即时通信

IMLib / IMKit

Android 5.X

2024-08-30

开发指导

更新时间:2024-08-30

欢迎使用融云即时通讯。本页面简单介绍了融云即时通讯架构、服务能力和 SDK 产品。

架构与服务

融云提供的即时通讯服务，不需要在 App 之外建立并行的用户体系，不用同步 App 下用户信息到融云，不影响 App 现有的系统架构与帐号体系，与现有业务体系能够实现完美融合。

融云的架构设计特点：

- 无需改变现有 App 的架构，直接嵌入现有代码框架中；
- 无需改变现有 App Server 的架构，独立部署一份用于用户授权的 Service 即可；
- 专注于提供通讯能力，使用私有的二进制通信协议，消息轻量、有序、不丢消息；
- 安全的身份认证和授权方式，无需担心 SDK 能力滥用（盗用身份的垃圾消息、垃圾群发）问题。

融云即时通讯产品支持单聊、群聊、超级群、聊天室 多种业务形态，提供丰富的客户端和服务端接口，大部分能力支持开箱即用。

业务类型介绍

单聊（Private）业务即一对一聊天。普通群组（Group）业务类似微信的群组。超级群与聊天室业务均不设用户总数上限。超级群（UltraGroup）[1] 类似 Discord，提供了一种新的群组业务形态，在超级群中提供公有/私有频道、用户组等功能，适用于构建超级社区。聊天室（Chatroom）只有在线用户可接收消息，广泛适用于直播、社区、游戏、广场交友、兴趣讨论等场景。融云的 IMKit 为 Android/iOS/Web 平台的单聊、普通群组业务提供了开箱即用的 UI 组件，其他情况下可以使用 IMLib SDK 构建您的业务体验。

单聊、群组、超级群、聊天室的主要差异如下：

功能	单聊（Private）	普通群组（Group）	超级群（UltraGroup）[1]	聊天室（Chatroom）
场景类比	类似微信私聊	类似微信群组	类似 Discord	聊天室
特性/优势	支持离线消息推送和历史消息记录漫游	支持离线消息推送和历史消息记录漫游，可用于兴趣群、办公群、客服服务沟通等	不限成员数量；支持修改已发消息；提供公有/私有频道、用户组等社群功能	不限成员数量；只有在线用户可接收消息，退出时清除本地历史消息
开通服务	不需要	不需要	需要	不需要
UI 组件	IMKit[2]	IMKit[2]	不提供	不提供
创建方式	无需创建	服务端 API	服务端 API	服务端 API；客户端加入时可自动创建
销毁/解散方式	不适用	服务端 API	服务端 API	服务端 API；具有自动销毁机制[3]
成员数量限制	不适用	群成员数上限 3000	不限	不限
用户加入限制	不适用	不限	最多加入 100 个群，每个群中可加入 50 个频道	默认仅可加入 1 个聊天室，可自行关闭限制[4]
获取加入前的消息	不适用	默认不允许，可关闭限制	默认不允许，可关闭限制	客户端加入聊天室即可获取最新消息，最多 50 条
客户端发送消息频率	每个客户端 5 条/秒[5]	每个客户端 5 条/秒[5]	每个客户端 5 条/秒[5]	每个客户端 5 条/秒[5]
服务端发送消息频率	6000 条/分钟[6]	20 条/秒[6]	100 条/秒[6]	100 条/秒[6]
扩展消息	支持	支持	支持	不支持
修改消息	不支持	不支持	支持	不支持
消息可靠度	100% 可靠	100% 可靠	100% 可靠	超出服务端消费上限的消息将被主动抛弃[7]
消息本地存储	移动端、PC 端支持	移动端、PC 端支持	移动端、PC 端支持	不支持
消息云端存储	需开通，可存储 6 - 36 个月[8]	需开通，可存储 6 - 36 个月[8]	默认存储 7 天，提供 3 - 36 个月存储服务[8]	需开通，可存储 2 - 36 个月[8]
离线缓存消息	默认 7 天离线消息缓存	默认 7 天离线消息缓存	不支持	不支持
消息本地搜索	支持	支持	支持	不支持
离线推送通知	支持	支持	支持，可调整推送频率	不支持

脚注：

1. 超级群业务仅限 IM 尊享版 使用。
2. IMKit 已支持 Android/iOS/Web 端。
3. 聊天室具有自动销毁机制。默认情况下，如果聊天室在指定时间内（默认 1 个小时）没有人说话，且没有人加入聊天室时，会把聊天室内所有成员踢出聊天室并销毁聊天室。您可以灵活调整聊天室的存活条件与存活时间。
4. 可允许单个用户加入多个聊天室，参考知识库文档：开通单个用户加入多个聊天室 。
5. 客户端不区分业务类型整体限制 5 条消息/秒，可付费上调。
6. 此处为服务端 API 默认频率，可付费上调。详细限频信息参见 API 接口列表 。
7. 聊天室消息量较大时，超出服务端消费上限的消息将被主动抛弃。您可通过用户白名单、消息白名单、自定义消息级别等服务，改变消息抛弃策略。如果用户在聊天室的用户白名单内，该用户所发送的消息在消息量大时也不会被抛弃。如需了解服务端消费上限与如何改变消息抛弃策略，可参见服务端文档消息优先级服务 、聊天室白名单服务 。
8. 参考知识库文档：单聊、群聊、聊天室、超级群在融云端历史消息存储时间分别是多长？ 。

前往融云产品文档 · 即时通讯 »

高级与扩展功能

IM 服务支持的高级与扩展功能，包括但不限于以下项目：

- **用户管理**：例如用户封禁、用户黑名单（拉黑）、用户白名单，群组及聊天室禁言、聊天室成员封禁等。
- **在线状态订阅**：将用户每一个终端在线、离线或登出后的状态，同步给应用开发者指定的服务器地址。
- **多设备在线消息同步**：同时支持桌面端、移动端、以及多个 Web 端之间的消息在线同步。
- **全量消息路由**：支持将单聊、群组、聊天室、超级群等的消息数据同步到应用开发者指定的服务器地址。
- **内容审核**：支持设置敏感词列表，过滤或替换消息中的敏感词。利用消息回调服务，可将消息先转发到应用开发者指定的服务器地址，由应用服务器判定是否可发送给目标接收者。
- **推送服务**：融云负责对接厂商推送平台，已覆盖小米、华为、荣耀、OPPO（适用于一加、realme）、vivo、魅族、FCM、APNs 手机系统级推送通道。支持标签推送、多种推送场景、推送统计、全量用户通知等特性。

部分功能需要在控制台开通服务后方可使用。部分为收费增值服务，详见即时通讯计费细则。

客户端 SDK

融云即时通讯（IM）客户端 SDK 提供丰富的组件与接口，大部分能力支持开箱即用。配合 IM 服务端 API 接口，可满足丰富的业务特性要求。

在集成融云 SDK 之前，我们建议使用快速上手教程与示例项目进行评估。

如何选择 SDK

IMLib 与 IMKit 是融云 IM 服务提供的两款经典的客户端 SDK。客户端功能在不同平台间基本保持一致。

- **IMLib** 是即时通讯能力库，封装了通信能力和会话、消息等对象。不含任何 UI 界面组件。

  IMLib 已支持绝大部分主流平台及框架，如 Android、iOS、Web、Flutter、React Native、Unity、微信小程序等。

- **IMKit** 是即时通讯界面库，集成了会话界面，并且提供了丰富的自定义功能。

  IMKit 已支持 Android、iOS 与 Web（要求 Web 5.X 版本）。

您可以根据业务需求进行选择：

- 基于 IMLib 开发应用，将融云即时通讯能力嵌入应用中，并自行开发产品的 UI 界面。
- 基于 IMKit 开发应用，将 IMKit 提供的界面组件直接集成到产品中，自定义界面组件功能，节省开发时间。您还可以使用融云提供的独立功能插件扩展 IMKit 的功能。

前往融云产品文档 · 客户端 SDK 体系 · IMLib · IMKit »

平台兼容性

IM 客户端 SDK 支持主流移动操作平台，客户端功能在多端基本保持一致，支持多平台互通。以下数据基于 5.X 版本 SDK。

平台/框架	接口语种	支持架构	说明
Android	Java	armeabi-v7a、arm64-v8a、x86、x86-64	系统版本 4.4 及以上
iOS	Objective-C	真机：arm64、armv7。模拟器： arm64（5.4.7+）、x86_64	系统版本 9.0 及以上
Web	Javascript	---	---
Electron	Javascript	详见下方 **Electron 版本与架构支持**	Electron 11.1.x、14.0.0、16.0.x、20.0.x
Flutter	dart	---	Flutter 2.0.0 及以上
React Native	Typescript	-	react-native 0.60 及以上
uni-app	Javascript	---	uni-app 2.8.1 及以上
Unity	C#	armeabi-v7a、arm64-v8a	---

- **Electron 版本与架构支持**：

  Electron 框架需要通过 Web 端 SDK 的 Electron 模块支持（详见 Electron 集成方案），适用于开发运行在 Windows、Linux、MacOS 平台的桌面版即时通讯应用。下表列出了目前已支持的 Electron 版本、桌面操作系统版本及 CPU 架构：

Electron 版本	平台	支持架构	备注
Electron 11.1.x	Windows	ia32 (x86)	`win32-ia32`
Electron 11.1.x	Linux	x64	`linux-x64`
Electron 11.1.x	Linux	arm64	`linux-arm64`
Electron 11.1.x	Mac	x64	`darwin-x64`
Electron 14.0.0	Windows	ia32 (x86)	`win32-ia32`
Electron 14.0.0	Mac	x64	`darwin-x64`

Electron 版本	平台	支持架构	备注
Electron 16.0.x	Windows	ia32 (x86)	`win32-ia32`
Electron 16.0.x	Mac	x64	`darwin-x64`
Electron 20.0.x	Windows	ia32 (x86)	`win32-ia32`
Electron 20.0.x	Mac	x64	`darwin-x64`
Electron 20.0.x	Mac	arm64	`darwin-arm64`

版本支持

IM 客户端 SDK 针对各平台/框架提供的最新版本如下（--- 表示暂未支持）：

SDK/平台	Android	iOS	Web	Electron	Flutter	React Native	Unity	uni-app	小程序
IMLib	5.6.x	5.6.x	5.9.x	5.9.x	5.4.x	5.2.x	5.1.x	5.4.x	5.9.x
IMKit	5.6.x	5.6.x	5.9.x	---	---	---	---	---	---
Global IM UIKit	1.0.x	1.0.x	1.0.x	1.0.x	---	---	---	---	---

集成 SDK 后包体积增量

- 集成 IM SDK 对应用安装包体积大小的增量（Android）
- 集成 IM SDK 对应用安装包体积大小的增量（iOS）

即时通讯服务端

即时通讯服务端提供一套 API 接口与多种语言的开源 SDK。

服务端 API

您可以使用服务端 API 将融云服务集成到您的即时通讯服务体系中，构建您即时通讯 App 的后台服务系统。例如，向融云获取用户身份令牌 (Token)，从 App 产品服务端向用户发送/撤回消息，或管理禁言用户列表。

> 前往融云即时通讯服务端 API 文档 · 集成必读 »

服务端 SDK

融云提供提供多个语言版本的开源服务端 SDK：

- server-sdk-java (GitHub) · (Gitee)
- server-sdk-php (GitHub) · (Gitee)
- server-sdk-go (GitHub) · (Gitee)

控制台

使用控制台，您可以对开发者账户和应用进行管理，开通高级服务，查看应用数据报表，和计费数据。

> 部分 IM 功能必须开通服务后方可使用。详见控制台服务管理页面。

即时通讯数据

如需在融云服务端长期存储单聊会话、群聊会话、聊天室会话的历史消息，您可以开通消息云存储服务。默认的长期存储时长与业务类型相关，可按需调整。该服务存储的数据仅供客户端获取历史消息时使用。

如果需要获取全部用户的消息历史，请开通 Server API 历史消息日志下载。开通后可使用服务端 API 获取最多三天的消息日志。

除此之外，您还可以开通全量消息路由服务，实时将消息同步到您的业务服务器。

您可以前往控制台的数据统计页面，查看即时通讯用户统计、业务统计、消息统计、业务健康检查等数据。开通相应服务后，还能获取如业务数据分析等数据。

融云不会利用客户的数据。同时融云提供完善的数据隐私保护策略。参见 SDK 隐私政策。

快速上手 快速上手

更新时间:2024-08-30

本教程是为了让新手快速了解融云即时通讯界面库（IMKit）。在本教程中，您可以体验集成 IMKit SDK 的基本流程和 IMKit 提供的 UI 界面。

前置条件

- 注册开发者账号。注册成功后，控制台会默认自动创建您的首个应用，默认生成开发环境下的 App Key，使用国内数据中心。
- 获取开发环境的应用 App Key。如不使用默认应用，请参考 如何创建应用，并获取对应环境 App Key 和 App Secret。

> 提示
>
> 每个应用具有两个不同的 App Key，分别对应开发环境与生产环境，两个环境之间数据隔离。在您的应用正式上线前，可切换到使用生产环境的 App Key，以便上线前进行测试和最终发布。

环境要求

- （SDK ≧ 5.6.3）使用 Android 5.0（API 21）或更高版本
- （SDK < 5.6.3）使用 Android 4.4（API 19）或更高版本

开始集成

IMKit 支持通过 Maven、本地 Module 、源码三种方式的方式集成。请提前在融云官网 SDK 下载页面或融云的 Maven 仓库查询最新版本。安装 IMKit 将同时集成即时通讯能力库 IMLib。其他插件可按需集成。

导入 SDK

本教程以在 Gradle 中添加远程依赖项为例，将 IMKit SDK 导入到您的应用工程中。请注意使用 融云的 Maven 仓库。

1. 打开根目录下的 `build.gradle`（**Project** 视图下），声明融云的 Maven 代码库。

```
allprojects {
repositories {
...
//融云 maven 仓库地址
maven {url "https://maven.rongcloud.cn/repository/maven-releases/"}
}
}
```

2. 在应用的 `build.gradle` 中，添加融云即时通讯界面库（IMKit）为远程依赖项。

```
dependencies {
...
//此处以集成 IMKit 库为例，您可以按需集成插件
api 'cn.rongcloud.sdk:im_kit:x.y.z'
}
```

> 提示
>
> 各个 SDK 的最新版本号可能不相同，还可能是 x.y.z.h，可前往 融云官网 SDK 下载页面 或 融云的 Maven 代码库 查询。

其他导入方式可参考导入 SDK。

使用 App Key 初始化

融云即时通讯客户端 SDK 核心类为 RongIM 和 IMCenter。在 Application 的 `onCreate()` 方法中，调用初始化方法，传入生产或开发环境的 App Key。

如果 SDK 版本 ≧ 5.4.2，请使用以下初始化方法。

```
String appKey = "YourAppKey"; // example: bos9p5rlcm2ba
InitOption initOption = new InitOption.Builder().build();

IMCenter.init(this, appKey, initOption);
```

初始化配置（`InitOption`）中封装了区域码（AreaCode），导航服务地址（`naviServer`）、文件服务地址（`fileServer`）、数据统计服务地址（`statisticServer`）配置，以及是否开启推送的开关（`enablePush`）和主进程开关（`isMainProcess`）。不传入任何配置表示全部使用默认配置。SDK 默认连接北京数据中心。

如果 App Key 不属于中国（北京）数据中心，则必须传入有效的初始化配置。初始化详细说明参见初始化。

获取用户 Token

用户 Token 是与用户 ID 对应的身份验证令牌，是应用程序的用户在融云的唯一身份标识。应用客户端在使用融云即时通讯功能前必须与融云建立 IM 连接，连接时必须传入 Token。

在实际业务运行过程中，应用客户端需要通过应用的服务端调用 IM Server API 申请取得 Token。详见 Server API 文档 注册用户 。

在本教程中，为了快速体验和测试 SDK，我们将使用控制台「北极星」开发者工具箱，从 API 调试页面调用 获取 Token 接口，获取到 **userId** 为 1 的用户的 Token。提交后，可在返回正文中取得 Token 字符串。

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"code":200,"userId":"1","token":"gxld6GHx3t1eDxof1qtxxYrQcjkbhl1V@sgyu.cn.example.com;sgyu.cn.example.com"}
```

建立 IM 连接

1. 监听 IM 连接状态的变化。建议在应用生命周期内设置。为了避免内存泄露，请在不需要监听时，将设置的监听器移除。详见连接状态监听。

```
private RongIMClient.ConnectionStatusListener connectionStatusListener = new RongIMClient.ConnectionStatusListener() {
@Override
public void onChanged(ConnectionStatus status) {
//开发者需要根据连接状态码，进行不同业务处理
}
};
public void setIMStatusListener() {
RongIM.setConnectionStatusListener(connectionStatusListener);
}
```

2. 在自定义登录页面，使用上一步获取的 token, 连接融云，即模拟 userId 为 1 的用户连接到融云服务器。

```
public class LoginActivity extends AppCompatActivity {
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_login);
TextView login = findViewById(R.id.login);
login.setOnClickListener(v -> {
String token = "后台获取的 token";
RongIM.connect(token, new RongIMClient.ConnectCallback() {
@Override
public void onSuccess(String userId) {
// 登录成功，跳转到默认会话列表页。
RouteUtils.routeToConversationListActivity(LoginActivity.this, "");
}

