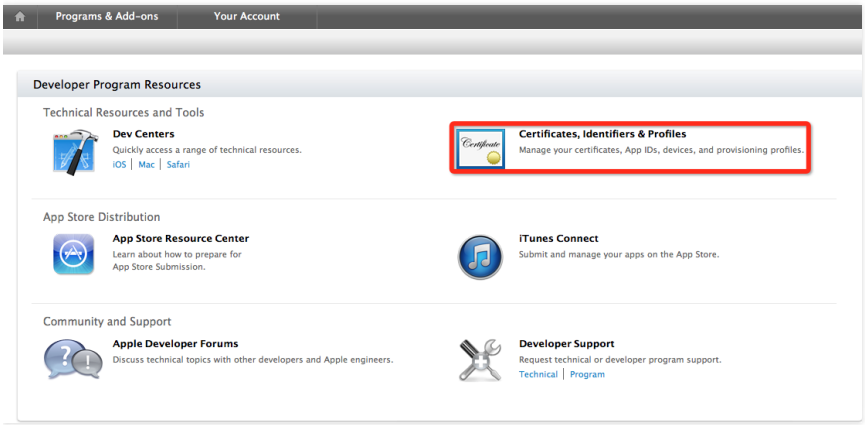
已经在本地生成了一个 TXIMDemoAPS.certSigningRequest 的 CSR 文件。

生成 App ID

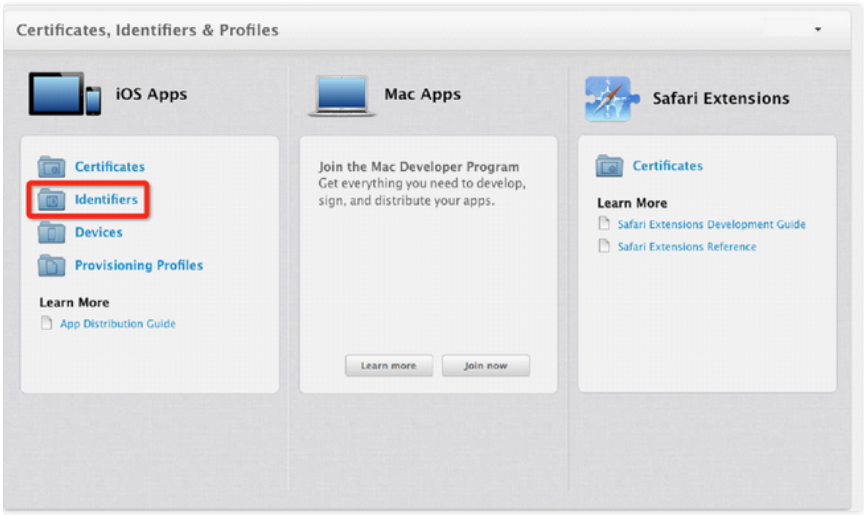
登录 developer.apple.com，选择 Member Center ：



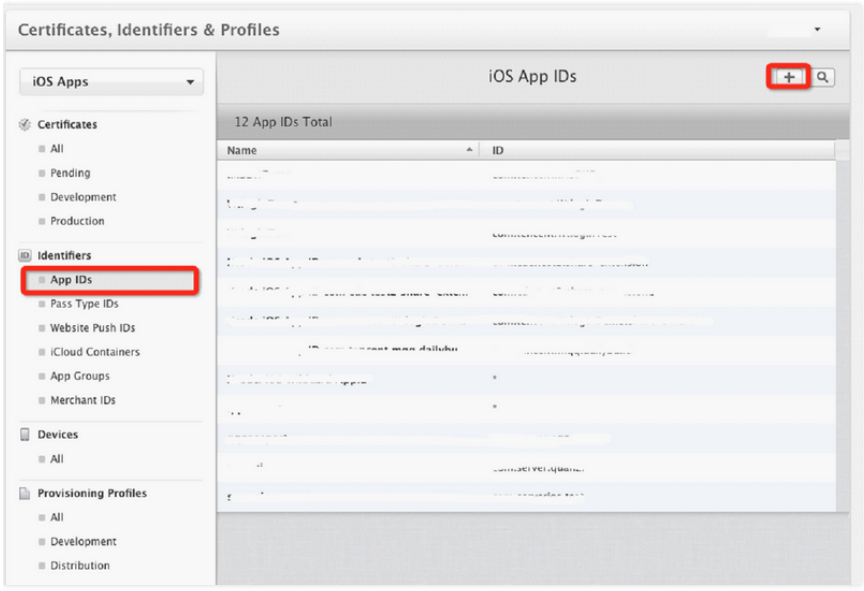
进入后选择 Certificates，Identifiers & Profiles：



