即时通信 IM

无 UI 集成

产品文档

腾讯云

版权所有：腾讯云计算（北京）有限责任公司

【版权声明】

©2013-2024 腾讯云版权所有

本文档著作权归腾讯云单独所有，未经腾讯云事先书面许可，任何主体不得以任何形式复制、修改、抄袭、传播全部或部分本文档内容。

【商标声明】

及其它腾讯云服务相关的商标均为腾讯云计算（北京）有限责任公司及其关联公司所有。本文档涉及的第三方主体的商标，依法由权利人所有。

【服务声明】

本文档意在向客户介绍腾讯云全部或部分产品、服务的当时的整体概况，部分产品、服务的内容可能有所调整。您所购买的腾讯云产品、服务的种类、服务标准等应由您与腾讯云之间的商业合同约定，除非双方另有约定，否则，腾讯云对本文档内容不做任何明示或默示的承诺或保证。

文档目录

版权所有：腾讯云计算（北京）有限责任公司

- React Native
- 接收消息
  - Android&iOS&Windows&Mac
  - Web
  - Flutter
  - Unity
  - React Native
- 历史消息
  - Android&iOS&Windows&Mac
  - Web
  - Flutter
  - Unity
  - React Native
- 转发消息
  - Android&iOS&Windows&Mac
  - Web
  - Flutter
  - Unity
  - React Native
- 消息变更
  - Android&iOS&Windows&Mac
  - Web
  - Flutter
  - Unity
  - React Native
- 插入消息
  - Android&iOS&Windows&Mac
  - Flutter
  - React Native
- 删除消息
  - Android&iOS&Windows&Mac
  - Web
  - Flutter
  - Unity
  - React Native
- 清空消息
  - Android&iOS&Windows&Mac
  - Web

版权所有：腾讯云计算（北京）有限责任公司

- Flutter
- Unity
- React Native

撤回消息

- Android&iOS&Windows&Mac
- Web
- Flutter
- Unity
- React Native

在线消息

- Android&iOS&Windows&Mac
- Web
- Flutter
- Unity
- React Native

已读回执

- Android&iOS&Windows&Mac
- Web
- Flutter
- Unity
- React Native

查询消息

- Android&iOS&Windows&Mac
- Web
- Flutter
- Unity
- React Native

群 @ 消息

- Android&iOS&Windows&Mac
- Flutter
- Unity
- React Native

群定向消息

- Android&iOS&Windows&Mac
- Web
- Flutter
- Unity
- React Native

- 消息免打扰
  - Android&iOS&Windows&Mac
  - Web
  - Flutter
  - Unity
  - React Native
- 消息扩展
  - Android&iOS&Windows&Mac
  - Web&小程序&uni-app
  - React Native
  - Flutter
  - Unity
- 消息回应
  - Android&iOS&Windows&Mac
  - Web
  - React Native
- 消息翻译
  - Android&iOS&Windows&Mac
  - Web
- 语音转文字
  - Android&iOS&Windows&Mac
  - Web
- 消息置顶
  - Android&iOS&Windows&Mac

会话相关

- 会话介绍
  - Android&iOS&Windows&Mac
  - Web
  - Flutter
  - Unity
  - React Native
- 会话列表
  - Android&iOS&Windows&Mac
  - Web
  - Flutter
  - Unity
  - React Native
- 获取会话

版权所有：腾讯云计算（北京）有限责任公司

- Android&iOS&Windows&Mac
- Web
- Flutter
- Unity
- React Native

会话未读数

- Android&iOS&Windows&Mac
- Web
- Flutter
- Unity
- React Native

置顶会话

- Android&iOS&Windows&Mac
- Web
- Flutter
- Unity
- React Native

删除会话

- Android&iOS&Windows&Mac
- Web
- Flutter
- Unity
- React Native

会话草稿

- Android&iOS&Windows&Mac
- Web
- Flutter
- Unity
- React Native

会话标记

- Android&iOS&Windows&Mac
- Web
- Flutter
- Unity

会话分组

- Android&iOS&Windows&Mac
- Web
- Flutter

版权所有：腾讯云计算（北京）有限责任公司

- 社群管理
  - Android&iOS&Windows&Mac
  - Web
  - Flutter
  - React Native
- 权限组
  - Android&iOS&Windows&Mac

用户管理

- 用户资料
  - Android&iOS&Windows&Mac
  - Web
  - Flutter
  - Unity
  - React Native
- 用户状态
  - Android&iOS&Windows&Mac
  - Web
  - Unity
- 好友管理
  - Android&iOS&Windows&Mac
  - Web
  - Flutter
  - Unity
  - React Native
- 好友分组
  - Android&iOS&Windows&Mac
  - Web
  - Flutter
  - Unity
  - React Native
- 黑名单
  - Android&iOS&Windows&Mac
  - Web
  - Flutter
  - Unity
  - React Native
- 关注&粉丝
  - Android&iOS&Windows&Mac

版权所有：腾讯云计算（北京）有限责任公司

- 离线推送
  - Android
  - iOS
    - APNs
    - VoIP
  - Flutter
  - React Native
- 本地搜索
  - 搜索消息
    - Android&iOS&Windows&Mac
    - Web
    - Flutter
    - Unity
    - React Native
  - 搜索好友
    - Android&iOS&Windows&Mac
    - Flutter
    - Unity
    - React Native
  - 搜索群组
    - Android&iOS&Windows&Mac
    - Flutter
    - Unity
    - React Native
  - 搜索群成员
    - Android&iOS&Windows&Mac
    - Flutter
    - Unity
    - React Native
- 信令相关
  - 信令管理
    - Android&iOS&Windows&Mac
    - iOS
    - Flutter
    - React Native
    - Windows

无 UI 集成
集成SDK
Android

最近更新时间：2024-07-05 15:24:08

本文主要介绍如何快速将 Chat SDK 集成到您的 Android 项目中。

开发环境要求

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

集成 SDK（aar）

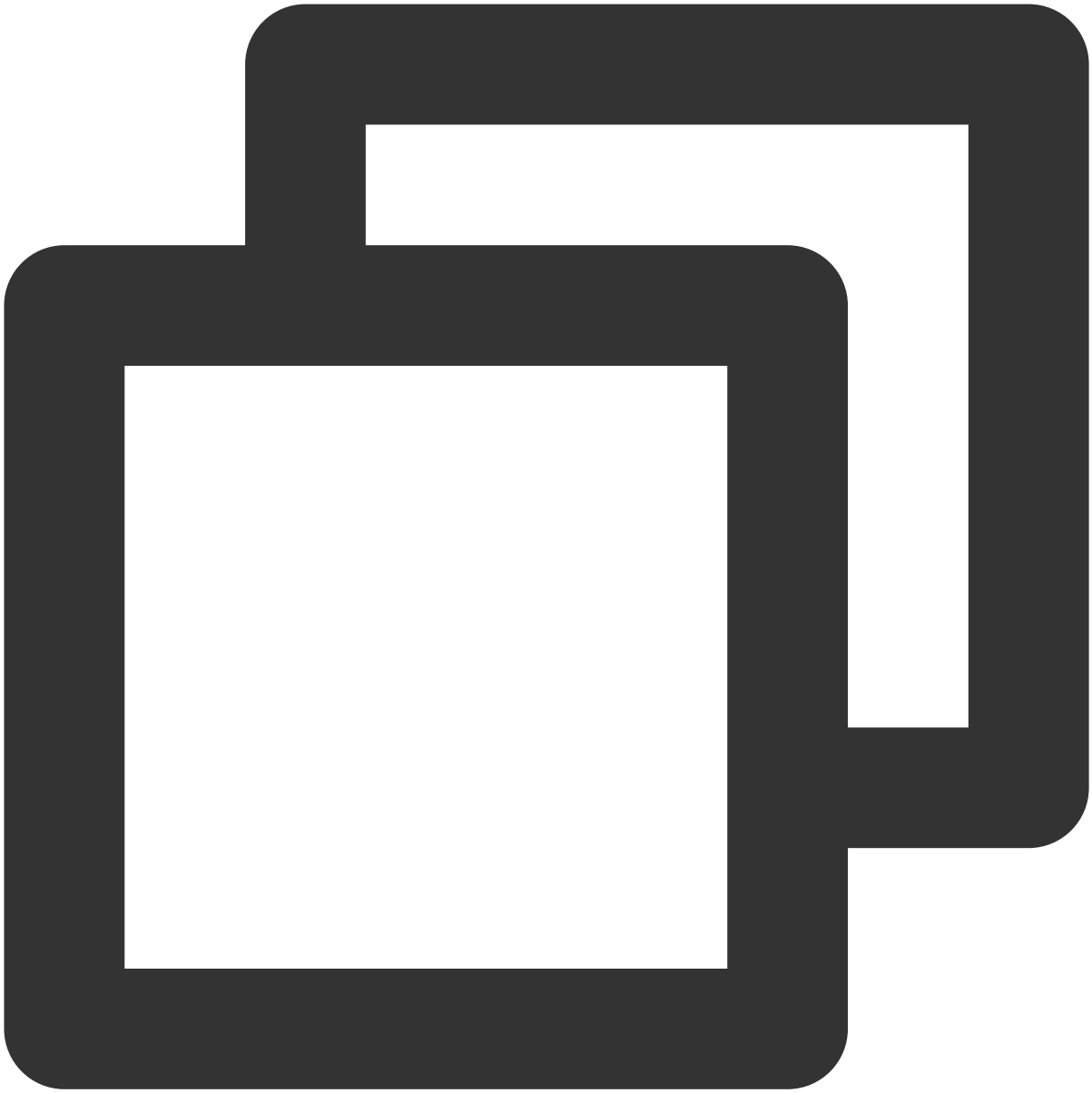
您可以选择使用 Gradle 自动加载的方式，或者手动下载 aar 再将其导入到您当前的工程项目中。

方法一：自动加载（aar）

您可以通过配置 gradle 自动下载更新已经发布到 Maven Central 库的 Chat SDK。
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：

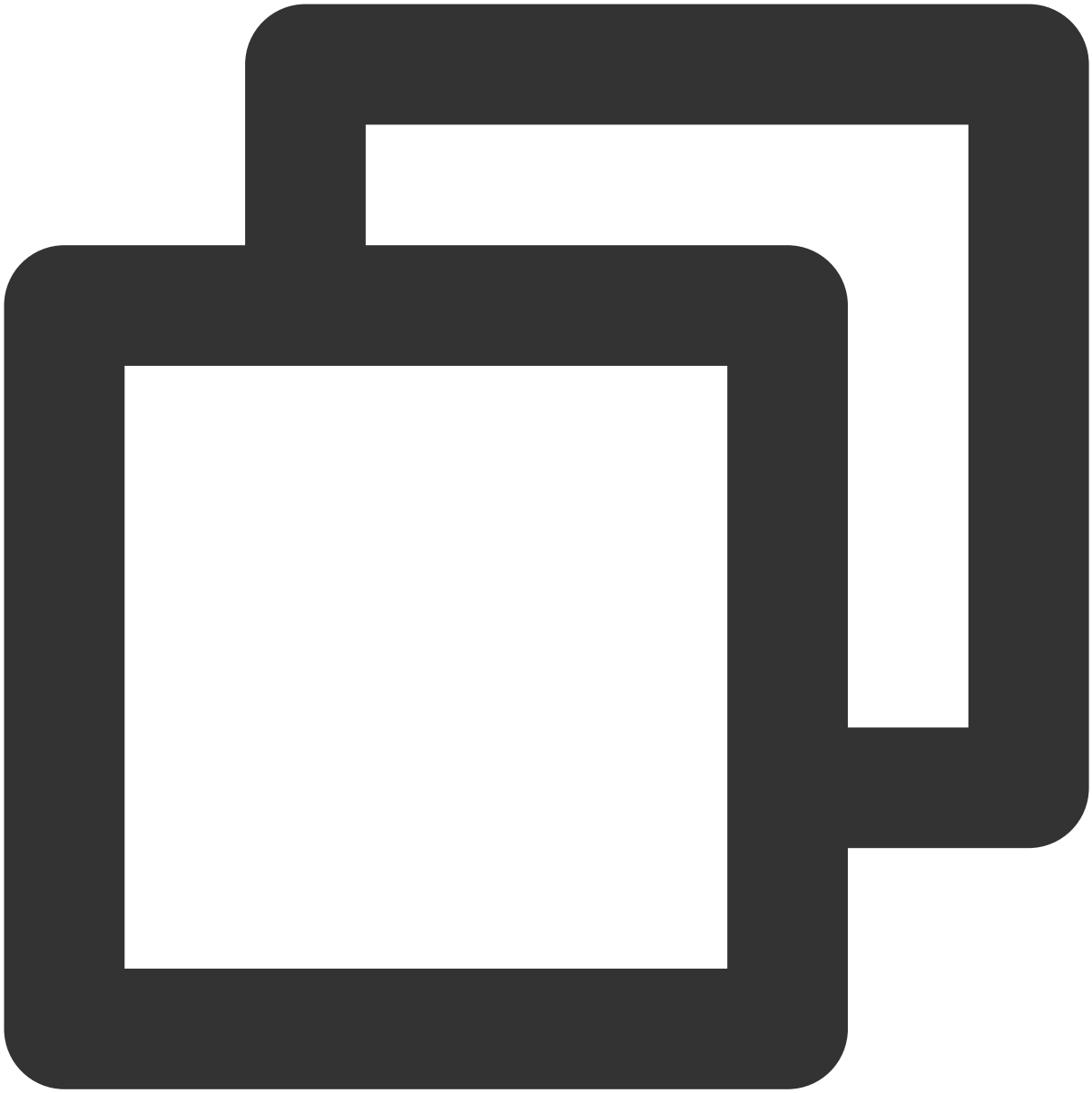
步骤1：添加 SDK 依赖

1. 找到 app 的 build.gradle，首先在 repositories 中添加 mavenCentral() 的依赖：



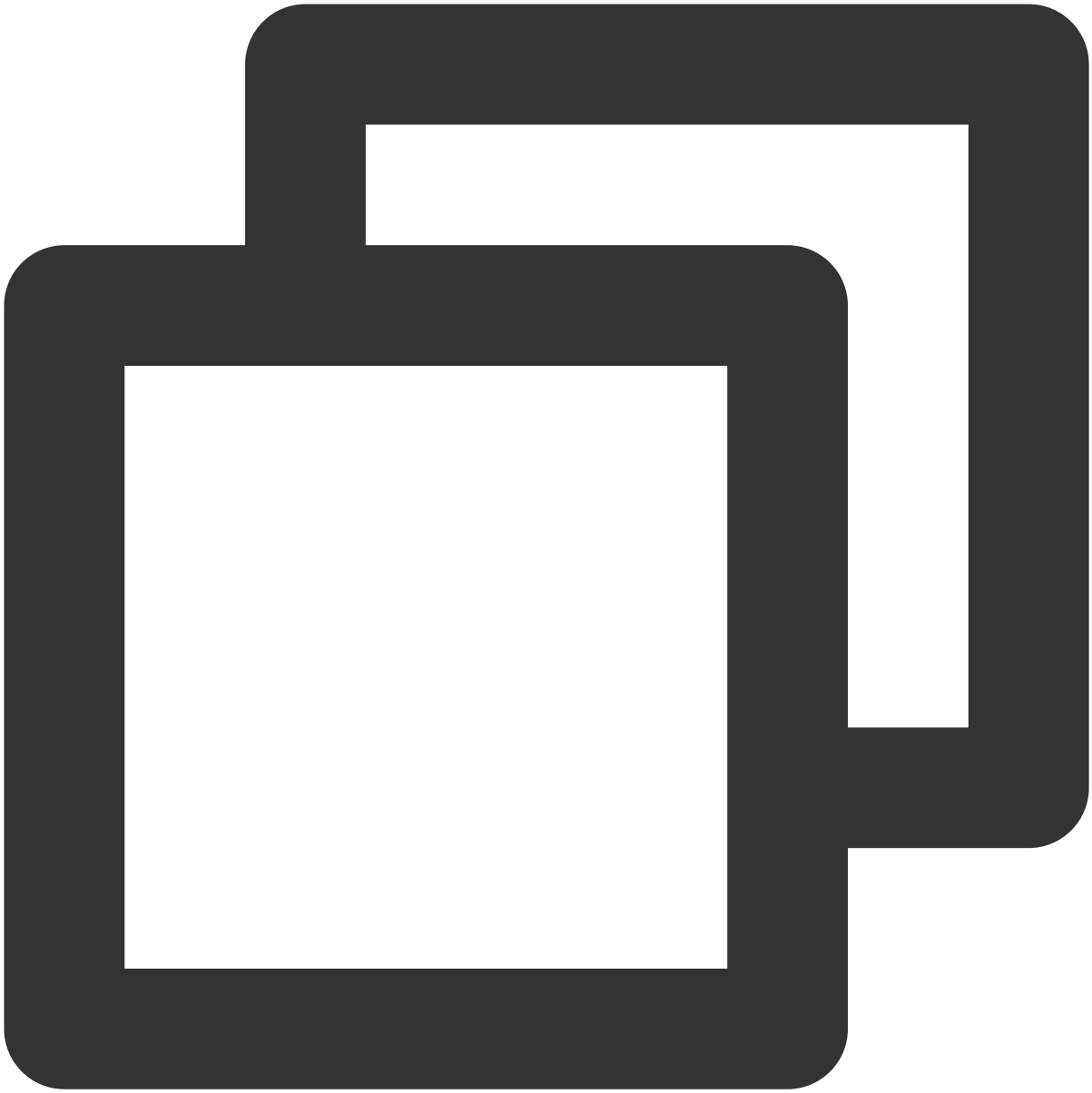
```
repositories {
    google()
    jcenter()
    // 增加 mavenCentral 仓库
    mavenCentral()
}
```

2. 然后在 dependencies 中添加 Chat SDK 的依赖：



```
dependencies {
    // 添加 Chat SDK，推荐填写最新的版本号
    api 'com.tencent.imsdk:imsdk-plus:版本号'

    // 如果您需要添加 Quic 插件，请取消下一行的注释（注意：要求插件版本号和 Chat SDK 版本号相同
    // api "com.tencent.imsdk:timquic-plugin:版本号"
}
```

其中的“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 `5.4.666` 为例：



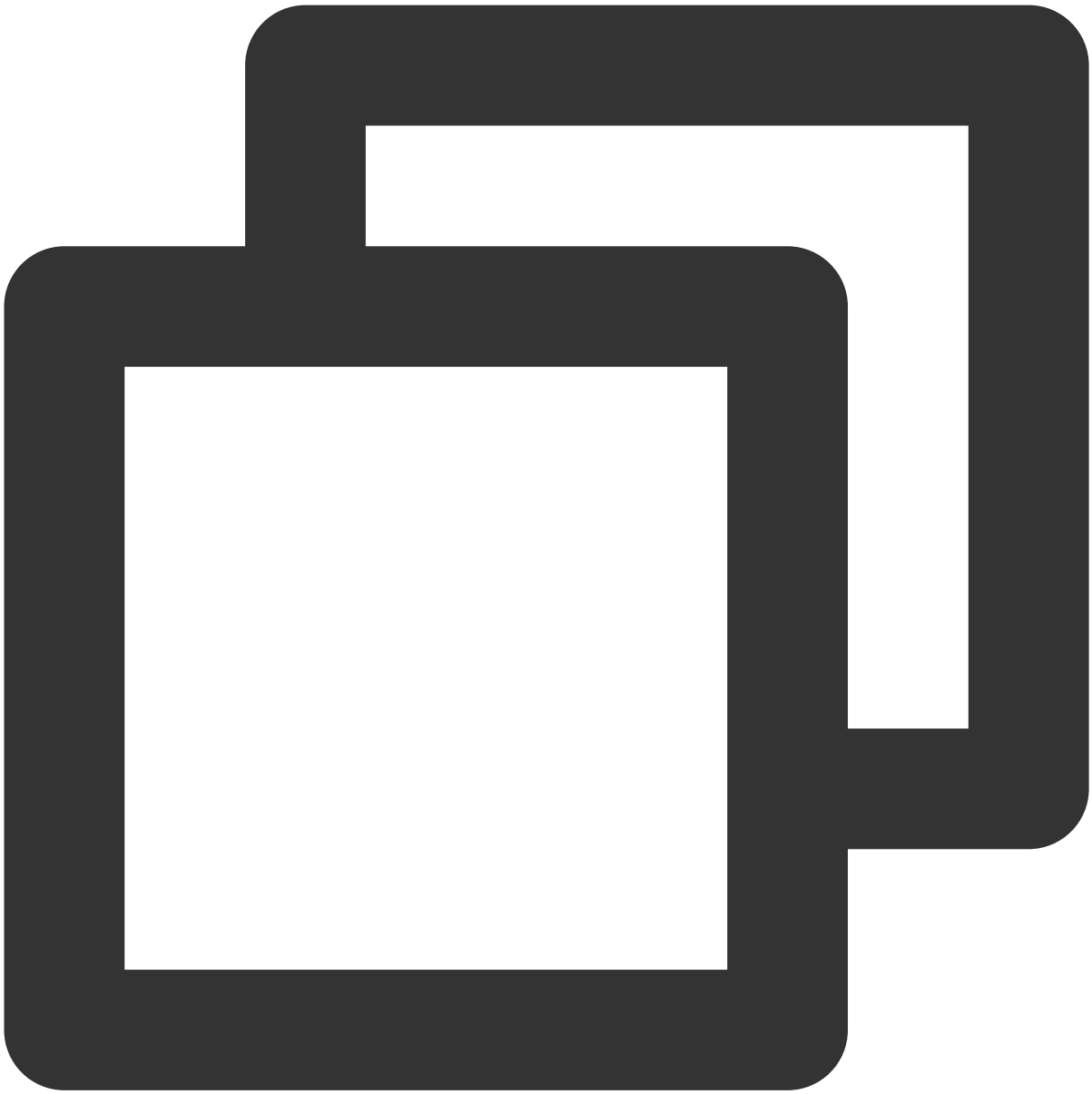
```
dependencies {
    api 'com.tencent.imsdk:imsdk-plus:5.4.666'
}
```

说明：

Quic 插件，提供 axp-quic 多路传输协议，弱网抗性更优，网络丢包率达到 70% 的条件下，仍然可以提供服务。仅对进阶版用户开放，请 购买进阶版 后可使用。为确保功能正常使用，请将 Chat SDK 更新至 7.7.5282 及其以上的版本。

步骤2：指定 App 使用架构

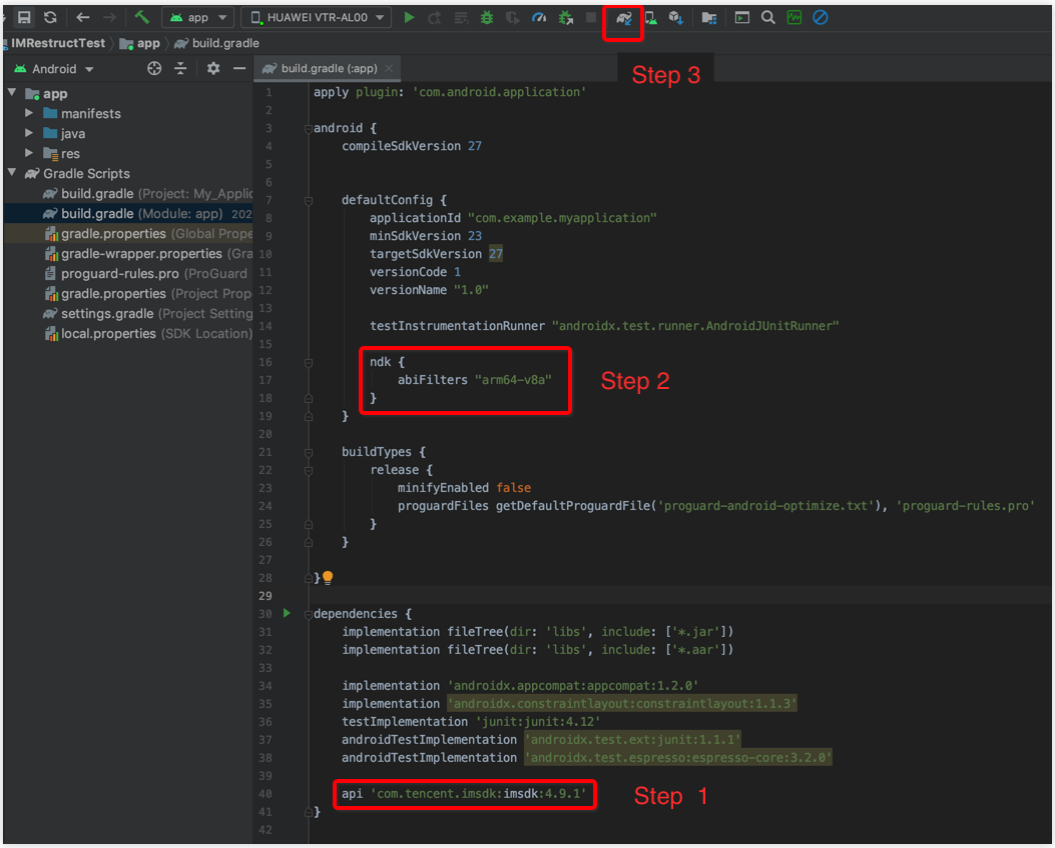
在 defaultConfig 中，指定 App 使用的 CPU 架构（从 Chat SDK 4.3.118 版本开始支持 armeabi-v7a，arm64-v8a，x86，x86_64）：



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

步骤3：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。



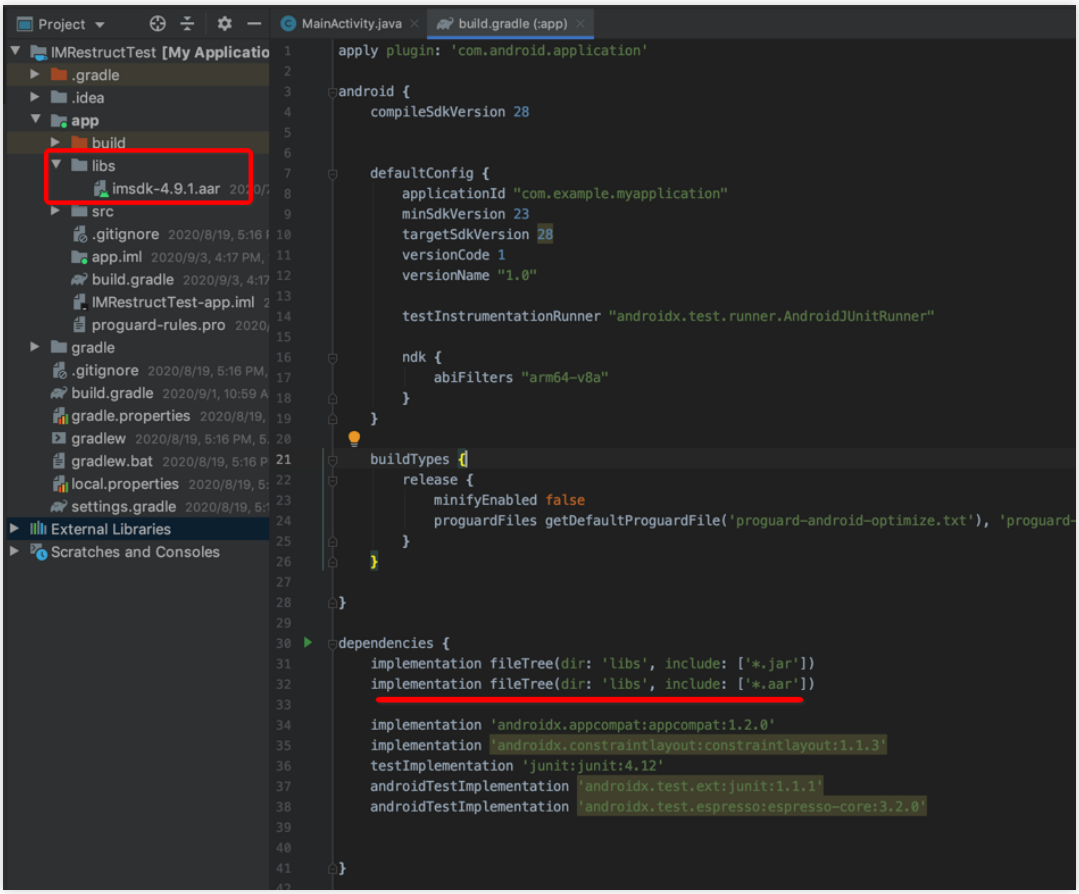
方法二：手动下载（aar）

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：

步骤1：下载 Chat SDK

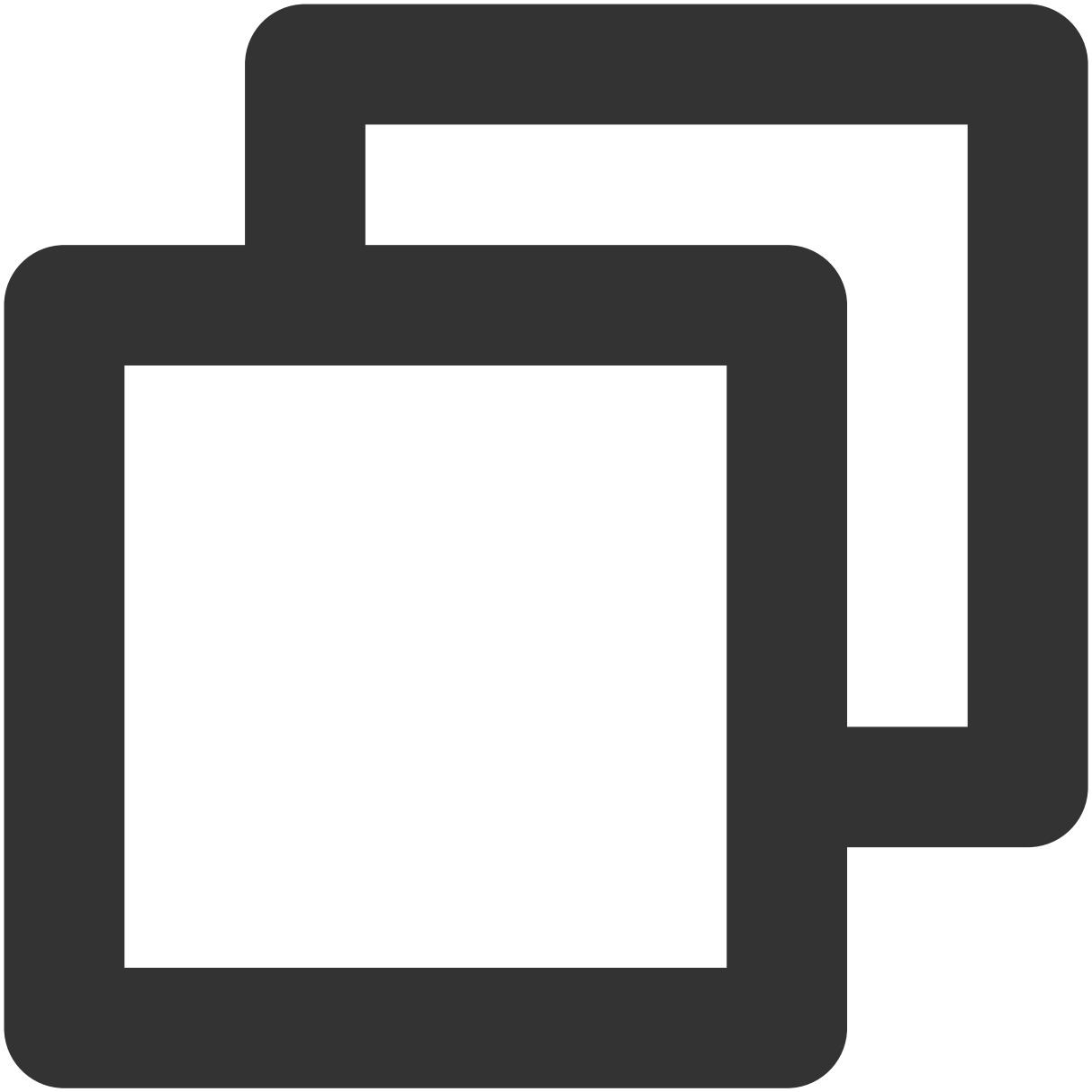
在 Github 上可以下载到最新版本的 SDK。

步骤2：拷贝 Chat SDK 到工程目录



步骤3：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 Chat SDK 4.3.118 版本开始支持 armeabi-v7a，arm64-v8a，x86，x86_64）：



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

说明：

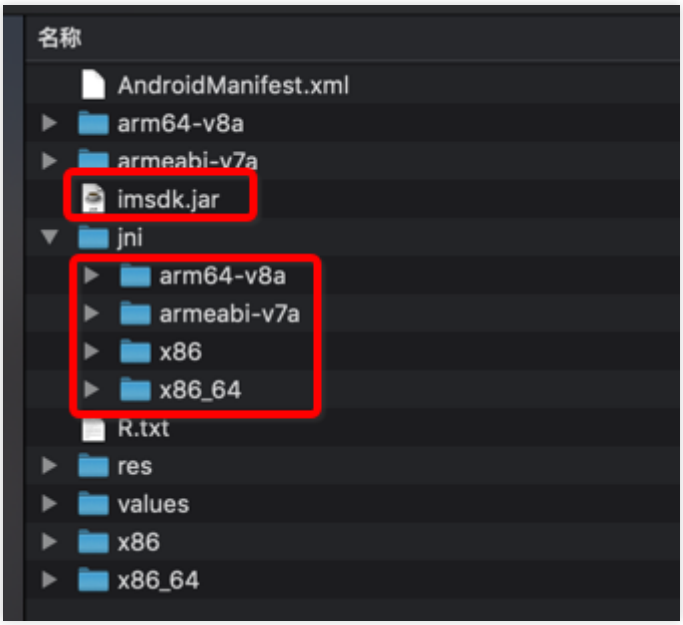
如果您需要添加 Quic 插件，请参考前面的步骤，手动下载集成 Quic 插件。

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 Chat SDK：

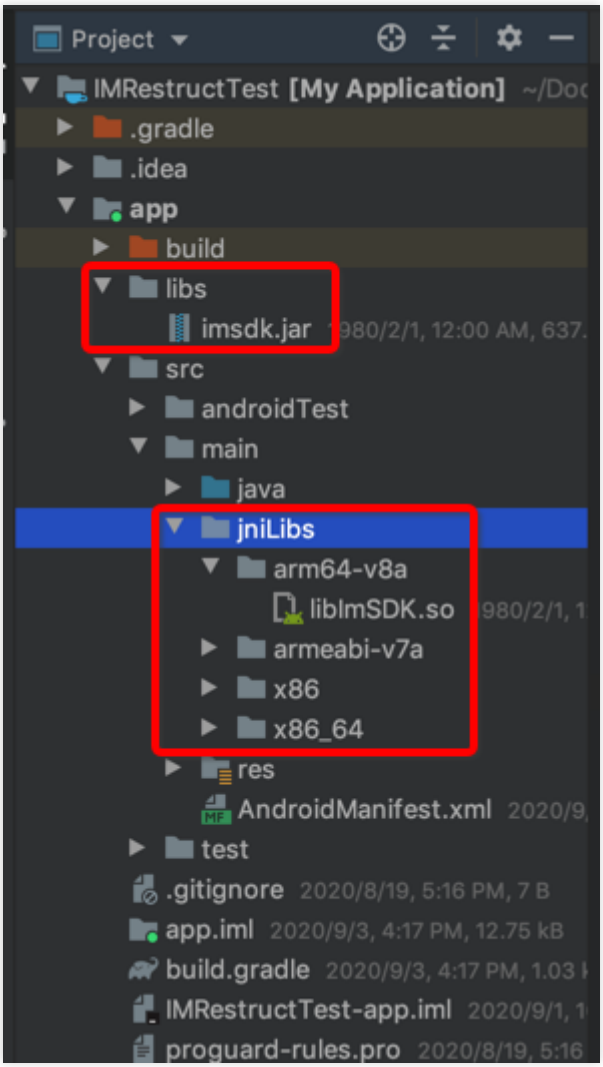
步骤1：下载解压 SDK

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 。



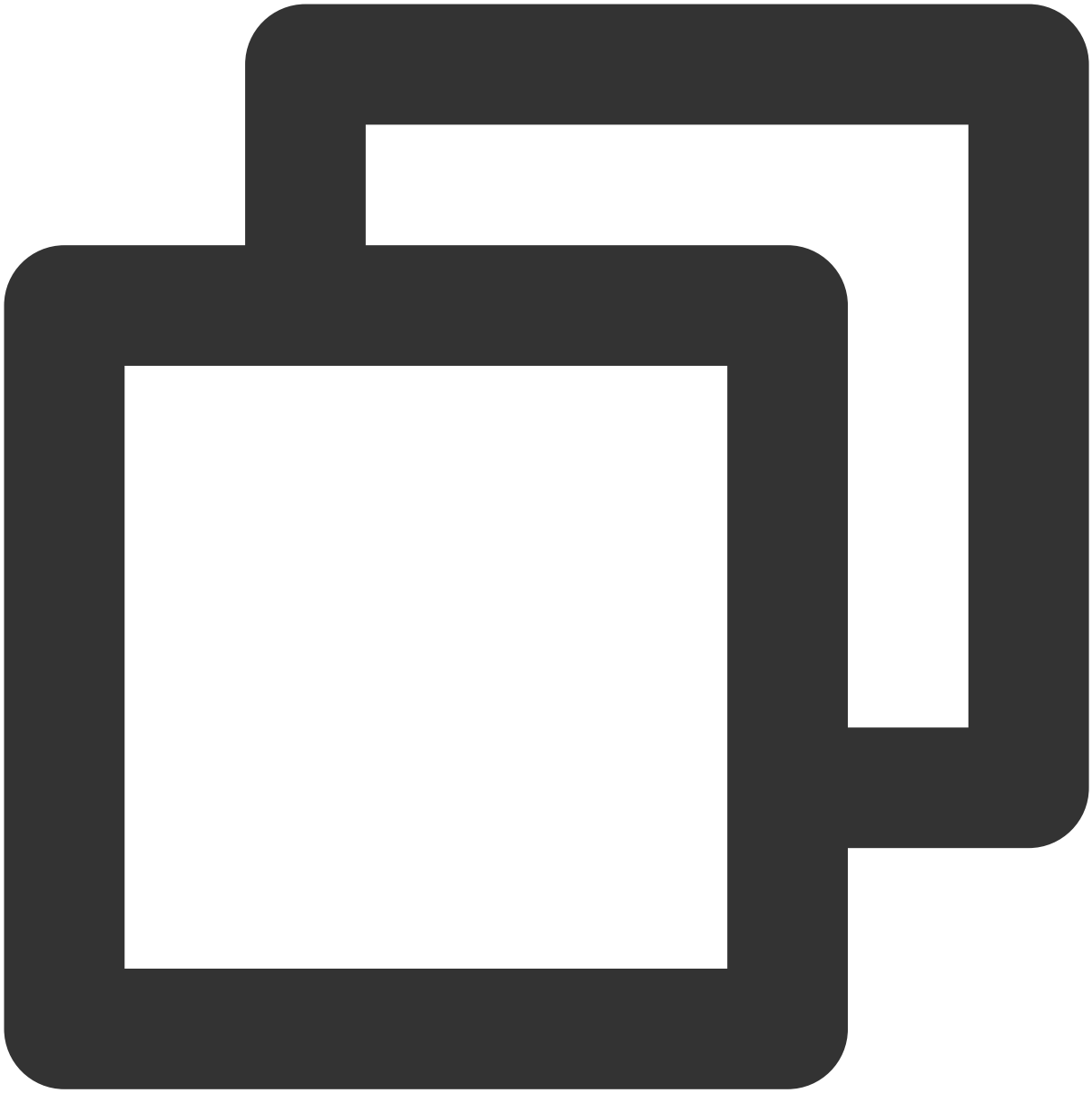
步骤2：拷贝 SDK 文件到工程目录

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：



步骤3：指定 App 使用架构并编译运行

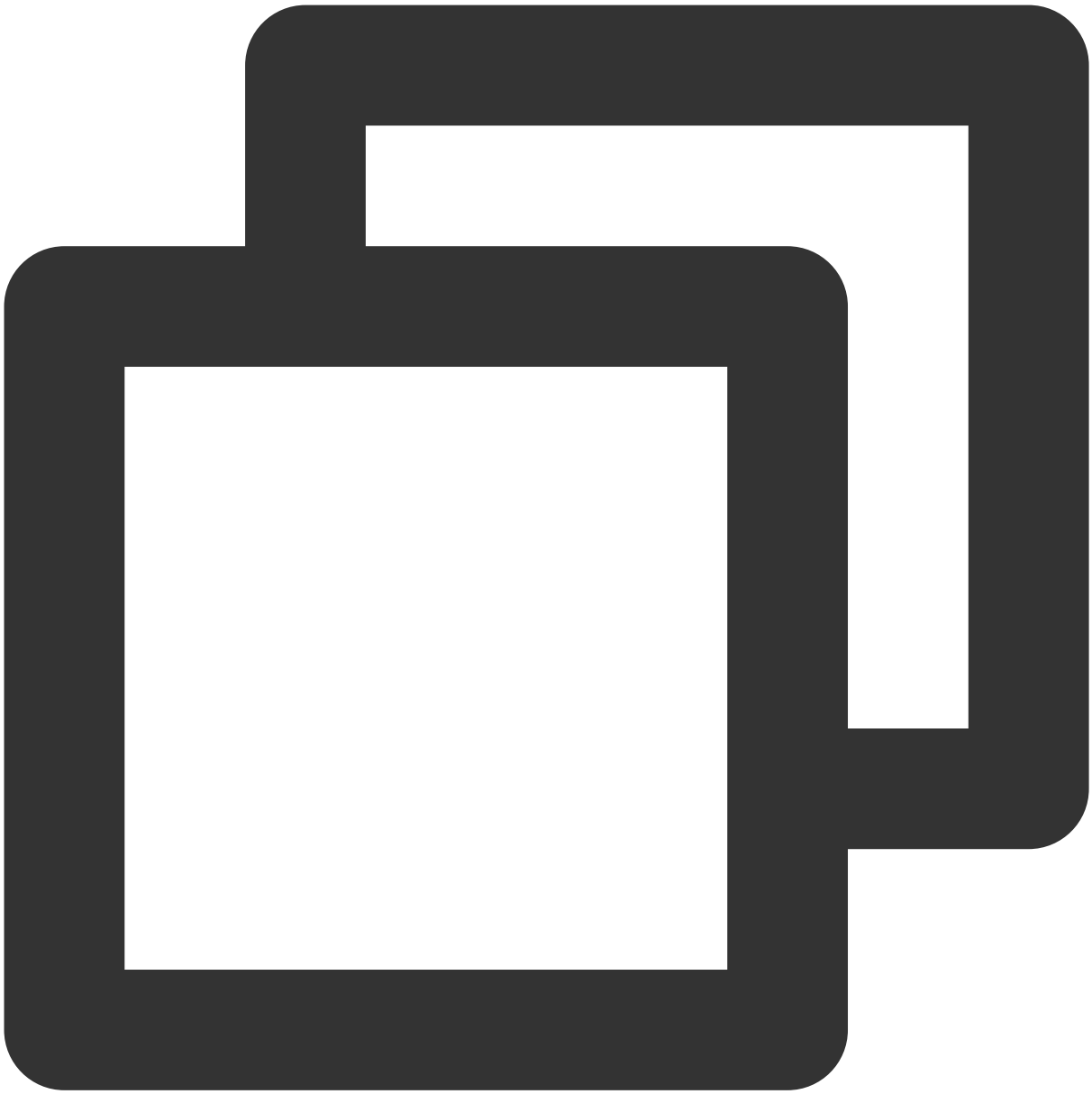
在 app/build.gradle 的 defaultConfig 中，指定 App 使用的 CPU 架构（从 Chat SDK 4.3.118 版本开始支持 armeabi-v7a，arm64-v8a，x86，x86_64）：



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

配置 App 权限

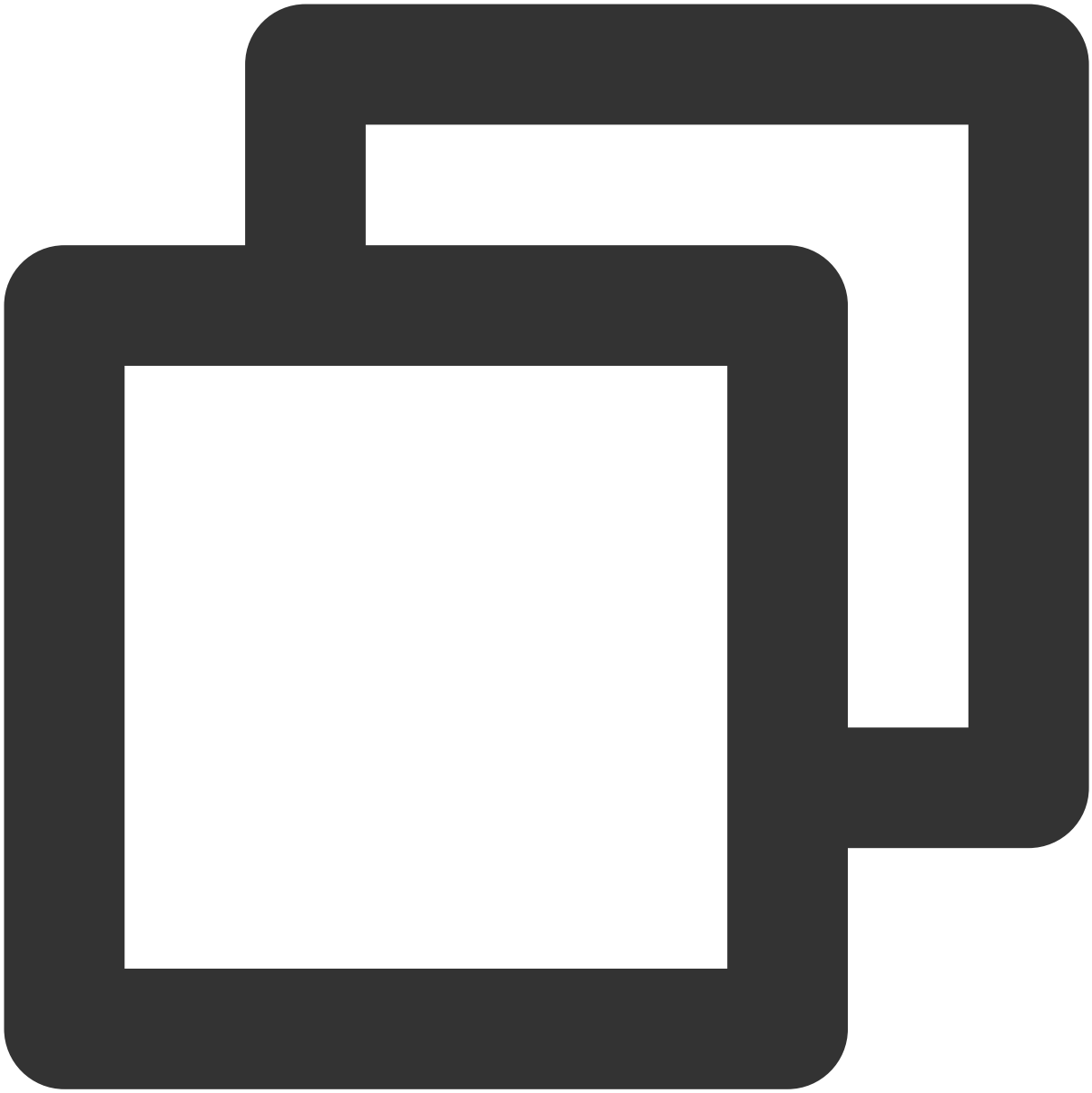
在 AndroidManifest.xml 中配置 App 的权限，Chat SDK 需要以下权限：



```
<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 Chat SDK 相关类加入不混淆名单：



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

iOS

最近更新时间：2024-07-05 15:24:08

本文主要介绍如何快速将 Chat SDK 集成到您的 iOS 项目中。

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。

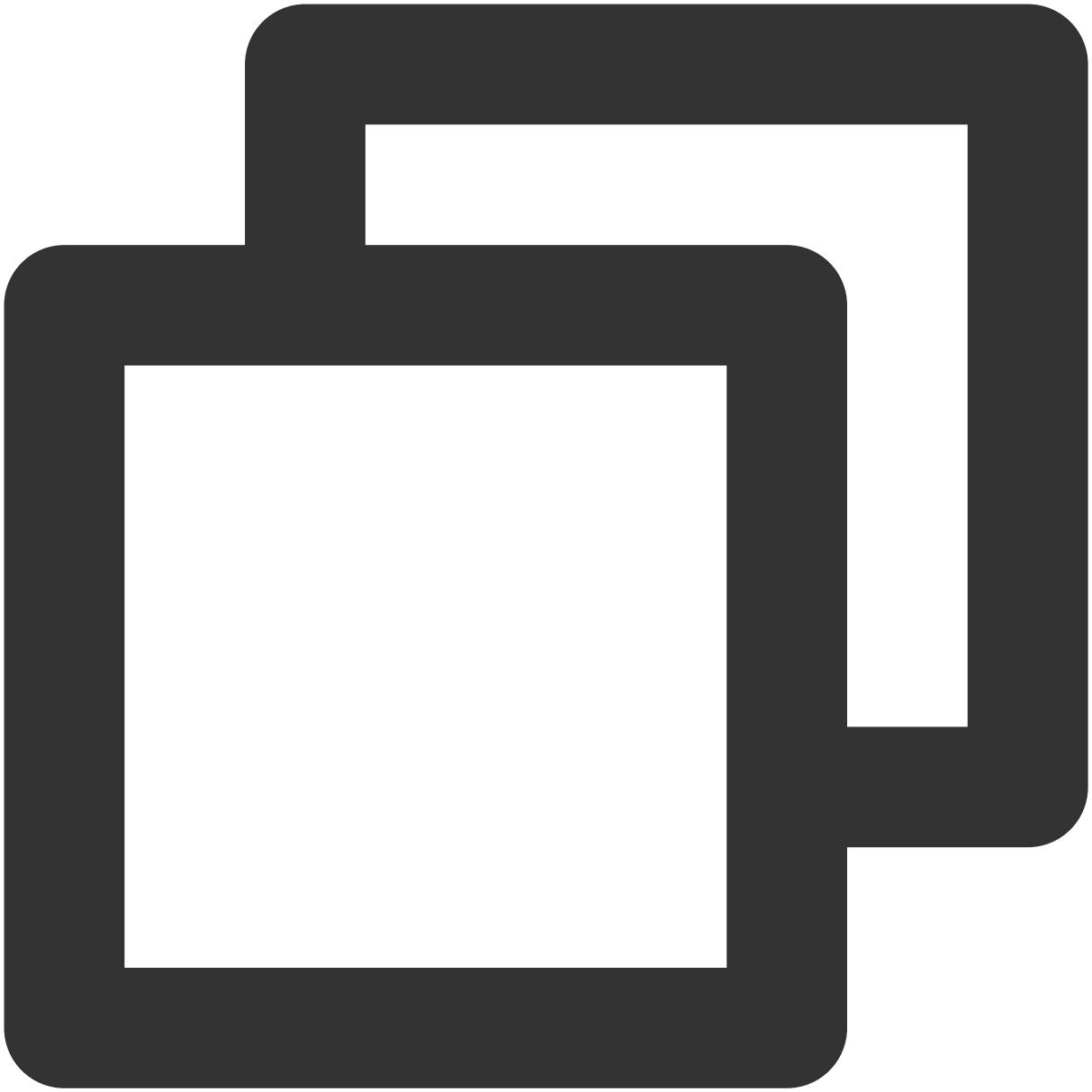
集成 SDK

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

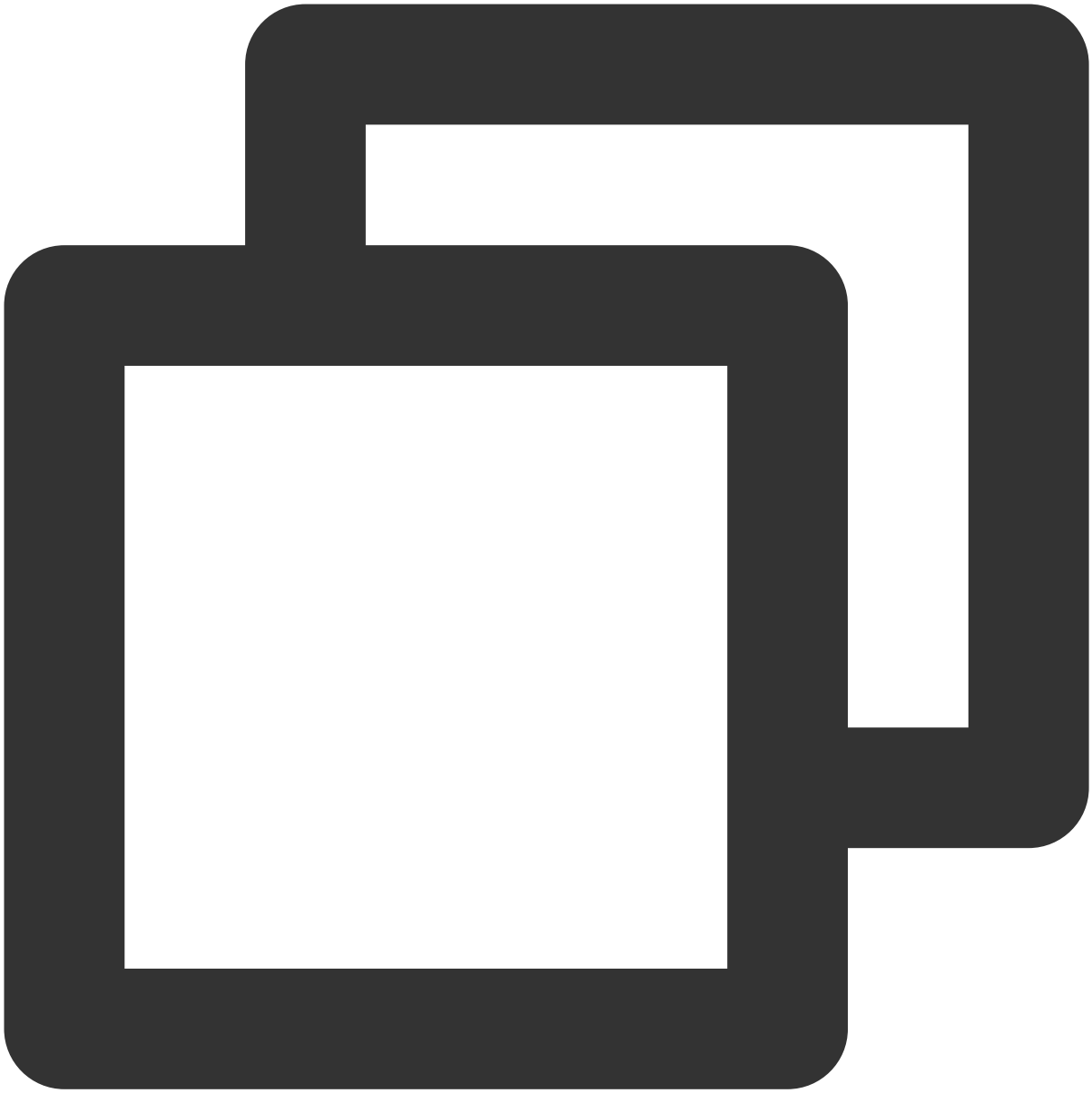
在终端窗口中输入如下命令（需要提前在 Mac 中安装 Ruby 环境）：



```
sudo gem install cocoapods
```

2. 创建 Podfile 文件

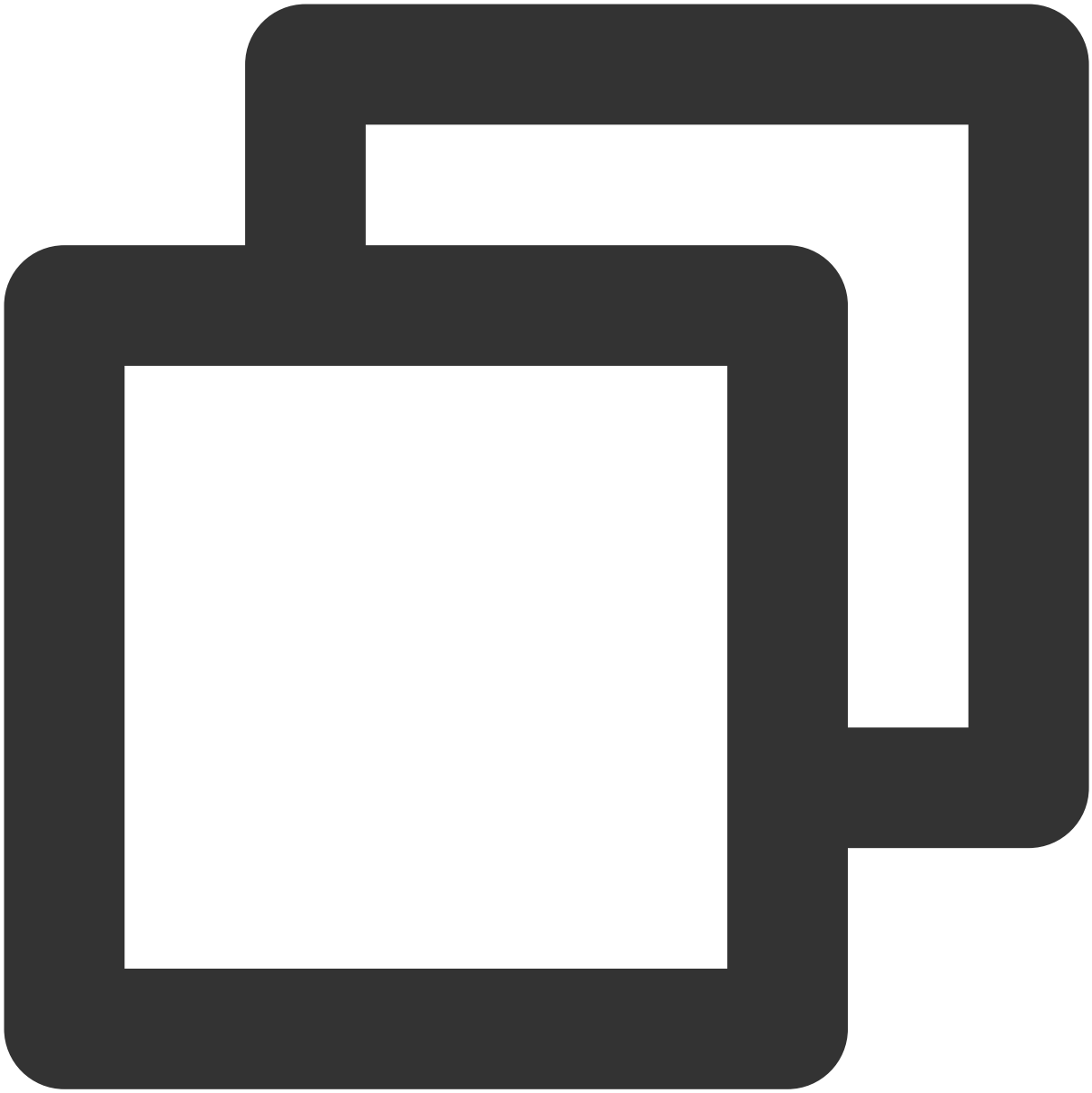
进入项目所在路径输入以下命令行，之后项目路径下会出现一个 Podfile 文件。



```
pod init
```

3. 编辑 Podfile 文件

请您按照如下方式设置 Podfile 文件：

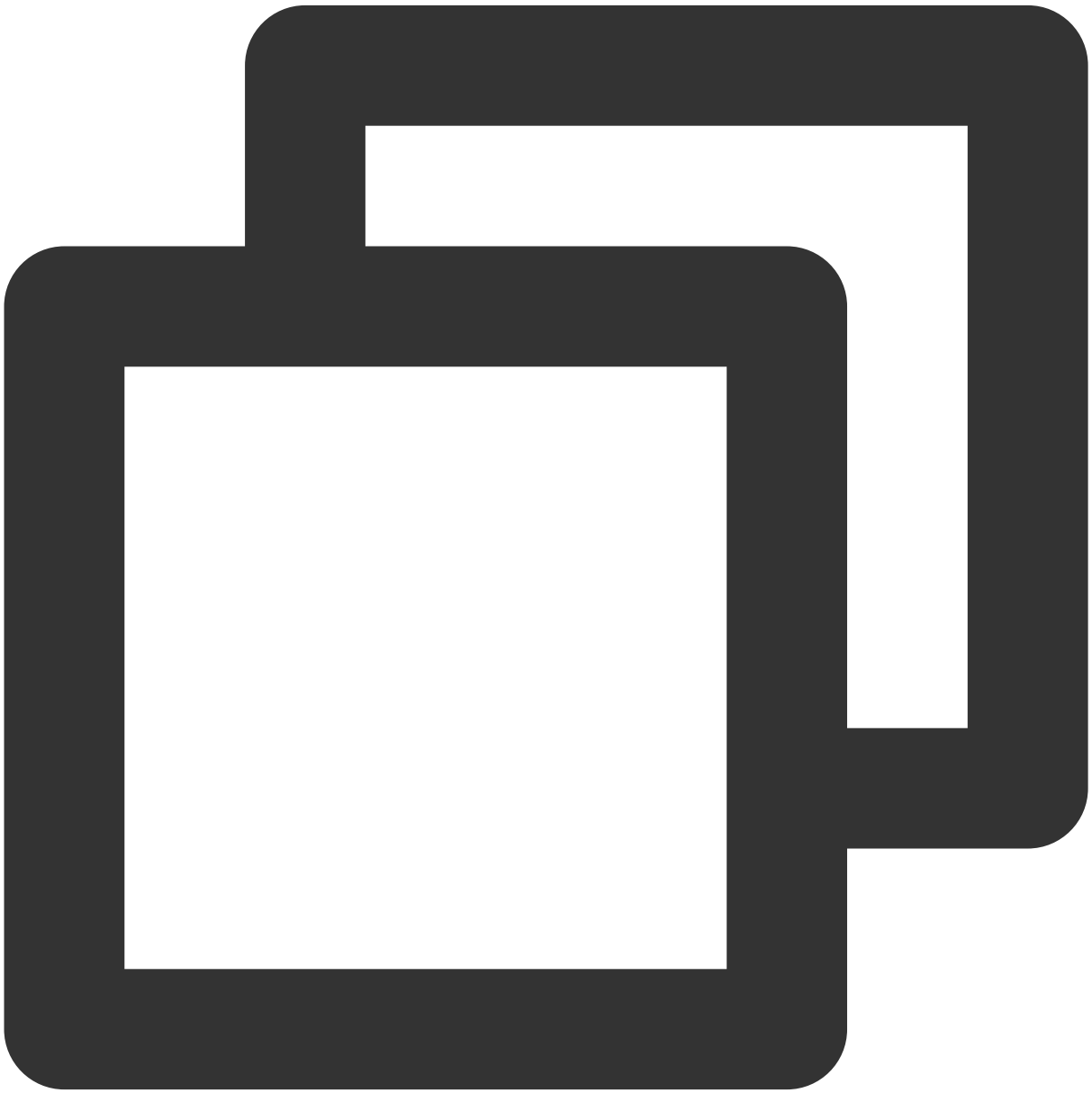


```
platform :ios, '8.0'
source 'https://github.com/CocoaPods/Specs.git'

target 'App' do
    # 添加 Chat SDK
    pod 'TXIMSDK_Plus_iOS'
    # pod 'TXIMSDK_Plus_iOS_XCFramework'
    # pod 'TXIMSDK_Plus_Swift_iOS_XCFramework'


    # 如果您需要添加 Quic 插件，请取消下一行的注释
```

```
    # 注意：
    # - 这个插件必须搭配 TXIMSDK_Plus_iOS 或 TXIMSDK_Plus_iOS_XCFramework 版本的 Chat S
    # - 对于 TXIMSDK_Plus_Swift_iOS_XCFramework 版本，不需要添加这个插件，如果您需要在这个
    # pod 'TXIMSDK_Plus_QuicPlugin'
end
```

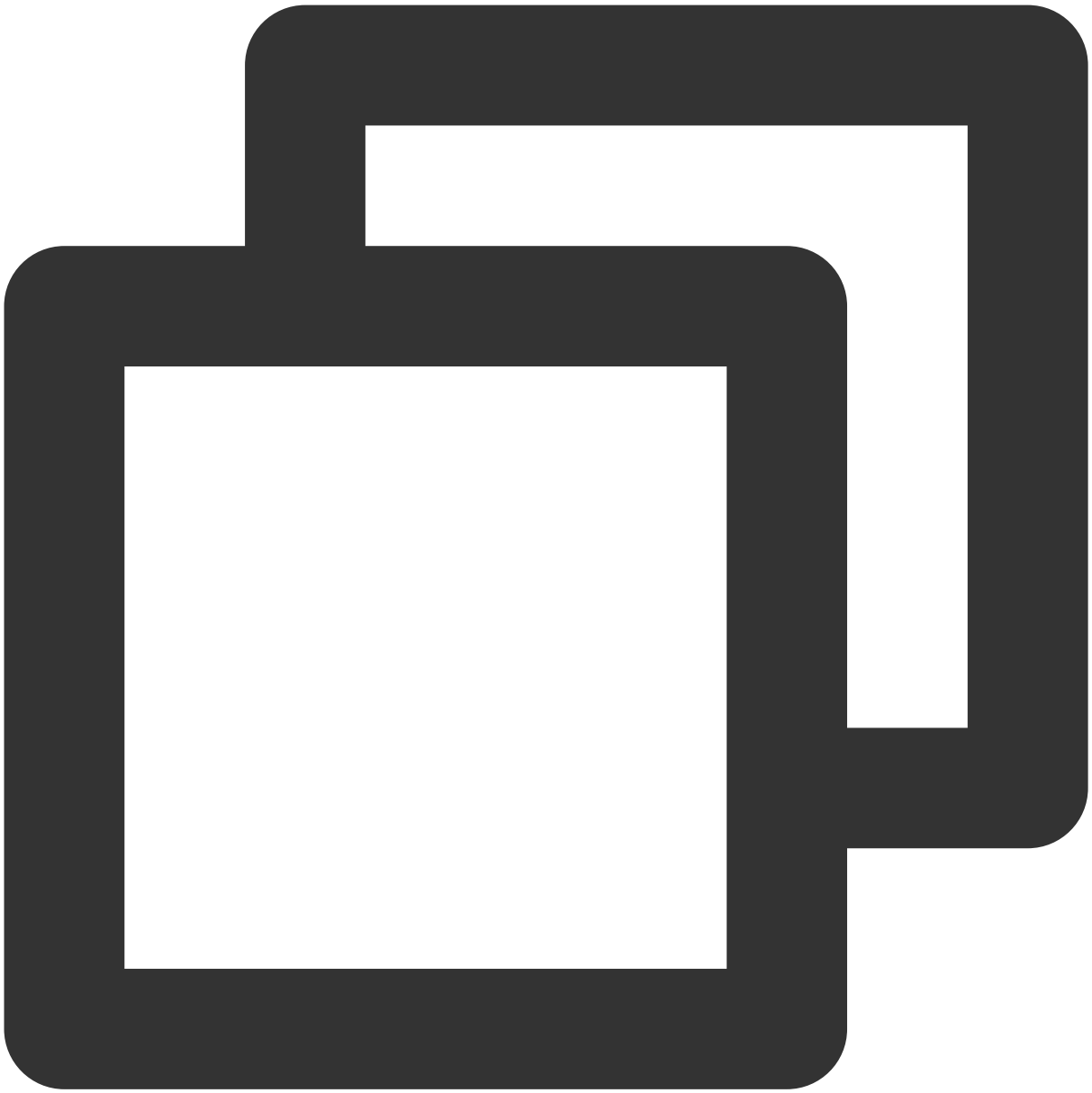
4. 更新并安装 SDK

在终端窗口中输入如下命令以更新本地库文件，并安装 Chat SDK：



```
pod install
```

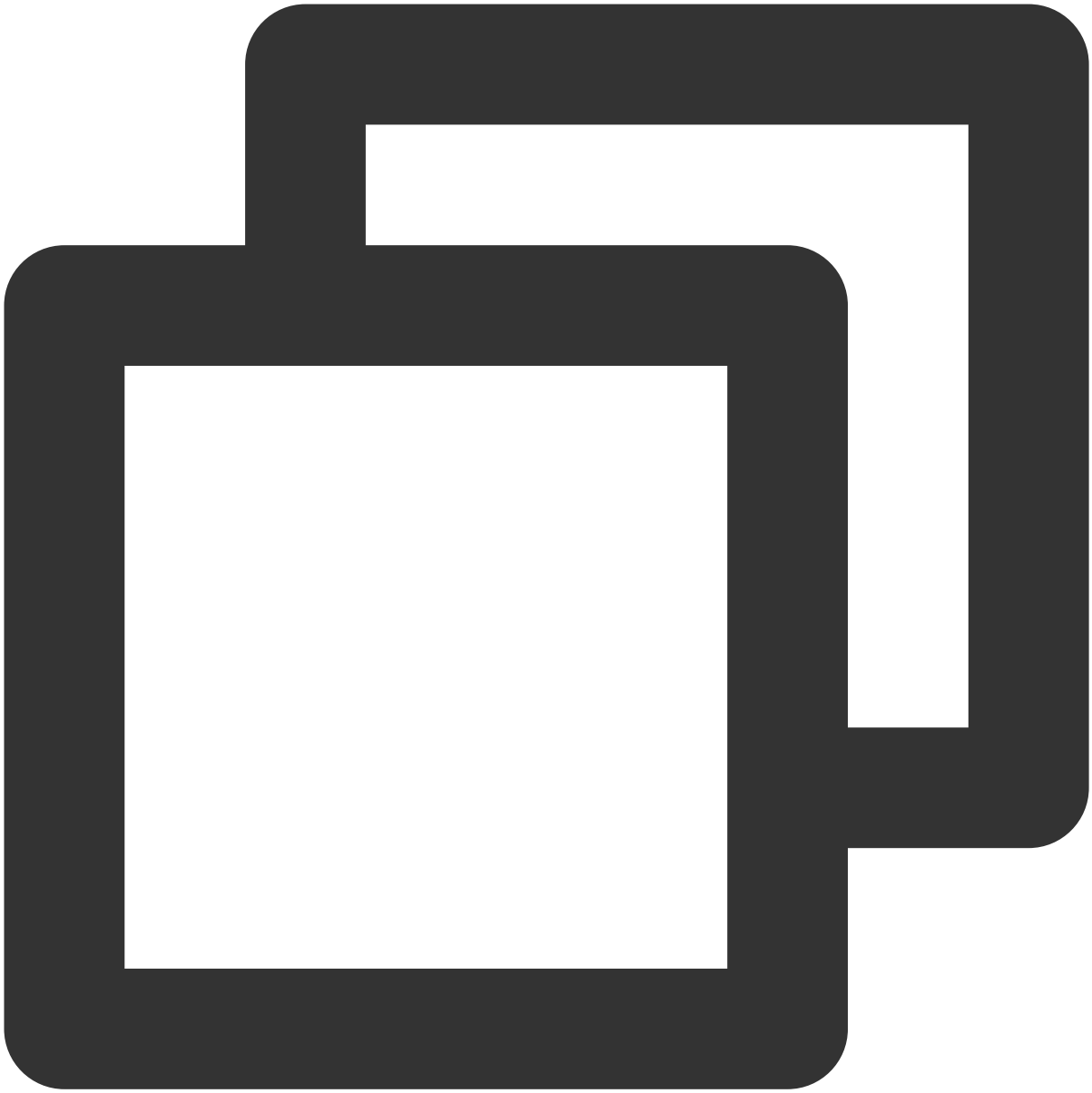
或使用以下命令更新本地库版本：



```
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。

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
```

说明：

Quic 插件，提供 axp-quic 多路传输协议，弱网抗性更优，网络丢包率达到 70% 的条件下，仍然可以提供服务。仅对进阶版用户开放，请 购买进阶版 后可使用。为确保功能正常使用，请将终端 SDK 更新至 7.7.5282 及其以上的版本。

如果您需要在 Swift 版本的 Chat SDK 中使用 Quic 功能，请您通过 Telegram 技术交流群 联系我们。

版权所有：腾讯云计算（北京）有限责任公司

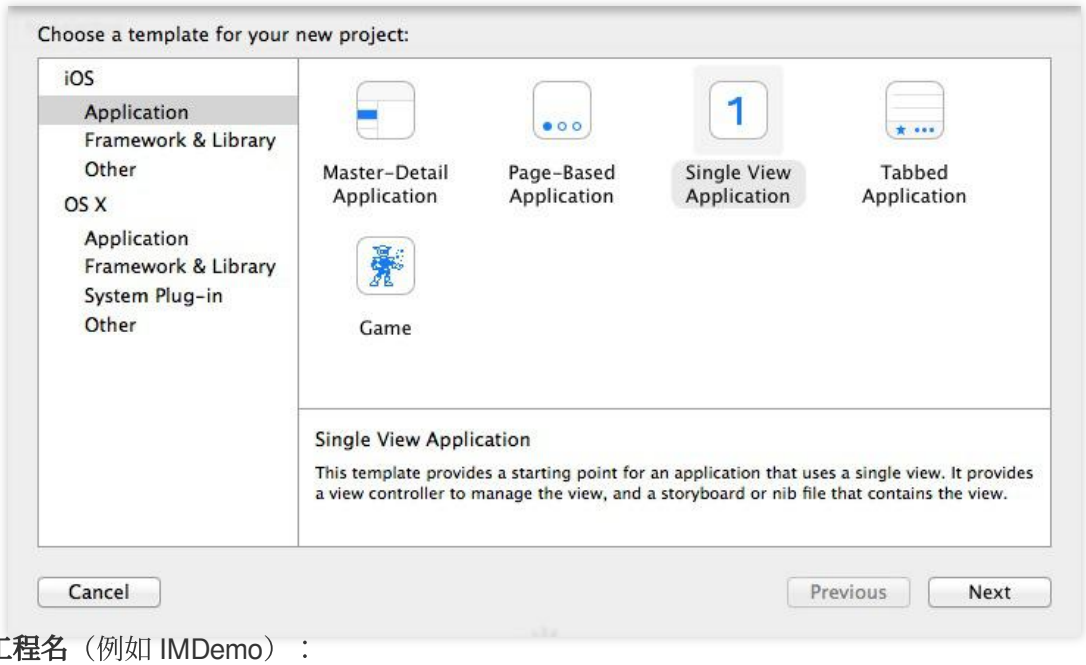
手动集成

1. 下载 SDK

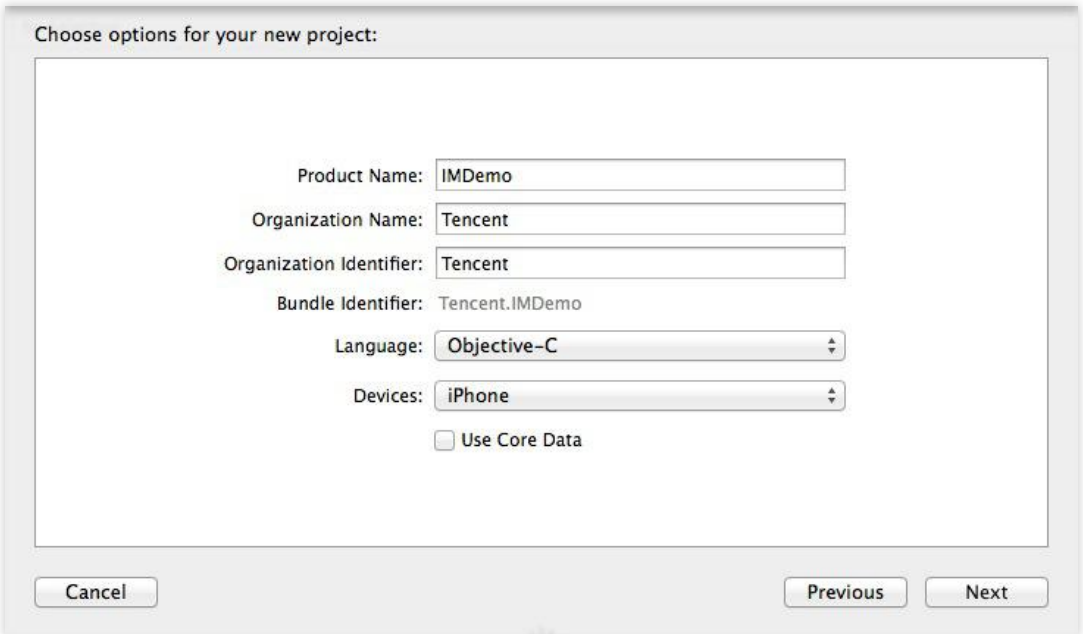
从 Github 下载最新版本 SDK。ImSDK_Plus.framework 是增强版 Chat SDK 的核心动态库文件。

2. 创建工程

创建一个新工程：

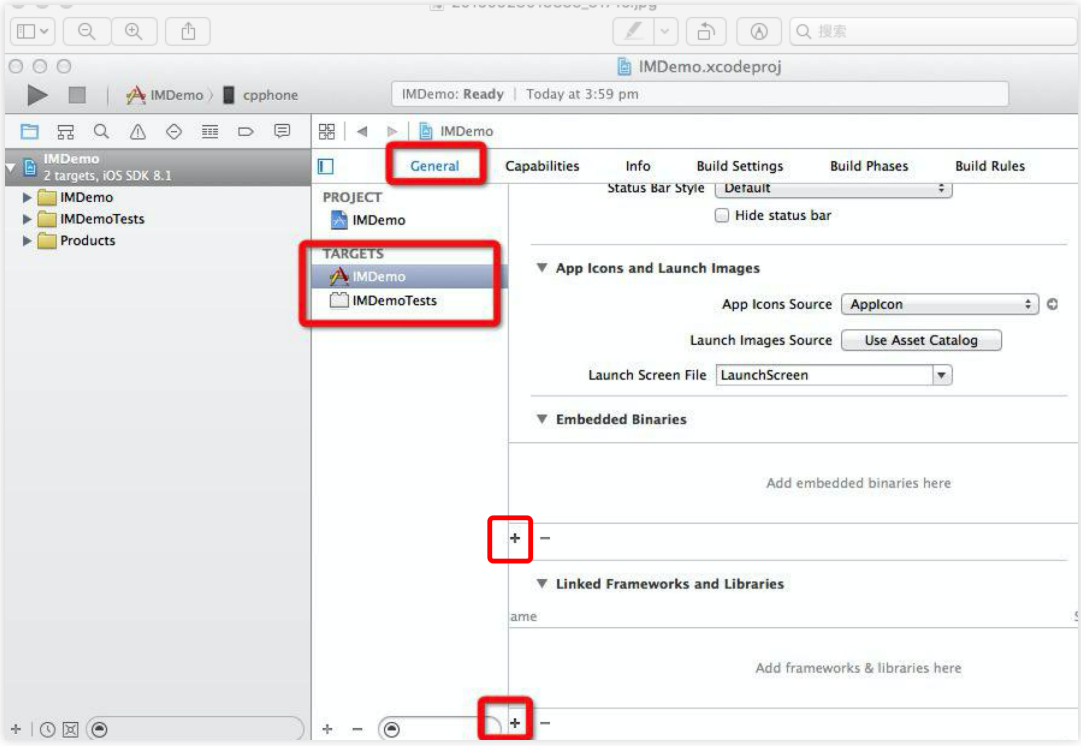


填入工程名（例如 IMDemo）：



3. 集成 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 ImSDK_Plus.framework。



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

说明

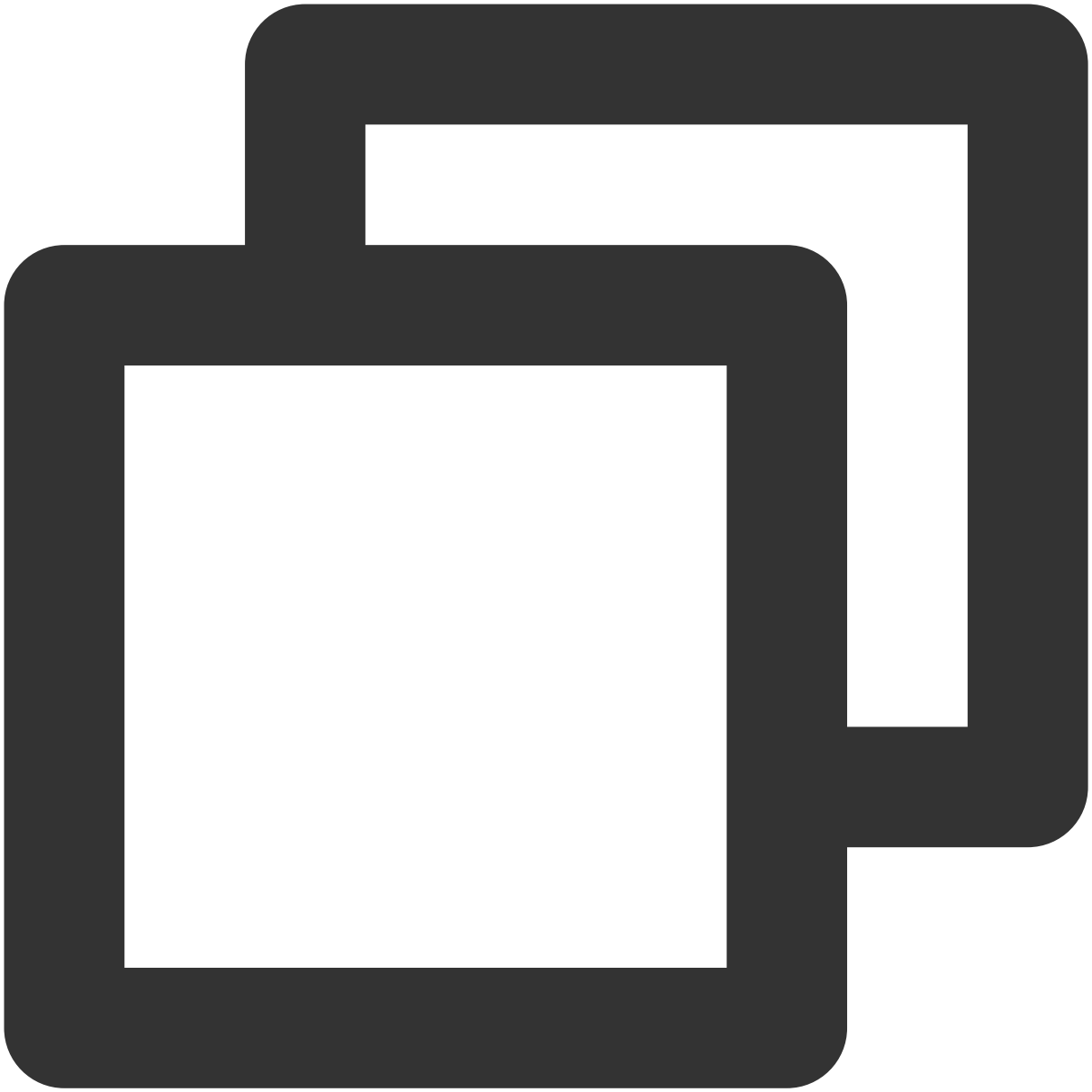
手动集成需要在 TARGET > General > Frameworks > Libraries and Embedded Content，将 ImSDK_Plus.framework 修改为 Embed&Sign。

如果您需要添加 Quic 插件，请参考前面的步骤，手动下载集成 Quic 插件。

引用 Chat SDK

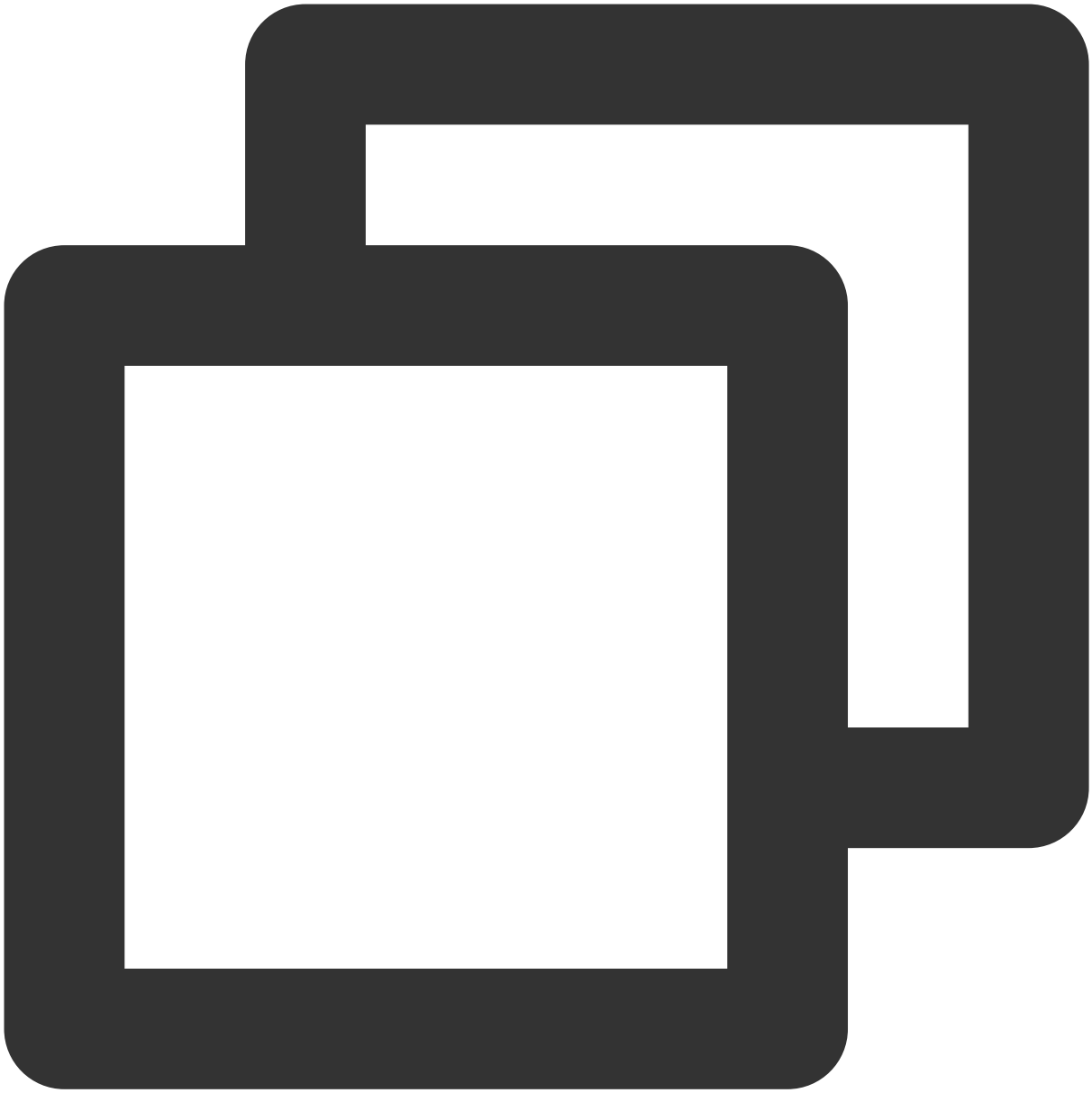
项目代码中使用 SDK 有两种方式：

方式 1，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。



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

方式 2，在项目需要使用 SDK API 的文件里，引入具体的头文件。



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

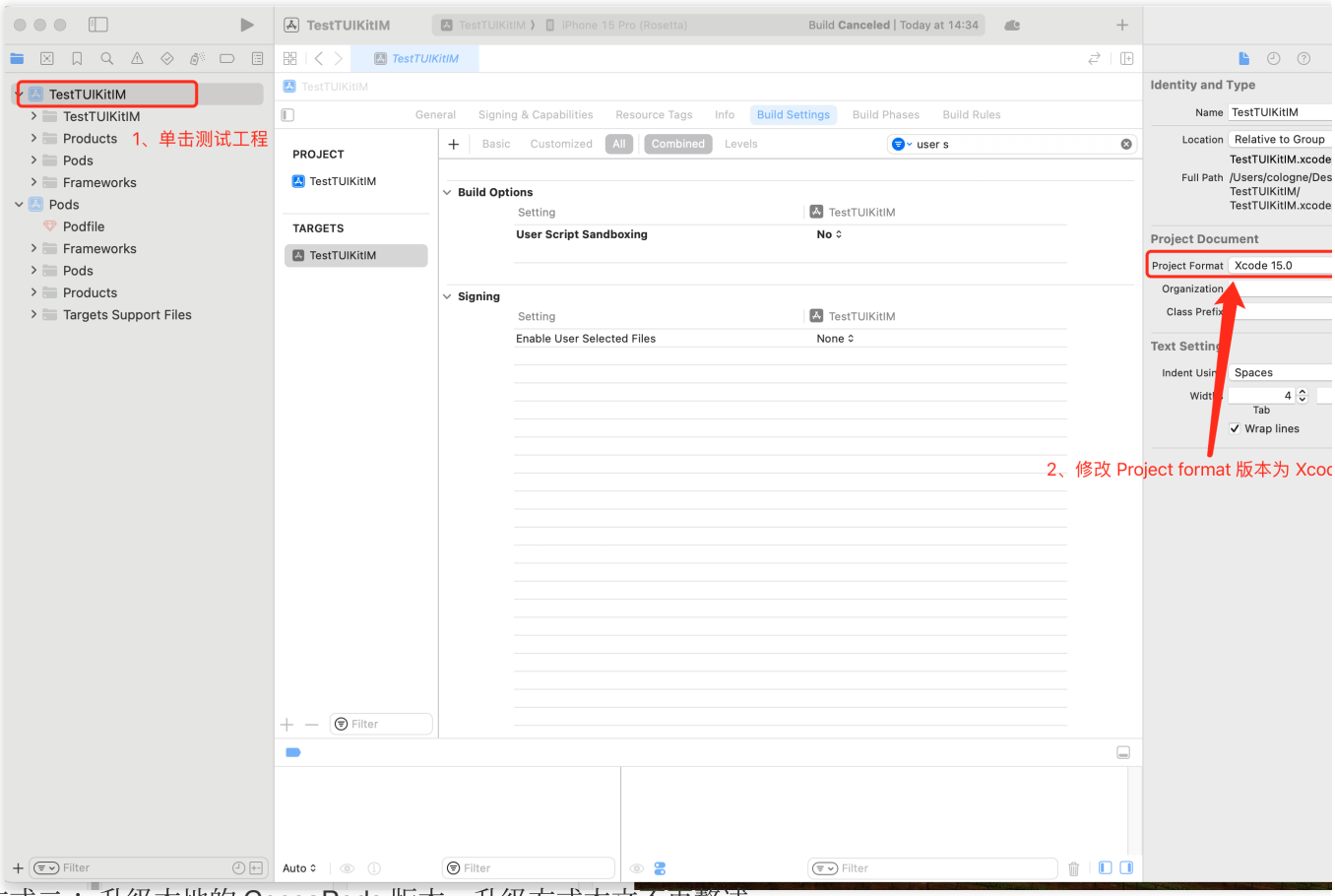
常见问题

[Xcodeproj] Unknown object version (60). (RuntimeError)

```
        from /usr/local/bin/pod:23:in `<main>'
/Library/Ruby/Gems/2.6.0/gems/xcodeproj-1.21.0/lib/xcodeproj/project.rb:228:i
ialize_from_file': [Xcodeproj] Unknown object version (60). (RuntimeError)
        from /Library/Ruby/Gems/2.6.0/gems/xcodeproj-1.21.0/lib/xcodeproj/pro
```

使用 Xcode15 创建新工程来集成 SDK 时，输入 pod install 后，可能会遇到此问题，原因是使用了较旧版本的 CocoaPods，此时有两种解决办法：

解决方式一：修改 Xcode 工程的 Project Format 版本。

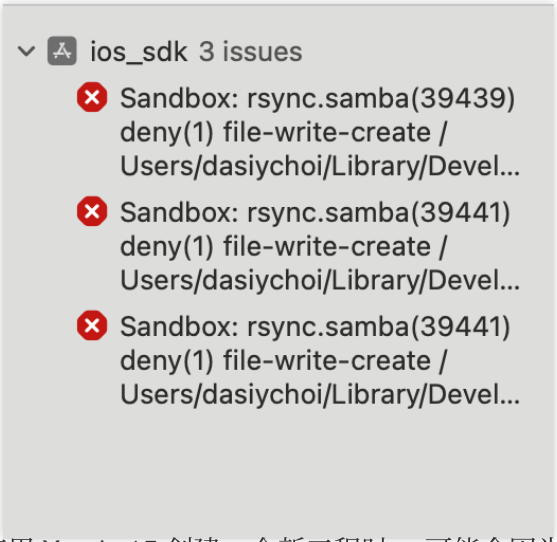


解决方式二：升级本地的 CocoaPods 版本，升级方式本文不再赘述。

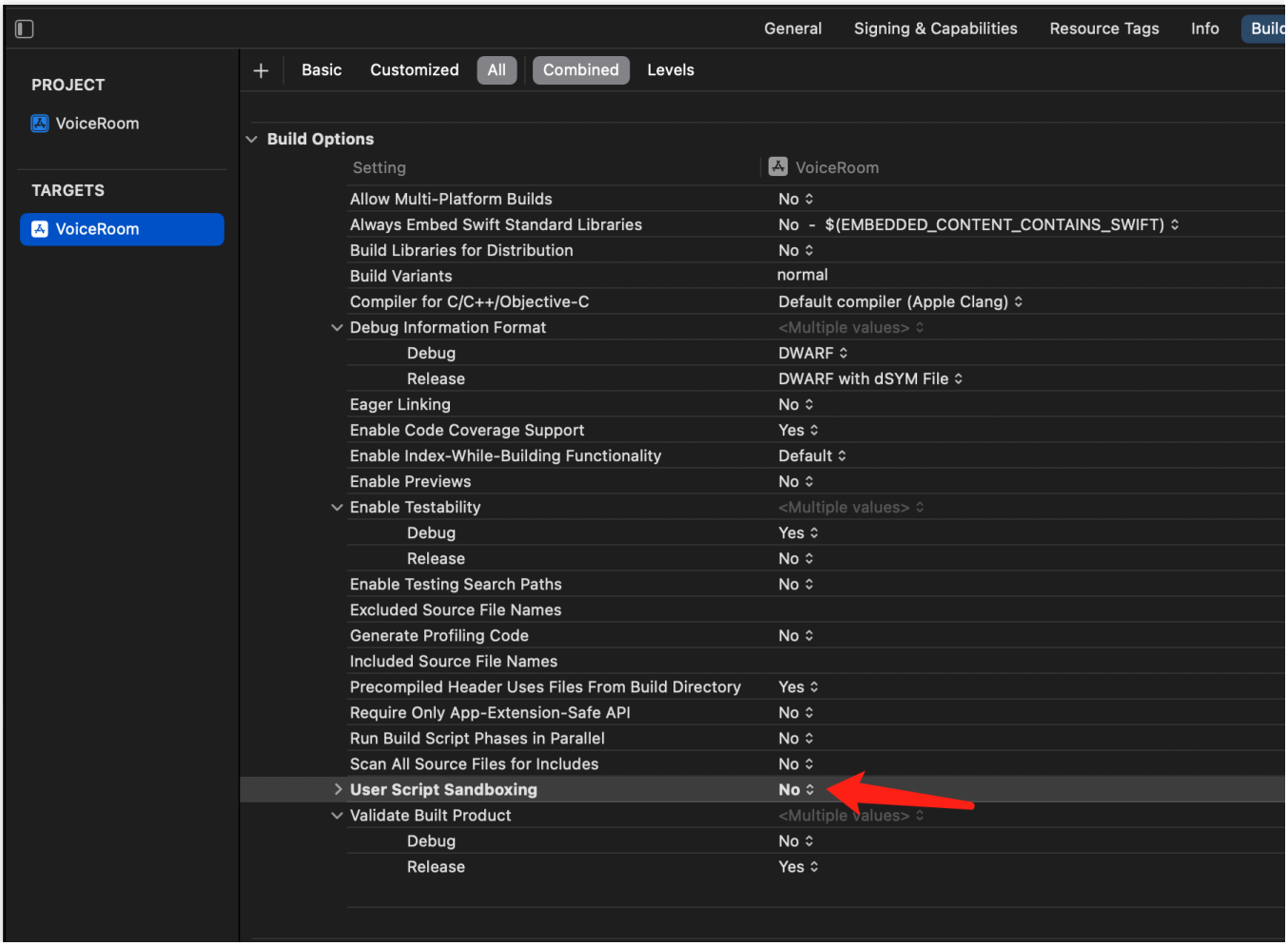
您可以在终端输入 `pod --version` 查看当前的Pods版本。

Xcode 15 开发者沙盒选项问题

Sandbox: bash(xxx) deny(1) file-write-create



当您使用 Xcode 15 创建一个新工程时， 可能会因为此选项导致编译运行失败，建议您关闭此选项。

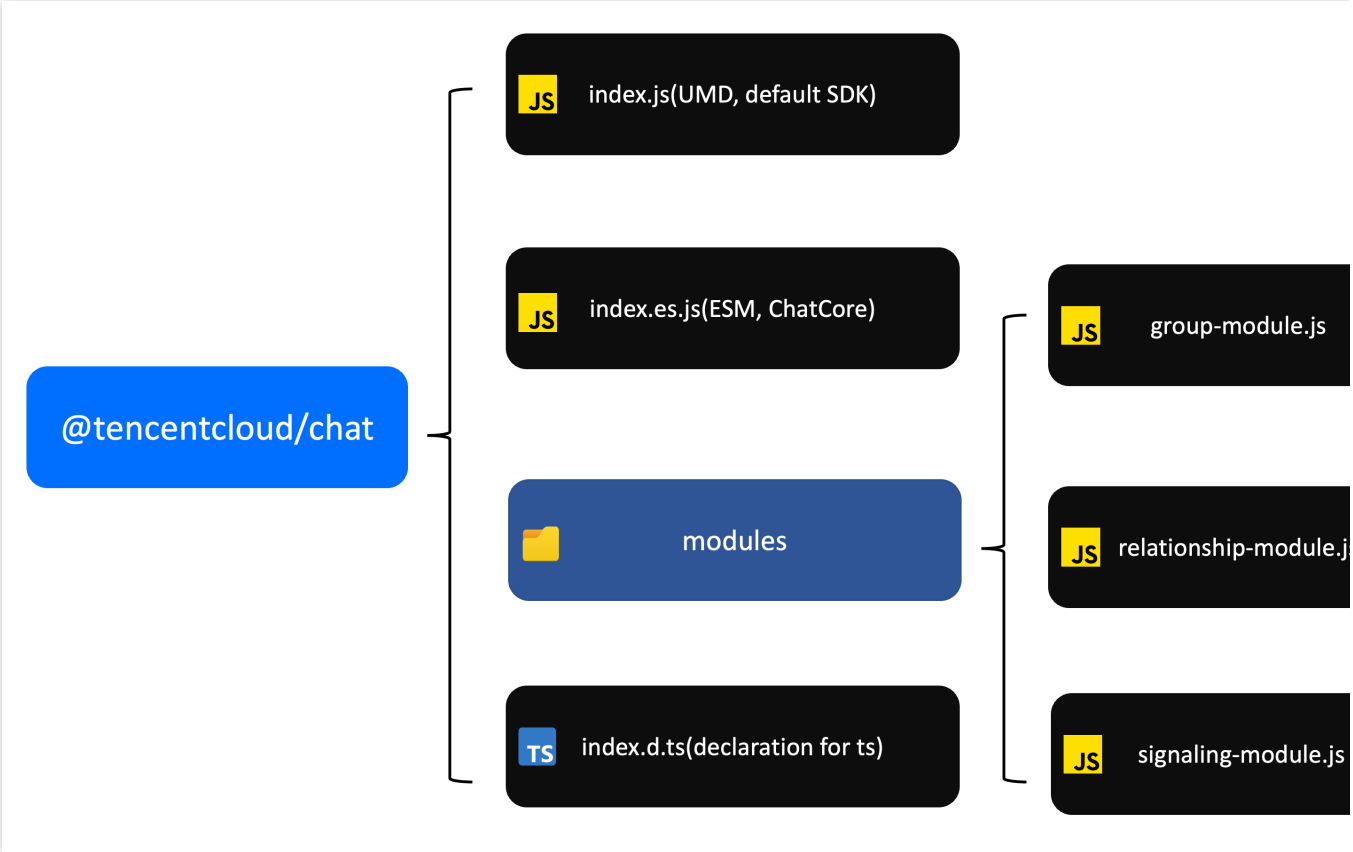


Web

最近更新时间：2024-07-10 16:31:08

本文主要介绍如何快速将腾讯云即时通信 Chat SDK 集成到您的 Web、小程序、uni-app 项目中。

Chat SDK 文件结构

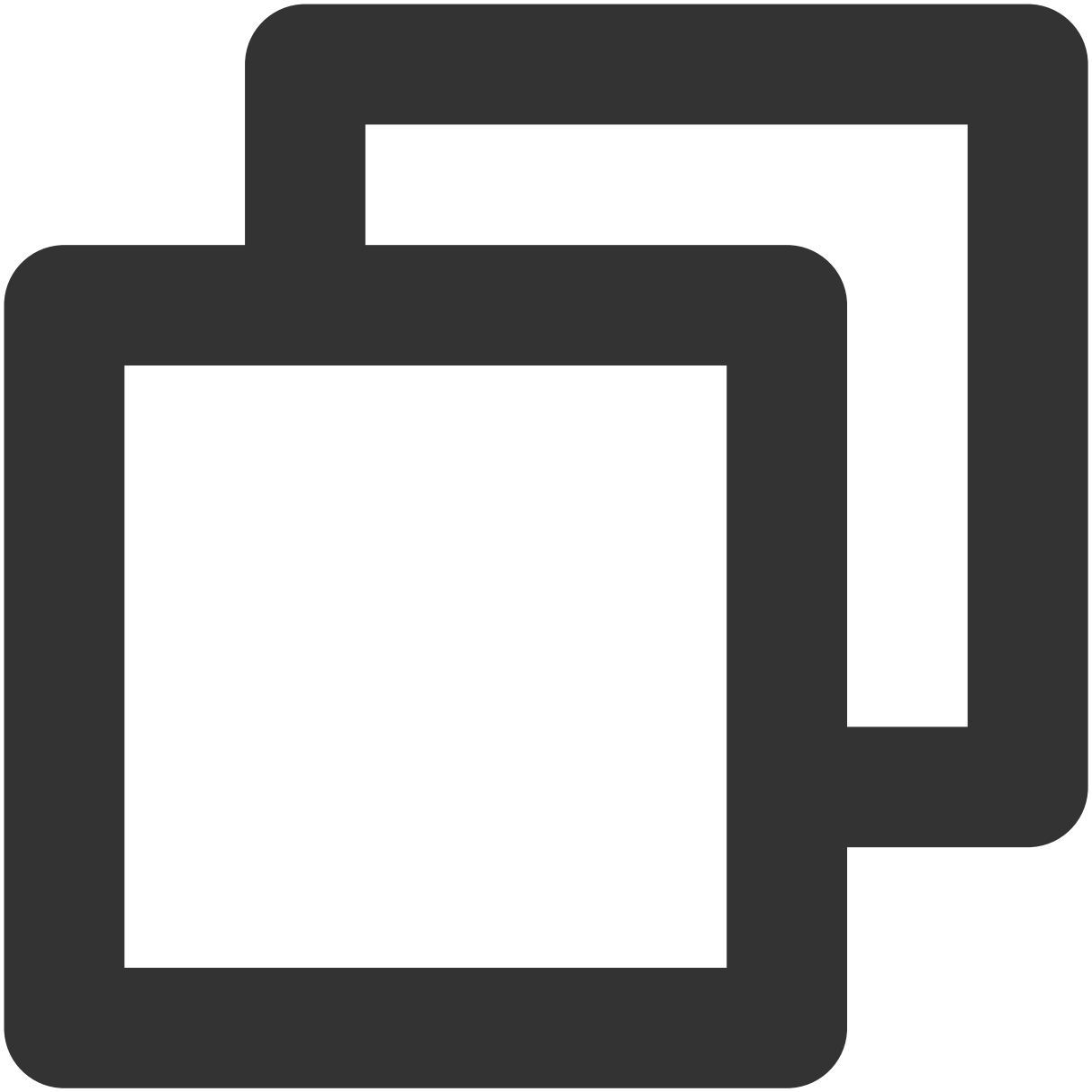


集成 SDK

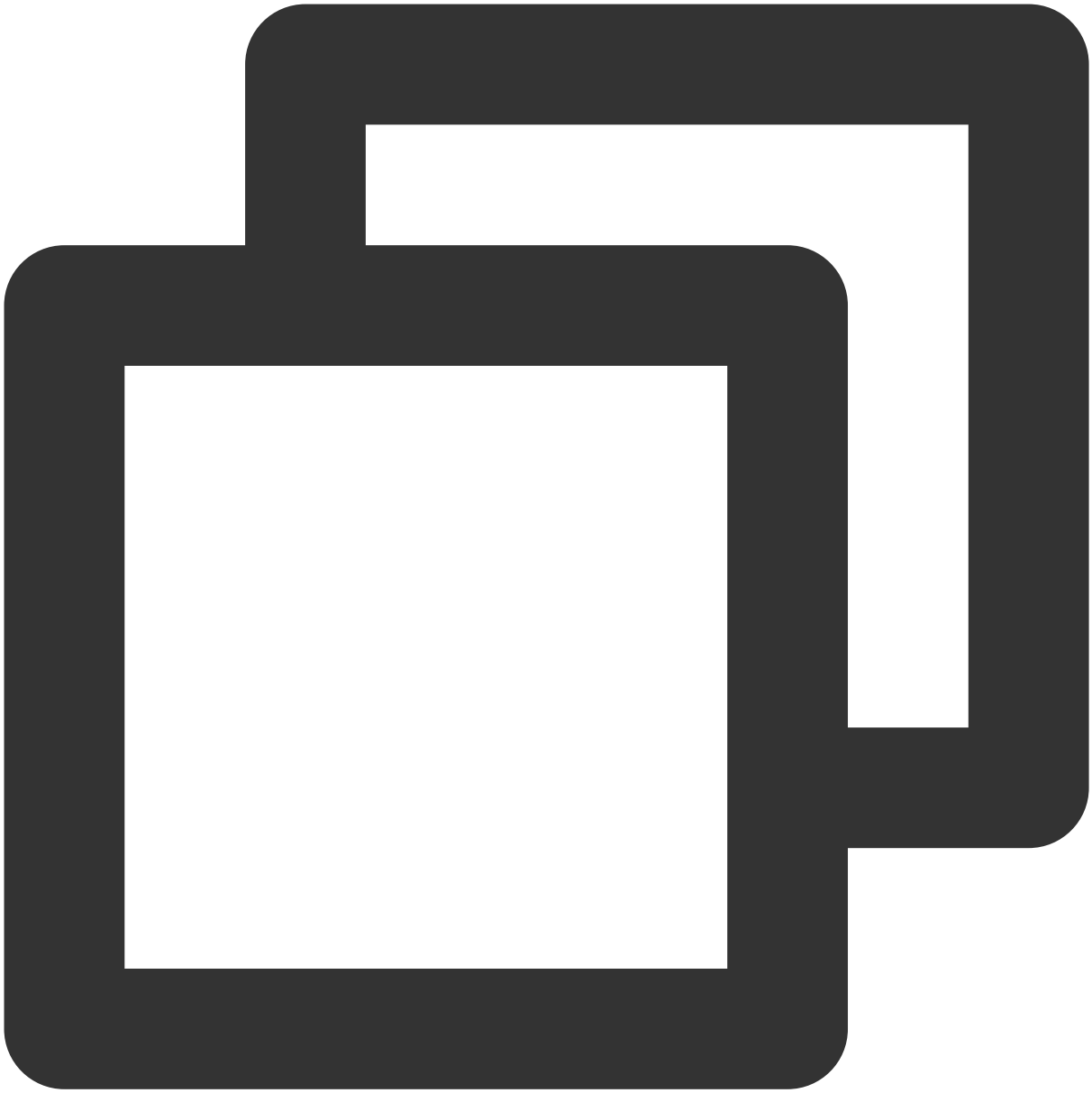
通过 npm 方式将 Chat SDK 集成到您的 Web、小程序、uni-app 项目中。
通过集成上传插件 tim-upload-plugin，实现更快更安全的富文本消息资源上传。

npm 集成（推荐）

在您的项目中使用 npm 安装相应的 Chat SDK 依赖。



```
npm install @tencentcloud/chat
// 发送图片、文件等消息需要腾讯云即时通信 IM 上传插件
npm install tim-upload-plugin --save
```



```
import TencentCloudChat from '@tencentcloud/chat';
import TIMUploadPlugin from 'tim-upload-plugin';

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
// `TencentCloudChat.create()` 方法对于同一个 `SDKAppID` 只会返回同一份实例
// SDK 实例通常用 chat 表示
let chat = TencentCloudChat.create(options);
```

```
// 设置 SDK 日志级别
// 0: 普通级别，日志量较多，接入时建议使用
// 1: release 级别，SDK 输出关键信息，生产环境时建议使用
chat.setLogLevel(0);
// chat.setLogLevel(1);

// 注册腾讯云即时通信 IM 上传插件
chat.registerPlugin({'tim-upload-plugin': TIMUploadPlugin});
```

下一步，初始化 SDK 。

相关资源

SDK 更新日志
SDK 接口文档

常见问题

1. 是否有开源的 UI 组件可以复用或者二次开发？

TencentCloud Chat 提供了各个平台的开源的 UIKit，供开发者复用和二次开发。请参考以下文档：

快速集成 TUIKit（React）
快速集成 TUIKit（Vue）

Flutter

最近更新时间：2024-04-09 16:36:56

本文主要介绍如何快速将腾讯云即时通信 IM SDK 集成到您的 Flutter 项目中。

环境要求

平台	版本
Flutter	2.2.0 及以上版本。
Android	Android Studio 3.5及以上版本，App 要求 Android 4.1及以上版本设备。
iOS	Xcode 11.0及以上版本，真机调试请确保您的项目已设置有效的开发者签名。

支持平台

我们致力于打造一套支持 Flutter 全平台的即时通信 IM SDK 及 TUIKit，帮助您一套代码，全平台运行。

平台	支持状态
iOS	支持
Android	支持
Web	支持，4.1.1+2版本起
macOS	支持，4.1.9版本起
Windows	支持，4.1.9版本起
混合开发 （将 Flutter SDK 添加至现有原生应用）	5.0.0版本起支持

说明：

Web/macOS/Windows 平台需要简单的几步额外引入，详情请查看本文 Web 兼容 和 Desktop兼容 部分。

体验 DEMO

在开始接入前，您可以体验我们的 DEMO，快速了解腾讯云 IM Flutter 跨平台 SDK 及 TUIKit 的能力。

版权所有：腾讯云计算（北京）有限责任公司

以下各版本 DEMO，均由同一 Flutter 项目制作打包而成。 Desktop(macOS/Windows)平台，SDK 已支持，DEMO 将于近期上线。

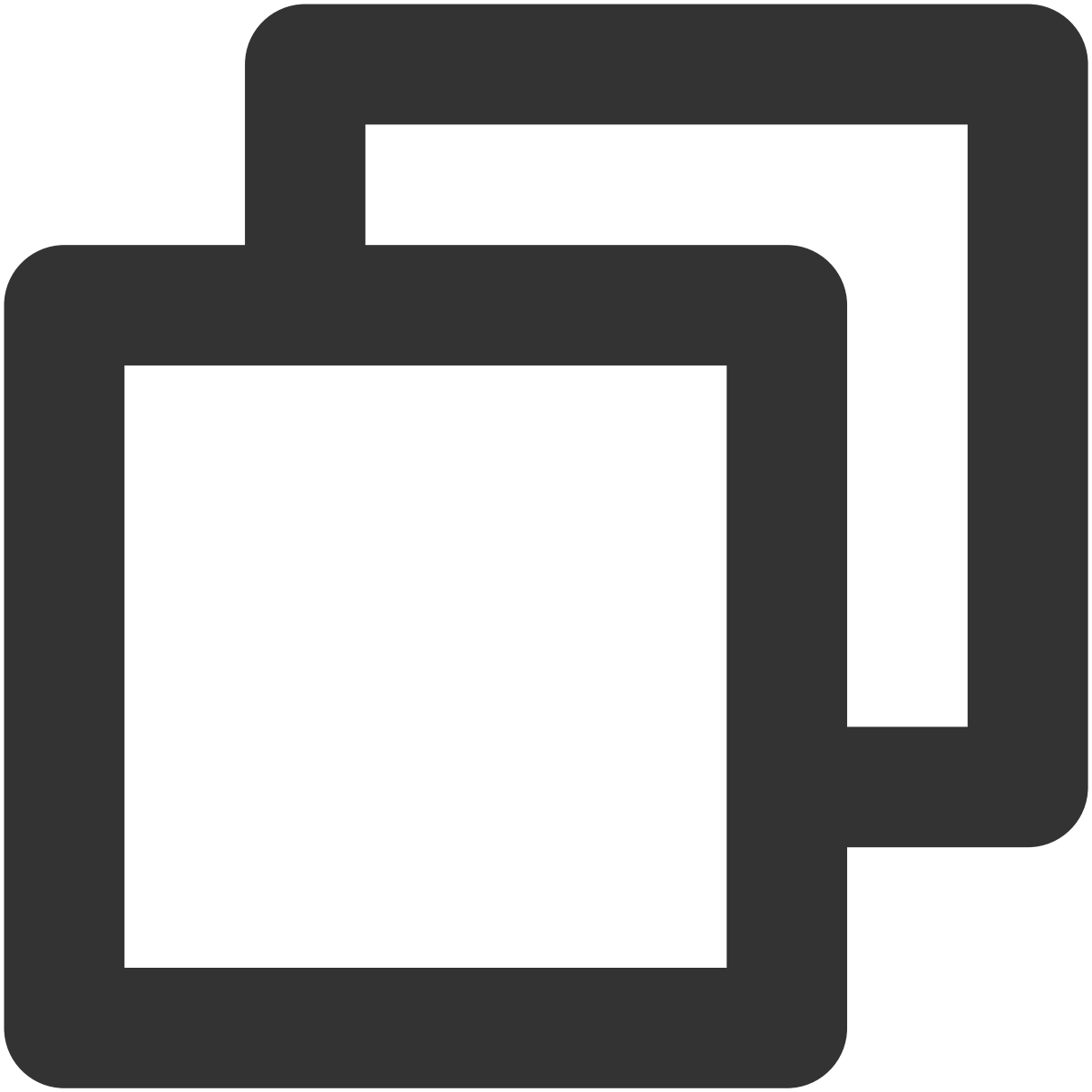
移动端 APP	WEB - H5
iOS/Android APP，自动判断平台下载	手机扫码体验在线Web版DEMO![](https://qcloudimg.tencent-cloud.cn/raw/3c79e8bb16dd0eeab35e894a690e0444.

集成 IM SDK

您可以通过 pub add 的方式直接集成腾讯云 IM SDK（Flutter），或者在 pubspec.yaml 中写入 IM SDK 的方式来集成。

flutter pub add 安装

在终端窗口中输入如下命令（需要提前安装 Flutter 环境）：

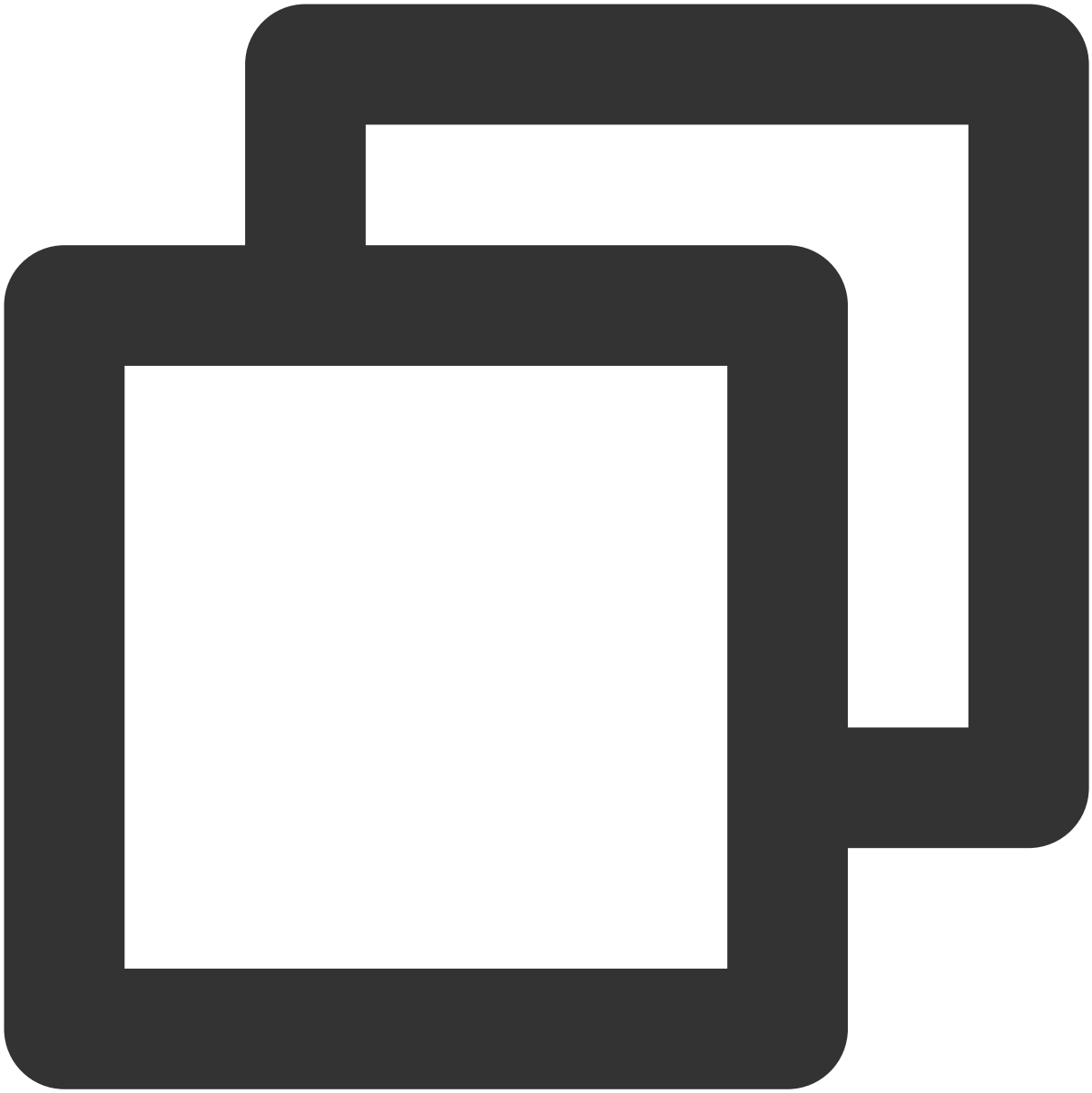


```
flutter pub add tencent_cloud_chat_sdk
```

说明：

如果您的项目还同时需要用于 Web 或 桌面端(macOS、Windows)，一些额外的步骤是需要的，具体请看文末。

在 pubspec.yaml 中写入



```
dependencies:
  tencent_cloud_chat_sdk: "最新版本" //可在https://pub.dev/packages/tencent_cloud_cha
```

此时您的 editor 或许会自动 flutter pub get，如果没有请您在命令行中手动输入 flutter pub get。
如果您的项目需要支持 Web，请在执行后续步骤前，查看Web兼容说明章节，引入JS文件。

Flutter for Web支持

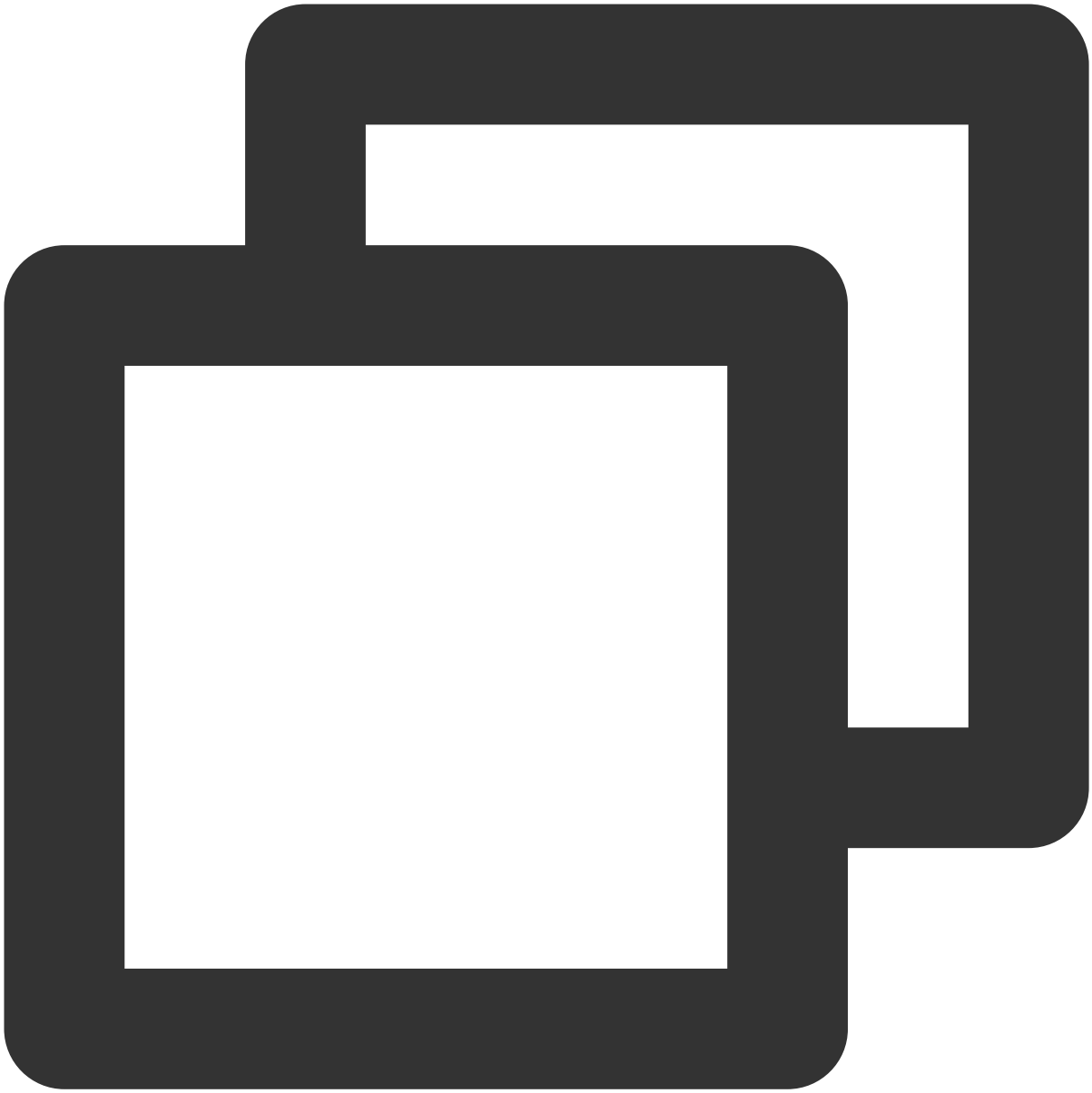
IM SDK(tencent_cloud_chat_sdk) 4.1.1+2版本起，可完美兼容Web端。
相比 Android 和 iOS 端，需要一些额外步骤。如下：

升级 Flutter 3.x 版本

Flutter 3.x 版本 针对 Web 性能做了较多优化，强烈建议您使用其来开发 Flutter Web 项目。

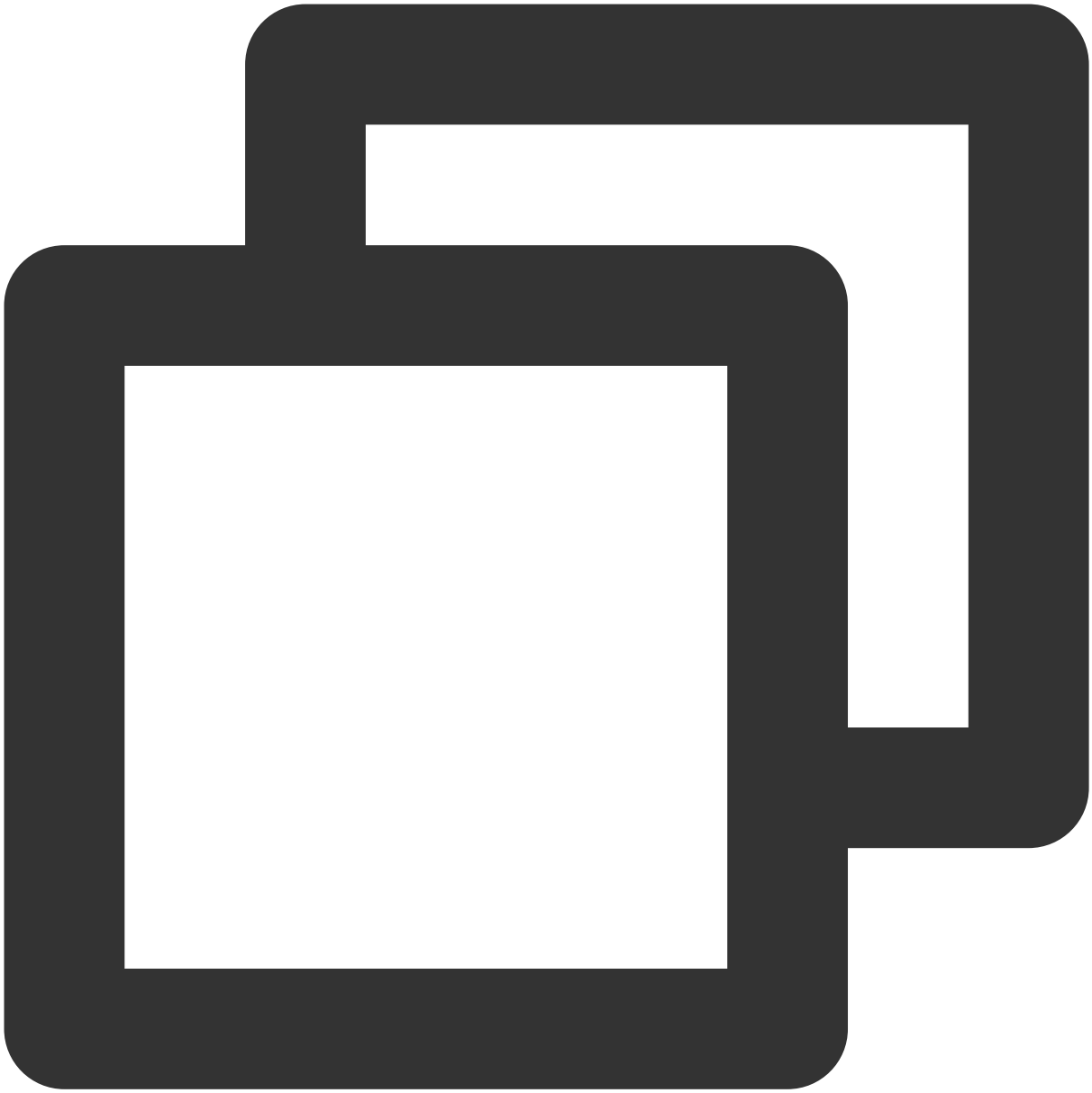
引入 JS

说明：
如果您现有的 Flutter 项目不支持 Web，请在项目根目录下运行 `flutter create .` 添加 Web 支持。
进入您项目的 `web/` 目录，使用 `npm` 或 `yarn` 安装相关JS依赖。初始化项目时，根据屏幕指引，进行即可。

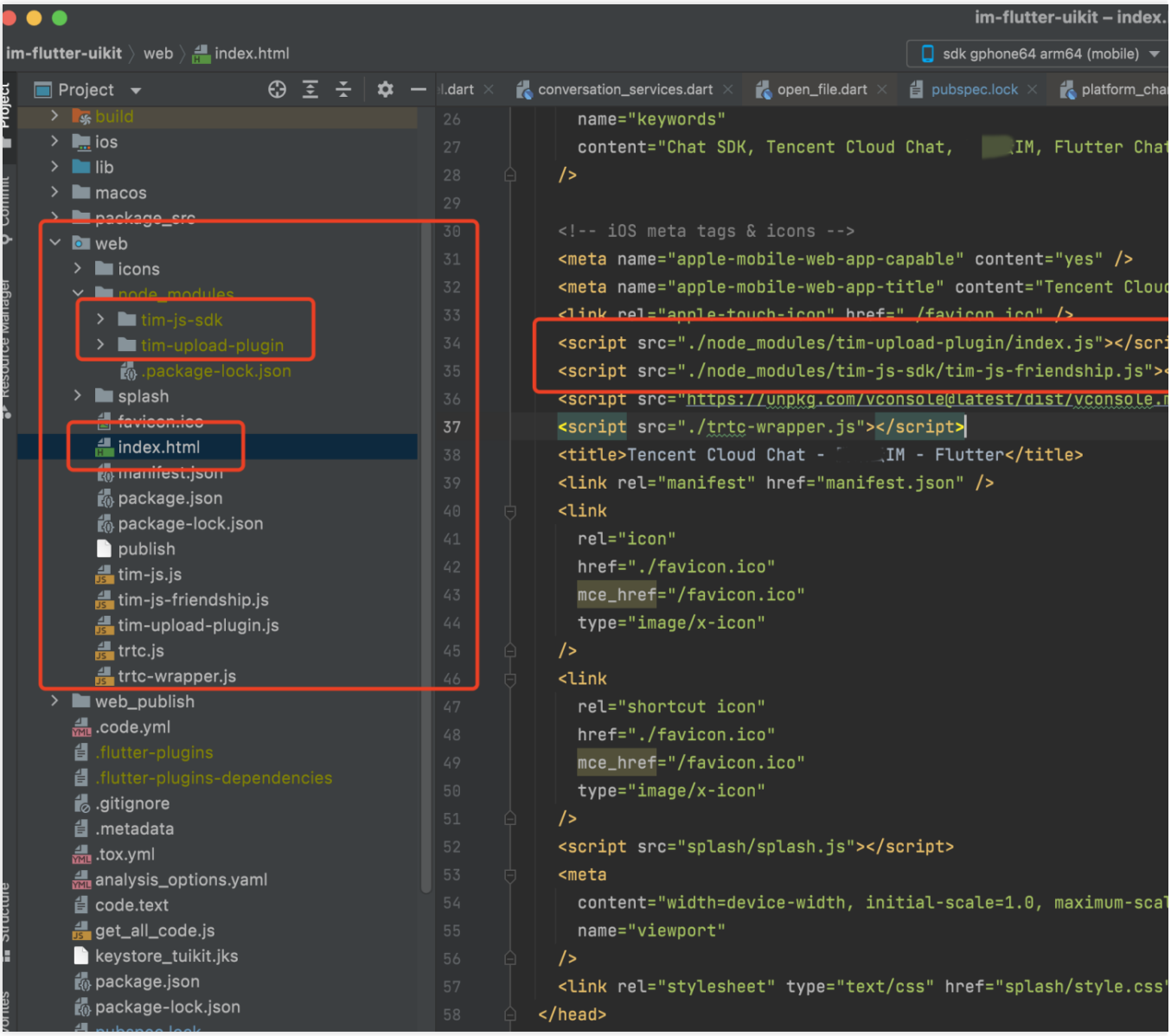


```
cd web

npm init

npm i tim-js-sdk

npm i tim-upload-plugin
```

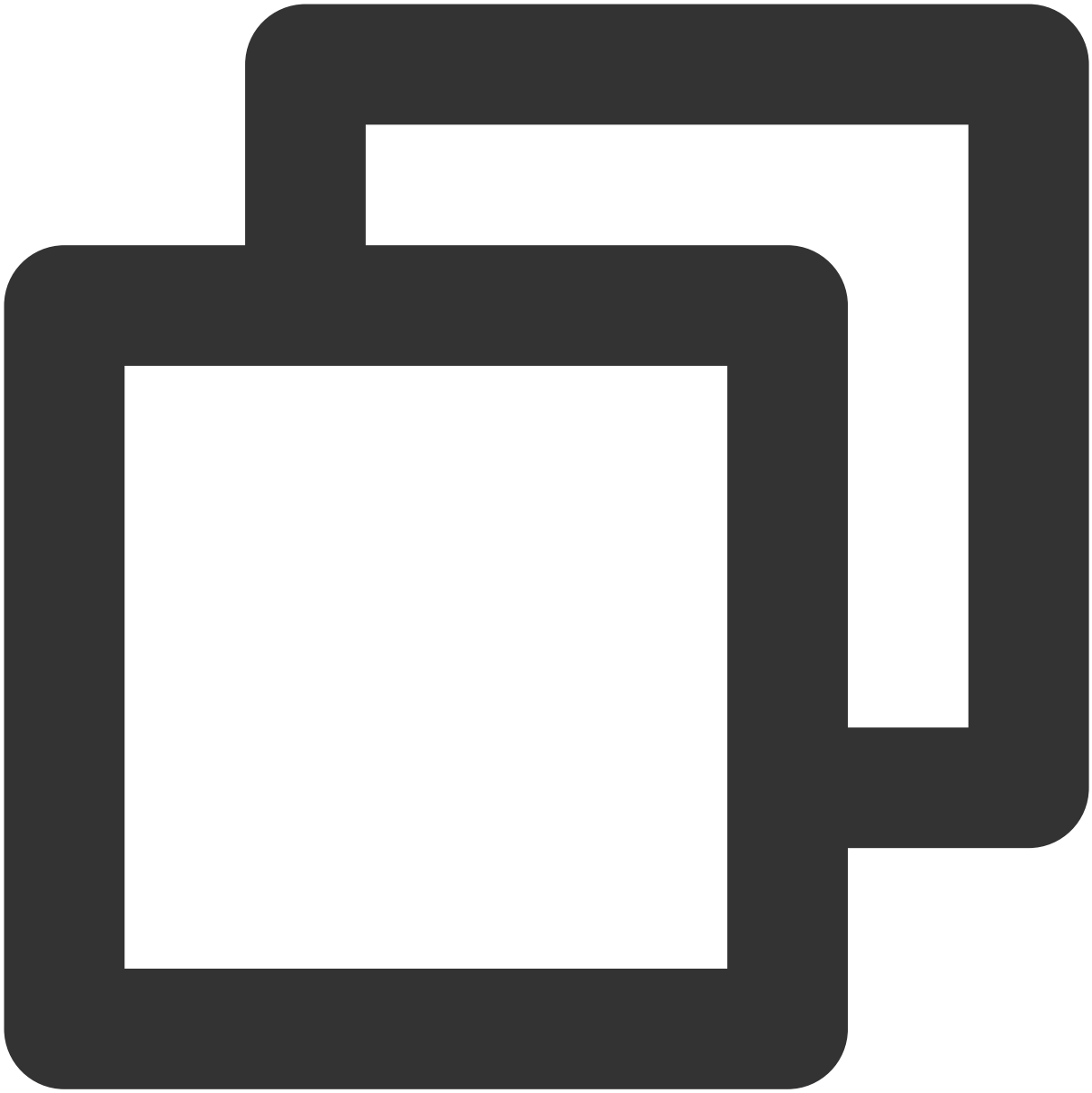
打开 `web/index.html` ，在 `<head> </head>` 间引入这JS文件。如下：



```
<script src="./node_modules/tim-upload-plugin/index.js"></script>
<script src="./node_modules/tim-js-sdk/tim-js-friendship.js"></script>
```



引入 Flutter for Web 增补 SDK



```
flutter pub add tencent_im_sdk_plugin_web
```

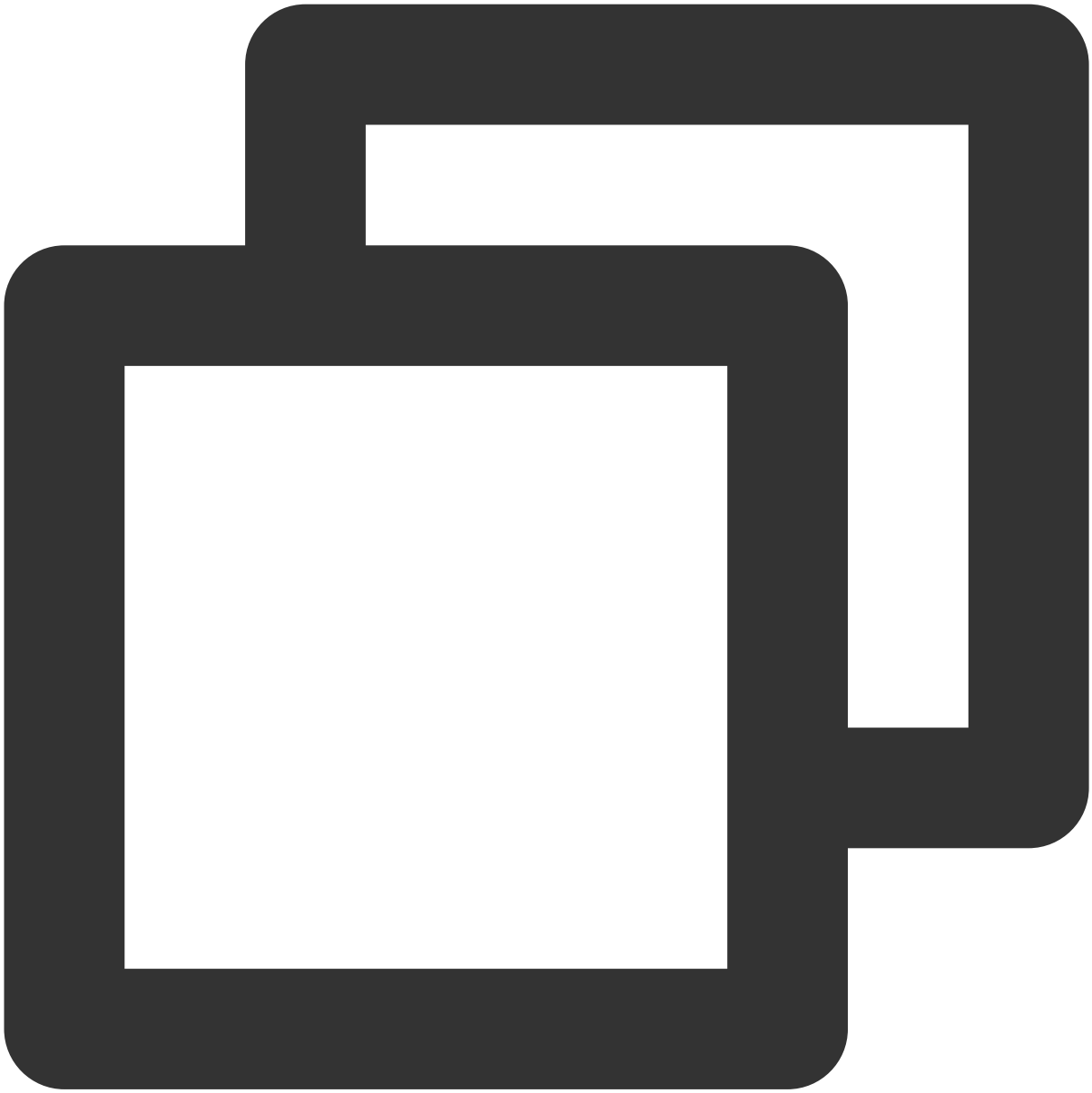
Flutter for Desktop(PC) 支持

我们的无 UI SDK(tencent_cloud_chat_sdk) 4.1.9 版本起，可完美兼容 macOS、Windows 端。
相比 Android 和 iOS 端，需要一些额外步骤。如下：

升级 Flutter 3.x 版本

从 Flutter 3.0 版本起，才可用于打包 desktop 端，因此，如需使用，请升级至 Flutter 3.x 版本。

引入 Flutter for Desktop 增补 SDK

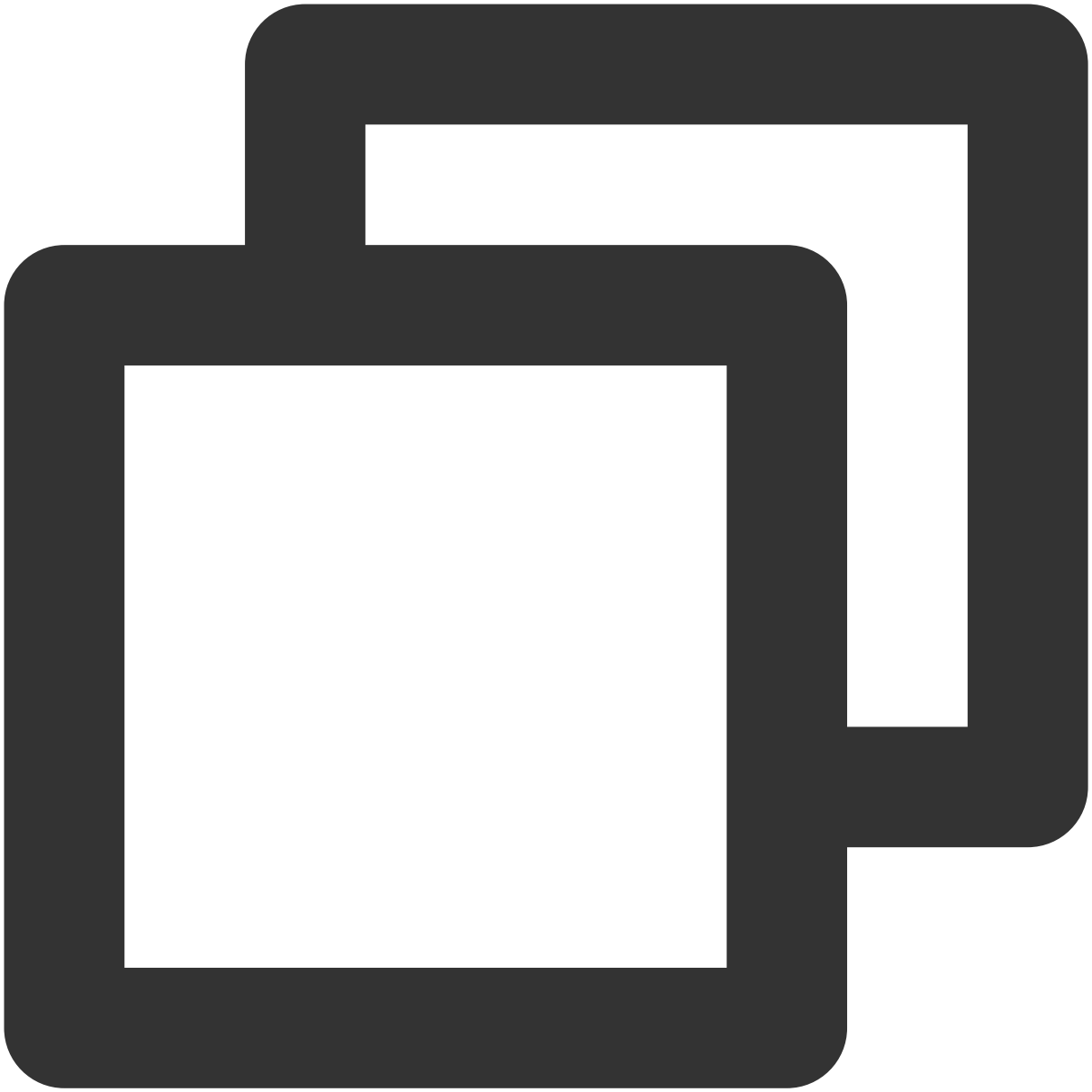


```
flutter pub add tencent_im_sdk_plugin_desktop
```

macOS 修改

打开 `macos/Runner/DebugProfile.entitlements` 文件。

在 `<dict></dict>` 中，加入如下 `key-value` 键值对。



```
<key>com.apple.security.app-sandbox</key>
<false/>
```

常见问题

flutter pub get/add 失败如何解决？

请参见官网配置 国内镜像。

Windows

最近更新时间：2024-07-24 16:56:49

本文主要介绍如何快速将腾讯云即时通信 IM SDK 集成到您的 Windows 项目中。

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 2019。

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 2019 工程中集成 SDK。

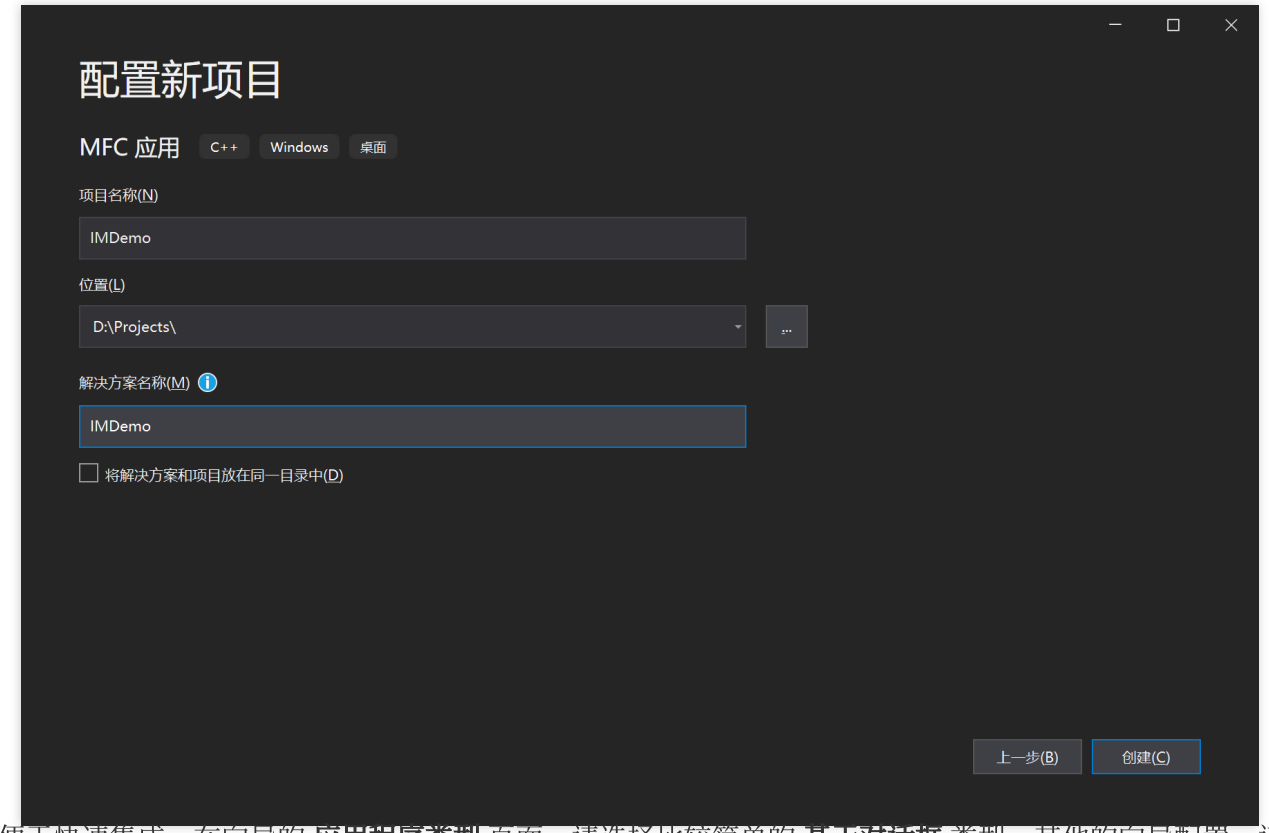
步骤1：下载 IM SDK

在 Github 下载 Windows IM SDK，下载并解压 IM SDK，为方便可将解压后的文件夹重命名为 ImSDK，其中包含以下几个部分：

目录名	说明
c_include	C 接口头文件
cpp_include	C++ 接口头文件
shared_lib\\Win32	**32 位 Release 模式**，采用 /MT 选项链接库文件
shared_lib\\Win64	**64 位 Release 模式**，采用 /MT 选项链接库文件

步骤2：新建工程

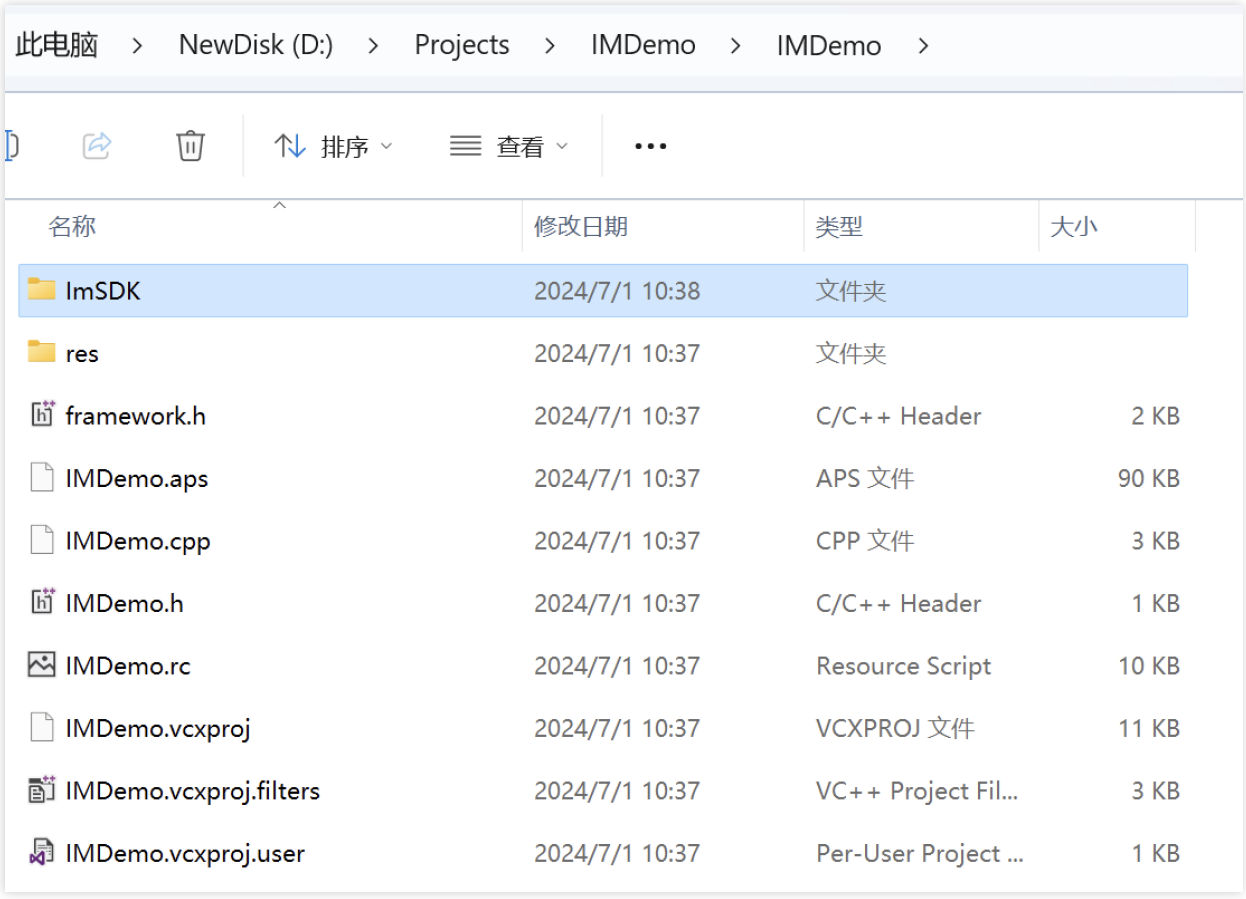
打开 Visual Studio，新建一个名为 IMDemo 的 MFC 应用程序（若 MFC 应用不在备选项前列，可借助上方的“搜索模板”进行查找），如下图所示：



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（即 步骤1 中获取的 ImSDK 文件夹）拷贝到 IMDemo.vcxproj 所在目录下，如下图所示：



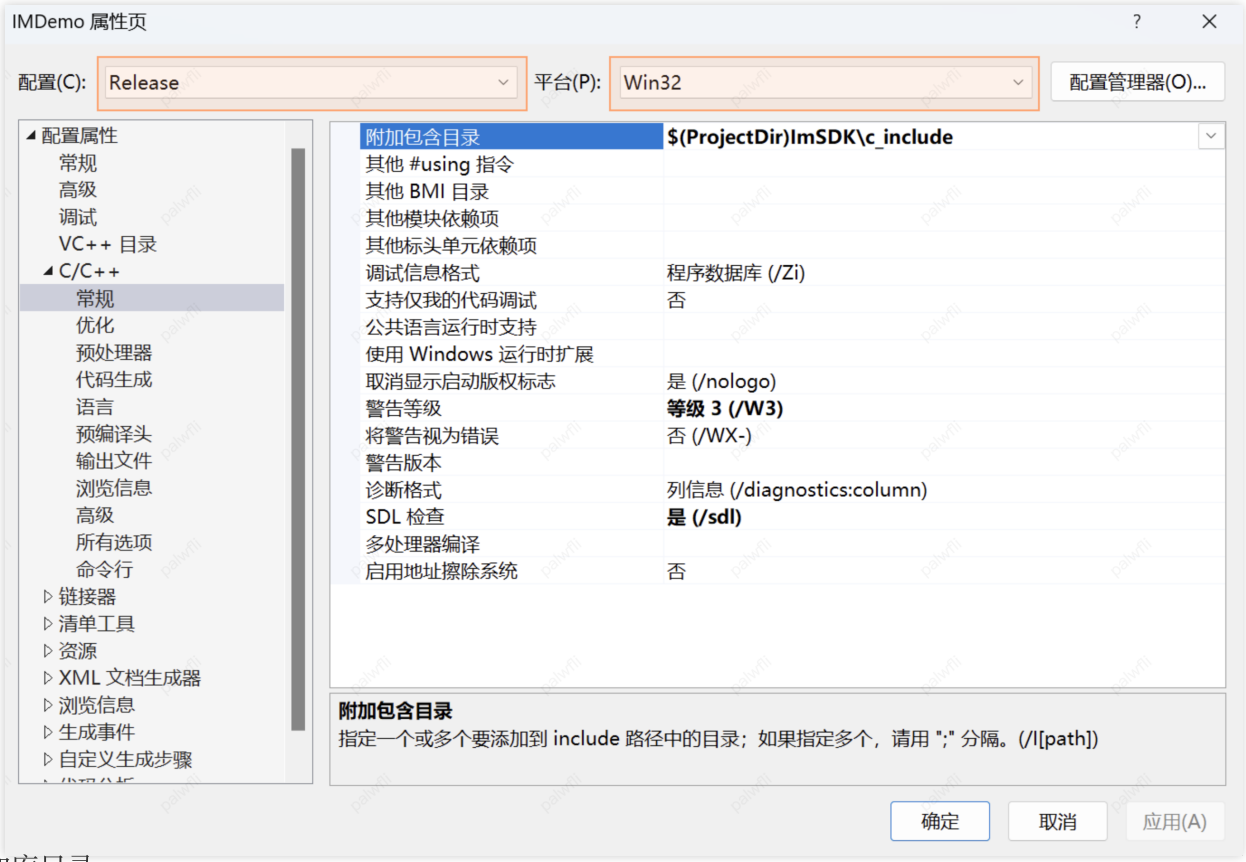
步骤4：修改工程配置

IM SDK 中提供了 **Release** 模式下 32 位和 64 位的动态库，针对这两类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 Release 模式** 为例，请按照以下步骤进行配置：

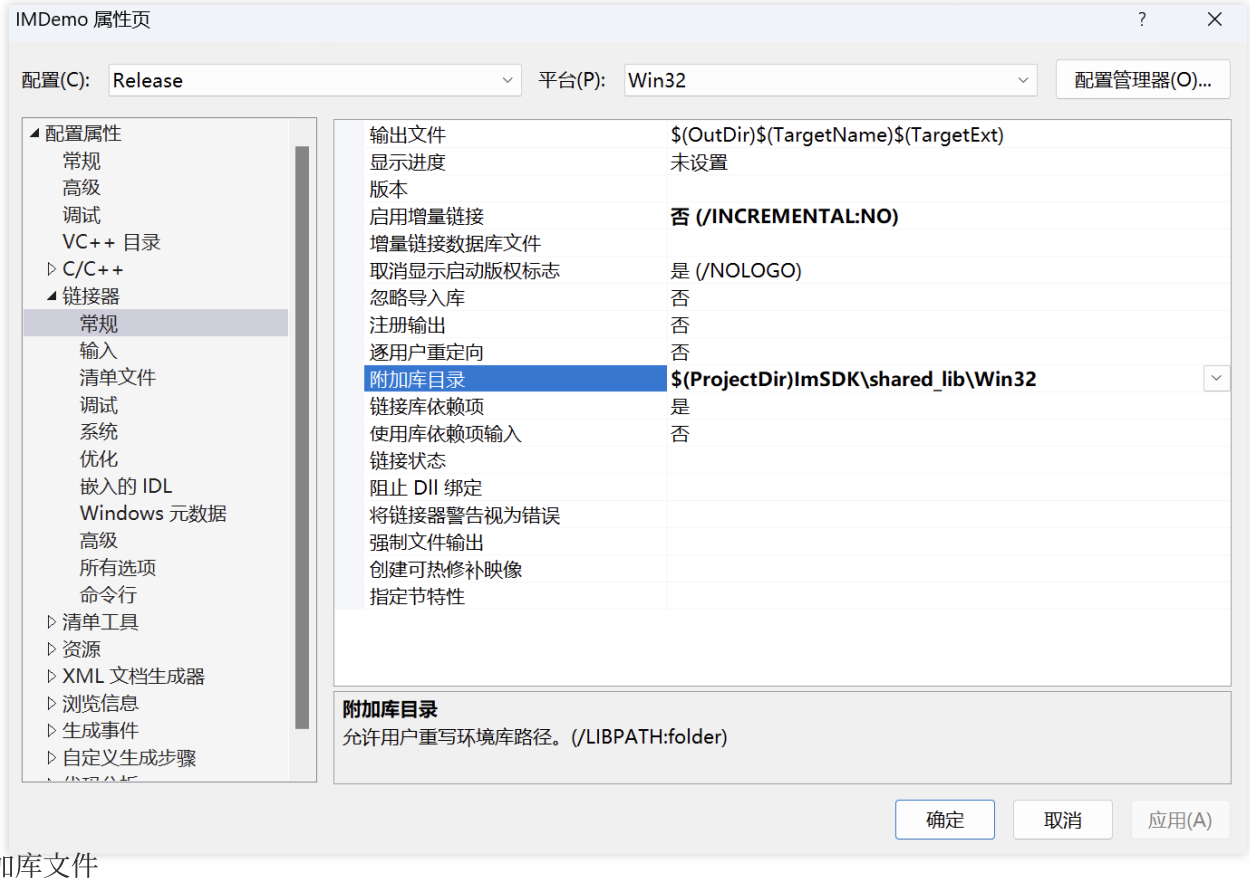
1. 添加包含目录

在 **C/C++ > 常规 > 附加包含目录**，添加 IM SDK C 接口的头文件目录 $(ProjectDir)ImSDK\\c_include，如下图所示：



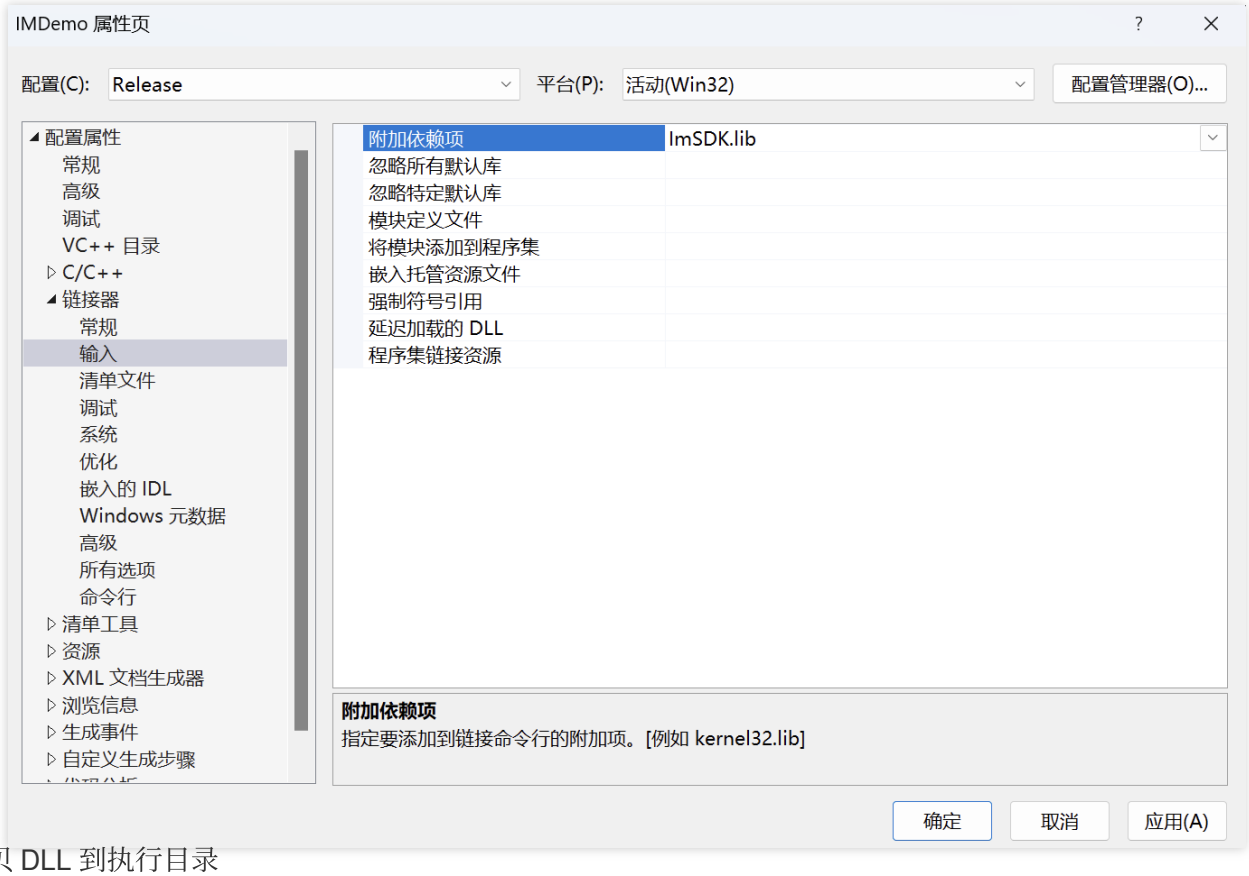
2. 添加库目录

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



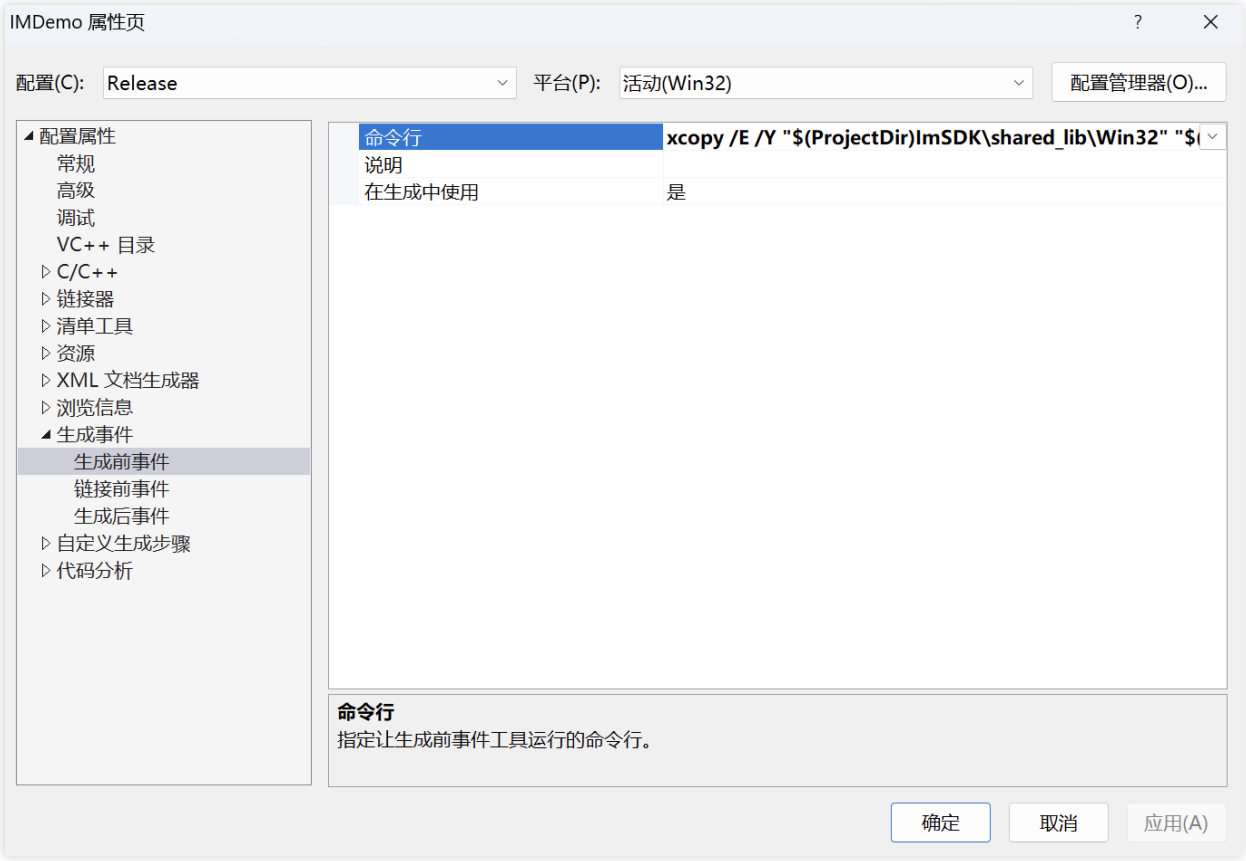
3. 添加库文件

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

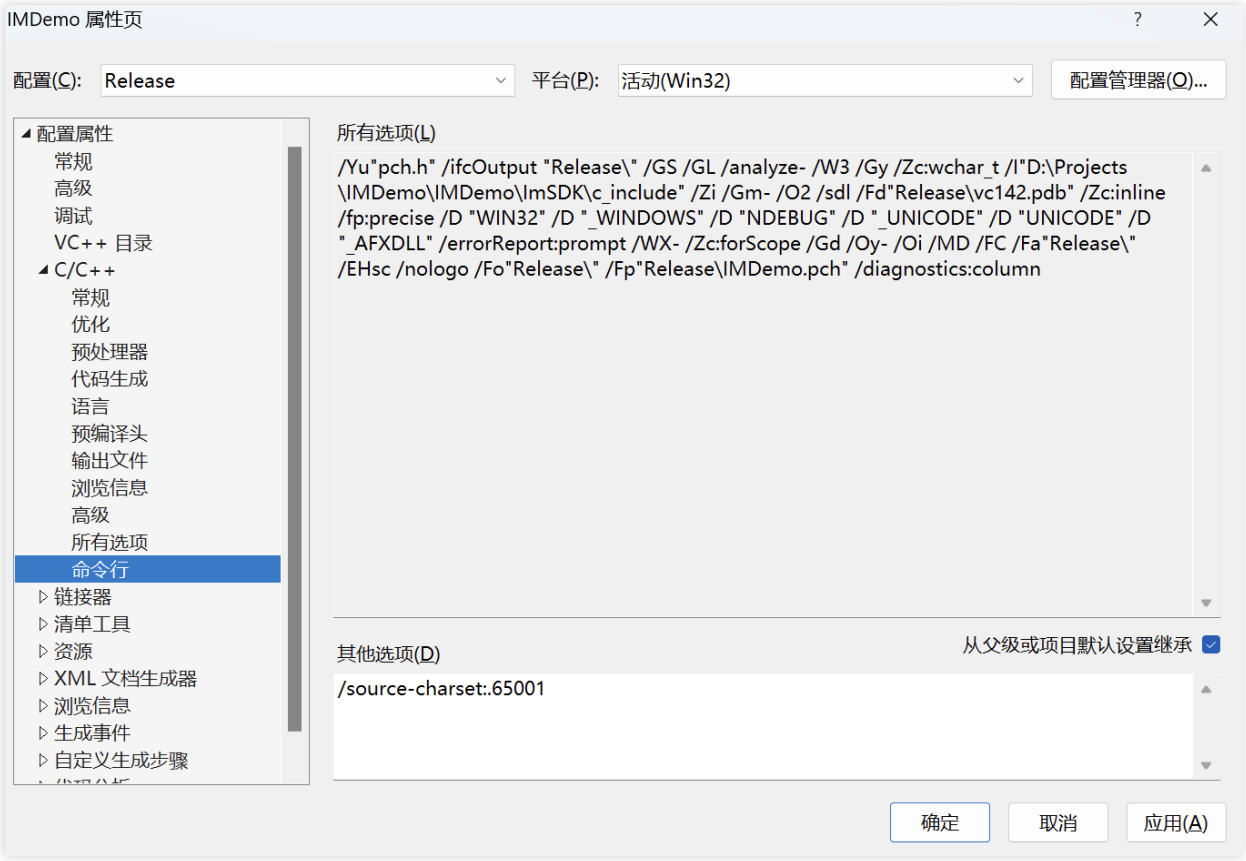
在 **生成事件 > 生成前事件 > 命令行**，输入 `xcopy /E /Y "$(ProjectDir)ImSDK\\shared_lib\\Win32" "$(OutDir)"`，拷贝 ImSDK.dll 动态库文件到程序生成目录，如下图所示：



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。

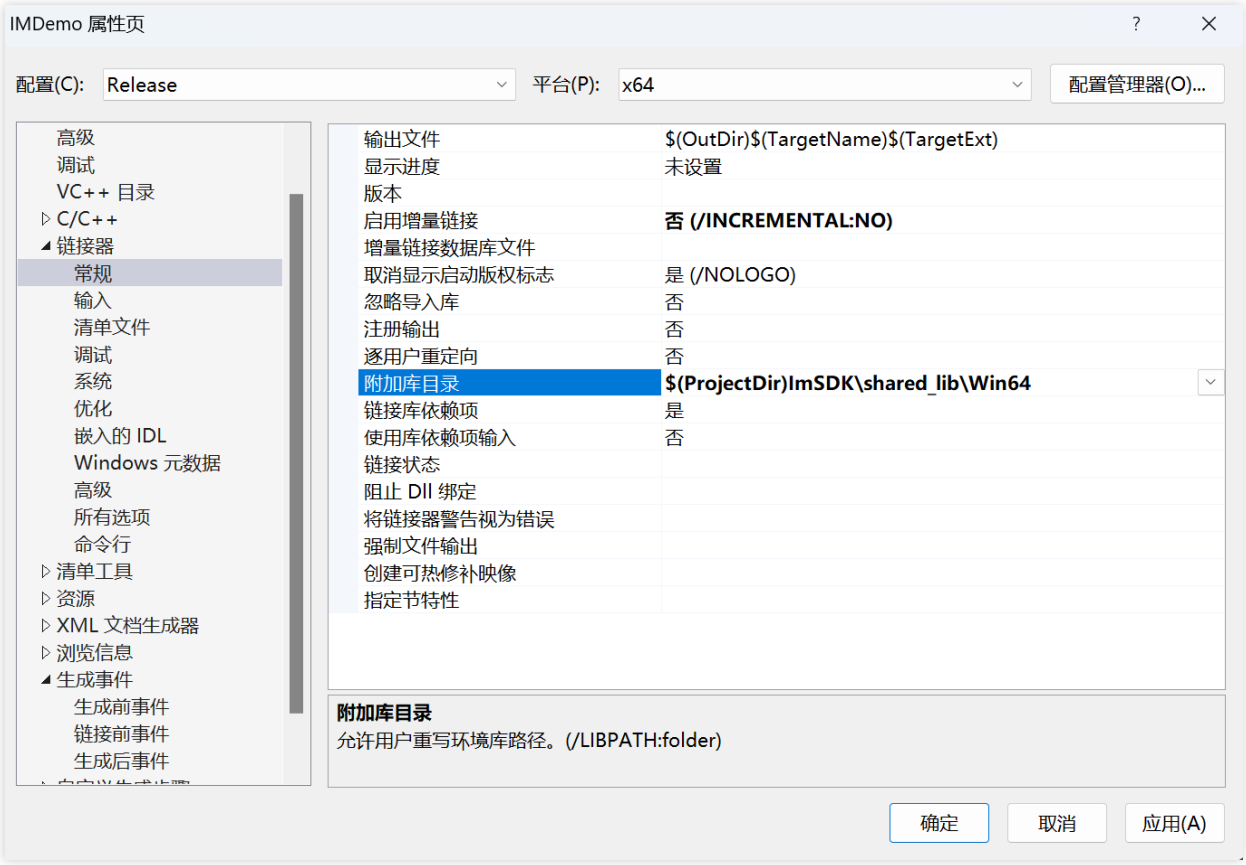
在 **C/C++** > **命令行** > **其他选项**，输入 `/source-charset:.65001` ，如下图所示：



64 位 Release 与 **32 位 Release** 的设置大部分都相同，不同在于 IM SDK 的库目录。具体如下：

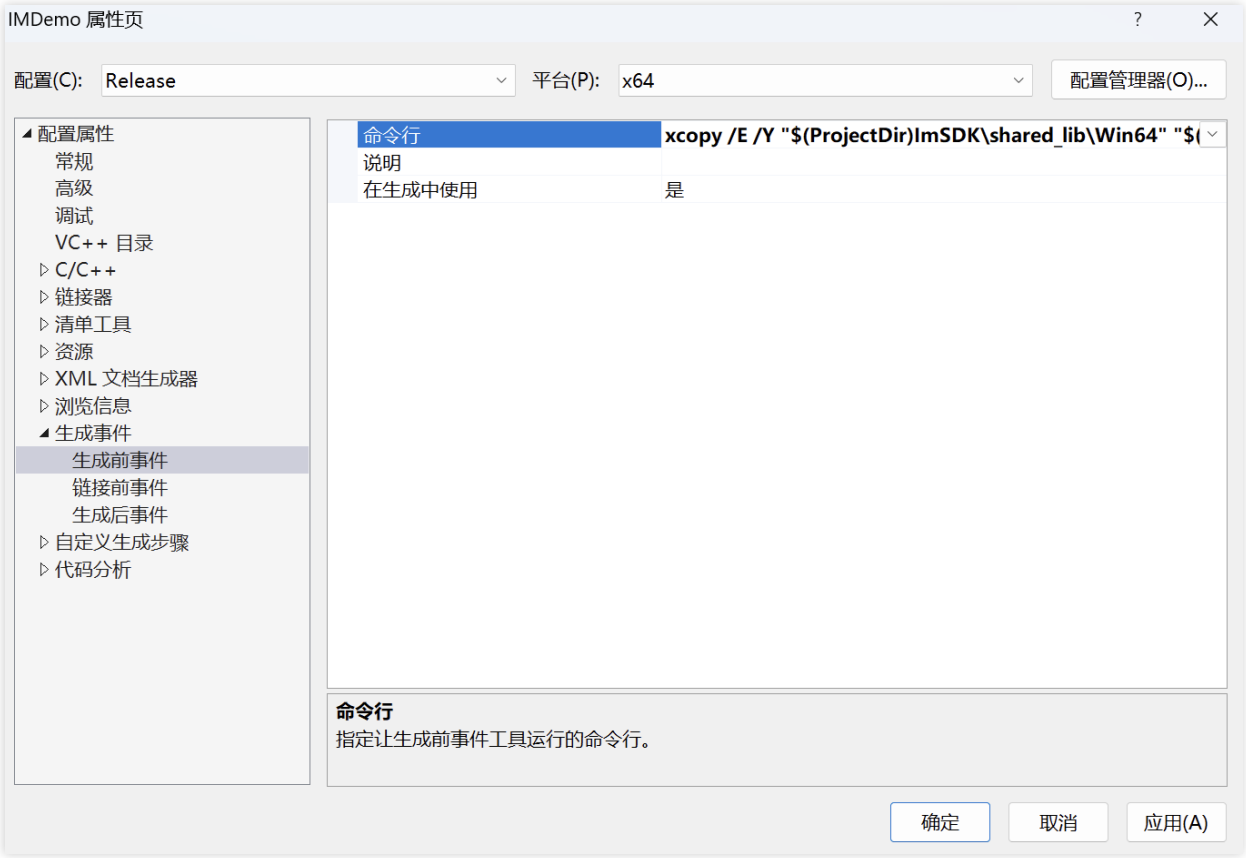
1. 添加库目录

在 **链接器** > **常规** > **附加库目录**，添加 IM SDK 库目录 $(ProjectDir)ImSDK\\shared_lib\\Win64，如下图所示：



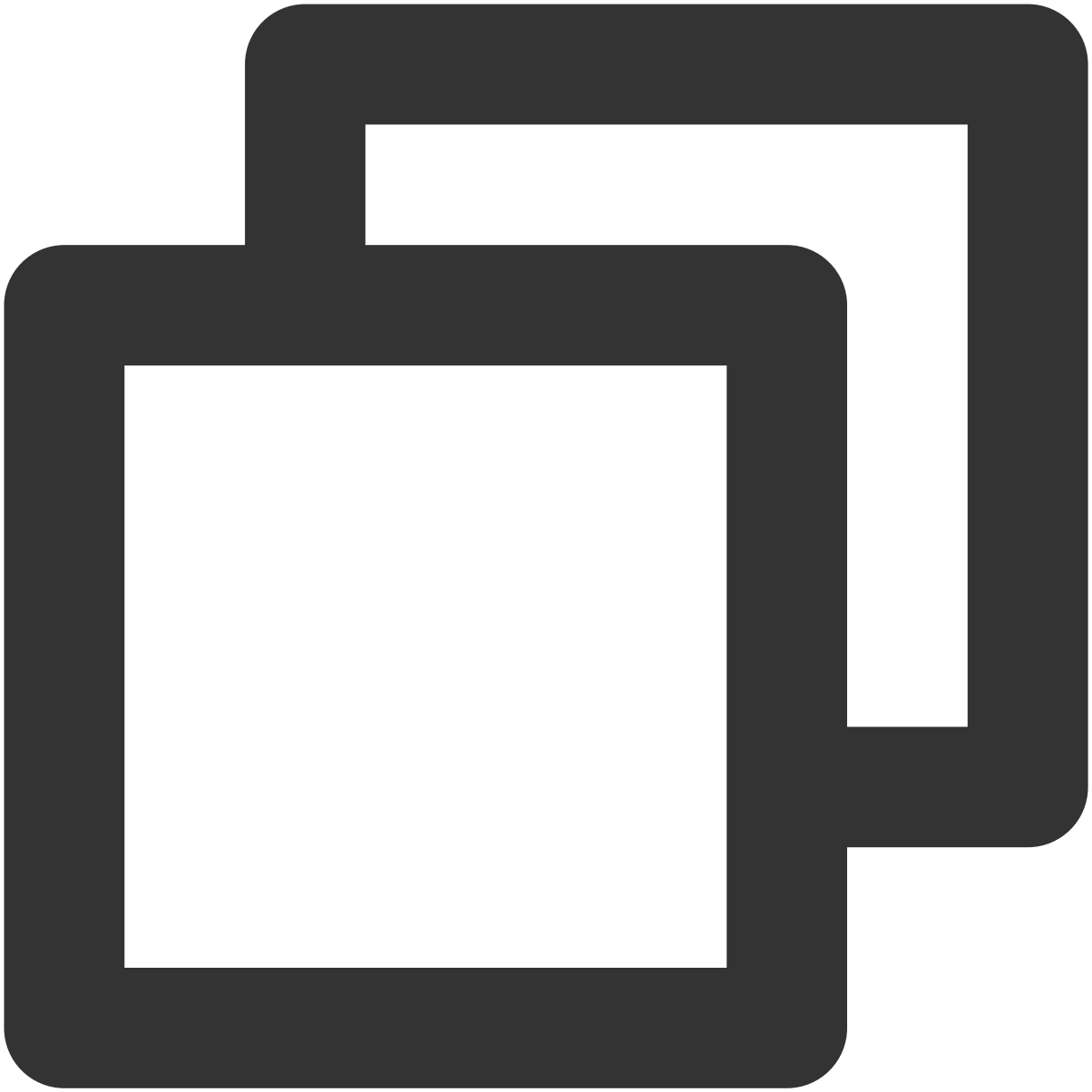
2. 拷贝 DLL 到执行目录

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



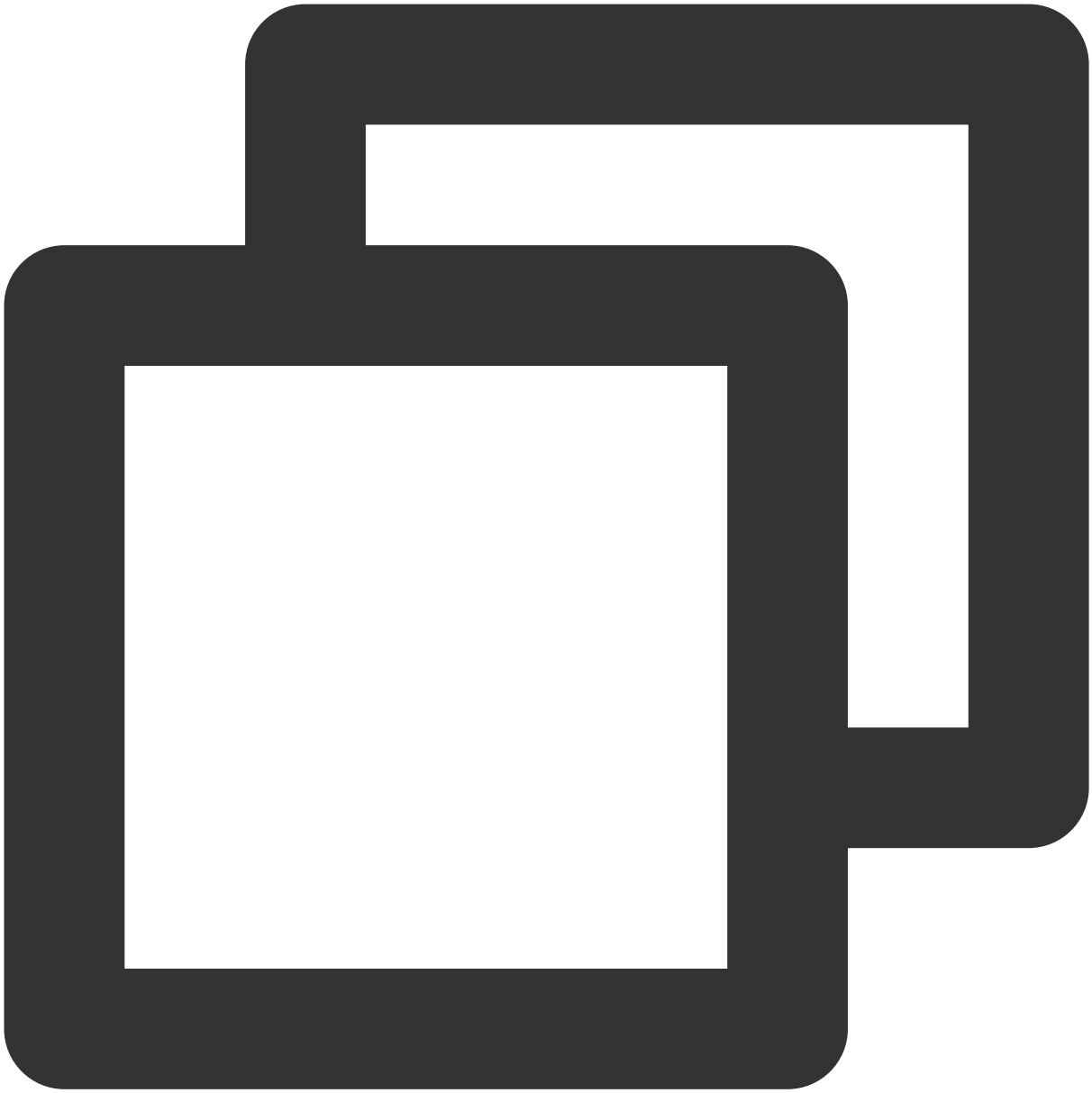
步骤5：打印 IM SDK 版本号

在 IMDemoDlg.cpp 文件中，添加头文件包含：



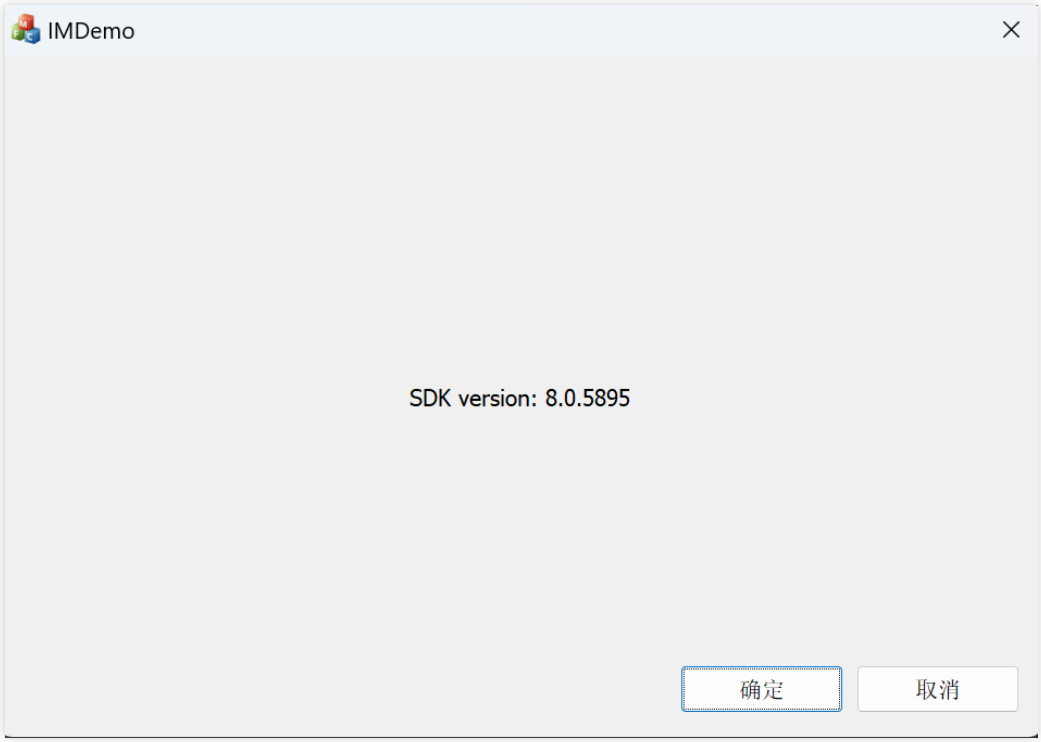
```
#include "TIMCloud.h"
```

在 IMDemoDlg.cpp 文件中找到 CIMDemoDlg::OnInitDialog 函数，在 return 前添加下面的测试代码：



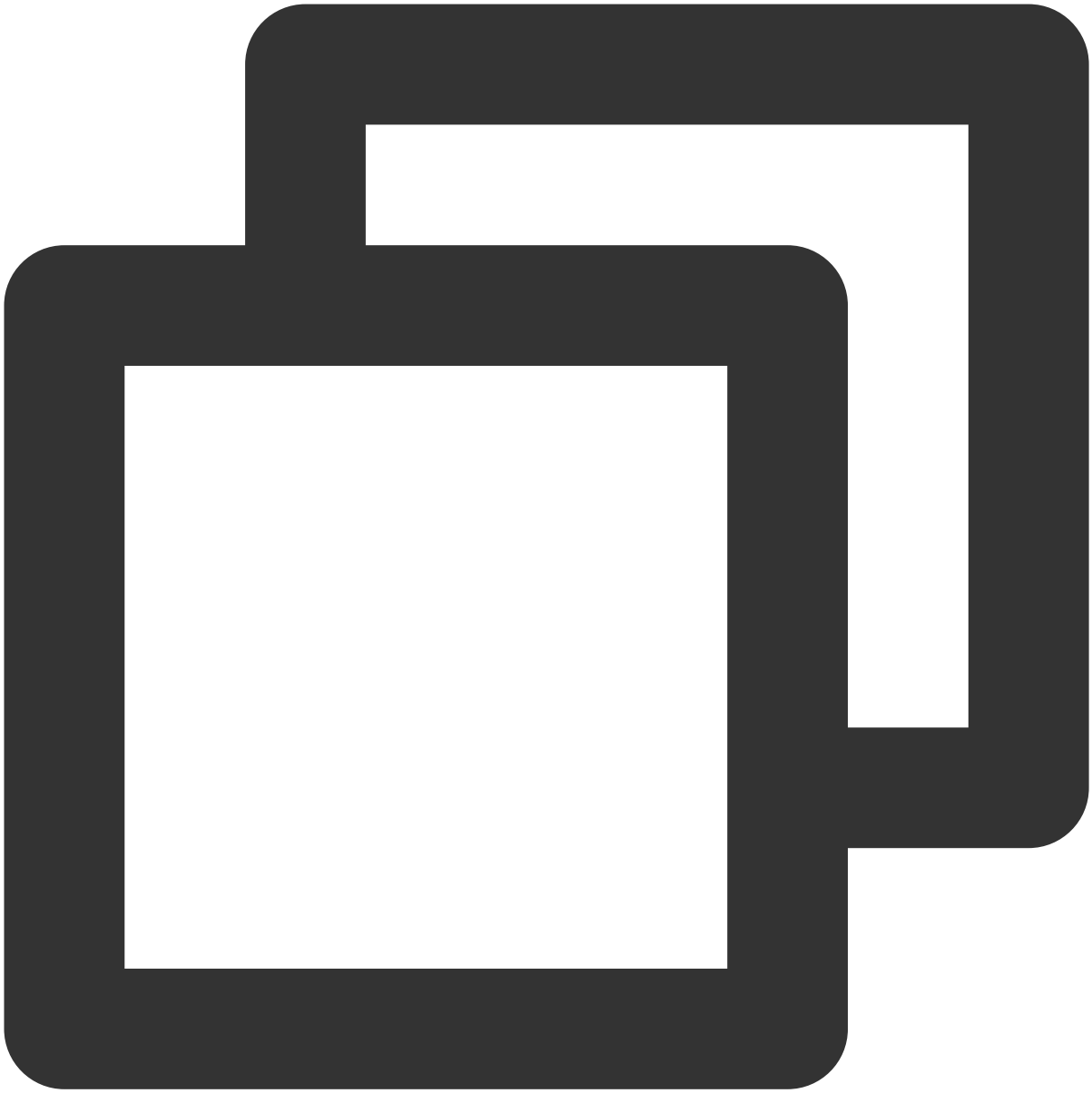
```
SetWindowText(L"IMDemo");
CString szText;
szText.Format(L"SDK version: %hs", TIMGetSDKVersion());
CWnd* pStatic = GetDlgItem(IDC_STATIC);
pStatic->SetWindowTextW(szText);
```

按键盘 F5 键运行，打印 IM SDK 的版本号，如下图所示：



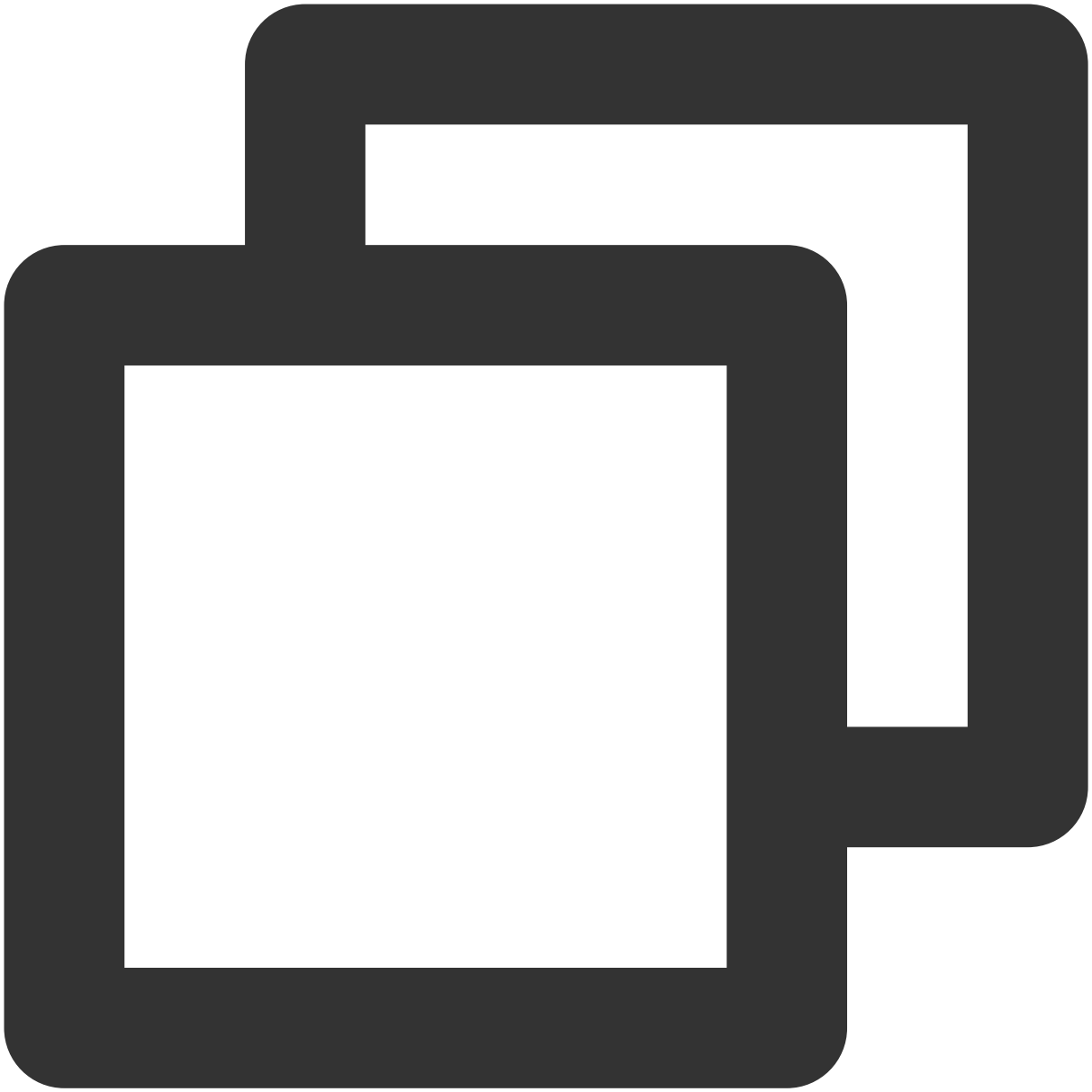
常见问题

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：

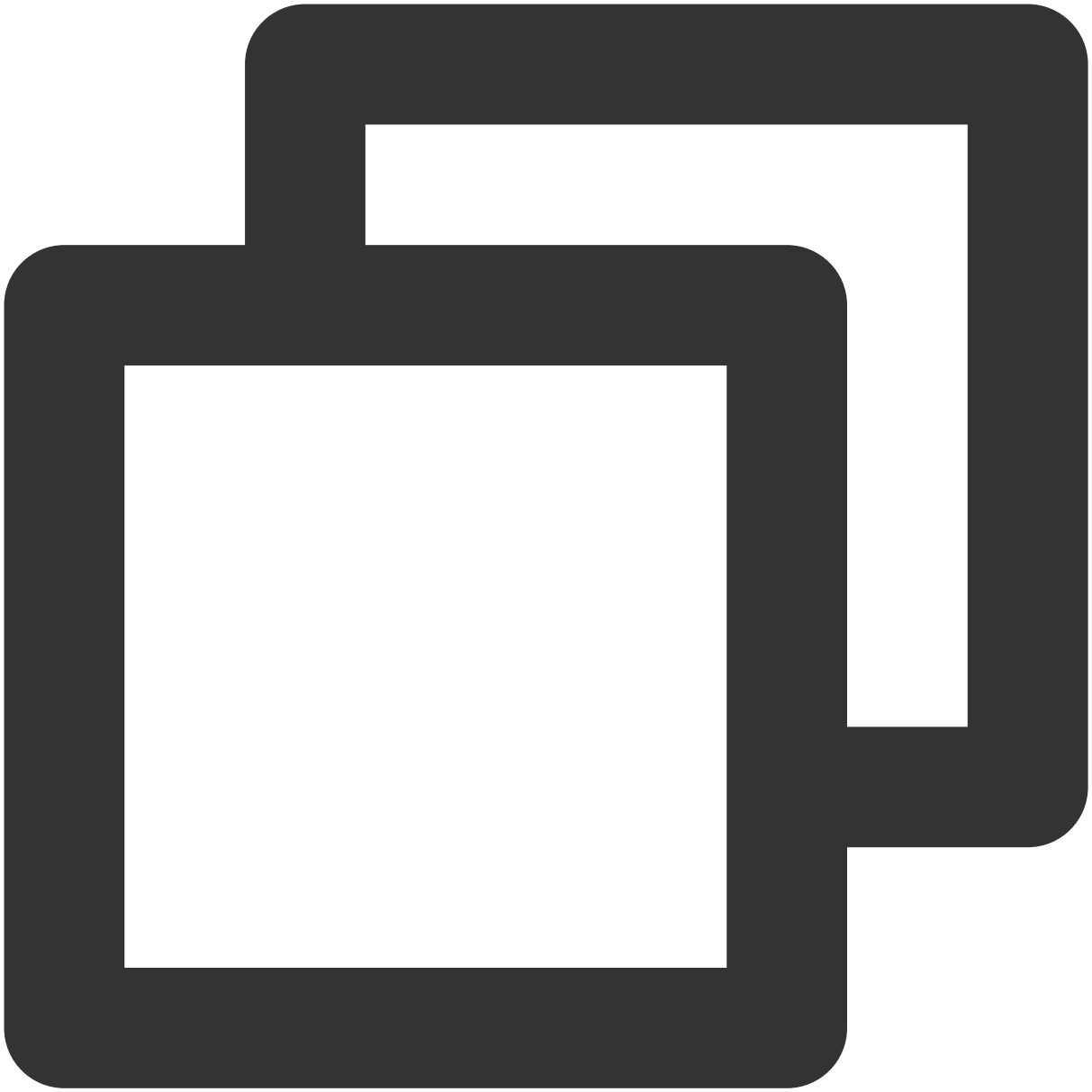