@Override
public void onError(RongIMClient.ConnectionErrorCode connectionErrorCode) {

}

@Override
public void onDatabaseOpened(RongIMClient.DatabaseOpenStatus databaseOpenStatus) {

}
});
});
}
}
```

SDK 已实现自动重连机制，请参见连接。

收发消息

对融云来说，只要提供对方的 userId，融云就可支持跟对方发起聊天。例如，A 需要 发送消息给 B，只需要将 B 的 userId 告知融云服务即可发送消息。

> 提示
>
> - 融云服务器提供消息发送能力，消息发送过程中默认不会做任何权限校验。
> - 好友关系由开发者的应用服务器自行维护。

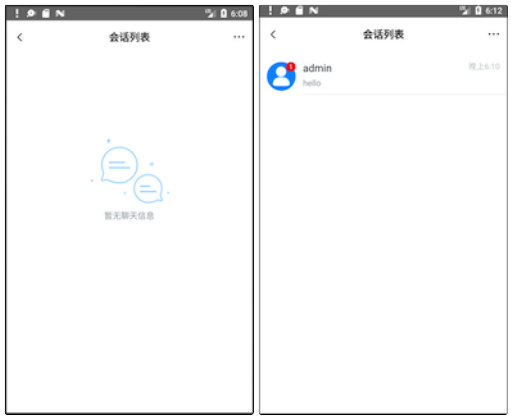
以下我们将利用 IMKit 默认提供的 UI 页面，进行简单的消息收发体验。

UI 界面

IMKit SDK 已默认提供会话列表页面和会话页面。客户端用户在会话列表页面可查看到当前所有的聊天会话，在会话页面可进行消息查看、回复、发送等活动。

首次登录成功后，即跳转到默认会话列表界面，一般会显示一个空会话列表。客户端接收到消息后，会自动在会话列表页面展示新会话。

下图展示了 IMKit SDK 默认提供的会话列表页面。以下直接以默认会话列表为例。



点击会话列表中的会话，将进入会话页面。在会话页面可发送消息。

IMKit 默认的会话列表页面与会话页面基于 Activity。IMKit 还支持以 Fragment 方式将两种页面集成到您应用自定义的 Activity 中。UI 支持自定义。具体请参见集成会话列表与集成会话界面。

如需显示昵称及头像信息，您需要设置一个用户信息提供者给 IMKit。 IMKit 通过用户信息提供者获取需要显示的用户资料数据。详情请参见用户信息。

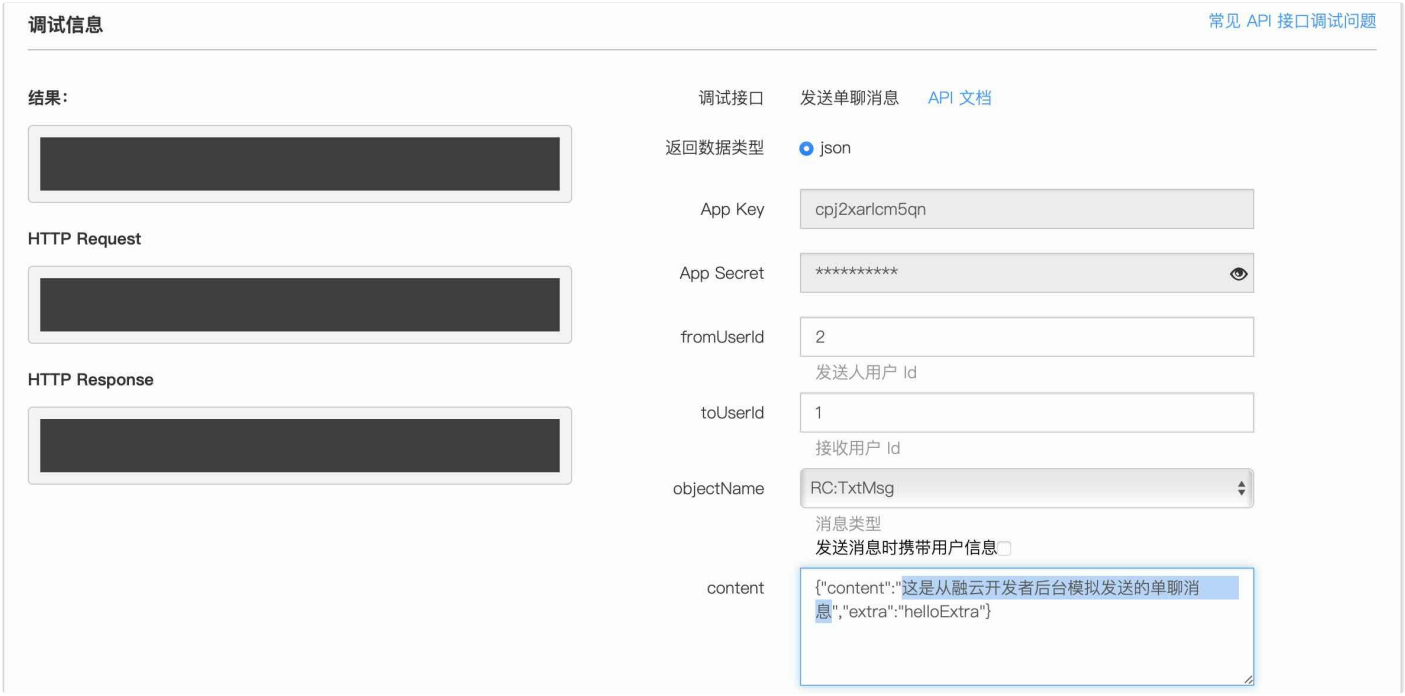
测试收发消息

在实际业务运行过程中，应用客户端可以通过用户 ID、群聊会话 ID、或聊天室 ID 等接收消息。

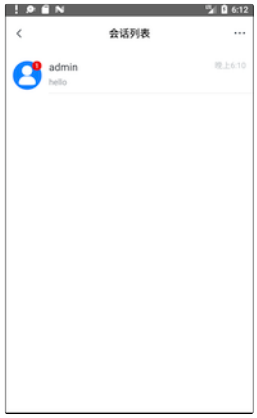
在本教程中，为了快速体验和测试 SDK，我们从控制台「北极星」开发者工具箱 IM Server API 调试 页面向当前登录的用户发送一条文本消息，模拟单聊会话。

1. 访问控制台「北极星」开发者工具箱的 IM Server API 调试 页面。
2. 在**消息**标签下，找到 **消息服务 > 发送单聊消息** 接口。

   以下模拟了从 UserId 为 2 的用户向 UserId 为 1 的用户发送一条文本消息。



3. 客户端接收到消息后，自动在会话列表页面展示新的单聊会话。



4. 点击会话，即可进入消息列表页面，发送消息。

后续步骤

以上步骤即 IMKit SDK 的快速集成与新手体验流程，您体验了基础 IM 通信能力和 UI 界面，更多详细介绍请参考后续各章节详细说明。

导入 SDK 导入 SDK

更新时间:2024-08-30

利用 Android Studio 中的 Gradle 构建系统，您可以轻松地将融云即时通讯界面库（IMKit）作为依赖项添加到您的构建中。

融云支持使用 Gradle 添加远程依赖项、导入本地库模块（Module）和导入源码三种方式，将 IMKit SDK 导入到您的应用工程中。

环境要求

- （SDK ≧ 5.6.3）使用 Android 5.0（API 21）或更高版本
- （SDK < 5.6.3）使用 Android 4.4（API 19）或更高版本

检查版本

在导入 SDK 前，您可以前往融云官网 SDK 下载页面 确认当前最新版本号。

Gradle

使用 Gradle，添加融云即时通讯界面库（IMKit）为远程依赖项。Android Studio 的配置在 Gradle 插件 7.0 以下版本、7.0 版本、和 7.1 及以上版本有所不同。请根据您当前的 Gradle 插件版本进行配置。本文以使用 Gradle 插件 7.0 以下版本为例。

> 由于 Jcenter 于2021年 5 月 4 日 停止提供远程仓库服务，远程仓库统一由 JCenter 迁移到新的融云私有仓库。
>
> 不再支持该地址集成：https://dl.bintray.com/rongcloud/maven。

1. 声明融云的 Maven 代码库，以使用 Gradle 插件 7.0 以下版本为例。打开根目录下的 `build.gradle`（**Project** 视图下）：

```
allprojects {
repositories {
...
//融云 maven 仓库地址
maven {url "https://maven.rongcloud.cn/repository/maven-releases/"}
}
}
```

2. 在应用的 `build.gradle` 中，添加融云即时通讯界面库（IMKit）为远程依赖项。

```
dependencies {
...
//此处以集成 IMKit 库为例，您可以按需集成插件
api 'cn.rongcloud.sdk:im_kit:x.y.z'
}
```

> 提示
>
> 各个 SDK 的最新版本号可能不相同，还可能是 x.y.z.h，可前往 融云官网 SDK 下载页面 或 融云的 Maven 代码库 查询。

Android 本地库模块 (Module)

在导入 SDK 前，您需要前往融云官网 SDK 下载页面，将 **即时通讯界面库 IMKit** 下载到本地。

1. 在 Android Studio 中打开工程后，依次点击 **File** > **New** > **Import Module**，找到下载的 Module 组件并导入。

2. 如果导入的内容中包含有插件的 aar 包，请移至 `app/libs` 目录下。

3. 打开根目录下的 `settings.gradle`（**Project** 视图下），添加 IMLib 本地库模块。

```
include ':IMKit'
include ':IMLib'
...
```

4. 在应用的 `build.gradle` 中，添加 IMLib 为本地库模块依赖项。

```
dependencies {
...
api project(':IMKit')
...
}
```

5. (可选) 以 Android 本地库模块导入 SDK 时默认不带 Javadoc。建议自行从融云的 Maven 代码库下载 Javadoc 并导入，以便于在 Android Studio 中即时查看。

   如需指导，请参见以下知识库链接：

   https://help.rongcloud.cn/t/topic/727

源代码方式

在导入 SDK 前，请先下载**融云开源工程**（GitHub · Gitee）到本地。

融云开源工程地址中 `imkit` 文件夹下的代码即为 IMKit SDK 的源码。

1. 拷贝 `imkit` 模块到您的工程。

2. 打开根目录下的 `settings.gradle`（**Project** 视图下），添加 IMKit 本地库模块。

```
include ':imkit'
```

3. 声明融云的 Maven 代码库，以使用 Gradle 插件 7.0 以下版本为例。打开根目录下的 `build.gradle`（**Project** 视图下）：

```
allprojects {
repositories {
...
//融云 maven 仓库地址
maven {url "https://maven.rongcloud.cn/repository/maven-releases/"}
}
}
```

4. 在应用的 `build.gradle` 中，添加融云即时通讯界面库（IMKit）为依赖项。

```
api project(':imkit')
```

5. (可选) 以源代码导入 SDK 时默认不带 Javadoc。建议自行从融云的 Maven 代码库下载 Javadoc 并导入，以便于在 Android Studio 中即时查看。

   如需指导，请参见以下知识库链接：

   https://help.rongcloud.cn/t/topic/727

初始化 初始化

更新时间:2024-08-30

在使用 SDK 其它功能前，必须先进行初始化。本文中将详细说明 IMKit SDK 初始化的方法。

首次使用融云的用户，我们建议先阅读 IMKit SDK 快速上手，以完成开发者账号注册等工作。

推送

推送是常见的基础功能。IMKit SDK 已集成融云自有推送，以及多家第三方推送，且会在 SDK 初始化后触发。因此，客户端的推送配置必须在初始化之前提供。详细说明请参见启用推送。

海外数据中心

- 如果您使用海外数据中心，且使用 5.4.2 及更新版本的开发版（dev）SDK，请注意在初始化配置中传入正确的区域码（AreaCode）。
- 如果您使用海外数据中心，且使用稳定版（stable） SDK，或使用早于 5.4.2 版本的开发版（dev）SDK，必须在初始化之前修改 IMLib SDK 连接的服务地址为海外数据中心地址。否则 SDK 默认连接中国国内数据中心服务地址。详细说明请参见配置海外数据中心服务地址。

进程

> 提示
>
> 从 5.3.0 版本开始，IMLib SDK 支持单进程。

SDK 可支持多进程和单进程机制。

如果 SDK 版本 < 5.3.3 或 ≥ 5.3.5，默认采用多进程机制，初始化之后，应用会启动以下进程：

1. 应用的主进程
2. `<应用包名>:ipc`。此进程是 IM 通信的核心进程，和主进程任务相互隔离。
3. `io.rong.push`：融云默认推送进程。该进程是否启动由推送通道的启用策略决定。详细说明可参考启用推送。

如果 SDK 版本 ≥ 5.3.3 且 < 5.3.5，默认采用单进程机制。

从 5.3.0 版本开始，可以在初始化前通过 RongCoreClient 的 `enableSingleProcess` 接口开启或关闭单进程。请注意，开启单进程后融云会占用主进程的部分内存。

```
//开启单进程
RongCoreClient.getInstance().enableSingleProcess(true);
```

初始化 SDK

> 提示
>
> 以下初始化方法要求 SDK 版本 ≥ 5.4.2。您可前往 融云官网 SDK 下载页面 查询最新开发版（Dev）和稳定版（Stable）的版本号。

请在 Application 的 `onCreate()` 方法中初始化 SDK，传入生产或开发环境的 App Key。

```
String appKey = "YourAppKey"; // example: bos9p5rlcm2ba
InitOption initOption = new InitOption.Builder().build();

IMCenter.init(this, appKey, initOption);
```

初始化配置（InitOption）中封装了区域码（AreaCode）配置。SDK 将通过区域码获取有效的导航服务地址、文件服务地址、数据统计服务地址、和日志服务地址等配置。

- 如果 App Key 属于中国（北京）数据中心，您无需传入任何配置，SDK 会使用默认配置。
- 如果 App Key 属于海外数据中心，则必须传入有效的区域码（AreaCode）配置。请务必在控制台核验当前 App Key 所属海外数据中心后，找到 AreaCode 中对应的枚举值进行配置。

例如，使用新加坡数据中心的应用的生产或开发环境的 App Key：

```
String appKey = "Singapore_dev_AppKey";
AreaCode areaCode = AreaCode.SG;

InitOption initOption = new InitOption.Builder()
.setAreaCode(areaCode)
.build();

IMCenter.init(context, appKey, initOption);
```

除区域码外，初始化配置（InitOption）中还封装了以下配置：

- 是否开启推送的开关（`enablePush`）：是否整体禁用推送。

- 主进程开关（`isMainProcess`）：是否为主进程。默认情况下由 SDK 判断进程。
- 导航服务地址（`naviServer`）：一般情况下不建议单独配置。SDK 内部默认使用与区域码对应的地址。
- 文件服务地址（`fileServer`）：仅限私有云使用。
- 数据统计服务地址（`statisticServer`）：一般情况下不建议单独配置。SDK 内部默认使用与区域码对应的地址。

```
String appKey = "Your_AppKey";
AreaCode areaCode = AreaCode.BJ;

InitOption initOption = new InitOption.Builder()
.setAreaCode(areaCode)
.enablePush(true)
.setFileServer("http(s)://fileServer")
.setNaviServer("http(s)://naviServer")
.setStatisticServer("http(s)://StatisticServer")
.build();

IMCenter.init(context, appKey, initOption);
```

初始化 SDK（旧版）

> 提示
>
> 如果您使用的开发版 SDK 版本号小于 5.4.2，或者稳定版 SDK 版本号小于等于 5.3.8，只能使用以下方式初始化。建议您及时升级 SDK 到最新版本。

请在 Application 的 `onCreate()` 方法中初始化 SDK，传入生产或开发环境的 App Key。您可以使用 RongCoreClient 或 `RongIMClient` 的初始化方法。

```
public class App extends Application {
@Override
public void onCreate() {
...
String appKey = "YourAppKey";
Boolean enablePush = true;
RongIM.init(this, appKey, enablePush);
}
}
```

参数	类型	说明
context	Context	应用上下文。
appKey	String	应用的开发环境或生产环境 AppKey，请勿混淆。如果您选择在 `AndroidManifest.xml` 设置 appKey，此处可不传。
enablePush	boolean	是否启用推送。默认 `true`，即开启推送功能。

仅为保持向后兼容，IMLib SDK 5x 仍支持在 `AndroidManifest.xml` 中配置 App Key。但考虑到 App Key 安全性，融云已不再建议使用这种配置方式。

您可以在用户接受隐私协议后，再进行初始化。

```
/**
* 应用启动时，判断用户是否已接受隐私协议，如果已接受，正常初始化；否则跳转到隐私授权页面请求用户授权。
*/
public class App extends Application {
@Override
public void onCreate() {
super.onCreate();

//伪代码，从 sp 里读取用户是否已接受隐私协议
boolean isPrivacyAccepted = getPrivacyStateFromSp();
//用户已接受隐私协议，进行初始化
if (isPrivacyAccepted) {
String appKey = "控制台创建的应用的 AppKey";
//第一个参数必须传应用上下文
RongIM.init(this.getApplicationContext(), appKey);
} else {
//用户未接受隐私协议，跳转到隐私授权页面。
goToPrivacyActivity();
}
...
}
}

/**
* 该类为隐私授权页面，示范如何在用户接受隐私协议后进行 IM 初始化。
*/
public class PrivacyActivity extends Activity implements View.OnClickListener {
...
@Override
public void onClick(View v) {
switch (v.getId()) {
case R.id.accept_privacy:
//伪代码，保存到 sp
savePrivacyStateToSp();