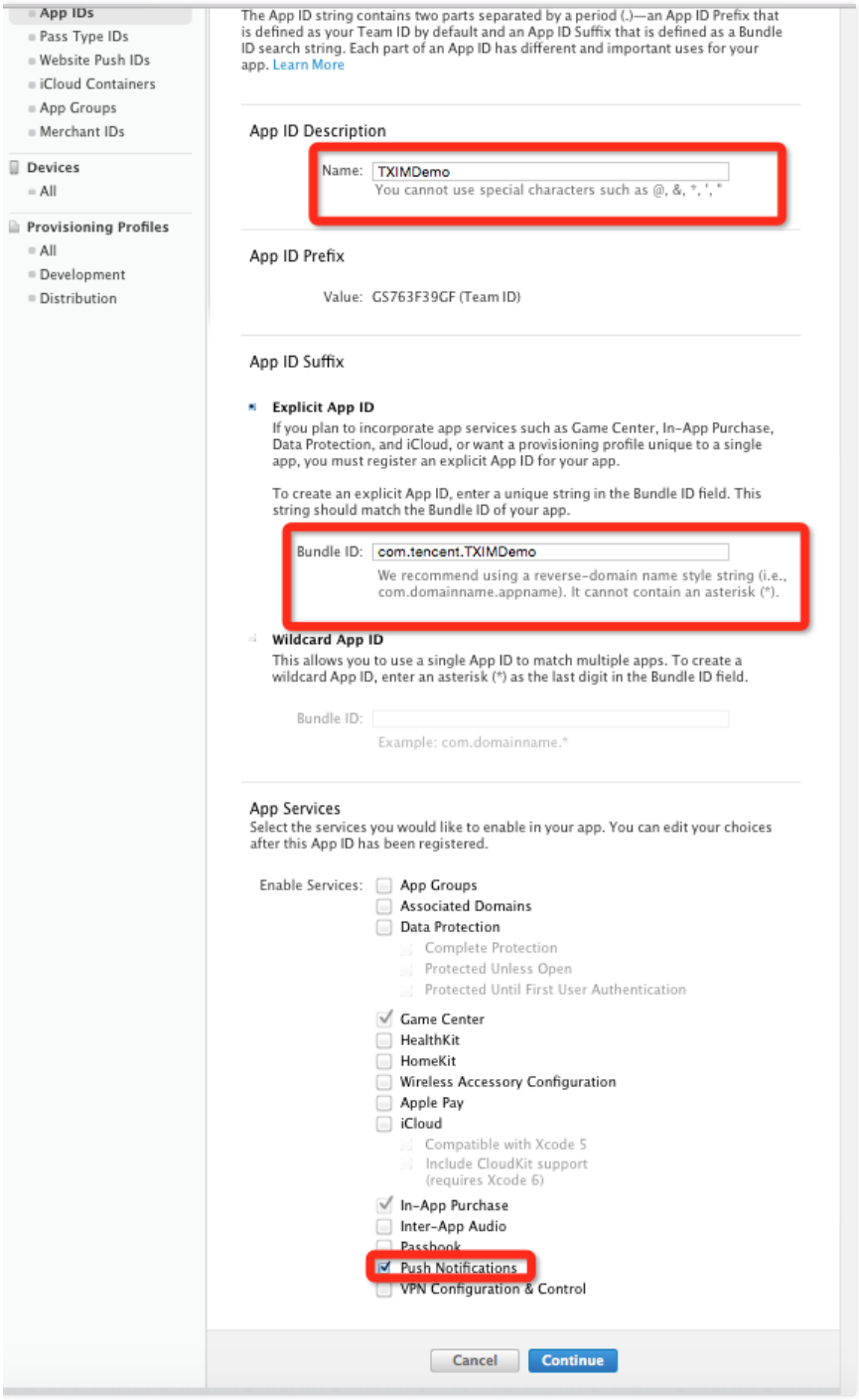
再选择 Identifiers 进行到 Identifiers 管理页：



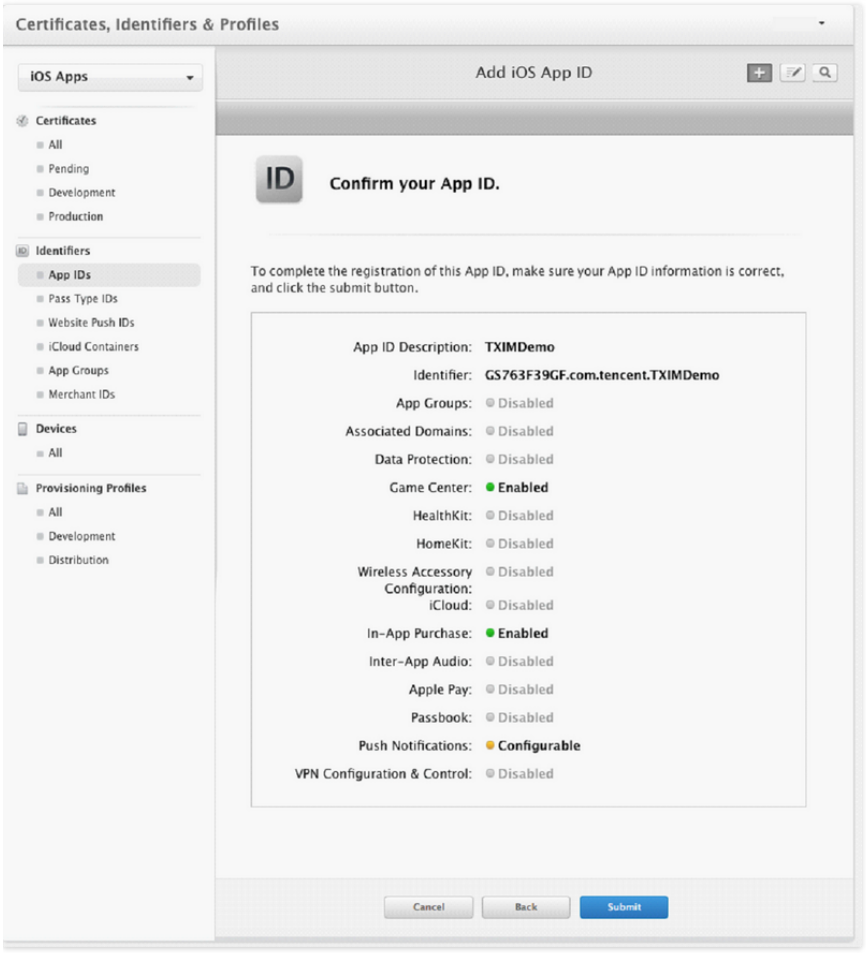
生成 App ID：
Identifiers 的右侧列表中，如果您已经配置您的应用，跳到第3步，否则单击“+”号添加 App ID。



输入您的 App ID 描述信息，可以输入工程名；Bunble ID（在工程的 General 信息中），一般格式为 com.youcompany.youprojname，选择需要支持 Push Notification，Continue：