```
fatal error C1083: 无法打开包括文件: “TIMCloud.h”: No such file or directory
```

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: virtu
```

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：


IMdemo.exe - 系统错误
由于找不到 ImSDK.dll，无法继续执行代码。重新安装程序可能会解决此问题。
确定

Mac

最近更新时间：2024-01-31 15:27:03

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

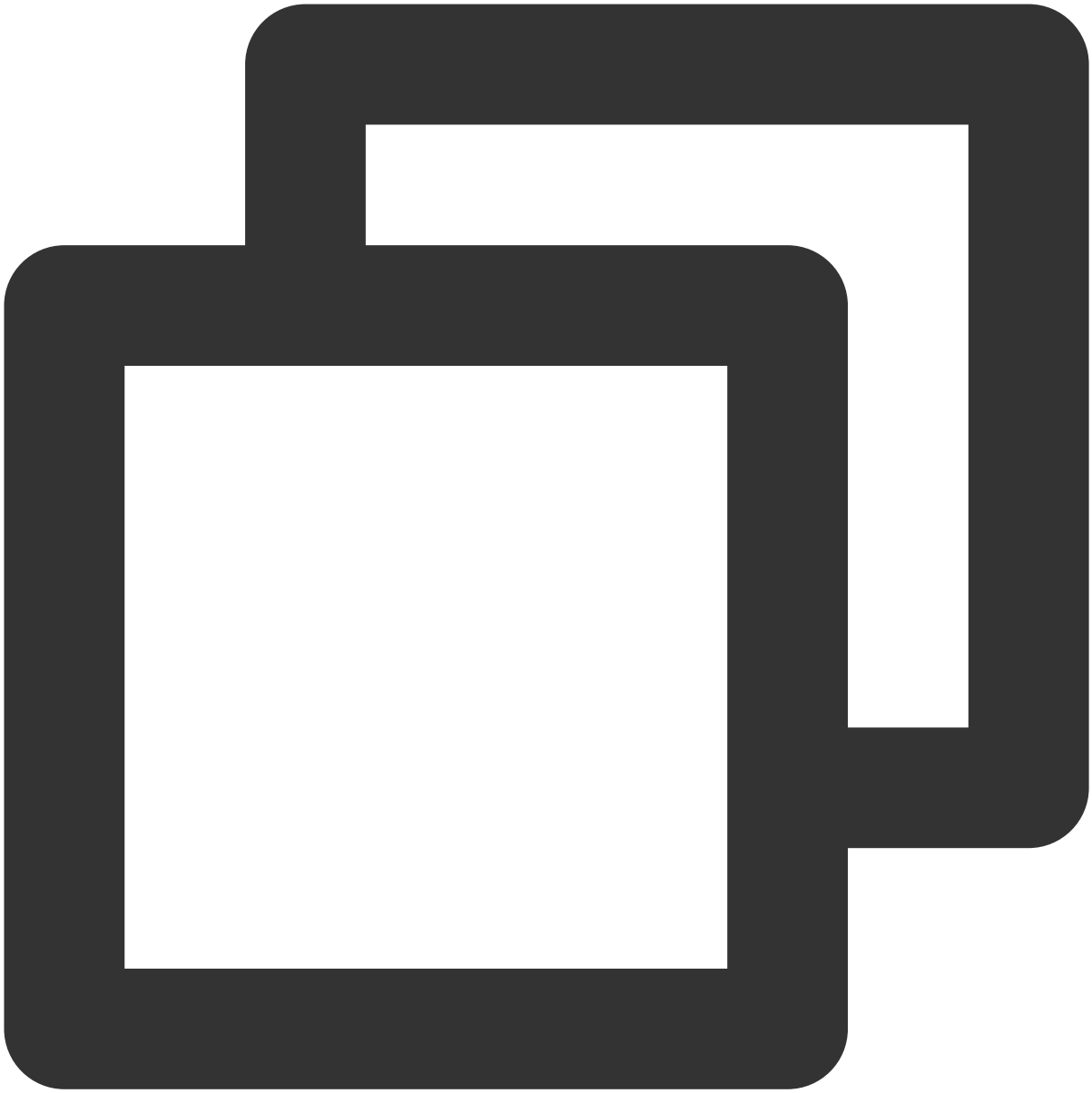
您可以选择使用 CocoaPods 自动加载的方式，或者先下载 SDK 再将其导入到您当前的工程项目中。

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）：

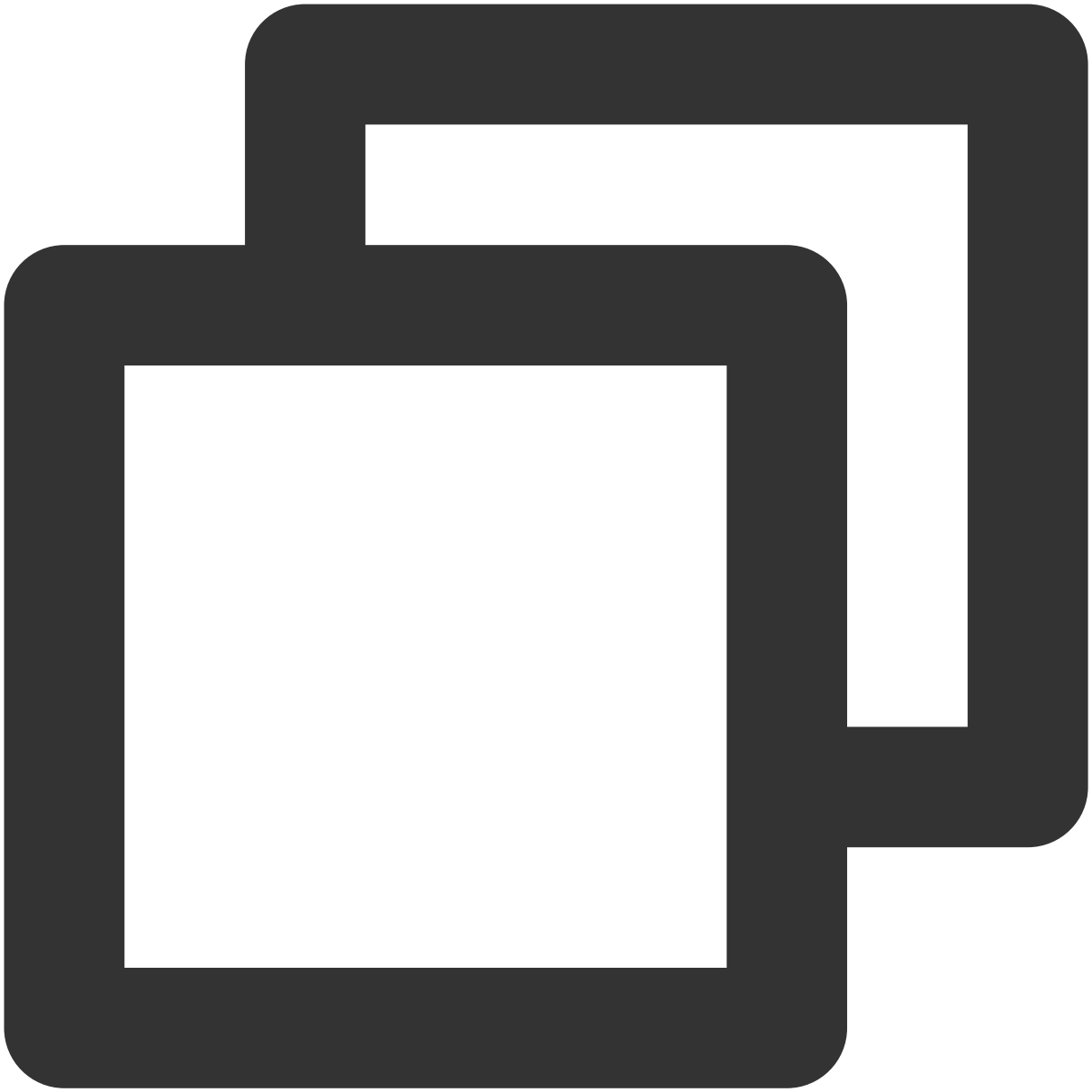
版权所有：腾讯云计算（北京）有限责任公司



```
sudo gem install cocoapods
```

2. 创建 Podfile 文件

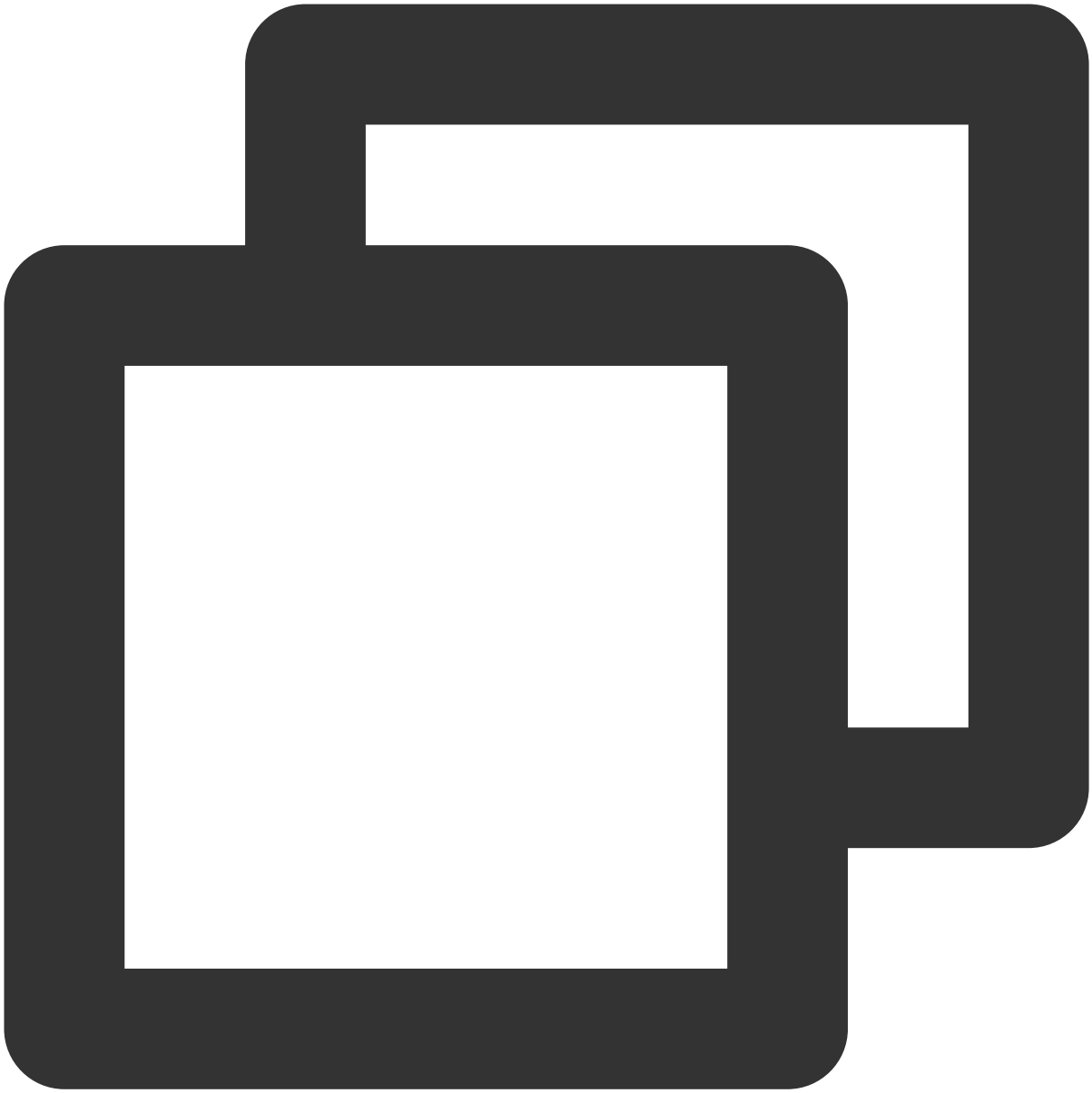
进入项目所在路径输入以下命令行，之后项目路径下会出现一个 Podfile 文件。



```
pod init
```

3. 编辑 Podfile 文件

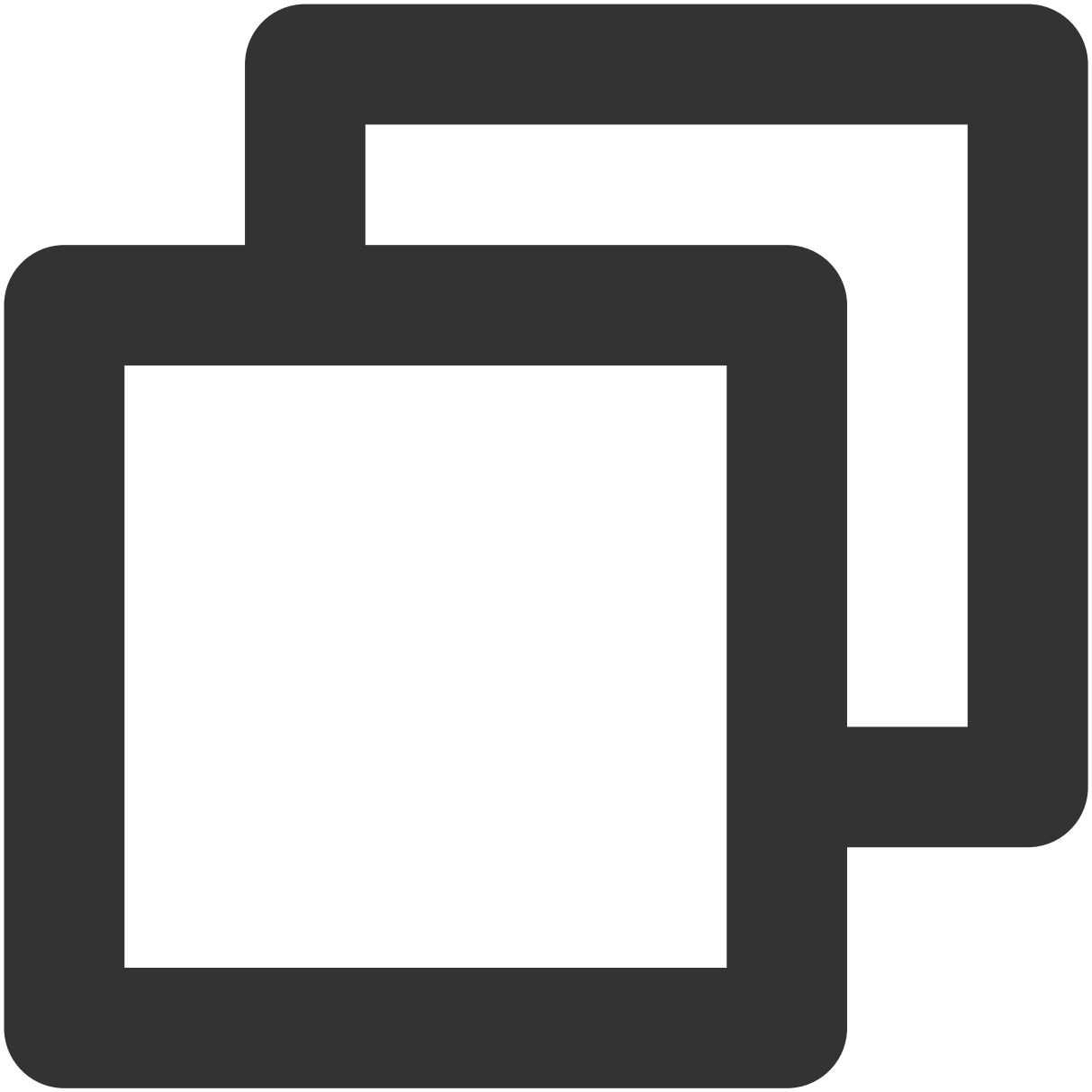
编辑 Podfile 文件，按如下方式设置：



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

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

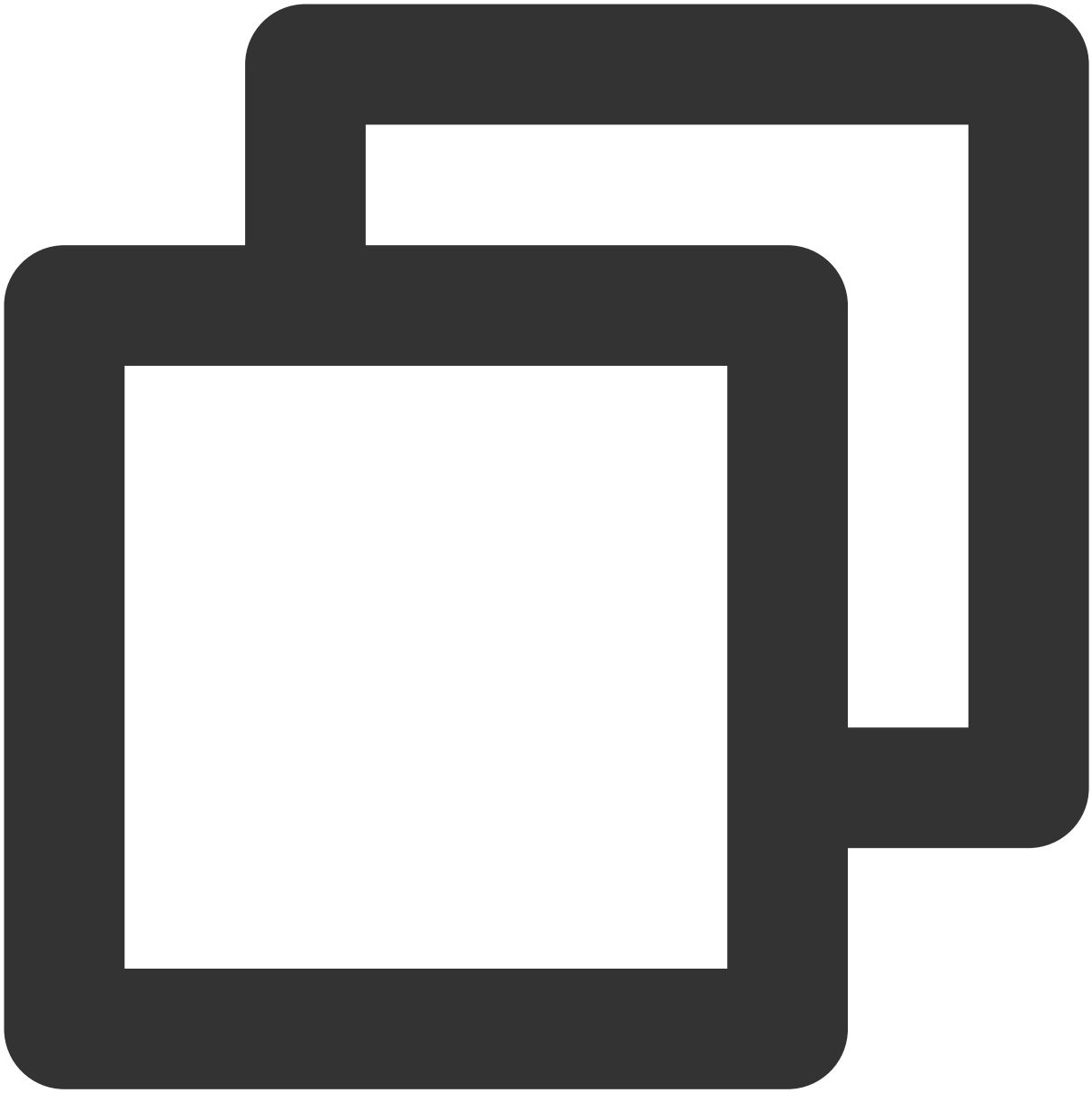
4. 更新并安装 SDK

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：

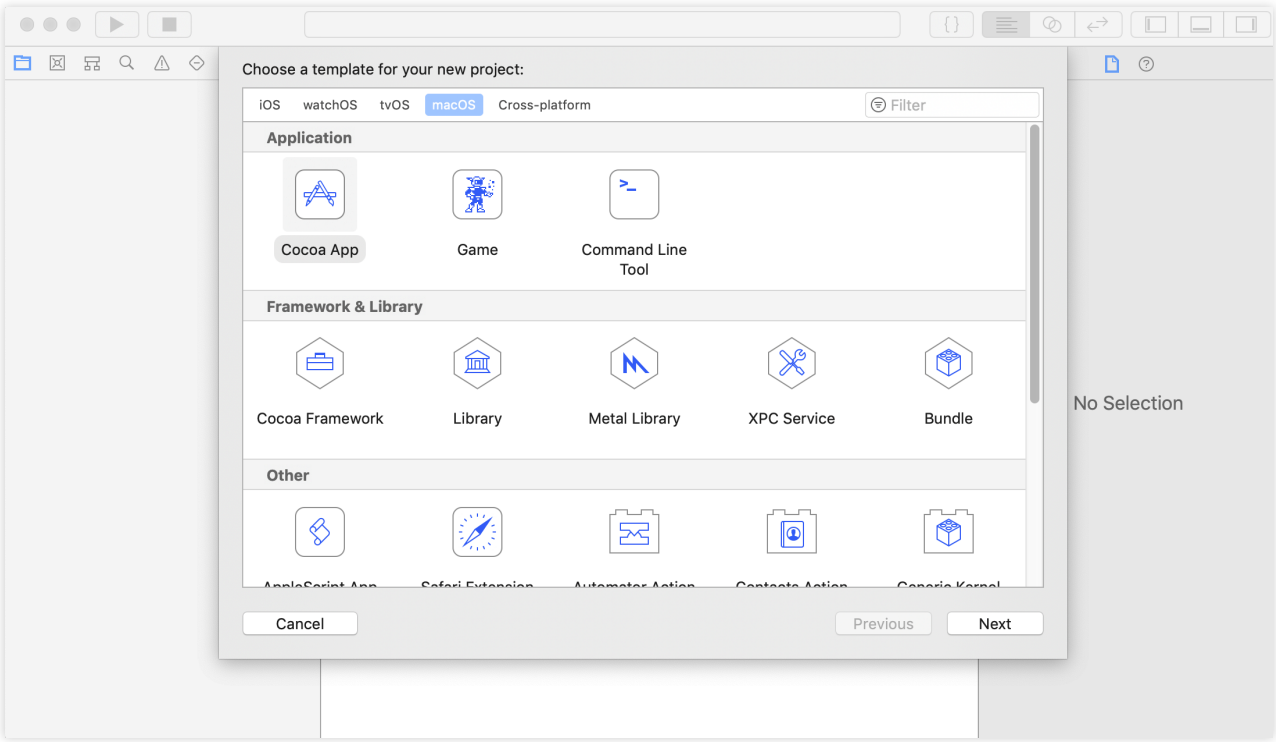
ImSDKForMac.framework 为 IM SDK 的核心动态库文件。

包名	介绍	ipa增量

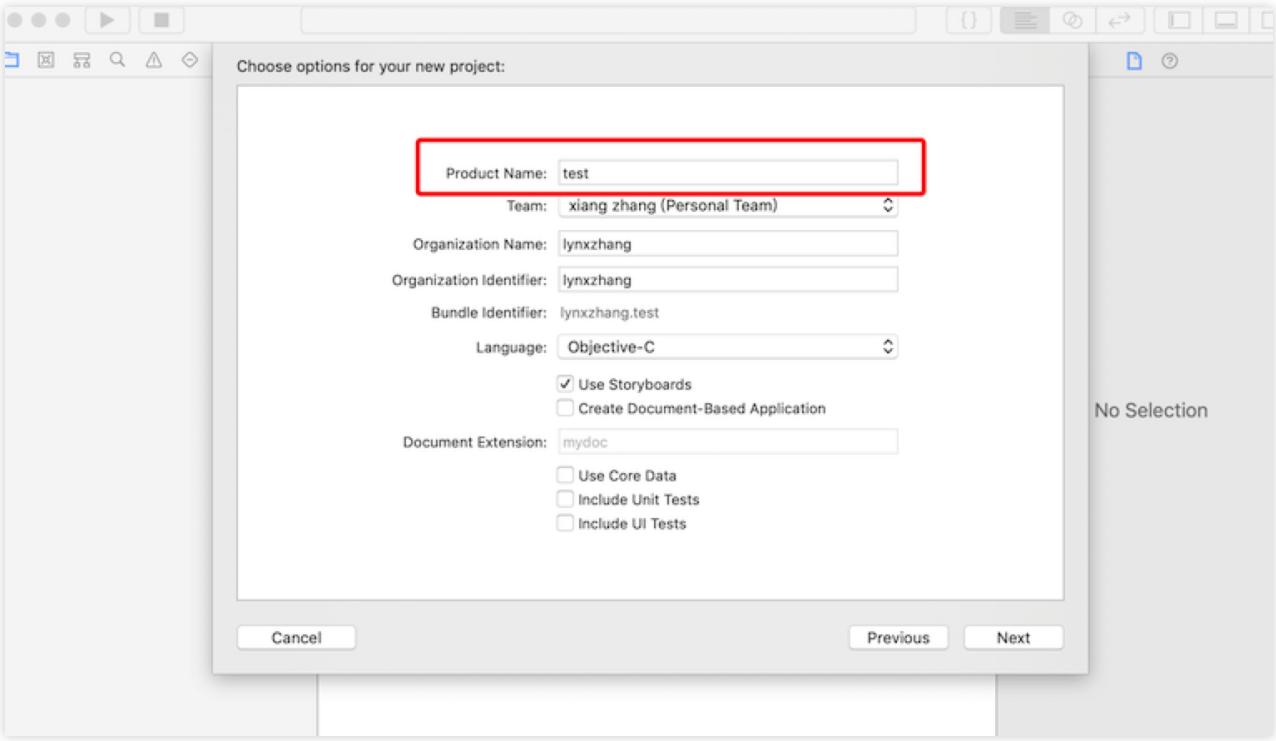
ImSDKForMac.framework	即时通信 IM 功能包	1.4M

2. 创建工程

创建一个新的工程：

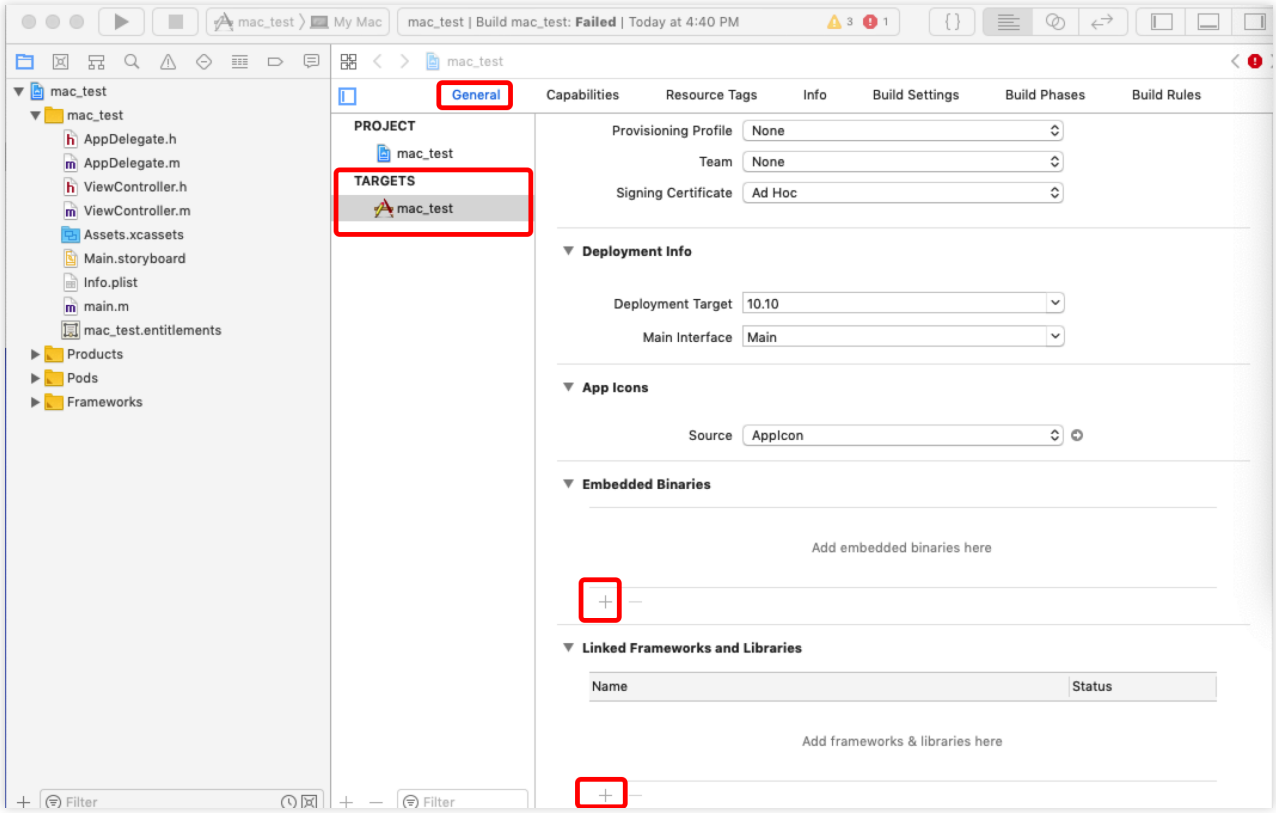


填入工程名：

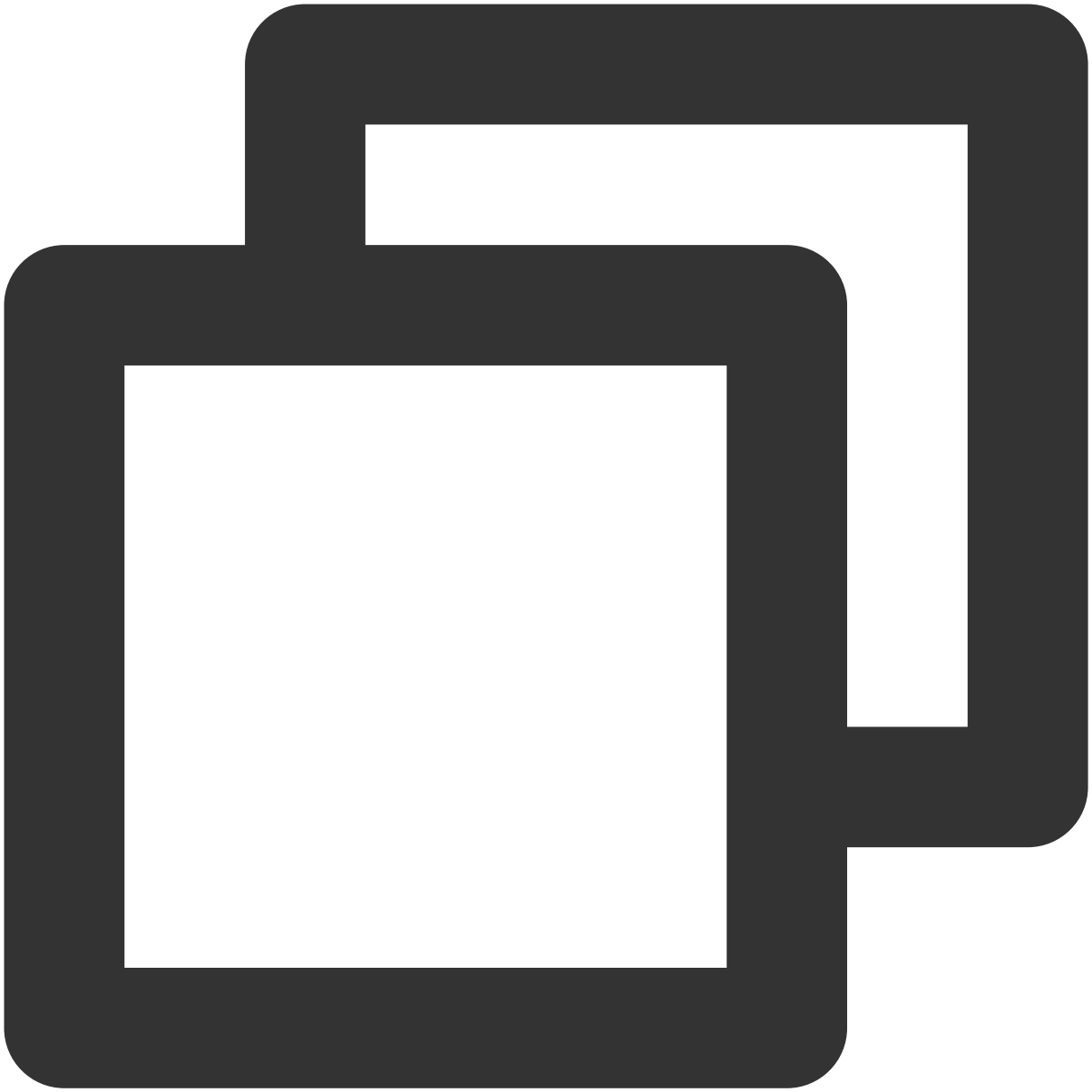


2. 集成 IM SDK

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。


mac_test | Build mac_test: Failed | Today at 4:40 PM
mac_test
General
Capabilities
Resource Tags
Info
Build Settings
Build Phases
Build Rules
PROJECT
mac_test
TARGETS
mac_test
Provisioning Profile
None
Team
None
Signing Certificate
Ad Hoc
Deployment Info
Deployment Target
10.10
Main Interface
Main
App Icons
Source
AppIcon
Embedded Binaries
Add embedded binaries here
Linked Frameworks and Libraries
Name
Status
Add frameworks & libraries here
AppDelegate.h
AppDelegate.m
ViewController.h
ViewController.m
Assets.xcassets
Main.storyboard
Info.plist
main.m
mac_test.entitlements
Products
Pods
Frameworks

添加依赖库：



```
ImSDKForMac.framework
```

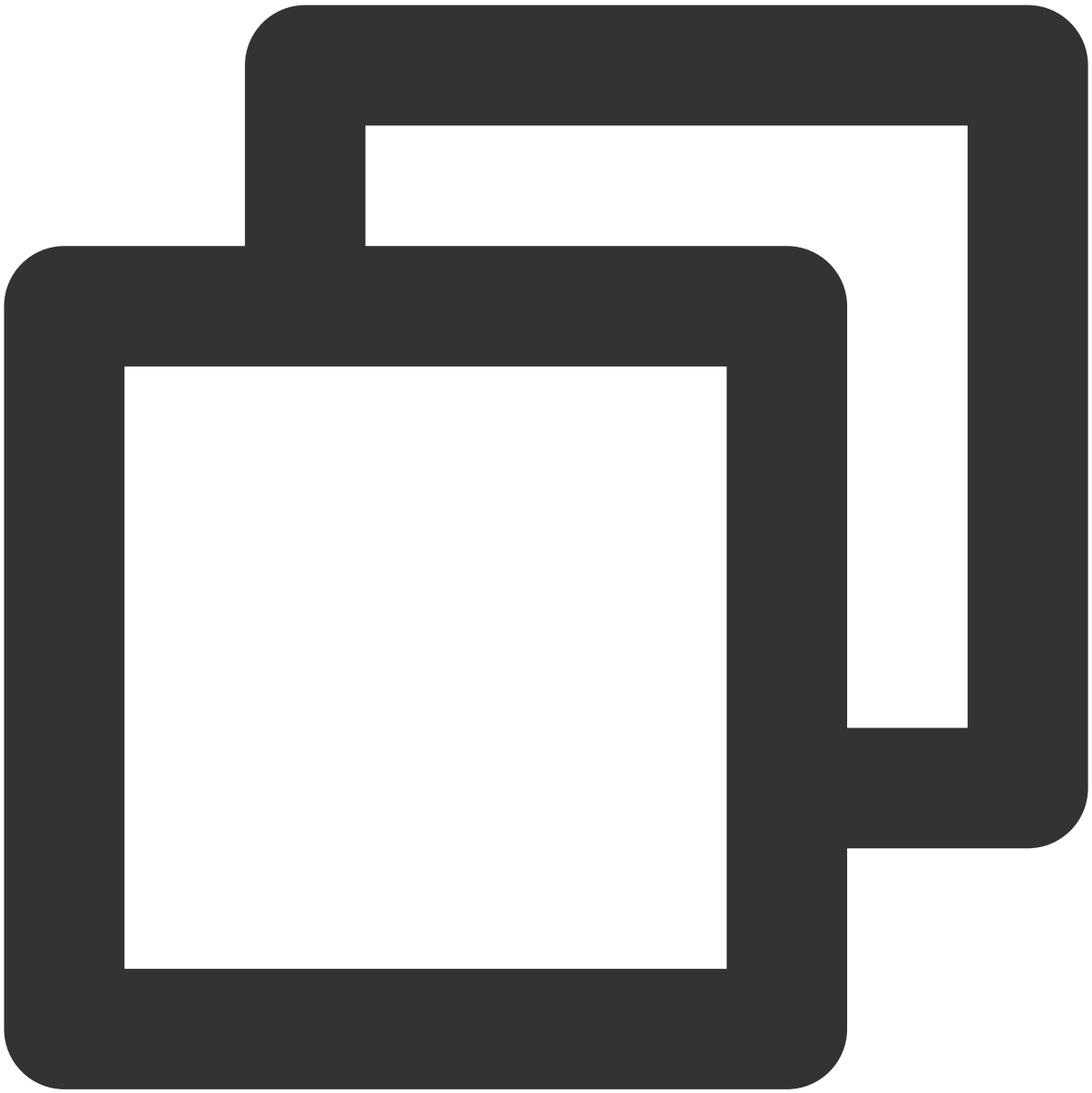
注意：

需要在**Build Setting-Other Linker Flags**添加 `-ObjC` 。

引用 IM SDK

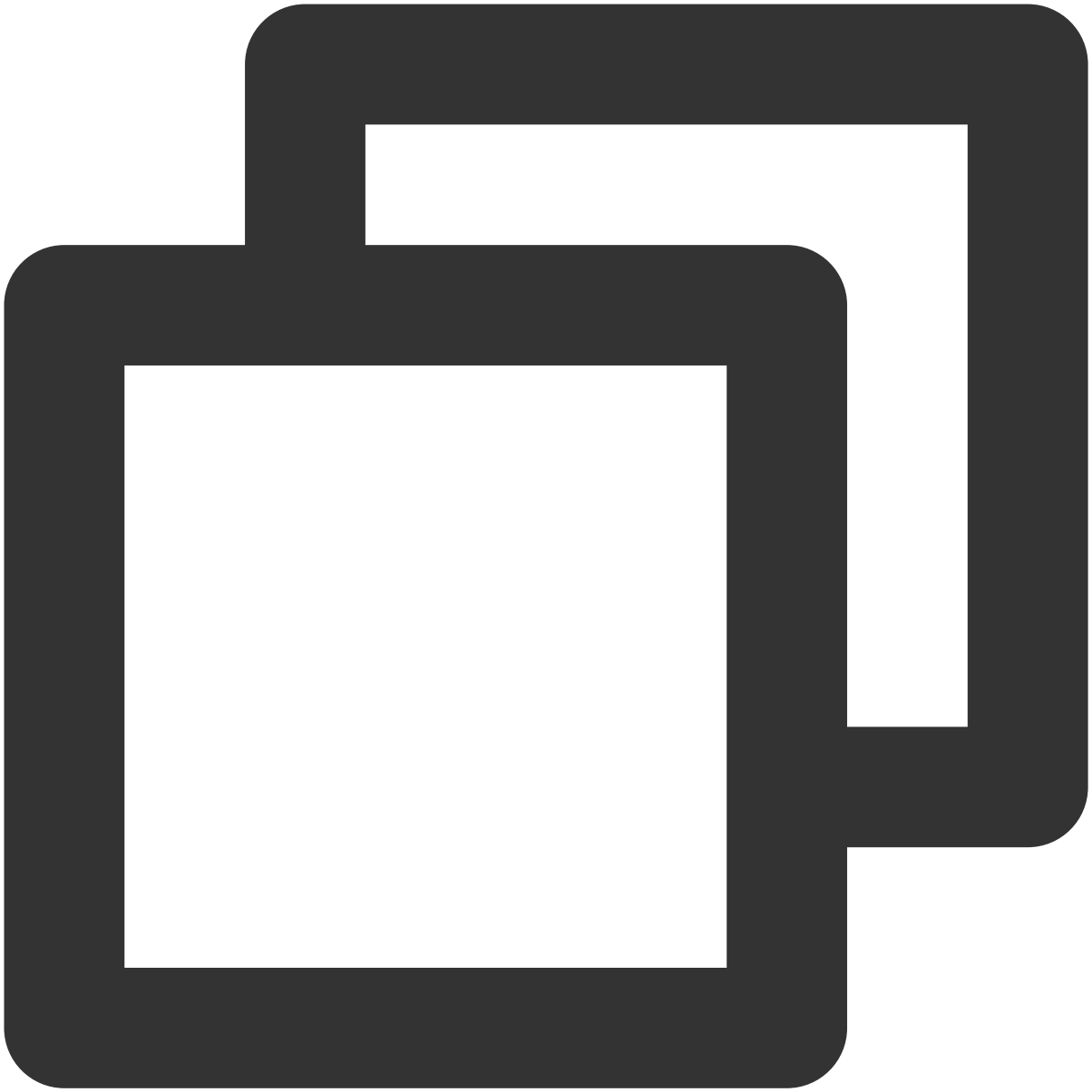
项目代码中使用 SDK 有两种方式：

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>
```

Unity

最近更新时间：2024-01-31 15:27:40

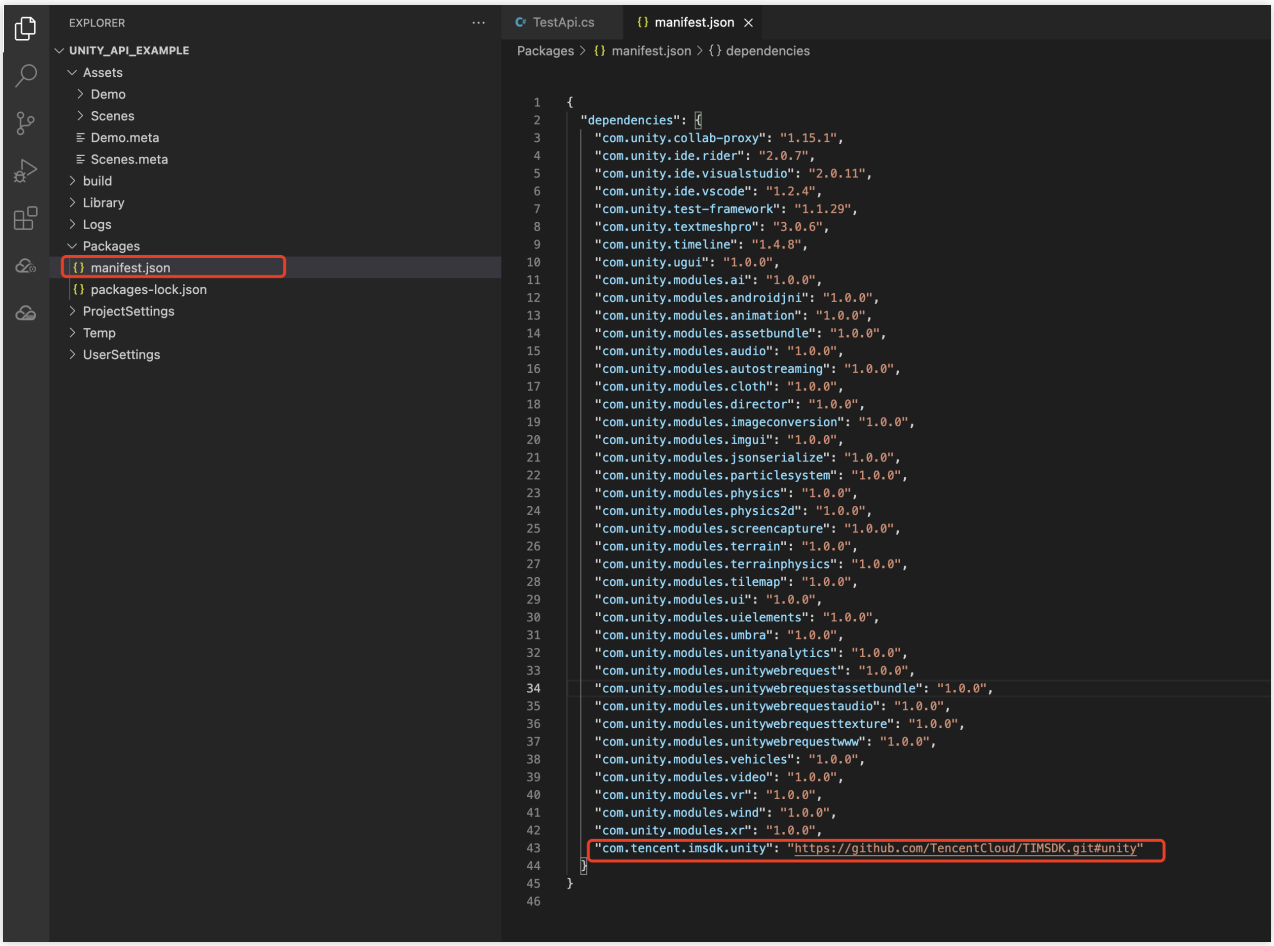
本文主要介绍如何快速地将腾讯云即时通信 IM SDK 集成到您的项目中，只要按照如下步骤进行配置，就可以完成 SDK 的集成工作。

环境要求

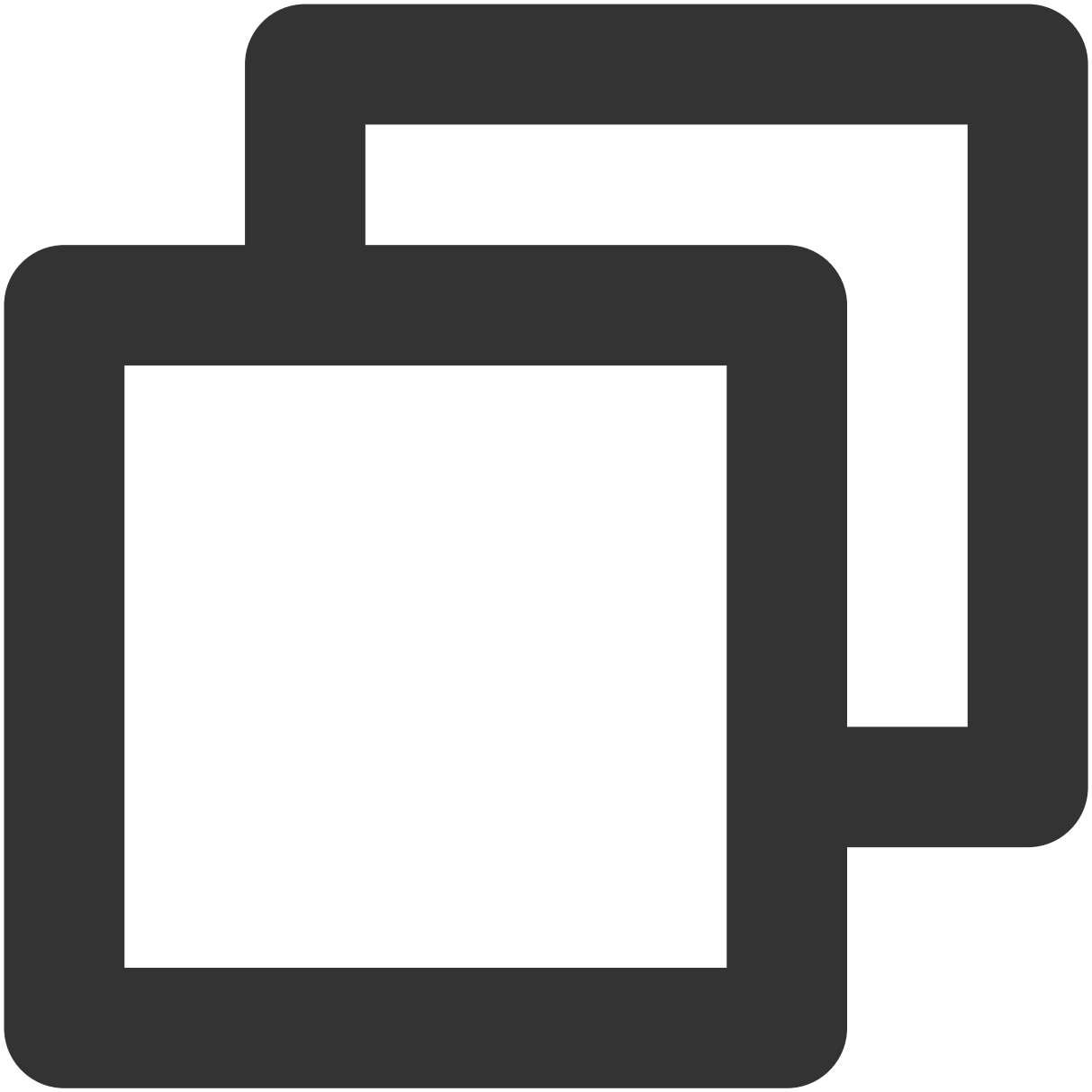
平台	版本
Unity	2019.4.15f1 及以上版本。
Android	Android Studio 3.5及以上版本，App 要求 Android 4.1及以上版本设备。
iOS	Xcode 11.0及以上版本，请确保您的项目已设置有效的开发者签名。

UPM 集成（推荐）

1. 修改 manifest.json 文件：

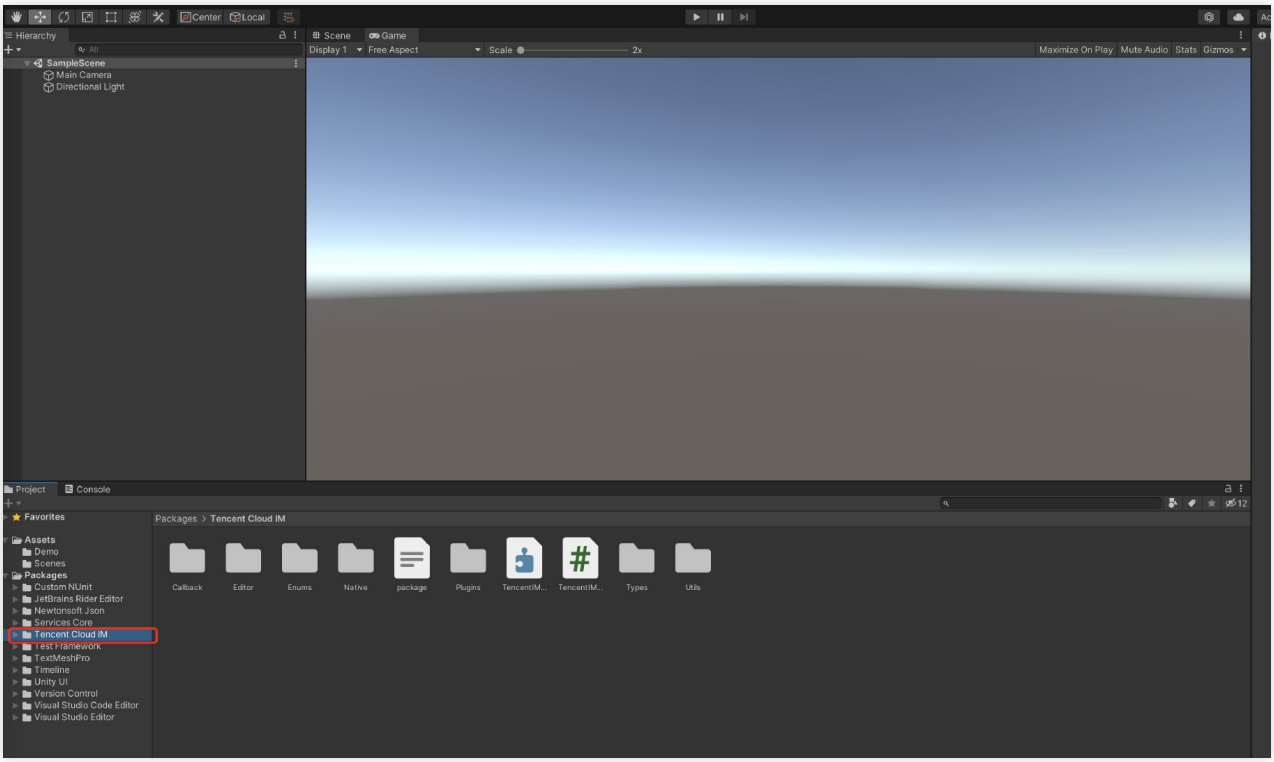

EXPLORER
UNITY_API_EXAMPLE
Assets
Demo
Scenes
Demo.meta
Scenes.meta
build
Library
Logs
Packages
manifest.json
packages-lock.json
ProjectSettings
Temp
UserSettings
TestApi.cs
manifest.json
Packages > manifest.json > { } dependencies
{
"dependencies": {
"com.unity.collab-proxy": "1.15.1",
"com.unity.ide.rider": "2.0.7",
"com.unity.ide.visualstudio": "2.0.11",
"com.unity.ide.vscode": "1.2.4",
"com.unity.test-framework": "1.1.29",
"com.unity.textmeshpro": "3.0.6",
"com.unity.timeline": "1.4.8",
"com.unity.ugui": "1.0.0",
"com.unity.modules.ai": "1.0.0",
"com.unity.modules.androidjni": "1.0.0",
"com.unity.modules.animation": "1.0.0",
"com.unity.modules.assetbundle": "1.0.0",
"com.unity.modules.audio": "1.0.0",
"com.unity.modules.autostreaming": "1.0.0",
"com.unity.modules.cloth": "1.0.0",
"com.unity.modules.director": "1.0.0",
"com.unity.modules.imageconversion": "1.0.0",
"com.unity.modules.imgui": "1.0.0",
"com.unity.modules.jsonserialize": "1.0.0",
"com.unity.modules.particlesystem": "1.0.0",
"com.unity.modules.physics": "1.0.0",
"com.unity.modules.physics2d": "1.0.0",
"com.unity.modules.screencapture": "1.0.0",
"com.unity.modules.terrain": "1.0.0",
"com.unity.modules.terrainphysics": "1.0.0",
"com.unity.modules.tilemap": "1.0.0",
"com.unity.modules.ui": "1.0.0",
"com.unity.modules.uielements": "1.0.0",
"com.unity.modules.umbra": "1.0.0",
"com.unity.modules.unityanalytics": "1.0.0",
"com.unity.modules.unitywebrequest": "1.0.0",
"com.unity.modules.unitywebrequestassetbundle": "1.0.0",
"com.unity.modules.unitywebrequestaudio": "1.0.0",
"com.unity.modules.unitywebrequesttexture": "1.0.0",
"com.unity.modules.unitywebrequestwww": "1.0.0",
"com.unity.modules.vehicles": "1.0.0",
"com.unity.modules.video": "1.0.0",
"com.unity.modules.vr": "1.0.0",
"com.unity.modules.wind": "1.0.0",
"com.unity.modules.xr": "1.0.0",
"com.tencent.imsdk.unity": "https://github.com/TencentCloud/TIMSDK.git#unity"
}
}

2. 修改如下：

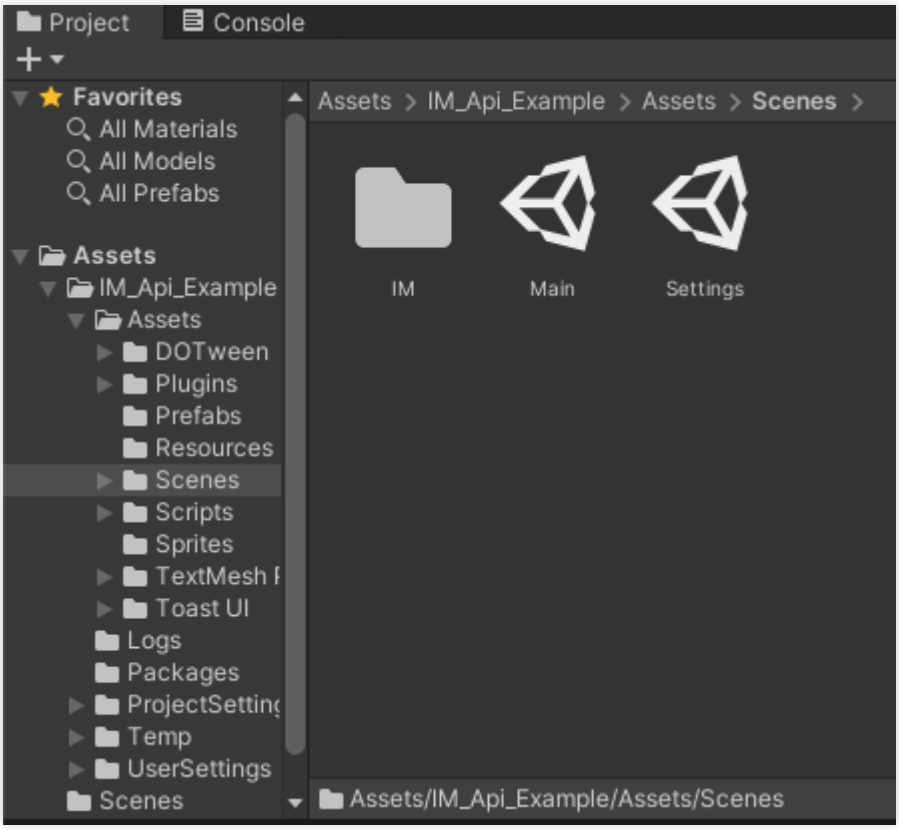


```
{
  "dependencies":{
    "com.tencent.imsdk.unity":"https://github.com/TencentCloud/chat-sdk-unity.git#u
  }
}
```

3. 在 Unity Editor 中打开项目，等候依赖加载完毕，确认Tencent Cloud IM 已经加载完成。



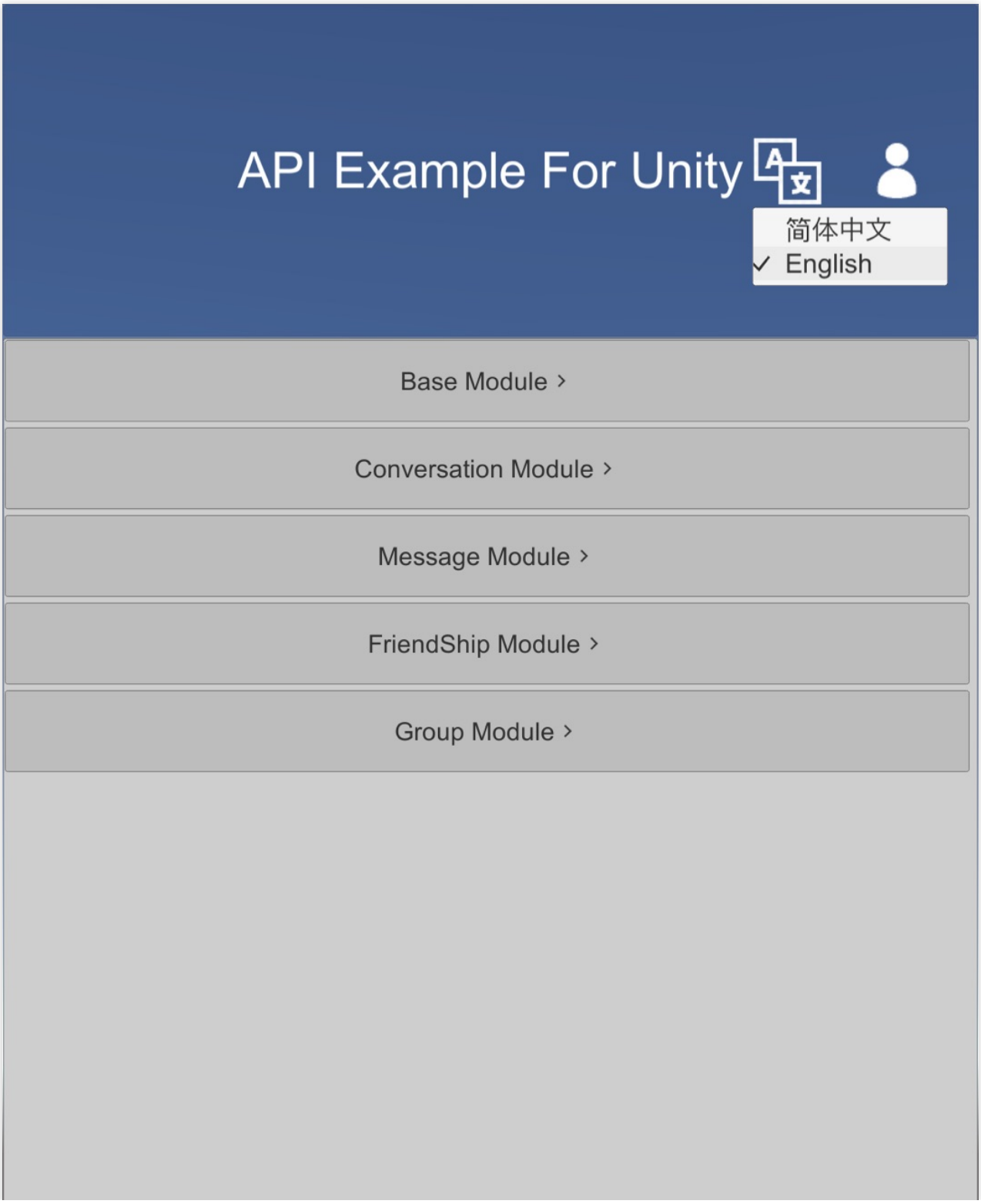
4. 该步骤为测试环节，您可下载 IM_Api_Example，解压后放入您的项目内。

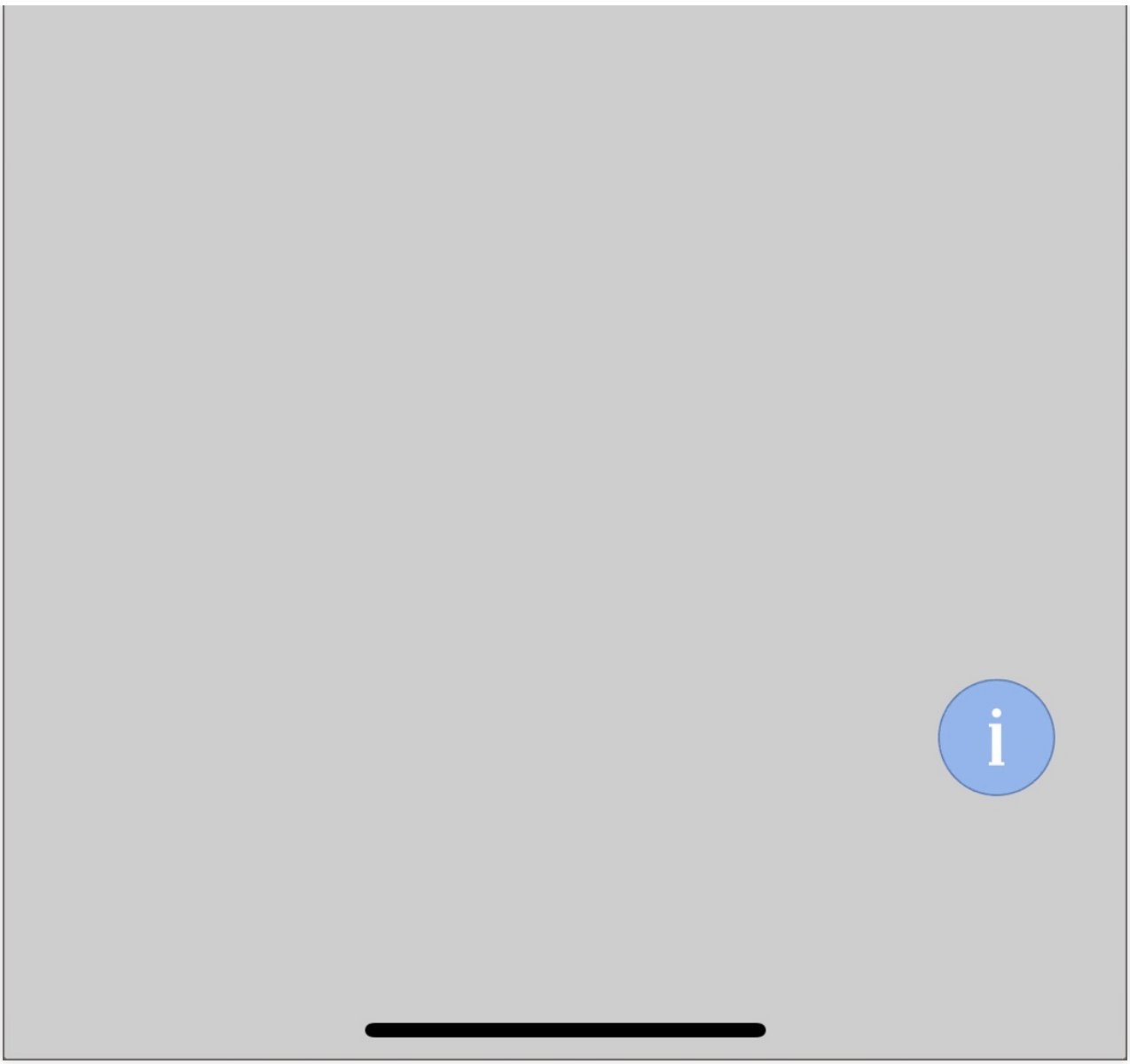


说明：

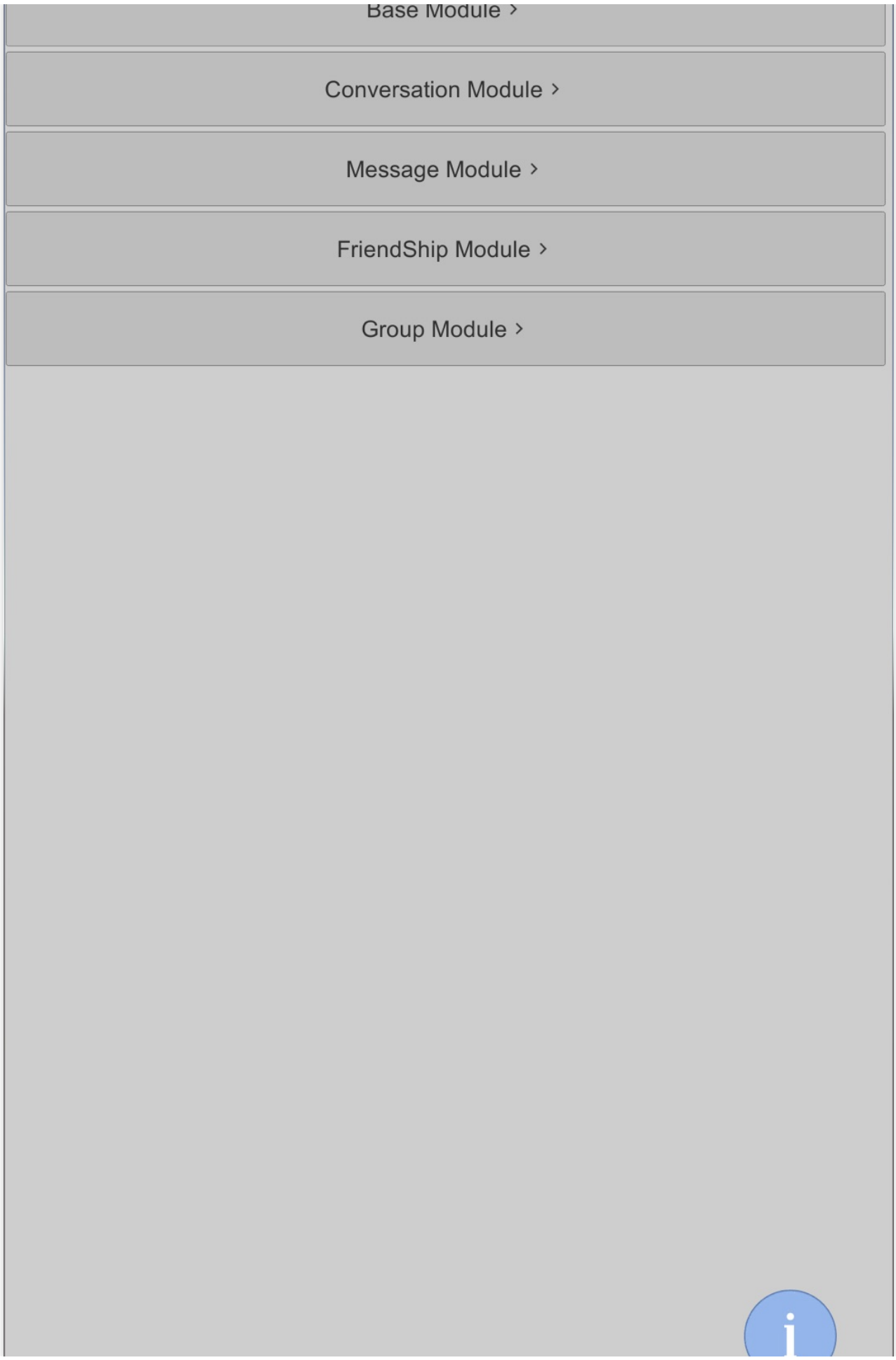
IM_Api_Example 是我们提供的用来测试 SDK 接口回调数据的 Demo，您也可以在项目开发早期通过调用我们提供的接口来对您的应用进行操作。

将 `IM_Api_Example/Assets` 文件夹下所有场景拖入 `Build Settings`，并保证 `Main` 场景的顺序在第一位。



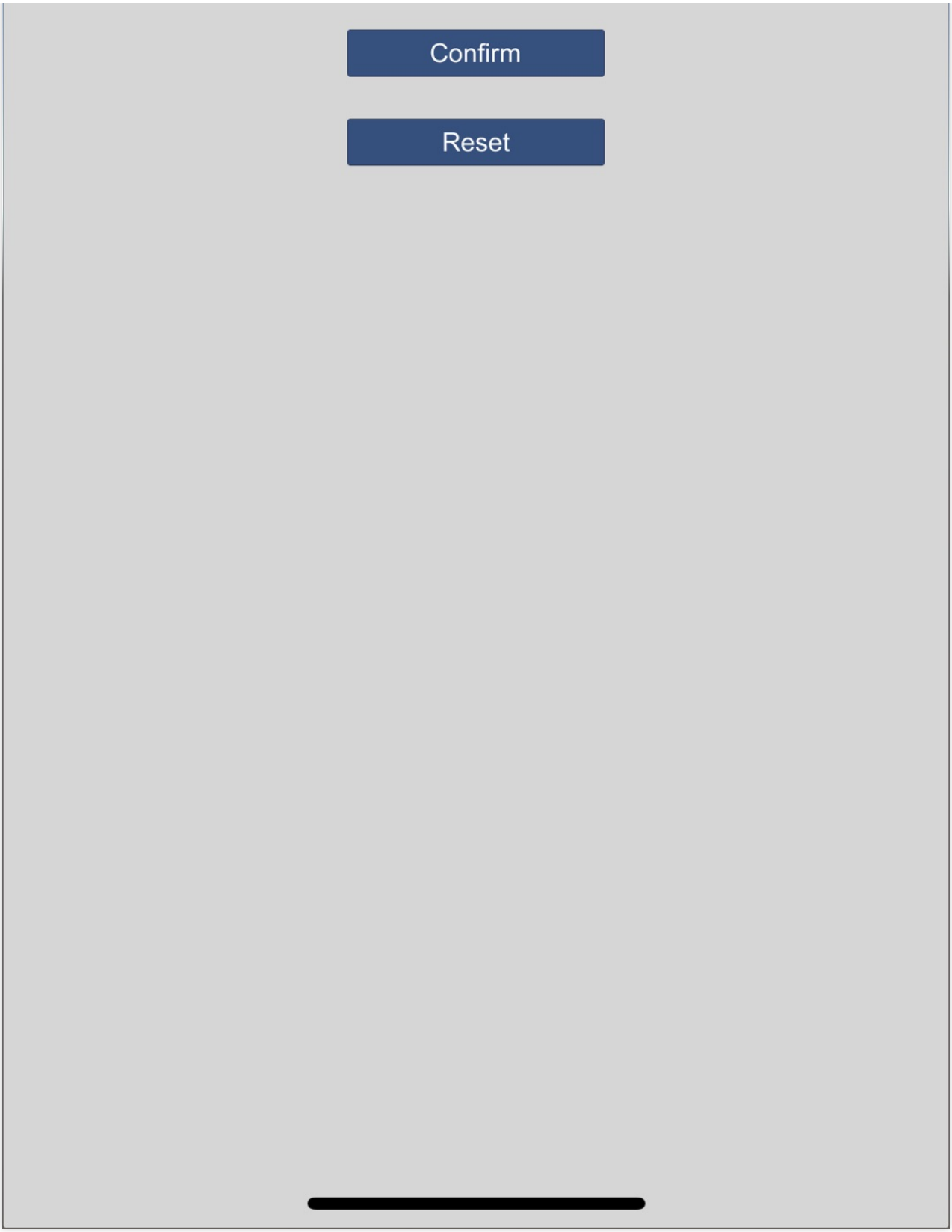


双击位于 `IM_Api_Example/Assets` 下的 Main Scene 来启动 Demo，您可以在这里选择语言。


Base Module
Conversation Module
Message Module
FriendShip Module
Group Module

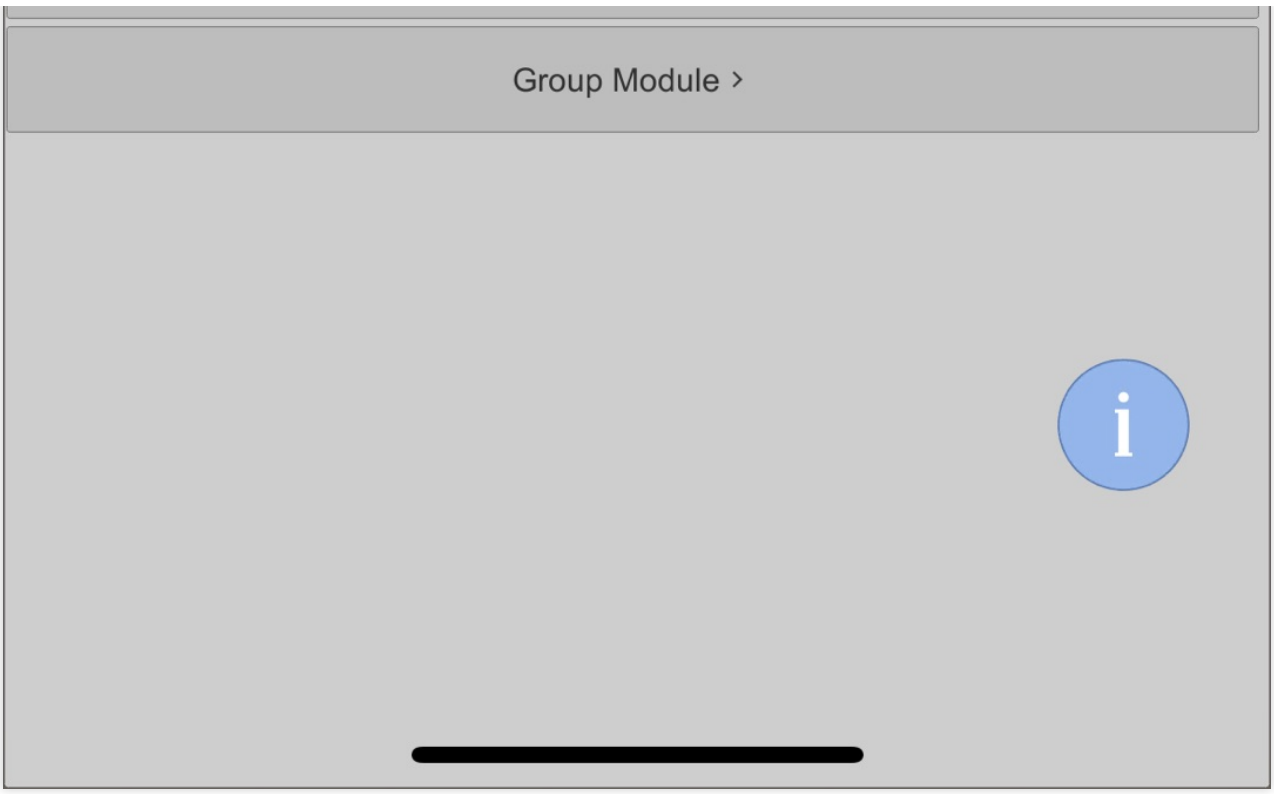
单击 Header 右侧的

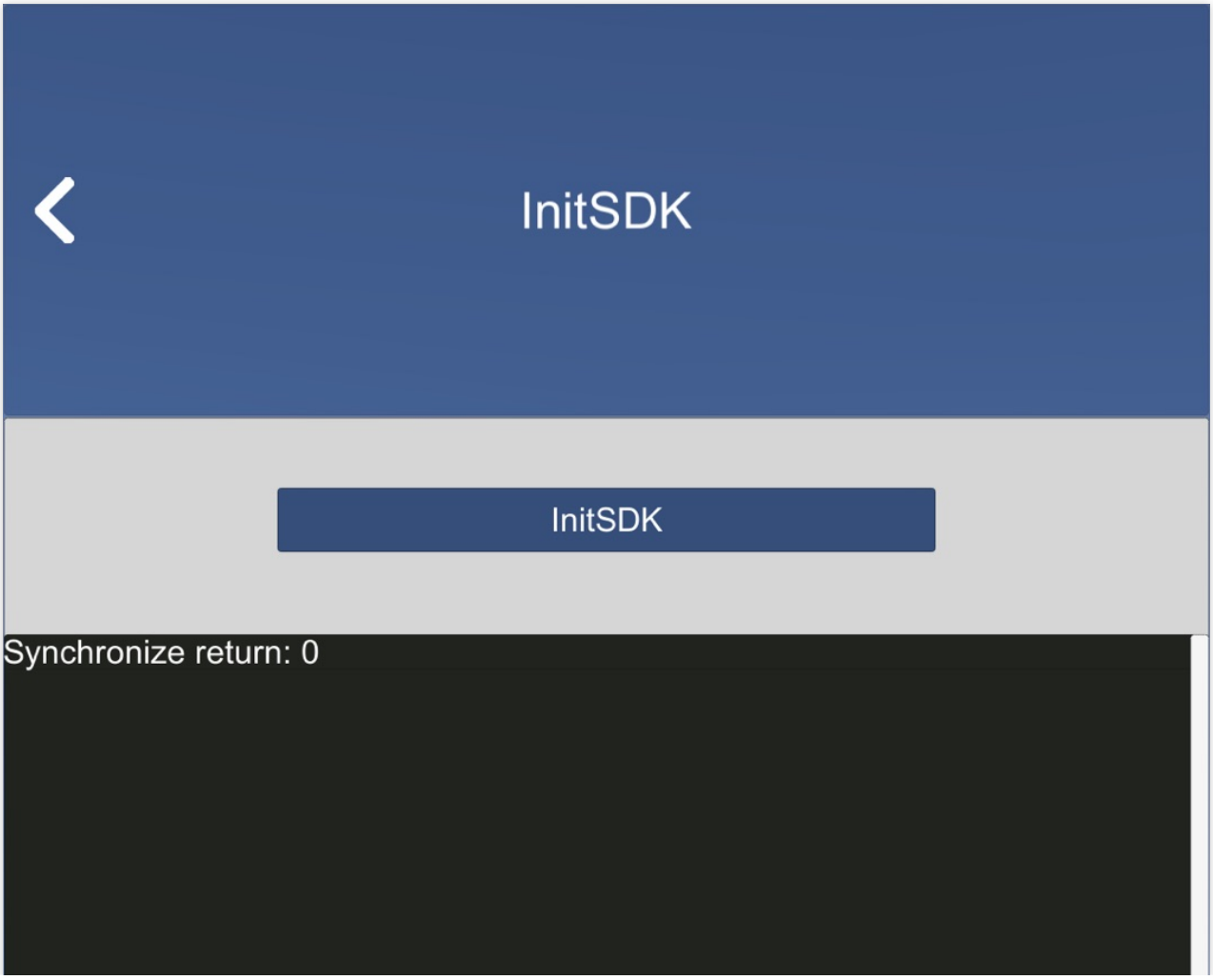
并填写相应信息。

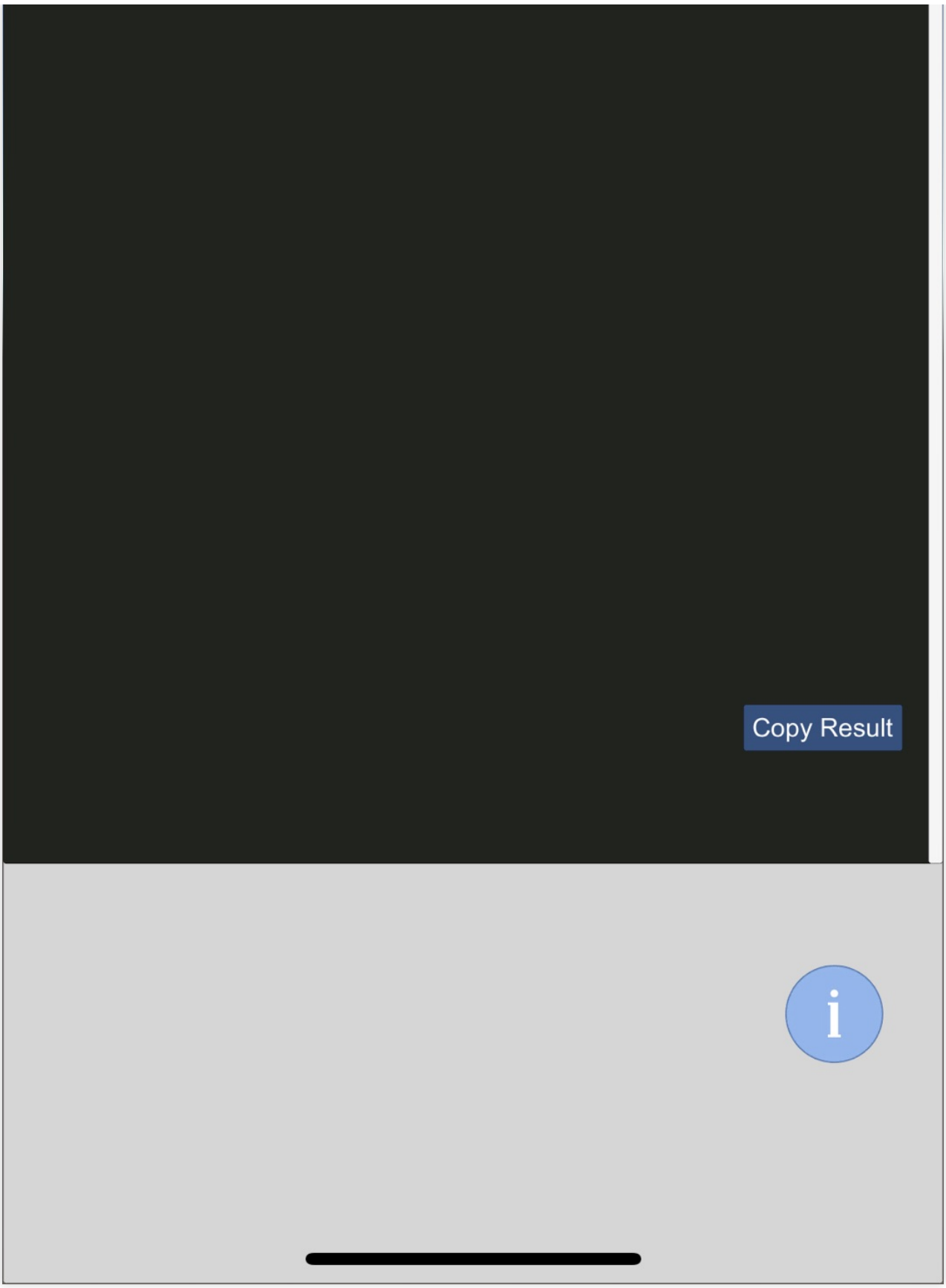


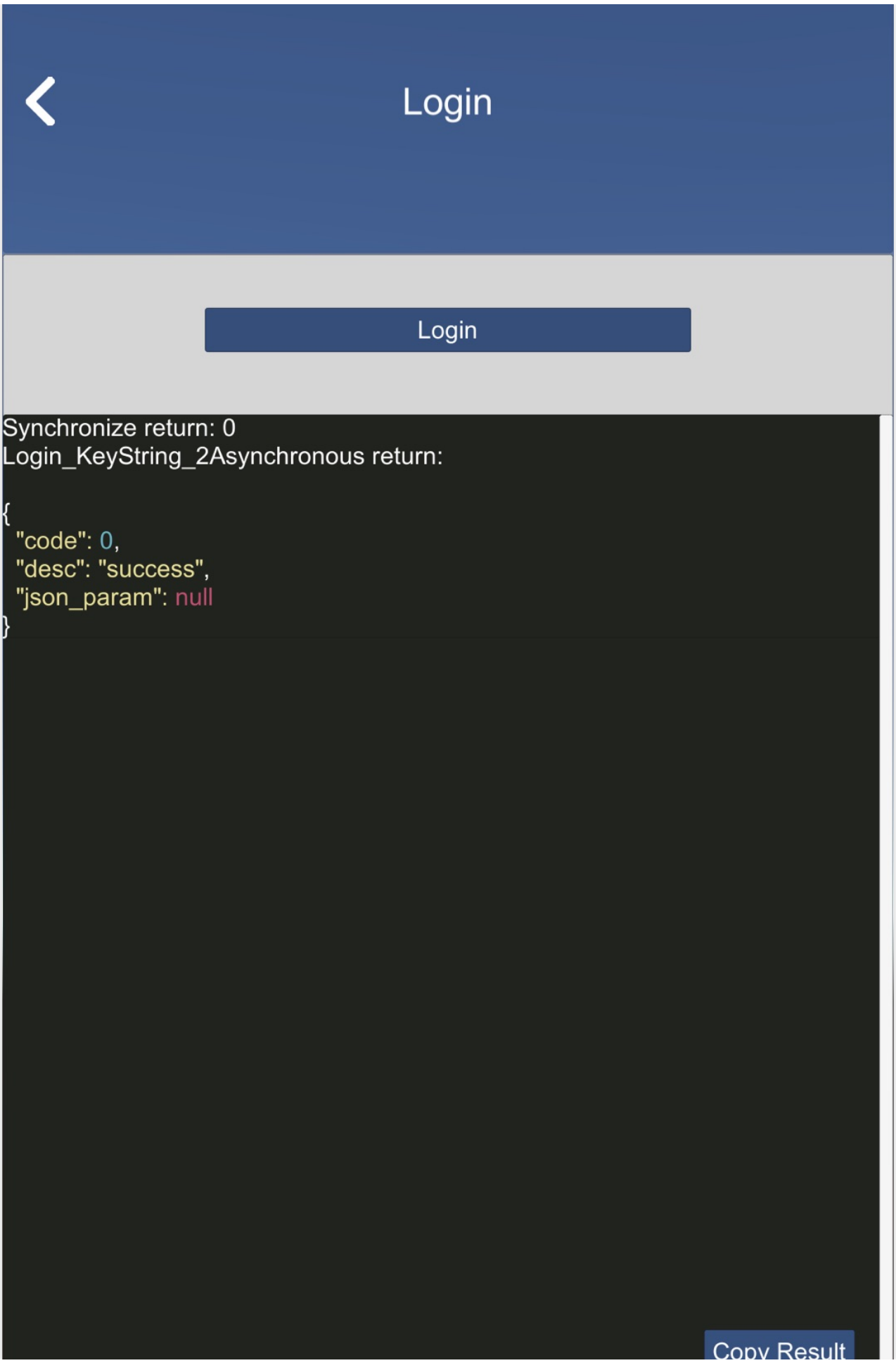
分别单击基础模块内的 InitSDK & Login 完成初始化和登录，接下来您可以自由调用 Api Example 里提供的接口。

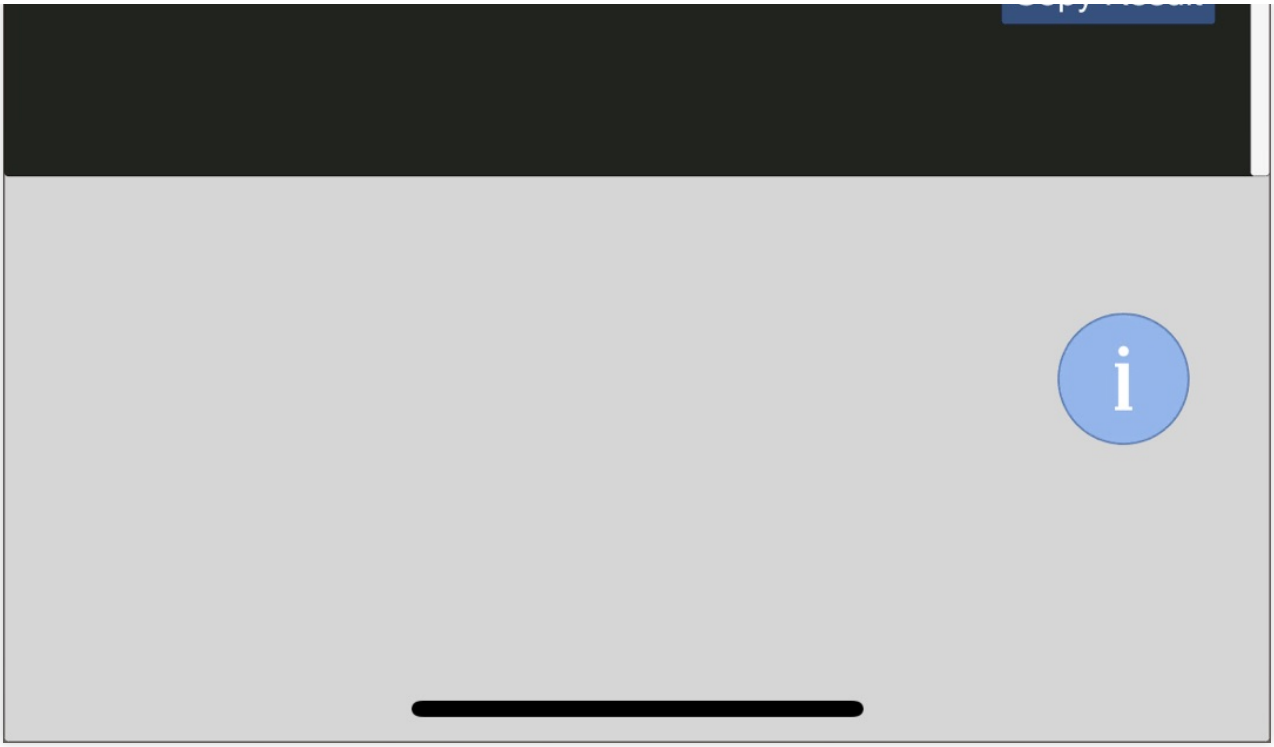

API Example For Unity
Base Module
InitSDK
AddEventListener
GetServerTime
Login
Logout
GetLoginUser
ProfileGetUserProfileList
GroupCreate
GroupJoin
GroupDelete
Conversation Module
Message Module
FriendShip Module


Group Module >


InitSDK
InitSDK
Synchronize return: 0


Copy Result


Login
Login
Synchronize return: 0
Login_KeyString_2Asynchronous return:
{
"code": 0,
"desc": "success",
"json_param": null
}
Copy Result



Unreal Engine

最近更新时间：2024-01-31 15:28:17

本文主要介绍如何快速地将腾讯云 IM SDK（Unreal Engine）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。

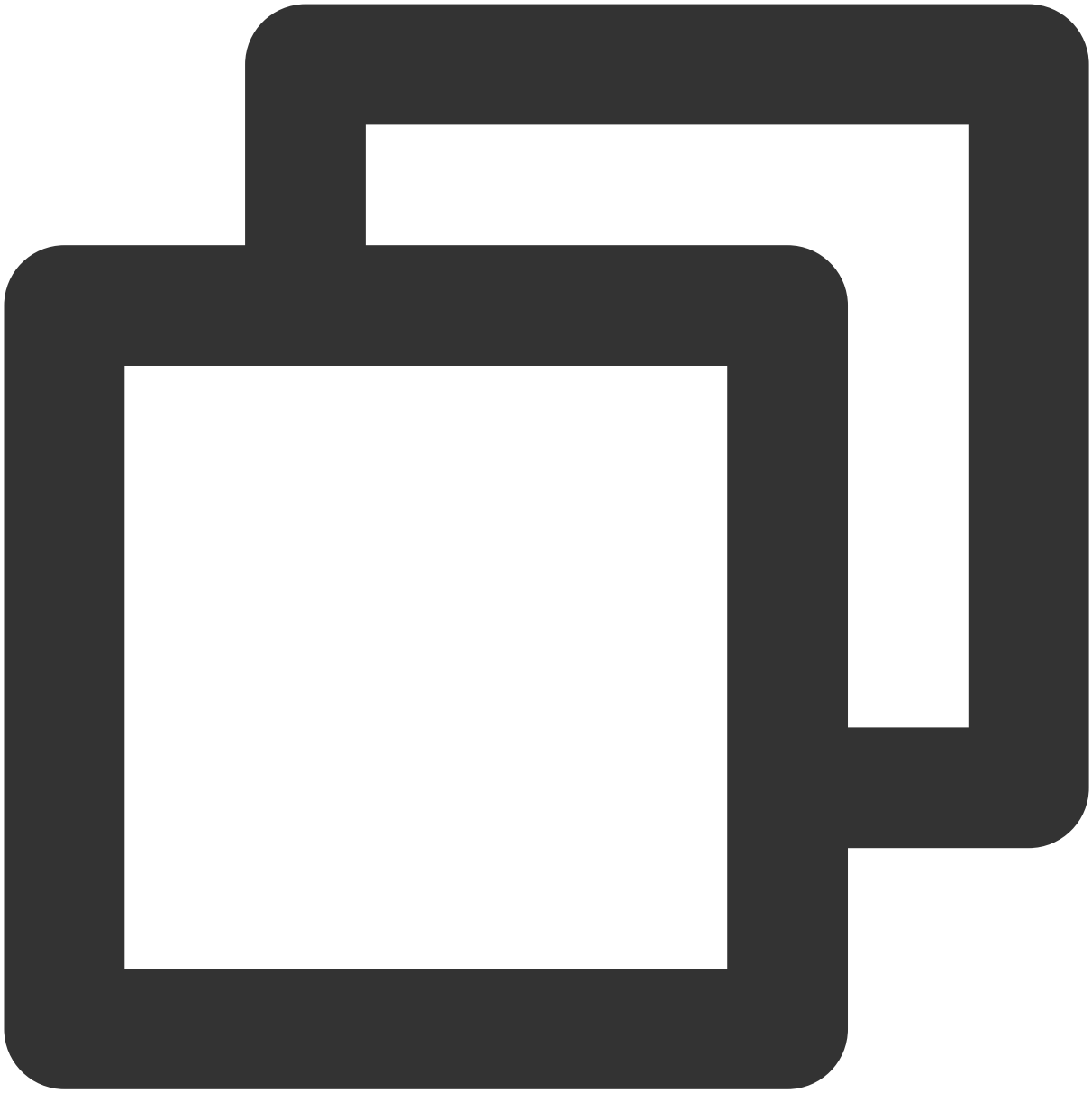
环境要求

建议 Unreal Engine 4.27.1 及以上版本。

开发端	环境
Android	Android Studio 4.0 及以上版本。 Visual Studio 2017 15.6 及以上版本。 只支持真机调试。
iOS & macOS	Xcode 11.0 及以上版本。 OSX 系统版本要求 10.11 及以上版本 。 请确保您的项目已设置有效的开发者签名。
Windows	操作系统：Windows 7 SP1 及以上版本（基于 x86-64 的 64 位操作系统）。 磁盘空间：除安装 IDE 和一些工具之外还应有至少 1.64 GB 的空间。 安装 Visual Studio 2019 。

集成 SDK

1. 下载 SDK 及配套的 SDK 源码。
2. 把项目中的 IM SDK 文件夹拷贝到您项目中的 **Source/[project_name]** 目录下，其中 **[project_name]** 表示你项目的名称。
3. 编辑你项目中的 **[project_name].Build.cs**文件。添加下面函数：



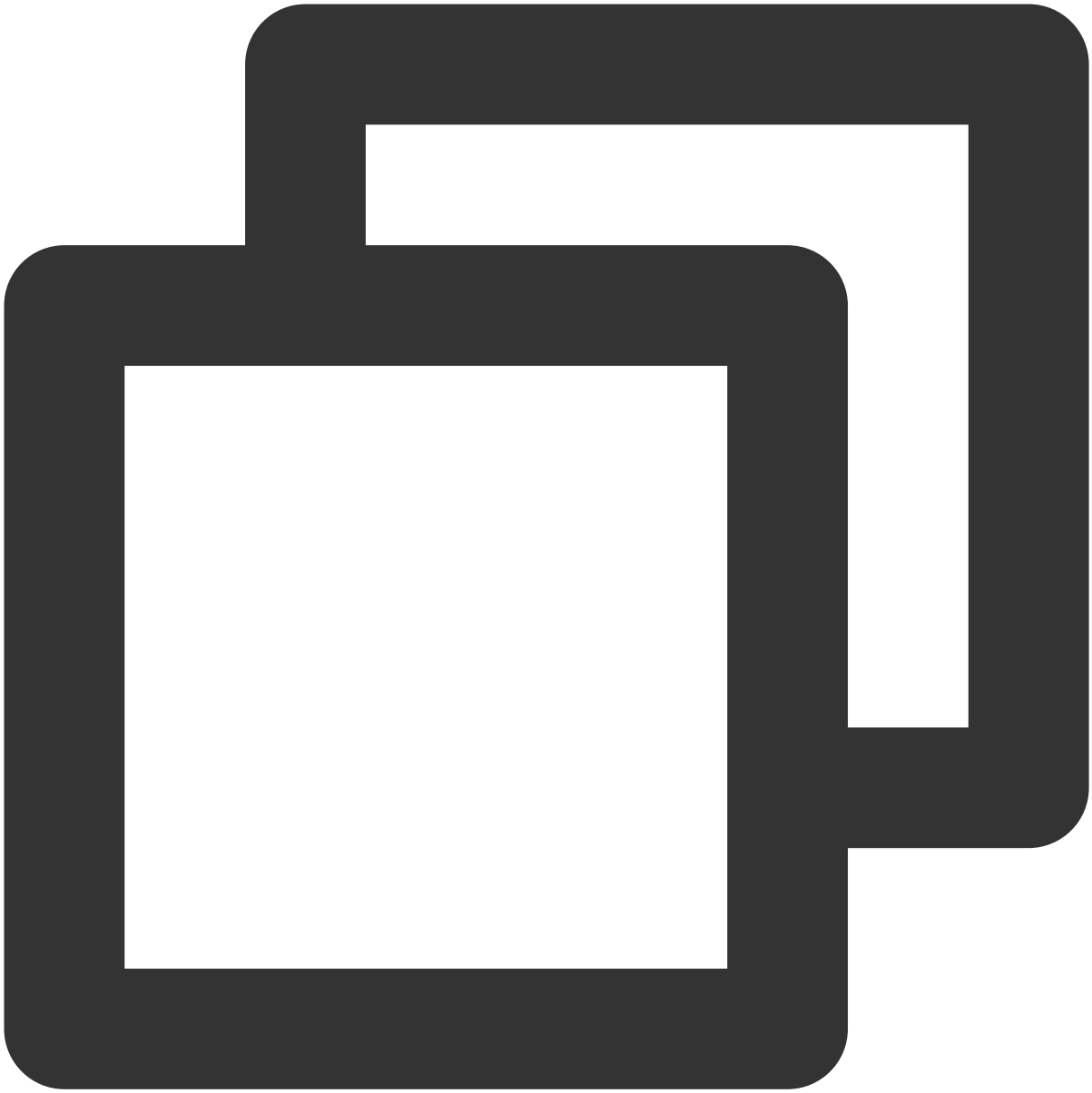
```
// 加载各个平台IM底层库
private void loadTIMSDK(ReadOnlyTargetRules Target) {
 string _TIMSDKPath = Path.GetFullPath(Path.Combine(ModuleDirectory, "TIMSDK"));
 bEnableUndefinedIdentifierWarnings = false;
 PublicIncludePaths.Add(Path.Combine(_TIMSDKPath, "include"));
 if (Target.Platform == UnrealTargetPlatform.Android) {
     PrivateDependencyModuleNames.AddRange(new string[] { "Launch" });
     AdditionalPropertiesForReceipt.Add(new ReceiptProperty("AndroidPlugin", Path.C

     string Architecture = "armeabi-v7a";
     // string Architecture = "arm64-v8a";
```

```
        // string Architecture = "armeabi";
        PublicAdditionalLibraries.Add(Path.Combine(ModuleDirectory,"TIMSDK", "Android"
    }
    else if (Target.Platform == UnrealTargetPlatform.IOS) {
        PublicAdditionalLibraries.AddRange(
            new string[] {
                "z","c++",
                "z.1.1.3",
                "sqlite3",
                "xml2"
            }
        );
    PublicFrameworks.AddRange(new string[]{
            "Security",
            "AdSupport",
            "CoreTelephony",
            "CoreGraphics",
            "UIKit"
        });
        PublicAdditionalFrameworks.Add(new UEBuildFramework("ImSDK_CPP",_TIMSDKPath+"/
    }
    else if(Target.Platform == UnrealTargetPlatform.Mac) {
        PublicAdditionalLibraries.AddRange(new string[] {
            "resolv",
            "z",
            "c++",
            "bz2",
            "sqlite3",
        });
    PublicFrameworks.AddRange(
            new string[] {
                "AppKit",
                "Security",
                "CFNetwork",
                "SystemConfiguration",
            });
        PublicFrameworks.Add(Path.Combine(_TIMSDKPath, "Mac", "Release","ImSDKForMac_C
    }
    else if (Target.Platform == UnrealTargetPlatform.Win64) {
        PublicAdditionalLibraries.Add(Path.Combine(_TIMSDKPath, "win64", "Release","Im
        PublicDelayLoadDLLs.Add(Path.Combine(_TIMSDKPath, "win64", "Release", "ImSDK.d
        RuntimeDependencies.Add("$(BinaryOutputDir)/ImSDK.dll", Path.Combine(_TIMSDKPa
    }
}
```

4. 在 **[project_name].Build.cs** 文件调用该函数：

```
private void loadTIMSDK(ReadOnlyTargetRules Target) {
    string _TIMSDKPath = Path.GetFullPath(Path.Combine(ModuleDirectory, "TIMSDK"));
    bEnableUndefinedIdentifierWarnings = false;
    PublicIncludePaths.Add(Path.Combine(_TIMSDKPath, "include"));
    if (Target.Platform == UnrealTargetPlatform.Android) {
        PrivateDependencyModuleNames.AddRange(new string[] { "Launch" });
        AdditionalPropertiesForReceipt.Add(new ReceiptProperty("AndroidPlugin", Path.Combine(ModuleDirectory, "TIMSDK", "Android", "APL_armv7.xml")));

        string Architecture = "armeabi-v7a";
        // string Architecture = "arm64-v8a";
        // string Architecture = "armeabi";
        PublicAdditionalLibraries.Add(Path.Combine(ModuleDirectory,"TIMSDK", "Android", Architecture, "libImSDK.so"));
    }
    else if (Target.Platform == UnrealTargetPlatform.IOS) {
        PublicAdditionalLibraries.AddRange(
            new string[] {
                "z","c++",
                "z.1.1.3",
                "sqlite3",
                "xml2"
            }
        );
        PublicFrameworks.AddRange(new string[]{
            "Security",
            "AdSupport",
            "CoreTelephony",
            "CoreGraphics",
            "UIKit"
        });
        PublicAdditionalFrameworks.Add(new UEBuildFramework("ImSDK_CPP",_TIMSDKPath+"/ios/ImSDK_CPP.framework.zip", ""));
    }
```

5. 到目前为止你已经集成了 IM SDK。可在你的 cpp 文件中使用 IM 的能力了。 `#include "V2TIMManager.h"`



```
// 获取sdk单例对象
V2TIMManager* timInstance = V2TIMManager::GetInstance();
// 获取sdk版本号
V2TIMString timString = timInstance->GetVersion();
// 初始化sdk
V2TIMSDKConfig timConfig;
timConfig.initPath = static_cast<V2TIMString>("D:\\\\");
timConfig.logPath = static_cast<V2TIMString>("D:\\\\");
bool isInit = timInstance->InitSDK(SDKAppID, timConfig);
```

版权所有：腾讯云计算（北京）有限责任公司

打包

macOS 端

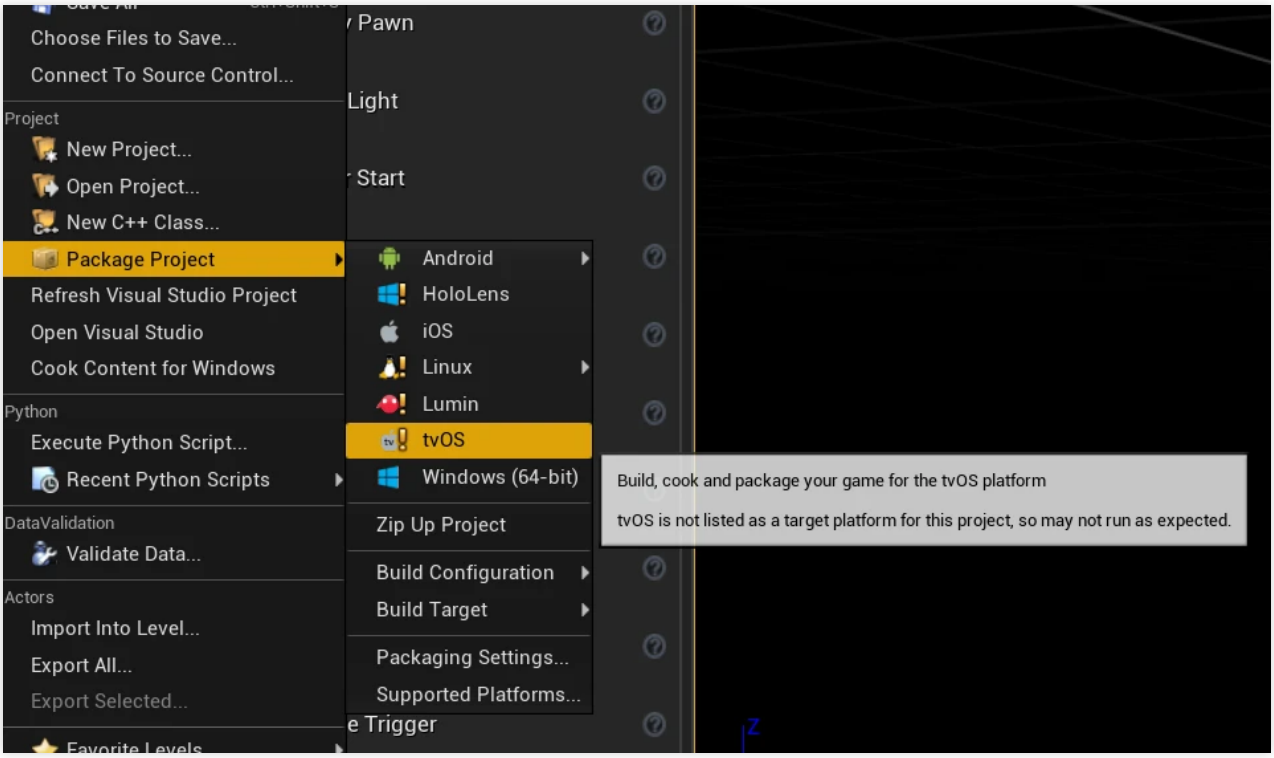
Windows 端

iOS 端

Android 端

File -> **Package Project** -> **Mac**。

File->**Package Project**->**Windows**->**Windows(64-bit)**。



打包项目。**File** -> **Package Project** -> **iOS**。

开发调试：详见 Android 快速入门。

打包项目：详见 打包 Android 项目。

IM Unreal Engine API 文档

更多接口介绍，请参见 API 概览。

版权所有：腾讯云计算（北京）有限责任公司

React Native

最近更新时间：2024-01-31 15:28:52

本文主要介绍如何快速将腾讯云即时通信 IM SDK 集成到您的 React Native 项目中。

环境要求

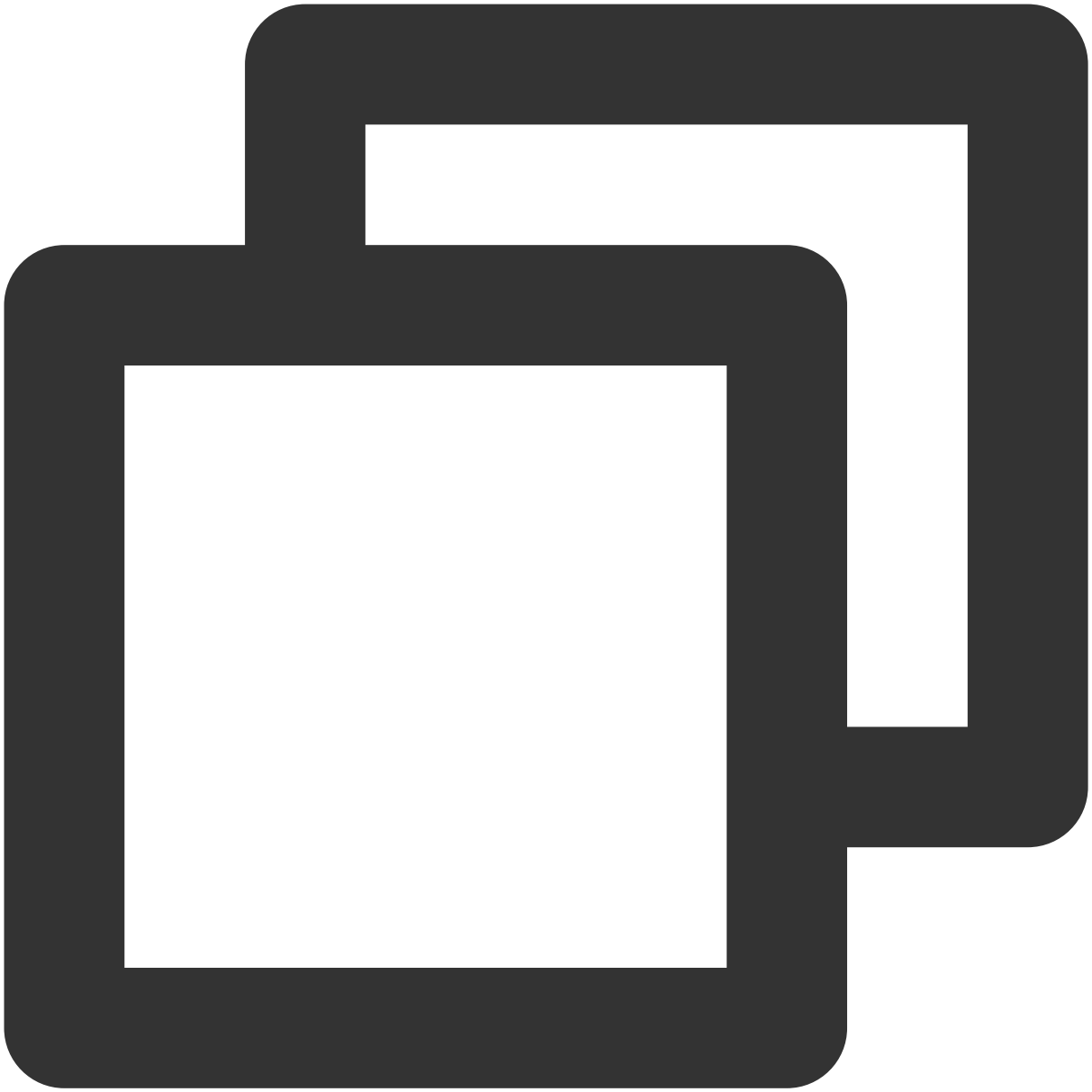
平台	版本
React Native	2.2.0 及以上版本。
Android	Android Studio 3.5 及以上版本，App 要求 Android 4.1 及以上版本设备。
iOS	Xcode 11.0 及以上版本，真机调试请确保您的项目已设置有效的开发者签名。

集成 IM SDK

您可以通过 npm 的方式直接集成腾讯云 IM SDK（React Native）。

npm 集成

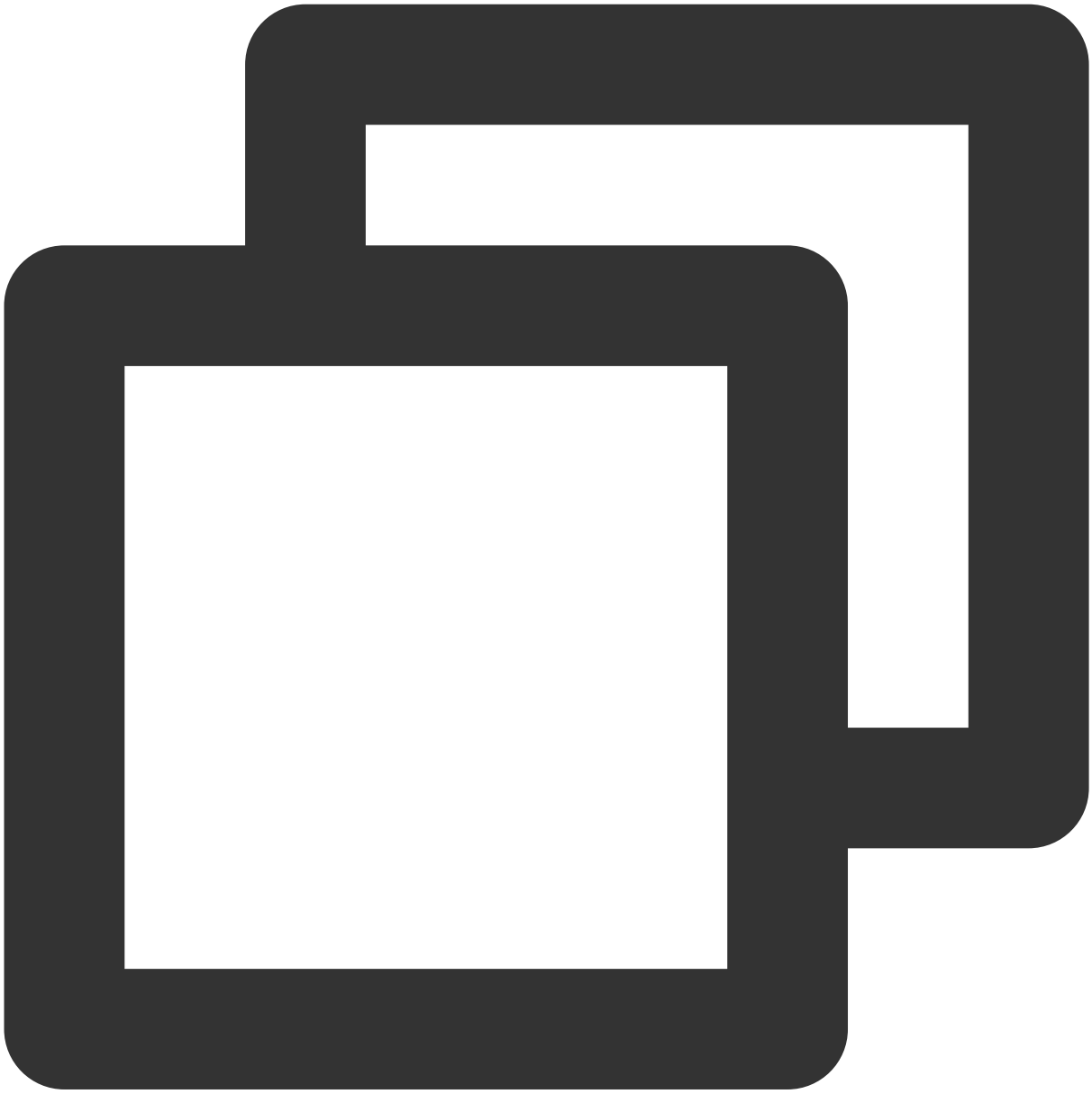
在终端窗口中输入如下命令：



```
# npm
npm install react-native-tim-js

# yarn
yarn add react-native-tim-js


# RN >= 0.60
cd ios && pod install

# RN < 0.60
```

```
react-native link react-native-tim-js
```

引入并初始化 SDK



```
import { TencentImSDKPlugin } from "react-native-tim-js";
```

初始化

Android&iOS&Windows&Mac

最近更新时间：2024-07-05 15:24:08

功能描述

在使用 SDK 的各项功能前，**必须**先进行初始化。大多数场景下，在应用生命周期内，您只需要进行一次 SDK 初始化。

初始化

初始化 SDK 需要操作以下步骤：

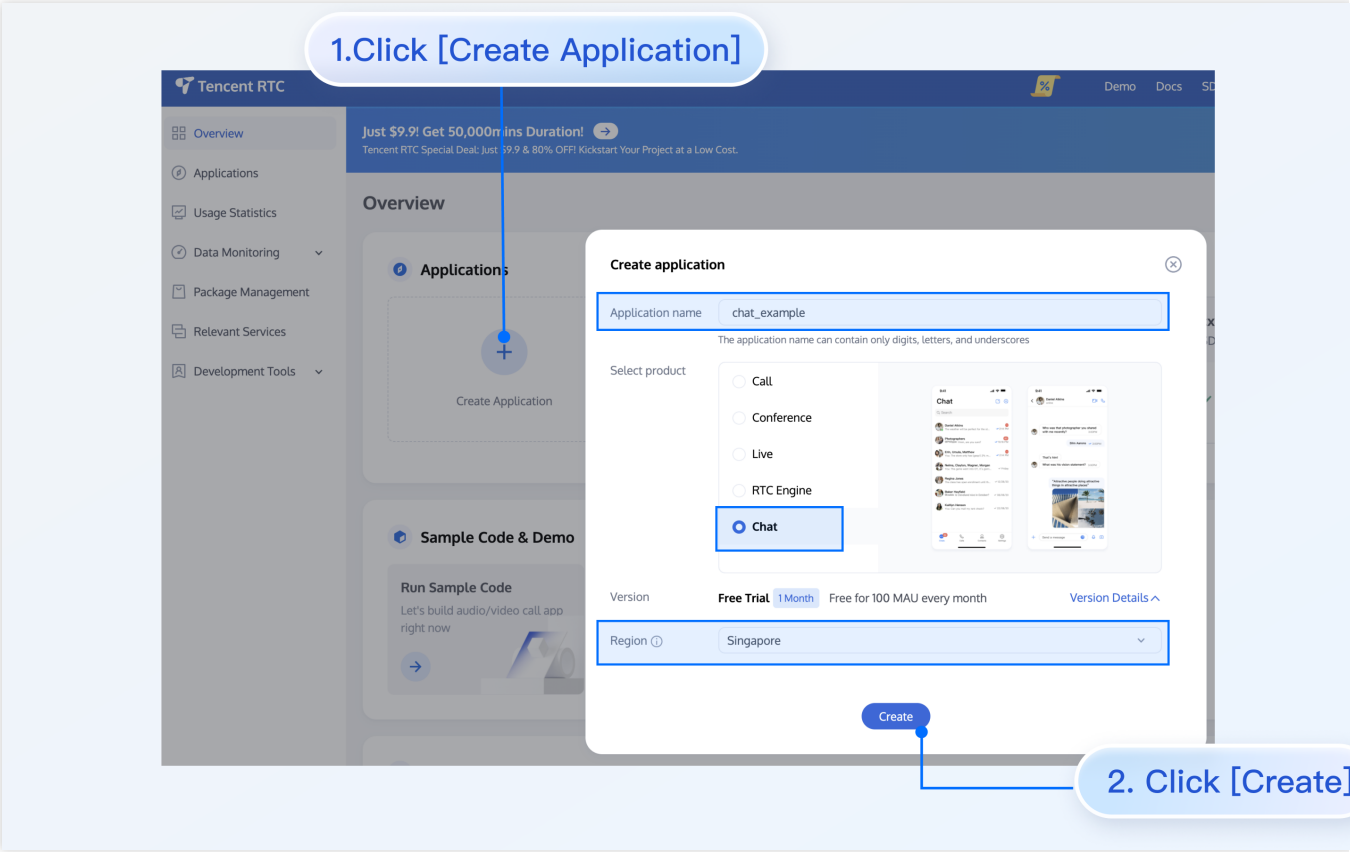
1. 准备 SDKAppID。
2. 配置 `V2TIMSDKConfig` 对象。
3. 添加 SDK 事件监听器。
4. 调用 `initSDK` 初始化 SDK。

下文我们将依次为您详细讲解具体的步骤内容。

准备 SDKAppID

SDKAppID 是用于区分客户账号的唯一标识。我们建议每一个独立的 App 都申请一个新的 SDKAppID。不同 SDKAppID 之间的消息是天然隔离的，不能互通。您必须拥有正确的 SDKAppID，才能进行初始化。

您可以在控制台查看所有的 SDKAppID，单击 `Create Applicaion` 按钮，创建新的 SDKAppID。



配置 V2TIMSDKConfig

初始化 SDK 前，您需要初始化一个 `V2TIMSDKConfig` (Android / iOS & Mac / Windows) 对象。该对象用于对 SDK 进行初始化配置，例如设置日志级别、设置日志监听回调。

设置日志级别

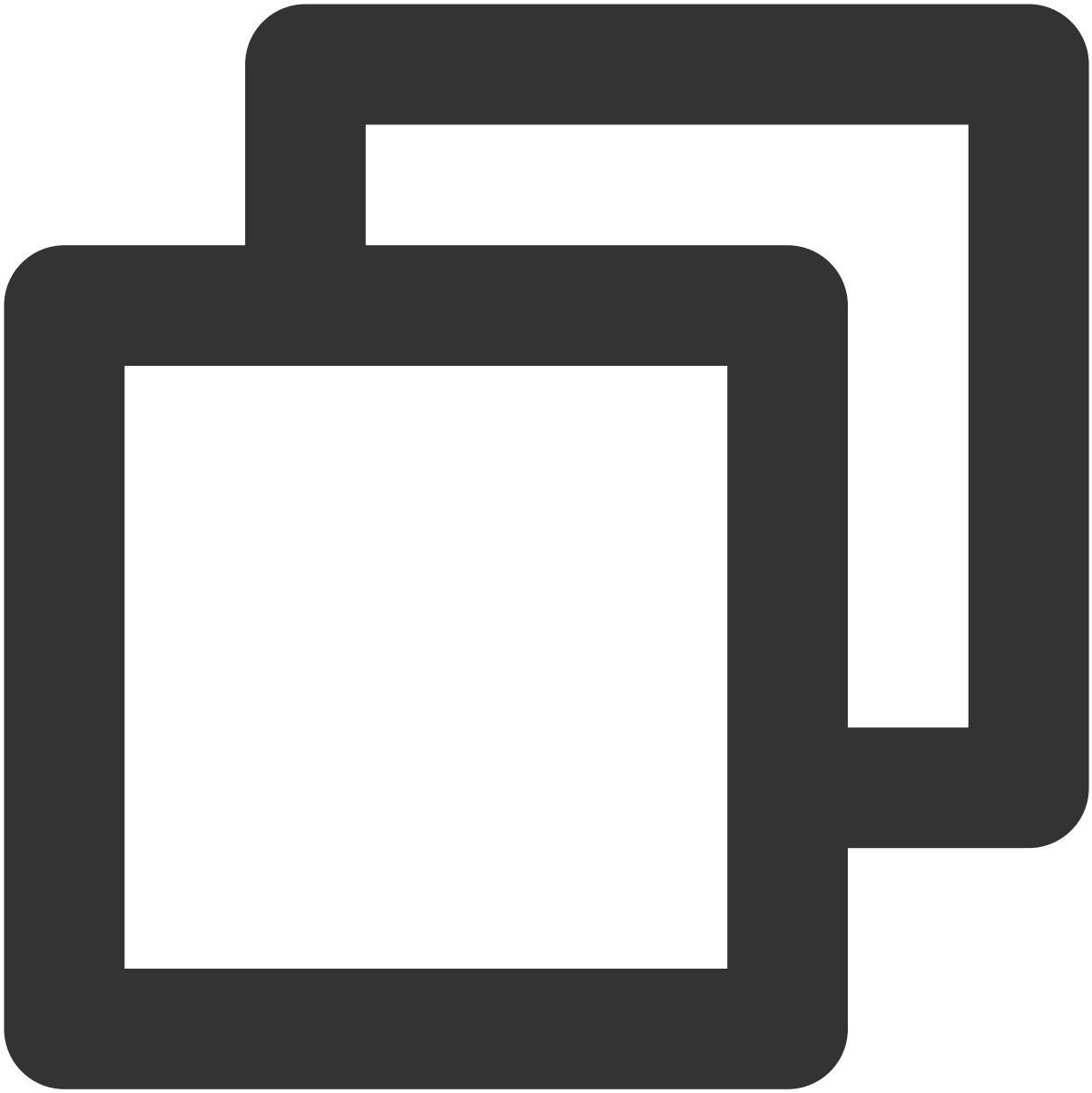
SDK 支持多种日志级别，如下表所示：

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 （默认的日志级别）
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

SDK 日志存储规则如下：
SDK 本地日志默认保存 7 天；SDK 在初始化时，会自动清理 7 天之前的日志。

在 Android 平台上，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。
iOS 平台 SDK 的日志默认存储于 `/Library/Caches/com_tencent_imsdk_log` 目录下。
Windows 平台 SDK 的日志默认存储于程序文件的运行目录下，比如：当程序运行在 `C:\\App\\` 目录下时，SDK 会将日志存储在 `C:\\App\\com_tencent_imsdk_log\\` 目录下。
从 4.7.1 版本开始，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。

版权所有：腾讯云计算（北京）有限责任公司



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

设置日志监听器

如果您需要实时监听 SDK 的日志，可以调用 `setLogListener` (Android / iOS & Mac / Windows) 设置日志监听器。

设置成功后，SDK 会实时将日志信息通过此回调抛出。

注意：

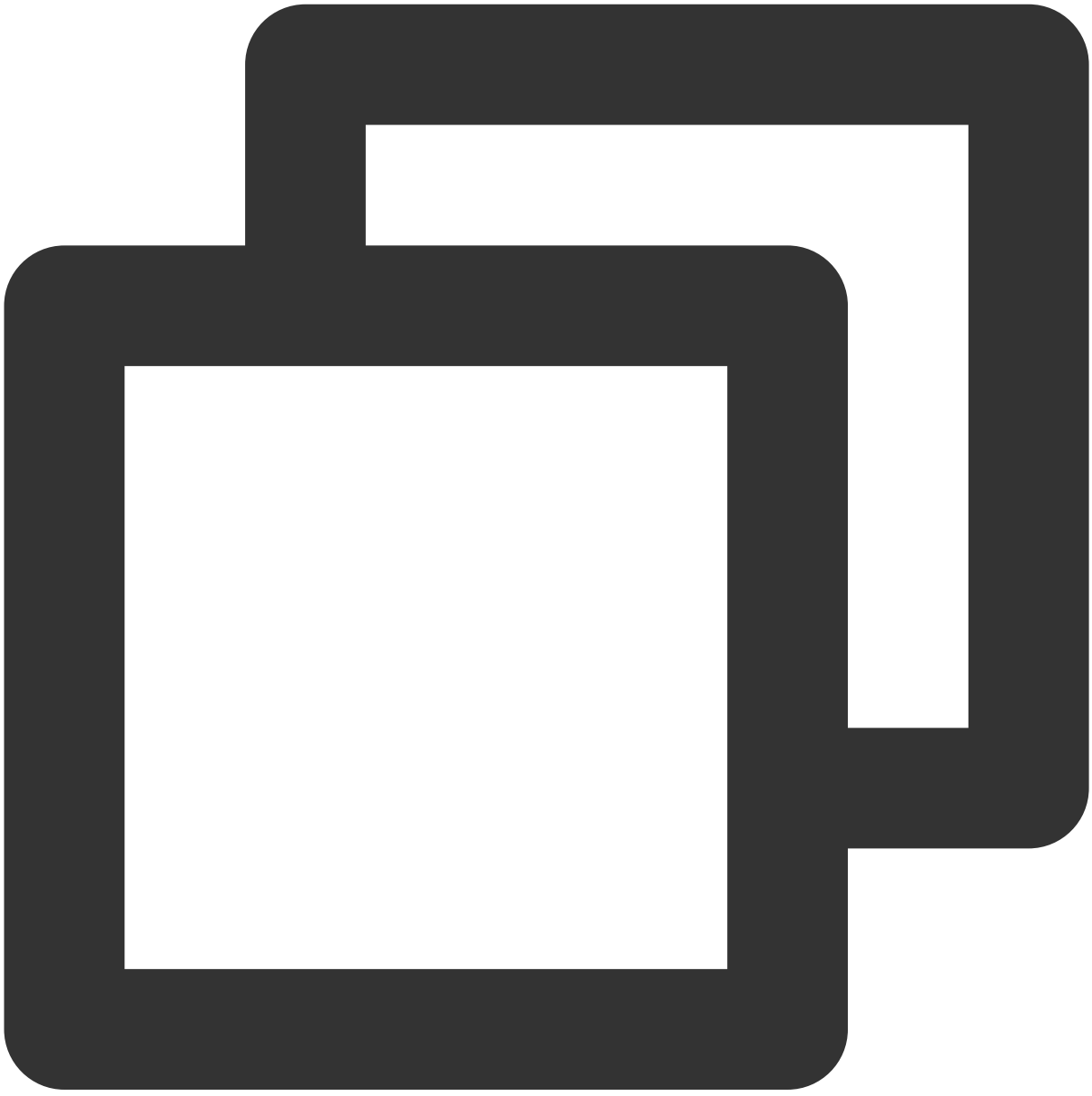
回调在主线程，日志回调可能比较频繁，请注意不要在回调里面同步处理太多耗时任务，可能会堵塞主线程。

配置 `V2TIMSDKConfig` 示例代码如下：

Android

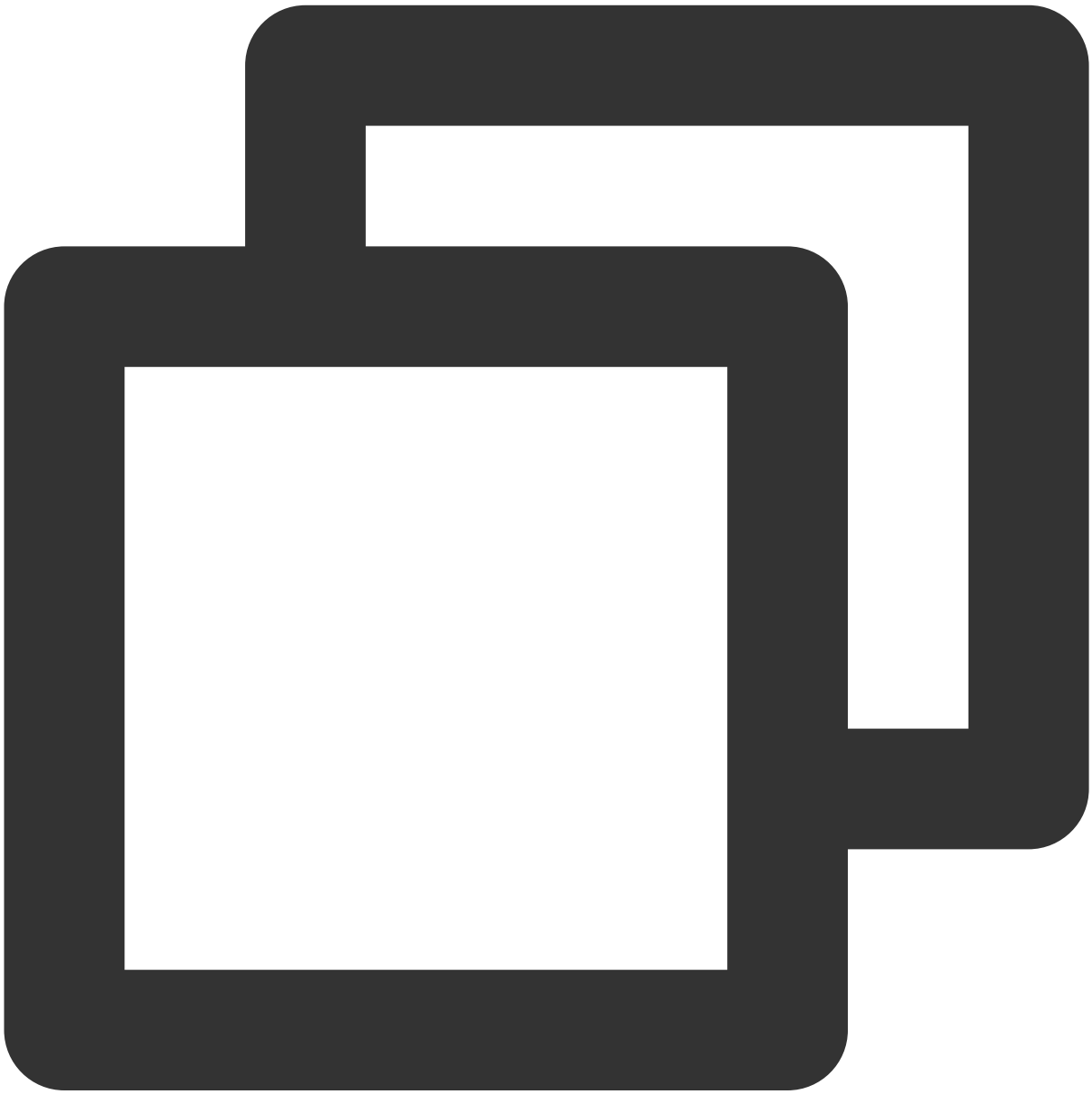
iOS & Mac

Windows



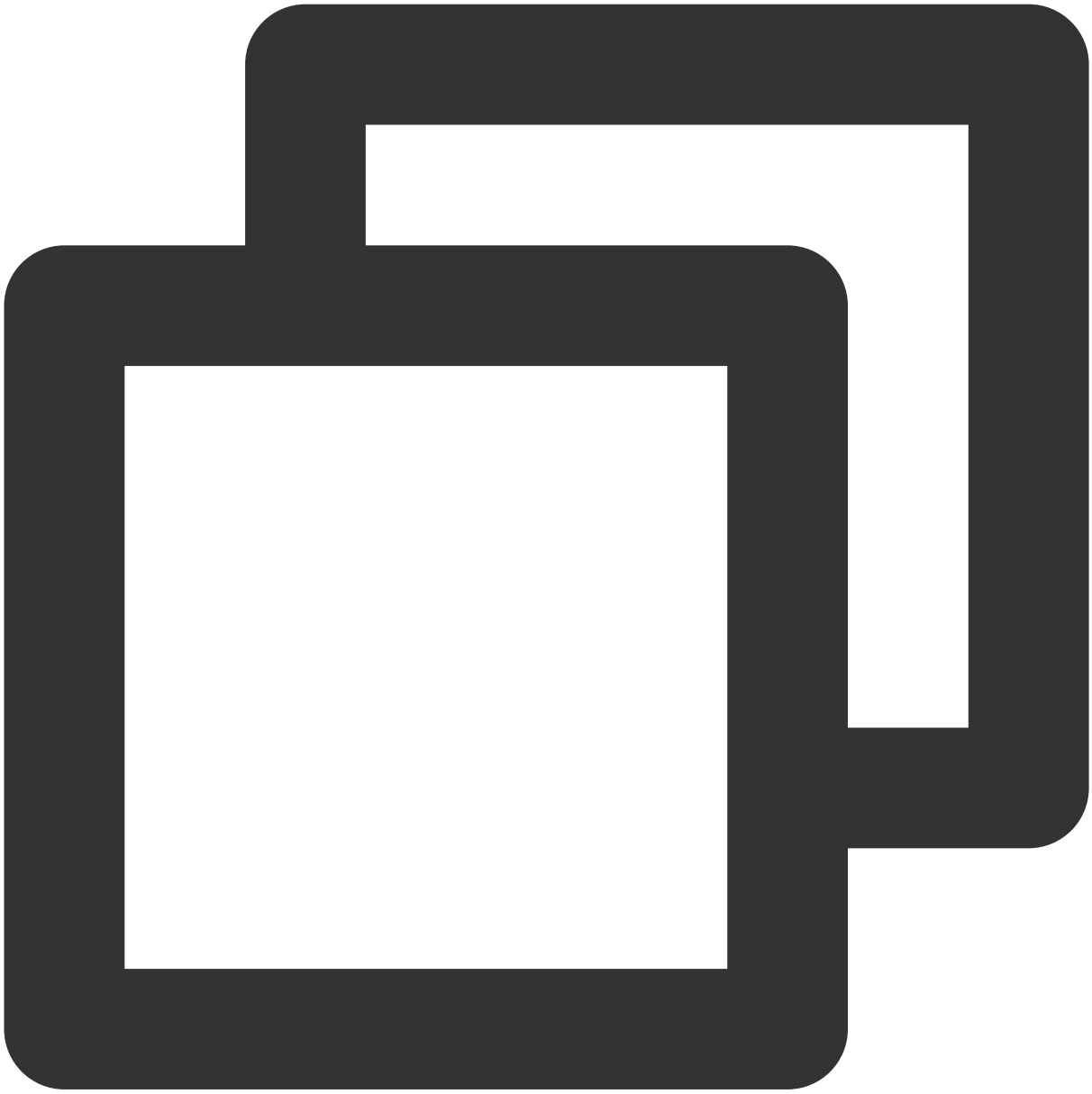
```
// 初始化 config 对象
V2TIMSDKConfig config = new V2TIMSDKConfig();
// 指定 log 输出级别
config.setLogLevel(V2TIMSDKConfig.V2TIM_LOG_INFO);
// 指定 log 监听器
config.setLogListener(new V2TIMLogListener() {
```

```
    @Override
    public void onLog(int logLevel, String logContent) {
        // logContent 为 SDK 日志内容
    }
});
```



```
// 初始化 config 对象
V2TIMSDKConfig *config = [[V2TIMSDKConfig alloc] init];
// 指定 log 输出级别
config.logLevel = V2TIM_LOG_INFO;
// 设置 log 监听器
```

```
config.logListener = ^(V2TIMLogLevel logLevel, NSString *logContent) {
    // logContent 为 SDK 日志内容
};
```



```
class LogListener final : public V2TIMLogListener {
public:
    LogListener() = default;
    ~LogListener() override = default;
    void OnLog(V2TIMLogLevel logLevel, const V2TIMString& logContent) override {
        // logContent 为 SDK 日志内容
}
```

```
};
// 注意 logListener 不能在 SDK 反初始化前释放，否则监听不到日志回调
LogListener logListener;

// 初始化 config 对象
V2TIMSDKConfig config;
// 指定 log 输出级别
config.logLevel = V2TIMLogLevel::V2TIM_LOG_INFO;
// 设置 log 监听器
config.logListener = &logListener;
```

添加 SDK 事件监听器

SDK 初始化后，会通过 `V2TIMSDKListener` 抛出一些事件，例如连接状态、登录票据过期等。
我们建议您调用 `addIMSDKListener` (Android / iOS & Mac/ Windows) 接口添加 SDK 事件监听器，在对应回调中做一些逻辑处理。

`V2TIMSDKListener` 相关回调如下表所示：

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	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。

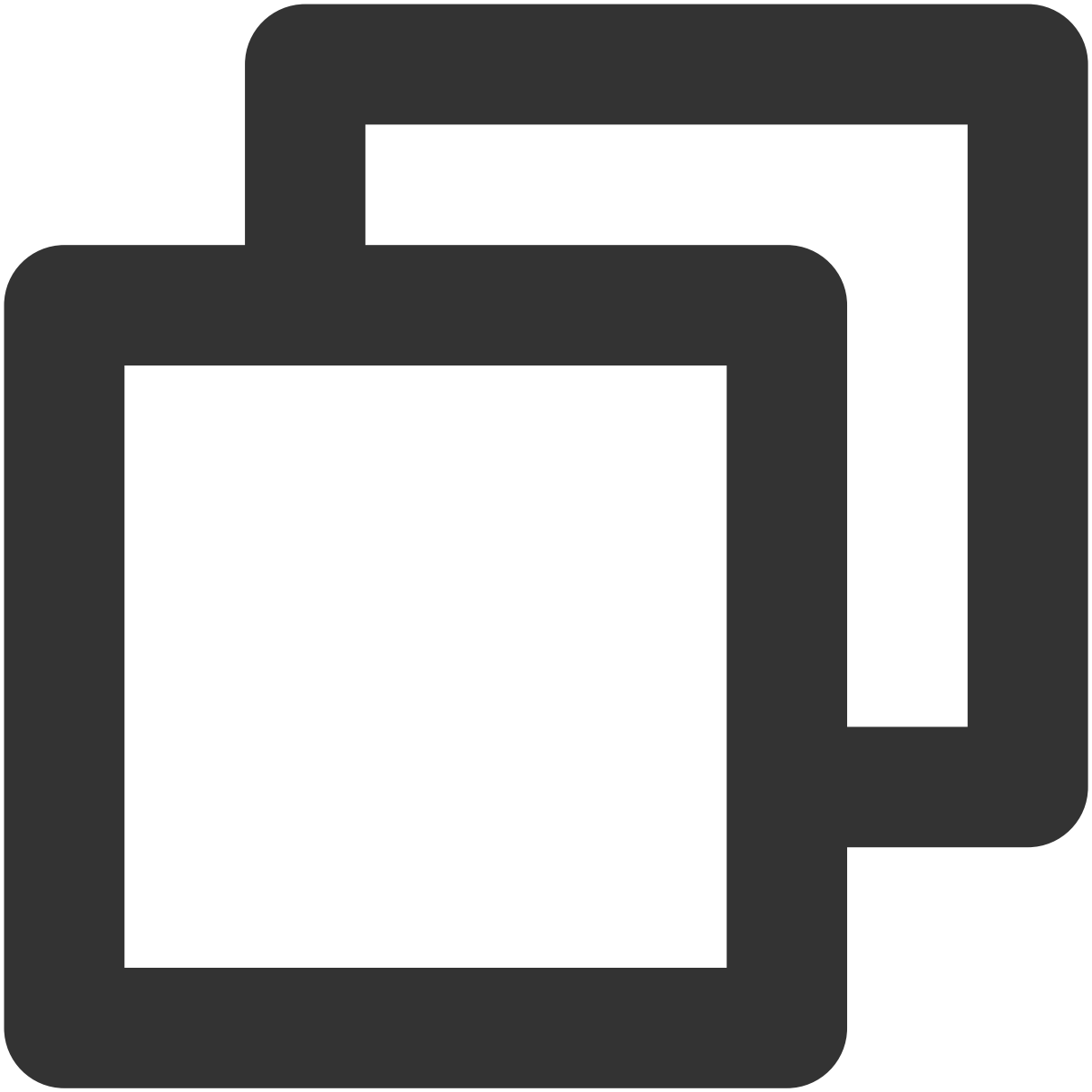
注意

如果收到 `onUserSigExpired` 回调，说明您登录用的 UserSig 票据已经过期，请使用新签发的 UserSig 进行重新登录。如果继续使用过期的 UserSig，会导致 SDK 登录进入死循环。
示例代码如下：

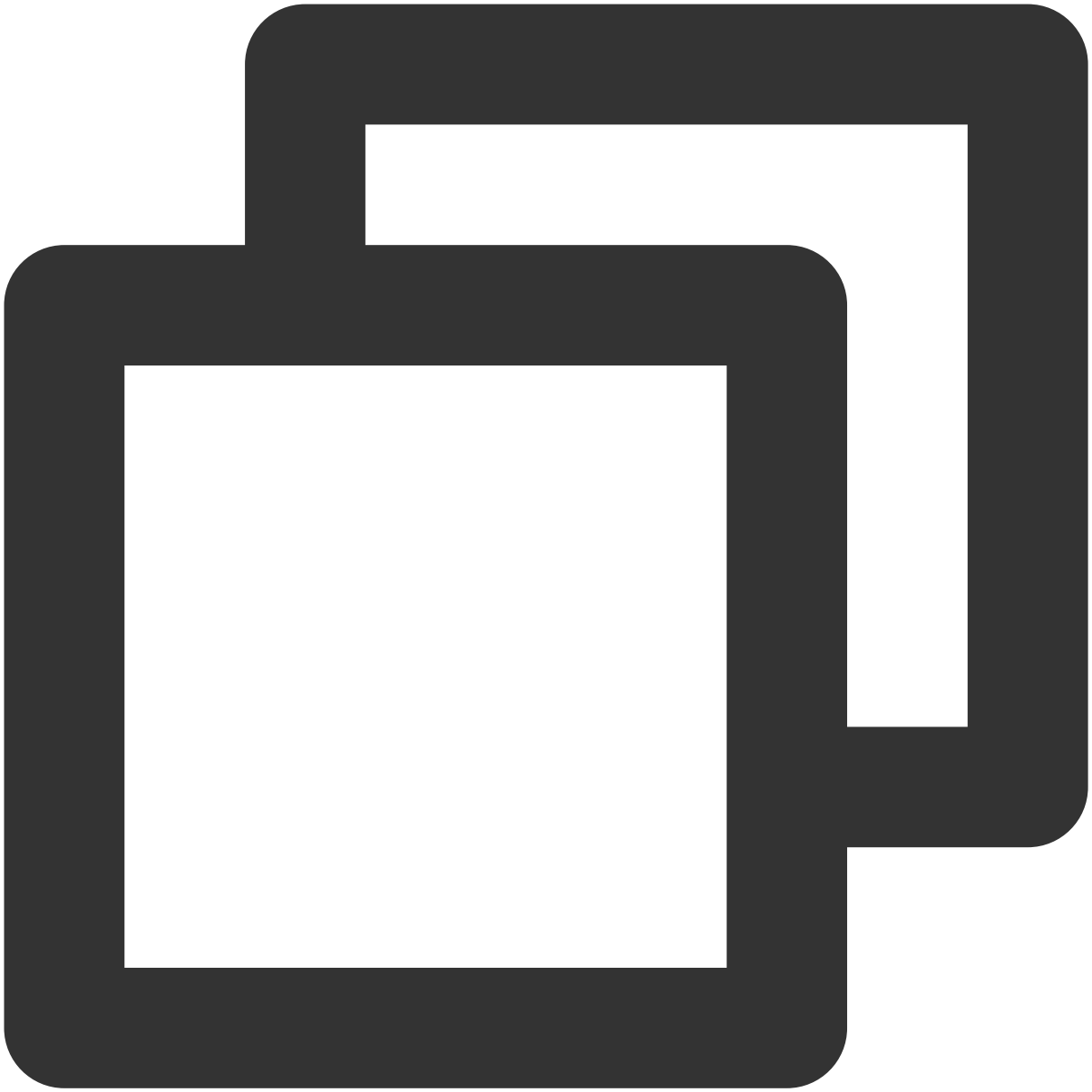
Android

iOS & Mac

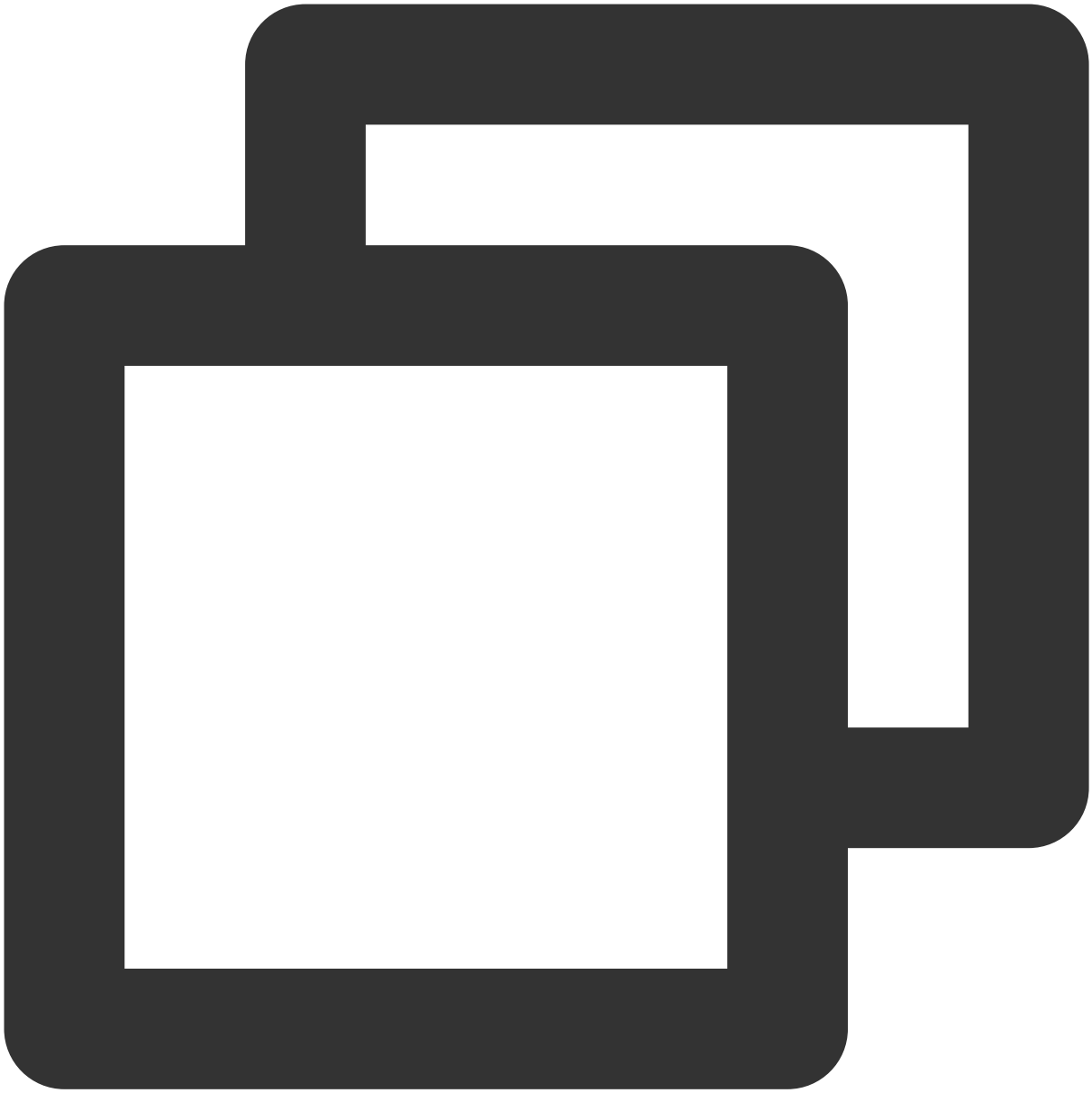
Windows



```
// sdkListener 类型为 V2TIMSDKListener
V2TIMManager.getInstance().addIMSDKListener(sdkListener);
```



```
// self 类型为 id<V2TIMSDKListener>
[[V2TIMManager sharedInstance] addIMSDKListener:self];
```



```
// sdkListener 是 V2TIMSDKListener 类的实例
V2TIMManager::GetInstance()->AddSDKListener(&sdkListener);
```

调用初始化接口

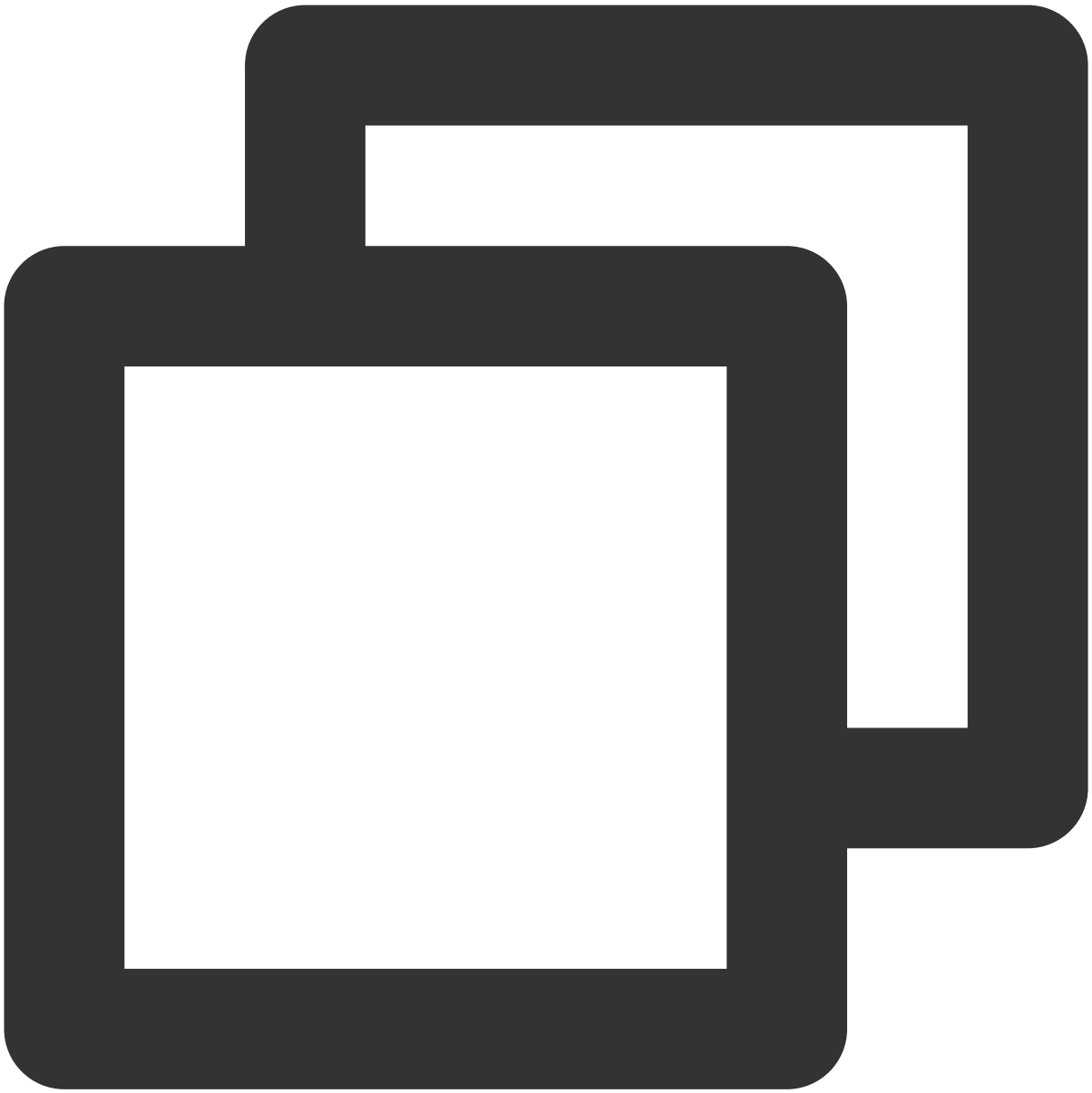
操作完上述步骤后，您可以调用 `initSDK` (Android / iOS & Mac/ Windows) 进行 SDK 初始化。

示例代码如下：

Android

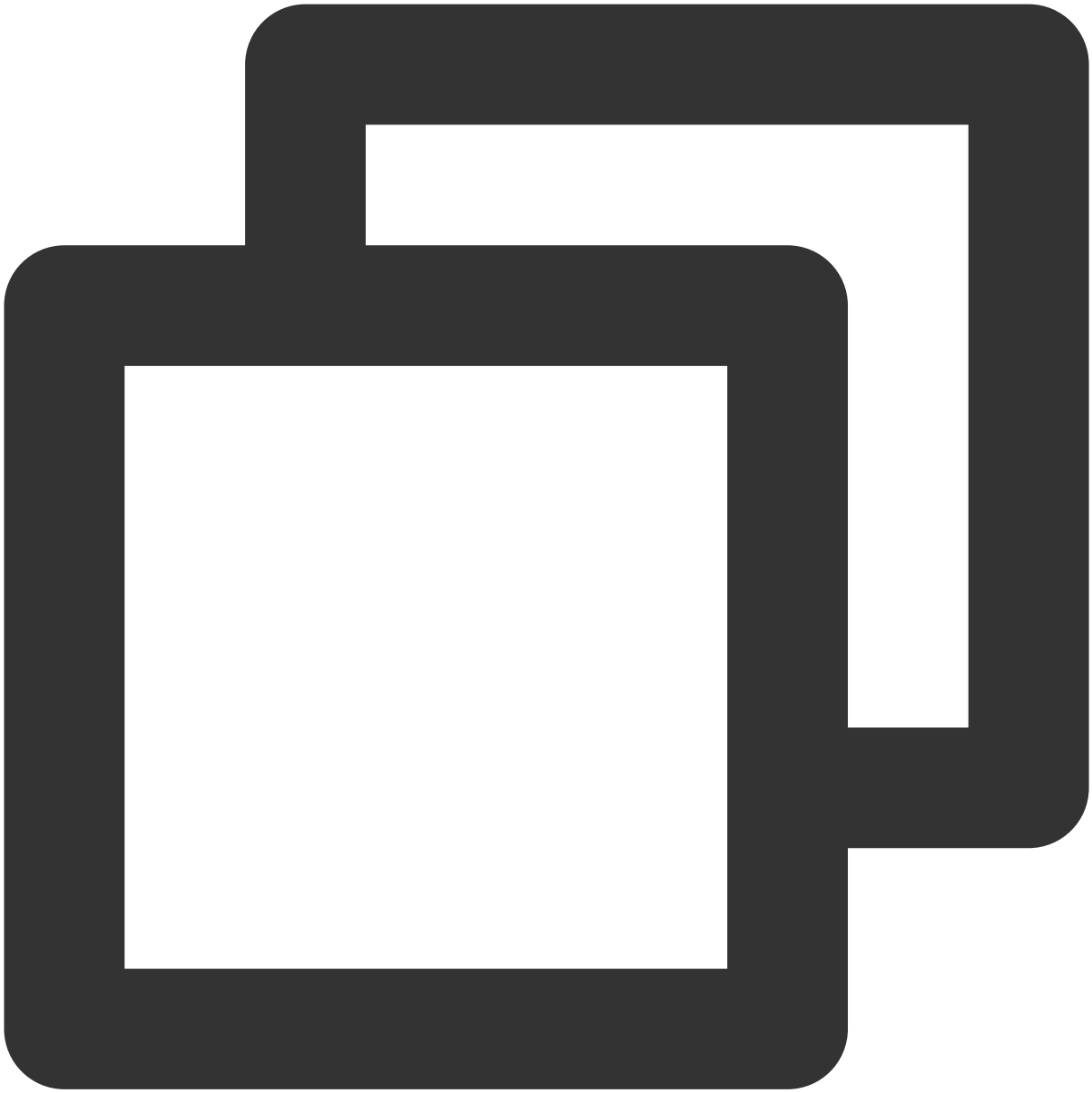
iOS & Mac

Windows



```
// 1. 从控制台获取应用 SDKAppID。
// 2. 初始化 config 对象。
V2TIMSDKConfig config = new V2TIMSDKConfig();
// 3. 指定 log 输出级别。
config.setLogLevel(V2TIMSDKConfig.V2TIM_LOG_INFO);
// 4. 添加 V2TIMSDKListener 的事件监听器，sdkListener 是 V2TIMSDKListener 的实现类，如果您
V2TIMManager.getInstance().addIMSDKListener(sdkListener);
// 5. 初始化 SDK，调用这个接口后，可以立即调用登录接口。
int  sdkAppID = 1400000000; // 请设置自己应用的 sdkAppID
```

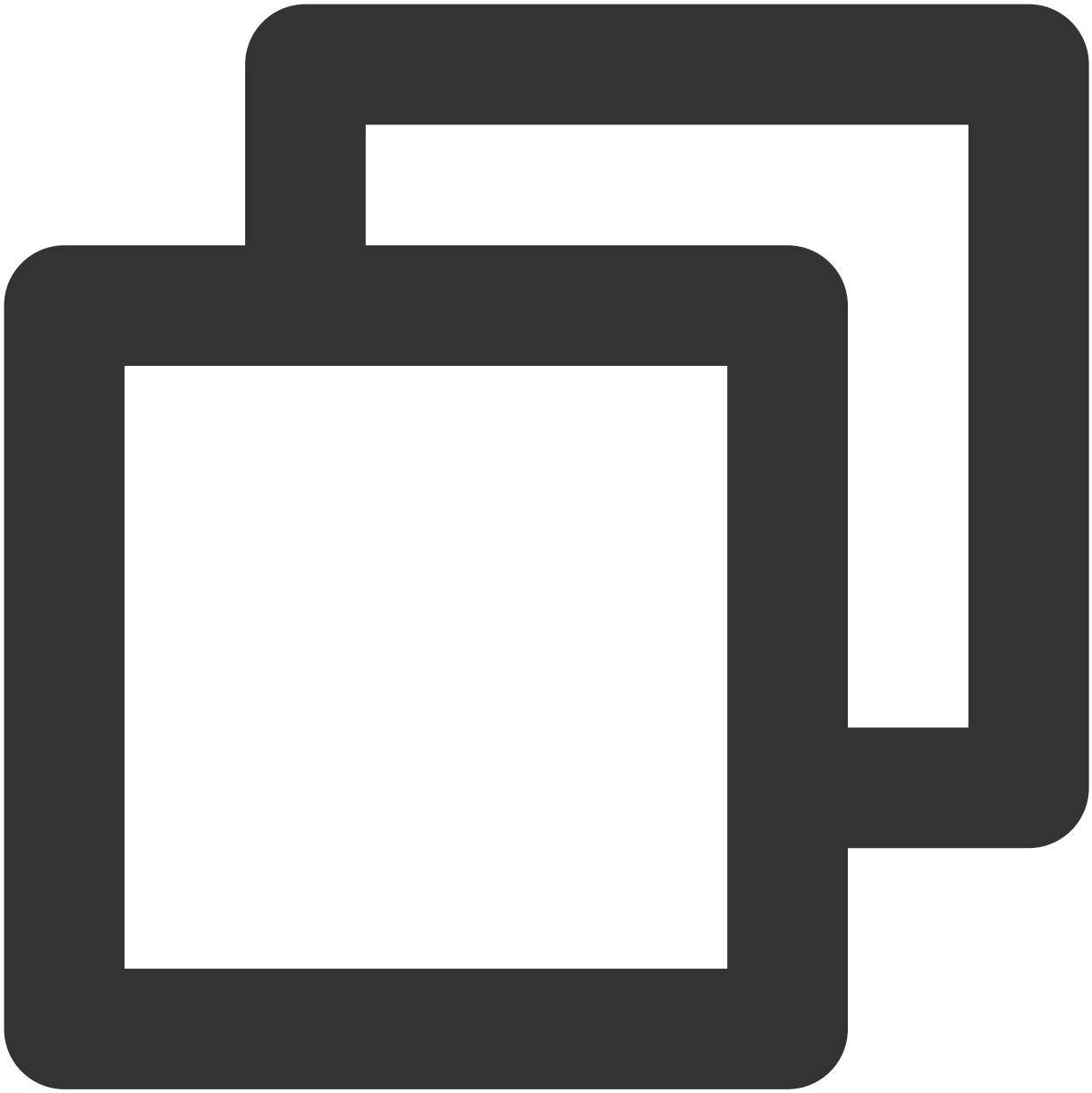
```
V2TIMManager.getInstance().initSDK(context, sdkAppID, config);
```



```
// 1. 从即时通信 IM 控制台获取应用 SDKAppID。
// 2. 初始化 config 对象
V2TIMSDKConfig *config = [[V2TIMSDKConfig alloc] init];
// 3. 指定 log 输出级别。
config.logLevel = V2TIM_LOG_INFO;
// 4. 添加 V2TIMSDKListener 的事件监听器，self 是 id<V2TIMSDKListener> 的实现类，如果您不
[[V2TIMManager sharedInstance] addIMSDKListener:self];
// 5. 初始化 SDK，调用这个接口后，可以立即调用登录接口。
int  sdkAppID = 1400000000; // 请设置自己应用的 sdkAppID
```

版权所有：腾讯云计算（北京）有限责任公司

```
[[V2TIMManager sharedInstance] initSDK:sdkAppID config:config];
```



```
// 1. 从即时通信 IM 控制台获取应用 SDKAppID。
// 2. 初始化 config 对象。
V2TIMSDKConfig config;
// 3. 指定 log 输出级别。
config.logLevel = V2TIMLogLevel::V2TIM_LOG_INFO;
// 4. 添加 V2TIMSDKListener 的事件监听器，sdkListener 是 V2TIMSDKListener 类的实例，如果您
V2TIMManager::GetInstance()->AddSDKListener(&sdkListener);
// 5. 初始化 SDK，调用这个接口后，可以立即调用登录接口。
int  sdkAppID = 1400000000; // 请设置自己应用的 sdkAppID
```

版权所有：腾讯云计算（北京）有限责任公司

```
V2TIMManager::GetInstance()->InitSDK(sdkAppID, config);
```

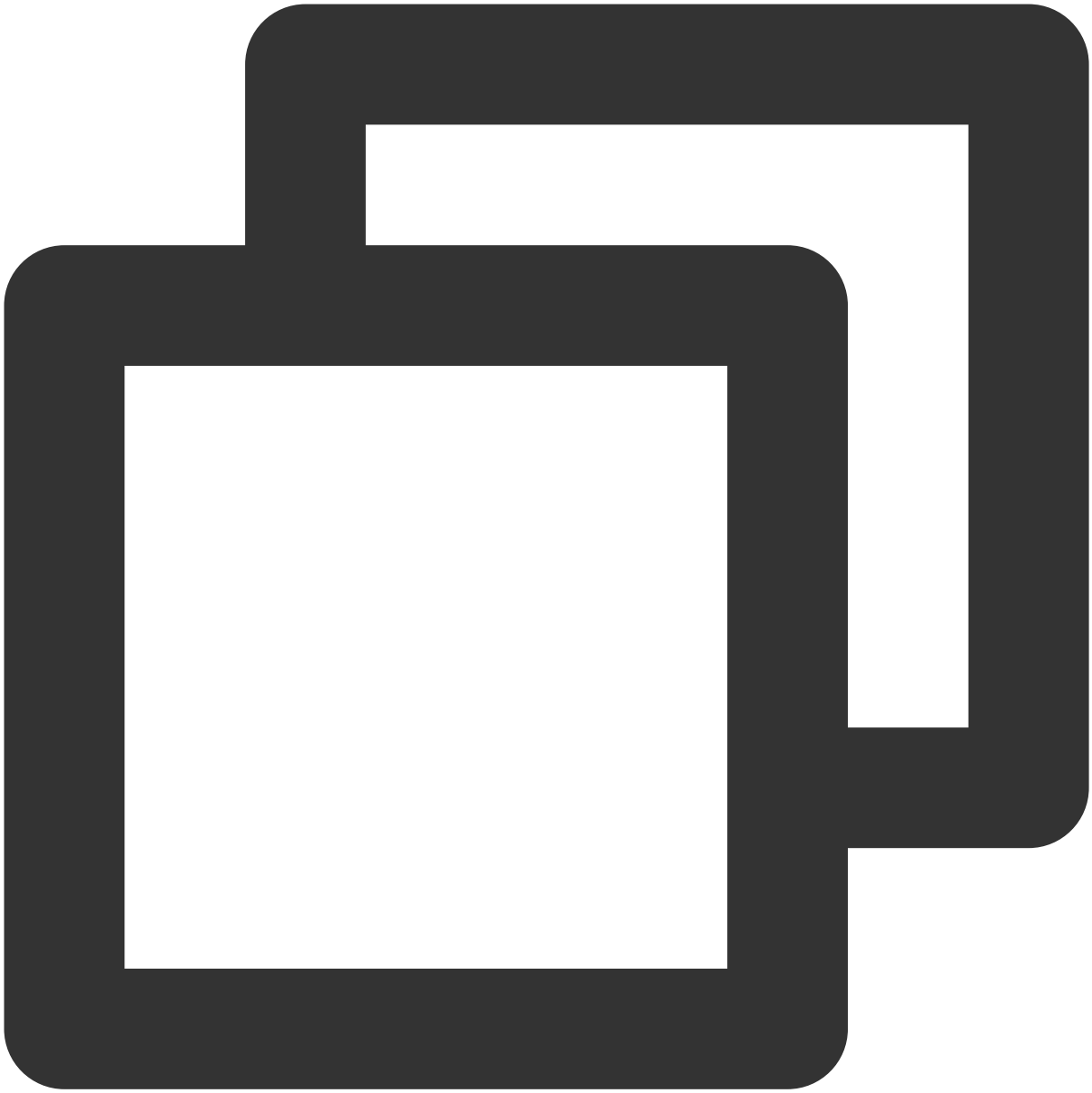
反初始化

普通情况下，如果您的应用生命周期跟 SDK 生命周期一致，退出应用前可以不进行反初始化。
但有些特殊场景，例如您只在进入特定界面后才初始化 SDK，退出界面后不再使用，可以对 SDK 进行反初始化。
反初始化需要操作 2 步：
1. 如果你前面调用了 `addIMSDKListener` 添加了 SDK 监听器，此时请调用 `removeIMSDKListener` (Android / iOS & Mac/ Windows) 移除 SDK 监听器。
2. 调用反初始化接口 `unInitSDK` (Android / iOS & Mac/ Windows)
示例代码如下：

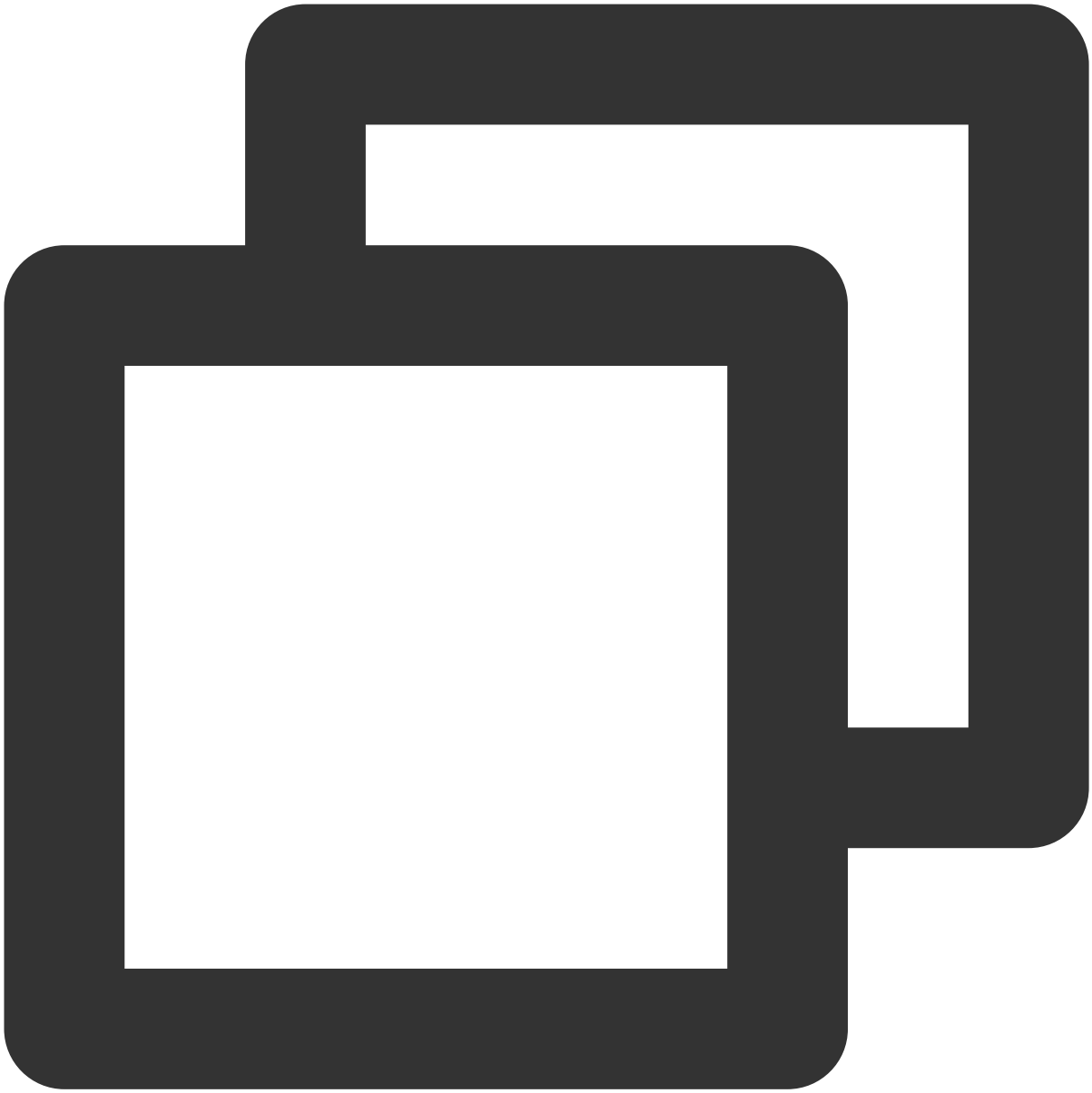
Android

iOS & Mac

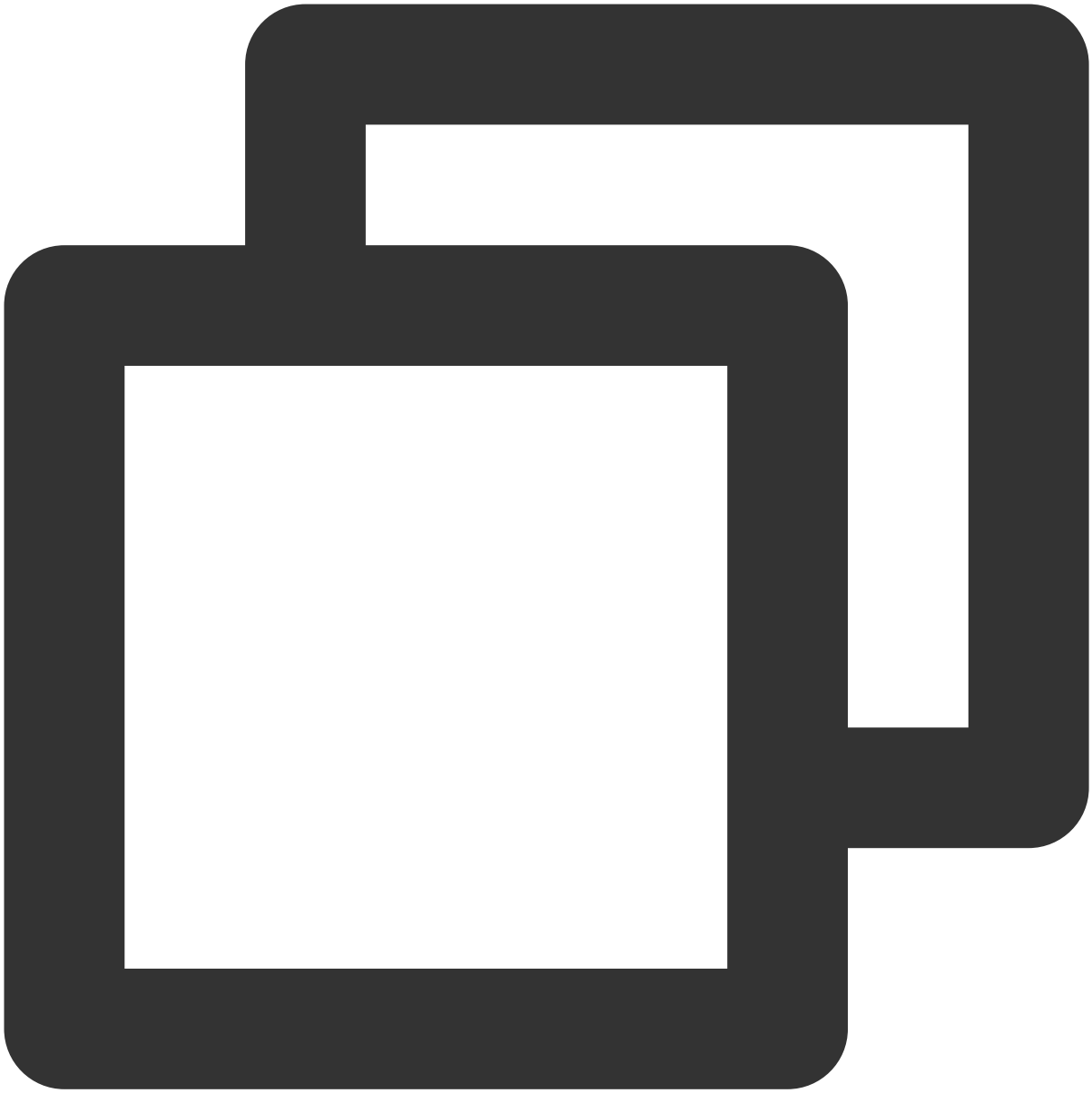
Windows



```
// 移除 V2TIMSDKListener 的事件监听器，sdkListener 是 V2TIMSDKListener 的实现类
V2TIMManager.getInstance().removeIMSDKListener(sdkListener);
// 反初始化 SDK
V2TIMManager.getInstance().unInitSDK();
```



```
// self 是 id<V2TIMSDKListener> 的实现类
[[V2TIMManager sharedInstance] removeIMSDKListener:self];
// 反初始化 SDK
[[V2TIMManager sharedInstance] unInitSDK];
```



```
// 移除 V2TIMSDKListener 的事件监听器，sdkListener 是 V2TIMSDKListener 类的实例
V2TIMManager::GetInstance()->RemoveSDKListener(&sdkListener);
// 反初始化 SDK
V2TIMManager::GetInstance()->UnInitSDK();
```

常见问题

在调用登录等其他接口时，发生错误，返回错误码是 6013 和错误描述是 "not initialized" 的信息。

在使用 SDK 登录、消息、群组、会话、关系链和资料、信令的功能前，必须先进行初始化。

Web

最近更新时间：2024-07-10 16:31:08

功能描述

在使用 Chat SDK 的各项功能前，**必须**先进行初始化。

初始化

初始化 SDK 需要操作以下步骤：

1. 准备 SDKAppID。
2. 调用 `TencentCloudChat.create` 初始化 SDK。
3. 监听 SDK 事件。

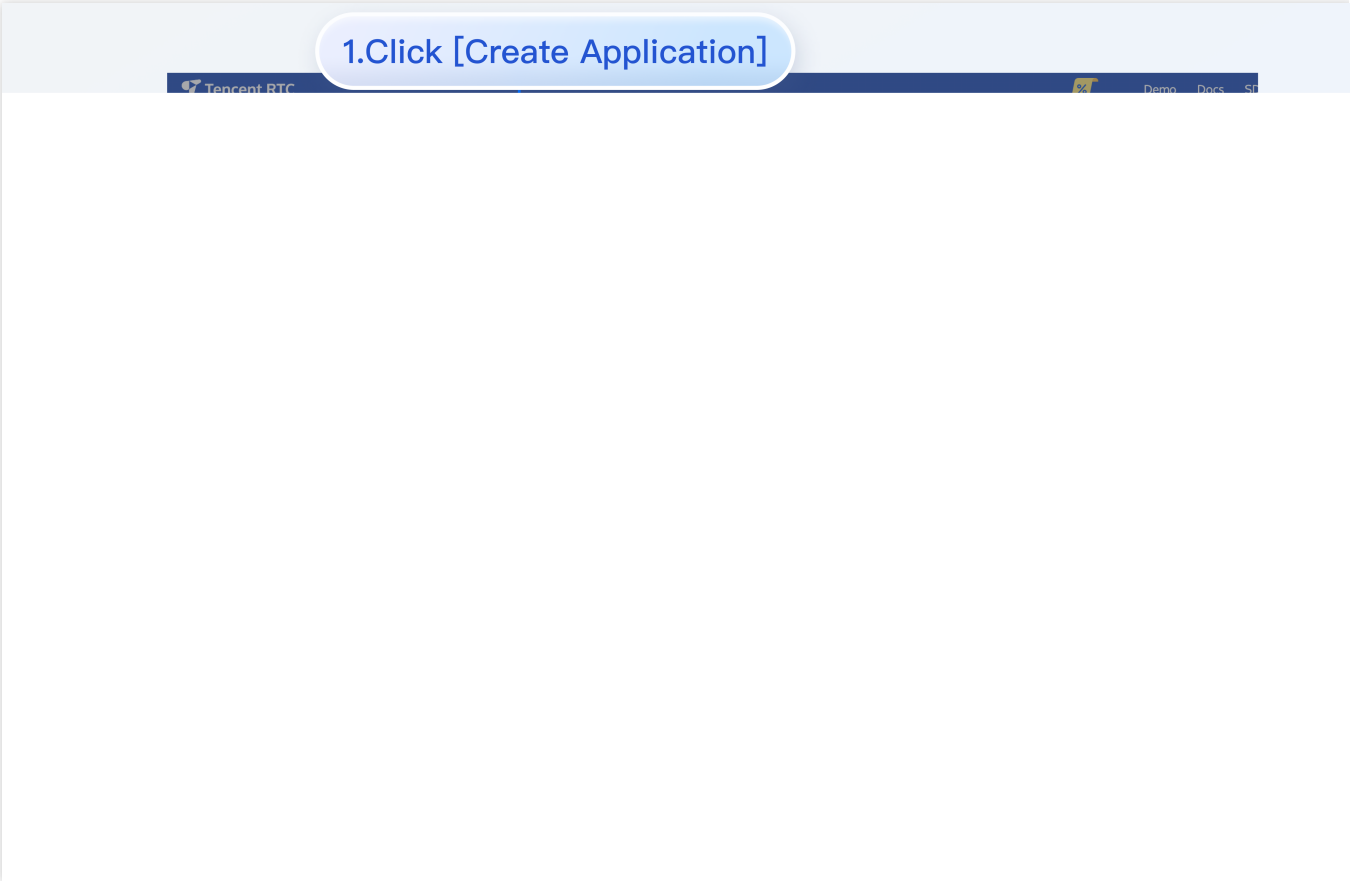
下面我们将分步骤依次为您详细讲解。

准备 SDKAppID

您必须拥有正确的 SDKAppID，才能进行初始化。

SDKAppID 是 Tencent Cloud Chat 服务区分客户账号的唯一标识。我们建议每一个独立的 App 都申请一个新的 SDKAppID。不同 SDKAppID 之间的消息是天然隔离的，不能互通。

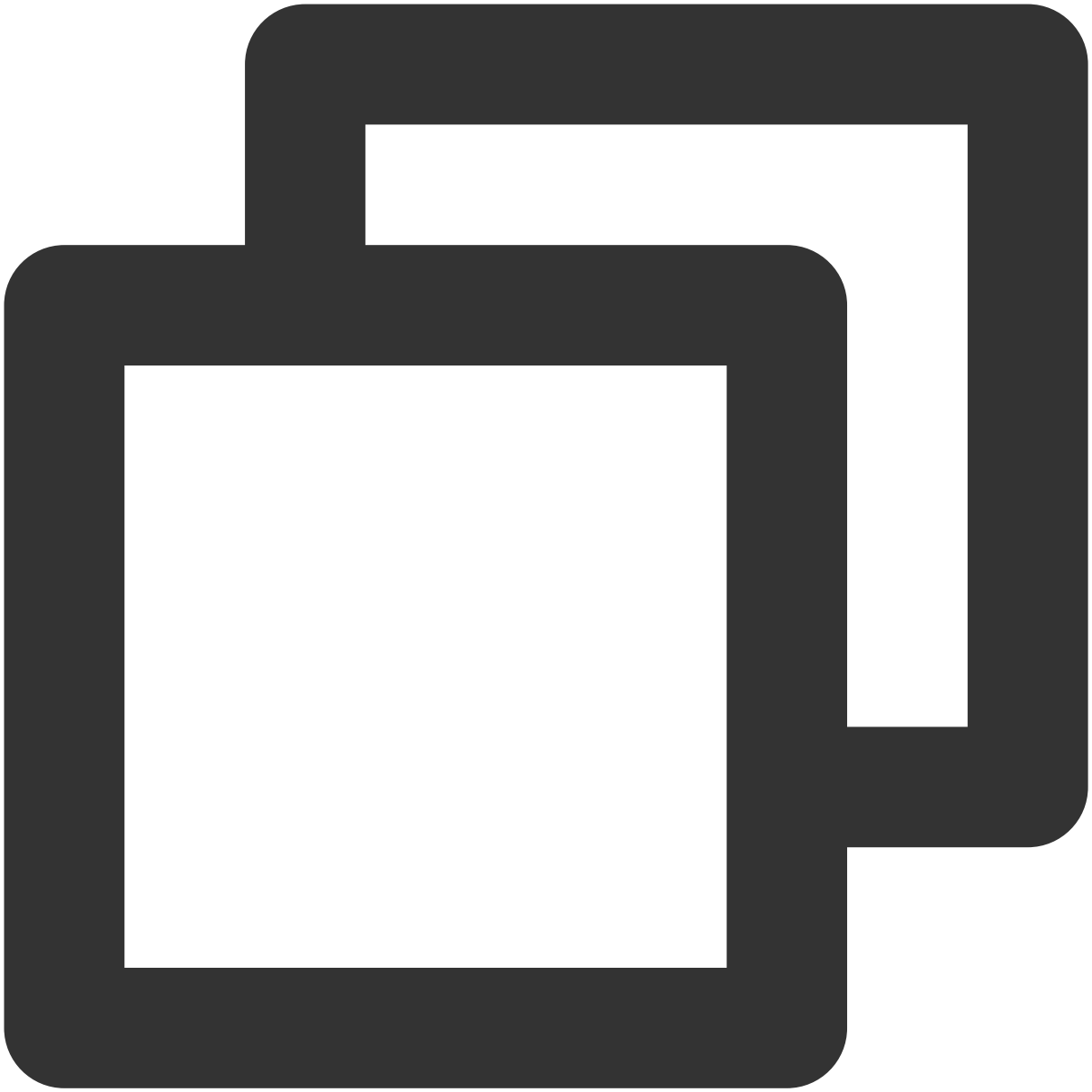
您可以在 Chat Console 查看所有的 SDKAppID，单击 `Create Applicaion` 按钮，创建新的 SDKAppID。



调用初始化接口

操作完上述步骤后，您可以调用 TencentCloudChat.create 初始化 SDK。

接口

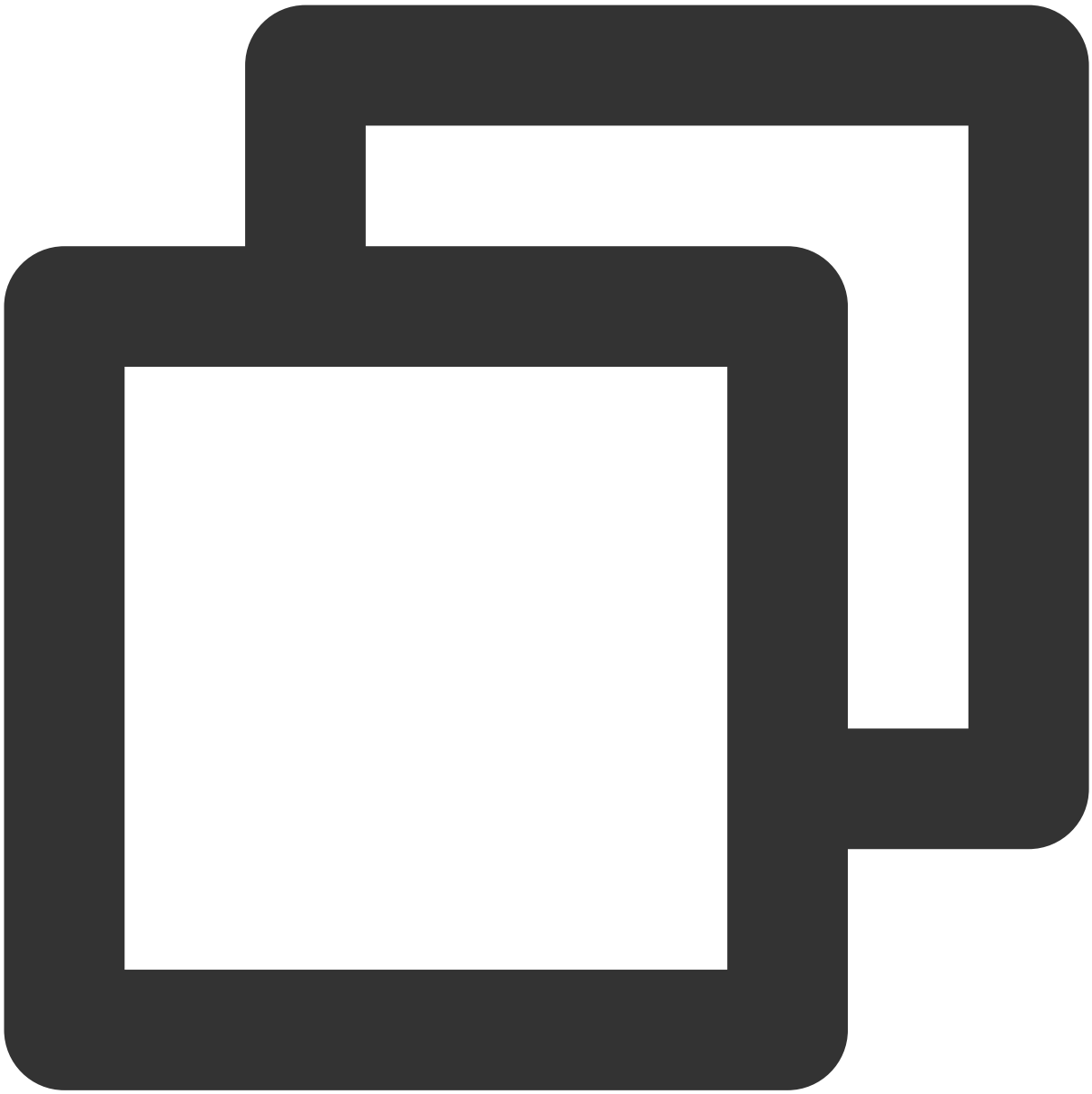


```
TencentCloudChat.create(options);
```

参数 options 为 Object 类型，包含的属性值如下：

Name	Type	Description
SDKAppID	Number	即时通信 IM 应用的 SDKAppID
proxyServer	String | undefined	WebSocket 服务器代理地址（小程序平台不支持使用 IP 地址）

示例



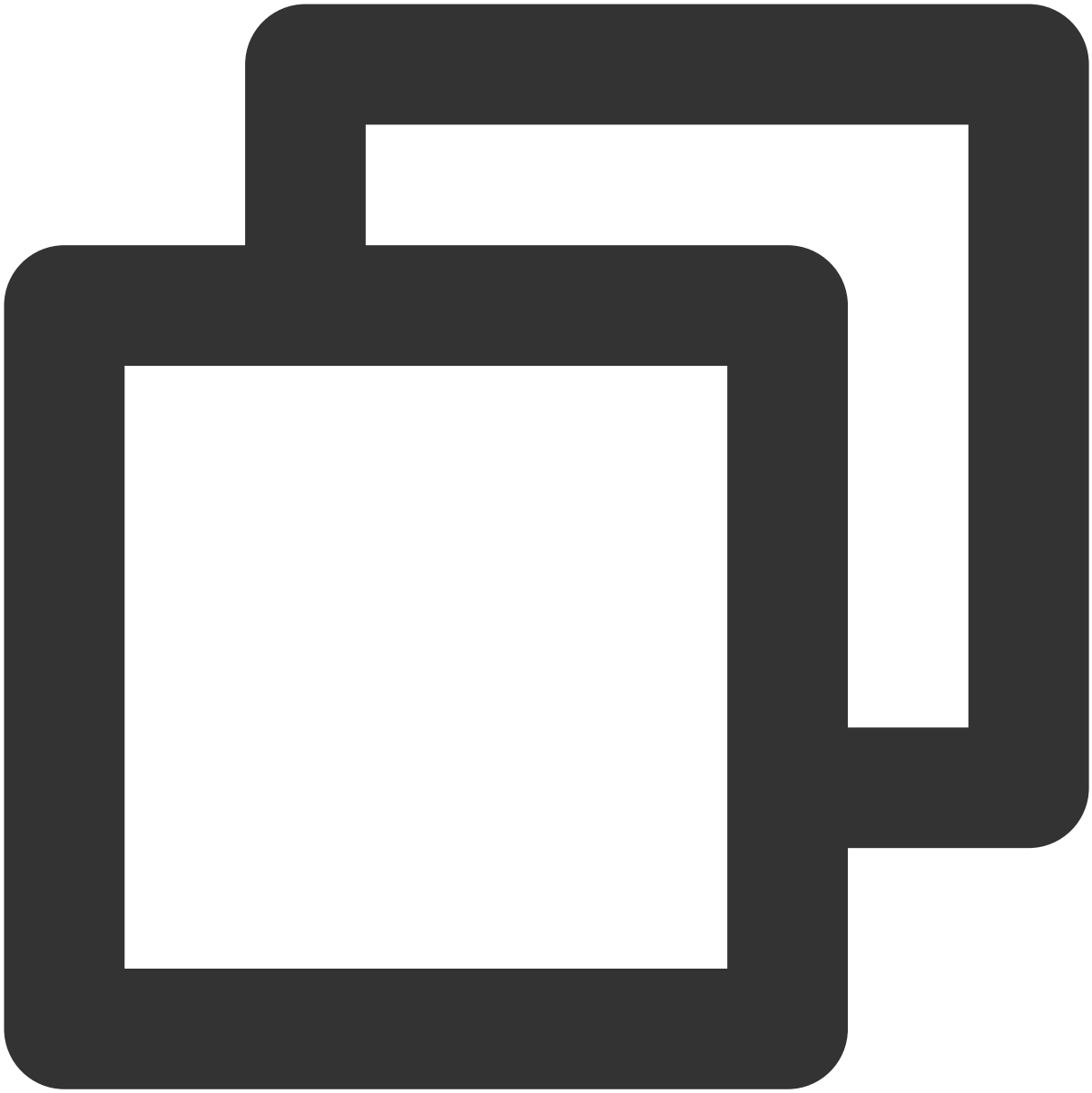
```
// 如果您已集成 v2.x 的 SDK，想升级到 V3 并且想尽可能地少改动项目代码，可以继续沿用 TIM
// import TIM from '@tencentcloud/chat';
import TencentCloudChat from '@tencentcloud/chat';
import TIMUploadPlugin from 'tim-upload-plugin';
import TIMProfanityFilterPlugin from 'tim-profanity-filter-plugin';

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，`TencentCloudChat.create()`方法对于同一个 `SDKAppID` 只会返回同一份实例
let chat = TencentCloudChat.create(options); // SDK 实例通常用 chat 表示

chat.setLogLevel(0); // 普通级别，日志量较多，接入时建议使用
// chat.setLogLevel(1); // release 级别，SDK 输出关键信息，生产环境时建议使用

// 注册腾讯云即时通信 IM 上传插件
chat.registerPlugin({'tim-upload-plugin': TIMUploadPlugin});