String appKey = "控制台创建的应用的 AppKey";
//第一个参数必须传应用上下文
RongIM.init(this.getApplicationContext(), appKey);
break;
default:
...
}
}
}
```

配置指南 配置指南

更新时间:2024-08-30

IMKit 全局配置旨在提供易于使用的功能配置，帮助您快速构建聊天应用程序。

配置说明

IMKit 全局配置按照以下模块进行划分。您可以在相应的文件中了解 IMKit 提供的所有全局配置。

类别	描述	API 文档	源码
特性配置	控制是否启用消息引用功能、是否禁用 emoji 表情输入、本地通知的铃声与震动提示等。	FeatureConfig	FeatureConfig.java
会话列表页面配置	控制会话列表每个会话的头像、每页拉取的会话条数、会话列表延迟刷新时间等。	ConversationListConfig	ConversationListConfig.java
会话页面配置	控制是否启用已读回执、消息撤回、消息重发、合并转发、进入会话页面默认拉取的历史消息数量等行为以及是否展示未读消息气泡、是否展示 @消息数提示等 UI 配置。	ConversationConfig	ConversationConfig.java
聚合会话配置	控制由多个会话聚合而成的聚合会话的标题、头像等。	GatheredConversationConfig	GatheredConversationConfig.java
通知配置	控制前台非会话页面接收到消息是否静默、通知的标题类型本地通知的分类（Category）等。	NotificationConfig	NotificationConfig.java

修改 IMKit 配置

IMKit 提供了 RongConfigCenter 类，作为修改 SDK 全局配置的入口。

`RongConfigCenter` 类初始化时会先从 rc_config.xml 文件中读取默认配置。您可以通过在应用程序目录下新建 `res/values/rc_config.xml` 覆盖默认配置，也可以通过 `RongConfigCenter` 类的方法动态修改 IMKit 配置。

配置示例：

```
// 禁用引用消息功能
RongConfigCenter.featureConfig().enableReference(false);

// 设置会话列表延时刷新时间（防止消息量过大导致卡顿）
RongConfigCenter.conversationListConfig().setDelayRefreshTime(100);

// 会话列表中每个条目的头像显示默认为矩形，可修改为圆角显示。
RongConfigCenter.featureConfig().setKitImageEngine(new GlideKitImageEngine() {
@Override
public void loadConversationListPortrait(@NonNull Context context, @NonNull String url, @NonNull ImageView imageView, Conversation conversation) {
Glide.with(context).load(url)
.apply(RequestOptions.bitmapTransform(new CircleCrop()))
.into(imageView);
}
});

// 会话页面中每条消息的头像显示默认为矩形，可修改为圆角显示。
RongConfigCenter.featureConfig().setKitImageEngine(new GlideKitImageEngine() {
@Override
public void loadConversationPortrait(@NonNull Context context, @NonNull String url, @NonNull ImageView imageView, Message message) {
Glide.with(context).load(url)
.apply(RequestOptions.bitmapTransform(new CircleCrop()))
.into(imageView);
}
});


// 修改 `needDeleteRemoteMessage` 属性为 `true` 后，IMKit 会同时删除本地与远端消息。(Since 5.2.3)
RongConfigCenter.conversationConfig().setNeedDeleteRemoteMessage(true);

// 设置前台处于非会话页面时，接收到消息后不通知。
RongConfigCenter.notificationConfig().setForegroundOtherPageAction(ForegroundOtherPageAction.Silent);
```

检查 IMKit 配置

IMKit 配置是实时应用的，修改后的配置将在下一次 UI 刷新或者操作时生效。建议在初始化 IMKit 后完成所有配置。

例外情况：快捷回复功能要求在初始化之前启用（`RongConfigCenter.featureConfig().enableQuickReply()`），否则该功能无法生效。

监听连接状态　　监听连接状态

更新时间:2024-08-30

客户端 SDK 为 App 提供了 IM 连接状态监听器 ConnectionStatusListener。通过监听 IM 连接状态的变化，App 可以进行不同业务处理，或在页面上给出提示。

连接状态监听器说明

ConnectionStatusListener 接口定义如下：

```
public interface ConnectionStatusListener {
void onChanged(ConnectionStatus status);
}
```

当连接状态发生变化时，SDK 会通过 onChanged() 方法，将当前的连接状态回调给开发者。各状态的具体说明请参考下表。

状态名称	状态值	说明
NETWORK_UNAVAILABLE	-1	网络不可用
CONNECTED	0	连接成功
CONNECTING	1	连接中
UNCONNECTED	2	未连接状态，即应用没有调用过连接方法
KICKED_OFFLINE_BY_OTHER_CLIENT	3	用户账号在其它设备登录，此设备被踢下线
TOKEN_INCORRECT	4	Token 过期时触发此状态
CONN_USER_BLOCKED	6	用户被控制台封禁
SIGN_OUT	12	用户主动断开连接的状态，见断开连接
SUSPEND	13	连接暂时挂起（多是由于网络问题导致），SDK 会在合适时机进行自动重连
TIMEOUT	14	连接超时，SDK 将停止连接，用户需要做超时处理，再自行调用连接接口进行连接

设置连接状态监听器

建议在应用生命周期内设置。为了避免内存泄露，请在不需要监听时，将设置的监听器移除。

```
private RongIMClient.ConnectionStatusListener connectionStatusListener = new RongIMClient.ConnectionStatusListener() {
@Override
public void onChanged(ConnectionStatus status) {
//开发者需要根据连接状态码，进行不同业务处理
}
};
public void setIMStatusListener() {
RongIM.setConnectionStatusListener(connectionStatusListener);
}
```

连接 连接

更新时间:2024-08-30

应用客户端成功连接到融云服务器后，才能使用融云即时通讯 SDK 的收发消息功能。

融云服务端在收到客户端发起的连接请求后，会根据连接请求里携带的用户身份验证令牌（Token 参数），判断是否允许用户连接。

前置条件

- 通过服务端 API 注册用户（获取 Token）。融云客户端 SDK 不提供获取 Token 方法。应用程序可以调用自身服务端，从融云服务端获取 Token。
  - 取得 Token 后，客户端可以按需保存 Token，供后续连接时使用。具体保存位置取决于应用程序客户端设计。如果 Token 未失效，就不必再向融云请求 Token。
  - Token 有效期可在控制台进行配置，默认为永久有效。即使重新生成了一个新 Token，未过期的旧 Token 仍然有效。Token 失效后，需要重新获取 Token。如有需要，可以主动调用服务端 API 作废 Token。
- 建议应用程序在连接之前设置连接状态监听。
- SDK 已完成初始化。

> 提示
>
> **请不要在客户端直接调用服务端 API 获取 Token。获取 Token 需要提供应用的 App Key 和 App Secret。客户端如果保存这些凭证，一旦被反编译，会导致应用的 App Key 和 App Secret 泄露。所以，请务必确保在应用服务端获取 Token。**

连接聊天服务器

请根据应用的业务需求与设计，自行决定合适的时机（登陆、注册、或其他时机以免无法进入应用主页），向融云聊天服务器发起连接请求。

请注意以下几点：

- 必须在 SDK 初始化之后，调用 `connect()` 方法进行连接。否则可能没有回调。
- SDK 本身有重连机制，在一个应用生命周期内不须多次调用 `connect()` 。否则可能触发多个回调，触发导致回调被清除。
- 在应用主进程内调用一次即可。

接口（可设置超时）

客户端用户首次连接聊天服务器时，建议调用带连接超时时间（timeLimit）的接口，并设置超时秒数。在网络较差等导致连接超时的情况下，您可以利用接口的超时错误回调，并在 UI 上提醒用户，例如建议客户端用户等待网络正常的时候再次连接。

一旦连接成功，SDK 的重连机制将立即开始生效，并接管所有的重连处理。当因为网络原因断线时，SDK 会不停重连直到连接成功为止，不需要您做额外的连接操作。应用主动断开连接，将退出重连机制，详见断开连接。其他情况请参见重连机制与重连互踢。

如果后续离线登录，建议将 `timeLimit` 参数设置为 0，可以在回调数据库打开（onDatabaseOpened）就进行页面跳转，优先展示本地历史数据。连接逻辑则完全托管给 SDK。

接口原型

```
RongIM.connect(token, timeLimit, connectCallback);
```

参数说明

参数	类型	说明
token	String	从服务端获取的 Token。
timeLimit	int	超时时间（秒）。超时后不再重连。取值 <=0 则将一直连接，直到连接成功或者发生业务错误。
connectCallback	ConnectCallback	连接回调。详见下文连接回调方法说明。

- `timeLimit` 参数说明：
  - `timeLimit` <= 0，IM 将一直连接，直到连接成功或者发生业务错误（如 token 非法）。
  - `timeLimit` > 0，IM 将最多连接 timeLimit 秒。如果在 timeLimit 秒内无法连接成功则不再进行重连，通过回调 onError 告知连接超时，您需要再自行调用 connect 接口。

示例代码

```
public void connectIM(String token){
int timeLimit = 0;
RongIM.connect("用户Token", timeLimit, new RongIMClient.ConnectCallback() {
@Override
public void onDatabaseOpened(RongIMClient.DatabaseOpenStatus code) {
if(RongIMClient.DatabaseOpenStatus.DATABASE_OPEN_SUCCESS.equals(code)) {
//本地数据库打开，跳转到会话列表页面
} else {
//数据库打开失败，可以弹出 toast 提示。
}
}

@Override
public void onSuccess(String s) {
//连接成功，如果 onDatabaseOpened() 时没有页面跳转，也可在此时进行跳转。
}

@Override
public void onError(RongIMClient.ConnectionErrorCode errorCode) {
if(errorCode.equals(RongIMClient.ConnectionErrorCode.RC_CONN_TOKEN_EXPIRE)) {
//从 APP 服务请求新 token，获取到新 token 后重新 connect()
} else if (errorCode.equals(RongIMClient.ConnectionErrorCode.RC_CONNECT_TIMEOUT)) {
//连接超时，弹出提示，可以引导用户等待网络正常的时候再次点击进行连接
} else {
//其它业务错误码，请根据相应的错误码作出对应处理。
}
}
})
}
```

接口（无超时设置）

如果客户端用户已成功登录过您的应用，且连接过融云聊天服务器，后续离线登录时建议使用此接口。

此时因用户已有历史数据，并不需要强依赖于连接成功。可以在回调数据库打开（onDatabaseOpened）就进行页面跳转，优先展示本地历史数据。连接逻辑完全托管给 SDK 即可。

> 提示
>
> 当网络较差时，有可能长时间不会回调。

调用此接口后，SDK 的重连机制将立即开始生效，并接管所有的重连处理。SDK 会不停重连直到连接成功为止，不需要您做额外的连接操作。应用主动断开连接，将退出重连机制，详见断开连接。其他情况请参见重连机制与重连互踢。

接口原型

```
RongIM.connect(token, connectCallback);
```

参数说明

参数	类型	说明
token	String	从服务端获取的用户身份令牌
connectCallback	ConnectCallback	连接回调。详见下文连接回调方法说明。

连接回调方法说明

连接回调 connectCallback 提供了以下三个回调方法：

- onDatabaseOpened(DatabaseOpenStatus code)

  本地数据库打开状态回调。当回调 DATABASE_OPEN_SUCCESS 时，说明本地数据库打开，此时可以拉取本地历史会话及消息，适用于离线登录场景。

- onSuccess(String userId)

  连接成功的回调，返回当前连接的用户 ID。

- onError(ConnectionErrorCode errorCode)

  连接失败并返回对应的连接错误码，开发者需要参考连接状态码进行不同业务处理。

  常见错误如下：

  - SDK 没有初始化即调用 connect()方法。
  - 应用客户端与应用服务器的 App Key 不一致，导致 TOKEN_INCORRECT 错误。

    融云的每个应用都提供用于隔离生产和开发环境的两套独立 App Key / Secret。由测试环境切换到生产环境时，很容易发生应用客户端与应用服务端 App Key 不一致的问题。
  - Token 已过期。参见获取 Token。

连接状态码

名称	值	说明
IPC_DISCONNECT	-2	IPC 进程意外终止。 可能原因： 1. 手机系统策略，导致 IPC 进程被回收或被解绑，应用层调用 IM 接口时会触发此问题，SDK 会做好自动重连，应用不需要额外处理。 2. 找不到对应 CPU 架构的 libRongIMLib.so 或 libsqlite.so，请确保集成了对应架构的 so
RC_CONN_ID_REJECT	31002	AppKey 错误，请检查您使用的 AppKey 是否正确
RC_CONN_TOKEN_INCORRECT	31004	Token 无效 请检查客户端初始化使用的 AppKey 和您服务器获取 token 时使用的 AppKey 是否一致.
RC_CONN_NOT_AUTHRORIZED	31005	App 校验未通过（开通了 App 校验功能，但是校验未通过）
RC_CONN_APP_BLOCKED_OR_DELETED	31008	应用被封禁或已删除。请检查您使用的 AppKey 是否被封禁或已删除。
RC_CONN_USER_BLOCKED	31009	连接失败，一般因为用户已被封禁。请检查您使用的 Token 是否正确，以及对应的 UserId 是否被封禁
RC_CONN_TOKEN_EXPIRE	31020	Token 过期 一般是因为在控制台设置了 token 过期时间，需要请求您的服务器重新获取 token 并再次用新的 token 建立连接.
RC_CLIENT_NOT_INIT	33001	SDK 没有初始化。在使用 SDK 任何功能之前，必须先初始化。
RC_CONNECTION_EXIST	34001	连接已存在，或正在重连中。
RC_CONNECT_TIMEOUT	34006	SDK 内部连接超时，调用可设置超时的连接接口，并设置有效的 timeLimit 值时会出现该错误 SDK 不会继续重连，需要 APP 手动调用 connect 接口进行连接
UNKNOWN	-1	未知错误

断开连接　　断开连接

更新时间:2024-08-30

在 App 需要执行切换用户登录、注销登录的操作时，需要断开与融云的 IM 连接。SDK 支持设置断开 IM 连接之后是否允许向用户发送消息推送通知。

> 提示
>
> SDK 在前后台切换或者网络出现异常都会自动重连，会保证连接的可靠性。除非 App 逻辑需要登出，否则不需要调用此方法进行手动断开。

断开连接（允许推送）

主动断开与融云服务端的 IM 连接，并设置断开连接后允许融云服务端进行远程推送。

```
RongIM.getInstance().disconnect()
```

如果融云服务端发现 App 客户端不在线（默认要求全部设备已下线），在接收新消息时，融云服务端会为该用户记录一条离线消息[?]，并触发融云服务端的推送服务。融云服务端会通过推送通道下发一条提醒到客户端 SDK。该提醒一般以通知形式展示在通知面板，提示用户有离线消息。

断开连接（不允许推送）

主动断开与融云服务端的 IM 连接，并设置断开连接后不允许融云服务端进行远程推送。

需要注销登录（登出）或切换 App 用户账号时，推荐使用以下方法：