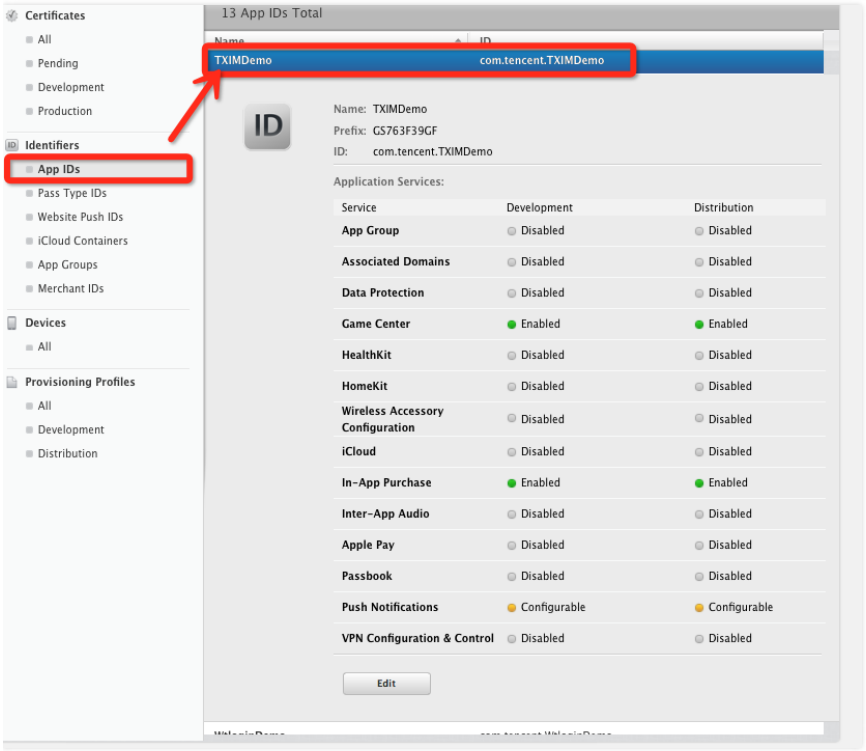


Submit 提交：

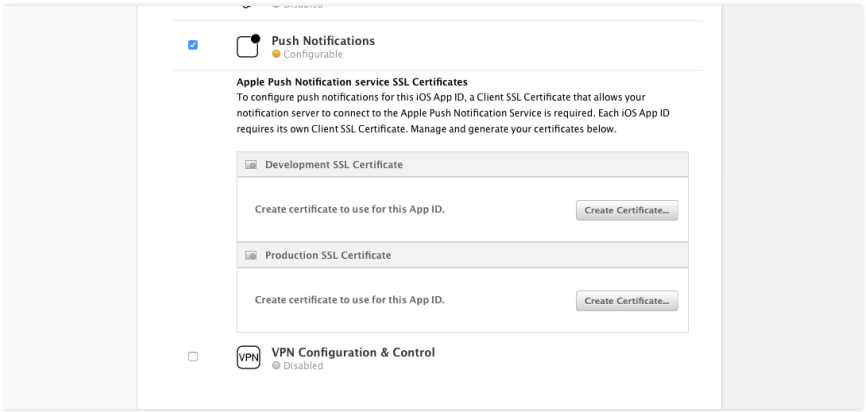


创建 App 的 APS 证书

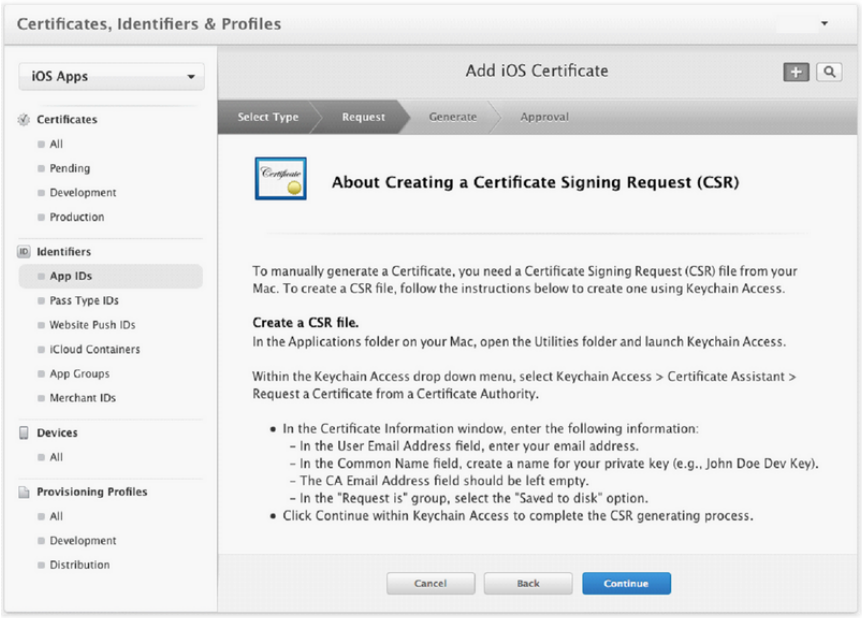
回到 App IDs 选择您需要推送的 App。展开单击“Edit”：



找到最底部的 Push Notifications，单击“Create Cerifcate...”创建 push 证书，这里**开发环境的证书和发布环境的证书需要分别创建，也就是相同的流程要走两遍**：

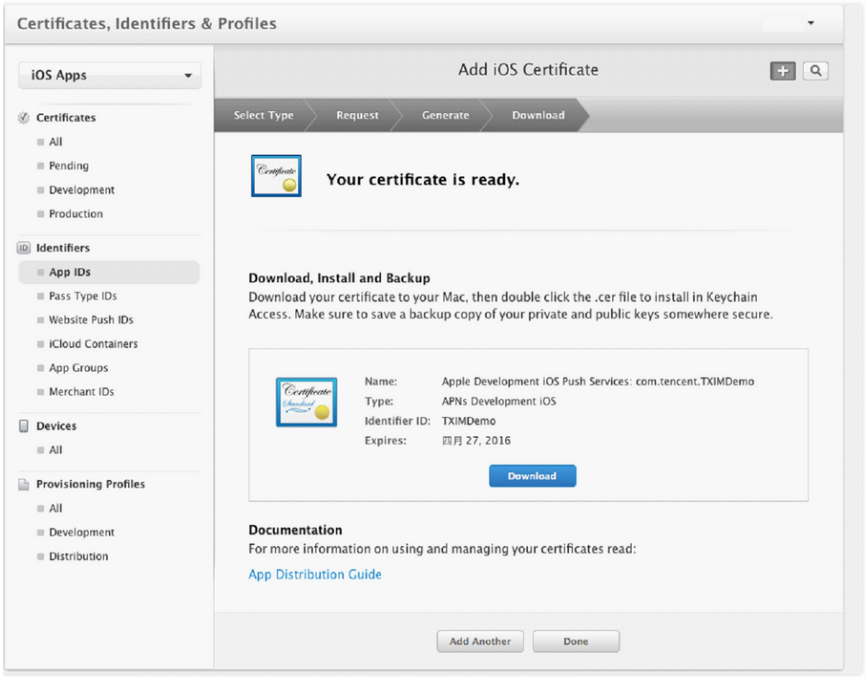


单击 Continue 继续：

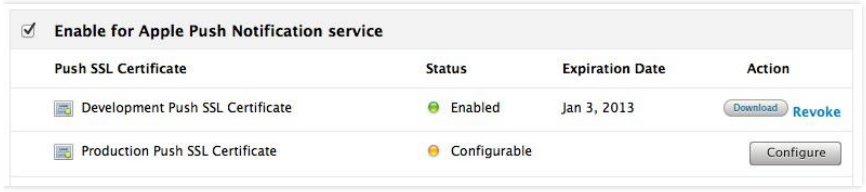


上传创建好 CSR 文件（参照第一节）xxx.certSigningRequest（例子为：TXIMDemoAPS.certSigningRequest），单击“Generate”：