// 注册腾讯云即时通信 IM 本地审核插件
chat.registerPlugin({'tim-profanity-filter-plugin': TIMProfanityFilterPlugin});
```

监听事件

SDK_READY

SDK 进入 ready 状态时触发，接入侧监听此事件，然后可调用 SDK 发送消息等 API，使用 SDK 的各项功能。

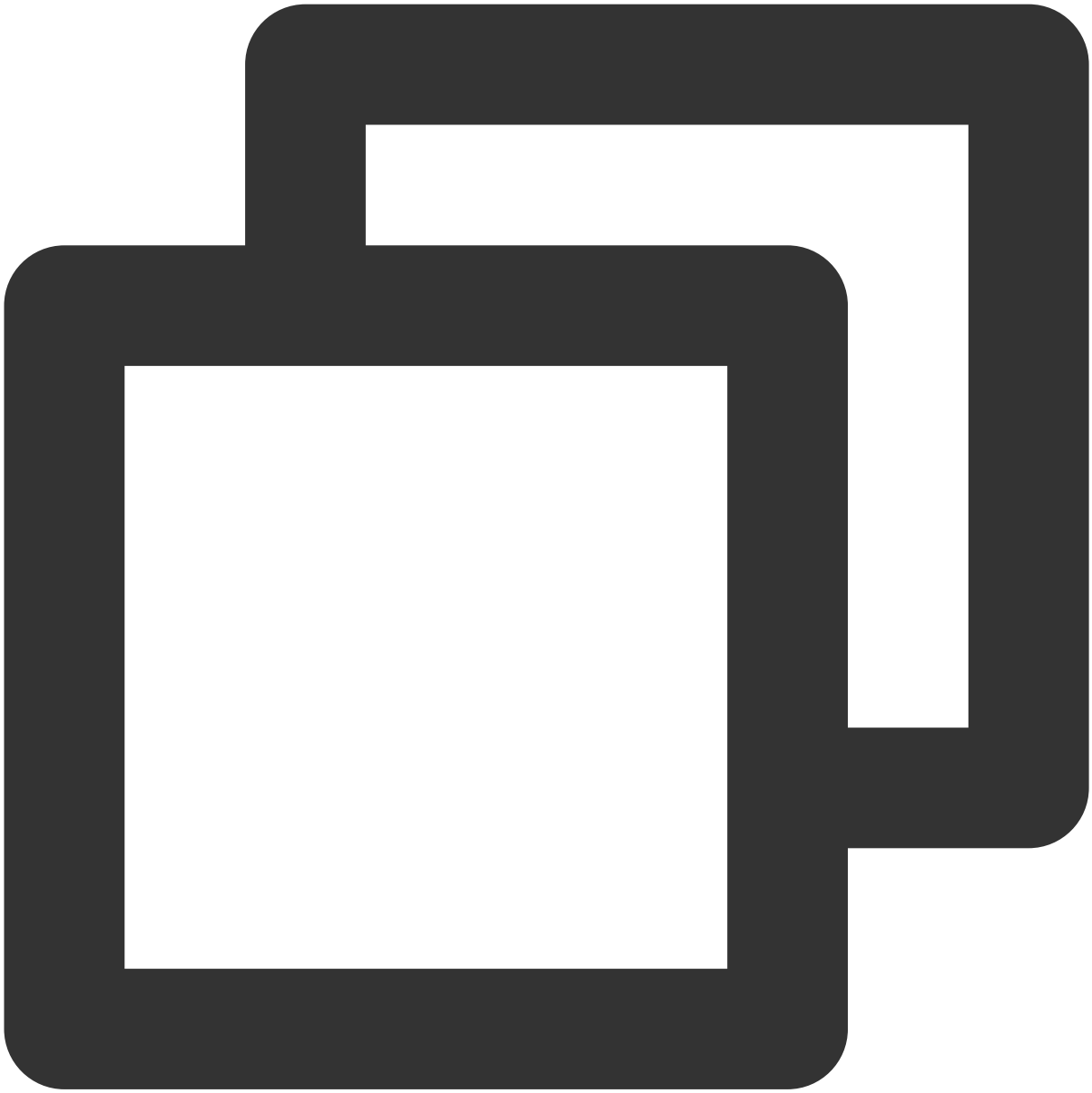


```
let onSdkReady = function(event) {
  let message = chat.createTextMessage({
    to: 'user1',
    conversationType: 'C2C',
    payload: { text: 'Hello world!' }
  });
  chat.sendMessage(message);
};
chat.on(TencentCloudChat.EVENT.SDK_READY, onSdkReady);
```

版权所有：腾讯云计算（北京）有限责任公司

SDK_NOT_READY

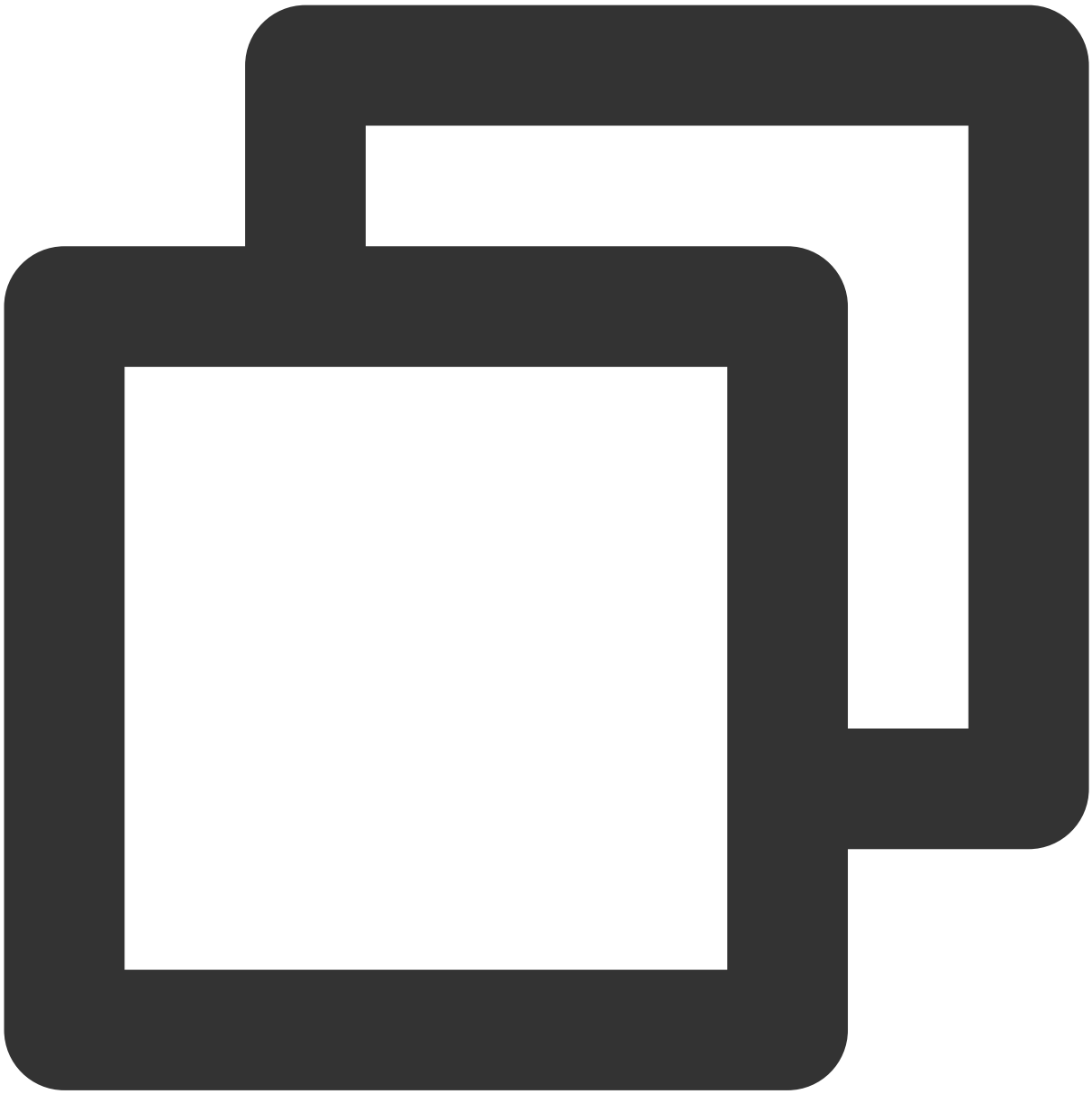
SDK 进入 not ready 状态时触发，此时接入侧将无法使用 SDK 发送消息等功能。如果想恢复使用，接入侧需调用 login 接口，驱动 SDK 进入 ready 状态。



```
let onSdkNotReady = function(event) {
  // chat.login({userID: 'your userID', userSig: 'your userSig'});
};
chat.on(TencentCloudChat.EVENT.SDK_NOT_READY, onSdkNotReady);
```

MESSAGE_RECEIVED

SDK 收到推送的单聊、群聊、群提示、群系统通知的新消息，接入侧可通过遍历 event.data 获取消息列表数据并渲染到页面。



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

MESSAGE_MODIFIED

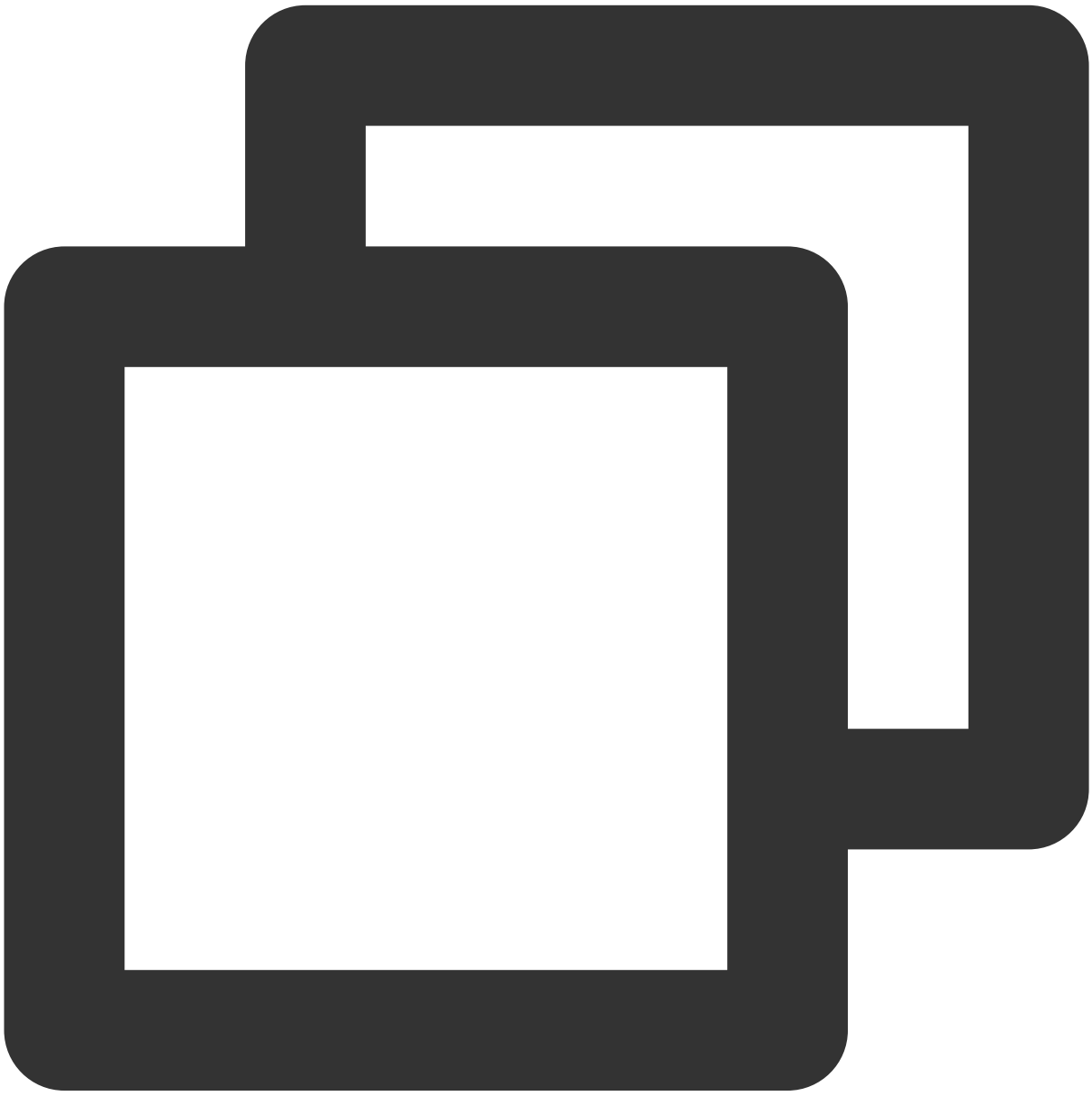
SDK 收到消息被修改的通知，消息发送方可通过遍历 event.data 获取消息列表数据并更新页面上同 ID 消息的内容。

版权所有：腾讯云计算（北京）有限责任公司

使用举例：

1、单聊用户或群成员发送了一个【Hello World】的消息并上屏，然后 SDK 收到了此消息被第三方回调修改的通知（例如被修改成了【Hello China】，对端或者其他群成员收到的是【Hello China】），此时 SDK 会触发此事件。

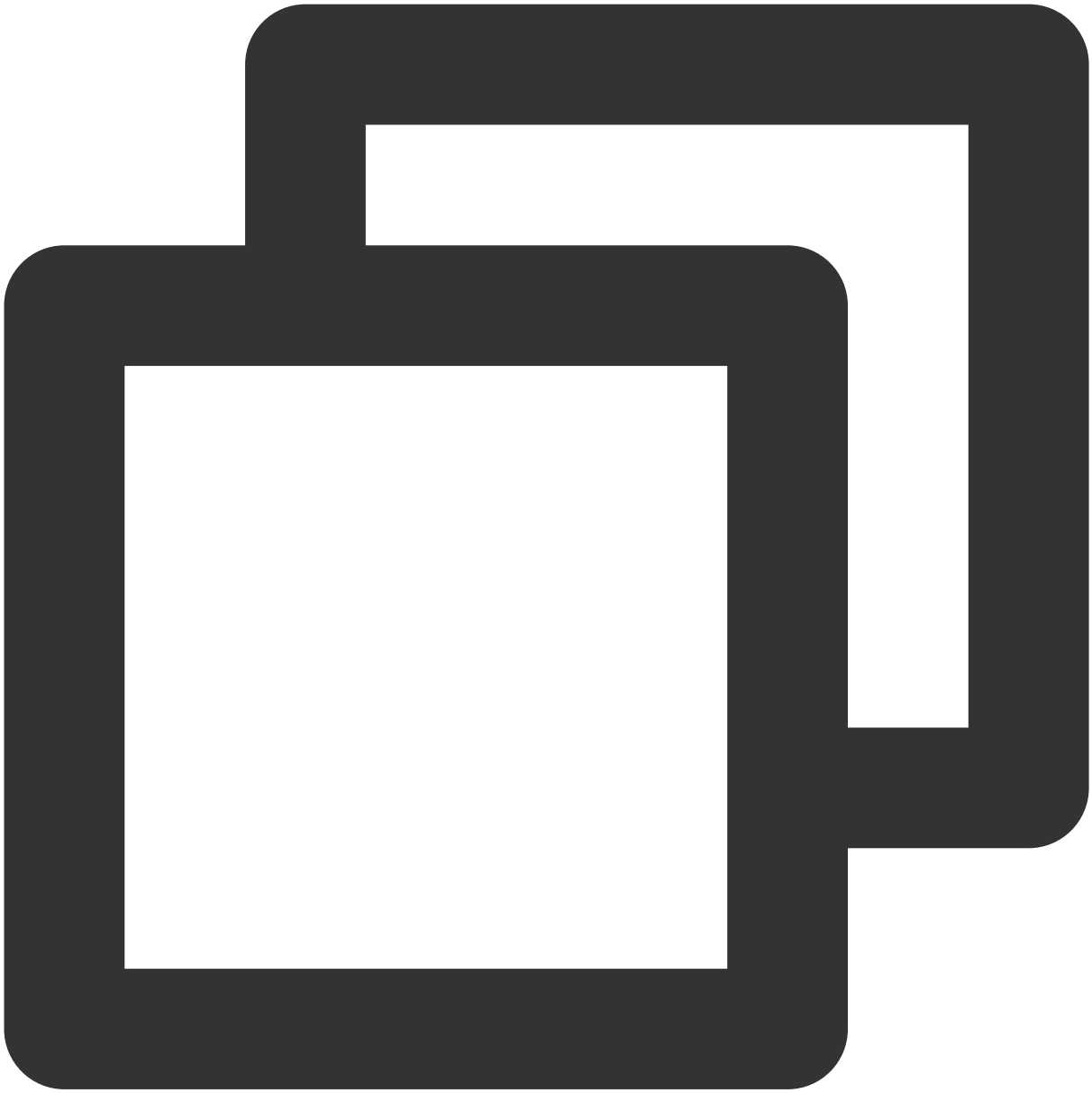
2、接入侧在事件回调里，遍历被修改的消息，替换页面上同 ID 消息的内容，将【Hello World】替换为【Hello China】，并重新渲染和展示。



```
let onMessageModified = function(event) {
  // event.data - 存储被修改过的 Message 对象的数组 - [Message]
};
chat.on(TencentCloudChat.EVENT.MESSAGE_MODIFIED, onMessageModified);
```

MESSAGE_REVOKED

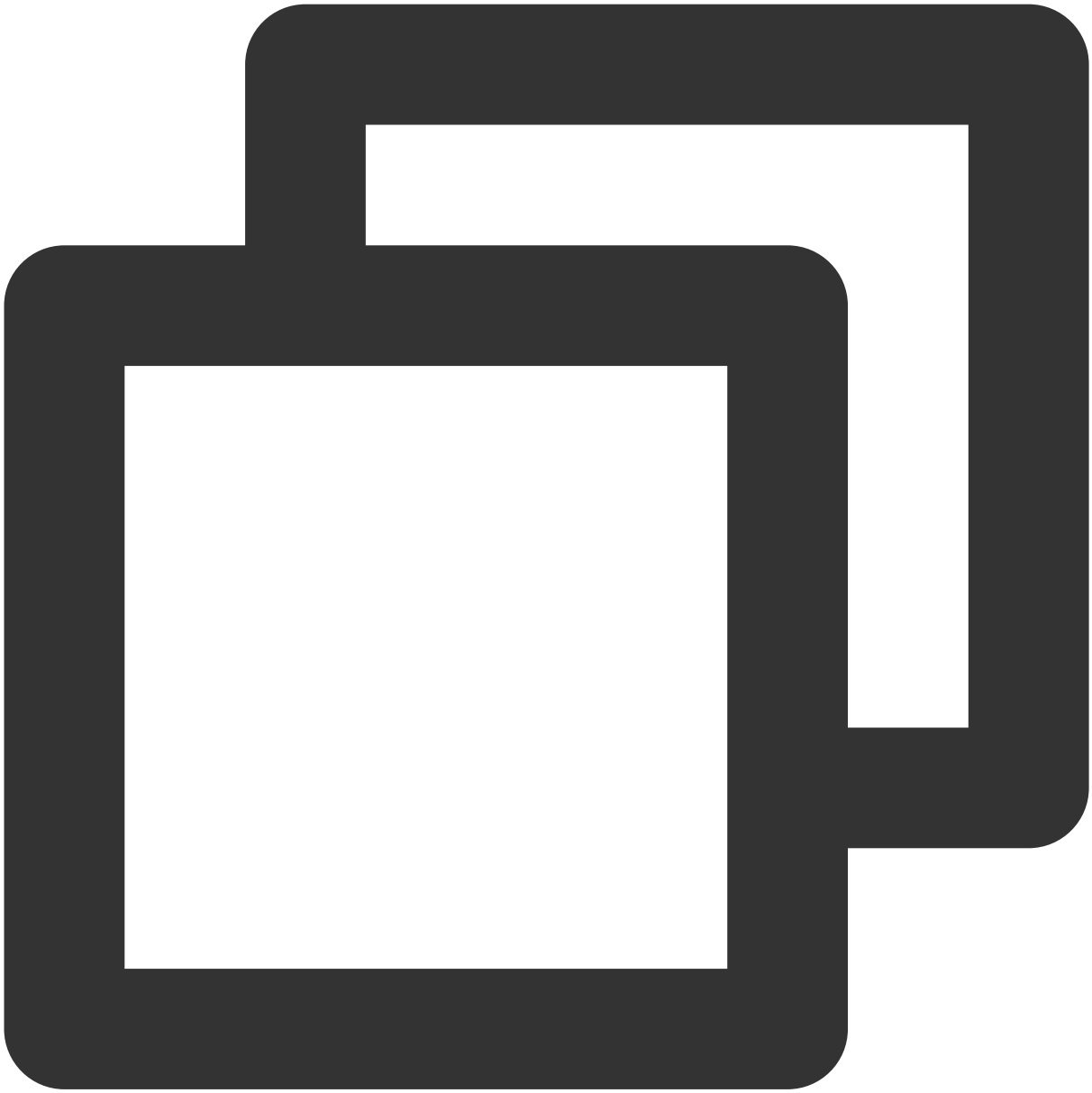
SDK 收到消息被撤回的通知，接入侧可通过遍历 event.data 获取被撤回的消息列表数据并渲染到页面，如单聊会话内可展示为 "对方撤回了一条消息"；群聊会话内可展示为 "XXX撤回了一条消息"。



```
let onMessageRevoked = function(event) {
  // event.data - 存储 Message 对象的数组 - [Message] - 每个 Message 对象的 isRevoked 属
};
chat.on(TencentCloudChat.EVENT.MESSAGE_REVOKED, onMessageRevoked);
```

MESSAGE_READ_BY_PEER

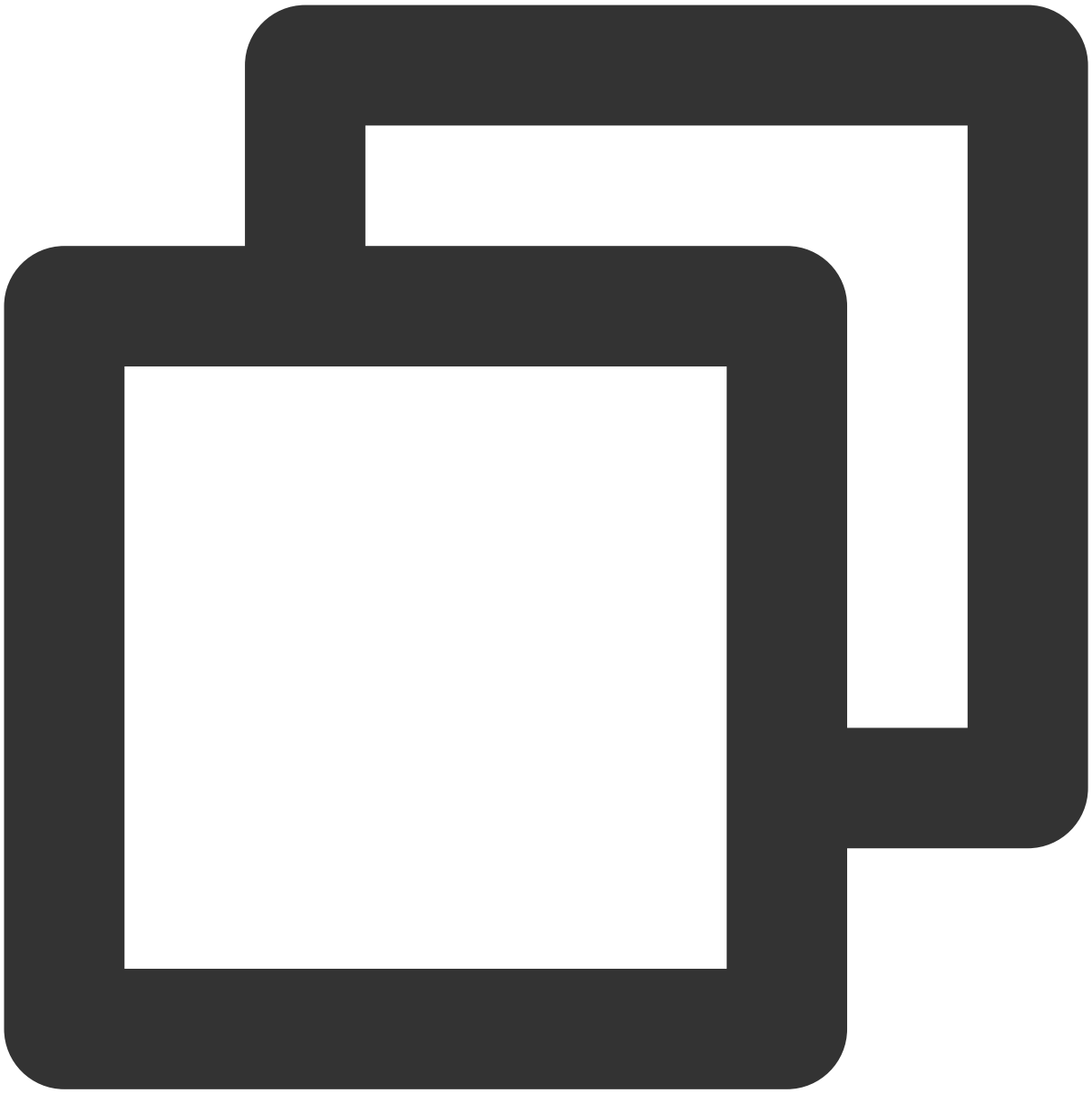
SDK 收到对端已读消息的通知，消息接收方调用 setMessageRead 已读上报成功后消息发送方会收到此事件。接入侧可通过遍历 event.data 获取对端已读的消息列表数据并渲染到页面，如单聊会话内可将自己发送的消息由“未读”状态改为“已读”。



```
let onMessageReadByPeer = function(event) {
  // event.data - 存储 Message 对象的数组 - [Message] - 每个 Message 对象的 isPeerRead
};
chat.on(TencentCloudChat.EVENT.MESSAGE_READ_BY_PEER, onMessageReadByPeer);
```

MESSAGE_READ_RECEIPT_RECEIVED

SDK 收到了消息的已读回执通知，消息接收方调用 sendMessageReadReceipt 时消息发送方会收到此事件。

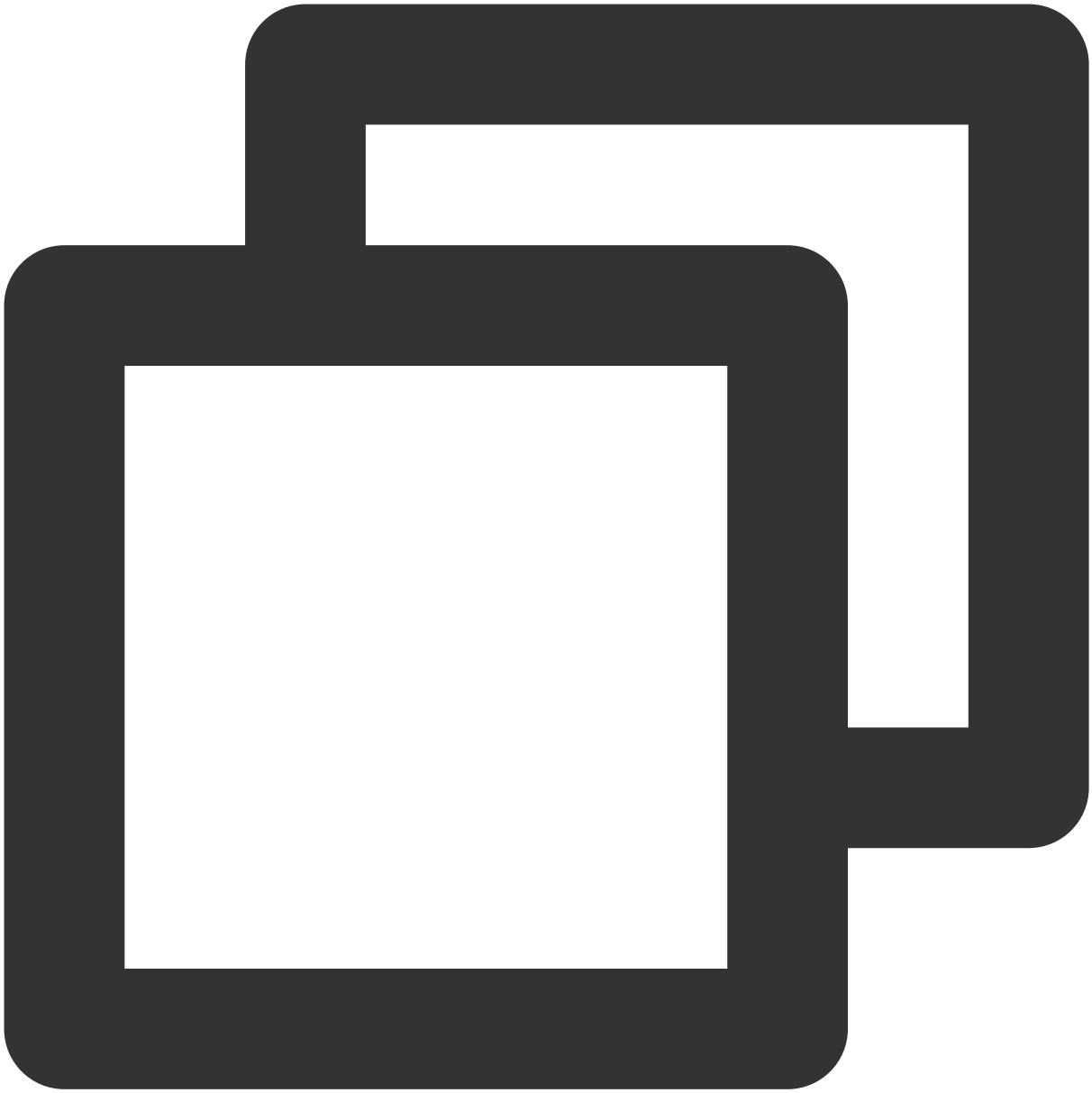


```
let onMessageReadReceiptReceived = function(event) {
  // event.data - 存储消息已读回执信息的数组
  const readReceiptInfoList = event.data;
  readReceiptInfoList.forEach((item) => {
    const { groupID, userID, messageID, readCount, unreadCount, isPeerRead } = item
    // messageID - 消息 ID
    // userID - C2C 消息接收方
    // isPeerRead - C2C 消息对端是否已读
    // groupID - 群组 ID
```

```
    // readCount - 群消息已读人数
    // unreadCount - 群消息未读人数
    const message = chat.findMessage(messageID);
    if (message) {
     if (message.conversationType === TencentCloudChat.TYPES.CONV_C2C) {
       if (message.readReceiptInfo.isPeerRead === true) {
         // 对端已读
       }
     } else if (message.conversationType === TencentCloudChat.TYPES.CONV_GROUP) {
      if (message.readReceiptInfo.unreadCount === 0) {
        // 全部已读
      } else {
        // message.readReceiptInfo.readCount - 消息最新的已读数
        // 如果想要查询哪些群成员已读了此消息，请使用 [getGroupMessageReadMemberList] 接口
      }
     }
    }
  });
}
chat.on(TencentCloudChat.EVENT.MESSAGE_READ_RECEIPT_RECEIVED, onMessageReadReceiptR
```

MESSAGE_EXTENSIONS_UPDATED

SDK 收到消息扩展更新通知，调用 setMessageExtensions 设置成功后，自己和对端用户（C2C）或群组成员（Group）都会收到此事件。

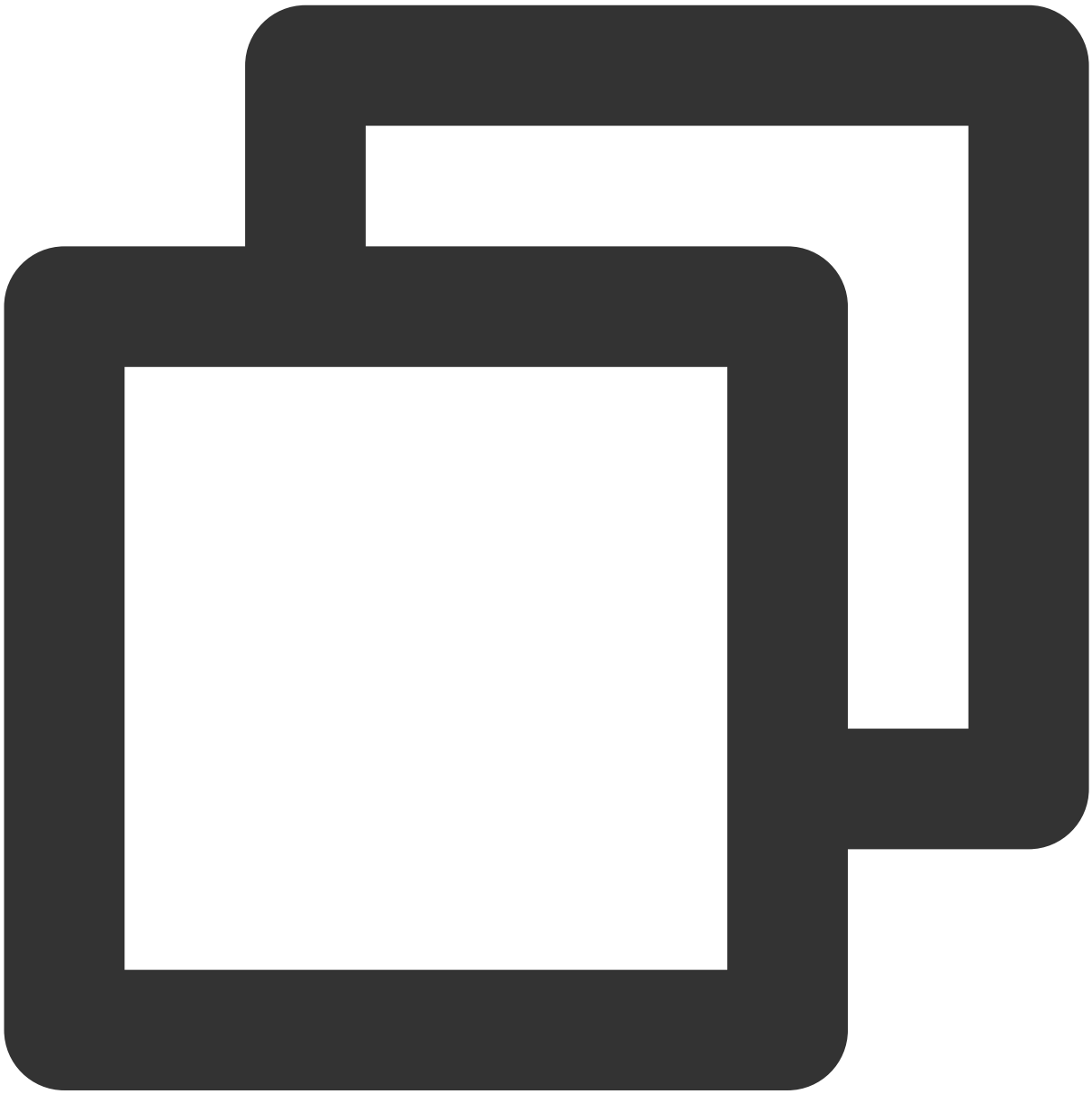


```
let onMessageExtensionsUpdated = function(event) {
  const { messageID, extensions } = event.data;
  // messageID - 消息 ID
  // extensions - 消息扩展信息
  extensions.forEach((item) => {
   const { key, value } = item;
   // key - 消息扩展 key
   // value - 消息扩展 key 对应的 value 值
  });
};
chat.on(TencentCloudChat.EVENT.MESSAGE_EXTENSIONS_UPDATED, onMessageExtensionsUpdat
```

版权所有：腾讯云计算（北京）有限责任公司

MESSAGE_EXTENSIONS_DELETED

SDK 收到消息扩展删除通知，调用 deleteMessageExtensions 删除成功后，自己和对端用户（C2C）或群组成员（Group）都会收到此事件。

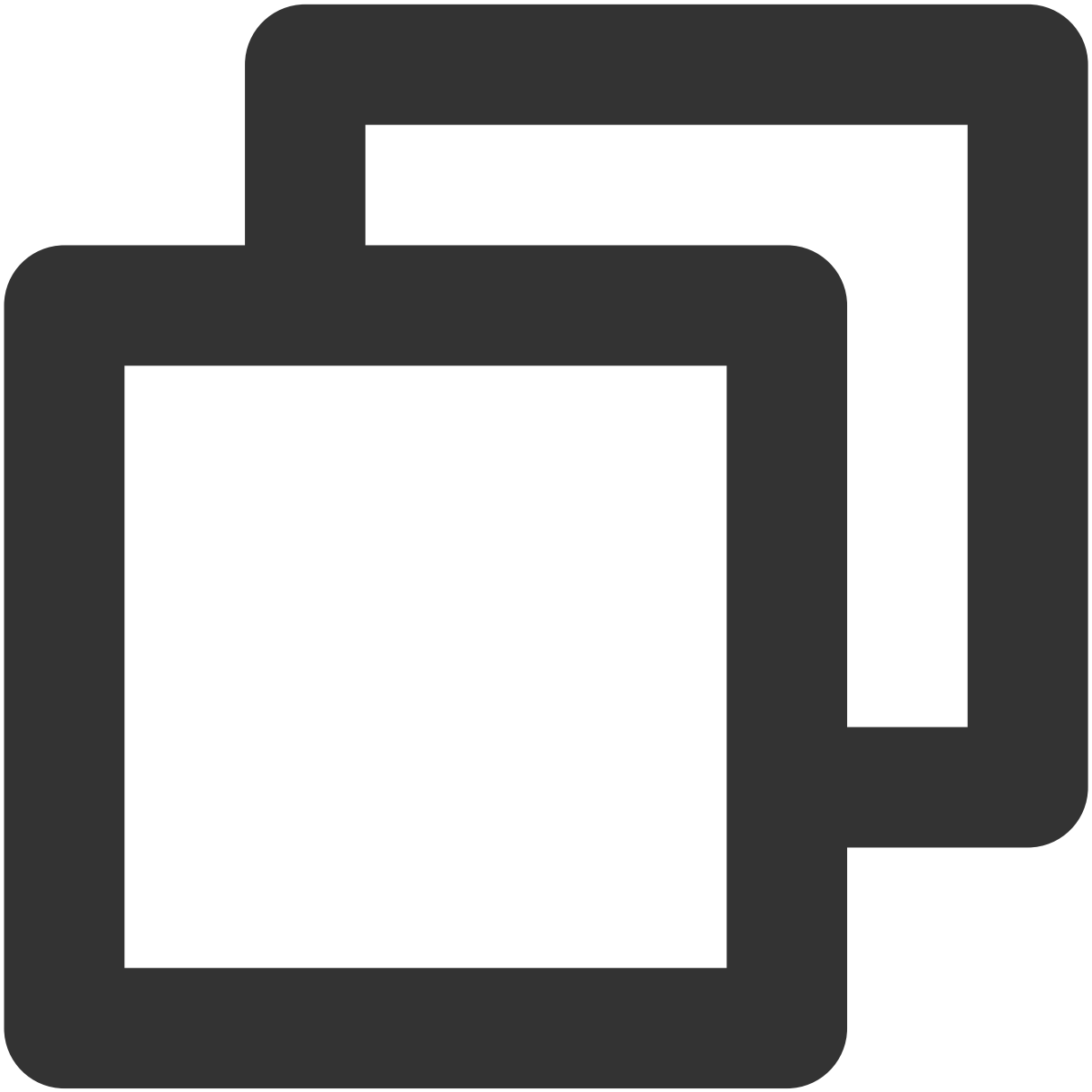


```
let onMessageExtensionsDeleted = function(event) {
  const { messageID, keyList } = event.data;
  // messageID - 消息 ID
  // keyList - 被删除的消息扩展 key 列表
  keyList.forEach((key) => {
```

```
    // console.log(key)
  });
};
chat.on(TencentCloudChat.EVENT.MESSAGE_EXTENSIONS_DELETED, onMessageExtensionsDelet
```

CONVERSATION_LIST_UPDATED

会话列表更新，event.data 是包含 Conversation 对象的数组。



```
let onConversationListUpdated = function(event) {
  console.log(event.data); // 包含 Conversation 实例的数组
```

版权所有：腾讯云计算（北京）有限责任公司

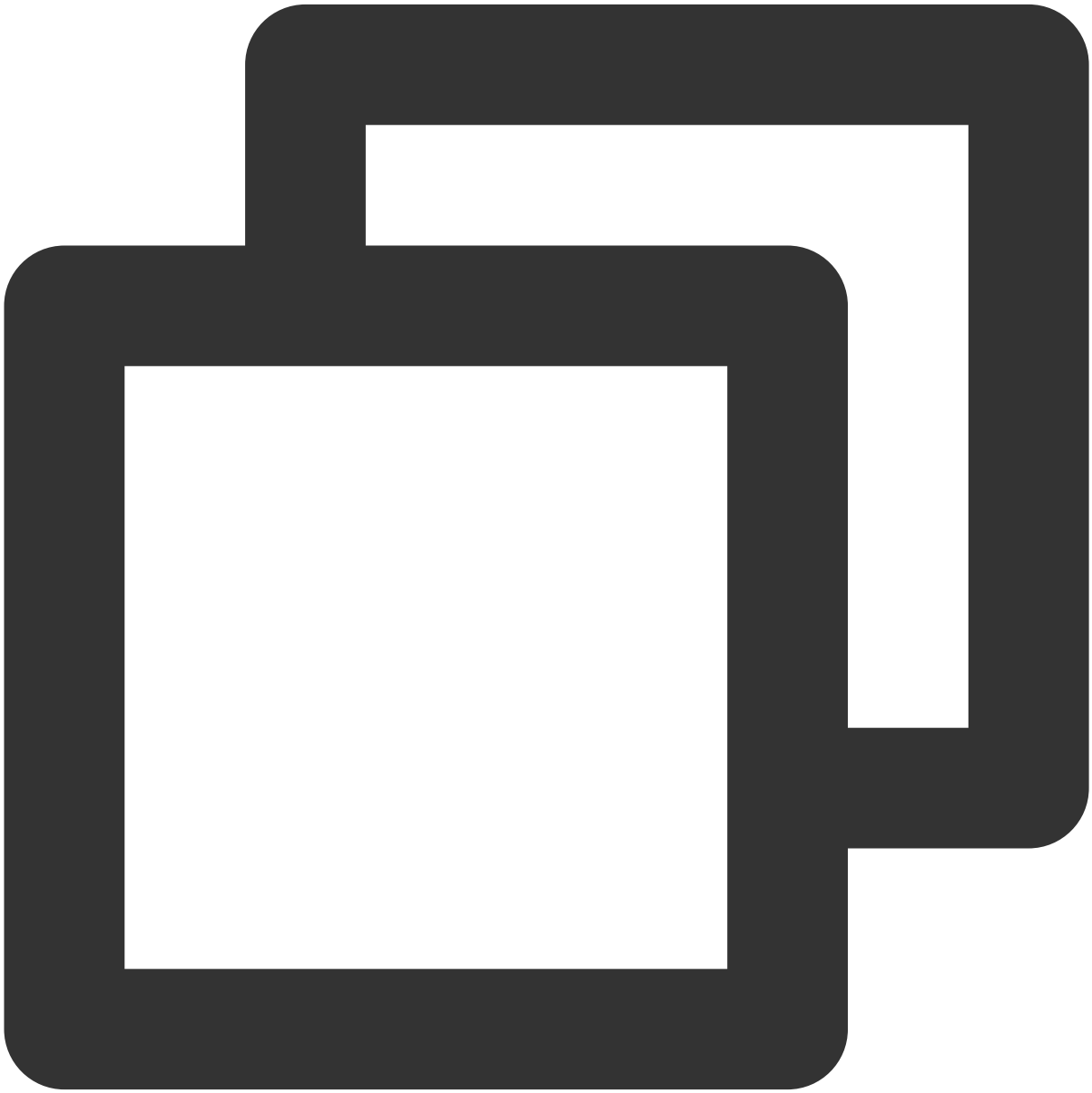
```
};
chat.on(TencentCloudChat.EVENT.CONVERSATION_LIST_UPDATED, onConversationListUpdated
```

TOTAL_UNREAD_MESSAGE_COUNT_UPDATED

会话未读总数更新，event.data 是当前单聊和群聊会话的未读总数。

注意：

1. 未读总数会减去设置为免打扰的会话的未读数。
2. 未读总数不计算系统会话的未读数。

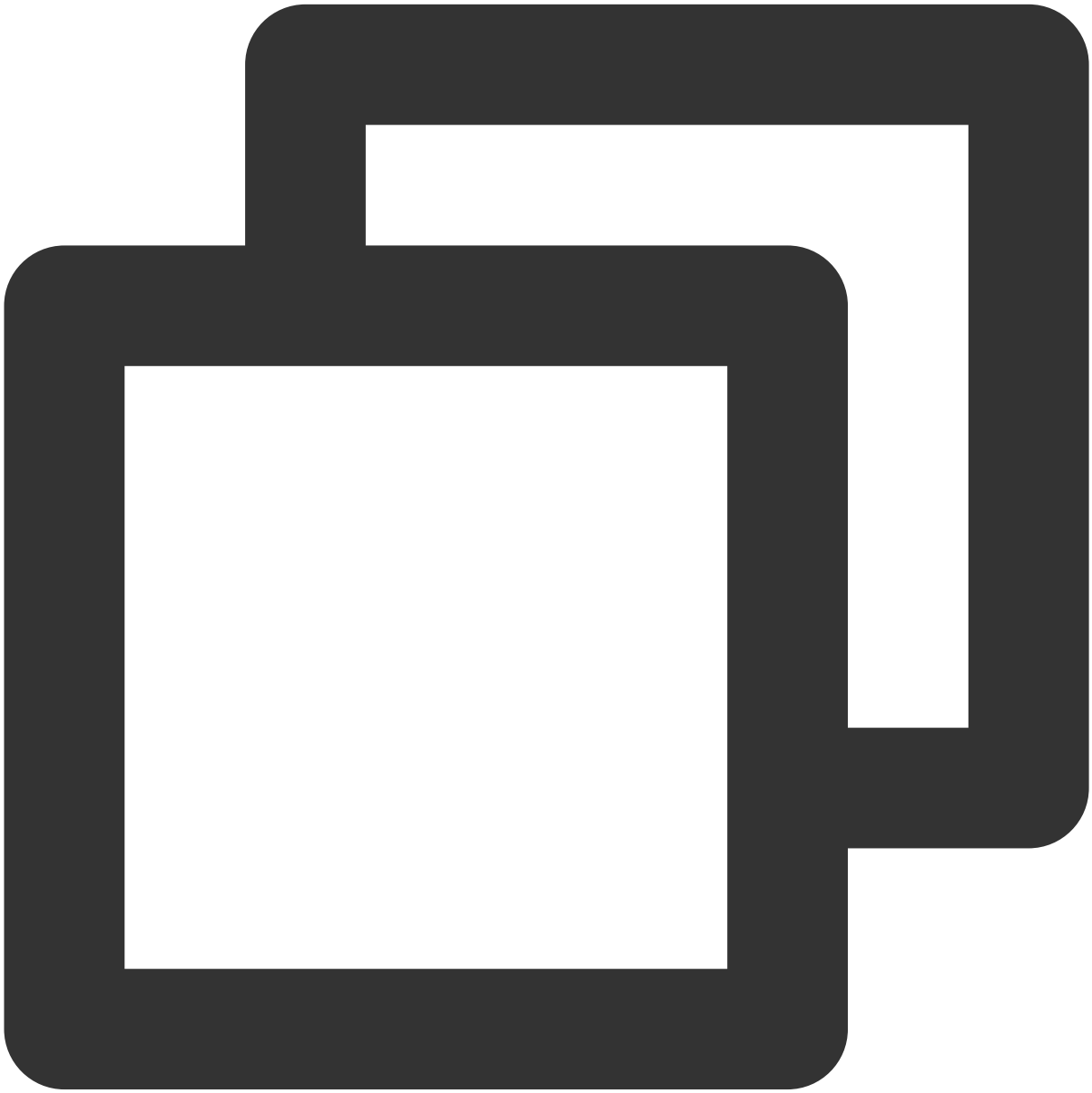


```
let onTotalUnreadMessageCountUpdated = function(event) {
```

```
  console.log(event.data); // 当前单聊和群聊会话的未读总数
};
chat.on(TencentCloudChat.EVENT.TOTAL_UNREAD_MESSAGE_COUNT_UPDATED, onTotalUnreadMes
```

CONVERSATION_GROUP_LIST_UPDATED

会话分组更新时触发（如创建会话分组、删除会话分组、重命名会话分组）。

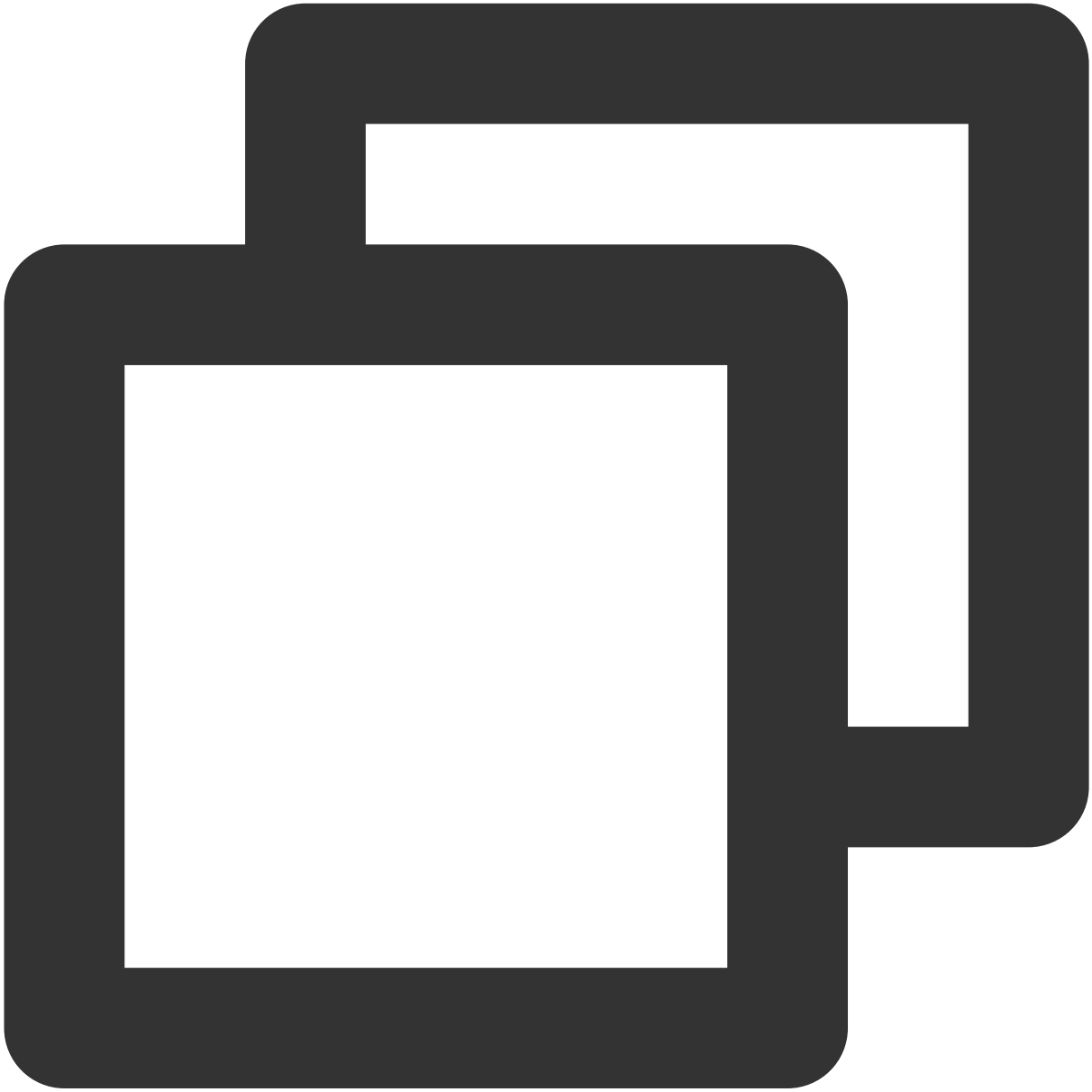


```
let onConversationGroupListUpdated = function(event) {
  console.log(event.data); // 全量的会话分组名列表
}
```

```
chat.on(TencentCloudChat.EVENT.CONVERSATION_GROUP_LIST_UPDATED, onConversationGroup
```

CONVERSATION_IN_GROUP_UPDATED

会话分组内的会话更新时触发（如添加会话到分组，或者从分组内删除会话）。

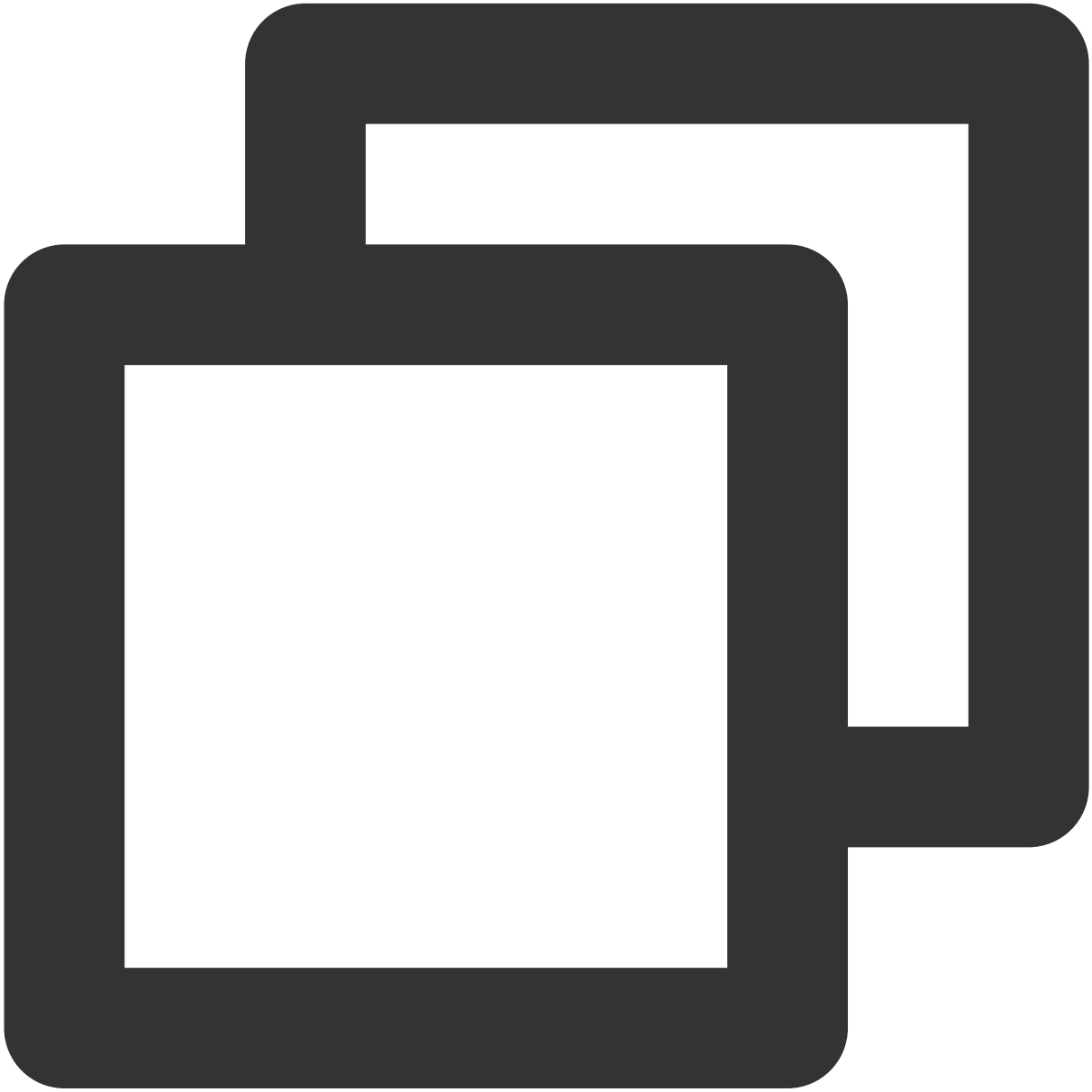


```
let onConversationInGroupUpdated = function(event) {
  const { groupName, conversationList }  = event.data;
  // groupName - 会话分组名
  // conversationList - 分组内的会话列表
}
```

```
chat.on(TencentCloudChat.EVENT.CONVERSATION_IN_GROUP_UPDATED, onConversationInGroup
```

GROUP_LIST_UPDATED

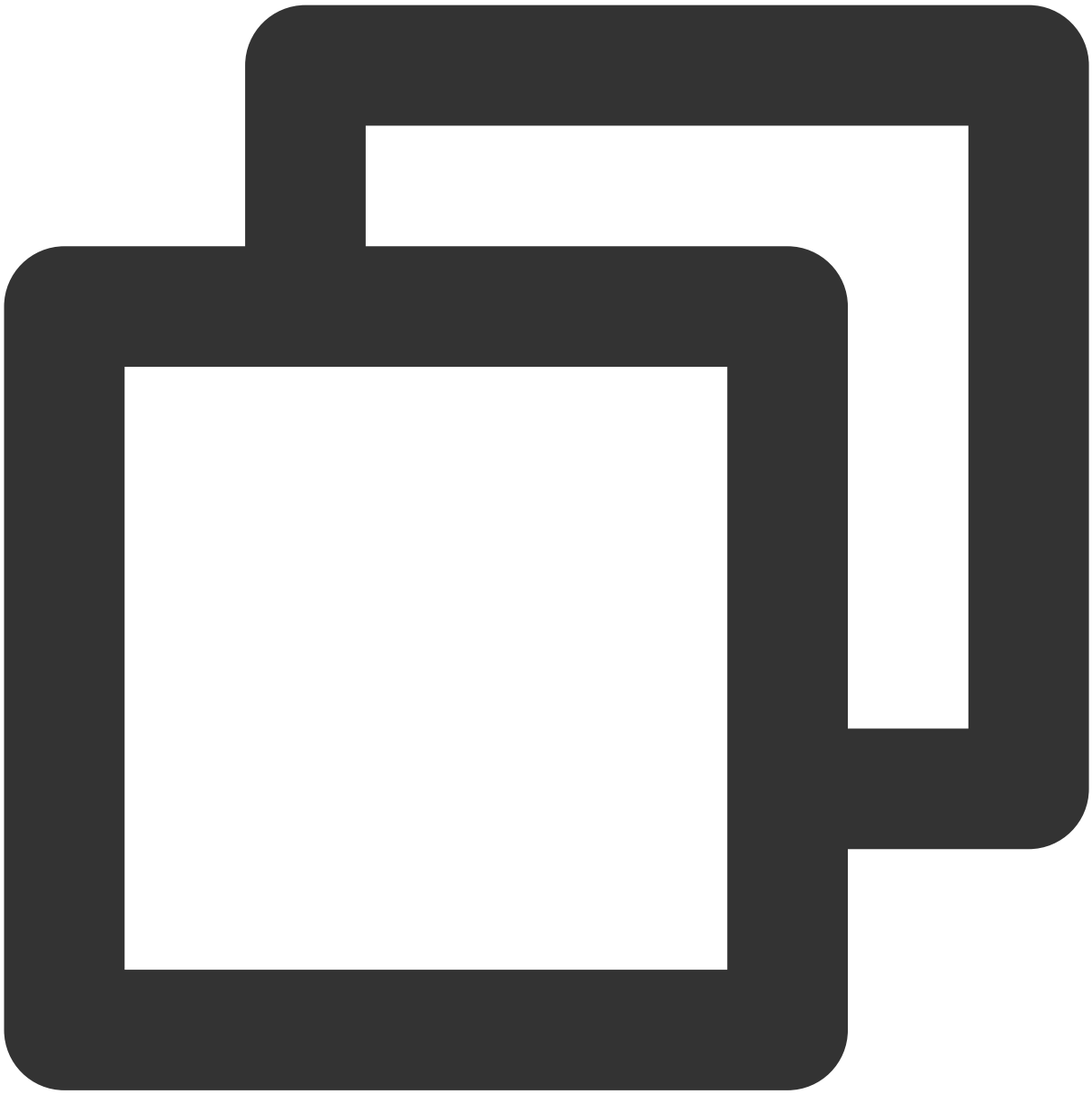
SDK 群组列表更新时触发，接入侧可通过遍历 event.data 获取群组列表数据并渲染到页面。



```
let onGroupListUpdated = function(event) {
   console.log(event.data);// 包含 Group 实例的数组
};
chat.on(TencentCloudChat.EVENT.GROUP_LIST_UPDATED, onGroupListUpdated);
```

GROUP_ATTRIBUTES_UPDATED

群属性更新时触发，接入侧可通过 event.data 获取到更新后的群属性数据。



```
let onGroupAttributesUpdated = function(event) {
   const groupID = event.data.groupID // 群组ID
   const groupAttributes = event.data.groupAttributes // 更新后的群属性
   console.log(event.data);
};
chat.on(TencentCloudChat.EVENT.GROUP_ATTRIBUTES_UPDATED, onGroupAttributesUpdated);
```

GROUP_COUNTER_UPDATED

SDK 收到群计数器更新通知，自己和其他群组成员都会收到此事件。

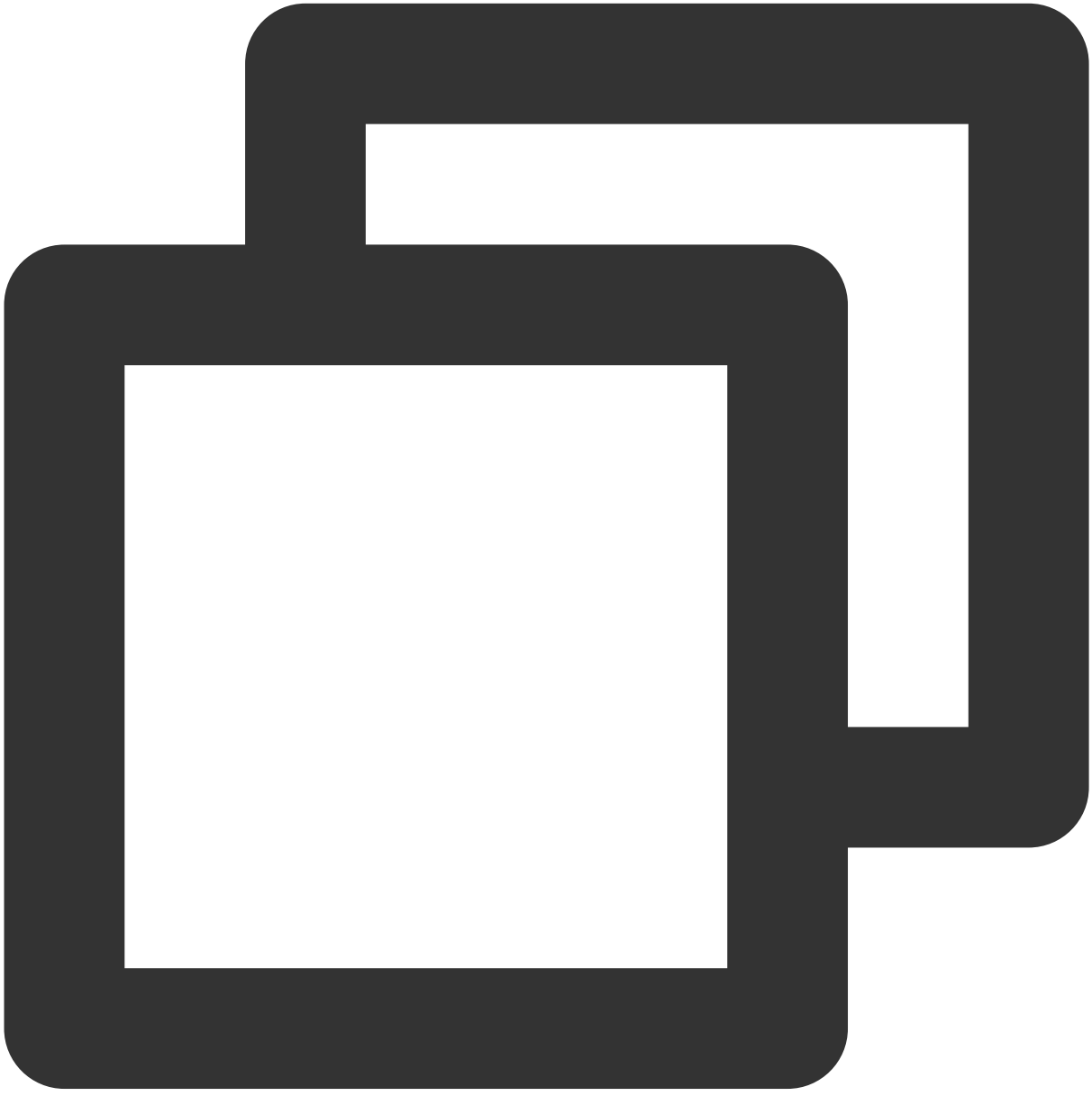
注意：

您调用以下接口操作成功后均会触发该事件：

setGroupCounters 设置群计数器。

increaseGroupCounter 递增群计数器。

decreaseGroupCounter 递减群计数器。

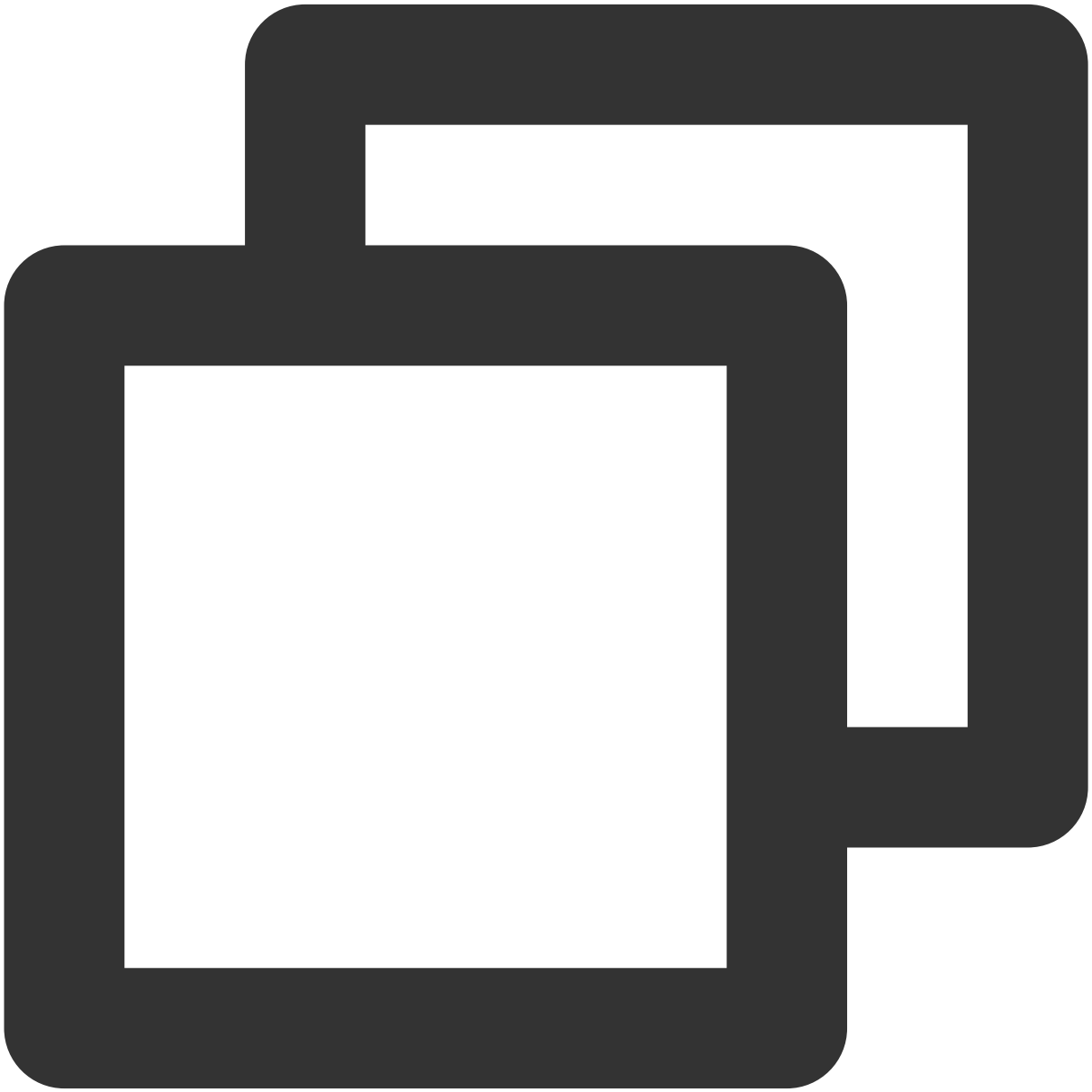


```
let onGroupCounterUpdated = function(event) {
  const { groupID, key, value } = event.data;
  // groupID - 群组 ID
```

```
  // key - 群计数器 key
  // value - 群计数器 key 对应的 value
};
chat.on(TencentCloudChat.EVENT.GROUP_COUNTER_UPDATED, onGroupCounterUpdated);
```

TOPIC_CREATED

创建话题时触发。

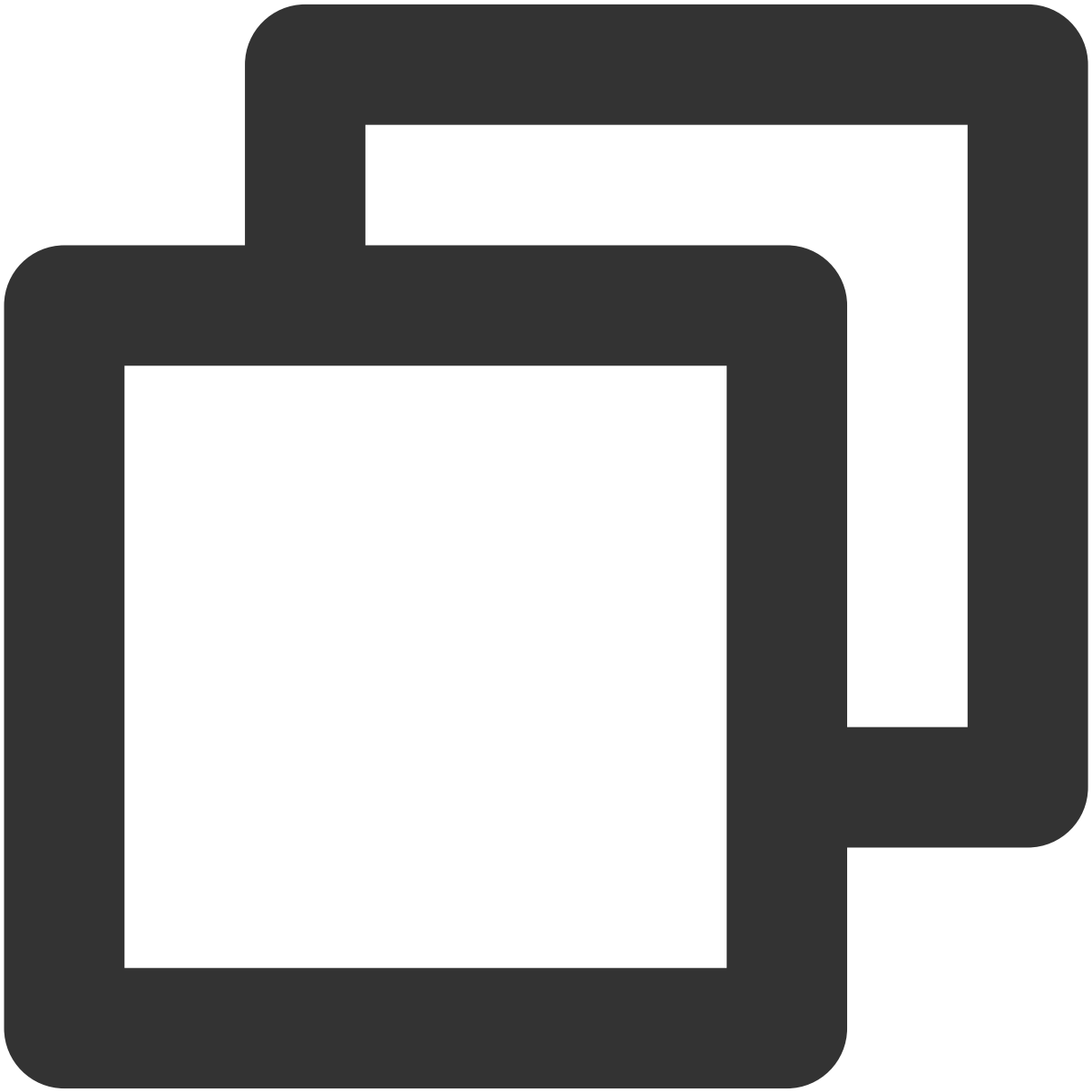


```
let onTopicCreated = function(event) {
   const groupID = event.data.groupID // 话题所属社群 ID
```

```
    const topicID = event.data.topicID // 话题 ID
    console.log(event.data);
};
chat.on(TencentCloudChat.EVENT.TOPIC_CREATED, onTopicCreated);
```

TOPIC_DELETED

删除话题时触发。

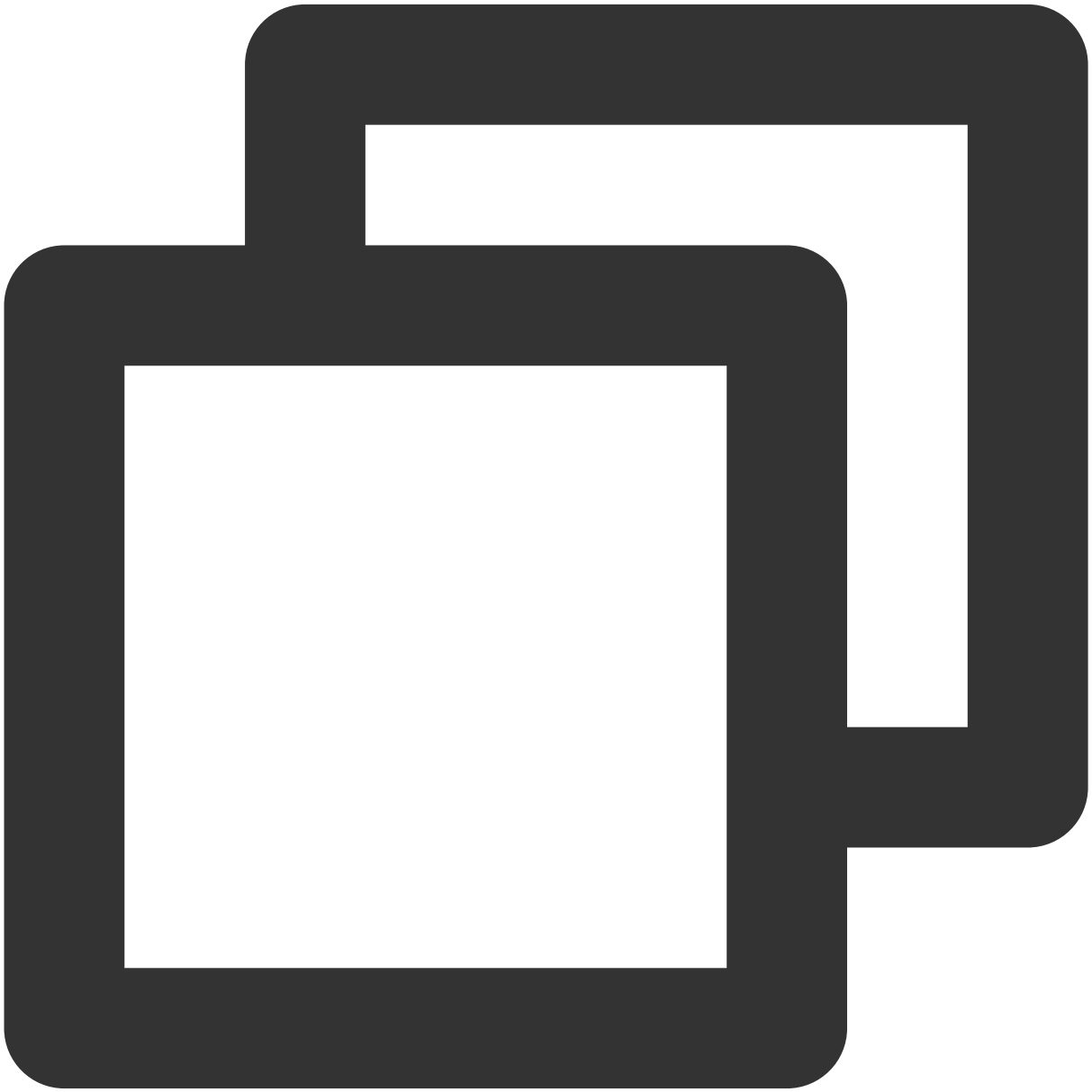


```
let onTopicDeleted = function(event) {
    const groupID = event.data.groupID // 话题所属社群 ID
```

```
    const topicIDList = event.data.topicIDList // 删除的话题 ID 列表
    console.log(event.data);
};
chat.on(TencentCloudChat.EVENT.TOPIC_DELETED, onTopicDeleted);
```

TOPIC_UPDATED

话题资料更新时触发。

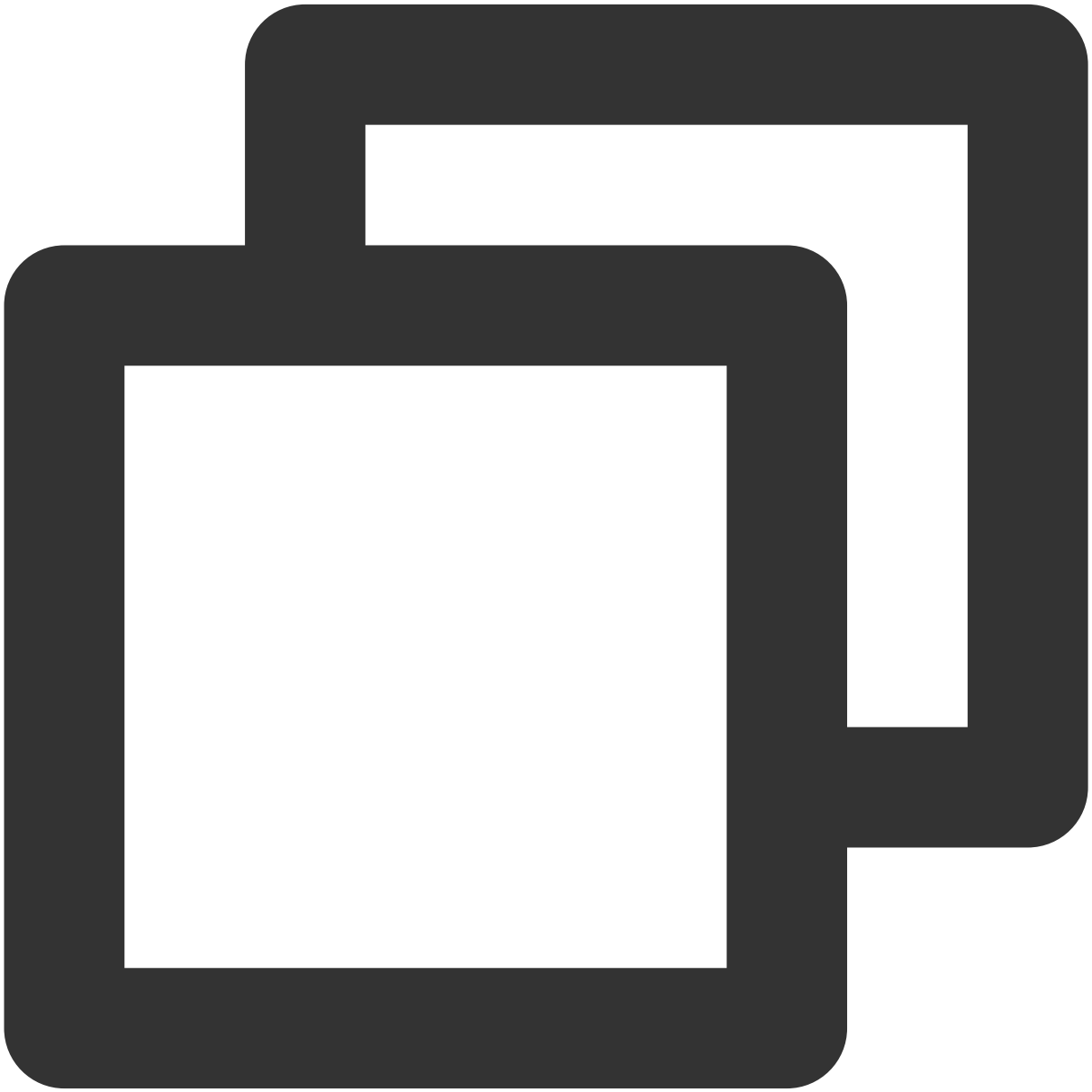


```
let onTopicUpdated = function(event) {
    const groupID = event.data.groupID // 话题所属社群 ID
```

```
    const topic = event.data.topic // 话题资料
    console.log(event.data);
};
chat.on(TencentCloudChat.EVENT.TOPIC_UPDATED, onTopicUpdated);
```

PROFILE_UPDATED

自己或好友的资料发生变更时触发，event.data 是包含 Profile 对象的数组。

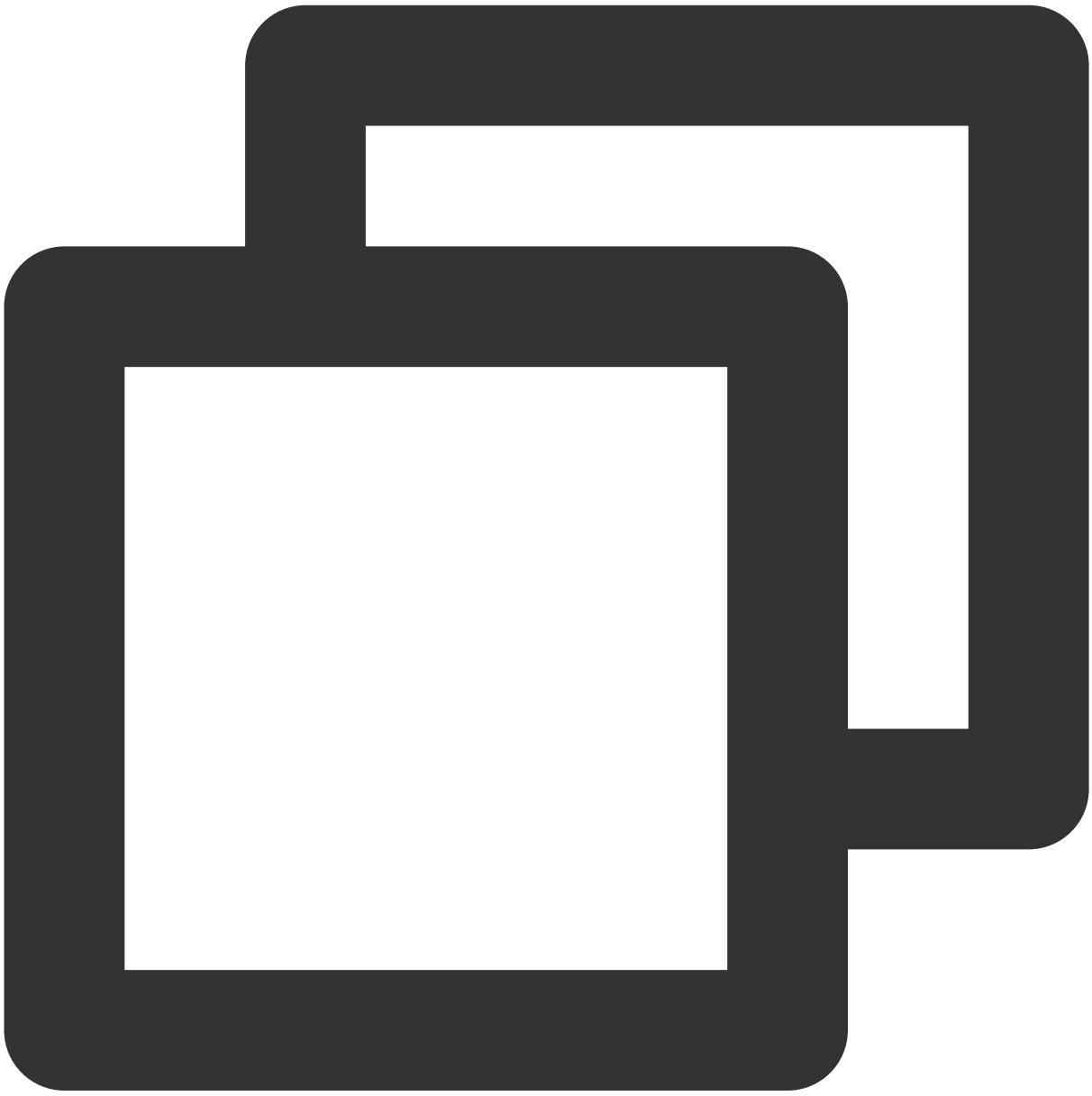


```
let onProfileUpdated = function(event) {
  console.log(event.data); // 包含 Profile 对象的数组
```

```
};
chat.on(TencentCloudChat.EVENT.PROFILE_UPDATED, onProfileUpdated);
```

USER_STATUS_UPDATED

已订阅用户或好友的状态变更（在线状态或自定义状态）时触发。

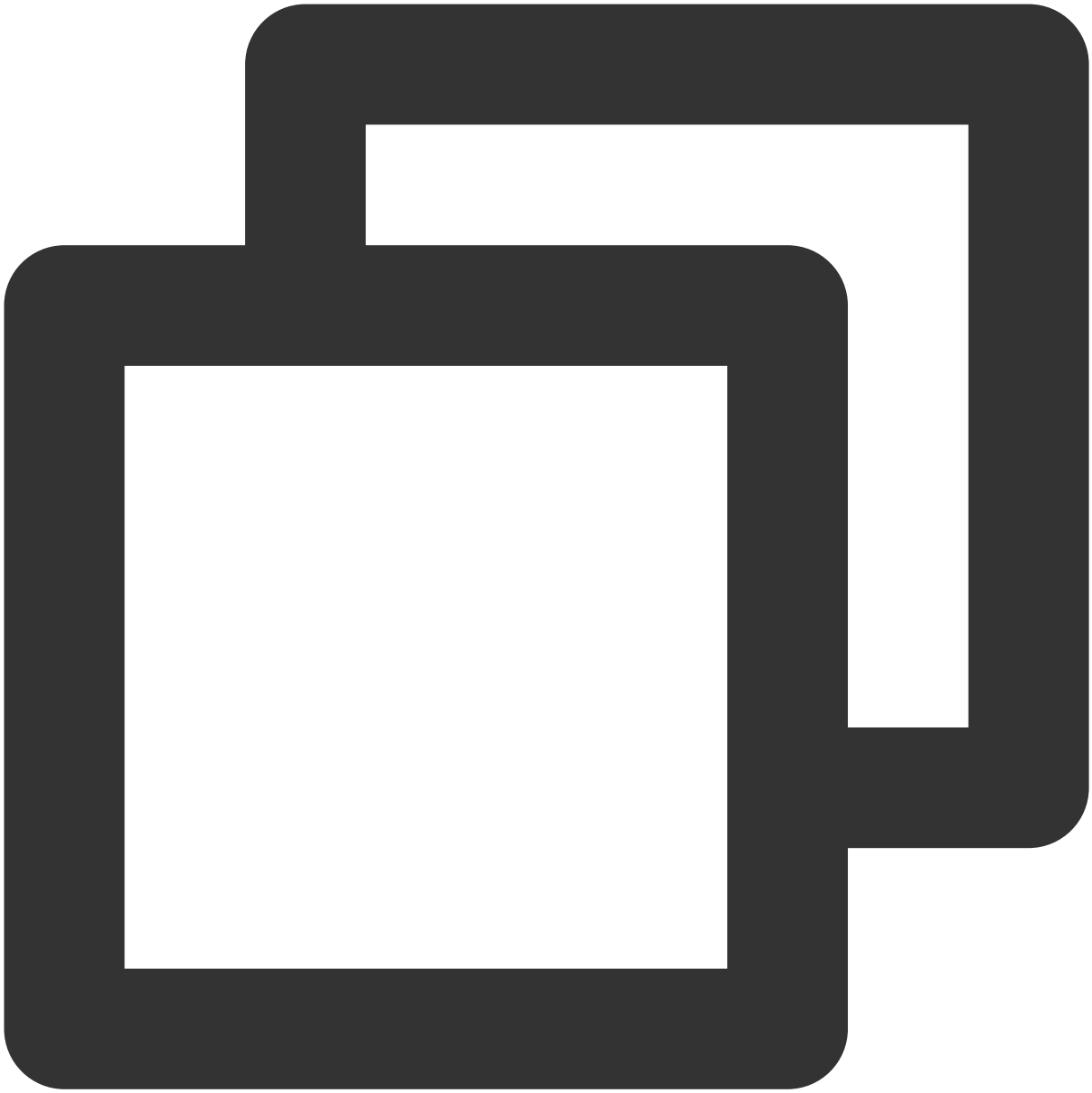


```
let onUserStatusUpdated = function(event) {
   console.log(event.data);
   const userStatusList = event.data;
   userStatusList.forEach((item) => {
```

```
      const { userID, statusType, customStatus } = item;
      // userID - 用户 ID
      // statusType - 用户状态，枚举值及说明如下：
      // TencentCloudChat.TYPES.USER_STATUS_UNKNOWN - 未知
      // TencentCloudChat.TYPES.USER_STATUS_ONLINE - 在线
      // TencentCloudChat.TYPES.USER_STATUS_OFFLINE - 离线
      // TencentCloudChat.TYPES.USER_STATUS_UNLOGINED - 未登录
      // customStatus - 用户自定义状态
    })
};
chat.on(TencentCloudChat.EVENT.USER_STATUS_UPDATED, onUserStatusUpdated);
```

BLACKLIST_UPDATED

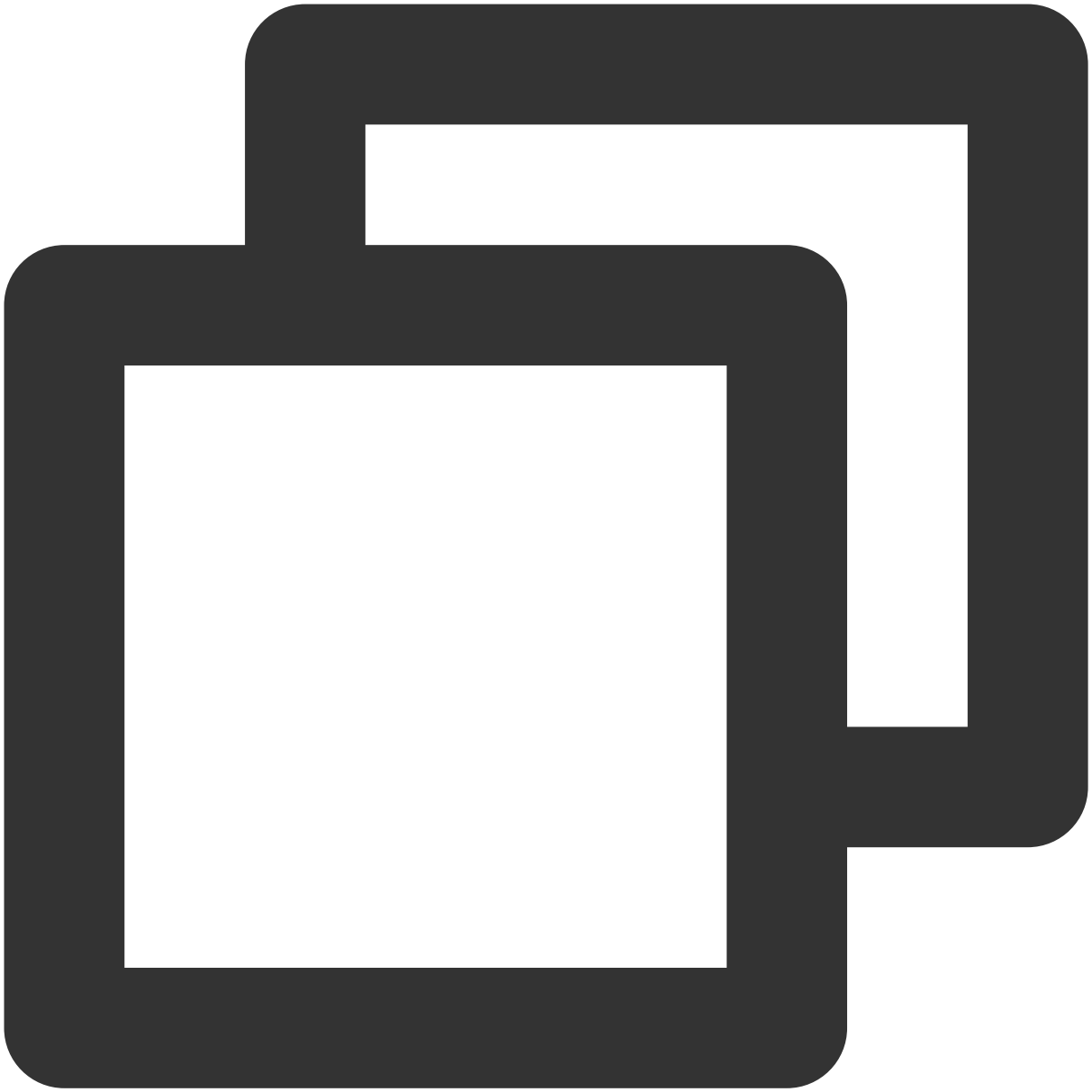
SDK 黑名单列表更新时触发。



```
let onBlacklistUpdated = function(event) {
  console.log(event.data); // 我的黑名单列表，结构为包含用户 userID 的数组
};
chat.on(TencentCloudChat.EVENT.BLACKLIST_UPDATED, onBlacklistUpdated);
```

FRIEND_LIST_UPDATED

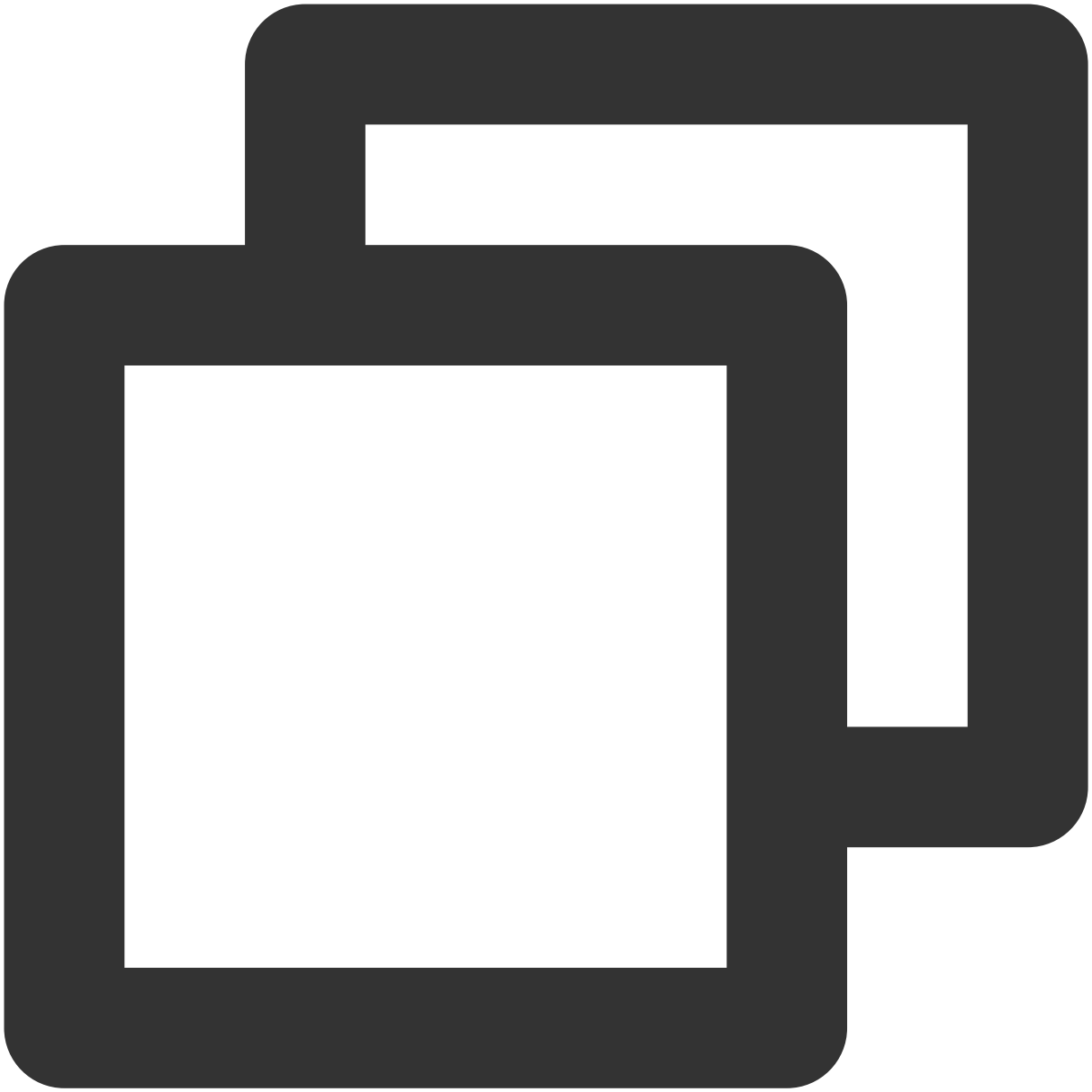
好友列表更新时触发。



```
let onFriendListUpdated = function(event) {
  console.log(event.data);
}
chat.on(TencentCloudChat.EVENT.FRIEND_LIST_UPDATED, onFriendListUpdated);
```

FRIEND_GROUP_LIST_UPDATED

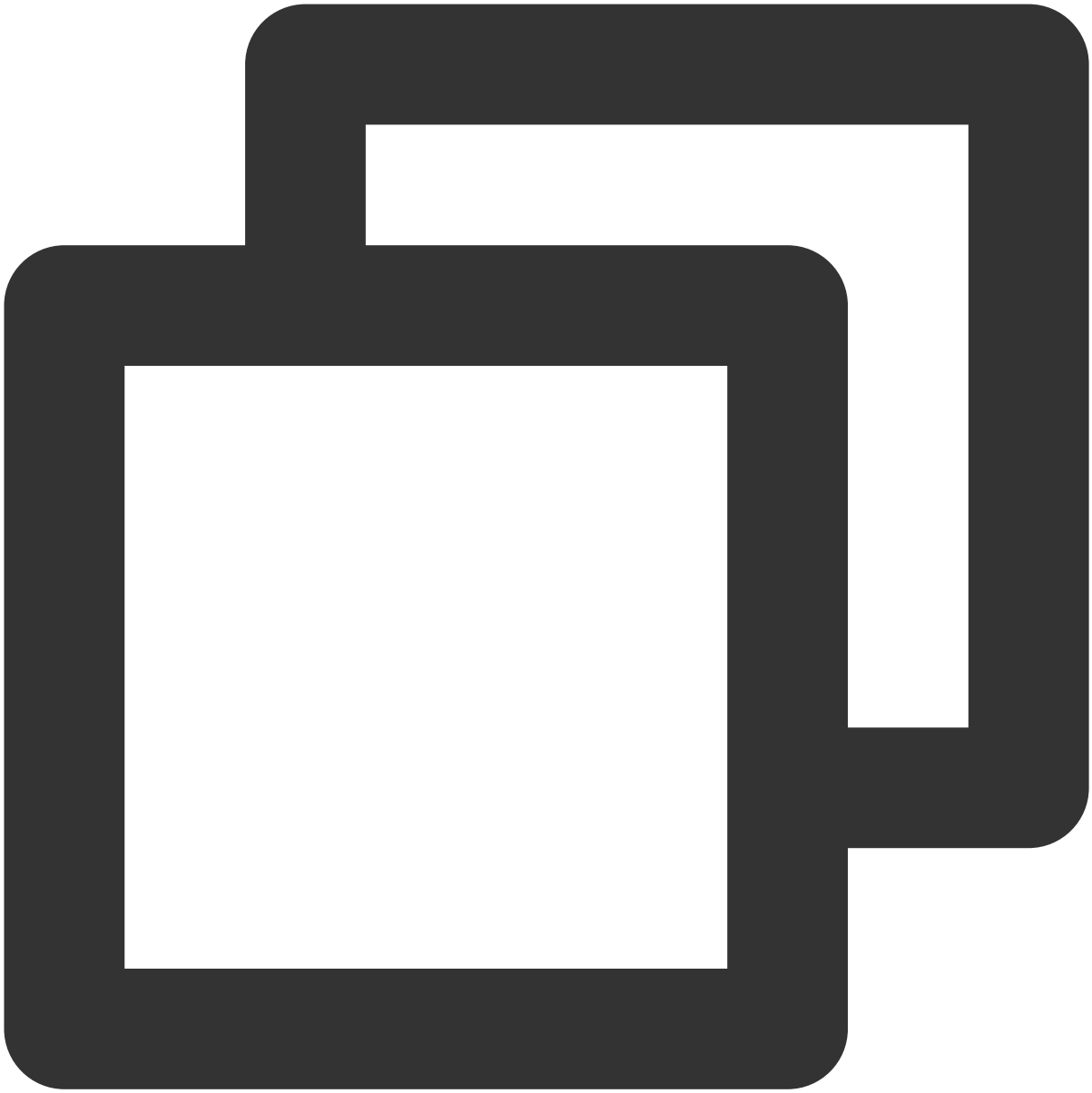
好友分组列表更新时触发。



```
let onFriendGroupListUpdated = function(event) {
  console.log(event.data);
}
chat.on(TencentCloudChat.EVENT.FRIEND_GROUP_LIST_UPDATED, onFriendGroupListUpdated)
```

FRIEND_APPLICATION_LIST_UPDATED

SDK 好友申请列表更新时触发。

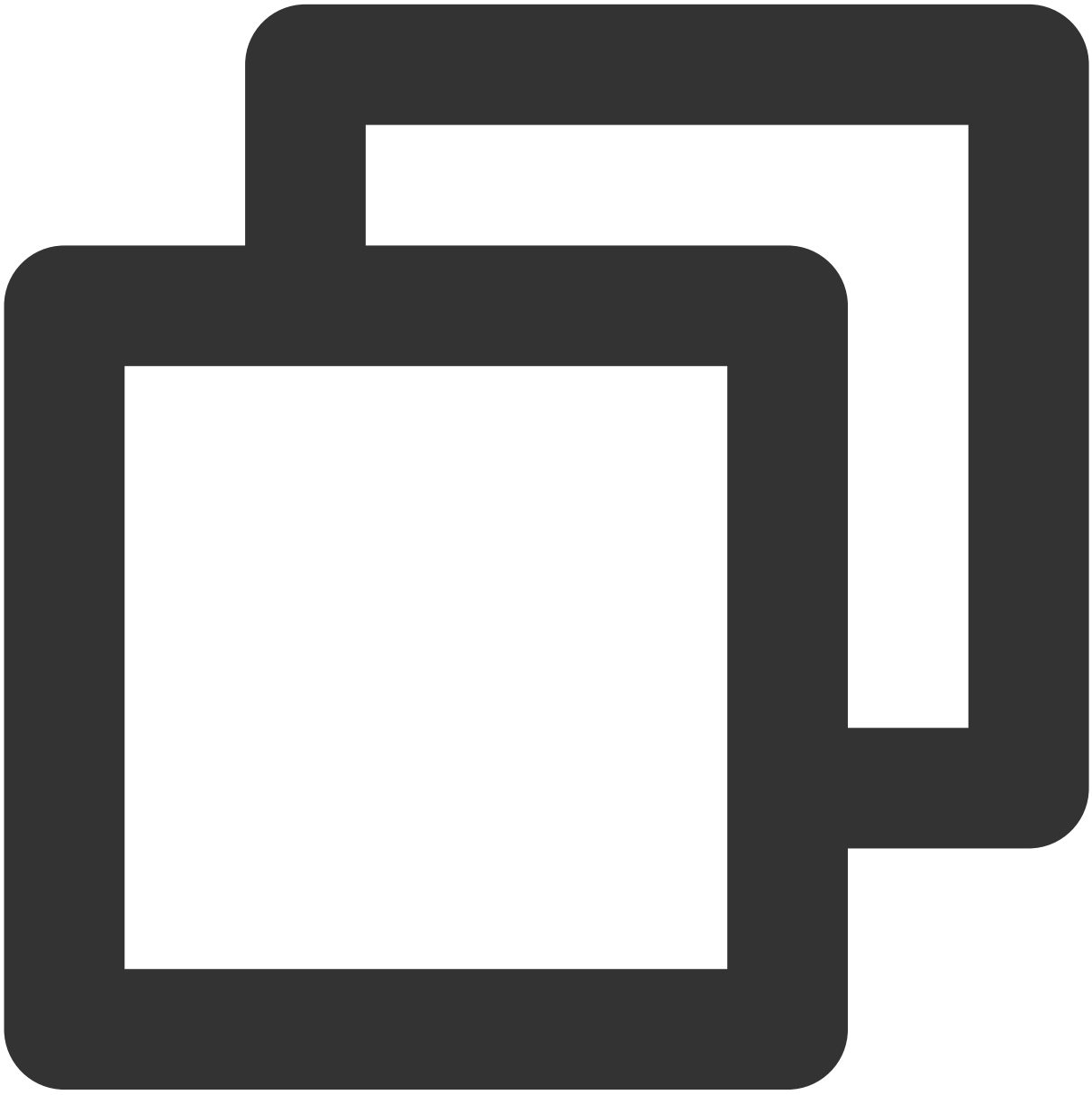


```
let onFriendApplicationListUpdated = function(event) {
  // friendApplicationList - 好友申请列表 - [FriendApplication]
  // unreadCount - 好友申请的未读数
  const { friendApplicationList, unreadCount } = event.data;
  // 发送给我的好友申请（即别人申请加我为好友）
  const applicationSentToMe = friendApplicationList.filter((friendApplication) => {
    return friendApplication.type === TencentCloudChat.TYPES.SNS_APPLICATION_SENT_T
  });
  // 我发送出去的好友申请（即我申请加别人为好友）
  const applicationSentByMe = friendApplicationList.filter((friendApplication) => {
    return friendApplication.type === TencentCloudChat.TYPES.SNS_APPLICATION_SENT_B
```

```
  });
};
chat.on(TencentCloudChat.EVENT.FRIEND_APPLICATION_LIST_UPDATED, onFriendApplication
```

KICKED_OUT

用户被踢下线时触发。

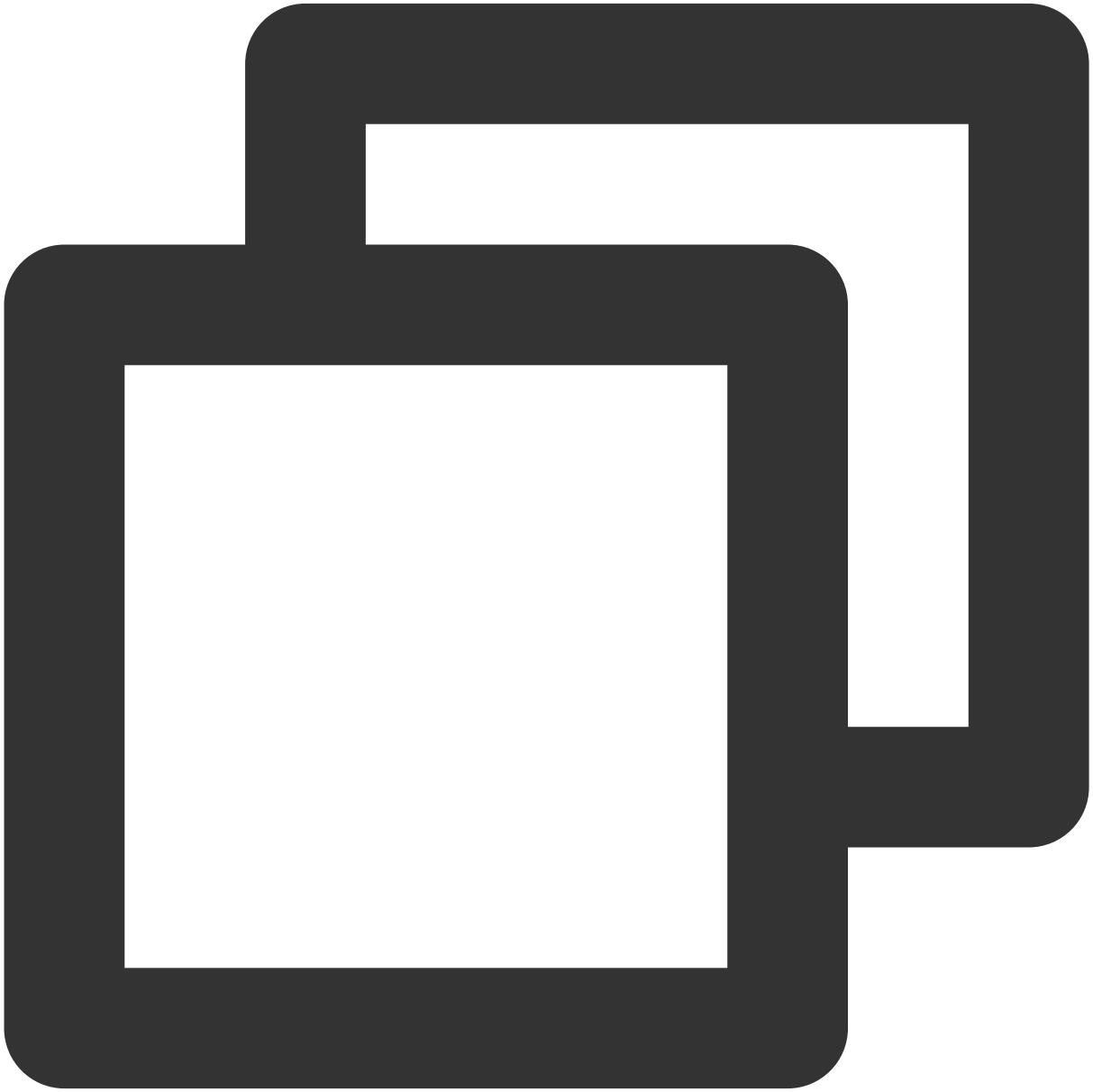


```
let onKickedOut = function(event) {
  console.log(event.data.type);
  // TencentCloudChat.TYPES.KICKED_OUT_MULT_ACCOUNT(Web端，同一账号，多页面登录被踢)
```

```
  // TencentCloudChat.TYPES.KICKED_OUT_MULT_DEVICE(同一账号，多端登录被踢)
  // TencentCloudChat.TYPES.KICKED_OUT_USERSIG_EXPIRED(签名过期)
  // TencentCloudChat.TYPES.KICKED_OUT_REST_API(REST API kick 接口踢出)
};
chat.on(TencentCloudChat.EVENT.KICKED_OUT, onKickedOut);
```

NET_STATE_CHANGE

网络状态发生改变。

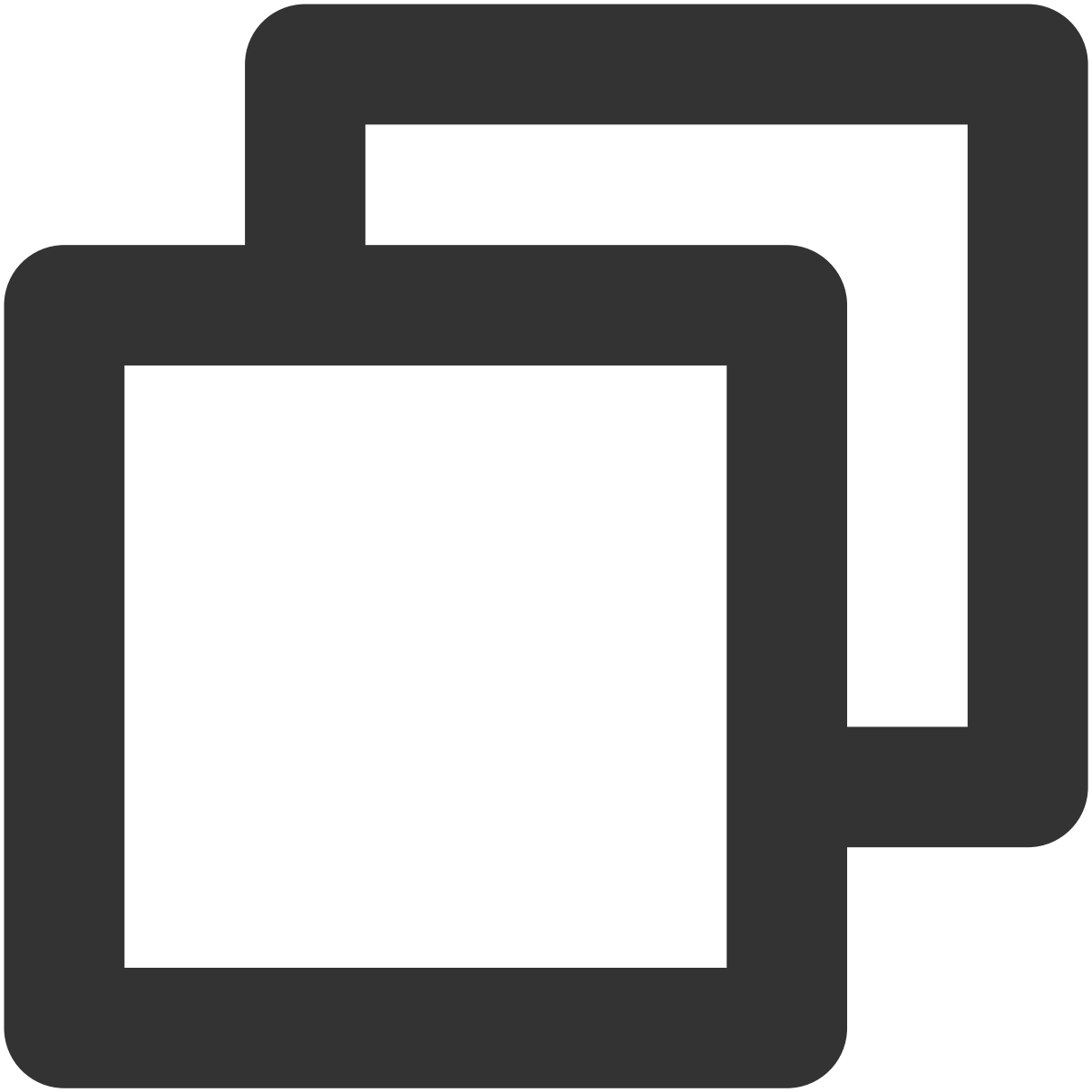


```
let onNetStateChange = function(event) {
```

```
    // event.data.state 当前网络状态，枚举值及说明如下：
    // TencentCloudChat.TYPES.NET_STATE_CONNECTED - 已接入网络
    // TencentCloudChat.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”
    // TencentCloudChat.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
};
chat.on(TencentCloudChat.EVENT.NET_STATE_CHANGE, onNetStateChange);
```

反初始化

销毁 SDK 实例。SDK 会先 logout，然后断开 WebSocket 长连接，并释放资源。



```
chat.destroy();
```

下一步，登录登出 。

Flutter

最近更新时间：2024-01-31 15:32:32

功能描述

在使用 IM SDK 的各项功能前，**必须**先进行初始化。
大多数场景下，在应用生命周期内，您只需要进行一次 IM SDK 初始化。

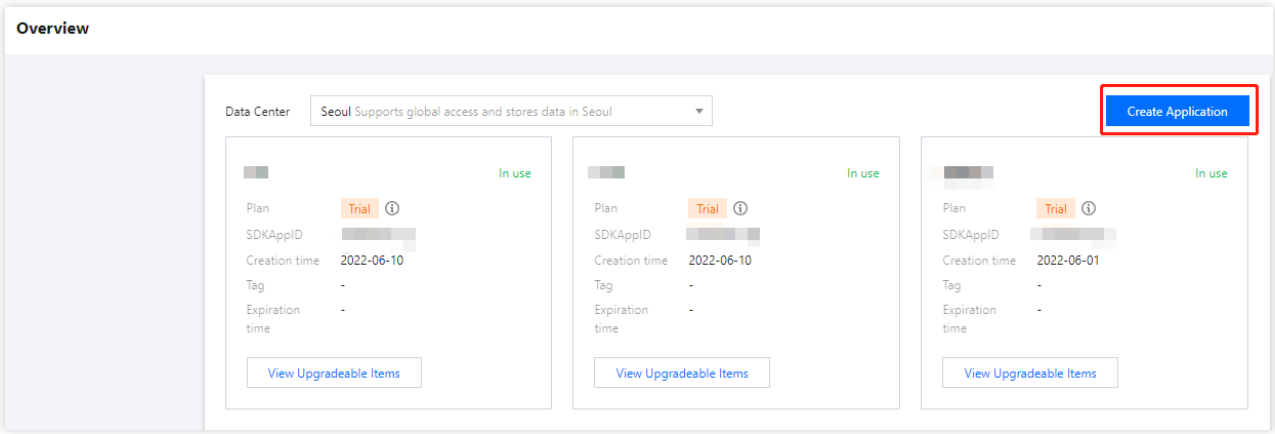
初始化

初始化 SDK 需要操作以下步骤：
1. 准备 SDKAppID。
2. 设置 LogLevelEnum。
3. 设置 SDK 事件监听器。
4. 调用 `initSDK` 初始化 SDK。

下文我们将依次为您详细讲解具体的步骤内容。

准备 SDKAppID

您必须拥有正确的 SDKAppID，才能进行初始化。
SDKAppID 是腾讯云 IM 服务用于区分客户帐号的唯一标识。我们建议每一个独立的 App 都申请一个新的 SDKAppID。不同 SDKAppID 之间的消息是天然隔离的，不能互通。
您可以在 即时通信 IM 控制台 查看所有的 SDKAppID，单击 **创建新应用** 按钮，可以创建新的 SDKAppID。



设置 LogLevelEnum

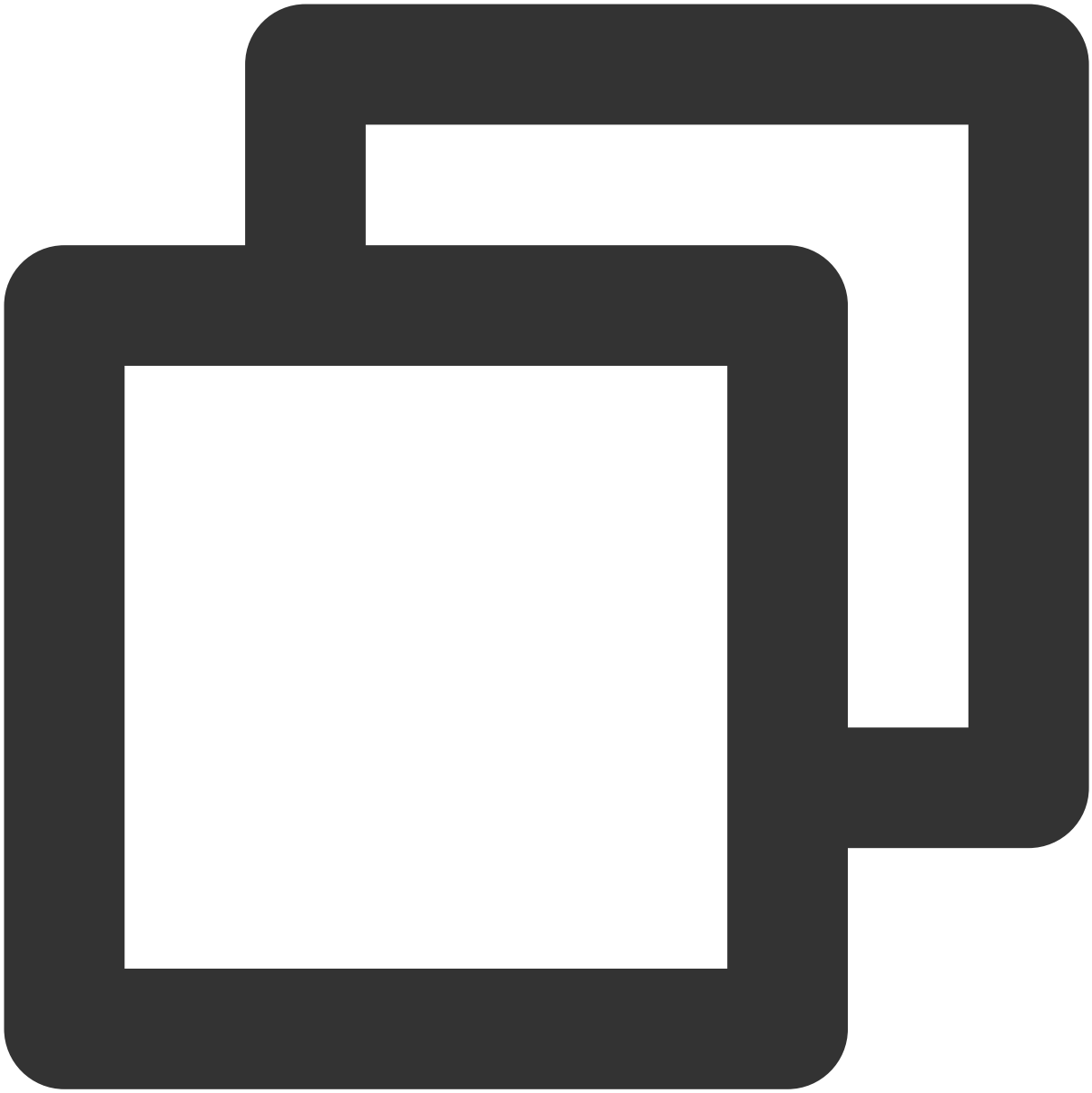
初始化 SDK 前，您需要初始化一个 `LogLevelEnum` (Dart) 对象。该对象用于对 SDK 进行日志级别设置。

设置日志级别

IM SDK 支持多种日志级别，如下表所示：

日志级别	LOG 输出量
LogLevelEnum.V2TIM_LOG_NONE	不输出任何 log
LogLevelEnum.V2TIM_LOG_DEBUG	输出 DEBUG，INFO，WARNING，ERROR 级别的 log （默认的日志级别）
LogLevelEnum.V2TIM_LOG_INFO	输出 INFO，WARNING，ERROR 级别的 log
LogLevelEnum.V2TIM_LOG_WARN	输出 WARNING，ERROR 级别的 log
LogLevelEnum.V2TIM_LOG_ERROR	输出 ERROR 级别的 log

SDK 日志存储规则如下：
IM SDK 本地日志默认保存 7 天；SDK 在初始化时，会自动清理 7 天之前的日志。
在 Android 平台上，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 4.7.1 版本开始，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
```

设置 SDK 事件监听器

SDK 初始化后，会通过 `V2TimSDKListener` 抛出一些事件，例如连接状态、登录票据过期等。

我们建议您在调用initSDK时传入 `V2TimSDKListener` (Dart) 接口添加 SDK 事件监听器，在对应回调中做一些逻辑处理。

`V2TimSDKListener` 相关回调如下表所示：

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	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帐号，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。

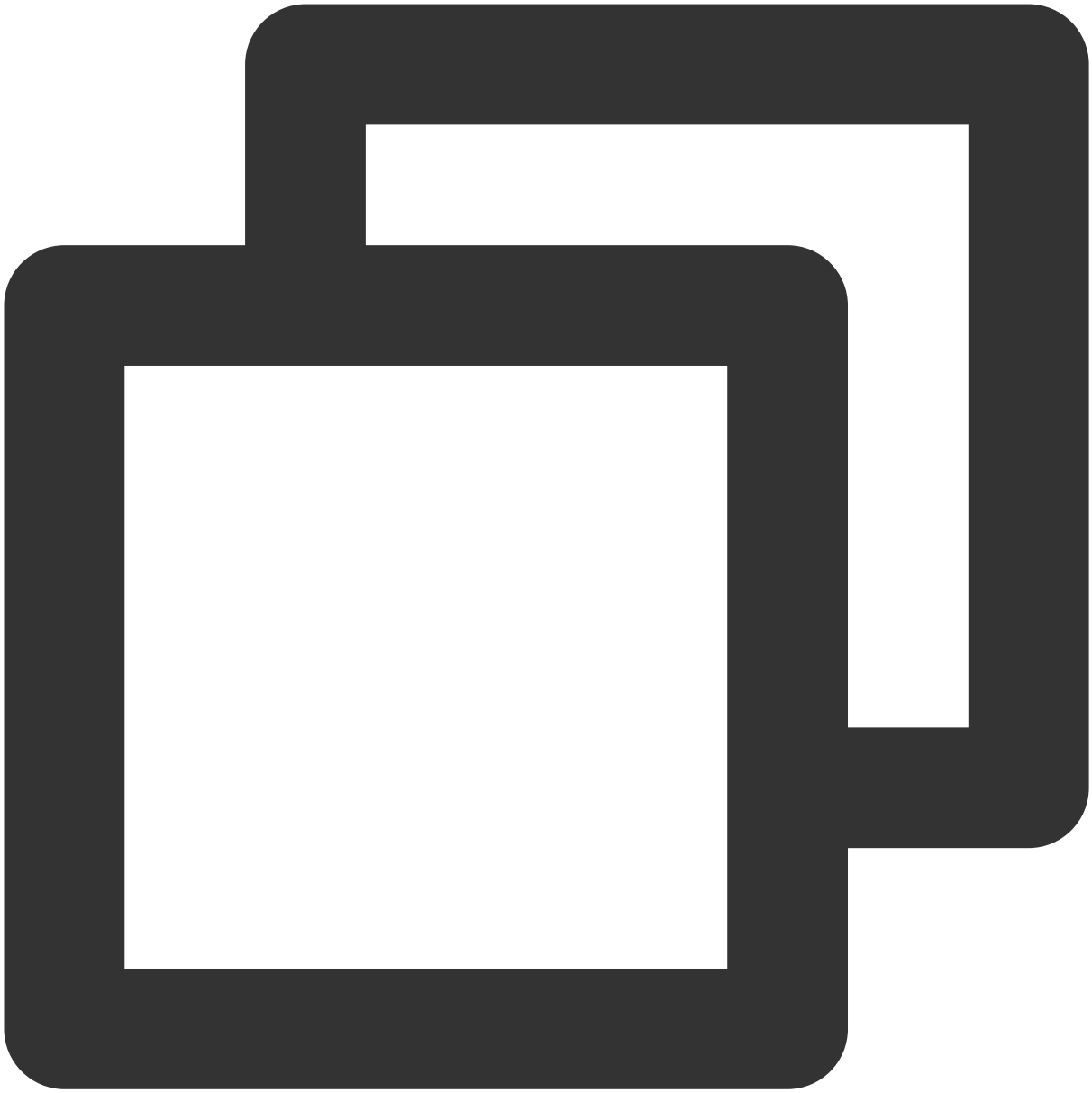
注意：

如果收到 `onUserSigExpired` 回调，说明您登录用的 UserSig 票据已经过期，请使用新签发的 UserSig 进行重新登录。如果继续使用过期的 UserSig，会导致 IM SDK 登录进入死循环。

调用初始化接口

操作完上述步骤后，您可以调用 `initSDK` (Dart) 进行 SDK 初始化。

示例代码如下：



```
// 1. 从即时通信 IM 控制台获取应用 SDKAppID。
int sdkAppID = 0;
// 2. 添加 V2TimSDKListener 的事件监听器，sdkListener 是 V2TimSDKListener 的实现类
V2TimSDKListener sdkListener = V2TimSDKListener(
  onConnectFailed: (int code, String error) {
    // 连接失败的回调函数
    // code 错误码
    // error 错误信息
  },
  onConnectSuccess: () {
    // SDK 已经成功连接到腾讯云服务器
```

```
    },
    onConnecting: () {
      // SDK 正在连接到腾讯云服务器
    },
    onKickedOffline: () {
      // 当前用户被踢下线，此时可以 UI 提示用户，并再次调用 V2TIMManager 的 login() 函数重
    },
    onSelfInfoUpdated: (V2TimUserFullInfo info) {
      // 登录用户的资料发生了更新
      // info登录用户的资料
    },
    onUserSigExpired: () {
      // 在线时票据过期：此时您需要生成新的 userSig 并再次调用 V2TIMManager 的 login() 函
    },
    onUserStatusChanged: (List<V2TimUserStatus> userStatusList) {
      //用户状态变更通知
      //userStatusList 用户状态变化的用户列表
      //收到通知的情况：订阅过的用户发生了状态变更（包括在线状态和自定义状态），会触发该回调
      //在 IM 控制台打开了好友状态通知开关，即使未主动订阅，当好友状态发生变更时，也会触发该回
      //同一个账号多设备登录，当其中一台设备修改了自定义状态，所有设备都会收到该回调
    },
  );
  // 3.初始化SDK
  V2TimValueCallback<bool> initSDKRes =
      await TencentImSDKPlugin.v2TIMManager.initSDK(
    sdkAppID: sdkAppID, // SDKAppID
    loglevel: LogLevelEnum.V2TIM_LOG_ALL, // 日志登记等级
    listener: sdkListener, // 事件监听器
  );
  if (initSDKRes.code == 0) {
    //初始化成功
  }
```

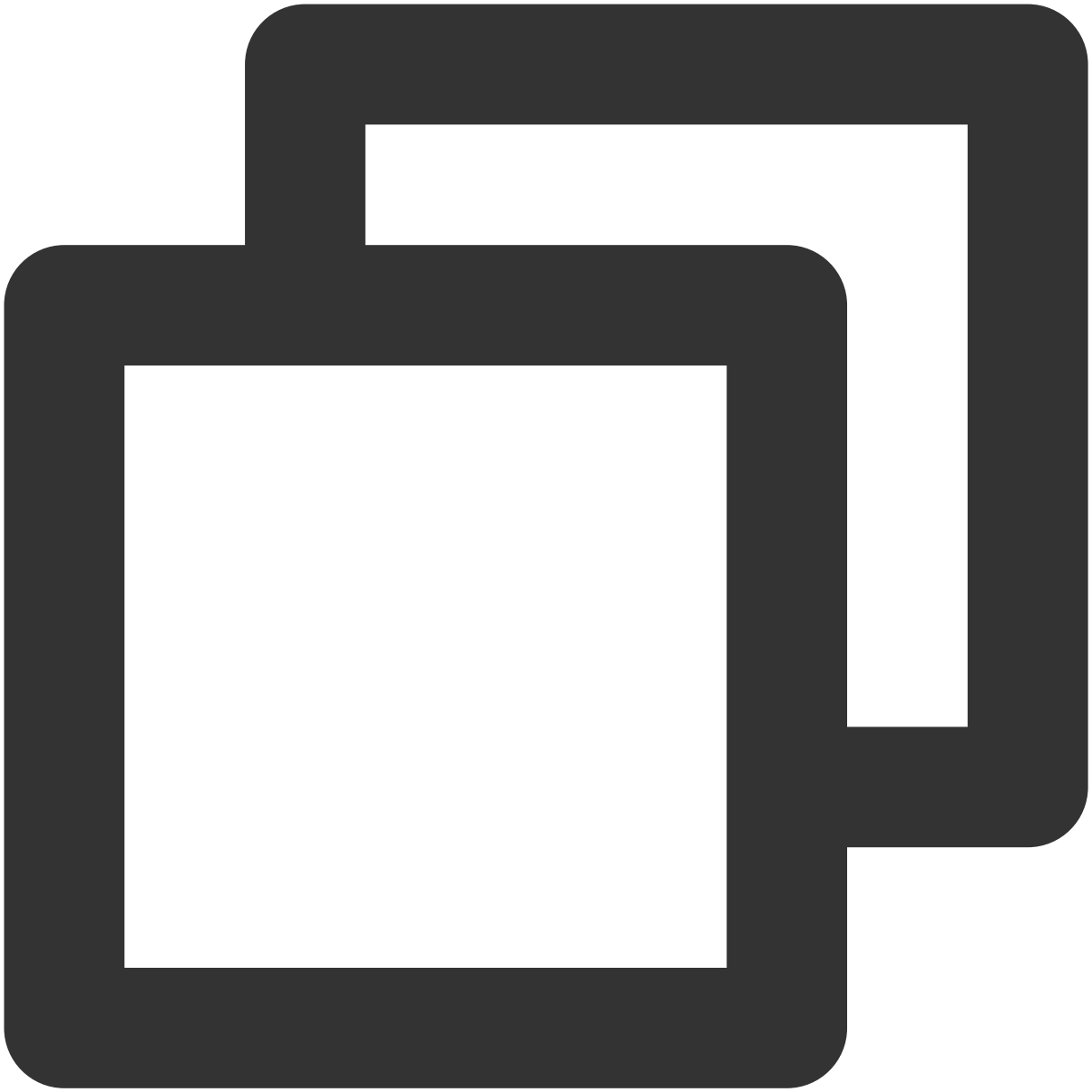
反初始化

普通情况下，如果您的应用生命周期跟 IM SDK 生命周期一致，退出应用前可以不进行反初始化。

但有些特殊场景，例如您只在进入特定界面后才初始化 IM SDK，退出界面后不再使用，可以对 IM SDK 进行反初始化。

反初始化需要操作一个步骤：调用反初始化接口 `unInitSDK` (Dart)

示例代码如下：



```
// 反初始化 SDK
TencentImSDKPlugin.v2TIMManager.unInitSDK();
```

常见问题

1. 在调用登录等其他接口时，发生错误，返回错误码是 6013 和错误描述是 "not initialized" 的信息。

在使用 IM SDK 登录、消息、群组、会话、关系链和资料、信令的功能前，必须先进行初始化。

Unity

最近更新时间：2023-09-20 09:58:06

功能描述

在使用 IM SDK 的各项功能前，**必须**先进行初始化。
大多数场景下，在应用生命周期内，您只需要进行一次 IM SDK 初始化。

初始化

初始化 SDK 需要操作以下步骤：
1. 准备 SDKAppID。
2. 设置 SdkConfig。
3. 设置 SDK 事件监听器。
4. 调用 `Init` 初始化 SDK。
下文我们将依次为您详细讲解具体的步骤内容。

准备 SDKAppID

您必须拥有正确的 SDKAppID，才能进行初始化。
SDKAppID 是腾讯云 IM 服务用于区分客户帐号的唯一标识。我们建议每一个独立的 App 都申请一个新的 SDKAppID。不同 SDKAppID 之间的消息是天然隔离的，不能互通。
您可以在 即时通信 IM 控制台 查看所有的 SDKAppID，单击 **创建新应用** 按钮，可以创建新的 SDKAppID。

设置 SdkConfig

初始化 SDK 前，您需要初始化一个 SdkConfig 对象。该对象用于设置本地 SDK 缓存、日志位置。
配置即时通信 IM 运行时的日志、数据的存储路径。

sdk_config_config_file_path

即时通信 IM 本地数据存储路径。
注意
该路径需要应用有可读写权限。

sdk_config_log_file_path

即时通信 IM 日志存储路径。

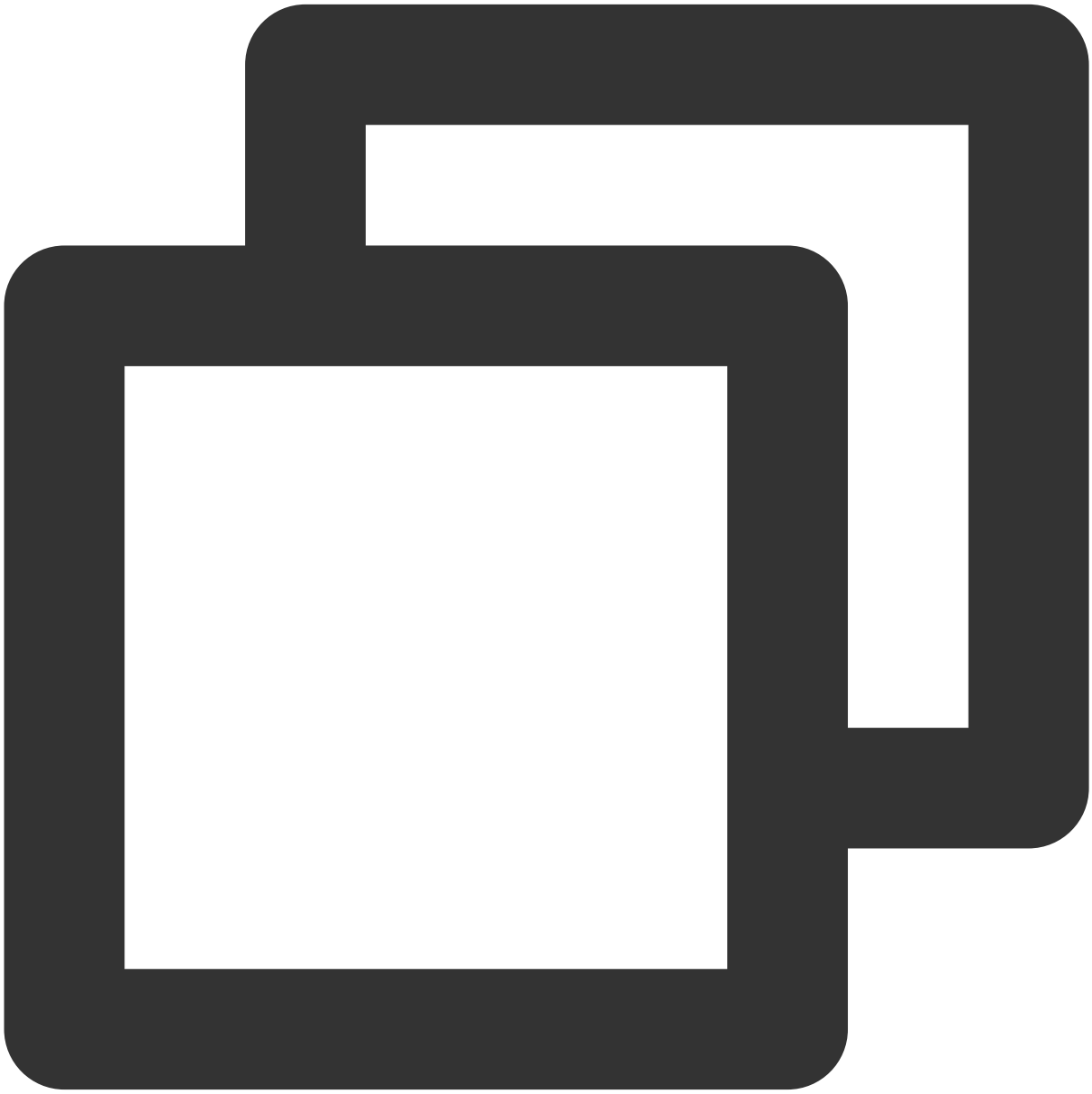
注意

该路径需要应用有可读写权限。

调用初始化接口

操作完上述步骤后，您可以调用 `Init` (点击查看详情) 进行 SDK 初始化。

示例代码如下：



```
namespace Com.Tencent.IM.Unity.UIKit{
    public static void Init() {
```

```
        string sdkappid = ""; // 从即时通信 IM 控制台获取应用 SDKAppID。
        SdkConfig sdkConfig = new SdkConfig();

        sdkConfig.sdk_config_config_file_path = Application.persistentDataPath + "/

        sdkConfig.sdk_config_log_file_path = Application.persistentDataPath + "/TIM

        TIMResult res = TencentIMSDK.Init(long.Parse(sdkappid), sdkConfig);
    }
}
```

注册 SDK 全局事件监听

SDK 初始化后，会通过诸如 `NetworkStatusListenerCallback`，`UserSigExpiredCallback` 等回调抛出一些事件，例如所示的连接状态、登录票据过期等。

我们建议您在调用 initSDK 后立即注册全局事件监听，在对应回调中做一些逻辑处理。

相关回调如下表所示：

事件回调	事件描述
RecvNewMsgCallback	注册收到新消息回调
MsgReadedReceiptCallback	设置消息已读回执回调
MsgRevokeCallback	设置接收的消息被撤回回调
MsgElemUploadProgressCallback	设置消息内元素相关文件上传进度回调
GroupTipsEventCallback	设置群组系统消息回调
GroupAttributeChangedCallback	设置群组属性变更回调
ConvTotalUnreadMessageCountChangedCallback	设置会话未读消息总数变更的回调
NetworkStatusListenerCallback	设置网络连接状态监听回调
KickedOfflineCallback	设置被踢下线通知回调
UserSigExpiredCallback	设置票据过期回调
OnAddFriendCallback	设置添加好友的回调
OnDeleteFriendCallback	设置删除好友的回调
UpdateFriendProfileCallback	设置更新好友资料的回调
FriendAddRequestCallback	设置好友添加请求的回调

版权所有：腾讯云计算（北京）有限责任公司

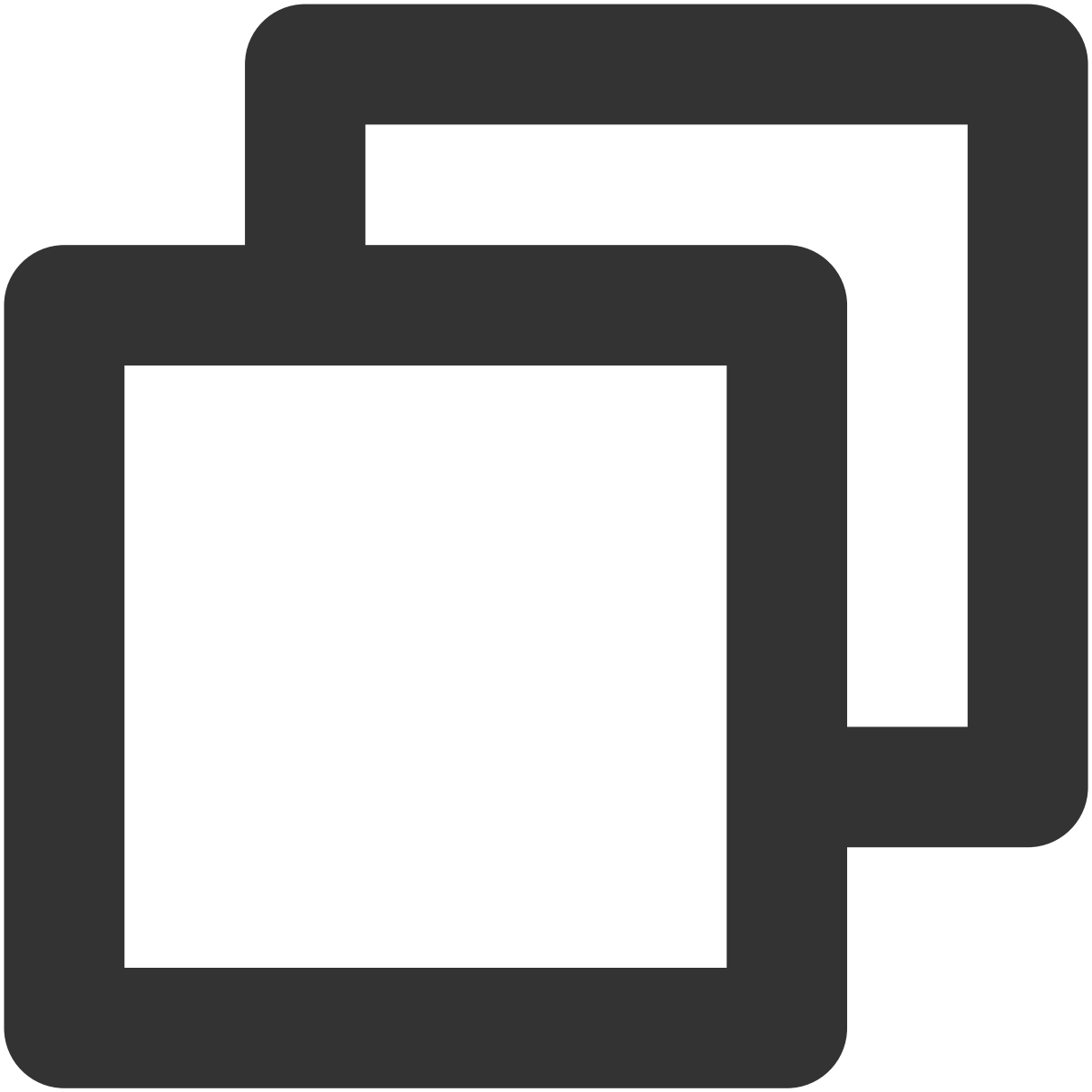
FriendApplicationListDeletedCallback	设置好友申请被删除的回调
FriendApplicationListReadCallback	设置好友申请已读的回调
FriendBlackListAddedCallback	设置黑名单新增的回调
FriendBlackListDeletedCallback	设置黑名单删除的回调
LogCallback	设置日志回调
MsgUpdateCallback	设置消息在云端被修改后回传回来的消息更新通知回调
MsgGroupMessageReadMemberListCallback	获取群消息已读群成员列表

注意

如果收到 `UserSigExpiredCallback` 回调，说明您登录用的 UserSig 票据已经过期，请使用新签发的 UserSig 进行重新登录。如果继续使用过期的 UserSig，会导致 IM SDK 登录进入死循环。

反初始化

普通情况下，如果您的应用生命周期跟 IM SDK 生命周期一致，退出应用前可以不进行反初始化。
但有些特殊场景，例如您只在进入特定界面后才初始化 IM SDK，退出界面后不再使用，可以对 IM SDK 进行反初始化。
反初始化需要操作一个步骤：调用反初始化接口 `unInit` (点击查看详情)
示例代码如下：



```
// 反初始化 SDK
TencentIMSDK.Uninit();
```

其他

调用 SDK 同步返回的结果，当 res 为 TIMResult.TIM_SUCC = 0 时接口调用成功。
在 SDK 初始化成功后，请立即添加需要用到的事件监听，以免消息遗漏。

常见问题

1. 在使用 IM SDK 登录、消息、群组、会话、关系链和资料、信令的功能前，必须先进行初始化。

React Native

最近更新时间：2024-01-31 15:33:04

功能描述

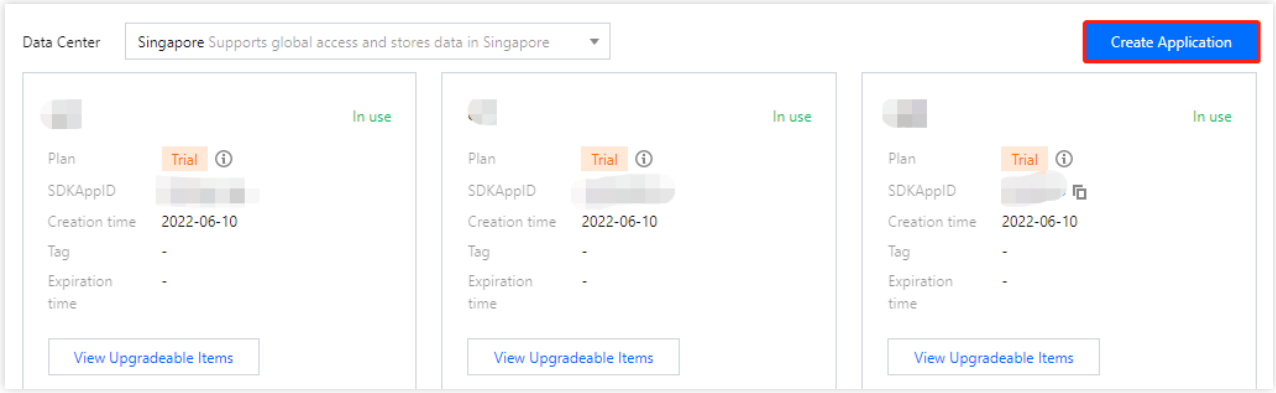
在使用 IM SDK 的各项功能前，**必须**先进行初始化。
大多数场景下，在应用生命周期内，您只需要进行一次 IM SDK 初始化。

初始化

初始化 SDK 需要操作以下步骤：
1. 准备 SDKAppID。
2. 设置 LogLevelEnum。
3. 设置 SDK 事件监听器。
4. 调用 `initSDK` 初始化 SDK。
下文我们将依次为您详细讲解具体的步骤内容。

准备 SDKAppID

您必须拥有正确的 SDKAppID，才能进行初始化。
SDKAppID 是腾讯云 IM 服务用于区分客户帐号的唯一标识。我们建议每一个独立的 App 都申请一个新的 SDKAppID。不同 SDKAppID 之间的消息是天然隔离的，不能互通。
您可以在 即时通信 IM 控制台 查看所有的 SDKAppID，单击 **创建新应用** 按钮，可以创建新的 SDKAppID。



设置 LogLevelEnum

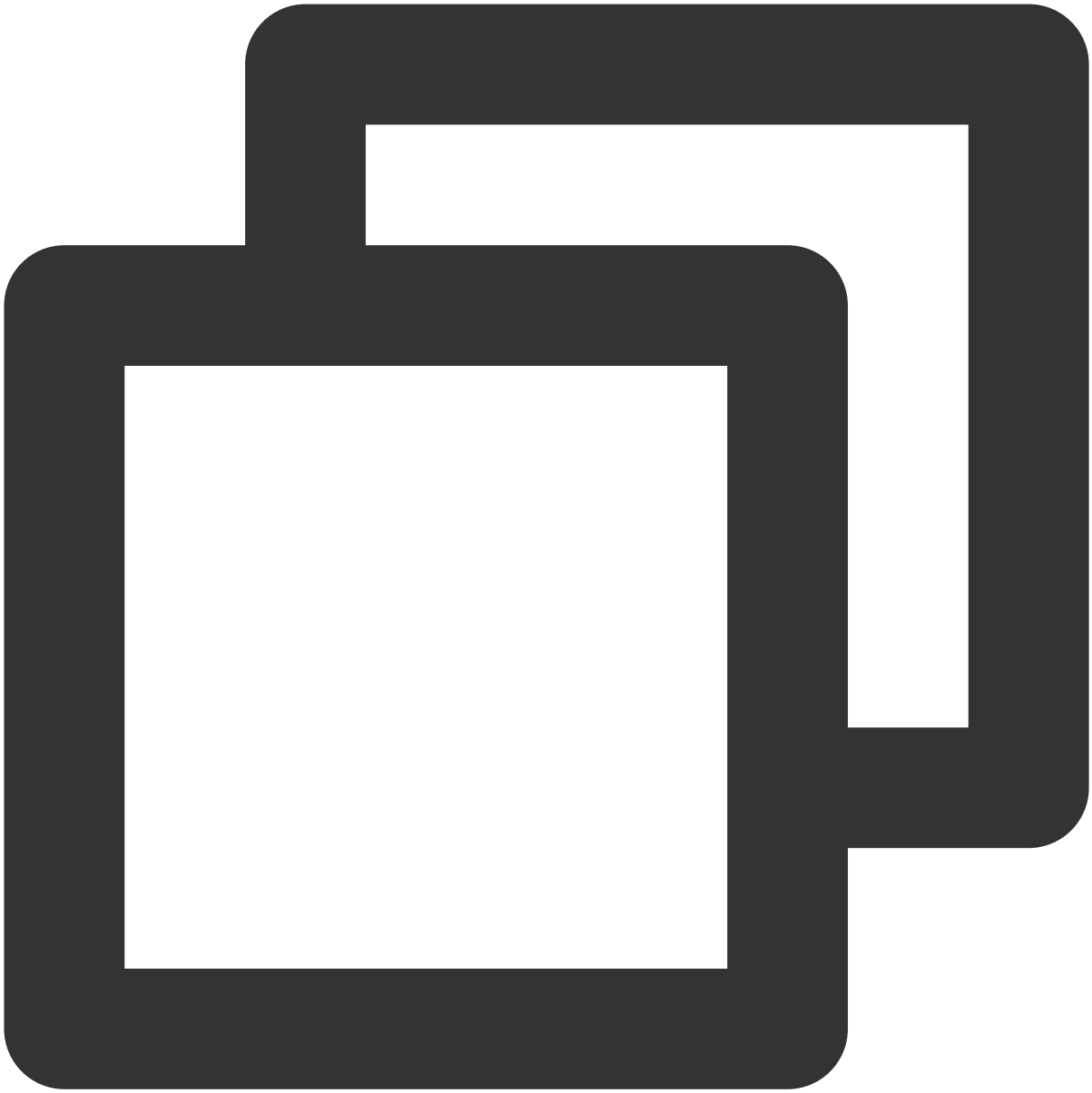
初始化 SDK 前，您需要初始化一个 `LogLevelEnum` 对象。该对象用于对 SDK 进行日志级别设置。

设置日志级别

IM SDK 支持多种日志级别，如下表所示：

日志级别	LOG 输出量
LogLevelEnum.V2TIM_LOG_NONE	不输出任何 log
LogLevelEnum.V2TIM_LOG_DEBUG	输出 DEBUG，INFO，WARNING，ERROR 级别的 log （默认的日志级别）
LogLevelEnum.V2TIM_LOG_INFO	输出 INFO，WARNING，ERROR 级别的 log
LogLevelEnum.V2TIM_LOG_WARN	输出 WARNING，ERROR 级别的 log
LogLevelEnum.V2TIM_LOG_ERROR	输出 ERROR 级别的 log

SDK 日志存储规则如下：
IM SDK 本地日志默认保存 7 天；SDK 在初始化时，会自动清理 7 天之前的日志。
在 Android 平台上，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 4.7.1 版本开始，IM SDK 的日志开始采用 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
```

设置 SDK 事件监听器

SDK 初始化后，会通过 `V2TimSDKListener` 抛出一些事件，例如连接状态、登录票据过期等。

我们建议您在调用 initSDK 时传入 `V2TimSDKListener` (Details) 接口添加 SDK 事件监听器，在对应回调中做一些逻辑处理。

`V2TimSDKListener` 相关回调如下表所示：

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	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帐号，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。

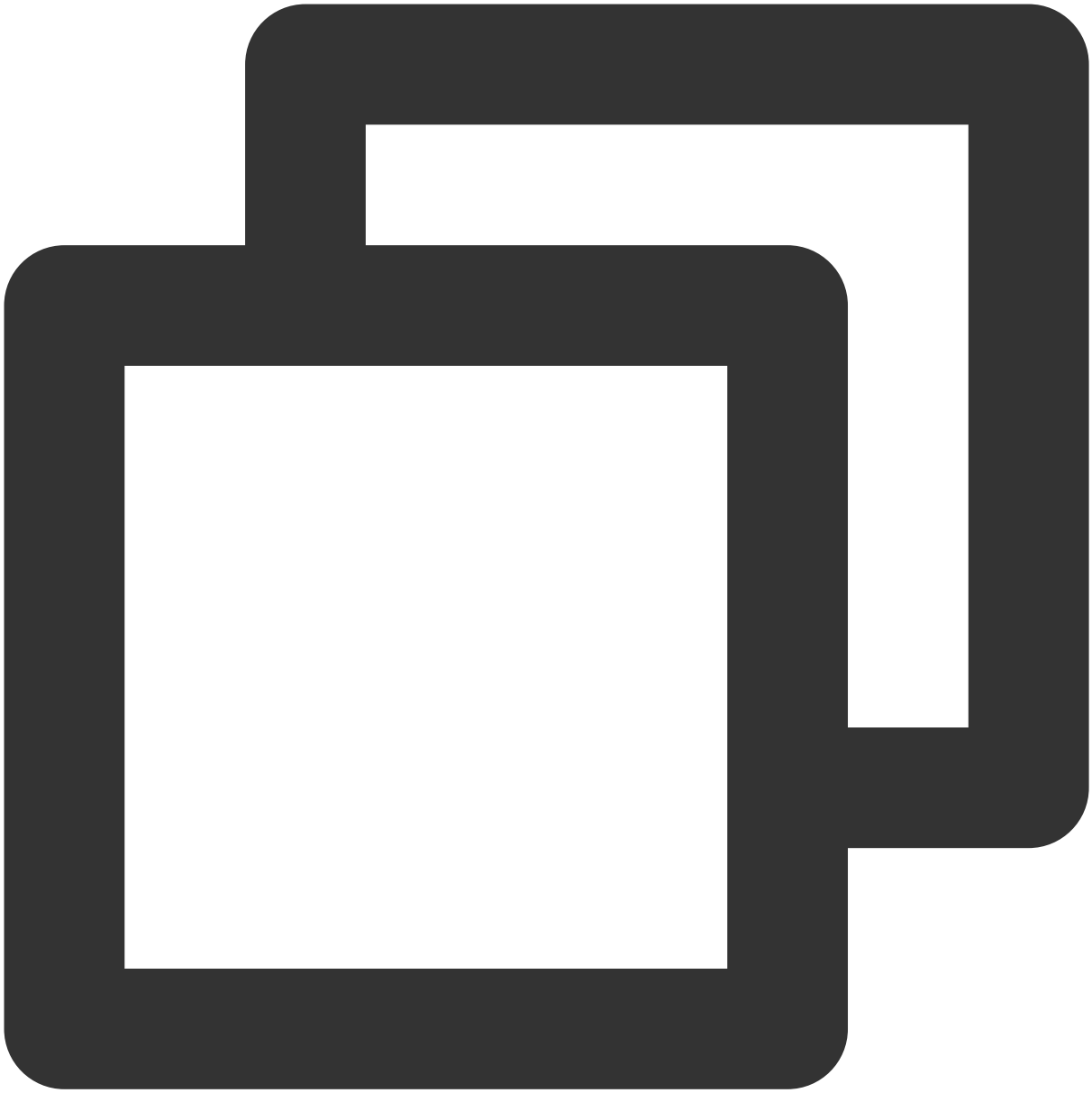
注意：

如果收到 `onUserSigExpired` 回调，说明您登录用的 UserSig 票据已经过期，请使用新签发的 UserSig 进行重新登录。如果继续使用过期的 UserSig，会导致 IM SDK 登录进入死循环。

调用初始化接口

操作完上述步骤后，您可以调用 `initSDK` (Details) 进行 SDK 初始化。

示例代码如下：

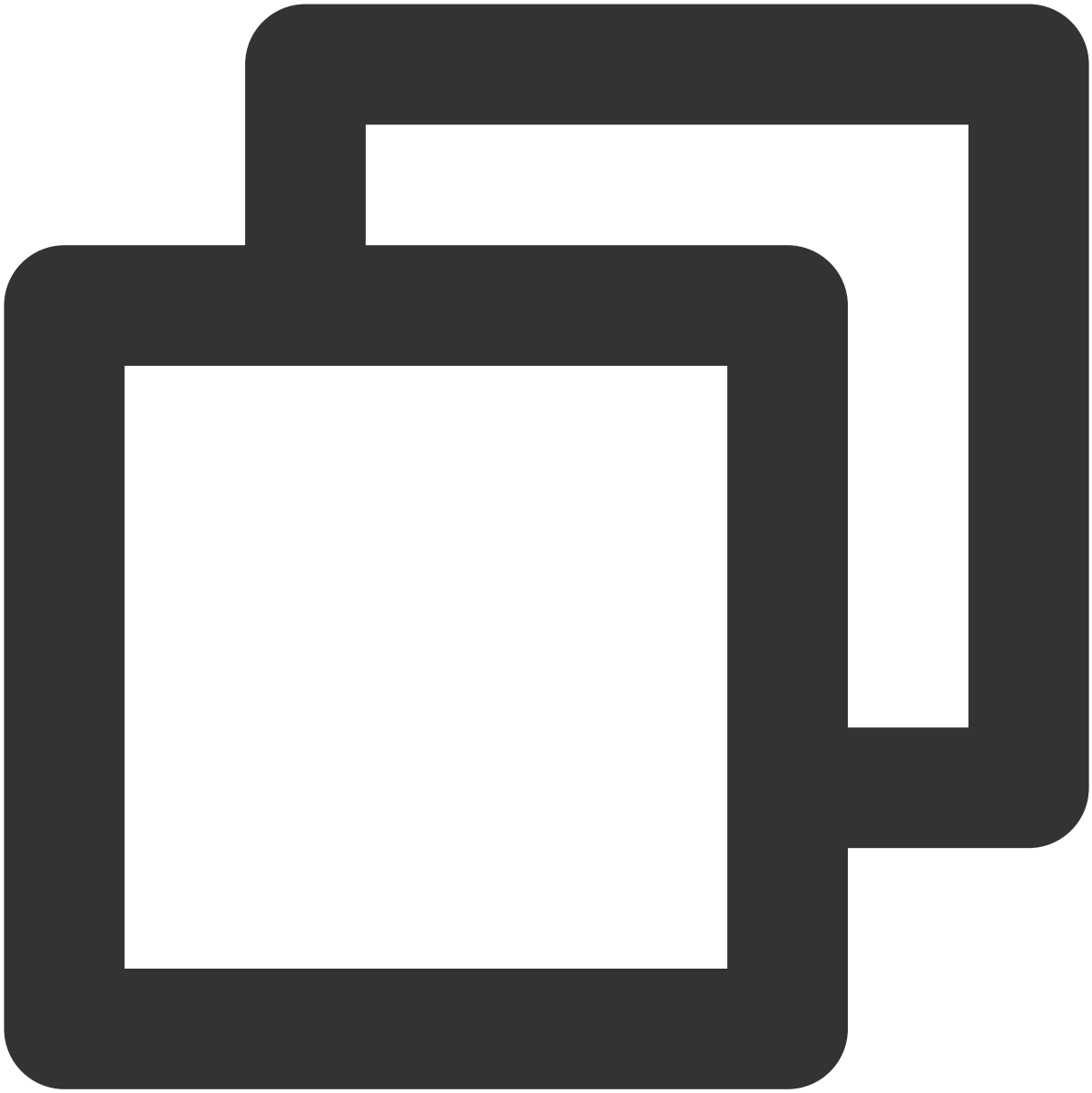


```
import { TencentImSDKPlugin, LogLevelEnum } from 'react-native-tim-js';


// 1. 从即时通信 IM 控制台获取应用 SDKAppID。
const sdkAppID = 0;
// 2. 添加 V2TimSDKListener 的事件监听器
const sdkListener = {
      onConnectFailed: (code, error) {},
      onConnectSuccess: () {},
      onConnecting: () {},
      onKickedOffline: () {},
```

```
      onSelfInfoUpdated: (V2TimUserFullInfo info) {},
      onUserSigExpired: () {},
};

// 3.初始化，成功之后可以注册事件。
TencentImSDKPlugin.v2TIMManager.initSDK(
      sdkAppID: sdkAppID,
      loglevel: LogLevelEnum.V2TIM_LOG_ALL,
      listener: sdkListener,
);
```

反初始化

普通情况下，如果您的应用生命周期跟 IM SDK 生命周期一致，退出应用前可以不进行反初始化。
但有些特殊场景，例如您只在进入特定界面后才初始化 IM SDK，退出界面后不再使用，可以对 IM SDK 进行反初始化。
反初始化需要操作一个步骤：调用反初始化接口 `unInitSDK` (Details)
示例代码如下：



```
import { TencentImSDKPlugin } from "react-native-tim-js";

// 反初始化 SDK
TencentImSDKPlugin.v2TIMManager.unInitSDK();
```

常见问题

1. 在调用登录等其他接口时，发生错误，返回错误码是 6013 和错误描述是 "not initialized" 的信息。

在使用 IM SDK 登录、消息、群组、会话、关系链和资料、信令的功能前，必须先进行初始化。

登录登出

Android&iOS&Windows&Mac

最近更新时间：2024-07-05 15:24:08

功能描述

初始化 Chat SDK 后，您需要调用 SDK 登录接口，验证账号身份，获得账号的功能使用权限。登录 SDK 成功后，才能正常使用消息、会话等功能。

注意：

1. 除了获取会话列表、拉取历史消息这两个接口以外，SDK 的各项功能接口必须在登录成功后才能调用。因此在使用其他功能之前，**请务必登录且确保登录成功**，否则可能导致功能异常或不可用！
2. 获取会话列表、拉取历史消息接口允许在登录失败的情况下调用，此时接口返回的是本地缓存的会话列表和历史消息，可用于无网络情况下展示。

登录

首次登录账号时，不需要先注册这个账号，直接登录即可，SDK 在登录过程中发现是未注册的账号，会自动注册。

您可以调用 `login` (Android / iOS & Mac / Windows) 接口进行登录。

`login` 接口的关键参数如下：

参数	含义	说明
UserID	登录用户唯一标识	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（-）。长度不超过 32 字节。
UserSig	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。

您需要在以下场景调用 `login` 接口：

App 启动后首次使用 SDK 的功能。

登录时票据过期：`login` 接口的回调会返回 `ERR_USER_SIG_EXPIRED（6206）` 或 `ERR_SVR_ACCOUNT_USERSIG_EXPIRED（70001）` 错误码，此时请您生成新的 userSig 重新登录。

在线时票据过期：用户在线期间也可能收到 `onUserSigExpired` (Android / iOS & Mac/ Windows) 回调，此时需要您生成新的 userSig 并重新登录。

在线时被踢下线：用户在线情况下被踢，SDK 会通过 `onKickedOffline` (Android / iOS & Mac / Windows) 回调通知给您，此时可以在 UI 提示用户，并调用 `login` 重新登录。

以下场景无需调用 `login` 接口：

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。

注意：

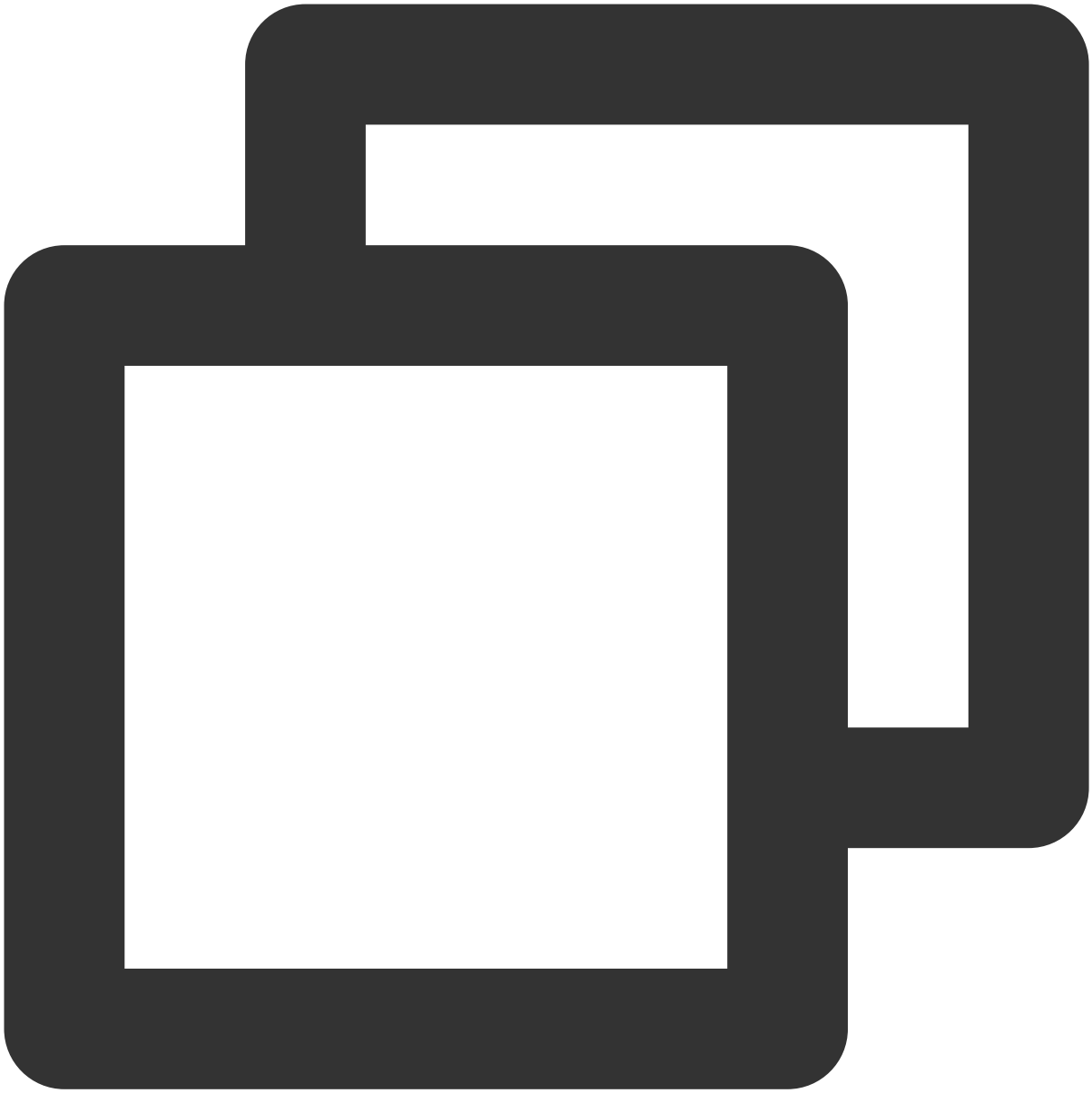
1. 调用 SDK 接口成功登录后，将会开始计算 MAU，请根据业务场景合理调用登录接口，避免出现 MAU 过高的情况。
2. 在一个 App 中，SDK 不支持多个账号同时在线，如果同时登录多个账号，只有最后登录的账号在线。

示例代码如下：

Android

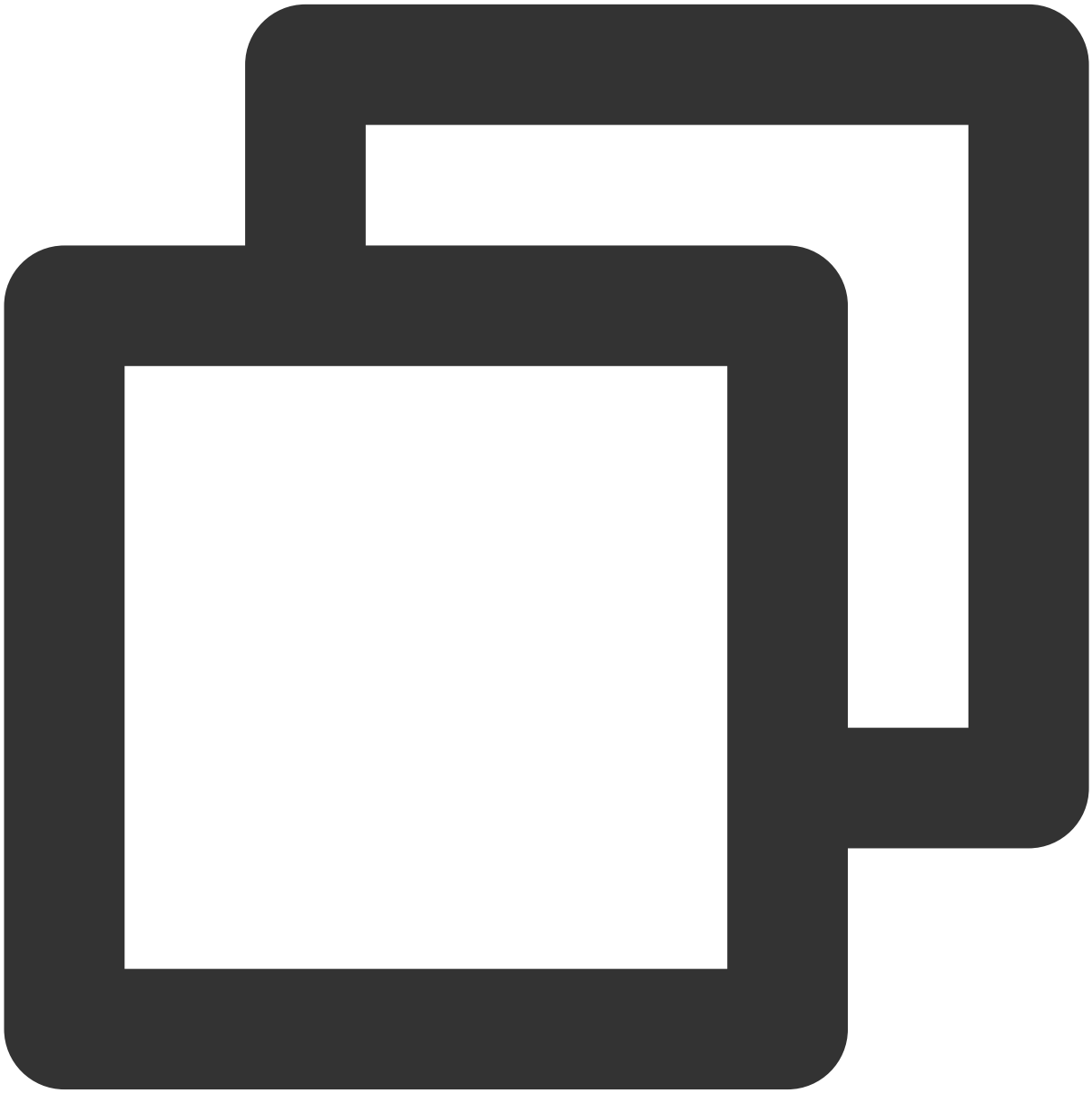
iOS & Mac

Windows



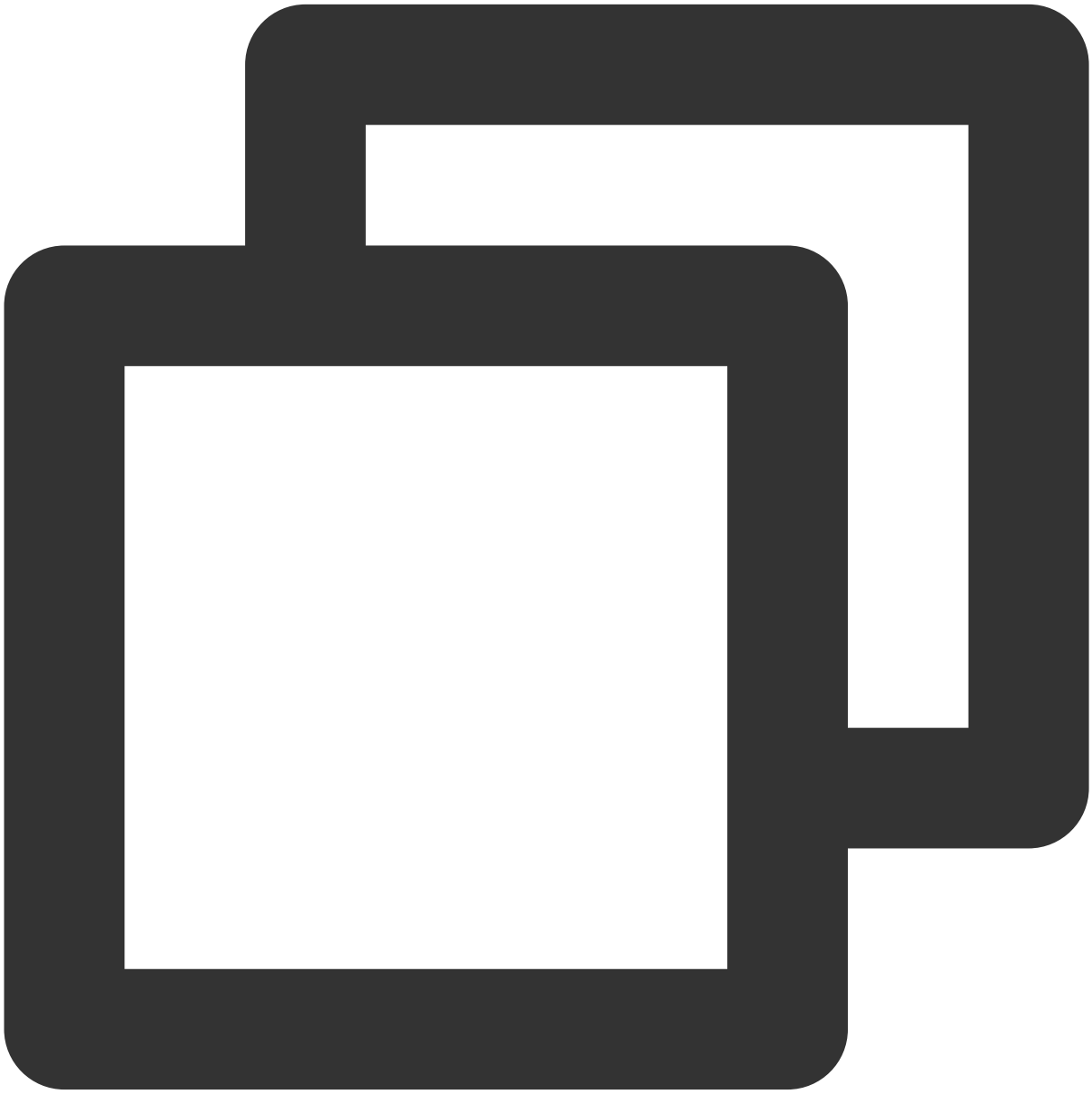
```
String userID = "your user id";
String userSig = "userSig from your server";
V2TIMManager.getInstance().login(userID, userSig, new V2TIMCallback() {
    @Override
    public void onSuccess() {
        Log.i("sdk", "success");
    }

    @Override
    public void onError(int code, String desc) {
        // 如果返回以下错误码，表示使用 UserSig 已过期，请您使用新签发的 UserSig 进行再次登录。
```

```
        // 1. ERR_USER_SIG_EXPIRED (6206)
        // 2. ERR_SVR_ACCOUNT_USERSIG_EXPIRED (70001)
        // 注意：其他的错误码，请不要在这里调用登录接口，避免 SDK 登录进入死循环。
        Log.i("sdk", "failure, code:" + code + ", desc:" + desc);
    }
});
```



```
NSString *userID = @"your user id";
NSString *userSig = @"userSig from your server";
[[V2TIMManager sharedInstance] login:userID userSig:userSig succ:^{
    NSLog(@"success");
```

```
} fail:^(int code, NSString *desc) {
    // 如果返回以下错误码，表示使用 UserSig 已过期，请您使用新签发的 UserSig 进行再次登录。
    // 1. ERR_USER_SIG_EXPIRED (6206)
    // 2. ERR_SVR_ACCOUNT_USERSIG_EXPIRED (70001)
    // 注意：其他的错误码，请不要在这里调用登录接口，避免 SDK 登录进入死循环。
    NSLog(@"failure, code:%d, desc:%@", code, desc);
}];
```



```
class LoginCallback final : public V2TIMCallback {
public:
    LoginCallback() = default;
```

```
    ~LoginCallback() override = default;

    using SuccessCallback = std::function<void()>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
    }
    void OnSuccess() override {
        if (success_callback_) {
            success_callback_();
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
};

V2TIMString userID = "your user id";
V2TIMString userSig = "userSig from your server";
auto callback = new LoginCallback;
callback->SetCallback(
    [=]() {
        std::cout << "success";
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 如果返回以下错误码，表示使用 UserSig 已过期，请您使用新签发的 UserSig 进行再次登录。
        // 1. ERR_USER_SIG_EXPIRED (6206)
        // 2. ERR_SVR_ACCOUNT_USERSIG_EXPIRED (70001)
        // 注意：其他的错误码，请不要在这里调用登录接口，避免 SDK 登录进入死循环。
        std::cout << "failure, code:" << error_code << ", desc:" << error_message.C
        delete callback;
    });
V2TIMManager::GetInstance()->Login(userID, userSig, callback);
```

获取登录用户

在登录成功后，通过调用 `getLoginUser` (Android / iOS & Mac / Windows) 获取登录用户 UserID。

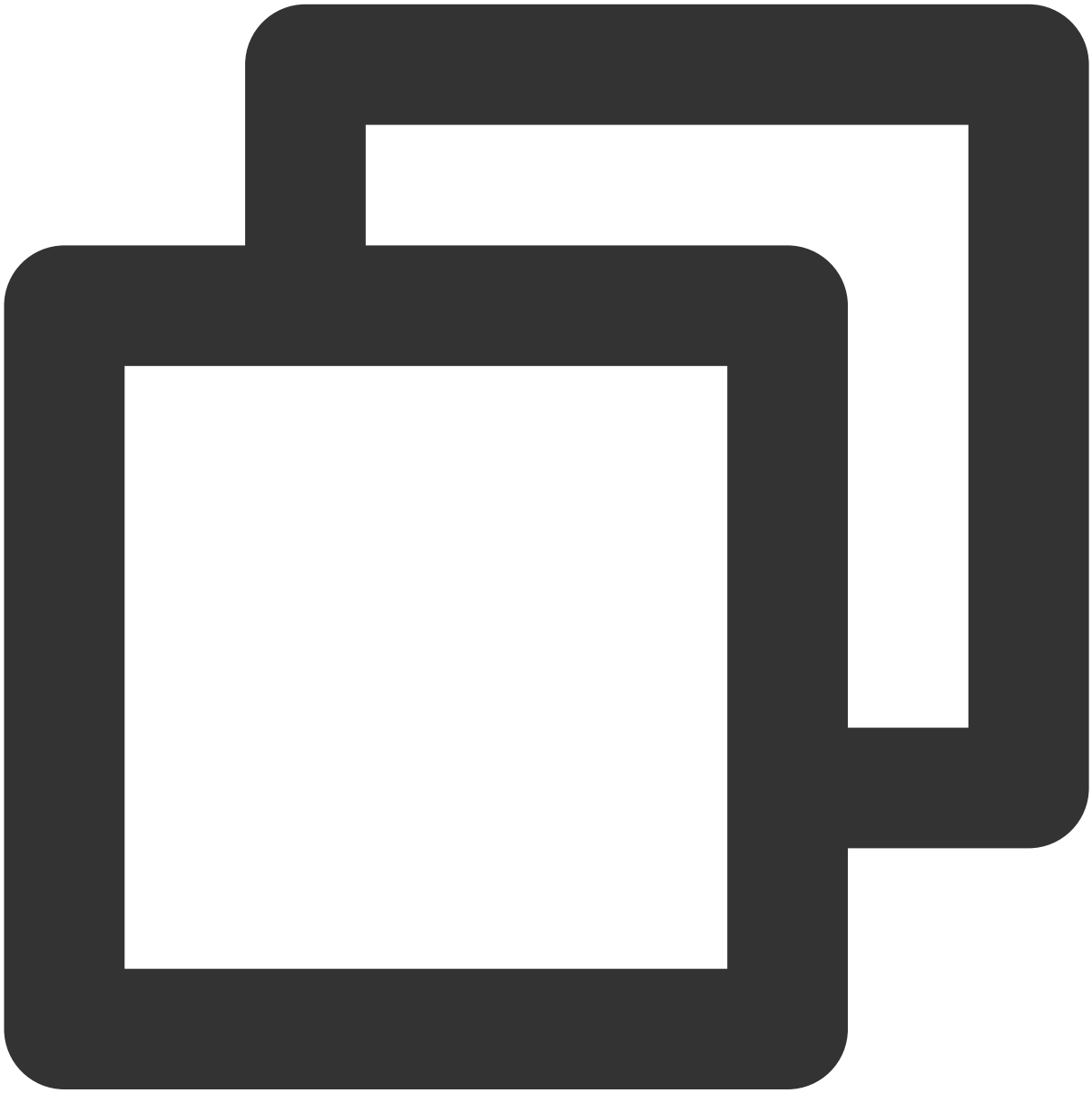
如果登录失败，获取的登录用户 UserID 为空。

示例代码如下：

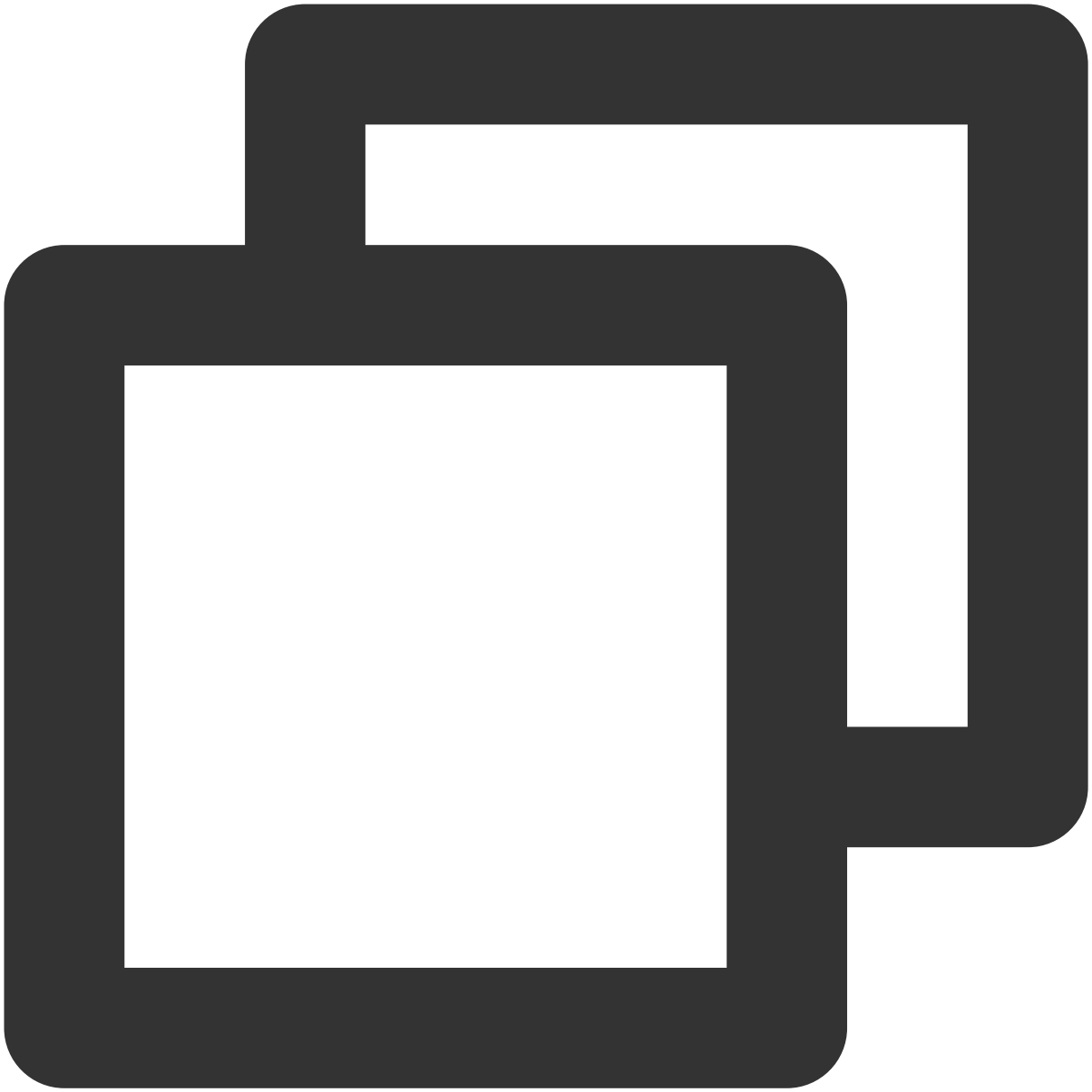
Android

iOS & Mac

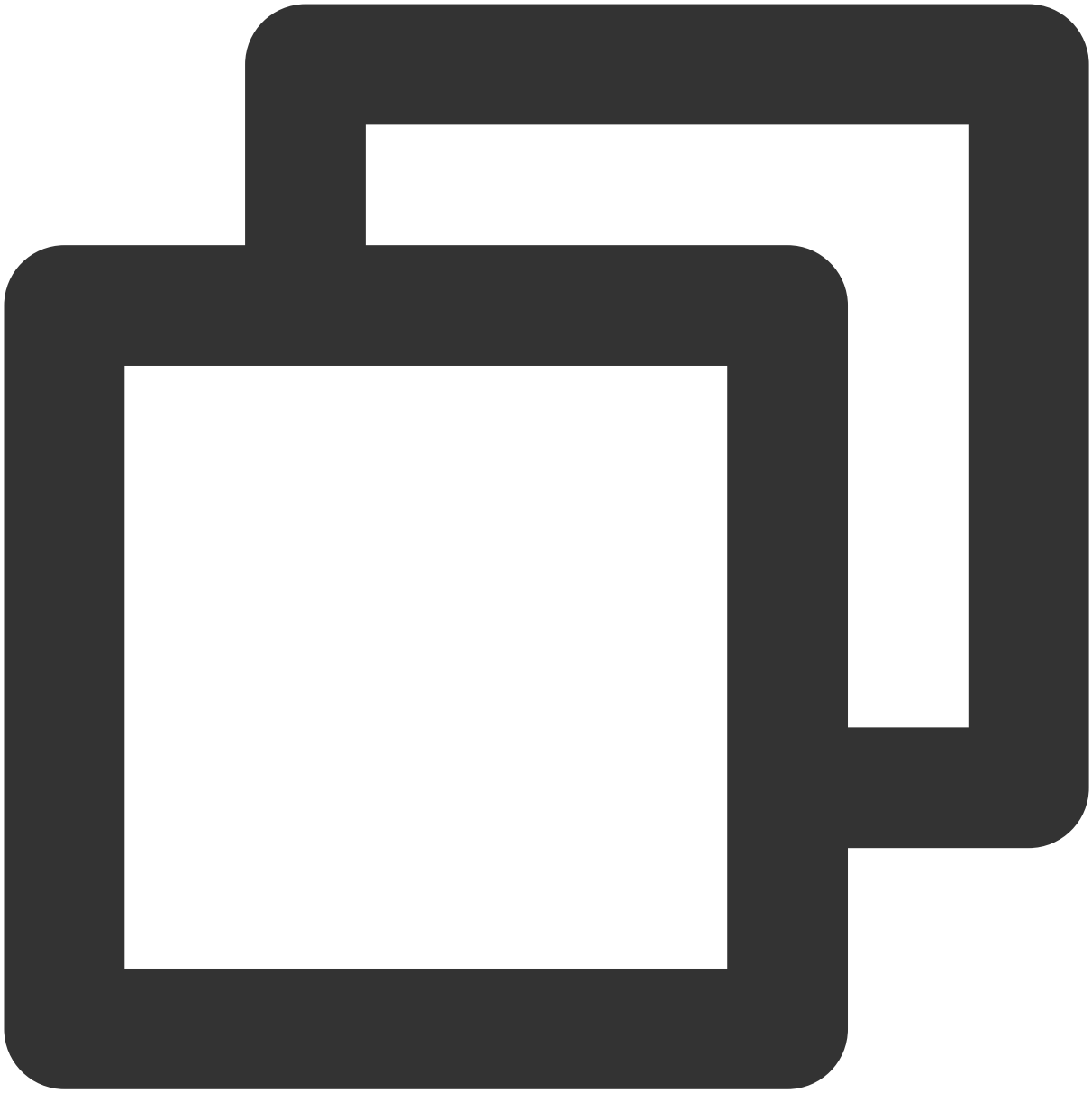
Windows



```
// 获取登录成功的用户 UserID
String loginUserID = V2TIMManager.getInstance().getLoginUser();
```



```
// 获取登录成功的用户 UserID
NSString *loginUserID = [[V2TIMManager sharedInstance] getLoginUser];
```



```
// 获取登录成功的用户 UserID
V2TIMString loginUserID = V2TIMManager::GetInstance()->GetLoginUser();
```

获取登录状态

通过调用 `getLoginStatus` (Android / iOS & Mac / Windows) 获取登录状态，如果用户已经处于已登录和登录中状态，请勿再频繁调用登录接口登录。SDK 支持的登录状态，如下表所示：

登录状态	含义

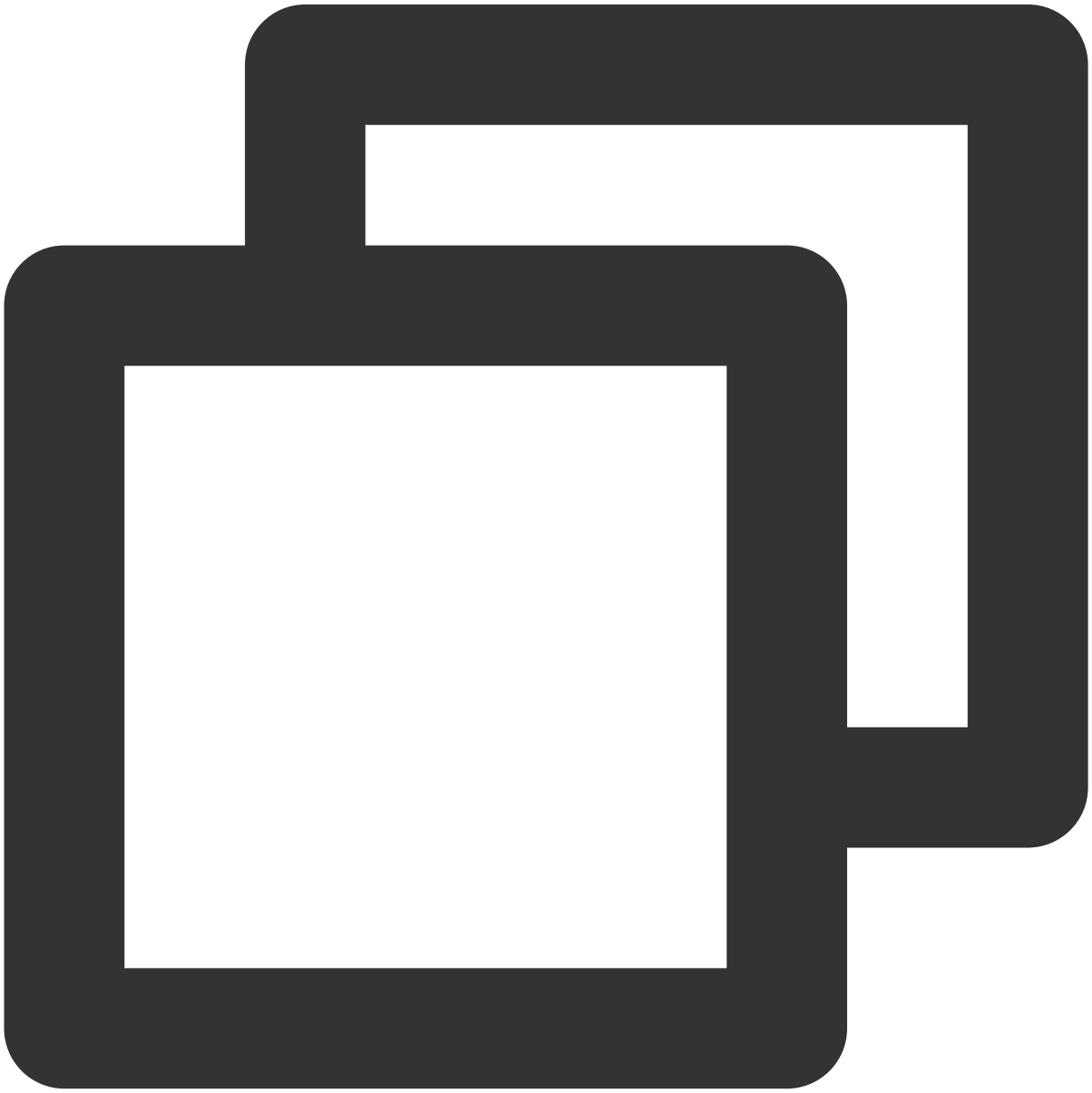
V2TIM_STATUS_LOGINED	已登录
V2TIM_STATUS_LOGINING	登录中
V2TIM_STATUS_LOGOUT	无登录

示例代码如下：

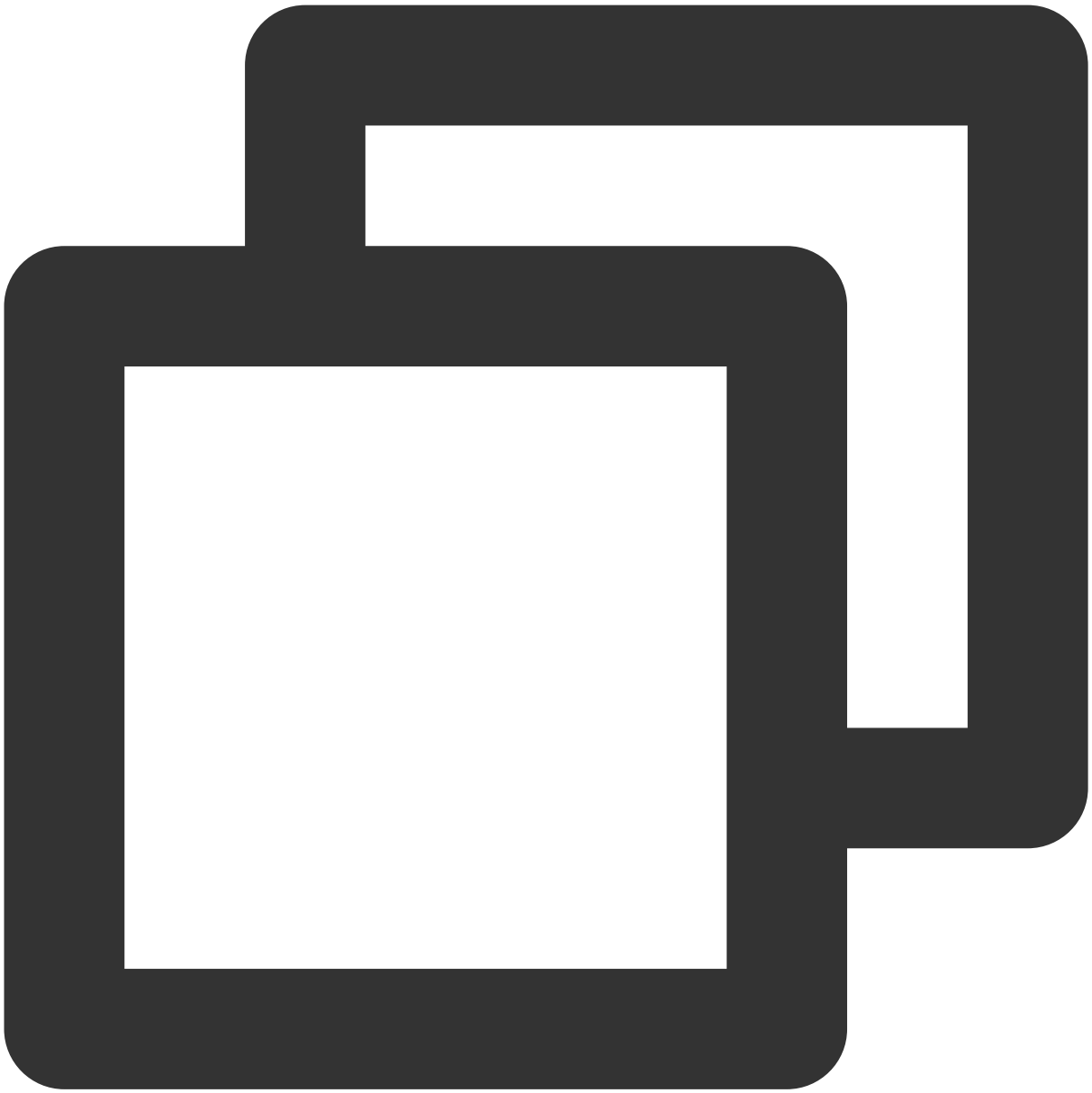
Android

iOS & Mac

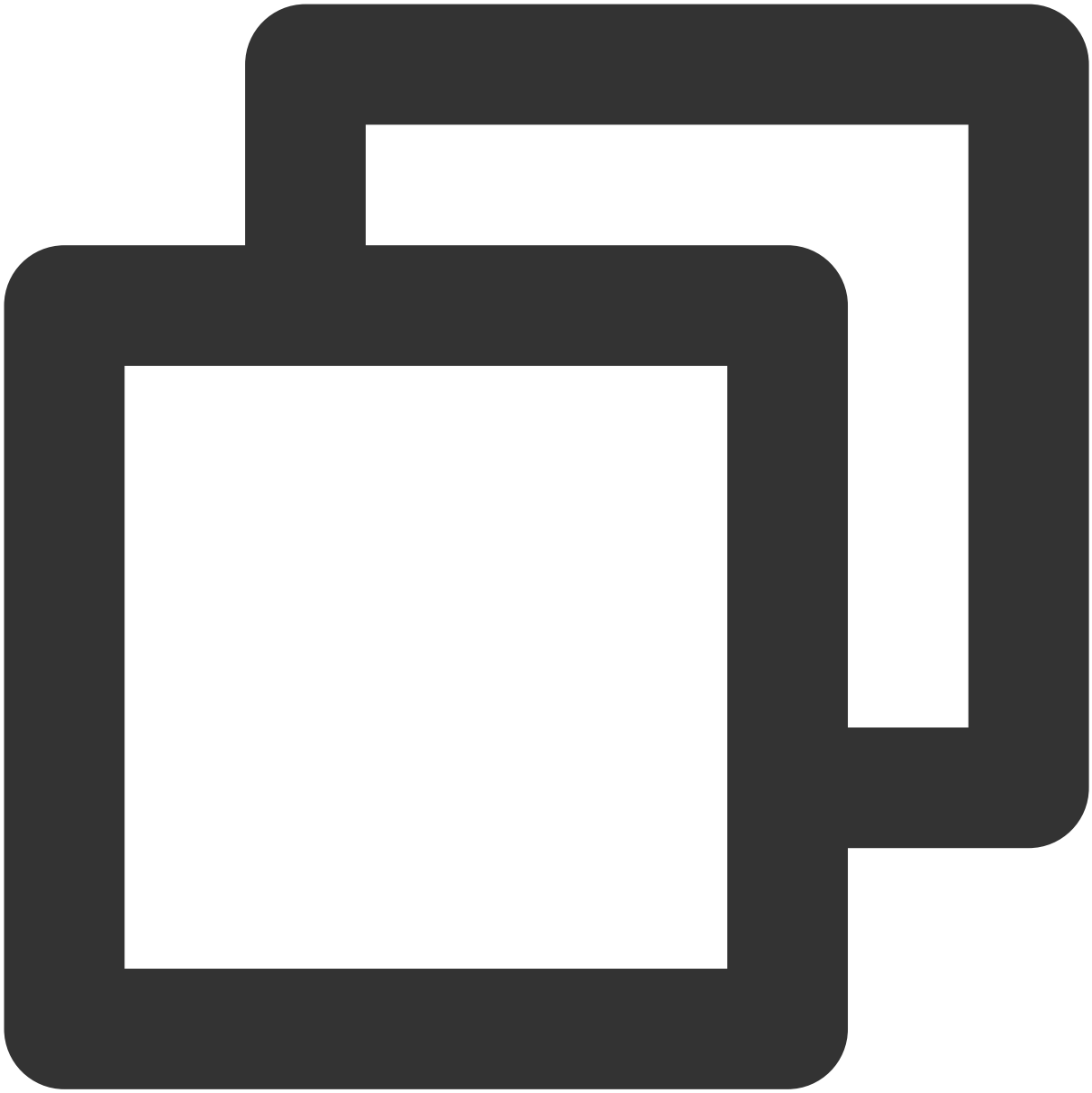
Windows