```
RongIM.getInstance().logout()
```

断开连接且不允许推送的情况下，融云服务端仅记录离线消息，但不会为当前设备触发推送服务。如果用户登录了多个设备，则在其他设备中最后一个登录的设备上可正常接收推送。在多设备场景下，App 可能需要保证设备间消息记录一致，可通过开启**多设备消息同步**实现。详见多设备消息同步。

重连机制与重连互踢　　重连机制与重连互踢 自动重连机制

更新时间:2024-08-30

SDK 内已实现自动重连机制，一旦连接成功，SDK 的重连机制将立即开始生效，并接管所有的重连处理。当因为网络原因断线时，SDK 内部会尝试重新建立连接，不需要您做额外的连接操作。

可能导致 SDK 断线重连的异常情况如下：

- **弱网环境**：可能出现 SDK 不停重连的情况。因为客户端 SDK 和融云服务端之间存在连接保活机制，一旦因如果网络太差导致心跳超时，SDK 就会触发重连操作，尝试重连直到连接成功。
- **无网环境**：SDK 的重连机制会暂停。一旦网络恢复，SDK 会进行重连操作。

> 提示
>
> 一旦触发连接错误的回调，SDK 将退出重连机制。请根据具体的状态码自行处理。

重连时间间隔

SDK 尝试重连时，时间间隔逐次变大，分别是 0.05s（5.6.2 之前为 0s）, 0.25s, 0.5s, 1s, 2s, 4s, 8s, 16s, 32s。之后每 64s 重试一次。

当 APP 切换到前台或者网络状态发生变化，重连时间会按照上面的时间间隔从头开始，保证这种情况下能尽快的连接成功。

主动退出重连机制

应用主动断开连接后，SDK 将退出重连机制，不再尝试重连。

重连互踢策略

重连互踢策略用于控制 SDK 自动重连成功时是否需要下线的设备。

即时通讯业务默认仅允许同一用户账号在单台移动端设备上登录。后登录的移动端设备一旦连接成功，则自动踢出之前登录的设备。在部分情况下，SDK 的重连机制可能会导致后登录设备无法正常在线。

例如，默认的重连互踢策略可能导致以下情况：

1. 用户张三尝试在移动设备 A 上登录，但因 A 设备网络不稳定导致未连接成功，触发了 SDK 的自动重连机制。
2. 用户此时尝试换到移动设备 B 上登录。B 设备连接成功，且用户可通过 B 设备正常使用即时通讯业务。
3. A 设备网络稳定之后，SDK 重连成功。因此时 A 设备为后上线设备，导致 B 设备被踢出。

修改 App 用户的重连互踢策略

如果 App 用户希望在以上场景中将重连成功的 A 设备下线，同时保持 B 设备登录，可以通过修改自己的重连互踢策略实现。

> 提示
>
> IMKit 不直接提供对应 API。如有需要，请调用 IMLib SDK 的 `setReconnectKickEnable` 方法。详见 IMLib 文档 重连机制与重连互踢。

多端同时在线

更新时间:2024-08-30

多端同时在线是指同一用户账号从多个平台同时连接到融云即时通讯服务的功能。默认情况下，融云即支持多端设备同时在线。该功能无需开通即可使用。

> 默认多端设备之间不会进行消息同步。如有需要，请开通多设备消息同步服务。

多端登录限制说明

默认的情况下，同一用户账号可在移动端、Web 端、桌面端、小程序端最多一个设备上同时在线。

平台类别	限制	IM SDK 支持平台列表
移动端	默认仅支持一个移动端设备连接。如需支持移动端多设备登录，请提交工单申请开通移动多端服务。	Android、iOS、Flutter、React Native、uni-app、Unity
桌面端	默认仅支持一个桌面端设备连接。	Electron 框架（通过 Web 端 SDK 的 Electron 模块支持）
Web 端	默认仅支持一个 Web 页面连接（每个浏览器标签页认为是一个连接）。在控制台自助开通多设备消息同步服务后，自动支持多 Web 页面连接。	Web
小程序端	默认仅支持一个小程序连接。如需支持小程序多设备登录，请提交工单申请开通小程序多端服务。	小程序

多设备消息同步

更新时间:2024-08-30

在即时通讯业务中，同一用户账号可能在多个设备上登录。**多设备消息同步**是融云服务端提供的一项服务，可用于同一用户账号的多个设备之间同步收发消息。

默认情况下，融云不会在设备之间同步消息。新消息被某一端设备收取后，其他端无法收取该消息。

适用场景

在融云即时通讯业务中，多设备消息同步适用于以下情况：

- 同一用户账号在多设备上同时在线（无论是否为同一端），希望同步收发消息。例如，用户可能拥有多个移动端设备，如两个 Android 设备、一个 iOS 设备。

> 注意：融云默认已支持多端同时在线，同一用户账号可在移动端、Web 端、桌面端、小程序端最多一个设备上同时在线。但是如果需要允许 App 用户同时在多个移动端设备或多个小程序端上在线，需要分别提交工单申请，详见多端同时在线。

- 同一用户账号换设备登录（无论是否曾在该设备登录过），希望同步收发的消息记录。例如用户从 Android 设备下线后，换到另一个设备从 Web 端登录。
- 同一用户账号在当前设备卸载重装 App，希望同步收发消息记录。

支持在多设备间同步的消息

并非所有消息均支持多设备消息同步。**状态消息**仅支持在多设备同时在线时同步接收，不在线的设备无法通过**多设备消息同步**收到消息。

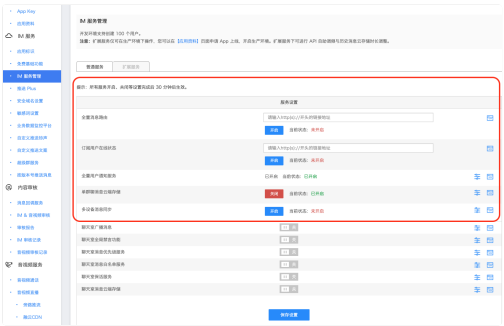
以下情况均属于状态消息：

- 融云内置消息类型中定义为状态消息类型的消息。内置状态类型消息的具体包括：正在输入状态消息（`RC:TypSts`）
- 自定义的状态消息类型的消息。详见各个客户端「自定义消息」文档。
- 使用服务端 API 状态消息接口发送的所有消息（不区分消息类型）均不支持同步。具体的 API 接口为发送单聊状态消息（`/statusmessage/private/publish.json`）、发送群聊状态消息（`/message/group/publish.json`）。

开通服务

请前往控制台，在 IM 服务管理 页面的**普通服务**标签下开通**多设备消息同步**服务。该服务在开发环境免费使用，默认为关闭状态。生产环境预存费用后才可开通服务。

> 服务开启、关闭设置完成后 30 分钟内生效。



对其他功能或业务的影响

多设备消息同步服务的状态对即时通讯业务中的离线补偿[?]、撤回消息、聊天室业务等有影响。

对离线补偿的影响

控制台的**多设备消息同步**服务包含了融云服务端离线补偿机制[?]的开关。

开通**多设备消息同步**服务后，融云服务端自动为 App 启用离线补偿机制。离线补偿机制会在以下场景触发：

- **换设备登录**。用户在新设备上登录后（无论是否曾在该设备登录过），SDK 可获取最近指定天数（默认离线补偿天数为 1 个自然日）在其他终端上发送和接收过的消息。
- **应用卸载重装**。消息与会话列表是存储在本地数据库，用户卸载 App 时会删除本地数据库。用户重新安装 App 后并再次连接时，会触发融云服务端离线补偿机制，SDK 可获取最近指定天数（默认离线补偿天数为 1 个自然日）在其他终端上发送和接收过的消息。

> 注意：在 **换设备登录** 或 **应用卸载重装** 场景下，离线补偿机制仅可获取到最近（默认离线补偿天数为 1 天，最大 7 天）的会话。早于该天数的会话无法通过离线补偿机制获取。因此，离线补偿后的会话列表可能与原设备上或卸载前的会话列表并不一致（您可能会有丢失部分会话的错觉）。

如需修改离线消息补偿的天数，可提交工单。建议谨慎设置离线补偿天数，当单用户消息量超大时，可能会因为补偿消息量过大，造成端上处理压力较大。

对 Web 平台连接数的影响

开通多设备消息同步服务后，可额外支持多 Web 页面连接（每个浏览器标签页也认为是一个连接），最多 10 个。

对撤回消息的影响

- 未开通多设备消息同步服务时，多端之间无法同步撤回的消息。
- 开通多设备消息同步服务后，消息发送端一旦撤回消息，如果用户在其他端在线，则其他端同步撤回该条已发送消息。如果用户在其他端不在线时，则在用户登录后同步撤回已发送的消息。

对服务端 API 发送消息的影响

通过服务端 API 发送消息时，部分接口可通过 `isIncludeSender` 指定消息发送者可否在客户端接收该已发消息。

- 未开通多设备消息同步服务时，仅在发送者已登录客户端（在线）的情况下，通过 Server API 发送的消息可即时同步到发送者的在线客户端，无法同步到离线的客户端。
- 开通多设备消息同步服务后，如果发送者未登录客户端（离线），通过 Server API 发送的消息可在再次上线时同步到发送者的在线客户端。

用户概述　　用户概述

更新时间:2024-08-30

App 用户需要接入融云服务，才能使用即时通讯服务。对于融云来说，用户是指持有由融云分发的有效 Token，接入并使用即时通讯服务的 App 用户。

注册用户

应用服务端（App Server）应向融云服务端提供 App 用户的用户 ID（`userId`），以向融云换取唯一用户 Token。对融云来说，这个以 `userId` 获取 Token 的步骤即注册用户，且必须通过调用 Server API 来完成。

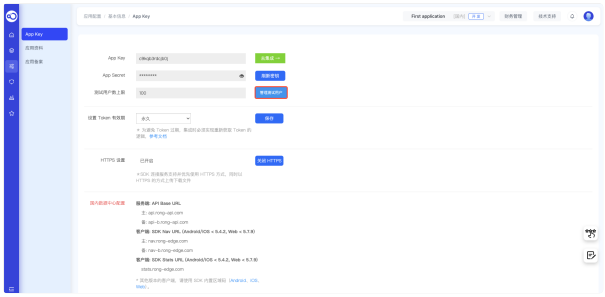
应用客户端必须持有有效 Token，才能成功连接到融云服务端，使用融云即时通讯服务。当 App 客户端用户向服务器发送登录请求时，服务器会查询数据库以检查连接请求是否匹配。

注册用户数限制

- **开发环境**中的注册用户数上限为 100 个。
- 在 **生产环境**中，升级为 **IM 旗舰版**或 **IM 尊享版**后不限制注册用户数。

删除用户

删除用户是指在应用的**开发环境**中，通过控制台删除已注册的测试用户，以控制开发环境中的测试用户总数。**生产环境**不支持该操作。



注销用户

注销用户是指在融云服务中删除用户数据。App 可使用该能力实现自身的用户销户功能，满足 App 上架或合规要求。

融云返回注销成功结果后，与用户 ID 相关数据即被删除。您可以向融云查询所有已注销用户的 ID。如有需要，您可以重新激活已被注销的用户 ID（注意，用户个人数据无法被恢复）。

仅 IM Server API 提供上述能力。

用户信息

用户信息泛指用户的昵称、头像，以及群组的群昵称、群头像等数据。融云服务端不提供用户信息托管维护服务。

在 IMKit SDK UI 中，如果需要在会话页面、好友列表等处显示用户及群组的头像、昵称等信息，需要由应用层提供相关数据。为方便 App 开发者，IMKit SDK 设计并提供了多个信息提供者接口，用于 SDK 向应用层获取用户信息。

好友关系

App 用户之间的好友关系需要由应用服务器（App Server）自行维护。融云不会同步或保存 App 端的好友关系数据。

如果需要对客户端用户之间的消息收发行为进行限制（例如，App 的所有 `userId` 泄漏，导致某个恶意用户可越过好友关系向任意用户发送消息），可以考虑使用用户白名单服务。用户一旦开启并设置白名单，则仅可接收来自该白名单中用户的消息。

用户管理接口

功能分类	功能描述	客户端 API	服务端 API
注册用户	使用 App 用户的用户 ID 向融云换取 Token。	不提供该 API	注册用户
删除用户	参见上文删除用户。	不提供该 API	不提供该 API
废弃 Token	废弃在特定时间点之前获取的 Token。	不提供该 API	作废 Token
注销用户	注销用户是指在融云服务中停用用户 ID，并删除用户个人数据。	不提供该 API	注销用户
查询已注销用户	获取已注销的用户 ID 列表。	不提供该 API	查询已注销用户
重新激活用户 ID	在融云服务中重新启用已注销用户的 ID。	不提供该 API	重新激活用户 ID
设置客户端本地的用户信息	设置用户信息提供者，由应用层负责提供数据。	设置用户信息提供者	不提供该 API

功能分类	功能描述	客户端 API	服务端 API
设置融云服务端的用户信息	设置在融云推送服务中使用的用户名称与头像。	不提供该 API	未提供单独的设置接口。在注册用户时必须提供用户信息。
获取客户端本地的用户信息	获取在会话页面、好友列表等处显示的用户头像、昵称等信息。	获取用户信息	不提供该 API
获取融云服务端的用户信息	获取用户在融云注册的信息，包括用户创建时间和服务端的推送服务使用的用户名称、头像 URL。	不提供该 API	获取信息
修改客户端本地的用户信息	修改在客户端本地数据库中保存的用户昵称、头像等信息。	刷新用户信息	不提供该 API
修改融云服务端的用户信息	修改在融云推送服务中使用的用户名称与头像。	不提供该 API	修改信息
封禁用户	禁止用户连接到融云即时通讯服务，并立即断开连接。可按时长解封或主动解封。查询被封禁用户的用户 ID、封禁结束时间。	不提供该 API	添加封禁用户、解除封禁用户、查询封禁用户
查询用户在线状态	查询某用户的在线状态。	不提供该 API	查询在线状态
黑名单管理	在用户的黑名单列表中添加、移除用户。在 A 用户黑名单的用户无法向 A 发送消息。IMKit 默认已处理被拉黑后的错误，页面会提示“您的消息已经发出，但被对方拒收”。 IMKit 未提供该 API 接口。此处客户端 API 为 IMLib 的 API 接口。	加入黑名单、移出黑名单、查询用户是否在黑名单中、获取黑名单列表	加入黑名单、移出黑名单、查询黑名单
用户白名单	用户一旦开启并设置白名单，则仅可接收来自该白名单中用户的消息。	不提供该 API	开启用户白名单、用户白名单状态查询、添加白名单、移出白名单、查询白名单

群组概述 群组概述

更新时间:2024-08-30

群聊是即时通讯类应用中常见的多人通讯方式，一般包含两个及以上的用户。融云的群组业务支持丰富的群组成员管理、禁言管理等特性，支持离线消息推送和历史消息记录漫游，可用于兴趣群、办公群、客服服务沟通等。IMKit 提供开箱即用的群聊会话 UI 组件。

服务配置

客户端 SDK 默认支持群组业务，不需要申请开通。部分基础功能与增值服务可以在控制台的免费基础功能和 IM 服务管理页面进行开通和配置。

- App Key 下可创建的群组数量无限制。单个用户可加入的群组数量无限制。
- 群组有容量上限，默认群组成员数量上限为 3000 人，可提交工单修改。
- 默认情况下，App Key 未开通**单群聊消息云端存储**服务。您可以自助开通，详见 开通单群聊消息云存储服务。如果是生产环境的 App Key，仅 **IM 旗舰版**、**IM 尊享版**可开通该服务。
- 默认情况下，用户只能查看他们加入群组后的群聊消息。开启服务后，新入群用户可以获取他们加入群组之前的群聊历史消息。详见开通新用户获取加入群组前历史消息服务。

客户端 SDK 使用须知

- 客户端 SDK（IMKit/IMLib）均不提供群组管理的 API。如需创建群组，必须由 App 服务器请求融云服务端 API 实现。其他操作例如解散群组、加入群组、退出群组等群组管理操作，均须由 App 服务器请求融云服务端 API 实现。详见下方群组管理功能。
- 群主、群管理员、群公告、邀请入群、群号搜索等均为群组业务逻辑，需在 App 侧自行实现。
- 融云只负责将消息传达给群组中的所有用户，不维护群组成员的资料（头像、名称、群成员名片等）。App 需要自行在业务服务器上维护相关数据，并实现 IMKit 的相关接口，向 IMKit 提供数据。参见下方文档：
  - 群组信息
  - 群成员用户信息
  - 群组成员列表

群组管理功能

对于客户端开发人员来说，创建群组、解散等基础管理操作只需要与 App 自身的业务服务端交互即可，由 App 服务端负责调用相应的融云服务端 API（Server API）接口完成相关操作。

服务端 API	功能描述
创建群组、解散群组	提供创建者用户 ID、群组 ID、和群名称，向融云服务端申请建群。如解散群组，则群成员关系不复存在。
加入群组、退出群组	加入群组后，默认可查看入群以后产生的新消息。退出群组后，不再接收该群的新消息。
刷新群组信息	修改在融云推送服务中使用的群组信息。
查询群组成员	查询指定群组所有成员的用户 ID 信息。
查询用户所在群组	根据用户 ID 查询该用户加入的所有群组，返回群组 ID 及群组名称。融云不存储群组资料信息，群组资料及群成员信息需要开发者在应用服务器自行维护，如应用服务端维护的用户群组关系有缺失时，可通过此接口来核对校验。
同步用户所在群组	向融云服务端同步指定用户当前所加入的所有群组，防止应用中的用户群组信息与融云服务端的用户所属群信息不一致。如果在集成融云服务前 App Server 上已有群组及成员数据，第一次连接融云服务器时，可使用此接口向融云同步已有的用户与群组对应关系。
禁言指定群成员	在指定的单个群组中或全部群组中，禁言一个或多个用户。被禁言用户可以接收查看群组中其他用户消息，但不能通过客户端 SDK 发送消息。
设置群组全体禁言	将群组全体成员禁言。被禁言群组的所有成员均不能发送消息，需要某些用户可以发言时，可将此用户加入到群禁言用户白名单中。
加入群组全体禁言白名单	群组被整体禁言后，禁言白名单中用户可以发送群消息。

用户信息　　用户信息

更新时间:2024-08-30

要在 IMKit UI 上展示用户头像、昵称等，需要应用层（App）主动向 IMKit SDK 提供**用户信息**（UserInfo）。

IMKit 使用 RongUserInfoManager 类统一管理以下数据。App 需要使用 `RongUserInfoManager` 向 IMKit 提供数据，用于在 UI 上展示。

- **用户信息**：包含昵称、头像
- **群组信息**：包含群组名称、群组头像
- **群成员用户信息**：仅支持群用户昵称

> 提示
>
> **用户信息、群组信息、群成员用户信息必须由应用开发者主动从 App 服务端获取，并提供给 SDK。融云不提供 App 用户与群组信息托管服务。融云服务端的用户昵称及头像仅用于推送服务。**

本文仅描述了应用层（App）如何为 IMKit SDK 提供用户昵称与头像（UserInfo）：

```
UserInfo userInfo = new UserInfo(userId, "userId 对应的名称", Uri.parse("userId 对应头像地址"));
```

刷新用户信息

如果 App 本地持有用户信息数据（例如当前登录用户的昵称和头像），可直接刷新本地缓存和数据库中存储的用户信息（头像与昵称）。刷新后，IMKit UI 会展示最新的用户信息 UserInfo。

刷新用户信息必须在 IMKit 已成功建立 IM 连接后操作，否则无法刷新本地数据。可能适用场景如下：

- App 首次启动，并成功建立 IM 连接以后，可以将自身业务所需的用户信息批量提供给 SDK，由 SDK 写入缓存与本地数据库，供后续使用。
- 在 IM 建立连接后，如果用户昵称、头像等信息变动，由 App 服务端通知客户端（例如使用消息），客户端调用接口刷新用户信息。