aps 证书创建成功了，单击 Download 下载到本地。（文件名：开发版本为 aps_development.cer，发布版本为 aps.cer）：



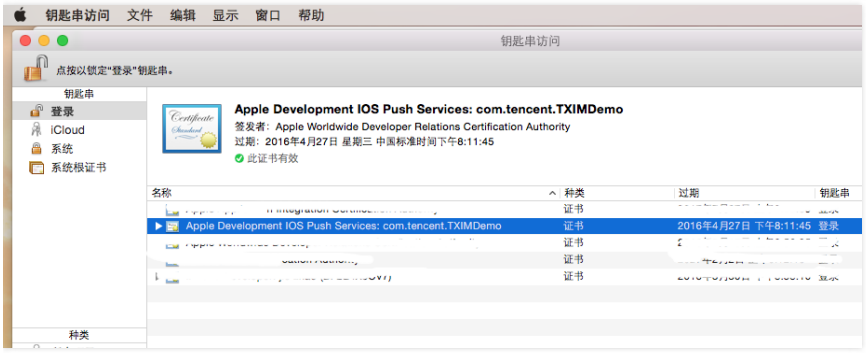
单击 Done，您会发现该环境的 push 配置状态已经 Enabled：



注意：有的 App ID 的 Apple Push Notification service 列是灰色的，并且不允许使用 Configure 按钮，这是因为 APNS 不支持带通配符的 App ID。

生成 Push 证书

导入证书
双击上一节下载的文件（aps_development.cer 和 aps.cer）将其安装到电脑，在"钥匙串访问"中，可以看到已经导入的证书。

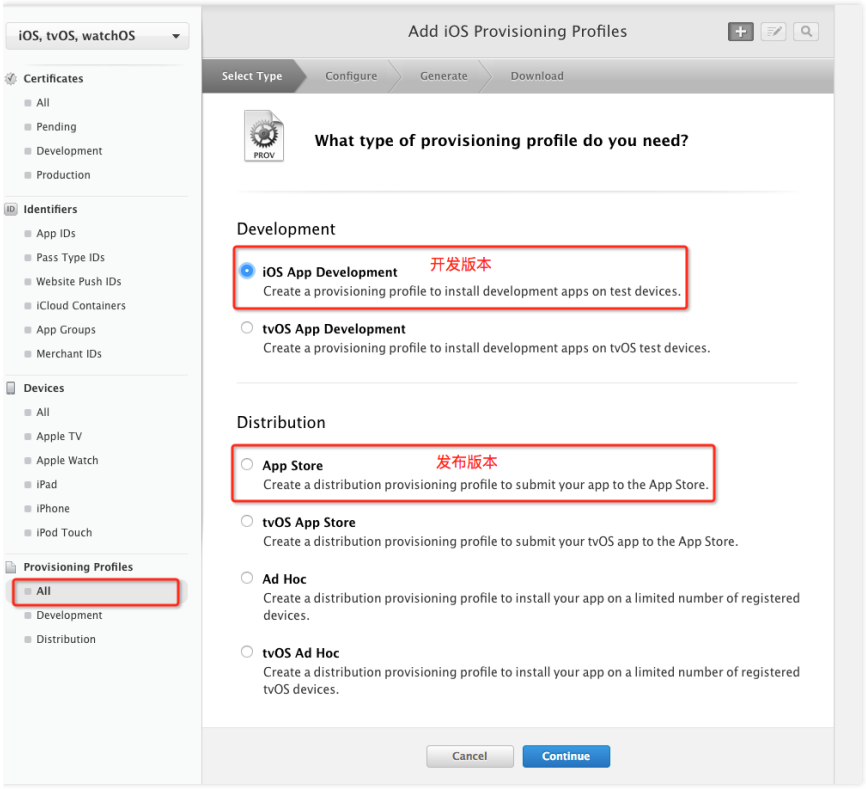


右键选择导出为 p12 文件，（例：存储为 TXIMDemoAPS.p12）：

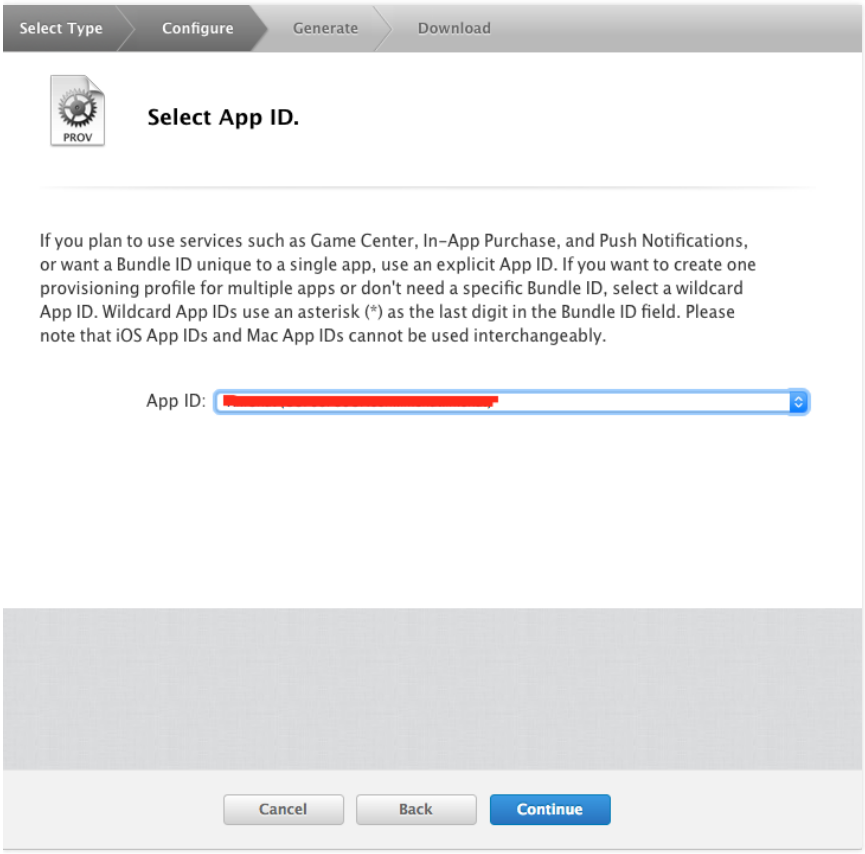
注意：开发版本证书只有在 debug 模式下开发的时候会生效，正式发布版本的证书，一定要使用正式版本的证书。