```
int loginStatus = V2TIMManager.getInstance().getLoginStatus();
```



```
V2TIMLoginStatus loginStatus = [[V2TIMManager sharedInstance] getLoginStatus];
```



```
V2TIMLoginStatus loginStatus = V2TIMManager::GetInstance()->GetLoginStatus();
```

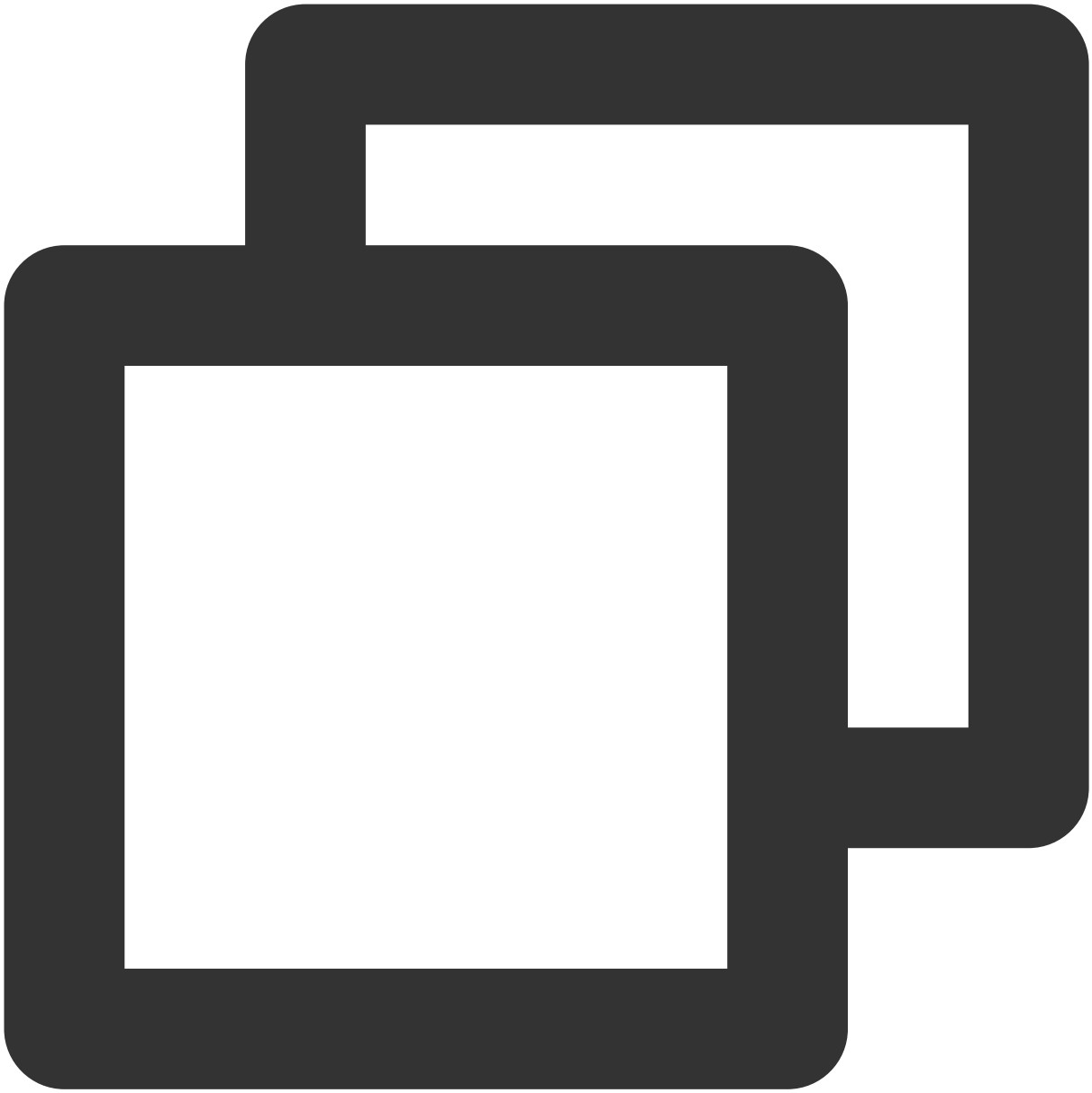
多端登录与互踢

您可以在 控制台 配置 SDK 的多端登录策略。多端登录策略有多种可选，例如“移动或桌面平台可有1种平台在线+Web可同时在线”、“不同平台均可同时在线”。相关配置请参考：登录设置。

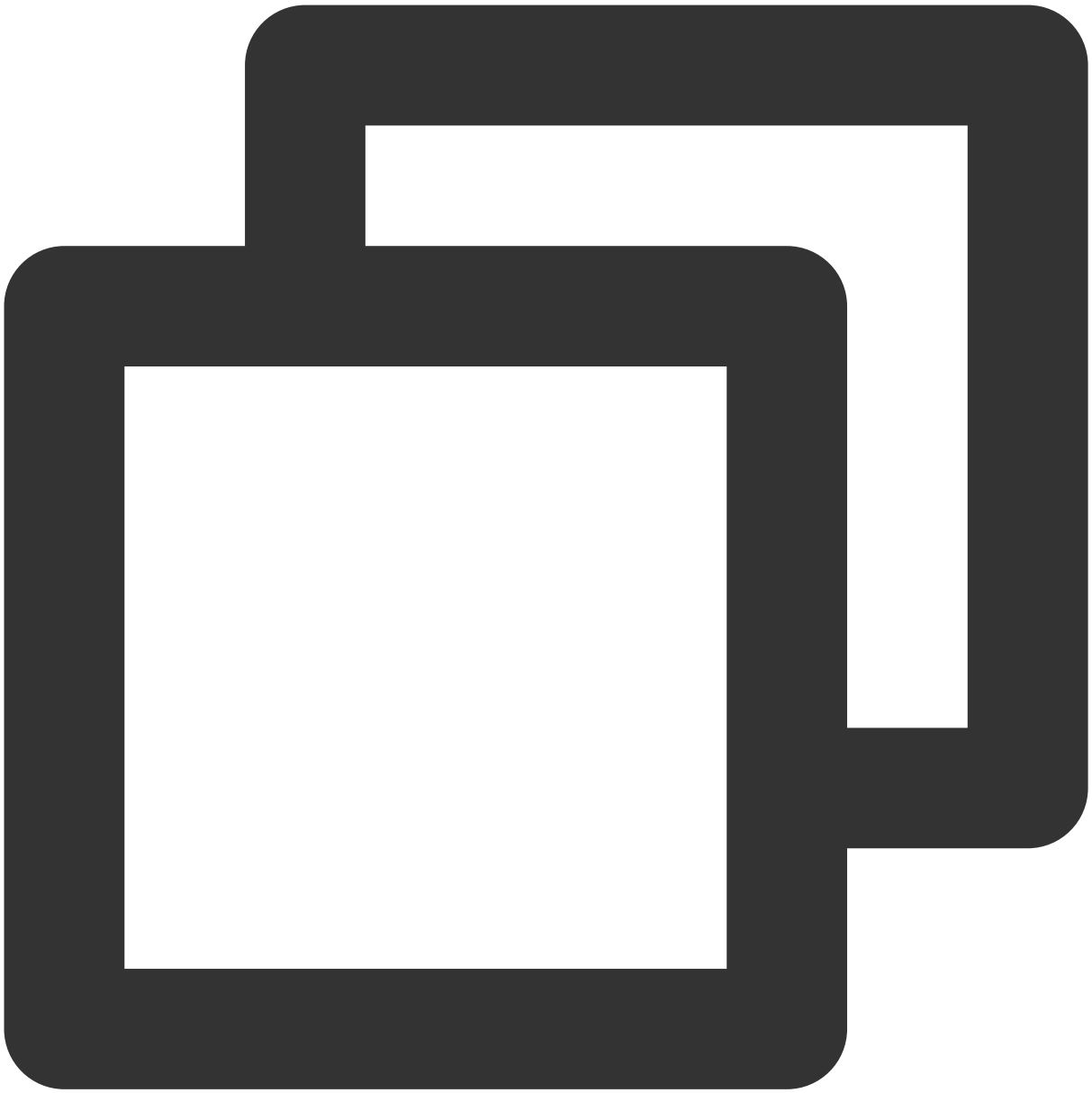
您可以在 控制台 配置 SDK 的同平台可登录实例数配置，即相同平台的设备可支持几个同时在线。此功能仅限进阶版使用。目前 Web 端可同时在线个数最多为 10 个。Android、iPhone、iPad、Windows、Mac 平台可同时在线设备个数最多为 3 个。相关配置请参考：登录设置。
调用 `login` 接口时，如果同一个账号的多端登录策略超出限制，新登录的实例会把之前已登录的实例踢下线。如果您在初始化时调用了 `addIMSDKListener` (Android / iOS & Mac / Windows) 添加了 SDK 监听器，对于之前已登录的一方，会收到 `onKickedOffline` (Android / iOS & Mac / Windows) 回调。

登出

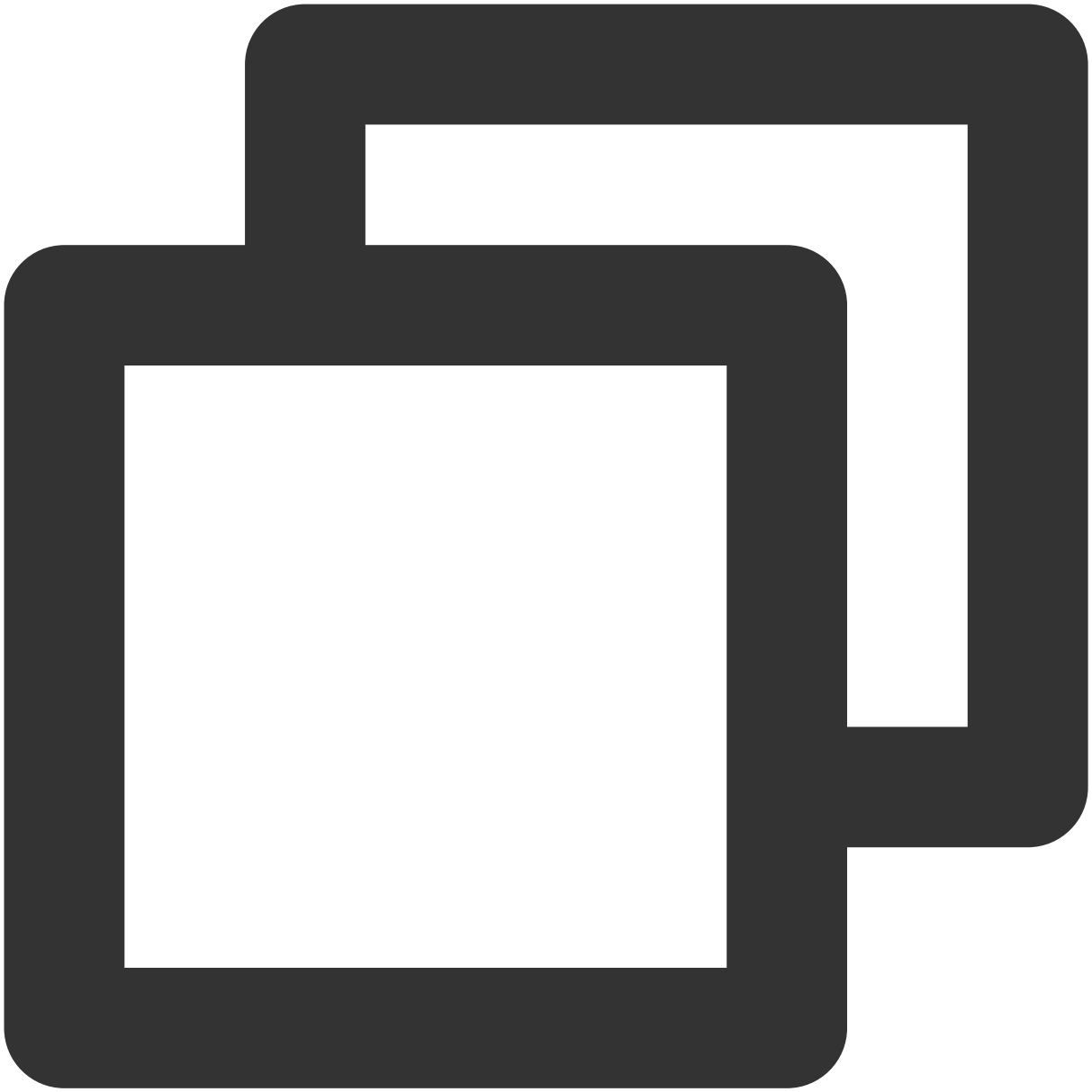
普通情况下，如果您的应用生命周期跟 SDK 生命周期一致，退出应用前可以不登出，直接退出即可。但有些特殊场景，例如您只在进入特定界面后才使用SDK，退出界面后不再使用，此时可以调用 `logout` (Android / iOS & Mac / Windows) 接口登出 SDK。登出成功后，不会再收到其他人发送的新消息。注意这种情况下，登出成功后还需要调 `unInitSDK` (Android / iOS & Mac / Windows) 对 SDK 进行反初始化。
示例代码如下：
Android
iOS & Mac
Windows



```
V2TIMManager.getInstance().logout(new V2TIMCallback() {
    @Override
    public void onSuccess() {
        Log.i("sdk", "success");
    }

    @Override
    public void onError(int code, String desc) {
        Log.i("sdk", "failure, code:" + code + ", desc:" + desc);
    }
});
```



```
[[V2TIMManager sharedInstance] logout:^{
    NSLog(@"success");
} fail:^(int code, NSString *desc) {
    NSLog(@"failure, code:%d, desc:%@", code, desc);
}];
```



```
class Callback final : public V2TIMCallback {
public:
    using SuccessCallback = std::function<void()>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;

    Callback() = default;
    ~Callback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
```

```
    }

    void OnSuccess() override {
        if (success_callback_) {
            success_callback_();
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
};

V2TIMString userID = "your user id";
V2TIMString userSig = "userSig from your server";
auto callback = new Callback{};
callback->SetCallback(
    [=]() {
        std::cout << "success";
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        std::cout << "failure, code:" << error_code << ", desc:" << error_message.C
        delete callback;
    });
V2TIMManager::GetInstance()->Logout(callback);
```

账号切换

如果您希望在应用中实现账号切换的需求，只需要每次切换账号时调用 `login` 即可。
例如已经登录了 alice，现在要切换到 bob，只需要直接 login bob 即可。login bob 前无需显式调用 logout alice，SDK 内部会自动处理。

Web

最近更新时间：2024-07-10 16:31:08

功能描述

用户登录 Chat 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

注意

登录成功，需等待 SDK 处于 ready 状态后（监听事件 TencentCloudChat.EVENT.SDK_READY）才能调用 sendMessage 等需要鉴权的接口。

默认情况下，不支持多实例登录，即如果此账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 `TencentCloudChat.EVENT.KICKED_OUT`，用户可在监听到事件后做相应处理。

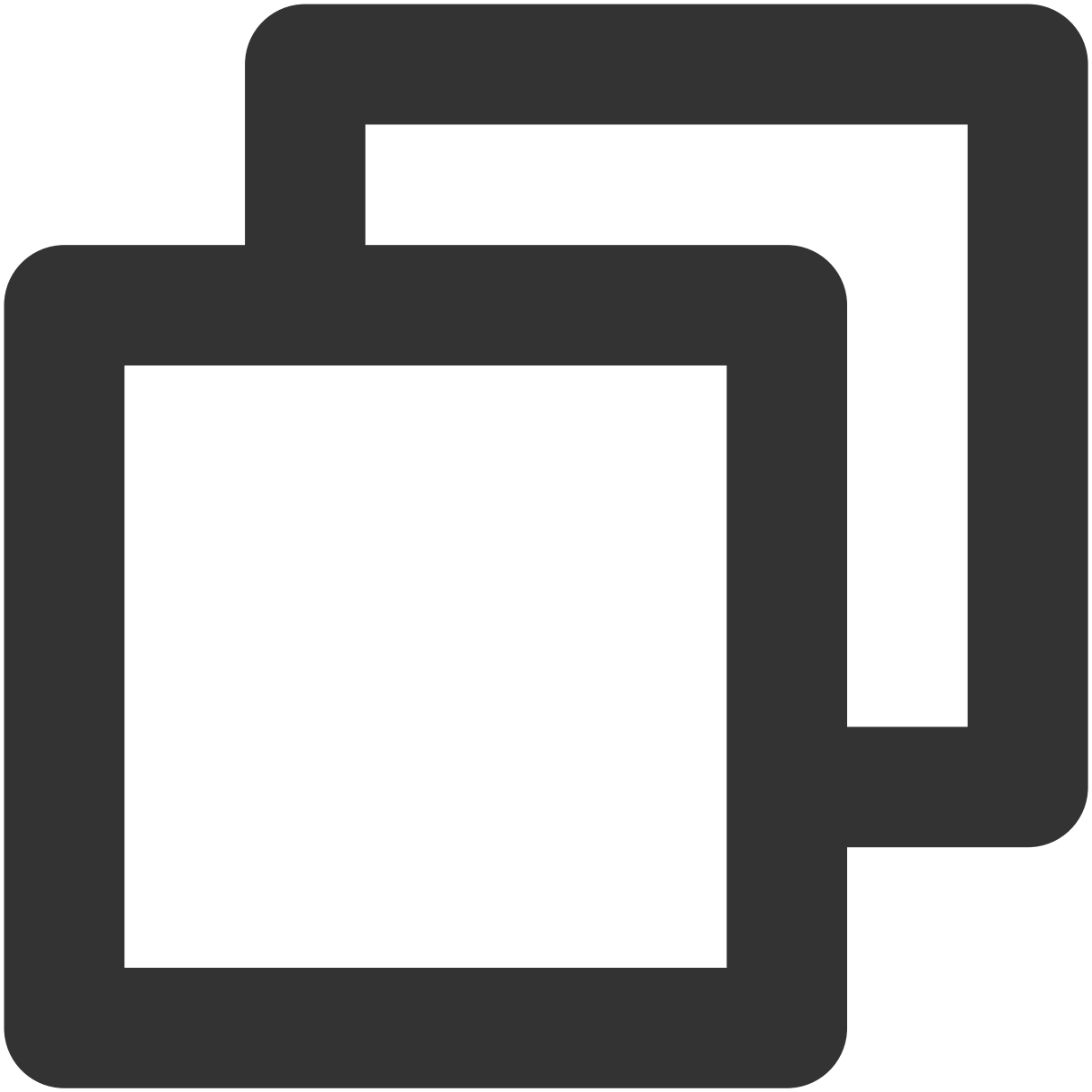
如需支持多实例登录（允许在多个网页中同时登录同一账号），请到 Chat Console，找到相应 SDKAppID，点击 **Application > Configuration > Login and Message > Login Settings > Max Login Instances per User per Platform** 配置实例个数。配置将在50分钟内生效。

小程序、小游戏和 uni-app（即使打包成 native app） 平台集成，登录后都算作 web 实例。

如果此接口返回错误码 60020，意味着您未购买套餐包，或套餐包已过期，或购买的套餐包正在配置中暂未生效。请登录 Chat 购买页面 重新购买套餐包。购买后，将在5分钟后生效。

如果在小程序平台此接口返回错误码 2801，请您检查小程序受信域名配置。

接口



```
chat.login(options);
```

参数

参数 options 为 Object 类型，包含的属性值如下：

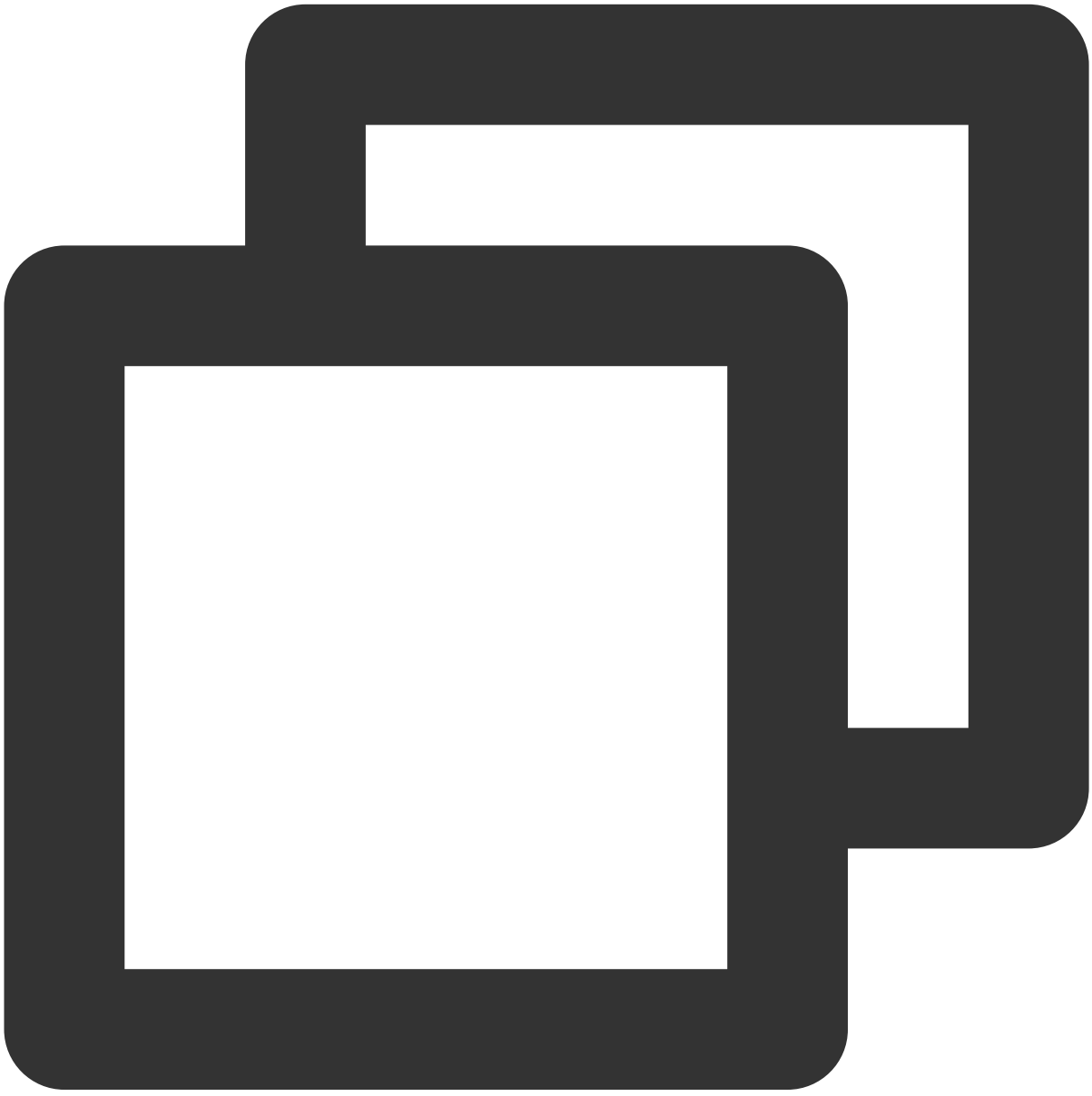
名称	类型	描述
userID	String	用户 ID。
userSig	String	用户登录 Tencent Cloud Chat 的密码，其本质是对 UserID 等信息加

		密后得到的密文。具体生成方法请参见 生成 UserSig。

返回值

`Promise`

示例



```
let promise = chat.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。
    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 Chat SDK，通常在切换账号的时候调用，清除登录态以及内存中的所有数据。

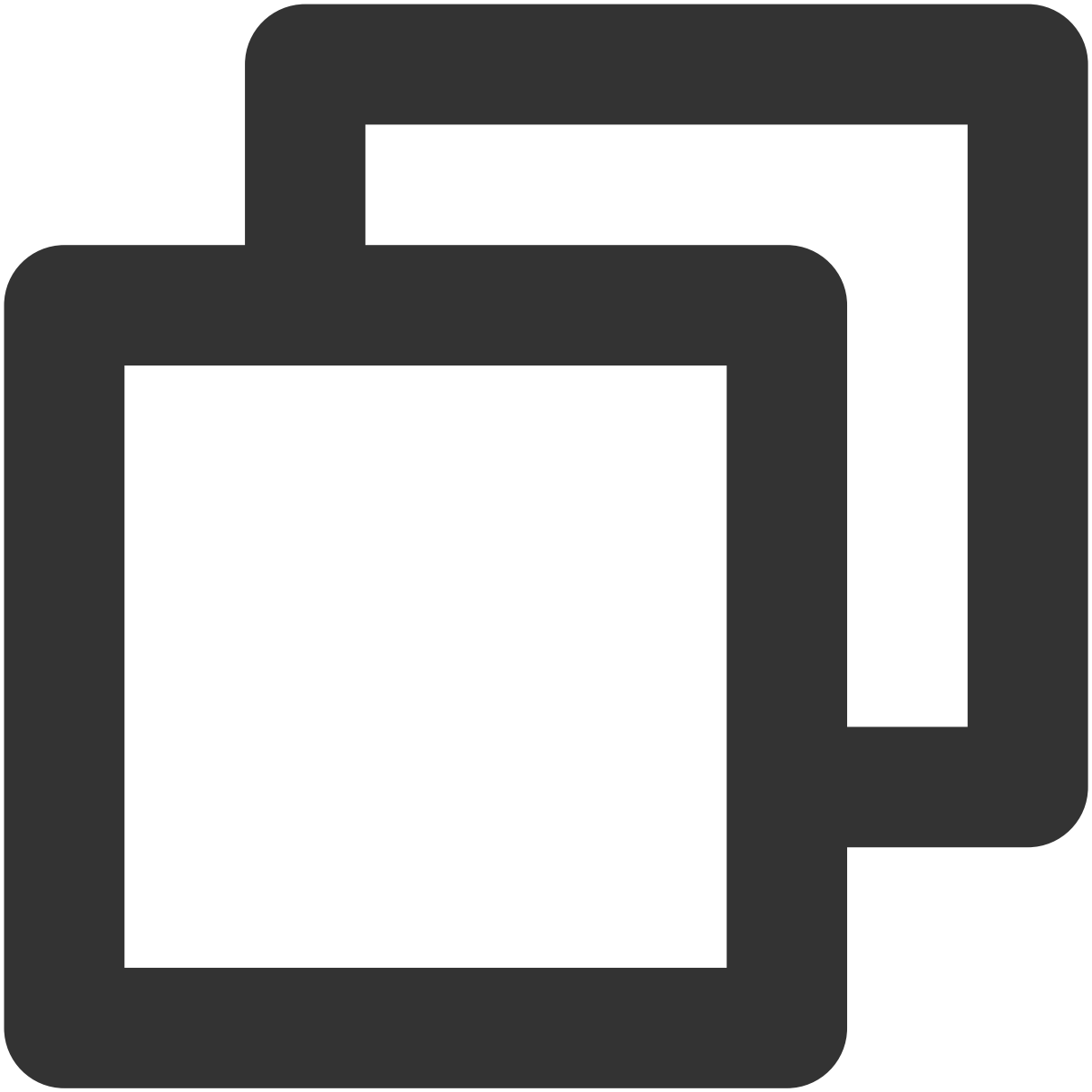
注意

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 Chat Console 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



```
chat.logout();
```

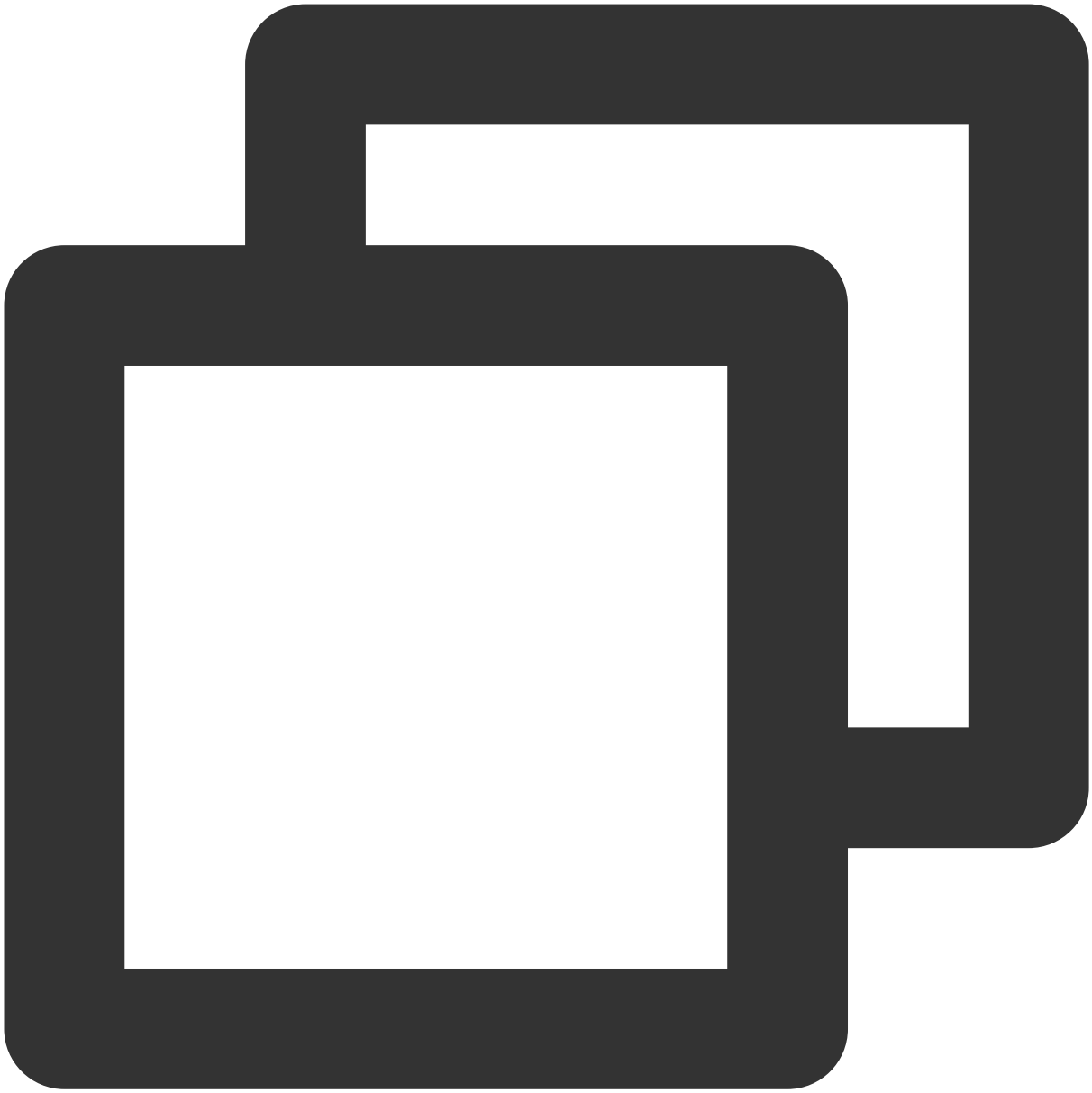
参数

无

返回值

`Promise`

示例

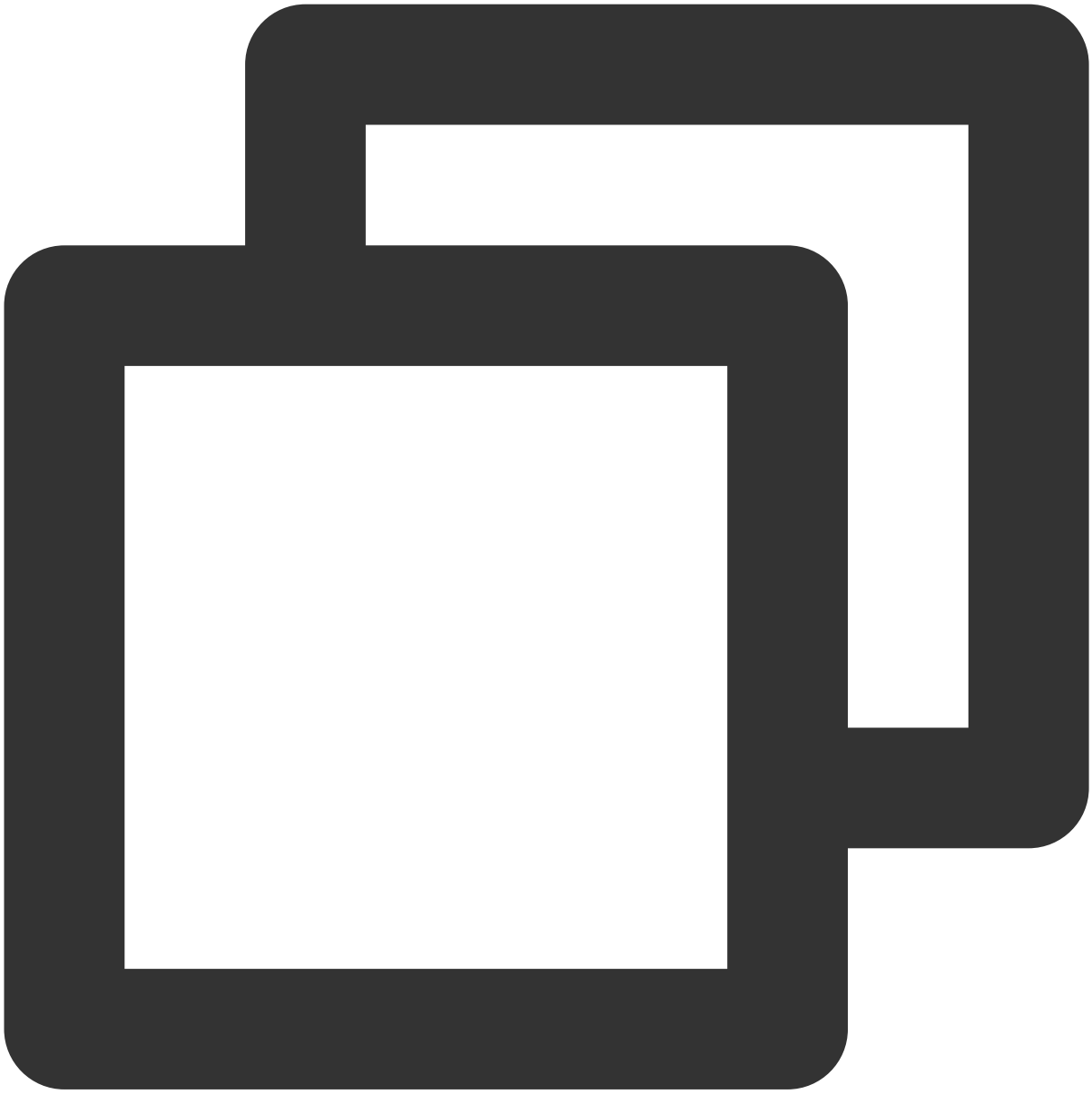


```
let promise = chat.logout();
promise.then(function(imResponse) {
  console.log(imResponse.data); // 登出成功
}).catch(function(imError) {
  console.warn('logout error:', imError);
});
```

销毁

销毁 SDK 实例。SDK 会先 logout，然后断开 WebSocket 长连接，并释放资源。

接口



```
chat.destroy();
```

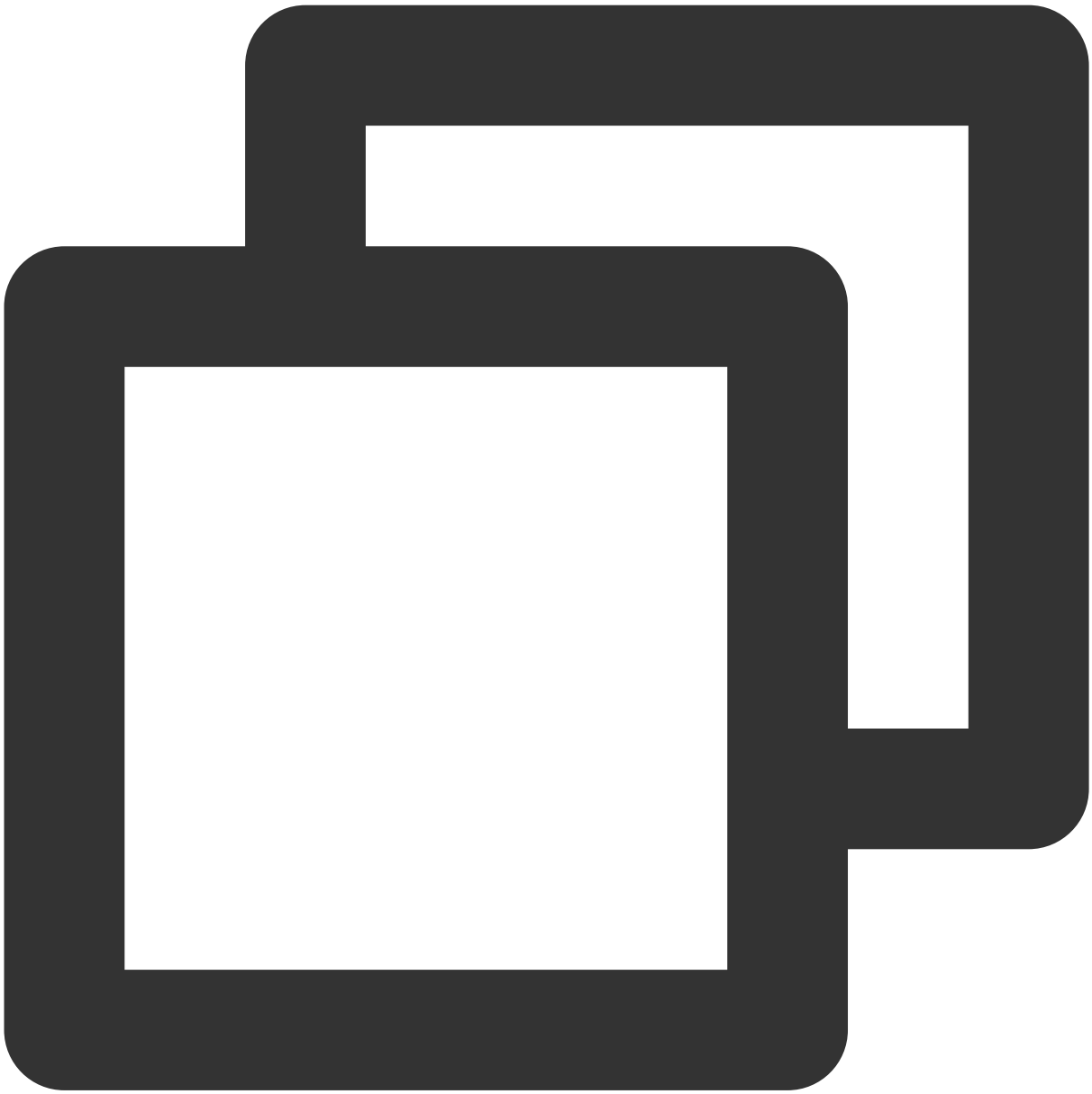
参数

无

返回值

`Promise`

示例



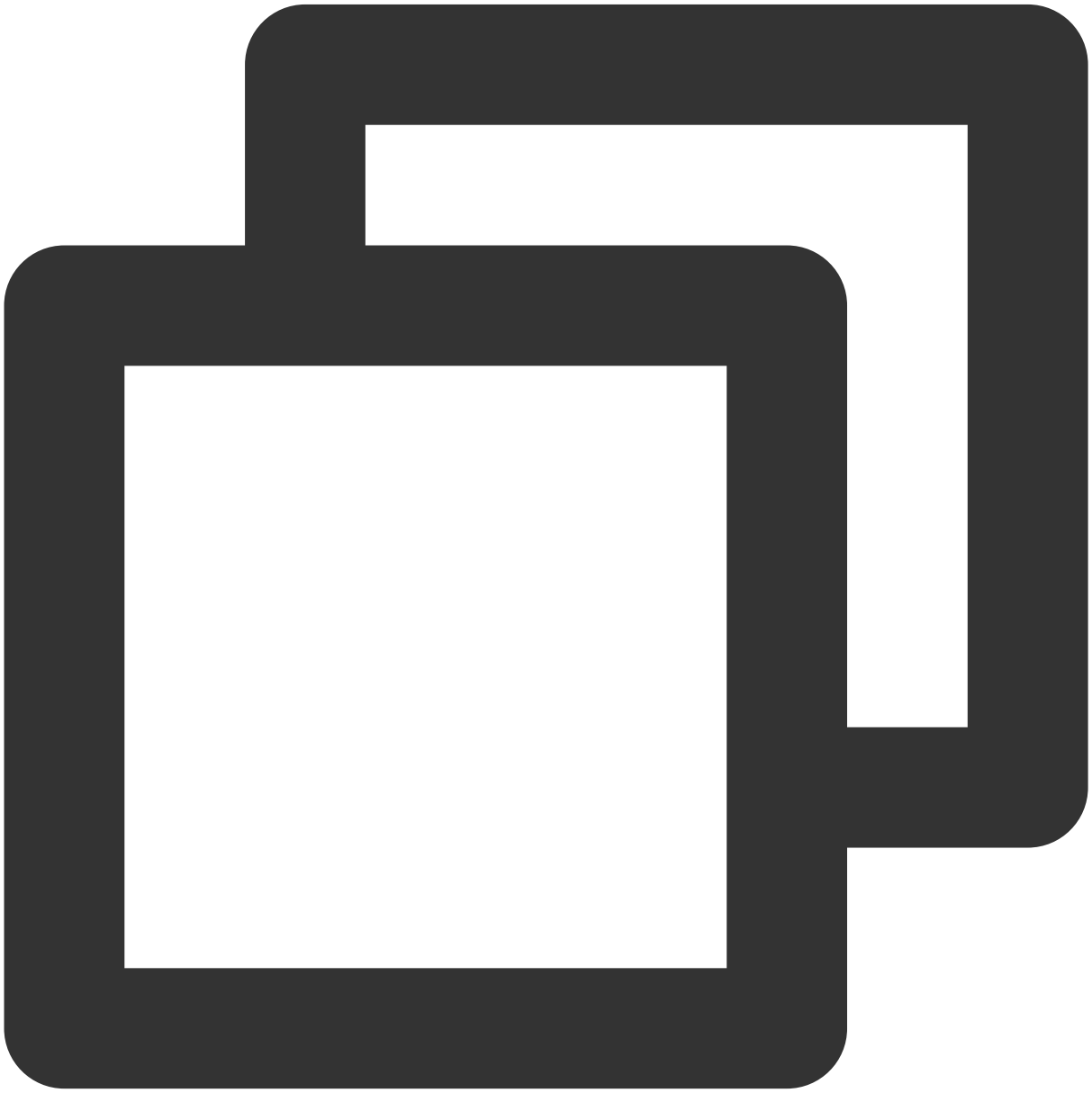
```
let promise = chat.destroy();
```

登录设置

默认情况下，SDK 不支持多实例登录，即如果此已在其他页面登录，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 `TencentCloudChat.EVENT.KICKED_OUT` ，用户可在监听到事件后做相应处理。

注意：

1. 如需支持 Web 多实例登录（允许在多个网页中同时登录同一账号），请到 Chat Console，找到相应 SDKAppID，**应用配置 > 功能配置 > 登录与消息 > Web 端可同时在线个数**配置实例个数。配置将在50分钟内生效。
2. 默认情况下，每种平台只支持1个终端在线（如 Android 和 Android 会互踢）。如果需要使用“同平台多设备在线”功能，请 升级进阶版 ，详见 价格说明。



```
let onKickedOut = function(event) {
```

```
  console.log(event.data.type);
  // TencentCloudChat.TYPES.KICKED_OUT_MULT_ACCOUNT(Web端，同一账号，多页面登录被踢)
  // TencentCloudChat.TYPES.KICKED_OUT_MULT_DEVICE(同一账号，多端登录被踢)
  // TencentCloudChat.TYPES.KICKED_OUT_USERSIG_EXPIRED(签名过期)
  // TencentCloudChat.TYPES.KICKED_OUT_REST_API(REST API kick 接口踢出)
};

chat.on(TencentCloudChat.EVENT.KICKED_OUT, onKickedOut);
```

Flutter

最近更新时间：2024-01-31 15:35:46

功能描述

初始化 IM SDK 后，您需要调用 SDK 登录接口，验证帐号身份，获得帐号的功能使用权限。登录 IM SDK 成功后，才能正常使用消息、会话等功能。

注意：

只有获取会话的接口可以在调用登录接口后立即调用。其他各项功能的接口，必须在 SDK 登录成功后才能调用。因此在使用其他功能之前，**请务必登录且确保登录成功**，否则可能导致功能异常或不可用！

登录

首次登录一个 IM 帐号时，不需要先注册这个帐号。在登录成功后，IM 自动完成这个帐号的注册。

您可以调用 `login` (点击查看详情) 接口进行登录。

`login` 接口的关键参数如下：

参数	含义	说明
UserID	登录用户唯一标识	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（-）。长度不超过 32 字节。
UserSig	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。

您需要在以下场景调用 `login` 接口：

App 启动后首次使用 IM SDK 的功能。

登录时票据过期：`login` 接口的回调会返回 `ERR_USER_SIG_EXPIRED（6206）` 或 `ERR_SVR_ACCOUNT_USERSIG_EXPIRED（70001）` 错误码，此时请您生成新的 userSig 重新登录。

在线时票据过期：用户在线期间也可能收到 `onUserSigExpired` (点击查看详情) 回调，此时需要您生成新的 userSig 并重新登录。

在线时被踢下线：用户在线情况下被踢，IM SDK 会通过 `onKickedOffline` (点击查看详情) 回调通知给您，此时可以在 UI 提示用户，并调用 `login` 重新登录。

以下场景无需调用 `login` 接口：

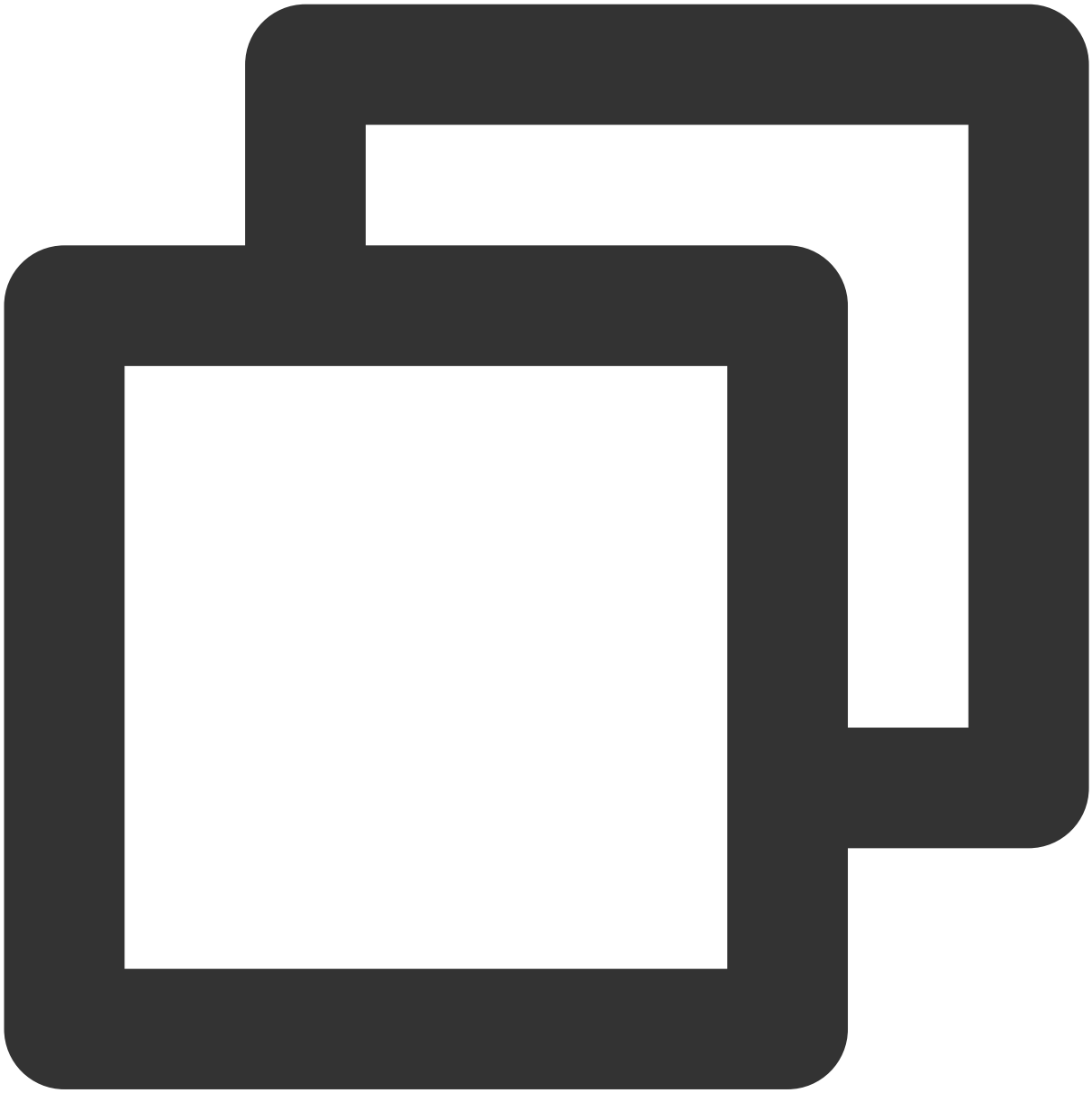
用户的网络断开并重新连接后，不需要调用 `login` 函数，IM SDK 会自动上线。

当一个登录过程在进行时，不需要进行重复登录。

说明：

1. 调用 IM SDK 接口成功登录后，将会开始计算 DAU，请根据业务场景合理调用登录接口，避免出现 DAU 过高的情况。
2. 在一个 App 中，IM SDK 不支持多个帐号同时在线，如果同时登录多个帐号，只有最后登录的帐号在线。

示例代码如下：

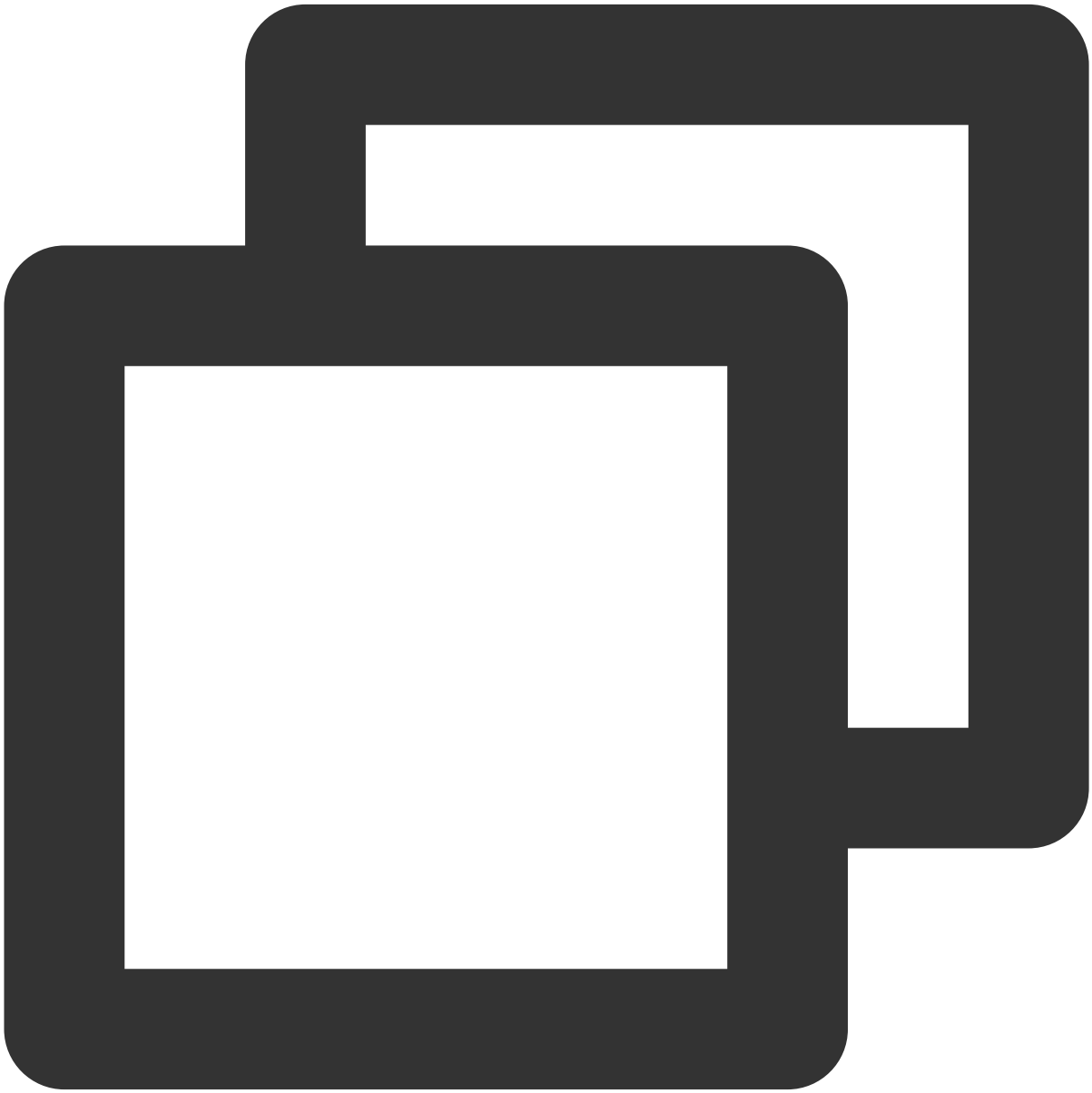


```
String userID = "your user id";
String userSig = "userSig from your server";
V2TimCallback res = await TencentImSDKPlugin.v2TIMManager.login(userID: userID, use
if(res.code == 0){
    // 登录成功逻辑
}else{
```

```
        // 登录失败逻辑
    }
```

获取登录用户

在登录成功后，通过调用 `getLoginUser` (点击查看详情) 获取登录用户 UserID。
如果登录失败，获取的登录用户 UserID 为空。
示例代码如下：



```
    // 在用户登陆成功之后可调用
    // 调用getLoginUser获取登录成功的用户 UserID
```

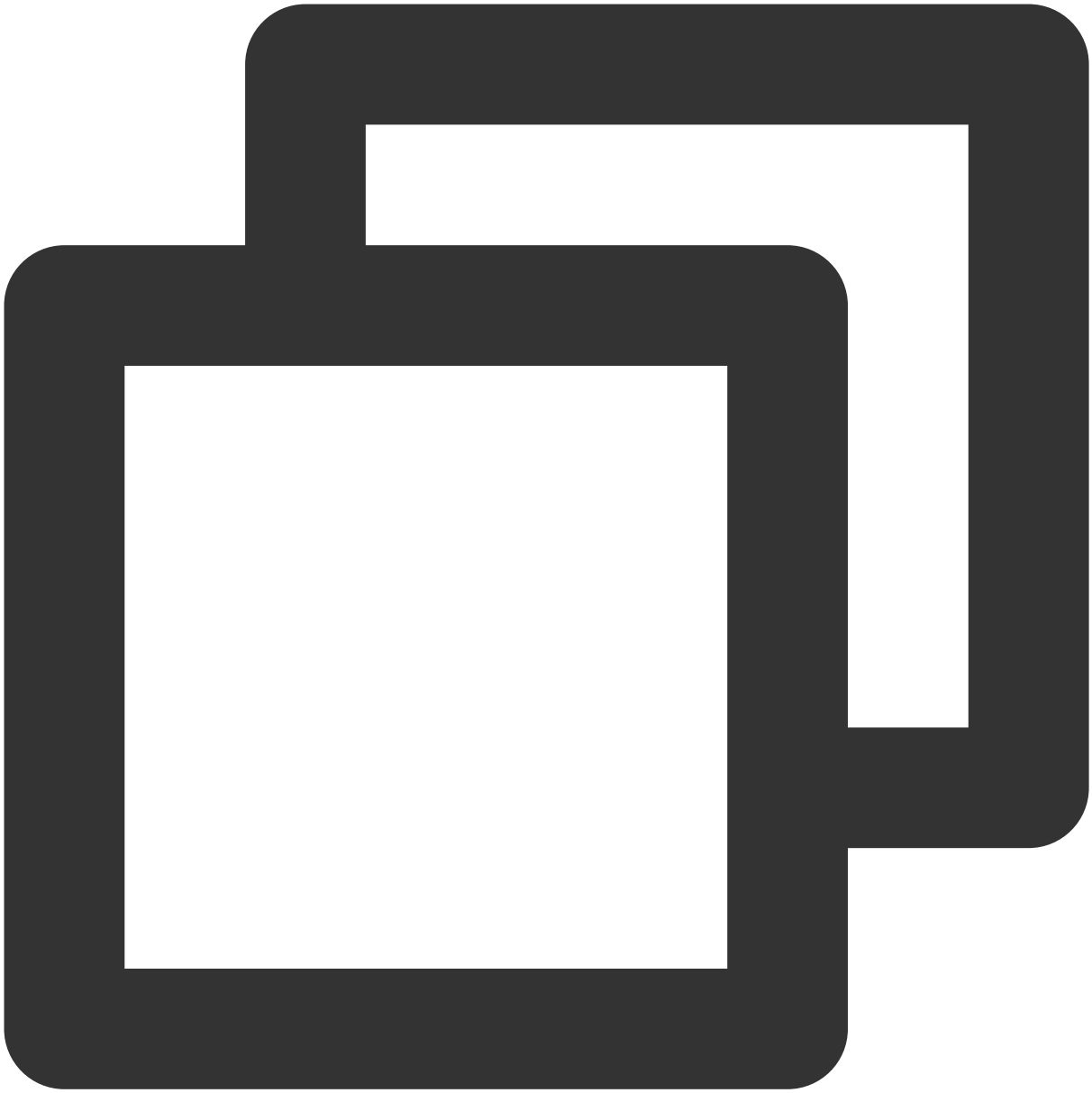
```
V2TimValueCallback<String> getLoginUserRes =
    await TencentImSDKPlugin.v2TIMManager.getLoginUser();
if (getLoginUserRes.code == 0) {
  //获取成功
  getLoginUserRes.data; // getLoginUserRes.data为查询到的登录用户的UserID
}
```

获取登录状态

通过调用 `getLoginStatus` (点击查看详情) 获取登录状态，如果用户已经处于已登录和登录中状态，请勿再频繁调用登录接口登录。IM SDK 支持的登录状态，如下表所示：

登录状态	含义
V2TIM_STATUS_LOGINED (1)	已登录
V2TIM_STATUS_LOGINING (2)	登录中
V2TIM_STATUS_LOGOUT (3)	未登录

示例代码如下：



```
// 在用户登陆成功之后可调用
// 调用getLoginStatus获取登录成功的用户的状态
V2TimValueCallback<int> getLoginStatusRes =
    await TencentImSDKPlugin.v2TIMManager.getLoginStatus();
if (getLoginStatusRes.code == 0) {
  int? status = getLoginStatusRes.data; // getLoginStatusRes.data为用户登录状态值
  if (status == 1) {
    // 已登录
  } else if (status == 2) {
    // 登录中
  } else if (status == 3) {
```

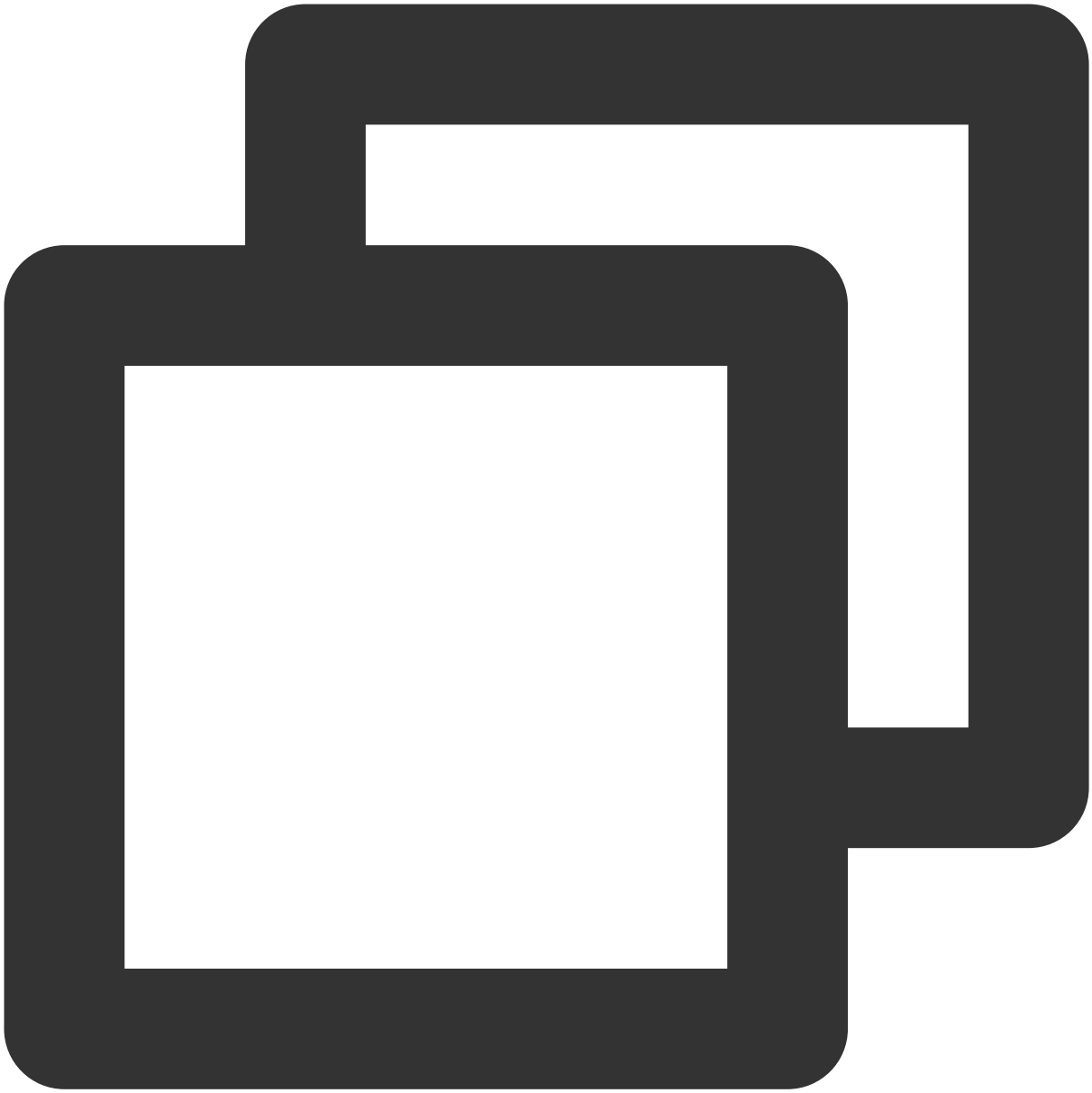
```
            // 未登录
        }
    }
```

多端登录与互踢

您可以在腾讯云控制台配置 IM SDK 的多端登录策略。
多端登录策略有多种可选，例如“移动或桌面平台可有1种平台在线+Web可同时在线”、“不同平台均可同时在线“。
相关配置请参考：登录设置。
您可以在腾讯云控制台配置 IM SDK 的同平台可登录实例数配置，即相同平台的设备可支持几个同时在线。
此功能仅限旗舰版使用。目前 Web 端可同时在线个数最多为 10 个。Android、iPhone、iPad、Windows、Mac（Flutter以实际编译结果为准） 平台可同时在线设备个数最多为 3 个。
相关配置请参考：登录设置。
调用 `login` (点击查看详情) 接口时，如果同一个帐号的多端登录策略超出限制，新登录的实例会把之前已登录的实例踢下线。
被踢下线的一方，会收到 `onKickedOffline` (点击查看详情) 回调。

登出

普通情况下，如果您的应用生命周期跟 IM SDK 生命周期一致，退出应用前可以不登出，直接退出即可。
但有些特殊场景，例如您只在进入特定界面后才使用 IM SDK，退出界面后不再使用，此时可以调用 `logout` (点击查看详情) 接口登出 SDK。登出成功后，不会再收到其他人发送的新消息。注意这种情况下，登出成功后还需要调 `unInitSDK` (点击查看详情) 对 SDK 进行反初始化。
示例代码如下：



```
V2TimCallback logoutRes = await TencentImSDKPlugin.v2TIMManager.logout();
if(logoutRes.code == 0){

}
```

帐号切换

如果您希望在应用中实现帐号切换的需求，只需要每次切换帐号时调用 `login` (点击查看详情) 即可。

例如已经登录了 alice，现在要切换到 bob，只需要直接 login bob 即可。login bob 前无需显式调用 logout alice，IM SDK 内部会自动处理。

Unity

最近更新时间：2024-01-31 15:36:14

功能描述

初始化 IM SDK 后，您需要调用 SDK 登录接口，验证帐号身份，获得帐号的功能使用权限。登录 IM SDK 成功后，才能正常使用消息、会话等功能。

注意：

只有获取会话的接口可以在调用登录接口后立即调用。其他各项功能的接口，必须在 SDK 登录成功后才能调用。因此在使用其他功能之前，**请务必登录且确保登录成功**，否则可能导致功能异常或不可用！

登录

首次登录一个 IM 帐号时，不需要先注册这个帐号。在登录成功后，IM 自动完成这个帐号的注册。

您可以调用 `Login` (Details) 接口进行登录。

`Login` 接口的关键参数如下：

参数	含义	说明
user_id	登录用户唯一标识	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（-）。长度不超过 32 字节。
user_sig	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。
callback	异步回调	可以为 NullValueCallback 或者为 ValueCallback `<String>` 。

您需要在以下场景调用 `Login` 接口：

App 启动后首次使用 IM SDK 的功能。

登录时票据过期：`Login` 接口的回调会返回 `ERR_USER_SIG_EXPIRED（6206）` 或 `ERR_SVR_ACCOUNT_USERSIG_EXPIRED（70001）` 错误码，此时请您生成新的 user_sig 重新登录。

在线时票据过期：用户在线期间也可能收到 `UserSigExpiredCallback` (Details) 回调，此时需要您生成新的 user_sig 并重新登录。

在线时被踢下线：用户在线情况下被踢，IM SDK 会通过 `KickedOfflineCallback` (Details) 回调通知给您，此时可以在 UI 提示用户，并调用 `Login` 重新登录。

以下场景无需调用 `Login` 接口：

用户的网络断开并重新连接后，不需要调用 `Login` 函数，IM SDK 会自动上线。

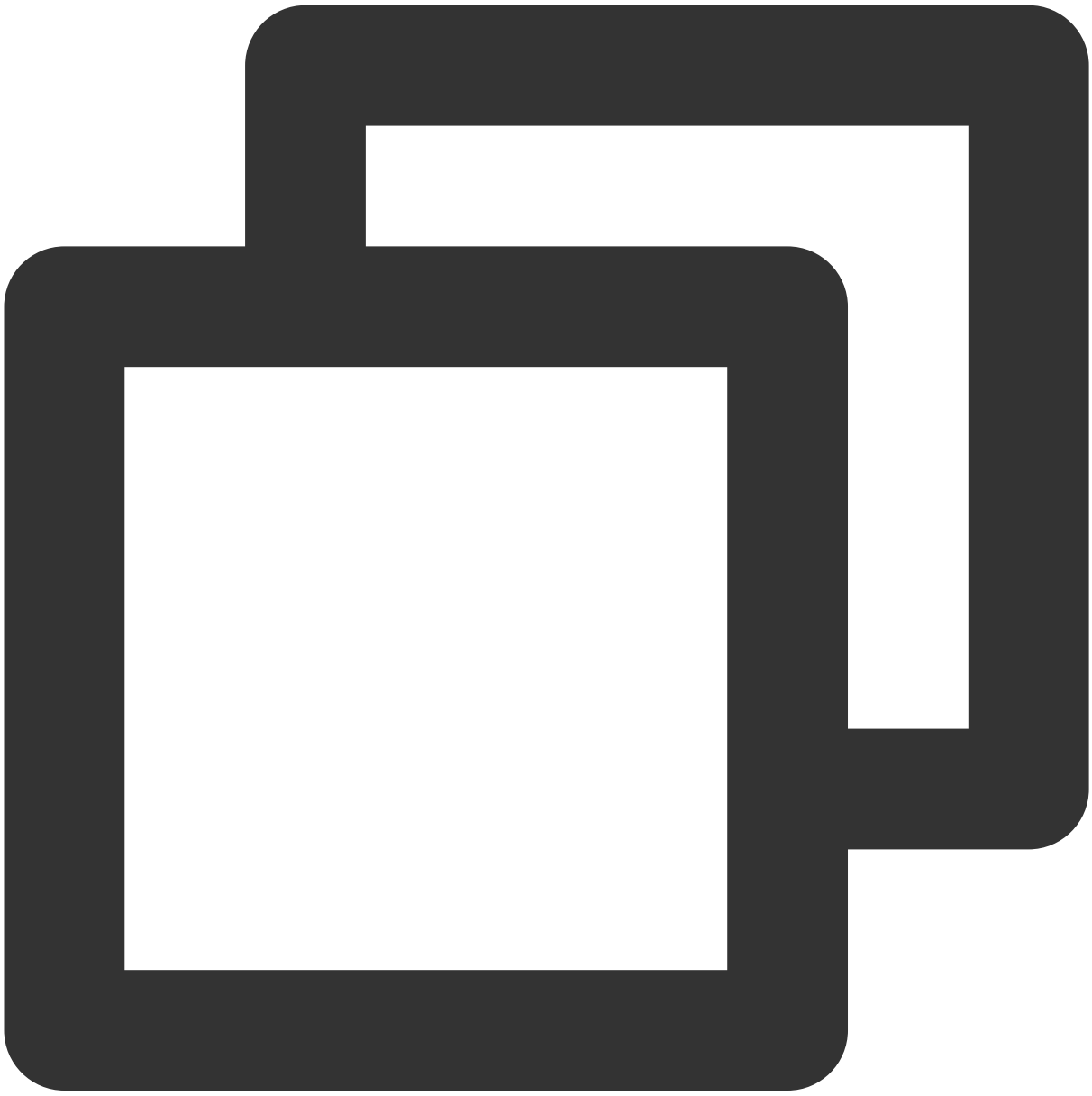
当一个登录过程在进行时，不需要进行重复登录。

版权所有：腾讯云计算（北京）有限责任公司

注意：

1. 调用 IM SDK 接口成功登录后，将会开始计算 DAU，请根据业务场景合理调用登录接口，避免出现 DAU 过高的情况。
2. 在一个 App 中，IM SDK 不支持多个帐号同时在线，如果同时登录多个帐号，只有最后登录的帐号在线。

示例代码如下：



```
public static void Login() {
  if (userid == "" || user_sig == "")
  {
      return;
```

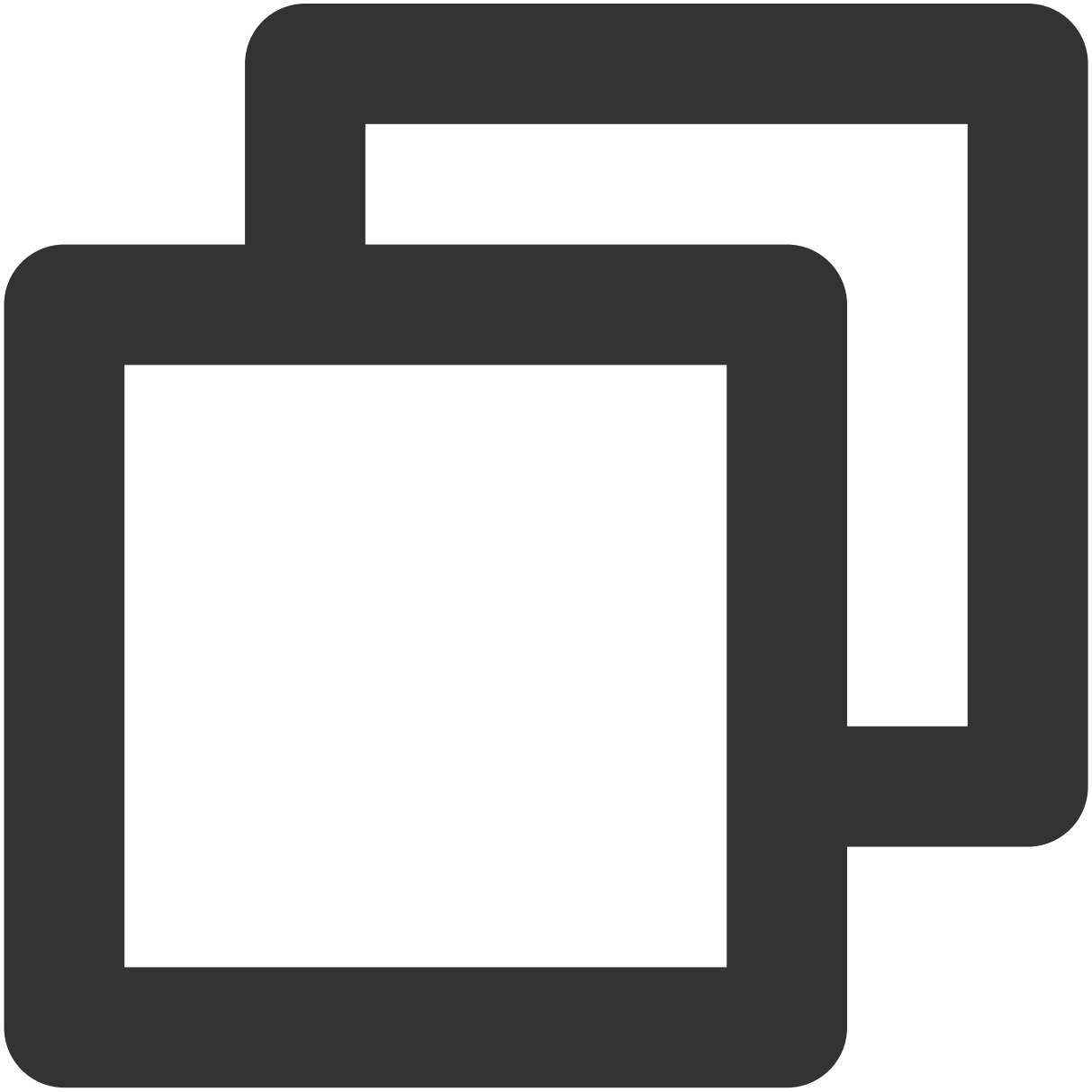
```
    }
    TIMResult res = TencentIMSDK.Login(userid, user_sig, (int code, string desc, stri
      // 处理登陆回调逻辑
    });
```

获取登录用户

在登录成功后，通过调用 `GetLoginUserID` (Details) 获取登录用户 UserID。

如果登录失败，获取的登录用户 UserID 为空。

示例代码如下：



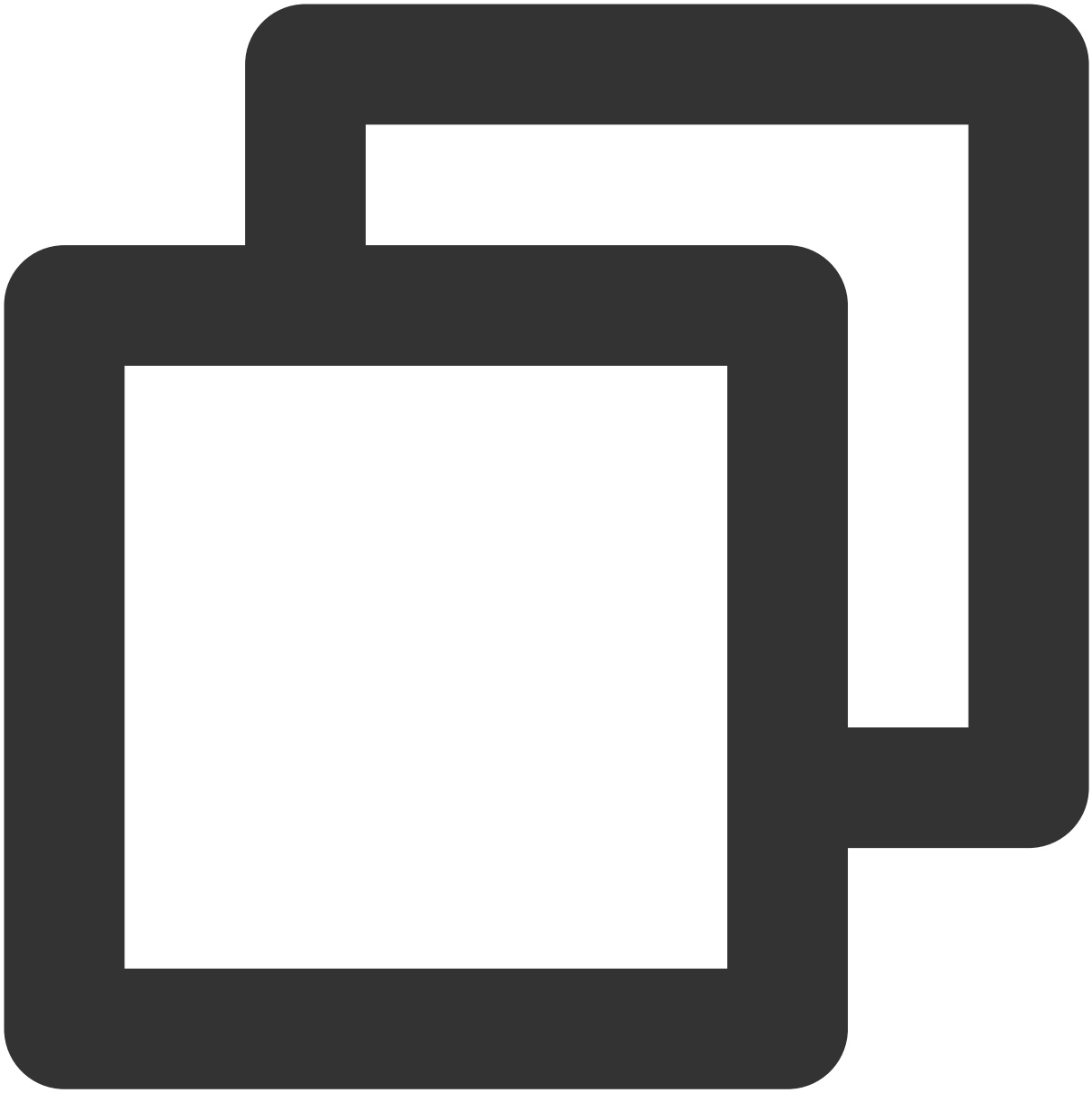
```
public static void GetLoginUserID()
  {
    StringBuilder userId = new StringBuilder(128);

    TIMResult res = TencentIMSDK.GetLoginUserID(userId);

    Debug.Log(userId.ToString());
  }
```

获取登录状态

通过调用 `GetLoginStatus` (Details) 获取登录状态，如果用户已经处于已登录和登录中状态，请勿再频繁调用登录接口登录。IM SDK 支持的登录状态，如下表所示：

登录状态	含义
kTIMLoginStatus_Logined	已登录
kTIMLoginStatus_Logining	登录中
kTIMLoginStatus_Logouting	登出中
kTIMLoginStatus_UnLogined	未登录

示例代码如下：



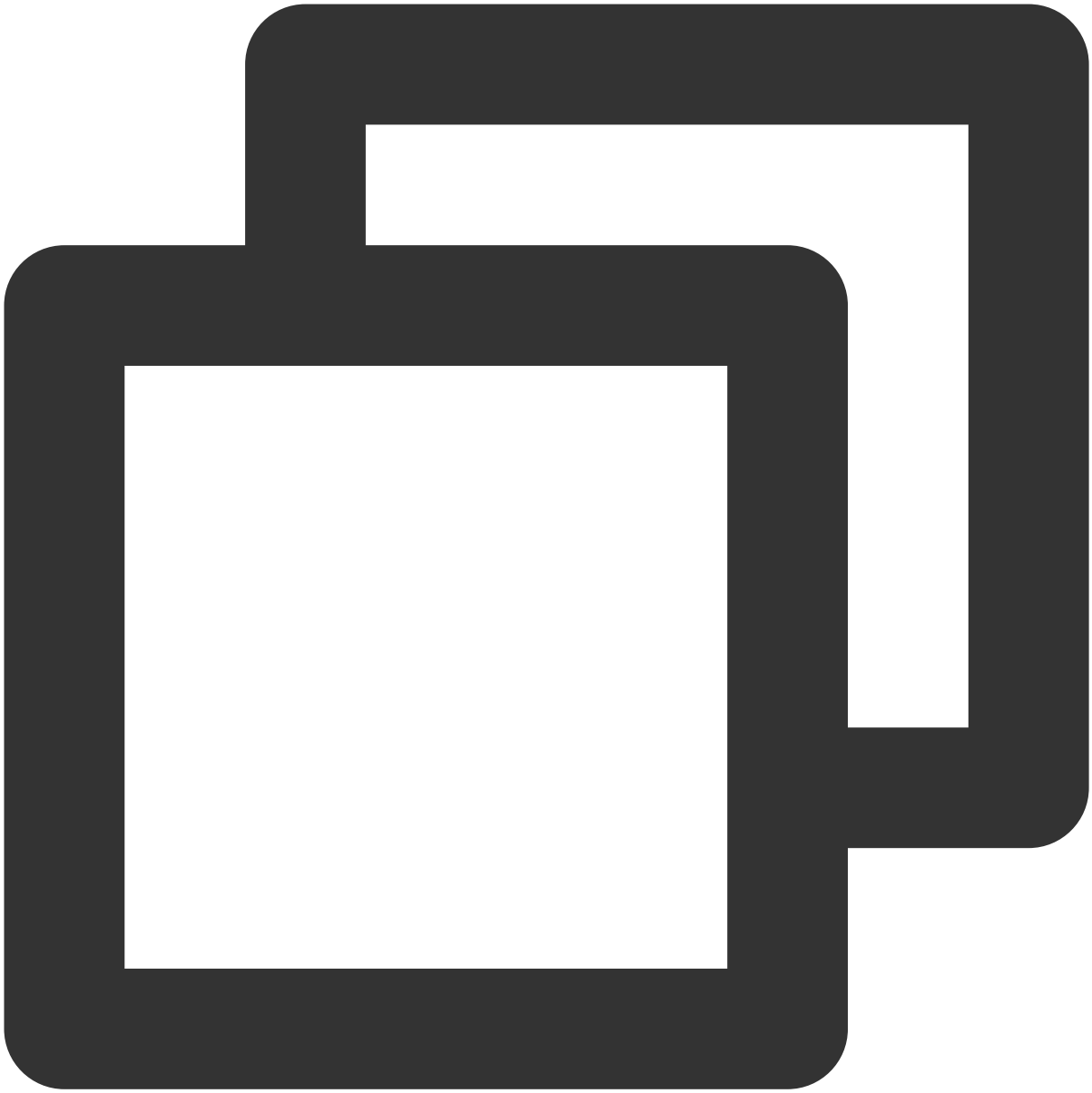
```
public static void GetLoginStatus()
 {
   TIMLoginStatus res = TencentIMSDK.GetLoginStatus();
 }
```

多端登录与互踢

您可以在腾讯云控制台配置 IM SDK 的多端登录策略。
多端登录策略有多种可选，例如“移动或桌面平台可有1种平台在线+Web可同时在线”、“不同平台均可同时在线“。
相关配置请参考：登录设置。
您可以在腾讯云控制台配置 IM SDK 的同平台可登录实例数配置，即相同平台的设备可支持几个同时在线。
此功能仅限旗舰版使用。目前 Web 端可同时在线个数最多为 10 个。Android、iPhone、iPad、Windows、Mac（Flutter以实际编译结果为准） 平台可同时在线设备个数最多为 3 个。
相关配置请参考：登录设置。
调用 `Login` (Details) 接口时，如果同一个帐号的多端登录策略超出限制，新登录的实例会把之前已登录的实例踢下线。
被踢下线的一方，会收到 `KickedOfflineCallback` (Details) 回调。

登出

普通情况下，如果您的应用生命周期跟 IM SDK 生命周期一致，退出应用前可以不登出，直接退出即可。
但有些特殊场景，例如您只在进入特定界面后才使用 IM SDK，退出界面后不再使用，此时可以调用 `Logout` (Details) 接口登出 SDK。登出成功后，不会再收到其他人发送的新消息。注意这种情况下，登出成功后还需要调 `Uninit` (Details) 对 SDK 进行反初始化。
示例代码如下：



```
public static void Logout()
  {
    TIMResult res = TencentIMSDK.Logout((int code, string desc, string json_param,
    // 处理登出回调逻辑
    });
  }
```

帐号切换

如果您希望在应用中实现帐号切换的需求，只需要每次切换帐号时调用 `Login` (Details) 即可。
例如已经登录了 alice，现在要切换到 bob，只需要直接 Login bob 即可。Login bob 前无需显式调用 logout alice，IM SDK 内部会自动处理。

常见问题

1. 在调用登录等其他接口时，发生错误，返回错误码是 6013 和错误描述是 "not initialized" 的信息。

React Native

最近更新时间：2024-01-31 15:36:40

功能描述

初始化 IM SDK 后，您需要调用 SDK 登录接口，验证帐号身份，获得帐号的功能使用权限。登录 IM SDK 成功后，才能正常使用消息、会话等功能。

注意：

只有获取会话的接口可以在调用登录接口后立即调用。其他各项功能的接口，必须在 SDK 登录成功后才能调用。因此在使用其他功能之前，**请务必登录且确保登录成功**，否则可能导致功能异常或不可用！

登录

首次登录一个 IM 帐号时，不需要先注册这个帐号。在登录成功后，IM 自动完成这个帐号的注册。

您可以调用 `login` (Details) 接口进行登录。

`login` 接口的关键参数如下：

参数	含义	说明
UserID	登录用户唯一标识	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（-）。长度不超过 32 字节。
UserSig	登录票据	由您的业务服务器进行计算以保证安全。计算方法请参见 UserSig 后台 API。

您需要在以下场景调用 `login` 接口：

App 启动后首次使用 IM SDK 的功能。

登录时票据过期：`login` 接口的回调会返回 `ERR_USER_SIG_EXPIRED（6206）` 或 `ERR_SVR_ACCOUNT_USERSIG_EXPIRED（70001）` 错误码，此时请您生成新的 userSig 重新登录。

在线时票据过期：用户在线期间也可能收到 `onUserSigExpired` (Details) 回调，此时需要您生成新的 userSig 并重新登录。

在线时被踢下线：用户在线情况下被踢，IM SDK 会通过 `onKickedOffline` (Details) 回调通知给您，此时可以在 UI 提示用户，并调用 `login` 重新登录。

以下场景无需调用 `login` 接口：

用户的网络断开并重新连接后，不需要调用 `login` 函数，IM SDK 会自动上线。

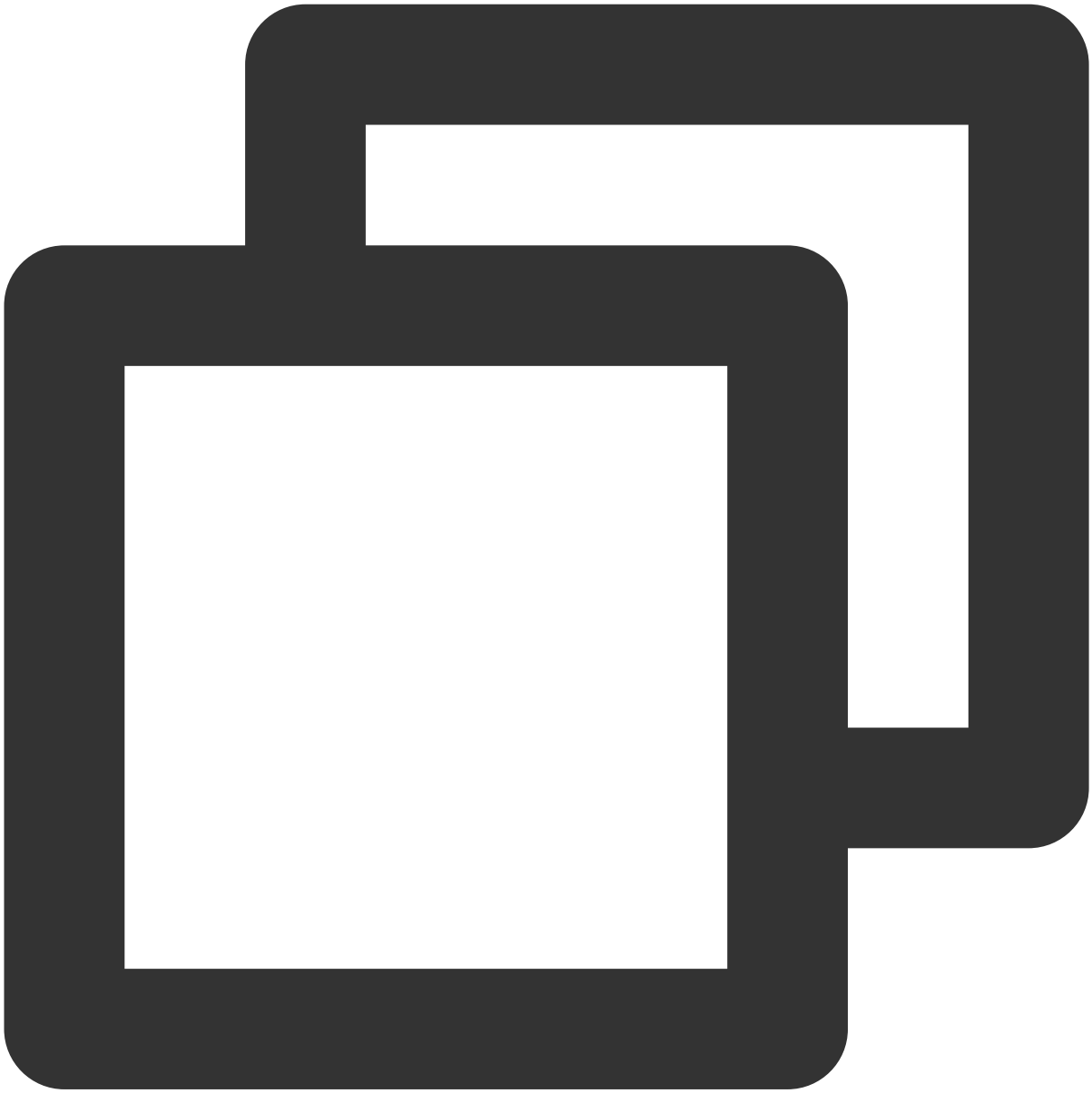
当一个登录过程在进行时，不需要进行重复登录。

说明：

调用 IM SDK 接口成功登录后，将会开始计算 DAU，请根据业务场景合理调用登录接口，避免出现 DAU 过高的情况。

在一个 App 中，IM SDK 不支持多个帐号同时在线，如果同时登录多个帐号，只有最后登录的帐号在线。

示例代码如下：

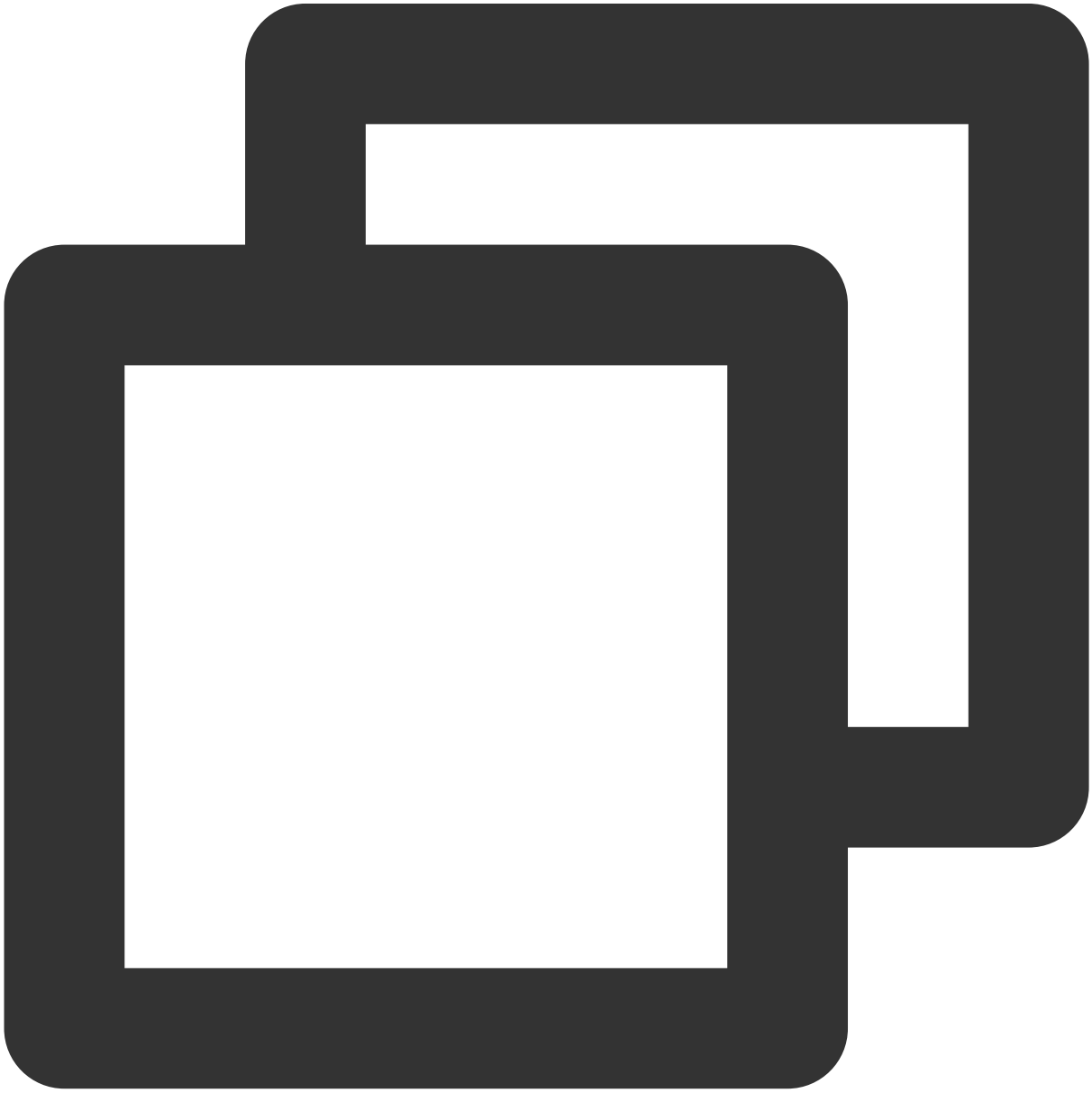


```
import { TencentImSDKPlugin } from "react-native-tim-js";

const userID = "your user id";
const userSig = "userSig from your server";
const res = await TencentImSDKPlugin.v2TIMManager.login(userID, userSig);
if (res.code == 0) {
```

```
    // 登录成功逻辑
} else {
    // 登录失败逻辑
}
```

获取登录用户

在登录成功后，通过调用 `getLoginUser` (Details) 获取登录用户 UserID。
如果登录失败，获取的登录用户 UserID 为空。
示例代码如下：



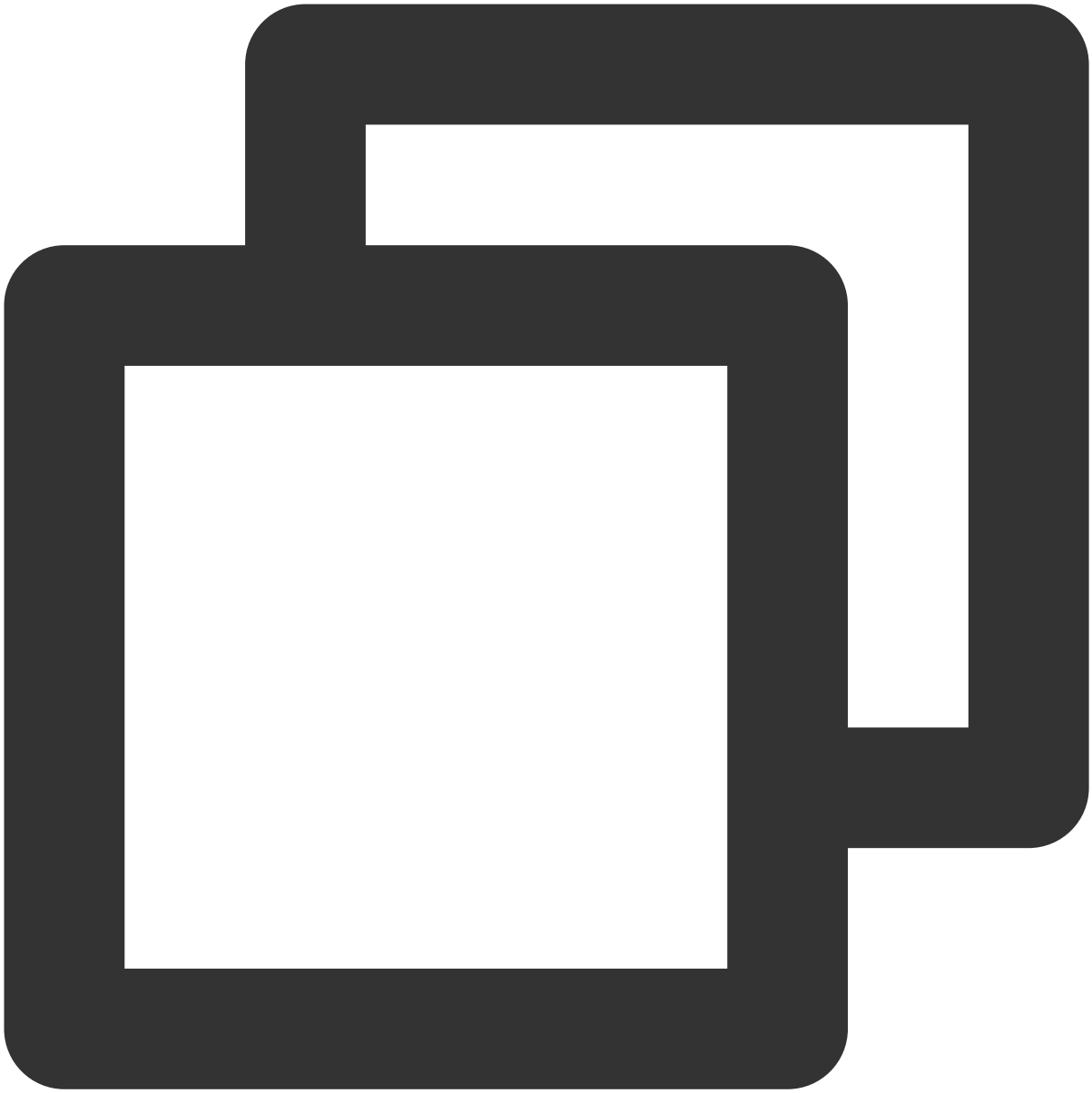
```
import { TencentImSDKPlugin } from "react-native-tim-js";

// 获取登录成功的用户 UserID
const getLoginUserRes = await TencentImSDKPlugin.v2TIMManager.getLoginUser();
if (getLoginUserRes.code == 0) {
  userID = getLoginUserRes.data;
}
```

获取登录状态

通过调用 `getLoginStatus` (Details) 获取登录状态，如果用户已经处于已登录和登录中状态，请勿再频繁调用登录接口登录。IM SDK 支持的登录状态，如下表所示：

登录状态	含义
V2TIM_STATUS_LOGINED (0)	已登录
V2TIM_STATUS_LOGINING (1)	登录中
V2TIM_STATUS_LOGOUT (2)	未登录

示例代码如下：



```
import { TencentImSDKPlugin } from "react-native-tim-js";

const getLoginStatusRes =
  await TencentImSDKPlugin.v2TIMManager.getLoginStatus();
if (getLoginStatusRes.code == 0) {
  const status = getLoginStatusRes.data;
  if (status == 0) {
    // 已登录
  } else if (status == 1) {
    // 登录中
  } else if (status == 2) {
```

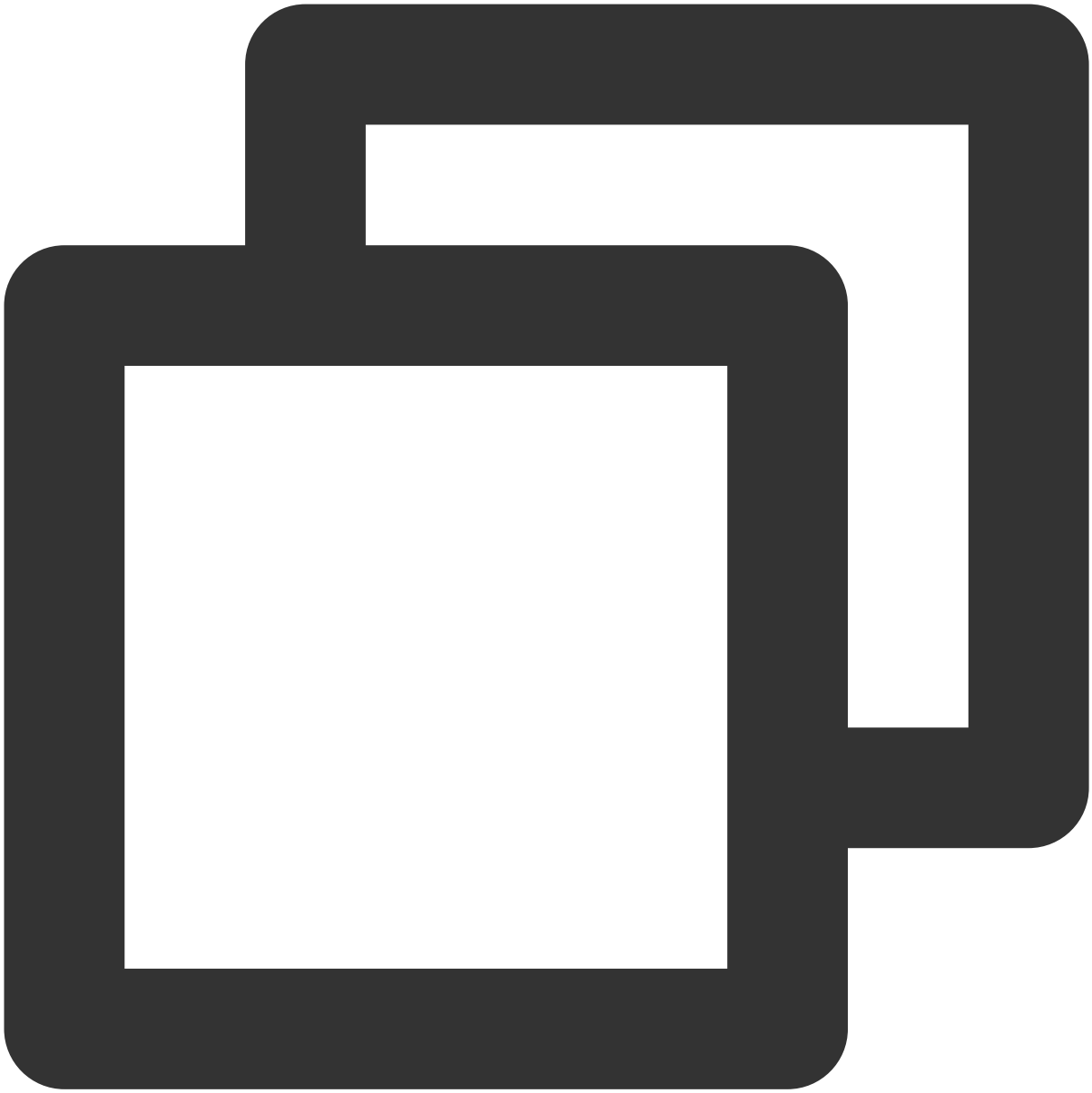
```
        // 未登录
    }
}
```

多端登录与互踢

您可以在腾讯云控制台配置 IM SDK 的多端登录策略。
多端登录策略有多种可选，例如“移动或桌面平台可有 1 种平台在线+Web 可同时在线”、“不同平台均可同时在线“。
相关配置请参见：登录设置。
您可以在腾讯云控制台配置 IM SDK 的同平台可登录实例数配置，即相同平台的设备可支持几个同时在线。
此功能仅限旗舰版使用。目前 Web 端可同时在线个数最多为 10 个。Android、iPhone、iPad、Windows、Mac（Flutter 以实际编译结果为准） 平台可同时在线设备个数最多为 3 个。
相关配置请参见：登录设置。
调用 `login` (Details) 接口时，如果同一个帐号的多端登录策略超出限制，新登录的实例会把之前已登录的实例踢下线。
被踢下线的一方，会收到 `onKickedOffline` (Details) 回调。

登出

普通情况下，如果您的应用生命周期跟 IM SDK 生命周期一致，退出应用前可以不登出，直接退出即可。
但有些特殊场景，例如您只在进入特定界面后才使用 IM SDK，退出界面后不再使用，此时可以调用 `logout` (Details) 接口登出 SDK。登出成功后，不会再收到其他人发送的新消息。注意这种情况下，登出成功后还需要调 `unInitSDK` (Details) 对 SDK 进行反初始化。
示例代码如下：



```
import { TencentImSDKPlugin } from "react-native-tim-js";

const logoutRes = await TencentImSDKPlugin.v2TIMManager.logout();
if (logoutRes.code == 0) {
}
```

帐号切换

如果您希望在应用中实现帐号切换的需求，只需要每次切换帐号时调用 `login` (Details) 即可。

例如已经登录了 alice，现在要切换到 bob，只需要直接 login bob 即可。login bob 前无需显式调用 logout alice，IM SDK 内部会自动处理。

消息相关
消息介绍
Android&iOS&Windows&Mac

最近更新时间：2024-08-01 16:06:00

消息类介绍

在 Chat SDK 中，消息类为 V2TIMMessage (Android / iOS & Mac / Windows) 。您在发送和接收消息过程中都会频繁地用到该类。

V2TIMMessage 类定义了以下内容：

属性	含义	说明
msgID	消息 ID	消息创建的时候为空，调用 `sendMessage` 的时候同步返回。
timestamp	消息时间戳	消息发送到服务端的时间。可用于消息排序。
sender	消息发送者的 userID	客户自己设置，跟登录时传入的 userID 一致。
nickName	消息发送者的昵称	客户自己设置。调用 `setSelfInfo` 设置及修改。详情可参考 用户资料。
friendRemark	消息发送者的好友备注	接收方使用。例如 alice 给好友 bob 备注为 "bob01"。当 bob 给 alice 发消息，此时对于 alice 而言，消息中的 `friendRemark` 为 "bob01"。调用 `setFriendInfo` 设置。
nameCard	发送者的群名片	仅群聊消息有效。例如 alice 修改自己的群名片为 "doctorA"，那么 alice 往群里发送的消息，群成员收到的消息 `nameCard` 字段值为 "doctorA"。接收者可以将这个字段优先作为用户名称的显示。 `nameCard` 需要调用 `setGroupMemberInfo` 设置。详情可参见 群成员资料。
faceURL	消息发送者头像	客户自己设置的头像 URL，可以通过它下载头像图片。
groupID	群组 ID	群聊消息中 `groupID` 为群组 ID；单聊消息中

版权所有：腾讯云计算（北京）有限责任公司

		`groupID` 为 nil。
userID	用户 ID	单聊消息中 `userID` 为对端用户 ID；群聊消息中 `userID` 为 nil。
seq	消息序列号	单聊消息的 `seq` 由本地生成，不能保证严格递增且唯一；群聊消息的 `seq` 由服务器生成，在当前群里的严格递增且唯一的。
random	消息随机码	SDK 内部生成。
status	消息发送状态	目前支持：发送中、发送成功、发送失败、被删除、导入到本地、被撤销。
isSelf	消息发送者是否是自己	可用于消息筛选。
needReadReceipt	消息是否需要已读回执	发送方设置。6.1 以上版本有效，需要购买进阶版。详情可参考 已读回执。
isBroadcastMessage	是否是广播消息	仅直播群支持广播消息。发送广播消息只能使用 REST API，SDK 仅接收。SDK 6.5.2803 增强版及以上版本支持，需要购买进阶版。详情可参见 接收广播消息。
priority	消息优先级	仅群聊消息有效。通过 `sendMessage` 接口设置。
groupAtUserList	群消息被 @ 的用户列表	仅群聊消息有效。列表中存储的是 userID。详情可参考 群 @ 消息。
elemType	消息类型	目前支持：文本、自定义内容、图片、语音、视频、文件、地理位置、表情、群 tips、合并转发消息。详情可参考 消息分类。
textElem	文本消息存储元素	发送文本消息需要创建并填充该元素；收到消息判断 `elemType` 为 TEXT 时，可以从 `textElem` 中解析出内容。详情可参考 发送消息 / 接收消息。
customElem	自定义消息存储元素	发送自定义消息需要创建并填充该元素；收到消息判断 `elemType` 为 CUSTOM 时，可以从 `customElem` 中解析出内容。
imageElem	图片消息存储元素	发送图片消息需要创建并填充该元素；收到消息判断 `elemType` 为 IMAGE 时，可以从 `imageElem` 中解析出内容。
soundElem	语音消息存储	发送语音消息需要创建并填充该元素；收到消息判断

	元素	`elemType` 为 SOUND 时，可以从 soundElem 中解析出内容。
videoElem	视频消息存储元素	发送视频消息需要创建并填充该元素；收到消息判断 `elemType` 为 VIDEO 时，可以从 `videoElem` 中解析出内容。
fileElem	文件消息存储元素	发送文件消息需要创建并填充该元素；收到消息判断 `elemType` 为 FILE 时，可以从 `fileElem` 中解析出内容。
locationElem	地理位置消息存储元素	发送地理位置消息需要创建并填充该元素；收到消息判断 `elemType` 为 LOCATION 时，可以从 `locationElem` 中解析出内容。
faceElem	表情消息存储元素	发送地理位置消息需要创建并填充该元素；收到消息判断 `elemType` 为 FACE 时，可以从 `faceElem` 中解析出内容。
mergerElem	合并消息存储元素	发送合并位置消息需要创建并填充该元素；收到消息判断 `elemType` 为 MERGER 时，可以从 `mergerElem` 中解析出内容。详情可参考 转发消息。
groupTipsElem	群 tips 消息存储元素	群 tips 消息目前只能由服务器发送。SDK 收到消息判断 `elemType` 为 GROUP_TIPS 时，可以从 `groupTipsElem` 中解析出内容。
cloudCustomData	消息自定义数据	发送方设置。内容由用户自定义。云端保存，会发送到对端，程序卸载重装后还能拉取到。
isExcludedFromUnreadCount	消息是否不计入会话未读数	发送方设置。默认为需要计入会话未读数。5.3.425 及以上版本支持。详情可参见 会话未读数。
isExcludedFromLastMessage	消息是否不计入会话 lastMsg	发送方设置。默认为需要计入会话 lastMsg。5.4.666 及以上版本支持。详情可参考 会话列表。
isExcludedFromContentModeration	消息是否不过内容审核（**云端审核**）	发送方设置。默认为需要过内容审核。7.1 及以上版本支持。只有在开通云端审核功能后设置才有效。
hasRiskContent	是否被标记为有安全风险的消息	暂时只支持语音和视频消息，只有在开通云端审核功能后才有效，如果您发送的语音或视频消息内容不合规，云端异步审核后会触发 SDK 的 `onRecvMessageModified` 回调，回调里的 message 对象该字段值为 true。

disableCloudMessagePreHook	是否禁用消息发送前云端回调	发送方设置。默认不禁用消息发送前云端回调。8.1及以上版本支持。
disableCloudMessagePostHook	是否禁用消息发送后云端回调	发送方设置。默认不禁用消息发送后云端回调。8.1及以上版本支持。
isRead	消息是否本端已读	如果是自己发的消息，默认已读。
isPeerRead	消息是否对端已读	仅单聊消息有效。
localCustomData	消息自定义数据	发送方设置。本地保存，不会发送到对端，程序卸载重装后失效。
localCustomInt	消息自定义数据	发送方设置。本地保存，不会发送到对端，程序卸载重装后失效。可以用来标记语音、视频消息是否已经播放。

消息分类

按照消息的发送目标，消息可以分为：“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”、“合并消息”、“群 tips 消息”等几种类型。

消息分类	API 关键词	说明
文本消息	TextElem	普通的文字消息。
自定义消息	CustomElem	一段二进制 buffer，通常用于传输您应用中的自定义信令。
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。

版权所有：腾讯云计算（北京）有限责任公司

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 100 MB。
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。
合并消息	MergerElem	适用于合并转发聊天记录等场景，最大支持 300 条消息合并。

消息存储策略

腾讯云 IM 消息按照消息存储策略，可以分为两种消息：在线消息、非在线消息。 在线消息是指只有当用户在线时才能接收到，离线后不会通过离线推送下发给用户。非在线消息是指无论用户是否在线，都能收到的消息。
在线消息会实时下发，不会存储在服务端。SDK 也不会存储在线消息。所以换设备或卸载后重新安装 App 拉取历史消息都不能拉到此类消息。
说明：
1. 直播群所有的消息都属于在线消息。
2. 全员推送的消息都属于在线消息。
非在线消息会被 SDK 和服务端存储。漫游服务器默认存储 7 天的消息。如果您希望存储超过 7 天，需要购买增值服务。
换设备或卸载后重新安装 App 拉取历史消息可以拉到此类消息。

Web

最近更新时间：2024-07-12 10:44:10

消息类介绍

Chat SDK 中 Message 表示消息对象，用于描述一条消息具有的属性，如类型、消息的内容、所属的会话 ID 等。

属性	类型	默认值	说明
ID	String	-	消息 ID。其 `${sende` Native Cha
type	String	-	消息类型，`Tencent` `Tencent` `Tencent` `Tencent` `Tencent` `Tencent` 息 `Tencent` `Tencent` 息 `Tencent` 息 `Tencent` 群系统通知
payload	Object	-	消息的内容 文本 图片 音频 视频 文件 自定义 合并 地理位置 群提示消息 群系统通知
conversationID	String	-	消息所属的

conversationType	String	-	消息所属会 `Tencent` Client, 端到 `Tencent` GROUP(群 `Tencent` SYSTEM(系
to	String	-	接收方的 us
from	String	-	发送方的 us 用户
flow	String	-	消息的流向 in：收到的 out：发出的
time	Number	-	消息时间戳
status	String	-	消息状态。 unSend：未 success：发 fail：发送失
isRevoked	Boolean	false	是否被撤回
priority	String	`TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL`	消息优先级
nick	String	''	消息发送者
avatar	String	''	消息发送者
isPeerRead	Boolean	false	C2C 消息对
nameCard	String	''	非直播群消 昵称），需
atUserList	Array	[]	群聊时此字
cloudCustomData	String	''	消息自定义 后还能拉取
isDeleted	Boolean	false	是否被删除
isModified	Boolean	false	是否被修改
needReadReceipt	Boolean	false	是否需要已 购买旗舰版

readReceiptInfo	Object	`{ readCount, unreadCount, isPeerRead }`	消息已读回 readCount getMessag 已读了消息 unreadCou getMessag isPeerReac 调用 sendMessa 执通知或拉
isBroadcastMessage	Boolean	false	对所有直播 购买旗舰版
isSupportExtension	Boolean	false	是否支持消 版套餐）
revoker	String | null	null	消息撤回者

Flutter

最近更新时间：2024-01-31 15:44:20

消息类介绍

在腾讯云 IM SDK 中，消息类为 `V2TIimessage` (Details) 。您在发送和接收消息过程中都会频繁地用到该类。`V2TimMessage` 类定义了以下内容：

属性	含义	说明
msgID	消息 ID	消息创建的时候为空，调用 `sendMessage` 的时候同步返回。
timestamp	消息时间戳	消息发送到服务端的时间。可用于消息排序。
sender	消息发送者的 userID	客户自己设置，跟 login 时传入的 userID 一致。
nickName	消息发送者的昵称	客户自己设置。调用 `setSelfInfo` 设置及修改。详情可参考用户资料。
friendRemark	消息发送者的好友备注	接收方使用。例如 alice 给好友 bob 备注为 "bob01"。当 bob 给 alice 发消息，此时对于 alice 而言，消息中的 friendRemark 为 "bob01"。调用 `setFriendInfo` 设置。
nameCard	发送者的群名片	仅群聊消息有效。例如 alice 修改自己的群名片为 "doctorA"，那么 alice 往群里发送的消息，群成员收到的消息 nameCard 字段值为 "doctorA"。接收者可以将这个字段优先作为用户名称的显示。调用 `setGroupMemberInfo` 设置。
faceURL	消息发送者头像	客户自己设置的头像 URL，可以通过它下载头像图片。
groupID	群组 ID	群聊消息中 groupID 为群组 ID；单聊消息中 groupID 为 nil。
userID	用户 ID	单聊消息中 userID 为对端用户 ID；群聊消息中 userID 为 nil。
seq	消息序列号	单聊消息的 seq 由本地生成，不能保证严格递增且唯一；群聊消息的 seq 由服务器生成，在当前群里的严格递增且唯一的。

版权所有：腾讯云计算（北京）有限责任公司

random	消息随机码	SDK 内部生成。
status	消息发送状态	目前支持：发送中、发送成功、发送失败、被删除、导入到本地、被撤销。
isSelf	消息发送者是否是自己	可用于消息筛选。
needReadReceipt	消息是否需要已读回执	发送方设置。6.1 以上版本有效，需要购买旗舰版套餐。详情可参考 已读回执
priority	消息优先级	仅群聊消息有效。通过 `sendMessage` 接口设置。
groupAtUserList	群消息被 @ 的用户列表	仅群聊消息有效。列表中存储的是 userID。详情可参考 群 @ 消息
elemType	消息类型	目前支持：文本、自定义内容、图片、语音、视频、文件、地理位置、表情、群 tips、合并转发消息。详情可参考 消息分类。
textElem	文本消息存储元素	发送文本消息需要创建并填充该元素；收到消息判断 elemType 为 TEXT 时，可以从 textElem 中解析出内容。详情可参考 发送消息 / 接收消息。
customElem	自定义消息存储元素	发送自定义消息需要创建并填充该元素；收到消息判断 elemType 为 CUSTOM 时，可以从 customElem 中解析出内容。
imageElem	图片消息存储元素	发送图片消息需要创建并填充该元素；收到消息判断 elemType 为 IMAGE 时，可以从 imageElem 中解析出内容。
soundElem	语音消息存储元素	发送语音消息需要创建并填充该元素；收到消息判断 elemType 为 SOUND 时，可以从 soundElem 中解析出内容。
videoElem	视频消息存储元素	发送视频消息需要创建并填充该元素；收到消息判断 elemType 为 VIDEO 时，可以从 videoElem 中解析出内容。
fileElem	文件消	发送文件消息需要创建并填充该元素；收到消息判断 elemType

	息存储元素	为 FILE 时，可以从 fileElem 中解析出内容。
locationElem	地理位置消息存储元素	发送地理位置消息需要创建并填充该元素；收到消息判断 elemType 为 LOCATION 时，可以从 locationElem 中解析出内容。
faceElem	表情消息存储元素	发送地理位置消息需要创建并填充该元素；收到消息判断 elemType 为 FACE 时，可以从 faceElem 中解析出内容。
mergerElem	合并消息存储元素	发送合并位置消息需要创建并填充该元素；收到消息判断 elemType 为 MERGER 时，可以从 mergerElem 中解析出内容。详情可参考 转发消息
groupTipsElem	群 tips 消息存储元素	群 tips 消息目前只能由服务器发送。SDK 收到消息判断 elemType 为 GROUP_TIPS 时，可以从 groupTipsElem 中解析出内容。详情可参考 接收消息
cloudCustomData	消息自定义数据	发送方设置。内容由用户自定义。云端保存，会发送到对端，程序卸载重装后还能拉取到。
isExcludedFromUnreadCount	消息是否不计入会话未读数	发送方设置。默认为需要计入会话未读数。5.3.425 及以上版本支持。
isExcludedFromLastMessage	消息是否不计入会话 lastMsg	发送方设置。默认为需要计入会话 lastMsg。5.4.666 及以上版本支持。
offlinePushInfo	自定义离线推送信息	详情可参考 离线推送
isRead	消息是否本端已读	如果是自己发的消息，默认已读。
isPeerRead	消息是否对端已读	仅单聊消息有效。
localCustomData	消息自	发送方设置。本地保存，不会发送到对端，程序卸载重装后失

	定义数据	效。
localCustomInt	消息自定义数据	发送方设置。本地保存，不会发送到对端，程序卸载重装后失效。可以用来标记语音、视频消息是否已经播放。

消息分类

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”、“合并消息” 等几种类型。

消息分类	API 关键词	说明
文本消息	TextElem	普通的文字消息。
自定义消息	CustomElem	一段二进制 buffer，通常用于传输您应用中的自定义信令。
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	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。

版权所有：腾讯云计算（北京）有限责任公司

息		
合并消息	MergerElem	最大支持 300 条消息合并。

消息存储策略

腾讯云 IM 消息按照消息存储策略，可以分为两种消息：在线消息、非在线消息。
在线消息是指只有当用户在线时才能接收到，离线后不会通过离线推送下发给用户。非在线消息是指无论用户是否在线，都能收到的消息。
在线消息会实时下发，不会存储在服务端。SDK 也不会存储在线消息。所以换设备或卸载后重新安装 App 拉取历史消息都不能拉到此类消息。
说明：
1. 直播群所有的消息都属于在线消息。
2. 全员推送的消息都属于在线消息。
非在线消息会被 SDK 和服务端存储。漫游服务器默认存储 7 天的消息。如果您希望存储超过 7 天，需要购买增值服务。服务内容和计费请查看 增值服务资费。
换设备或卸载后重新安装 App 拉取历史消息可以拉到此类消息。

Unity

最近更新时间：2024-01-31 15:45:02

消息类介绍

在腾讯云 IM SDK 中，消息类为 `Message`（点击查看详情）。您在发送和接收消息过程中都会频繁地用到该类。

消息字段含义

字段	含义
message_elem_array	消息内元素列表
message_conv_id	消息所属会话 ID
message_conv_type	消息所属会话类型
message_sender	消息的发送者
message_priority	消息优先级
message_client_time	客户端时间
message_server_time	服务端时间
message_is_from_self	消息是否来自自己
message_platform	发送消息的平台
message_is_read	消息是否已读
message_is_online_msg	消息是否是在线消息，false 表示普通消息，true 表示在线消息，默认为 false
message_is_peer_read	消息是否被会话对方已读
message_need_read_receipt	消息是否需要已读回执（6.1 以上版本有效，需要您购买旗舰版套餐），群消息在使用该功能之前，需要先到 IM 控制台设置已读回执支持的群类型
message_status	消息当前状态
message_target_group_member_array	指定群消息接收成员（定向消息）；不支持群 @ 消息设置，不

	支持社群（Community）和直播群（AVChatRoom）消息设置；该字段设置后，消息会不计入会话未读数。
message_unique_id	消息的唯一标识，推荐使用 kTIMMsgMsgId
message_msg_id	消息的唯一标识
message_rand	消息的随机码
message_has_sent_receipt	是否已经发送了已读回执（只有Group 消息有效）
message_group_receipt_read_count	这个字段是内部字段，不推荐使用，推荐调用 TIMMsgGetMessageReadReceipts 获取群消息已读回执
message_group_receipt_unread_count	这个字段是内部字段，不推荐使用，推荐调用 TIMMsgGetMessageReadReceipts 获取群消息已读回执
message_seq	消息序列
message_custom_int	自定义整数值字段（本地保存，不会发送到对端，程序卸载重装后失效）
message_custom_str	自定义数据字段（本地保存，不会发送到对端，程序卸载重装后失效）
message_cloud_custom_str	消息自定义数据（云端保存，会发送到对端，程序卸载重装后还能拉取到）
message_is_excluded_from_unread_count	消息是否不计入未读计数：默认为 NO，表明需要计入未读计数，设置为 YES，表明不需要计入未读计数
message_is_forward_message	是否是转发消息
message_group_at_user_array	群消息中被 @ 的用户 UserID 列表（即该消息都 @ 了哪些人），如果需要 @ALL，请传入 kImSDK_MesssageAtALL 字段
message_sender_profile	消息的发送者的用户资料
message_sender_group_member_info	消息发送者在群里面的信息，只有在群会话有效。目前仅能获取字段 kTIMGroupMemberInfoIdentifier、kTIMGroupMemberInfoNameCard 其他的字段建议通过 TIMGroupGetMemberInfoList 接口获取
message_offlie_push_config	消息的离线推送设置
message_excluded_from_last_message	是否作为会话的 lasgMessage，true - 不作为，false - 作为

版权所有：腾讯云计算（北京）有限责任公司

消息分类

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，只有接收者可以收到该消息
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”、“图片消息”、“视频消息”、“语音消息”、“文件消息”、“位置消息”、“合并消息”、“群 Tips 消息”等几种类型。

消息分类	API 关键词	说明
文本消息	kTIMElem_Text	即普通的文字消息
自定义消息	kTIMElem_Custom	即一段二进制 buffer，通常用于传输您应用中的自定义信令
图片消息	kTIMElem_Image	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	kTIMElem_Video	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	kTIMElem_Sound	支持语音是否播放红点展示
文件消息	kTIMElem_File	文件消息最大支持100MB
位置消息	kTIMElem_Location	地理位置消息由位置描述、经度（longitude ）和纬度（latitude）三个字段组成
合并消息	kTIMElem_Merge	最大支持 300 条消息合并
群 Tips 消息	kTIMElem_GroupTips	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

版权所有：腾讯云计算（北京）有限责任公司

消息存储策略

腾讯云 IM 消息按照消息存储策略，可以分为两种消息：在线消息、非在线消息。
在线消息是指只有当用户在线时才能接收到，离线后不会通过离线推送下发给用户。非在线消息是指无论用户是否在线，都能收到的消息。
在线消息会实时下发，不会存储在服务端。SDK 也不会存储在线消息。所以换设备或卸载后重新安装 App 拉取历史消息都不能拉到此类消息。
说明：
1. 直播群所有的消息都属于在线消息。
2. 全员推送的消息都属于在线消息。
非在线消息会被 SDK 和服务端存储。漫游服务器默认存储 7 天的消息。如果您希望存储超过 7 天，需要购买增值服务。服务内容和计费请查看 增值服务资费。
换设备或卸载后重新安装 App 拉取历史消息可以拉到此类消息。例如所有类型的普通消息。

版权所有：腾讯云计算（北京）有限责任公司

React Native

最近更新时间：2024-01-31 15:45:40

消息类介绍

在腾讯云 IM SDK 中，消息类为 `V2TimMessage` (Details) 。您在发送和接收消息过程中都会频繁地用到该类。`V2TimMessage` 类定义了以下内容：

属性	含义	说明
msgID	消息 ID	消息创建的时候为空，调用 `sendMessage` 的时候同步返回。
timestamp	消息时间戳	消息发送到服务端的时间。可用于消息排序。
sender	消息发送者的 userID	客户自己设置，跟 login 时传入的 userID 一致。
nickName	消息发送者的昵称	客户自己设置。调用 `setSelfInfo` 设置及修改。详情可参见用户资料。
friendRemark	消息发送者的好友备注	接收方使用。例如 alice 给好友 bob 备注为 "bob01"。当 bob 给 alice 发消息，此时对于 alice 而言，消息中的 friendRemark 为 "bob01"。调用 `setFriendInfo` 设置。
nameCard	发送者的群名片	仅群聊消息有效。例如 alice 修改自己的群名片为 "doctorA"，那么 alice 往群里发送的消息，群成员收到的消息 nameCard 字段值为 "doctorA"。接收者可以将这个字段优先作为用户名称的显示。调用 `setGroupMemberInfo` 设置。
faceURL	消息发送者头像	客户自己设置的头像 URL，可以通过它下载头像图片。
groupID	群组 ID	群聊消息中 groupID 为群组 ID；单聊消息中 groupID 为 nil。
userID	用户 ID	单聊消息中 userID 为对端用户 ID；群聊消息中 userID 为 nil。
seq	消息序列号	单聊消息的 seq 由本地生成，不能保证严格递增且唯一；群聊消息的 seq 由服务器生成，在当前群里的严格递增且唯一的。

版权所有：腾讯云计算（北京）有限责任公司

random	消息随机码	SDK 内部生成。
status	消息发送状态	目前支持：发送中、发送成功、发送失败、被删除、导入到本地、被撤销。
isSelf	消息发送者是否是自己	可用于消息筛选。
needReadReceipt	消息是否需要已读回执	需要购买旗舰版套餐。详情可参见 已读回执
priority	消息优先级	仅群聊消息有效。通过 `sendMessage` 接口设置。
groupAtUserList	群消息被 @ 的用户列表	仅群聊消息有效。列表中存储的是 userID。详情可参见 群 @ 消息
elemType	消息类型	目前支持：文本、自定义内容、图片、语音、视频、文件、地理位置、表情、群 tips、合并转发消息。详情可参见 消息分类。
textElem	文本消息存储元素	发送文本消息需要创建并填充该元素；收到消息判断 elemType 为 TEXT 时，可以从 textElem 中解析出内容。详情可参见 发送消息 / 接收消息。
customElem	自定义消息存储元素	发送自定义消息需要创建并填充该元素；收到消息判断 elemType 为 CUSTOM 时，可以从 customElem 中解析出内容。
imageElem	图片消息存储元素	发送图片消息需要创建并填充该元素；收到消息判断 elemType 为 IMAGE 时，可以从 imageElem 中解析出内容。
soundElem	语音消息存储元素	发送语音消息需要创建并填充该元素；收到消息判断 elemType 为 SOUND 时，可以从 soundElem 中解析出内容。
videoElem	视频消息存储元素	发送视频消息需要创建并填充该元素；收到消息判断 elemType 为 VIDEO 时，可以从 videoElem 中解析出内容。
fileElem	文件消	发送文件消息需要创建并填充该元素；收到消息判断 elemType

	息存储元素	为 FILE 时，可以从 fileElem 中解析出内容。
locationElem	地理位置消息存储元素	发送地理位置消息需要创建并填充该元素；收到消息判断 elemType 为 LOCATION 时，可以从 locationElem 中解析出内容。
faceElem	表情消息存储元素	发送地理位置消息需要创建并填充该元素；收到消息判断 elemType 为 FACE 时，可以从 faceElem 中解析出内容。
mergerElem	合并消息存储元素	发送合并位置消息需要创建并填充该元素；收到消息判断 elemType 为 MERGER 时，可以从 mergerElem 中解析出内容。详情可参见 转发消息
groupTipsElem	群 tips 消息存储元素	群 tips 消息目前只能由服务器发送。SDK 收到消息判断 elemType 为 GROUP_TIPS 时，可以从 groupTipsElem 中解析出内容。详情可参见 接收消息
cloudCustomData	消息自定义数据	发送方设置。内容由用户自定义。云端保存，会发送到对端，程序卸载重装后还能拉取到。
isExcludedFromUnreadCount	消息是否不计入会话未读数	发送方设置。默认为需要计入会话未读数。5.3.425 及以上版本支持。
isExcludedFromLastMessage	消息是否不计入会话 lastMsg	发送方设置。默认为需要计入会话 lastMsg。5.4.666 及以上版本支持。
offlinePushInfo	自定义离线推送信息	详情可参见 离线推送
isRead	消息是否本端已读	如果是自己发的消息，默认已读。
isPeerRead	消息是否对端已读	仅单聊消息有效。
localCustomData	消息自	发送方设置。本地保存，不会发送到对端，程序卸载重装后失

	定义数据	效。
localCustomInt	消息自定义数据	发送方设置。本地保存，不会发送到对端，程序卸载重装后失效。可以用来标记语音、视频消息是否已经播放。

消息分类

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”、“合并消息” 等几种类型。

消息分类	API 关键词	说明
文本消息	TextElem	普通的文字消息。
自定义消息	CustomElem	一段二进制 buffer，通常用于传输您应用中的自定义信令。
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	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。

版权所有：腾讯云计算（北京）有限责任公司

息		
合并消息	MergerElem	最大支持 300 条消息合并。

消息存储策略

腾讯云 IM 消息按照消息存储策略，可以分为两种消息：在线消息、非在线消息。
在线消息是指只有当用户在线时才能接收到，离线后不会通过离线推送下发给用户。非在线消息是指无论用户是否在线，都能收到的消息。
在线消息会实时下发，不会存储在服务端。SDK 也不会存储在线消息。所以换设备或卸载后重新安装 App 拉取历史消息都不能拉到此类消息。
说明：
1. 直播群所有的消息都属于在线消息。
2. 全员推送的消息都属于在线消息。
非在线消息会被 SDK 和服务端存储。漫游服务器默认存储 7 天的消息。如果您希望存储超过 7 天，需要购买增值服务。服务内容和计费请查看 增值服务资费。
换设备或卸载后重新安装 App 拉取历史消息可以拉到此类消息。

发送消息
Android&iOS&Windows&Mac

最近更新时间：2024-04-23 17:59:37

功能描述

发送消息方法在核心类 `V2TIMManager` 和 `V2TIMMessageManager(Android)` / `V2TIMManager(Message)(iOS & Mac)` 中。
支持发送文本、自定义、富媒体消息，消息类型都是 `V2TIMMessage` 。
`V2TIMMessage` 中可以携带 `V2TIMElem` 的不同类型子类，表示不同类型的消息。

重点接口说明

接口 `sendMessage` (Android / iOS & Mac / Windows) 是发送消息中最核心的接口。该接口支持发送所有类型的消息。

说明

下文中提到的发消息高级接口，指的都是 `sendMessage` 。

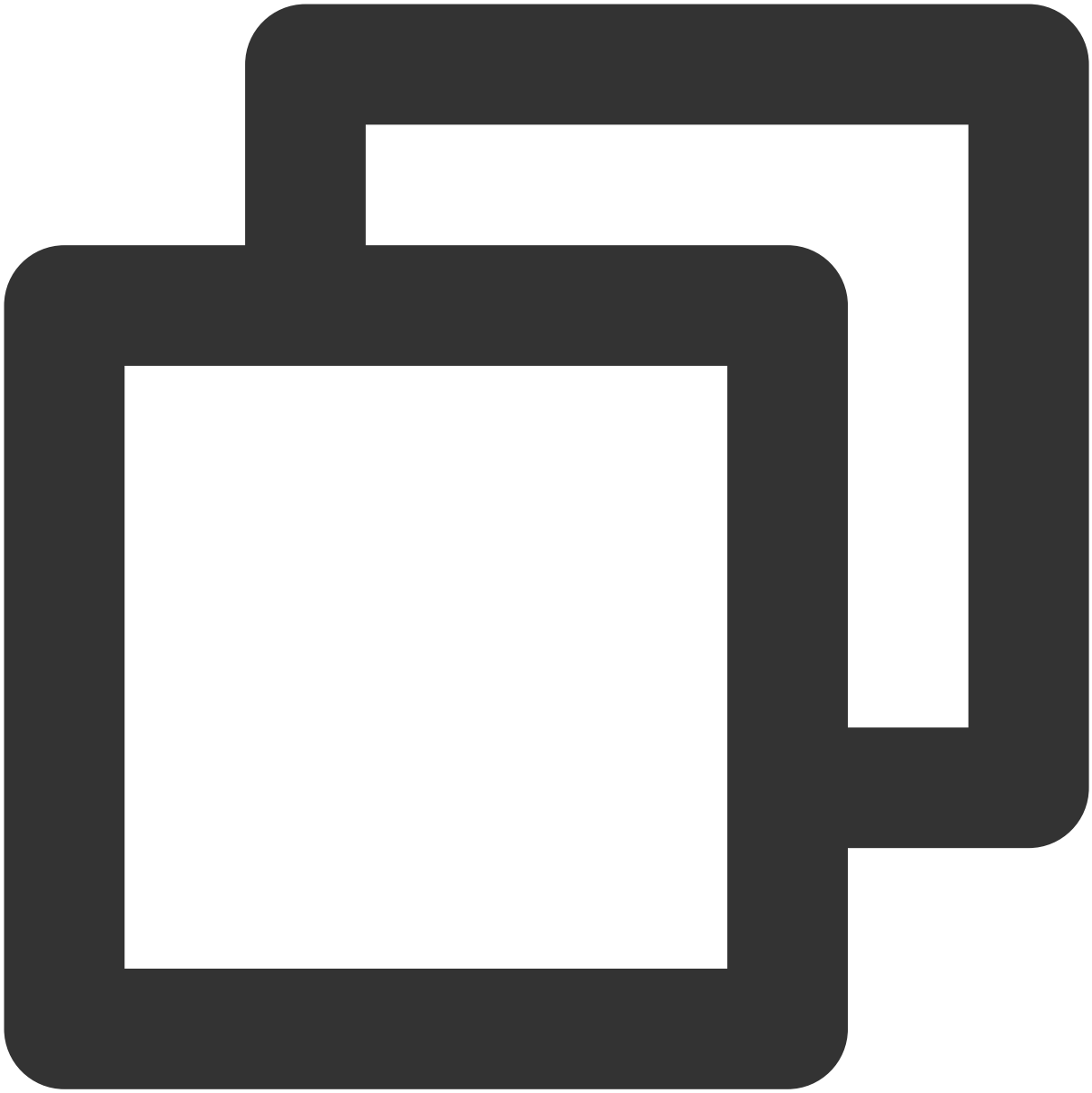
接口说明如下：

Android

iOS & Mac

Windows

方法原型：

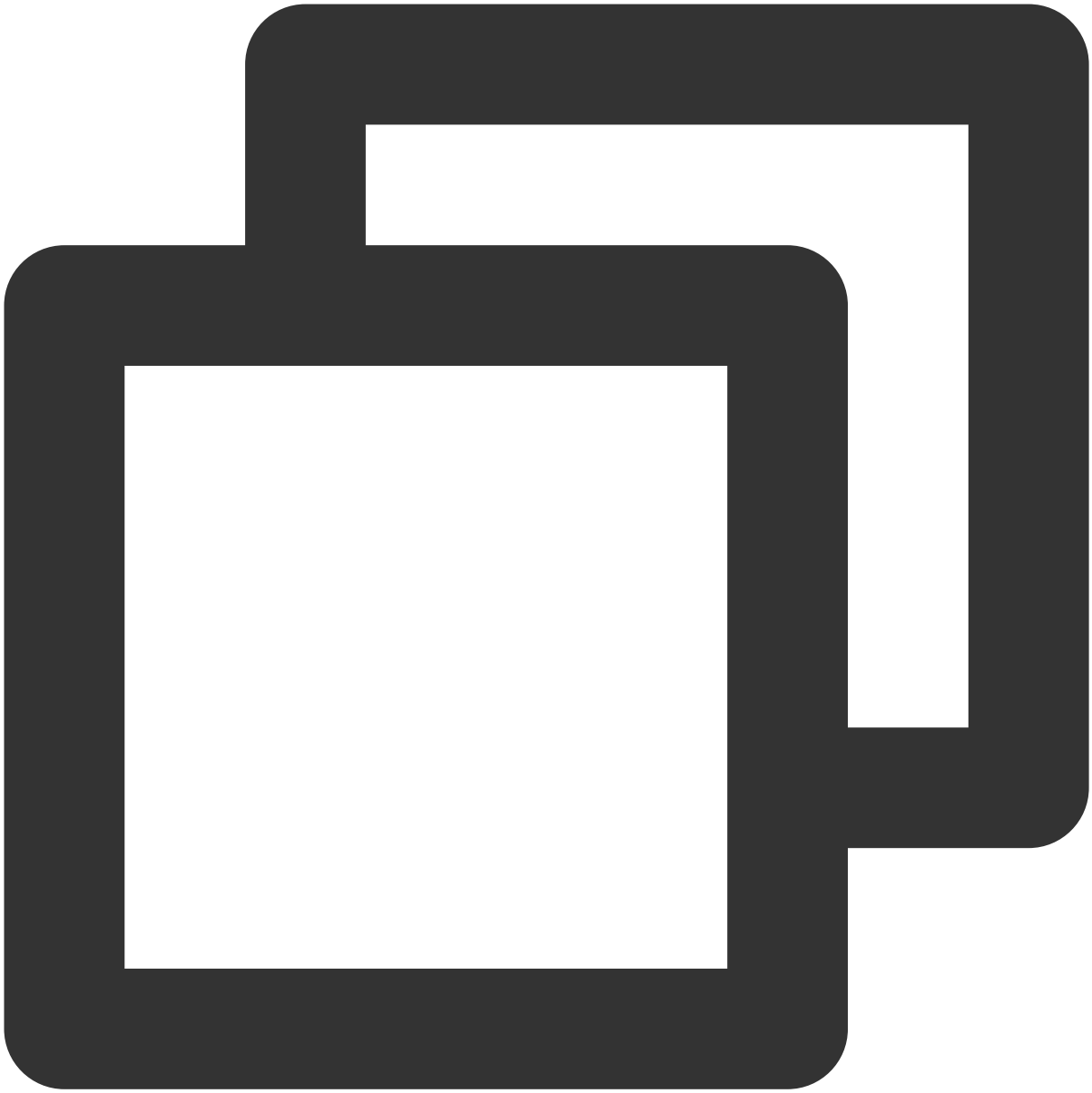


```
// V2TIMMessageManager
public abstract String sendMessage(
    V2TIMMessage message,
    String receiver,
    String groupID,
    int priority,
    boolean onlineUserOnly,
    V2TIMOfflinePushInfo offlinePushInfo,
    V2TIMSendCallback<V2TIMMessage> callback);
```

参数说明：

参数	含义	单聊有效	群聊有效	说明
message	消息对象	YES	YES	需要通过对应的 `createXxxMessage` 接口先行创建，`Xxx` 表示具体的类型。
receiver	单聊消息接收者 userID	YES	NO	如果是发送 C2C 单聊消息，只需要指定 `receiver` 即可。
groupID	群聊 groupID	NO	YES	如果是发送群聊消息，只需要指定 `groupID` 即可。
priority	消息优先级	NO	YES	请把重要消息设置为高优先级（例如红包、礼物消息），高频且不重要的消息设置为低优先级（例如点赞消息）。
onlineUserOnly	是否只有在线用户才能收到	YES	YES	如果设置为 `true`，接收方历史消息拉取不到，常被用于实现 “对方正在输入” 或群组里的非重要提示等弱提示功能。
offlinePushInfo	离线推送信息	YES	YES	离线推送时携带的标题和内容。
callback	发送回调	YES	YES	包含上传进度回调、发送成功回调、发送失败回调。

方法原型：



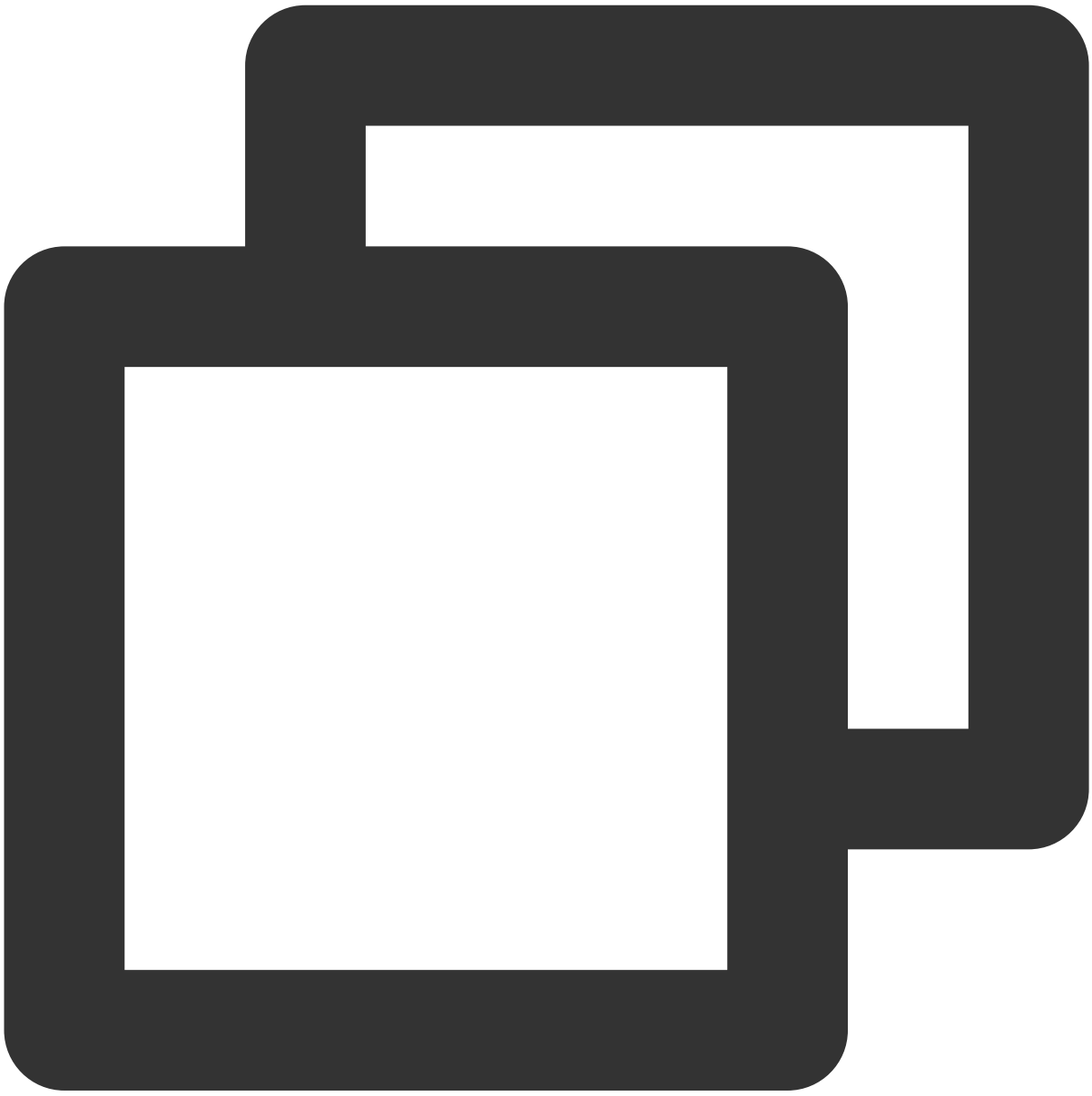
```
// V2TIMManager+Message.h
- (NSString *)sendMessage:(V2TIMMessage *)message
                 receiver:(NSString *)receiver
                  groupID:(NSString *)groupID
                 priority:(V2TIMMessagePriority)priority
           onlineUserOnly:(BOOL)onlineUserOnly
          offlinePushInfo:(V2TIMOfflinePushInfo *)offlinePushInfo
                 progress:(V2TIMProgress)progress
                     succ:(V2TIMSucc)succ
                     fail:(V2TIMFail)fail;
```

参数说明：

参数	含义	单聊有效	群聊有效	说明
message	消息对象	YES	YES	需要通过对应的 `createXxxMessage` 接口先行创建，`Xxx` 表示具体的类型。
receiver	单聊消息接收者 userID	YES	NO	如果是发送 C2C 单聊消息，只需要指定 `receiver` 即可。
groupID	群聊 groupID	NO	YES	如果是发送群聊消息，只需要指定 `groupID` 即可。
priority	消息优先级	NO	YES	请把重要消息设置为高优先级（例如红包、礼物消息），高频且不重要的消息设置为低优先级（例如点赞消息）。
onlineUserOnly	是否只有在线用户才能收到	YES	YES	如果设置为 `YES`，接收方历史消息拉取不到，常被用于实现 “对方正在输入” 或群组里的非重要提示等弱提示功能。
offlinePushInfo	离线推送信息	YES	YES	离线推送时携带的标题和内容。
progress	文件上传进度	YES	YES	文件上传进度。当发送消息中包含图片、语音、视频、文件等富媒体消息时才有效，纯文本、表情、定位消息不会回调。
succ	消息发送成功回调	YES	YES	——

fail	消息发送失败回调	YES	YES	回调失败错误码、错误描述。

方法原型：



```
// V2TIMMessageManager
virtual V2TIMString SendMessage(
    V2TIMMessage& message,
    const V2TIMString& receiver,
```

```
    const V2TIMString& groupID,
    V2TIMMessagePriority priority,
    bool onlineUserOnly,
    const V2TIMOfflinePushInfo& offlinePushInfo,
    V2TIMSendCallback* callback);
```

参数说明：

参数	含义	单聊有效	群聊有效	说明
message	消息对象	YES	YES	需要通过对应的 `CreateXxxMessage` 接口先行创建，`Xxx` 表示具体的类型。
receiver	单聊消息接收者 userID	YES	NO	如果是发送 C2C 单聊消息，只需要指定 `receiver` 即可。
groupID	群聊 groupID	NO	YES	如果是发送群聊消息，只需要指定 `groupID` 即可。
priority	消息优先级	NO	YES	请把重要消息设置为高优先级（例如红包、礼物消息），高频且不重要的消息设置为低优先级（例如点赞消息）。
onlineUserOnly	是否只有在线用户才能收到	YES	YES	如果设置为 `true`，接收方历史消息拉取不到，常被用于实现 “对方正在输入” 或群组里的非重要提示等弱提示功能。
offlinePushInfo	离线推送信息	YES	YES	离线推送时携带的标题和内容。
callback	发送回调	YES	YES	包含上传进度回调、发送成功回调、发送失败回调。

注意

如果 `groupID` 和 `receiver` 同时设置，表示给 `receiver` 发送定向群消息。详情请参考 群定向消息。

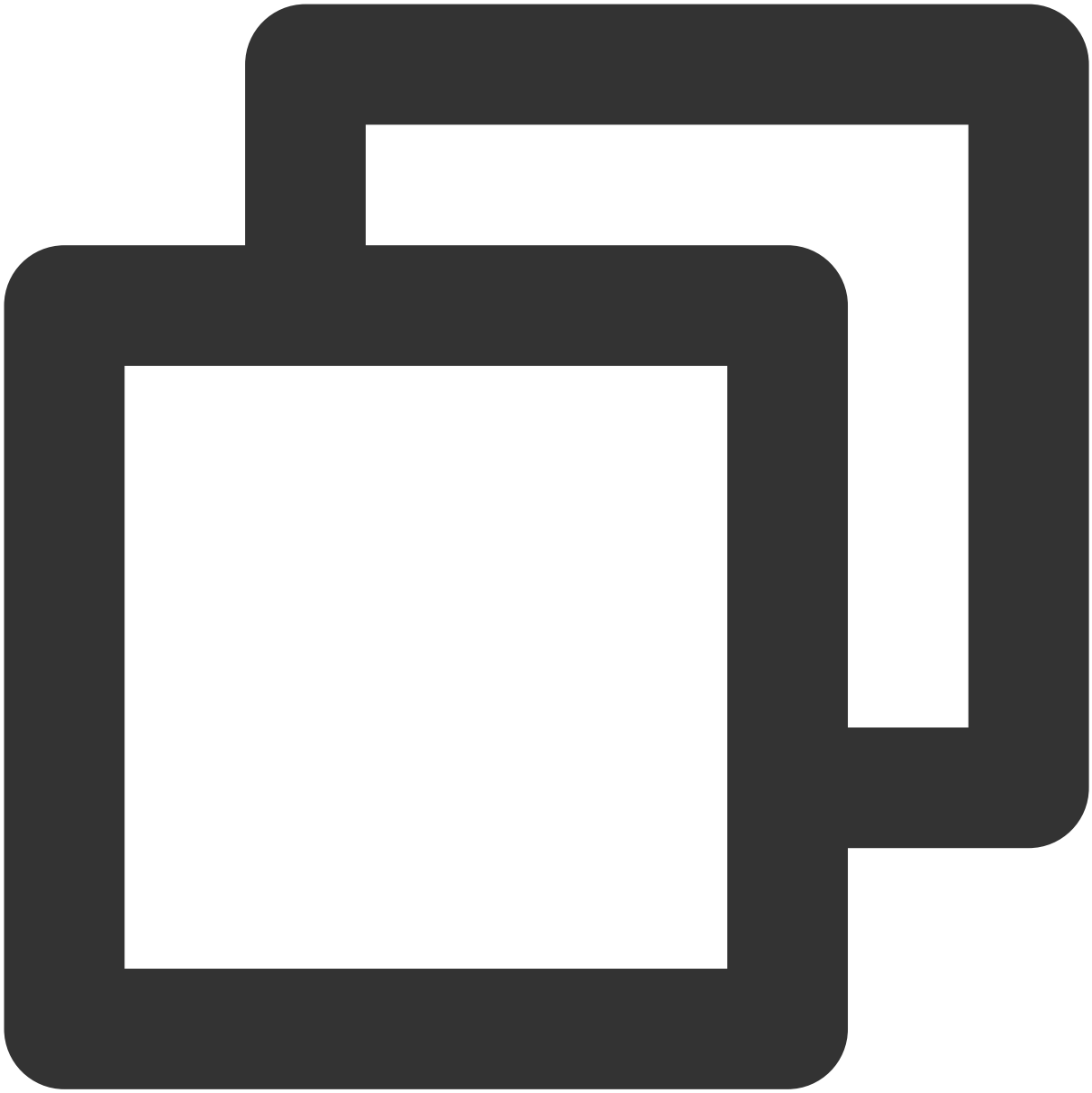
发送文本消息

文本消息区分单聊和群聊，涉及的接口、传参有所区别。
发送文本消息可以采用两种接口：普通接口和高级接口。高级接口比普通接口能设置更多的发送参数（例如优先级、离线推送信息等）。
普通接口参考下文具体描述，高级接口就是上文中提到的 `sendMessage` 。

单聊文本消息

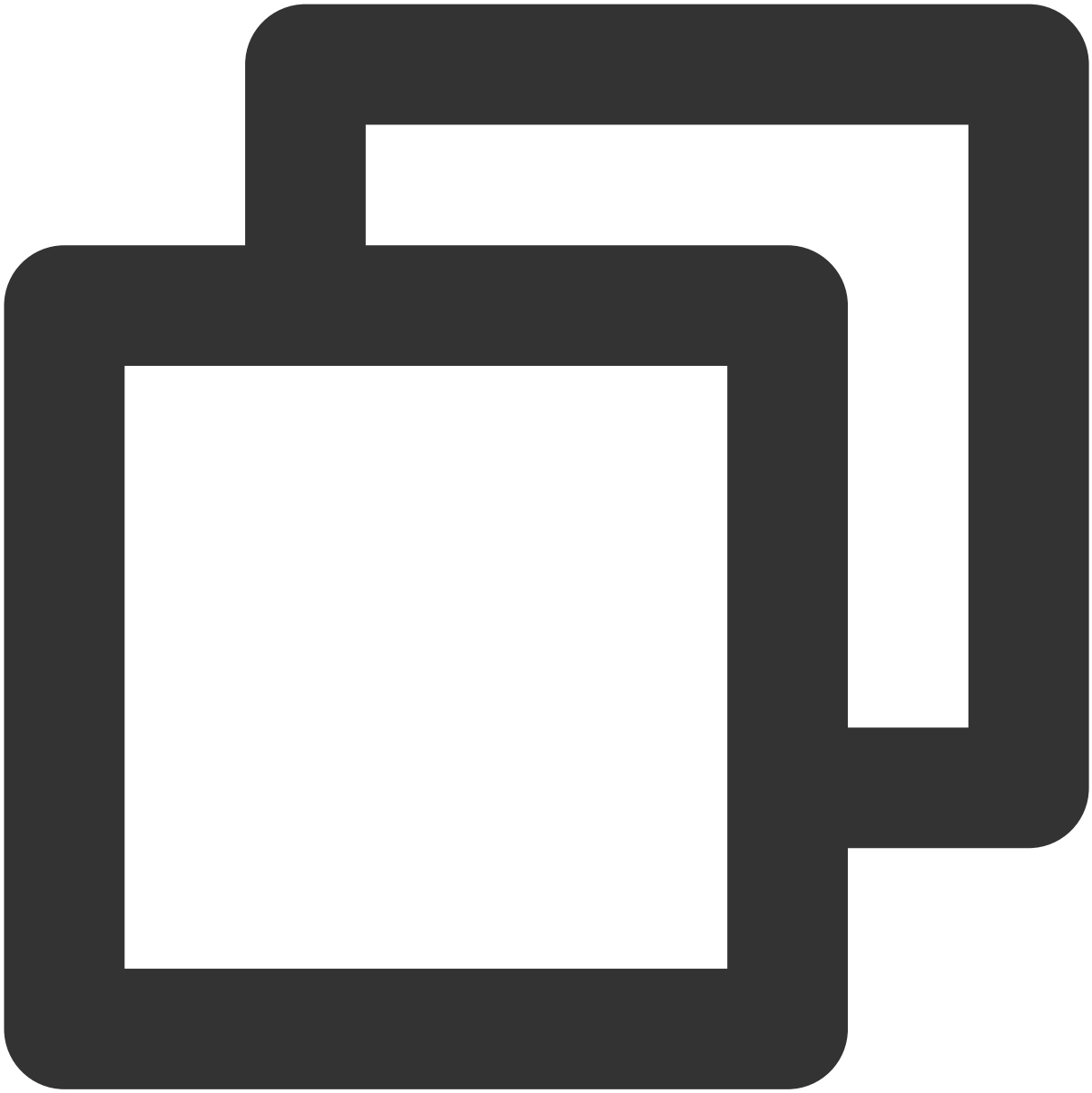
普通接口

调用 `sendC2CTextMessage` (Android / iOS & Mac / Windows) 发送单聊文本消息，直接传入消息内容和接收者的 userID 即可。
示例代码如下：
Android
iOS & Mac
Windows

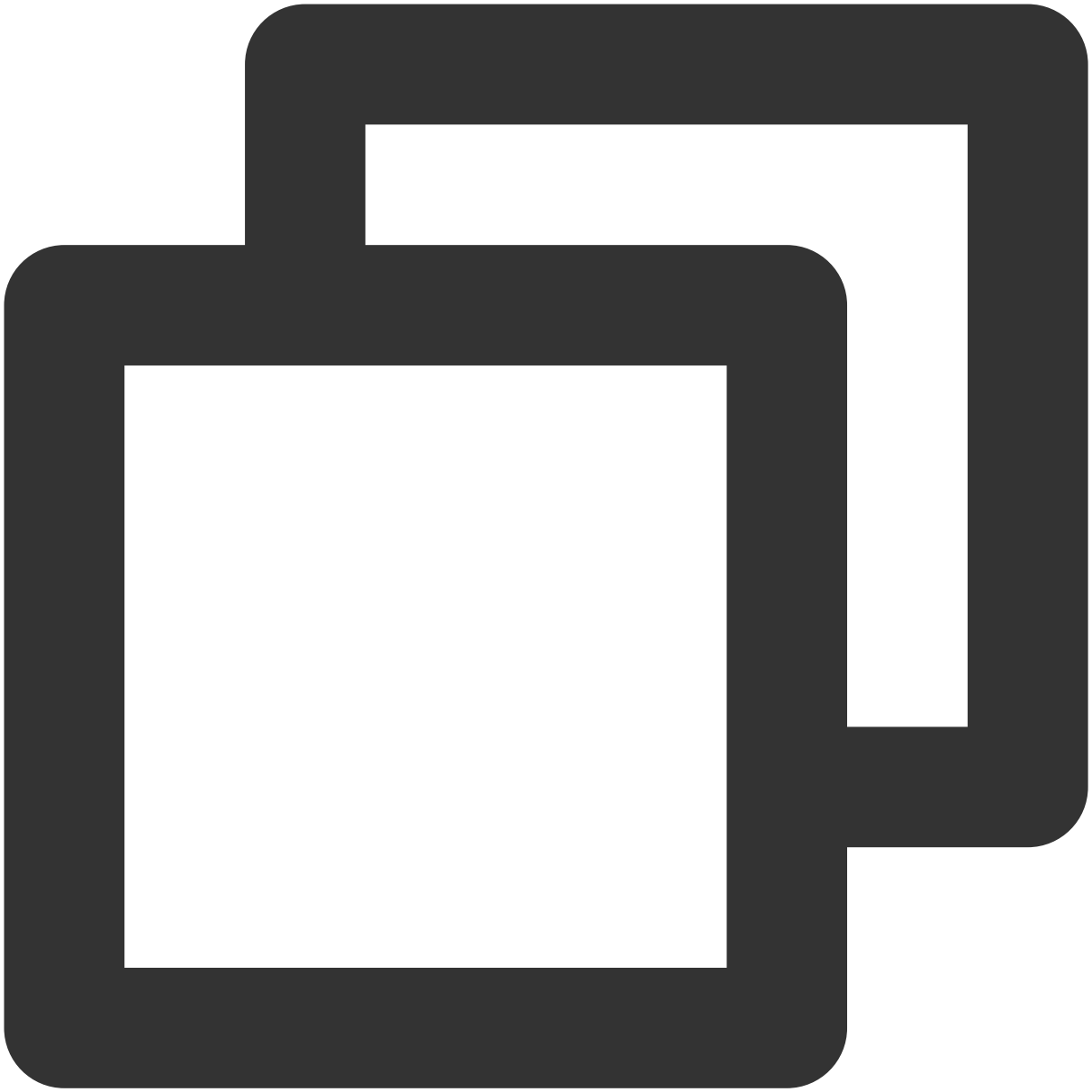


```
// API 返回 msgID，按需使用
String msgID = V2TIMManager.getInstance().sendC2CTextMessage("单聊文本消息", "receive
    @Override
    public void onSuccess(V2TIMMessage message) {
        // 发送单聊文本消息成功
    }

    @Override
    public void onError(int code, String desc) {
        // 发送单聊文本消息失败
    }
```

```
});
```



```
// API 返回 msgID，按需使用
NSString *msgID = [[V2TIMManager sharedInstance] sendC2CTextMessage:@"单聊文本消息"
                                                                 to:@"receiver_user
                                                               succ:^{
    // 发送单聊文本消息成功
} fail:^(int code, NSString *msg) {
    // 发送单聊文本消息失败
}];
```



```
class SendCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;

    SendCallback() = default;
    ~SendCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     ProgressCallback progress_callback) {
```

```
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }

    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

auto callback = new SendCallback{};
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 发送单聊文本消息成功
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 发送单聊文本消息失败
        delete callback;
    },
    [=](uint32_t progress) {
        // 文本消息不会回调进度
    });
// API 返回 msgID，按需使用
V2TIMString msgID =
    V2TIMManager::GetInstance()->SendC2CTextMessage("单聊文本消息", "receiver_userID"
```

高级接口

调用高级接口发送单聊文本消息分两步：

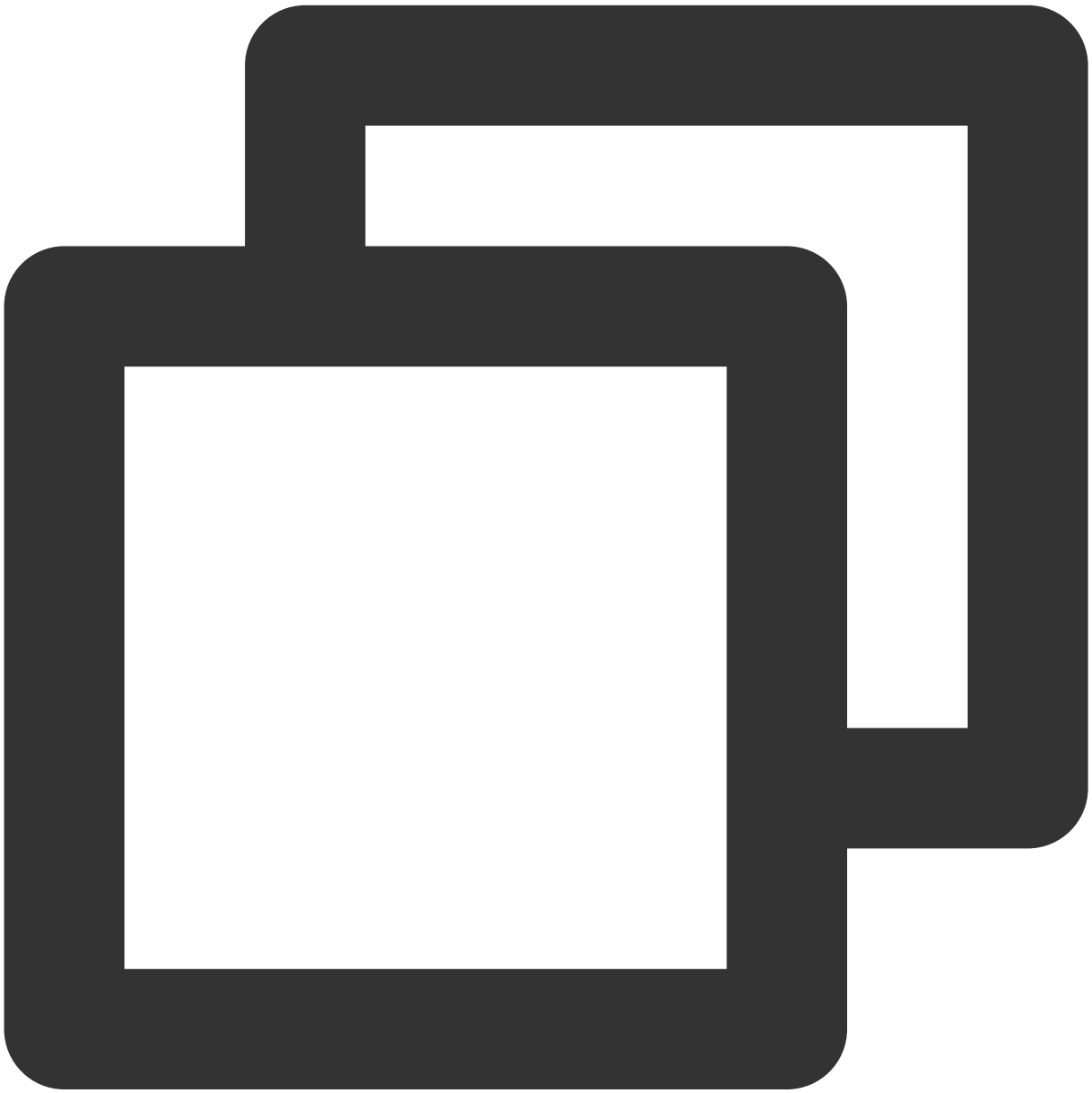
1. 调用 `createTextMessage` (Android / iOS & Mac / Windows) 创建文本消息。
2. 调用 `sendMessage` (Android / iOS & Mac / Windows) 发送消息。

示例代码如下：

Android

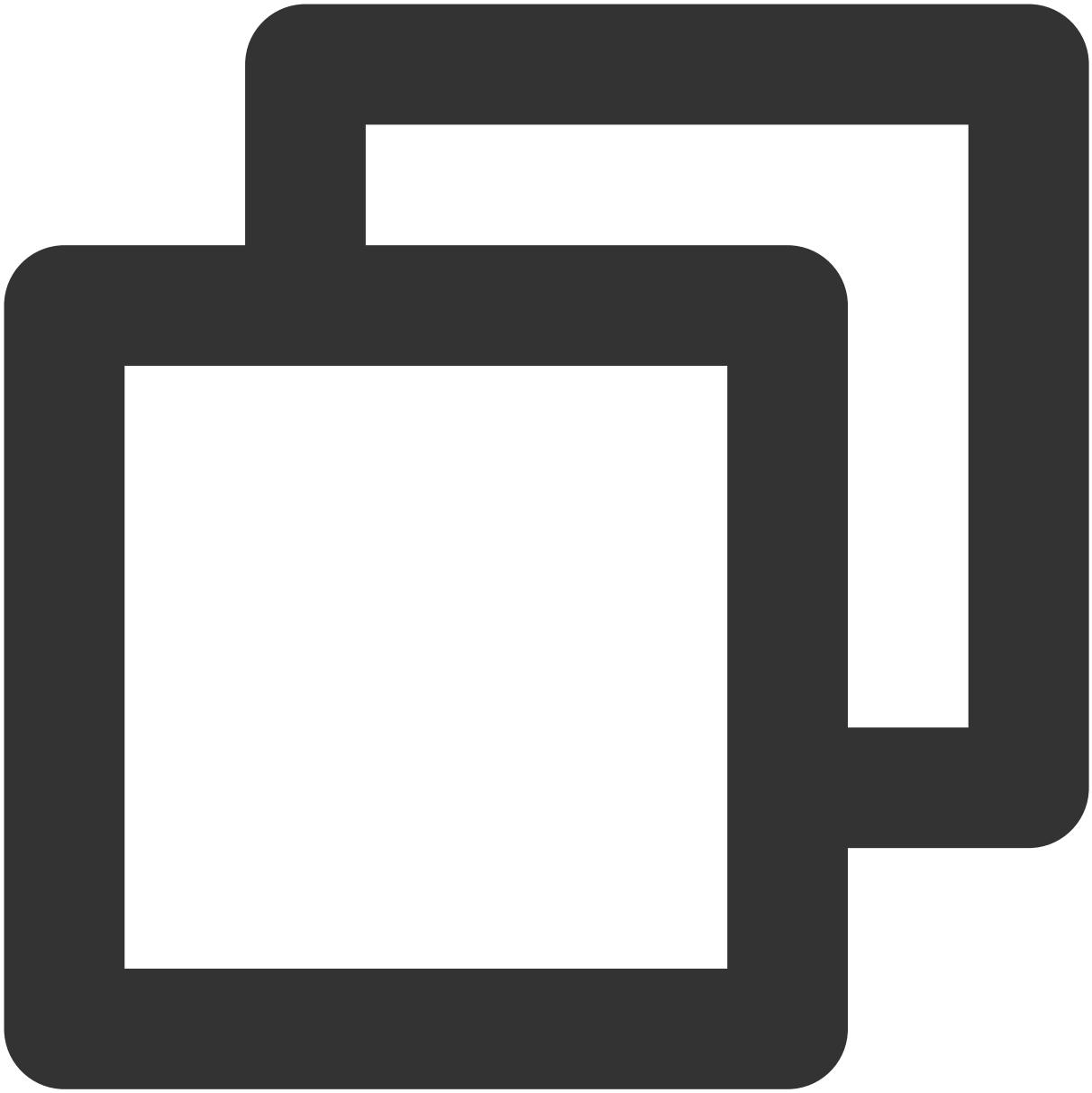
iOS & Mac

Windows



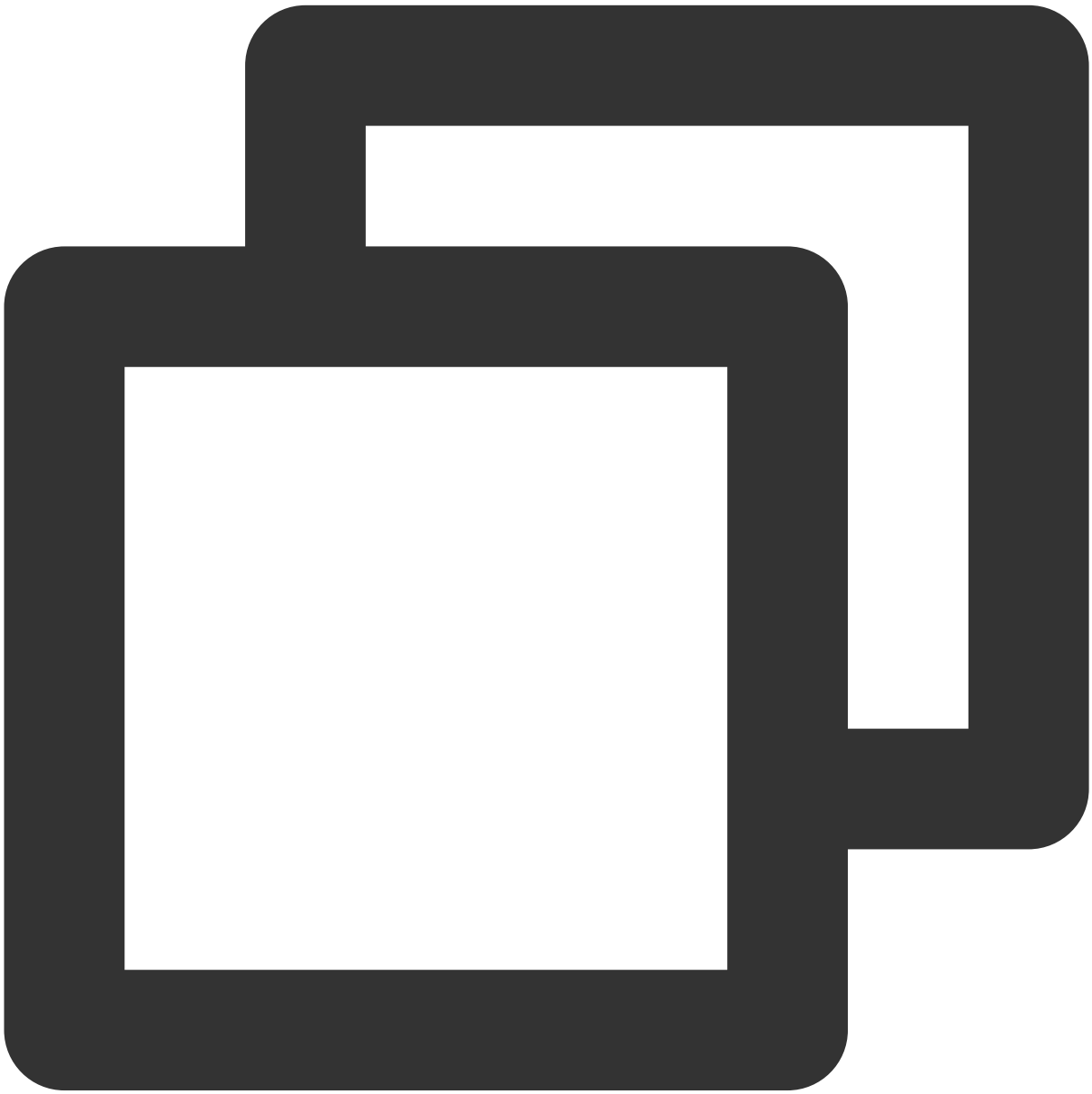
```
// 创建文本消息
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createTextMessage("con
// 发送消息
```

```
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "userID", null, V2TIMMes
    @Override
    public void onProgress(int progress) {
        // 文本消息不会回调进度
    }

    @Override
    public void onSuccess(V2TIMMessage message) {
        // 文本消息发送成功
    }

    @Override
    public void onError(int code, String desc) {
        // 文本消息发送失败
    }
});
```



```
// 创建文本消息
V2TIMMessage *message = [[V2TIMManager sharedInstance] createTextMessage:@"content"
// 发送消息
[V2TIMManager.sharedInstance sendMessage:message
                                receiver:@"userID"
                                 groupID:nil
                                priority:V2TIM_PRIORITY_NORMAL
                          onlineUserOnly:NO
                         offlinePushInfo:nil
                                progress:nil
                                succ:^{
```

```
    // 文本消息发送成功
}
                                fail:^(int code, NSString *desc) {
    // 文本消息发送失败
}];
```



```
class SendCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;
```

```
    SendCallback() = default;
    ~SendCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     ProgressCallback progress_callback) {
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }

    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

// 创建文本消息
V2TIMMessage v2TIMMessage = V2TIMManager::GetInstance()->GetMessageManager()->Creat
// 发送消息
auto callback = new SendCallback{};
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 文本消息发送成功
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 文本消息发送失败
        delete callback;
    },
    [=](uint32_t progress) {
```

```
        // 文本消息不会回调进度
    });
V2TIMManager::GetInstance()->GetMessageManager()->SendMessage(
    v2TIMMessage, "userID", {}, V2TIMMessagePriority::V2TIM_PRIORITY_NORMAL, false,
```

群聊文本消息

普通接口

调用 `sendGroupTextMessage` (Android / iOS & Mac / Windows) 发送群聊文本消息，直接传递消息内容、群聊的 groupID 和消息优先级即可。

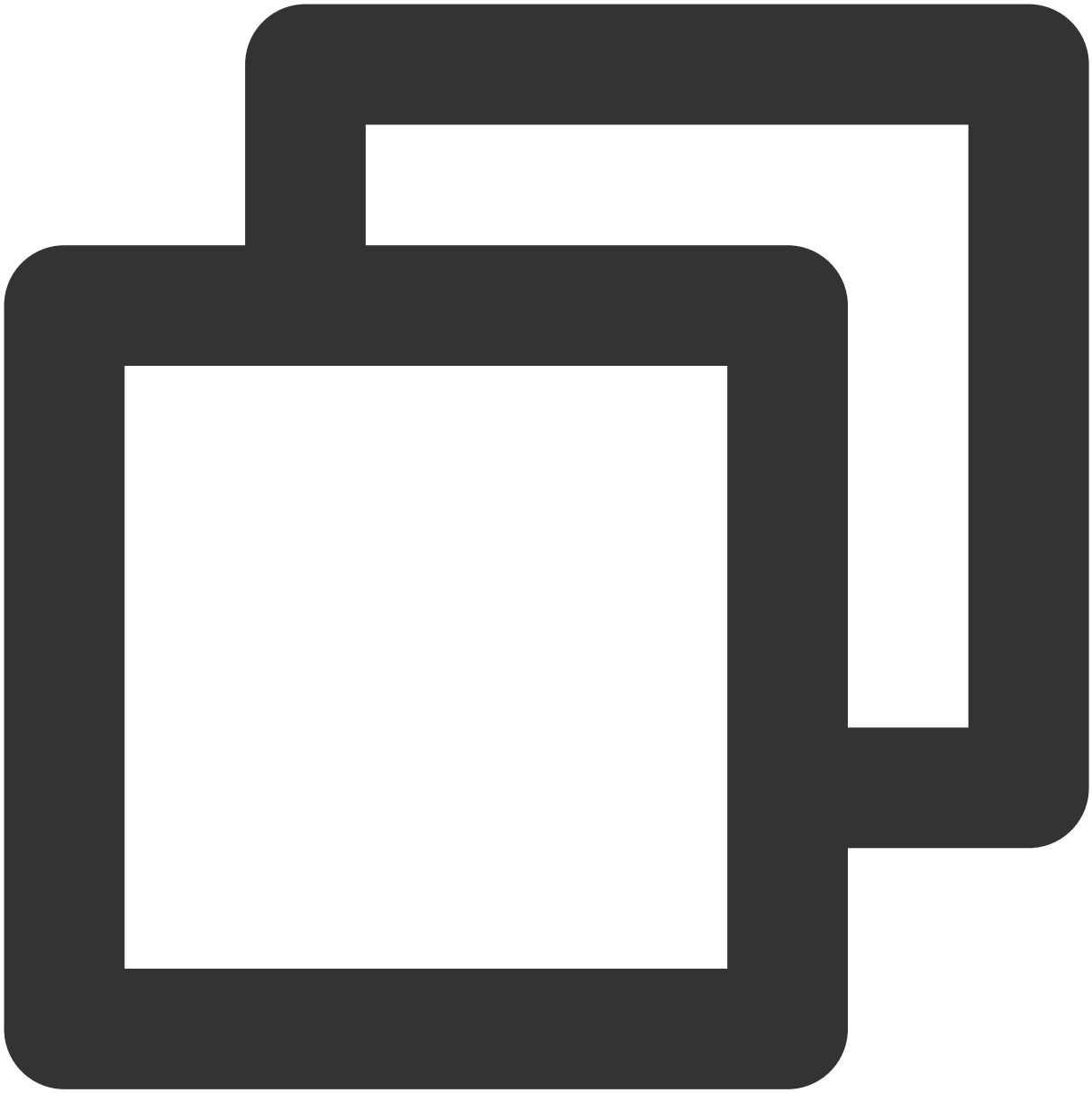
消息优先级可参考 `V2TIMMessagePriority` 定义。

示例代码如下：

Android

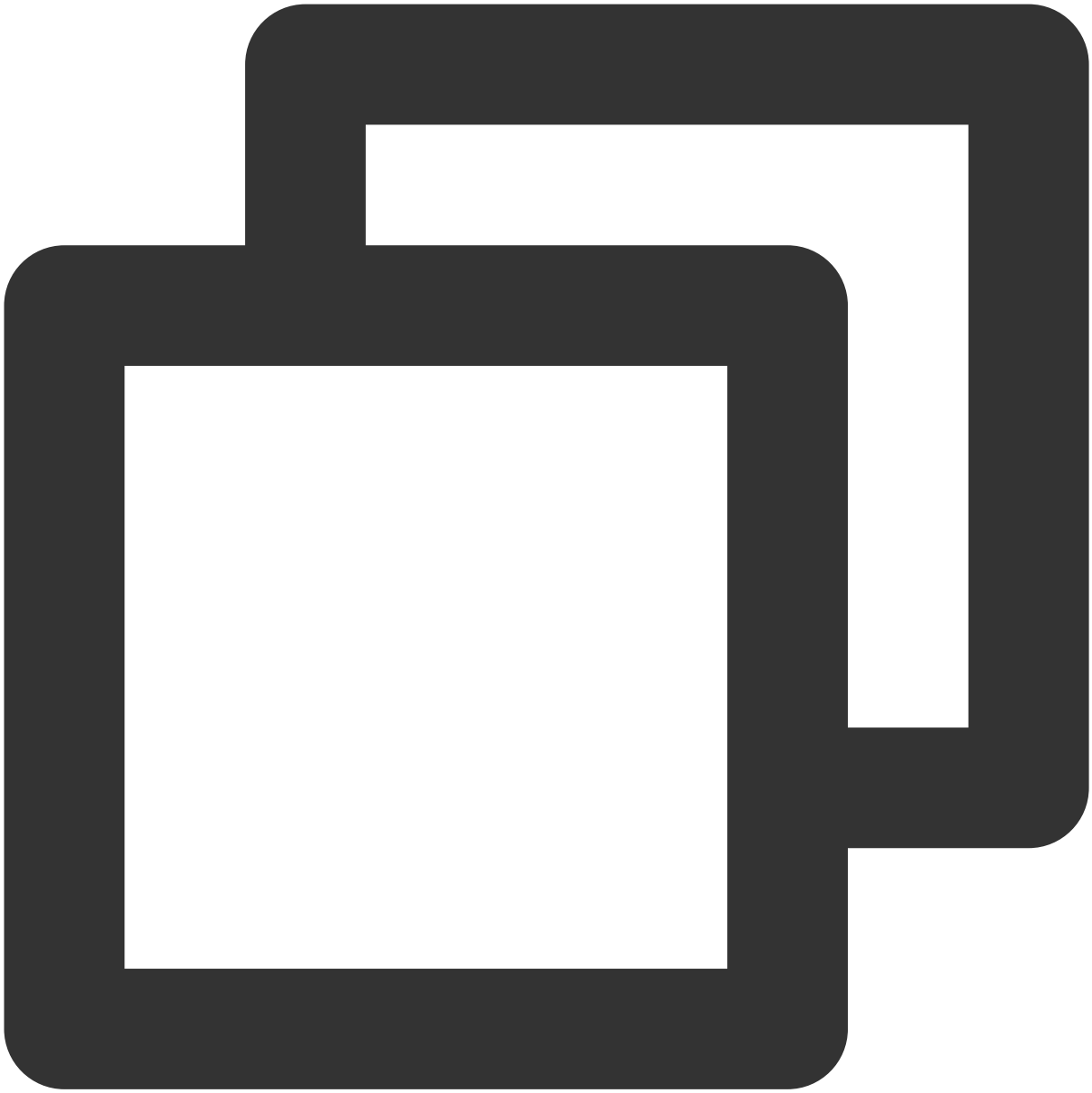
iOS & Mac

Windows

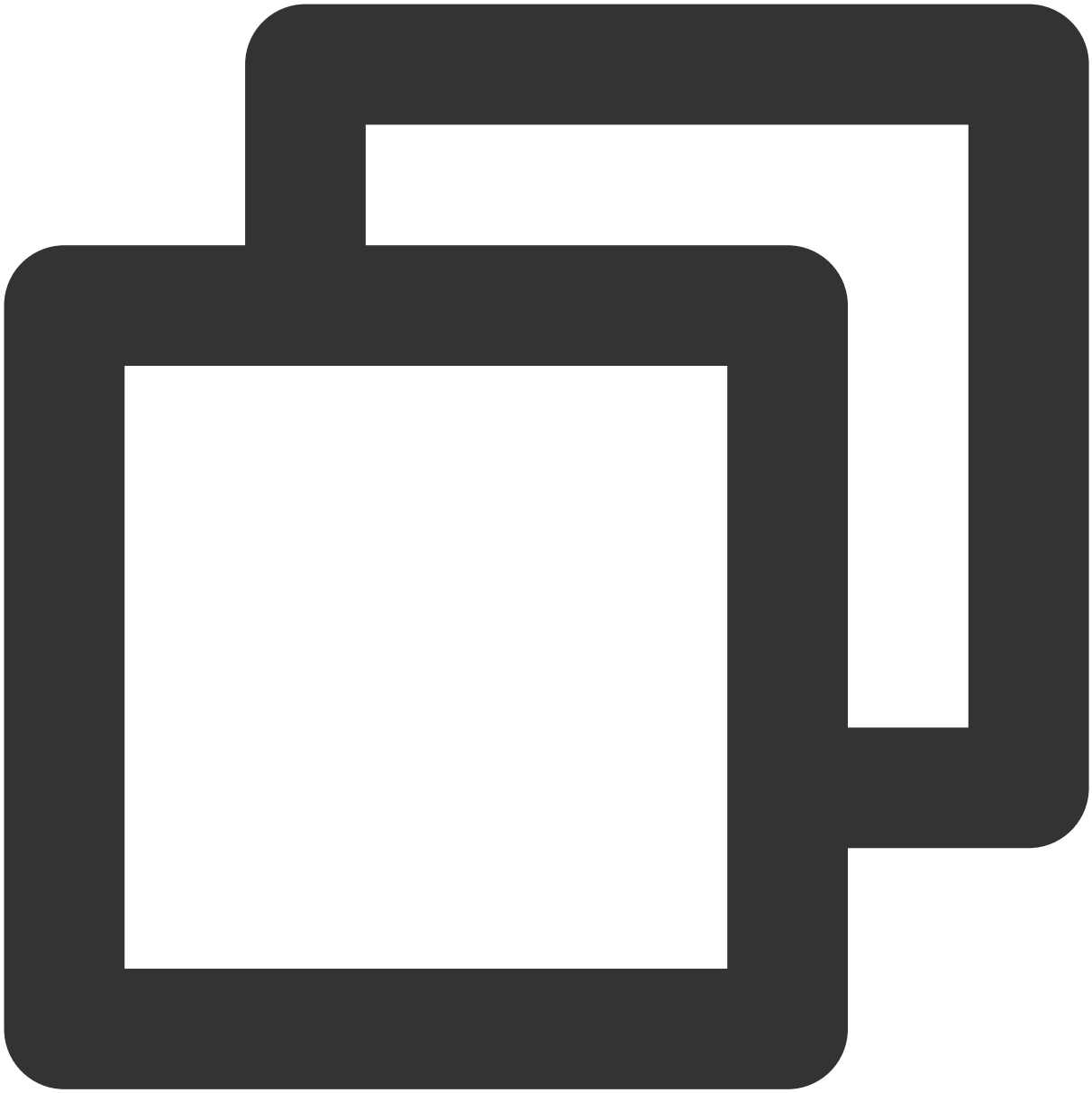


```
// API 返回 msgID，按需使用
String msgID = V2TIMManager.getInstance().sendGroupTextMessage("群聊文本消息", "group
    @Override
    public void onSuccess(V2TIMMessage message) {
        // 发送群聊文本消息成功
    }

    @Override
    public void onError(int code, String desc) {
        // 发送群聊文本消息失败
    }
```

```
});
```



```
// API 返回 msgID，按需使用
NSString *msgID = [[V2TIMManager sharedInstance] sendGroupTextMessage:@"群聊文本消息"
                                                                   to:@"groupID"  /
                                                             priority:V2TIM_PRIORIT
                                                                 succ:^{
    // 群聊文本消息发送成功
} fail:^(int code, NSString *msg) {
    // 群聊文本消息发送失败
}];
```



```
class SendGroupTextMessageCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;

    SendGroupTextMessageCallback() = default;
    ~SendGroupTextMessageCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
```

```
                    ProgressCallbackprogress_callback) {
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }
    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

auto callback = new SendGroupTextMessageCallback;
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 发送群聊文本消息成功
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 发送群聊文本消息失败
        delete callback;
    },
    [=](uint32_t progress) {
        // 文本消息不会回调进度
    });
// API 返回 msgID，按需使用
V2TIMString msgID = V2TIMManager::GetInstance()->SendGroupTextMessage(
    "群聊文本消息", "groupID", V2TIMMessagePriority::V2TIM_PRIORITY_NORMAL, callback)
```

高级接口

调用高级接口发送群聊文本消息分两步：

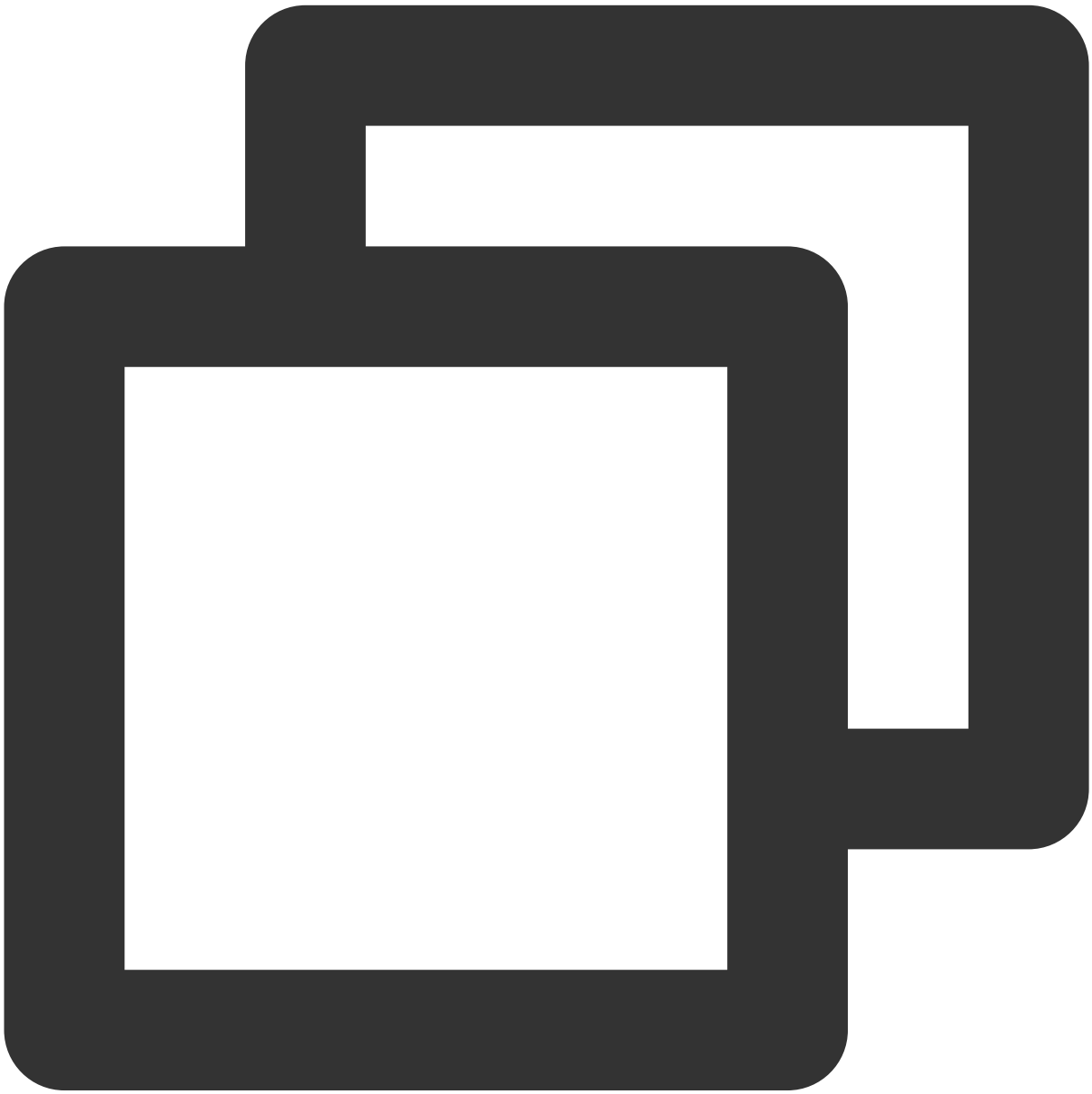
1. 调用 `createTextMessage` (Android / iOS & Mac / Windows) 创建文本消息。
2. 调用 `sendMessage` (Android / iOS & Mac / Windows) 发送消息。