```
UserInfo userInfo = new UserInfo(userId, "userId 对应的名称", Uri.parse("userId 对应头像地址"));
RongUserInfoManager.getInstance().refreshUserInfoCache(userInfo);
```

如果 App 本地不持有数据，推荐在 IMKit 需要展示数据时动态提供用户信息。

动态提供用户信息

从 IMKit 5.X 版本开始，SDK 设计了「用户信息提供者」UserDataProvider.UserInfoProvider 接口类。如果 IMKit 无法从 `RongUserInfoManager` 中获取用户信息，将触发 `UserInfoProvider` 的 `getUserInfo` 回调方法。App 应在该回调中提供 SDK 所需要的用户头像与昵称。

获取用户信息数据后，SDK 会自动设置、刷新用户头像与昵称，以及实现相关 UI 展示。

```
public interface UserInfoProvider {
/**
* 获取用户信息。
*
* @param userId 用户 Id。
* @return 用户信息。
*/
UserInfo getUserInfo(String userId);
}
```

步骤 1：设置用户信息提供者

使用 `RongUserInfoManager` 的 setUserInfoProvider 方法设置用户信息提供者。必须在 SDK 初始化之后，建立 IM 连接之前设置。建议在应用生命周期内设置。

```
// 允许 SDK 在本地持久化存储用户信息
boolean isCacheUserInfo = true;

RongUserInfoManager.getInstance().setUserInfoProvider(new UserDataProvider.UserInfoProvider() {
@Override
public UserInfo getUserInfo(String userId) {
...// 此处需要 App 提供用户信息。
}
}, isCacheUserInfo);
```

参数	类型	说明
userInfoProvider	UserDataProvider.UserInfoProvider	用户信息提供者接口
isCacheUserInfo	boolean	是否持久化存储用户信息到 SDK 的本地数据库。`true` 表示存储。`false` 表示不存储。

> 提示
>
> 建议设置 isCacheUserInfo 为 true，即在本地持久化存储用户信息。

在 App 的生命周期中，如果 SDK 获取过用户的信息，便会在内存中缓存该信息。允许持久化存储后，SDK 优先从本地数据库中获取用户信息，App 下次启动时数据仍然可用。SDK 在处理对应信息时默认行为如下：

1. 当 SDK 需要在 UI 上显示用户信息时，首先从内存中查询已获取的数据。
2. 如果 SDK 可从缓存或本地数据库中查询到所需信息，将直接将数据返回 UI 层并刷新 UI。
3. 如果 SDK 未能从缓存或本地数据库查询到所需信息，则将触发 UserDataProvider.UserInfoProvider 的回调方法，并尝试从应用层获取信息。收到应用层提供的相应信息后，SDK 将刷新 UI。

步骤 2：提供用户信息给 SDK

在需要展示用户信息时（例如会话列表页面、会话页面），IMKit 首先会根据用户 ID 逐个调用 RongUserInfoManager 的 getUserInfo 方法获取用户信息。如果 SDK 无法在缓存或本地数据库中查询到所需用户信息，则会触发 UserDataProvider.UserInfoProvider 的回调方法，要求 App 提供用户信息数据。

请在 UserDataProvider.UserInfoProvider 的 getUserInfo 回调触发时，向 SDK 提供用户信息数据。

- **异步获取用户信息，再手动刷新**： App 可以使用该方式避免耗时操作影响 UI。

1. App 需要在 getUserInfo 方法中直接返回 null，同时，App 应在该方法中触发自行获取 userId 的用户信息的逻辑。注意，该步骤会将 userId 的用户信息临时置为空。

```
RongUserInfoManager.getInstance().setUserInfoProvider(new UserDataProvider.UserInfoProvider {
@Override
public UserInfo getUserInfo(String userId) {
...// 在需要展示用户信息时（例如会话列表页面、会话页面），IMKit 首先会根据用户 ID 逐个调用 getUserInfo。
// 此处由 App 自行完成异步请求用户信息的逻辑。后续通过 refreshUserInfoCache 提供给 SDK。
return null;
}

}, true);
```

2. App 成功获取 userId 的用户信息数据后，再调用 RongUserInfoManager 的 refreshUserInfoCache 方法，手动刷新 userId 用户信息。SDK 在收到该用户的信息后会刷新 UI。详见刷新用户信息。

```
UserInfo userInfo = new UserInfo(userId, "userId 对应的名称", Uri.parse("userId 对应的头像地址"))
RongUserInfoManager.getInstance().refreshUserInfoCache(userInfo);
```

- **同步返回用户信息**：App 也可以直接返回 userId 的用户信息。SDK 在收到该用户的信息后会刷新 UI。

```
RongUserInfoManager.getInstance().setUserInfoProvider(new UserDataProvider.UserInfoProvider() {
@Override
public UserInfo getUserInfo(String userId) {
UserInfo userInfo = new UserInfo(userId, "userId 对应的名称", Uri.parse("userId 对应的头像地址"))
return userInfo;
}
}, true);
```

获取用户信息

App 可以主动调用 RongUserInfoManager 的 getUserInfo 方法获取用户信息。SDK 的行为如下：

1. 首先尝试从本地缓存获取应用层提供的数据。如果在设置用户信息提供者时，已授权 SDK 在本地数据库中存储用户信息（即 `isCacheUserInfo` 为 `true`），SDK 还会尝试从本地数据库中获取用户信息。
2. 如果本地没有相关信息的数据，SDK 会触发 `UserInfoProvider` 的 `getUserInfo` 回调方法。如果您的 App 应用层已在该回调中提供数据，则 SDK 可成功获取用户信息 UserInfo。

```
String userId = "用户 Id"
UserInfo userInfo = RongUserInfoManager.getInstance().getUserInfo(userId)
```

输入参数	类型	说明
userId	String	用户 ID

群组信息　　群组信息

更新时间:2024-08-30

要在 IMKit UI 上展示群组的头像（非群成员的头像）、群名称等，需要应用层（App）主动向 IMKit SDK 提供群组信息（Group）。

IMKit 使用 RongUserInfoManager 类统一管理以下数据。App 需要使用 RongUserInfoManager 向 IMKit 提供数据，用于在 UI 上展示。

- **用户信息**：包含昵称、头像
- **群组信息**：包含群组名称、群组头像
- **群成员用户信息**：仅支持群用户昵称

> 提示
>
> 用户信息、群组信息、群成员用户信息必须由应用开发者主动从 App 服务端获取，并提供给 SDK。融云不提供 App 用户与群组信息托管服务。融云服务端的用户昵称及头像仅用于推送服务。

本文仅描述了应用层（App）如何为 IMKit SDK 提供群组信息（Group），以实现在 IMKit UI 上展示群组的头像（非群成员的头像）、群名称等功能。

```
Group group = new Group(groupId, groupName, groupPortrait);
// 5.3.0 及之后，新增 String 类型的 extra 字段
Group group = new Group(groupId, groupName, groupPortrait, extra);
```

刷新群组信息

如果 App 本地持有群组信息数据，可直接刷新本地缓存和数据库中存储的群组信息（群组名称与群组头像）。刷新后，IMKit UI 会展示最新的群组信息 Group。

刷新群组信息必须在 IMKit 已成功建立 IM 连接后操作，否则无法刷新本地数据。可能适用场景如下：

- App 首次启动，并成功建立 IM 连接以后，可以将自身业务所需的用户信息批量提供给 SDK，由 SDK 写入缓存与本地数据库，供后续使用。
- 在 IM 建立连接后，如果群组名称、头像等信息变动，由 App 服务端通知客户端（例如使用消息），客户端调用接口刷新群组信息。

```
Group group = new Group(groupId, groupName, groupPortrait);
RongUserInfoManager.getInstance().refreshGroupInfoCache(group);
```

如果 App 本地不持有数据，推荐在 IMKit 需要展示数据时动态提供群组信息。

动态提供群组信息

从 IMKit 5.X 版本开始，SDK 设计了「群组信息提供者」UserDataProvider.GroupInfoProvider 接口类。如果 IMKit 无法从 RongUserInfoManager 中获取群组名称与群组头像，将触发 GroupInfoProvider 的 getGroupInfo 回调方法。App 应在该回调中提供 SDK 所需要的群组名称与群组头像。

获取群组信息数据后，SDK 会自动设置、刷新群组的名称和头像，以及实现相关 UI 展示。

```
public interface GroupInfoProvider {
/**
* 获取群组信息。
*
* @param groupId 群组 ID.
* @return 群组信息。
*/
Group getGroupInfo(String groupId);
}
```

步骤 1：设置群组信息提供者

在 IMKit 中，您可以使用 RongUserInfoManager 的 setGroupInfoProvider 方法设置群组信息提供者。必须在 SDK 初始化之后，建立 IM 连接之前设置。建议在应用生命周期内设置。

```
// 允许 SDK 在本地持久化存储群组信息
boolean isCacheGroupInfo = true;

RongUserInfoManager.getInstance().setGroupInfoProvider(new UserDataProvider.GroupInfoProvider() {
@Override
public Group getGroupInfo(String groupId) {
...// 此处需要 App 提供群组信息。
}
}, isCacheGroupInfo);
```

参数	类型	说明
groupProvider	UserDataProvider.GroupInfoProvider	群组信息提供者接口
isCacheGroupInfo	boolean	是否持久化存储群组信息到 SDK 的本地数据库。`true` 表示存储。`false` 表示不存储。

> 提示
>
> 建议设置 isCacheGroupInfo 为 true，即在本地持久化存储群组信息。

在 App 的生命周期中，如果 SDK 获取过群组的信息，便会在内存中缓存该信息。允许持久化存储后，SDK 优先从本地数据库中获取群组信息，App 下次启动时数据仍然可用。SDK 在处理对应信息时默认行为如下：

1. 当 SDK 需要在 UI 上显示群组信息时，首先从内存中查询已获取的数据。
2. 如果 SDK 可从缓存或本地数据库中查询到所需信息，将直接将数据返回 UI 层并刷新 UI。
3. 如果 SDK 未能从缓存或本地数据库查询到所需信息，则将触发 UserDataProvider.GroupInfoProvider 的回调方法，并尝试从应用层获取信息。收到应用层提供的相应信息后，SDK 将刷新 UI。

步骤 2：提供群组信息给 SDK

在需要展示群组信息时（例如会话列表页面、会话页面），IMKit 会为根据群组 ID 逐个调用 RongUserInfoManager 的 getGroupInfo 方法获取群组信息。如果 SDK 无法在缓存或本地数据库中查询到所需群组信息，将触发 UserDataProvider.GroupInfoProvider 的回调方法，要求 App 提供群组信息数据。

请在 UserDataProvider.GroupInfoProvider 的 getGroupInfo 回调触发时，向 SDK 提供群组信息数据。

- **异步获取群组信息，再手动刷新**： App 可以使用该方式避免耗时操作影响 UI。

  1. App 需要在 getGroupInfo 方法中直接返回 null，同时，App 应在该方法中触发自行获取 groupId 的群组信息的逻辑。注意，该步骤会将 groupId 的用户信息临时置为空。

```
RongUserInfoManager.getInstance().setGroupInfoProvider(new UserDataProvider.GroupInfoProvider {
@Override
public Group getGroupInfo(String groupId) {
...// 在需要展示群组信息时（例如会话列表页面、会话页面），IMKit 首先会根据群组 ID 逐个调用 getGroupInfo
// 此处由 App 自行完成异步请求群组信息的逻辑。后续通过 refreshGroupInfoCache 提供给 SDK。
return null;
}

}, true);
```

  2. App 成功获取 groupId 的群组信息数据后，再调用 RongUserInfoManager 的 refreshGroupInfoCache 方法，手动刷新 groupId 的群组信息。SDK 收到该群组的信息后会刷新 UI。详见刷新群组信息。

```
Group group = new Group(groupId, groupName, groupPortrait);
RongUserInfoManager.getInstance().refreshGroupInfoCache(group);
```

- **同步返回群组信息**：App 也可以直接返回 groupId 的群组信息。SDK 在收到该群组的信息后会刷新 UI。

```
RongUserInfoManager.getInstance().setGroupInfoProvider(new UserDataProvider.GroupInfoProvider() {
@Override
public Group getGroupInfo(String groupId) {
Group group = new Group(groupId, groupName, groupPortrait);
return userInfo;
}
}, true);
```

获取群组信息

App 可以主动调用 RongUserInfoManager 的 getGroupInfo 方法获取群组信息。SDK 的行为如下：

1. 首先尝试从本地缓存获取应用层提供的数据。如果在设置群组信息提供者时，已授权 SDK 在本地数据库中存储群组信息（即 `isCacheGroupInfo` 为 `true`），SDK 还会尝试从本地数据库中获取群组信息。
2. 如果本地没有相关信息的数据，SDK 会触发 `UserInfoProvider` 的 `getGroupInfo` 回调方法。如果您的 App 应用层已在该回调中提供数据，则 SDK 可成功获取群组信息 Group。

```
Group group = RongUserInfoManager.getInstance().getGroupInfo(groupId);
```

群成员用户信息　　群成员用户信息

更新时间:2024-08-30

要在 IMKit UI 上展示群成员昵称，需要应用层（App）主动向 IMKit SDK 提供**群成员用户信息**（GroupUserInfo）。

IMKit 使用 RongUserInfoManager 类统一管理以下数据。App 需要使用 RongUserInfoManager 向 IMKit 提供数据，用于在 UI 上展示。

- **用户信息**：包含昵称、头像
- **群组信息**：包含群组名称、群组头像
- **群成员用户信息**：仅支持群用户昵称

> 提示
>
> 用户信息、群组信息、群成员用户信息必须由应用开发者主动从 App 服务端获取，并提供给 SDK。融云不提供 App 用户与群组信息托管服务。融云服务端的用户昵称及头像仅用于推送服务。

本文仅描述了应用层（App）如何为 IMKit SDK 提供群成员昵称与头像（GroupUserInfo）。

```
GroupUserInfo groupUserInfo = new GroupUserInfo(groupId, userId, nickname);
// 5.3.0 及之后，新增 String 类型的 extra 字段
GroupUserInfo groupUserInfo = new GroupUserInfo(groupId, userId, nickname, extra);
```

刷新群成员用户信息

如果 App 本地持有群组用户信息数据（群昵称），可直接刷新本地缓存和数据库中存储的群组用户信息数据。刷新后，IMKit UI 会展示最新的群成员用户信息 GroupUserInfo。

刷新群组用户信息必须在 IMKit 已成功建立 IM 连接后操作，否则无法刷新本地数据。可能适用场景如下：

- App 首次启动，并成功建立 IM 连接以后，可以将自身业务所需的群组用户信息批量提供给 SDK，由 SDK 写入缓存与本地数据库，供后续使用。
- 在 IM 建立连接后，如果群组用户昵称变动，由 App 服务端通知客户端（例如使用消息），客户端调用接口刷新群组用户信息。

```
RongUserInfoManager.getInstance().refreshGroupUserInfoCache(groupUserInfo);
```

如果 App 本地不持有数据，推荐在 IMKit 需要展示数据时动态提供群成员用户信息。

动态提供群成员用户信息

从 IMKit 5.X 版本开始，SDK 设计了「群成员用户信息提供者」UserDataProvider.GroupUserInfoProvider 接口类。如果 IMKit 无法从 RongUserInfoManager 中获取群成员用户信息，将触发 GroupUserInfoProvider 的 getGroupUserInfo 回调方法。App 应在该回调中提供 SDK 所需要的群成员昵称。

获取群成员用户信息数据后，SDK 会自动设置、群成员的群内昵称，以及实现相关 UI 展示。

```
/**
* GroupUserInfo提供者。
*/
public interface GroupUserInfoProvider {
/**
* 获取GroupUserInfo。
*
* @param groupId 群组id。
* @param userId 用户id。
* @return GroupUserInfo。
*/
GroupUserInfo getGroupUserInfo(String groupId, String userId);
}
```

步骤 1：设置群成员用户信息提供者

使用 RongUserInfoManager 的 setGroupUserInfoProvider 方法设置群成员用户信息提供者。必须在 SDK 初始化之后，建立 IM 连接之前设置。建议在应用生命周期内设置。

```
// 允许 SDK 在本地持久化存储群成员用户信息
boolean isCacheGroupUserInfo = true;