生成 Provisioning Profile 文件（PP 文件）

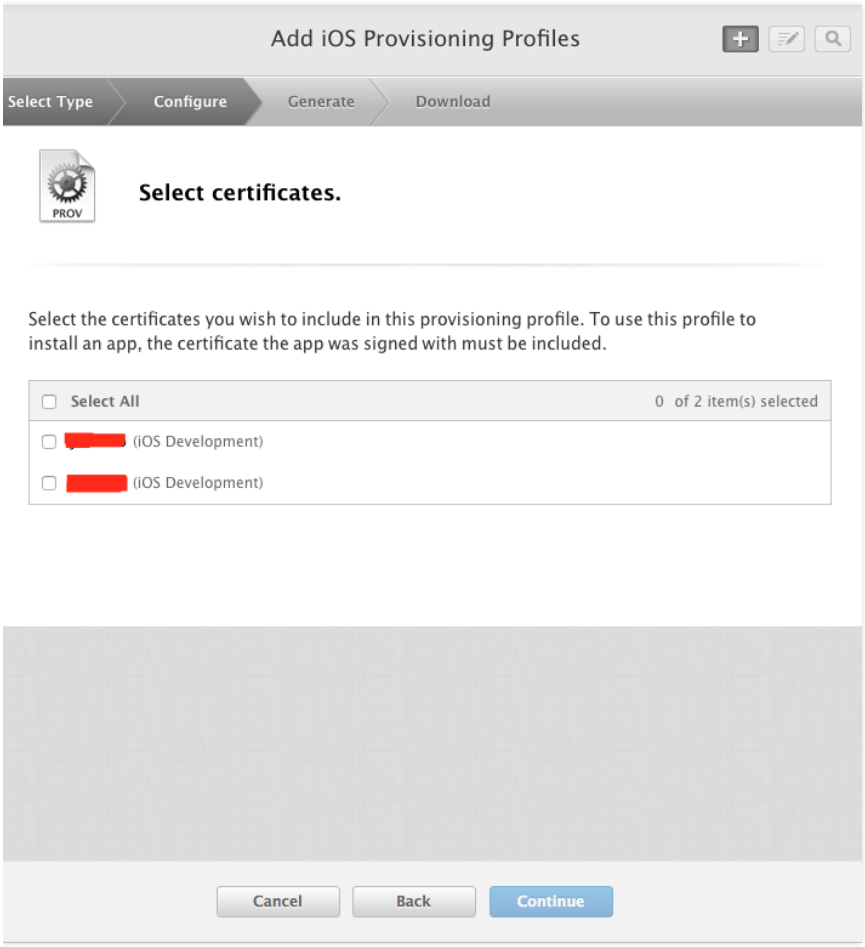
生成对应的描述文件，这里演示开发版描述文件的创建（发布版本的创建流程一样，用户可以自行操作），单击"Continue"