示例代码如下：

Android

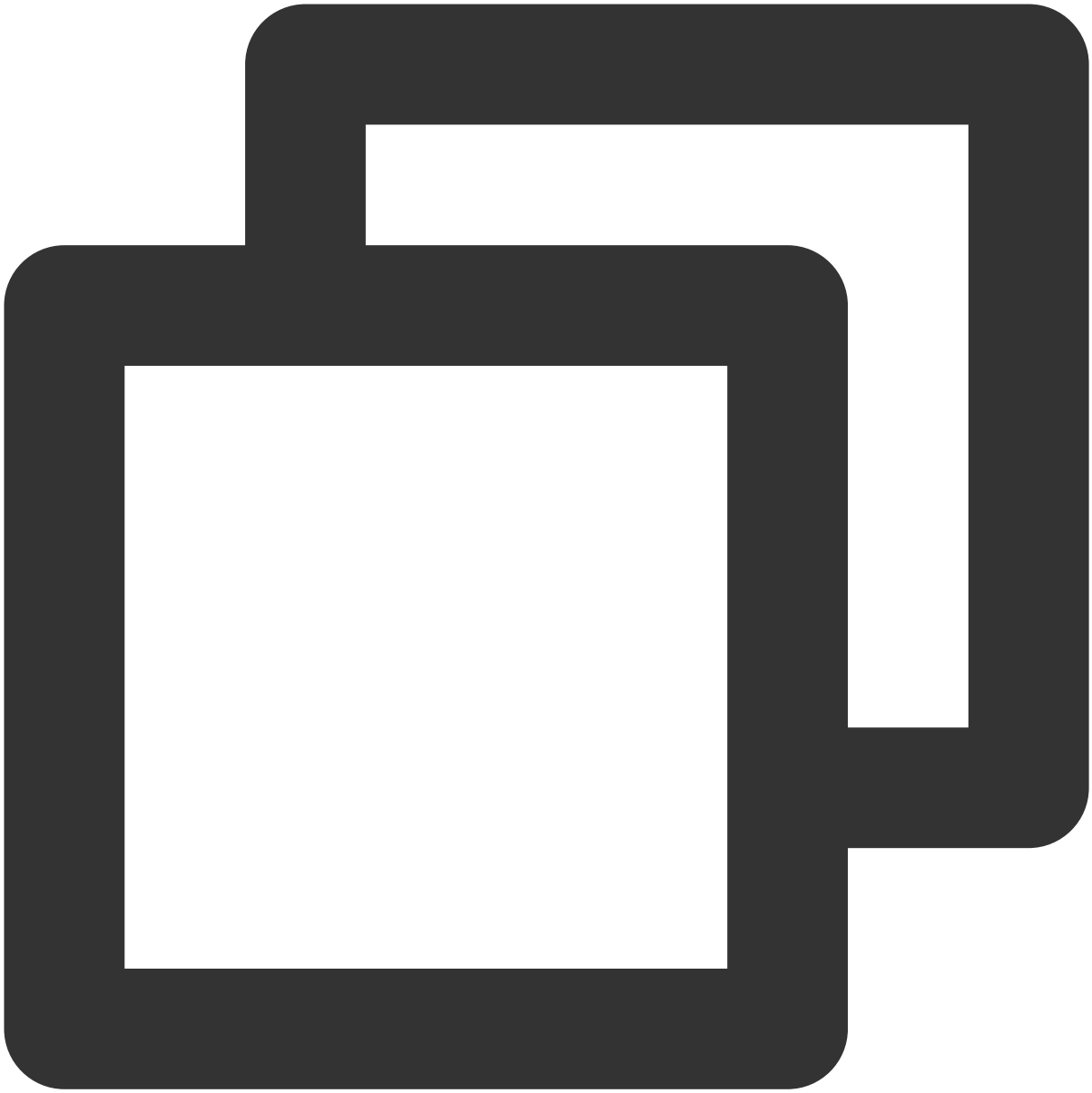
iOS & Mac

Windows



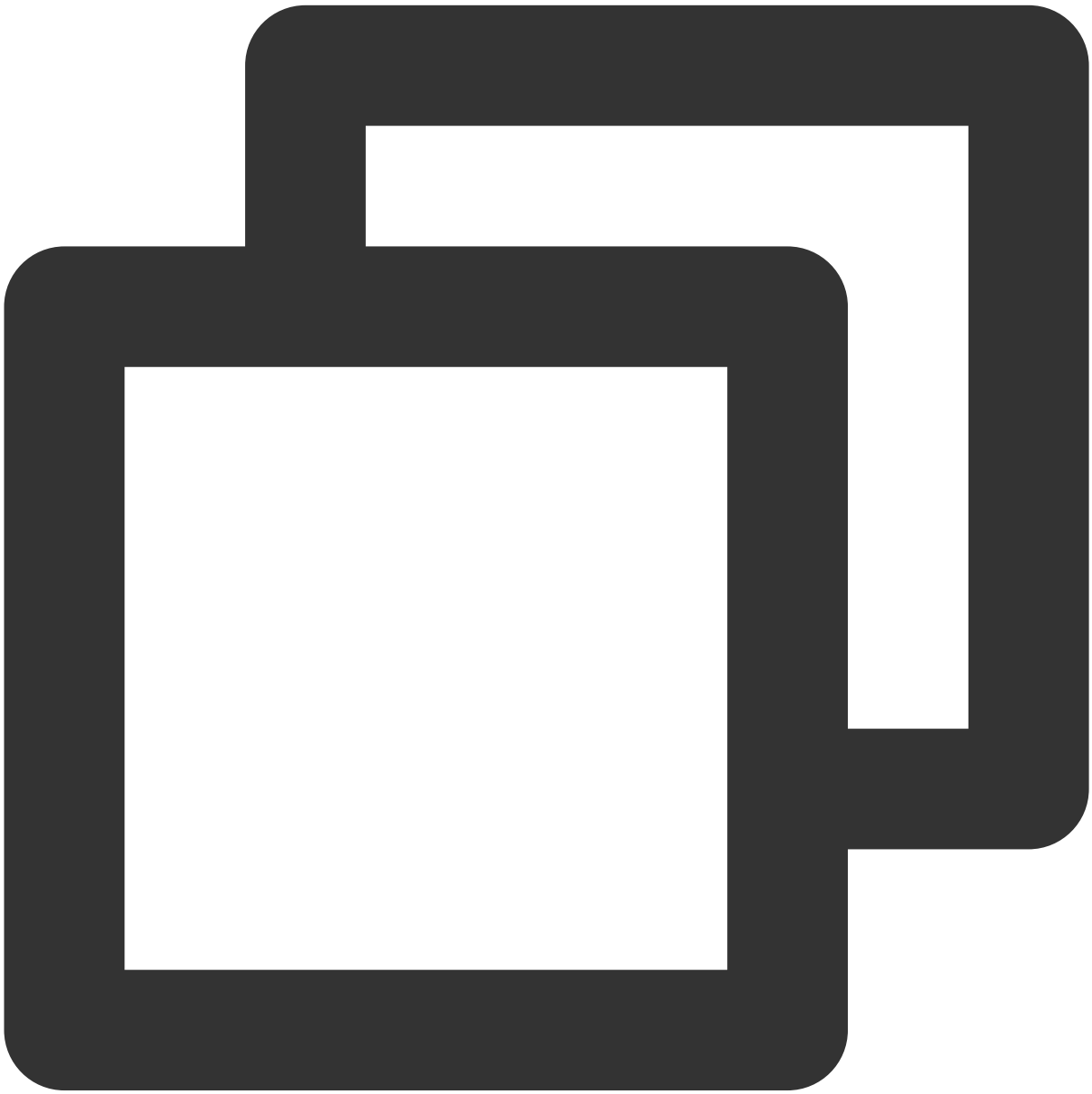
```
// 创建文本消息
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createTextMessage("con
// 发送消息
```

```
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, null, "receiver_groupID"
    @Override
    public void onProgress(int progress) {
        // 文本消息不会回调进度
    }

    @Override
    public void onSuccess(V2TIMMessage message) {
        // 发送群聊文本消息成功
    }

    @Override
    public void onError(int code, String desc) {
        // 发送群聊文本消息失败
    }
});
```



```
// 创建文本消息
V2TIMMessage *message = [[V2TIMManager sharedInstance] createTextMessage:content];
// 发送消息
[V2TIMManager.sharedInstance sendMessage:message
                                receiver:nil
                                 groupID:@"receiver_groupID"  // 群聊的 groupID
                                priority:V2TIM_PRIORITY_NORMAL // 消息优先级
                          onlineUserOnly:NO   // 仅在线用户接收
                         offlinePushInfo:nil  // 离线推送自定义信息
                                progress:nil
                                    succ:^{
```

```
    // 文本消息发送成功
}
                                    fail:^(int code, NSString *desc) {
    // 文本消息发送失败
}];
```



```
class SendCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;
```

```
    SendCallback() = default;
    ~SendCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     ProgressCallback progress_callback) {
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }

    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

// 创建文本消息
V2TIMMessage v2TIMMessage = V2TIMManager::GetInstance()->GetMessageManager()->Creat
// 发送消息
auto callback = new SendCallback{};
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 发送群聊文本消息成功
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 发送群聊文本消息失败
        delete callback;
    },
    [=](uint32_t progress) {
```

```
            // 文本消息不会回调进度
        });
    V2TIMManager::GetInstance()->GetMessageManager()->SendMessage(
        v2TIMMessage, {}, "receiver_groupID", V2TIMMessagePriority::V2TIM_PRIORITY_NORM
```

发送自定义消息

自定义消息区分单聊和群聊，涉及的接口或者传参有所区别。发送自定义消息可以采用两种接口：普通接口和高级接口。

高级接口即上文中已介绍过的 `sendMessage` (Android / iOS & Mac / Windows)，比普通接口能设置更多的发送参数（例如优先级、离线推送信息等）。

单聊自定义消息

普通接口

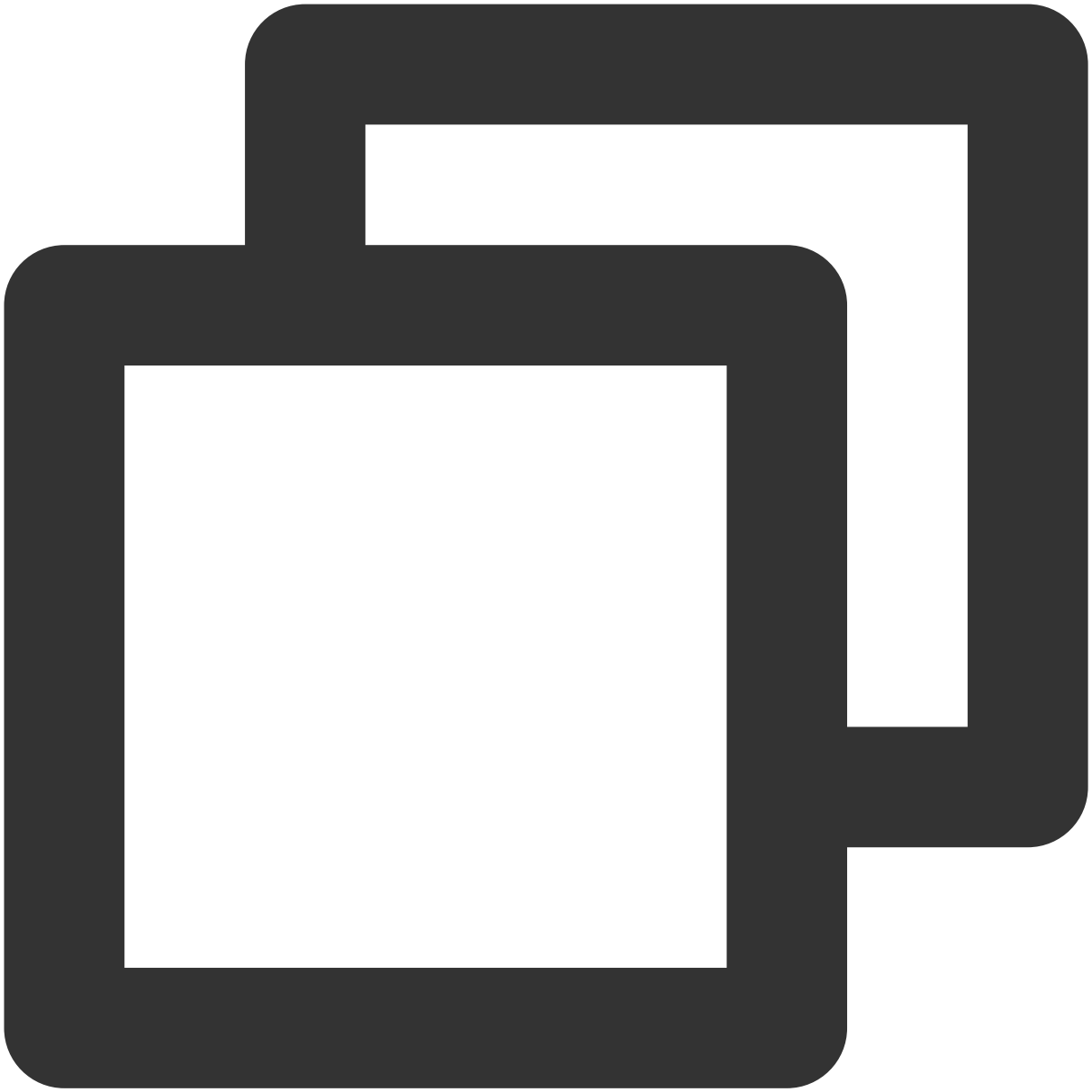
调用 `sendC2CCustomMessage` (Android / iOS & Mac / Windows) 发送单聊自定义消息，直接传递消息二进制内容、单聊接收者 userID 即可。

示例代码如下：

Android

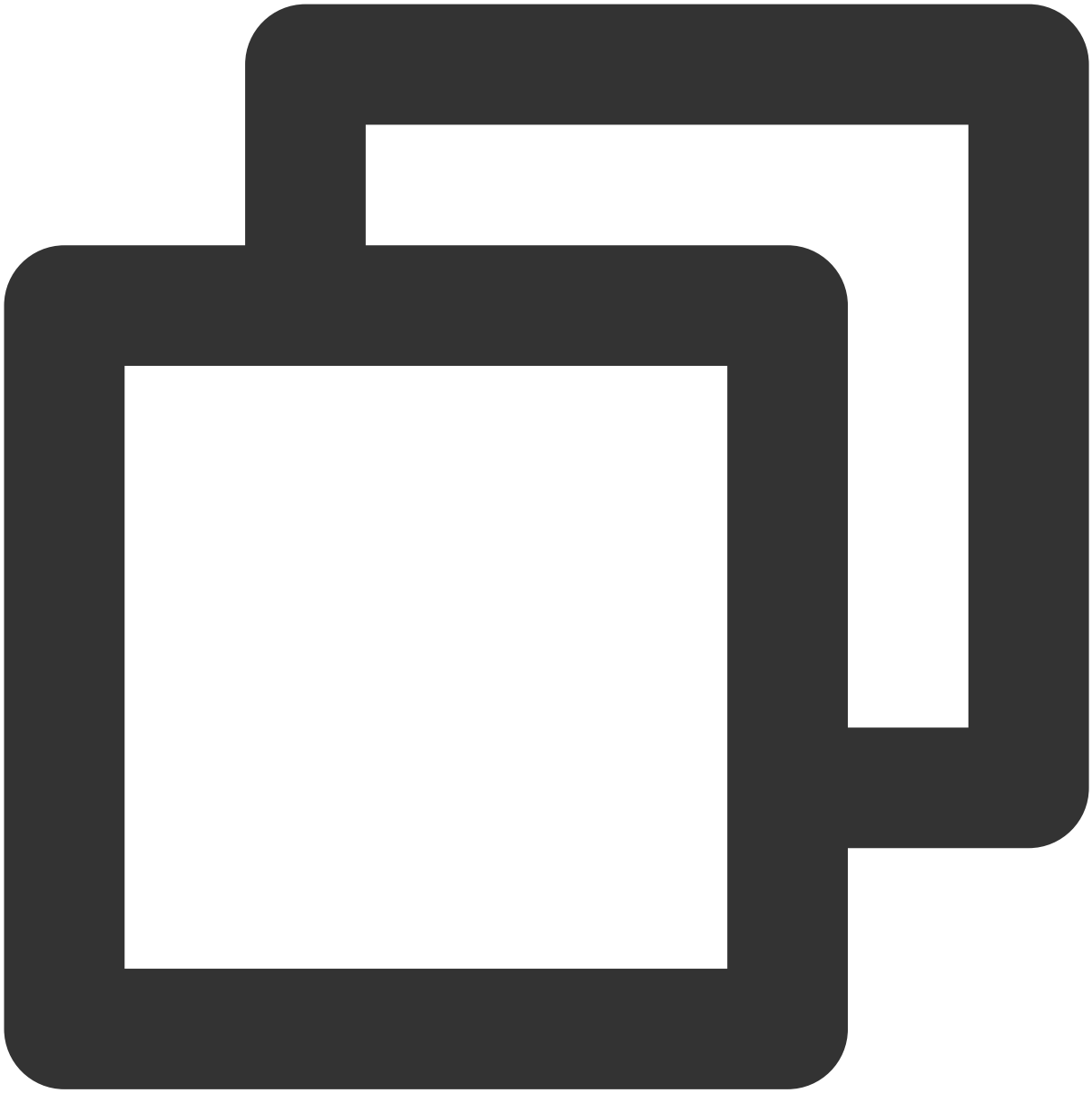
iOS & Mac

Windows

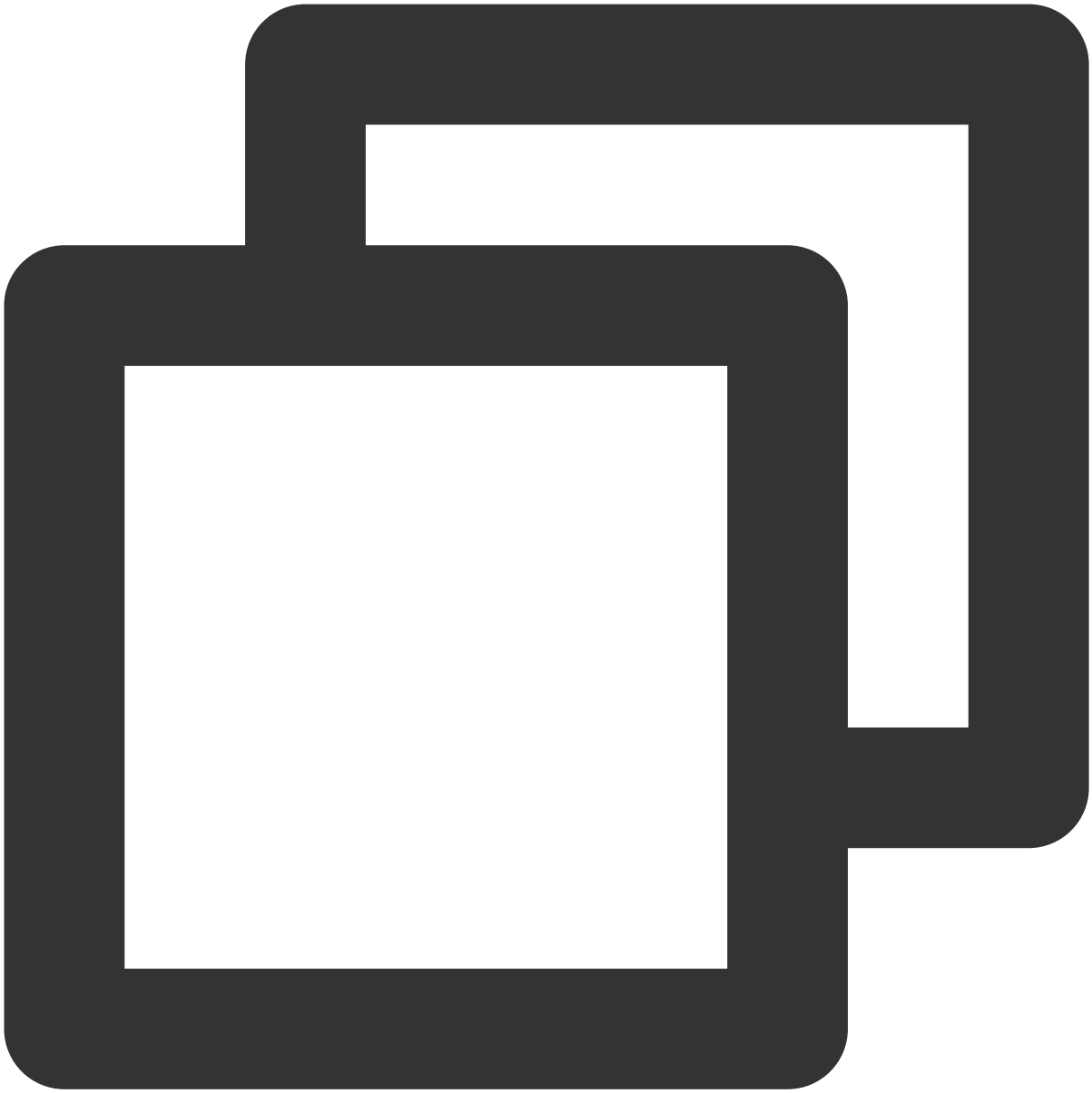


```
String msgID = V2TIMManager.getInstance().sendC2CCustomMessage("单聊自定义消息".getByt
    @Override
    public void onSuccess(V2TIMMessage message) {
        // 发送单聊自定义消息成功
    }

    @Override
    public void onError(int code, String desc) {
        // 发送单聊自定义消息失败
    }
```

```
});
```



```
NSData *customData = [@"单聊自定义消息"  dataUsingEncoding:NSUTF8StringEncoding];
NSString *msgID = [[V2TIMManager sharedInstance] sendC2CCustomMessage:customData
                                                                   to:@"receiver_us
                                                                 succ:^{
    // 单聊自定义消息发送成功
}
                                                                 fail:^(int code, NS
    // 单聊自定义消息发送失败
}];
```



```
class SendC2CCustomMessageCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;

    SendC2CCustomMessageCallback() = default;
    ~SendC2CCustomMessageCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
```

```
                    ProgressCallback progress_callback) {
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }
    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

auto callback = new SendCallback{};
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 发送单聊自定义消息成功
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 发送单聊自定义消息失败
        delete callback;
    },
    [=](uint32_t progress) {
        // 自定义消息不会回调进度
    });
V2TIMString str = u8"单聊自定义消息";
V2TIMBuffer customData = {reinterpret_cast<const uint8_t*>(str.CString()), str.Size
V2TIMString msgID = V2TIMManager::GetInstance()->SendC2CCustomMessage(customData, "
```

高级接口

调用高级接口发送单聊自定义消息分两步：

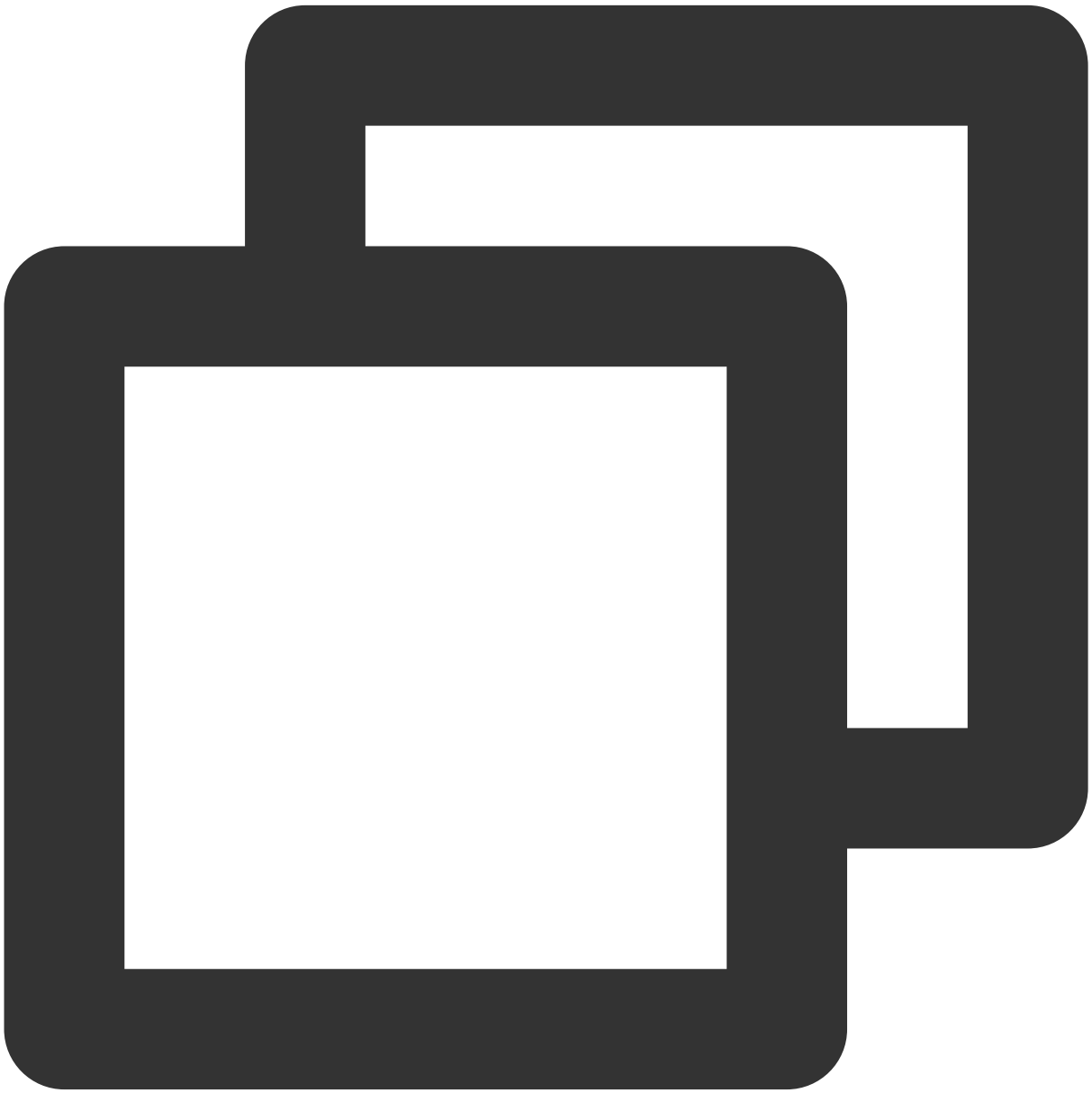
1. 调用 `createCustomMessage` (Android / iOS & Mac / Windows) 创建自定义消息。
2. 调用 `sendMessage` (Android / iOS & Mac / Windows) 发送消息。

示例代码如下：

Android

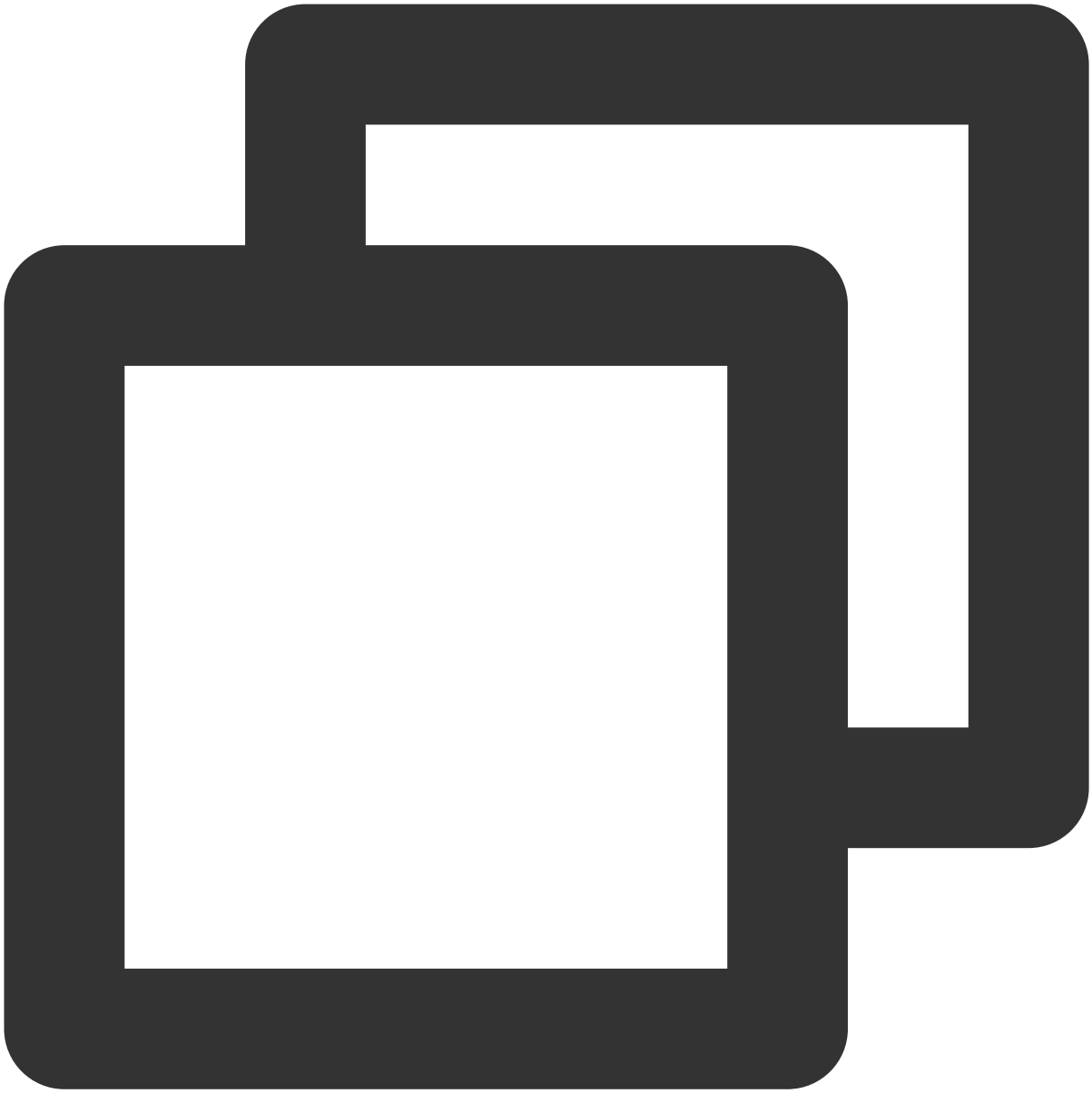
iOS & Mac

Windows



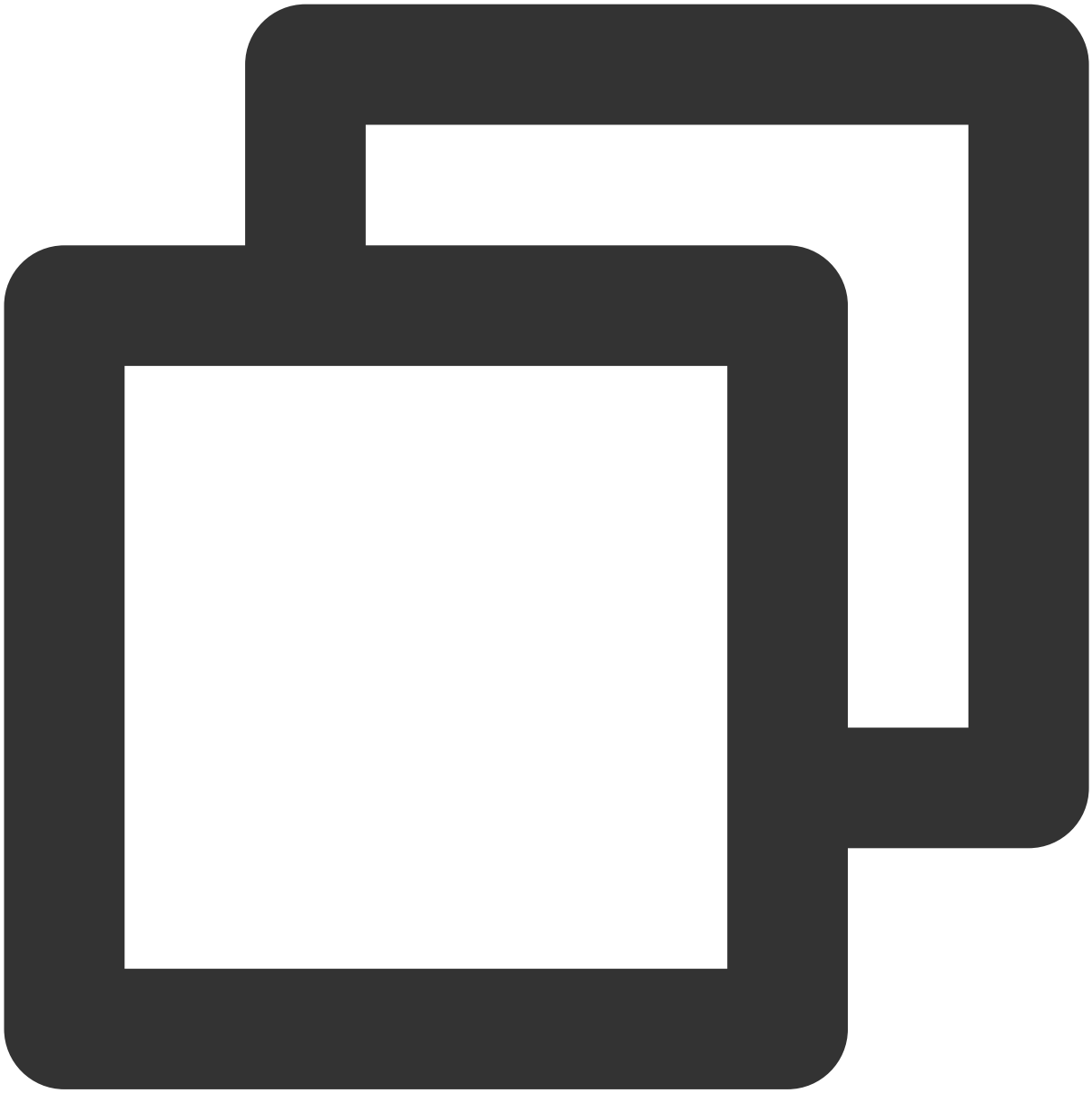
```
// 创建自定义消息
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createCustomMessage("自
// 发送消息
```

```
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "receiver_userID", null,
    @Override
    public void onProgress(int progress) {
        // 自定义消息不会回调进度
    }

    @Override
    public void onSuccess(V2TIMMessage message) {
        // 发送单聊自定义消息成功
    }

    @Override
    public void onError(int code, String desc) {
        // 发送单聊自定义消息失败
    }
});
```



```
V2TIMMessage *message = [[V2TIMManager sharedInstance] createCustomMessage:data];
[[V2TIMManager sharedInstance] sendMessage:message
                                  receiver:@"receiver_userID"  // 接收者 userID
                                   groupID:nil
                                  priority:V2TIM_PRIORITY_DEFAULT  // 消息优先级
                            onlineUserOnly:NO
                           offlinePushInfo:nil
                                  progress:nil
                                      succ:^{
    // 单聊自定义消息发送成功
} fail:^(int code, NSString *desc) {
```

```
    // 单聊自定义消息发送失败
}];
```



```
class SendCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;

    SendCallback() = default;
    ~SendCallback() override = default;
```

```
    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     ProgressCallback progress_callback) {
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }

    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

// 创建自定义消息
V2TIMString str = u8"单聊自定义消息";
V2TIMBuffer customData = {reinterpret_cast<const uint8_t*>(str.CString()), str.Size
V2TIMMessage v2TIMMessage = V2TIMManager::GetInstance()->GetMessageManager()->Creat
// 发送消息
auto callback = new SendCallback{};
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 发送单聊自定义消息成功
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 发送单聊自定义消息失败
        delete callback;
    },
    [=](uint32_t progress) {
        // 自定义消息不会回调进度
```

```
    });
V2TIMManager::GetInstance()->GetMessageManager()->SendMessage(
    v2TIMMessage, "receiver_userID", {}, V2TIMMessagePriority::V2TIM_PRIORITY_NORMA
```

群聊自定义消息

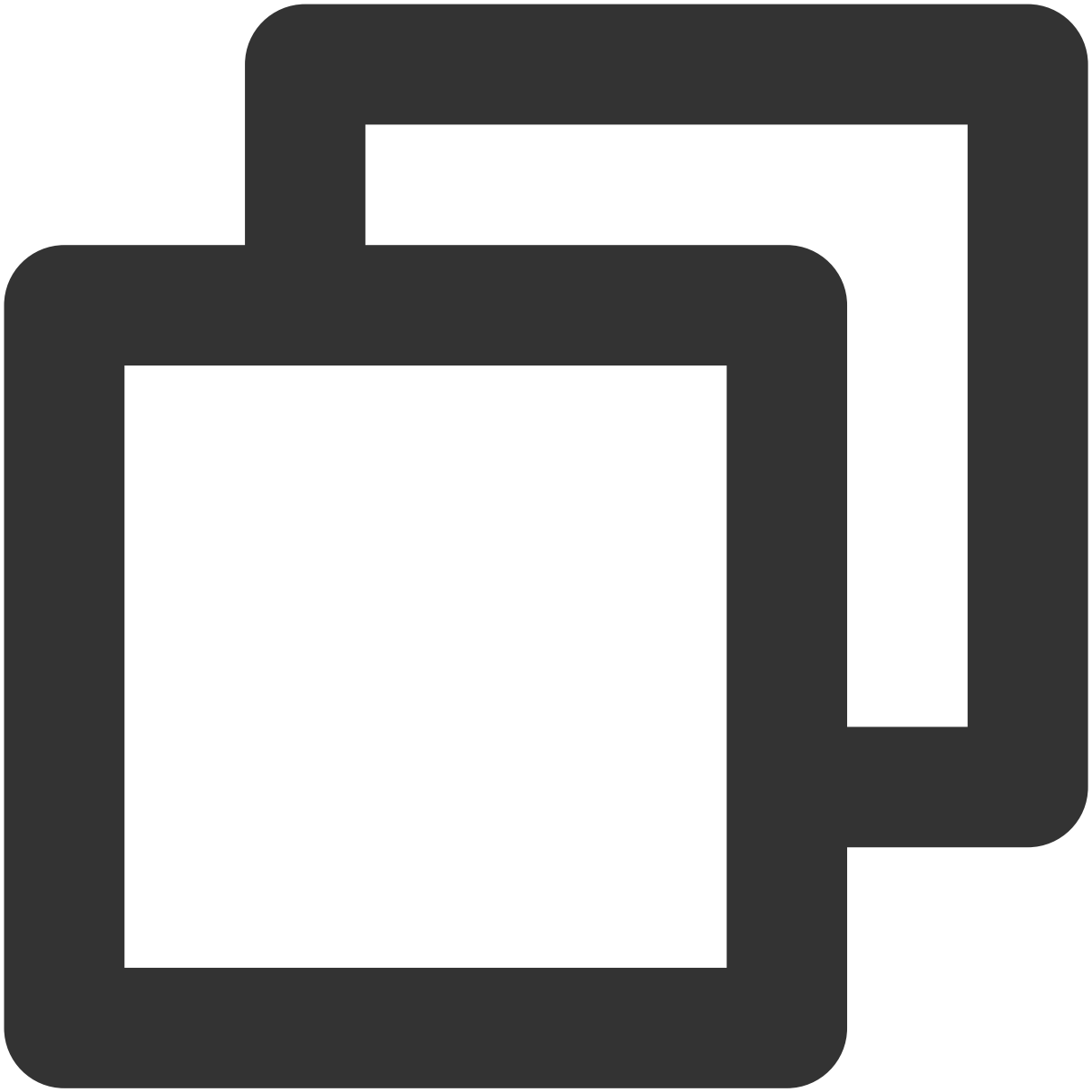
普通接口

调用 `sendGroupCustomMessage` (Android / iOS & Mac / Windows) 发送群聊自定义消息，直接传递消息二进制内容、群聊 groupID 和优先级即可。
消息优先级可参考 `V2TIMMessagePriority` 定义。
代码示例如下：

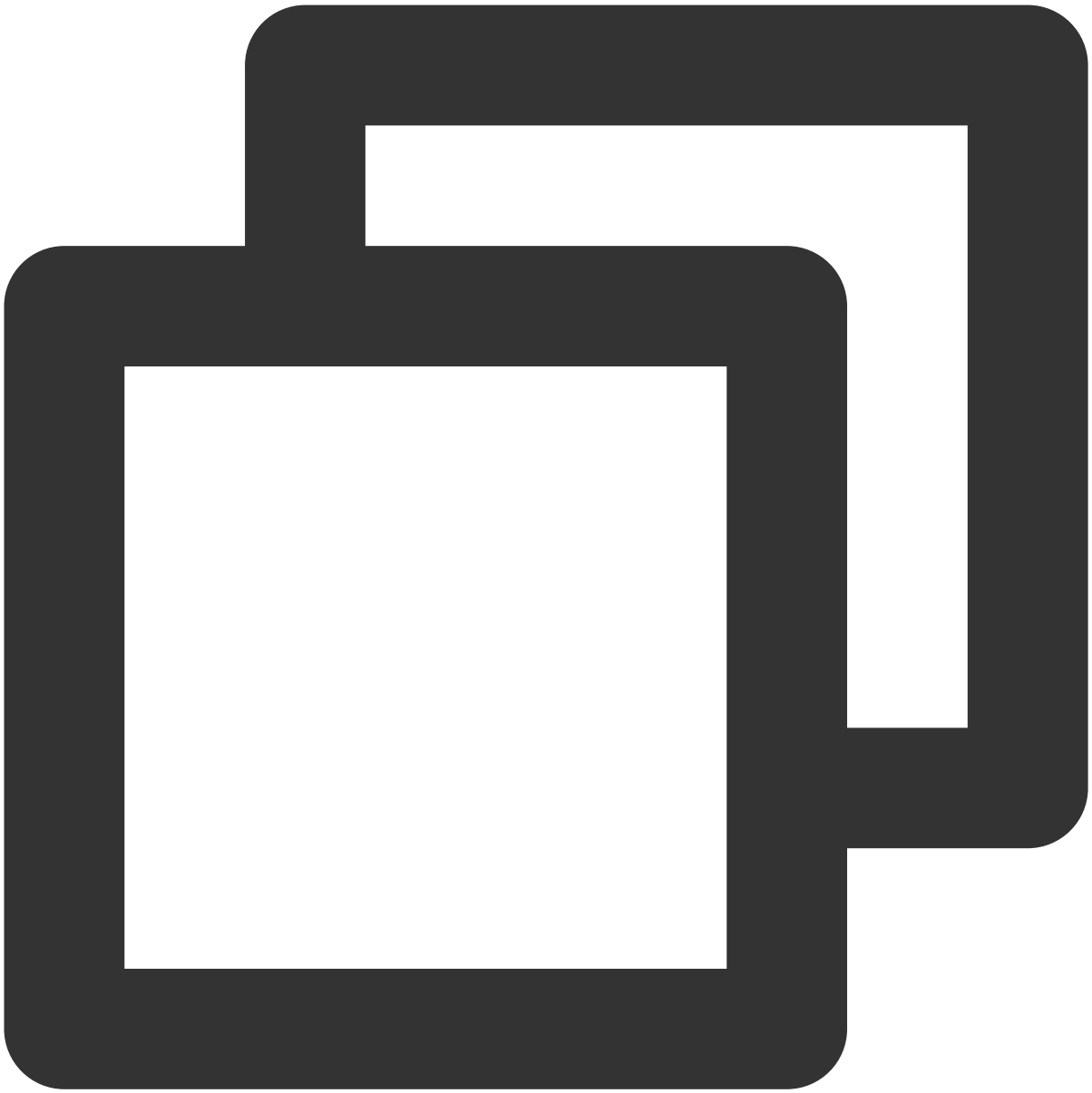
Android

iOS & Mac

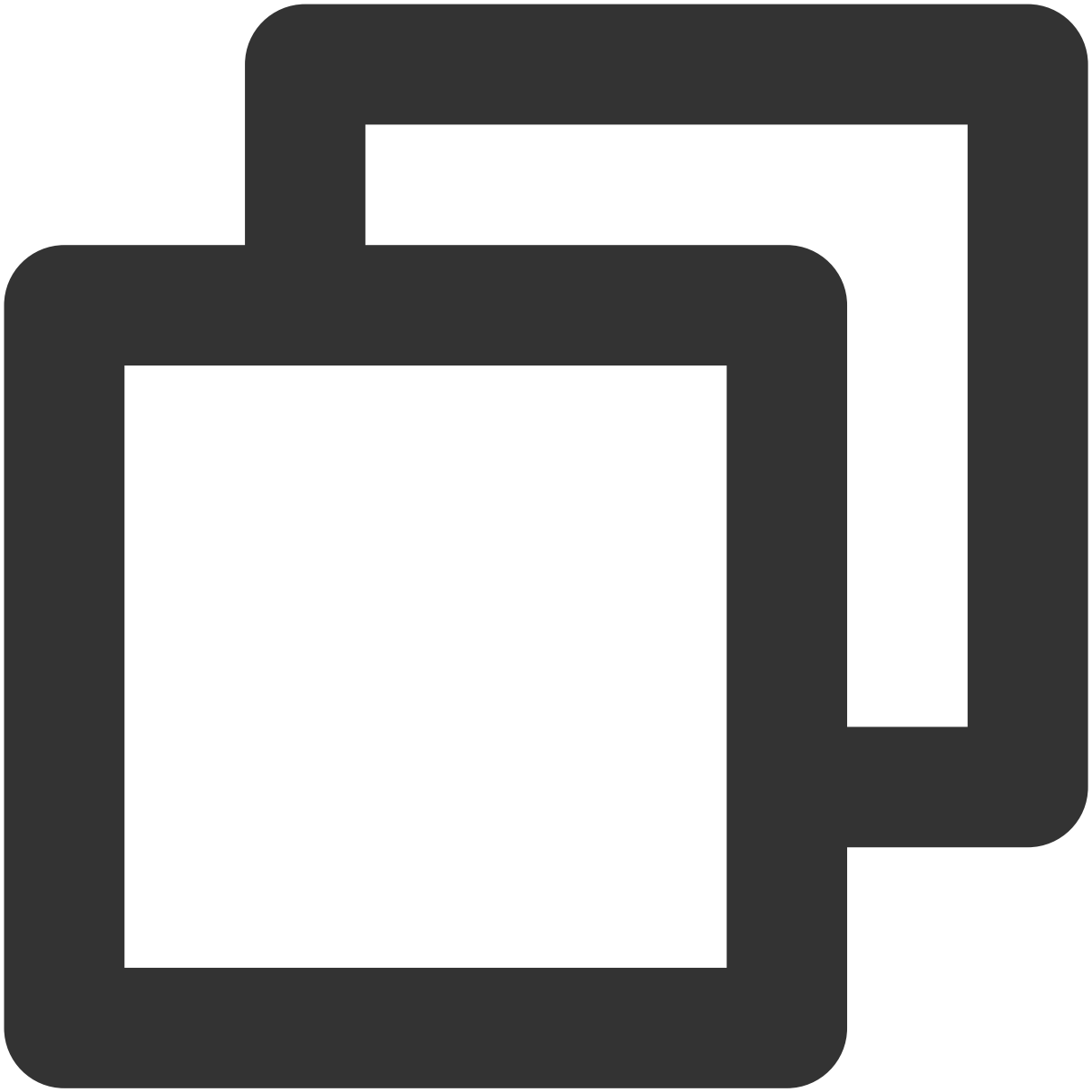
Windows



```
String msgID = V2TIMManager.getInstance().sendGroupCustomMessage("群聊自定义消息".getB
    @Override
    public void onSuccess(V2TIMMessage message) {
        // 发送群聊自定义消息成功
    }

    @Override
    public void onError(int code, String desc) {
        // 发送群聊自定义消息失败
    }
});
```



```
NSData *customData = [@"群聊自定义消息"  dataUsingEncoding:NSUTF8StringEncoding];
NSString *msgID = [[V2TIMManager sharedInstance] sendGroupCustomMessage:customData
                                                                     to:@"receiver_
                                                               priority:V2TIM_PRIOR
                                                                   succ:^{
    // 群聊自定义消息发送成功
} fail:^(int code, NSString *msg) {
    // 群聊自定义消息发送失败
}];
```



```
class SendCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;

    SendCallback() = default;
    ~SendCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     ProgressCallback progress_callback) {
```

```
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }

    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

auto callback = new SendCallback{};
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 发送群聊自定义消息成功
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 发送群聊自定义消息失败
        delete callback;
    },
    [=](uint32_t progress) {
        // 自定义消息不会回调进度
    });
V2TIMString str = u8"群聊自定义消息";
V2TIMBuffer customData = {reinterpret_cast<const uint8_t*>(str.CString()), str.Size
V2TIMString msgID = V2TIMManager::GetInstance()->SendGroupCustomMessage(
    customData, "groupID", V2TIMMessagePriority::V2TIM_PRIORITY_NORMAL, callback);
```

高级接口

调用高级接口发送群聊自定义消息分两步：

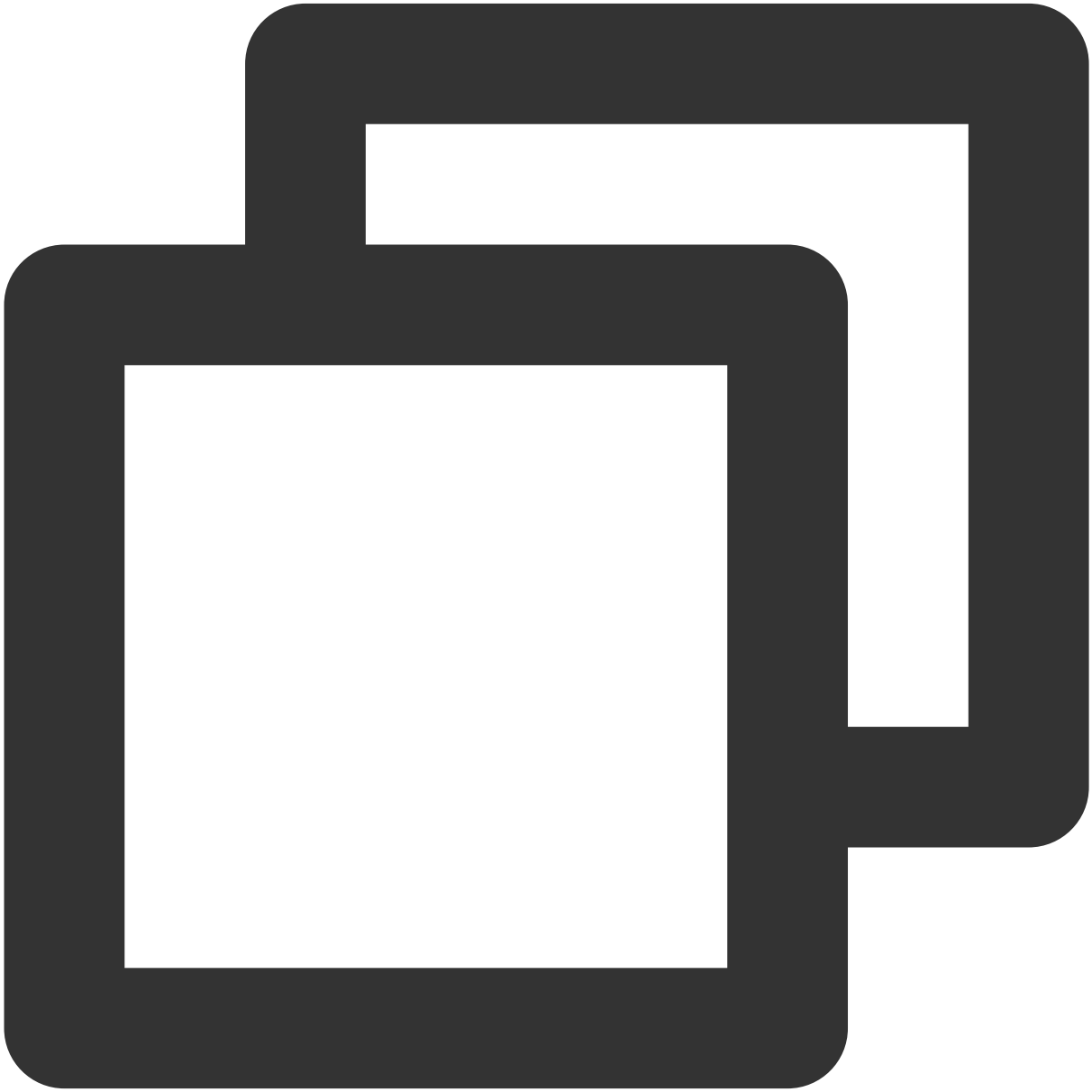
1. 调用 `createCustomMessage` (Android / iOS & Mac / Windows) 创建自定义消息。
2. 调用 `sendMessage` (Android / iOS & Mac / Windows) 发送消息。

代码示例如下：

Android

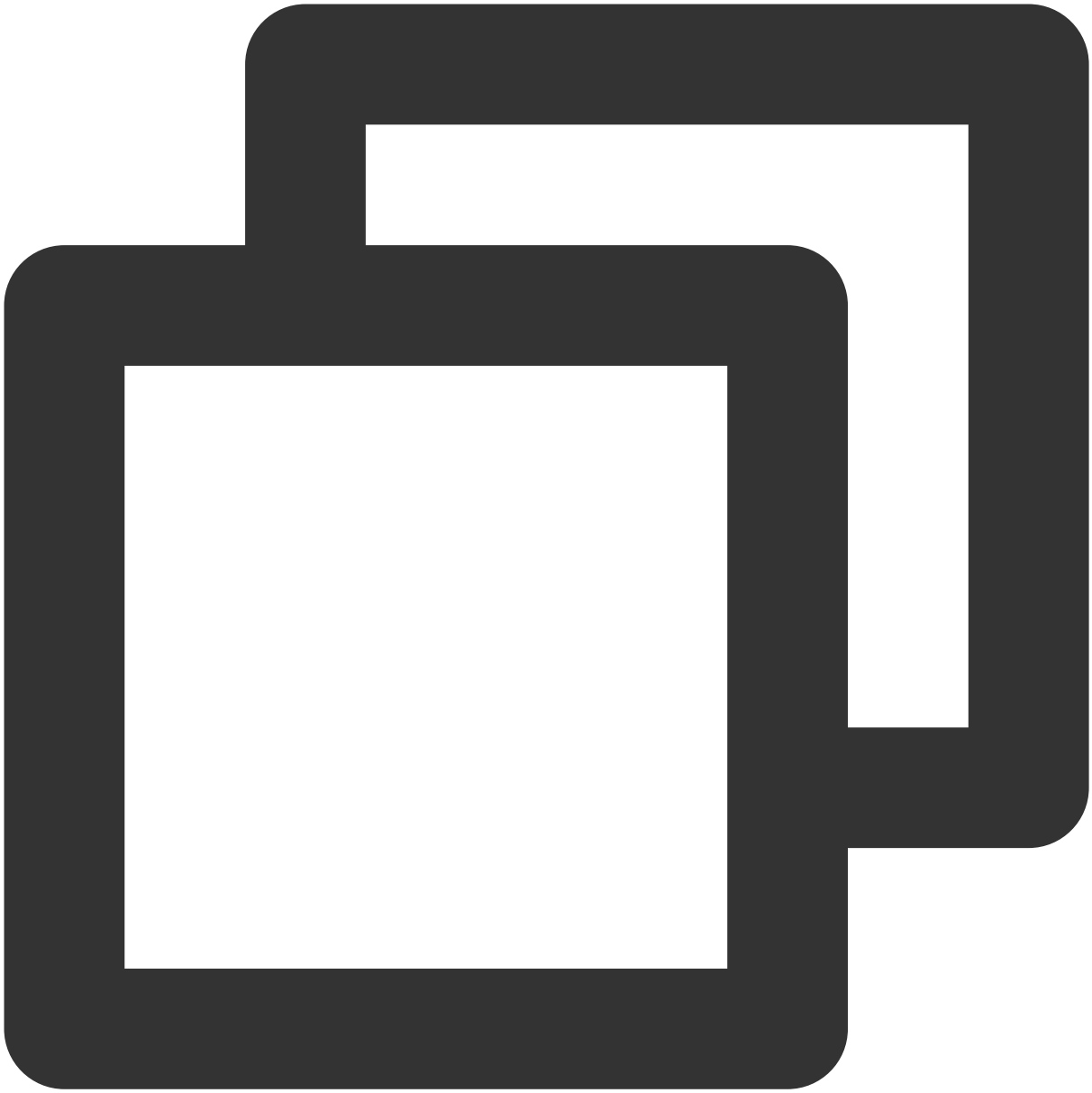
iOS & Mac

Windows



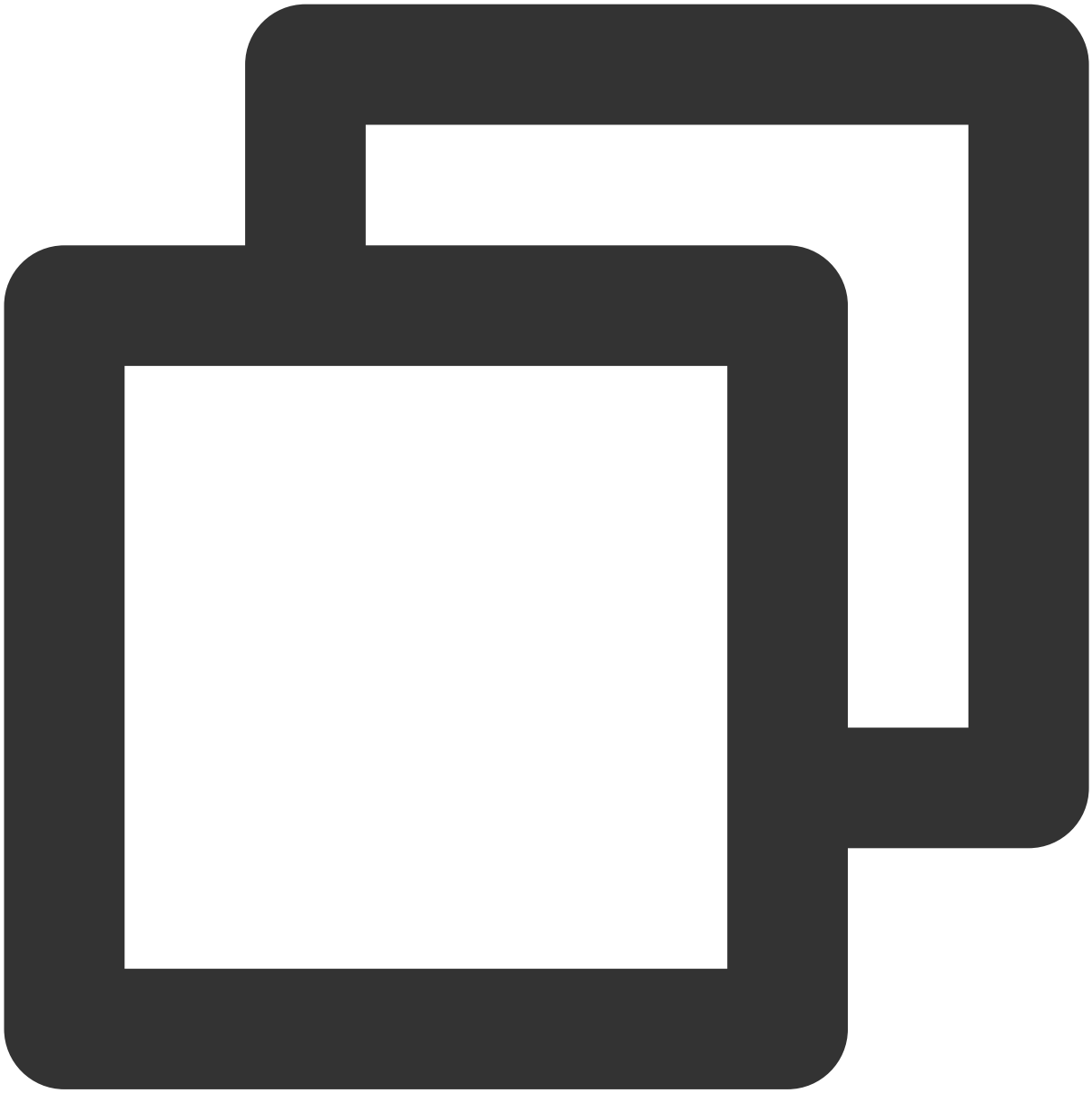
```
// 创建自定义消息
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createCustomMessage("群
```

```
// 发送消息
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, null, "receiver_groupID"
    @Override
    public void onProgress(int progress) {
        // 自定义消息不会回调进度
    }

    @Override
    public void onSuccess(V2TIMMessage message) {
        // 发送群聊自定义消息成功
    }

    @Override
    public void onError(int code, String desc) {
        // 发送群聊自定义消息失败
    }
});
```



```
V2TIMMessage *message = [[V2TIMManager sharedInstance] createCustomMessage:data];
[[V2TIMManager sharedInstance] sendMessage:message
                                  receiver:nil
                                   groupID:@"receiver_groupID"  // 群聊的 groupID
                                  priority:V2TIM_PRIORITY_DEFAULT  // 消息优先级
                            onlineUserOnly:NO
                           offlinePushInfo:nil
                                  progress:nil
                                      succ:^{
    // 群聊自定义消息发送成功
} fail:^(int code, NSString *desc) {
```

```
    // 群聊自定义消息发送失败
}];
```



```
class SendCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;

    SendCallback() = default;
    ~SendCallback() override = default;
```

```
    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     ProgressCallback progress_callback) {
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }

    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

// 创建自定义消息
V2TIMString str = u8"群聊自定义消息";
V2TIMBuffer customData = {reinterpret_cast<const uint8_t*>(str.CString()), str.Size
V2TIMMessage v2TIMMessage = V2TIMManager::GetInstance()->GetMessageManager()->Creat
// 发送消息
auto callback = new SendCallback{};
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 发送群聊自定义消息成功
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 发送群聊自定义消息失败
        delete callback;
    },
    [=](uint32_t progress) {
        // 自定义消息不会回调进度
```

```
    });
V2TIMManager::GetInstance()->GetMessageManager()->SendMessage(
    v2TIMMessage, {}, "receiver_groupID", V2TIMMessagePriority::V2TIM_PRIORITY_NORM
```

发送富媒体消息

富媒体消息的发送步骤如下：

1. 调用 `createXxxMessage` 创建指定类型的富媒体消息对象，其中 Xxx 表示具体的消息类型。
2. 调用 `sendMessage` (Android / iOS & Mac / Windows) 发送消息。
3. 在消息回调中获取消息是否发送成功或失败。

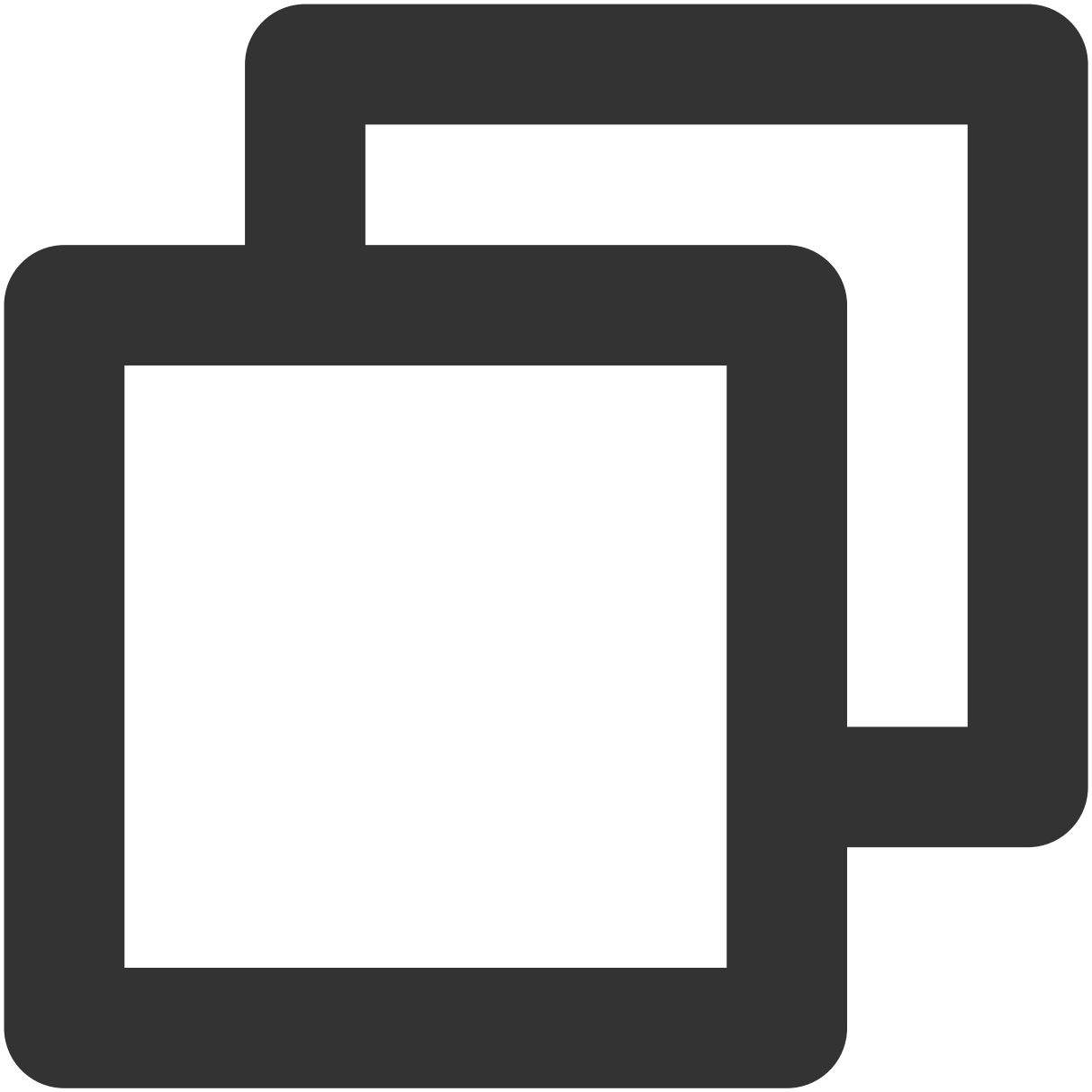
图片消息

创建图片消息需要先获取到本地图片路径。
发送消息过程中，会先将图片文件上传至服务器，同时回调上传进度。上传成功后再发送消息。
示例代码如下：

Android

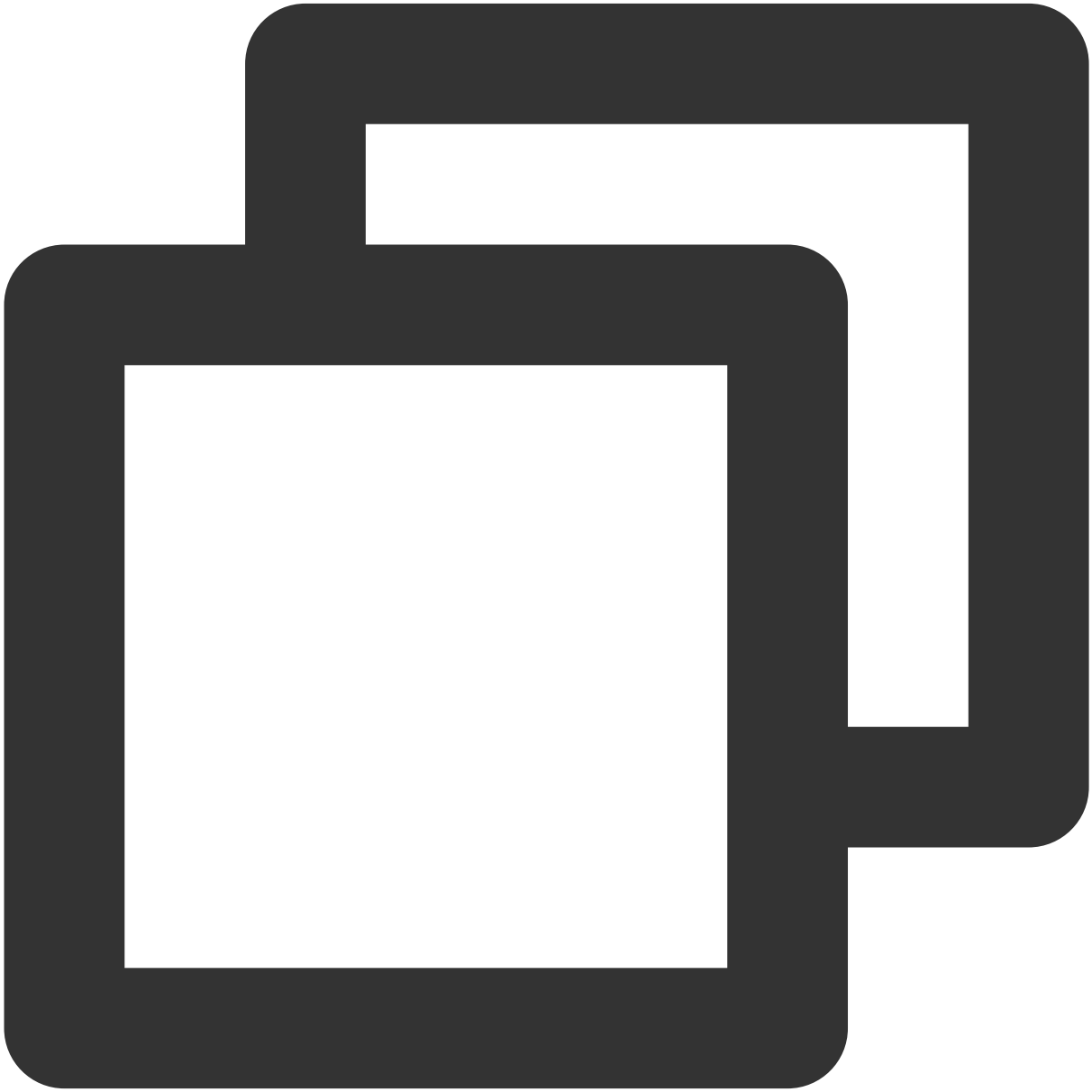
iOS & Mac

Windows



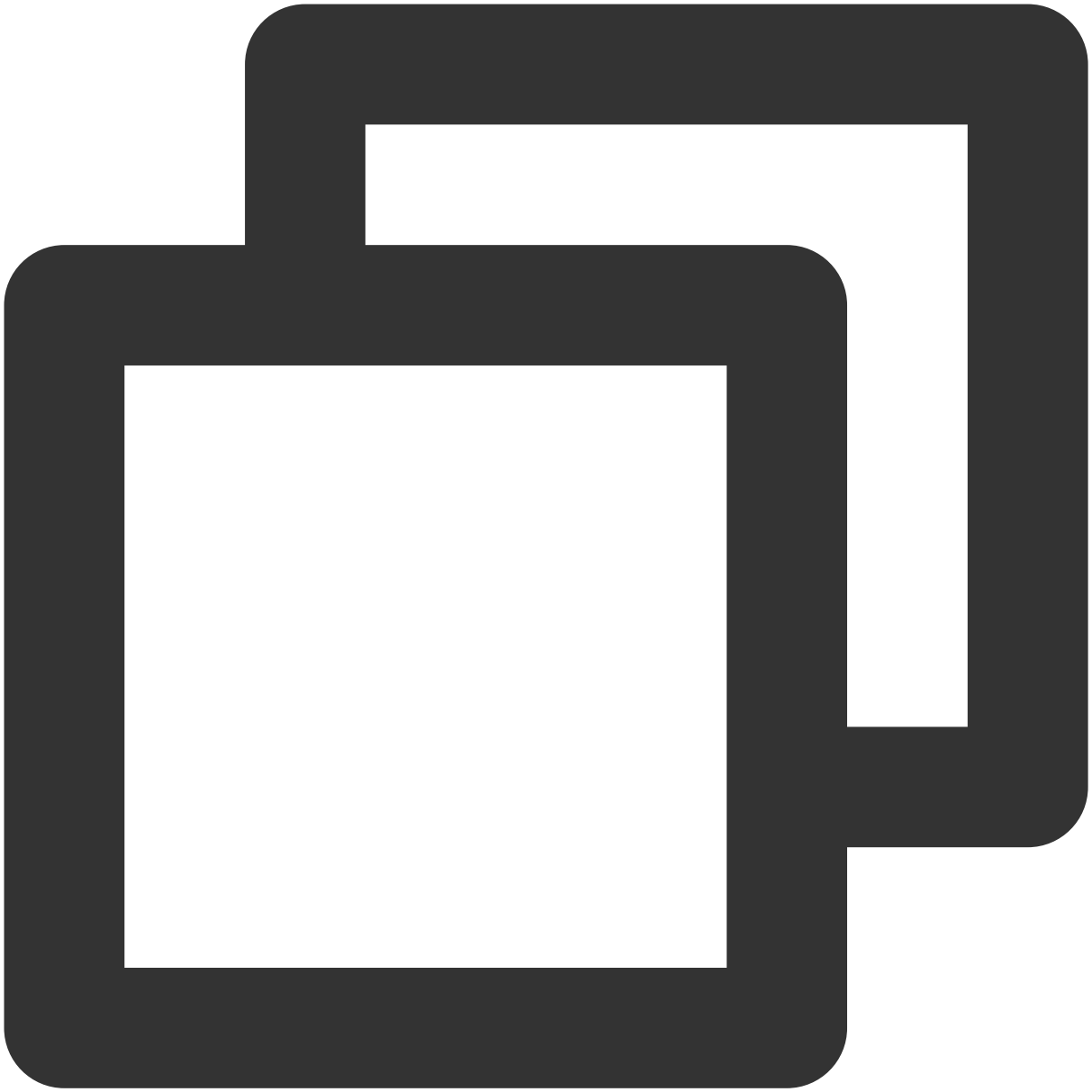
```
// 创建图片消息
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createImageMessage("/s
// 发送消息
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "receiver_userID", null,
    @Override
    public void onProgress(int progress) {
        // 图片上传进度, progress 取值 [0, 100]
    }

    @Override
    public void onSuccess(V2TIMMessage message) {
```

```
            // 图片消息发送成功
        }

        @Override
        public void onError(int code, String desc) {
            // 图片消息发送失败
        }
    });
```



```
// 获取本地图片路径
NSString *imagePath = [[NSBundle mainBundle] pathForResource:@"test" ofType:@"png"]
```

```
// 创建图片消息
V2TIMMessage *message = [[V2TIMManager sharedInstance] createImageMessage:imagePath
// 发送消息
[[V2TIMManager sharedInstance] sendMessage:message
                                  receiver:@"userID"
                                   groupID:nil
                                  priority:V2TIM_PRIORITY_DEFAULT
                            onlineUserOnly:NO
                           offlinePushInfo:nil
                                  progress:^(uint32_t progress) {
    // 图片上传进度, progress 取值 [0, 100]
} succ:^{
    // 图片消息发送成功
} fail:^(int code, NSString *desc) {
    // 图片消息发送成功
}];
```



```
class SendCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;

    SendCallback() = default;
    ~SendCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     ProgressCallback progress_callback) {
```

```
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }

    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

// 创建图片消息
V2TIMMessage v2TIMMessage =
    V2TIMManager::GetInstance()->GetMessageManager()->CreateImageMessage("./File/Xx
// 发送消息
auto callback = new SendCallback{};
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 图片消息发送成功
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 图片消息发送失败
        delete callback;
    },
    [=](uint32_t progress) {
        // 图片上传进度, progress 取值 [0, 100]
    });
V2TIMManager::GetInstance()->GetMessageManager()->SendMessage(
    v2TIMMessage, "receiver_userID", {}, V2TIMMessagePriority::V2TIM_PRIORITY_NORMA
```

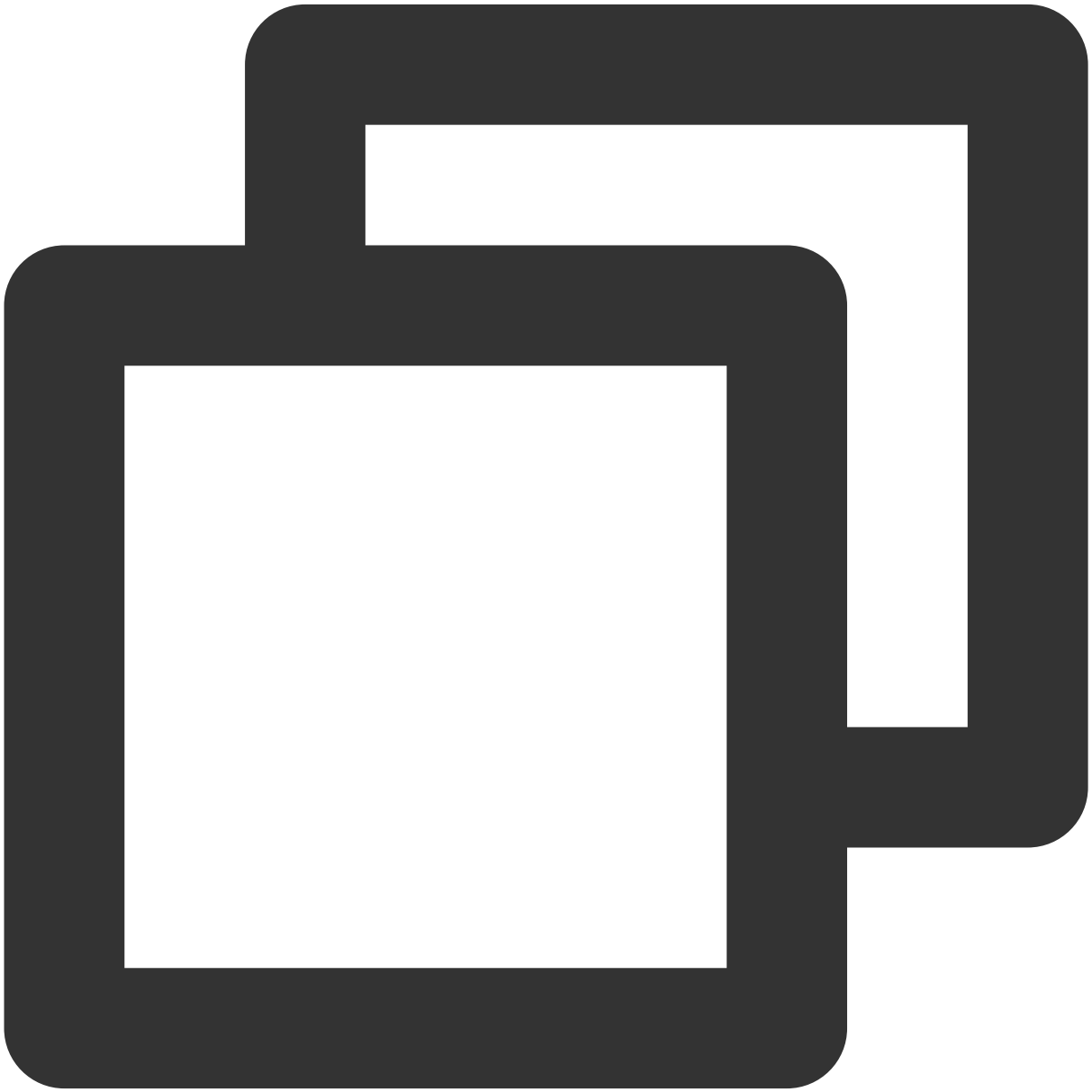
语音消息

创建语音消息需要先获取到本地语音文件路径和语音时长，其中语音时长可用于接收端 UI 显示。
发送消息过程中，会先将语音文件上传至服务器，同时回调上传进度。上传成功后再发送消息。
示例代码如下：

Android

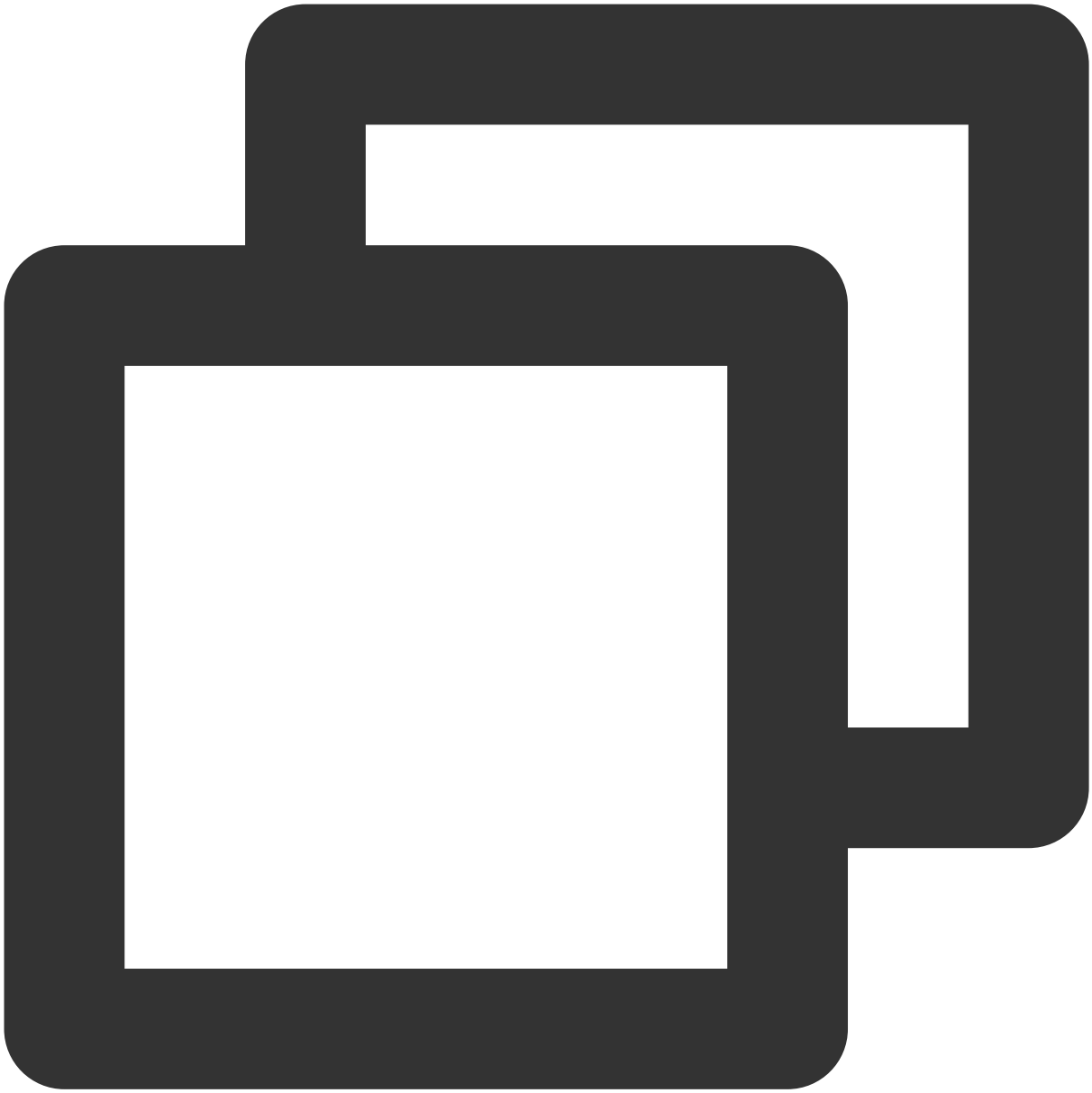
iOS & Mac

Windows



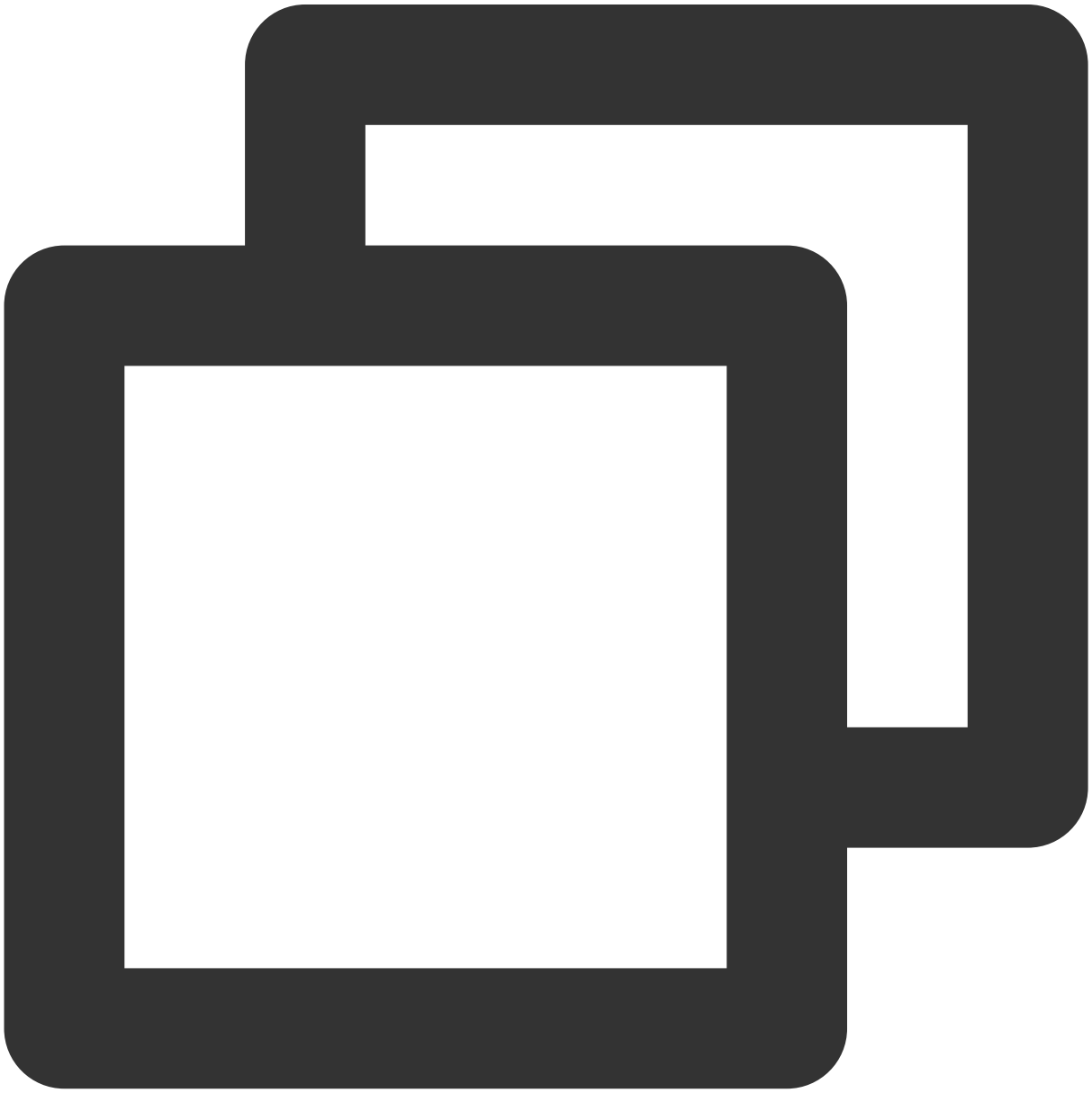
```
// 创建语音消息
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createSoundMessage("/s
```

```
// 发送消息
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "receiver_userID", null,
    @Override
    public void onProgress(int progress) {
        // 语音上传进度, progress 取值 [0, 100]
    }

    @Override
    public void onSuccess(V2TIMMessage message) {
        // 语音消息发送成功
    }

    @Override
    public void onError(int code, String desc) {
        // 语音消息发送失败
    }
});
```



```
// 获取本地语音文件路径
NSString *soundPath = [[NSBundle mainBundle] pathForResource:@"test" ofType:@"m4a"]
// 获取语音素材的时长（时长仅用于 UI 显示）
AVURLAsset *asset = [AVURLAsset assetWithURL:[NSURL fileURLWithPath:soundPath]];
CMTime time = [asset duration];
int duration = ceil(time.value/time.timescale);
// 创建语音消息
V2TIMMessage *message = [[V2TIMManager sharedInstance] createSoundMessage:soundPath
// 发送消息
[[V2TIMManager sharedInstance] sendMessage:message
                                  receiver:@"userID"
```

```
                                    groupID:nil
                                   priority:V2TIM_PRIORITY_DEFAULT
                             onlineUserOnly:NO
                            offlinePushInfo:nil
                                   progress:^(uint32_t progress) {
    // 语音上传进度, progress 取值 [0, 100]
} succ:^{
    // 语音消息发送成功
} fail:^(int code, NSString *desc) {
    // 语音消息发送失败
}];
```



```
class SendCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;

    SendCallback() = default;
    ~SendCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     ProgressCallback progress_callback) {
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }

    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

// 创建语音消息
V2TIMMessage v2TIMMessage =
    V2TIMManager::GetInstance()->GetMessageManager()->CreateSoundMessage("./File/Xx
// 发送消息
auto callback = new SendCallback{};
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 语音消息发送成功
```

```
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 语音消息发送失败
        delete callback;
    },
    [=](uint32_t progress) {
        // 语音上传进度，progress 取值 [0, 100]
    });
V2TIMManager::GetInstance()->GetMessageManager()->SendMessage(
    v2TIMMessage, "receiver_userID", {}, V2TIMMessagePriority::V2TIM_PRIORITY_NORMA
```

视频消息

创建视频消息需要先获取到本地视频文件路径、视频时长和视频快照，其中时长和快照可用于接收端 UI 显示。

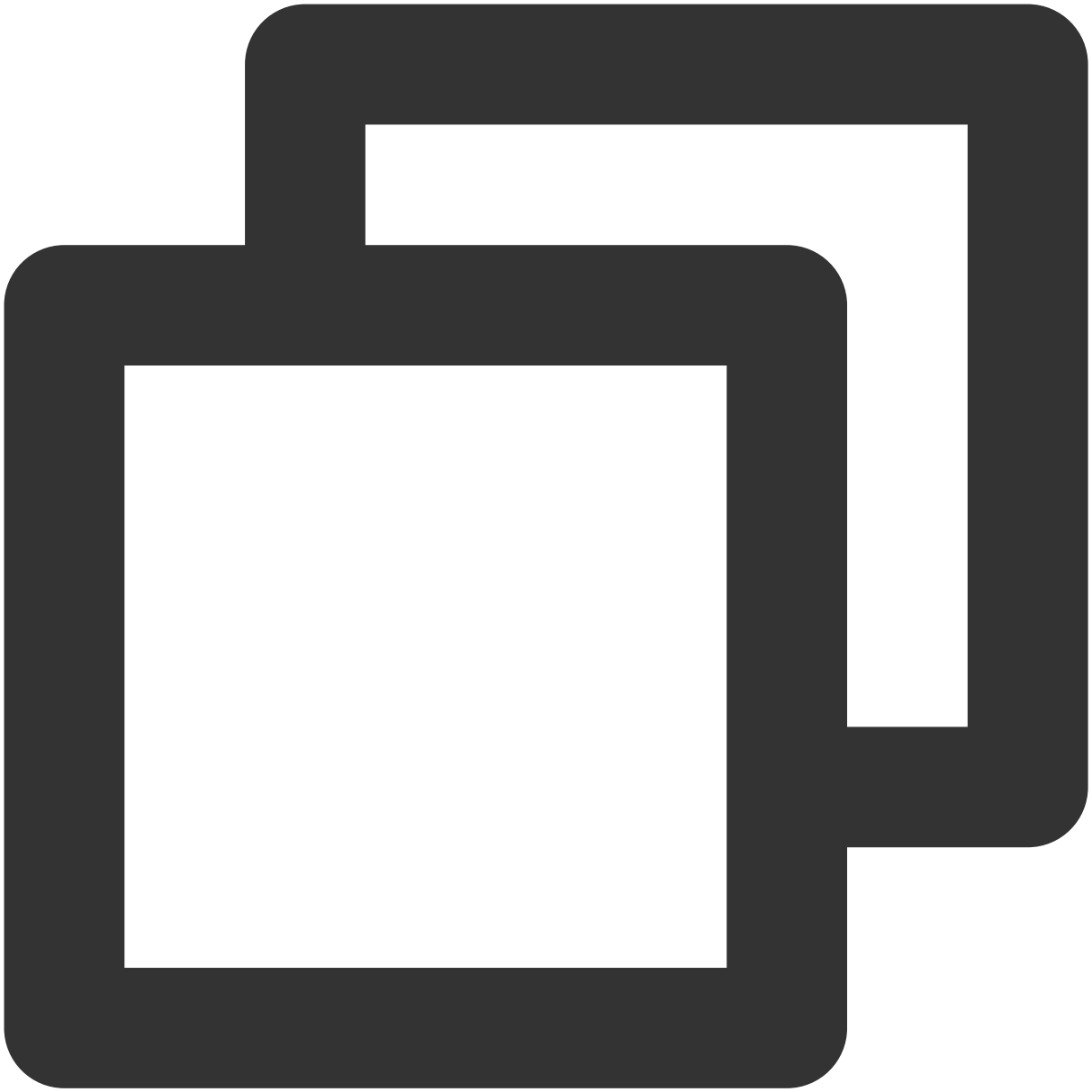
发送消息过程中，会先将视频上传至服务器，同时回调上传进度。上传成功后再发送消息。

示例代码如下：

Android

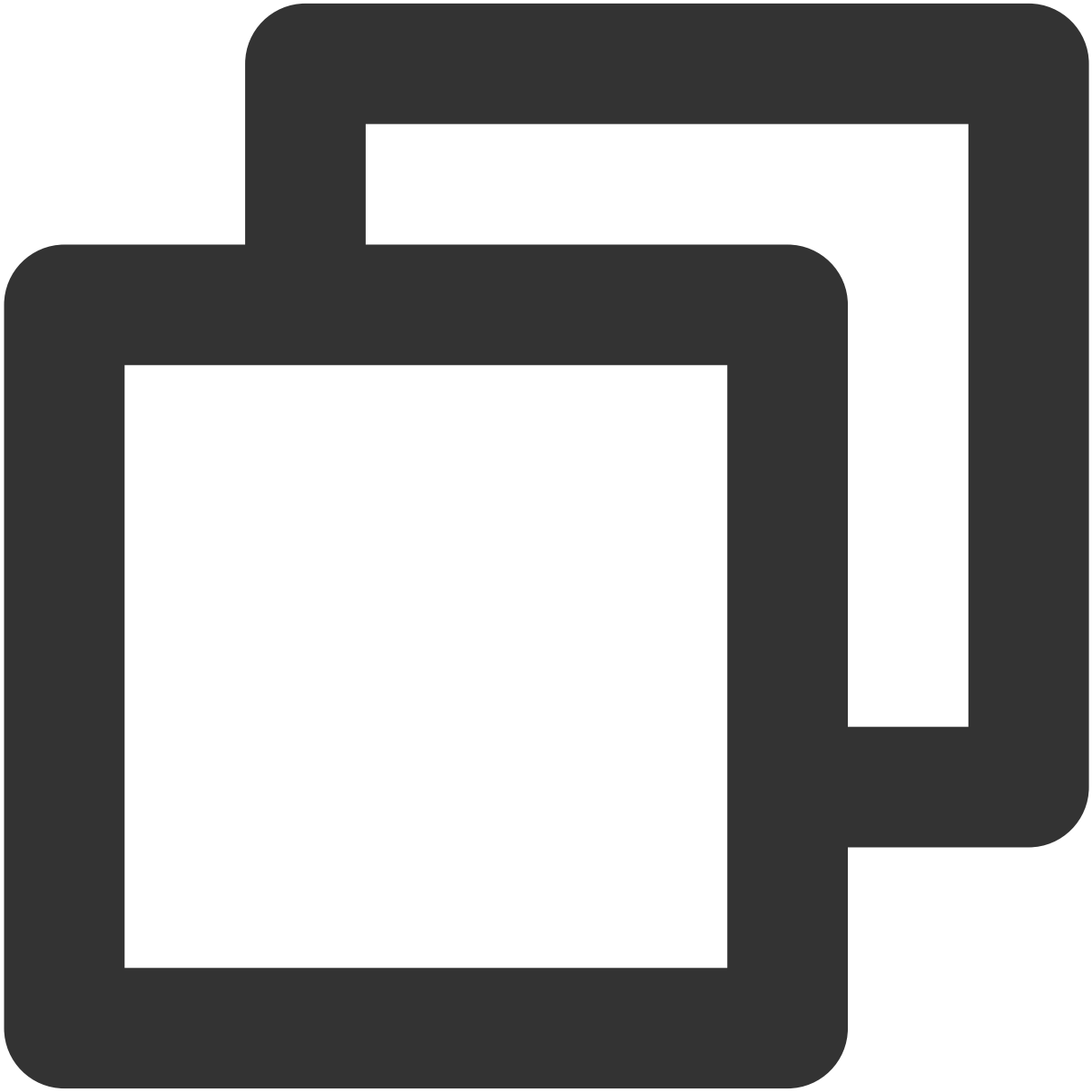
iOS & Mac

Windows



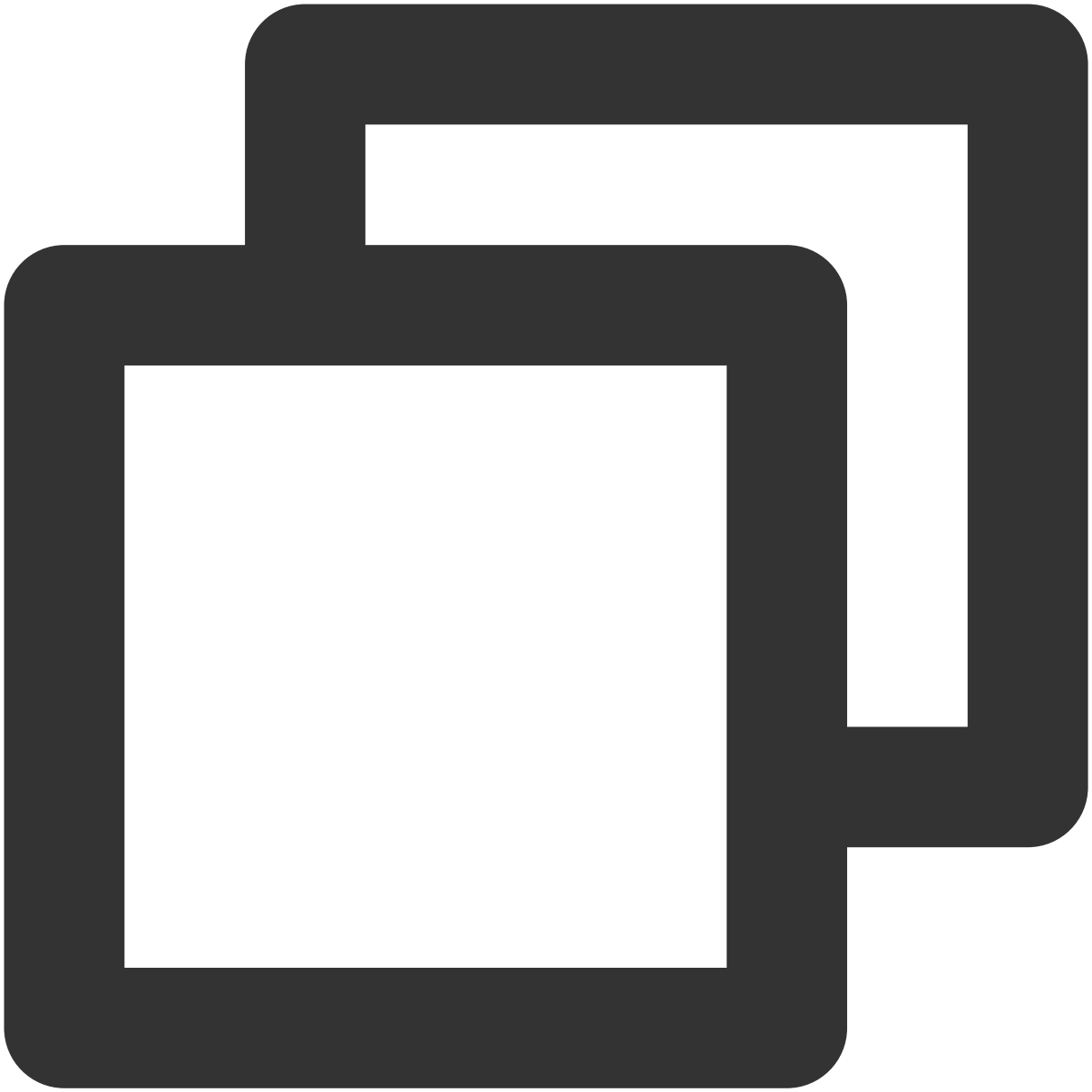
```
// 创建视频消息
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createVideoMessage("/s
// 发送消息
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "receiver_userID", null,
    @Override
    public void onProgress(int progress) {
        // 视频上传进度, progress 取值 [0, 100]
    }

    @Override
    public void onSuccess(V2TIMMessage message) {
```

```
            // 视频消息发送成功
        }

        @Override
        public void onError(int code, String desc) {
            // 视频消息发送失败
        }
    });
```



```
// 获取本地视频文件路径
NSString *videoPath = [[NSBundle mainBundle] pathForResource:@"test" ofType:@"mp4"]
```

```
// 获取本地视频快照路径
NSString *snapShotPath = [[NSBundle mainBundle] pathForResource:@"testpng" ofType:@
// 获取视频时长
AVURLAsset *asset = [AVURLAsset assetWithURL:[NSURL fileURLWithPath:path]];
CMTime time = [asset duration];
int duration = ceil(time.value/time.timescale);
// 创建视频消息
V2TIMMessage *message = [[V2TIMManager sharedInstance] createVideoMessage:videoPath
                                                                     type:@"mp4"
                                                                 duration:duration
                                                             snapshotPath:snapShotP
// 发送消息
[[V2TIMManager sharedInstance] sendMessage:message
                                  receiver:@"userID"
                                   groupID:nil
                                  priority:V2TIM_PRIORITY_DEFAULT
                            onlineUserOnly:NO
                           offlinePushInfo:nil
                                  progress:^(uint32_t progress) {
    // 视频上传进度, progress 取值 [0, 100]
} succ:^{
    // 视频消息发送成功
} fail:^(int code, NSString *desc) {
    // 视频消息发送失败
}];
```



```
class SendCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;

    SendCallback() = default;
    ~SendCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     ProgressCallback progress_callback) {
```

```
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }

    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

// 创建视频消息
V2TIMMessage v2TIMMessage = V2TIMManager::GetInstance()->GetMessageManager()->Creat
    "./File/Xxx.mp4", "mp4", 10, "./File/Xxx.jpg");
// 发送消息
auto callback = new SendCallback{};
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 视频消息发送成功
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 视频消息发送失败
        delete callback;
    },
    [=](uint32_t progress) {
        // 视频上传进度, progress 取值 [0, 100]
    });
V2TIMManager::GetInstance()->GetMessageManager()->SendMessage(
    v2TIMMessage, "receiver_userID", {}, V2TIMMessagePriority::V2TIM_PRIORITY_NORMA
```

文件消息

创建文件消息需要先获取到本地文件路径。

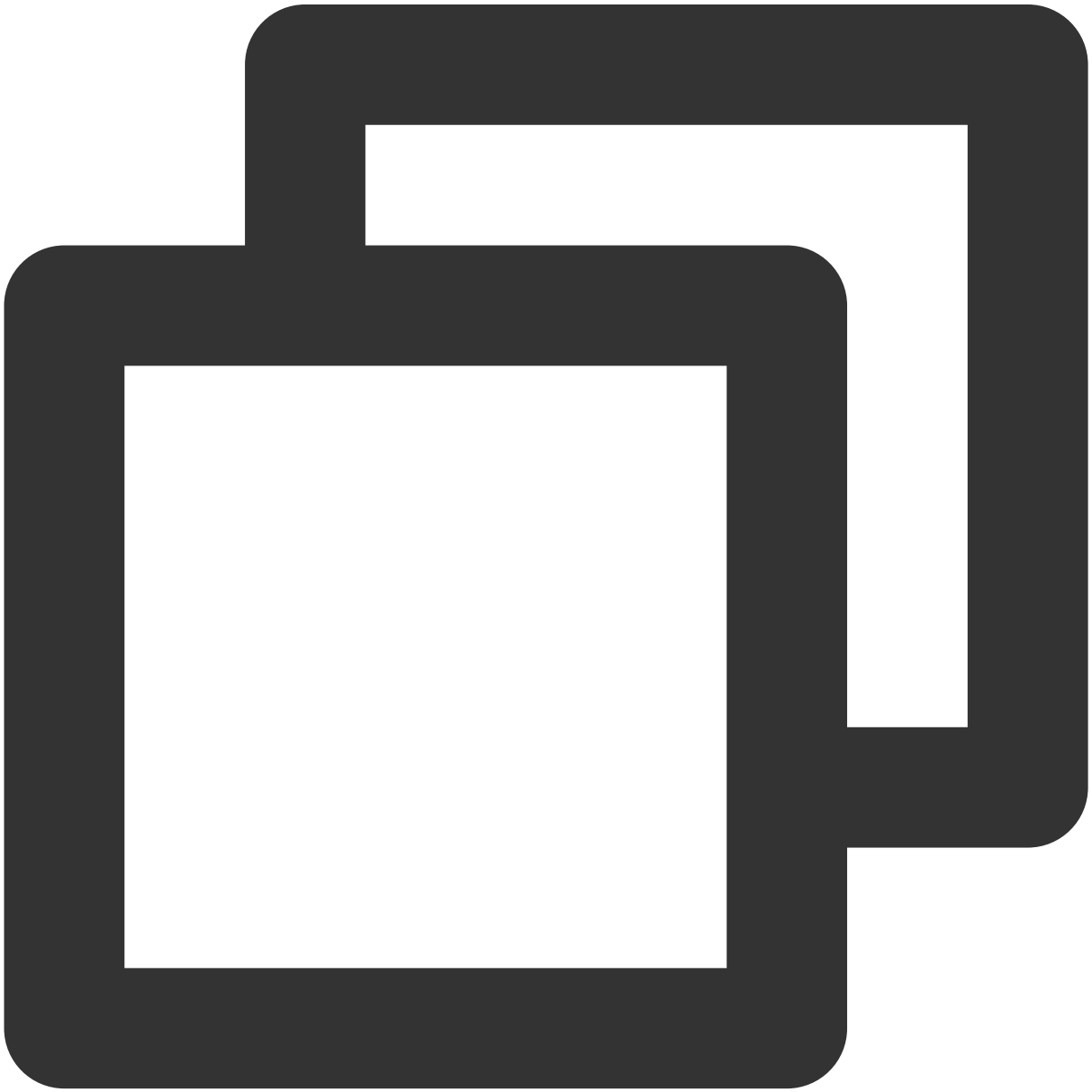
发送消息过程中，会先将文件上传至服务器，同时回调上传进度。上传成功后再发送消息。

示例代码如下：

Android

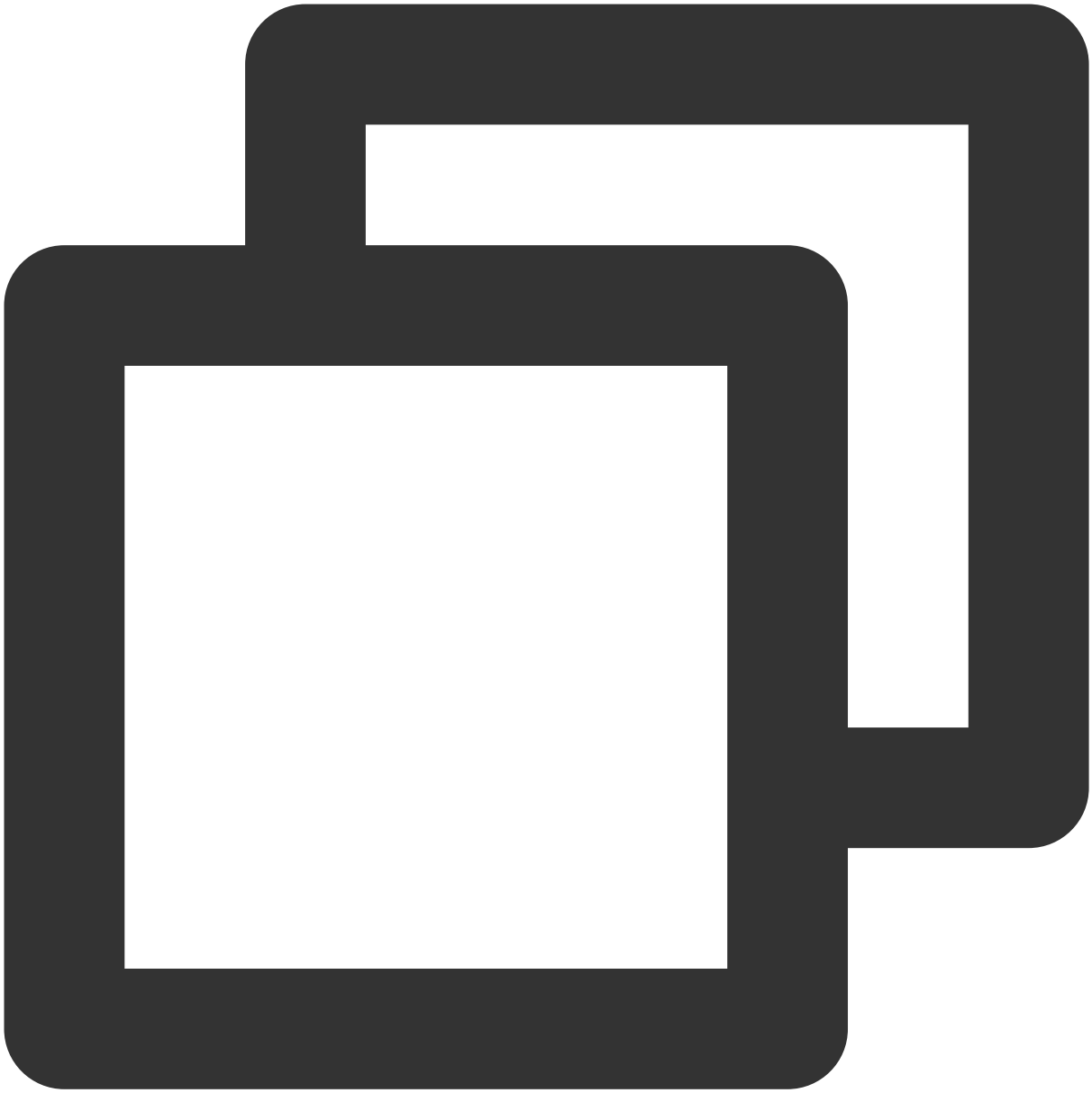
iOS & Mac

Windows



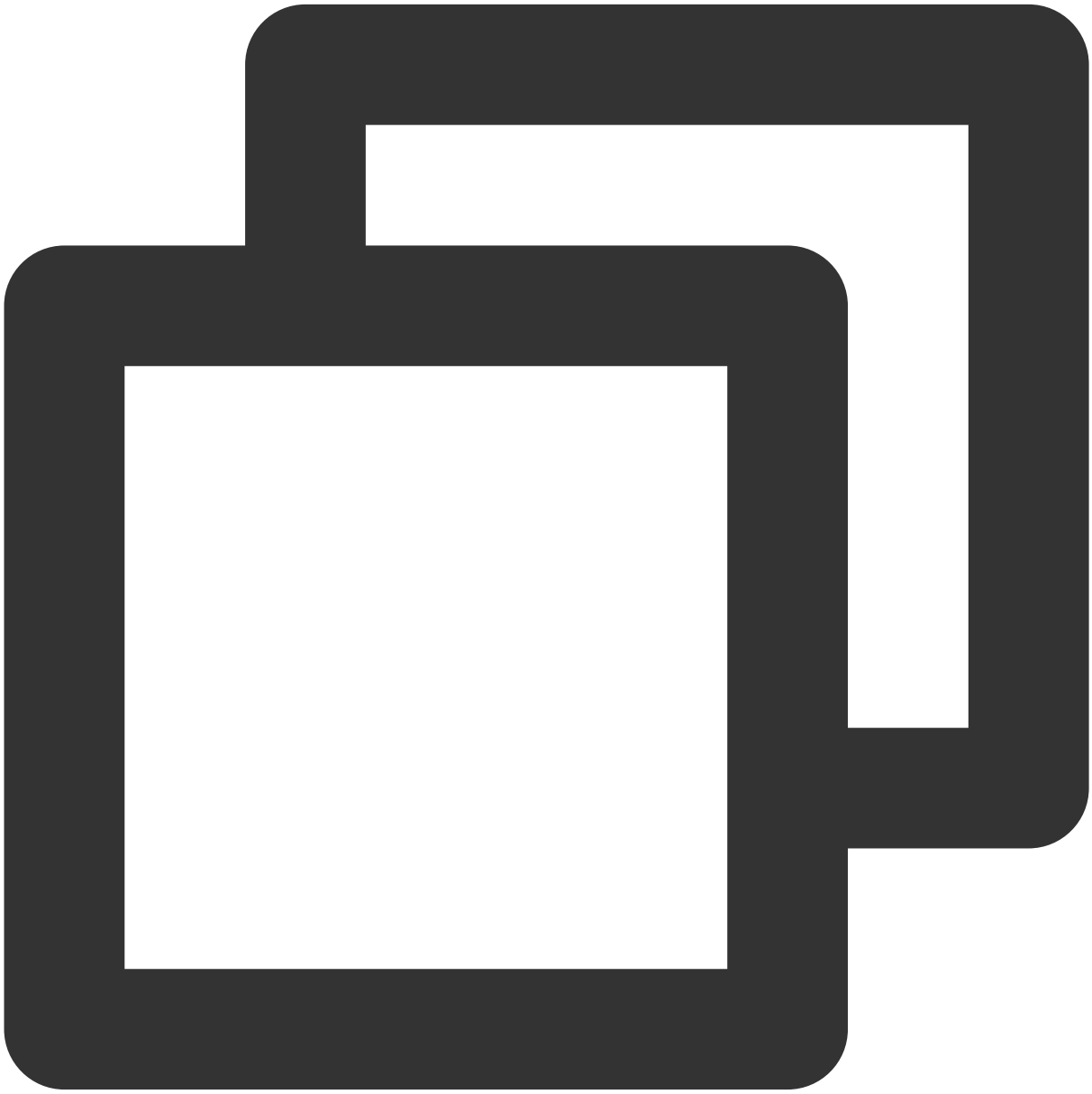
```
// 创建文件消息
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createFileMessage("/sd
```

```
// 发送消息
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "receiver_userID", null,
    @Override
    public void onProgress(int progress) {
        // 文件上传进度, progress 取值 [0, 100]
    }

    @Override
    public void onSuccess(V2TIMMessage message) {
        // 文件消息发送成功
    }

    @Override
    public void onError(int code, String desc) {
        // 文件消息发送失败
    }
});
```



```
// 获取本地文件路径
NSString *filePath = [[NSBundle mainBundle] pathForResource:@"test" ofType:@"mp4"];
// 创建文件消息
V2TIMMessage *message = [[V2TIMManager sharedInstance] createFileMessage:filePath f
// 发送消息
[[V2TIMManager sharedInstance] sendMessage:message
                                  receiver:@"userID"
                                   groupID:nil
                                  priority:V2TIM_PRIORITY_DEFAULT
                           onlineUserOnly:NO
                          offlinePushInfo:nil
```

```
                                    progress:^(uint32_t progress) {
    // 文件上传进度, progress 取值 [0, 100]
} succ:^{
    // 文件消息发送成功
} fail:^(int code, NSString *desc) {
    // 文件消息发送失败
}];
```



```
class SendCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
```

```
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;

    SendCallback() = default;
    ~SendCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     ProgressCallback progress_callback) {
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }

    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

// 创建文件消息
V2TIMMessage v2TIMMessage =
    V2TIMManager::GetInstance()->GetMessageManager()->CreateFileMessage("./File/Xxx
// 发送消息
auto callback = new SendCallback{};
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 文件消息发送成功
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 文件消息发送失败
```

```
        delete callback;
    },
    [=](uint32_t progress) {
        // 文件上传进度，progress 取值 [0, 100]
    });
V2TIMManager::GetInstance()->GetMessageManager()->SendMessage(
    v2TIMMessage, "receiver_userID", {}, V2TIMMessagePriority::V2TIM_PRIORITY_NORMA
```

定位消息

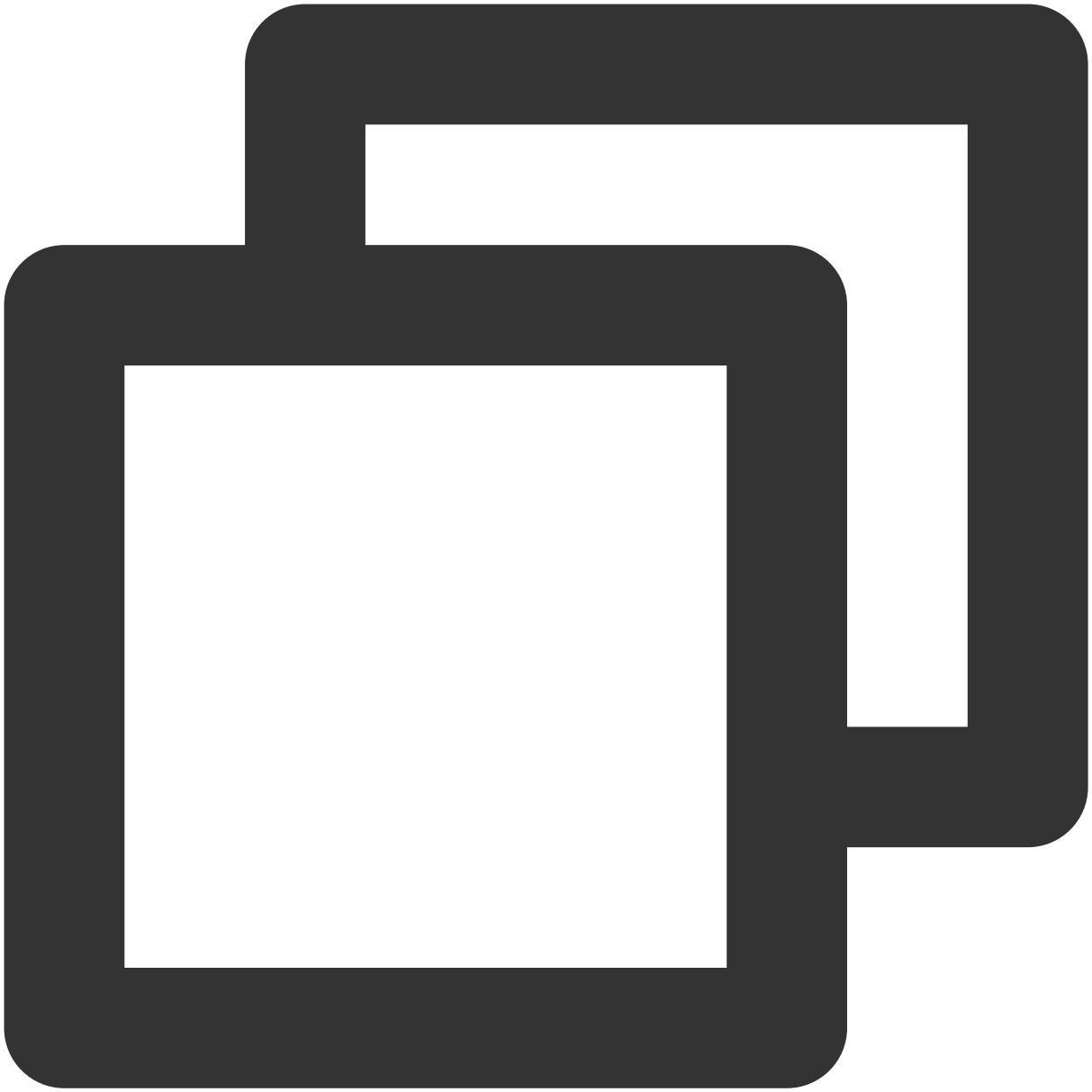
定位消息会直接发送经纬度，一般需要配合地图控件显示。

示例代码如下：

Android

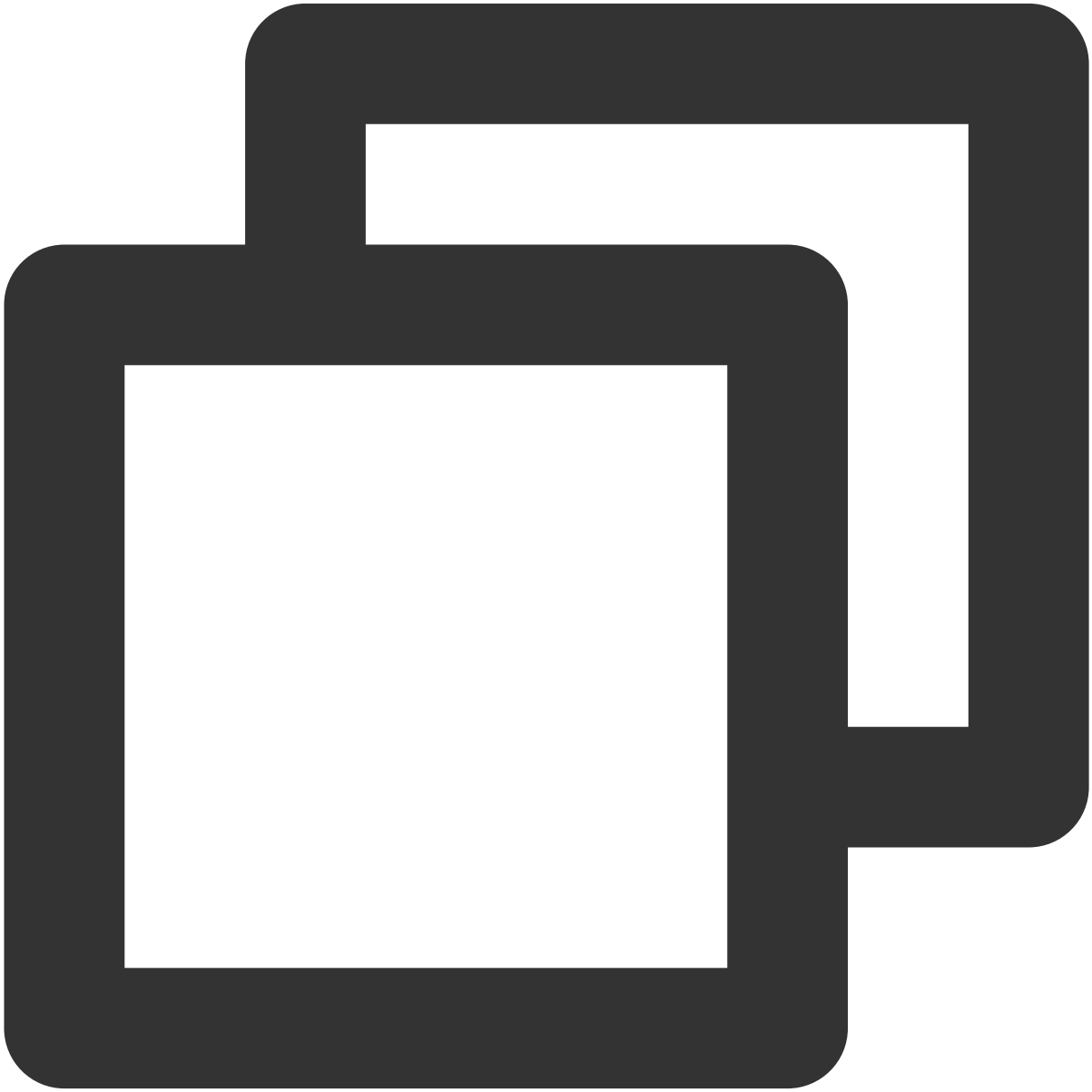
iOS & Mac

Windows



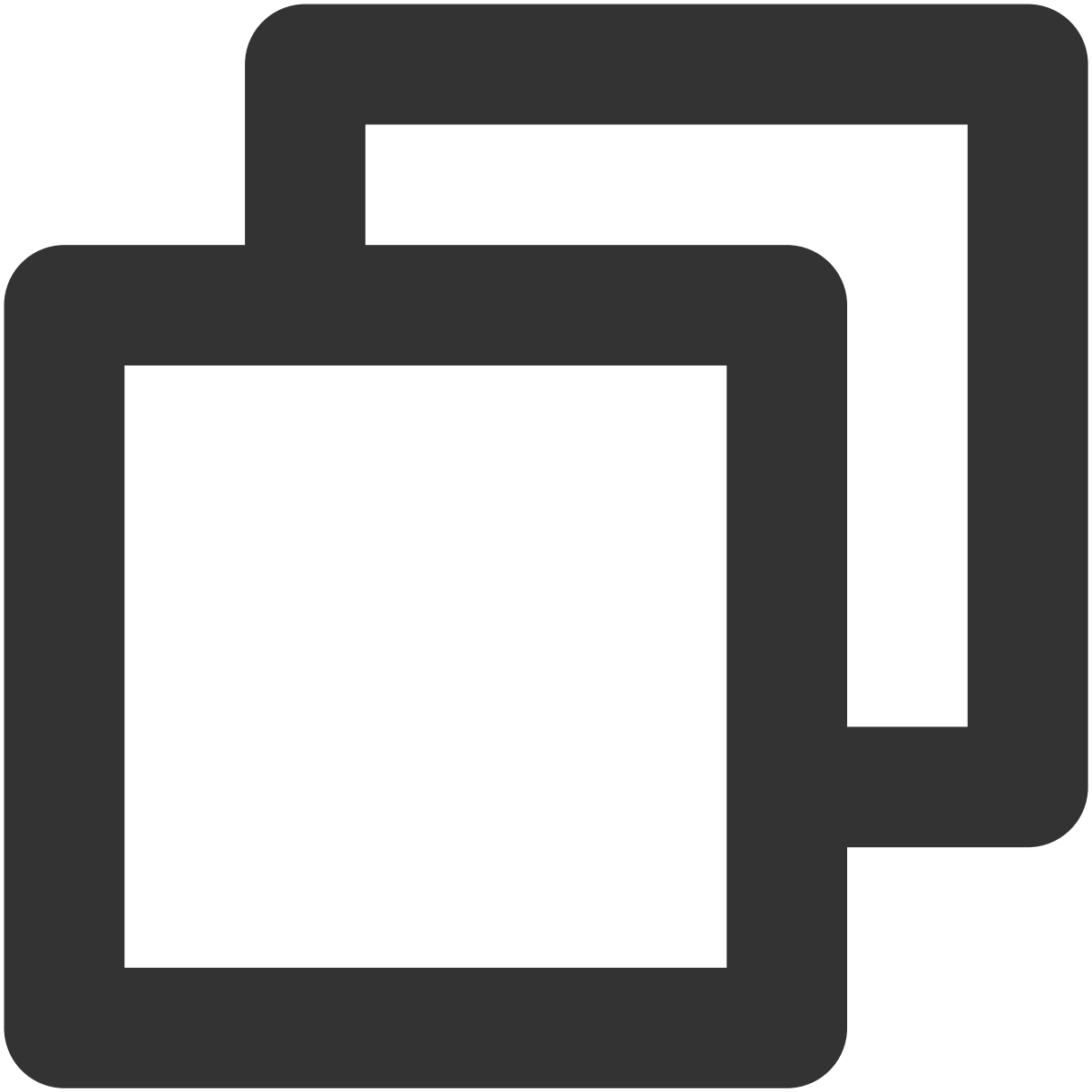
```
// 创建定位消息
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createLocationMessage(
// 发送消息
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "receiver_userID", null,
    @Override
    public void onProgress(int progress) {
        // 定位消息不回调进度
    }

    @Override
    public void onSuccess(V2TIMMessage message) {
```

```
            // 定位消息发送成功
        }

        @Override
        public void onError(int code, String desc) {
            // 定位消息发送失败
        }
    });
```



```
// 创建定位消息
V2TIMMessage *message = [[V2TIMManager sharedInstance] createLocationMessage:@"发送地
```

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



```
class SendCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;

    SendCallback() = default;
    ~SendCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     ProgressCallback progress_callback) {
```

```
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }

    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

// 创建定位消息
V2TIMMessage v2TIMMessage =
    V2TIMManager::GetInstance()->GetMessageManager()->CreateLocationMessage("地理位置
// 发送消息
auto callback = new SendCallback{};
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 定位消息发送成功
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 定位消息发送失败
        delete callback;
    },
    [=](uint32_t progress) {
        // 定位消息不回调进度
    });
V2TIMManager::GetInstance()->GetMessageManager()->SendMessage(
    v2TIMMessage, "receiver_userID", {}, V2TIMMessagePriority::V2TIM_PRIORITY_NORMA
```

表情消息

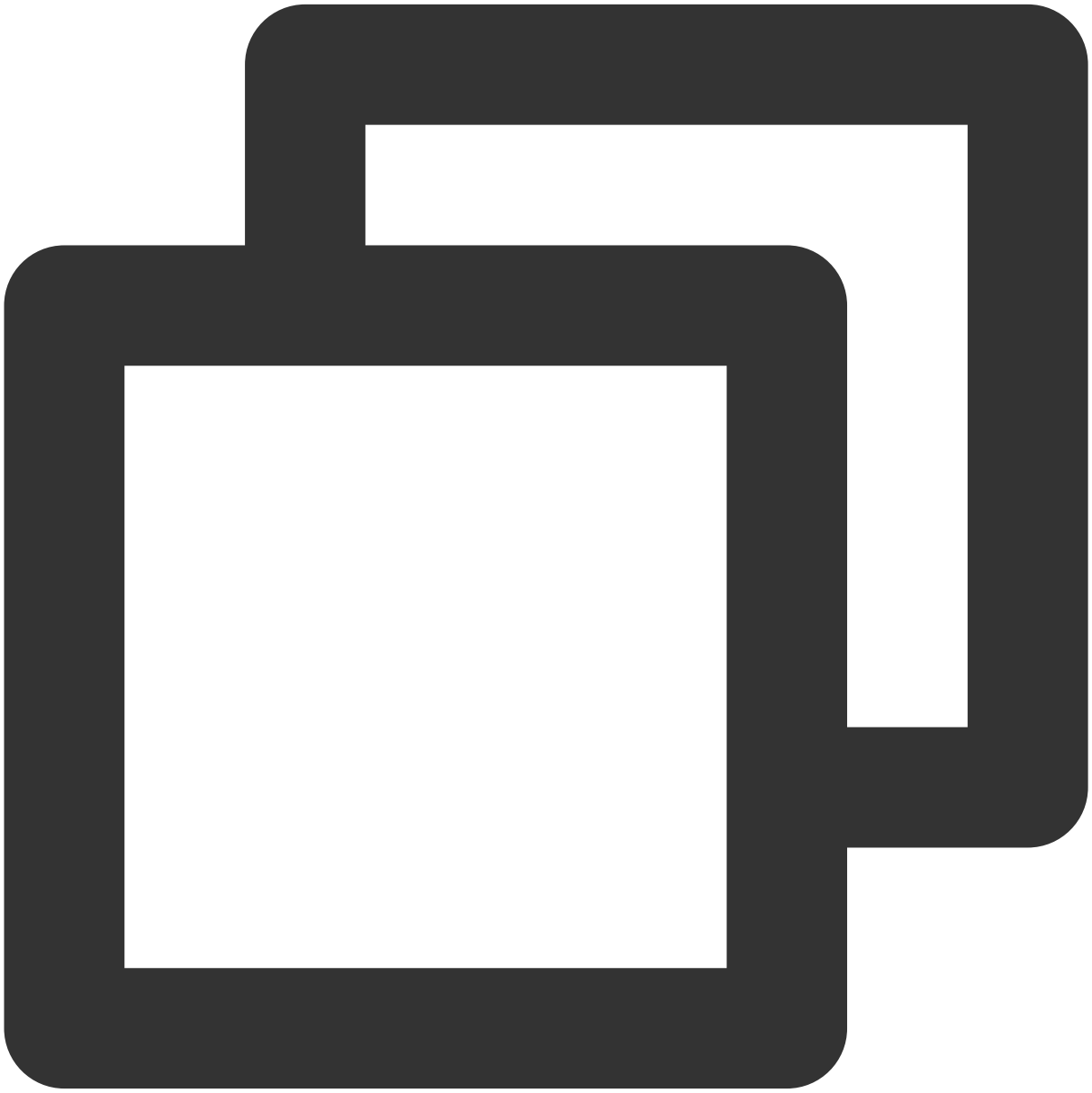
表情消息会直接发送表情编码，通常接收端需要将其转换成对应的表情 icon。

示例代码如下：

Android

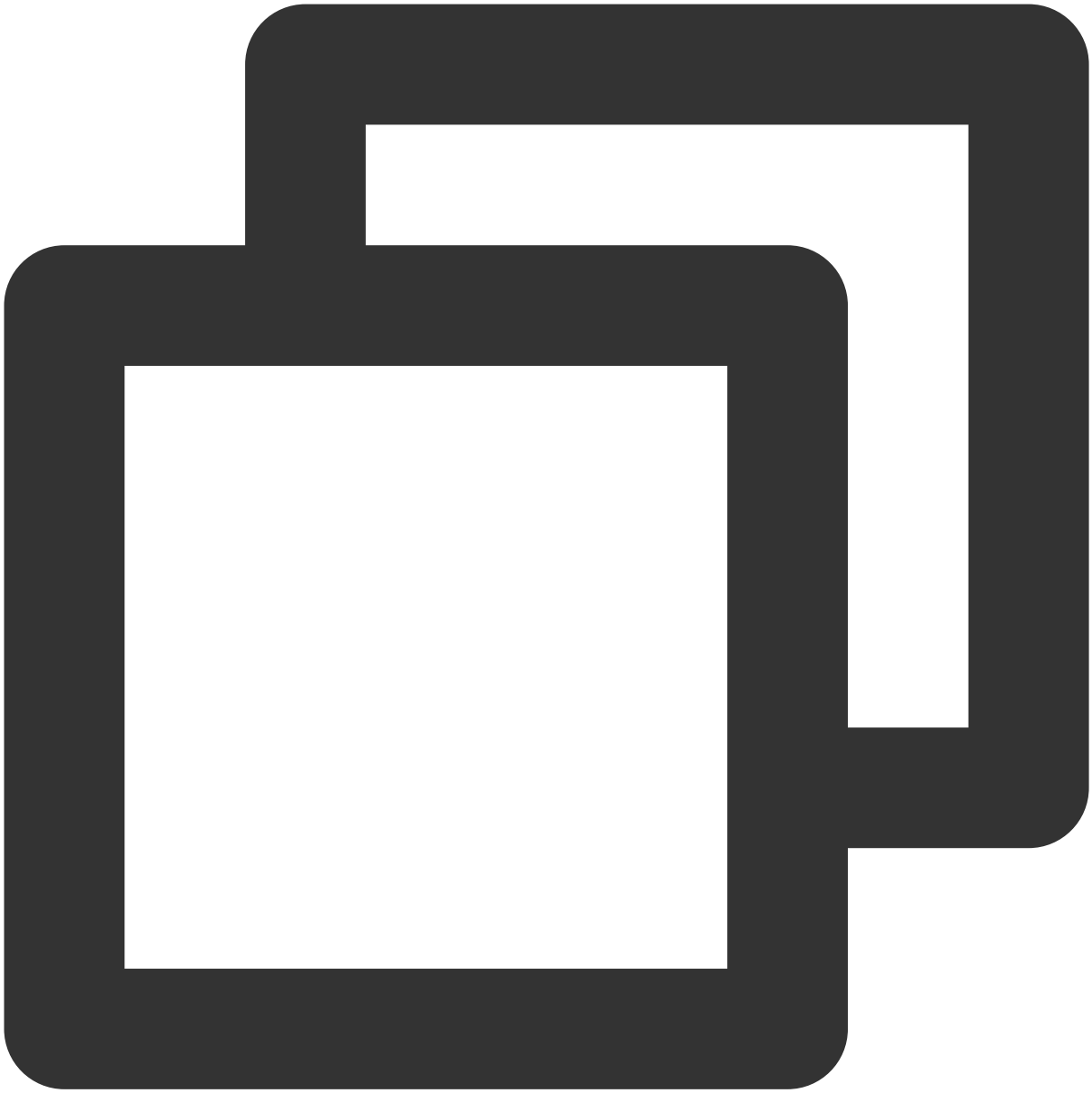
iOS & Mac

Windows



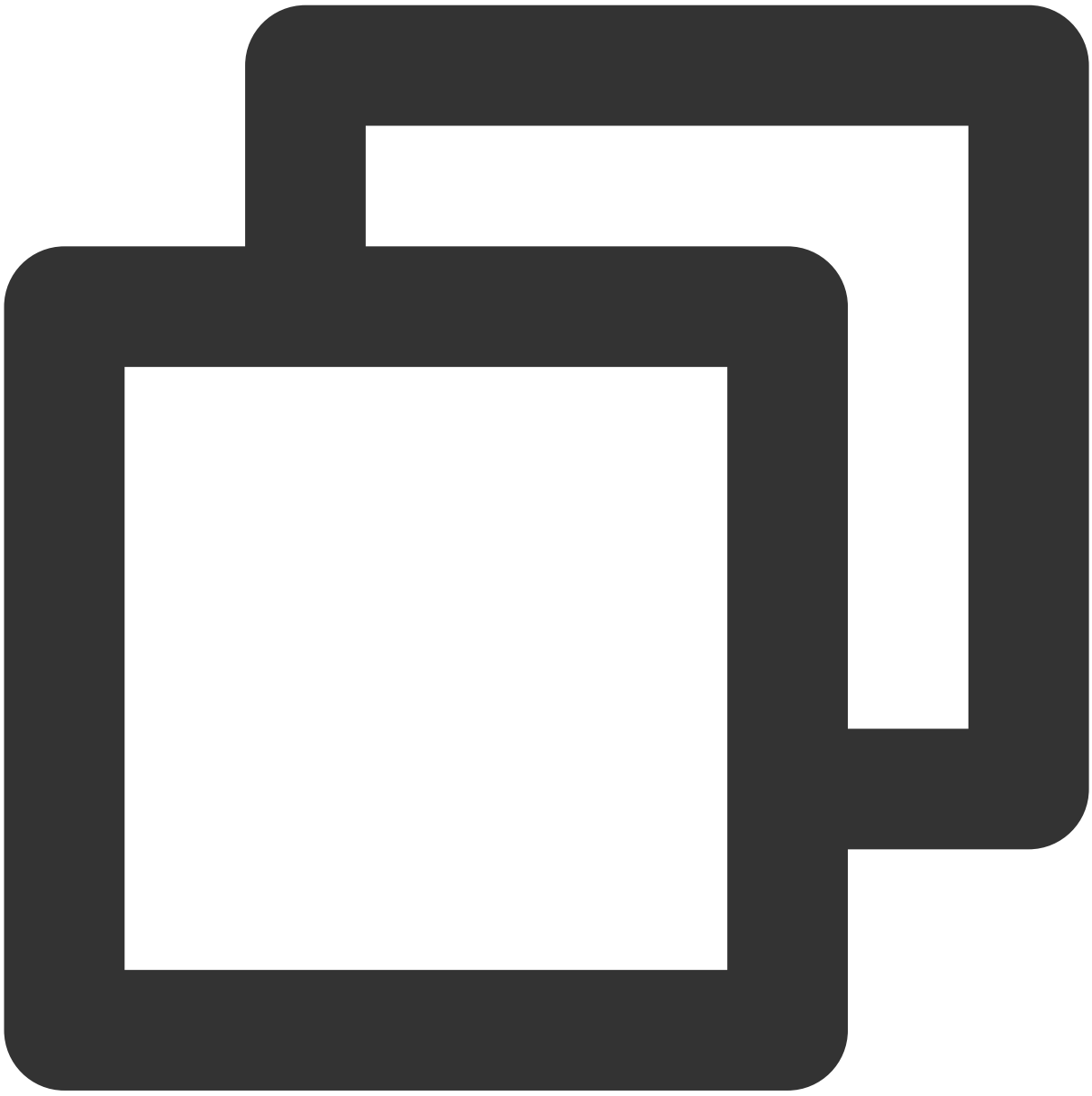
```
// 创建表情消息
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createFaceMessage(1, "
// 发送消息
```

```
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "receiver_userID", null,
    @Override
    public void onProgress(int progress) {
        // 表情消息不回调进度
    }

    @Override
    public void onSuccess(V2TIMMessage message) {
        // 表情消息发送成功
    }

    @Override
    public void onError(int code, String desc) {
        // 表情消息发送失败
    }
});
```



```
// 创建表情消息
V2TIMMessage *message = [[V2TIMManager sharedInstance] createFaceMessage:1 data:[@"
// 发送消息
[[V2TIMManager sharedInstance] sendMessage:message
                                  receiver:@"userID"
                                   groupID:nil
                                  priority:V2TIM_PRIORITY_DEFAULT
                            onlineUserOnly:NO
                           offlinePushInfo:nil
                                  progress:nil
                                      succ:^{
```

```
    // 表情消息发送成功
} fail:^(int code, NSString *desc) {
    // 表情消息发送失败
}];
```



```
class SendCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;
```

```
    SendCallback() = default;
    ~SendCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     ProgressCallback progress_callback) {
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }

    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

// 创建表情消息
V2TIMString str = u8"tt00";
V2TIMBuffer data = {reinterpret_cast<const uint8_t*>(str.CString()), str.Size()};
V2TIMMessage v2TIMMessage = V2TIMManager::GetInstance()->GetMessageManager()->Creat
// 发送消息
auto callback = new SendCallback{};
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 表情消息发送成功
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 表情消息发送失败
        delete callback;
    },
```

```
    [=](uint32_t progress) {
        // 表情消息不回调进度
    });
V2TIMManager::GetInstance()->GetMessageManager()->SendMessage(
    v2TIMMessage, "receiver_userID", {}, V2TIMMessagePriority::V2TIM_PRIORITY_NORMA
```

发送多个 Elem 的消息

如果您的消息需要多个 elem，可以在创建 Message 对象后，通过 Message 对象的 Elem 成员调用 `appendElem` (Android / iOS & Mac / Windows) 方法添加下一个 elem 成员。

`appendElem` 仅支持在原有的 `V2TIMElem` （此 Elem 类型不限）后面追加 `V2TIMTextElem` 、 `V2TIMCustomElem` 、 `V2TIMFaceElem` 和 `V2TIMLocationElem` 四种类型的元素。

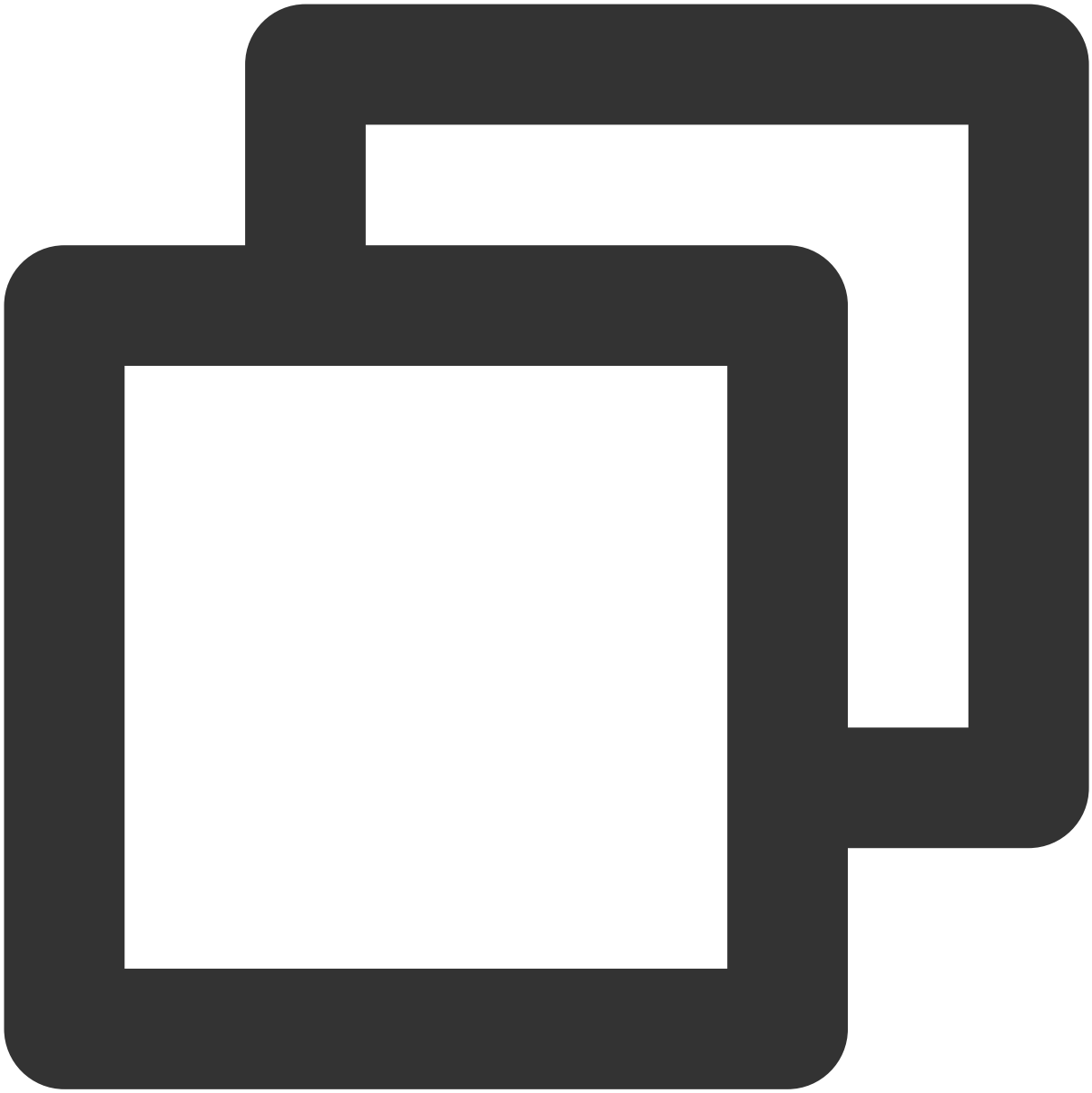
因此 “图片 + 文本”、“视频 + 文本”、“位置 + 文本” 这几种形式均可支持，但是 “图片 + 图片”、“文本 + 图片” 等不支持。

以文本消息 + 自定义消息为例，示例代码如下：

Android

iOS & Mac

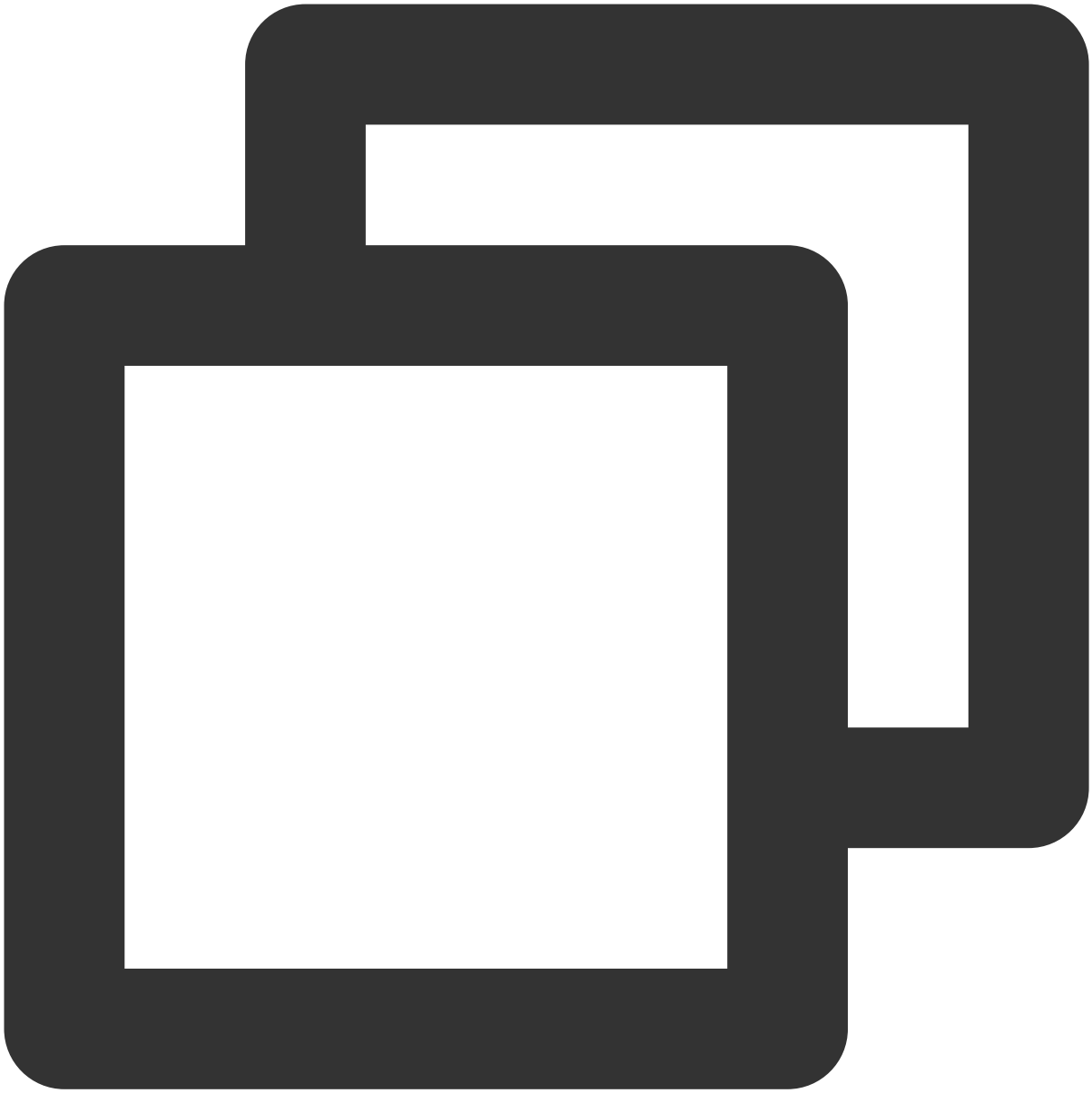
Windows



```
// 创建文本消息
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createTextMessage("tes
// 创建自定义 elem
V2TIMCustomElem customElem = new V2TIMCustomElem();
customElem.setData("自定义消息".getBytes());
// 将自定义 elem 添加到 message.textElem 中
v2TIMMessage.getTextElem().appendElem(customElem);
// 发送消息
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "receiver_userID", null,
    @Override
    public void onProgress(int progress) {
```

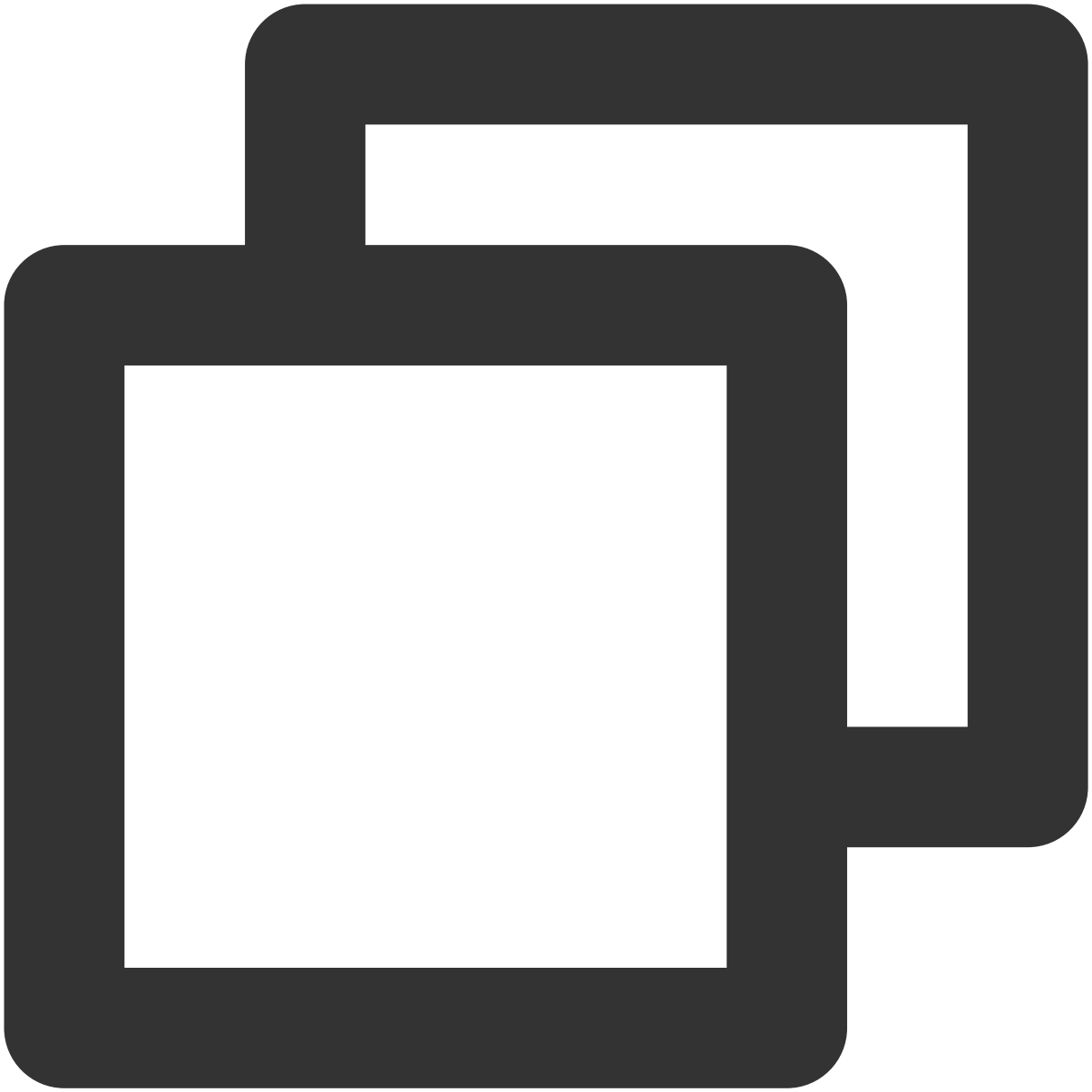
版权所有：腾讯云计算（北京）有限责任公司

```
            // 不回调进度
        }

        @Override
        public void onSuccess(V2TIMMessage message) {
            // 多 elem 消息发送成功
        }

        @Override
        public void onError(int code, String desc) {
            // 多 elem 消息发送失败
        }
    });
```



```
// 创建文本消息
V2TIMMessage *message = [[V2TIMManager sharedInstance] createTextMessage:@"text"];

// 创建自定义 elem
V2TIMCustomElem *customElem = [[V2TIMCustomElem alloc] init];
customElem.data = [@"自定义消息" dataUsingEncoding:NSUTF8StringEncoding];

// 将自定义 elem 添加到 message.textElem 中
[message.textElem appendElem:customElem];

// 发送消息
```

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



```
class SendCallback final : public V2TIMSendCallback {
public:
    using SuccessCallback = std::function<void(const V2TIMMessage&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using ProgressCallback = std::function<void(uint32_t)>;

    SendCallback() = default;
    ~SendCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     ProgressCallback progress_callback) {
```

```
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        progress_callback_ = std::move(progress_callback);
    }

    void OnSuccess(const V2TIMMessage& message) override {
        if (success_callback_) {
            success_callback_(message);
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnProgress(uint32_t progress) override {
        if (progress_callback_) {
            progress_callback_(progress);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    ProgressCallback progress_callback_;
};

// 创建文本消息
V2TIMMessage v2TIMMessage = V2TIMManager::GetInstance()->GetMessageManager()->Creat
// 创建自定义 elem
V2TIMCustomElem customElem;
V2TIMString str = u8"tt00";
customElem.data = {reinterpret_cast<const uint8_t*>(str.CString()), str.Size()};
// 将自定义 elem 添加到 message.elemList 中
v2TIMMessage.elemList.PushBack(&customElem);
// 发送消息
auto callback = new SendCallback{};
callback->SetCallback(
    [=](const V2TIMMessage& message) {
        // 多 elem 消息发送成功
        delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
        // 多 elem 消息发送失败
        delete callback;
    },
    [=](uint32_t progress) {
```

```
        // 不回调进度
    });
V2TIMManager::GetInstance()->GetMessageManager()->SendMessage(
    v2TIMMessage, "receiver_userID", {}, V2TIMMessagePriority::V2TIM_PRIORITY_NORMA
```

接口限制

功能特性	限制项	限制说明
单聊/群聊	内容长度	单聊、群聊消息，单条消息最大长度限制为 12K。
	发送频率	单聊消息：客户端发送单聊消息无限制；REST API 发送有频率限制，可查看相应接口的文档。 群聊消息：每个群限 40 条/秒（针对所有群类型、所有平台接口）。不同群内发消息，限频互不影响。
	接收频率	单聊和群聊均无限制。
	单个文件大小	发送文件消息时，SDK 最大支持发送单个文件大小为 100MB。

说明：

1. 消息数量超过限制后，后台优先下发优先级相对较高的消息，同等优先级的消息随机排序。如果同一秒内高优先级消息总数超过 40 条/秒，高优先级消息也会被抛弃。
2. 被频控限制的消息，不会下发，不会存入历史消息，但会给发送人返回成功，会触发 群内发言之前回调，但不会触发 群内发言之后回调。
3. REST API 发送群组消息接口调用限频默认为 200次/秒，与上述 “每个群发送消息限 40 条/秒” 是不同概念，请区分开。

更多限制请参见文档 使用限制。

Web

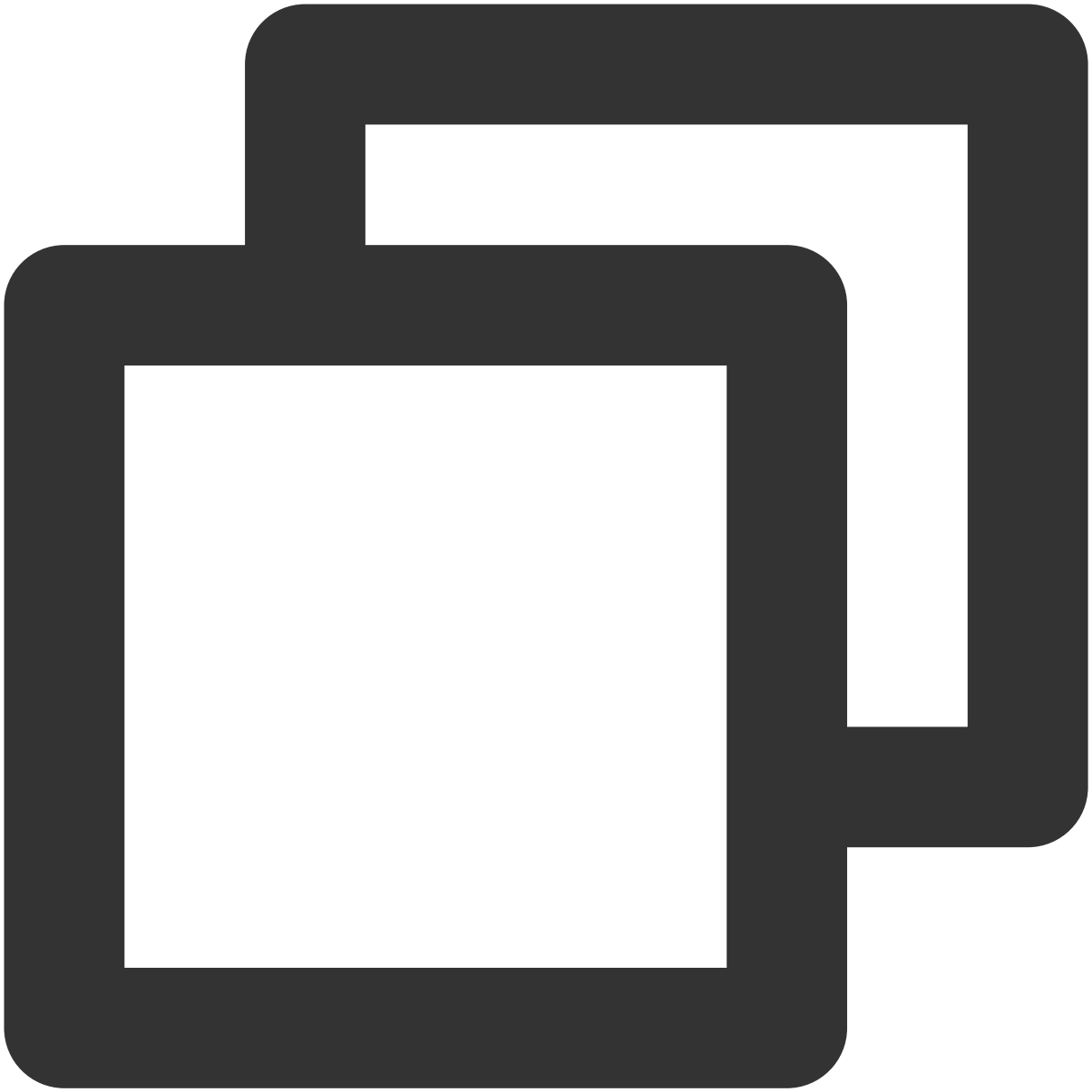
最近更新时间：2024-07-10 16:31:08

创建消息

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



```
chat.createTextMessage(options);
```

参数

参数 options 为 Object 类型，包含的属性值如下：

Name	Type	Default	Description
to	String	-	消息接收方的 us
conversationType	String	-	会话类型，取

			值 TencentCl 端会话） 或 TencentCl 组会话）
priority	String	TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	消息内容的容器
cloudCustomData	String	''	消息自定义数据 重装后还能拉取
receiverList	Array | undefined	-	定向接收消息的
isSupportExtension	Boolean	false	是否支持消息扩 买旗舰版套餐）

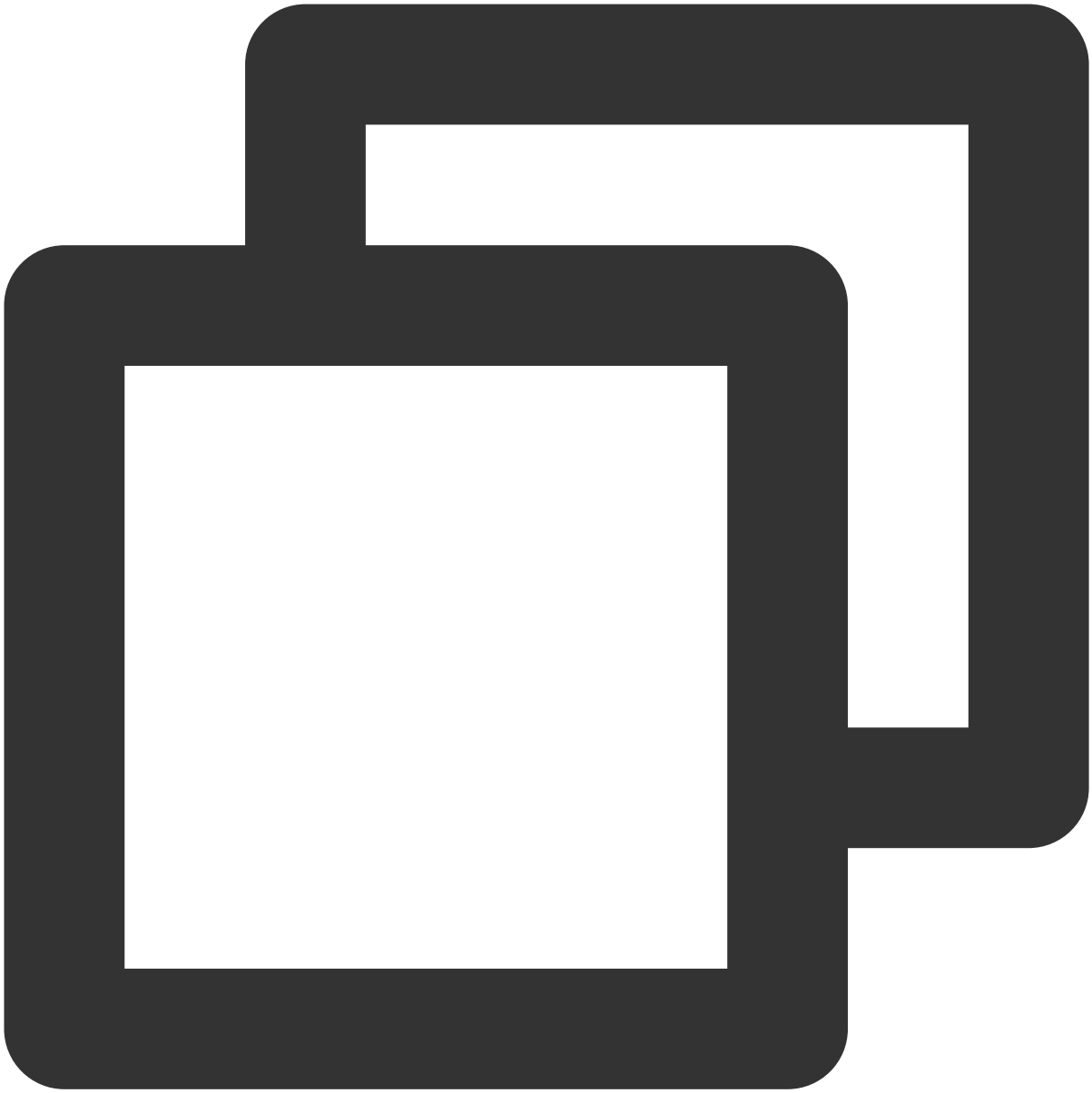
payload 的描述如下：

Name	Type	Description
text	String	消息文本内容

返回值

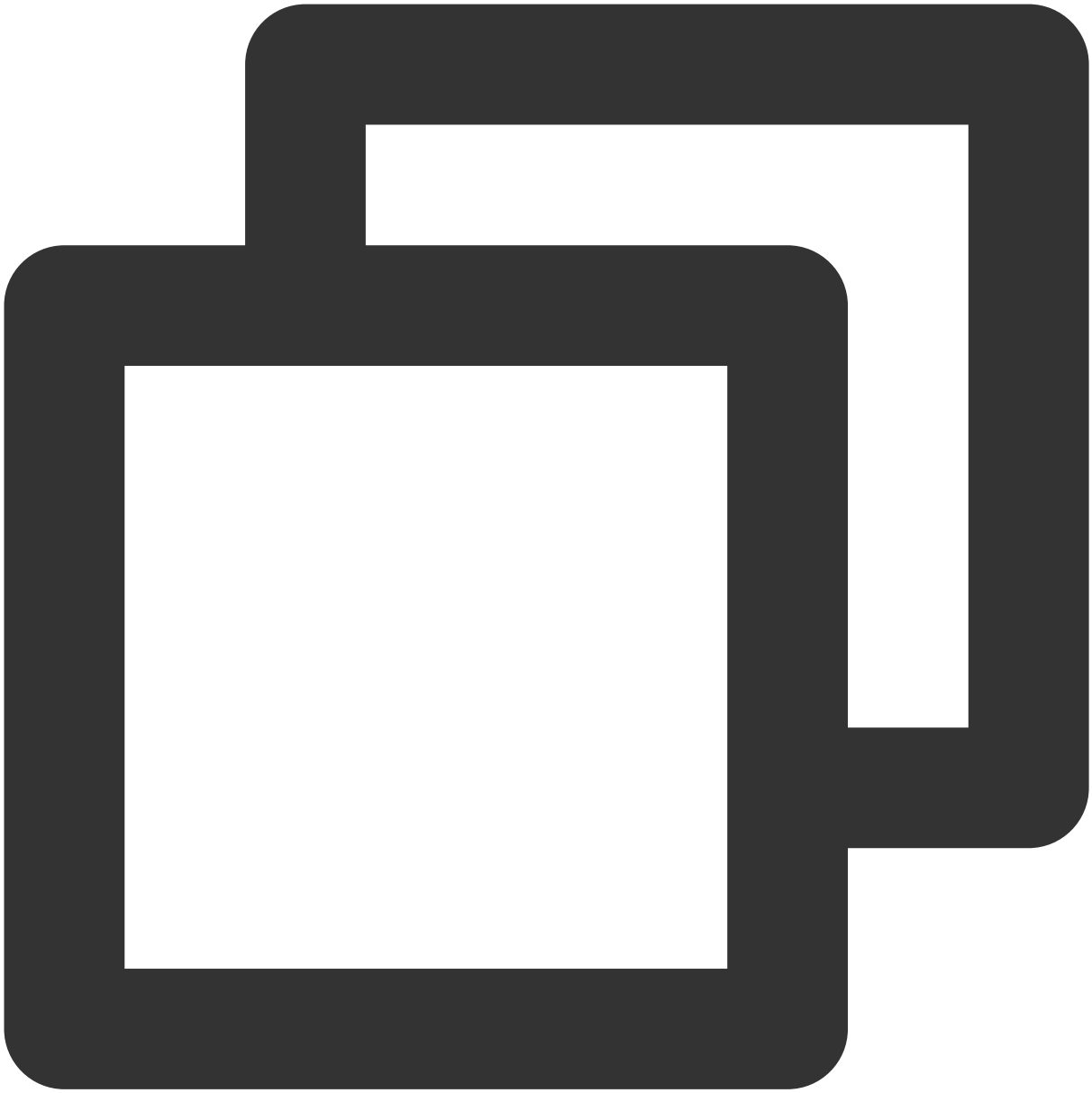
Message

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 = chat.createTextMessage({
  to: 'user1',
  conversationType: TencentCloudChat.TYPES.CONV_C2C,
  // 消息优先级，用于群聊。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
  // priority: TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL,
  payload: {
    text: 'Hello world!'
  },
  // 如果您发消息需要已读回执，需购买旗舰版套餐，并且创建消息时将 needReadReceipt 设置为 true
```

```
  needReadReceipt: true
  // 消息自定义数据（云端保存，会发送到对端，程序卸载重装后还能拉取到）
  // cloudCustomData: 'your cloud custom data'
});
// 2. 发送消息
let promise = chat.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);
});
```



```
// 发送群定向消息
// 注意：群定向消息不计入会话未读，receiverList 最大支持50个接收者。
let message = chat.createTextMessage({
  to: 'group1',
  conversationType: TencentCloudChat.TYPES.CONV_GROUP,
  payload: {
    text: 'Hello world!'
  },
  // 如果您需要发群定向消息，需购买旗舰版套餐，并且创建消息时通过 receiverList 指定消息接收者
  receiverList: ['user0', 'user1']
});
```

```
// 发送消息
let promise = chat.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);
});
```

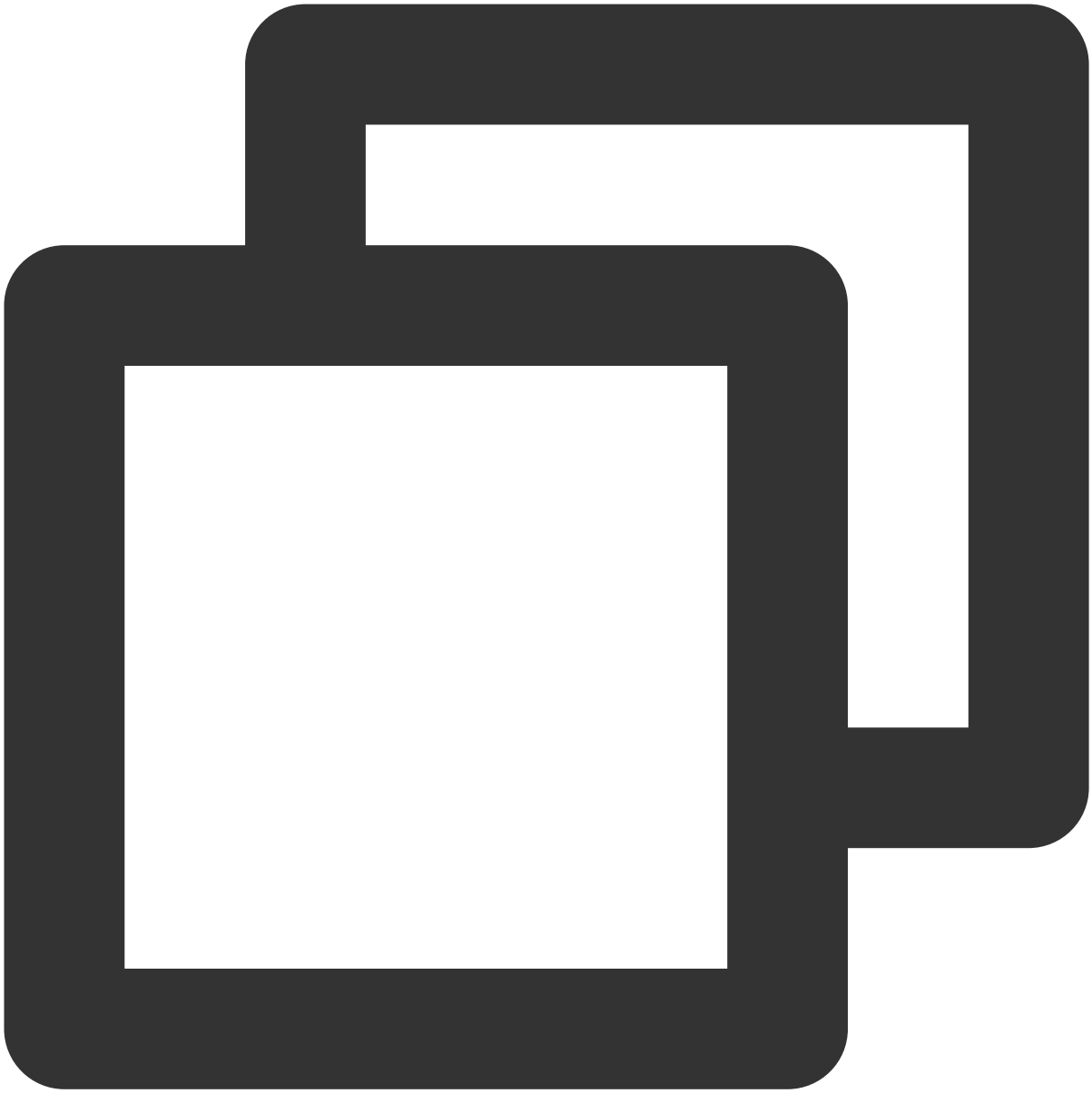
创建@消息

创建可以附带 @ 提醒功能的文本消息的接口，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。

注意

1. 此接口仅用于群聊，且社群和社群下的话题不支持 @ALL。
2. 创建群 @ 消息不支持指定消息接收者。

接口



```
chat.createTextAtMessage(options);
```

参数

参数 options 为 Object 类型，包含的属性值如下：

Name	Type	Default	Description
to	String	-	消息接收方的 userID
conversationType	String	-	会话类型，取

			值 `TencentCloudC` 端会话） 或 `TencentCloudC` 组会话）
priority	String	TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	消息内容的容器
cloudCustomData	String	''	消息自定义数据（云 重装后还能拉取到）

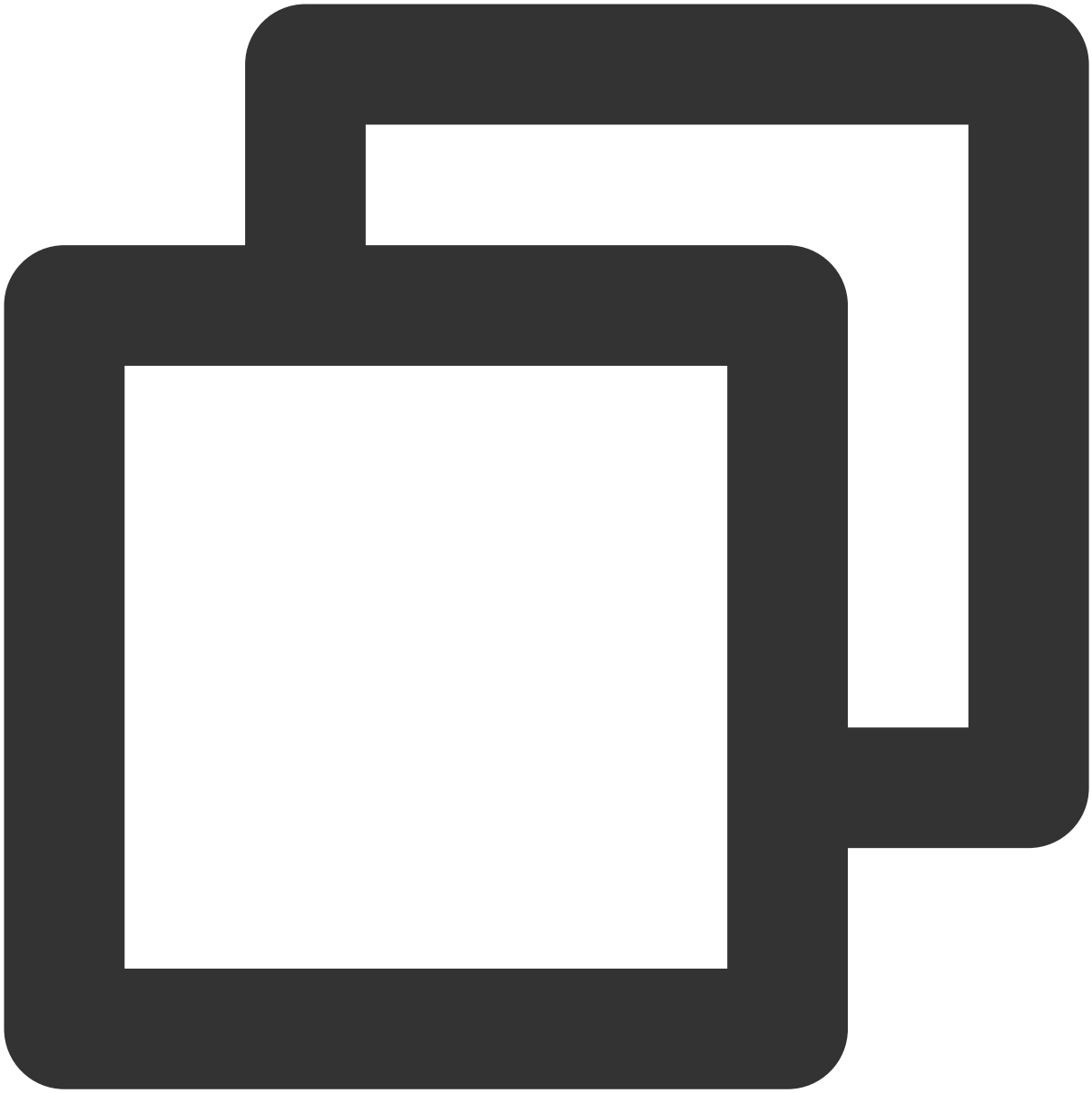
payload 的描述如下：

Name	Type	Description
text	String	消息文本内容
atUserList	Array	需要 @ 的用户列表，如果需要 @ALL，请传入 TencentCloudChat.TYPES.MSG_AT_ALL 。 举个例子，假设该条文本消息希望 @ 提醒 denny 和 lucy 两个用户，同时又希望 @ 所有人，atUserList 传 ['denny', 'lucy', TencentCloudChat.TYPES.MSG_AT_ALL]

返回值

Message

示例



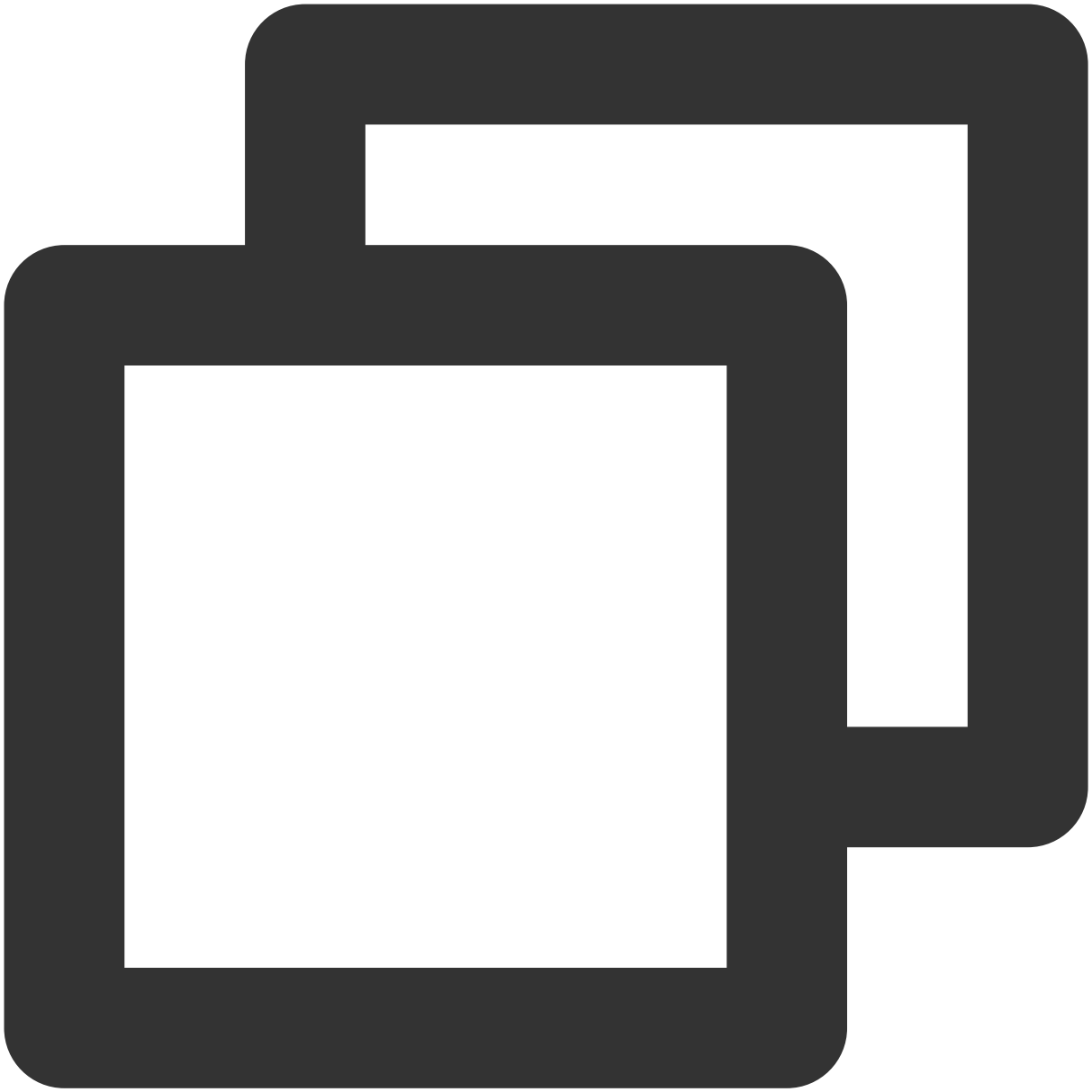
```
// 发送文本消息，Web 端与小程序端相同
// 1. 创建消息实例，接口返回的实例可以上屏
let message = chat.createTextAtMessage({
  to: 'group1',
  conversationType: TencentCloudChat.TYPES.CONV_GROUP,
  // 消息优先级，用于群聊
  // priority: TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL,
  payload: {
    text: '@denny @lucy @所有人 今晚聚餐，收到的请回复1',
    // 'denny' 'lucy' 都是 userID，而非昵称
    atUserList: ['denny', 'lucy', TencentCloudChat.TYPES.MSG_AT_ALL]
```

```
  },
  // 消息自定义数据（云端保存，会发送到对端，程序卸载重装后还能拉取到）
  // cloudCustomData: 'your cloud custom data'
});
// 2. 发送消息
let promise = chat.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);
});
```

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。

接口



```
chat.createImageMessage(options);
```

参数

参数 options 为 Object 类型，包含的属性值如下表所示：

Name	Type	Default	Description
to	String	-	消息的接收方
conversationType	String	-	会话类型，取值

			`TencentCloudC` 或 `TencentCloudC`
priority	String	TencentCloudChat.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
cloudCustomData	String	''	消息自定义数据（云 序卸载重装后还能拉

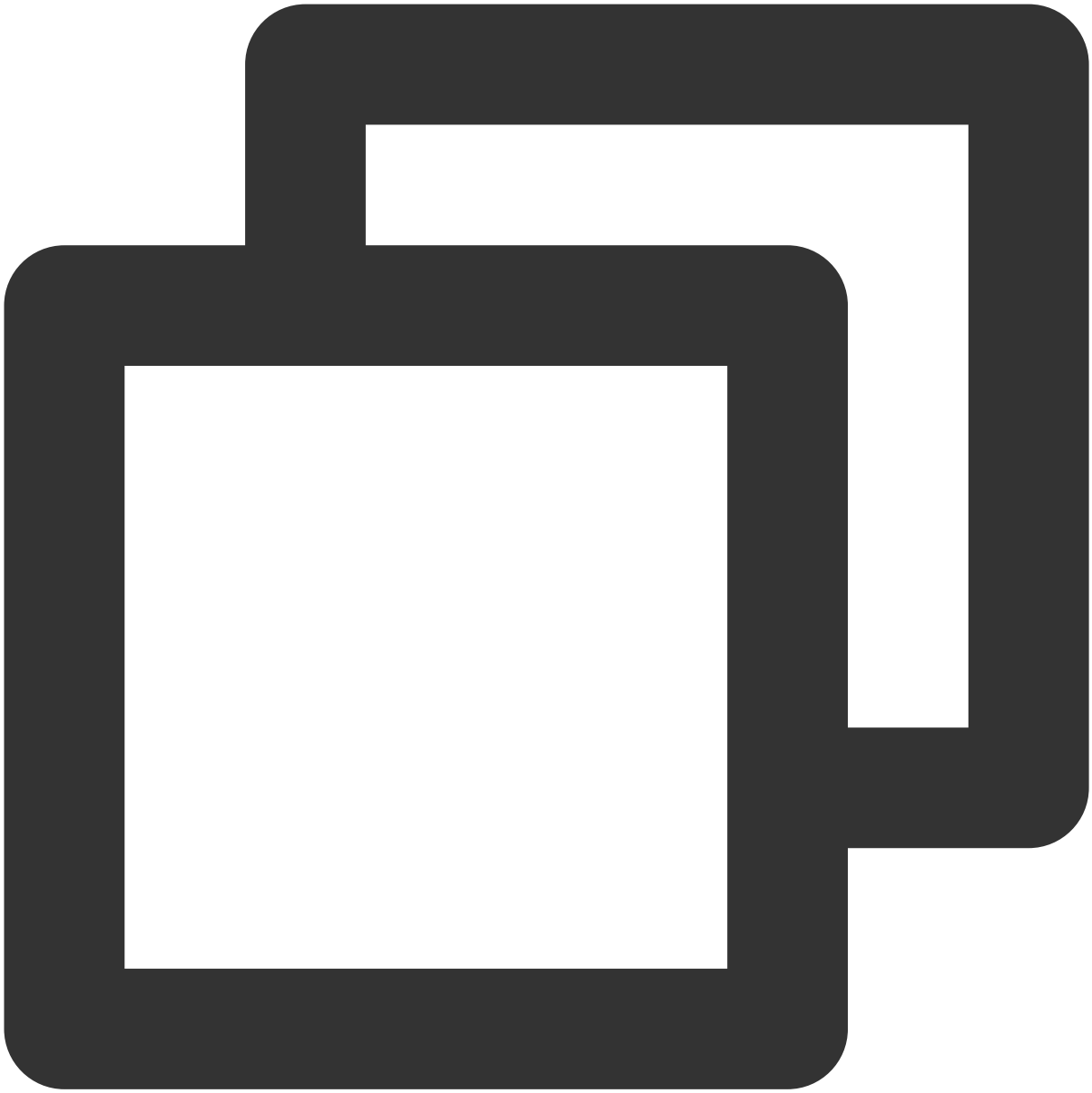
payload 的描述如下：

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

返回值

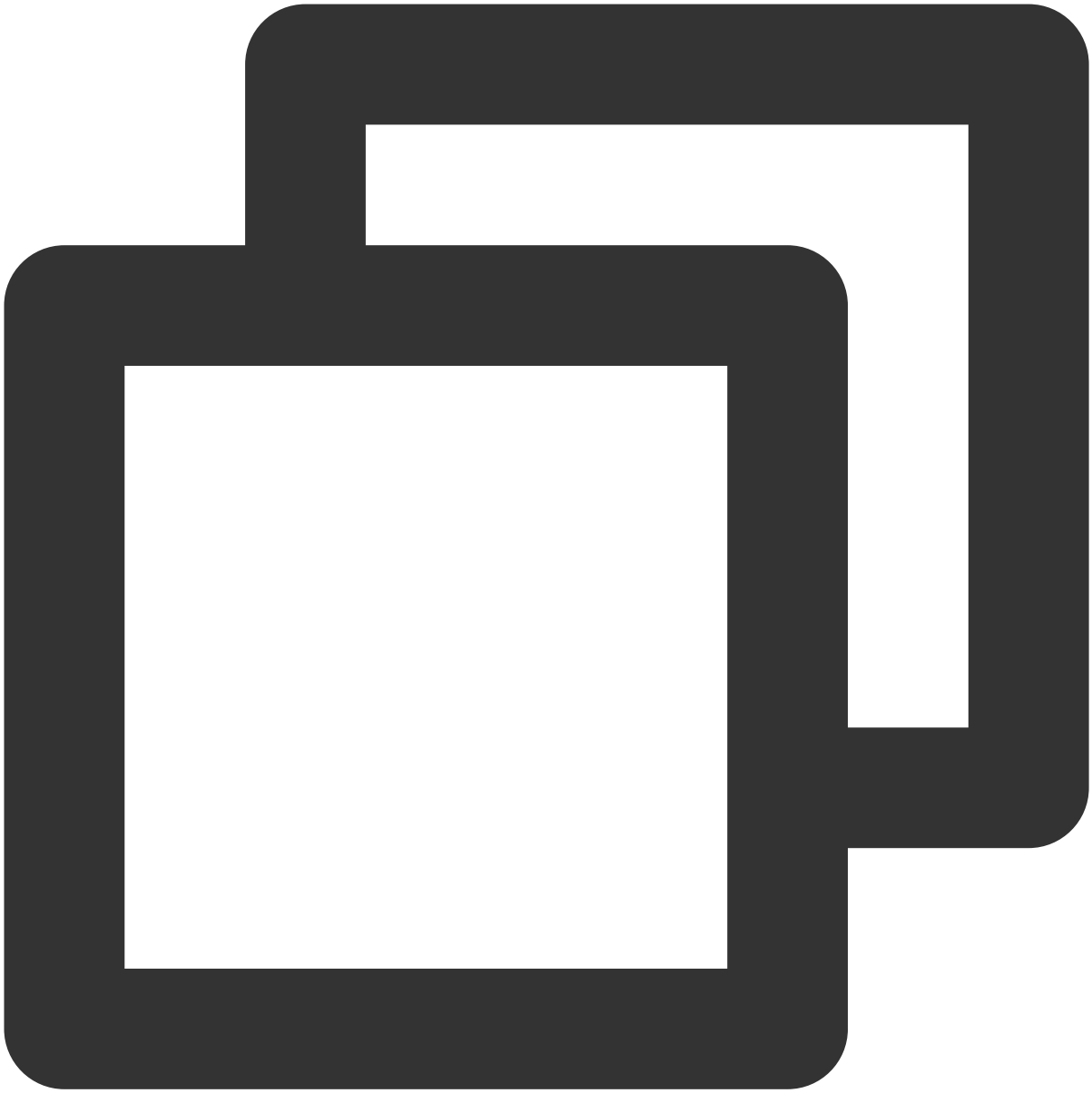
Message

示例



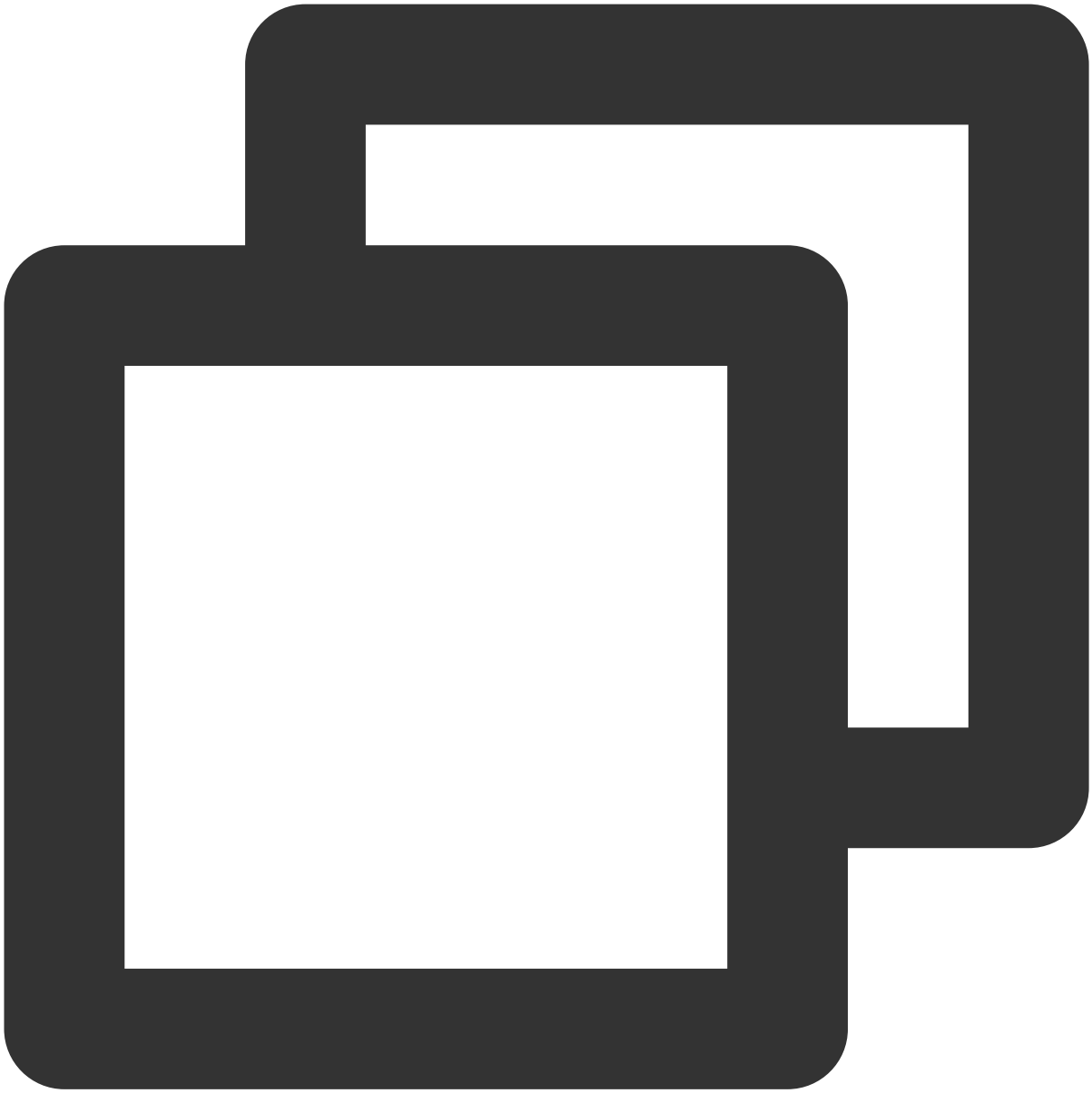
```
// Web 端发送图片消息示例1 - 传入 DOM 节点
// 1. 创建消息实例，接口返回的实例可以上屏
let message = chat.createImageMessage({
  to: 'user1',
  conversationType: TencentCloudChat.TYPES.CONV_C2C,
  // 消息优先级，用于群聊
  // priority: TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL,
  payload: {
    file: document.getElementById('imagePicker'),
  },
  // 消息自定义数据（云端保存，会发送到对端，程序卸载重装后还能拉取到）
```

```
  // cloudCustomData: 'your cloud custom data'
  onProgress: function(event) { console.log('file uploading:', event) }
});

// 2. 发送消息
let promise = chat.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);
});
```



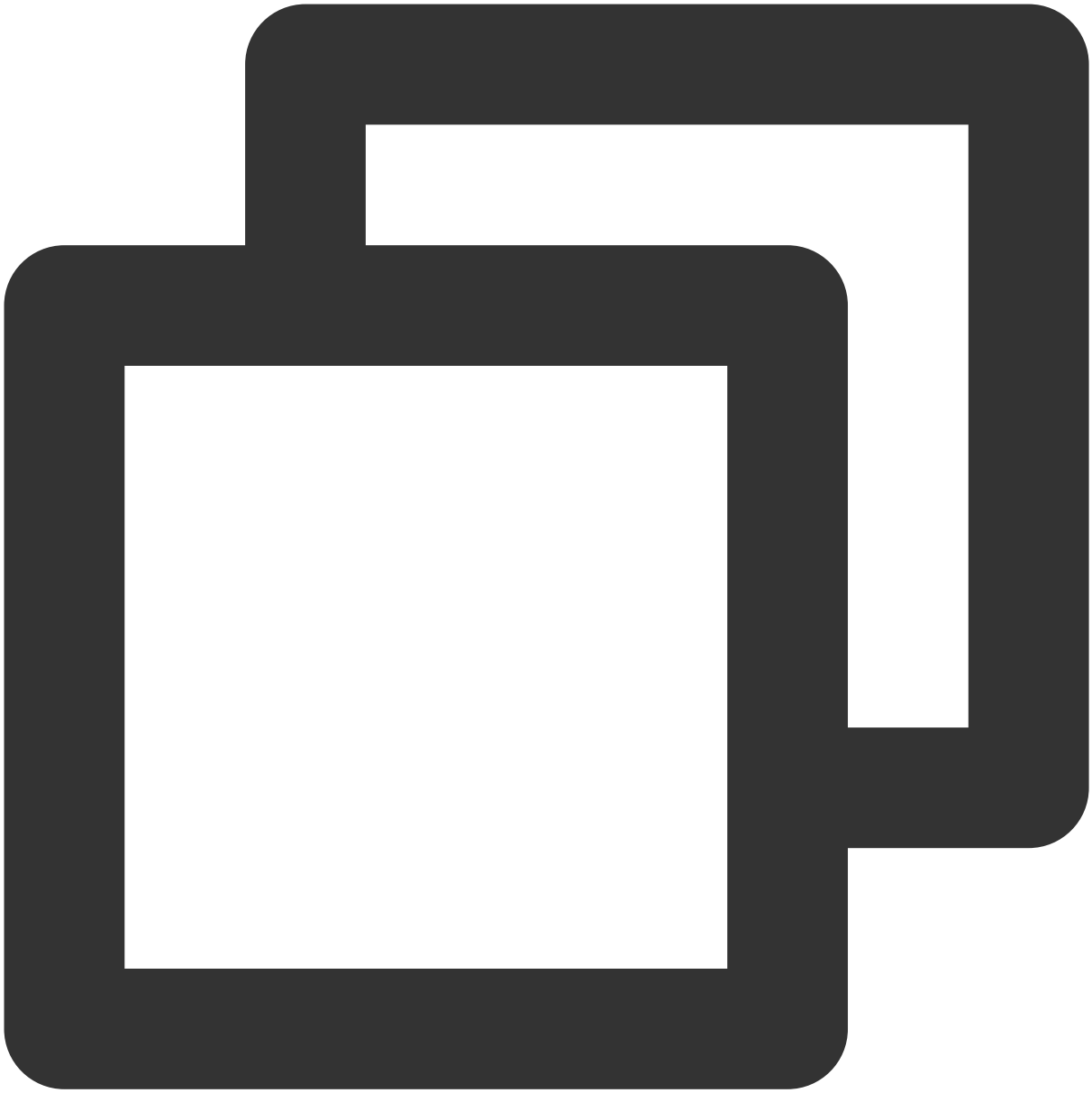
```
// Web 端发送图片消息示例2- 传入 File 对象
// 先在页面上添加一个 id 为 "testPasteInput" 的消息输入框
// 如 <input type="text" id="testPasteInput" placeholder="截图后粘贴到输入框中" size="3
document.getElementById('testPasteInput').addEventListener('paste', function(e) {
  let clipboardData = e.clipboardData;
  let file;
  let fileCopy;
  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
    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 = chat.createImageMessage({
    to: 'user1',
    conversationType: TencentCloudChat.TYPES.CONV_C2C,
    payload: {
      file: file
    },
    onProgress: function(event) { console.log('file uploading:', event) }
  });