RongUserInfoManager.getInstance().setGroupUserInfoProvider(new UserDataProvider.GroupUserInfoProvider() {
@Override
public GroupUserInfo getGroupUserInfo(String groupId, String userId) {
...// 此处需要 App 提供群成员用户信息。
}
}, isCacheGroupUserInfo);
```

参数	类型	说明
groupUserInfoProvider	UserDataProvider.GroupUserInfoProvider	群成员用户信息提供者接口
isCacheGroupUserInfo	boolean	是否持久化存储群成员用户信息到 SDK 的本地数据库。`true` 表示存储。`false` 表示不存储。

> 提示
>
> 建议设置 `isCacheGroupUserInfo` 为 `true`，即在本地持久化存储群成员用户信息。

在 App 的生命周期中，如果 SDK 获取过群成员用户信息，便会在内存中缓存该信息。允许持久化存储后，SDK 优先从本地数据库中获取群成员用户信息，App 下次启动时数据仍然可用。SDK 在处理对应信息时默认行为如下：

1. 当 SDK 需要在 UI 上显示群成员用户信息时，首先从内存中查询已获取的数据。
2. 如果 SDK 可从缓存或本地数据库中查询到所需信息，将直接将数据返回 UI 层并刷新 UI。
3. 如果 SDK 未能从缓存或本地数据库查询到所需信息，则将触发 UserDataProvider.GroupUserInfoProvider 的回调方法，并尝试从应用层获取信息。收到应用层提供的相应信息后，SDK 将刷新 UI。

步骤 2：提供群成员用户信息给 SDK

需要展示群成员用户信息时（例如会话列表页面、群聊会话页面），IMKit 会为这些用户逐个调用 RongUserInfoManager 的 getGroupUserInfo 方法获取用户信息。如果 SDK 无法在缓存或本地数据库中查询到所需数据，则会触发 UserDataProvider.GroupUserInfoProvider 的回调方法，要求 App 提供群成员信息数据。

请在 UserDataProvider.GroupUserInfoProvider 的 getGroupUserInfo 回调触发时，向 SDK 提供的群成员用户信息。

- **异步获取群成员用户信息，再手动刷新**：App 可以使用该方式避免耗时操作影响 UI。

  1. App 应在 getGroupUserInfo 方法直接返回 null，同时，App 需要在该方法中触发自行获取 userId 的群组用户信息的逻辑。注意，该步骤会将 userId 在群组内的昵称临时置为空。

```
RongUserInfoManager.getInstance().setGroupUserInfoProvider(new UserDataProvider.GroupUserInfoProvider() {
@Override
public GroupUserInfo getGroupUserInfo(String groupId, String userId) {
... // 在需要展示群组信息时（例如会话列表页面、会话页面），IMKit 首先会根据群组 ID 和用户 ID 逐次调用 getGroupUserInfo。
// 此处由 App 自行完成异步请求用户信息的逻辑。后续通过 refreshGroupUserInfoCache 提供给 SDK。
return null;
}
}, true);
```

  2. App 成功获取 userId 的在群组内的昵称数据后，再调用 refreshGroupUserInfoCache 方法，手动刷新群组成员昵称。SDK 在收到该群成员用户信息后会刷新 UI。详见刷新群成员用户信息。

- **同步返回群成员信息**：App 也可以直接返回 userId 的群成员用户信息。SDK 在收到该数据后会刷新 UI。

```
RongUserInfoManager.getInstance().setGroupUserInfoProvider(new UserDataProvider.GroupUserInfoProvider() {
@Override
public GroupUserInfo getGroupUserInfo(String groupId, String userId) {
GroupUserInfo groupUserInfo = new GroupUserInfo(groupId, userId, "小花");
return groupUserInfo; //同步返回群成员昵称。
}
}, true);
```

获取群成员用户信息

App 可以主动调用 RongUserInfoManager 的 getGroupUserInfo 方法获取群成员用户信息。SDK 的行为如下：

1. 首先尝试从本地缓存获取应用层提供的数据。如果在设置群成员用户信息提供者时，已授权 SDK 在本地数据库中存储用户信息（即 `isCacheGroupUserInfo` 为 `true`），SDK 还会尝试从本地数据库中获取群成员用户信息。
2. 如果本地没有相关信息的数据，SDK 会触发 `GroupUserInfoProvider` 的 `getGroupUserInfo` 回调方法。如果您的 App 应用层已在该回调中提供数据，则 SDK 可成功获取群成员用户信息 GroupUserInfo。

```
GroupUserInfo groupUserInfo = RongUserInfoManager.getInstance().getGroupUserInfo(groupId, userId);
```

输入参数	类型	说明
groupId	String	群组 ID
userId	String	用户 ID

群组成员列表　　群组成员列表

更新时间:2024-08-30

本文描述了应用层（App）如何为 IMKit SDK 提供群组成员数据，用于在群聊会话中输入 @符号时弹出的默认选人界面、发起群组音视频通话的选人界面。设置完成后，在 IMKit UI 中上需要展示群成员列表时可正常显示群组成员的头像、用户名或群内昵称。

> 提示
>
> 融云服务端不向 SDK 提供用户与群组信息托管服务。因此 SDK 所需要的群组信息必须由应用开发者主动从 App 服务端获取，并提供给 SDK。

了解群组成员提供者接口

SDK 在 RongMentionManager 类中定义了群组成员提供者（IGroupMembersProvider）。在 IMKit UI 需要展示群成员列表时，会触发 IGroupMembersProvider 的 getGroupMembers 方法，向应用层获取群组成员信息。

应用开发者必须实现 RongMentionManager 的 IGroupMembersProvider 和 IGroupMemberCallback 接口，SDK 才能获取到 App 的群组成员数据。如果 SDK 无法获取到群组成员，选人界面会显示为空列表。

```
public interface IGroupMembersProvider {
void getGroupMembers(String groupId, IGroupMemberCallback callback);
}

public interface IGroupMemberCallback {
void onGetGroupMembersResult(List<UserInfo> members);
}
```

设置群组成员提供者

在 init 之后调用 RongIM 的 setGroupMembersProvider 方法设置群组成员信息提供者。SDK 所需要的群组成员数据必须由应用开发者主动从应用层（例如 App 服务端）获取。

在 IMKit UI 需要展示群成员列表时，SDK 会触发 getGroupMembers 回调并提供群组 ID。您可以在方法中根据群组 ID 获取群组内用户 ID，再根据用户 ID 从应用层查询用户信息，然后通过 IGroupMemberCallback 将 UserInfo 列表提供给 SDK。

如果 App 实现了群成员用户信息提供者，可用于获取群成员昵称。详见群成员用户信息。

```
RongIM.getInstance().setGroupMembersProvider(new RongMentionManager.IGroupMembersProvider() {
@Override
public void getGroupMembers(String groupId, RongMentionManager.IGroupMemberCallback iGroupMemberCallback) {

//groupId 是群组id 可以根据群组 ID 获取群组内用户 ID，根据 ID 获取用户信息并返回
//代码如下
List<UserInfo> list=new ArrayList();
UserInfo userInfo=new UserInfo("userid1","小花22", Uri.parse(""));
...
list.add(userInfo);
list.add...

iGroupMemberCallback.onGetGroupMembersResult(list); // 调用 callback 的 onGetGroupMembersResult 回传群组信息
}
});
```

监听用户数据变更 监听用户数据变更

更新时间:2024-08-30

通过设置用户数据变更监听器，您可以监听到用户信息、群组信息或群成员用户信息（昵称）的变更。

设置用户数据变更监听器

在应用生命周期内，初始化之后、连接之前，调用以下方法设置全局信息变更监听器 UserDataObserver 。

```
RongUserInfoManager.getInstance().addUserDataObserver(userDataObserver);
```

UserDataObserver 提供了以下回调方法：

- `onUserUpdate()` ：用户信息发生变更时会回调此方法。
- `onGroupUpdate()` ：群信息发生变更时回调此方法。
- `onGroupUserInfoUpdate` ：群成员用户信息（昵称）发生变更时回调此方法。

详细定义如下：

```
/**
* 用户信息变更观察者，所有回调都在 ui 线程
*/
public interface UserDataObserver {
/**
* 用户信息发生变更时的回调方法。
* @param info 变更后的用户信息。
*/
void onUserUpdate(UserInfo info);

/**
* 群组信息发生变更时的回调方法。
* @param group 变更后的群信息。
*/
void onGroupUpdate(Group group);

/**
* 群成员用户信息（昵称）发生变更时的回调方法。
* @param groupUserInfo 变更后的群昵称信息。
*/
void onGroupUserInfoUpdate(GroupUserInfo groupUserInfo);
}
```

移除用户数据变更监听器

通过以下方法可以将已设置的监听器移除。

```
RongUserInfoManager.getInstance().removeUserDataObserver(userDataObserver);
```

会话列表页面 会话列表页面

更新时间:2024-08-30

会话列表页面展示了当前用户设备上的所有会话。一旦 SDK 的本地消息数据库生成消息，SDK 就会生成会话列表，并按照时间倒序排列，置顶会话会排在最前。

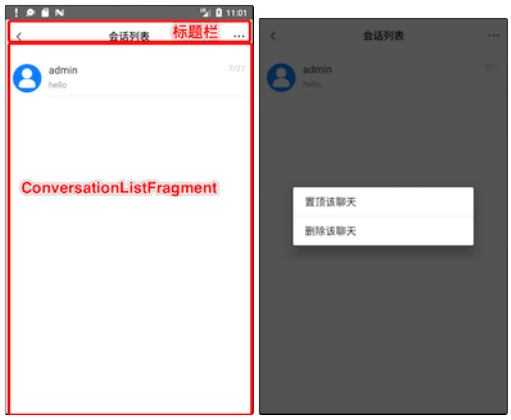
IMKit 提供基于 Activity 类和基于 Fragment 类实现的会话列表页面。

- **基于 Activity**：IMKit SDK 默认提供的会话列表。页面支持一个标题栏和一个会话列表。在 SDK 内部页面需要跳转到会话列表时，将跳转到默认的会话列表 Activity。
- **基于 Fragment**：您可以在应用 Activity 中集成 IMKit 提供的会话列表 Fragment，即自定义会话列表 Activity。注意，您需要将自定义会话列表 Activity 注册到 IMKit SDK 中，以替换 IMKit 默认的会话列表 Activity。

会话列表界面

会话列表页面一般由标题栏和会话列表两部分组成。

> 提示
>
> IMKit 在默认会话列表页面 Activity（RongConversationListActivity）包含了标题栏的实现。如果基于 Fragment 构建会话列表页面，请自行实现标题栏。



标题栏

IMKit 仅在默认会话列表页面 Activity（RongConversationListActivity）实现了标题栏。默认展示「会话列表」。

会话列表

会话列表组件按时间倒序显示所有会话的列表，置顶会话会排在最前。长按会话列表中的会话时显示弹窗菜单。IMKit 默认已实现了置顶会话和删除会话功能。

会话列表的每个项目的视图是在 `PrivateConversationProvider.java` 和其他展示模板中创建和自定义的。您还可以继承使用 `BaseDataProcessor`，聚合（折叠）或过滤需要在列表中展示的会话项目。

> 提示
>
> 如果开启了多设备消息同步，在 **换新设备登录** 或 **应用卸载重装** 场景下，离线补偿机制仅可获取到最近（默认离线补偿天数为 1 天，最大 7 天）的单聊、群聊会话消息。早于该天数的会话无法通过离线补偿机制获取。因此，离线补偿后的会话列表可能与原设备上或卸载前的会话列表并不一致（您可能会有丢失部分会话的错觉）。

用法

IMKit 提供了 Activity 和 Fragment 来创建会话列表页面。`RongConversationListActivity` 是默认提供的会话列表页面。您可以根据需要，选择直接使用默认的会话列表页面 Activity，或者继承 `RongConversationListActivity`，或者使用 ConversationListFragment 构建自定义会话列表 Activity。

IMKit 默认已经实现在会话列表页点击某条会话时，跳转到对应的会话页面。

启动会话列表页面

启动会话列表 Activity 有两种启动方式：通过调用 SDK 内置的 Activity 路由器启动，或通过 intent 跳转。您也可以自定义会话列表页的启动方式，实现从应用的其它 Activity 启动会话列表页面。

使用 RouteUtils

`RouteUtils` 是 IMKit 内置的 Activity 路由器，封装了 SDK 内部各页面跳转方法，可避免应用程序进行重复的 intent 封装。

> 提示
>
> 如果您构建了自定义的会话列表 Activity，必须先向 `RouteUtils` 注册自定义 Activity，才能通过这种方式跳转到自定义会话列表 Activity，否则跳转的是 SDK 默认的 `RongConversationListActivity`。

跳转到会话列表 Activity:

```
RouteUtils.routeToConversationListActivity(context, title);
```

参数	类型	说明
context	Context	activity 上下文
title	String	会话列表页面标题。如果传空，会显示为默认标题“会话列表”。

通过 intent 跳转

您可以自己组装 Intent， 使用 Android 默认机制进行跳转。

```
Intent intent = new Intent(context, MyConversationListActivity.class);
context.startActivity(intent);
```

继承默认会话页面 Activity

您可以通过继承 RongConversationListActivity 创建自定义会话列表 Activity，实现对会话列表页面的调整。注意，必须向 RouteUtils 注册自定义的会话列表 Activity。注册之后，当 SDK 需要跳转到会话列表时，会跳转到您注册的自定义会话列表 Activity。注册方法在应用生命周期内、主进程中调用一次即可。

```
RouteUtils.registerActivity(RouteUtils.RongActivityType.ConversationListActivity, MyConversationListActivity.class);
```

使用 Fragment 构建自定义会话列表 Activity

IMKit 提供的会话列表 Fragment 类名为 ConversationListFragment。为方便应用程序自定义会话列表页面，推荐继承使用会话列表 Fragment 类，构建自定义的会话列表 Activity。注意，自定义会话列表 Activity 需要被注册到 IMKit SDK 中，以替换 IMKit 默认的会话列表 Activity。

1. 您可以在应用 Activity 中集成 IMKit 提供的会话列表 Fragment。以下示例声明一个新的MyConversationListActivity。

```
<activity
android:name=".ui.activity.MyConversationListActivity"
android:launchMode="singleTop"
android:screenOrientation="portrait"
android:windowSoftInputMode="stateHidden|adjustResize">
</activity>
```

2. 描述 Activity 布局。此处以 activity_conversation_list.xml 为例。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="match_parent">
<FrameLayout
android:id="@+id/container" //此容器用于动态放置 fragment
android:layout_width="match_parent"
android:layout_height="match_parent" />
</LinearLayout>
```

3. 继承 IMKit 的 ConversationListFragment 创建子类，例如 AppConversationListFragment。在 MyConversationListActivity 的 onCreate() 方法中使用您自定义的会话列表 Fragment 子类。

```
class MyConversationListActivity extends FragmentActivity {

@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_conversation_list);

ConversationListFragment conversationListFragment=new AppConversationListFragment();
FragmentManager manager = getSupportFragmentManager();
FragmentTransaction transaction = manager.beginTransaction();
transaction.replace(R.id.container, conversationListFragment);
transaction.commit();
}
}
```

4. 向 RouteUtils 注册自定义的会话列表 Activity，以替换 IMKit 默认的会话列表 Activity。方法在应用生命周期内、主进程中注册一次即可。

```
RouteUtils.registerActivity(RouteUtils.RongActivityType.ConversationListActivity, MyConversationListActivity.class);
```

参数	类型	说明
activityType	RongActivityType	IMKit 内置的 Activity 类型枚举。此处为 `RongActivityType.ConversationListActivity`
class	Class<? extends Activity>	应用自定义的会话列表 Activity 的类名。

注册完成后，在 SDK 内部页面需要跳转到会话列表场景时，SDK 会自动跳转到您注册的应用 Activity。

定制化

修改会话头像形状

会话列表中的每个会话项目上均会显示一个头像图标，即会话头像（不是聊天页面内中消息列表中的头像）。单聊会话显示对方用户头像，群聊会话显示群组头像，聚合会话显示默认头像或应用程序主动设置的头像。IMKit 支持通过 IMKit 全局配置单独控制会话列表中的会话头像样式。

会话头像显示默认为矩形，可修改为圆形。头像显示大小默认值为 48dp*48dp。请在 App 初始化之后调用以下代码进行设置：

```
RongConfigCenter.featureConfig().setKitImageEngine(new GlideKitImageEngine() {
@Override
public void loadConversationListPortrait(@NonNull Context context, @NonNull String url, @NonNull ImageView imageView, Conversation conversation) {
Glide.with(context).load(url)
.apply(RequestOptions.bitmapTransform(new CircleCrop()))
.into(imageView);
}
});
```

如需修改聊天页面内消息上显示的头像样式，请转至会话页面。

设置默认加载会话条目的数量

IMKit 在加载会话列表时（进入会话列表页面、上拉、下拉），默认从本地加载 100 条会话数据。您可以修改默认加载的会话条目数量。

```
RongConfigCenter.conversationListConfig().setCountPerPage();
```

> 提示
>
> 客户端的会话列表是根据本地消息生成的，因此会话列表仅从本地获取。

自定义长按会话的弹窗样式

IMKit 默认在长按会话时显示以下弹窗。您可以在设置会话列表监听器 中描述的 `onConversationLongClick()` 方法中实现自定义处理逻辑，并返回 `true`，以修改弹窗样式。

添加自定义头、尾、空布局

> 提示
>
> IMKit 从 5.1.0 版本开始支持该功能。

IMKit 的 ConversationListFragment 支持在会话列表中添加自定义的头（Header）、尾（Footer）和空（Empty）视图。

1. 继承 IMKit 的 ConversationListFragment 创建子类。
2. 在子类中实现 `addHeaderView()`、`addFooterView()` 和 `setEmptyView()` 等方法，
3. 在子类的 `onViewCreated()` 方法中按需调用自身的 `addHeaderVew()`、`addFooterView()` 和 `setEmptyView()` 等方法。

```
@Override
public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
super.onViewCreated(view, savedInstanceState);
addHeaderView(view);
addFooterView(view);
setEmptyView(view);
}

/**
* @param view 自定义列表 header view
*/
public void addHeaderView(View view) {
mAdapter.addHeaderView(view);
}

/**
* @param view 自定义列表 footer view
*/
public void addFooterView(View view) {
mAdapter.addFootView(view);
}

/**
* @param view 自定义列表 空数据 view
*/
public void setEmptyView(View view) {
mAdapter.setEmptyView(view);
}

/**
* @param emptyId 自定义列表 空数据的 LayoutId
*/
public void setEmptyView(@LayoutRes int emptyId) {
mAdapter.setEmptyView(emptyId);
}
```