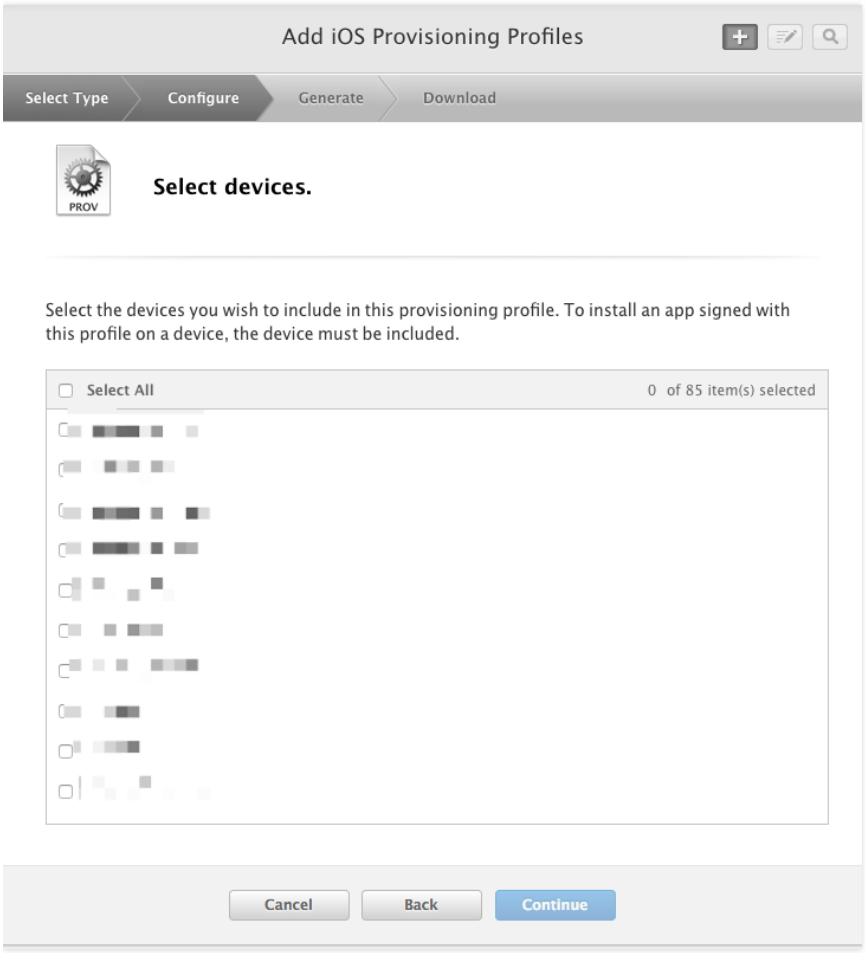
选择3.3步骤中创建推送证书那个 App ID，单击"Continue"，



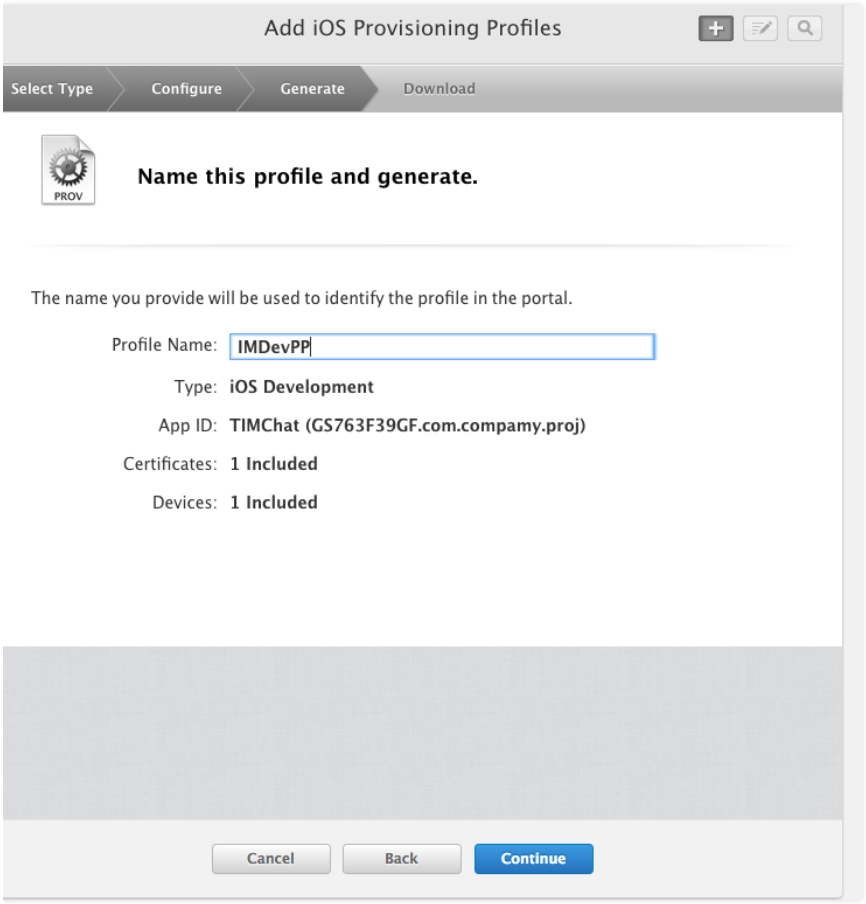
选择3.3中创建的开发版推送证书（创建发布版描述文件时，选择3.3中创建的发布版推送证书），单击"Continue"，



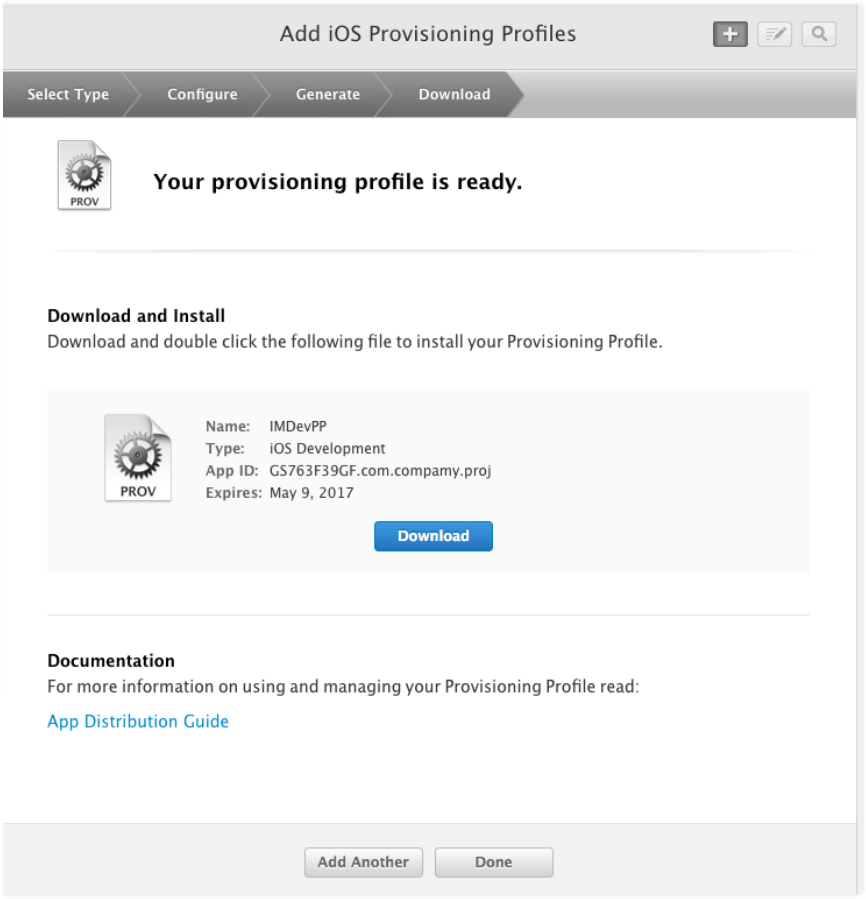
选择需要加入开发的设备，只有加入了的设备才能进行真机调试，创建发布版本时没有这个步骤，单击"Continue",