  // 2. 发送消息
  let promise = chat.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 = chat.createImageMessage({
      to: 'user1',
      conversationType: TencentCloudChat.TYPES.CONV_C2C,
      payload: { file: res },
```

```
        onProgress: function(event) { console.log('file uploading:', event) }
      });
      // 3. 发送图片
      let promise = chat.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);
      });
    }
})
```



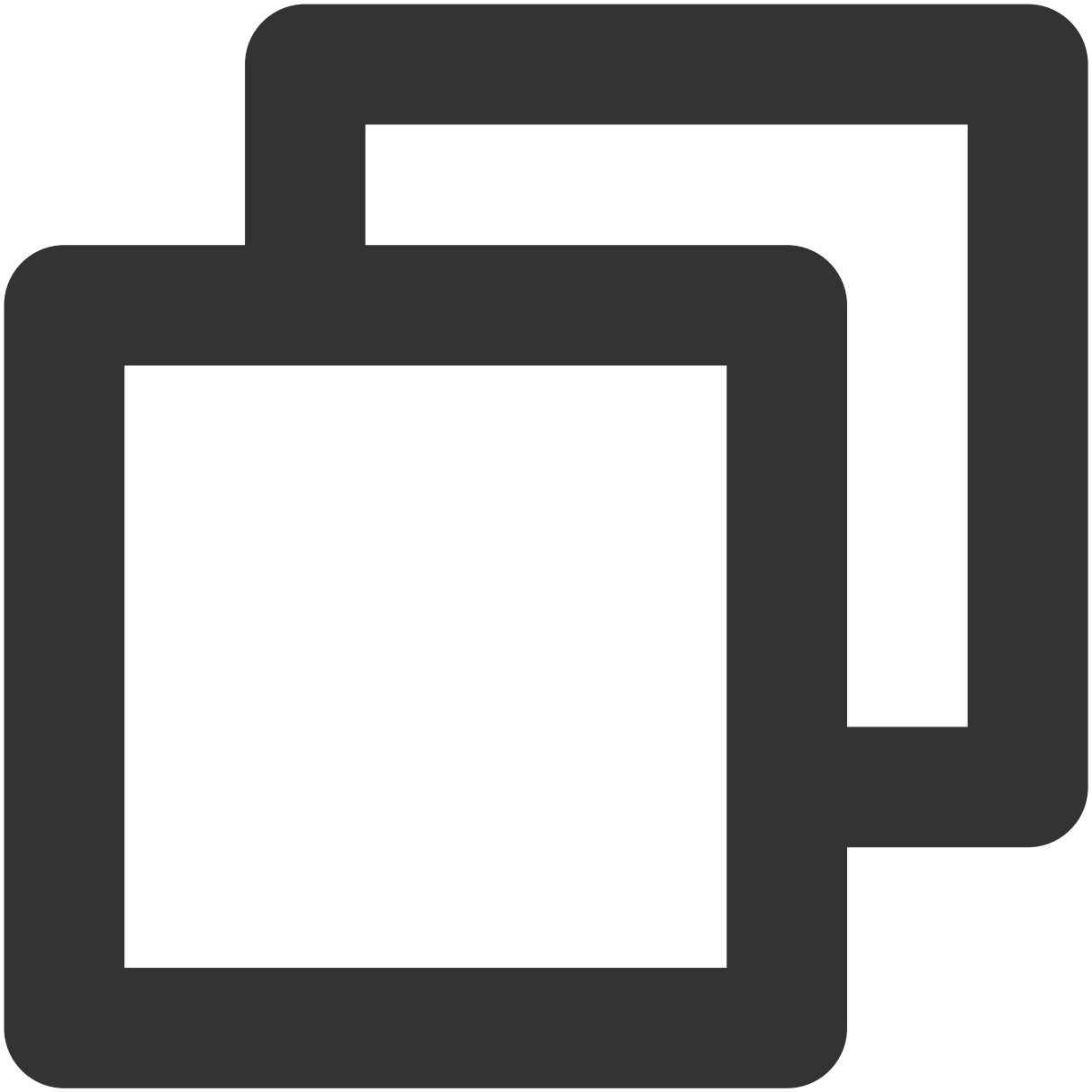
```
// uni-app 发送图片
// 从基础库 2.21.0 开始， wx.chooseImage 停止维护，请使用 uni.chooseMedia 代替
// 1. 选择图片
uni.chooseMedia({
  count: 1,
  mediaType: ['image'], // 图片
  sizeType: ['original', 'compressed'], // 可以指定是原图还是压缩图，默认二者都有
  sourceType: ['album'], // 从相册选择
  success: function(res) {
    let message = chat.createImageMessage({
      to: 'user1',
```

```
      conversationType: TencentCloudChat.TYPES.CONV_C2C,
      payload: { file: res },
      onProgress: function(event) { console.log('file uploading:', event) }
    });
    // 2. 发送消息
    let promise = chat.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);
    });
  }
});
```

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。

接口



```
chat.createAudioMessage(options);
```

参数

参数 options 为 Object 类型，包含的属性值如下：

Name	Type	Default	Description
to	String	-	消息的接收方
conversationType	String	-	会话类型，取值

			`TencentCloudCh` 或 `TencentCloudCh`
priority	String	TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	消息内容的容器
cloudCustomData	String	''	消息自定义数据（云 序卸载重装后还能拉

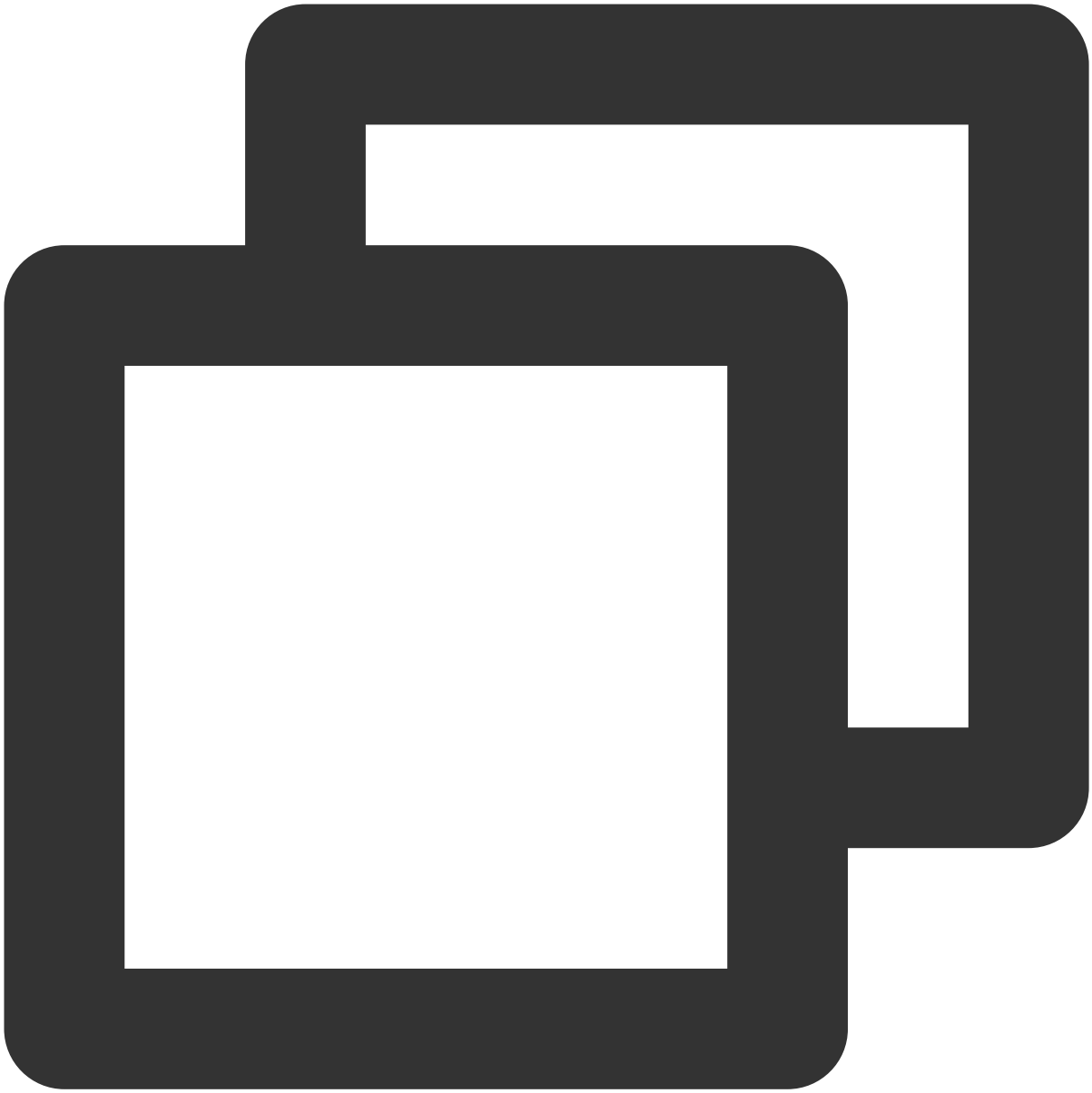
payload 的描述如下：

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

返回值

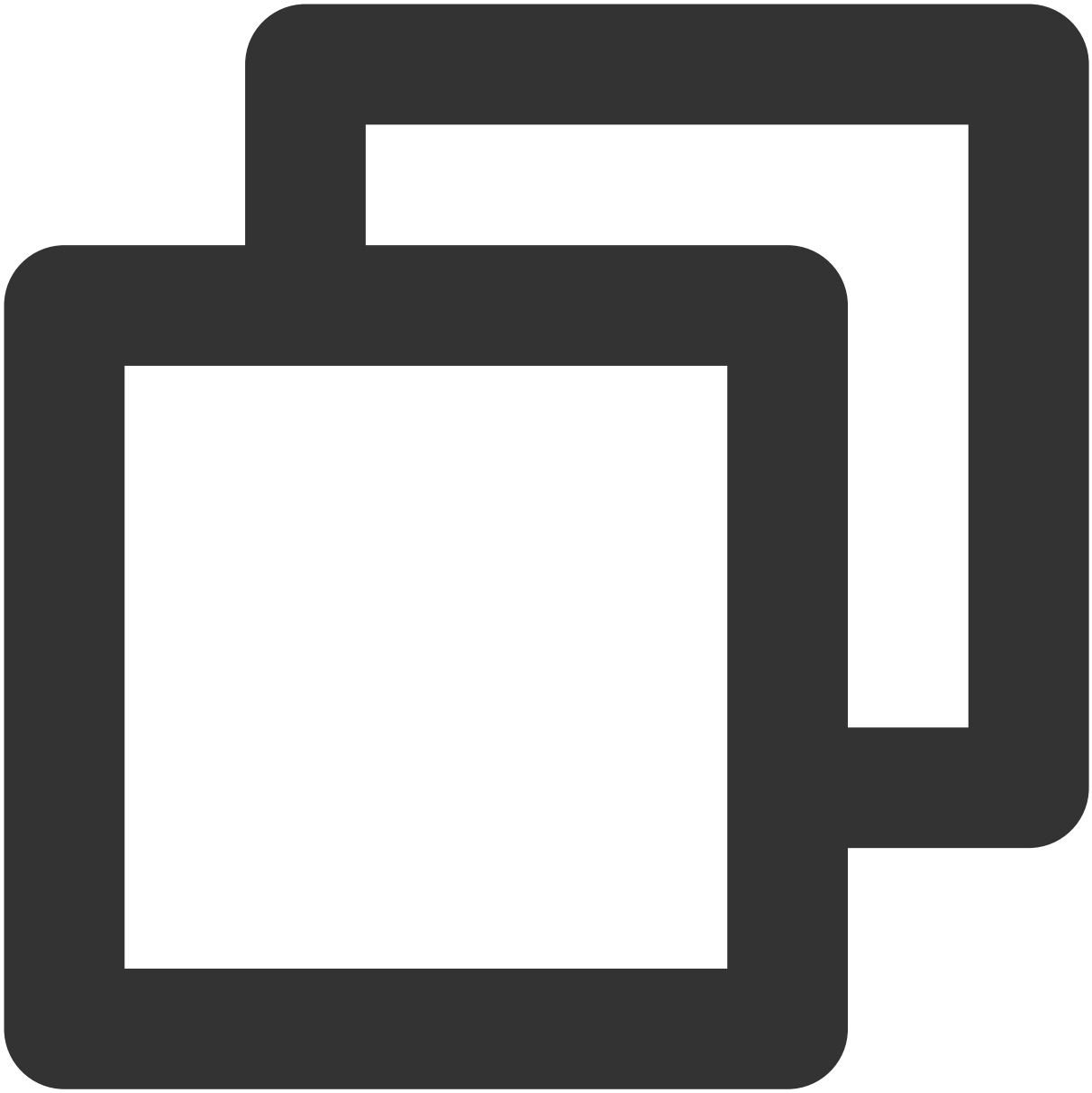
Message

示例



```
// 示例：使用官方的 RecorderManager 进行录音
// 参考 https://developers.weixin.qq.com/minigame/dev/api/media/recorder/RecorderMar
// 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 Chat 全平台（Android、iOS和Web
};

// 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 = chat.createAudioMessage({
    to: 'user1',
    conversationType: TencentCloudChat.TYPES.CONV_C2C,
    // 消息优先级，用于群聊
    // priority: TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL,
    payload: {
      file: res
    },
    // 消息自定义数据（云端保存，会发送到对端，程序卸载重装后还能拉取到）
    // cloudCustomData: 'your cloud custom data'
  });

  // 5. 发送消息
  let promise = chat.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);
```



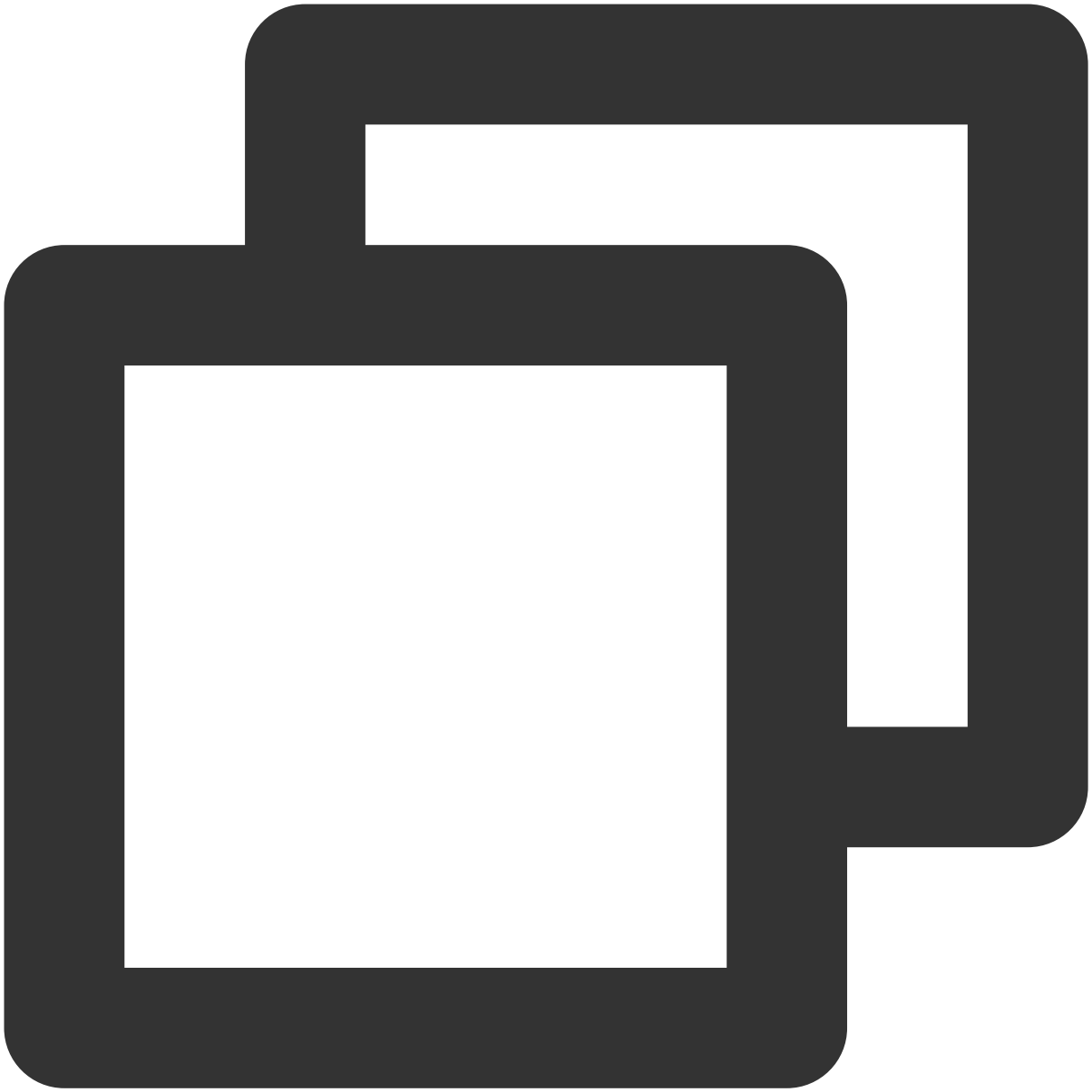
```
// 在 Web 端创建语音消息并发送
// 示例：使用第三方库 js-audio-recorder 录制音频
// 1. 开始录制
let recorder = new Recorder({
  // 采样位数，支持 8 或 16，默认是16
  sampleBits: 16,
  // 采样率，支持 11025、16000、22050、24000、44100、48000，根据浏览器默认值，我的chrome是4
  sampleRate: 16000,
  // 声道，支持 1 或 2，  默认是1
  numChannels: 1,
});
```

```
let startTs;
recorder.start().then(() => {
  // 开始录音，记录起始时间戳
  startTs = Date.now();
}, (error) => {
  // 出错了
  console.log(`${error.name} : ${error.message}`);
});

// 2. 结束录制
recorder.stop();

// 3. 计算录音时长，获取 WAV 数据
let duration = Date.now() - startTs; // 单位：ms
let wavBlob = recorder.getWAVBlob();

// 4. blob 数据转成 File 对象
let audioFile = new File([wavBlob], 'hello.wav', { type: 'wav' });
audioFile.duration = duration;

// 5. 创建音频消息
let message = chat.createAudioMessage({
  to: 'user1',
  conversationType: 'C2C',
  payload: {
    file: audioFile
  }
});

// 6. 发送消息
let promise = chat.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);
});
```

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。

接口



```
chat.createVideoMessage(options);
```

参数

参数 options 为 Object 类型，包含的属性值如下：

Name	Type	Default	Description
to	String	-	消息的接收方
conversationType	String	-	会话类型，取值 `TencentCloudC`

			或 `TencentCloudCh`
priority	String	TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	消息内容的容器
cloudCustomData	String	''	消息自定义数据（云 卸载重装后还能拉取

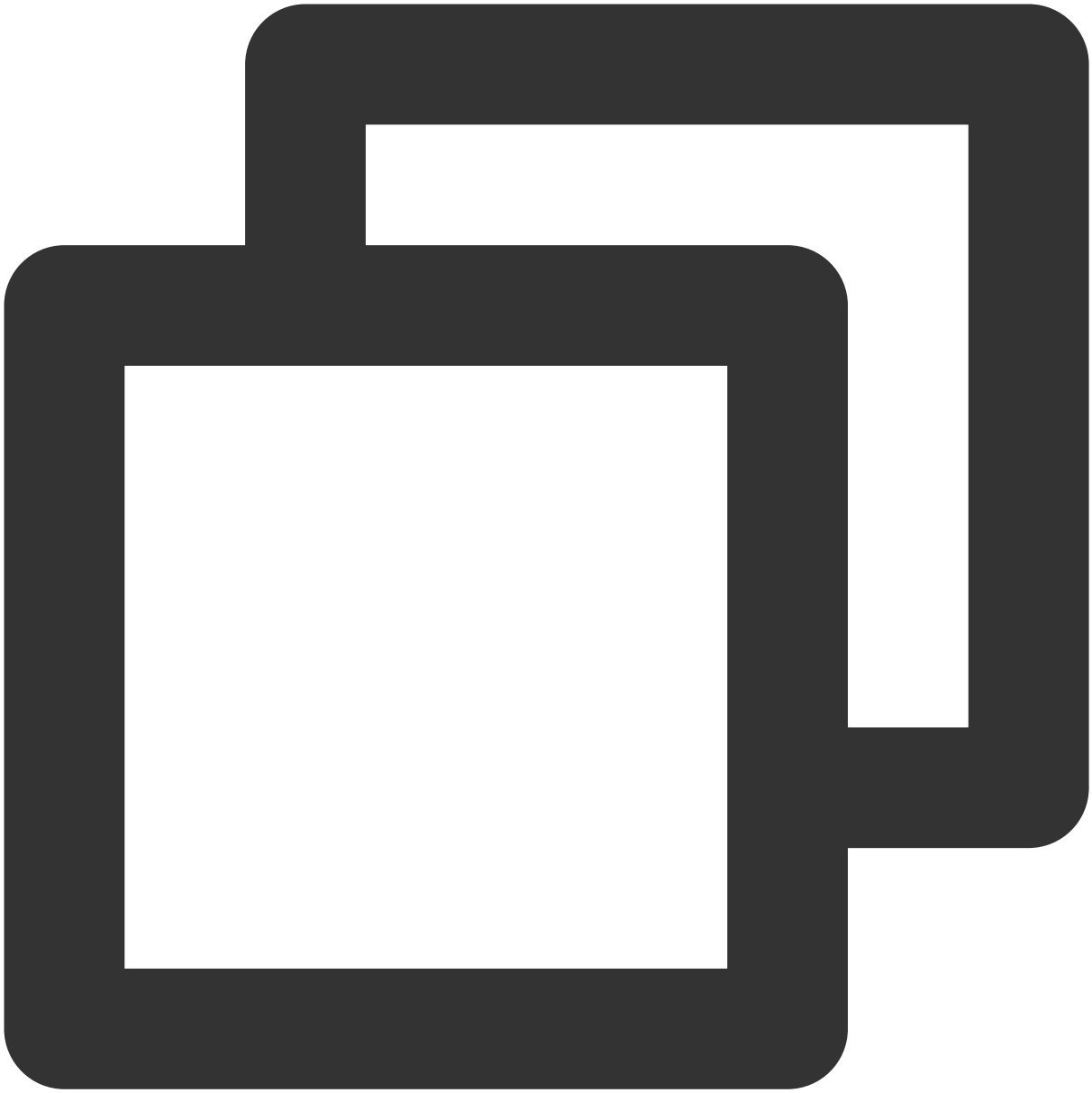
payload 的描述如下：

Name	Type	Description
file	HTMLInputElement | File | Object	自定义消息的数据字段

返回值

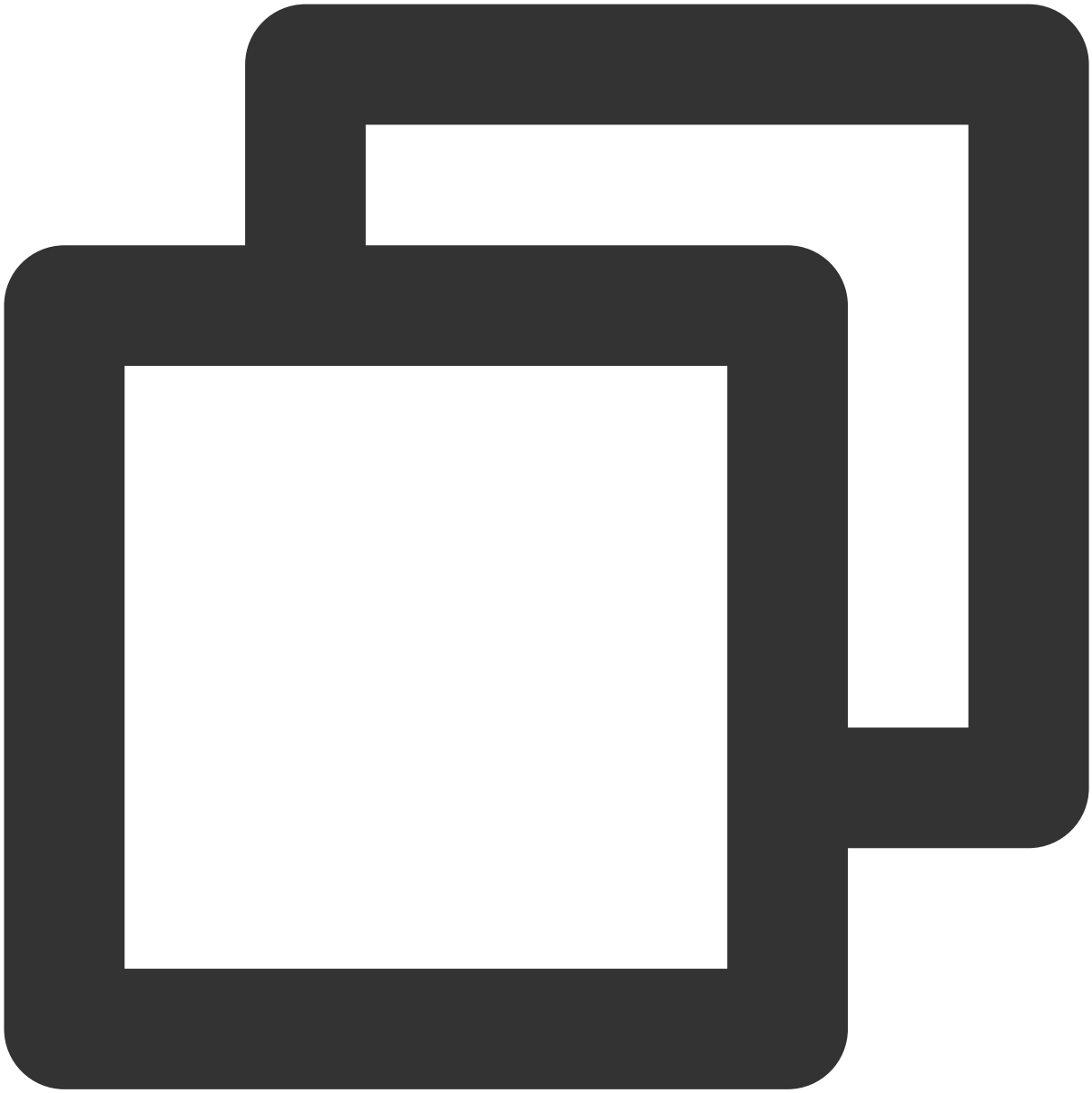
Message

示例



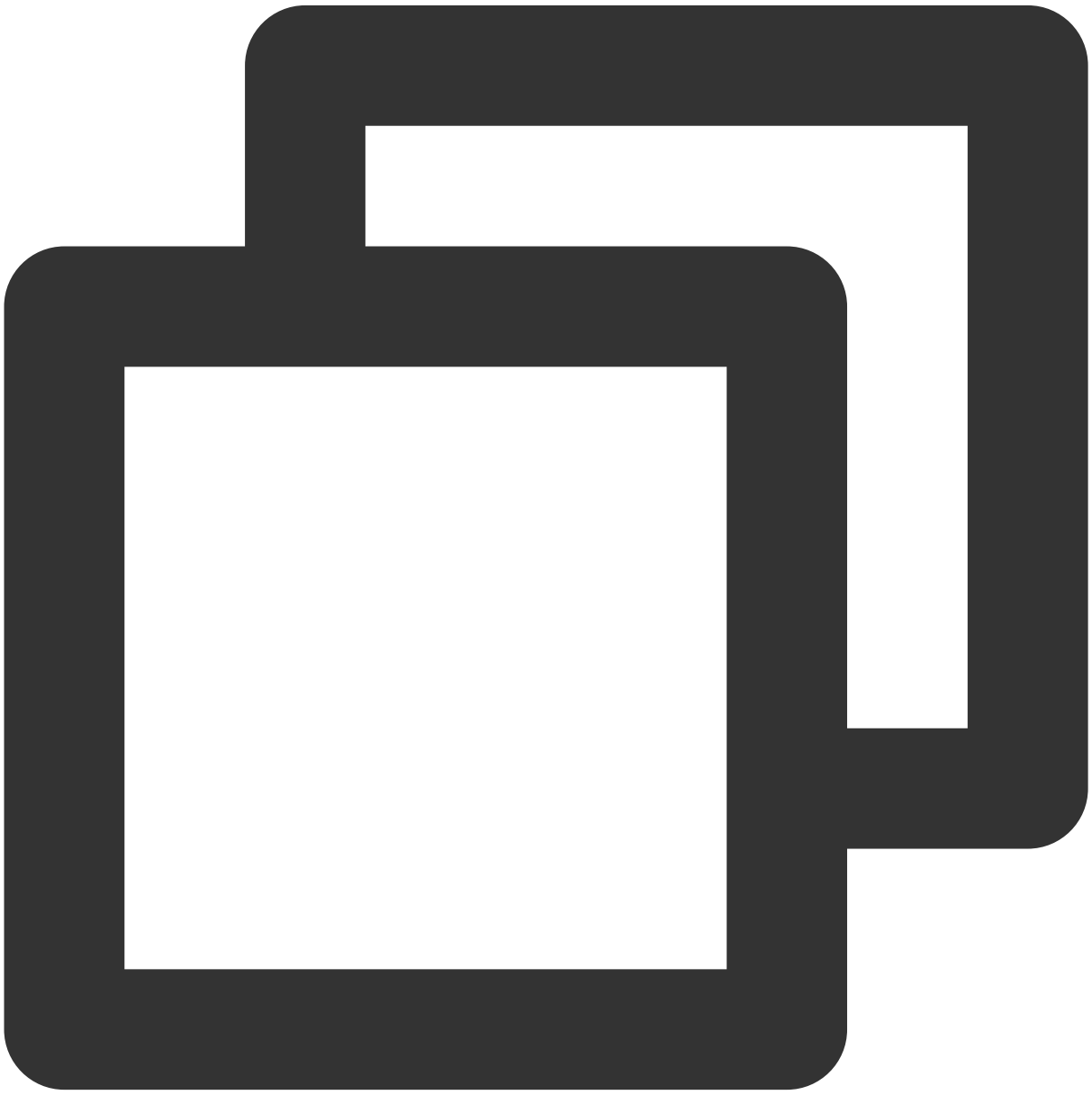
```
// 小程序端发送视频消息示例：
// 接口详情请查阅 https://developers.weixin.qq.com/miniprogram/dev/api/media/video/wx
// 1. 调用小程序接口选择视频
wx.chooseVideo({
  sourceType: ['album', 'camera'], // 来源相册或者拍摄
  maxDuration: 60, // 设置最长时间60s
  camera: 'back', // 后置摄像头
  success (res) {
    // 2. 创建消息实例，接口返回的实例可以上屏
    let message = chat.createVideoMessage({
      to: 'user1',
```

```
      conversationType: TencentCloudChat.TYPES.CONV_C2C,
      // 消息优先级，用于群聊
      // priority: TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL,
      payload: {
        file: res
      },
      // 消息自定义数据（云端保存，会发送到对端，程序卸载重装后还能拉取到）
      // cloudCustomData: 'your cloud custom data'
      onProgress: function(event) { console.log('file uploading:', event) }
    })
    // 3. 发送消息
    let promise = chat.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
// 1. 获取视频：传入 DOM 节点
// 2. 创建消息实例
const message = chat.createVideoMessage({
  to: 'user1',
  conversationType: TencentCloudChat.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 = chat.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);
});
```



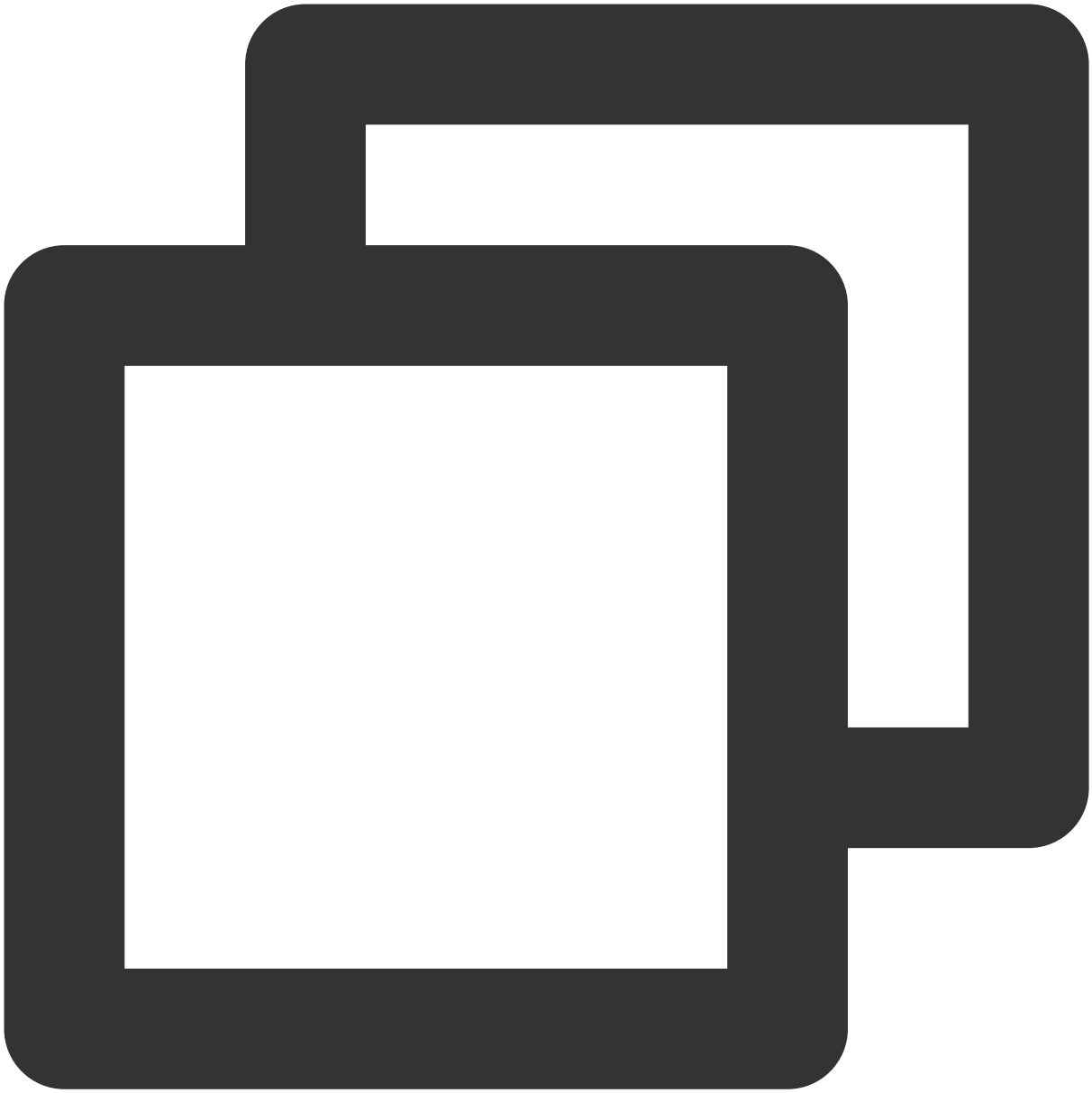
```
// uni-app 发送视频
```

```
// 1. 选择视频
uni.chooseVideo({
  count: 1,
  sourceType: ['camera', 'album'], // album 从相册选视频，camera 使用相机拍摄，默认为：['
  maxDuration: 60, // 设置最长时间60s
  camera: 'back', // 后置摄像头
  success: function(res) {
    let message = chat.createVideoMessage({
      to: 'user1',
      conversationType: TencentCloudChat.TYPES.CONV_C2C,
      payload: { file: res },
      onProgress: function(event) { console.log('file uploading:', event) }
    });
    // 2. 发送消息
    let promise = chat.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);
    });
  }
})
```

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。

接口



```
chat.createCustomMessage(options);
```

参数

参数 options 为 Object 类型，包含的属性值如下：

Name	Type	Default	Description
to	String	-	消息接收方的 userID
conversationType	String	-	会话类型，取

			值 `TencentCloudC` 或 `TencentCloudCh`
priority	String	TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	消息内容的容器
cloudCustomData	String	''	消息自定义数据（云 卸载重装后还能拉取

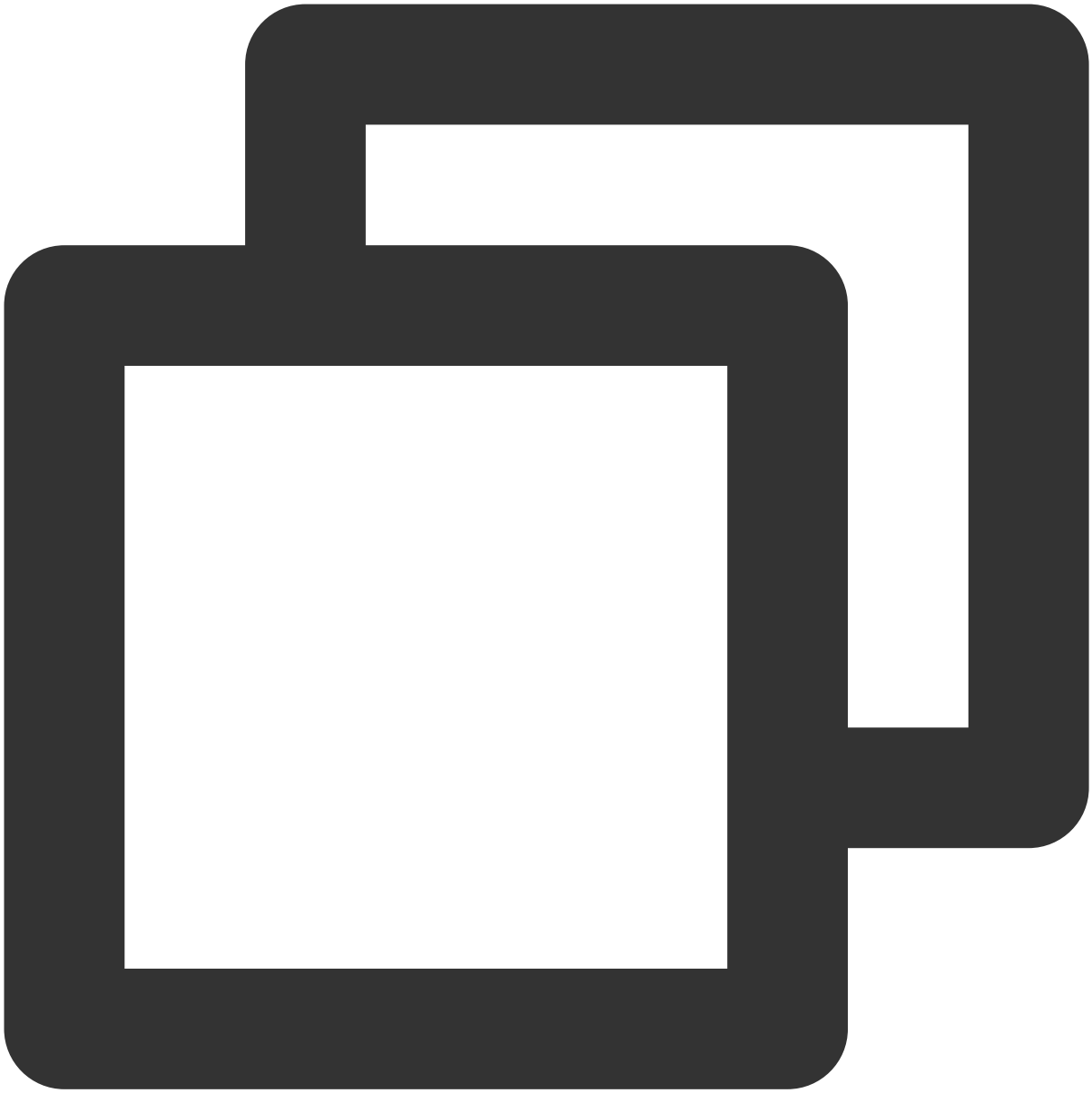
payload 的描述如下：

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

返回值

Message

示例



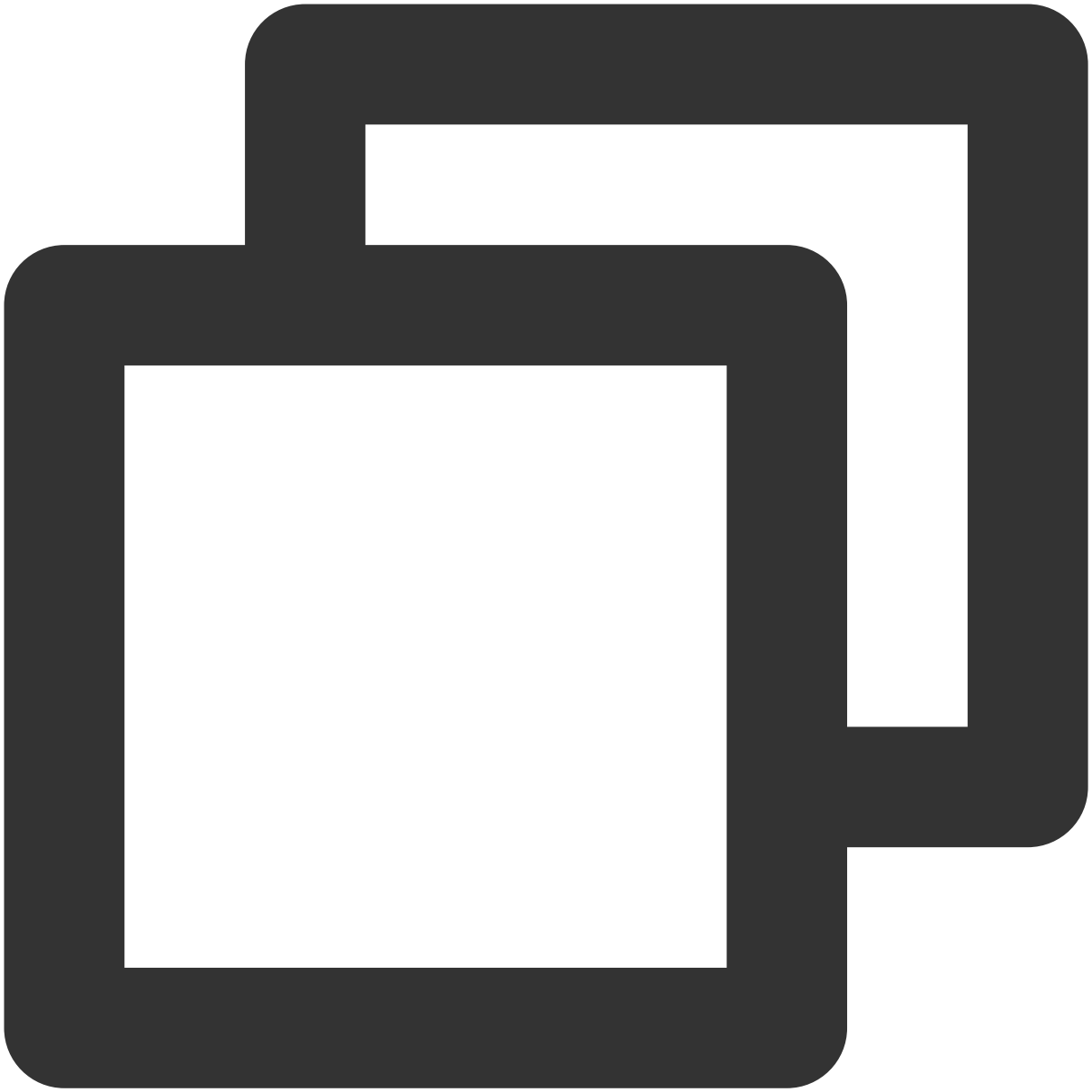
```
// 示例：利用自定义消息实现投骰子功能
// 1. 定义随机函数
function random(min, max) {
  return Math.floor(Math.random() * (max - min + 1) + min);
}
// 2. 创建消息实例，接口返回的实例可以上屏
let message = chat.createCustomMessage({
  to: 'user1',
  conversationType: TencentCloudChat.TYPES.CONV_C2C,
  // 消息优先级，用于群聊
  // priority: TencentCloudChat.TYPES.MSG_PRIORITY_HIGH,
```

```
  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 = chat.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);
});
```

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。

接口



```
chat.createFaceMessage(options);
```

参数

参数 options 为 Object 类型，包含的属性值如下：

Name	Type	Default	Description
to	String	-	消息接收方的 userID
conversationType	String	-	会话类型，取值

			`TencentCloudCh` 或 `TencentCloudCh`
priority	String	TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	消息内容的容器
cloudCustomData	String	''	消息自定义数据（云 序卸载重装后还能拉

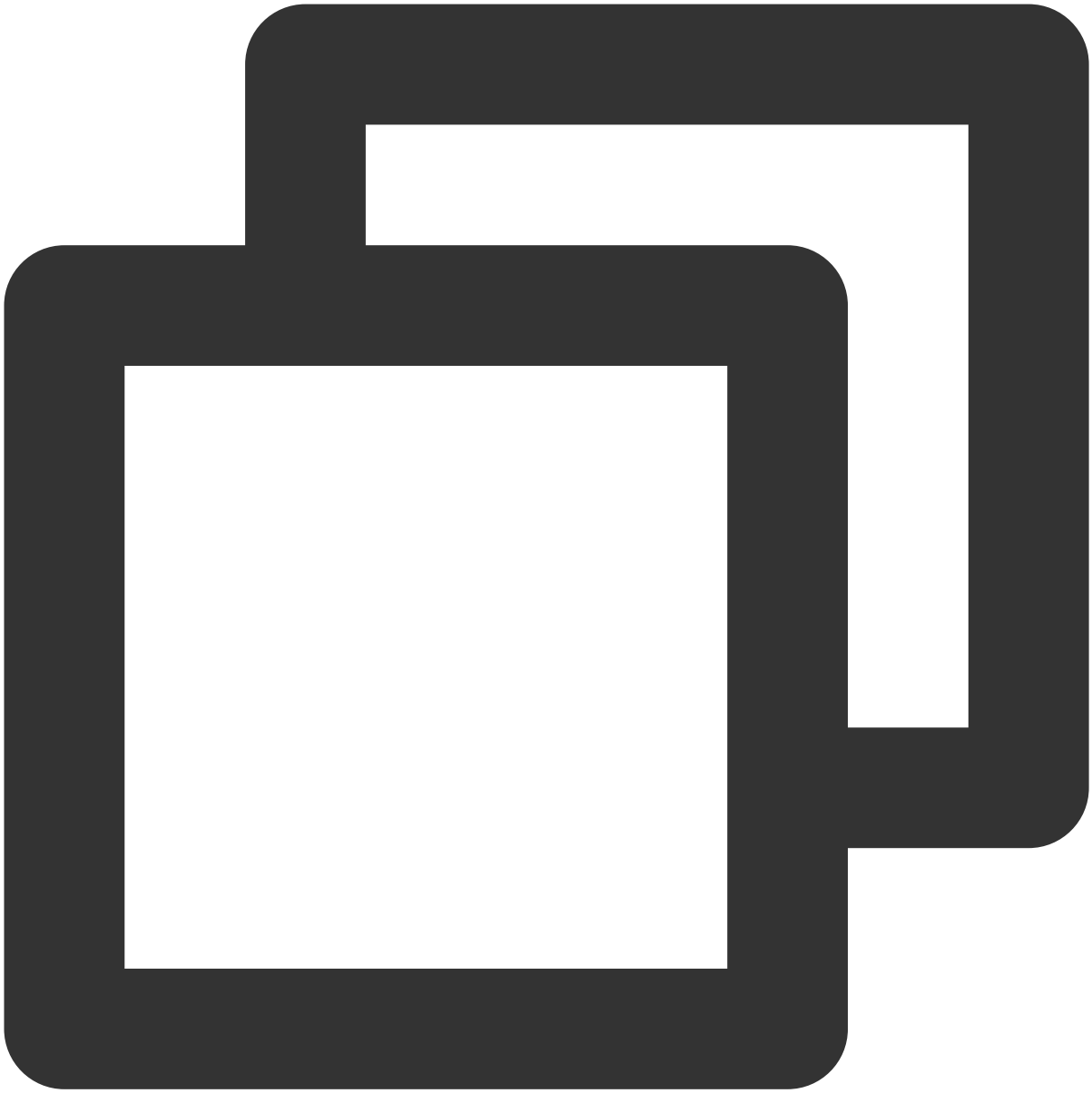
payload 的描述如下：

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

返回值

Message

示例



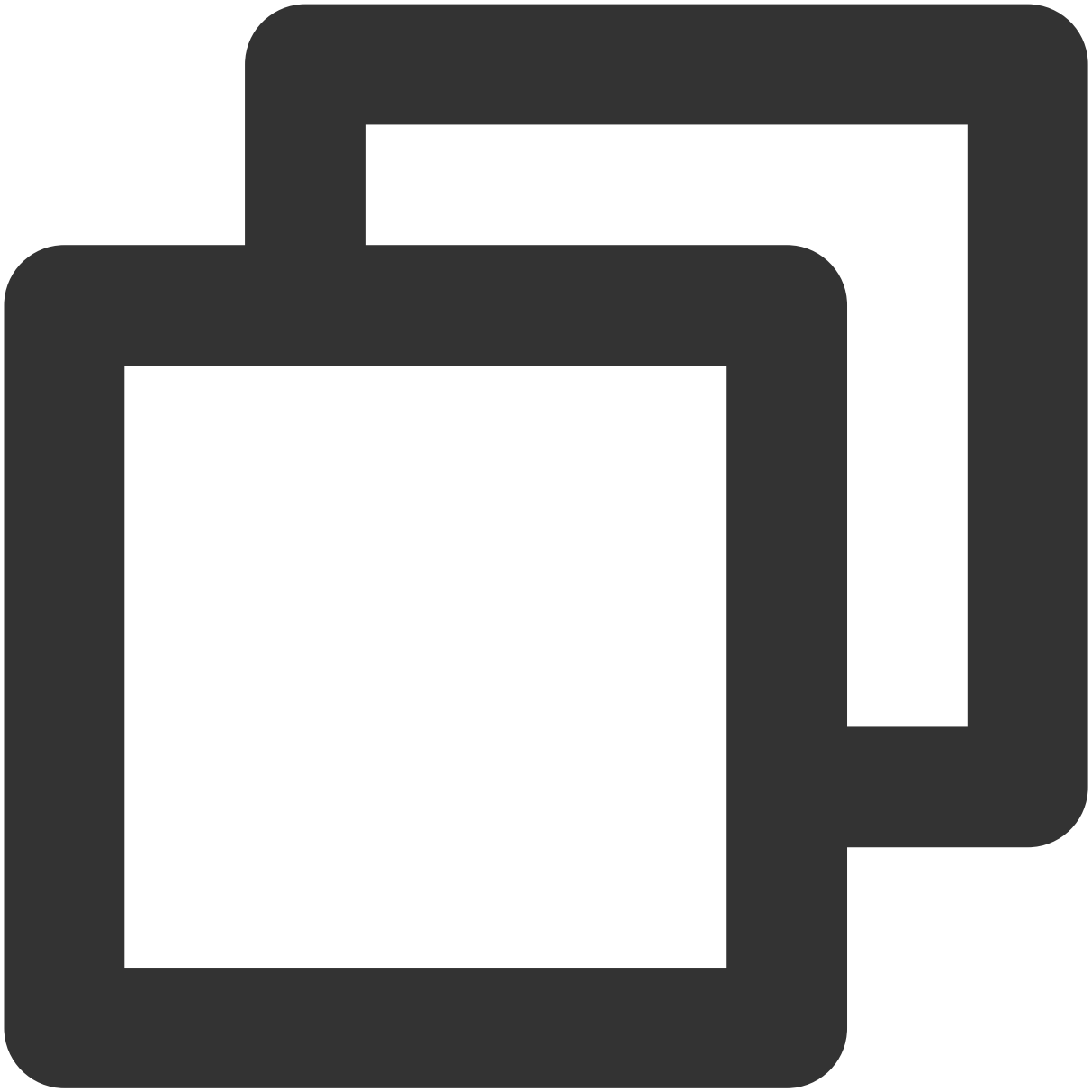
```
// 发送表情消息，Web端与小程序端相同。
// 1. 创建消息实例，接口返回的实例可以上屏
let message = chat.createFaceMessage({
  to: 'user1',
  conversationType: TencentCloudChat.TYPES.CONV_C2C,
  // 消息优先级，用于群聊
  // priority: TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL,
  payload: {
    index: 1, // Number 表情索引，用户自定义
    data: 'tt00' // String 额外数据
  },
```

```
  // 消息自定义数据（云端保存，会发送到对端，程序卸载重装后还能拉取到）
  // cloudCustomData: 'your cloud custom data'
});
// 2. 发送消息
let promise = chat.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);
});
```

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。

接口



```
chat.createFileMessage(options);
```

参数

参数 options 为 Object 类型，包含的属性值如下：

Name	Type	Default	Description
to	String	-	消息接收方的 userI
conversationType	String	-	会话类型，取值

			`TencentCloudC` 或 `TencentCloudC`
priority	String	TencentCloudChat.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
cloudCustomData	String	''	消息自定义数据（云 序卸载重装后还能拉

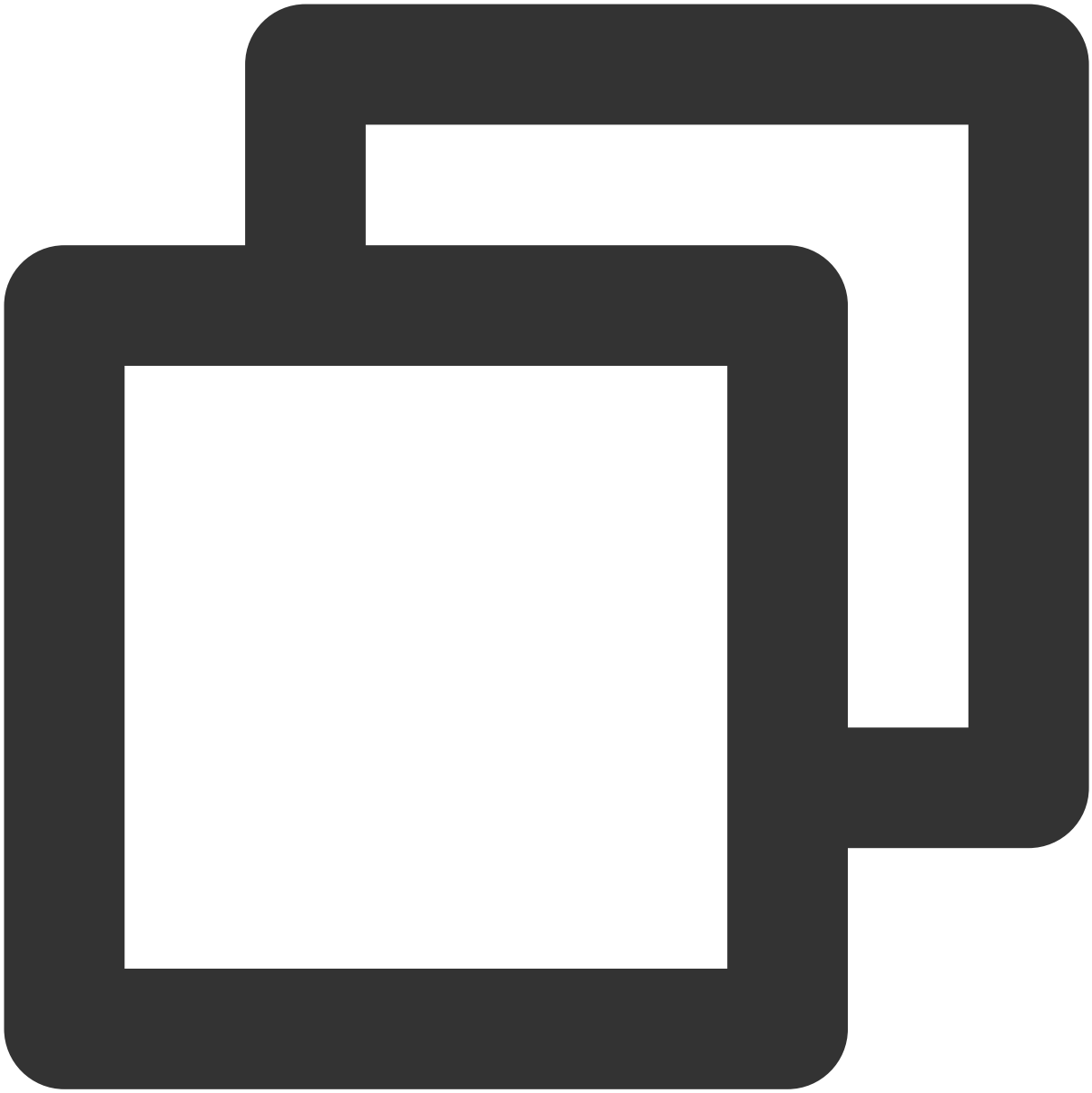
payload 的描述如下：

Name	Type	Description
file	HTMLInputElement | File | Object	用于选择文件的 DOM 节点（Web）或者 File 对象（Web）或者 Object（uni.chooseFile 接口的 success 回调参数），SDK 会读取其中的数据并上传文件

返回值

Message

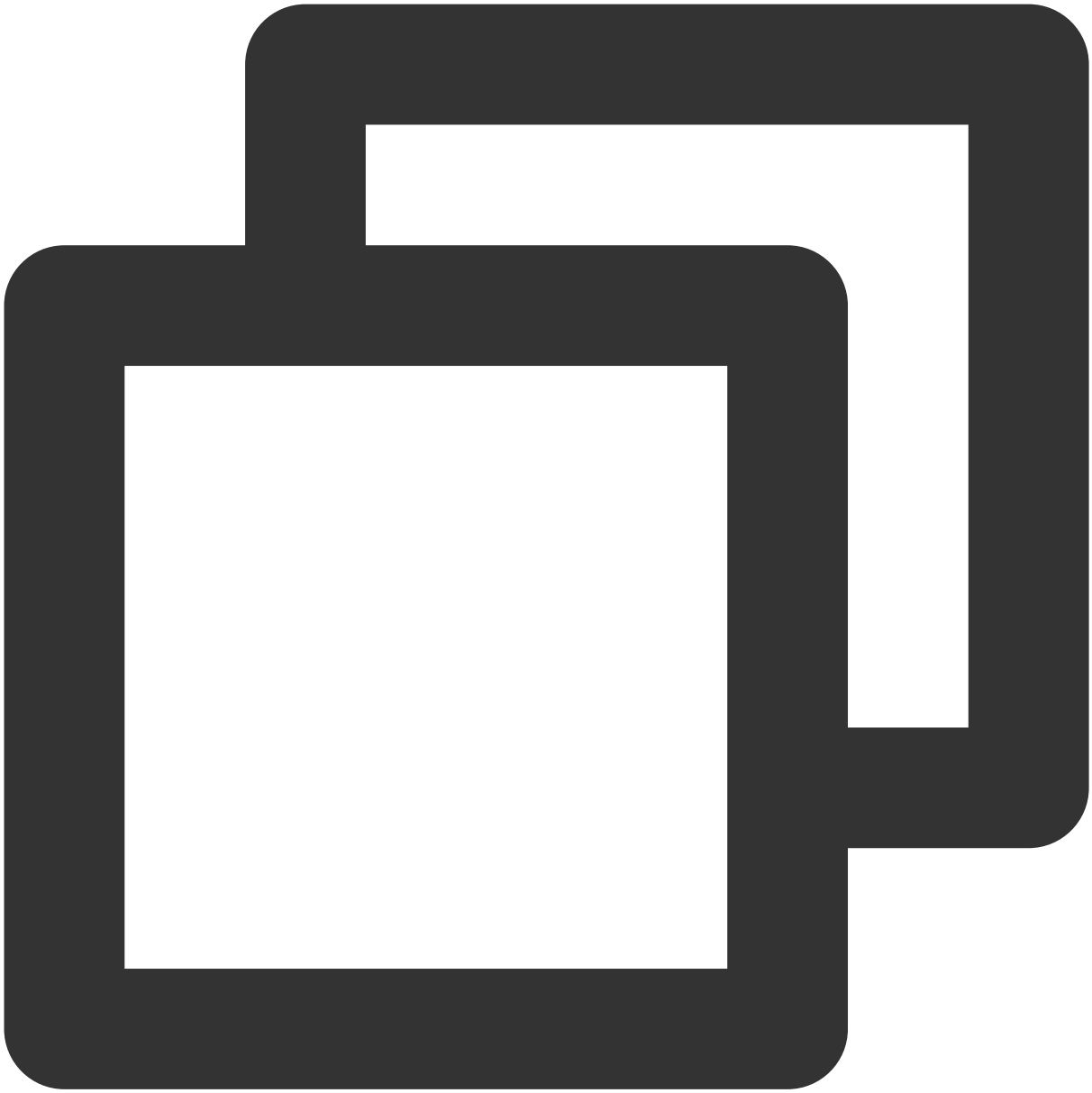
示例



```
// 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 = chat.createFileMessage({
  to: 'user1',
  conversationType: TencentCloudChat.TYPES.CONV_C2C,
  // 消息优先级，用于群聊
  // priority: TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL,
  payload: {
    file: document.getElementById('filePicker'),
  },
  // 消息自定义数据（云端保存，会发送到对端，程序卸载重装后还能拉取到）
```

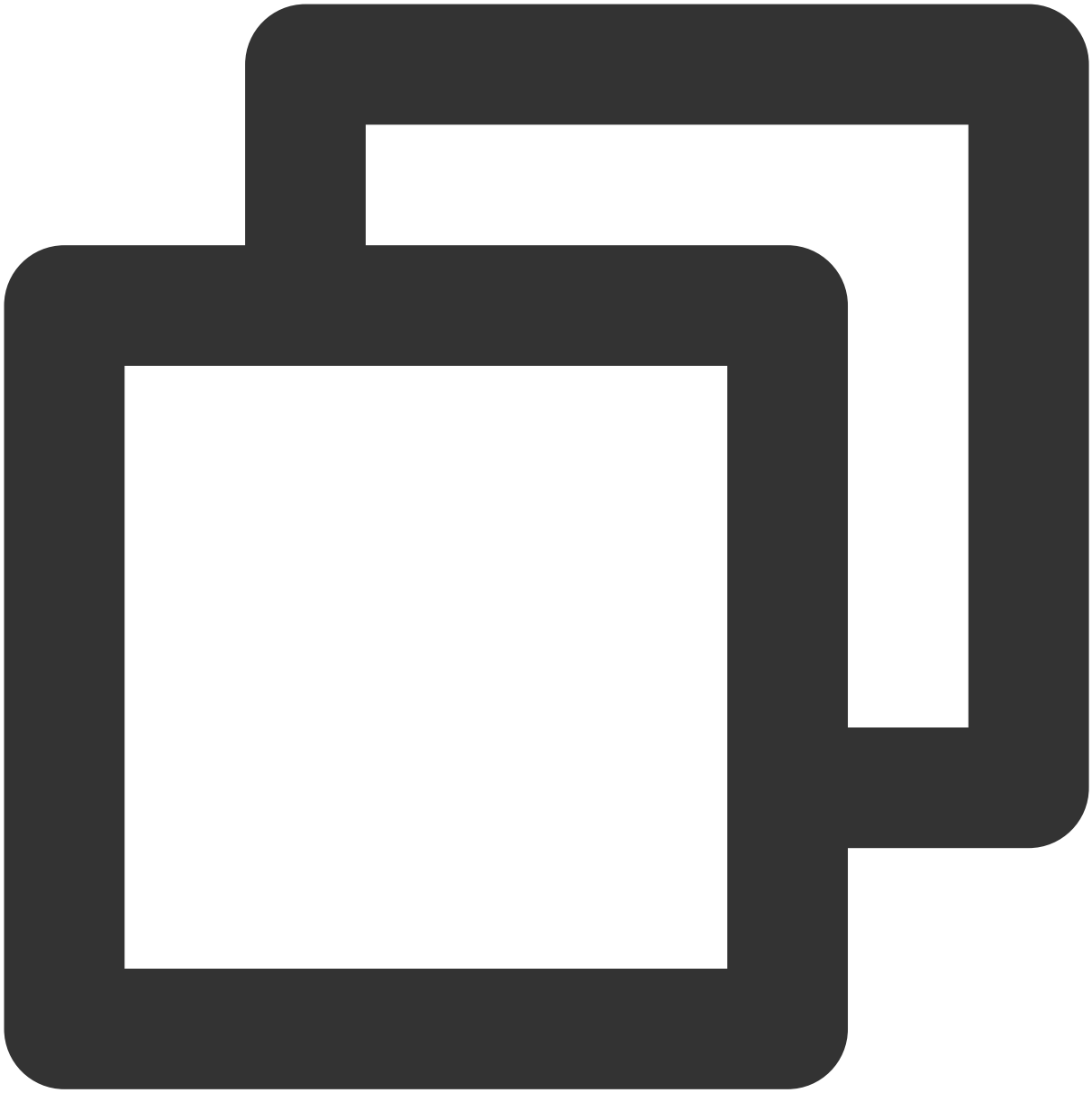
版权所有：腾讯云计算（北京）有限责任公司

```
  // cloudCustomData: 'your cloud custom data'
  onProgress: function(event) { console.log('file uploading:', event) }
});

// 2. 发送消息
let promise = chat.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);
});
```



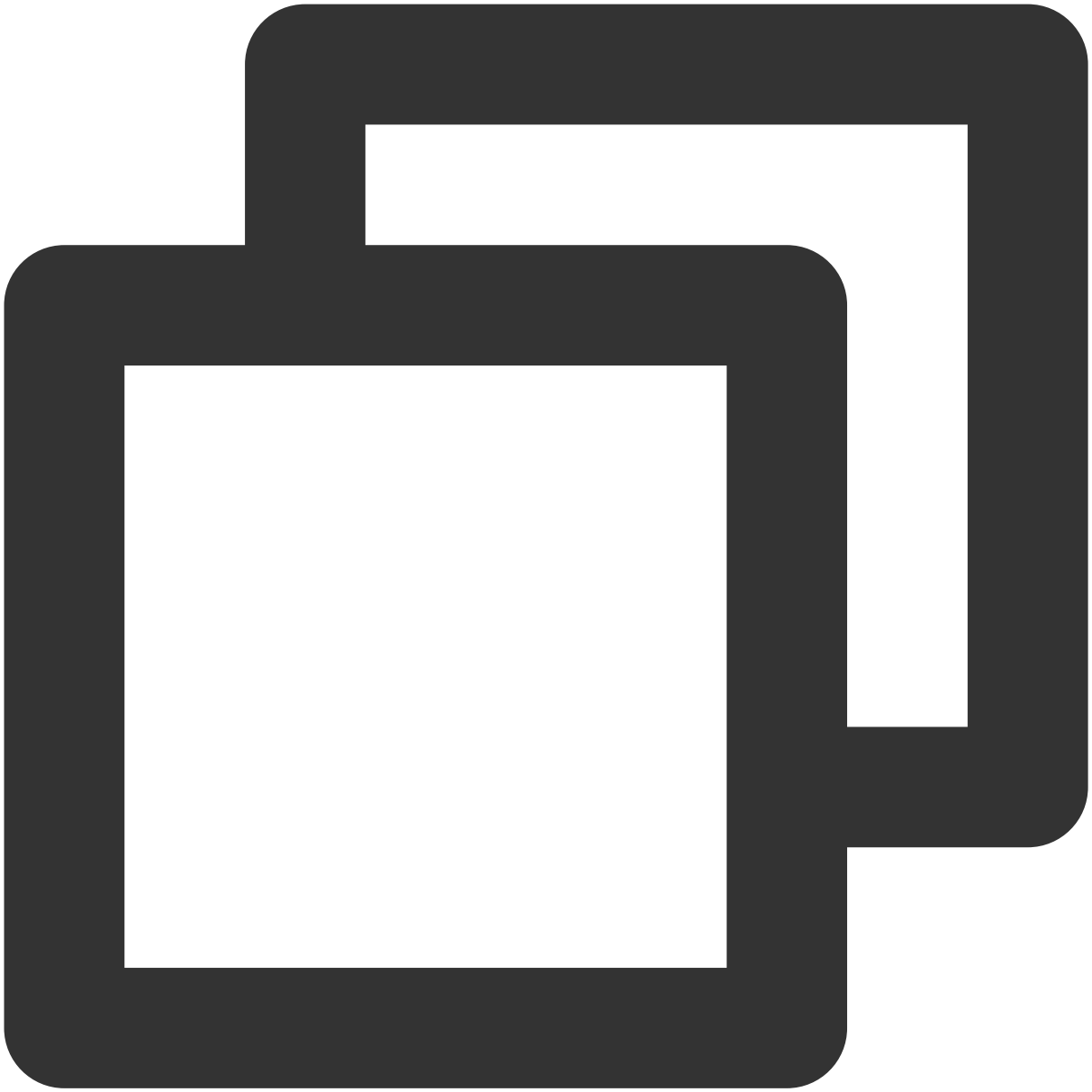
```
// Web 端发送文件消息示例2- 传入 File 对象
// 先在页面上添加一个 id 为 "testPasteInput" 的消息输入框
// 如 <input type="text" id="testPasteInput" placeholder="截图后粘贴到输入框中" size="3
document.getElementById('testPasteInput').addEventListener('paste', function(e) {
  let clipboardData = e.clipboardData;
  let file;
  let fileCopy;
  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
    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 = chat.createFileMessage({
    to: 'user1',
    conversationType: TencentCloudChat.TYPES.CONV_C2C,
    payload: {
      file: file
    },
    onProgress: function(event) { console.log('file uploading:', event) }
  });

  // 2. 发送消息
  let promise = chat.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);
  });
});
```



```
// uni-app 发送文件
// 1. 选择文件
uni.chooseFile({
  count: 1,
  extension:['.zip','.doc'],
  success: function(res) {
    let message = chat.createFileMessage({
      to: 'user1',
      conversationType: TencentCloudChat.TYPES.CONV_C2C,
      payload: { file: res },
      onProgress: function(event) { console.log('file uploading:', event) }
```

```
    });
    // 2. 发送消息
    let promise = chat.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);
    });
  }
});
```



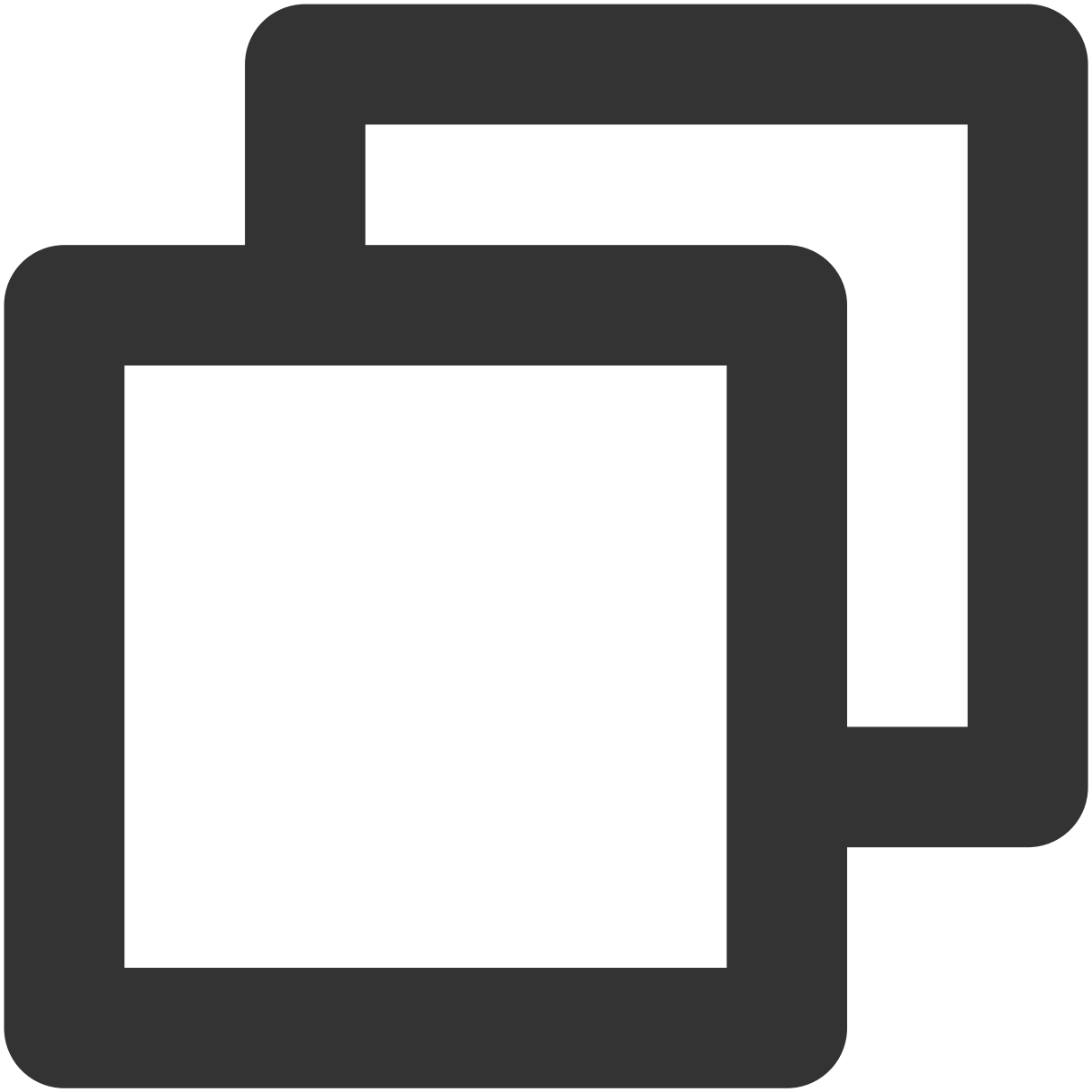
```
// 手机端发送文件
// wx.chooseMessageFile 基础库 2.5.0 开始支持，低版本需做兼容处理
// qq.chooseMessageFile 基础库 1.18.0 开始支持，低版本需做兼容处理
// 1. 从客户端会话选择文件
wx.chooseMessageFile({
  count: 1,
  type: 'all', // 从所有文件选择
  success: (res) => {
    const message = chat.createFileMessage({
      to: 'user1',
      conversationType: TencentCloudChat.TYPES.CONV_C2C,
```

```
        payload: { file: res },
        onProgress: function(event) { console.log('file uploading:', event) }
      });
      // 2. 发送消息
      let promise = chat.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);
      });
    }
  });
```

创建地理位置消息

创建地理位置消息的接口，此接口返回一个消息实例，可以在需要发送地理位置消息时调用 发送消息 接口发送消息。

接口



```
chat.createLocationMessage(options);
```

参数

参数 options 为 Object 类型，包含的属性值如下：

Name	Type	Default	Description
to	String	-	消息接收方的 userID
conversationType	String	-	会话类型，取值

			`TencentCloudCh` `TencentCloudCh`
priority	String	TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	消息内容的容器
cloudCustomData	String	''	消息自定义数据（云 重装后还能拉取到）

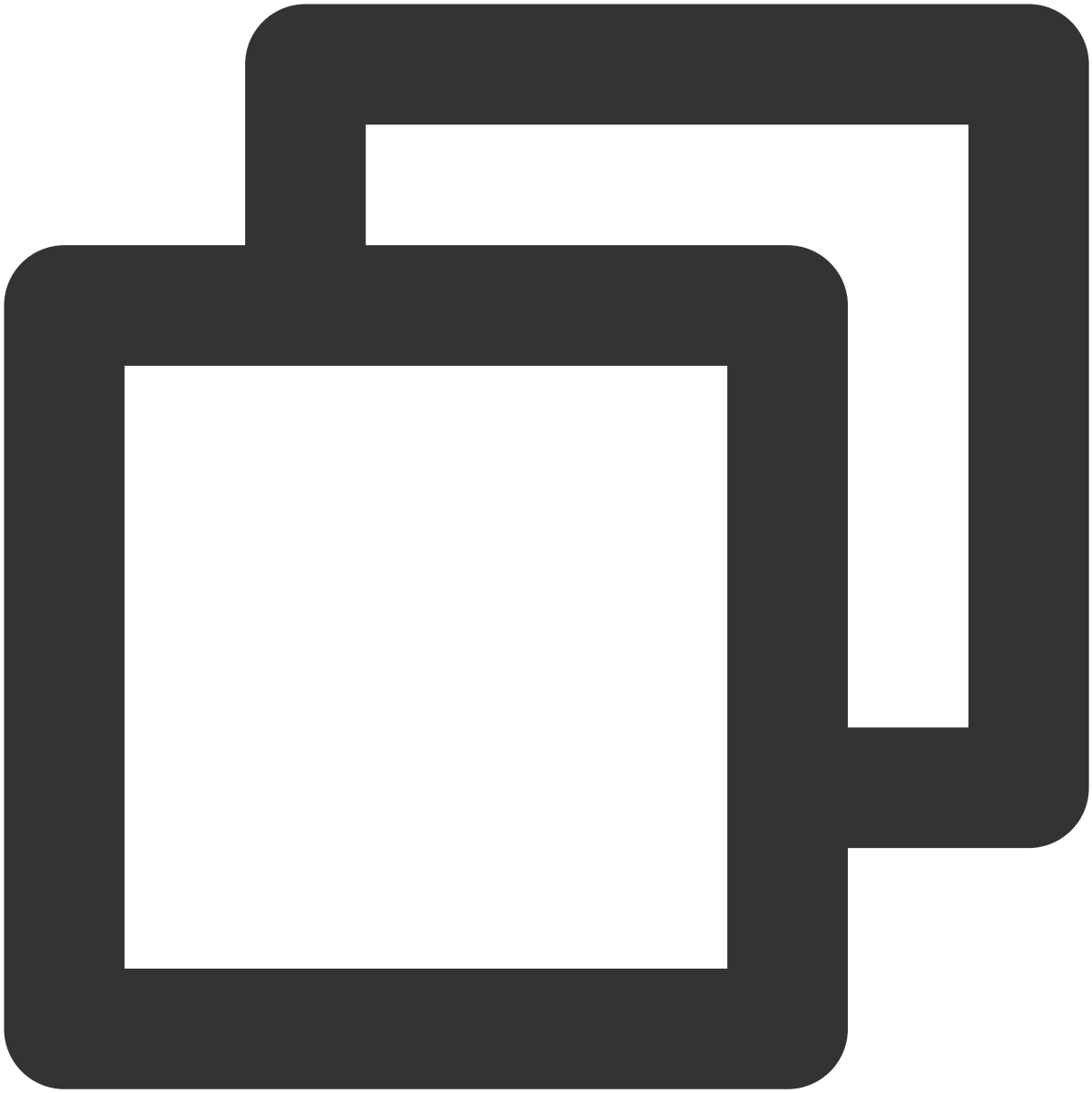
payload 的描述如下：

Name	Type	Description
description	String	地理位置描述信息
longitude	Number	经度
latitude	Number	纬度

返回值

Message

示例



```
// 发送地理位置消息，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 = chat.createLocationMessage({
  to: 'user1',
  conversationType: TencentCloudChat.TYPES.CONV_C2C,
  // 消息优先级，用于群聊
  // priority: TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL,
  payload: {
    description: '深圳市深南大道10000号腾讯大厦',
    longitude: 113.941079, // 经度
    latitude: 22.546103 // 纬度
```

```
  }
});
// 2. 发送消息
let promise = chat.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);
});
```

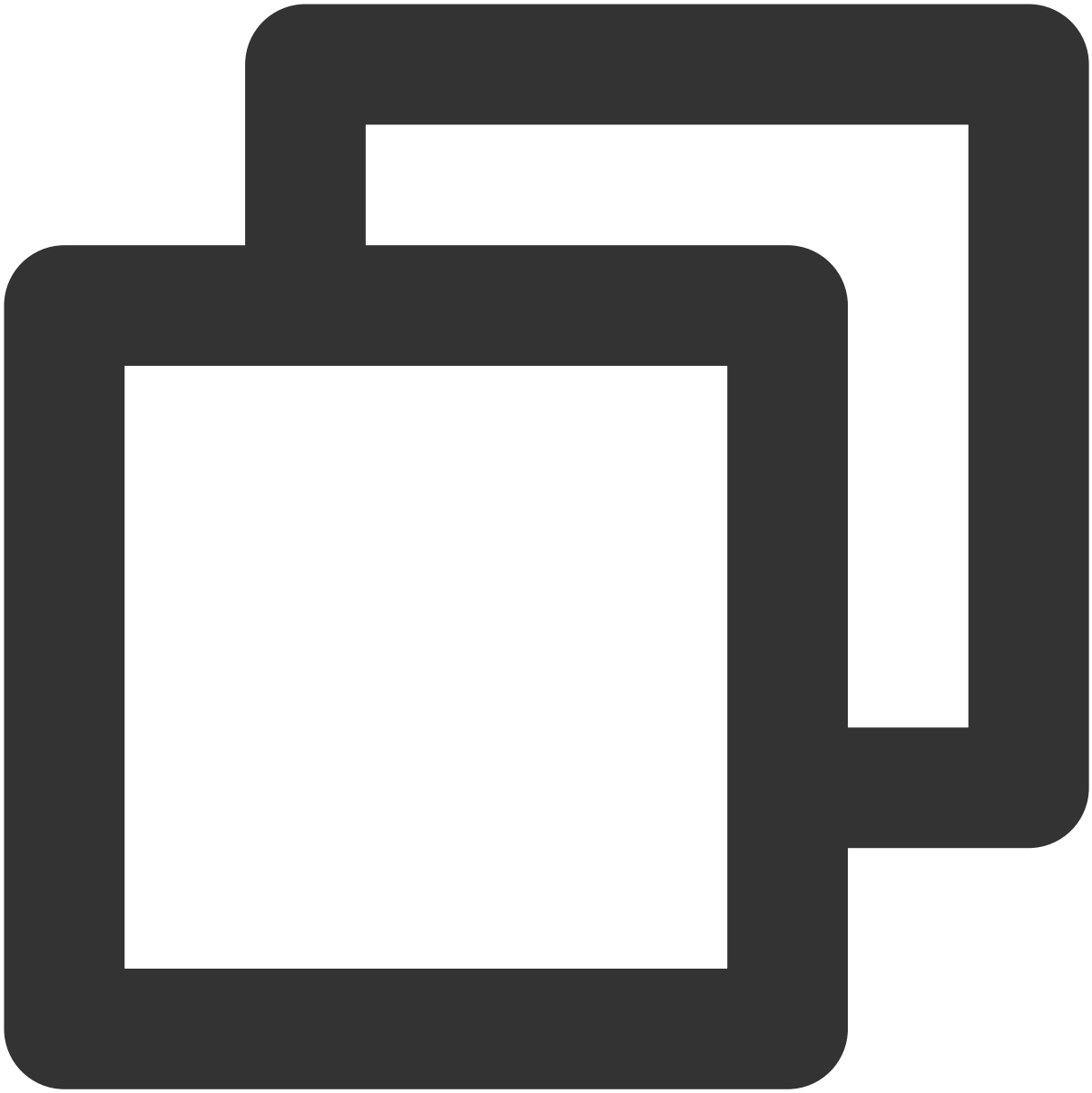
创建合并消息

创建合并消息的接口，此接口返回一个消息实例，可以在需要发送合并消息时调用 发送消息 接口发送消息。

注意

1. 不支持合并已发送失败的消息，如果消息列表内传入了已发送失败的消息，则创建消息接口会报错。

接口



```
chat.createMergerMessage(options);
```

参数

参数 options 为 Object 类型，包含的属性值如下：

Name	Type	Default	Description
to	String	-	消息接收方的 userID
conversationType	String	-	会话类型，取值

			`TencentCloudCh` 或 `TencentCloudCh`
priority	String	TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL	消息优先级
payload	Object	-	消息内容的容器
cloudCustomData	String	''	消息自定义数据（云 序卸载重装后还能拉

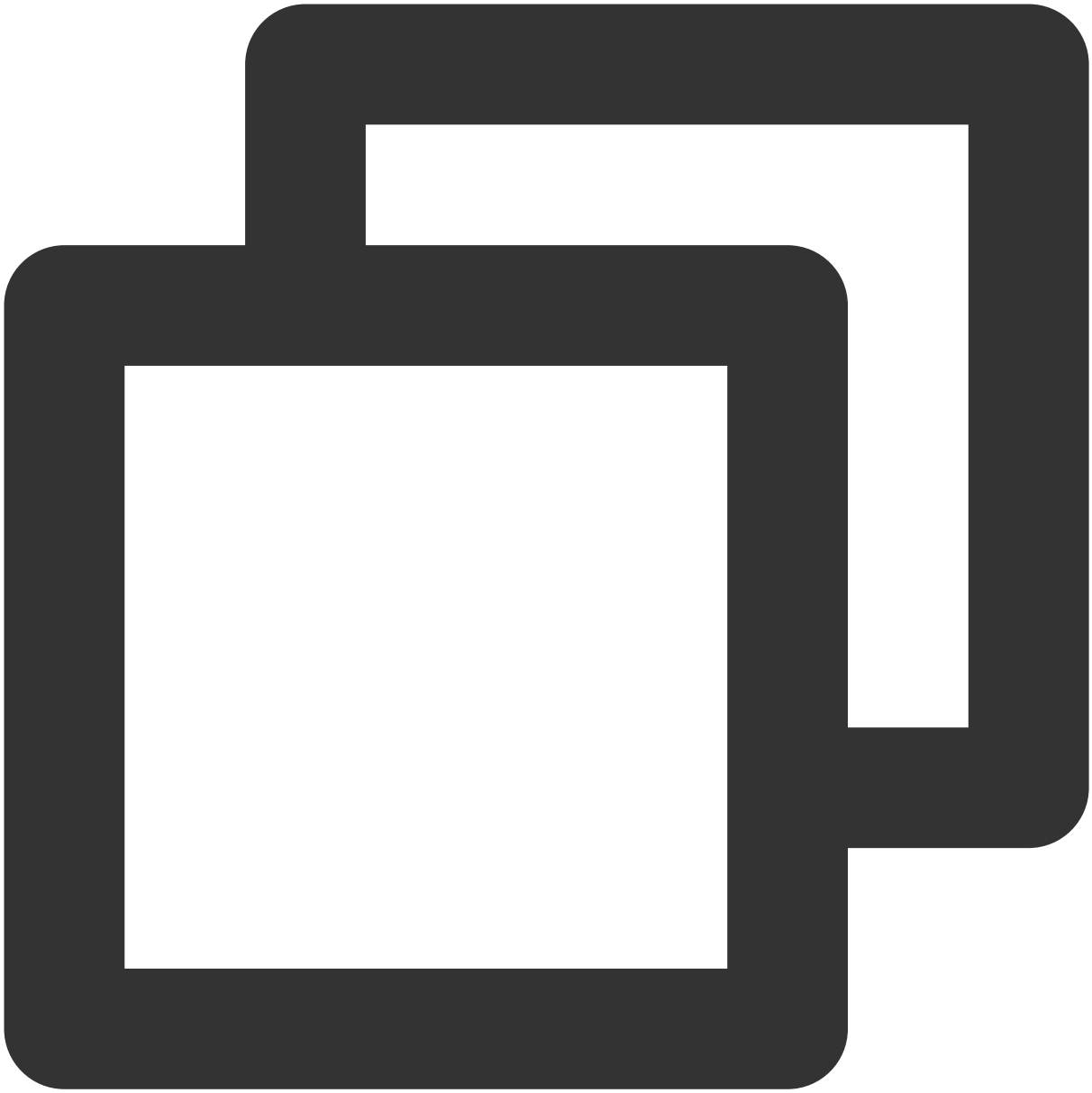
payload 的描述如下：

Name	Type	Description
messageList	Array	合并的消息列表
title	String	合并的标题，例如："大湾区前端人才中心的聊天记录"
abstractList	String	摘要列表，不同的消息类型可以设置不同的摘要信息，例如：文本消息可以设置为：sender: text，图片消息可以设置为：sender: [图片]，文件消息可以设置为：sender: [文件]
compatibleText	String	兼容文本，低版本 SDK 如果不支持合并消息，默认会收到一条文本消息，文本消息的内容为 ${compatibleText}，必填

返回值

Message

示例



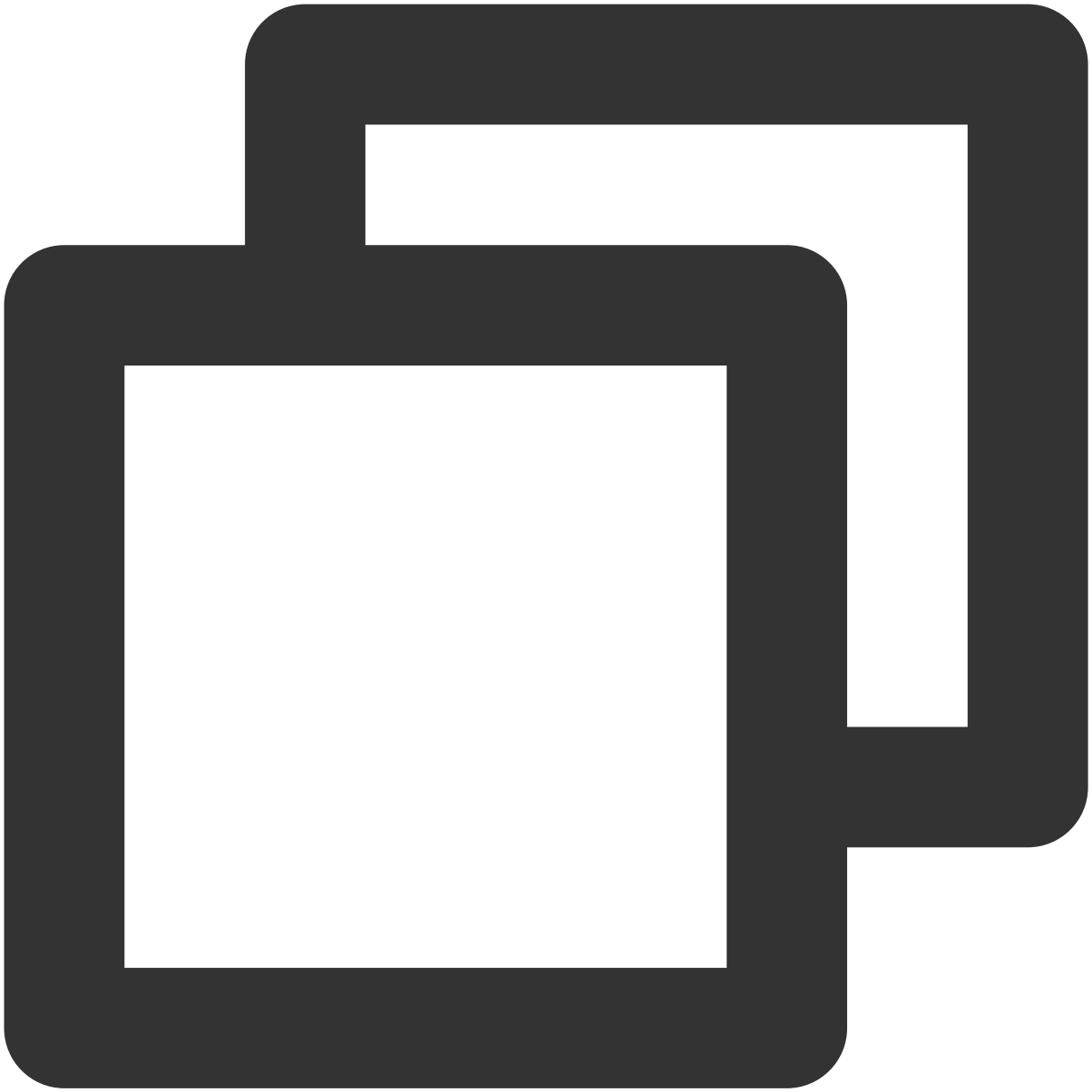
```
// 1. 将群聊消息转发到 c2c 会话
// message1 message2 message3 是群聊消息
let mergerMessage = chat.createMergerMessage({
  to: 'user1',
  conversationType: TencentCloudChat.TYPES.CONV_C2C,
  payload: {
    messageList: [message1, message2, message3],
    title: '大湾区前端人才中心的聊天记录',
    abstractList: ['allen: 666', 'iris: [图片]', 'linda: [文件]'],
    compatibleText: '请升级 Chat SDK 到v2.10.1或更高版本查看此消息'
  },
```

```
  // 消息自定义数据（云端保存，会发送到对端，程序卸载重装后还能拉取到）
  // cloudCustomData: 'your cloud custom data'
});

// 2. 发送消息
let promise = chat.sendMessage(mergerMessage);
promise.then(function(imResponse) {
  // 发送成功
  console.log(imResponse);
}).catch(function(imError) {
  // 发送失败
  console.warn('sendMessage error:', imError);
});
```

下载合并消息

如果发送方发送的合并消息较大，SDK 会将此消息存储到云端，消息接收方查看消息时，需要先把消息从云端下载到本地。

接口



```
chat.downloadMergerMessage(message);
```

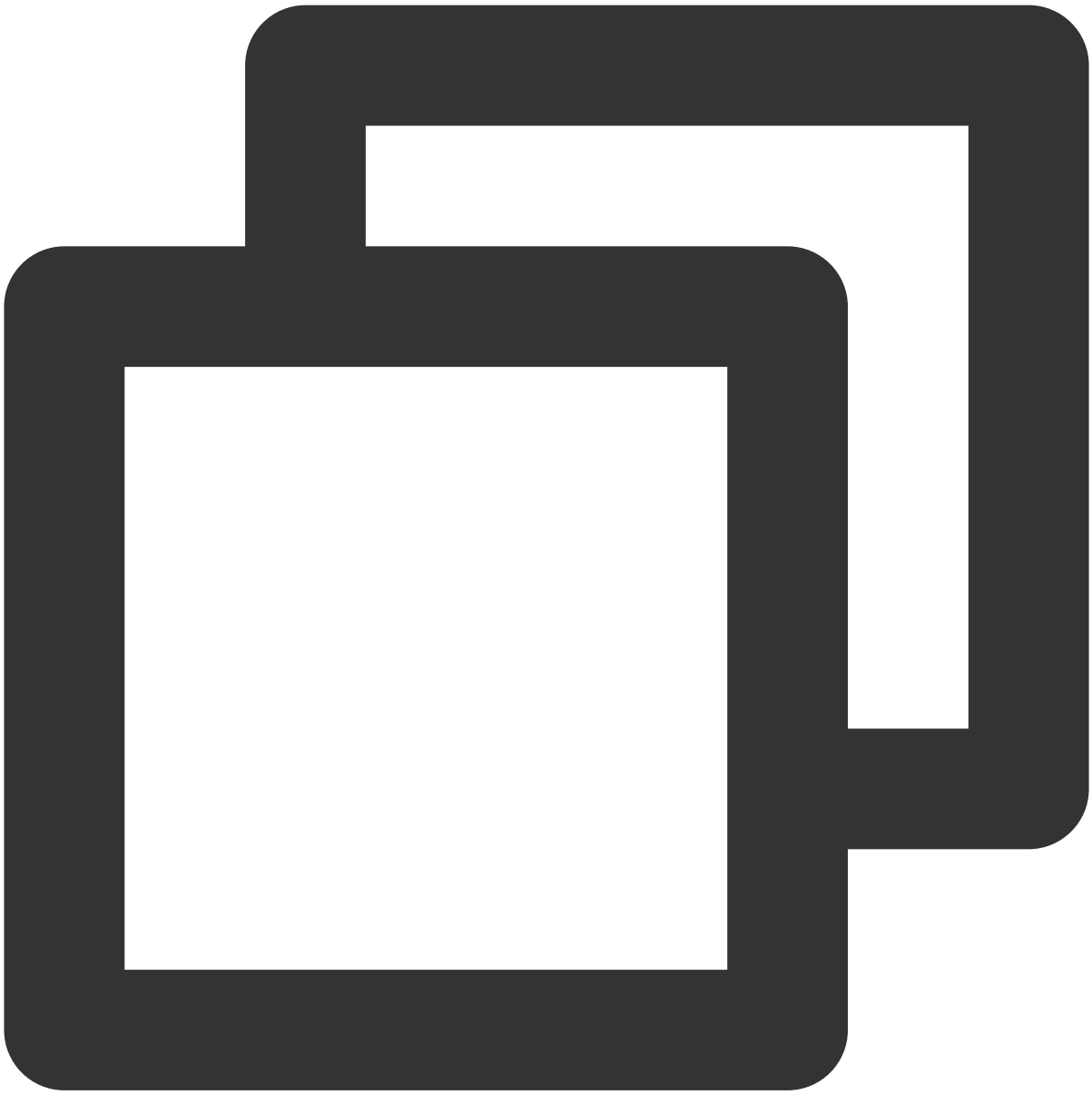
参数

Name	Type	Description
message	Message	消息实例

返回值

`Promise`

示例



```
// downloadKey 存在说明收到的合并消息存储在云端，需要先下载
if (message.type === TencentCloudChat.TYPES.MSG_MERGER && message.payload.downloadK
  let promise = chat.downloadMergerMessage(message);
  promise.then(function(imResponse) {
    // 下载成功后，SDK会更新 message.payload.messageList 等信息
    console.log(imResponse.data);
  }).catch(function(imError) {
```

```
    // 下载失败
    console.warn('downloadMergerMessage error:', imError);
  });
}
```

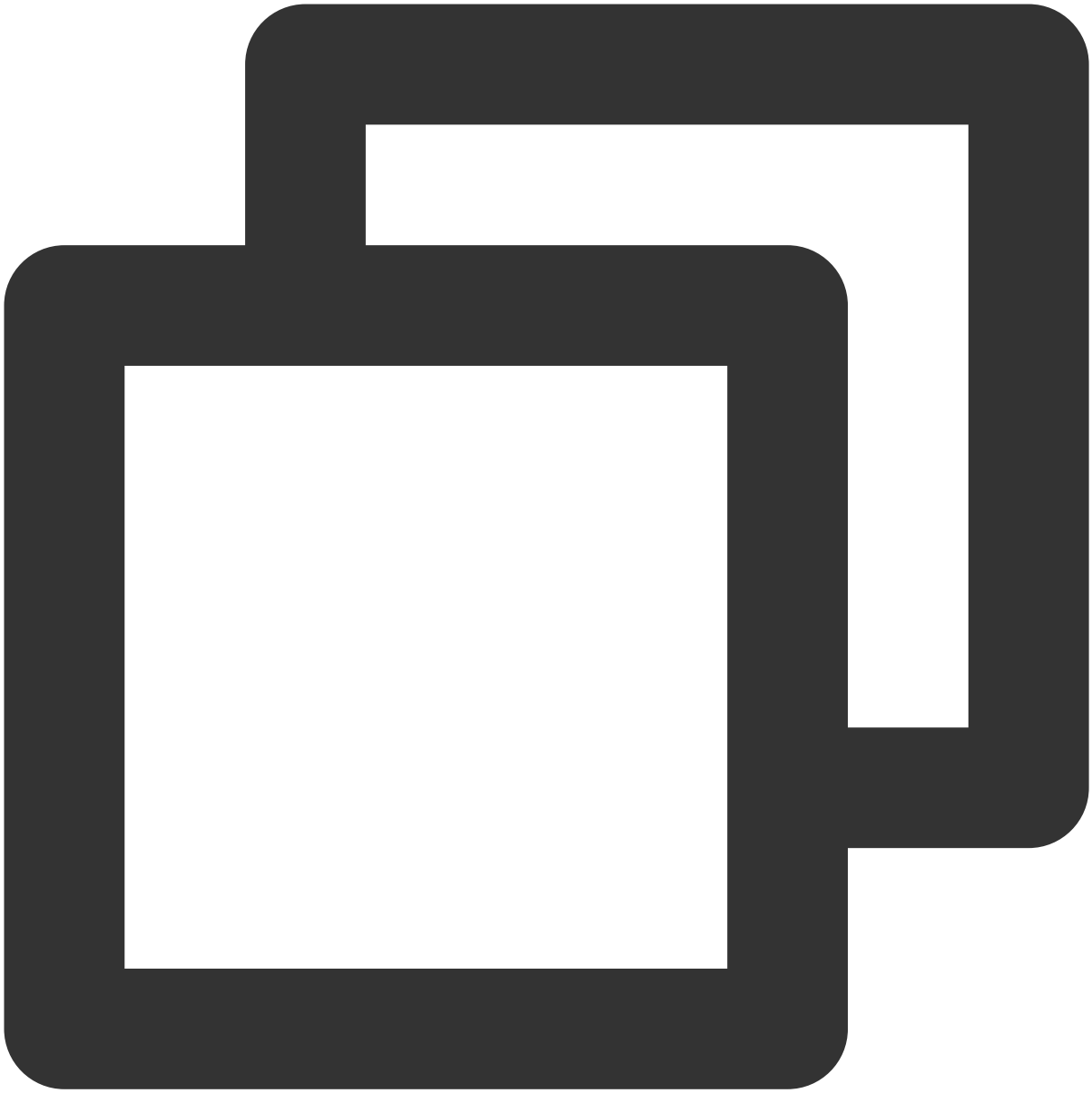
逐条转发消息

如果您需要转发单条消息，可以先通过 createForwardMessage 接口创建一条和原消息内容完全一样的转发消息，再调用 sendMessage 接口把转发消息发送出去。

注意

1. 支持单条转发和逐条转发。

接口

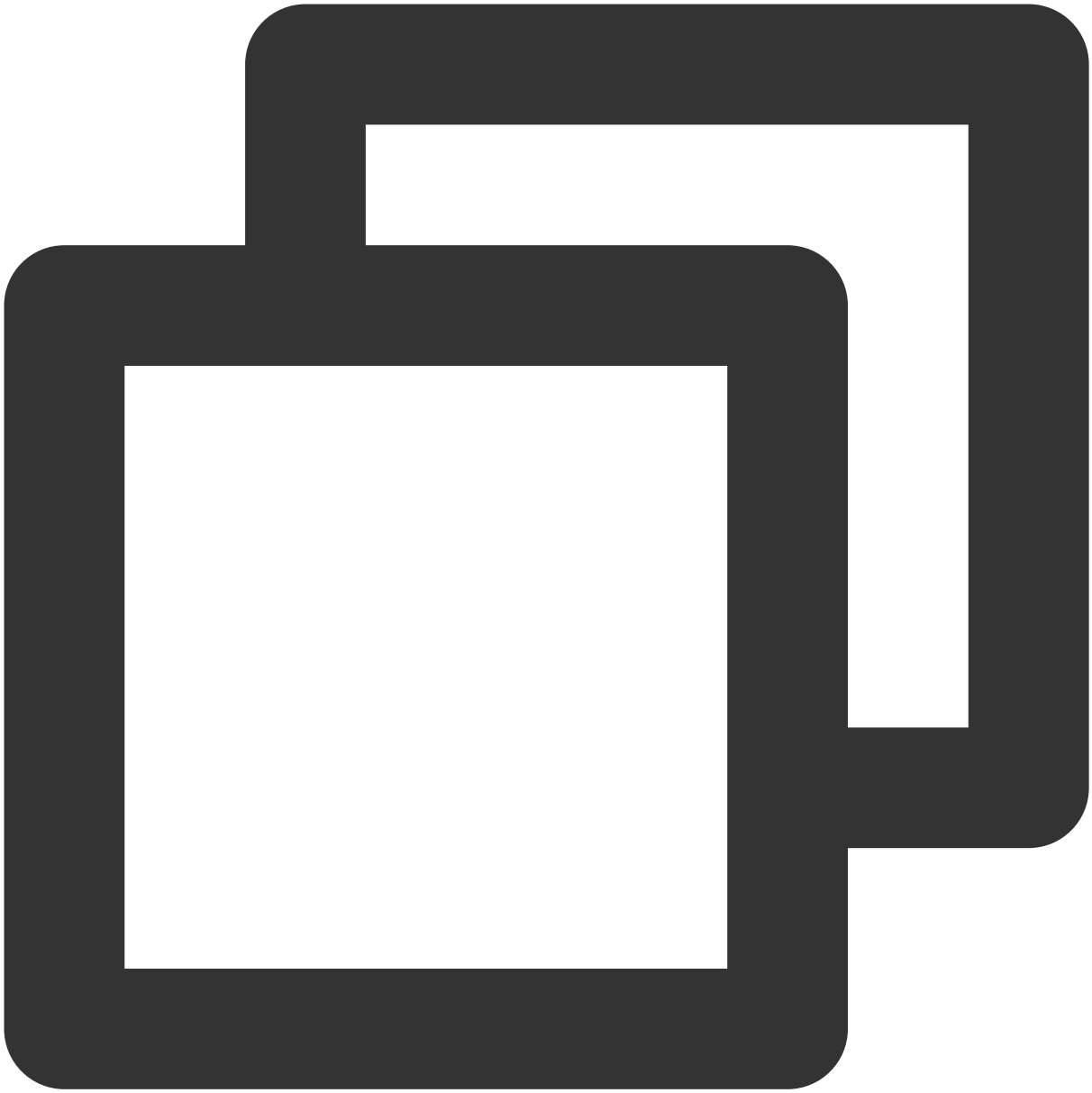


```
chat.createForwardMessage(message);
```

参数

Name	Type	Description
message	Message	消息实例

示例



```
let forwardMessage = chat.createForwardMessage({
  to: 'user1',
  conversationType: TencentCloudChat.TYPES.CONV_C2C,
  // 消息优先级，用于群聊
  // priority: TencentCloudChat.TYPES.MSG_PRIORITY_NORMAL,
  payload: message, // 消息实例，已收到的或自己已发出的消息
  // 消息自定义数据（云端保存，会发送到对端，程序卸载重装后还能拉取到）
  // cloudCustomData: 'your cloud custom data'
});
// 2. 发送消息
let promise = chat.sendMessage(forwardMessage);
```

版权所有：腾讯云计算（北京）有限责任公司

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

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 @ 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

创建地理位置消息

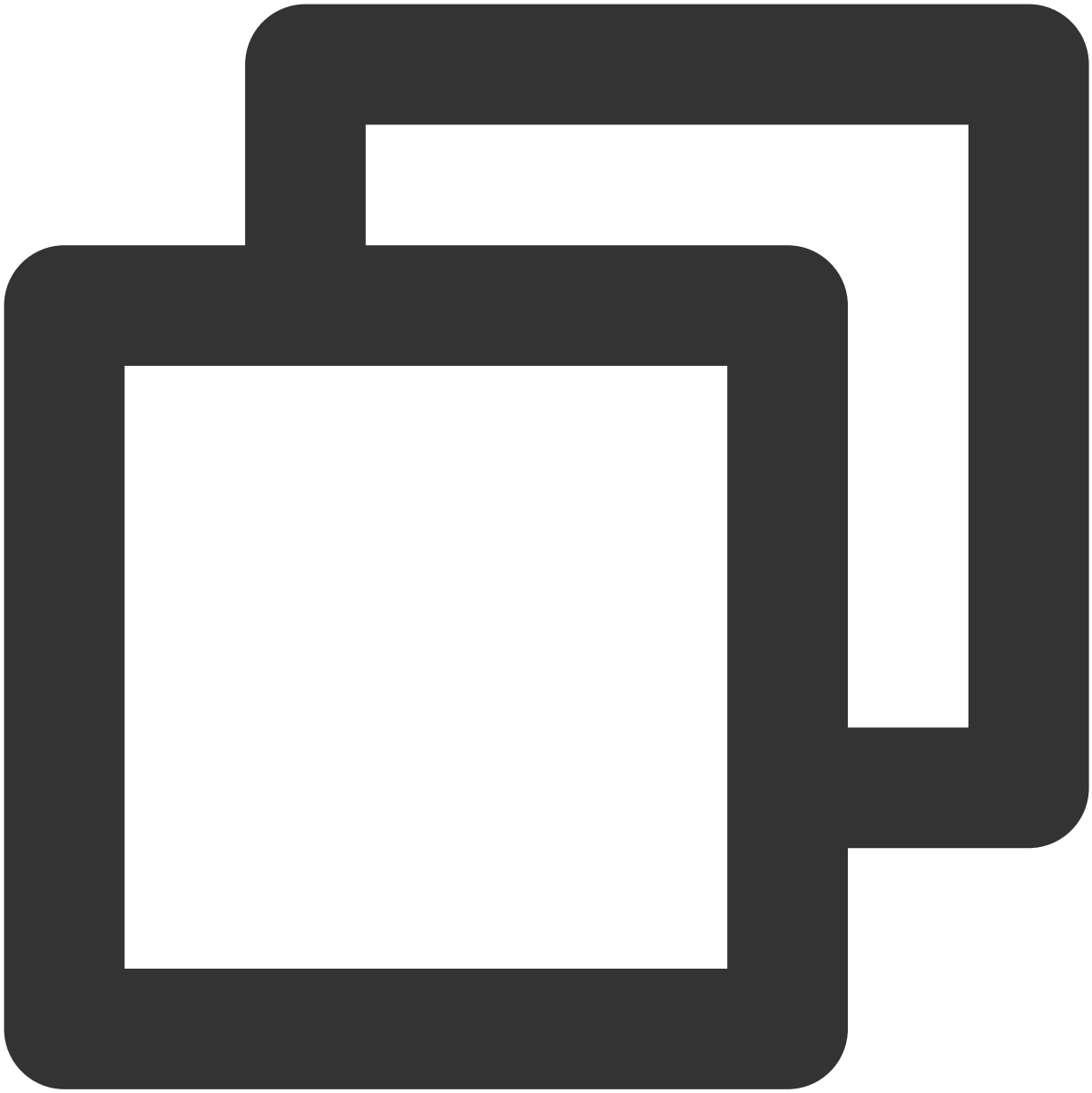
创建合并消息

创建转发消息

注意

1. 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： `TencentCloudChat.EVENT.SDK_READY` - sdk 处于 ready 状态时触发 `TencentCloudChat.EVENT.SDK_NOT_READY` - sdk 处于 not ready 状态时触发。
2. 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 TencentCloudChat.EVENT.MESSAGE_RECEIVED。
3. 本实例发送的消息，不会触发事件 TencentCloudChat.EVENT.MESSAGE_RECEIVED。同账号从其他端（或通过 REST API）发送的消息，会触发事件 TencentCloudChat.EVENT.MESSAGE_RECEIVED。
4. 离线推送仅适用于终端（Android 或 iOS），Web 不支持。
5. onlineUserOnly 和 messageControlInfo 不能同时使用。
6. 支持普通社群和话题消息不计未读。
7. 向【系统会话】发消息，SDK 将返回错误码 2106。
8. 消息包体大小限制为12KB，且不可调整，超出将导致消息发送失败，错误码 80002。
9. 启用好友关系检查后（【Chat 控制台】>【应用配置】>【功能配置】>【登录与消息】>【好友关系检查】），Chat 会在发起单聊时检查好友关系，仅允许好友之间发送单聊消息，陌生人发送单聊消息时 SDK 返回错误码 20009。

版权所有：腾讯云计算（北京）有限责任公司



```
chat.sendMessage(options);
```

参数

参数 options 为 Object 类型，包含的属性值如下：

Name	Type	Description
message	Message	消息实例
options	Object	消息发送选项（消息内容的容器），选填

options 的描述如下：

Name	Type	Description
onlineUserOnly	Boolean	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	离线推送 配置
messageControlInfo	Object	消息控制配置

offlinePushInfo 的描述如下：

Name	Type	Description
disablePush	Boolean	true 关闭离线推送；false 开启离线推送（默认）
disableVoipPush	Boolean	true 关闭 voip 推送（默认）；false 开启 voip 推送（开启 voip 推送需要同时开启离线推送）
title	String	离线推送标题，该字段为 iOS 和 Android 共用
description	String	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	离线推送透传内容
ignoreIOSBadge	Boolean	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	离线推送设置 OPPO 手机 8.0 系统及以上的渠道 ID

messageControlInfo 的描述如下：

Name	Type	Description
excludedFromUnreadCount	Boolean	true 消息不更新会话 unreadCount（消息存漫游），默认值 false
excludedFromLastMessage	Boolean	true 消息不更新会话 lastMessage（消息存漫游），默认值 false

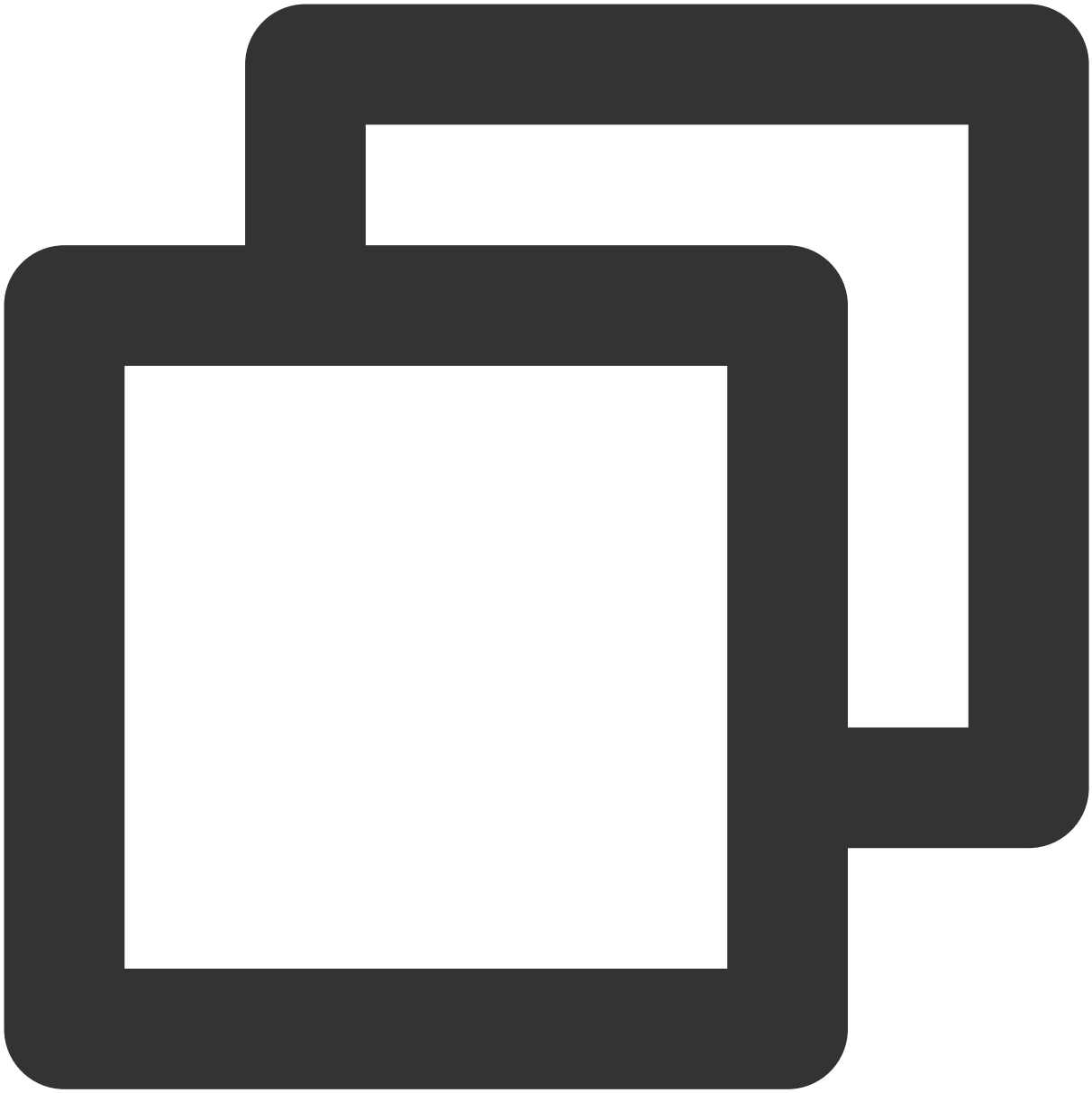
版权所有：腾讯云计算（北京）有限责任公司

excludedFromContentModeration	Boolean	消息是否不过内容审核（包含【本地审核】和【云端审核】） 只有在开通【本地审核】或【云端审核】功能后，excludedFromContentModeration 设置才有效，设置为 true，消息不过内容审核，设置为 false 消息过内容审核。

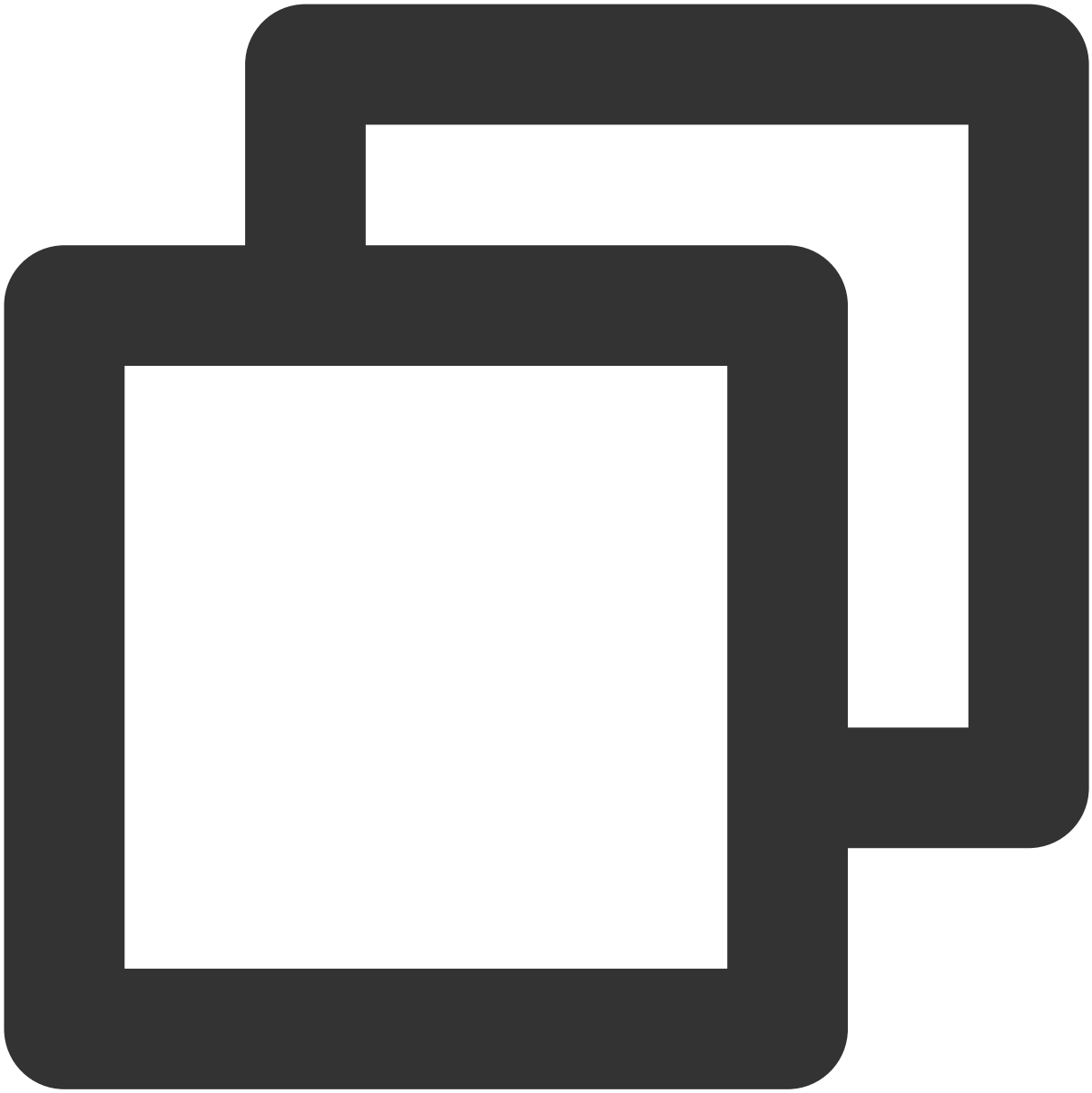
返回

`Promise`

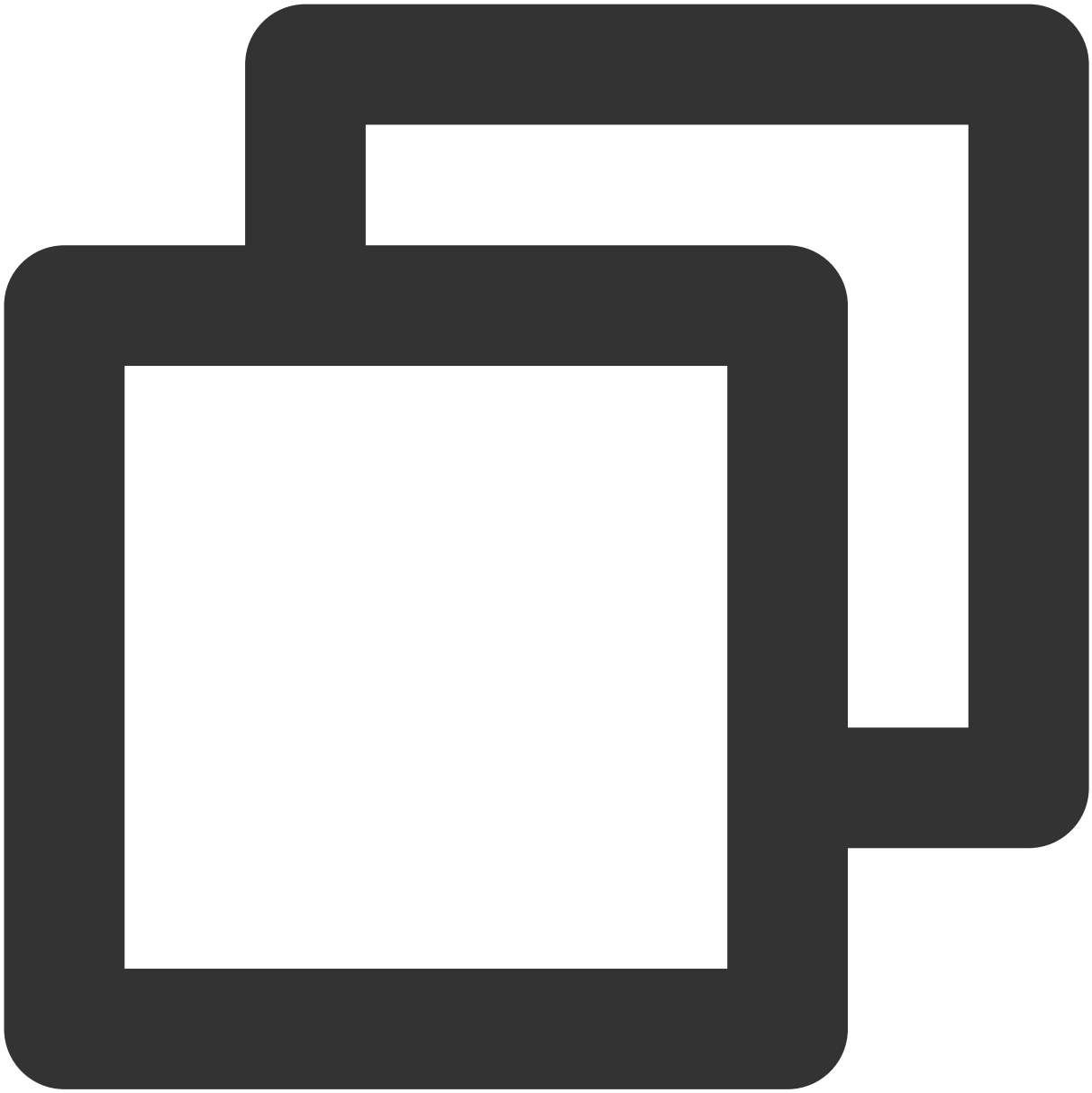
示例



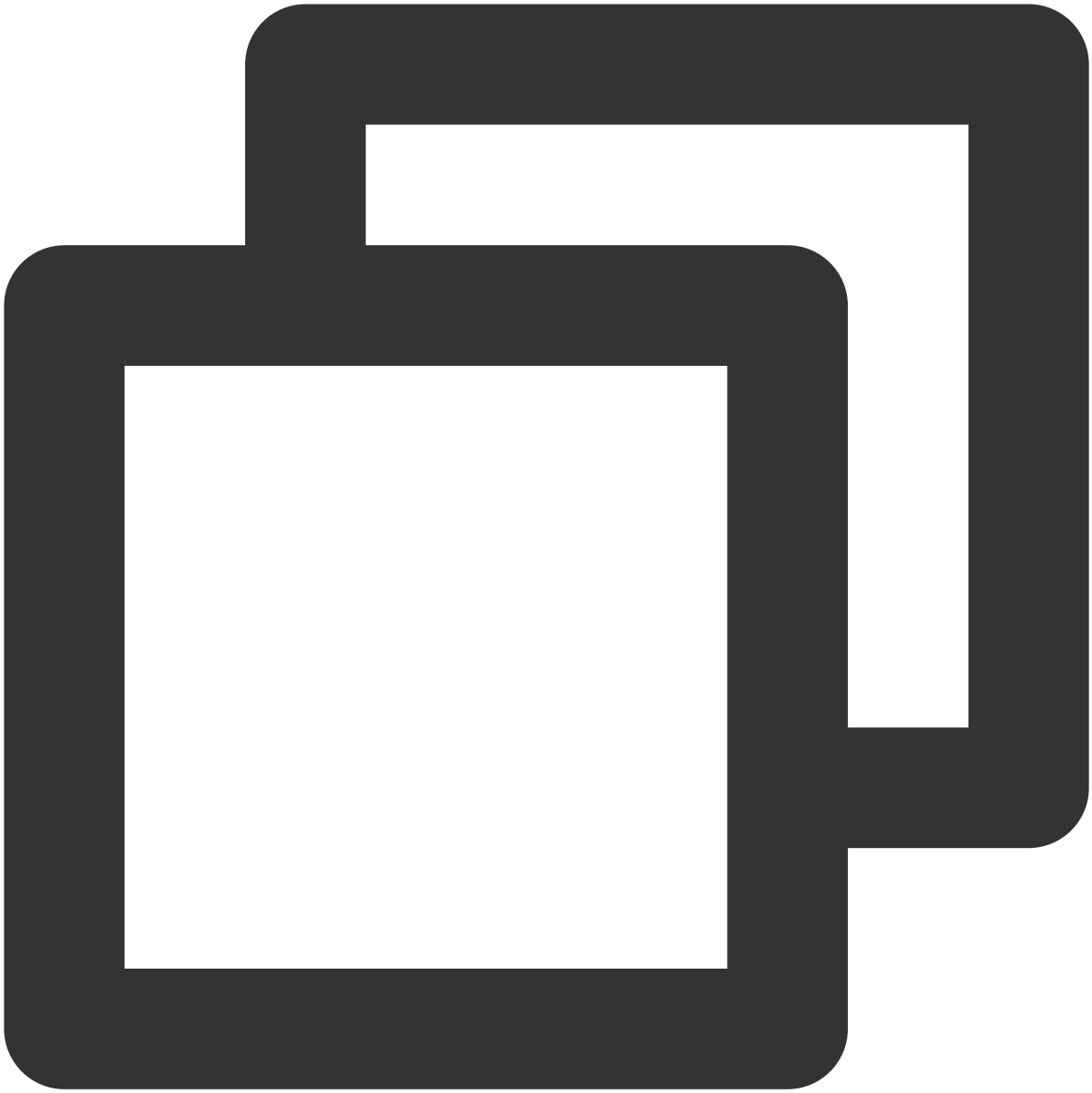
```
// 如果接收方不在线，则消息将存入漫游，且进行离线推送（在接收方 App 退后台或者进程被 kill 的情况下
// 离线推送的标题和内容使用默认值。
// 离线推送的说明请参考 https://intl.cloud.tencent.com/document/product/1047/33525
chat.sendMessage(message);
```



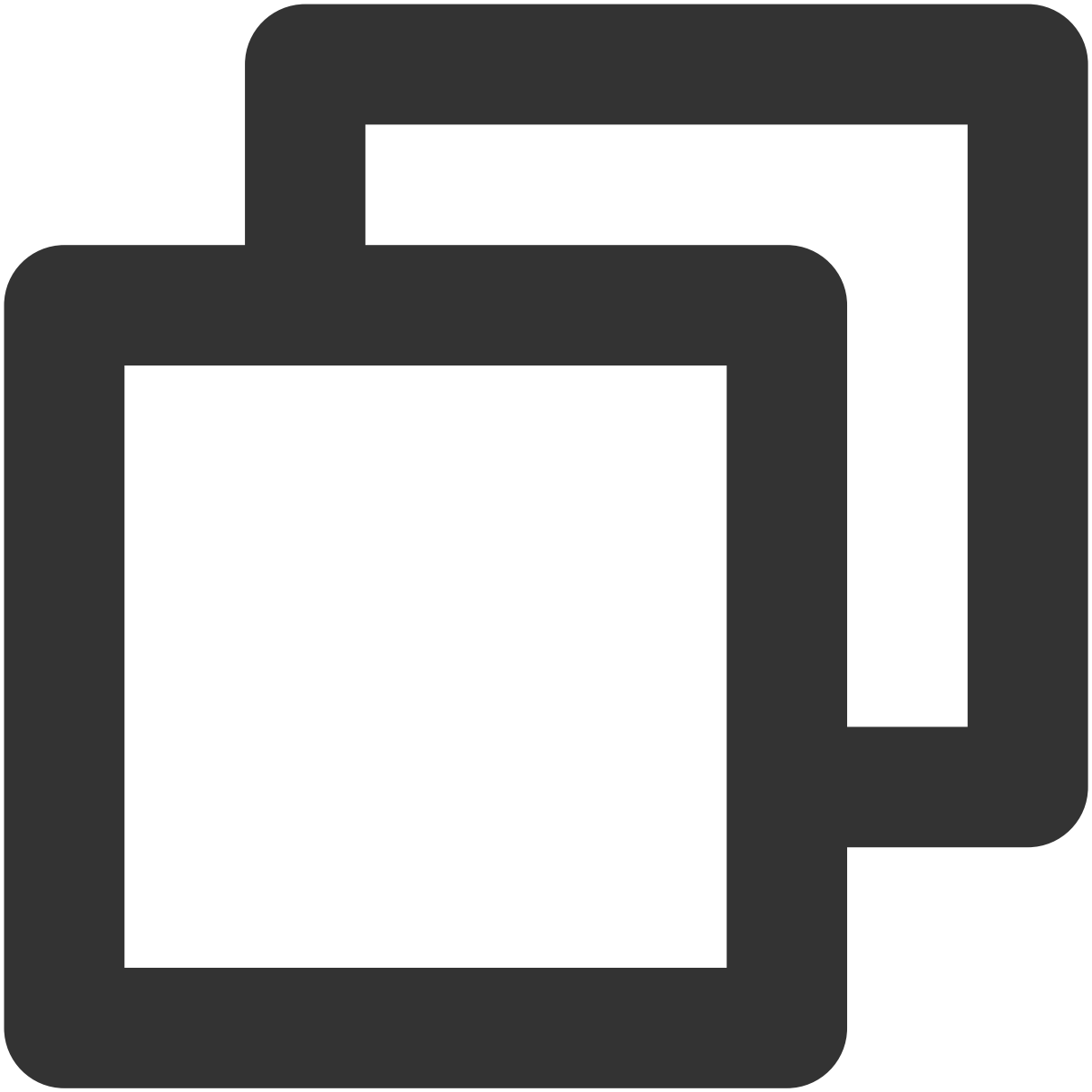
```
chat.sendMessage(message, {
  onlineUserOnly: true // 如果接收方不在线，则消息不存入漫游，且不会进行离线推送
});
```



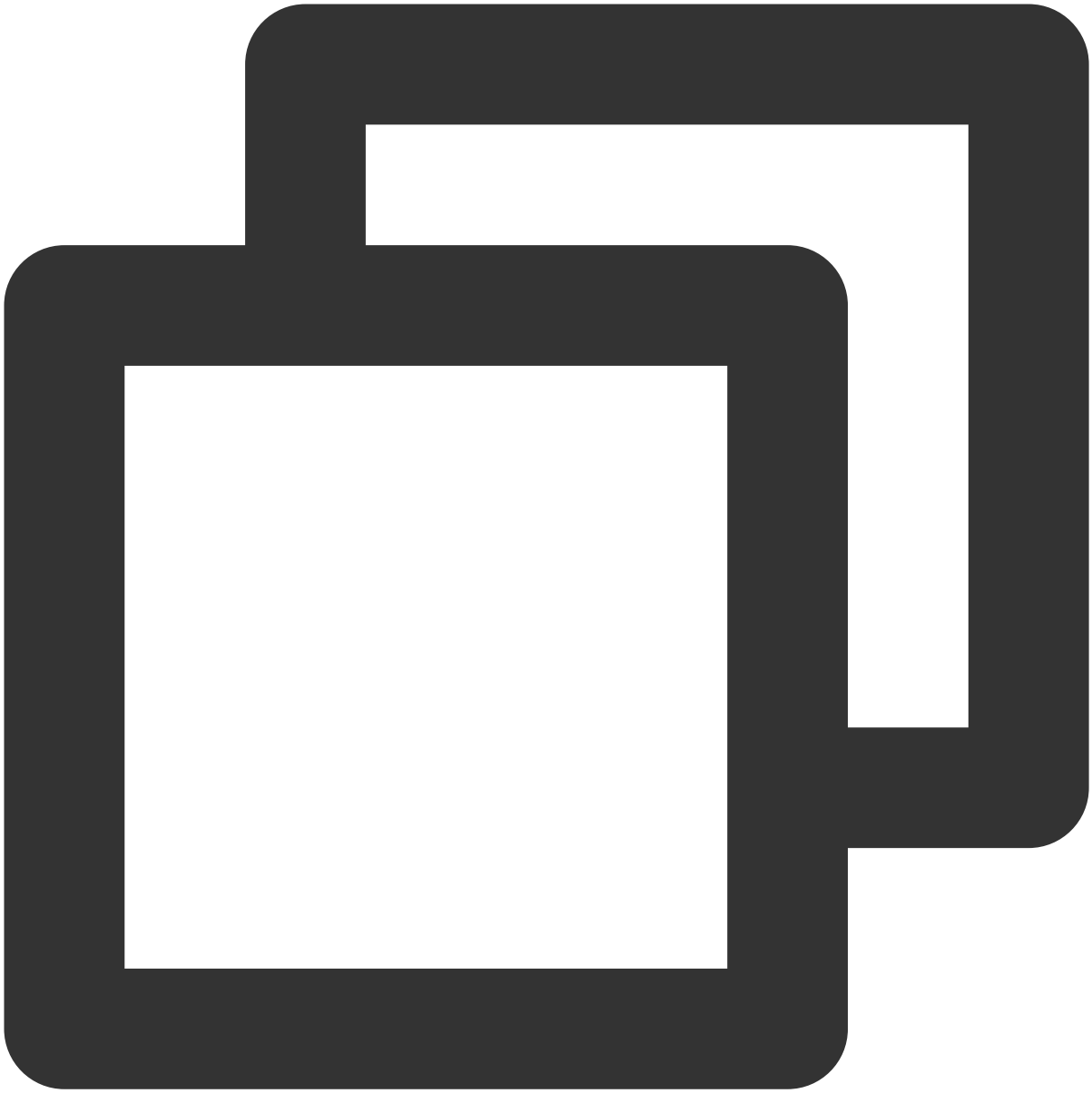
```
chat.sendMessage(message, {
  offlinePushInfo: {
    disablePush: true // 如果接收方不在线，则消息将存入漫游，但不进行离线推送
  }
});
```



```
chat.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
  // 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
  }
});
```



```
chat.sendMessage(message, {
  messageControlInfo: {
    excludedFromUnreadCount: true, // 消息不更新会话 unreadCount（消息存漫游）
    excludedFromLastMessage: true // 消息不更新会话 lastMessage（消息存漫游）
  }
});
```



```
// 消息不过内容审核
chat.sendMessage(message, {
  messageControlInfo: {
    excludedFromContentModeration: true,
  }
});
```

接口限制

功能特性	限制项	限制说明
单聊/群聊	内容长度	单聊、群聊消息，单条消息最大长度限制为 12K。
	发送频率	单聊消息：客户端发送单聊消息无限制；REST API 发送有频率限制，可查看相应接口的文档。 群聊消息：每个群限 40 条/秒（针对所有群类型、所有平台接口）。不同群内发消息，限频互不影响。
	接收频率	单聊和群聊均无限制。
	单个文件大小	发送文件消息时，SDK 最大支持发送单个文件大小为 100MB。

说明：

1. 消息数量超过限制后，后台优先下发优先级相对较高的消息，同等优先级的消息随机排序。如果同一秒内高优先级消息总数超过 40 条/秒，高优先级消息也会被抛弃。
2. 被频控限制的消息，不会下发，不会存入历史消息，但会给发送人返回成功，会触发 群内发言之前回调，但不会触发 群内发言之后回调。
3. REST API 发送群组消息接口调用限频默认为 200次/秒，与上述 “每个群发送消息限 40 条/秒” 是不同概念，请区分开。

更多限制请参见文档 使用限制。

Flutter

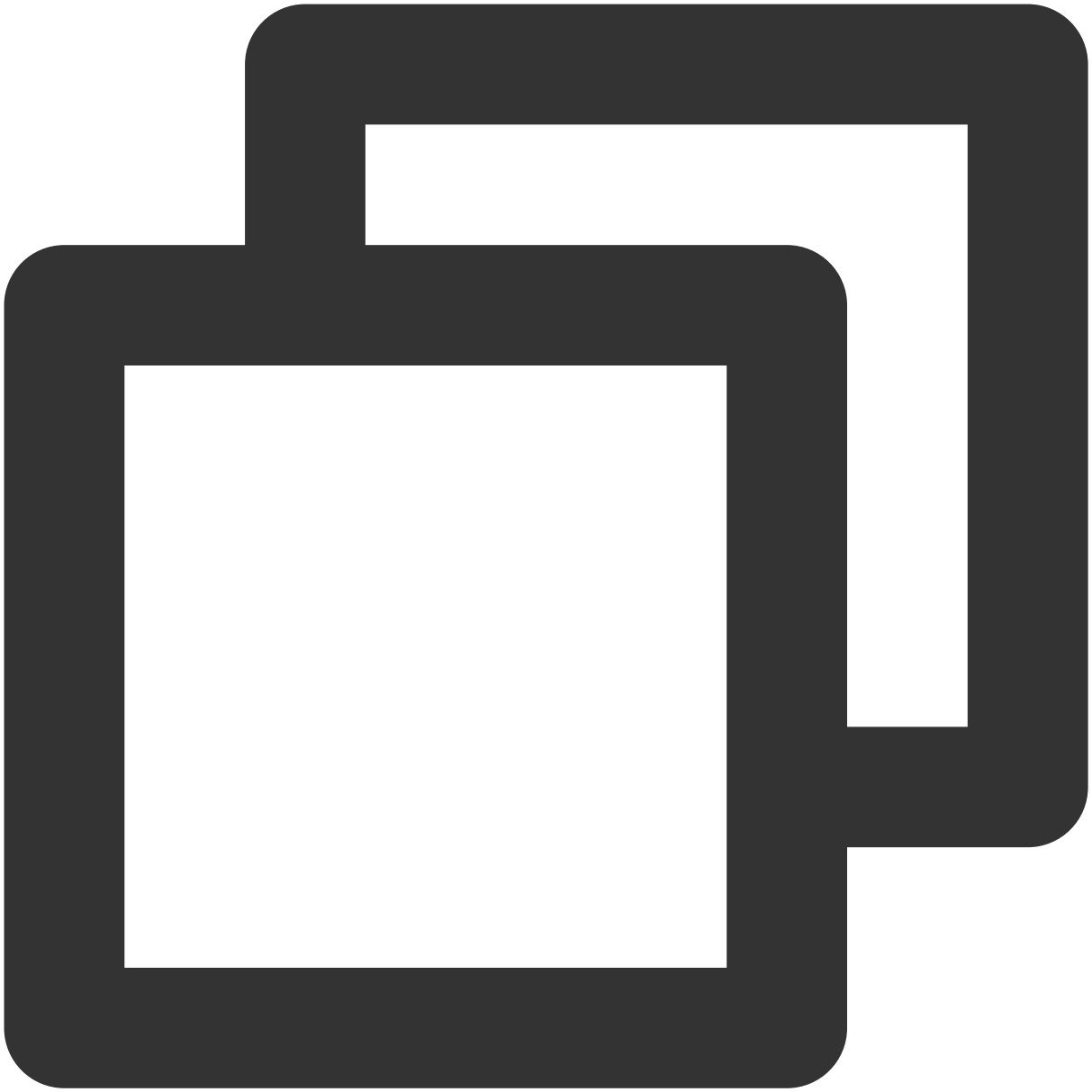
最近更新时间：2024-01-31 15:48:59

功能描述

发送消息方法在核心类 `TencentImSDKPlugin.v2TIMManager.getMessageManager()` 中。
支持发送文本、自定义、富媒体消息，消息类型都是 `V2TimMessage` 。
`V2TimMessage` 中可以携带不同类型子类，表示不同类型的消息。

重点接口说明

接口 `sendMessage` (点击查看详情) 是发送消息中最核心的接口。该接口支持发送所有类型的消息。
说明：
下文中提到的发消息高级接口，指的都是 `sendMessage` 。
接口说明如下：



```
Future<V2TimValueCallback<V2TimMessage>> sendMessage(
{
  required String id,
  required String receiver,
  required String groupID,
  int priority = 0,
  bool onlineUserOnly = false,
  bool needReadReceipt = false,
  bool isExcludedFromUnreadCount = false,
  bool isExcludedFromLastMessage = false,
  Map<String, dynamic>? offlinePushInfo,
```

```
  String? cloudCustomData,
  String? localCustomData,
}
)
```

参数说明：

参数	含义	单聊有效	群聊有效	说明
id	创建消息返回的id	YES	YES	需要通过对应的 `createXxxMessage` 接口先行创建
receiver	单聊消息接收者 userID	YES	NO	如果是发送 C2C 单聊消息，只需要指定 receiver 即可
groupID	群聊 groupID	NO	YES	如果是发送群聊消息，只需要指定 groupID 即可
priority	消息优先级	NO	YES	请把重要消息设置为高优先级（例如红包、礼物消息），高频且不重要的消息设置为低优先级（例如点赞消息）
onlineUserOnly	是否只有在线用户才能收到	YES	YES	如果设置为 YES，接收方历史消息拉取不到，常被用于实现”对方正在输入”或群组里的非重要提示等弱提示功能
offlinePushInfo	离线推送信息	YES	YES	离线推送时携带的标题和内容
needReadReceipt	发送群消息是否支持已读	NO	YES	发送群消息是否支持已读
isExcludedFromUnreadCount	发送消息是否计入会话未读数	YES	YES	如果设置为 true，发送消息不会计入会话未读，默认为 false
isExcludedFromLastMessage	发送消息是否计入会话 lastMessage	YES	YES	如果设置为 true，发送消息不会计入会话 lastMessage，默认为 false
cloudCustomData	消息云端数据	YES	YES	消息附带的额外的数据，存云端，消息的接受者可以访问到
localCustomData	消息本地数据	YES	YES	消息附带的额外的数据，存本地，消息的接受者不可以访问到，App 卸载后数据丢失

版权所有：腾讯云计算（北京）有限责任公司

说明：
如果 groupID 和 receiver 同时设置，表示给 receiver 发送定向群消息。具体请参考 群定向消息。

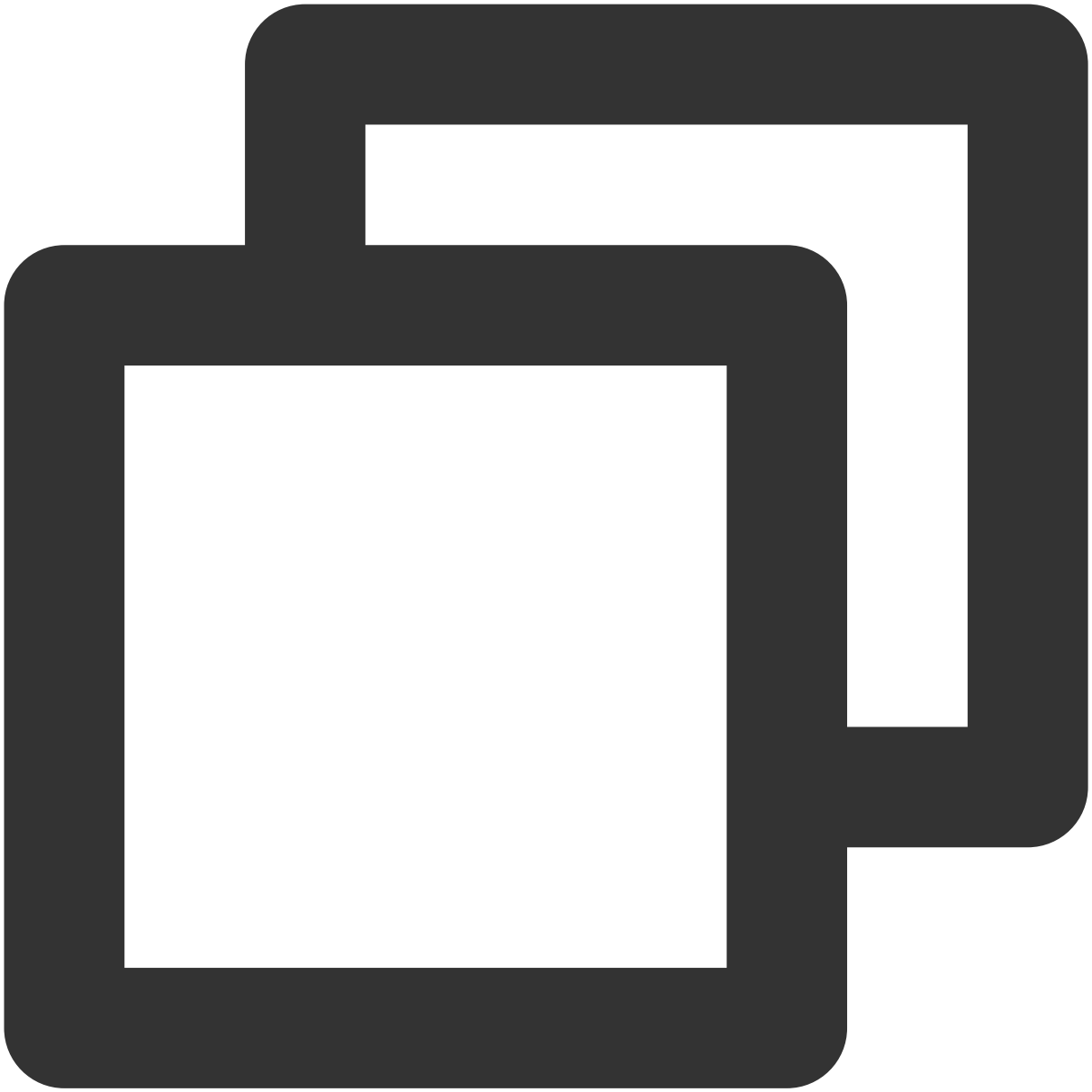
发送文本消息

文本消息区分单聊和群聊，涉及的接口、传参有所区别。
发送文本消息可以采用两种接口：普通接口和高级接口。高级接口比普通接口能设置更多的发送参数（例如优先级、离线推送信息等）。
普通接口参考下文具体描述，高级接口就是上文中提到的 `sendMessage` 。

单聊文本消息

高级接口

调用高级接口发送单聊文本消息分两步：
1. 调用 `createTextMessage` (点击查看详情) 创建文本消息。
2. 调用 `sendMessage` (点击查看详情) 发送消息。
示例代码如下：



```
// 创建文本消息
V2TimValueCallback<V2TimMsgCreateInfoResult> createTextMessageRes =
    await TencentImSDKPlugin.v2TIMManager
        .getMessageManager()
        .createTextMessage(
          text: "test", // 文本信息
        );
if (createTextMessageRes.code == 0) {
  // 文本信息创建成功
  String? id = createTextMessageRes.data?.id;
  // 发送文本消息
```

```
        // 在sendMessage时，若只填写receiver则发个人用户单聊消息
        //                 若只填写groupID则发群组消息
        //                 若填写了receiver与groupID则发群内的个人用户，消息在群聊中显示，只有
        V2TimValueCallback<V2TimMessage> sendMessageRes =
            await TencentImSDKPlugin.v2TIMManager.getMessageManager().sendMessage(
                id: id!, // 创建的messageid
                receiver: "userID", // 接收人id
                groupID: "groupID", // 接收群组id
                priority: MessagePriorityEnum.V2TIM_PRIORITY_DEFAULT, // 消息优先级
                onlineUserOnly:
                    false, // 是否只有在线用户才能收到，如果设置为 true ，接收方历史消息拉取不到
                isExcludedFromUnreadCount: false, // 发送消息是否计入会话未读数
                isExcludedFromLastMessage: false, // 发送消息是否计入会话 lastMessage
                needReadReceipt:
                    false, // 消息是否需要已读回执（只有 Group 消息有效，6.1 及以上版本支持，需
                offlinePushInfo: OfflinePushInfo(), // 离线推送时携带的标题和内容
                cloudCustomData: "", // 消息云端数据，消息附带的额外的数据，存云端，消息的接受
                localCustomData:
                    "" // 消息本地数据，消息附带的额外的数据，存本地，消息的接受者不可以访问到，A
                );
        if (sendMessageRes.code == 0) {
          // 发送成功
        }
    }
```

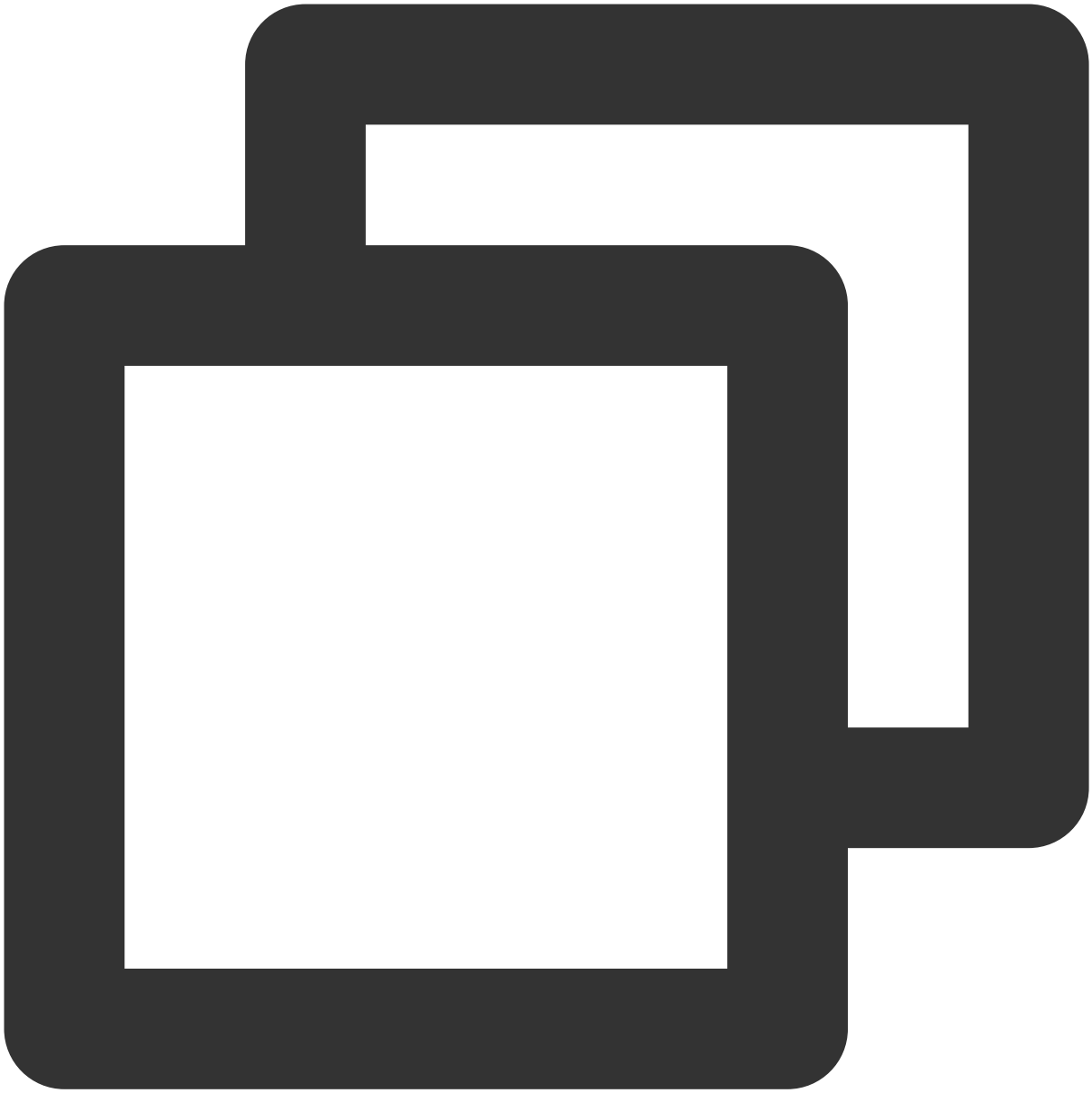
群聊文本消息

高级接口

调用高级接口发送群聊文本消息分两步：

1. 调用 `createTextMessage` (点击查看详情) 创建文本消息。
2. 调用 `sendMessage` (点击查看详情) 发送消息。

示例代码如下：



```
// 创建文本消息
V2TimValueCallback<V2TimMsgCreateInfoResult> createTextMessageRes =
    await TencentImSDKPlugin.v2TIMManager
        .getMessageManager()
        .createTextMessage(
          text: "test", // 文本信息
        );
if (createTextMessageRes.code == 0) {
  // 文本信息创建成功
  String? id = createTextMessageRes.data?.id;
  // 发送文本消息
```

```
        // 在sendMessage时，若只填写receiver则发个人用户单聊消息
        //                 若只填写groupID则发群组消息
        //                 若填写了receiver与groupID则发群内的个人用户，消息在群聊中显示，只有
        V2TimValueCallback<V2TimMessage> sendMessageRes =
            await TencentImSDKPlugin.v2TIMManager.getMessageManager().sendMessage(
                id: id!, // 创建的messageid
                receiver: "userID", // 接收人id
                groupID: "groupID", // 接收群组id
                priority: MessagePriorityEnum.V2TIM_PRIORITY_DEFAULT, // 消息优先级
                onlineUserOnly:
                    false, // 是否只有在线用户才能收到，如果设置为 true ，接收方历史消息拉取不到
                isExcludedFromUnreadCount: false, // 发送消息是否计入会话未读数
                isExcludedFromLastMessage: false, // 发送消息是否计入会话 lastMessage
                needReadReceipt:
                    false, // 消息是否需要已读回执（只有 Group 消息有效，6.1 及以上版本支持，需
                offlinePushInfo: OfflinePushInfo(), // 离线推送时携带的标题和内容
                cloudCustomData: "", // 消息云端数据，消息附带的额外的数据，存云端，消息的接受
                localCustomData:
                    "" // 消息本地数据，消息附带的额外的数据，存本地，消息的接受者不可以访问到，A
                );
        if (sendMessageRes.code == 0) {
          // 发送成功
        }
    }
```

发送自定义消息

自定义消息区分单聊和群聊，涉及的接口或者传参有所区别。发送自定义消息可以采用两种接口：普通接口和高级接口。

高级接口即上文中已介绍过的 `sendMessage` (点击查看详情)，比普通接口能设置更多的发送参数（例如优先级、离线推送信息等）。

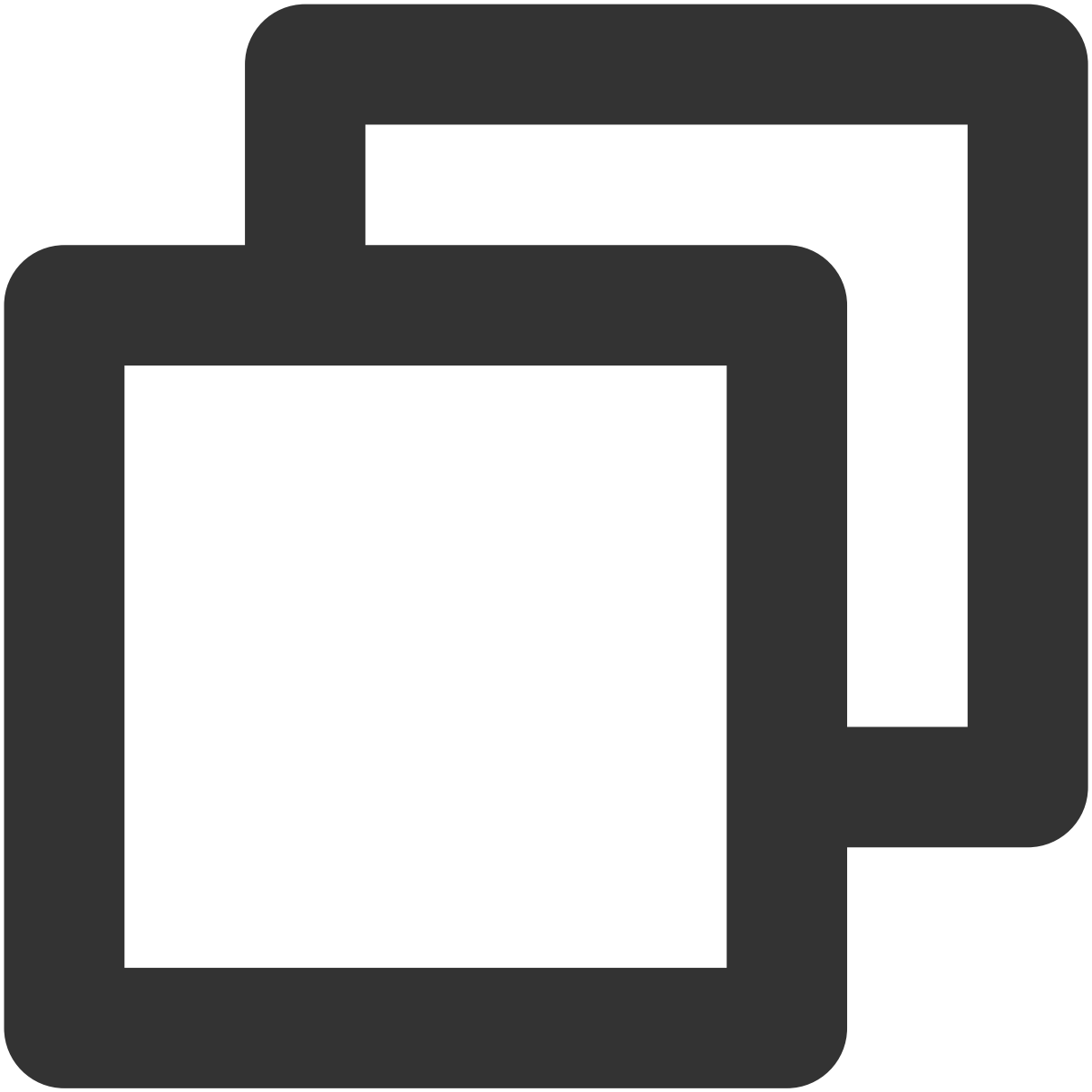
单聊自定义消息

高级接口

调用高级接口发送单聊自定义消息分两步：

1. 调用 `createCustomMessage` (点击查看详情) 创建自定义消息。
2. 调用 `sendMessage` (点击查看详情) 发送消息。

示例代码如下：



```
// 创建自定义消息
V2TimValueCallback<V2TimMsgCreateInfoResult> createCustomMessageRes = await Tencent
    data: '自定义data',
    desc: '自定义desc',
    extension: '自定义extension',
  );
  if(createCustomMessageRes.code == 0){
    String id =  createCustomMessageRes.data.id;
    // 发送自定义消息
    V2TimValueCallback<V2TimMessage> sendMessageRes = await TencentImSDKPlugin.v2TI
    if(sendMessageRes.code == 0){
```

```
        // 发送成功
      }
    }
```

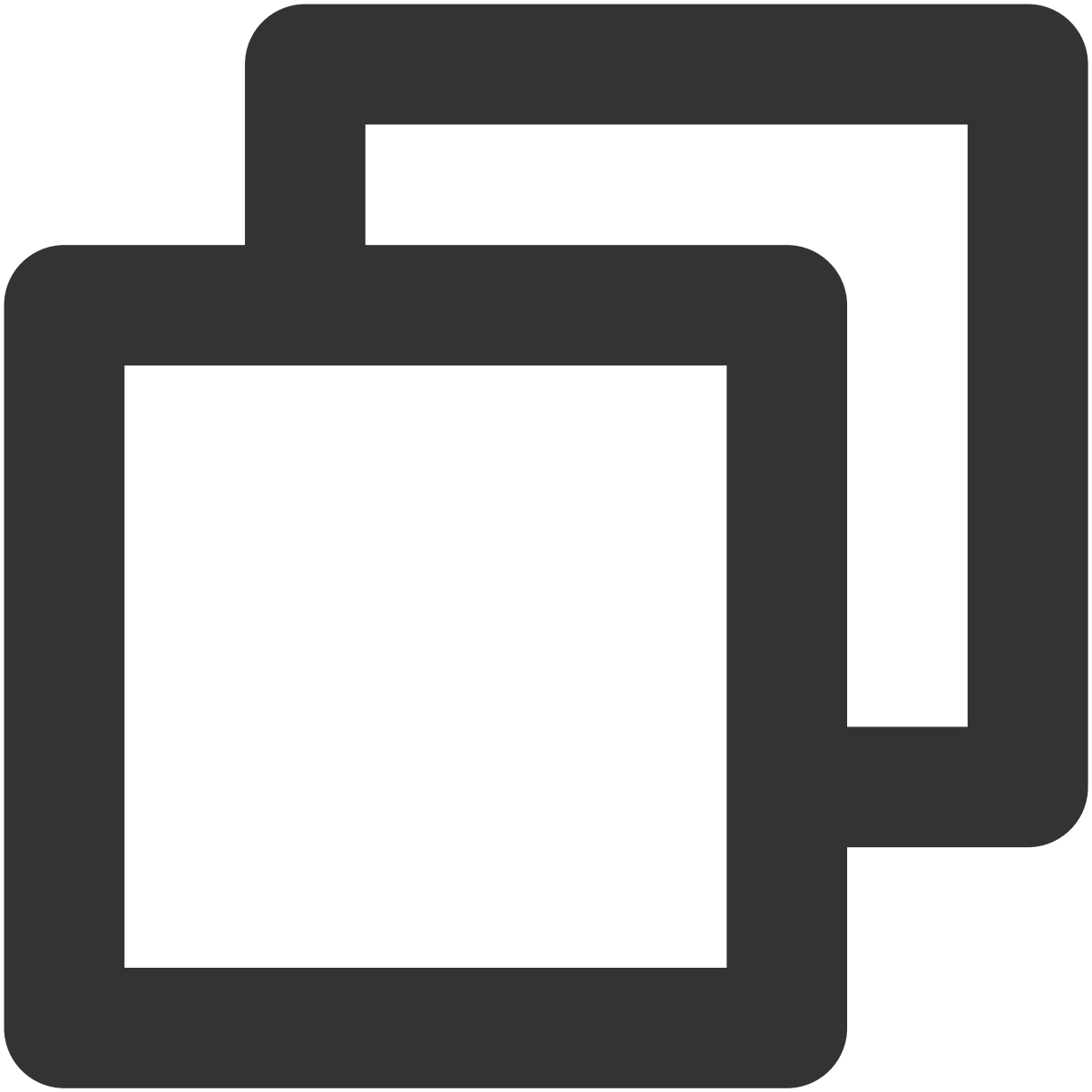
群聊自定义消息

高级接口

调用高级接口发送群聊自定义消息分两步：

1. 调用 `createCustomMessage` (点击查看详情) 创建自定义消息。
2. 调用 `sendMessage` (点击查看详情) 发送消息。

代码示例如下：



```
// 创建自定义消息
V2TimValueCallback<V2TimMsgCreateInfoResult> createCustomMessageRes = await Tencent
    data: '自定义data',
    desc: '自定义desc',
    extension: '自定义extension',
  );
  if(createCustomMessageRes.code == 0){
    String id =  createCustomMessageRes.data.id;
    // 发送自定义消息
    V2TimValueCallback<V2TimMessage> sendMessageRes = await TencentImSDKPlugin.v2TI
    if(sendMessageRes.code == 0){
```

```
        // 发送成功
      }
    }
```

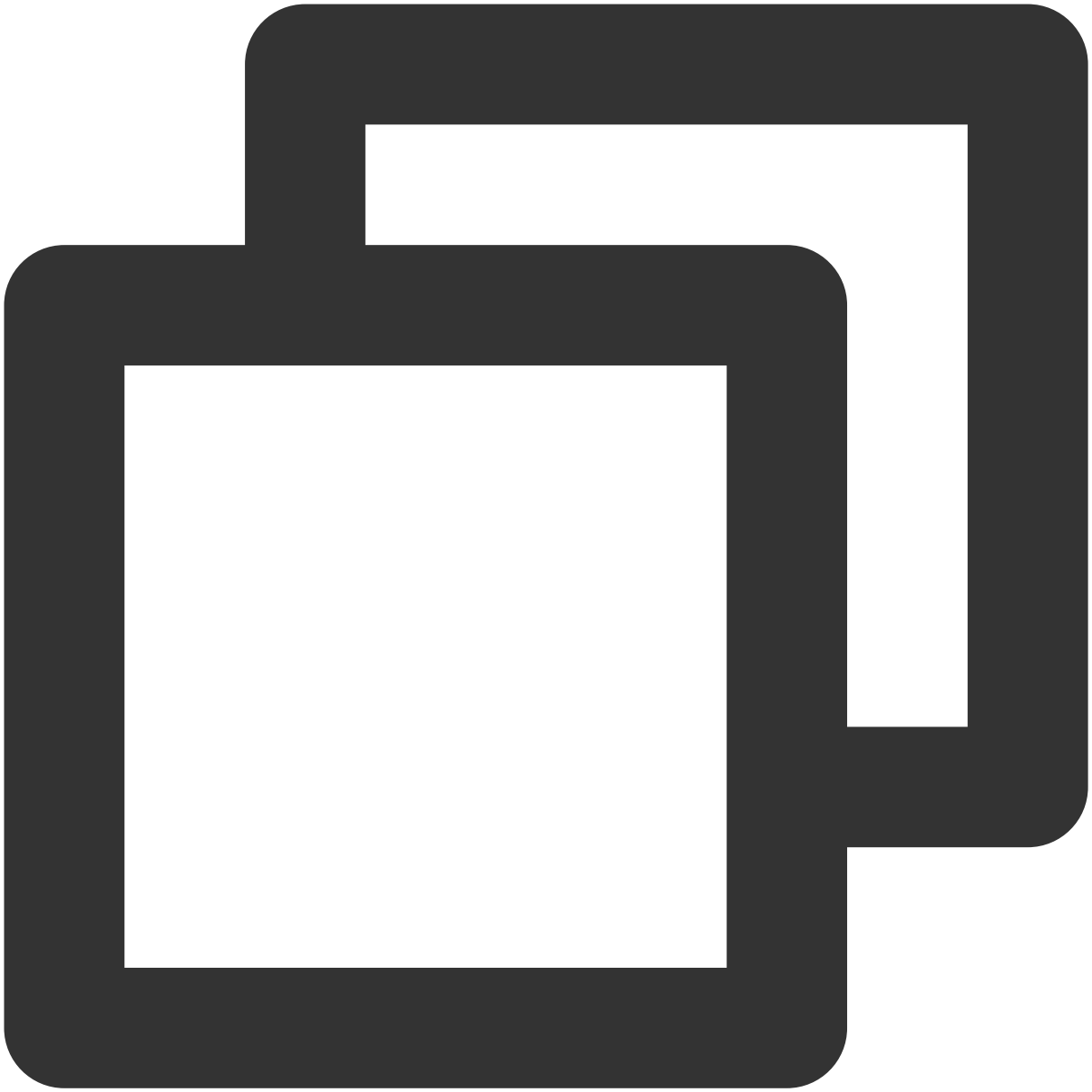
发送富媒体消息

富媒体消息发送没有普通接口，都需要使用高级接口，步骤是：

1. 调用 `createXxxMessage` 创建指定类型的富媒体消息对象，其中 Xxx 表示具体的消息类型。
2. 调用 `sendMessage` (点击查看详情) 发送消息。
3. 在消息回调中获取消息是否发送成功或失败。

图片消息

创建图片消息需要先获取到本地图片路径。
发送消息过程中，会先将图片文件上传至服务器，同时回调上传进度。上传成功后再发送消息。
如果您的项目需要支持Web，请查看Web兼容说明章节，发送图片的方式与移动端有不一致之处。
示例代码如下：

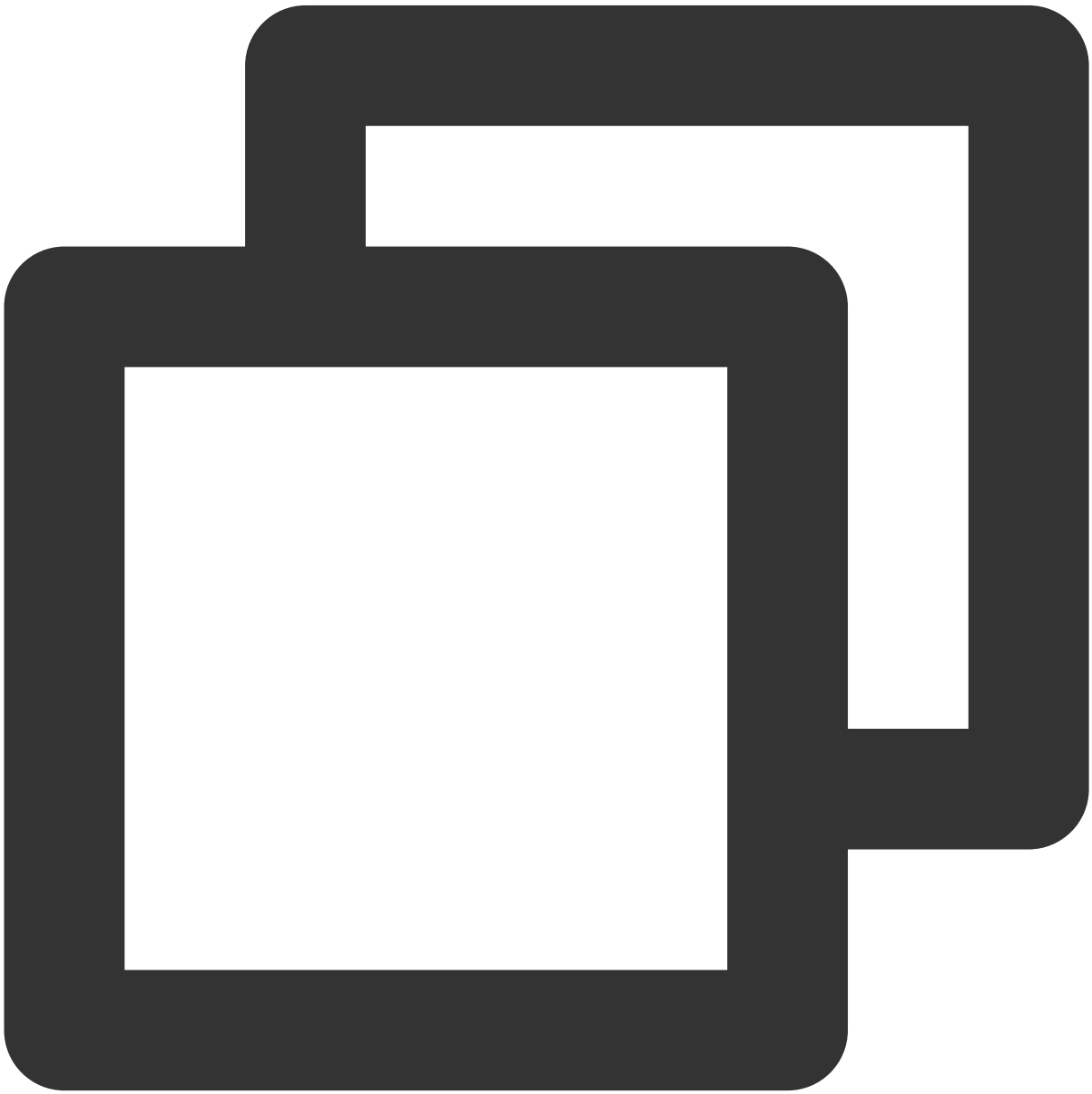


```
V2TimValueCallback<V2TimMsgCreateInfoResult> createImageMessageRes = await TencentI
        imagePath: "本地图片绝对路径",
);
  if (createImageMessageRes.code == 0) {
    String id = createImageMessageRes.data.id;
    V2TimValueCallback<V2TimMessage> sendMessageRes = await TencentImSDKPlugin
        .v2TIMManager
        .getMessageManager()
        .sendMessage(id: id, receiver: "userID", groupID: "groupID");
    if (sendMessageRes.code == 0) {
      // 发送成功
```

```
    }
  }
```

语音消息

创建语音消息需要先获取到本地语音文件路径和语音时长，其中语音时长可用于接收端 UI 显示。
发送消息过程中，会先将语音文件上传至服务器，同时回调上传进度。上传成功后再发送消息。
示例代码如下：

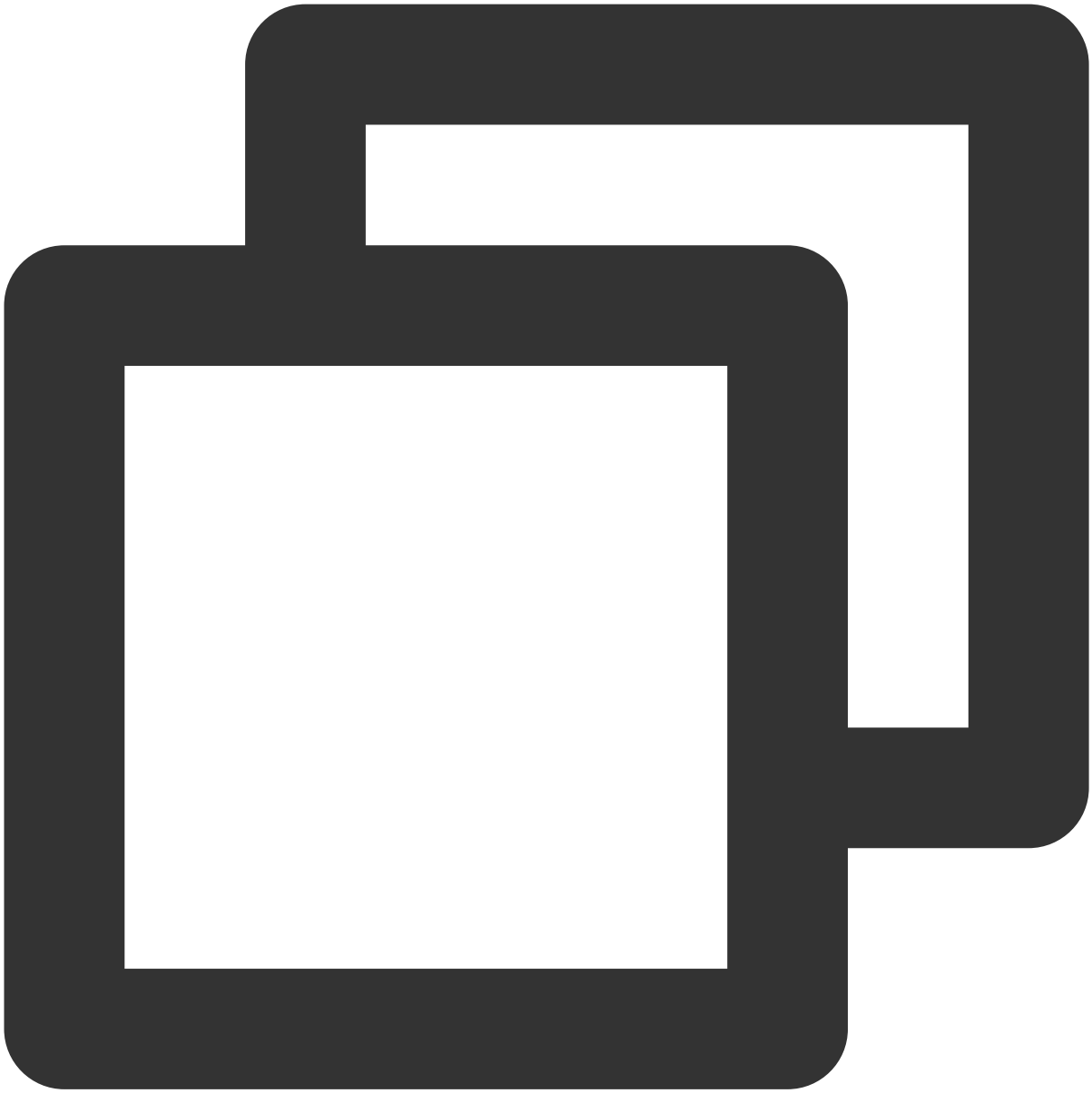


```
V2TimValueCallback<V2TimMsgCreateInfoResult> createSoundMessageRes =
      await TencentImSDKPlugin.v2TIMManager.getMessageManager().createSoundMessage(
```

```
        soundPath: "本地录音文件绝对路径",
        duration: 10,// 录音时长
      );
  if (createSoundMessageRes.code == 0) {
    String id = createSoundMessageRes.data.id;
    V2TimValueCallback<V2TimMessage> sendMessageRes = await TencentImSDKPlugin
        .v2TIMManager
        .getMessageManager()
        .sendMessage(id: id, receiver: "userID", groupID: "groupID");
    if (sendMessageRes.code == 0) {
      // 发送成功
    }
  }
```

视频消息

创建视频消息需要先获取到本地视频文件路径、视频时长和视频快照，其中时长和快照可用于接收端 UI 显示。
发送消息过程中，会先将视频上传至服务器，同时回调上传进度。上传成功后再发送消息。
如果您的项目需要支持Web，请查看Web兼容说明章节，发送视频的方式与移动端有不一致之处。
示例代码如下：



```
V2TimValueCallback<V2TimMsgCreateInfoResult> createVideoMessageRes =
      await TencentImSDKPlugin.v2TIMManager
          .getMessageManager()
          .createVideoMessage(
            videoFilePath: "本地视频文件绝对路径",
            type: "mp4", // 视频类型
            duration: 10,// 视频时长
            snapshotPath: "本地视频封面文件绝对路径",
          );
  if (createVideoMessageRes.code == 0) {
    String id = createVideoMessageRes.data.id;
```

```
    V2TimValueCallback<V2TimMessage> sendMessageRes = await TencentImSDKPlugin
        .v2TIMManager
        .getMessageManager()
        .sendMessage(id: id, receiver: "userID", groupID: "groupID");
    if (sendMessageRes.code == 0) {
      // 发送成功
    }
  }
```

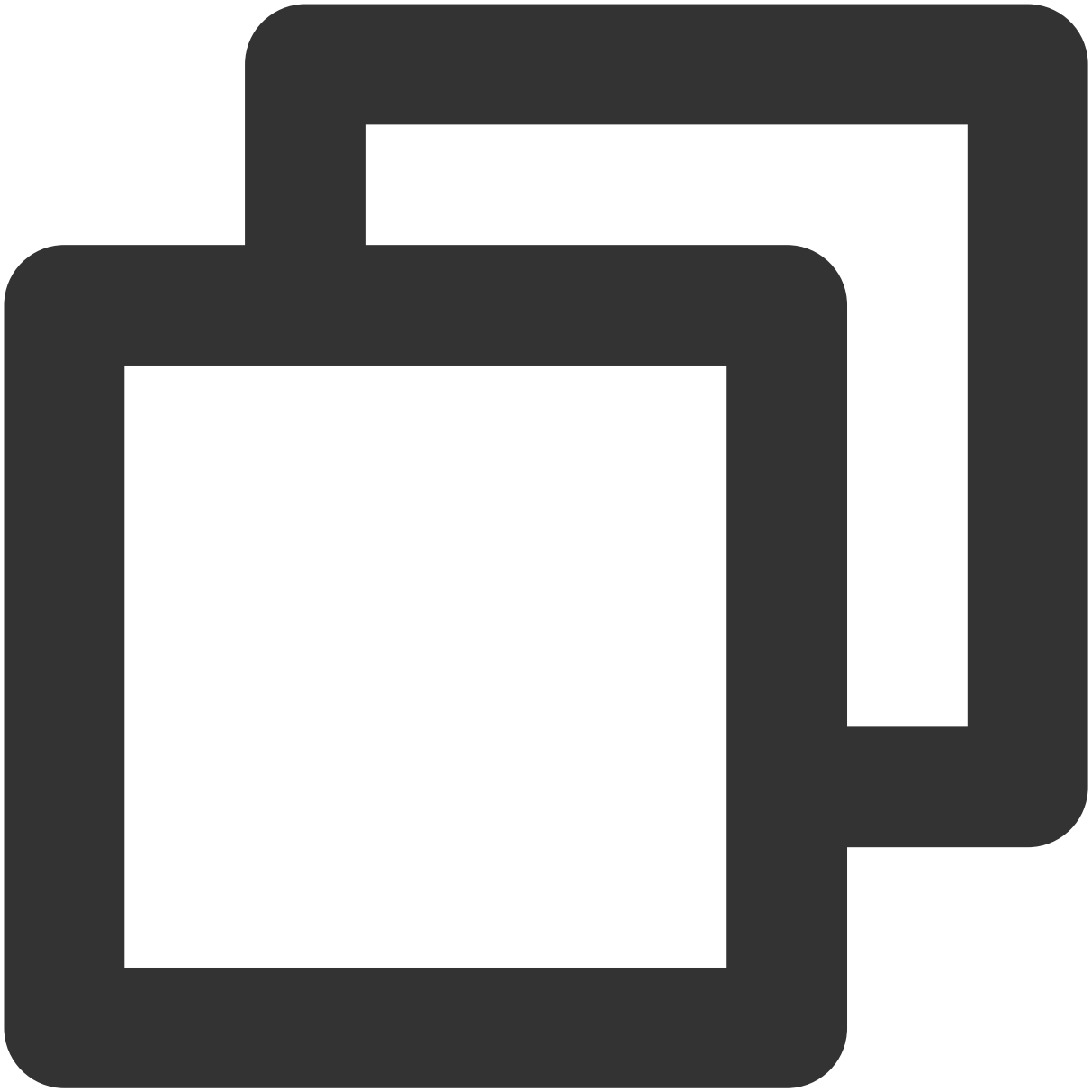
文件消息

创建文件消息需要先获取到本地文件路径。

发送消息过程中，会先将文件上传至服务器，同时回调上传进度。上传成功后再发送消息。

如果您的项目需要支持Web，请查看Web兼容说明章节，发送文件的方式与移动端有不一致之处。

示例代码如下：

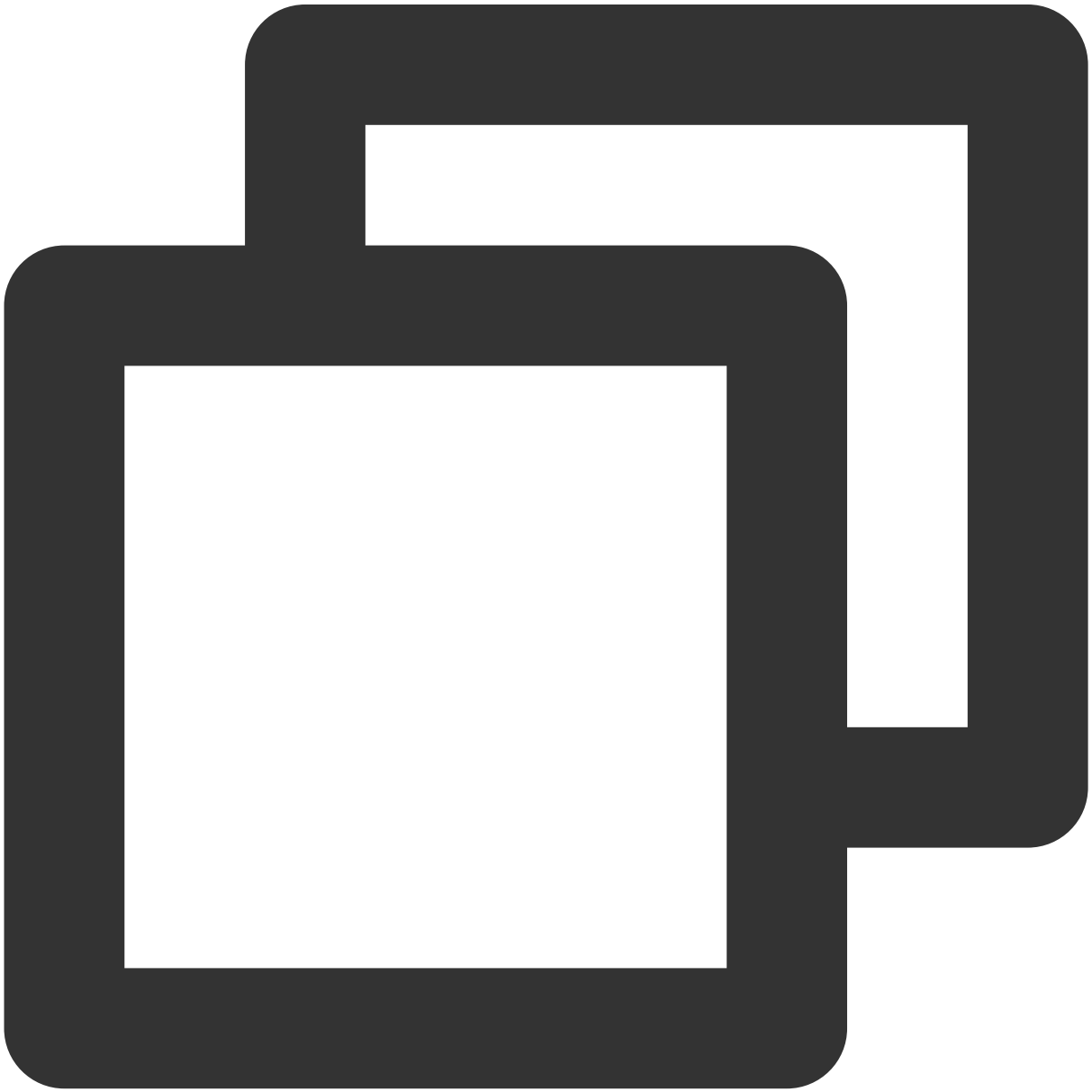


```
V2TimValueCallback<V2TimMsgCreateInfoResult> createFileMessageRes =
      await TencentImSDKPlugin.v2TIMManager
          .getMessageManager()
          .createFileMessage(
            filePath: "本地文件绝对路径",
            fileName: "文件名",
          );
  if (createFileMessageRes.code == 0) {
    String id = createFileMessageRes.data.id;
    V2TimValueCallback<V2TimMessage> sendMessageRes = await TencentImSDKPlugin
        .v2TIMManager
```

```
        .getMessageManager()
        .sendMessage(id: id, receiver: "userID", groupID: "groupID");
    if (sendMessageRes.code == 0) {
      // 发送成功
    }
  }
```

位置消息

位置消息会直接发送经纬度，一般需要配合地图控件显示。
示例代码如下：

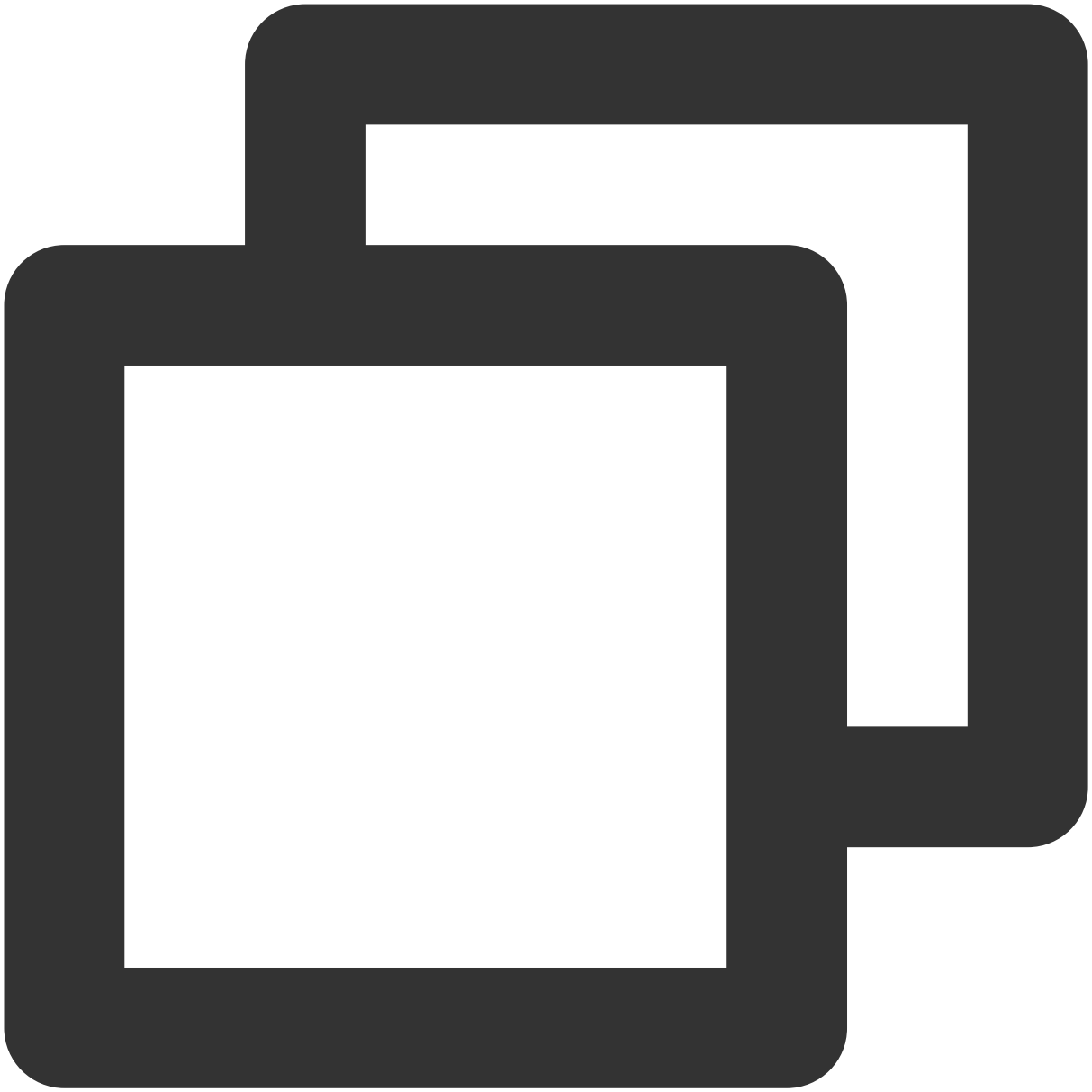


```
V2TimValueCallback<V2TimMsgCreateInfoResult> createLocationMessage =
     await TencentImSDKPlugin.v2TIMManager
         .getMessageManager()
         .createLocationMessage(
           desc: "深圳市南山区深南大道",//位置信息摘要
           longitude: 34,// 经度
           latitude: 20, // 纬度
         );
 if (createLocationMessage.code == 0) {
   String id = createLocationMessage.data.id;
   V2TimValueCallback<V2TimMessage> sendMessageRes = await TencentImSDKPlugin
       .v2TIMManager
       .getMessageManager()
       .sendMessage(id: id, receiver: "userID", groupID: "groupID");
   if (sendMessageRes.code == 0) {
     // 发送成功
   }
 }
```

表情消息

表情消息会直接发送表情编码，通常接收端需要将其转换成对应的表情 icon。

示例代码如下：



```
V2TimValueCallback<V2TimMsgCreateInfoResult> createFaceMessageRes =
      await TencentImSDKPlugin.v2TIMManager
          .getMessageManager()
          .createFaceMessage(
            index: 0,
            data: "",
          );
  if (createFaceMessageRes.code == 0) {
    String id = createFaceMessageRes.data.id;
    V2TimValueCallback<V2TimMessage> sendMessageRes = await TencentImSDKPlugin
        .v2TIMManager
```