输入 PP 文件的名称，这里以 IMDevPP 为例



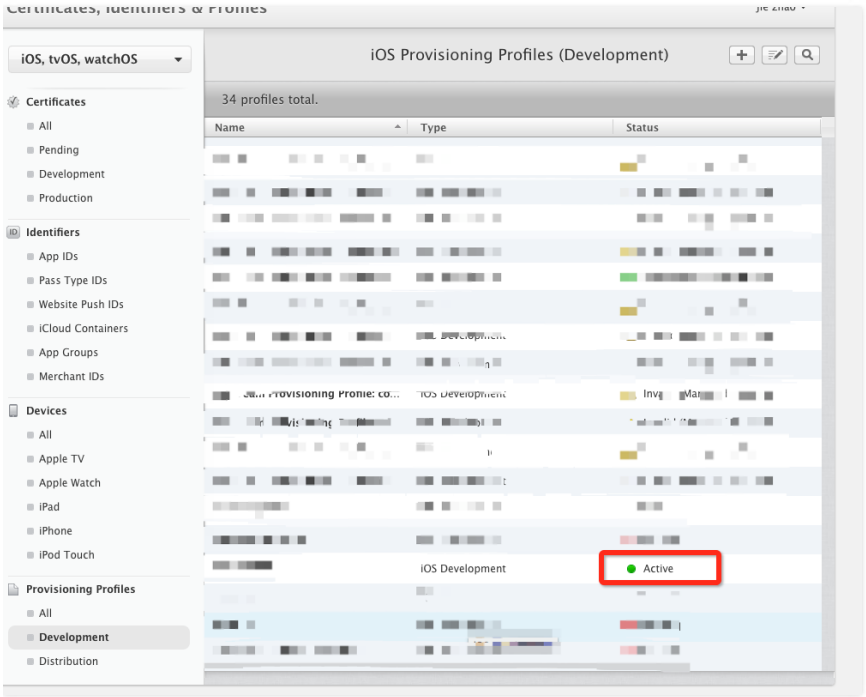
生成 PP 文件完成



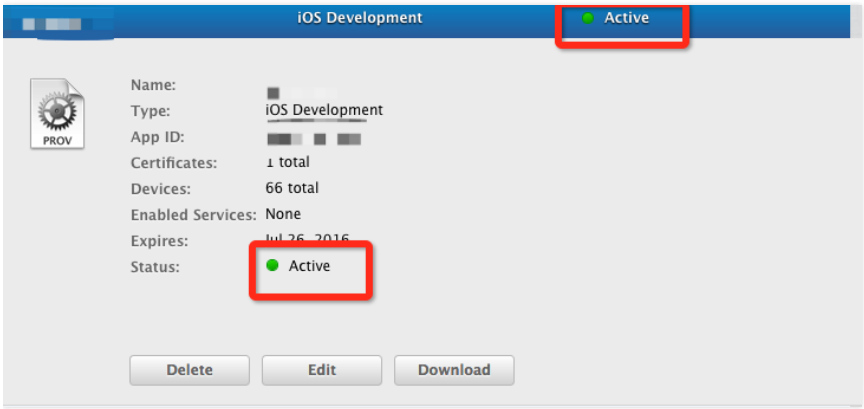
注：以上所有的步骤，除了生成 p12 文件时必须要下载证书安装到本地，其它生成的文件都可以不下载到本地。

检查生成的 PP 文件

查看 pp 文件的状态一定要为 Active

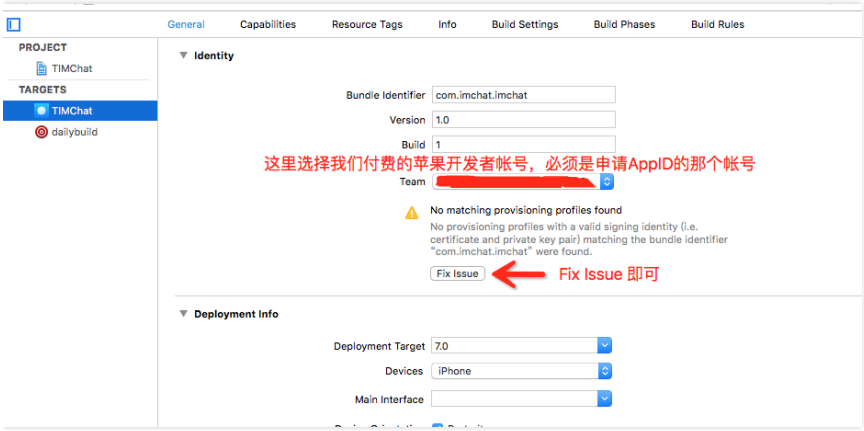


单击 PP 文件进入详细资料页面，状态也要是 Active



Xcode 中的配置

新版 Xcode 已经不需要手动配置证书和描述文件了，只需在 General 中选择正确的 Team，Fix Issue 即可，这也是上面所说的不用下载证书到本地安装的原因



离线推送（iOS）

最近更新时间：2020-02-26 01:11:25

配置离线推送

如想要接收 APNs 离线消息通知，需要在腾讯云管理平台提交 Push 证书，在客户端每次登录时，获取并通过 API 接口上报 Token。APNs 推送功能只用于通知用户，如果 App 在前台，以 `onNewMessage` 回调获取新消息为准，`didReceiveRemoteNotification` 获取到的消息可以忽略。详细推送原理请参见 Apple Push Notification Service。

申请 APNs 证书

申请 APNs 证书的具体操作步骤请参见 Apple 推送证书申请。

上传证书到控制台

1. 登录 即时通信 IM 控制台。
2. 单击目标应用卡片，进入应用的基础配置页面。
3. 单击【iOS平台推送设置】右侧的【添加证书】。
4. 选择证书类型，上传 iOS 证书（p.12），设置证书密码，单击【确认】。

> 注意：
> - 上传证书名最好使用全英文（尤其不能使用括号等特殊字符）。
> - 上传证书需要设置密码，无密码收不到推送。
> - 发布 App Store 的证书需要设置为生产环境，否则无法收到推送。
> - 上传的 p12 证书必须是自己申请的真实有效的证书。

5. 待推送证书信息生成后，记录证书的 ID。

客户端实现 APNs 推送

客户端要实现接收 APNs 推送，需要实现以下几个步骤，更详细的操作步骤可参考 即时通信 iOS IM SDK 离线推送视频。

向苹果后台请求 DeviceToken

```
- (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)];
}
}
/**
* 在 AppDelegate 的回调中会返回 deviceToken，需要在登录后上报给腾讯云后台
/**
-(void)application:(UIApplication *)app didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)deviceToken
{
//记录下 Apple 返回的 deviceToken
_deviceToken = deviceToken;
}
```

登录 IM SDK 后上传 Token 到腾讯云

> 注意：

busiID 需要与控制台分配的证书 ID 保持一致。

```
__weak typeof(self) ws = self;
//这里如果使用了 TUIKit，请在 TUKit 登录回调里面设置 Token，如果没有使用，请在 TIMManager 的 login 回调里面设置 Token。
[[TUIKit sharedInstance] loginKit:identifier userSig:userSig succ:^{
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。*/
#if kAppStoreVersion
// App Store 版本
#if DEBUG
param.busiId = 2383;
#else
param.busiId = 2382;
#endif
#else
//企业证书 ID
param.busiId = 2516;
#endif
[param setToken:ws.deviceToken];
[[TIMManager sharedInstance] setToken:param succ:^{
NSLog(@"-----> 上传 token 成功 ");
} fail:^(int code, NSString *msg) {
NSLog(@"-----> 上传 token 失败 ");
}];
} fail:^(int code, NSString *msg) {
NSLog(@"登录失败！");
}];
}
```

App 进入后台时上报切后台事件

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
__block UIBackgroundTaskIdentifier bgTaskID;
bgTaskID = [application beginBackgroundTaskWithExpirationHandler:^ {
//不管有没有完成，结束 background_task 任务
[application endBackgroundTask: bgTaskID];
bgTaskID = UIBackgroundTaskInvalid;
}];
//获取未读计数
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);
}];
}
```

App 进入前台时上报切前台事件