```
        .getMessageManager()
        .sendMessage(id: id, receiver: "userID", groupID: "groupID");
    if (sendMessageRes.code == 0) {
      // 发送成功
    }
  }
```

Flutter for Web支持

IM SDK(tencent_cloud_chat_sdk) 4.1.1+2版本起，可完美兼容 Web 端。相比 Android 和 iOS 端，发送媒体及文件，步骤有不一致之处。如下：

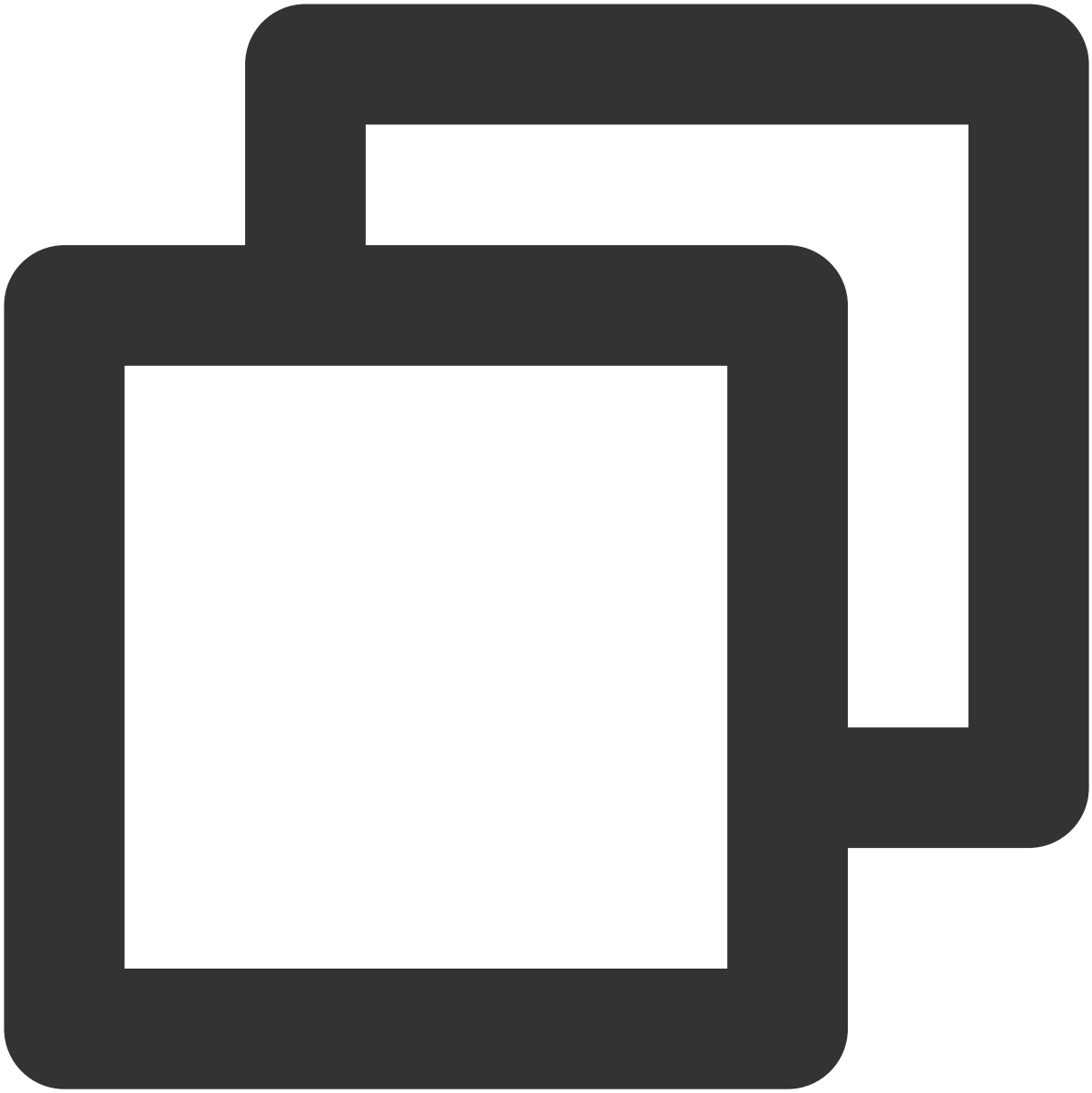
由于 Web 特性，创建媒体及文件消息时，无法直接传入路径至 SDK。需要根据 Element ID 获取 input 的 DOM 节点，将选择文件后的 input DOM 传入。

选取媒体建议使用 image_picker 包。

选取文件建议使用 file_picker 包。

示例代码中 `getElementById` 的值，若和F12控制台看到 input 的 id 不一致，请以实际为准。

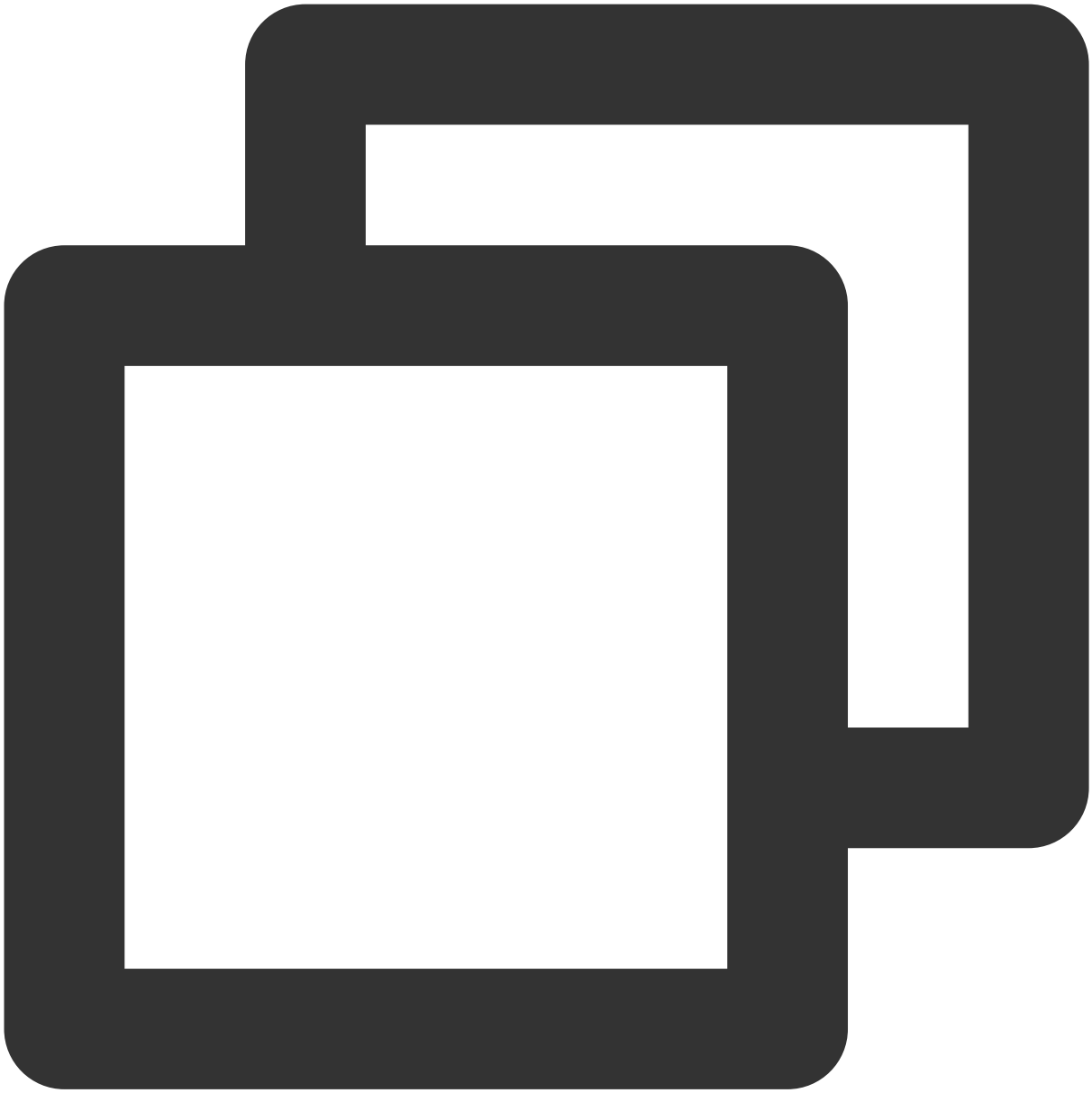
发送图片



```
final ImagePicker _picker = ImagePicker();

_sendImageFileOnWeb() async {
  final pickedFile = await _picker.pickImage(source: ImageSource.gallery);
  final imageContent = await pickedFile!.readAsBytes();
  fileName = pickedFile.name;
  tempFile = File(pickedFile.path);
  fileContent = imageContent;

  html.Node? inputElem;
  inputElem = html.document
```

```
        .getElementById("__image_picker_web-file-input")
        ?.querySelector("input");
    final convID = widget.conversationID;
    final convType =
    widget.conversationType == 1 ? ConvType.c2c : ConvType.group;
    final createImageMessageRes = await TencentImSDKPlugin.v2TIMManager
      .getMessageManager()
      .createImageMessage(inputElement: inputElement);
    if (createImageMessageRes.code == 0) {
      String id = createImageMessageRes.data.id;
      V2TimValueCallback<V2TimMessage> sendMessageRes = await TencentImSDKPlugin
          .v2TIMManager
          .getMessageManager()
          .sendMessage(id: id, receiver: "userID", groupID: "groupID");
      if (sendMessageRes.code == 0) {
        // 发送成功
      }
    }
  }
```

发送视频

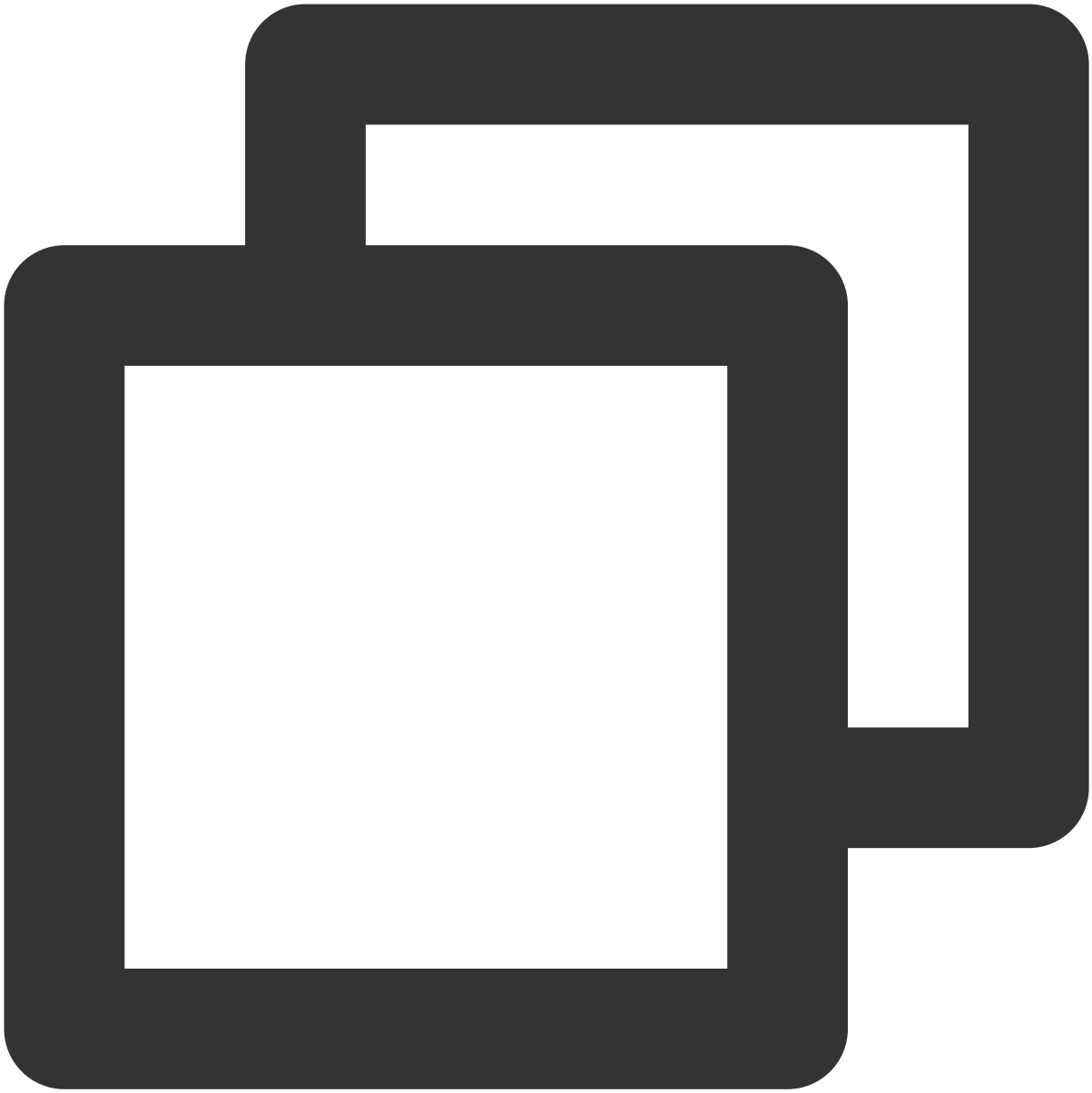


```
final ImagePicker _picker = ImagePicker();

_sendVideoFileOnWeb() async {
  final pickedFile = await _picker.pickVideo(source: ImageSource.gallery);
  final videoContent = await pickedFile!.readAsBytes();
  fileName = pickedFile.name ?? "";
  tempFile = File(pickedFile.path);
  fileContent = videoContent;

  if(fileName!.split(".")[fileName!.split(".").length - 1] != "mp4"){
    Toast.showToast("视频消息仅限 mp4 格式", context);
```

版权所有：腾讯云计算（北京）有限责任公司

```
    return;
  }

  html.Node? inputElem;
  inputElem = html.document
      .getElementById("__image_picker_web-file-input")
      ?.querySelector("input");
  final convID = widget.conversationID;
  final convType =
  widget.conversationType == 1 ? ConvType.c2c : ConvType.group;
  final createVideoMessageRes = await TencentImSDKPlugin.v2TIMManager
    .getMessageManager()
    .createVideoMessage(inputElement: inputElement, videoFilePath: "", type: "", du
  if (createVideoMessageRes.code == 0) {
    String id = createVideoMessageRes.data.id;
    V2TimValueCallback<V2TimMessage> sendMessageRes = await TencentImSDKPlugin
        .v2TIMManager
        .getMessageManager()
        .sendMessage(id: id, receiver: "userID", groupID: "groupID");
    if (sendMessageRes.code == 0) {
      // 发送成功
    }
  }
}
```

发送文件



```
_sendFileOnWeb(){
  final convID = widget.conversationID;
  final convType =
      widget.conversationType == 1 ? ConvType.c2c : ConvType.group;
  FilePickerResult? result = await FilePicker.platform.pickFiles();
  if (result != null && result.files.isNotEmpty) {
    html.Node? inputElem;
    inputElem = html.document
        .getElementById("__file_picker_web-file-input")
        ?.querySelector("input");
    fileName = result.files.single.name;
```

```
    final createFileMessageRes = await TencentImSDKPlugin.v2TIMManager
        .getMessageManager()
        .createFileMessage(inputElement: inputElement, filePath: "", fileName: file
    if (createFileMessageRes.code == 0) {
    String id = createFileMessageRes.data.id;
    V2TimValueCallback<V2TimMessage> sendMessageRes = await TencentImSDKPlugin
        .v2TIMManager
        .getMessageManager()
        .sendMessage(id: id, receiver: "userID", groupID: "groupID");
    if (sendMessageRes.code == 0) {
      // 发送成功
    }
  }
  }
}
```

Unity

最近更新时间：2024-01-31 15:51:55

功能描述

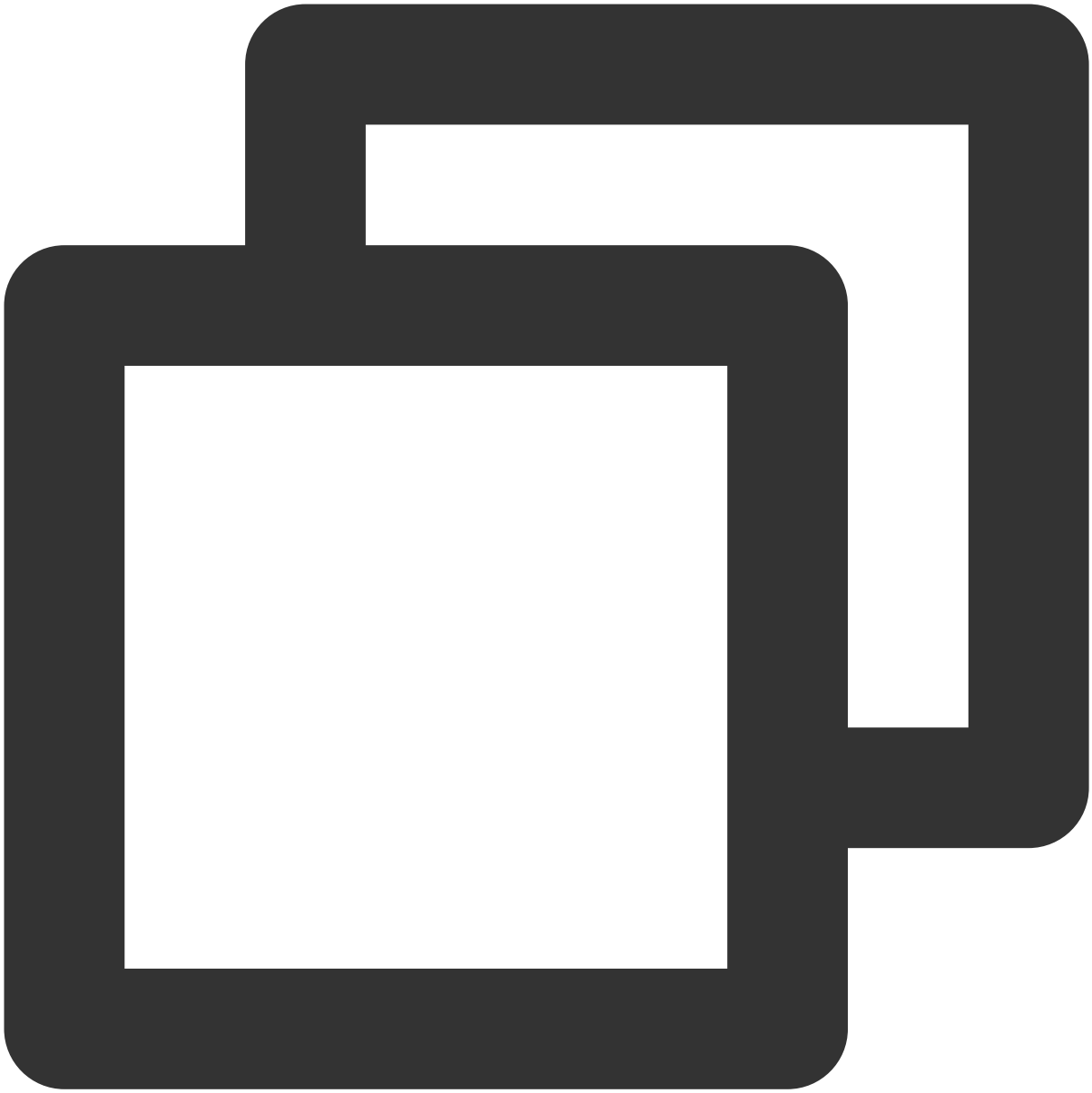
支持发送文本、自定义、富媒体消息，消息类型都是 `Message` 。

重点接口说明

接口 `MsgSendMessage` (点击查看详情) 是发送消息中最核心的接口。该接口支持发送所有类型的消息。
接口说明如下：

Type	Name	Description
System.String	conv_id	会话ID
TIMConvType	conv_type	会话类型 TIMConvType
Message	message	消息体 Message
System.Text.StringBuilder	message_id	承接消息 ID 的 StringBuilder
ValueCallback | ValueCallback	callback	异步回调

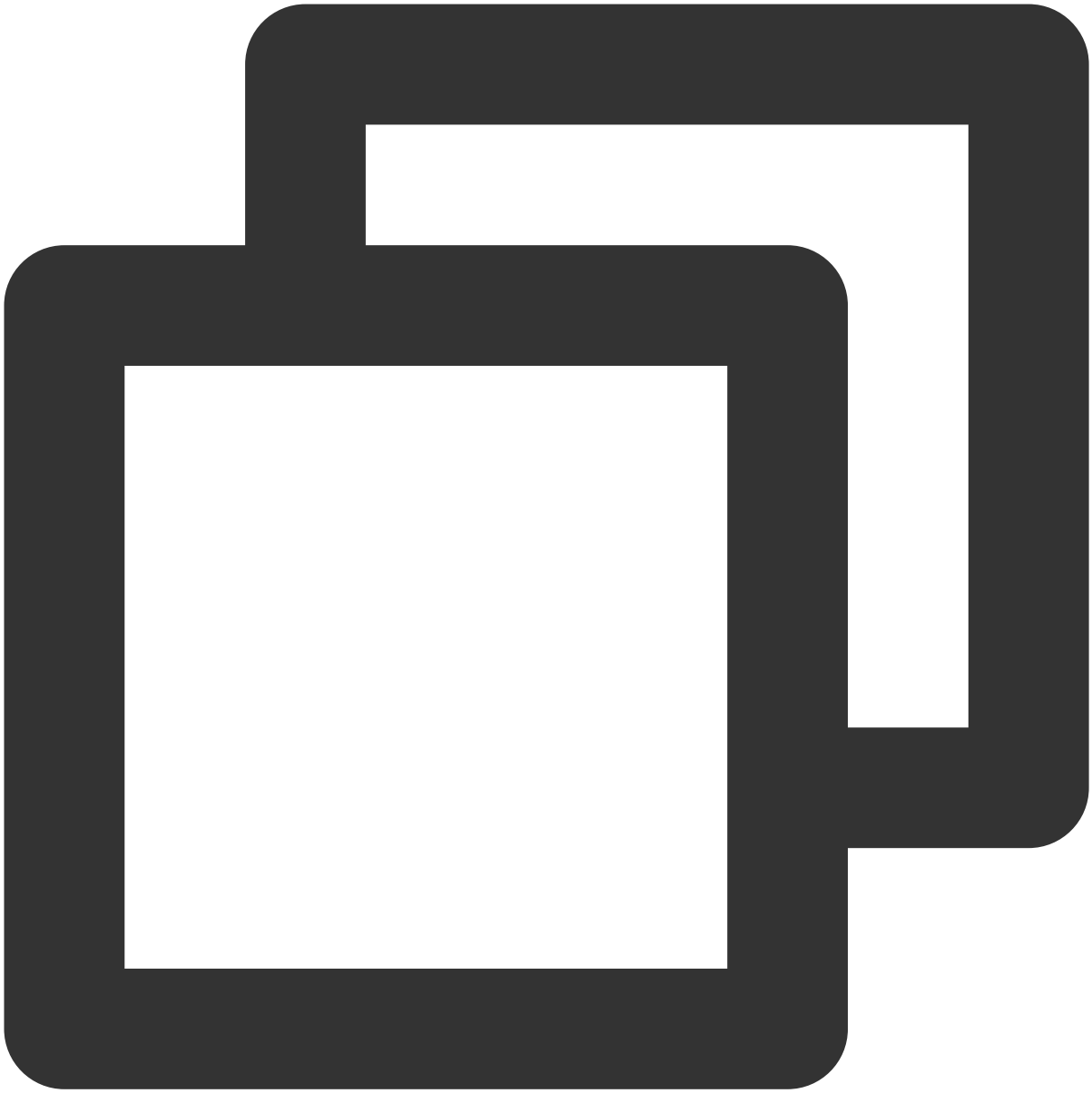
发送文本消息



```
public static void MsgSendMessage() {
        string conv_id = ""; // c2c 消息会话 ID 为 userID, 群消息会话 ID 为 groupID
        Message message = new Message
        {
          message_conv_id = conv_id,
          message_conv_type = TIMConvType.kTIMConv_C2C, // 群消息为TIMConvType.kTIMC
          message_elem_array = new List<Elem>
          {
            new Elem
            {
              elem_type = TIMElemType.kTIMElem_Text,
```

```
                text_elem_content =  "这是一个普通文本消息"
              }
            }
          };
          StringBuilder messageId = new StringBuilder(128);

          TIMResult res = TencentIMSDK.MsgSendMessage(conv_id, TIMConvType.kTIMConv_C
            // 消息发送异步结果
          });
                // 消息发送同步返回的消息ID messageId
}
```

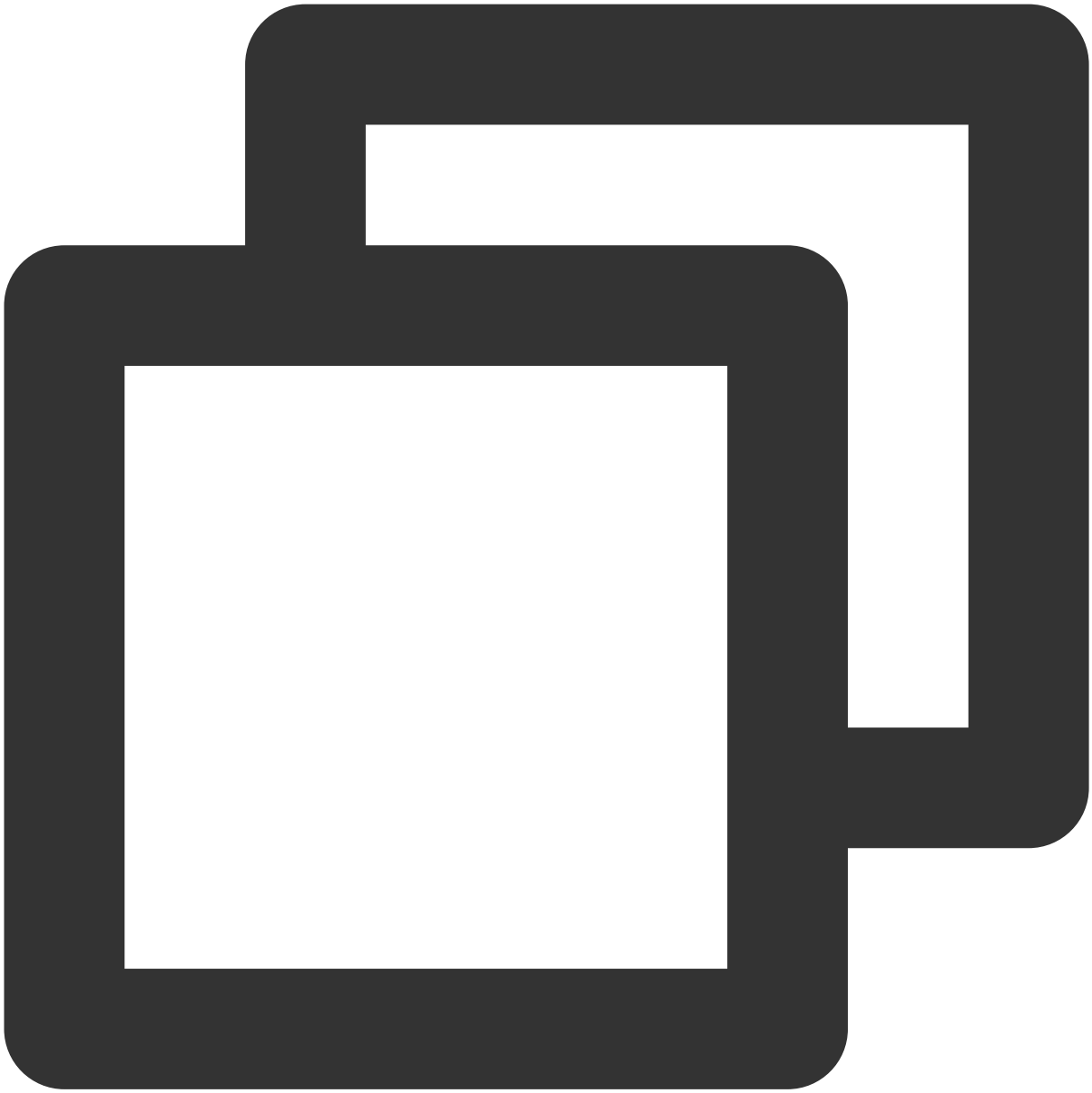
发送图片消息



```
public static void MsgSendMessage() {
        string conv_id = ""; // c2c 消息会话 ID 为 userID, 群消息会话 ID 为 groupID
        Message message = new Message
        {
          message_conv_id = conv_id,
          message_conv_type = TIMConvType.kTIMConv_C2C, // 群消息为TIMConvType.kTIMC
          message_elem_array = new List<Elem>
          {
            new Elem
            {
              elem_type = TIMElemType.kTIMElem_Image,
```

```
                image_elem_orig_path =  "/Users/xxx/xxx.png", // 文件绝对路径
                image_elem_level = TIMImageLevel.kTIMImageLevel_Orig // 原图发送
              }
            }
          };
          StringBuilder messageId = new StringBuilder(128);

          TIMResult res = TencentIMSDK.MsgSendMessage(conv_id, TIMConvType.kTIMConv_C
            // 消息发送异步结果
          });
                // 消息发送同步返回的消息ID messageId
}
```

发送语音消息

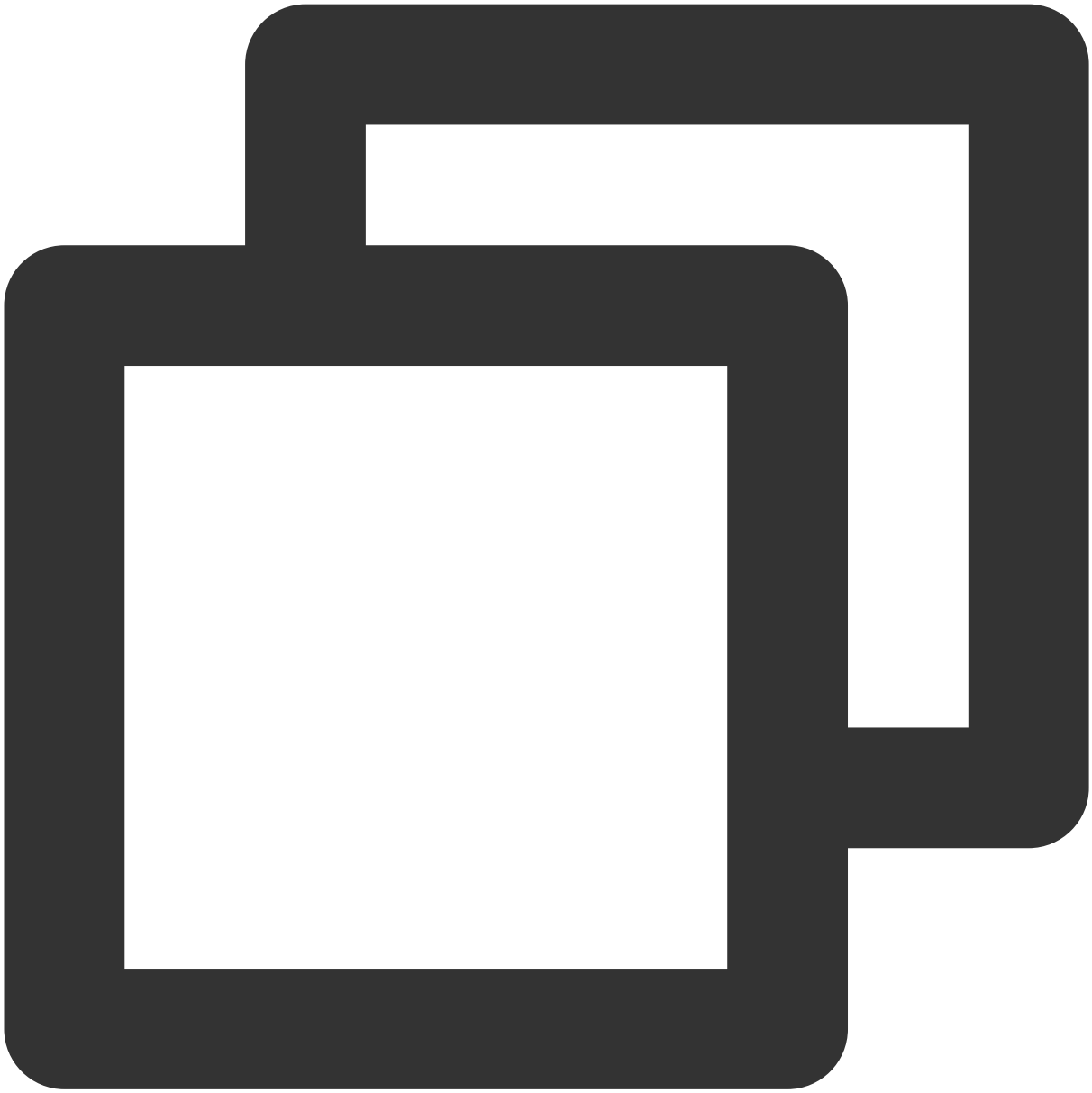


```
public static void MsgSendMessage() {
        string conv_id = ""; // c2c 消息会话 ID 为 userID, 群消息会话 ID 为 groupID
        Message message = new Message
        {
          message_conv_id = conv_id,
          message_conv_type = TIMConvType.kTIMConv_C2C, // 群消息为TIMConvType.kTIMC
          message_elem_array = new List<Elem>
          {
            new Elem
            {
              elem_type = TIMElemType.kTIMElem_Sound,
```

版权所有：腾讯云计算（北京）有限责任公司

```
                sound_elem_file_path =  "/Users/xxx/xxx.mp3", // 文件绝对路径
                sound_elem_file_size = 10  // 语音时长

            }
          }
        };
        StringBuilder messageId = new StringBuilder(128);

        TIMResult res = TencentIMSDK.MsgSendMessage(conv_id, TIMConvType.kTIMConv_C
          // 消息发送异步结果
        });
              // 消息发送同步返回的消息ID messageId
}
```

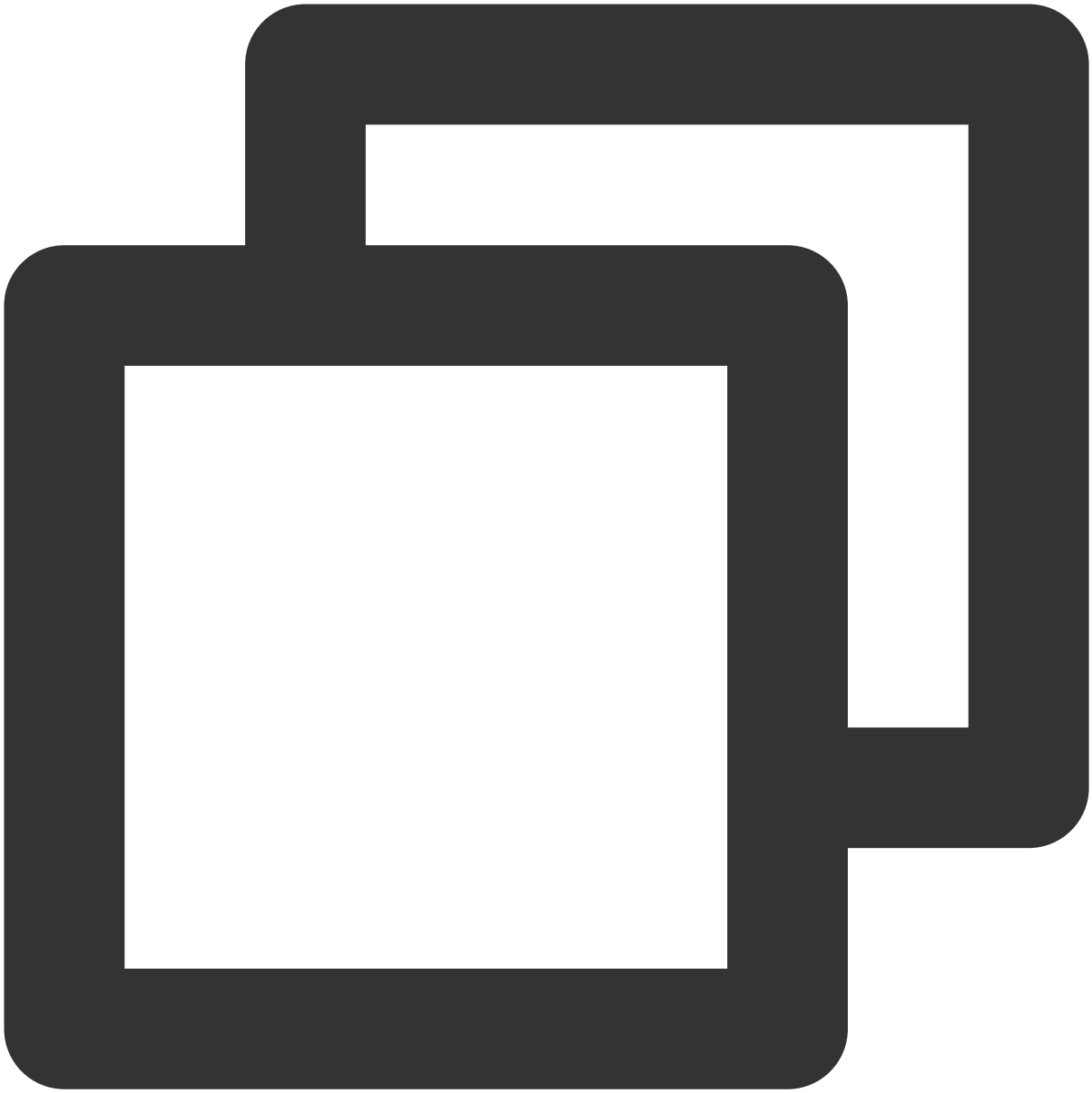
发送视频消息



```
public static void MsgSendMessage() {
        string conv_id = ""; // c2c 消息会话 ID 为 userID, 群消息会话 ID 为 groupID
        Message message = new Message
        {
          message_conv_id = conv_id,
          message_conv_type = TIMConvType.kTIMConv_C2C, // 群消息为TIMConvType.kTIMC
          message_elem_array = new List<Elem>
          {
            new Elem
            {
              elem_type = TIMElemType.kTIMElem_Video,
```

```
                video_elem_video_path =  "/Users/xxx/xxx.mp4", // 文件绝对路径
                video_elem_video_type = "mp4",  // 视频类型
                video_elem_video_duration = 10, // 视频时长

                video_elem_image_path = "本地视频封面文件绝对路径",
                video_elem_image_type = "png", // 视频截图文件类型
                video_elem_image_size = 100, // 视频截图文件大小
                video_elem_image_width = 1920, // 视频截图文件宽
                video_elem_image_height = 1080, // 视频截图文件高
            }
          }
        };
        StringBuilder messageId = new StringBuilder(128);

        TIMResult res = TencentIMSDK.MsgSendMessage(conv_id, TIMConvType.kTIMConv_C
          // 消息发送异步结果
        });
              // 消息发送同步返回的消息ID messageId
}
```

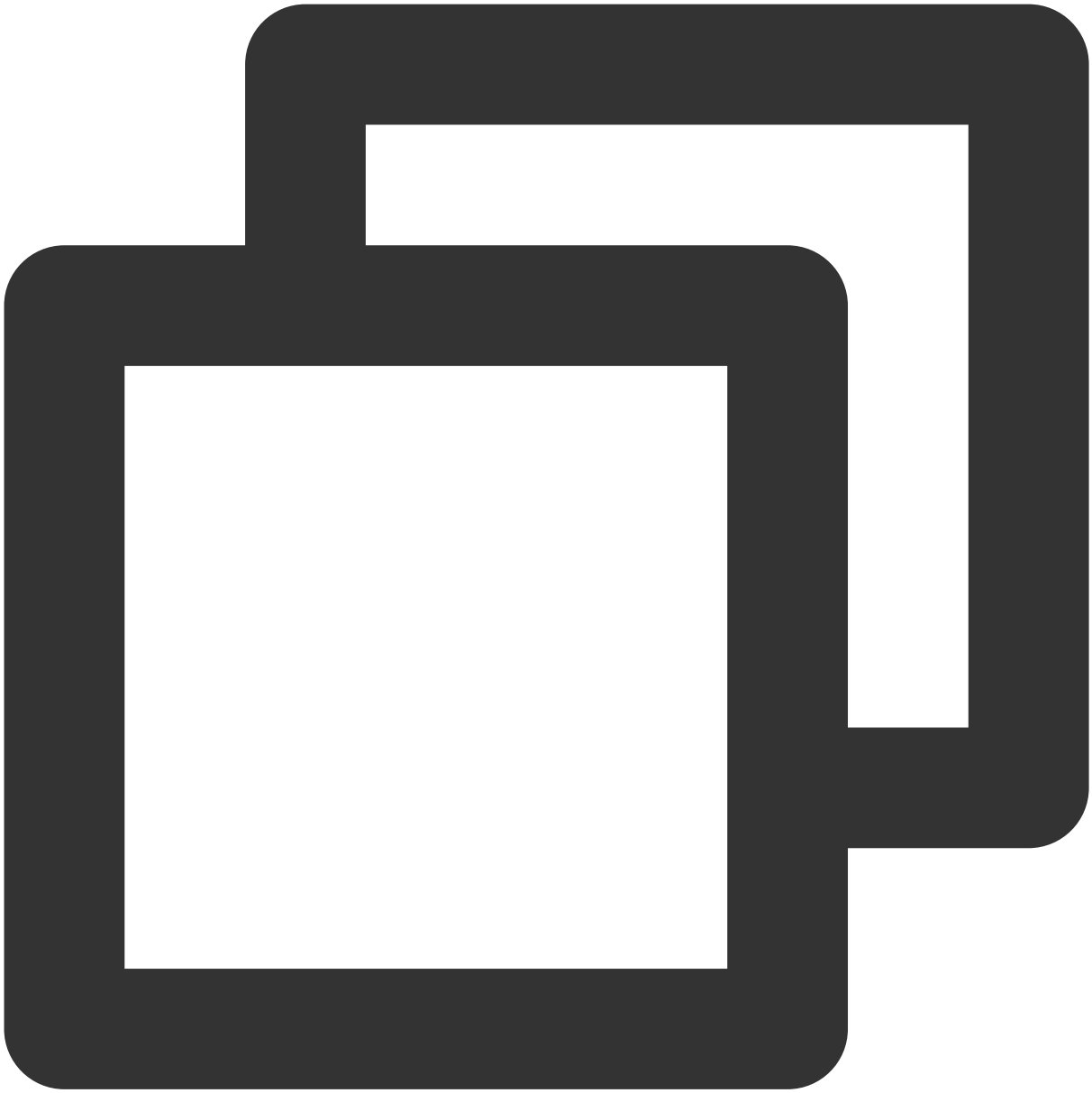
发送文件消息



```
public static void MsgSendMessage() {
        string conv_id = ""; // c2c 消息会话 ID 为 userID, 群消息会话 ID 为 groupID
        Message message = new Message
        {
          message_conv_id = conv_id,
          message_conv_type = TIMConvType.kTIMConv_C2C, // 群消息为TIMConvType.kTIMC
          message_elem_array = new List<Elem>
          {
            new Elem
            {
              elem_type = TIMElemType.kTIMElem_File,
```

```
                file_elem_file_path =  "/Users/xxx/xxx.x", // 文件绝对路径
                file_elem_file_name = "文件名",
            }
        }
    };
    StringBuilder messageId = new StringBuilder(128);

    TIMResult res = TencentIMSDK.MsgSendMessage(conv_id, TIMConvType.kTIMConv_C
      // 消息发送异步结果
    });
          // 消息发送同步返回的消息ID messageId
}
```

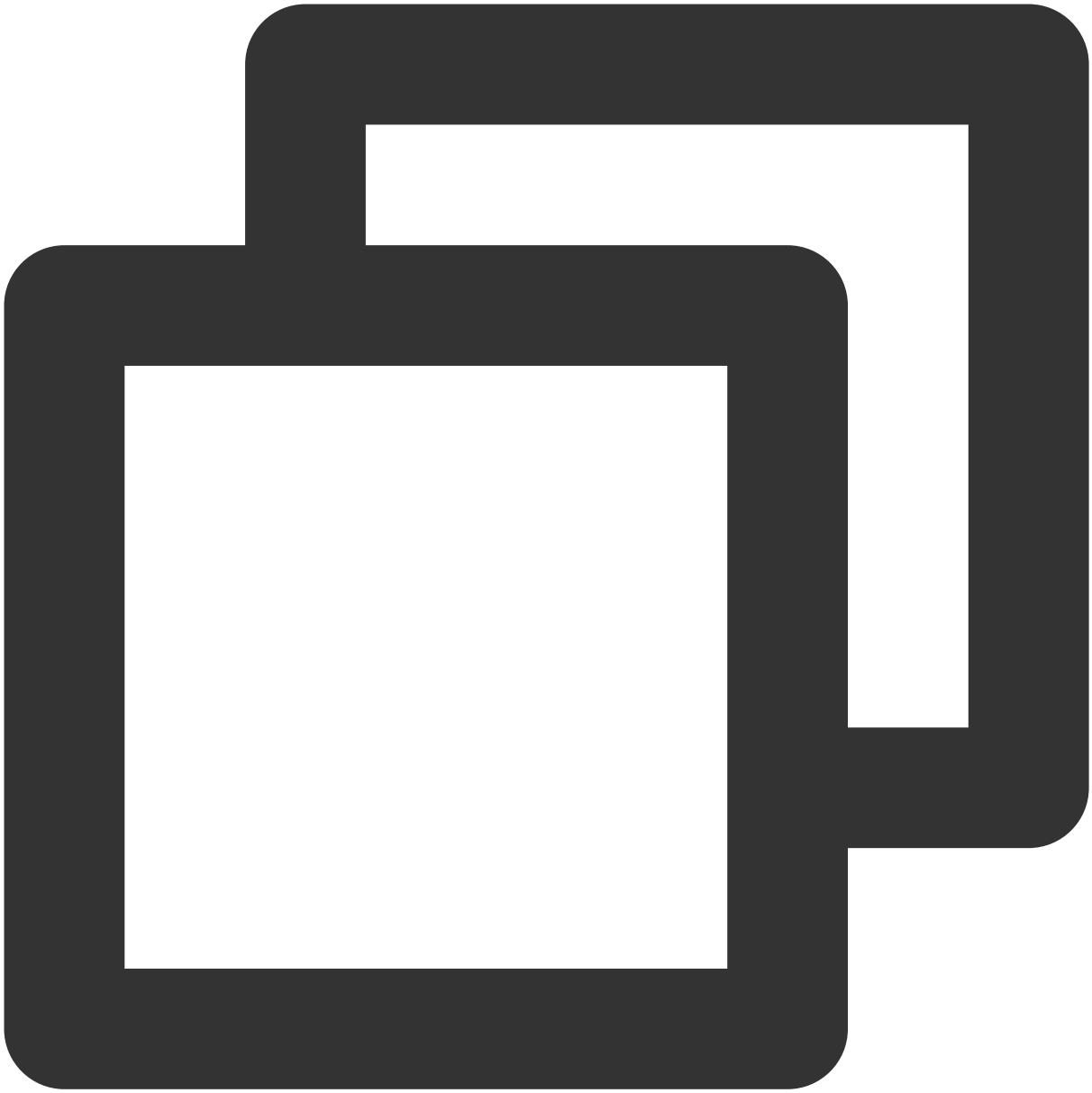
发送定位消息



```
public static void MsgSendMessage() {
        string conv_id = ""; // c2c 消息会话 ID 为 userID, 群消息会话 ID 为 groupID
        Message message = new Message
        {
          message_conv_id = conv_id,
          message_conv_type = TIMConvType.kTIMConv_C2C, // 群消息为TIMConvType.kTIMC
          message_elem_array = new List<Elem>
          {
            new Elem
            {
              elem_type = TIMElemType.kTIMElem_Location,
```

```
                location_elem_desc =  "深圳市南山区深南大道", // 位置信息摘要
                location_elem_longitude = 34, // 经度
                location_elem_latitude = 20 // 纬度
              }
            }
          };
          StringBuilder messageId = new StringBuilder(128);

          TIMResult res = TencentIMSDK.MsgSendMessage(conv_id, TIMConvType.kTIMConv_C
            // 消息发送异步结果
          });
                // 消息发送同步返回的消息ID messageId
  }
```

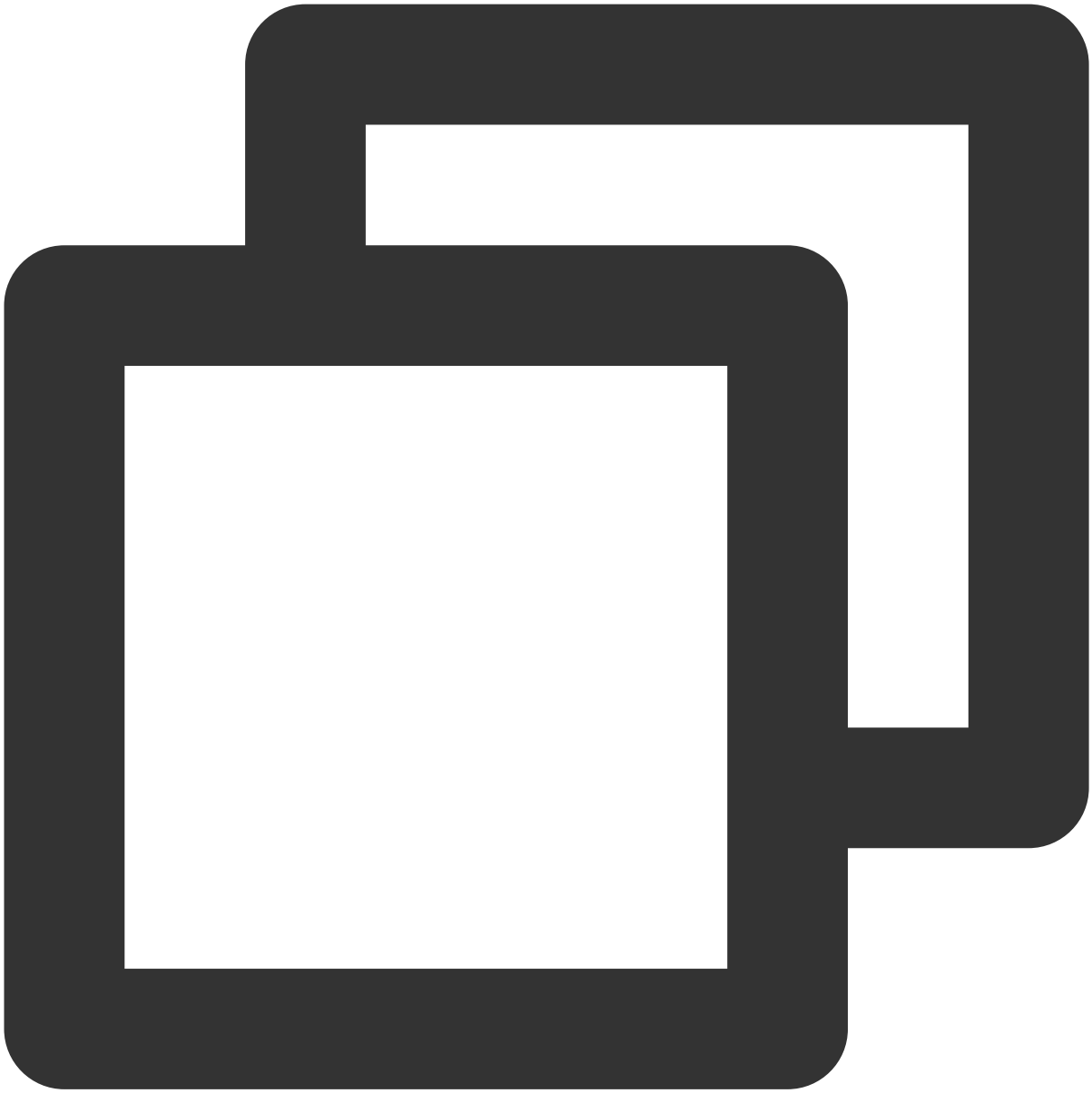
发送表情消息



```
public static void MsgSendMessage() {
        string conv_id = ""; // c2c 消息会话 ID 为 userID, 群消息会话 ID 为 groupID
        Message message = new Message
        {
          message_conv_id = conv_id,
          message_conv_type = TIMConvType.kTIMConv_C2C, // 群消息为TIMConvType.kTIMC
          message_elem_array = new List<Elem>
          {
            new Elem
            {
              elem_type = TIMElemType.kTIMElem_Face,
```

```
                face_elem_index = 0,
                face_elem_buf = ""
              }
            }
          };
          StringBuilder messageId = new StringBuilder(128);

          TIMResult res = TencentIMSDK.MsgSendMessage(conv_id, TIMConvType.kTIMConv_C
            // 消息发送异步结果
          });
                // 消息发送同步返回的消息ID messageId
}
```

发送自定义消息



```
public static void MsgSendMessage() {
        string conv_id = ""; // c2c 消息会话 ID 为 userID, 群消息会话 ID 为 groupID
        Message message = new Message
        {
          message_conv_id = conv_id,
          message_conv_type = TIMConvType.kTIMConv_C2C, // 群消息为TIMConvType.kTIMC
          message_elem_array = new List<Elem>
          {
            new Elem
            {
              elem_type = TIMElemType.kTIMElem_Custom,
```

```
                custom_elem_data = "",
                custom_elem_desc = "",
                custom_elem_ext = ""
              }
            }
          };
          StringBuilder messageId = new StringBuilder(128);

          TIMResult res = TencentIMSDK.MsgSendMessage(conv_id, TIMConvType.kTIMConv_C
            // 消息发送异步结果
          });
                // 消息发送同步返回的消息ID messageId
}
```

React Native

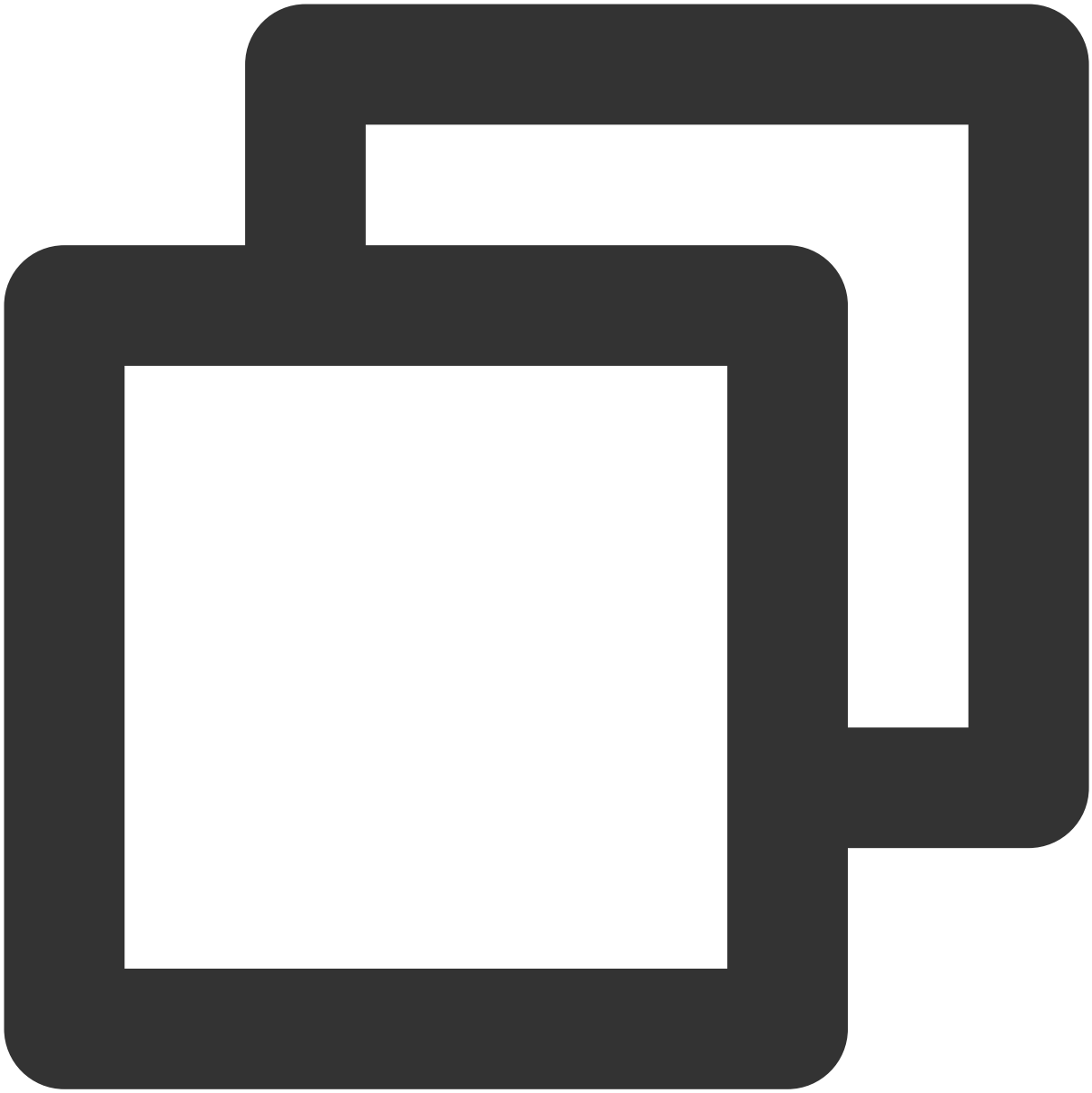
最近更新时间：2024-01-31 15:52:40

功能描述

发送消息方法在核心类 `TencentImSDKPlugin.v2TIMManager.getMessageManager()` 中。
支持发送文本、自定义、富媒体消息，消息类型都是 `V2TimMessage` 。
`V2TimMessage` 中可以携带不同类型子类，表示不同类型的消息。

重点接口说明

接口 `sendMessage` (Details) 是发送消息中最核心的接口。该接口支持发送所有类型的消息。
说明：
下文中提到的发消息高级接口，指的都是 `sendMessage` 。
接口说明如下：



```
public sendMessage({
        id,
        receiver,
        groupID,
        onlineUserOnly = false,
        isExcludedFromLastMessage = false,
        isExcludedFromUnreadCount = false,
        needReadReceipt = false,
        offlinePushInfo,
        cloudCustomData,
        localCustomData,
```

```
        priority = MessagePriorityEnum.V2TIM_PRIORITY_NORMAL,
    }: {
        id: string;
        receiver: string;
        groupID: string;
        onlineUserOnly?: boolean;
        isExcludedFromUnreadCount?: boolean;
        isExcludedFromLastMessage?: boolean;
        needReadReceipt?: boolean;
        offlinePushInfo?: V2TimOfflinePushInfo;
        cloudCustomData?: string;
        localCustomData?: string;
        priority?: MessagePriorityEnum;
    })
```

参数说明：

参数	含义	单聊有效	群聊有效	说明
id	创建消息返回的id	YES	YES	需要通过对应的 `createXxxMessage` 接口先行创建
receiver	单聊消息接收者 userID	YES	NO	如果是发送 C2C 单聊消息，只需要指定 receiver 即可
groupID	群聊 groupID	NO	YES	如果是发送群聊消息，只需要指定 groupID 即可
priority	消息优先级	NO	YES	请把重要消息设置为高优先级（例如红包、礼物消息），高频且不重要的消息设置为低优先级（例如点赞消息）
onlineUserOnly	是否只有在线用户才能收到	YES	YES	如果设置为 YES，接收方历史消息拉取不到，常被用于实现”对方正在输入”或群组里的非重要提示等弱提示功能
offlinePushInfo	离线推送信息	YES	YES	离线推送时携带的标题和内容
needReadReceipt	发送群消息是否支持已读	NO	YES	发送群消息是否支持已读
isExcludedFromUnreadCount	发送消息是否计入会话未读数	YES	YES	如果设置为 true，发送消息不会计入会话未读，默认为 false

isExcludedFromLastMessage	发送消息是否计入会话 lastMessage	YES	YES	如果设置为 true，发送消息不会计入会话 lastMessage，默认为 false
cloudCustomData	消息云端数据	YES	YES	消息附带的额外的数据，存云端，消息的接受者可以访问到
localCustomData	消息本地数据	YES	YES	消息附带的额外的数据，存本地，消息的接受者不可以访问到，App 卸载后数据丢失

说明：
如果 groupID 和 receiver 同时设置，表示给 receiver 发送定向群消息。具体请参考 群定向消息。

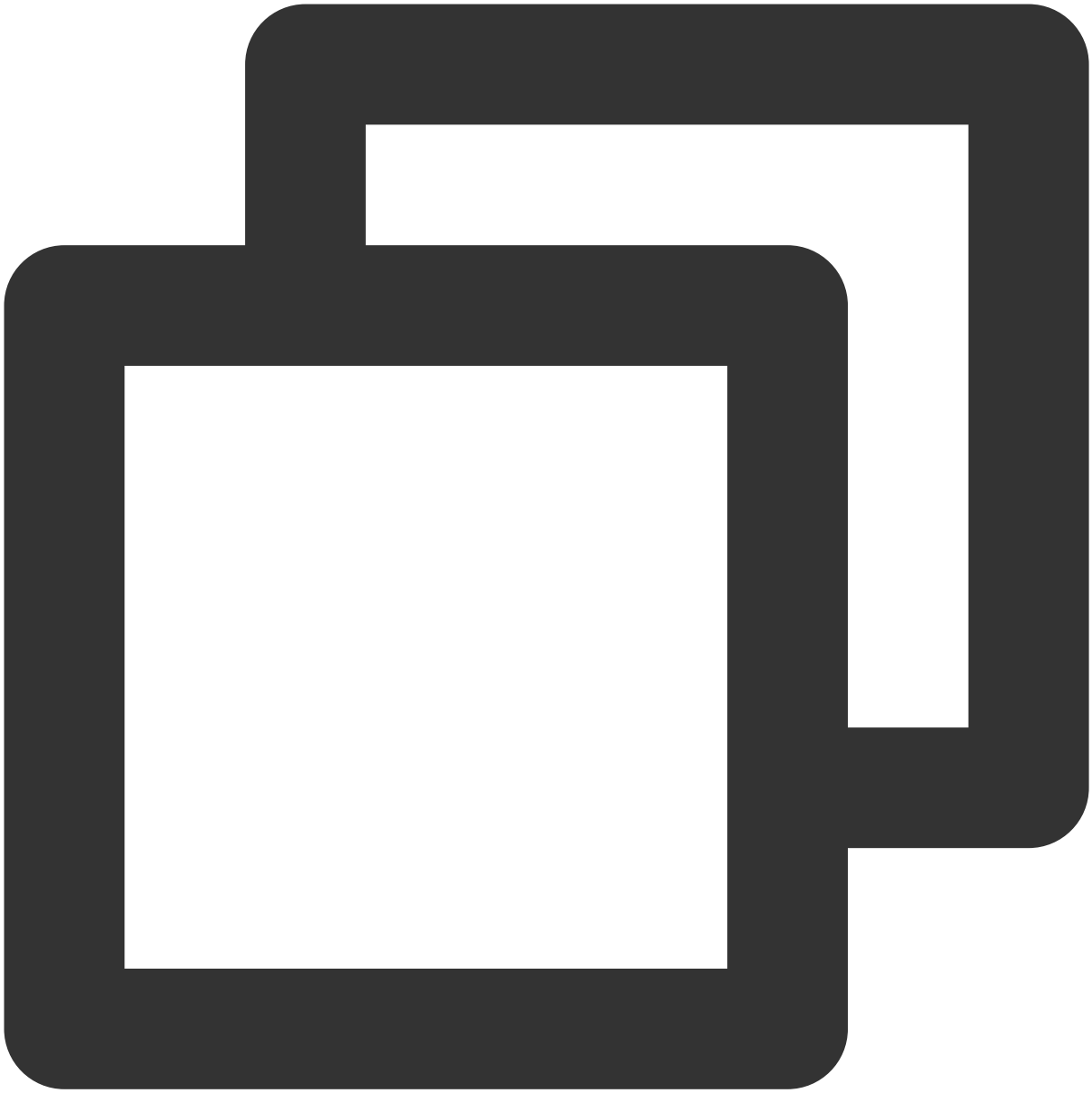
发送文本消息

文本消息区分单聊和群聊，涉及的接口、传参有所区别。
发送文本消息可以采用两种接口：普通接口和高级接口。高级接口比普通接口能设置更多的发送参数（例如优先级、离线推送信息等）。
普通接口参考下文具体描述，高级接口就是上文中提到的 `sendMessage` 。

单聊文本消息

高级接口

调用高级接口发送单聊文本消息分两步：
1. 调用 `createTextMessage` (Details) 创建文本消息。
2. 调用 `sendMessage` (Details) 发送消息。
示例代码如下：



```
import { TencentImSDKPlugin } from 'react-native-tim-js';

// 创建文本消息
const createTextMessage = await TencentImSDKPlugin.v2TIMManager.getMessageManager()
 if(createTextMessage.code == 0){
    String id =  createTextMessage.data.id;

       // 发送文本消息
   const sendMessageRes = await TencentImSDKPlugin.v2TIMManager.getMessageManager()
    if(sendMessageRes.code == 0){
      // 发送成功
```

```
        }
    }
```

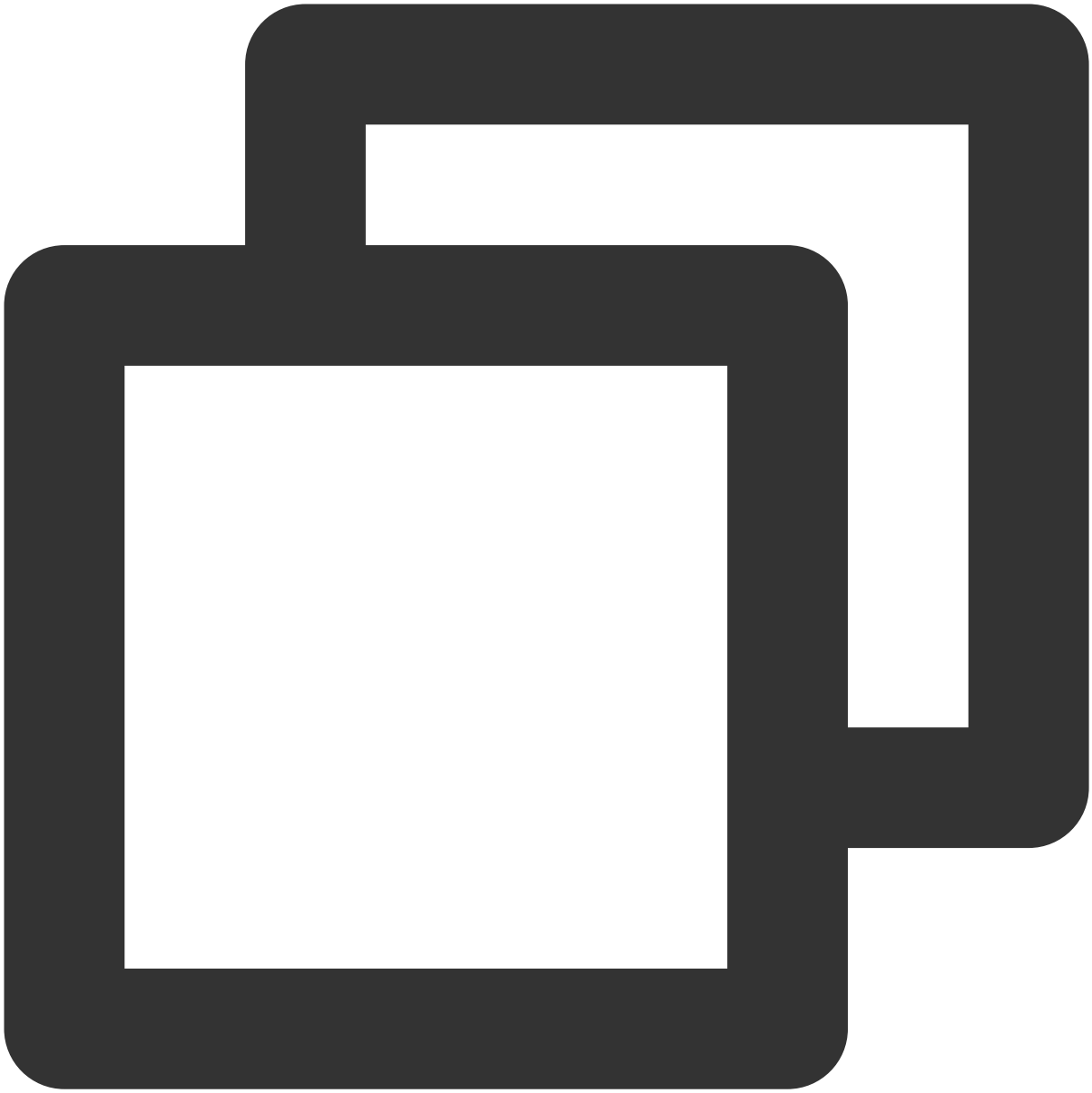
群聊文本消息

高级接口

调用高级接口发送群聊文本消息分两步：

1. 调用 `createTextMessage` (Details) 创建文本消息。
2. 调用 `sendMessage` (Details) 发送消息。

示例代码如下：



```
// 创建文本消息
const text = "test";
const atUserList = [];

const createTextAtMessageRes = await TencentImSDKPlugin.v2TIMManager.getMessageMana
 if(createTextAtMessageRes.code == 0){
    const id =  createTextAtMessageRes.data.id;

       // 发送文本消息
    const sendMessageRes = await TencentImSDKPlugin.v2TIMManager.getMessageManager(
    if(sendMessageRes.code == 0){
```

```
        // 发送成功
      }
    }
```

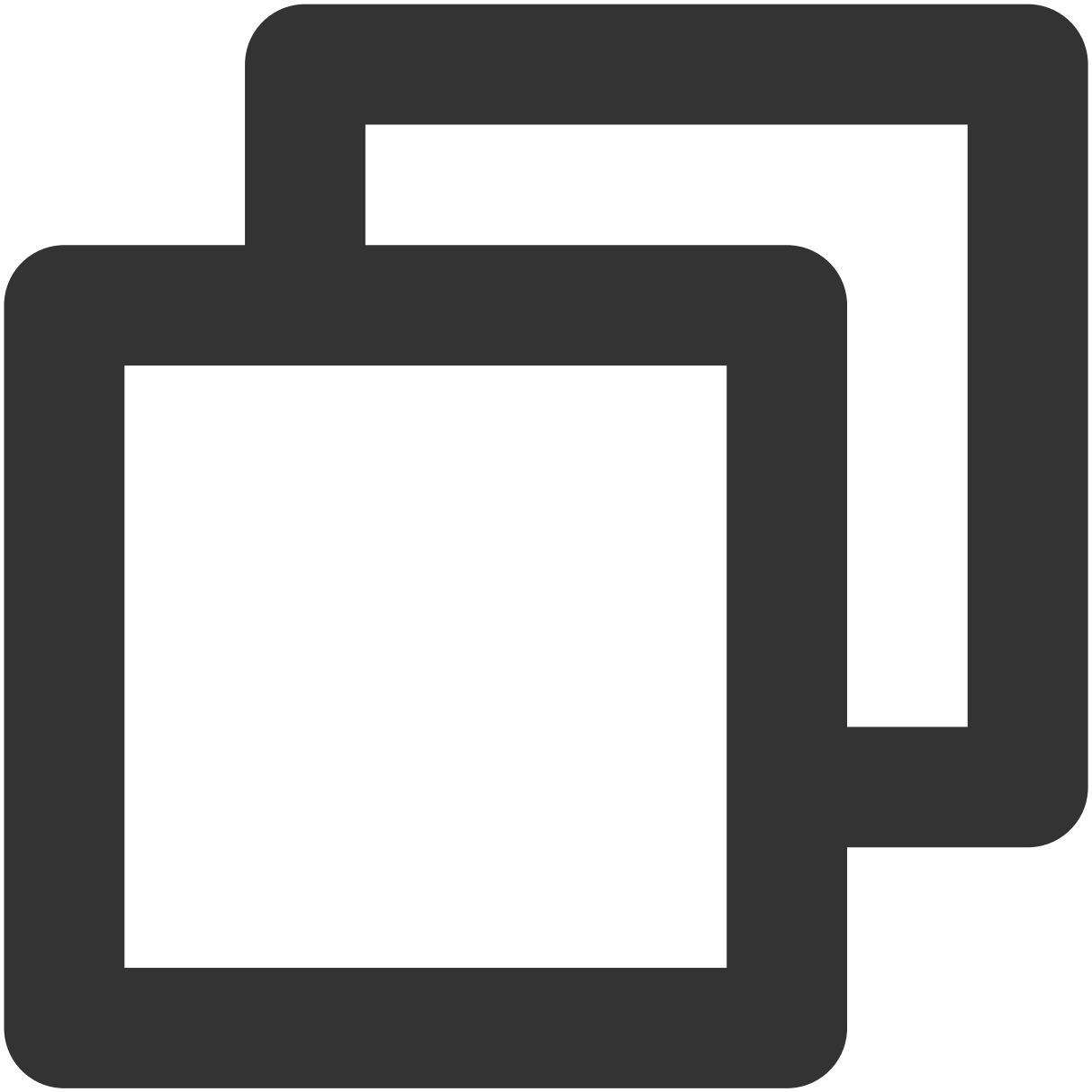
发送自定义消息

自定义消息区分单聊和群聊，涉及的接口或者传参有所区别。发送自定义消息可以采用两种接口：普通接口和高级接口。
高级接口即上文中已介绍过的 `sendMessage` (Details)，比普通接口能设置更多的发送参数（例如优先级、离线推送信息等）。

单聊自定义消息

高级接口

调用高级接口发送单聊自定义消息分两步：
1. 调用 `createCustomMessage` (Details) 创建自定义消息。
2. 调用 `sendMessage` (Details) 发送消息。
示例代码如下：



```
import { TencentImSDKPlugin } from 'react-native-tim-js';

// 创建自定义消息
const createCustomMessageRes = await TencentImSDKPlugin.v2TIMManager.getMessageMana
    data: '自定义data',
    desc: '自定义desc',
    extension: '自定义extension',
  );
  if(createCustomMessageRes.code == 0){
    const id =  createCustomMessageRes.data.id;
    // 发送自定义消息
```

```
    const sendMessageRes = await TencentImSDKPlugin.v2TIMManager.getMessageManager(
    if(sendMessageRes.code == 0){
      // 发送成功
    }
  }
```

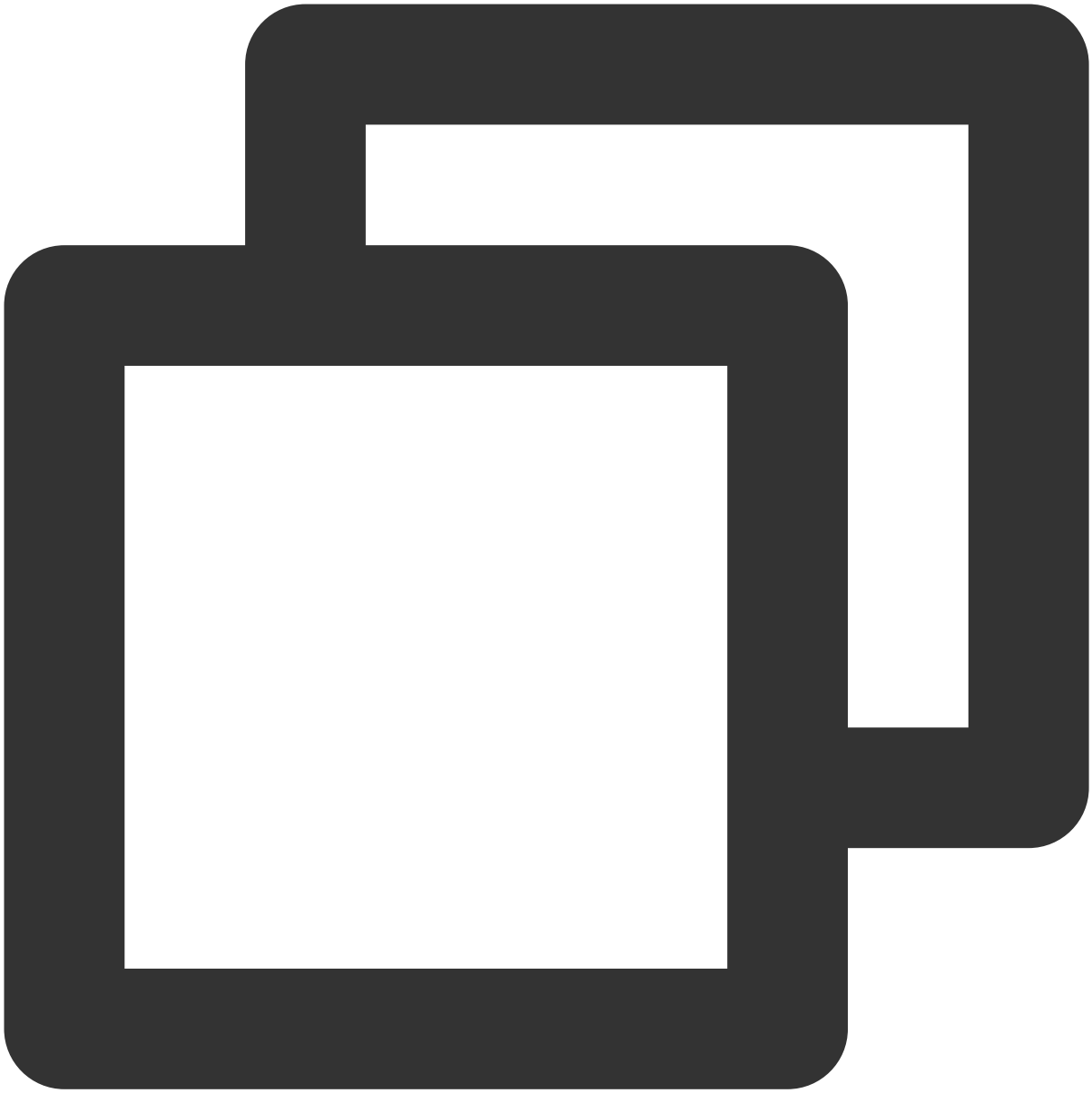
群聊自定义消息

高级接口

调用高级接口发送群聊自定义消息分两步：

1. 调用 `createCustomMessage` (Details) 创建自定义消息。
2. 调用 `sendMessage` (Details) 发送消息。

代码示例如下：



```
// 创建自定义消息
import { TencentImSDKPlugin } from 'react-native-tim-js';

// 创建自定义消息
const createCustomMessageRes = await TencentImSDKPlugin.v2TIMManager.getMessageMana
    data: '自定义data',
    desc: '自定义desc',
    extension: '自定义extension',
  );
  if(createCustomMessageRes.code == 0){
    const id =  createCustomMessageRes.data.id;
```

```
    // 发送自定义消息
    const sendMessageRes = await TencentImSDKPlugin.v2TIMManager.getMessageManager(
    if(sendMessageRes.code == 0){
      // 发送成功
    }
  }
```

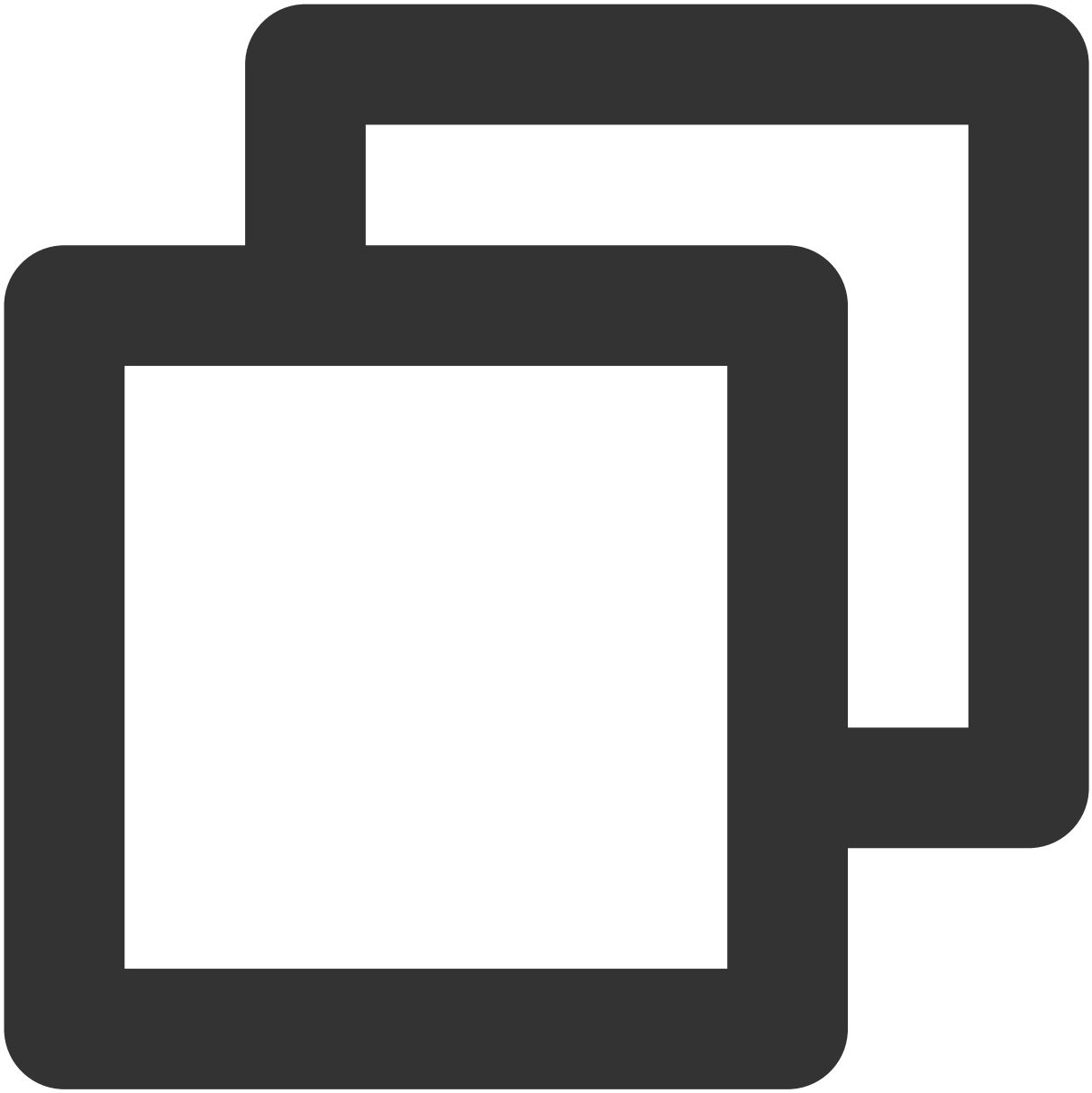
发送富媒体消息

富媒体消息发送没有普通接口，都需要使用高级接口，步骤是：

1. 调用 `createXxxMessage` 创建指定类型的富媒体消息对象，其中 Xxx 表示具体的消息类型。
2. 调用 `sendMessage` (Details) 发送消息。
3. 在消息回调中获取消息是否发送成功或失败。

图片消息

创建图片消息需要先获取到本地图片路径。
发送消息过程中，会先将图片文件上传至服务器，同时回调上传进度。上传成功后再发送消息。
示例代码如下：

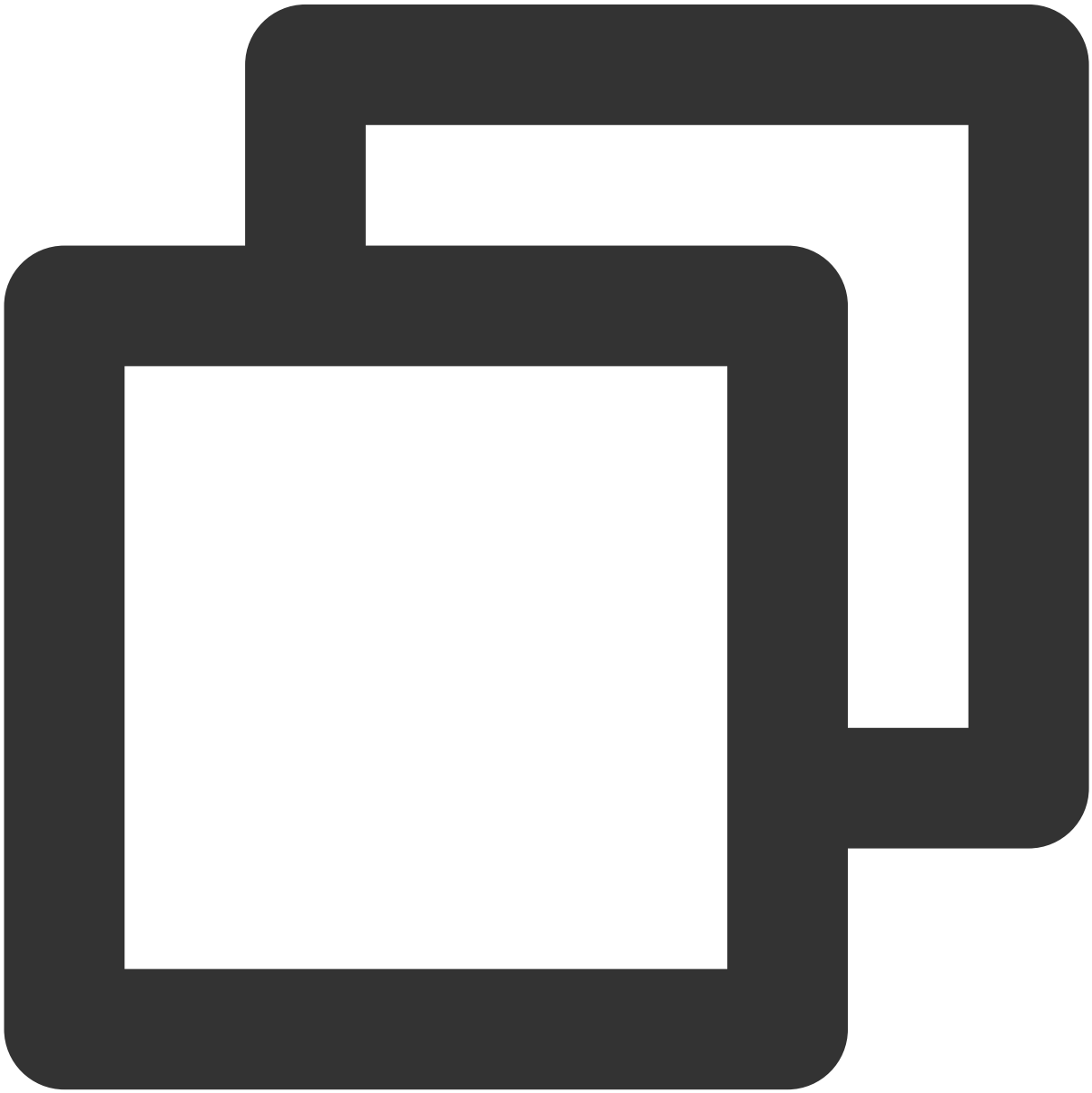


```
import { TencentImSDKPlugin } from 'react-native-tim-js';

const imagePath = '本地图片绝对路径';
const createImageMessageRes = await TencentImSDKPlugin.v2TIMManager.getMessageManag
  if (createImageMessageRes.code == 0) {
    const id = createImageMessageRes.data.id;
    const sendMessageRes = await TencentImSDKPlugin
        .v2TIMManager
        .getMessageManager()
        .sendMessage(id: id, receiver: "userID", groupID: "groupID");
    if (sendMessageRes.code == 0) {
```

```
        // 发送成功
      }
    }
```

语音消息

创建语音消息需要先获取到本地语音文件路径和语音时长，其中语音时长可用于接收端 UI 显示。
发送消息过程中，会先将语音文件上传至服务器，同时回调上传进度。上传成功后再发送消息。
示例代码如下：

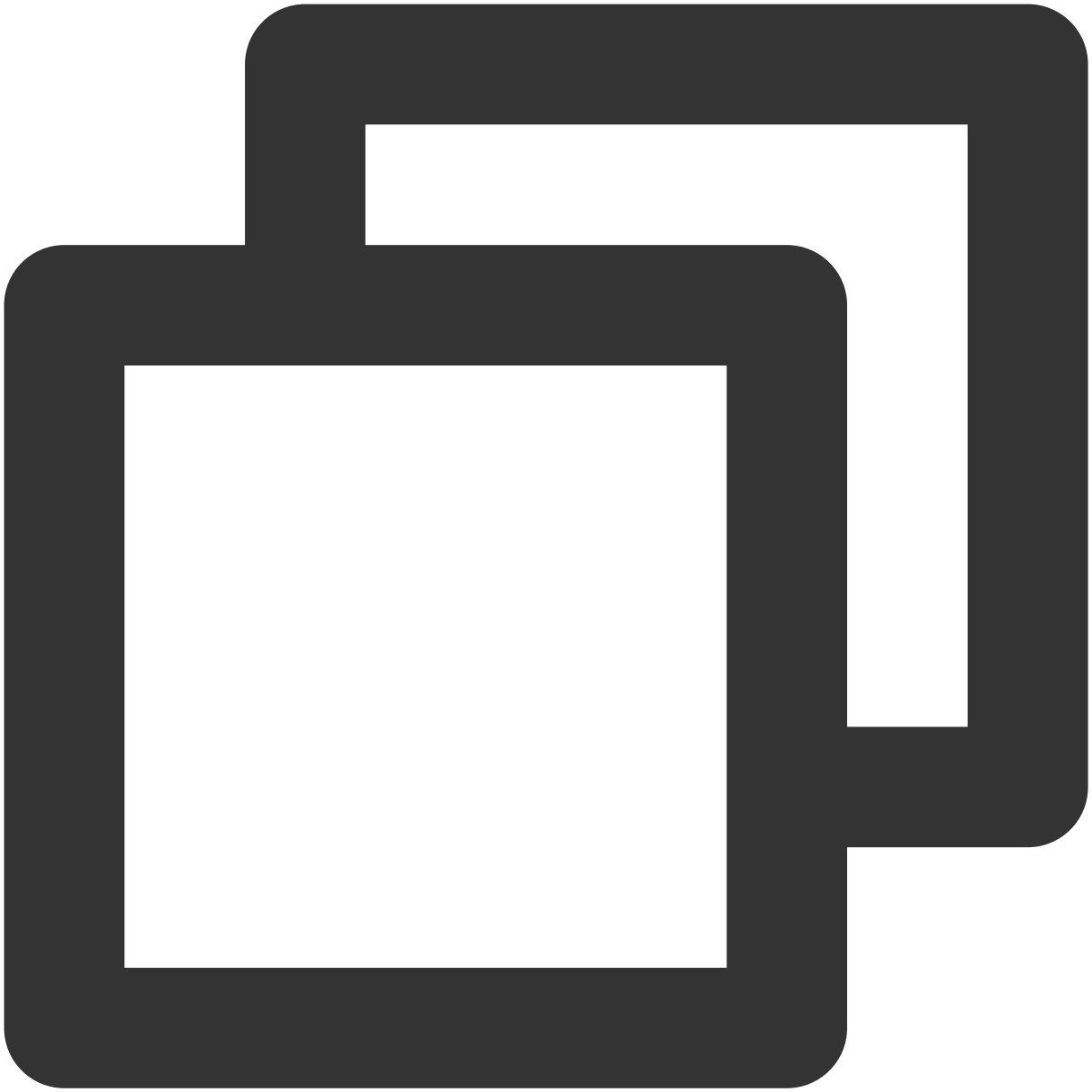


```
import { TencentImSDKPlugin } from 'react-native-tim-js';
```

```
const soundPath = '本地录音文件绝对路径';
const duration = 10; // 录音时长
const createSoundMessageRes =
      await TencentImSDKPlugin.v2TIMManager.getMessageManager().createSoundMessage(
        soundPath,
        duration// 录音时长
      );
  if (createSoundMessageRes.code == 0) {
    const id = createSoundMessageRes.data.id;
    const sendMessageRes = await TencentImSDKPlugin
        .v2TIMManager
        .getMessageManager()
        .sendMessage(id: id, receiver: "userID", groupID: "groupID");
    if (sendMessageRes.code == 0) {
      // 发送成功
    }
  }
```

视频消息

创建视频消息需要先获取到本地视频文件路径、视频时长和视频快照，其中时长和快照可用于接收端 UI 显示。
发送消息过程中，会先将视频上传至服务器，同时回调上传进度。上传成功后再发送消息。
示例代码如下：



```
import { TencentImSDKPlugin } from 'react-native-tim-js';

const videoFilePath = '本地视频文件绝对路径';
const type = "mp4"; // 视频类型
const duration = 10; // 视频时长
const snapshotPath = "本地视频封面文件绝对路径";
const createVideoMessageRes =
      await TencentImSDKPlugin.v2TIMManager
          .getMessageManager()
          .createVideoMessage(
            videoFilePath,
```

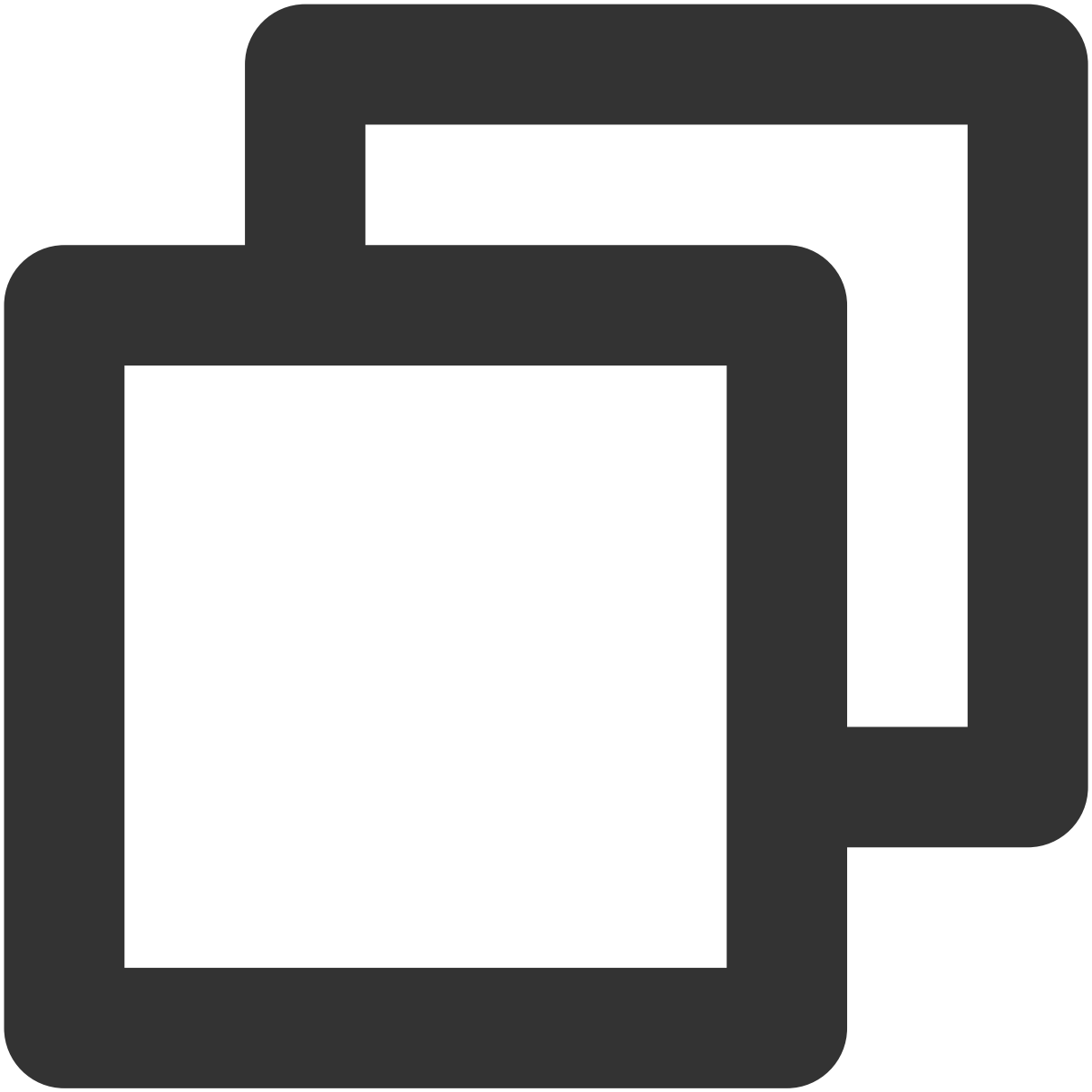
```
            type, // 视频类型
            duration,// 视频时长
            snapshotPath,
          );
  if (createVideoMessageRes.code == 0) {
    const id = createVideoMessageRes.data.id;
    const sendMessageRes = await TencentImSDKPlugin
        .v2TIMManager
        .getMessageManager()
        .sendMessage(id: id, receiver: "userID", groupID: "groupID");
    if (sendMessageRes.code == 0) {
      // 发送成功
    }
  }
```

文件消息

创建文件消息需要先获取到本地文件路径。

发送消息过程中，会先将文件上传至服务器，同时回调上传进度。上传成功后再发送消息。

示例代码如下：



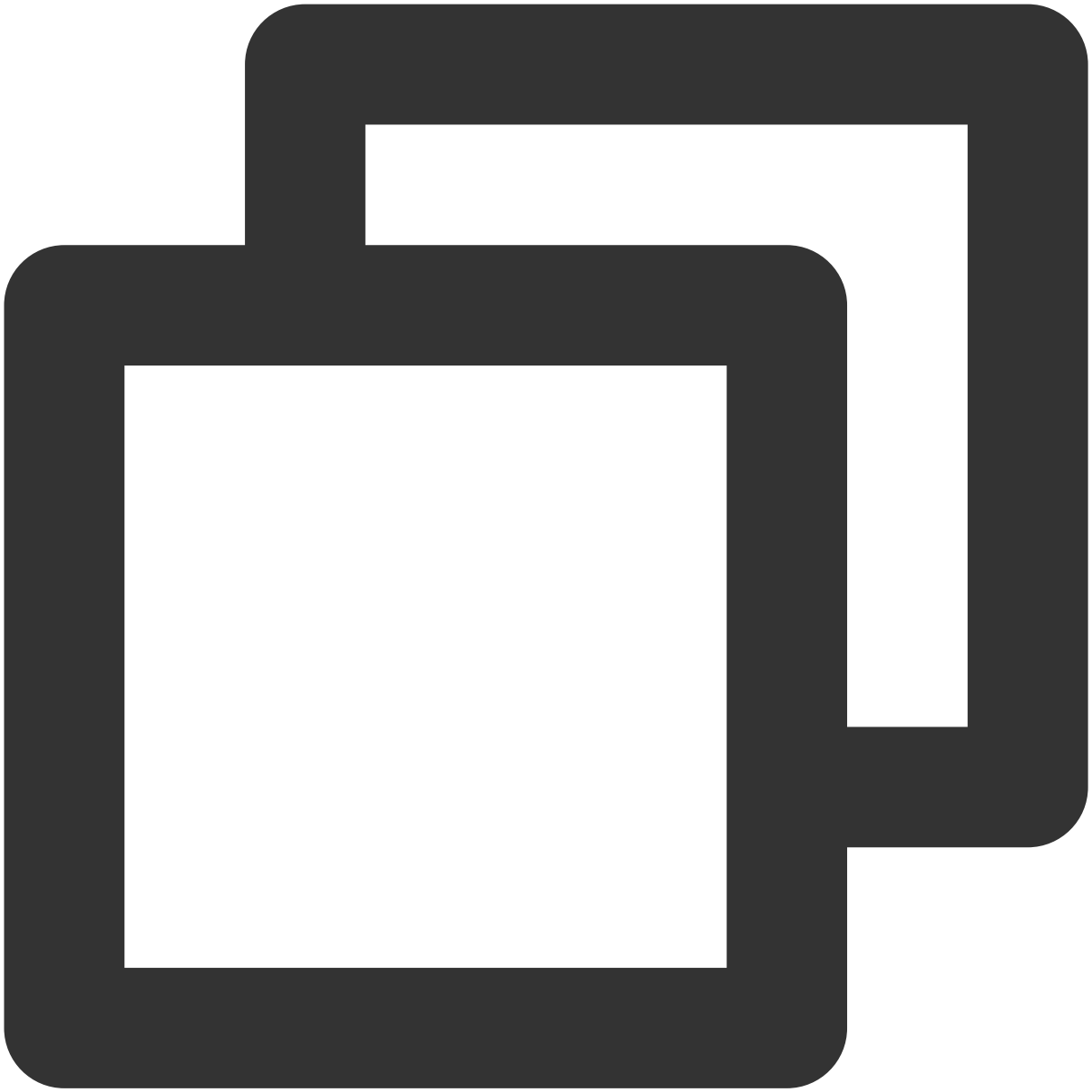
```
import { TencentImSDKPlugin } from 'react-native-tim-js';

const filePath = "本地文件绝对路径";
const fileName = "文件名";
const createFileMessageRes =
      await TencentImSDKPlugin.v2TIMManager
          .getMessageManager()
          .createFileMessage(
            filePath,
            fileName,
          );
```

版权所有：腾讯云计算（北京）有限责任公司

```
  if (createFileMessageRes.code == 0) {
    const id = createFileMessageRes.data.id;
    const sendMessageRes = await TencentImSDKPlugin
        .v2TIMManager
        .getMessageManager()
        .sendMessage(id: id, receiver: "userID", groupID: "groupID");
    if (sendMessageRes.code == 0) {
      // 发送成功
    }
  }
```

定位消息

定位消息会直接发送经纬度，一般需要配合地图控件显示。
示例代码如下：

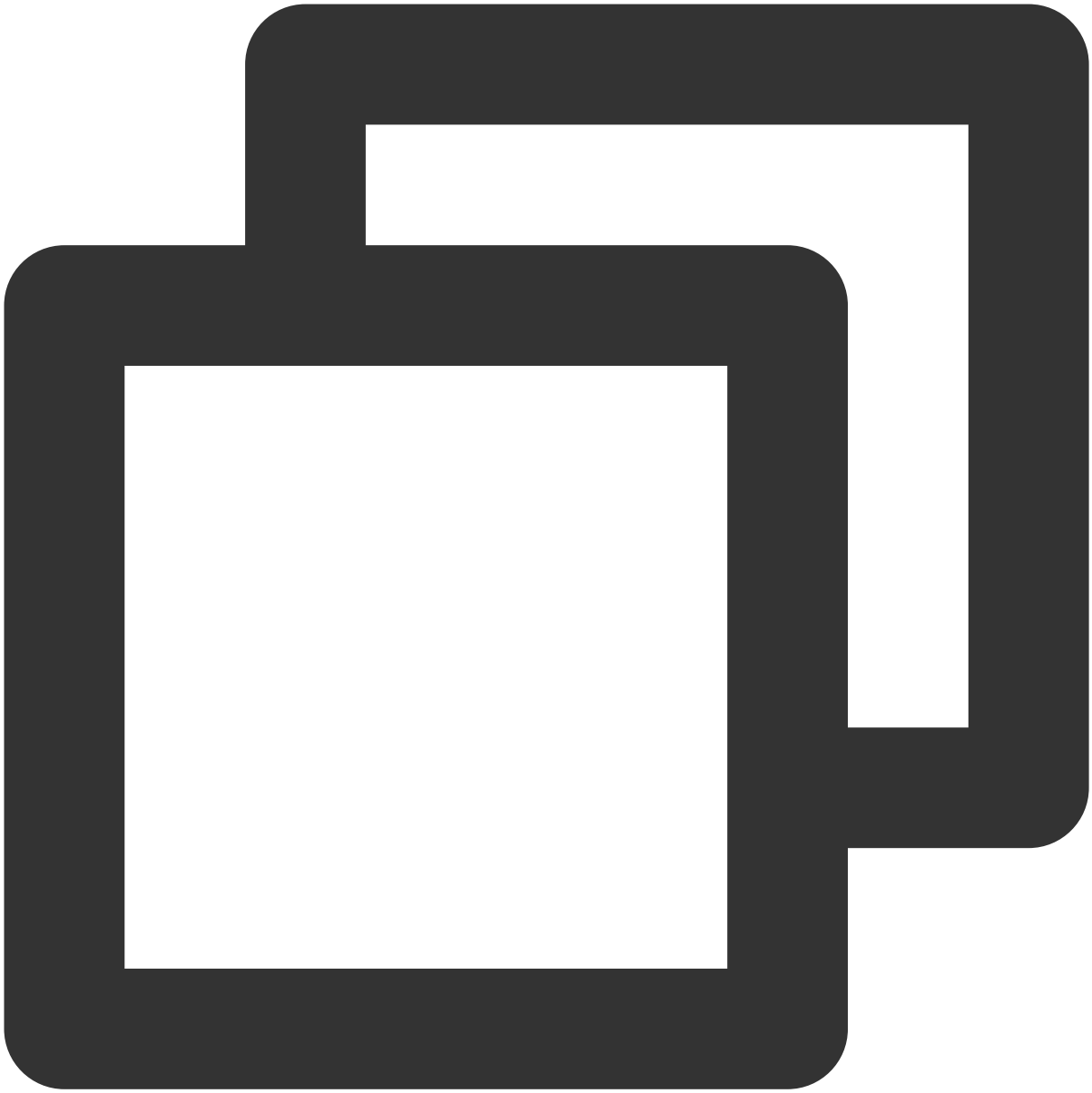


```
import { TencentImSDKPlugin } from 'react-native-tim-js';

const desc = "深圳市南山区深南大道"
const longitude = 34;
const latitude = 20;
const createLocationMessage =
      await TencentImSDKPlugin.v2TIMManager
          .getMessageManager()
          .createLocationMessage(
            desc,//位置信息摘要
            longitude,// 经度
```

```
            latitude, // 纬度
          );
  if (createLocationMessage.code == 0) {
    const id = createLocationMessage.data.id;
    const sendMessageRes = await TencentImSDKPlugin
        .v2TIMManager
        .getMessageManager()
        .sendMessage(id: id, receiver: "userID", groupID: "groupID");
    if (sendMessageRes.code == 0) {
      // 发送成功
    }
  }
```

表情消息

表情消息会直接发送表情编码，通常接收端需要将其转换成对应的表情 icon。
示例代码如下：



```
const createFaceMessageRes =
      await TencentImSDKPlugin.v2TIMManager
          .getMessageManager()
          .createFaceMessage(
            0,
            "",
          );
  if (createFaceMessageRes.code == 0) {
    const id = createFaceMessageRes.data.id;
    const sendMessageRes = await TencentImSDKPlugin
        .v2TIMManager
```

```
        .getMessageManager()
        .sendMessage(id: id, receiver: "userID", groupID: "groupID");
    if (sendMessageRes.code == 0) {
      // 发送成功
    }
  }
```

接收消息
Android&iOS&Windows&Mac

最近更新时间：2024-07-05 15:24:08

功能描述

通过 `addSimpleMsgListener` 监听接收文本、自定义消息，相关回调在 `V2TIMSimpleMsgListener` 协议中定义。
通过 `addAdvancedMsgListener` 监听接收所有类型消息（文本、自定义、富媒体消息），相关回调在 `V2TIMAdvancedMsgListener` 协议中定义。

设置消息监听器

SDK 提供了 2 种消息监听器，简单消息监听器 `V2TIMSimpleMsgListener` 和高级消息监听器 `V2TIMAdvancedMsgListener` 。
两者的区别在于:
1. 简单消息监听器**只能**接收文本、自定义消息。如果您的业务只需要这两种消息，可以仅使用简单消息监听器。
2. 高级消息监听器可以接收**所有**类型的消息。如果您的业务还需要支持富媒体、合并消息等其他类型，请使用高级消息监听器。

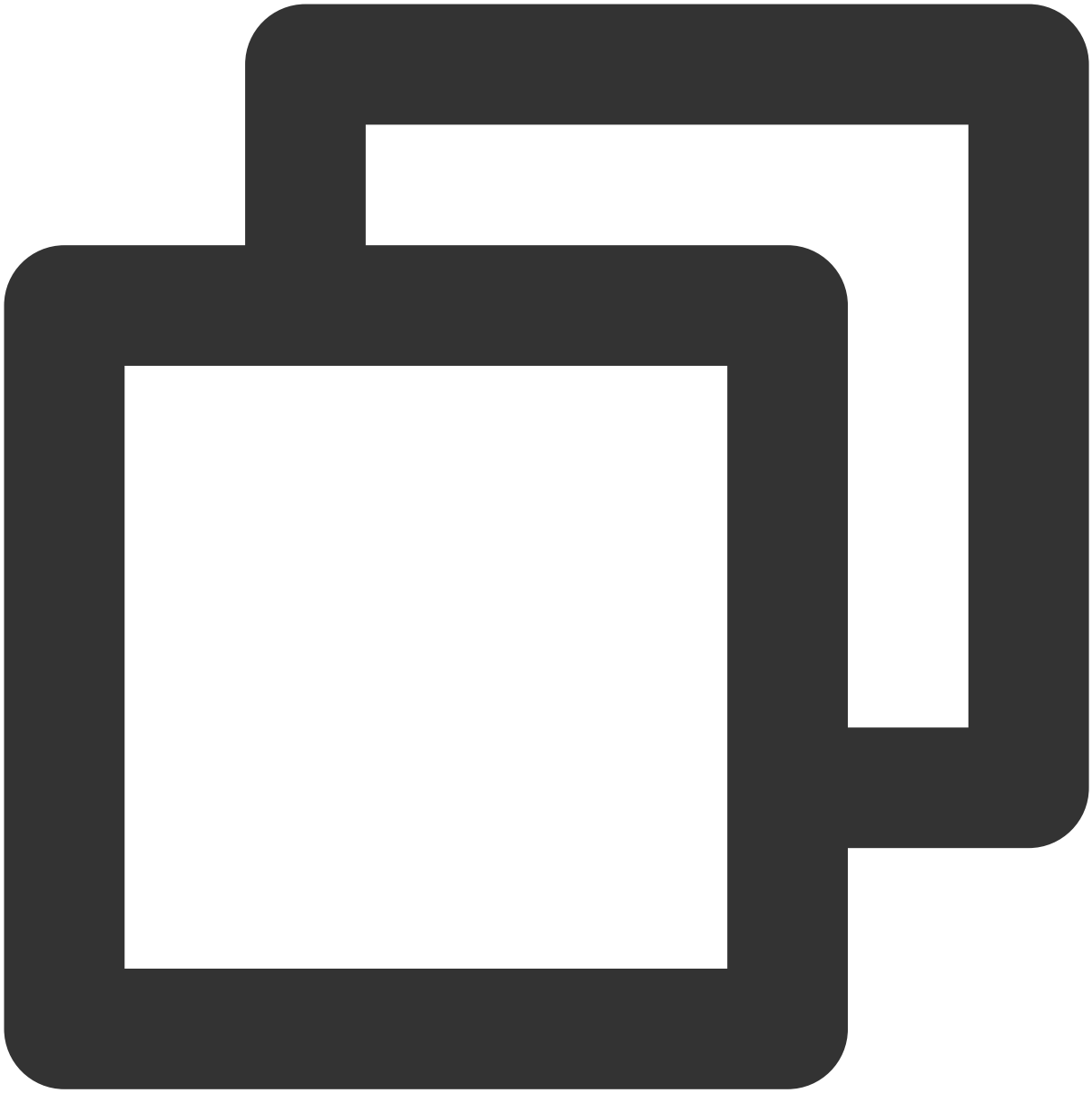
注意

1. `addSimpleMsgListener` 和 `addAdvancedMsgListener` 请使用其中之一，**切勿混用**，以免产生不可预知的逻辑 bug。
2. 如果想要正常接收下面各种类型的消息，**必须**先添加消息监听器，否则无法正常接收。

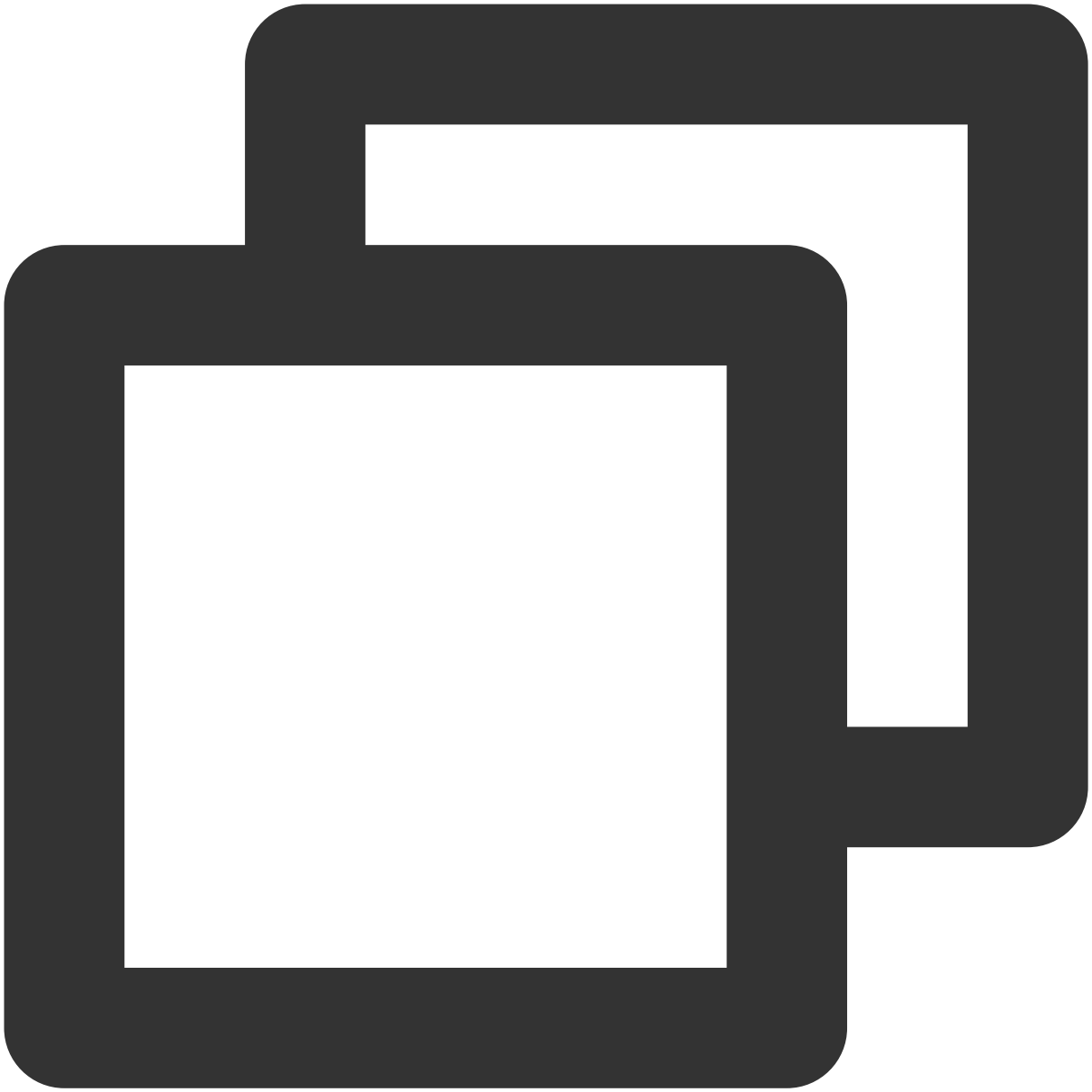
简单消息监听器

添加监听器

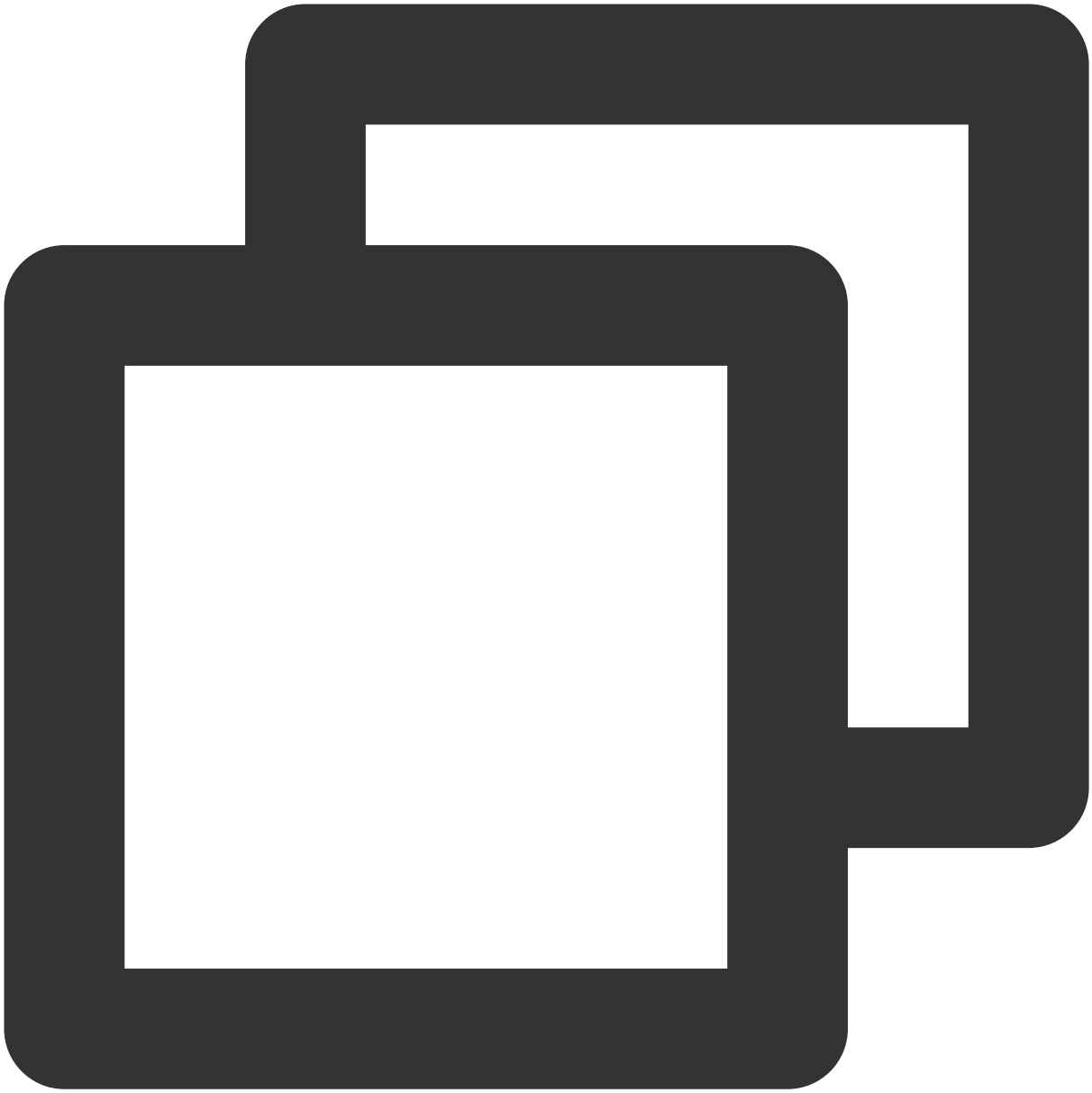
接收方调用 `addSimpleMsgListener` (Android / iOS & Mac / Windows) 添加简单消息监听器。一般建议在比较靠前的时间点调用，例如聊天消息界面初始化后，确保 App 能及时收到消息。
示例代码如下：
Android
iOS & Mac
Windows



```
V2TIMManager.getInstance().addSimpleMsgListener(simpleMsgListener);
```



```
// self 为 id<V2TIMSignalingListener>
[[V2TIMManager sharedInstance] addSimpleMsgListener:self];
```



```
class SimpleMsgListener final : public V2TIMSimpleMsgListener {
    // 成员 ...
};

// 添加基本消息的事件监听器，注意在移除监听器之前需要保持 simpleMsgListener 的生命期，以免接收不
SimpleMsgListener simpleMsgListener;
V2TIMManager::GetInstance()->AddSimpleMsgListener(&simpleMsgListener);
```

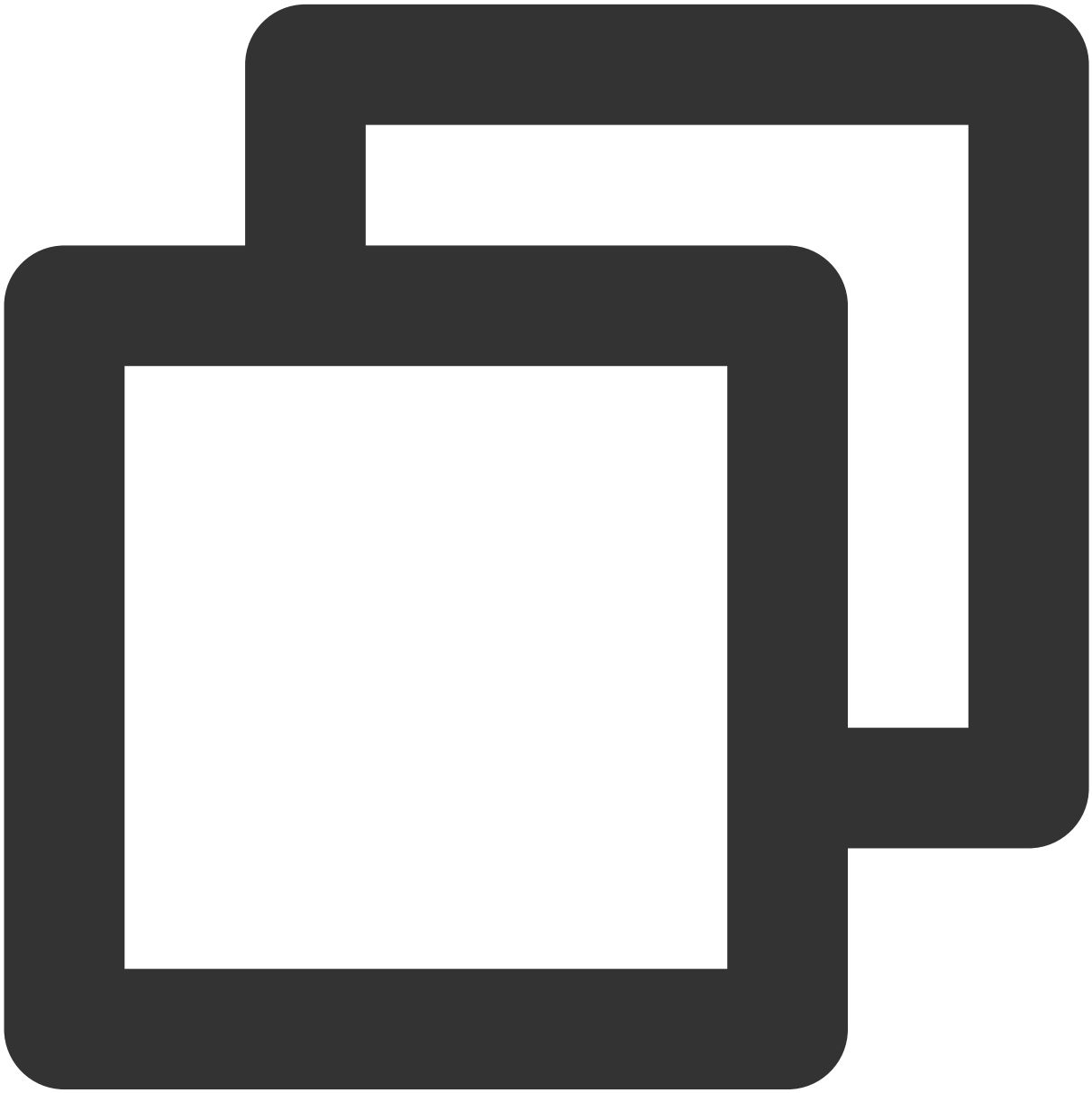
监听器回调事件

添加成功简单消息监听器后，接收方可以在 `V2TIMSimpleMsgListener` (Android / iOS & Mac / Windows) 的回调中接收不同类型消息，说明如下：

Android

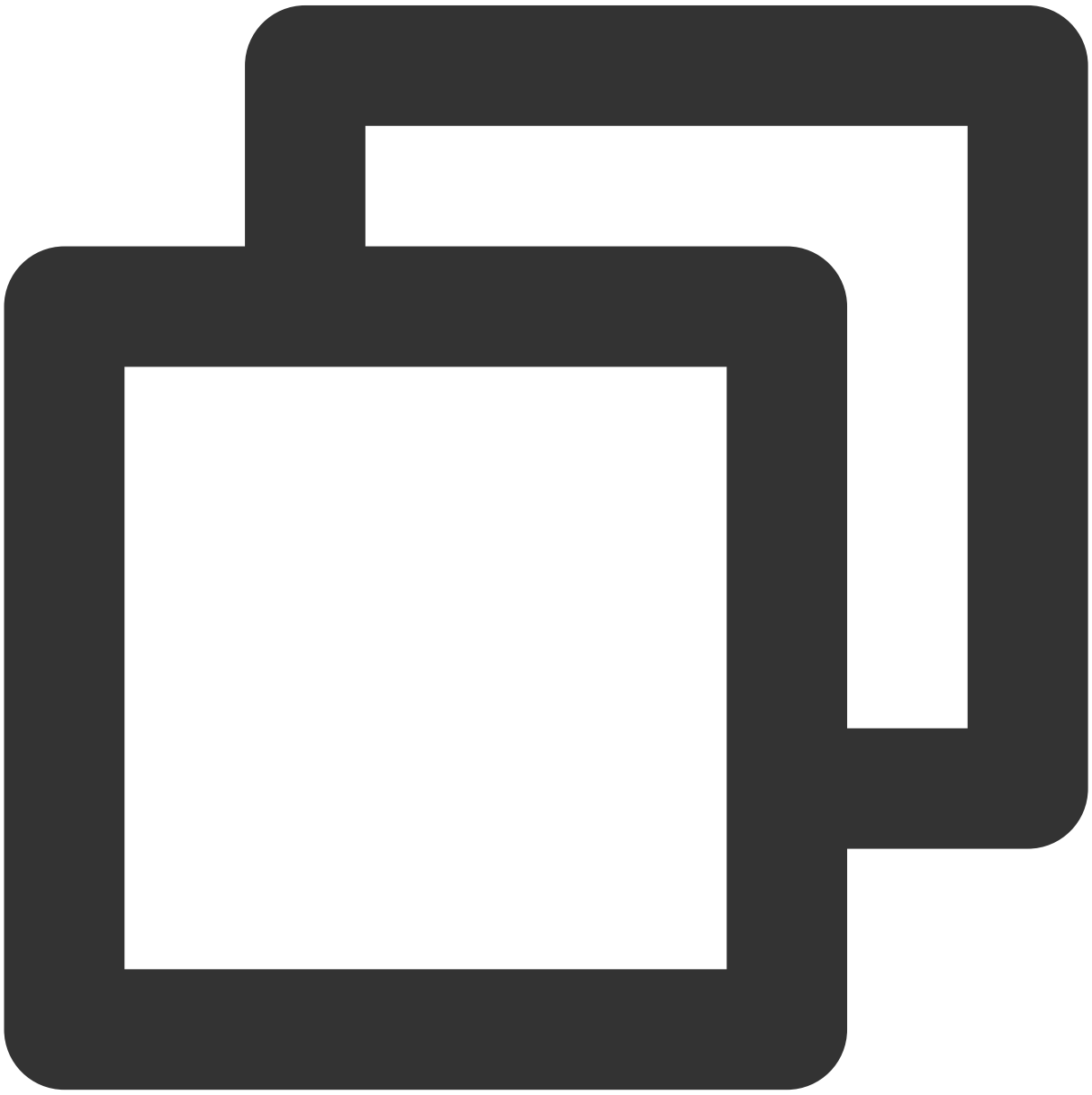
iOS & Mac

Windows



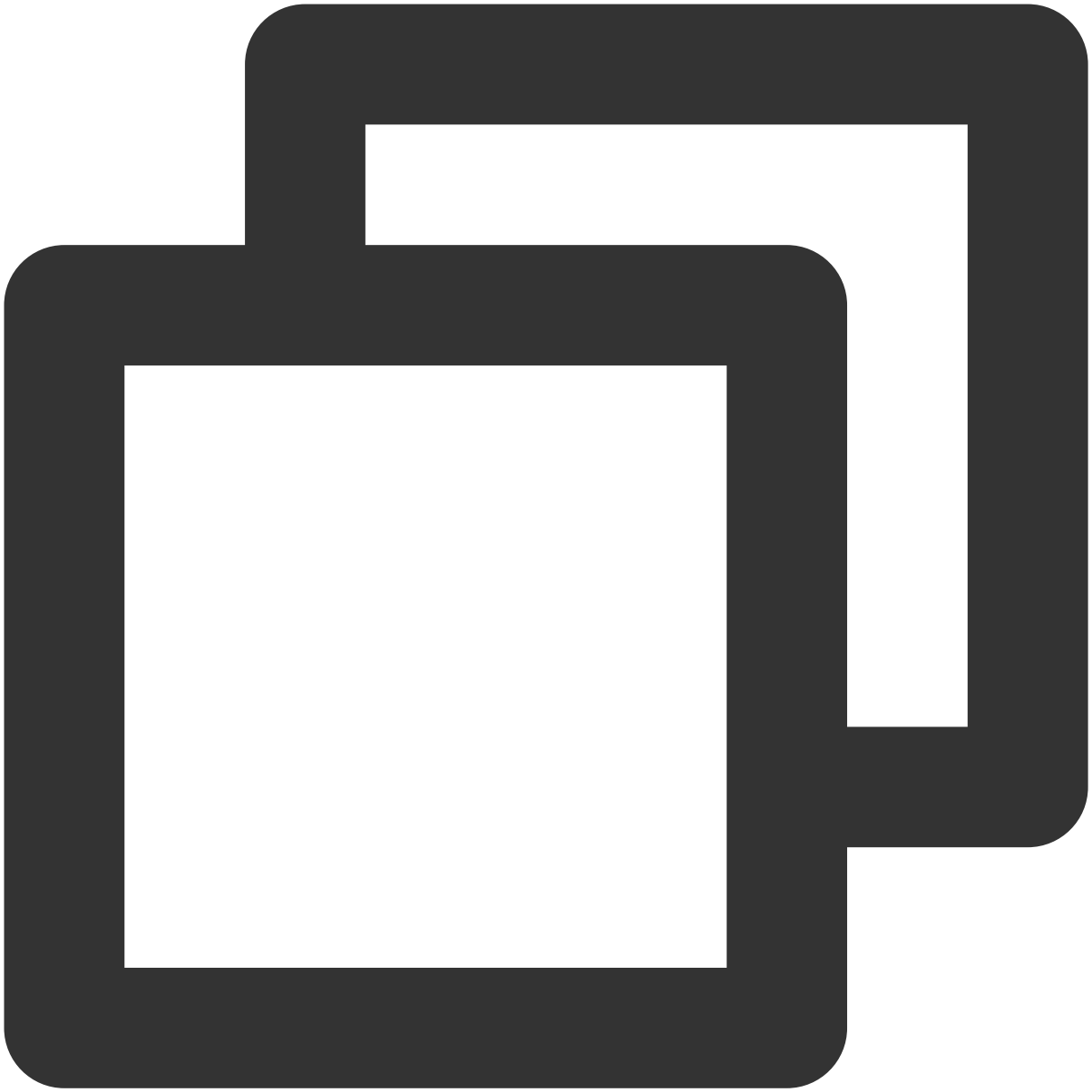
```
public abstract class V2TIMSimpleMsgListener {
    // 收到 C2C 文本消息
    public void onRecvC2CTextMessage(String msgID, V2TIMUserInfo sender, String tex
```

```
    // 收到 C2C 自定义（信令）消息
    public void onRecvC2CCustomMessage(String msgID, V2TIMUserInfo sender, byte[] c

    // 收到群文本消息
    public void onRecvGroupTextMessage(String msgID, String groupID, V2TIMGroupMemb

    // 收到群自定义（信令）消息
    public void onRecvGroupCustomMessage(String msgID, String groupID, V2TIMGroupMe
}
```



```
/// IMSDK 基本消息回调
```

```
@protocol V2TIMSimpleMsgListener <NSObject>
@optional

/// 收到 C2C 文本消息
- (void)onRecvC2CTextMessage:(NSString *)msgID  sender:(V2TIMUserInfo *)info text:(

/// 收到 C2C 自定义（信令）消息
- (void)onRecvC2CCustomMessage:(NSString *)msgID  sender:(V2TIMUserInfo *)info cust

/// 收到群文本消息
- (void)onRecvGroupTextMessage:(NSString *)msgID groupID:(NSString *)groupID sender

/// 收到群自定义（信令）消息
- (void)onRecvGroupCustomMessage:(NSString *)msgID groupID:(NSString *)groupID send
@end
```



```
class SimpleMsgListener final : public V2TIMSimpleMsgListener {
public:
    SimpleMsgListener() = default;
    ~SimpleMsgListener() override = default;

    // 收到 C2C 文本消息
    void OnRecvC2CTextMessage(const V2TIMString& msgID, const V2TIMUserFullInfo& se
                              const V2TIMString& text) override {}

    // 收到 C2C 自定义（信令）消息
    void OnRecvC2CCustomMessage(const V2TIMString& msgID, const V2TIMUserFullInfo&
```

```
                                const V2TIMBuffer& customData) override {}

    // 收到群文本消息
    void OnRecvGroupTextMessage(const V2TIMString& msgID, const V2TIMString& groupI
                                const V2TIMGroupMemberFullInfo& sender, const V2TIM

    // 收到群自定义（信令）消息
    void OnRecvGroupCustomMessage(const V2TIMString& msgID, const V2TIMString& grou
                                  const V2TIMGroupMemberFullInfo& sender,
                                  const V2TIMBuffer& customData) override {}
};
```

移除监听器

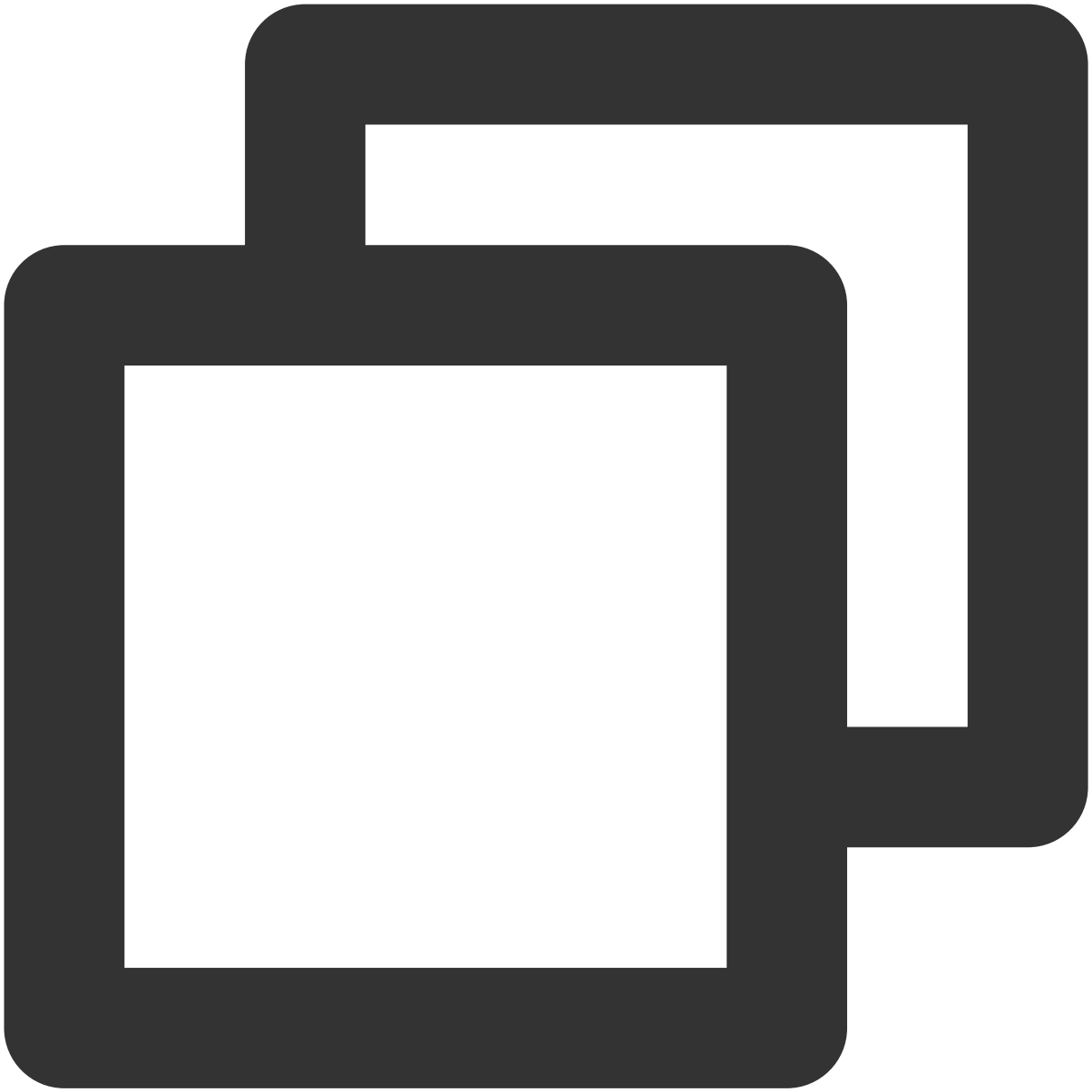
如果想停止接收消息，接收方可调用 `removeSimpleMsgListener` (Android / iOS & Mac / Windows) 移除简单消息监听器。

示例代码如下：

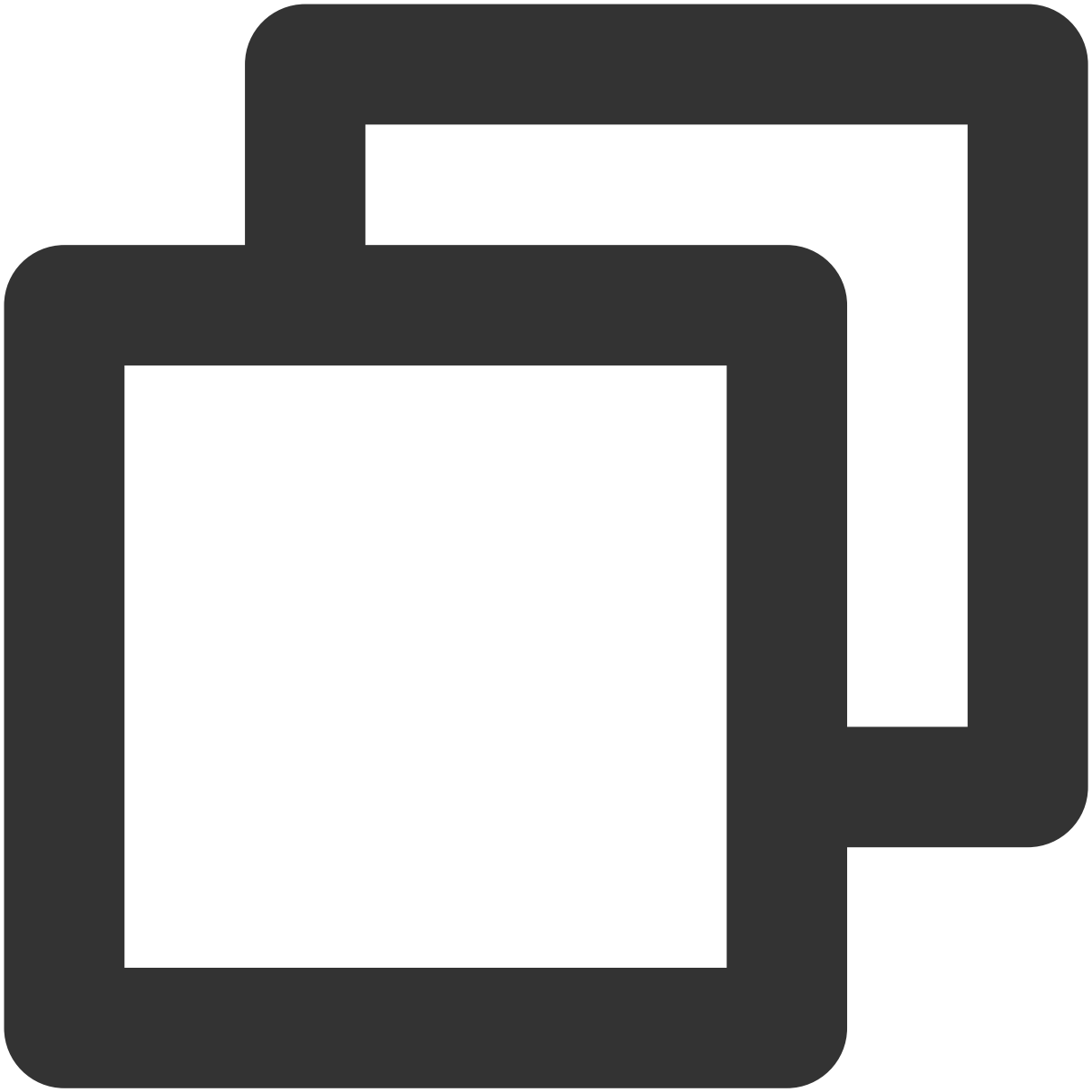
Android

iOS & Mac

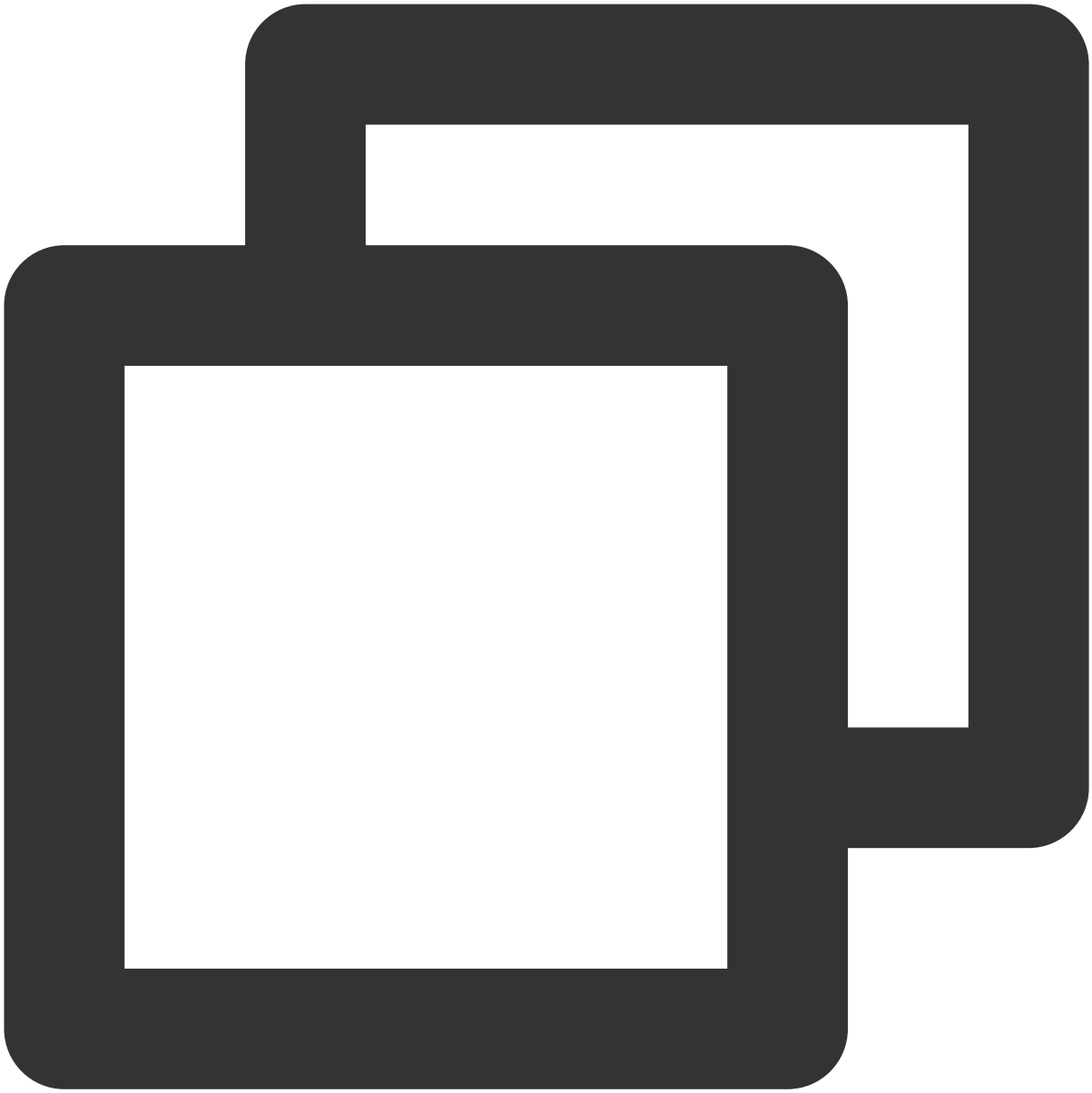
Windows



```
V2TIMManager.getInstance().removeSimpleMsgListener(simpleMsgListener);
```



```
// self 为 id<V2TIMSignalingListener>
[[V2TIMManager sharedInstance] removeSimpleMsgListener:self];
```



```
class SimpleMsgListener final : public V2TIMSimpleMsgListener {
    // 成员 ...
};

// simpleMsgListener 是 SimpleMsgListener 的实例
V2TIMManager::GetInstance()->RemoveSimpleMsgListener(&simpleMsgListener);
```

高级消息监听器

添加监听器

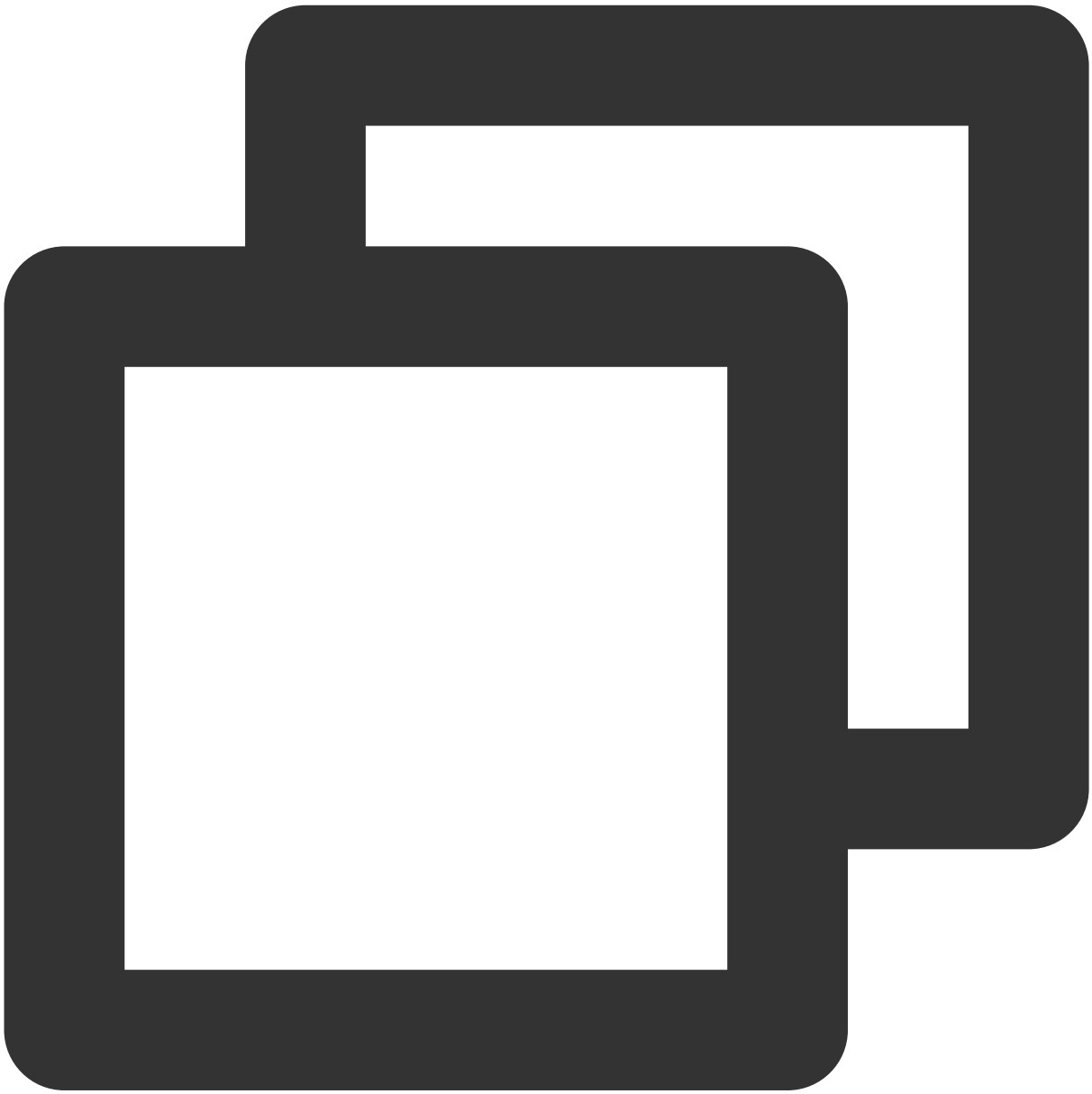
接收方调用 `addAdvancedMsgListener` (Android / iOS & Mac / Windows) 添加高级消息监听器。一般建议在比较靠前的时间点调用，例如例如聊天消息界面初始化后，确保 App 能及时收到消息。

示例代码如下：

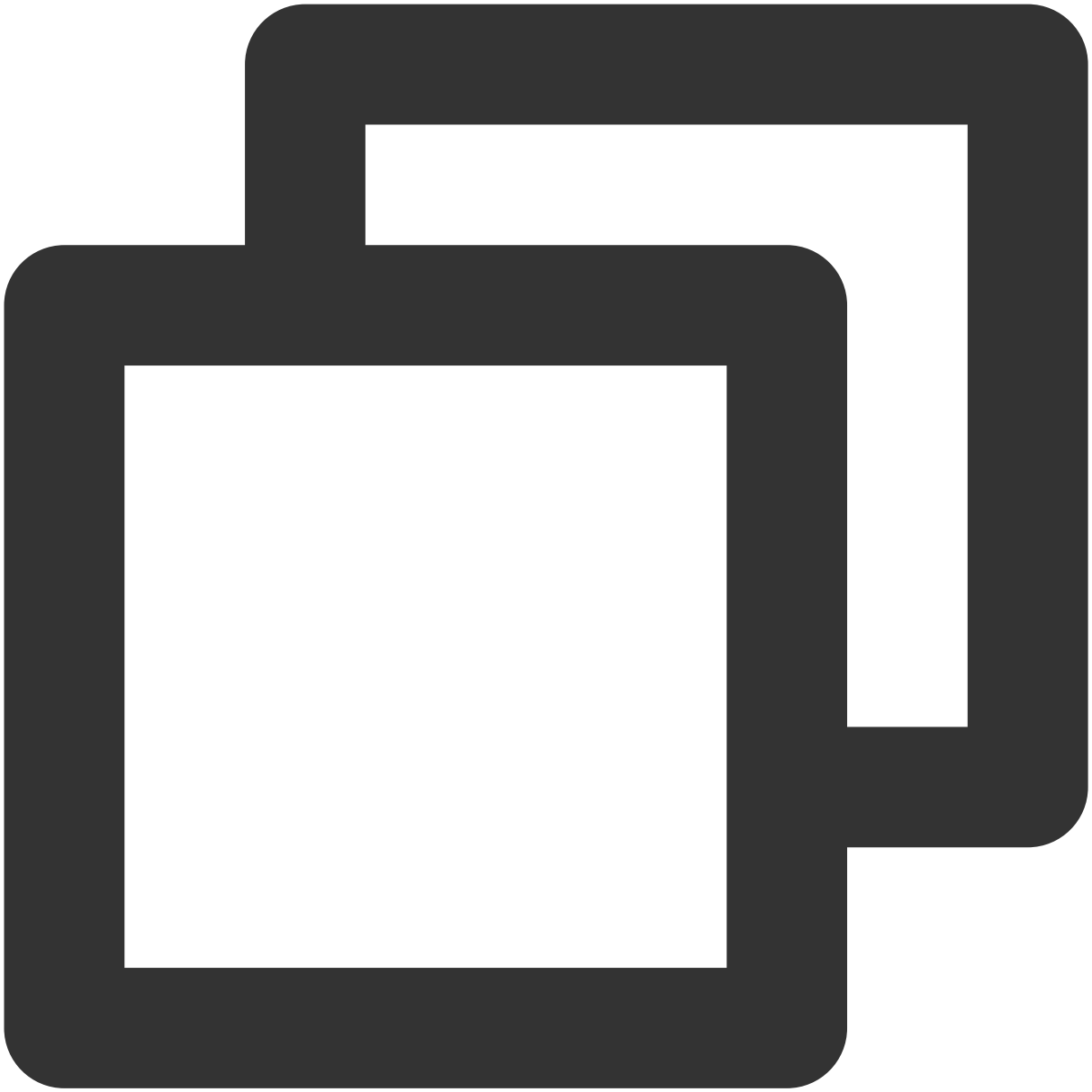
Android

iOS & Mac

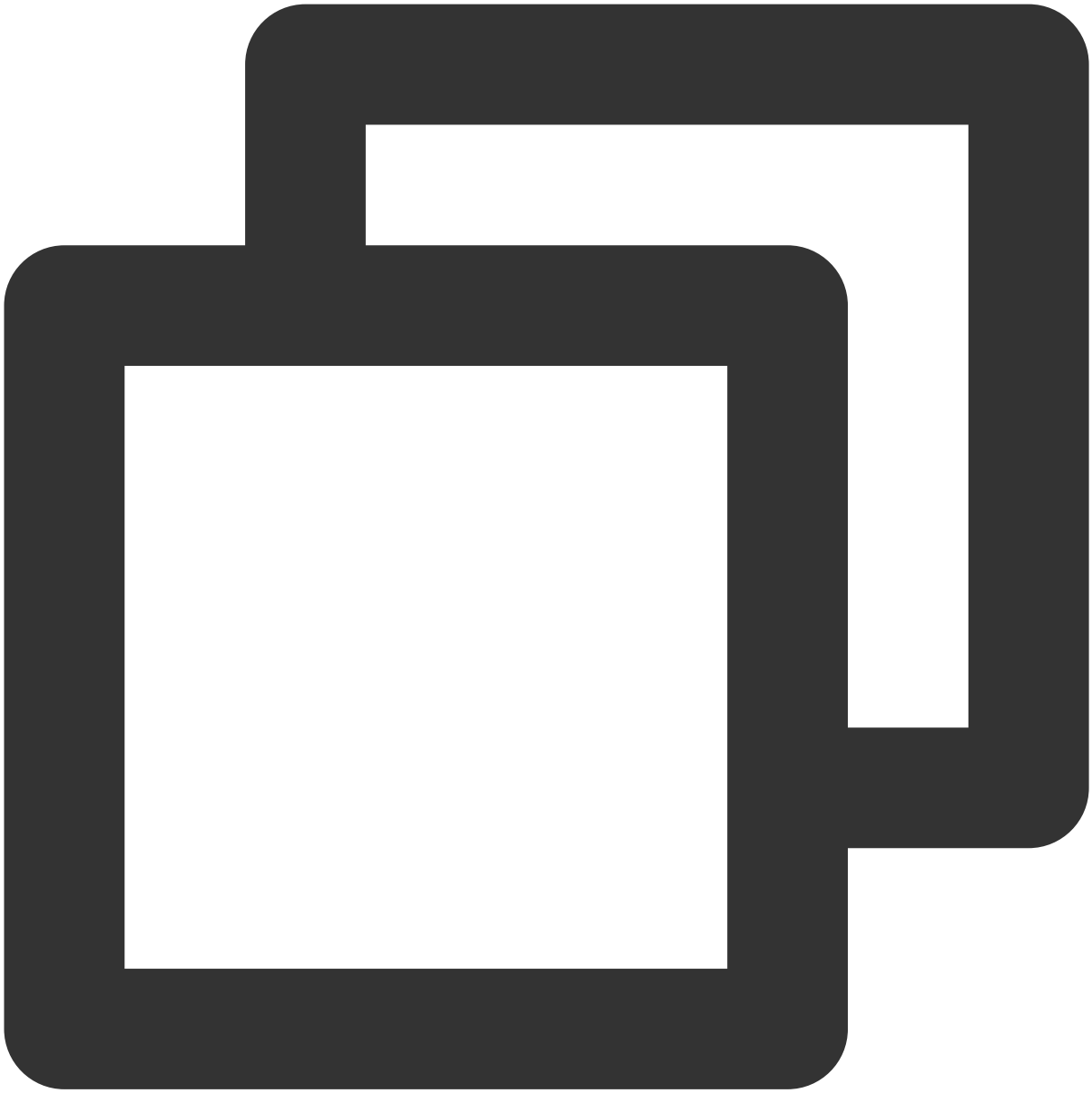
Windows



```
V2TIMManager.getMessageManager().addAdvancedMsgListener(advancedMsgListener);
```



```
// self 为 id<V2TIMAdvancedMsgListener>
[[V2TIMManager sharedInstance] addAdvancedMsgListener:self];
```



```
class AdvancedMsgListener final : public V2TIMAdvancedMsgListener {
    // 成员 ...
};

// 添加高级消息的事件监听器，注意在移除监听器之前需要保持 advancedMsgListener 的生命期，以免接收
AdvancedMsgListener advancedMsgListener;
V2TIMManager::GetInstance()->GetMessageManager()->AddAdvancedMsgListener(&advancedM
```

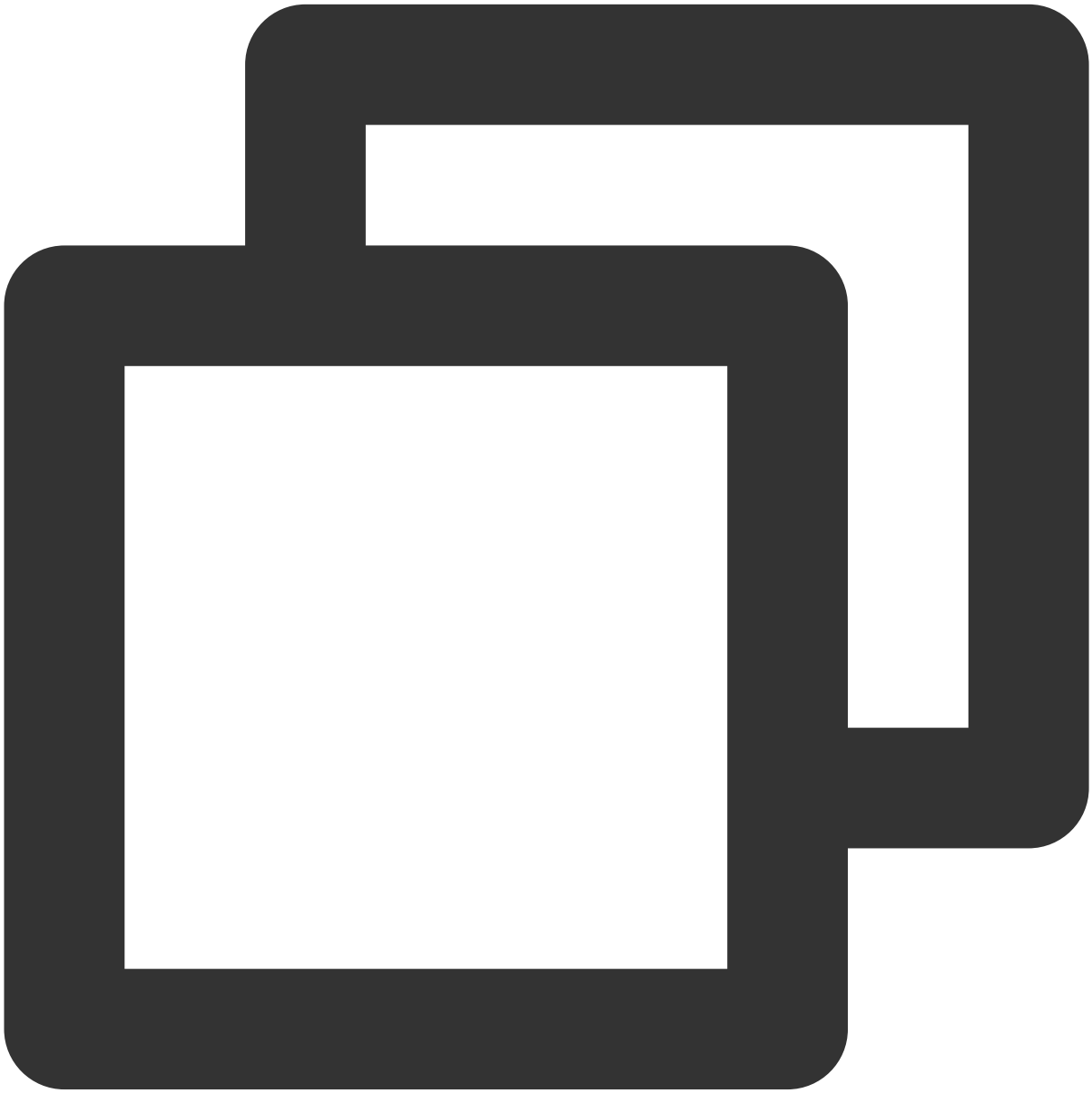
监听器回调事件

添加成功高级消息监听器后，接收方可以在 `V2TIMAdvancedMsgListener` (Android / iOS & Mac / Windows) 的回调中接收不同类型消息，说明如下：

Android

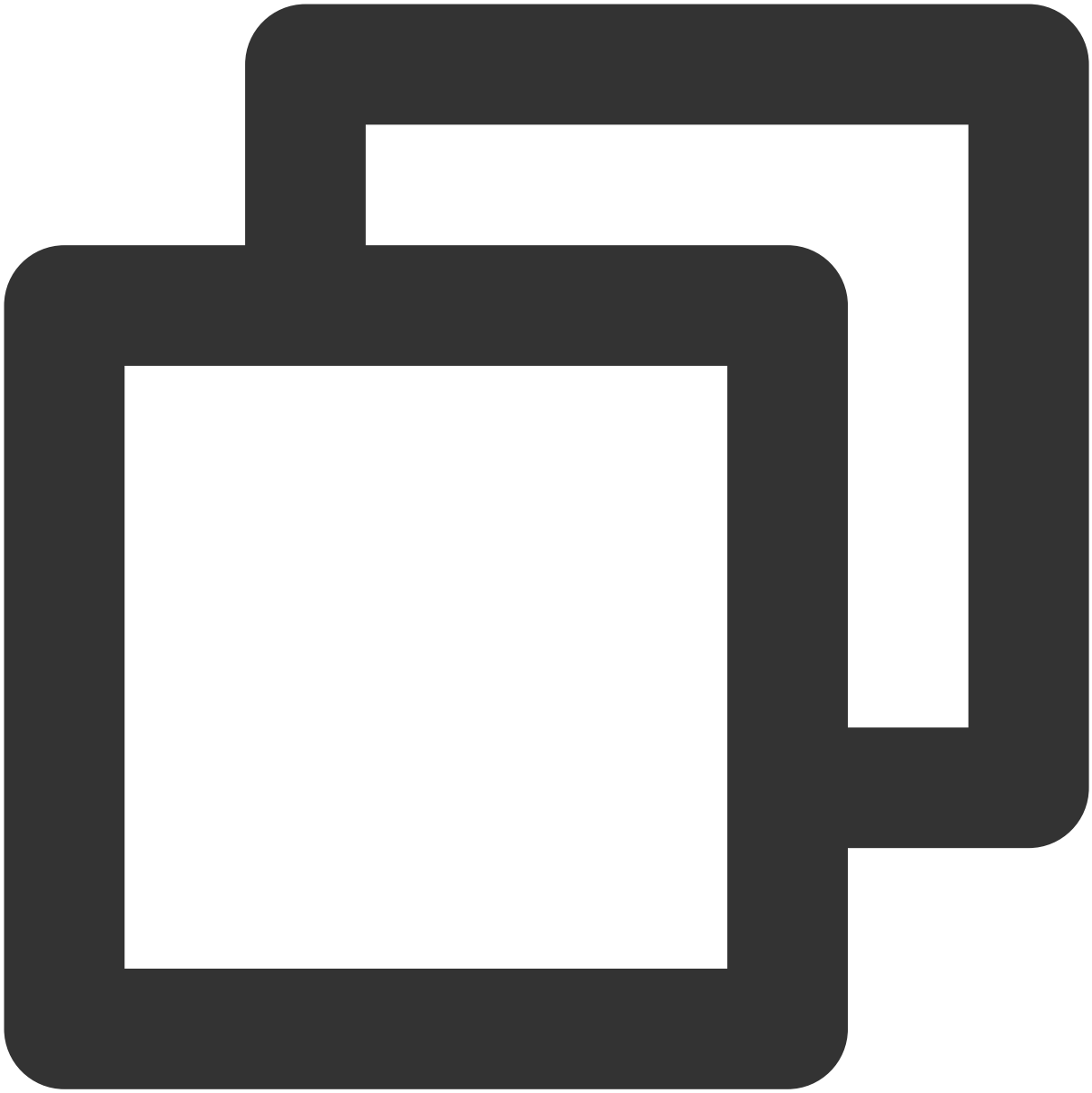
iOS & Mac

Windows



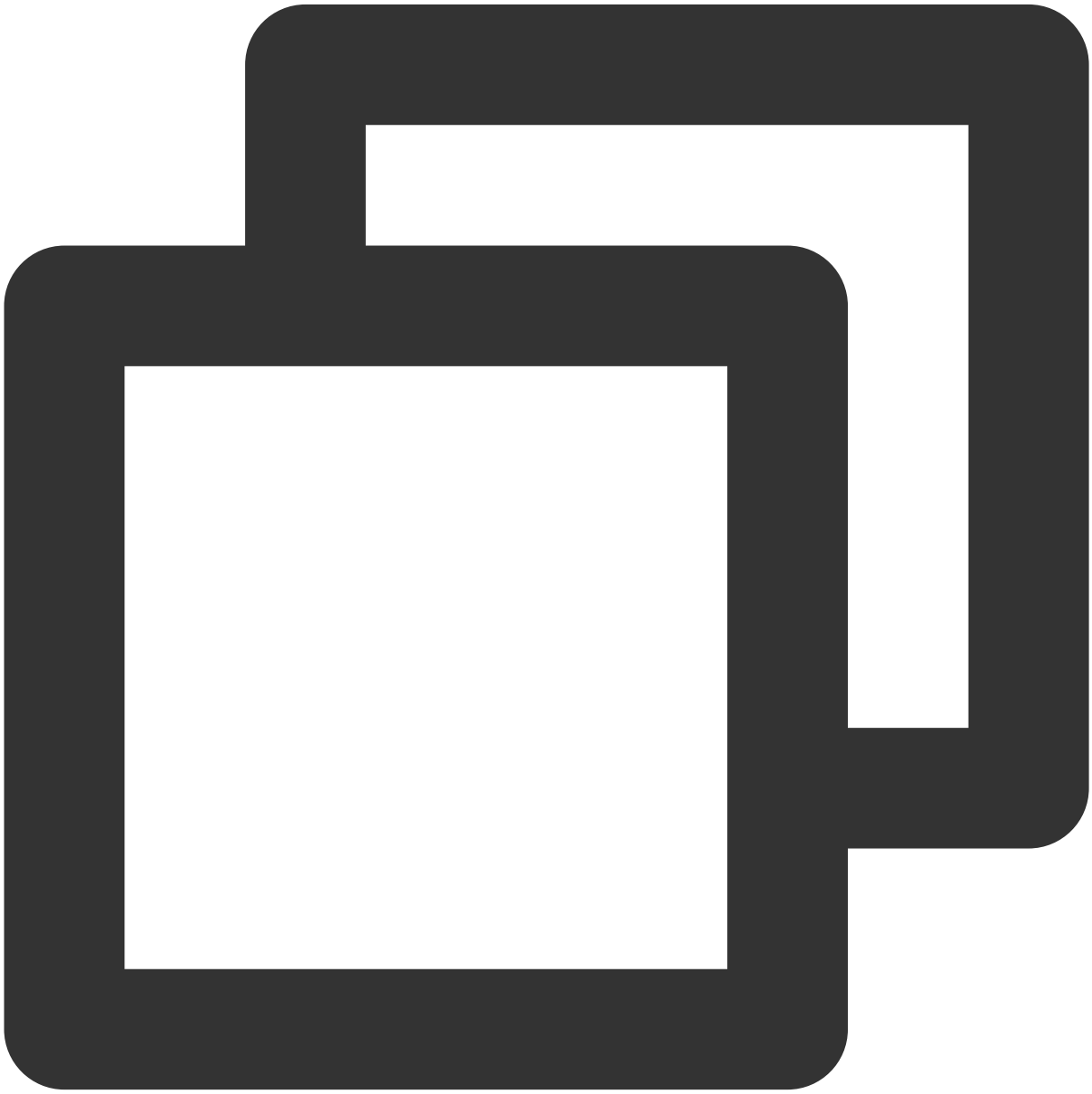
```
public abstract class V2TIMAdvancedMsgListener {
    // 收到新消息
    public void onRecvNewMessage(V2TIMMessage msg) {}
```

```
    // C2C 对端用户会话已读通知（对端用户调用 markC2CMessageAsRead，自己会收到该通知）
    public void onRecvC2CReadReceipt(List<V2TIMMessageReceipt> receiptList) {}

    // 消息已读回执通知（如果自己发送的消息支持已读回执，消息接收端调用 sendMessageReadReceipt
    public void onRecvMessageReadReceipts(List<V2TIMMessageReceipt> receiptList) {}

    // 收到消息撤回的通知
    public void onRecvMessageRevoked(String msgID) {}

    // 消息内容被修改
    public void onRecvMessageModified(V2TIMMessage msg) {}
}
```



```
/// 高级消息监听器
@protocol V2TIMAdvancedMsgListener <NSObject>
@optional
/// 收到新消息
- (void)onRecvNewMessage:(V2TIMMessage *)msg;

/// 消息已读回执通知（如果自己发的消息支持已读回执，消息接收端调用了 sendMessageReadReceipts 接
- (void)onRecvMessageReadReceipts:(NSArray<V2TIMMessageReceipt *> *)receiptList;

/// C2C 对端用户会话已读通知（如果对端用户调用 markC2CMessageAsRead 接口，自己会收到该通知）
- (void)onRecvC2CReadReceipt:(NSArray<V2TIMMessageReceipt *> *)receiptList;
```

```
/// 收到消息撤回
- (void)onRecvMessageRevoked:(NSString *)msgID;

/// 消息内容被修改
- (void)onRecvMessageModified:(V2TIMMessage *)msg;
@end
```



```
class AdvancedMsgListener final : public V2TIMAdvancedMsgListener {
public:
    AdvancedMsgListener() = default;
```

```
    ~AdvancedMsgListener() override = default;

    // 收到新消息
    void OnRecvNewMessage(const V2TIMMessage& message) override {}

    // C2C 对端用户会话已读通知（对端用户调用 markC2CMessageAsRead，自己会收到该通知）
    void OnRecvC2CReadReceipt(const V2TIMMessageReceiptVector& receiptList) overrid

    // 消息已读回执通知（如果自己发送的消息支持已读回执，消息接收端调用
    // sendMessageReadReceipts，自己会收到该通知）
    void OnRecvMessageReadReceipts(const V2TIMMessageReceiptVector& receiptList) ov

    // 收到消息撤回的通知
    void OnRecvMessageRevoked(const V2TIMString& messageID) override {}

    // 消息内容被修改
    void OnRecvMessageModified(const V2TIMMessage& message) override {}
};
```

移除监听器

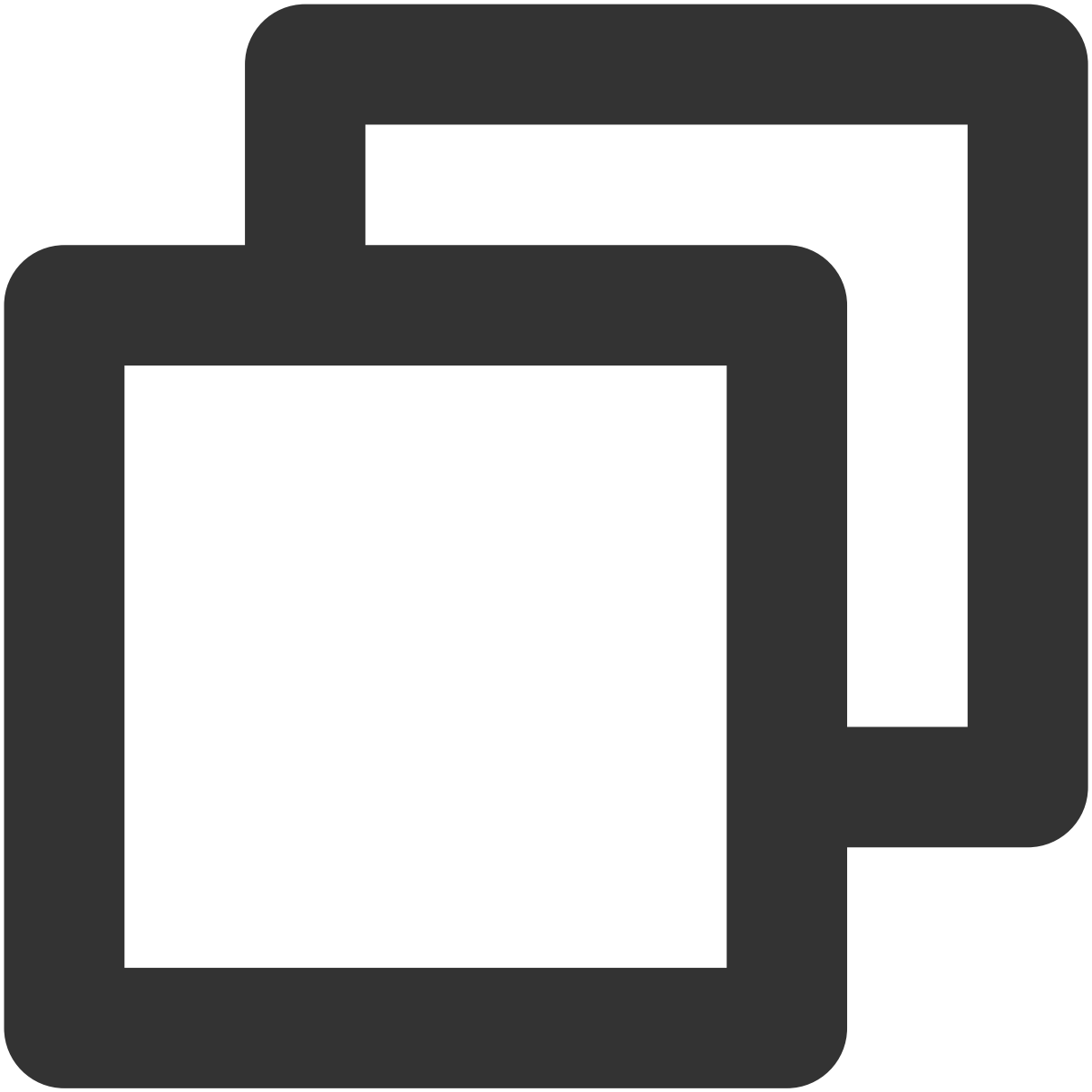
如果想停止接收消息，接收方可调用 `removeAdvancedMsgListener` (Android / iOS & Mac / Windows) 移除高级消息监听器。

示例代码如下：

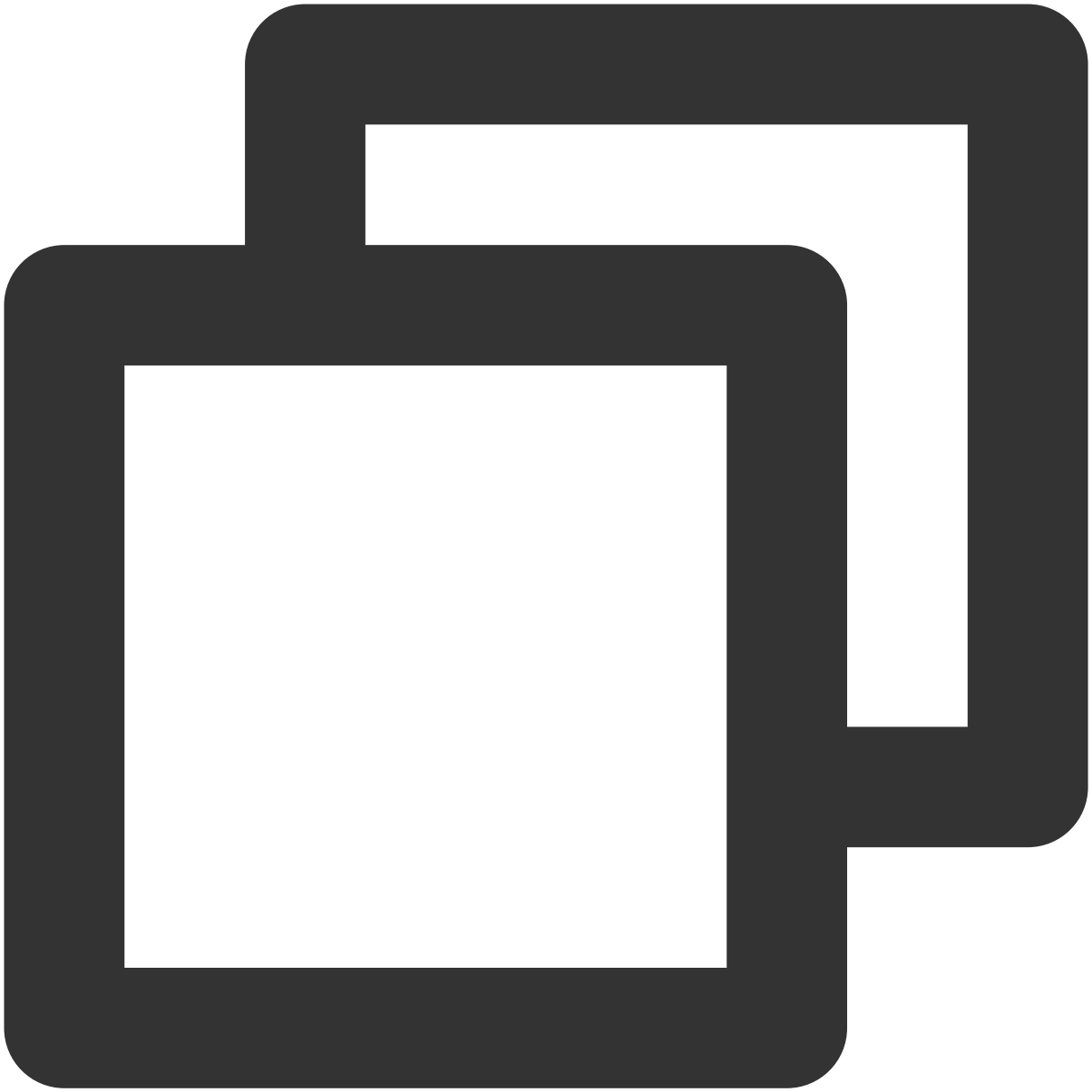
Android

iOS & Mac

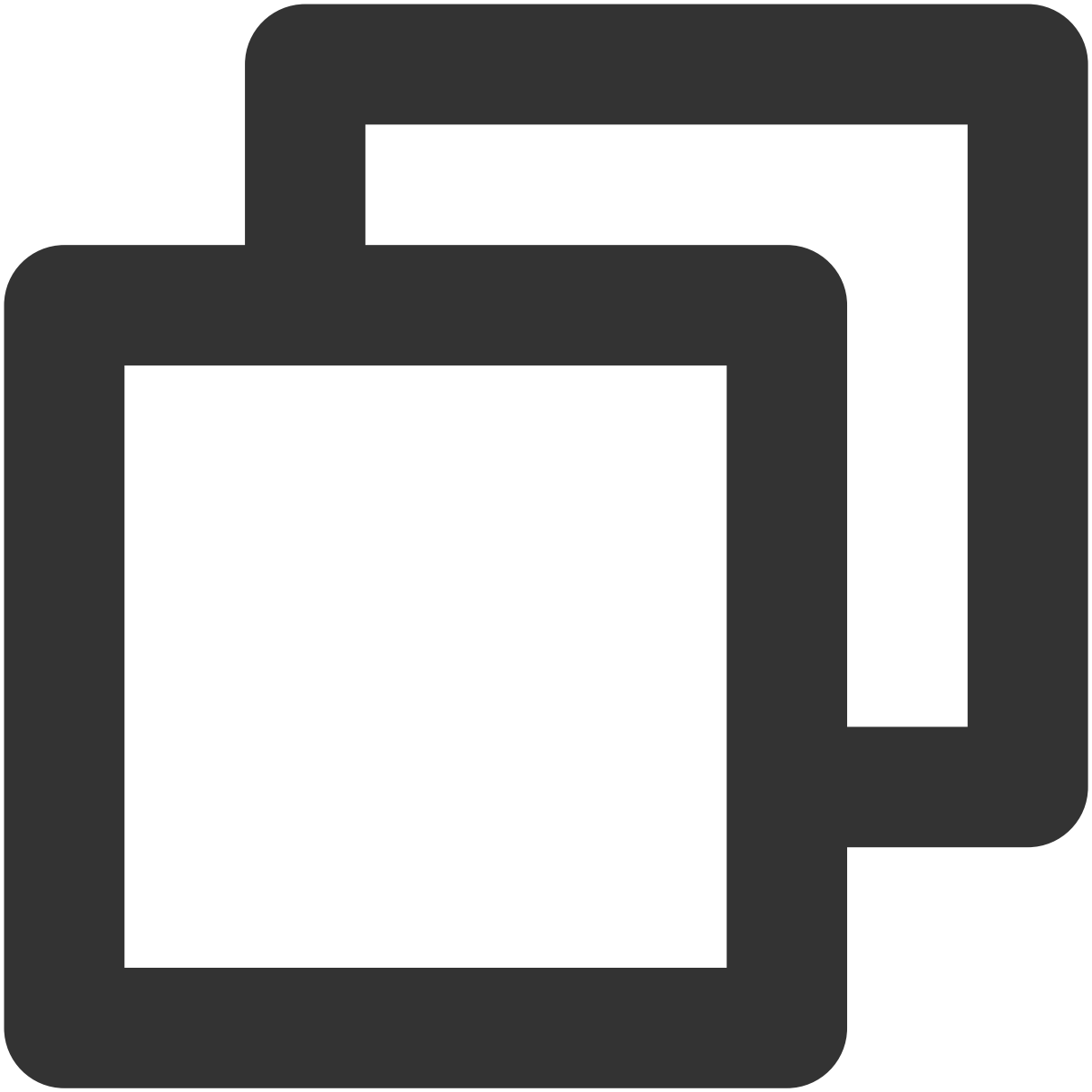
Windows



```
V2TIMManager.getMessageManager().removeAdvancedMsgListener(advancedMsgListener);
```



```
// self 为 id<V2TIMAdvancedMsgListener>
[[V2TIMManager sharedInstance] removeAdvancedMsgListener:self];
```



```
class AdvancedMsgListener final : public V2TIMAdvancedMsgListener {
    // 成员 ...
};

// advancedMsgListener 是 AdvancedMsgListener 的实例
V2TIMManager::GetInstance()->GetMessageManager()->RemoveAdvancedMsgListener(&advanc
```

接收文本消息

使用简单消息监听器接收

单聊文本消息

接收方使用简单消息监听器接收单聊文本消息，需要以下几步：

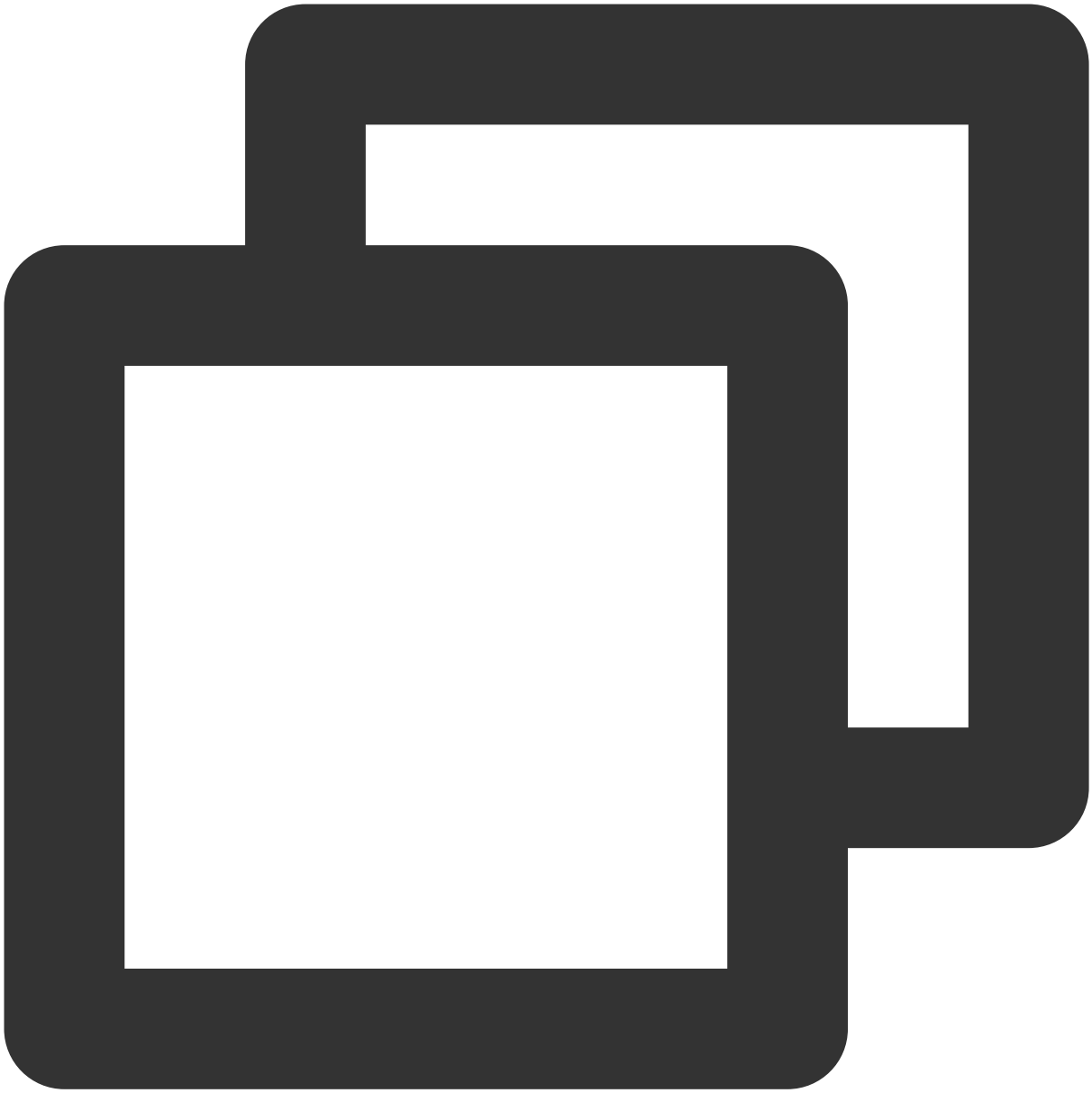
1. 调用 `addSimpleMsgListener` 设置事件监听器。
2. 监听 `onRecvC2CTextMessage` (Android / iOS & Mac / Windows) 回调，在其中接收文本消息。
3. 希望停止接收消息，调用 `removeSimpleMsgListener` 移除监听。该步骤不是必须的，客户可以按照业务需求调用。

代码示例如下：

Android

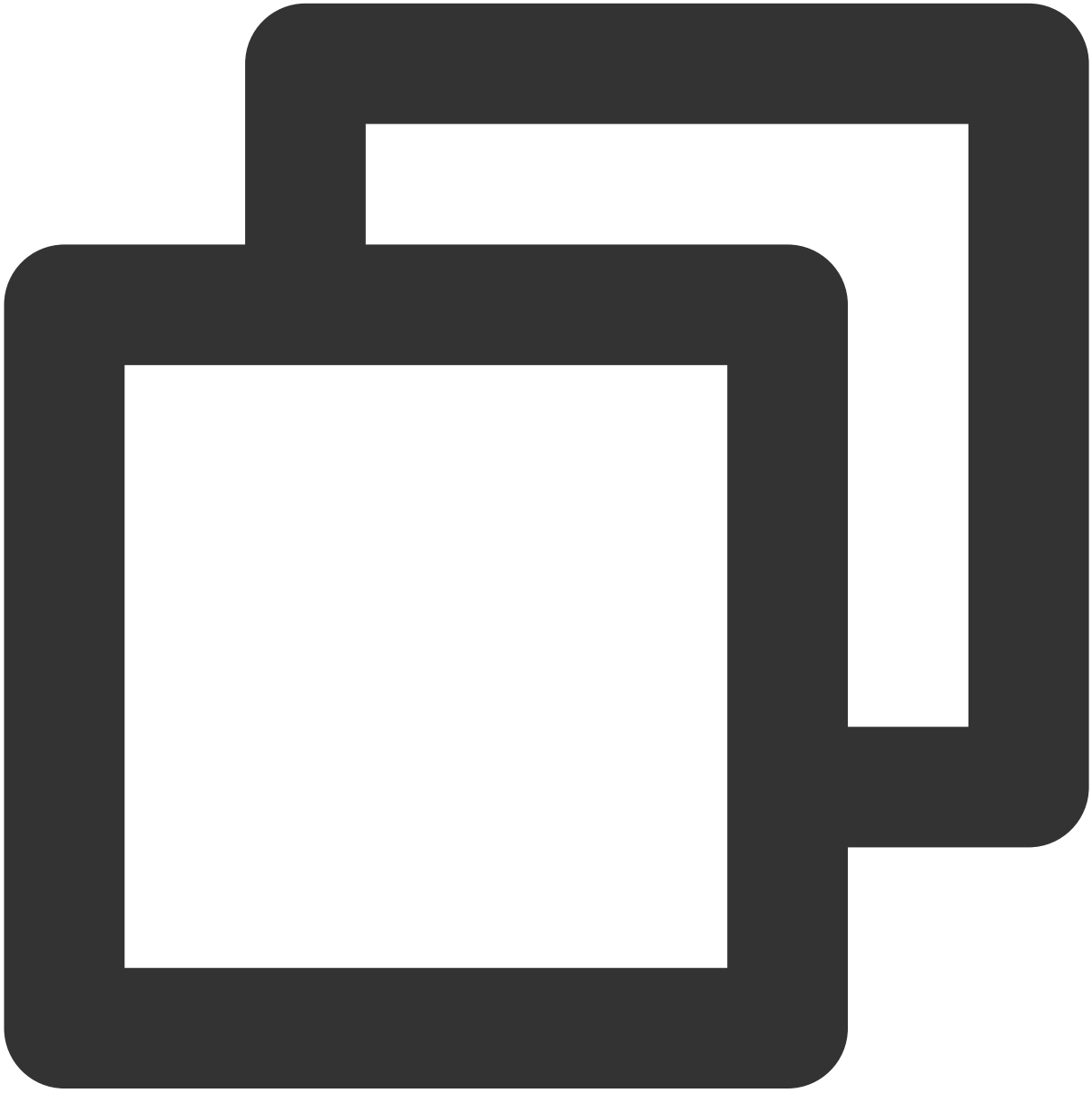
iOS & Mac

Windows



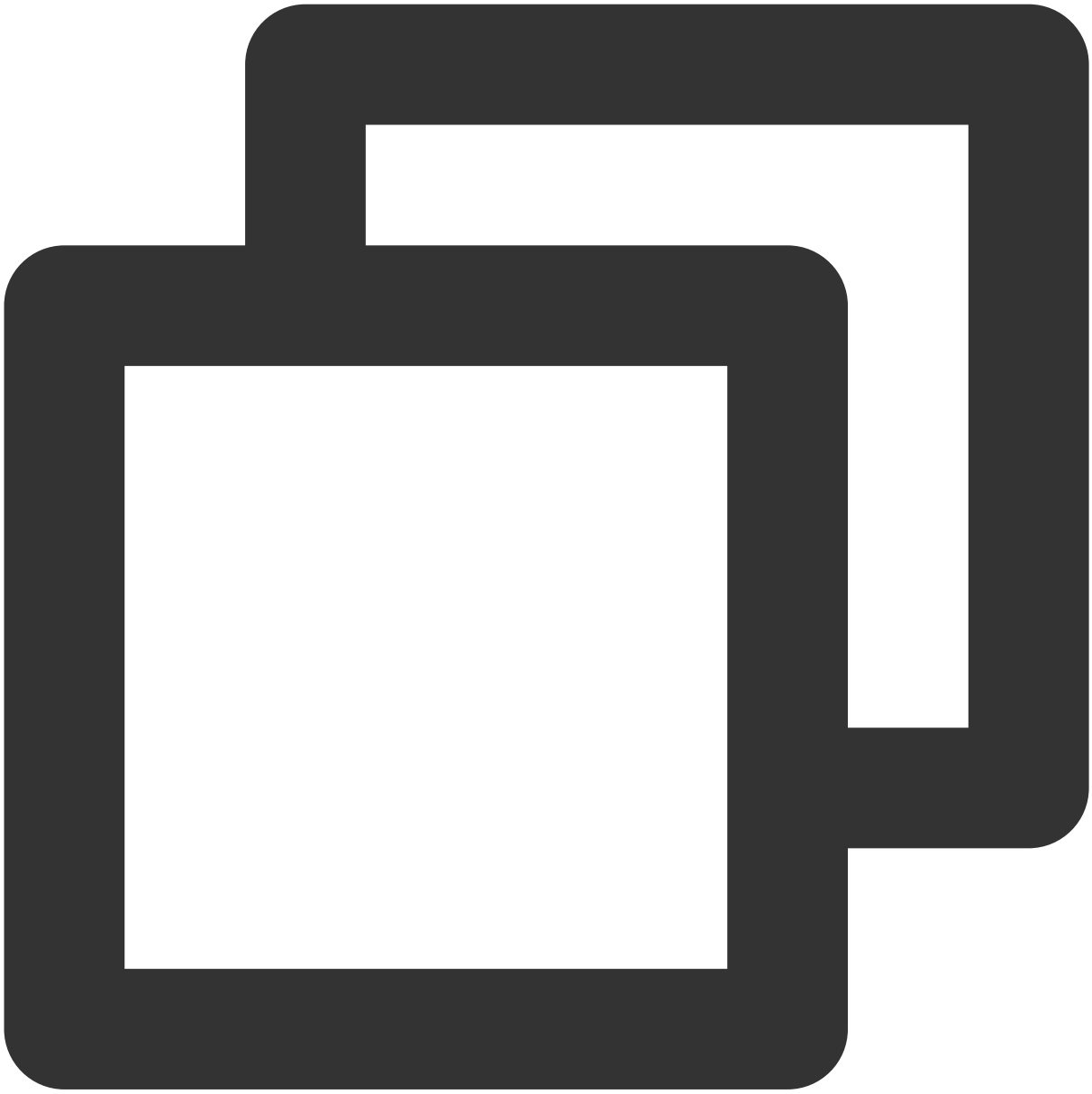
```
// 设置事件监听器
V2TIMManager.getInstance().addSimpleMsgListener(simpleMsgListener);

// 接收单聊文本消息
/**
 * 收到 C2C 文本消息
 *
 * @param msgID 消息唯一标识
 * @param sender 发送方信息
 * @param text 发送内容
 */
```

```
public void onRecvC2CTextMessage(String msgID, V2TIMUserInfo sender, String text) {
    // 可解析消息并展示到 UI
}
```



```
// 设置事件监听器
[[V2TIMManager sharedInstance] addSimpleMsgListener:self];

/// 接收单聊文本消息
/// @param msgID 消息 Id
/// @param info 发送者信息
/// @param text 文本内容
```

```
- (void)onRecvC2CTextMessage:(NSString *)msgID sender:(V2TIMUserInfo *)info text:(N
    // 可解析消息并展示到 UI
}
```



```
class SimpleMsgListener final : public V2TIMSimpleMsgListener {
public:
    /**
     * 收到 C2C 文本消息
     *
     * @param msgID 消息唯一标识
     * @param sender 发送方信息
```

```
     * @param text 发送内容
     */
    void OnRecvC2CTextMessage(const V2TIMString& msgID, const V2TIMUserFullInfo& se
                              const V2TIMString& text) override {
        // 可以解析消息并展示到 UI，比如：
        std::cout << "text:" << std::string{text.CString(), text.Size()} << std::en
    }
    // 其他成员 ...
};

// 添加基本消息的事件监听器，注意在移除监听器之前需要保持 simpleMsgListener 的生命期，以免接收不
SimpleMsgListener simpleMsgListener;
V2TIMManager::GetInstance()->AddSimpleMsgListener(&simpleMsgListener);
```

群聊文本消息

接收方使用简单消息监听器接收群聊文本消息，需要以下几步：

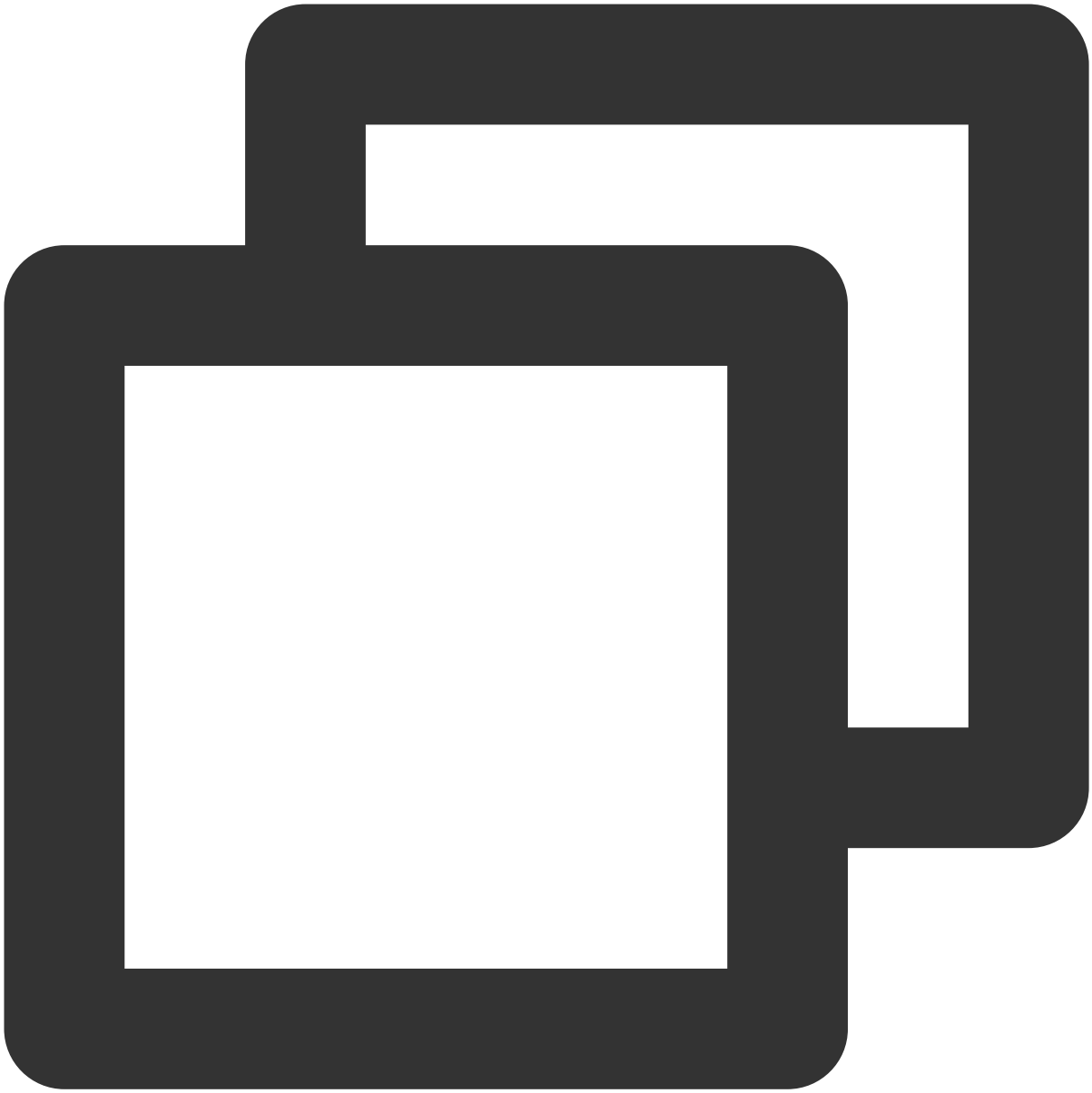
1. 调用 `addSimpleMsgListener` 设置事件监听器。
2. 监听 `onRecvGroupTextMessage` (Android / iOS & Mac / Windows) 回调，在其中接收文本消息。
3. 希望停止接收消息，调用 `removeSimpleMsgListener` 移除监听。该步骤不是必须的，客户可以按照业务需求调用。

代码示例如下：

Android

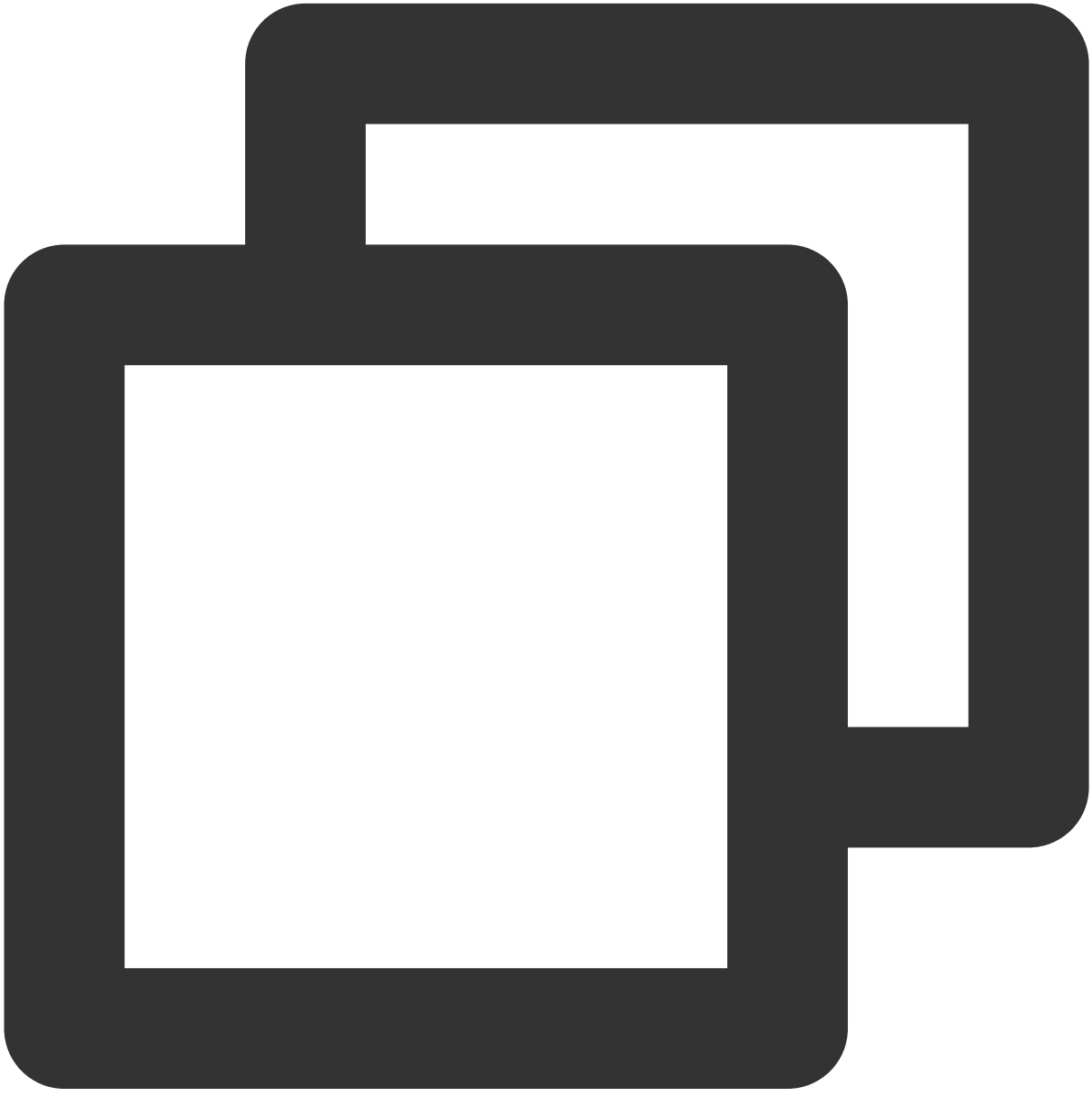
iOS & Mac

Windows



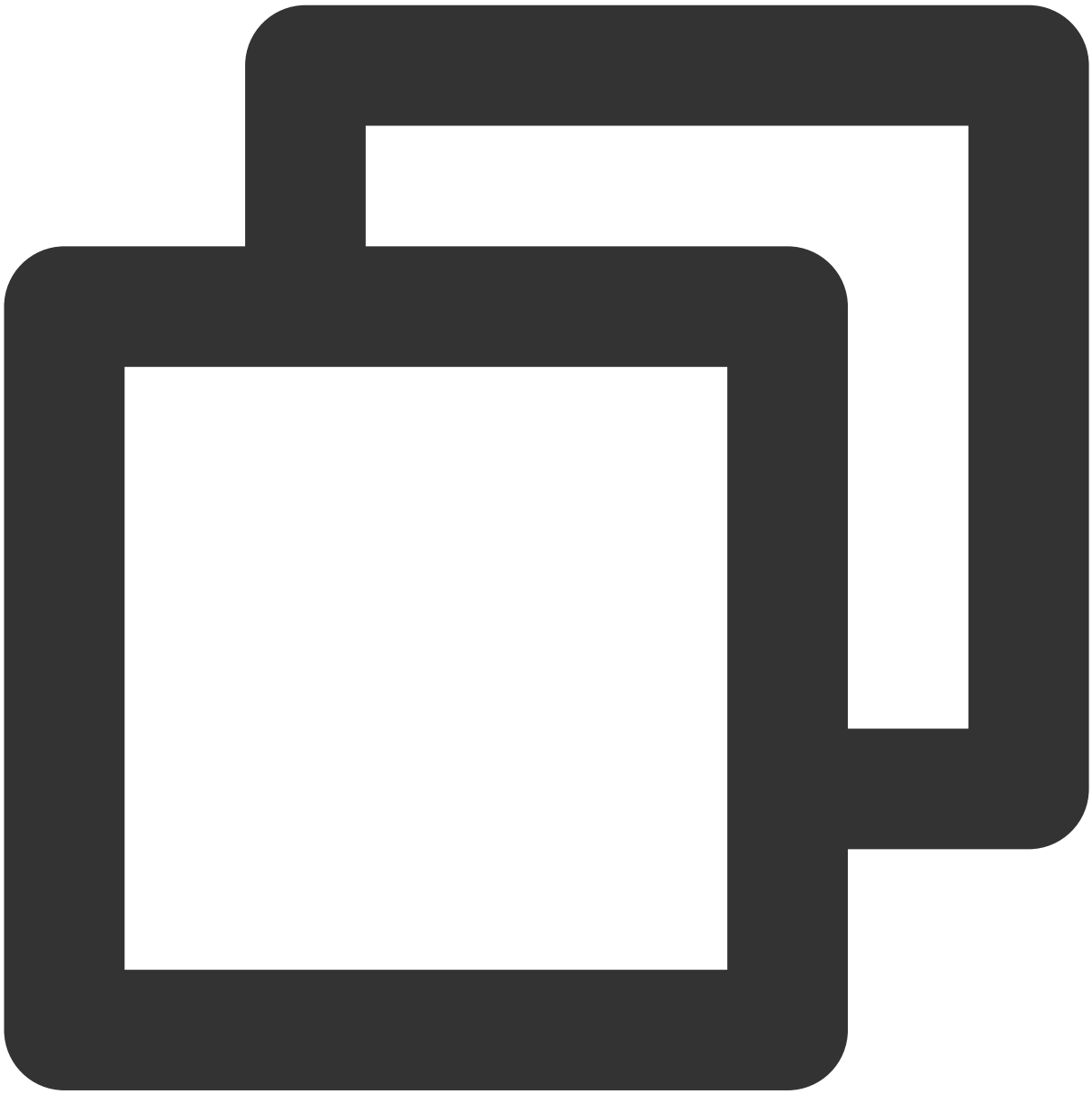
```
// 设置事件监听器
V2TIMManager.getInstance().addSimpleMsgListener(simpleMsgListener);

// 接收群聊文本消息
/**
 * 收到群文本消息
 *
 * @param msgID 消息唯一标识
 * @param groupID 群 ID
 * @param sender 发送方群成员信息
 * @param text 发送内容
```

```
 */
public void onRecvGroupTextMessage(String msgID, String groupID, V2TIMGroupMemberIn
    // 可解析消息并展示到 UI
}
```



```
// 设置事件监听器
[[V2TIMManager sharedInstance] addSimpleMsgListener:self];

/// 接收群聊文本消息
/// @param msgID 消息 Id
/// @param groupID 群组 ID
```

```
/// @param info 发送者信息
/// @param text 文本内容
- (void)onRecvGroupTextMessage:(NSString *)msgID groupID:(NSString *)groupID sender
    // 可解析消息并展示到 UI
}
```



```
class SimpleMsgListener final : public V2TIMSimpleMsgListener {
public:
    /**
     * 收到群文本消息
     *
```

```
     * @param msgID 消息唯一标识
     * @param groupID 群 ID
     * @param sender 发送方群成员信息
     * @param text 发送内容
     */
    void OnRecvGroupTextMessage(const V2TIMString& msgID, const V2TIMString& groupI
                                const V2TIMGroupMemberFullInfo& sender, const V2TIM
        // 可以解析消息并展示到 UI，比如：
        std::cout << "text:" << std::string{text.CString(), text.Size()} << std::en
    }
    // 其他成员 ...
};

// 添加基本消息的事件监听器，注意在移除监听器之前需要保持 simpleMsgListener 的生命期，以免接收不
SimpleMsgListener simpleMsgListener;
V2TIMManager::GetInstance()->AddSimpleMsgListener(&simpleMsgListener);
```

使用高级消息监听器接收

接收方使用高级消息监听器接收单聊、群聊文本消息，需要以下几步：

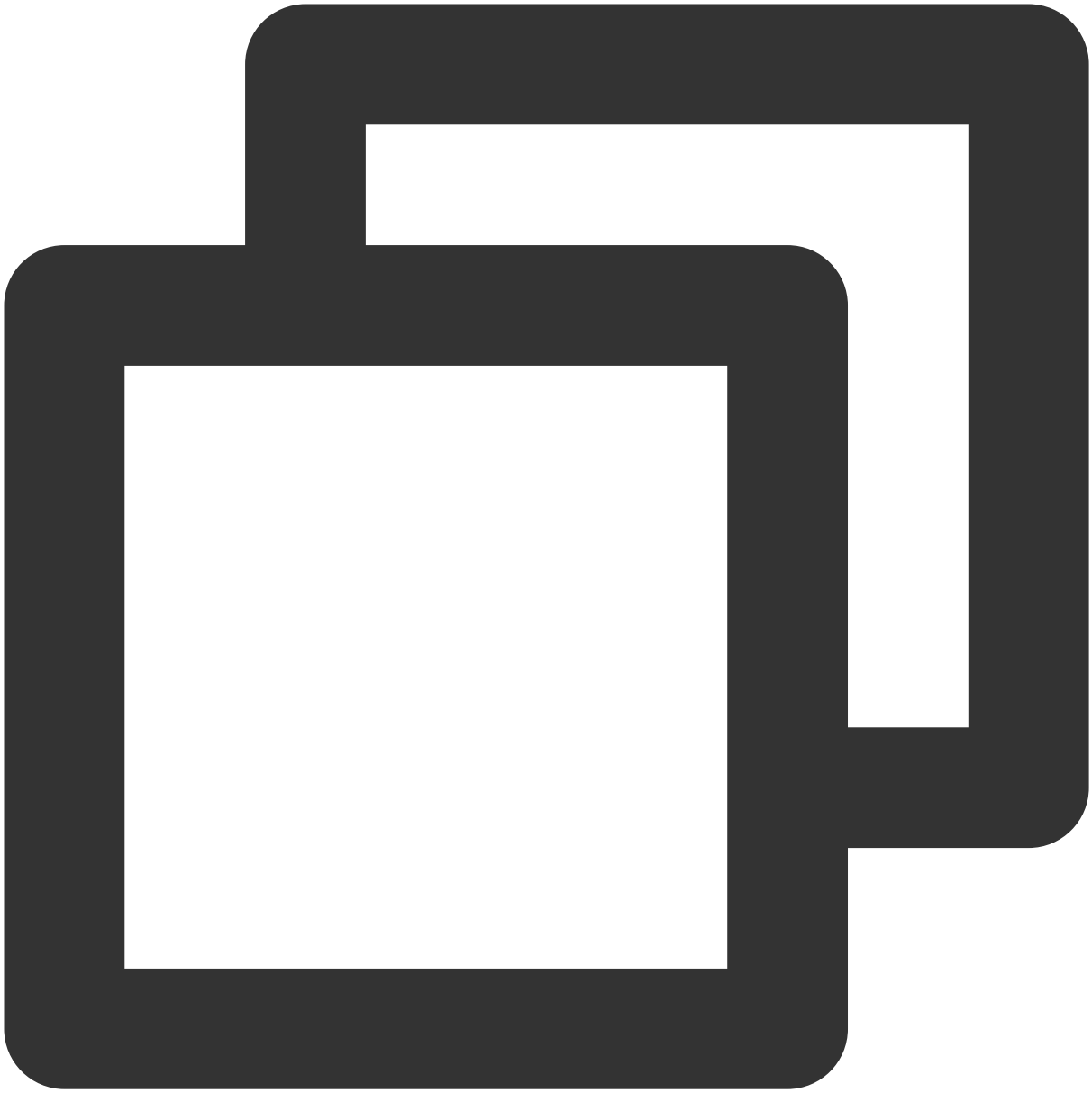
1. 调用 `addAdvancedMsgListener` 设置事件监听器。
2. 监听 `onRecvNewMessage` (Android / iOS & Mac / Windows) 回调，在其中接收文本消息。
3. 希望停止接收消息，调用 `removeAdvancedMsgListener` 移除监听。该步骤不是必须的，客户可以按照业务需求调用。

代码示例如下：

Android

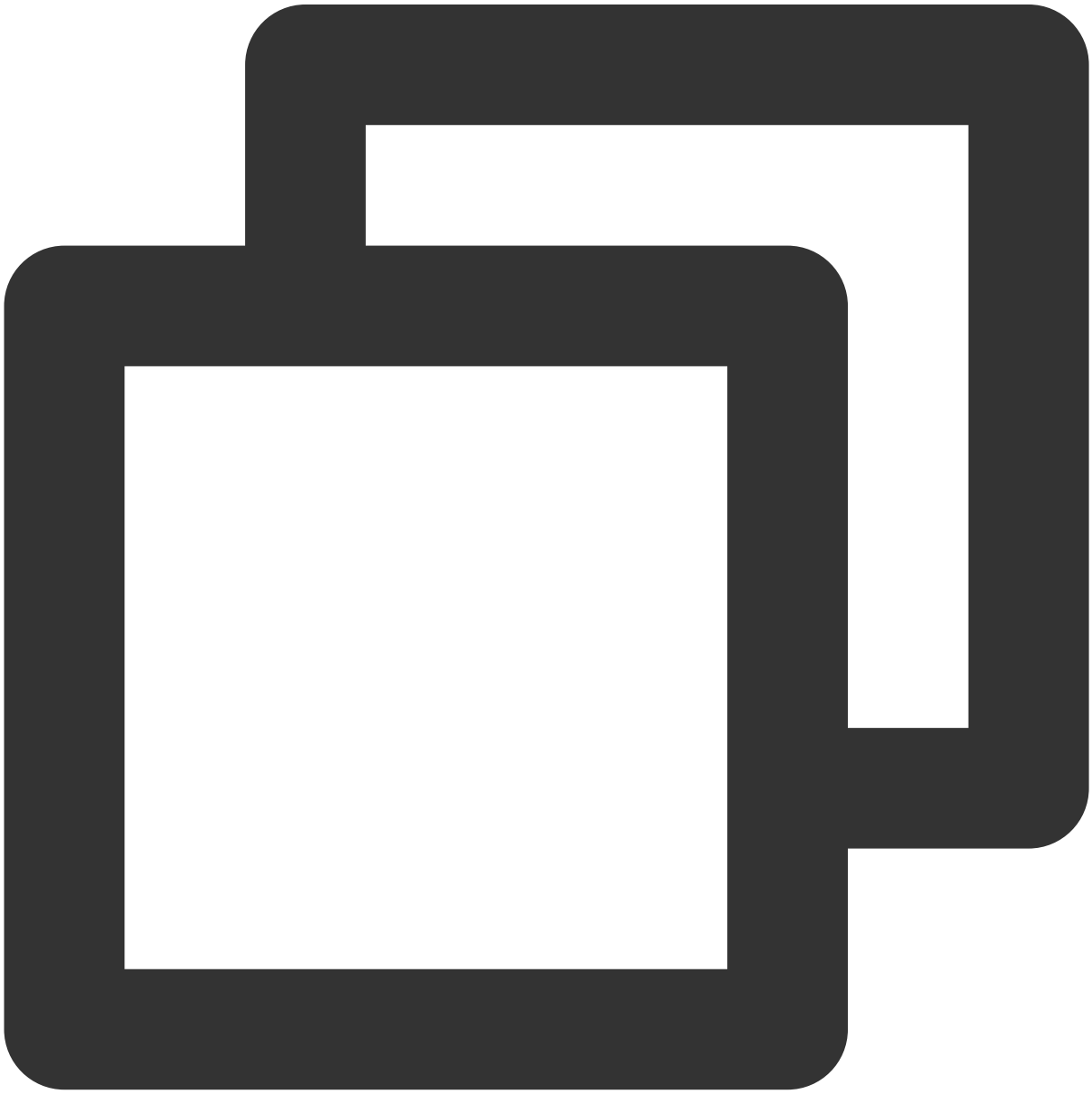
iOS & Mac

Windows

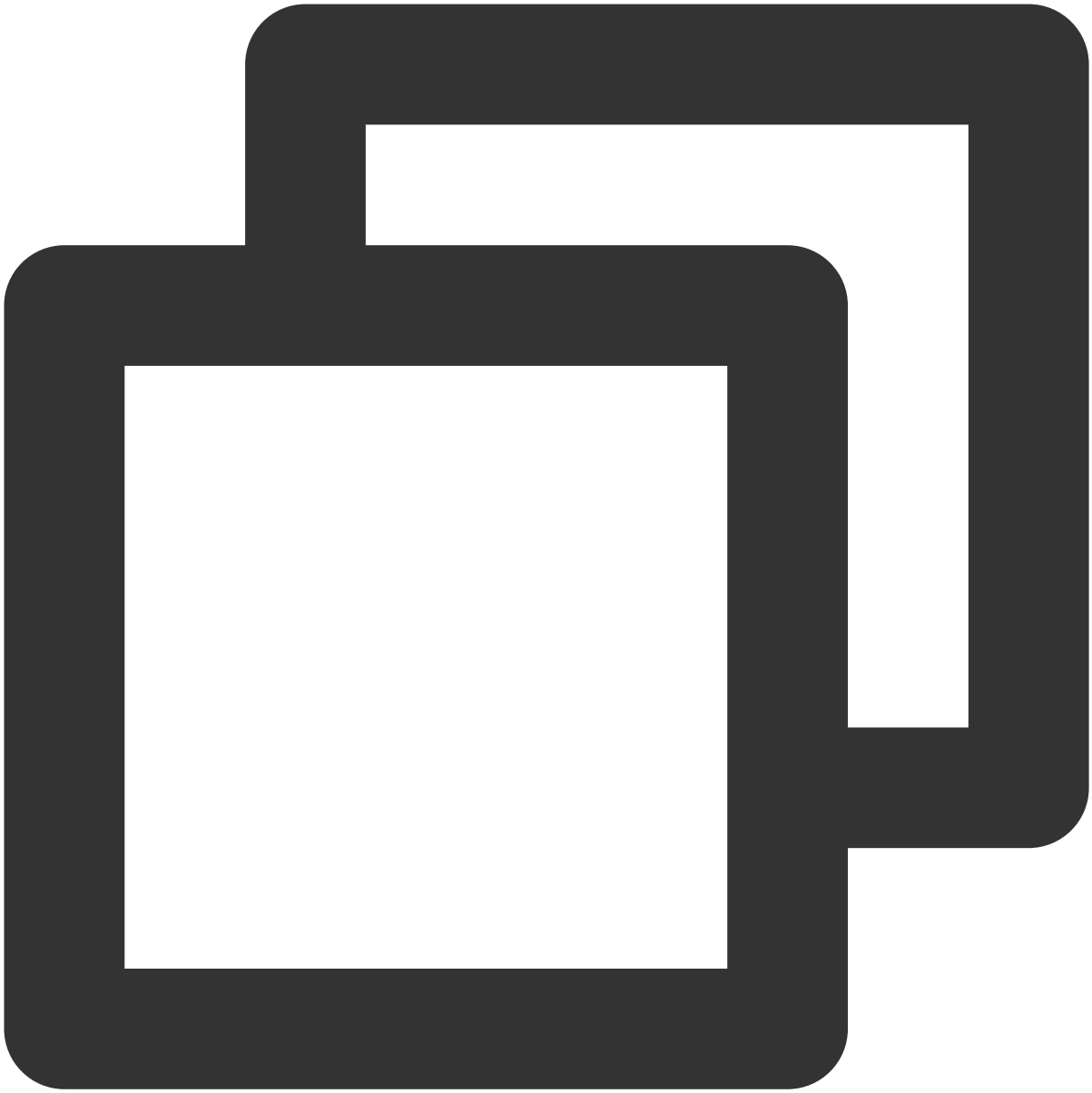


```
// 设置事件监听器
V2TIMManager.getMessageManager().addAdvancedMsgListener(advancedMsgListener);

/**
 * 收到新消息
 * @param msg 消息
 */
public void onRecvNewMessage(V2TIMMessage msg) {
    // 解析出 groupID 和 userID
    String groupID = msg.getGroupID();
    String userID = msg.getUserID();
```

```
    // 判断当前是单聊还是群聊：
    // 如果 groupID 不为空，表示此消息为群聊；如果 userID 不为空，表示此消息为单聊

    // 解析出 msg 中的文本消息
    if (msg.getElemType() == V2TIMMessage.V2TIM_ELEM_TYPE_TEXT) {
        V2TIMTextElem textElem = msg.getTextElem();
        String text = textElem.getText();
        Log.i("onRecvNewMessage", "text:" + text);
    }
}
```



```
// 设置事件监听器
[[V2TIMManager sharedInstance] addAdvancedMsgListener:self];

/// 接收消息
/// @param msg 消息对象
- (void)onRecvNewMessage:(V2TIMMessage *)msg {
    // 解析出 groupID 和 userID
    NSString *groupID = msg.groupID;
    NSString *userID = msg.userID;

    // 判断当前是单聊还是群聊：
    // 如果 groupID 不为空，表示此消息为群聊；如果 userID 不为空，表示此消息为单聊

    // 解析出 msg 中的文本消息
    if (msg.elemType == V2TIM_ELEM_TYPE_TEXT) {
        V2TIMTextElem *textElem = msg.textElem;
        NSString *text = textElem.text;
        NSLog(@"onRecvNewMessage, text: %@", text);
    }
}
```



```
class AdvancedMsgListener final : public V2TIMAdvancedMsgListener {
public:
    /**
     * 收到新消息
     *
     * @param message 消息
     */
    void OnRecvNewMessage(const V2TIMMessage& message) override {
        // 解析出 groupID 和 userID
        V2TIMString groupID = message.groupID;
        V2TIMString userID = message.userID;
```

```
        // 判断当前是单聊还是群聊：
        // 如果 groupID 不为空，表示此消息为群聊；如果 userID 不为空，表示此消息为单聊

        // 解析出 message 中的文本消息
        if (message.elemList.Size() == 1) {
            V2TIMElem* elem = message.elemList[0];
            if (elem->elemType == V2TIMElemType::V2TIM_ELEM_TYPE_TEXT) {
                // 文本消息
                auto textElem = static_cast<V2TIMTextElem*>(elem);
                // 消息文本
                V2TIMString text = textElem->text;
                // 可以解析消息并展示到 UI，比如：
                std::cout << "text:" << std::string{text.CString(), text.Size()} <<
            }
        }
    }
    // 其他成员 ...
};

// 添加高级消息的事件监听器，注意在移除监听器之前需要保持 advancedMsgListener 的生命期，以免接
AdvancedMsgListener advancedMsgListener;
V2TIMManager::GetInstance()->GetMessageManager()->AddAdvancedMsgListener(&advancedM
```

接收自定义消息

使用简单消息监听器接收

单聊自定义消息

接收方使用简单消息监听器接收单聊自定义消息，需要以下几步：

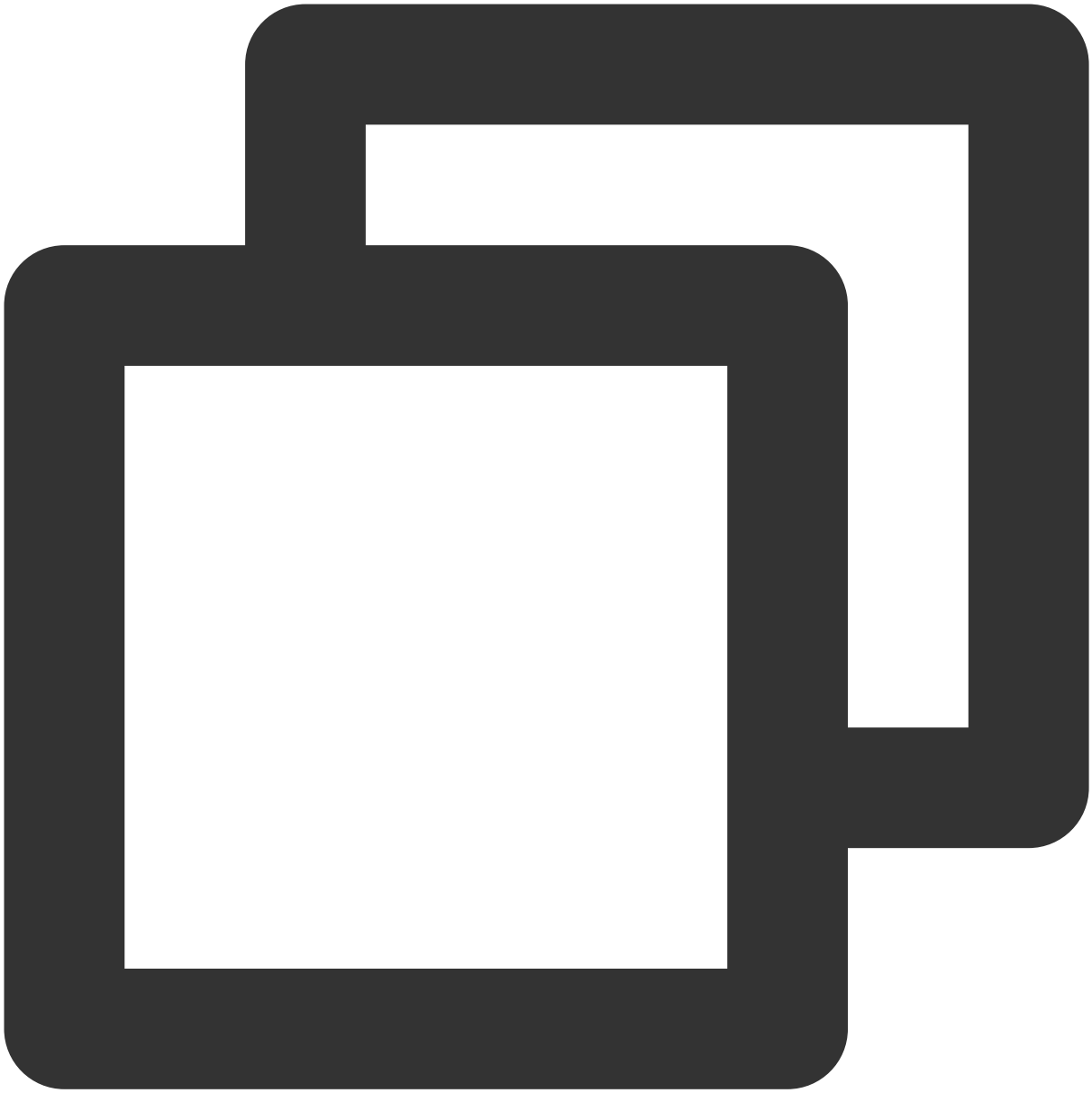
1. 调用 `addSimpleMsgListener` 设置事件监听器。
2. 监听 `onRecvC2CCustomMessage` (Android / iOS & Mac / Windows) 回调，在其中接收单聊自定义消息。
3. 希望停止接收消息，调用 `removeSimpleMsgListener` 移除监听。该步骤不是必须的，客户可以按照业务需求调用。

代码示例如下：

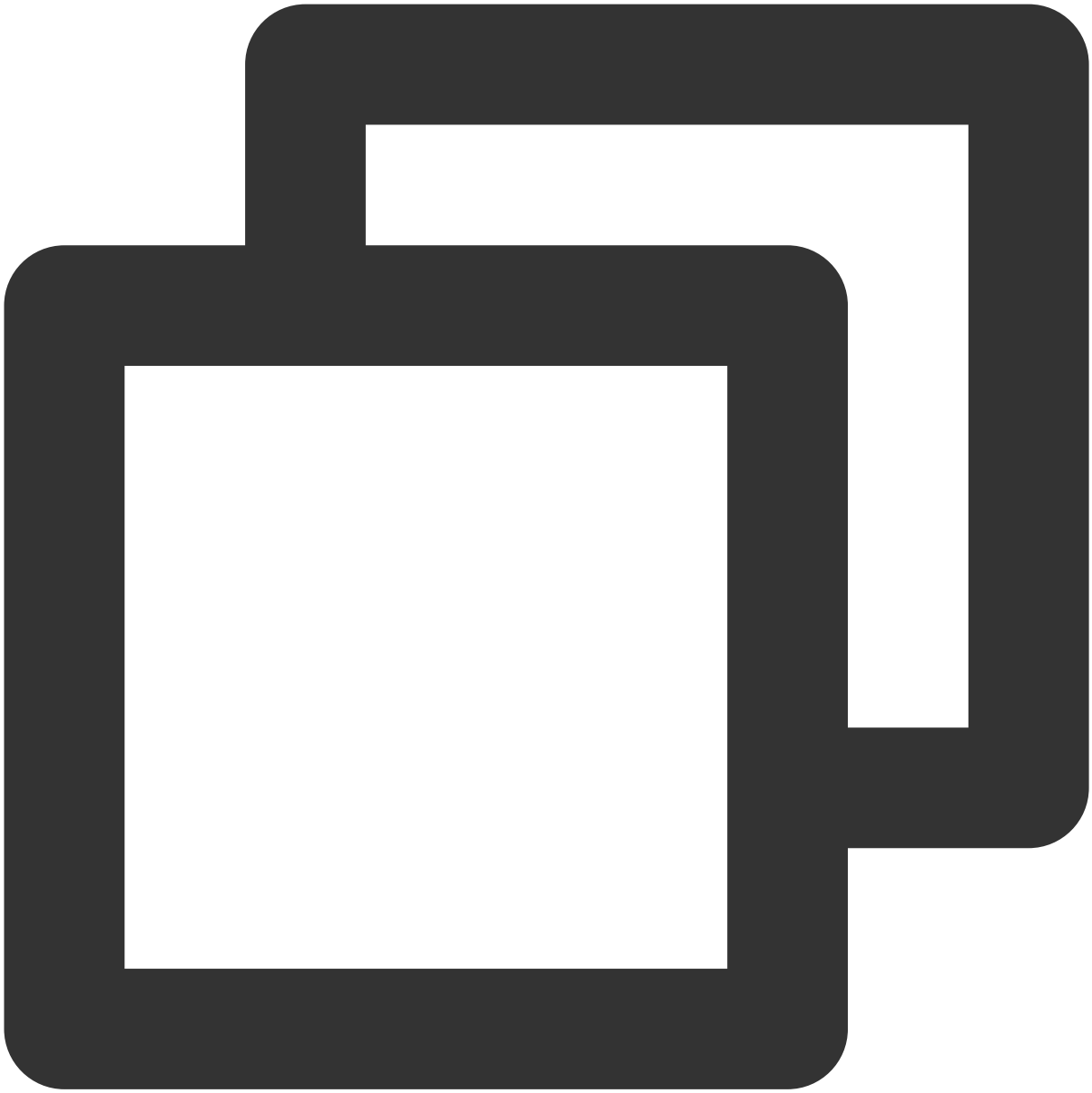
Android

iOS & Mac

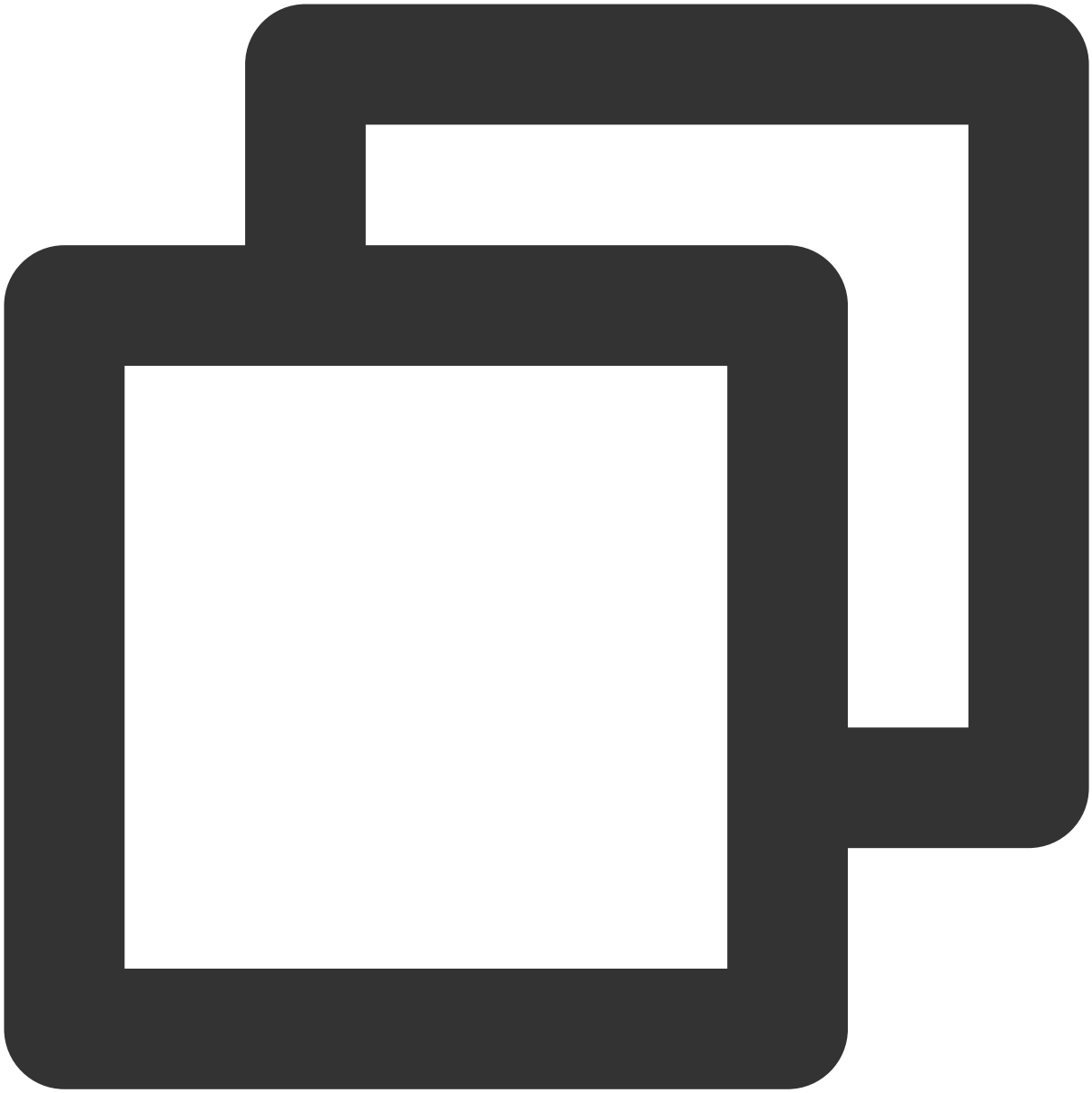
Windows



```
/**
 * 接收单聊自定义消息
 * @param msgID 消息 ID
 * @param sender 发送方信息
 * @param customData 发送内容
 */
public void onRecvC2CCustomMessage(String msgID, V2TIMUserInfo sender, byte[] custo
    Log.i("onRecvC2CCustomMessage", "msgID:" + msgID + ", from:" + sender.getNickNa
}
```



```
/// 接收单聊自定义消息
/// @param msgID 消息 ID
/// @param info 发送者信息
/// @param data 自定义消息二进制内容
- (void)onRecvC2CCustomMessage:(NSString *)msgID sender:(V2TIMUserInfo *)info custo
    NSLog(@"onRecvC2CCustomMessage, msgID: %@, sender: %@, customData: %@", msgID,
}
```



```
class SimpleMsgListener final : public V2TIMSimpleMsgListener {
public:
    /**
     * 收到 C2C 自定义（信令）消息
     *
     * @param msgID 消息唯一标识
     * @param sender 发送方信息
     * @param customData 发送内容
     */
    void OnRecvC2CCustomMessage(const V2TIMString& msgID, const V2TIMUserFullInfo&
                                const V2TIMBuffer& customData) override {
```

```
        // 可以解析消息并展示到 UI，比如当 customData 为文本时：
        std::cout << "customData:"
                  << std::string{reinterpret_cast<const char*>(customData.Data()),
                  << std::endl;
    }
    // 其他成员 ...
};

// 添加基本消息的事件监听器，注意在移除监听器之前需要保持 simpleMsgListener 的生命期，以免接收不
SimpleMsgListener simpleMsgListener;
V2TIMManager::GetInstance()->AddSimpleMsgListener(&simpleMsgListener);
```

群聊自定义消息

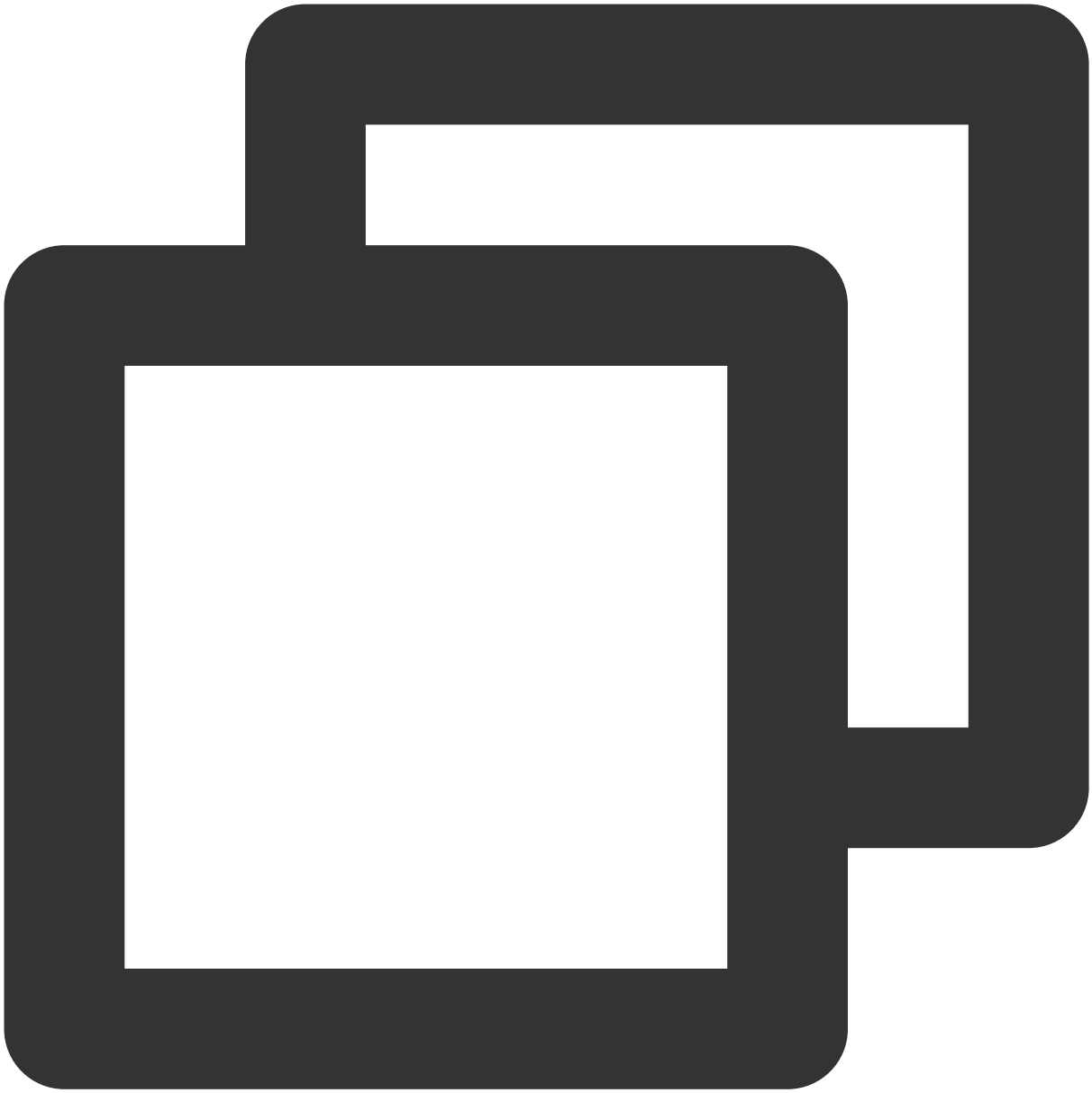
接收方使用简单消息监听器接收群聊自定义消息，需要以下几步：

1. 调用 `addSimpleMsgListener` 设置事件监听器。
2. 监听 `onRecvGroupCustomMessage` (Android / iOS & Mac / Windows) 回调，在其中接收群聊自定义消息。
3. 希望停止接收消息，调用 `removeSimpleMsgListener` 移除监听。该步骤不是必须的，客户可以按照业务需求调用。

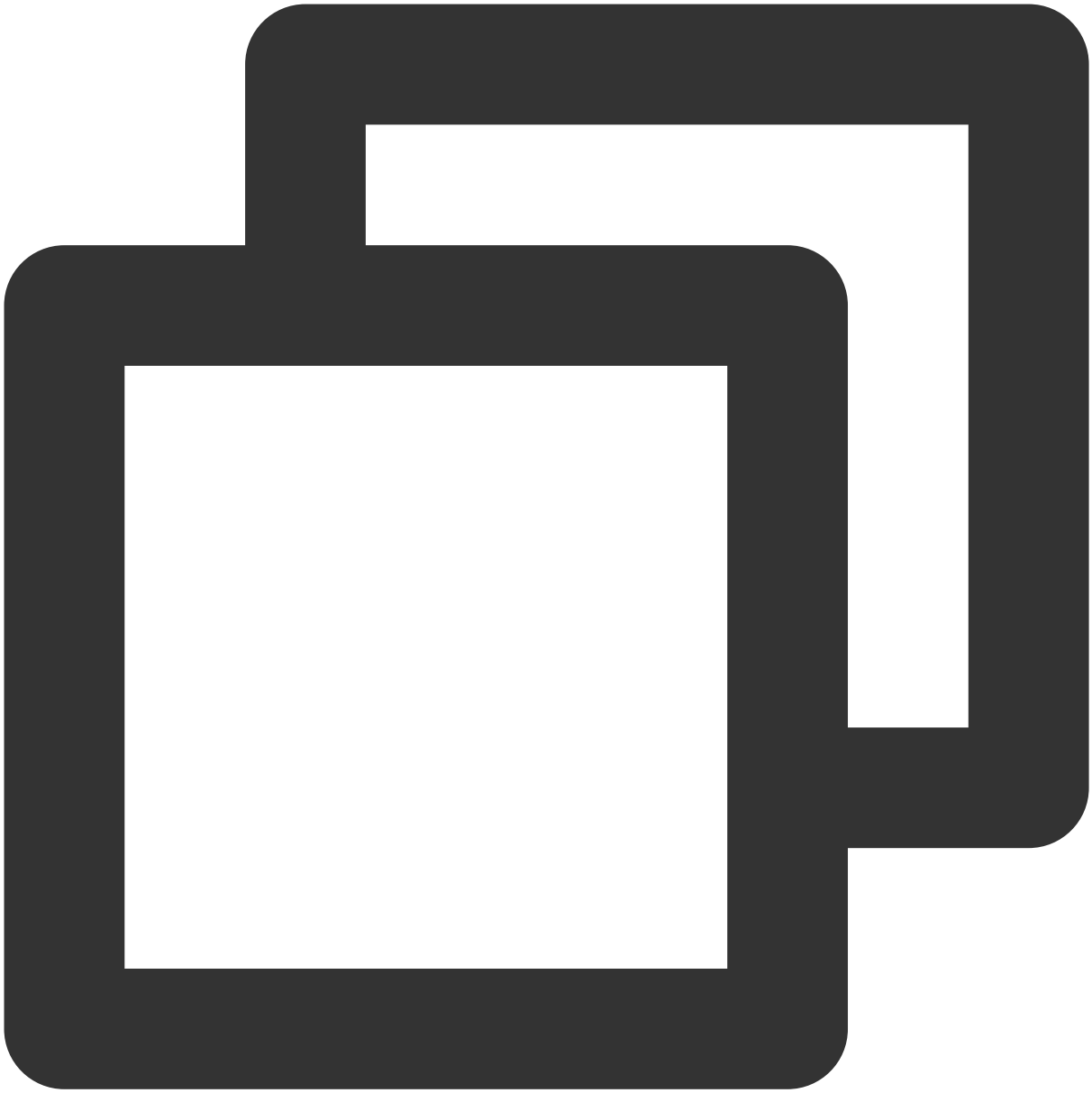
Android

iOS & Mac

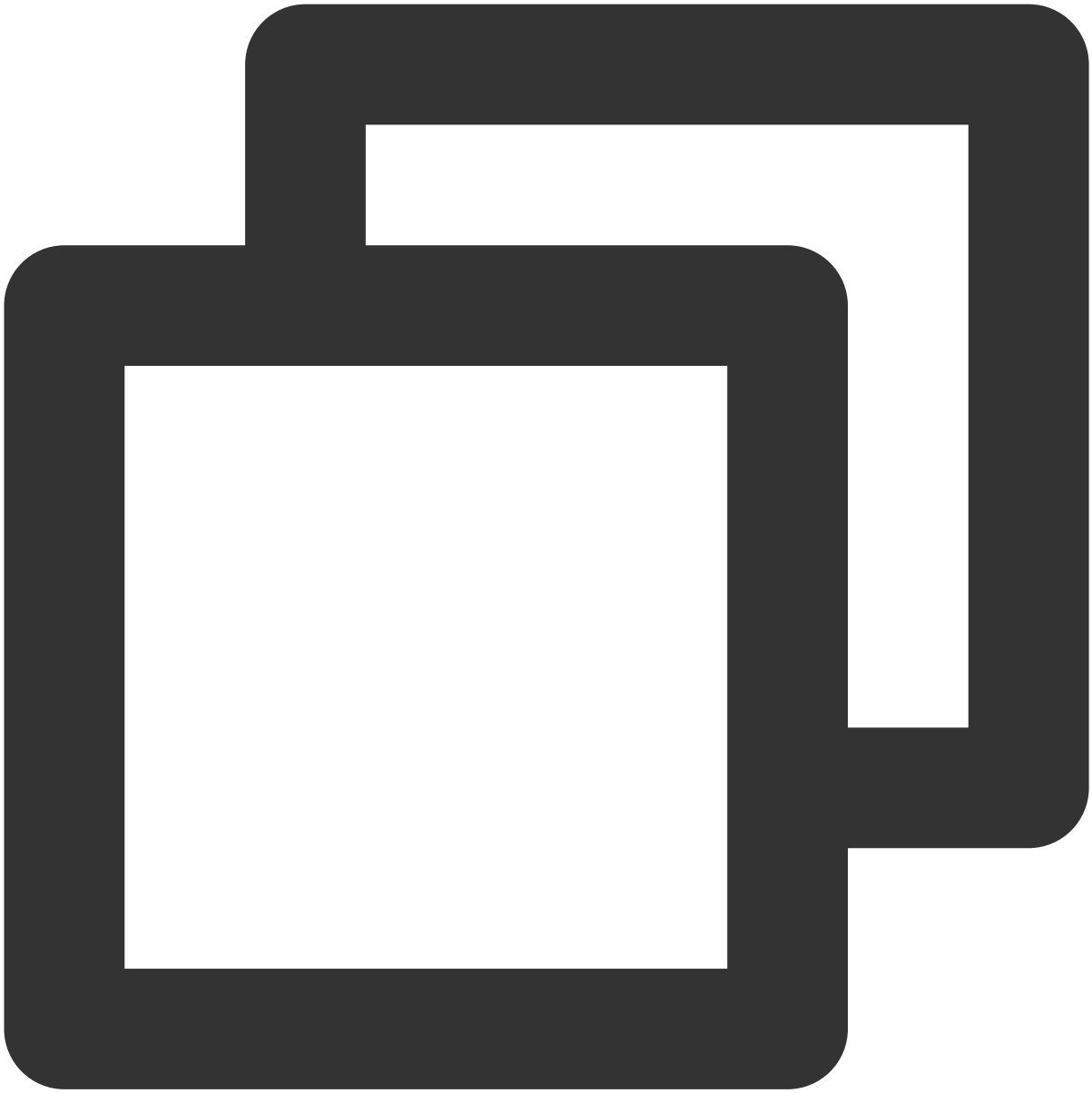
Windows



```
/**
 * 接收群聊自定义消息
 * @param msgID 消息 ID
 * @param groupID 群 ID
 * @param sender 发送方群成员信息
 * @param customData 发送内容
 */
public void onRecvGroupCustomMessage(String msgID, String groupID, V2TIMGroupMember
    Log.i("onRecvGroupCustomMessage", "msgID:" + msgID + ", groupID:" + groupID + "
}
```



```
/// 接收群聊自定义消息
/// @param msgID 消息 ID
/// @param groupID 群组 ID
/// @param info 发送者信息
/// @param text 自定义消息二进制内容
- (void)onRecvGroupCustomMessage:(NSString *)msgID groupID:(NSString *)groupID send
    NSLog(@"onRecvGroupCustomMessage, msgID: %@, groupID: %@, sender: %@, customDat
}
```



```
class SimpleMsgListener final : public V2TIMSimpleMsgListener {
public:
    /**
     * 收到群自定义（信令）消息
     *
     * @param msgID 消息唯一标识
     * @param groupID 群 ID
     * @param sender 发送方群成员信息
     * @param customData 发送内容
     */
    void OnRecvGroupCustomMessage(const V2TIMString& msgID, const V2TIMString& grou
```

```
                                    const V2TIMGroupMemberFullInfo& sender,
                                    const V2TIMBuffer& customData) override {
        // 可以解析消息并展示到 UI，比如当 customData 为文本时：
        std::cout << "customData:"
                  << std::string{reinterpret_cast<const char*>(customData.Data()),
                  << std::endl;
    }
    // 其他成员 ...
};

// 添加基本消息的事件监听器，注意在移除监听器之前需要保持 simpleMsgListener 的生命期，以免接收不
SimpleMsgListener simpleMsgListener;
V2TIMManager::GetInstance()->AddSimpleMsgListener(&simpleMsgListener);
```

使用高级消息监听器接收

接收方使用高级消息监听器接收单聊、群聊自定义消息，需要以下几步：

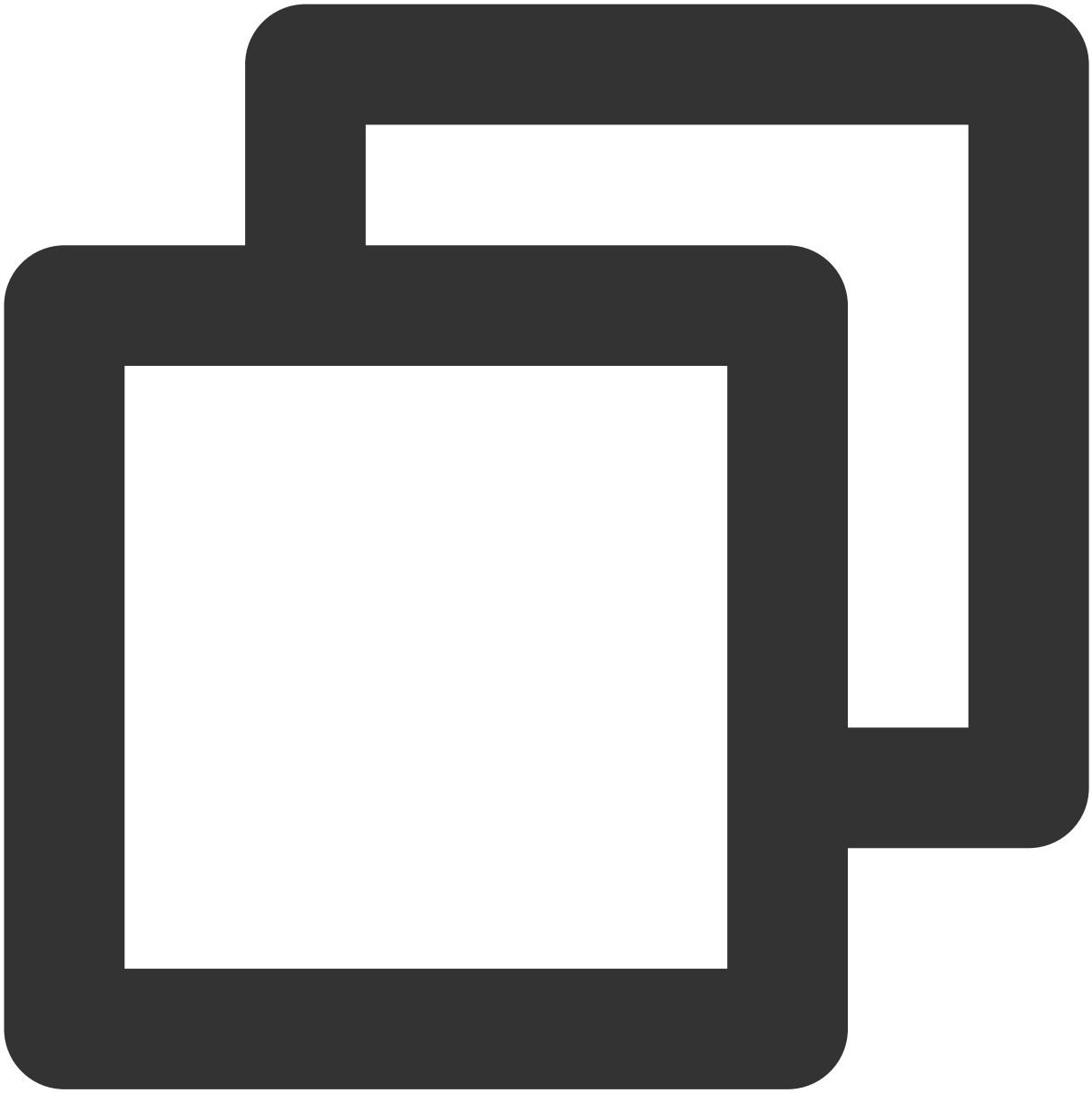
1. 调用 `addAdvancedMsgListener` 设置事件监听器。
2. 监听 `onRecvNewMessage` (Android / iOS & Mac / Windows) 回调，在其中接收自定义消息。
3. 希望停止接收消息，调用 `removeAdvancedMsgListener` 移除监听。该步骤不是必须的，客户可以按照业务需求调用。

代码示例如下：

Android

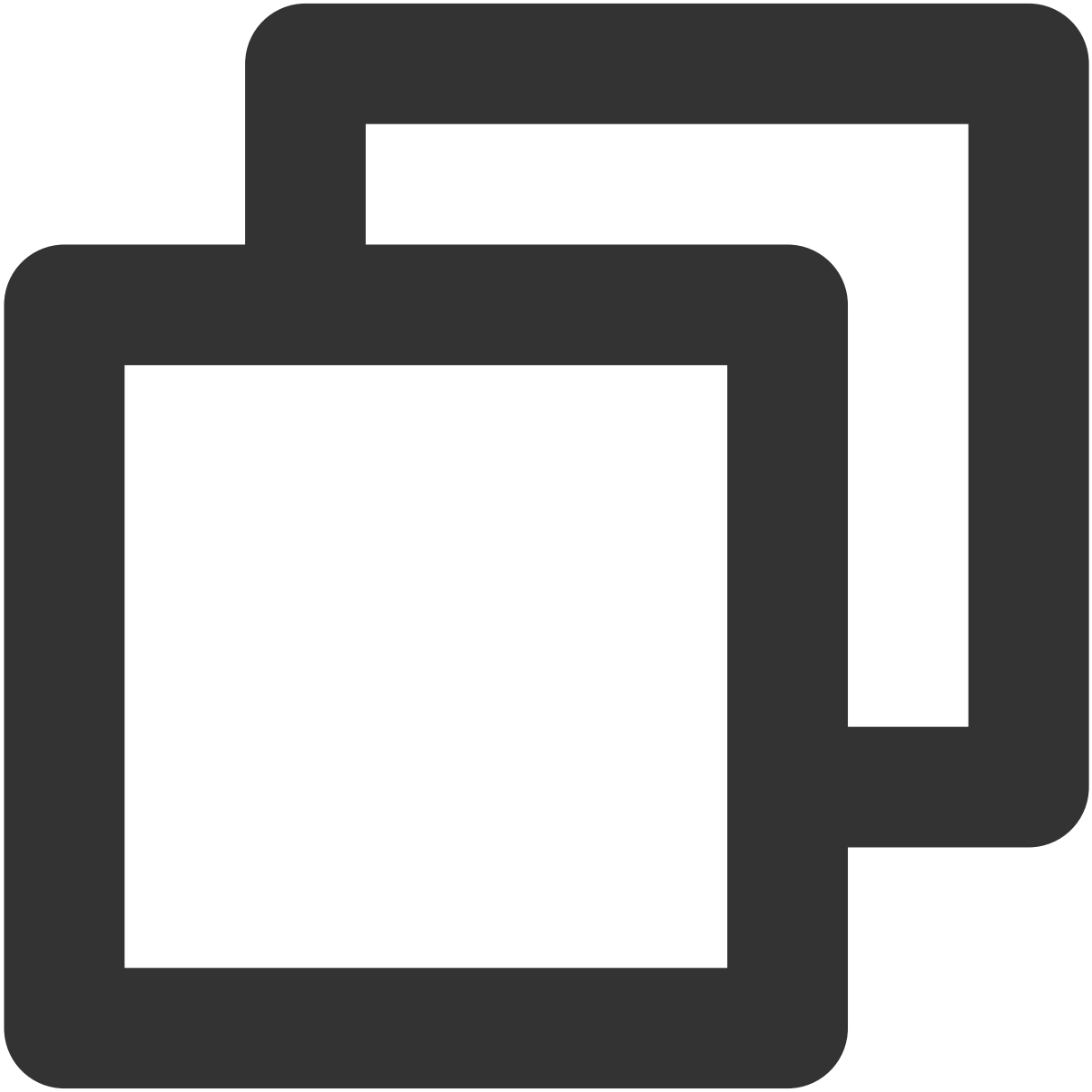
iOS & Mac

Windows



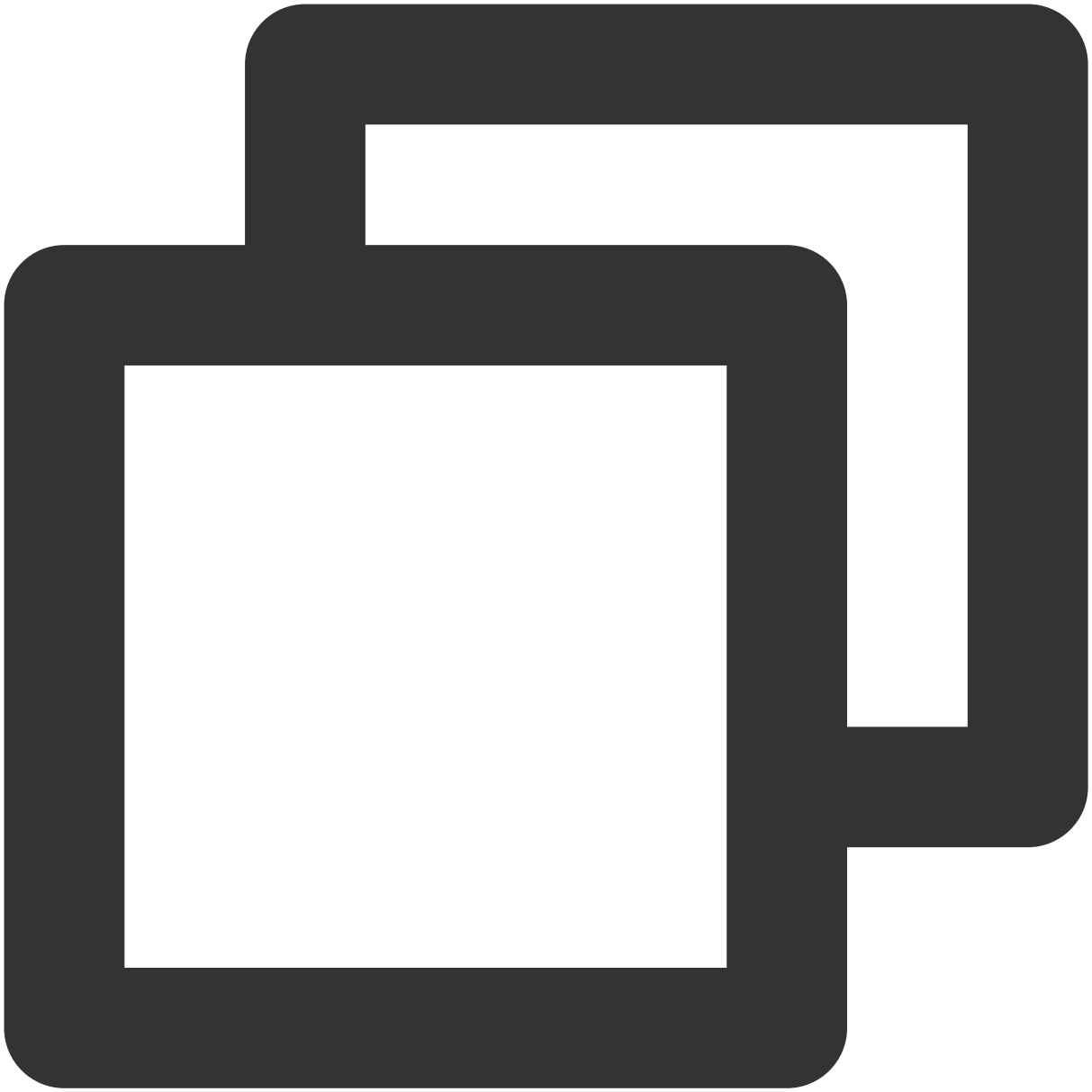
```
// 设置事件监听器
V2TIMManager.getMessageManager().addAdvancedMsgListener(v2TIMAdvancedMsgListener);

// 接收消息
public void onRecvNewMessage(V2TIMMessage msg) {
    // 解析出 groupID 和 userID
    String groupID = msg.getGroupID();
    String userID = msg.getUserID();

    // 判断当前是单聊还是群聊：
    // 如果 groupID 不为空，表示此消息为群聊；如果 userID 不为空，表示此消息为单聊
```

```
    // 解析出 msg 中的自定义消息
    if (msg.getElemType() == V2TIMMessage.V2TIM_ELEM_TYPE_CUSTOM) {
        V2TIMCustomElem customElem = msg.getCustomElem();
        String data = new String(customElem.getData());
        Log.i("onRecvNewMessage", "customData:" + data);
    }
}
```



```
// 设置事件监听器
[[V2TIMManager sharedInstance] addAdvancedMsgListener:self];
```

```
/// 接收消息
/// @param msg 消息对象
- (void)onRecvNewMessage:(V2TIMMessage *)msg {
    // 解析出 groupID 和 userID
    NSString *groupID = msg.groupID;
    NSString *userID = msg.userID;

    // 判断当前是单聊还是群聊：
    // 如果 groupID 不为空，表示此消息为群聊；如果 userID 不为空，表示此消息为单聊

    // 解析出 msg 中的自定义消息
    if (msg.elemType == V2TIM_ELEM_TYPE_CUSTOM) {
        V2TIMCustomElem *customElem = msg.customElem;
        NSData *customData = customElem.data;
        NSLog(@"onRecvNewMessage, customData: %@", customData);
    }
}
```



```
class AdvancedMsgListener final : public V2TIMAdvancedMsgListener {
public:
    /**
     * 收到新消息
     *
     * @param message 消息
     */
    void OnRecvNewMessage(const V2TIMMessage& message) override {
        // 解析出 groupID 和 userID
        V2TIMString groupID = message.groupID;
        V2TIMString userID = message.userID;
```

```
        // 判断当前是单聊还是群聊：
        // 如果 groupID 不为空，表示此消息为群聊；如果 userID 不为空，表示此消息为单聊

        // 解析出 message 中的自定义消息
        if (message.elemList.Size() == 1) {
            V2TIMElem* elem = message.elemList[0];
            if (elem->elemType == V2TIMElemType::V2TIM_ELEM_TYPE_CUSTOM) {
                // 自定义消息
                auto customElem = static_cast<V2TIMCustomElem*>(elem);
                // 自定义消息二进制数据
                V2TIMBuffer data = customElem->data;
                // 可以解析消息并展示到 UI，比如当 data 为文本时：
                std::cout << "data:"
                          << std::string{reinterpret_cast<const char*>(data.Data())
                          << std::endl;
            }
        }
    }
    // 其他成员 ...
};

// 添加高级消息的事件监听器，注意在移除监听器之前需要保持 advancedMsgListener 的生命期，以免接
AdvancedMsgListener advancedMsgListener;
V2TIMManager::GetInstance()->GetMessageManager()->AddAdvancedMsgListener(&advancedM
```

接收富媒体消息

接收富媒体消息**只能使用**高级消息监听器，需要以下几步：

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` (Android / iOS & Mac / Windows) 获取消息 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。
4. 希望停止接收消息，调用 `removeAdvancedMsgListener` 移除监听。该步骤不是必须的，客户可以按照业务需求调用。

图片消息

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 会在发送图片消息的时候自动生成微缩图、大图，客户不需要关心）：
大图：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：将原图等比压缩。压缩后宽、高中较小的一个等于 198 像素。

版权所有：腾讯云计算（北京）有限责任公司

接收端收到图片消息后，我们推荐您调用 SDK 的 `downloadImage` (Android / iOS & Mac / Windows) 将图片下载到本地，再取出图片渲染到 UI 层。

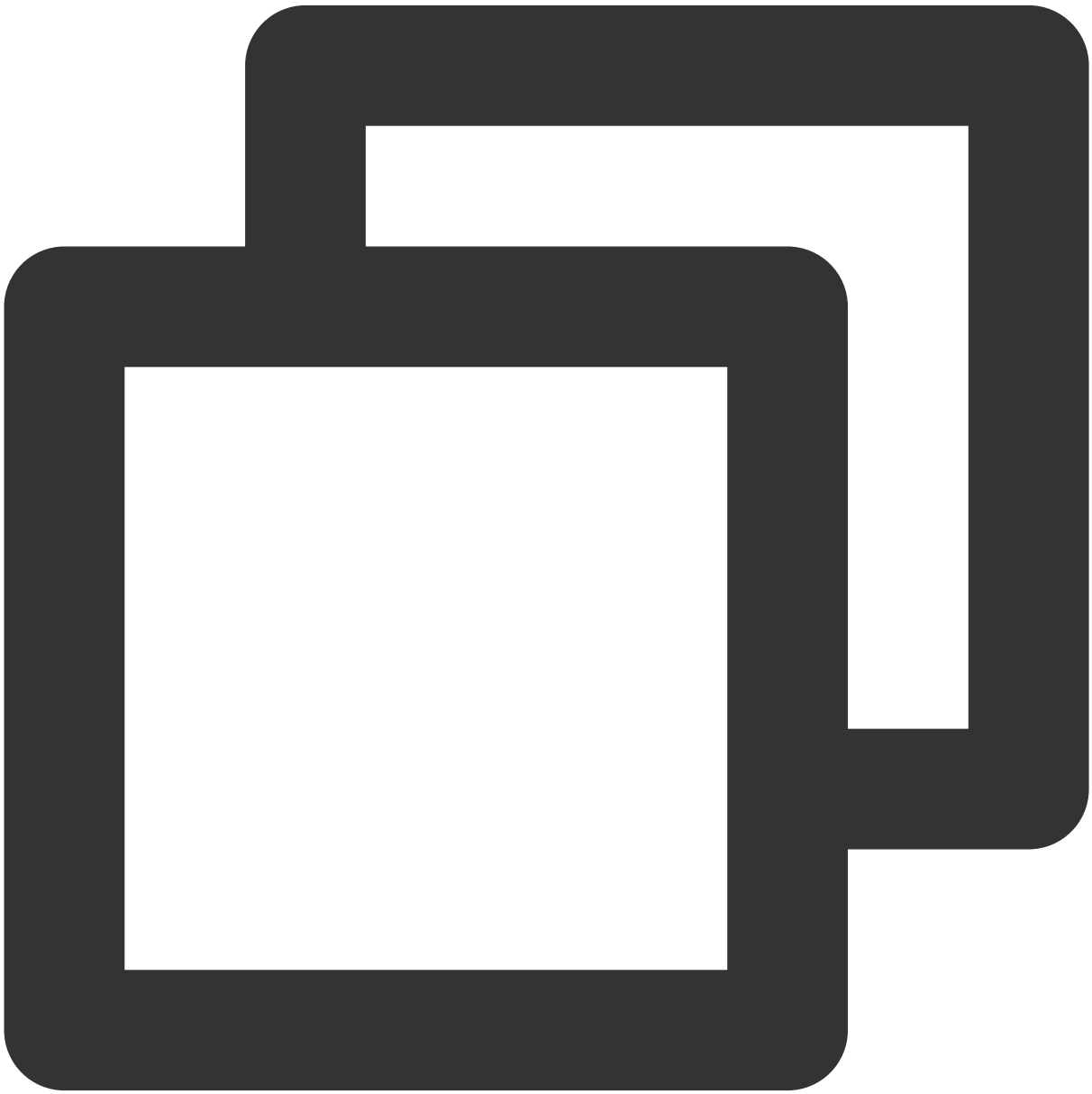
为了避免重复下载，节省资源，我们推荐您将 `V2TIMImage` 对象的 `uuid` 属性值设置到图片的下载路径中，作为图片的标识。

示例代码向您演示如何从 `V2TIMMessage` 中解析出图片消息内容：

Android

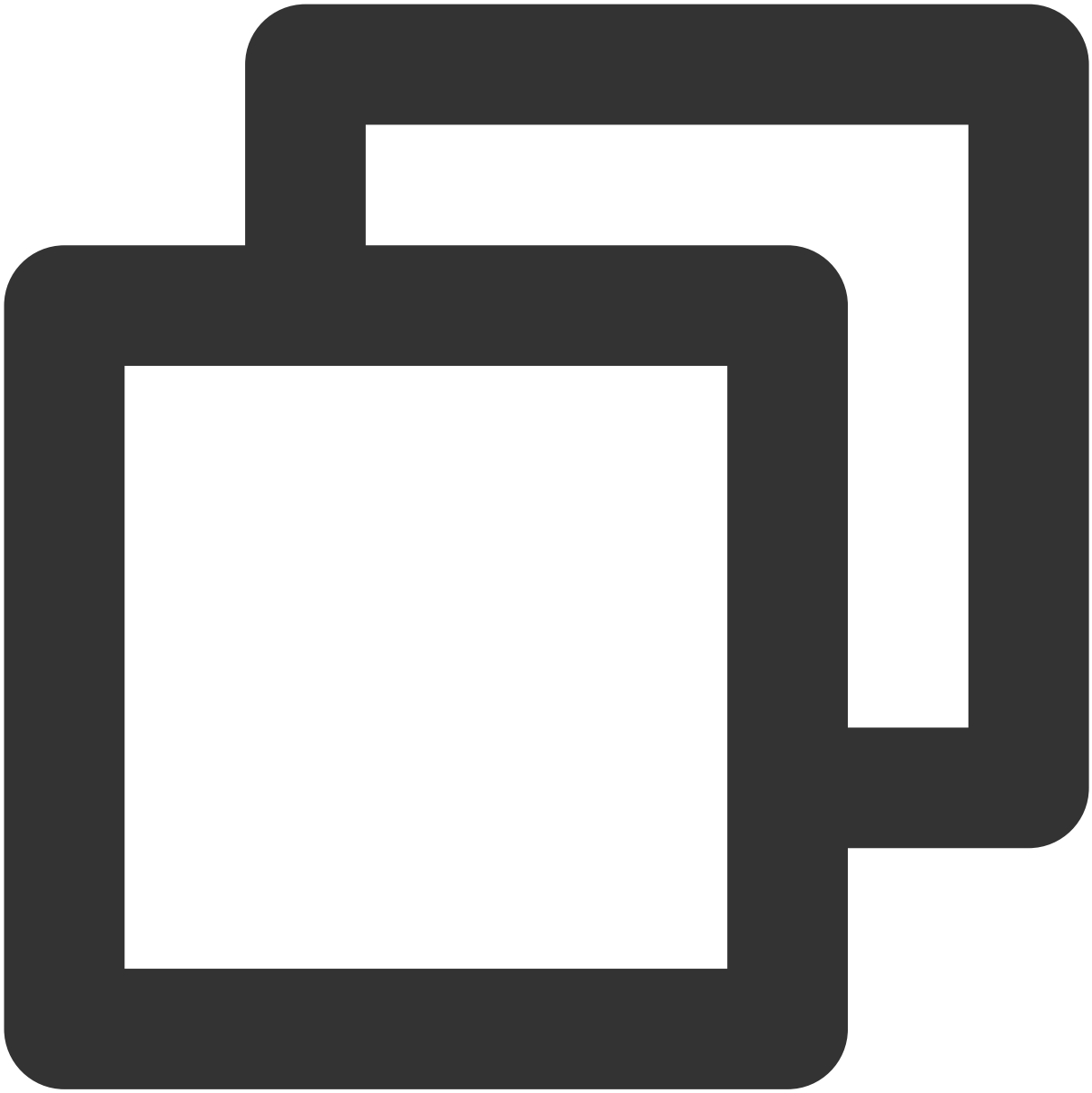
iOS & Mac

Windows



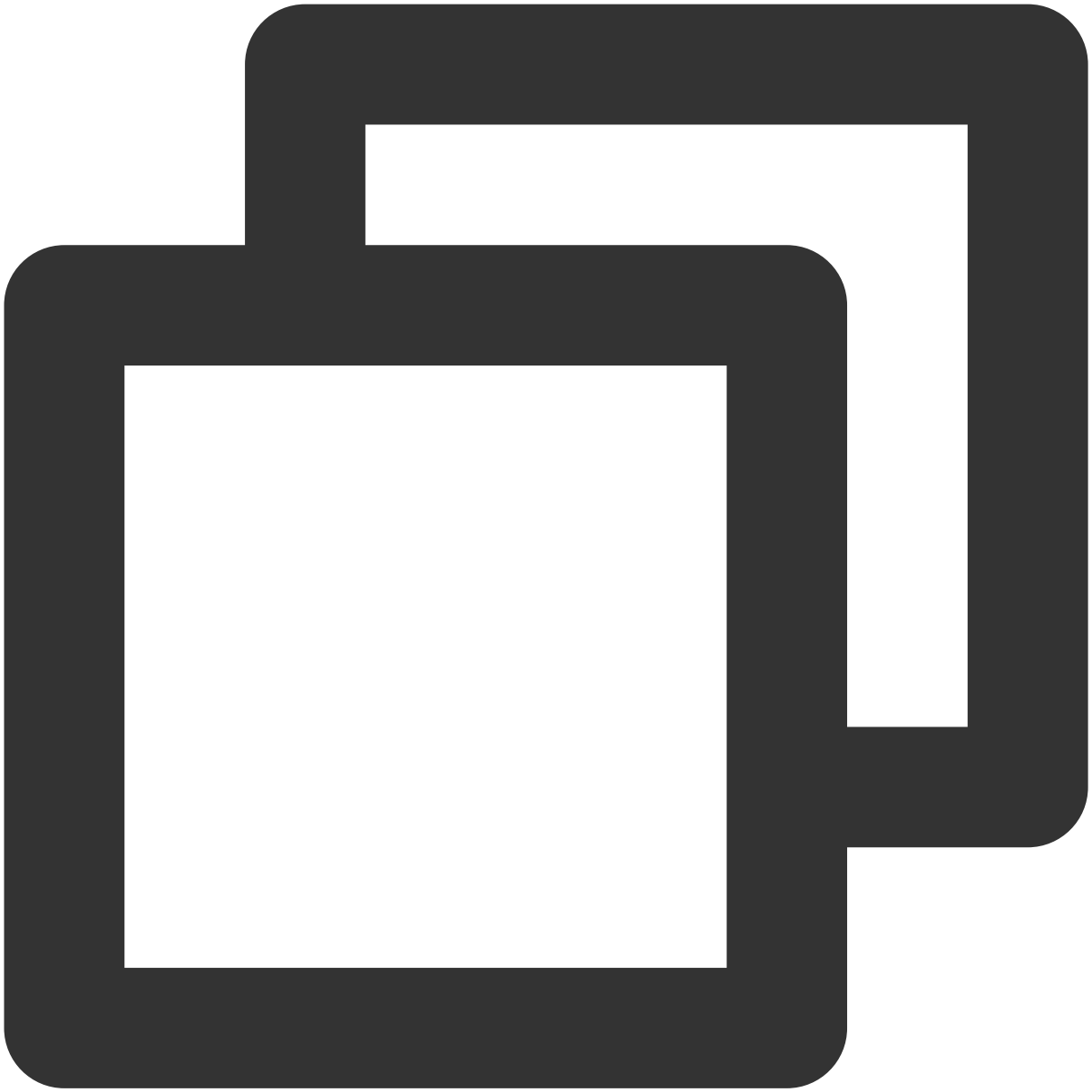
```
public void onRecvNewMessage(V2TIMMessage msg) {
```

```
    if (msg.getElemType() == V2TIMMessage.V2TIM_ELEM_TYPE_IMAGE) {
        // 图片消息
        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 会在发送图片消息的
        // 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) {
            // 图片 ID，内部标识，可用于外部缓存 key
            String uuid = v2TIMImage.getUUID();
            // 图片类型，三种类型，分别为 V2TIM_IMAGE_TYPE_ORIGIN（原图），V2TIM_IMAGE_TY
            int imageType = v2TIMImage.getType();
            // 图片大小（字节）
            int size = v2TIMImage.getSize();
            // 图片宽度
            int width = v2TIMImage.getWidth();
            // 图片高度
            int height = v2TIMImage.getHeight();
            // 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);
            // 判断 imagePath 下有没有已经下载过的图片文件
            if (!imageFile.exists()) {
                // 下载图片
                v2TIMImage.downloadImage(imagePath, new V2TIMDownloadCallback() {
                    @Override
                    public void onProgress(V2TIMElem.V2ProgressInfo progressInfo) {
                        // 下载进度回调：已下载大小 v2ProgressInfo.getCurrentSize()；总
                    }
                    @Override
                    public void onError(int code, String desc) {
                        // 下载失败
                    }
                    @Override
                    public void onSuccess() {
                        // 下载完成
                    }
                });
            } else {
                // 图片已存在
            }
        }
    }
}
```



```
- (void)onRecvNewMessage:(V2TIMMessage *)msg {
    if (msg.elemType == V2TIM_ELEM_TYPE_IMAGE) {
        V2TIMImageElem *imageElem = msg.imageElem;
        // 原图、大图、微缩图列表
        NSArray<V2TIMImage *> *imageList = imageElem.imageList;
        for (V2TIMImage *timImage in imageList) {
            // 图片 ID，内部标识，可用于外部缓存 key
            NSString *uuid = timImage.uuid;
            // 图片类型
            V2TIMImageType type = timImage.type;
            // 图片大小（字节）
```

```
            int size = timImage.size;
            // 图片宽度
            int width = timImage.width;
            // 图片高度
            int height = timImage.height;
            // 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() stringByAppendingPathComp
            // 判断 imagePath 下有没有已经下载过的图片文件
            if (![[NSFileManager defaultManager] fileExistsAtPath:imagePath]) {
                // 下载图片
                [timImage downloadImage:imagePath progress:^(NSInteger curSize, NSI
                    // 下载进度
                    NSLog(@"下载图片进度：curSize：%lu,totalSize:%lu",curSize,totalSiz
                } succ:^{
                    // 下载成功
                    NSLog(@"下载图片完成");
                } fail:^(int code, NSString *msg) {
                    // 下载失败
                    NSLog(@"下载图片失败：code：%d,msg:%@",code,msg);
                }];
            } else {
                // 图片已存在
            }
            NSLog(@"图片信息：uuid:%@, type:%ld, size:%d, width:%d, height:%d", uuid,
        }
    }
}
```



```
class DownloadCallback final : public V2TIMDownloadCallback {
public:
    using SuccessCallback = std::function<void()>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using DownLoadProgressCallback = std::function<void(uint64_t, uint64_t)>;

    DownloadCallback() = default;
    ~DownloadCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     DownLoadProgressCallback download_progress_callback) {
```

```
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        download_progress_callback_ = std::move(download_progress_callback);
    }
    void OnSuccess() override {
        if (success_callback_) {
            success_callback_();
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnDownLoadProgress(uint64_t currentSize, uint64_t totalSize) override {
        if (download_progress_callback_) {
            download_progress_callback_(currentSize, totalSize);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    DownLoadProgressCallback download_progress_callback_;
};

class AdvancedMsgListener final : public V2TIMAdvancedMsgListener {
public:
    /**
     * 收到新消息
     *
     * @param message 消息
     */
    void OnRecvNewMessage(const V2TIMMessage& message) override {
        // 解析出 groupID 和 userID
        V2TIMString groupID = message.groupID;
        V2TIMString userID = message.userID;

        // 判断当前是单聊还是群聊：
        // 如果 groupID 不为空，表示此消息为群聊；如果 userID 不为空，表示此消息为单聊

        // 解析出 message 中的图片消息
        if (message.elemList.Size() == 1) {
            V2TIMElem* elem = message.elemList[0];
            if (elem->elemType == V2TIMElemType::V2TIM_ELEM_TYPE_IMAGE) {
                // 图片消息
                auto imageElem = static_cast<V2TIMImageElem*>(elem);
```

```
                // 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
                // 会在发送图片消息的时候自动生成微缩图、大图，客户不需要关心）
                // 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。
                V2TIMImageVector imageList = imageElem->imageList;
                for (size_t i = 0; i < imageList.Size(); ++i) {
                    V2TIMImage& image = imageList[i];
                    /// 图片 ID，内部标识，可用于外部缓存 key
                    V2TIMString uuid = image.uuid;
                    /// 图片类型
                    V2TIMImageType type = image.type;
                    /// 图片大小（type == V2TIMImageType::V2TIM_IMAGE_TYPE_ORIGIN 有效
                    uint64_t size = image.size;
                    /// 图片宽度
                    uint32_t width = image.width;
                    /// 图片高度
                    uint32_t height = image.height;
                    // 设置图片下载路径 path，这里可以用 uuid 作为标识，避免重复下载
                    std::filesystem::path imagePath = u8"./File/Image/"s + uuid.CSt
                    // 判断 imagePath 下有没有已经下载过的图片文件
                    if (!std::filesystem::exists(imagePath)) {
                        std::filesystem::create_directories(imagePath.parent_path()
                        auto callback = new DownloadCallback{};
                        callback->SetCallback(
                            [=]() {
                                // 下载完成
                                delete callback;
                            },
                            [=](int error_code, const V2TIMString& error_message) {
                                // 下载失败
                                delete callback;
                            },
                            [=](uint64_t currentSize, uint64_t totalSize) {
                                // 下载进度回调：已下载大小 currentSize; 总文件大小 total
                            });
                        image.DownloadImage(imagePath.string().c_str(), callback);
                    } else {
                        // 图片已存在
                    }
                }
            }
        }
    }
    // 其他成员 ...
};

// 添加高级消息的事件监听器，注意在移除监听器之前需要保持 advancedMsgListener 的生命期，以免接收
```

```
AdvancedMsgListener advancedMsgListener;
V2TIMManager::GetInstance()->GetMessageManager()->AddAdvancedMsgListener(&advancedM
```

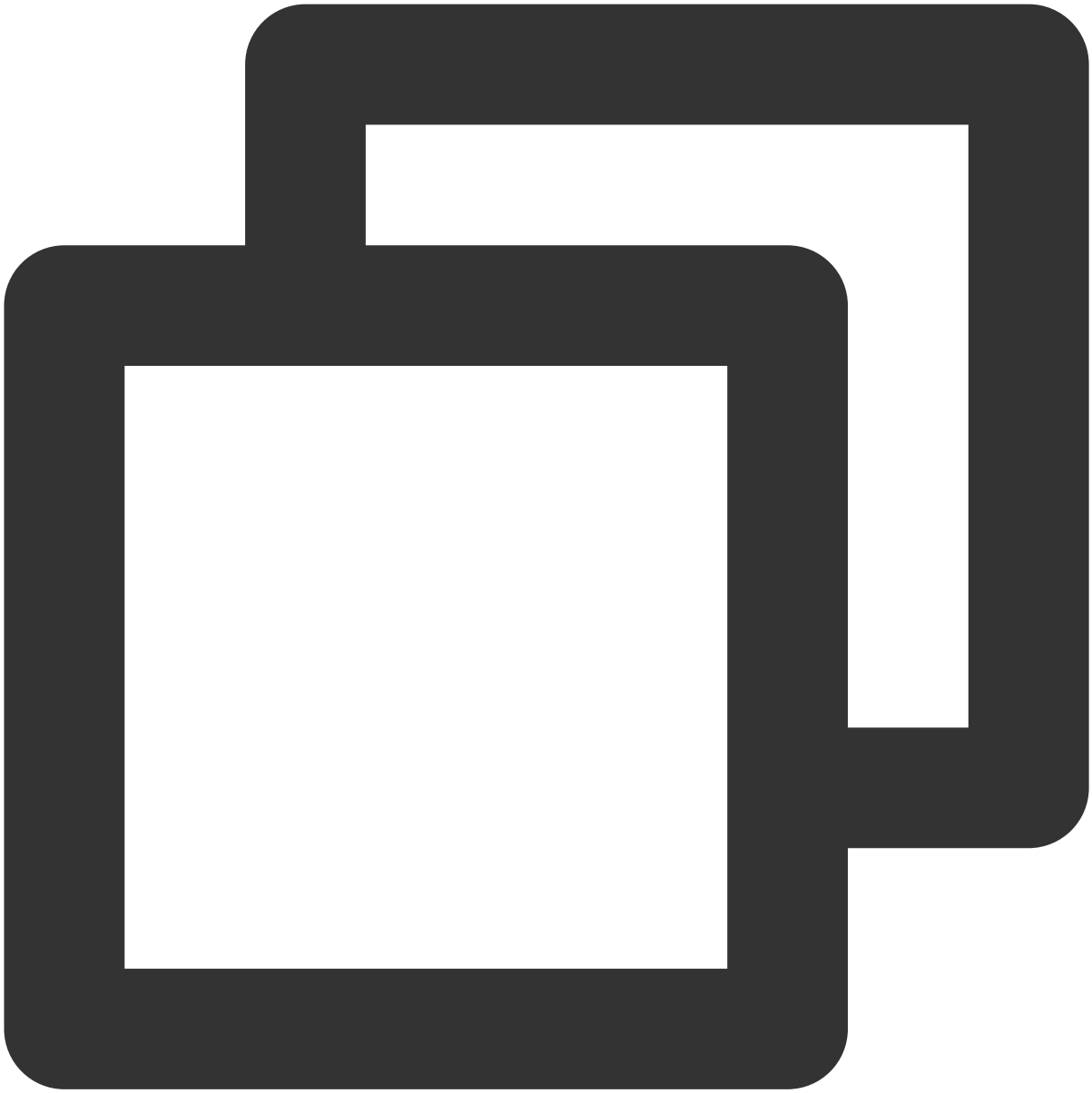
视频消息

接收方收到视频消息后，一般需要在聊天界面显示一个视频预览图，当用户点击消息后，才会触发视频的播放。
所以这里需要两步：
1. 下载视频截图。我们推荐您调用 SDK 的 `downloadSnapshot` (Android / iOS & Mac / Windows) 进行下载。
2. 下载视频。我们推荐您调用 SDK 的 `downloadVideo` (Android / iOS & Mac / Windows) 进行下载。
为了避免重复下载，节省资源，我们推荐您将 `V2TIMVideoElem` 对象的 `videoUUID` 属性值设置到视频的下载路径中，作为视频的标识。
示例代码向您演示如何从 `V2TIMMessage` 中解析出视频消息内容：

Android

iOS & Mac

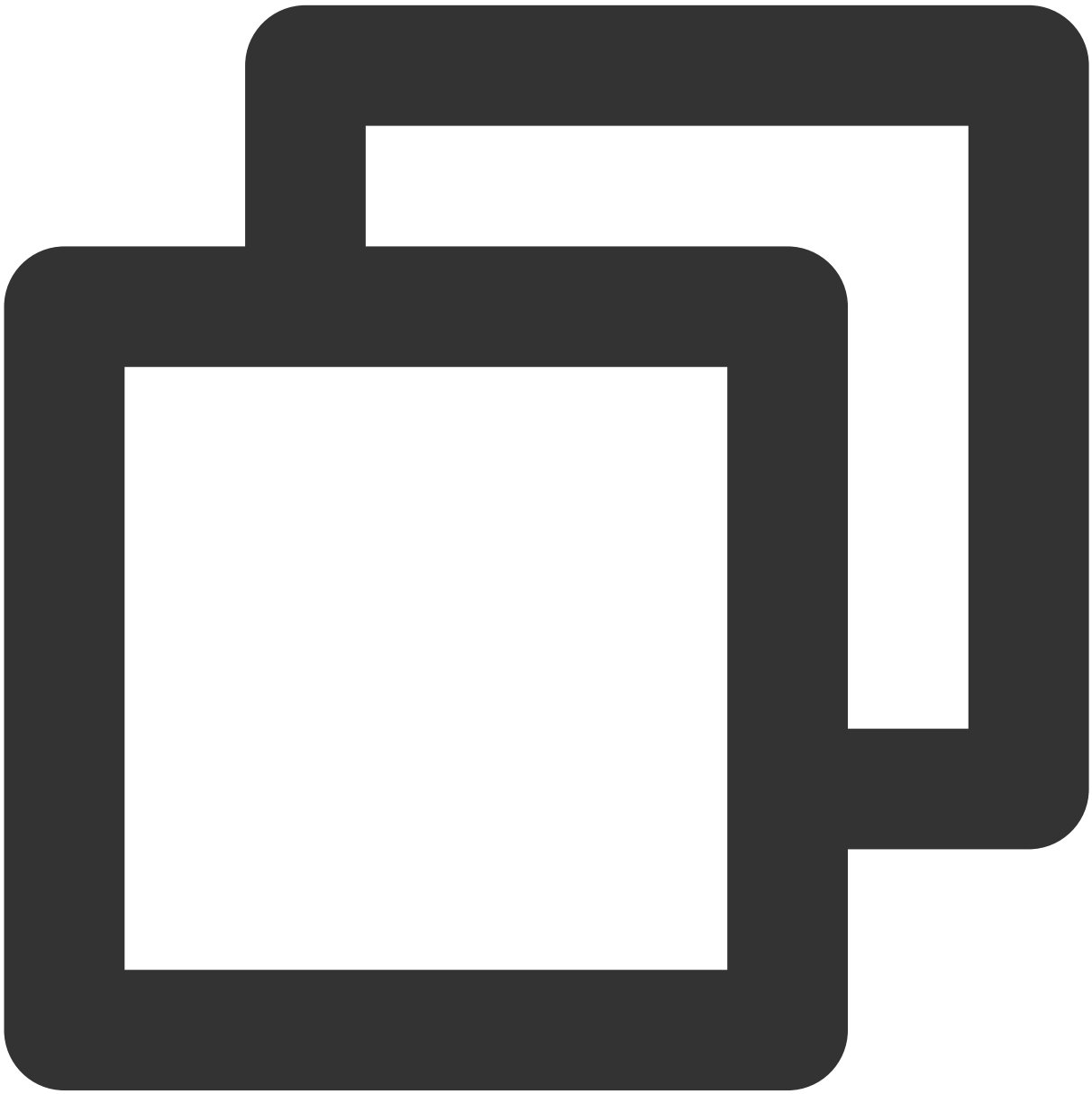
Windows



```
public void onRecvNewMessage(V2TIMMessage msg) {
    if (msg.getElemType() == V2TIMMessage.V2TIM_ELEM_TYPE_VIDEO) {
        // 视频消息
        V2TIMVideoElem v2TIMVideoElem = msg.getVideoElem();
        // 视频截图 ID,内部标识，可用于外部缓存 key
        String snapshotUUID = v2TIMVideoElem.getSnapshotUUID();
        // 视频截图文件大小
        int snapshotSize = v2TIMVideoElem.getSnapshotSize();
        // 视频截图宽
        int snapshotWidth = v2TIMVideoElem.getSnapshotWidth();
        // 视频截图高
```

```
int snapshotHeight = v2TIMVideoElem.getSnapshotHeight();
// 视频 ID,内部标识，可用于外部缓存 key
String videoUUID = v2TIMVideoElem.getVideoUUID();
// 视频文件大小
int videoSize = v2TIMVideoElem.getVideoSize();
// 视频时长
int duration = v2TIMVideoElem.getDuration();
// 设置视频截图文件路径，这里可以用 uuid 作为标识，避免重复下载
String snapshotPath = "/sdcard/im/snapshot/" + "myUserID" + snapshotUUID;
File snapshotFile = new File(snapshotPath);
if (!snapshotFile.exists()) {
    v2TIMVideoElem.downloadSnapshot(snapshotPath, new V2TIMDownloadCallback
        @Override
        public void onProgress(V2TIMElem.V2ProgressInfo progressInfo) {
            // 下载进度回调：已下载大小 v2ProgressInfo.getCurrentSize()；总文件大
        }
        @Override
        public void onError(int code, String desc) {
            // 下载失败
        }
        @Override
        public void onSuccess() {
            // 下载完成
        }
    });
} else {
    // 文件已存在
}

// 设置视频文件路径，这里可以用 uuid 作为标识，避免重复下载
String videoPath = "/sdcard/im/video/" + "myUserID" + videoUUID;
File videoFile = new File(videoPath);
if (!videoFile.exists()) {
    v2TIMVideoElem.downloadVideo(videoPath, new V2TIMDownloadCallback() {
        @Override
        public void onProgress(V2TIMElem.V2ProgressInfo progressInfo) {
        // 下载进度回调：已下载大小 v2ProgressInfo.getCurrentSize()；总文件大小 v
        }
        @Override
        public void onError(int code, String desc) {
        // 下载失败
        }
        @Override
        public void onSuccess() {
        // 下载完成
        }
    });
```

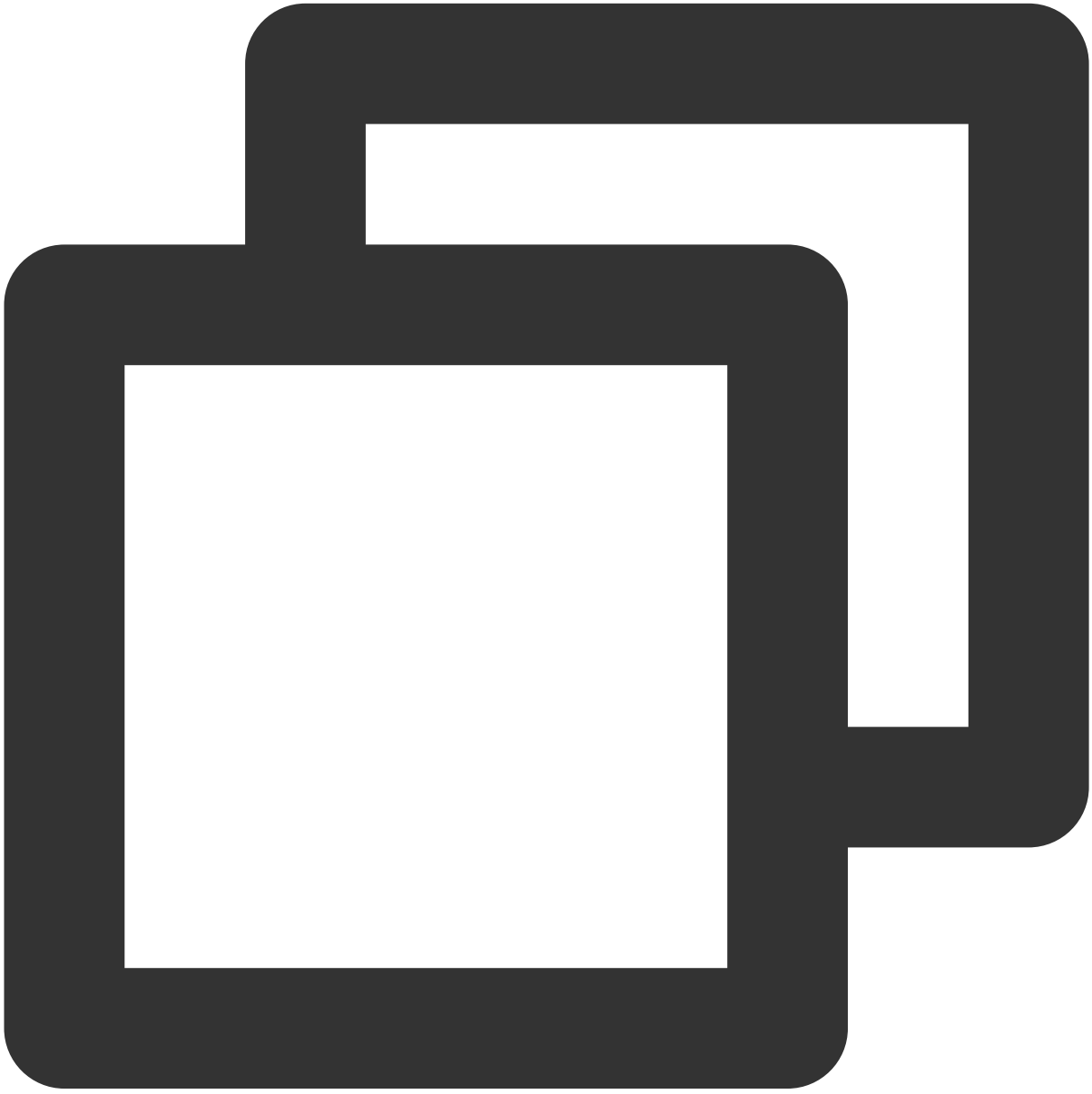
```
        } else {
            // 文件已存在
        }
    }
}
```



```
- (void)onRecvNewMessage:(V2TIMMessage *)msg {
    if (msg.elemType == V2TIM_ELEM_TYPE_VIDEO) {
        V2TIMVideoElem *videoElem = msg.videoElem;
        // 视频截图 ID,内部标识，可用于外部缓存 key
        NSString *snapshotUUID = videoElem.snapshotUUID;
```

```
        // 视频截图文件大小
        int snapshotSize = videoElem.snapshotSize;
        // 视频截图宽
        int snapshotWidth = videoElem.snapshotWidth;
        // 视频截图高
        int snapshotHeight = videoElem.snapshotHeight;
        // 视频 ID,内部标识，可用于外部缓存 key
        NSString *videoUUID = videoElem.videoUUID;
        // 视频文件大小
        int videoSize = videoElem.videoSize;
        // 视频时长
        int duration = videoElem.duration;
        // 设置视频截图文件路径，这里可以用 uuid 作为标识，避免重复下载
        NSString *snapshotPath = [NSTemporaryDirectory() stringByAppendingPathCompo
        if (![[NSFileManager defaultManager] fileExistsAtPath:snapshotPath]) {
            // 下载视频截图
            [videoElem downloadSnapshot:snapshotPath progress:^(NSInteger curSize,
                // 下载进度
                NSLog(@"%@", [NSString stringWithFormat:@"下载视频截图进度：curSize：%
            } succ:^{
                // 下载成功
                NSLog(@"下载视频截图完成");
            } fail:^(int code, NSString *msg) {
                // 下载失败
                NSLog(@"%@", [NSString stringWithFormat:@"下载视频截图失败：code：%d,ms
            }];
        } else {
            // 视频截图已存在
        }
        NSLog(@"视频截图信息：snapshotUUID:%@, snapshotSize:%d, snapshotWidth:%d, snap

        // 设置视频文件路径，这里可以用 uuid 作为标识，避免重复下载
        NSString *videoPath = [NSTemporaryDirectory() stringByAppendingPathComponen
        if (![[NSFileManager defaultManager] fileExistsAtPath:videoPath]) {
            // 下载视频
            [videoElem downloadVideo:videoPath progress:^(NSInteger curSize, NSInte
                // 下载进度
                NSLog(@"%@", [NSString stringWithFormat:@"下载视频进度：curSize：%lu,t
            } succ:^{
                // 下载成功
                NSLog(@"下载视频完成");
            } fail:^(int code, NSString *msg) {
                // 下载失败
                NSLog(@"%@", [NSString stringWithFormat:@"下载视频失败：code：%d,msg:%
            }];
        } else {
            // 视频已存在
```

```
        }
        NSLog(@"视频信息：videoUUID:%@, videoSize:%d, duration:%d, videoPath:%@", vid
    }
}
```



```
class DownloadCallback final : public V2TIMDownloadCallback {
public:
    using SuccessCallback = std::function<void()>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using DownLoadProgressCallback = std::function<void(uint64_t, uint64_t)>;
```

```
    DownloadCallback() = default;
    ~DownloadCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     DownLoadProgressCallback download_progress_callback) {
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        download_progress_callback_ = std::move(download_progress_callback);
    }
    void OnSuccess() override {
        if (success_callback_) {
            success_callback_();
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnDownLoadProgress(uint64_t currentSize, uint64_t totalSize) override {
        if (download_progress_callback_) {
            download_progress_callback_(currentSize, totalSize);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    DownLoadProgressCallback download_progress_callback_;
};

class AdvancedMsgListener final : public V2TIMAdvancedMsgListener {
public:
    /**
     * 收到新消息
     *
     * @param message 消息
     */
    void OnRecvNewMessage(const V2TIMMessage& message) override {
        // 解析出 groupID 和 userID
        V2TIMString groupID = message.groupID;
        V2TIMString userID = message.userID;

        // 判断当前是单聊还是群聊：
        // 如果 groupID 不为空，表示此消息为群聊；如果 userID 不为空，表示此消息为单聊

        // 解析出 message 中的视频消息
```

```
if (message.elemList.Size() == 1) {
    V2TIMElem* elem = message.elemList[0];
    if (elem->elemType == V2TIMElemType::V2TIM_ELEM_TYPE_VIDEO) {
        // 视频消息
        auto videoElem = static_cast<V2TIMVideoElem*>(elem);
        // 视频 ID,内部标识，可用于外部缓存 key
        V2TIMString videoUUID = videoElem->videoUUID;
        // 视频大小
        uint64_t videoSize = videoElem->videoSize;
        // 视频类型
        V2TIMString videoType = videoElem->videoType;
        // 视频时长
        uint32_t duration = videoElem->duration;
        // 截图 ID,内部标识，可用于外部缓存 key
        V2TIMString snapshotUUID = videoElem->snapshotUUID;
        // 截图 size
        uint64_t snapshotSize = videoElem->snapshotSize;
        // 截图宽
        uint32_t snapshotWidth = videoElem->snapshotWidth;
        // 截图高
        uint32_t snapshotHeight = videoElem->snapshotHeight;

        // 设置视频文件路径，这里可以用 uuid 作为标识，避免重复下载
        std::filesystem::path videoPath = u8"./File/Video/"s + videoUUID.CS
        // 判断 videoPath 下有没有已经下载过的视频文件
        if (!std::filesystem::exists(videoPath)) {
            std::filesystem::create_directories(videoPath.parent_path());
            auto callback = new DownloadCallback{};
            callback->SetCallback(
                [=]() {
                    // 下载完成
                    delete callback;
                },
                [=](int error_code, const V2TIMString& error_message) {
                    // 下载失败
                    delete callback;
                },
                [=](uint64_t currentSize, uint64_t totalSize) {
                    // 下载进度回调：已下载大小 currentSize; 总文件大小 totalSize
                });
            videoElem->DownloadVideo(videoPath.string().c_str(), callback);
        } else {
            // 视频文件已存在
        }

        // 设置视频截图文件路径，这里可以用 uuid 作为标识，避免重复下载
        std::filesystem::path snapshotPath = u8"./File/Snapshot/"s + snapsh
```

```
                // 判断 snapshotPath 下有没有已经下载过的视频截图文件
                if (!std::filesystem::exists(snapshotPath)) {
                    std::filesystem::create_directories(snapshotPath.parent_path())
                    auto callback = new DownloadCallback{};
                    callback->SetCallback(
                        [=]() {
                            // 下载完成
                            delete callback;
                        },
                        [=](int error_code, const V2TIMString& error_message) {
                            // 下载失败
                            delete callback;
                        },
                        [=](uint64_t currentSize, uint64_t totalSize) {
                            // 下载进度回调：已下载大小 currentSize; 总文件大小 totalSize
                        });
                    videoElem->DownloadSnapshot(snapshotPath.string().c_str(), call
                } else {
                    //视频截图文件已存在
                }
            }
        }
    }
    // 其他成员 ...
};

// 添加高级消息的事件监听器，注意在移除监听器之前需要保持 advancedMsgListener 的生命期，以免接
AdvancedMsgListener advancedMsgListener;
V2TIMManager::GetInstance()->GetMessageManager()->AddAdvancedMsgListener(&advancedM
```

语音消息

接收端收到语音消息后，我们推荐您调用 SDK 的 `downloadSound` (Android / iOS & Mac / Windows) 将语音下载到本地，再获取本地语音文件播放。

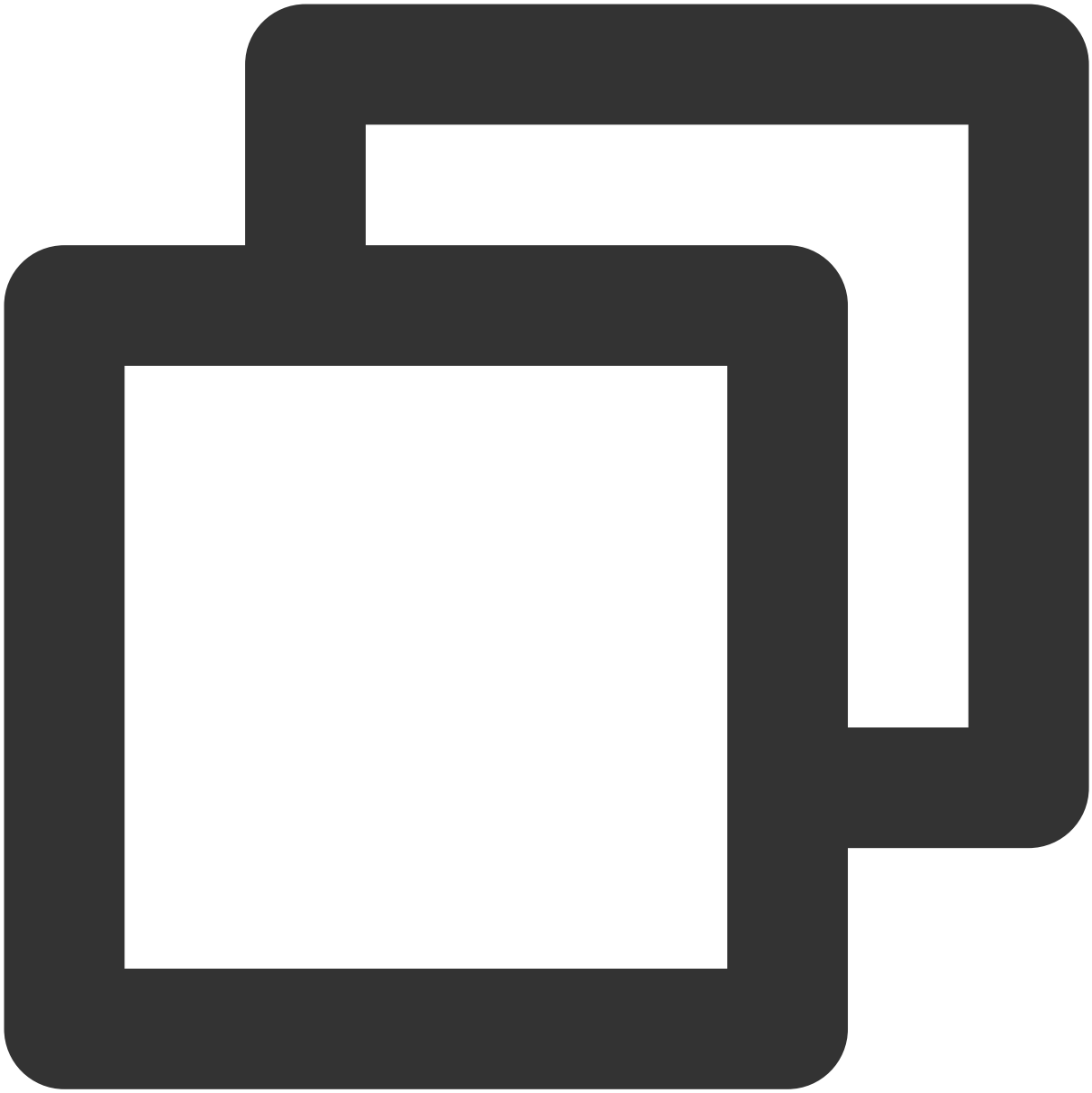
为了避免重复下载，节省资源，我们推荐您将 `V2TIMSoundElem` 对象的 `uuid` 属性值设置到语音的下载路径中，作为语音的标识。

示例代码向您演示如何从 `V2TIMMessage` 中解析出语音消息内容：

Android

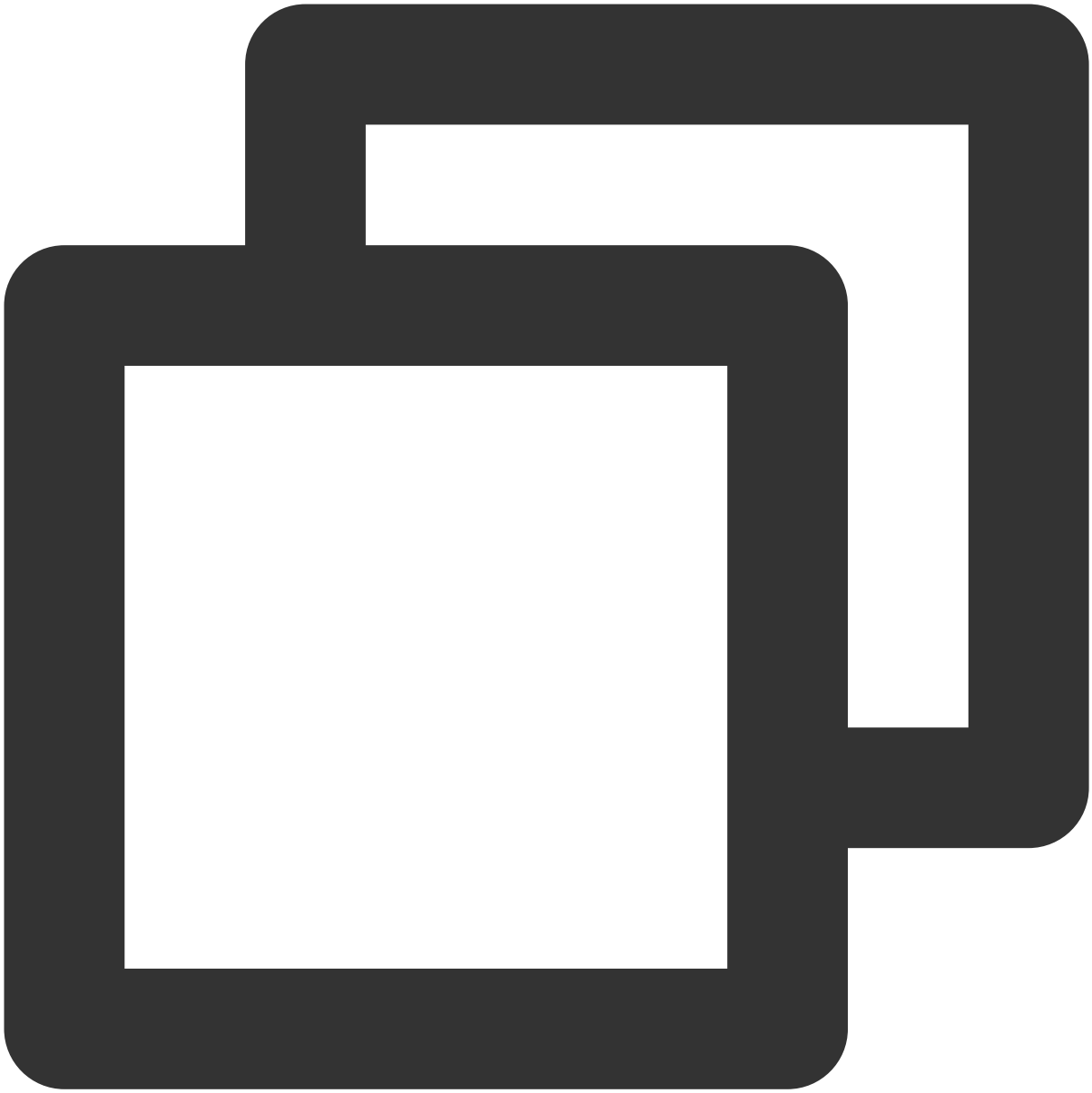
iOS & Mac

Windows



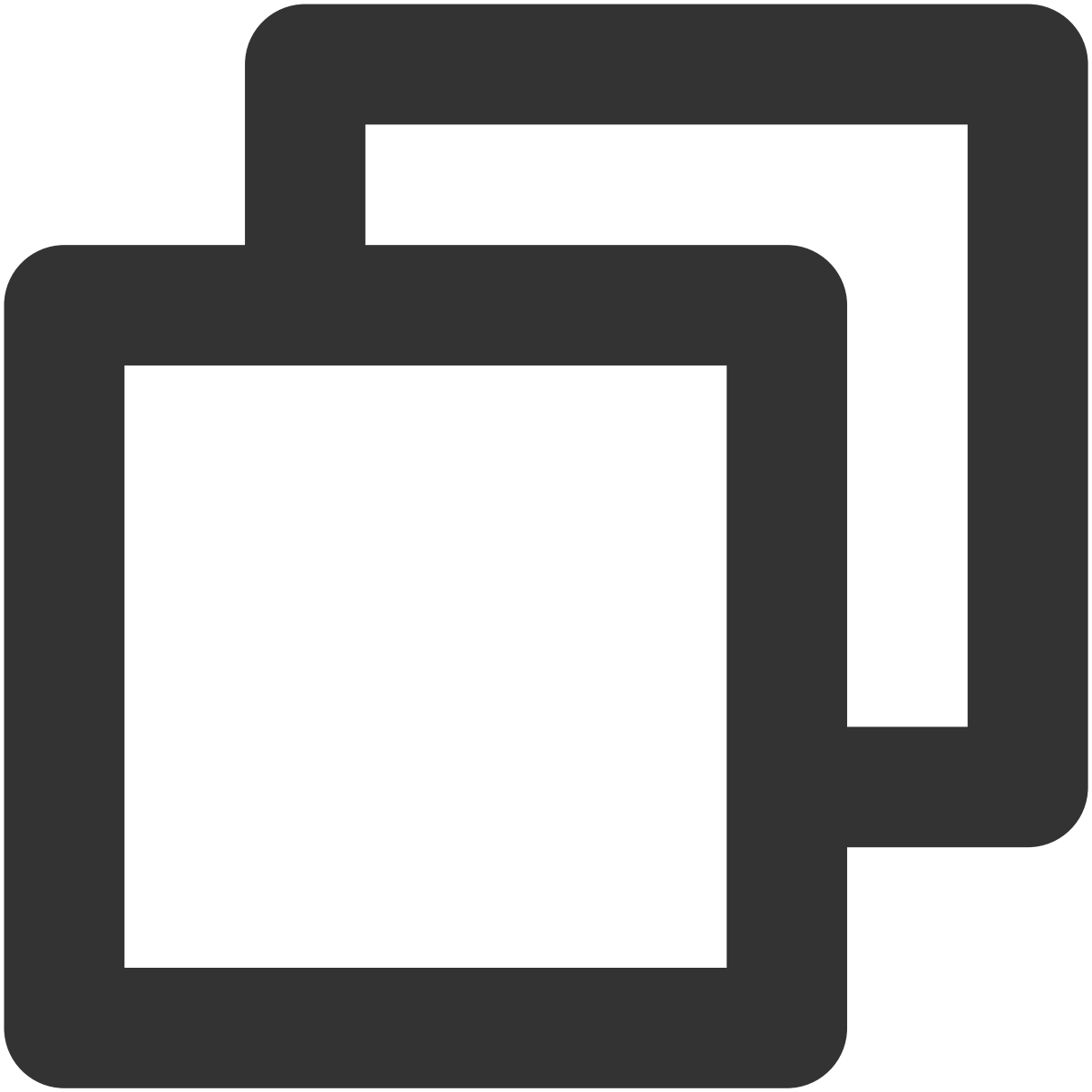
```
public void onRecvNewMessage(V2TIMMessage msg) {
    if (msg.getElemType() == V2TIMMessage.V2TIM_ELEM_TYPE_SOUND) {
        // 语音消息
        V2TIMSoundElem v2TIMSoundElem = msg.getSoundElem();
        // 语音 ID,内部标识，可用于外部缓存 key
        String uuid = v2TIMSoundElem.getUUID();
        // 语音文件大小
        int dataSize = v2TIMSoundElem.getDataSize();
        // 语音时长
        int duration = v2TIMSoundElem.getDuration();
        // 设置语音文件路径 soundPath，这里可以用 uuid 作为标识，避免重复下载
```

```
        String soundPath = "/sdcard/im/sound/" + "myUserID" + uuid;
        File imageFile = new File(soundPath);
        // 判断 soundPath 下有没有已经下载过的语音文件
        if (!imageFile.exists()) {
            v2TIMSoundElem.downloadSound(soundPath, new V2TIMDownloadCallback() {
                @Override
                public void onProgress(V2TIMElem.V2ProgressInfo progressInfo) {
                    // 下载进度回调：已下载大小 v2ProgressInfo.getCurrentSize()；总文件大
                }
                @Override
                public void onError(int code, String desc) {
                    // 下载失败
                }
                @Override
                public void onSuccess() {
                    // 下载完成
                }
            });
        } else {
            // 文件已存在
        }
    }
}
```



```
- (void)onRecvNewMessage:(V2TIMMessage *)msg {
    if (msg.elemType == V2TIM_ELEM_TYPE_SOUND) {
        V2TIMSoundElem *soundElem = msg.soundElem;
        // 语音 ID,内部标识，可用于外部缓存 key
        NSString *uuid = soundElem.uuid;
        // 语音文件大小
        int dataSize = soundElem.dataSize;
        // 语音时长
        int duration = soundElem.duration;
        // 设置语音文件路径 soundPath，这里可以用 uuid 作为标识，避免重复下载
        NSString *soundPath = [NSTemporaryDirectory() stringByAppendingPathComponen
```

```
        // 判断 soundPath 下有没有已经下载过的语音文件
        if (![[NSFileManager defaultManager] fileExistsAtPath:soundPath]) {
            // 下载语音
            [soundElem downloadSound:soundPath progress:^(NSInteger curSize, NSInte
                // 下载进度
                NSLog(@"下载语音进度：curSize：%lu,totalSize:%lu",curSize,totalSize);
            } succ:^{
                // 下载成功
                NSLog(@"下载语音完成");
            } fail:^(int code, NSString *msg) {
                // 下载失败
                NSLog(@"下载语音失败：code：%d,msg:%@",code,msg);
            }];
        } else {
            // 语音已存在
        }
        NSLog(@"语音信息：uuid:%@, dataSize:%d, duration:%d, soundPath:%@", uuid, dat
    }
}
```



```
class DownloadCallback final : public V2TIMDownloadCallback {
public:
    using SuccessCallback = std::function<void()>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using DownLoadProgressCallback = std::function<void(uint64_t, uint64_t)>;

    DownloadCallback() = default;
    ~DownloadCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     DownLoadProgressCallback download_progress_callback) {
```

```
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        download_progress_callback_ = std::move(download_progress_callback);
    }
    void OnSuccess() override {
        if (success_callback_) {
            success_callback_();
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnDownLoadProgress(uint64_t currentSize, uint64_t totalSize) override {
        if (download_progress_callback_) {
            download_progress_callback_(currentSize, totalSize);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    DownLoadProgressCallback download_progress_callback_;
};

class AdvancedMsgListener final : public V2TIMAdvancedMsgListener {
public:
    /**
     * 收到新消息
     *
     * @param message 消息
     */
    void OnRecvNewMessage(const V2TIMMessage& message) override {
        // 解析出 groupID 和 userID
        V2TIMString groupID = message.groupID;
        V2TIMString userID = message.userID;

        // 判断当前是单聊还是群聊：
        // 如果 groupID 不为空，表示此消息为群聊；如果 userID 不为空，表示此消息为单聊

        // 解析出 message 中的语音消息
        if (message.elemList.Size() == 1) {
            V2TIMElem* elem = message.elemList[0];
            if (elem->elemType == V2TIMElemType::V2TIM_ELEM_TYPE_SOUND) {
                // 语音消息
                auto soundElem = static_cast<V2TIMSoundElem*>(elem);
```

```
                // 语音消息内部 ID
                V2TIMString uuid = soundElem->uuid;
                // 语音数据大小
                uint64_t dataSize = soundElem->dataSize;
                // 语音长度（秒）
                uint32_t duration = soundElem->duration;
                // 设置语音文件路径，这里可以用 uuid 作为标识，避免重复下载
                std::filesystem::path soundPath = u8"./File/Sound/"s + uuid.CString
                // 判断 soundPath 下有没有已经下载过的语音文件
                if (!std::filesystem::exists(soundPath)) {
                    std::filesystem::create_directories(soundPath.parent_path());
                    auto callback = new DownloadCallback{};
                    callback->SetCallback(
                        [=]() {
                            // 下载完成
                            delete callback;
                        },
                        [=](int error_code, const V2TIMString& error_message) {
                            // 下载失败
                            delete callback;
                        },
                        [=](uint64_t currentSize, uint64_t totalSize) {
                            // 下载进度回调：已下载大小 currentSize; 总文件大小 totalSize
                        });
                    soundElem->DownloadSound(soundPath.string().c_str(), callback);
                } else {
                    // 语音文件已存在
                }
            }
        }
    }
    // 其他成员 ...
};

// 添加高级消息的事件监听器，注意在移除监听器之前需要保持 advancedMsgListener 的生命期，以免接收
AdvancedMsgListener advancedMsgListener;
V2TIMManager::GetInstance()->GetMessageManager()->AddAdvancedMsgListener(&advancedM
```

文件消息

接收端收到文件消息后，我们推荐您调用 SDK 的 `downloadFile` (Android / iOS & Mac / Windows) 将文件下载到本地，再获取本地文件展示。

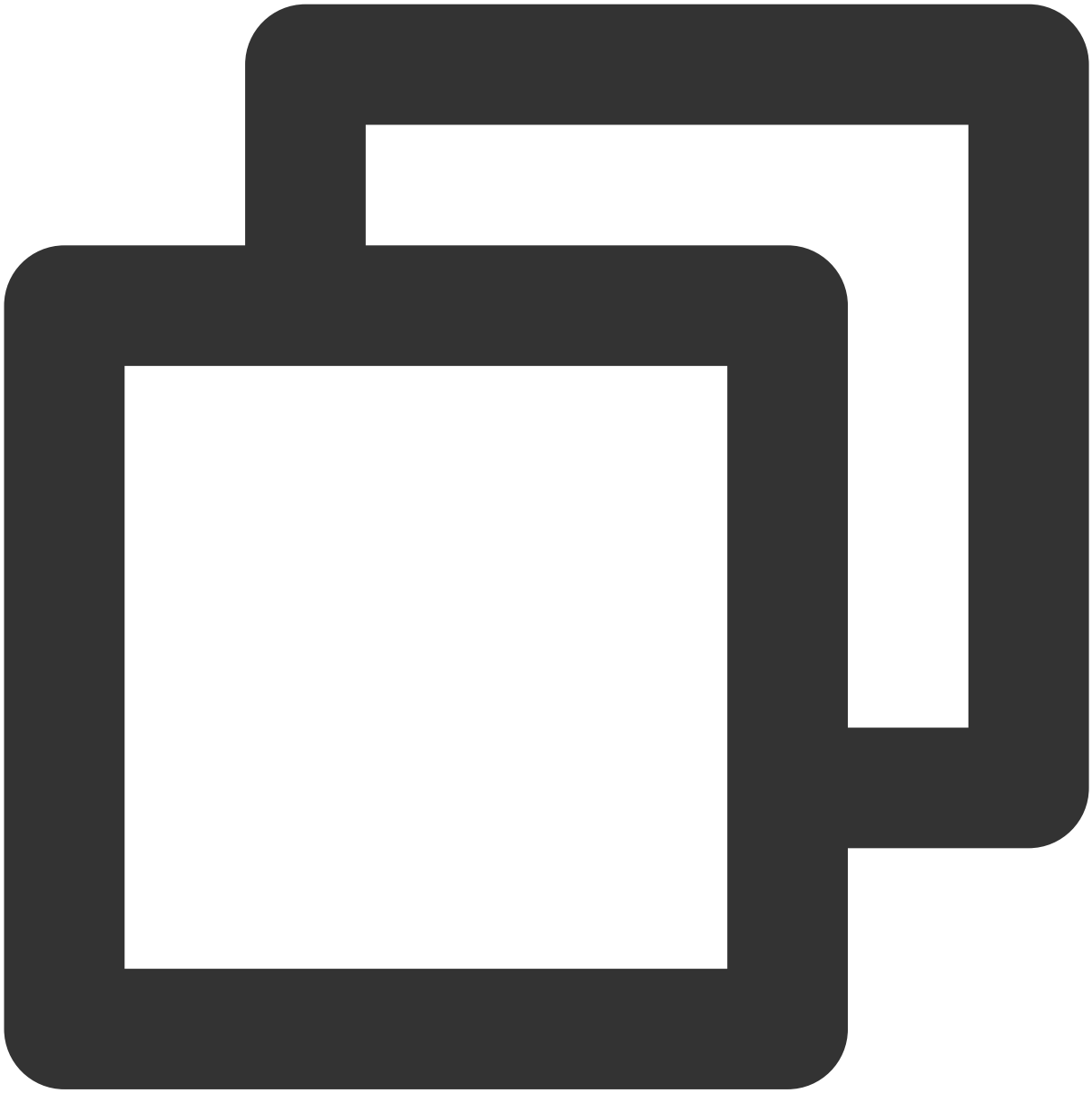
为了避免重复下载，节省资源，我们推荐您将 `V2TIMFileElem` 对象的 `uuid` 属性值设置到文件的下载路径中，作为文件的标识。

示例代码向您演示如何从 `V2TIMMessage` 中解析出文件消息内容：

Android

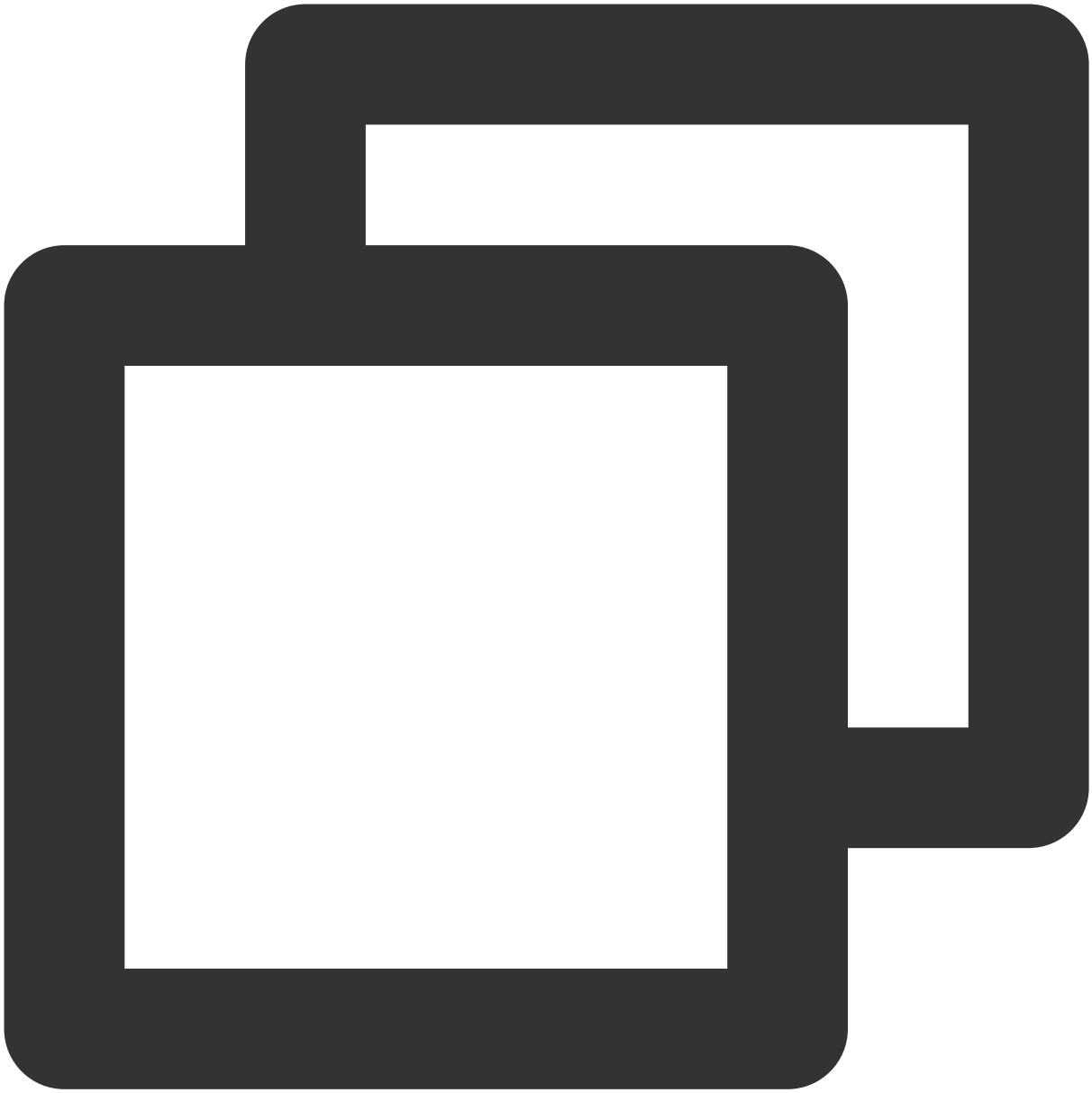
iOS & Mac

Windows



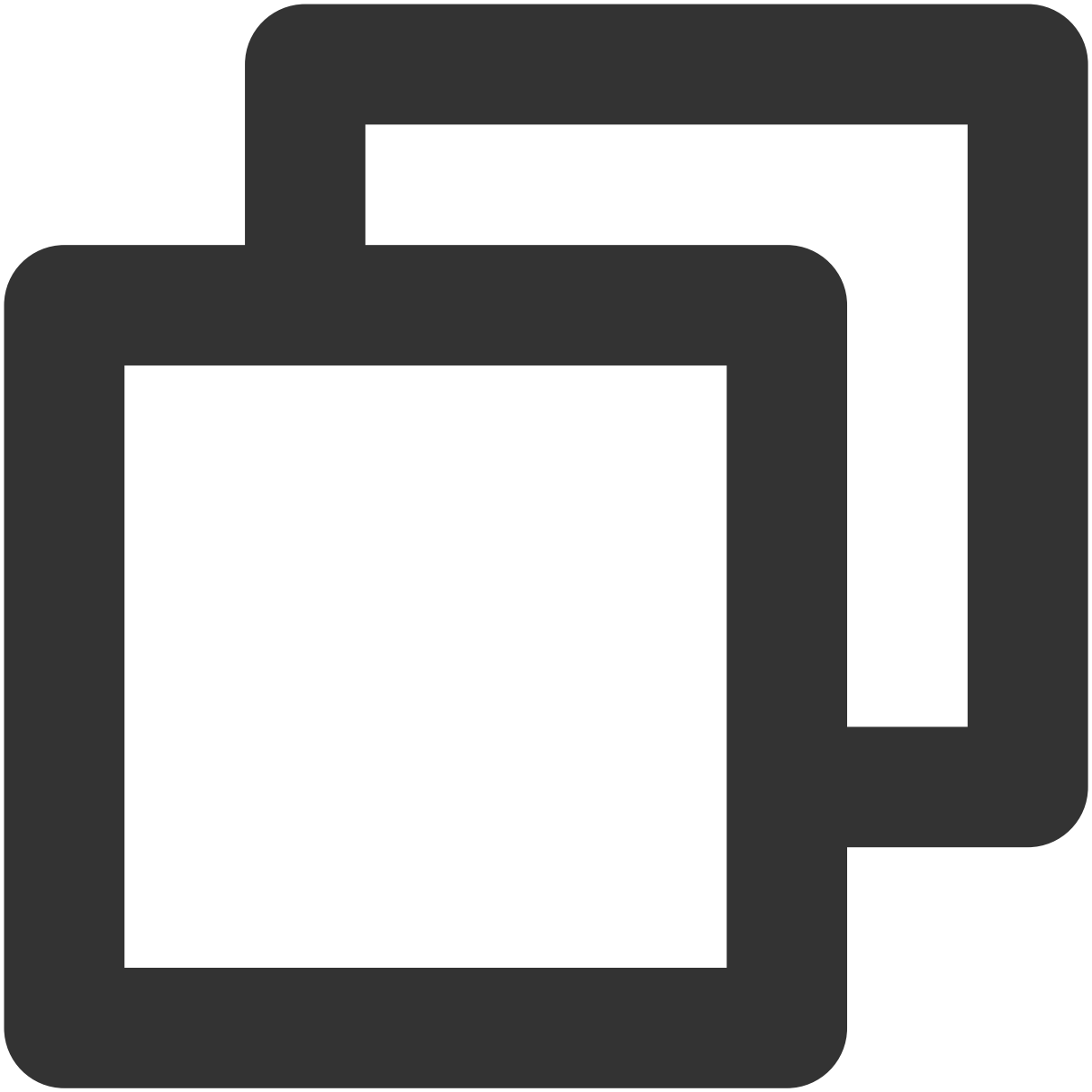
```
public void onRecvNewMessage(V2TIMMessage msg) {
    if (msg.getElemType() == V2TIMMessage.V2TIM_ELEM_TYPE_FILE) {
        // 文件消息
        V2TIMFileElem v2TIMFileElem = msg.getFileElem();
        // 文件 ID,内部标识，可用于外部缓存 key
        String uuid = v2TIMFileElem.getUUID();
        // 文件名称
```

```
        String fileName = v2TIMFileElem.getFileName();
        // 文件大小
        int fileSize = v2TIMFileElem.getFileSize();
        // 设置文件路径，这里可以用 uuid 作为标识，避免重复下载
        String filePath = "/sdcard/im/file/" + "myUserID" + uuid;
        File file = new File(filePath);
        if (!file.exists()) {
            v2TIMFileElem.downloadFile(filePath, new V2TIMDownloadCallback() {
                @Override
                public void onProgress(V2TIMElem.V2ProgressInfo progressInfo) {
                    // 下载进度回调：已下载大小 v2ProgressInfo.getCurrentSize()；总文件大
                }
                @Override
                public void onError(int code, String desc) {
                    // 下载失败
                }
                @Override
                public void onSuccess() {
                    // 下载完成
                }
            });
        } else {
            // 文件已存在
        }
    }
}
```



```
- (void)onRecvNewMessage:(V2TIMMessage *)msg {
    if (msg.elemType == V2TIM_ELEM_TYPE_FILE) {
        V2TIMFileElem *fileElem = msg.fileElem;
        // 文件 ID,内部标识，可用于外部缓存 key
        NSString *uuid = fileElem.uuid;
        // 文件名称
        NSString *filename = fileElem.filename;
        // 文件大小
        int fileSize = fileElem.fileSize;
        // 设置文件路径，这里可以用 uuid 作为标识，避免重复下载
        NSString *filePath = [NSTemporaryDirectory() stringByAppendingPathComponent
```

```
        if (![[NSFileManager defaultManager] fileExistsAtPath:filePath]) {
            // 下载文件
            [fileElem downloadFile:filePath progress:^(NSInteger curSize, NSInteger
                // 下载进度
                NSLog(@"%@", [NSString stringWithFormat:@"下载文件进度：curSize：%lu,t
            } succ:^{
                // 下载成功
                NSLog(@"下载文件完成");
            } fail:^(int code, NSString *msg) {
                // 下载失败
                NSLog(@"%@", [NSString stringWithFormat:@"下载文件失败：code：%d,msg:%
            }];
        } else {
            // 文件已存在
        }
        NSLog(@"文件信息：uuid:%@, filename:%@, fileSize:%d, filePath:%@", uuid, file
    }
}
```



```
class DownloadCallback final : public V2TIMDownloadCallback {
public:
    using SuccessCallback = std::function<void()>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;
    using DownLoadProgressCallback = std::function<void(uint64_t, uint64_t)>;

    DownloadCallback() = default;
    ~DownloadCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
                     DownLoadProgressCallback download_progress_callback) {
```

```
        success_callback_ = std::move(success_callback);
        error_callback_ = std::move(error_callback);
        download_progress_callback_ = std::move(download_progress_callback);
    }
    void OnSuccess() override {
        if (success_callback_) {
            success_callback_();
        }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
        if (error_callback_) {
            error_callback_(error_code, error_message);
        }
    }
    void OnDownLoadProgress(uint64_t currentSize, uint64_t totalSize) override {
        if (download_progress_callback_) {
            download_progress_callback_(currentSize, totalSize);
        }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
    DownLoadProgressCallback download_progress_callback_;
};

class AdvancedMsgListener final : public V2TIMAdvancedMsgListener {
public:
    /**
     * 收到新消息
     *
     * @param message 消息
     */
    void OnRecvNewMessage(const V2TIMMessage& message) override {
        // 解析出 groupID 和 userID
        V2TIMString groupID = message.groupID;
        V2TIMString userID = message.userID;

        // 判断当前是单聊还是群聊：
        // 如果 groupID 不为空，表示此消息为群聊；如果 userID 不为空，表示此消息为单聊

        // 解析出 message 中的文件消息
        if (message.elemList.Size() == 1) {
            V2TIMElem* elem = message.elemList[0];
            if (elem->elemType == V2TIMElemType::V2TIM_ELEM_TYPE_FILE) {
                // 文件消息
                auto fileElem = static_cast<V2TIMFileElem*>(elem);
```

```
                // 文件 ID,内部标识，可用于外部缓存 key
                V2TIMString uuid = fileElem->uuid;
                // 文件显示名称
                V2TIMString filename = fileElem->filename;
                // 文件大小
                uint64_t fileSize = fileElem->fileSize;
                // 设置文件路径，这里可以用 uuid 作为标识，避免重复下载
                std::filesystem::path filePath = u8"./File/File/"s + uuid.CString()
                // 判断 filePath 下有没有已经下载过的文件
                if (!std::filesystem::exists(filePath)) {
                    std::filesystem::create_directories(filePath.parent_path());
                    auto callback = new DownloadCallback{};
                    callback->SetCallback(
                        [=]() {
                            // 下载完成
                            delete callback;
                        },
                        [=](int error_code, const V2TIMString& error_message) {
                            // 下载失败
                            delete callback;
                        },
                        [=](uint64_t currentSize, uint64_t totalSize) {
                            // 下载进度回调：已下载大小 currentSize; 总文件大小 totalSize
                        });
                    fileElem->DownloadFile(filePath.string().c_str(), callback);
                } else {
                    // 文件已存在
                }
            }
        }
    }
    // 其他成员 ...
};

// 添加高级消息的事件监听器，注意在移除监听器之前需要保持 advancedMsgListener 的生命期，以免接收
AdvancedMsgListener advancedMsgListener;
V2TIMManager::GetInstance()->GetMessageManager()->AddAdvancedMsgListener(&advancedM
```

地理位置消息

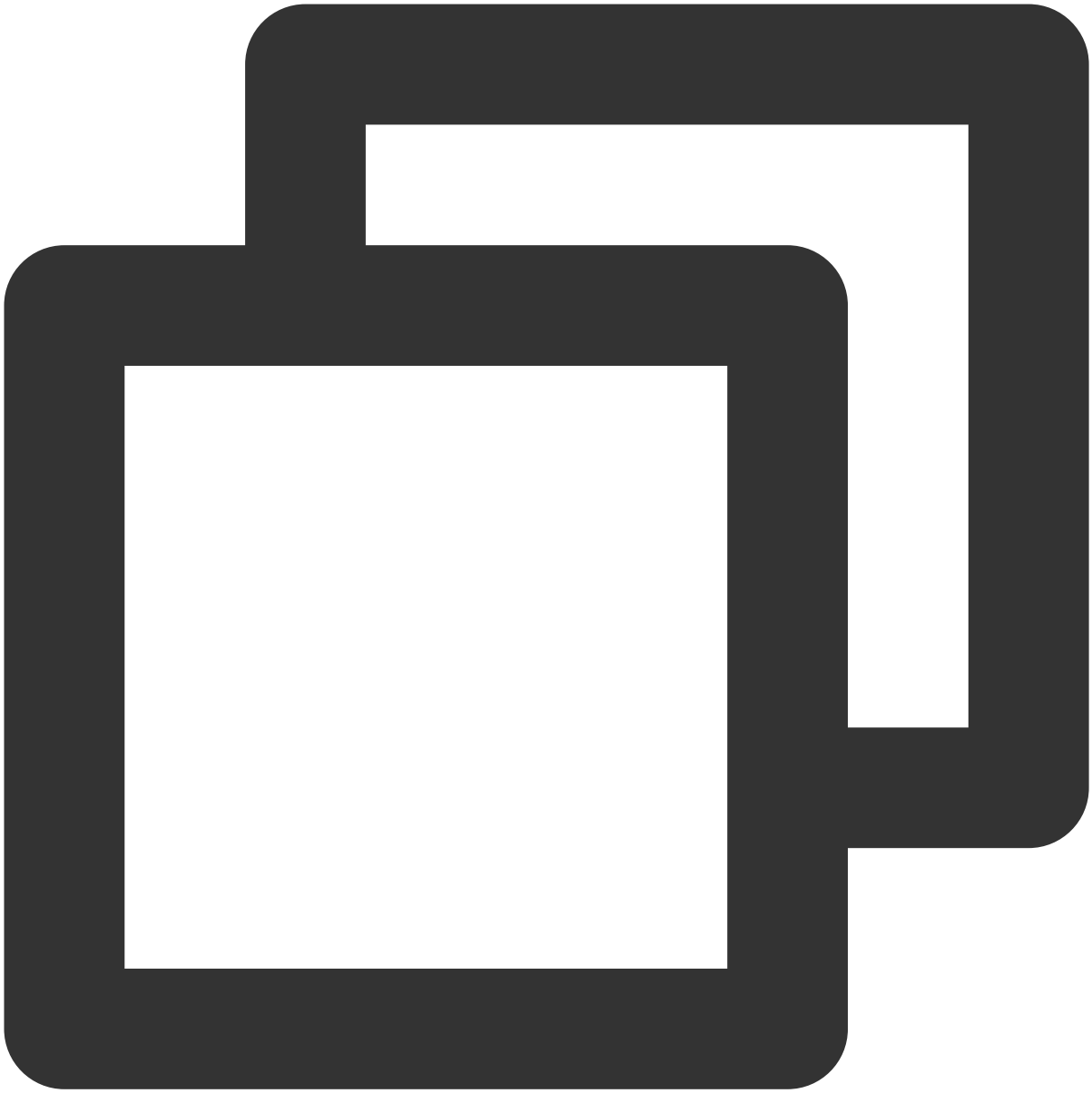
接收到地理位置消息后，接收放可直接从 `V2TIMLocationElem` 中解析出经纬度信息。

示例代码向您演示如何从 `V2TIMMessage` 中解析出地理位置消息内容：

Android

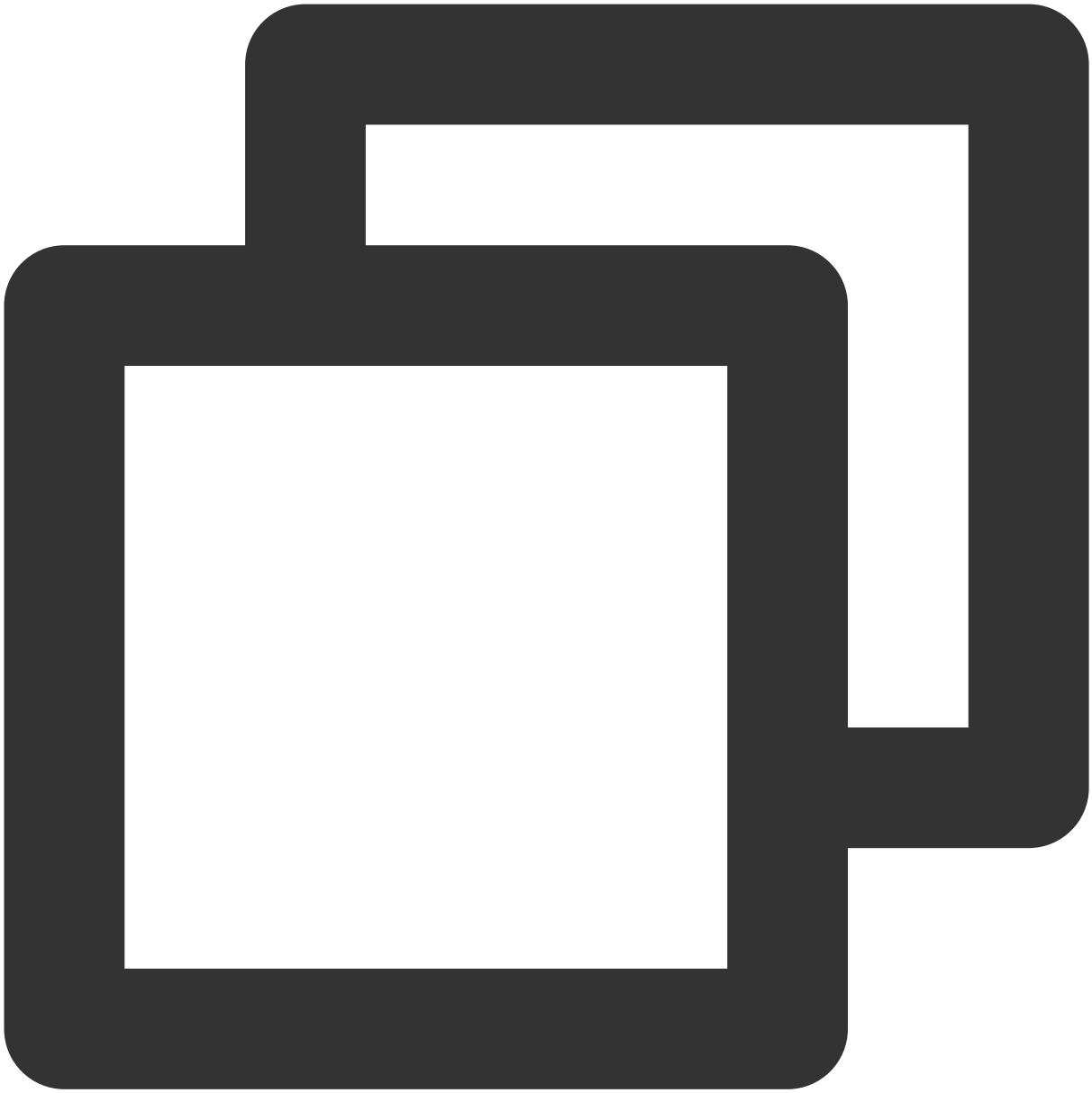
iOS & Mac

Windows



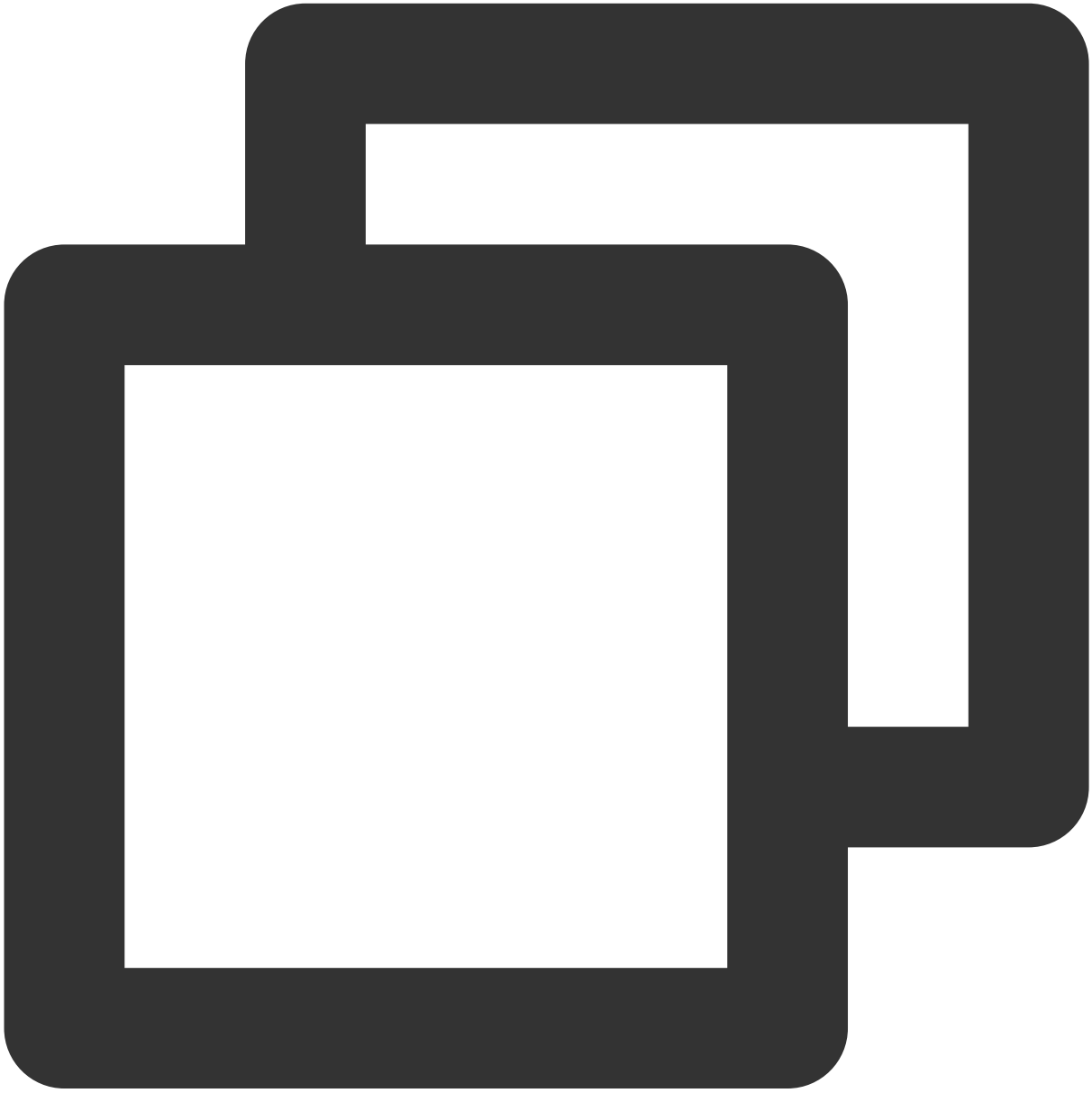
```
public void onRecvNewMessage(V2TIMMessage msg) {
    if (msg.getElemType() == V2TIMMessage.V2TIM_ELEM_TYPE_LOCATION) {
        // 地理位置消息
        V2TIMLocationElem v2TIMLocationElem = msg.getLocationElem();
        // 地理位置信息描述
        String desc = v2TIMLocationElem.getDesc();
        // 经度
        double longitude = v2TIMLocationElem.getLongitude();
        // 纬度
        double latitude = v2TIMLocationElem.getLatitude();
    }
```

```
}
```



```
- (void)onRecvNewMessage:(V2TIMMessage *)msg {
    if (msg.elemType == V2TIM_ELEM_TYPE_LOCATION) {
        V2TIMLocationElem *locationElem = msg.locationElem;
        // 地理位置信息描述
        NSString *desc = locationElem.desc;
        // 经度
        double longitude = locationElem.longitude;
        // 纬度
        double latitude = locationElem.latitude;
```

```
        NSLog(@"地理位置信息：desc：%@, longitude:%f, latitude:%f", desc, longitude, l
    }
}
```



```
class AdvancedMsgListener final : public V2TIMAdvancedMsgListener {
public:
    /**
     * 收到新消息
     *
     * @param message 消息
     */
```

```
    void OnRecvNewMessage(const V2TIMMessage& message) override {
        // 解析出 groupID 和 userID
        V2TIMString groupID = message.groupID;
        V2TIMString userID = message.userID;

        // 判断当前是单聊还是群聊：
        // 如果 groupID 不为空，表示此消息为群聊；如果 userID 不为空，表示此消息为单聊

        // 解析出 message 中的地理位置消息
        if (message.elemList.Size() == 1) {
            V2TIMElem* elem = message.elemList[0];
            if (elem->elemType == V2TIMElemType::V2TIM_ELEM_TYPE_LOCATION) {
                // 地理位置消息
                auto locationElem = static_cast<V2TIMLocationElem*>(elem);
                // 地理位置描述信息
                V2TIMString desc = locationElem->desc;
                // 经度，发送消息时设置
                double longitude = locationElem->longitude;
                // 纬度，发送消息时设置
                double latitude = locationElem->latitude;
            }
        }
    }
    // 其他成员 ...
};

// 添加高级消息的事件监听器，注意在移除监听器之前需要保持 advancedMsgListener 的生命期，以免接
AdvancedMsgListener advancedMsgListener;
V2TIMManager::GetInstance()->GetMessageManager()->AddAdvancedMsgListener(&advancedM
```

表情消息

SDK 仅为表情消息提供消息透传的通道，消息内容字段参考 V2TIMFaceElem (Android / iOS & Mac / Windows) 定义。其中 `index` 和 `data` 的内容由客户自定义。

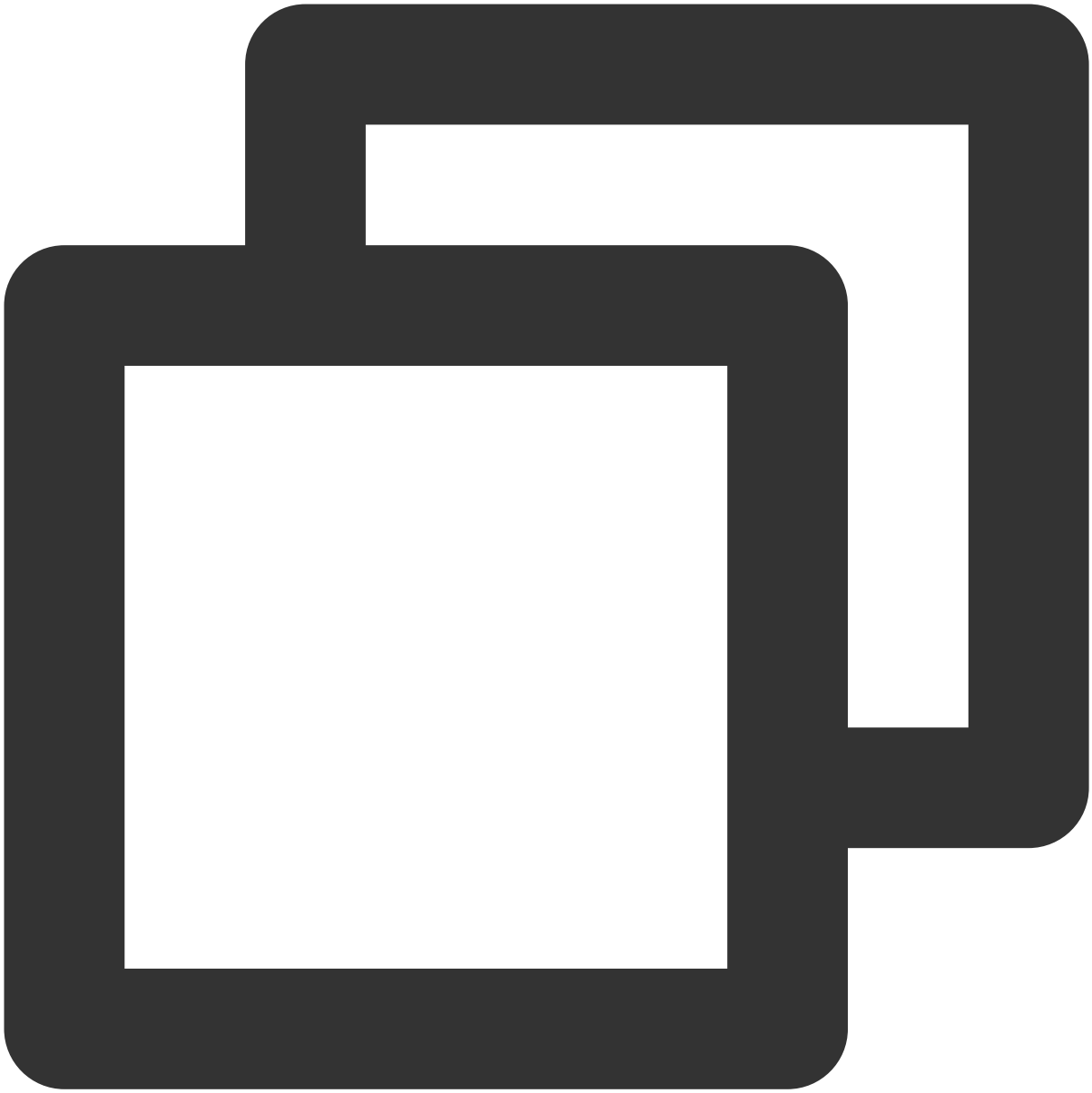
例如发送方可设置 index = 1, data = "x12345"，表示 “微笑” 表情。 接收方收到表情消息后解析出 1 和 "x12345"，按照预设的规则将其展示为 “微笑” 表情。

示例代码向您演示如何从 `V2TIMMessage` 中解析出表情消息内容：

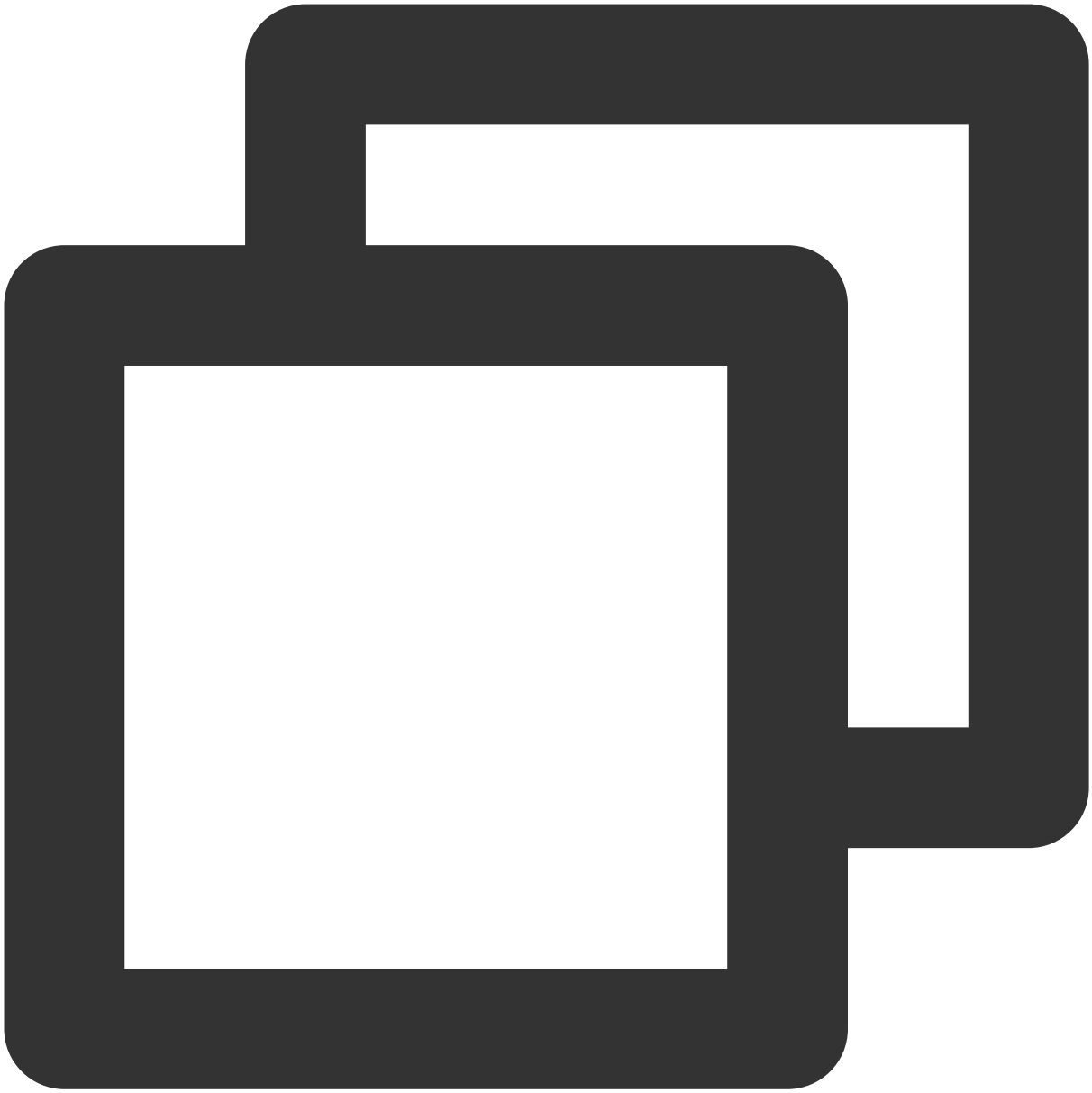
Android

iOS & Mac

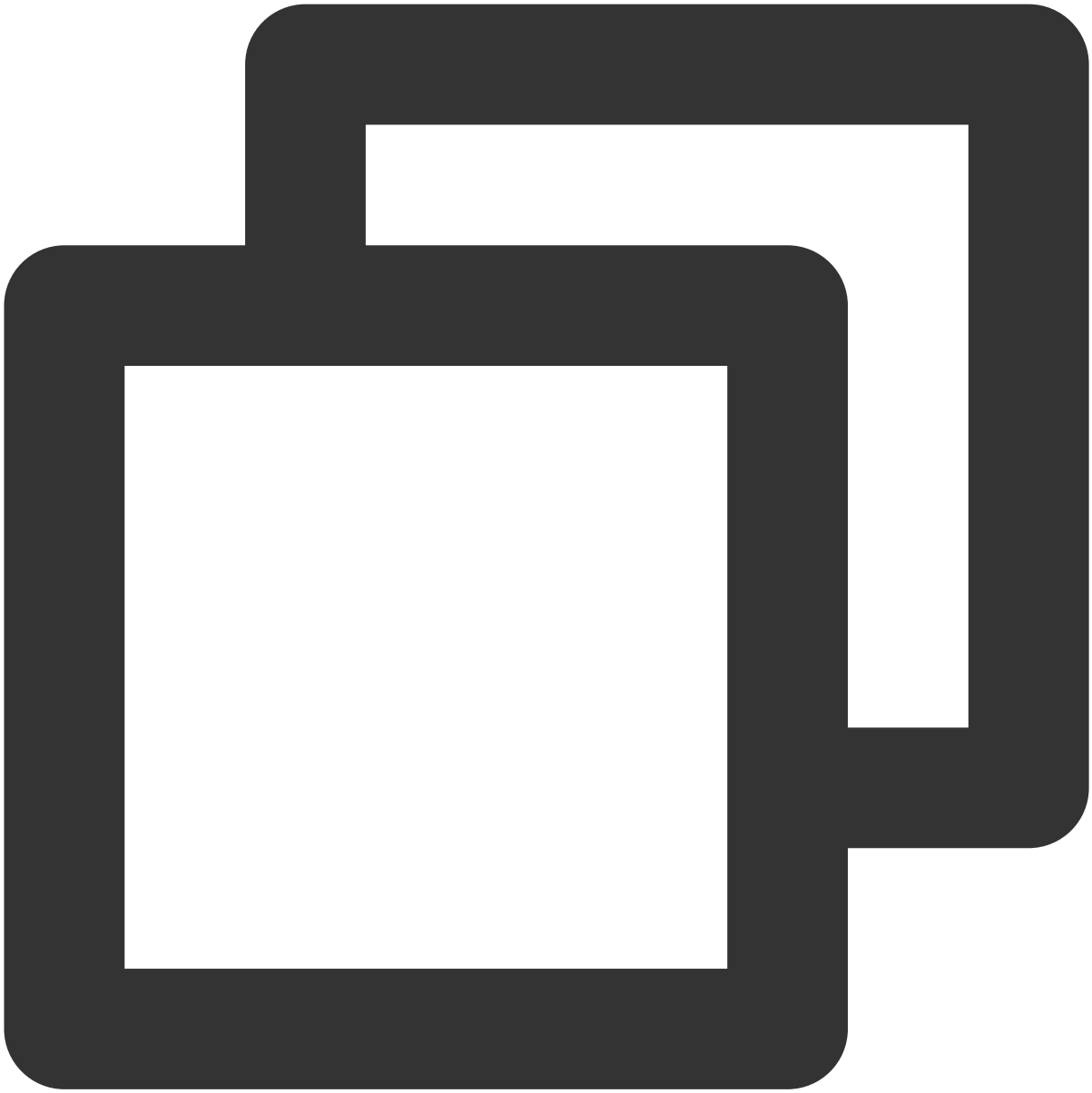
Windows



```
public void onRecvNewMessage(V2TIMMessage msg) {
    if (msg.getElemType() == V2TIMMessage.V2TIM_ELEM_TYPE_FACE) {
        // 表情消息
        V2TIMFaceElem v2TIMFaceElem = msg.getFaceElem();
        // 表情所在的位置
        int index = v2TIMFaceElem.getIndex();
        // 表情自定义数据
        byte[] data = v2TIMFaceElem.getData();
    }
}
```



```
- (void)onRecvNewMessage:(V2TIMMessage *)msg {
    if (msg.elemType == V2TIM_ELEM_TYPE_FACE) {
        V2TIMFaceElem *faceElem = msg.faceElem;
        // 表情所在的位置
        int index = faceElem.index;
        // 表情自定义数据
        NSData *data = faceElem.data;
        NSLog(@"表情信息：index: %d, data: %@", index, data);
    }
}
```



```
class AdvancedMsgListener final : public V2TIMAdvancedMsgListener {
public:
    /**
     * 收到新消息
     *
     * @param message 消息
     */
    void OnRecvNewMessage(const V2TIMMessage& message) override {
        // 解析出 groupID 和 userID
        V2TIMString groupID = message.groupID;
        V2TIMString userID = message.userID;
```

```
        // 判断当前是单聊还是群聊：
        // 如果 groupID 不为空，表示此消息为群聊；如果 userID 不为空，表示此消息为单聊

        // 解析出 message 中的表情消息
        if (message.elemList.Size() == 1) {
            V2TIMElem* elem = message.elemList[0];
            if (elem->elemType == V2TIMElemType::V2TIM_ELEM_TYPE_FACE) {
                // 表情消息
                auto faceElem = static_cast<V2TIMFaceElem*>(elem);
                // 表情所在的位置
                uint32_t index = faceElem->index;
                // 表情自定义数据
                V2TIMBuffer data = faceElem->data;
            }
        }
    }
    // 其他成员 ...
};

// 添加高级消息的事件监听器，注意在移除监听器之前需要保持 advancedMsgListener 的生命期，以免接收
AdvancedMsgListener advancedMsgListener;
V2TIMManager::GetInstance()->GetMessageManager()->AddAdvancedMsgListener(&advancedM
```

接收广播消息

您只能调用 REST API 直播群广播消息 向同一个 AppID 下面所有的直播群发送广播消息，所有直播群在线群成员均能收到此消息。

SDK 不能发送广播消息，只能接收来自于 REST API 的消息。

接收消息依然是在上文所述的 `onRecvNewMessage` 通知里。您可以在收到消息后，根据消息的 `isBroadcastMessage` 判断该消息是否是广播消息。

注意：

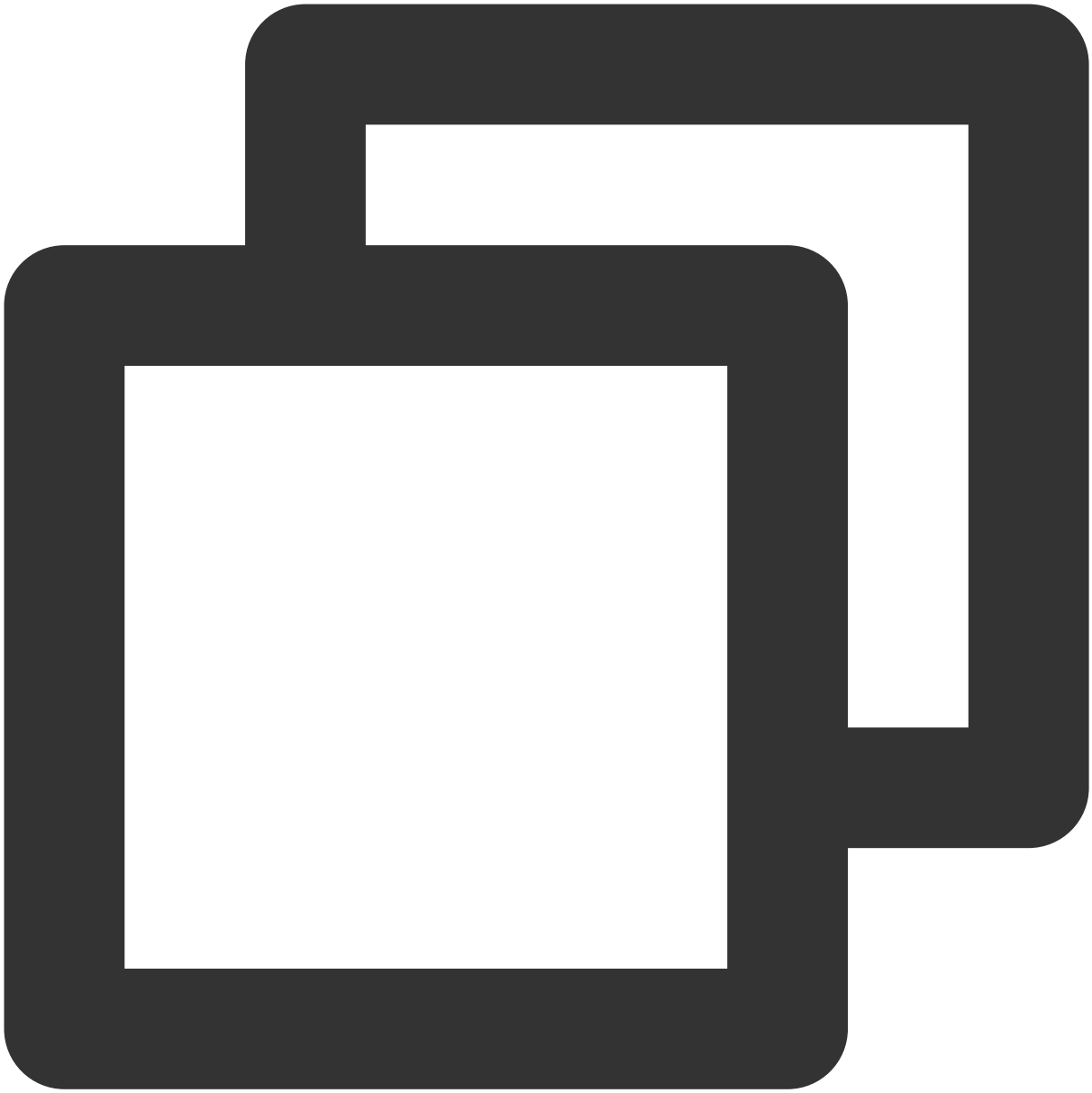
1. 该功能仅对进阶版客户开放，购买进阶版后可使用。
2. 仅增强版 SDK 6.5.2803 及以上版本支持。
3. 仅直播群支持广播消息，其他类型群组暂不支持。

示例代码如下：

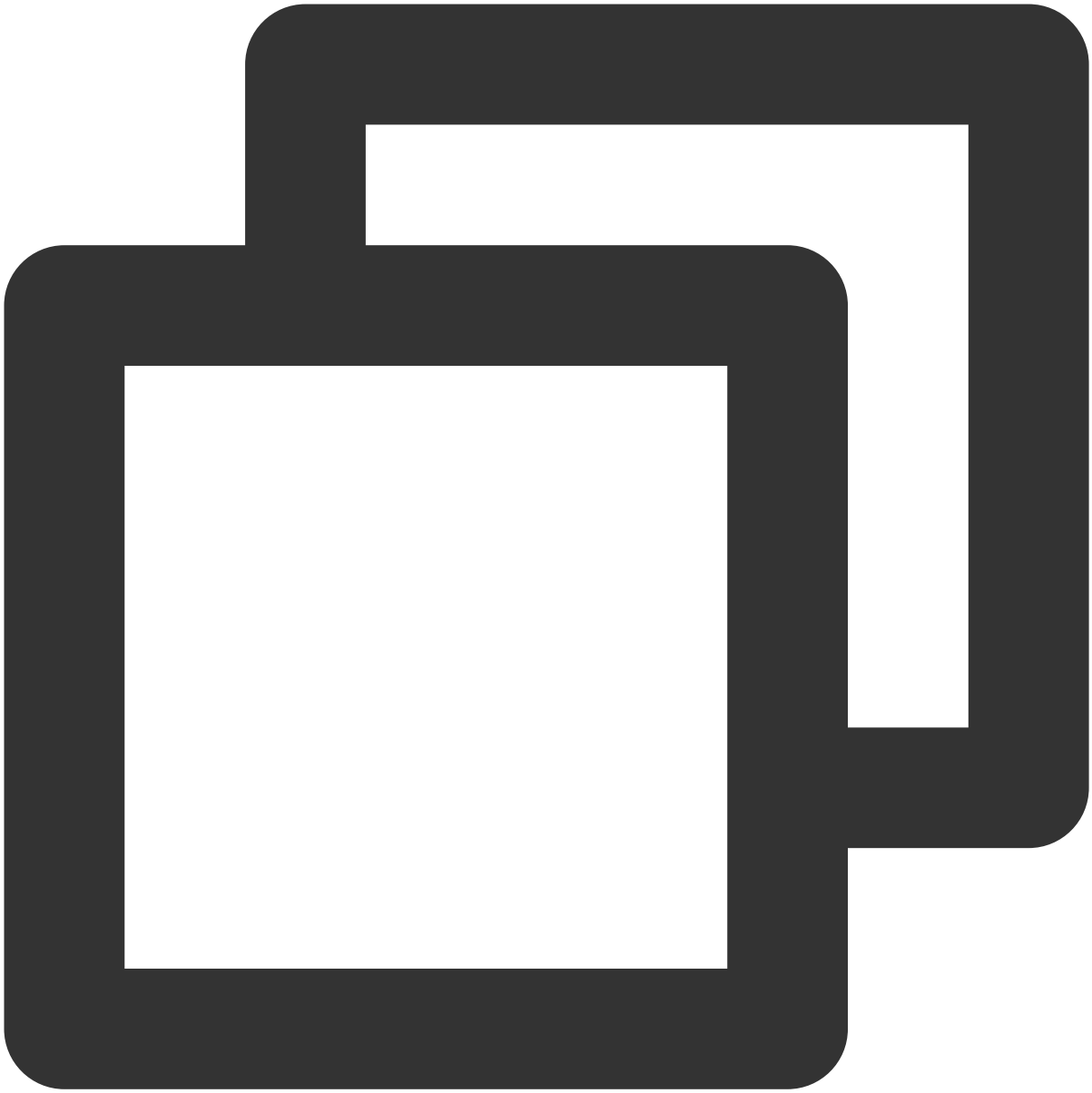
Android

iOS & Mac

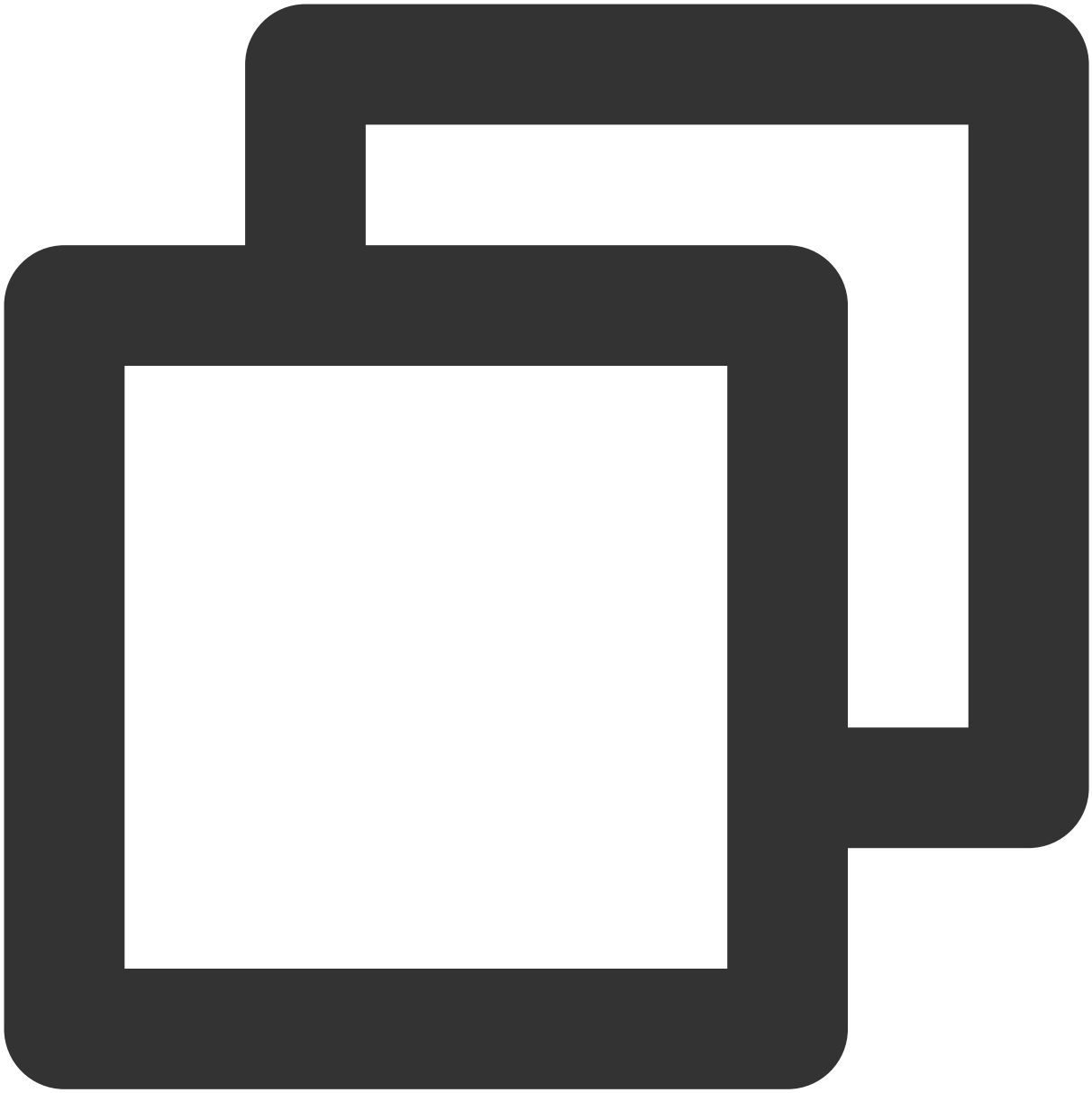
Windows



```
public void onRecvNewMessage(V2TIMMessage msg) {
    if (msg.isBroadcastMessage) {
        // 收到了广播消息
    }
}
```



```
- (void)onRecvNewMessage:(V2TIMMessage *)msg {
    if (msg.isBroadcastMessage) {
        // 收到了广播消息
    }
}
```



```
class AdvancedMsgListener final : public V2TIMAdvancedMsgListener {
public:
    /**
     * 收到新消息
     *
     * @param message 消息
     */
    void OnRecvNewMessage(const V2TIMMessage& message) override {
        if (message.isBroadcastMessage) {
            // 收到了广播消息
        }
```

```
    }
    // 其他成员 ...
};

// 添加高级消息的事件监听器，注意在移除监听器之前需要保持 advancedMsgListener 的生命期，以免接
AdvancedMsgListener advancedMsgListener;
V2TIMManager::GetInstance()->GetMessageManager()->AddAdvancedMsgListener(&advancedM
```

接收多个 Elem 的消息

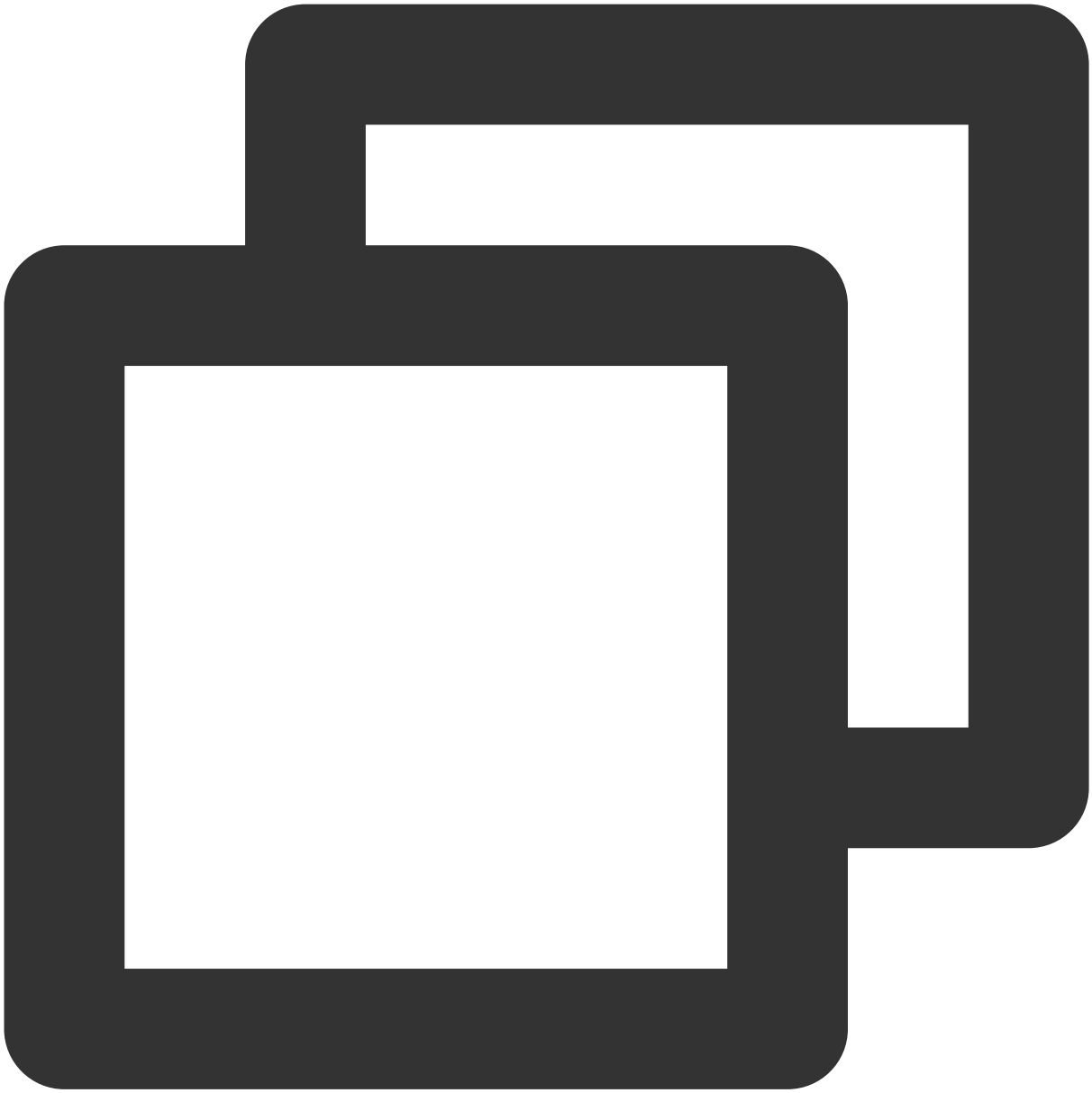
1. 通过 Message 对象正常解析出第一个 Elem 对象。
2. 通过第一个 Elem 对象的 `nextElem` 方法获取下一个 Elem 对象，如果下一个 Elem 对象存在，会返回 Elem 对象实例，如果不存在，会返回 nil/null。

示例代码如下：

Android

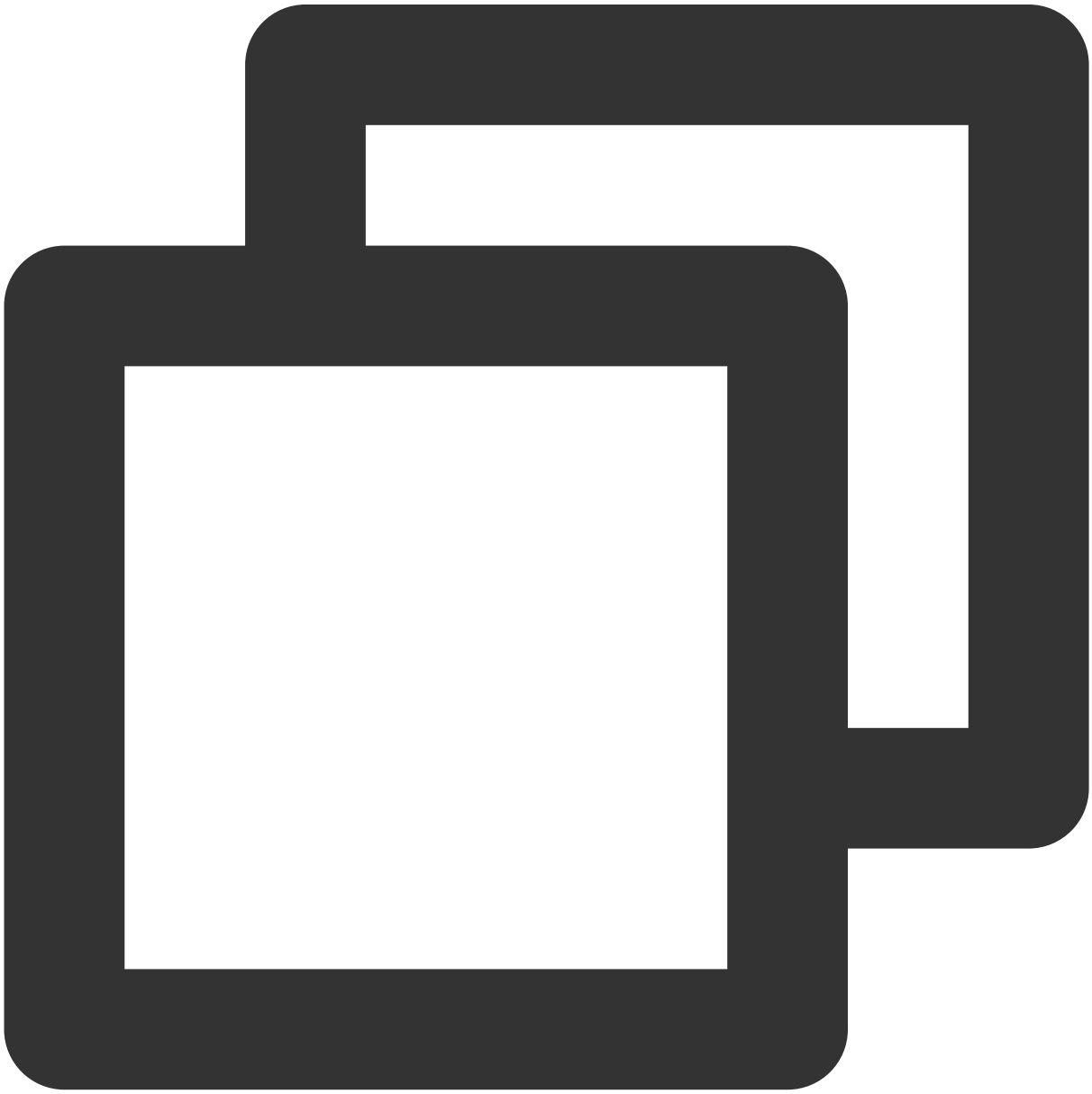
iOS & Mac

Windows

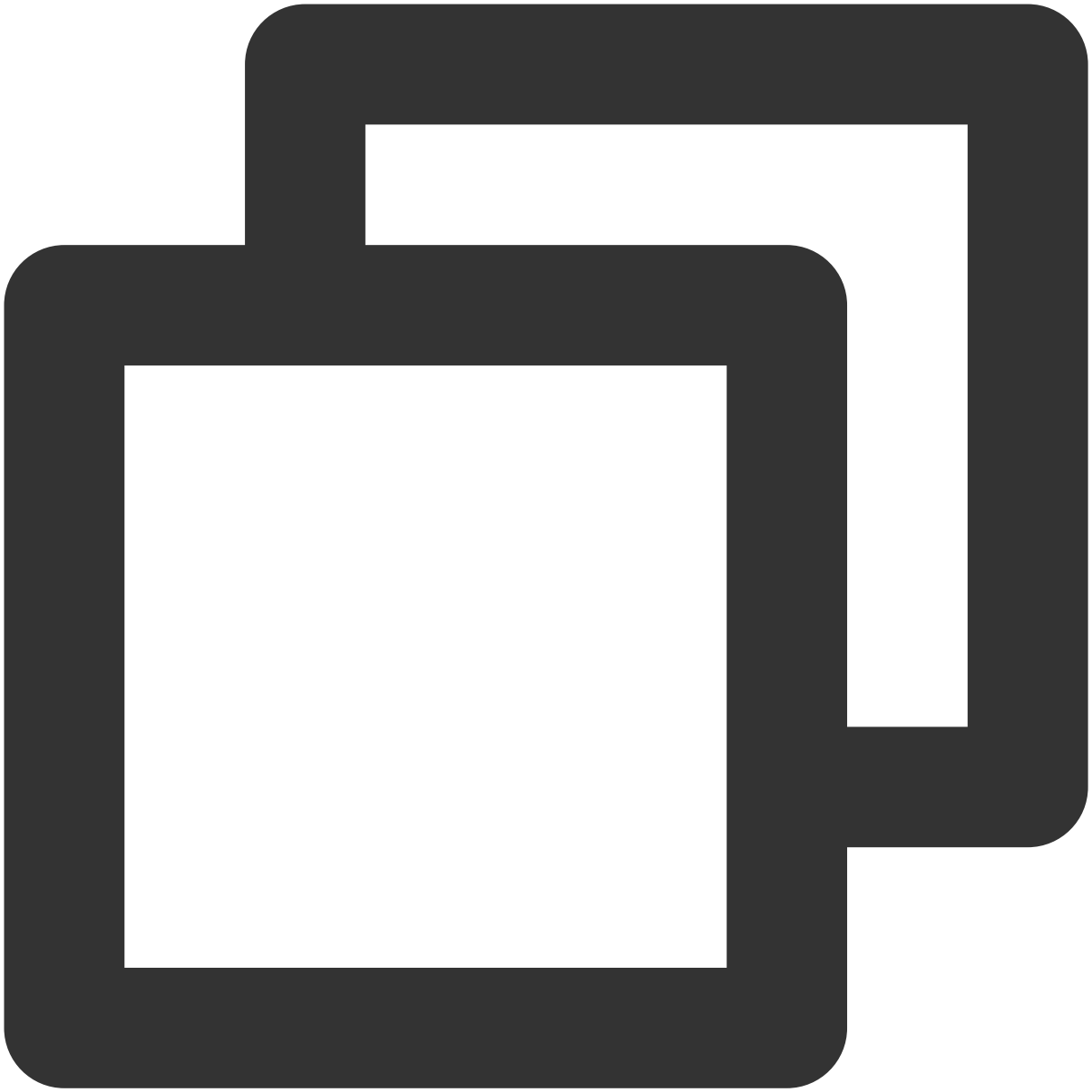


```
@Override
public void onRecvNewMessage(V2TIMMessage msg) {
    // 查看第一个 Elem
    int elemType = msg.getElemType();
    if (elemType == V2TIMMessage.V2TIM_ELEM_TYPE_TEXT) {
        // 文本消息
        V2TIMTextElem v2TIMTextElem = msg.getTextElem();
        String text = v2TIMTextElem.getText();
        // 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
    }
}
```



```
- (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
    }
}
```



```
class AdvancedMsgListener final : public V2TIMAdvancedMsgListener {
public:
    /**
     * 收到新消息
     *
     * @param message 消息
     */
    void OnRecvNewMessage(const V2TIMMessage& message) override {
        // 解析出 groupID 和 userID
        V2TIMString groupID = message.groupID;
        V2TIMString userID = message.userID;
```

```
// 判断当前是单聊还是群聊：
// 如果 groupID 不为空，表示此消息为群聊；如果 userID 不为空，表示此消息为单聊

// 解析出 message 中的多个 Elem 消息
// 所有的 Elem 都保存在 message.elemList 中，访问这些 Elem 需要遍历 message.elemI
for (size_t i = 0; i < message.elemList.Size(); ++i) {
    V2TIMElem* elem = message.elemList[i];
    switch (elem->elemType) {
        case V2TIMElemType::V2TIM_ELEM_TYPE_NONE: {
            // 未知消息
        } break;
        case V2TIMElemType::V2TIM_ELEM_TYPE_TEXT: {
            // 文本消息
            auto textElem = static_cast<V2TIMTextElem*>(elem);
        } break;
        case V2TIMElemType::V2TIM_ELEM_TYPE_CUSTOM: {
            // 自定义消息
            auto customElem = static_cast<V2TIMCustomElem*>(elem);
        } break;
        case V2TIMElemType::V2TIM_ELEM_TYPE_IMAGE: {
            // 图片消息
            auto imageElem = static_cast<V2TIMImageElem*>(elem);
        } break;
        case V2TIMElemType::V2TIM_ELEM_TYPE_SOUND: {
            // 语音消息
            auto soundElem = static_cast<V2TIMSoundElem*>(elem);
        } break;
        case V2TIMElemType::V2TIM_ELEM_TYPE_VIDEO: {
            // 视频消息
            auto videoElem = static_cast<V2TIMVideoElem*>(elem);
        } break;
        case V2TIMElemType::V2TIM_ELEM_TYPE_FILE: {
            // 文件消息
            auto fileElem = static_cast<V2TIMFileElem*>(elem);
        } break;
        case V2TIMElemType::V2TIM_ELEM_TYPE_LOCATION: {
            // 地理位置消息
            auto locationElem = static_cast<V2TIMLocationElem*>(elem);
        } break;
        case V2TIMElemType::V2TIM_ELEM_TYPE_FACE: {
            // 表情消息
            auto faceElem = static_cast<V2TIMFaceElem*>(elem);
        } break;
        case V2TIMElemType::V2TIM_ELEM_TYPE_GROUP_TIPS: {
            // 群 Tips 消息
            auto mergerElem = static_cast<V2TIMMergerElem*>(elem);
```

```
                } break;
                case V2TIMElemType::V2TIM_ELEM_TYPE_MERGER: {
                    // 合并消息
                    auto groupTipsElem = static_cast<V2TIMGroupTipsElem*>(elem);
                } break;
                default: {
                } break;
            }
        }
    }
    // 其他成员 ...
};

// 添加高级消息的事件监听器，注意在移除监听器之前需要保持 advancedMsgListener 的生命期，以免接
AdvancedMsgListener advancedMsgListener;
V2TIMManager::GetInstance()->GetMessageManager()->AddAdvancedMsgListener(&advancedM
```

Web

最近更新时间：2023-08-03 11:01:36

功能描述

接收消息需要接入侧监听 MESSAGE_RECEIVED 事件。

监听事件

注意

请在调用 login 接口前调用此接口监听事件，避免漏掉 SDK 派发的事件。