```
- (void)applicationDidBecomeActive:(UIApplication *)application {
[[TIMManager sharedInstance] doForeground:^() {
```

```
NSLog(@"doForegroud Succ");
} fail:^(int code, NSString * err) {
NSLog(@"Fail: %d->%@", code, err);
}];
}
```

推送格式

推送格式示例如下图所示。

通用推送规则

对于单聊消息，APNs 推送规则如下，其中昵称是发送方用户昵称，如果未设置昵称，则只显示内容。

```
昵称:内容
```

对于群聊消息，APNs 推送规则如下，其中名称为群名片或者发送者昵称，优先级为 `群名片` > `群名` 。

```
名称(群名):内容
```

不同类型消息推送规则

APNs 推送内容部分由消息体中各个 `Elem` 内容组成，不同 `Elem` 的离线消息展示效果如下表所示。

参数	说明
文本 Elem	直接显示内容
语音 Elem	显示 [语音]
文件 Elem	显示 [文件]
图片 Elem	显示 [图片]
自定义 Elem	显示 `desc` 字段内容，若自定义消息 `desc` 为空，则该消息不进行离线推送

多 App 互通

如果将多个 App 中的 `SDKAppID` 设置为相同值，则可以实现多 App 互通。不同 App 需要使用不同的推送证书，您需要为每一个 App 申请 APNs 证书 并完成 离线推送配置。

推送提示音

设置自定义推送提示音

IM SDK 提供了设置用户声音的接口，可按需自定义设置单聊消息提示音和群组消息提示音，也可在用户级别设置是否接收推送。

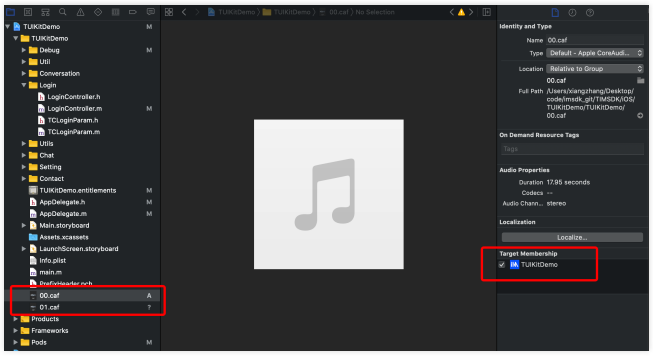
```
/**
* APNs 配置
*/
@interface TIMAPNSConfig : NSObject
/**
* 是否开启推送：0-不进行设置 1-开启推送 2-关闭推送
*/
@property(nonatomic,assign) uint32_t openPush;
/**
* C2C 消息声音,不设置传入 nil
*/
@property(nonatomic,retain) NSString * c2cSound;
/**
* Group 消息声音,不设置传入 nil
*/
@property(nonatomic,retain) NSString * groupSound;
@end
@interface TIMManager : NSObject
/**
* 设置 APNS 配置
*
* @param config APNS 配置
* @param succ 成功回调
* @param fail 失败回调
*
* @return 0 成功
*/
-(int) setAPNS:(TIMAPNSConfig*)config succ:(TIMSucc)succ fail:(TIMFail)fail;
@end
```

参数说明

参数	说明
config	openPush：是否开启推送。0表示不进行设置，1表示开启推送，2表示关闭推送 c2cSound：表示单聊消息提示音，需设置为文件名（含后缀） groupSound：表示群组消息提示音，需设置为文件名（含后缀）
succ	成功回调
fail	失败回调

操作步骤

1. 把音频文件集成到工程中。



2. 登录成功后调用 `setToken` 接口设置 token 和 busiID 信息。
3. 调用 `setAPNS` 接口设置音频文件信息。

> 说明：
> 只需要设置音频文件的**文件名称（含后缀）**即可。

```
[[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) {
```

获取推送消息提示音

如需获取推送消息提示音，可使用 `getAPNSConfig` 获取，此接口每次都从服务器同步数据，不会进行本地缓存。

```
@interface TIMManager : NSObject
/**
* 获取 APNS 配置
*
* @param succ 成功回调，返回配置信息
* @param fail 失败回调
*
* @return 0 成功
*/
-(int) getAPNSConfig:(TIMAPNSConfigSucc)succ fail:(TIMFail)fail;
@end
```

参数说明如下表所示：

参数	说明
succ	成功回调，返回 TIMAPNSConfig 结构体

参数	说明
fail	失败回调

自定义离线消息属性

如果需要定制每条消息的展示文本、扩展字段、提示音、是否推送属性，可以在消息设置 `TIMOfflinePushInfo` ，此条消息在推送时，会替换用户原有的默认属性，实现每条消息定制化推送。例如，填入 `kIOSOfflinePushNoSound` 到 `sound` 属性时接收端强制为静音提示。

```
/**
填入 sound 字段表示接收时不会播放声音
*/
extern NSString * const kIOSOfflinePushNoSound;
@interface TIMAndroidOfflinePushConfig : NSObject
/**
* 离线推送时展示标签
*/
@property(nonatomic,retain) NSString * title;
/**
* Android 离线 Push 时声音字段信息
*/
@property(nonatomic,retain) NSString * sound;
/**
* 离线推送时通知形式
*/
@property(nonatomic,assign) TIMAndroidOfflinePushNotifyMode notifyMode;
@end
@interface TIMIOSOfflinePushConfig : NSObject
/**
* 离线 Push 时声音字段信息
*/
@property(nonatomic,retain) NSString * sound;
/**
* 忽略 badge 计数
*/
@property(nonatomic,assign) BOOL ignoreBadge;
@end
@interface TIMOfflinePushInfo : NSObject
/**
* 自定义消息描述信息，做离线 Push 时文本展示
*/
@property(nonatomic,retain) NSString * desc;
/**
* 离线 Push 时扩展字段信息
*/
@property(nonatomic,retain) NSString * ext;
/**
* 推送规则标志
*/
@property(nonatomic,assign) TIMOfflinePushFlag pushFlag;
/**
* iOS 离线推送配置
*/
@property(nonatomic,retain) TIMIOSOfflinePushConfig * iosConfig;
/**
* Android 离线推送配置
*/
@property(nonatomic,retain) TIMAndroidOfflinePushConfig * androidConfig;
@end
```