是否展示网络连接状态提示

当连接断开或者重连时，SDK 会在会话列表页面与会话页面顶部展示连接状态提示栏。

如果需要在会话列表页面上禁用该提示，您可以在应用 `res/values` 目录下创建 `rc_config.xml` 文件，将以下变量的值设为 `false` 来关闭提示栏。该 XML 配置项属于 IMKit 全局默认配置，会同时禁用会话页面与会话列表页面的连接状态提示栏。

```
<bool name="rc_is_show_warning_notification">false</bool>
```

是否清除通知栏通知

IMKit 支持在打开者会话列表页面和会话页面时，清除通知面板的所有通知。IMKit 中此开关默认关闭，即不清除通知。

如果需要在会话列表页面时清除通知，您可以在应用 `res/values` 目录下创建 `rc_config.xml` 文件，将以下变量的值设为 `true`。该 XML 配置项属于 IMKit 全局默认配置，会同时影响会话页面与会话列表页面的默认行为。

```
<!--进入会话或会话列表界面，是否清除通知栏通知-->
<bool name="rc_wipe_out_notification_message">true</bool>
```

通过 XML 资源自定义 UI

IMKit 支持更改会话列表与其中会话条目的展示样式。如果默认展示样式不能满足需求，您可以通过修改 XML 资源文件修改背景、字体颜色或调整布局里各组件位置等。

IMKit 资源文件	位置	说明
rc_conversationlist_fragment.xml	res/layout	会话列表的展示样式。它是整个输入框的容器。复制 IMKit 源码文件的全部内容，在应用程序项目下创建 `/res/layout` 下创建同名文件。
rc_conversationlist_item.xml	res/layout	会话列表中的每一条会话（item）的展示样式。复制 IMKit 源码文件的全部内容，在应用程序项目下创建 `/res/layout` 下创建同名文件。

以上 XML 资源也可从 官网 SDK 下载 页面获取。

> 提示
>
> 修改时不要删除 XML 中的 view，不要修改 ID，否则会导致 SDK 崩溃。

自定义会话条目的展示模板

下表列出了 IMKit 中内置的会话列表项展示模板。如果需要修改会话列表项的展示逻辑，建议您自定义展示模板，并注册使用。

模板类名	适用场景
PrivateConversationProvider.java	单聊会话的展示模板。
BaseConversationProvider.java	会话模板基类，缺省模板，没有匹配到其它模板的会话将以此模板展示。

模板类名	适用场景
ConversationListEmptyProvider.java	空会话列表展示模板 会话列表没有数据时的展示模板。

1. 创建 CustomConversationProvider，继承会话模板基类 BaseConversationProvider。

2. 重写 isItemViewType() 和其它需要自定义的方法。

```
public class CustomConversationProvider extends BaseConversationProvider {
@Override
public boolean isItemViewType(BaseUiConversation item) {
//根据业务需要，判断 item 是该模板需要处理的会话时，返回 true, 否则返回 false
//此处以自定义私聊会话模板为例
return item.mCore.getConversationType().equals(Conversation.ConversationType.PRIVATE);
}

@Override
public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
//根据业务需要，自定义处理
return super.onCreateViewHolder(parent, viewType);
}

@Override
public void bindViewHolder(ViewHolder holder, BaseUiConversation uiConversation, int position, List<BaseUiConversation> list,
IViewProviderListener<BaseUiConversation> listener) {
//根据业务需要，自定义处理
super.bindViewHolder(holder, uiConversation, position, list, listener);
}
}
```

3. 注册自定义模板，或替换 SDK 内置模板。

```
//获取会话模板管理器
ProviderManager<BaseUiConversation> providerManager = RongConfigCenter.conversationListConfig().getProviderManager();

//用自定义模板替换 SDK 原有私聊会话展示模板
providerManager.replaceProvider(PrivateConversationProvider.class, new CustomConversationProvider());

//注册一个自定义模板
providerManager.replaceProvider(new CustomSystemConversationProvider());
```

其他定制化

- 删除会话：IMKit 默认在长按会话时显示以下弹窗，实现了删除会话功能。如果已有实现无法满足您的需求，可以使用 IMCenter 提供的删除会话 API。
- 获取会话: IMKit SDK 已经实现了一整套会话获取和展示逻辑，一般不需要额外调用获取会话相关 API。如果已有实现无法满足您的需求，可以使用 IMLib SDK 中相关 API。
- 会话草稿: IMKit 中默认已实现了获取会话草稿、删除草稿的功能和页面刷新，您不需要额外调用 API。如果已有实现无法满足您的需求，可以使用 IMKit 和 IMLib 提供的 API。
- 会话置顶: IMKit 中默认已实现了置顶会话、取消置顶等功能和页面刷新，您不需要额外调用 API。如果已有实现无法满足您的需求，可以使用 IMKit 和 IMLib 提供的 API。
- 自定义处理会话列表数据：支持按会话类型或自定义规则过滤会话，在会话列表展示过滤后的项目。

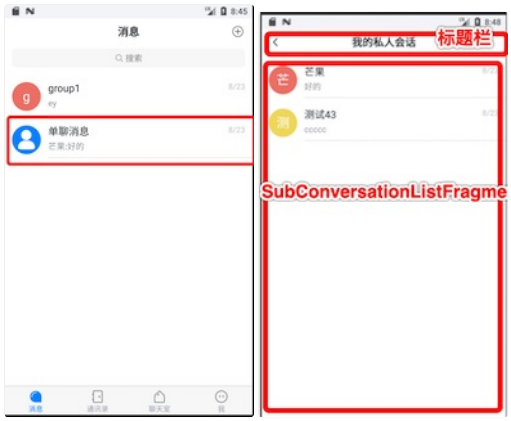
按类型聚合会话 按类型聚合会话

更新时间:2024-08-30

IMKit 支持在会话列表中按照会话类型进行聚合（折叠）。设置聚合显示的会话类型后，该类型的会话在会话列表中仅被展示为一个聚合条目。默认情况下，IMKit 的会话列表页面会以平铺方式展示所有本地会话。

下图展示了设置了单聊类型会话聚合显示的效果。在会话列表中，所有单聊会话折叠为“单聊消息”（左侧），点击后跳转到聚合会话列表（右侧）。

> 提示
>
> IMKit 在默认的聚合会话页面 Activity（RongSubConversationListActivity）包含了标题栏的实现。如果基于 Fragment 构建聚合会话列表页面，请自行实现标题栏。



局限

- 会话列表中的聚合会话项目不支持置顶操作。

设置聚合显示的会话类型

聚合会话功能支持单聊、群聊、系统会话类型。如需聚合显示，需要继承 BaseDataProcessor 实现自定义数据处理器，在打开会话列表页面之前提供给 IMKit。

1. 声明自定义的数据处理器类 MyAggregateDataProcessor，继承 SDK 的数据处理器 BaseDataProcessor。重写 isGathered 的方法，。当会话类型为需要聚合显示的会话类型时，返回 true。下例中我们聚合显示所有系统会话。

```
public class MyAggregateDataProcessor extends BaseDataProcessor<Conversation> {
/**
* 自定义会话列表页面支持的会话类型，此处设置为支持单聊、群组和系统会话。
*/
@Override
public Conversation.ConversationType[] supportedTypes() {
Conversation.ConversationType[] types = {Conversation.ConversationType.PRIVATE, Conversation.ConversationType.GROUP, Conversation.ConversationType.SYSTEM};
return types;
}

/**
* 自定义需要聚合显示的会话。此处设置单聊会话聚合显示。
*/
@Override
public boolean isGathered(Conversation.ConversationType type) {
if (type.equals(Conversation.ConversationType.PRIVATE)) {
return true;
} else {
return false;
}
}

}
```

2. 在打开会话列表页面之前，使用方法设置会话列表数据处理器。

```
RongConfigCenter.conversationListConfig().setDataProcessor(new MyAggregateDataProcessor());
```

聚合会话列表页面

设置会话聚合后，在会话列表页面中，被聚合会话会被折叠为一个聚合会话条目（GatheredConversation）。点击该条目后，SDK 会跳转到聚合会话列表页面。

IMKit 提供基于 Activity 类和基于 Fragment 类实现的聚合会话列表页面。

- **基于 Activity**：IMKit 提供了默认的聚合会话列表页面 `RongSubConversationListActivity`。页面支持一个标题栏和一个聚合会话列表。在 SDK 内部页面需要跳转到聚合会话列表时，默认会跳转到该 Activity。
- **基于 Fragment**：您可以使用 IMKit 的 `SubConversationListFragment` 构建自定义聚合会话列表页面。注意，您需要将自定义会话列表 Activity 注册到 IMKit SDK 中，以替换 IMKit 默认的聚合会话列表 Activity。

启动默认聚合会话列表页面

IMKit SDK 默认提供的聚合会话列表页面 Activity。点击聚合会话时，默认跳转到以下聚合会话列表 `RongSubConversationListActivity`。

如果您需要在 App 的其它页面跳转到聚合会话列表 Activity，可以使用 IMKit 内置的 Activity 路由器 `RouteUtils`。

> 提示
>
> 如果您构建了自定义的会话列表 Activity，必须先向 `RouteUtils` 注册自定义 Activity，才能通过这种方式跳转到自定义会话列表 Activity，否则跳转的是 SDK 默认的 `RongSubConversationListActivity`。

```
RouteUtils.routeToSubConversationListActivity(context, conversationType, title);
```

参数	类型	说明
context	Context	activity 上下文
type	Conversation.ConversationType	会话类型。
title	String	会话列表页面标题。如果传空，会显示为默认标题 “会话列表”。

> 提示
>
> 数据处理器通过抽象类 `BaseDataProcessor` 支持的版本为 5.1.3.x 系列中的稳定版，以及 5.1.5 及之后所有版本。其它版本需要使用 `DataProcessor` 接口自行实现一个 `DataProcessor` 类。关于数据处理器的详细说明，可参见自定义数据处理。

使用 Fragment 构建聚合会话列表 Activity

如果 IMKit 提供的默认聚合会话列表 `RongSubConversationListActivity` 不能满足需求，推荐继承使用 IMKit 提供的聚合会话列表 Fragment 类 `SubConversationListFragment` 构建自定义聚合会话列表页面。

> 提示
>
> App 自定义的聚合会话列表 Activity 必须注册到 IMKit SDK 中，以替换 IMKit 默认的聚合会话列表 Activity。

1. 您可以在应用 Activity 中集成 IMKit 提供的聚合会话列表 Fragment。以下示例声明一个新的`MyAggregateConversationListActivity`。

```
<activity
android:name="xxx.xxx.MyAggregateConversationListActivity"
android:screenOrientation="portrait"
android:windowSoftInputMode="stateHidden|adjustResize">
<intent-filter>
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />
<data
android:host="${`applicationId`}"
android:pathPrefix="/subconversationlist"
android:scheme="rong" />
</intent-filter>
</activity>
```

2. 实现聚合会话列表布局。此处以 `activity_subconversation_list.xml` 布局为例。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="match_parent">
<FrameLayout
android:id="@+id/container"
android:layout_width="match_parent"
android:layout_height="match_parent" />
</LinearLayout>
```

3. 继承 IMKit 的 `SubConversationListFragment` 创建子类，例如 `AppSubConversationListFragment`。在 `MyAggregateConversationListActivity` 的 `onCreate` 方法中创建并使用 `SubConversationListFragment`。

```
SubConversationListFragment subconversationListFragment = new AppSubConversationListFragment();
FragmentManager manager = getSupportFragmentManager();
FragmentTransaction transaction = manager.beginTransaction();
transaction.replace(R.id.container, subconversationListFragment);
transaction.commit();
```

4. 向 RouteUtils 注册自定义的聚合会话列表 Activity，以替换 IMKit 默认的聚合会话列表 Activity。方法在应用生命周期内、主进程中注册一次即可。

```
RouteUtils.registerActivity(RouteUtils.RongActivityType.SubConversationListActivity, MyAggregateConversationListActivity.class);
```

参数	类型	说明
activityType	RongActivityType	IMKit 内置的 Activity 类型枚举，此处为 RongActivityType.SubConversationListActivity
className	Class<? extends Activity>	应用自定义的聚合会话列表 Activity 的类名。

注册完成后，在 SDK 内部页面需要跳转到聚合会话列表场景时，SDK 会自动跳转到您注册的应用 Activity。

定制化

设置会话按类型聚合后，会话列表页面会展示对应的聚合会话（GatheredConversation）条目。您可以为聚合会话设置头像与昵称（标题）。

设置聚合会话头像

您可以为会话列表页面中的聚合会话（GatheredConversation）设置自定义头像。默认使用融云默认头像。

```
RongConfigCenter.gatheredConversationConfig().setGatherConversationPortrait(conversationType, portraitUri);
```

参数	类型	说明
conversationType	ConversationType	聚合显示的会话类型
portraitUri	Uri	自定义的聚合会话的头像资源 uri

设置聚合会话的标题

您可以为在会话列表页面中的聚合会话（GatheredConversation）设置自定义标题。如未设置，使用融云默认标题规则。默认标题与会话类型相关，下表列出了各会话类型的聚合显示条目的标题字符串：

聚合会话类型	会话标题	资源名称
单聊	”单聊消息“	R.string.rc_gathered_conversation_private_title
群聊	”群聊消息“	R.string.rc_gathered_conversation_group_title
系统	”系统消息“	R.string.rc_gathered_conversation_system_title

如果需要更改聚合会话标题，可以在进入会话列表之前，调用下面方法配置：

```
RongConfigCenter.gatheredConversationConfig().setConversationTitle(conversationType, title);
```

参数	类型	说明
conversationType	ConversationType	聚合显示的会话类型
title	String	自定义的聚合会话标题

如果使用 IMKit 在默认的聚合会话页面 Activity（RongSubConversationListActivity），聚合会话的标题也会显示在页面标题栏中。

会话页面　　会话页面

更新时间:2024-08-30

会话页面即应用程序中的聊天页面，主要由消息列表和输入区两部分组成。IMKit 提供默认的会话页面 Activity 类和基于 Fragment 类实现的会话页面。

- **基于 Activity**：IMKit SDK 提供了默认会话页面 `RongConversationActivity`。页面包含标题栏、消息列表和输入区域。在会话列表页点击某条会话时，会跳转到对应的会话页面。应用程序可以直接使用 `RongConversationActivity` 或通过继承创建自定义的会话 Activity。
- **基于 Fragment**：您可以在应用 Activity 中集成 IMKit 提供的会话 Fragment。

聊天界面

聊天界面一般由三个部分组成：标题栏、消息列表、输入区域。

> 提示
>
> IMKit 在默认会话页面 Activity（RongConversationActivity）包含了标题栏的实现。如果基于 Fragment 构建会话页面，请自行实现标题栏。

标题栏

IMKit 仅在默认的会话页面 RongConversationActivity 中实现了会话页面标题栏。标题栏用于显示会话的标题，该标题可以是群组名称，也可以是一对一聊天中另一个用户的名称。在单聊会话中，标题栏可显示对方的输入状态。标题栏的视图是在 TitleBar 中创建和控制的。

标题栏提供两个按钮，一个位于组件的左侧，另一个位于组件的右上角。当点击左按钮时，会调用 Activity 的 finish() 方法来退出当前屏幕。右侧菜单按钮默认不展示。在通过继承 RongConversationActivity 创建的子类中，您可以通过 mTitleBar 的 setRightVisible 方法展示右侧按钮，通过 setOnRightIconClickListener 设置监听器，在用户点击右侧按钮后跳转到您自行实现的页面或进行其他自定义操作。

如果您使用 ConversationFragment 构建自己的会话页面，则默认不包含标题栏。您可以参考 IMKit 的标题栏组件自行实现标题栏，详见 TitleBar.java 。

消息列表

消息列表组件按时间顺序显示所有消息的列表。当前用户发送的消息与其他成员发送的消息区分显示。消息列表的视图是在 BaseMessageItemProvider 和各类型消息的展示模板中创建和自定义的。您还可以使用 MessageListAdapter 自定义列表视图中的每个项目。

输入区域

消息输入组件是用户可以输入文本、文件、语音、图像、视频等消息的地方。输入区域的视图是在 RongExtension 中统一控制的。

用法

IMKit 提供了 Activity 和 Fragment 来创建会话页面。RongConversationActivity 是默认提供的会话页面，在会话列表页点击某条会话时，会跳转到对应的会话页面。您可以根据需要，选择直接使用默认的会话页面 Activity，继承 RongConversationActivity，或者使用 ConversationFragment 构建自定义会话 Activity。

启动会话页面

启动会话页面 Activity 有两种方式：通过调用 SDK 内置的 Activity 路由器启动，或通过 intent 跳转。您也可以自定义会话页启动方式，实现从应用的其它 Activity 启动会话页面。

> 提示
>
> IMKit 中默认已实现了会话页跳转逻辑。默认情况下，点击会话列表页的会话项目后，SDK 将自动跳转到 RongConversationActivity，应用层无需处理。

使用 RouteUtils

RouteUtils 是 IMKit 内置的 Activity 路由器，封装了 SDK 内部各页面跳转方法，可避免应用程序进行重复的 intent 封装。

> 提示
>
> 如果您构建了自定义的会话 Activity，必须先向 RouteUtils 注册自定义 Activity，才能通过这种方式跳转到自定义会话 Activity，否则跳转的是 SDK 默认的 RongConversationActivity。

跳转到会话 Activity:

```
String targetId = "userId";
ConversationIdentifier conversationIdentifier = new ConversationIdentifier(Conversation.ConversationType.PRIVATE, targetId);

RouteUtils.routeToConversationActivity(context, conversationIdentifier, false, bundle)
```

参数	类型	说明
context	Context	Activity 上下文
conversationIdentifier	ConversationIdentifier	会话类型
disableSystemEmoji	Boolean	是否隐藏 SDK 内置 Emoji 表情。true 为隐藏，false 为不隐藏。
bundle	Bundle	扩展参数

通过 intent 跳转

您可以自己组装 Intent， 使用 Android 默认机制进行跳转。

```
Intent intent = new Intent(context, ConversationActivity.class);
intent.putExtra(RouteUtils.CONVERSATION_TYPE, type.getName().toLowerCase());
intent.putExtra(RouteUtils.TARGET_ID, targetId);
intent.putExtra(RouteUtils.DISABLE_SYSTEM_EMOJI, false);
intent.putExtras(bundle);
context.startActivity(intent);
```

继承默认会话页面 Activity

您可以通过继承 RongConversationActivity 创建自定义会话 Activity，实现对会话页面的调整。注意，必须向 RouteUtils 注册自定义的会话 Activity。注册之后，自定义的会话 Activity 将替换 IMKit 默认的会话 Activity。替换完成后，当在会话列表页点击会话时，SDK 会跳转到您注册的自定义会话 Activity。注册方法在应用生命周期内、主进程中调用一次即可。

```
RouteUtils.registerActivity(RouteUtils.RongActivityType.ConversationActivity, MyConversationActivity.class)
```

使用 Fragment 构建自定义会话 Activity

为方便应用程序自定义会话页面，推荐您继承使用会话 Fragment 类 ConversationFragment 构建自定义的会话 Activity。您需要将自定义会话 Activity 注册到 IMKit SDK 中，以替换 IMKit 默认的会话 Activity。

1. 您可以在应用 Activity 中集成 IMKit 提供的会话 Fragment。以下示例声明一个新的 MyConversationActivity。

```
<activity
android:name="xxx.xxx.MyConversationActivity"
android:screenOrientation="portrait"
android:windowSoftInputMode="stateHidden|adjustResize">
</activity>
```

2. 实现 Activity 布局。此处以 activity_conversation.xml 为例。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="match_parent">
<FrameLayout
android:id="@+id/container"
android:layout_width="match_parent"
android:layout_height="match_parent" />
</LinearLayout>
```

3. 通过继承 IMKit 的 ConversationFragment 创建子类，在 MyConversationActivity 的 onCreate() 方法中使用您自定义的会话 Fragment 子类。

```
class MyConversationActivity extends FragmentActivity {
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
super.setContentView(R.layout.activity_conversation);

// 使用 ConversationFragment 的子类添加会话界面
ConversationFragment conversationFragment = new AppConversationFragment();
FragmentManager manager = getSupportFragmentManager();
FragmentTransaction transaction = manager.beginTransaction();
transaction.replace(R.id.container, conversationFragment);
transaction.commit();
}
}
```

4. 向 RouteUtils 注册自定义的会话 Activity，以替换 IMKit 默认的会话 Activity。替换完成后，当在会话列表页点击会话时，SDK 会跳转到您注册的自定义会话 Activity。注册方法在应用生命周期内、主进程中调用一次即可。

```
RouteUtils.registerActivity(RouteUtils.RongActivityType.ConversationActivity, MyConversationActivity.class)
```

定制化

您可以通过修改 IMKit 全局配置和重写 RongConversationActivity 或 ConversationFragment 的方法来自定义聊天页面。

在开始定制化之前，建议您首先继承 SDK 内置会话页面 RongConversationActivity 或继承使用 ConversationFragment，创建并实现自己的会话页面 Activity。

> 提示
>
> 会话页面中部分样式与行为受 IMKit 全局配置影响。如需了解全局配置，参见 [配置指南]。

修改用户头像形状与大小

会话页面的消息列表中显示的用户头像（指消息上显示的头像）可以通过 IMKit 全局配置修改。

头像形状默认为矩形，可修改为圆角显示。头像显示大小默认值为 40dp*40dp。请在 SDK 初始化之后调用以下代码进行设置：

```
RongConfigCenter.featureConfig().setKitImageEngine(new GlideKitImageEngine() {
@Override
public void loadConversationPortrait(@NonNull Context context, @NonNull String url, @NonNull ImageView imageView, Message message) {
Glide.with(context).load(url)
.apply(RequestOptions.bitmapTransform(new CircleCrop()))
.into(imageView);
}
});
```

> 提示
>
> 修改 IMKit 全局配置会影响 IMKit 中所有用户头像的显示。

隐藏用户昵称

IMKit 仅支持在单聊会话页面隐藏用户昵称。

```
// 设置是否显示用户昵称，仅支持单聊会话
RongConfigCenter.conversationConfig().setShowReceiverUserTitle(true);
```

隐藏默认 Emoji 表情

> 提示
>
> SDK 从 5.2.3 开始支持禁用内置 Emoji。禁用后，表情面板不再显示 Emoji 标签页。

配置方式详见表情区域。

设置默认拉取历史消息数

IMKit 在进入会话页面时和下拉页面时获取历史消息，默认拉取 10 条。优先从本地拉取，本地消息拉取完之后，如果已开通单群聊历史消息云存储功能，IMKit 还可从远端拉取历史消息。您可以修改默认拉取的历史消息数。

> 提示
>
> 拉取远端历史消息要求已开通单群聊历史消息云存储功能。

请在进入会话页面前设置。

```
// 默认拉取历史消息数量
RongConfigCenter.conversationConfig().setConversationHistoryMessageCount(10);
// 默认拉取远端历史消息数量
RongConfigCenter.conversationConfig().setConversationRemoteMessageCount(10);
```

> 提示
>
> 您可以自助开通单群聊历史消息云存储功能，详见开通单群聊消息云存储服务。

修改长按消息菜单中的删除行为

IMKit 会话页面默认已在长按消息弹出的菜单中实现了删除消息的选项。

- 如果 IMKit 版本 ≥ 5.6.3，默认实现的删除功能会删除本地消息，如果 App Key 已开通单群聊消息云端存储服务，则同步删除远端消息。
- 如果 IMKit 版本 < 5.6.3，默认实现的删除功能仅删除本地消息，如果 App Key 已开通单群聊消息云端存储服务，不会删除服务端历史消息记录中的该条消息。用户再次获取历史消息、或触发离线消息补偿（卸载重装、换设备登录）时，该消息可能会再次出现在聊天页面中。如需删除远端消息，应用程序可修改删除按钮的默认行为。

如需修改 IMKit 会话页面中删除消息选项的默认行为，方法如下：

- 如果 SDK 版本 ≥ 5.2.3，可通过 IMKit 全局配置修改 `needDeleteRemoteMessage` 属性。`true` 表示需要 IMKit 同步删除本地与远端消息。`false` 表示仅从本地消息数据库中删除消息。

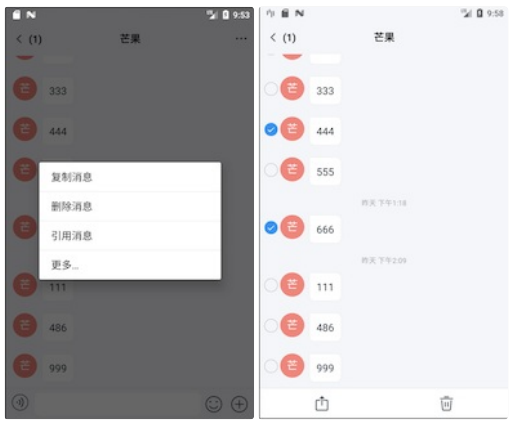
```
RongConfigCenter.conversationConfig().setNeedDeleteRemoteMessage(true);
```

- 如果 5.1.3.19 ≦ SDK 版本 < 5.2.3，您需要自定义长按消息弹窗菜单，并添加自定义的删除操作。在实现自定义删除消息时调用 IMCenter 的 `deleteRemoteMessages` 方法，该方法可同时删除本地与远端消息，并刷新 UI。详见自定义长按消息菜单选项。

> 提示
>
> 如果已有实现无法满足您的需求，可以直接使用 IMKit 核心类 IMCenter 提供的 API。具体的核心类、API 与 使用方法，详见 删除消息。

修改多选消息数量限制

IMKit 会话页面默认实现消息多选能力，在 IMKit 会话页面的消息列表中长按单条消息后，SDK 会弹出菜单，其中包含一个可进入下级菜单的「更多」选项，点击后会进入消息选择界面。



IMKit 默认最多选择 100 条消息进行后续操作。您可以通过全局配置修改多选消息数量上限：

```
RongConfigCenter.conversationConfig().rc_max_message_selected_count = 100;
```

如果修改 IMKit 默认配置，您可以在应用 res/values 目录下创建 rc_config.xml 文件，修改以下配置项的值。

```
<integer name="rc_max_message_selected_count">100</integer>
```

隐藏长按消息弹窗中的更多按钮

在 IMKit 会话页面的消息列表中长按单条消息后，SDK 会弹出菜单，包含默认实现和展示的功能选项，例如「删除消息」、「引用消息」等，以及一个可进入下级菜单的「更多」选项。IMKit 支持通过修改全局配置隐藏弹窗中的「更多」选项。

请在进入会话页面前设置。

```
RongConfigCenter.conversationConfig().setShowMoreClickAction(true);
```

> 提示
>
> 如果需要自定义菜单选项的显示名称、顺序、以及自行增删菜单选项，请参见自定义长按消息菜单。

是否展示网络连接状态提示

当连接断开或者重连时，SDK 会在会话列表页面与会话页面顶部展示连接状态提示栏。

您可以通过全局配置，在会话页面上禁用该提示：

```
RongConfigCenter.conversationConfig().rc_is_show_warning_notification = false;
```

如果修改 IMKit 默认配置，您可以在应用 res/values 目录下创建 rc_config.xml 文件，将以下变量的值设为 false 来关闭提示栏。该 XML 配置项会同时禁用会话页面与会话列表页面的连接状态提示栏。

```
<bool name="rc_is_show_warning_notification">false</bool>
```

是否清除通知栏通知

IMKit 支持在打开者会话页面与会话列表页面时，清除通知面板的所有通知。IMKit 中此开关默认关闭，即不清除通知。

您可以通过全局配置，设置在打开会话页面时清除所有通知：

```
RongConfigCenter.featureConfig().rc_wipe_out_notification_message = true;
```

如果修改 IMKit 默认配置，您可以在应用 res/values 目录下创建 rc_config.xml 文件，将以下变量的值设为 true。该 XML 配置项会同时影响会话页面与会话列表页面的默认行为。

```
<!--进入会话或会话列表界面，是否清除通知栏通知-->
<bool name="rc_wipe_out_notification_message">true</bool>
```

添加自定义头、尾、空布局

提示

IMKit 从 5.1.0 版本开始支持该功能。

IMKit 的 ConversationFragment 支持在消息列表中添加自定义的头（Header）、尾（Footer）和空（Empty）视图。

1. 继承 IMKit 的 ConversationFragment 创建子类。
2. 在子类中实现 addHeaderView()、addFooterView() 和 setEmptyView() 等方法，
3. 在子类的 onViewCreated() 方法中按需调用自身的 addHeaderVew()、addFooterView() 和 setEmptyView() 等方法。

```
@Override
public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
super.onViewCreated(view, savedInstanceState);
addHeaderView(view);
addFooterView(view);
setEmptyView(view);
}
/**
* @param view 自定义列表 header view
*/
public void addHeaderView(View view) {
mAdapter.addHeaderView(view);
}

/**
* @param view 自定义列表 footer view
*/
public void addFooterView(View view) {
mAdapter.addFootView(view);
}

/**
* @param view 自定义列表 空数据 view
*/
public void setEmptyView(View view) {
mAdapter.setEmptyView(view);
}

/**
* @param emptyId 自定义列表 空数据的 LayoutId
*/
public void setEmptyView(@LayoutRes int emptyId) {
mAdapter.setEmptyView(emptyId);
}
```

其他定制化

您可以在以下文档中继续了解如何自定义会话页面：

- 未读消息数
- 修改消息的展示样式
- 表情区域
- 输入区域
- 事件监听

提示

您可以直接参考 IMKit 源码中的 RongConversationActivity.java 和 ConversationFragment.java 了解更多属性和方法。

输入区域　　输入区域

更新时间:2024-08-30

IMKit 的输入区域是在 RongExtension 统一创建和控制的，支持自定义输入模式、自定义扩展区域（插件）、以及自定义表情。

> 提示
>
> 下图输入区从左至右依次是语音/文本切换按钮、内容输入框、表情面板按钮、扩展面板按钮。

修改输入栏组合模式

IMKit 输入栏提供语音/文本切换、内容输入、扩展区域功能，并支持修改输入组合模式。例如，您可以移除语音/文本切换按钮和扩展面板，仅保留内容输入功能。

目前提供了多种排列组合模式：

组合模式	XML 资源	枚举
语音/文本切换功能+内容输入功能+扩展面板	SCE	STYLE_SWITCH_CONTAINER_EXTENSION
语音/文本切换功能+内容输入功能	SC	STYLE_CONTAINER
扩展面板+内容输入功能	EC	---
内容输入功能+扩展面板	CE	STYLE_SWITCH_CONTAINER
内容输入功能	C	STYLE_CONTAINER

如需全局修改，可通过修改全局默认输入模式实现。请复制 rc_conversation_fragment.xml 的全部内容，在应用程序项目的 res/layout/ 下创建同名文件，找到 RongExtension 组件后添加 app:RCStyle="SCE"，更改默认输入显示形式。

如需动态修改，需要继承 IMKit 的 ConversationFragment 实现自定义的会话页面 Activity，在 ConversationFragment 子类中重写 setInputPanelStyle() 方法，该方法接受的枚举值可参见上表。

```
public class ChatFragment extends ConversationFragment {

@Override
public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
super.onViewCreated(view, savedInstanceState);
mRongExtension.getInputPanel().setInputPanelStyle(STYLE_CONTAINER_EXTENSION);

}
}
```

隐藏输入栏中的表情面板按钮

> 提示
>
> IMKit SDK 从 5.3.2 版本开始提供该功能。

在 App 不需要提供表情输入功能时，可通过 IMKit 全局配置隐藏进入表情面板的按钮。

在进入会话页面前隐藏展开表情面板的按钮。

```
RongConfigCenter.featureConfig().setHideEmojiButton(true);
```

输入区域扩展面板

融云将点击输入栏 + 号按钮时展示的面板称为**扩展面板**，**扩展面板**上展示当前可用的插件。

扩展面板插件列表

IMKit 完整插件列表如下所示。扩展面板中默认仅包含内置插件（图库、文件）。如需其他插件，请集成 IMKit 提供的插件库。

所属模块名称	插件名称	说明
内置插件	ImagePlugin	图库插件
内置插件	FilePlugin	文件插件
LocationExtensionModule	DefaultLocationPlugin	仅包括位置功能的插件（未开启实时共享时使用），集成 `locationKit` 库可用。
LocationExtensionModule	CombineLocationPlugin	包括位置和实时位置共享两个功能的插件，集成 `locationKit` 库可用。
SightExtensionModule	SightPlugin	小视频插件，集成 `sight` 库后可用。
ContactCardExtensionModule	ContactCardPlugin	名片插件，集成 `contactcard` 库后可用。
RecognizeExtensionModule	RecognizePlugin	语音输入插件，集成 `recognizer` 库后可用。
RongCallModule	AudioPlugin	语音通话插件，集成 CallKit 后可用。
RongCallModule	VideoPlugin	视频通话插件，集成 CallKit 后可用。

添加语音输入转文字插件

IMKit 基于讯飞 SDK 开发了语音转文字插件库 `recognizer`。IMKit 集成 `recognizer` 库后，在扩展面板会自动生成语音输入功能入口。用户点击语音输入可以将录入的语音转成文字并展示在输入栏。

1. 下载 IMKit 源码，下载后将 `recognizer` 目录拷贝到您应用工程中。

2. 在工程根目录下的 `settings.gradle` 中增加配置。

```
include ':recognizer'
```

3. 在应用的 `build.gradle` 中添加依赖。

```
// 插件版本需要与主 SDK 版本保持一致。
implementation project(path: ':recognizer')
```

4. 在进入会话页之前，通过 `RongExtensionManager`，向 IMKit 的输入区域 `RongExtension` 中注册语音转文字模块 `RecognizeExtensionModule`，该模块会向扩展面板中添加语音输入插件 `RecognizePlugin`。建议在应用生命周期内统一配置。

```
RongExtensionManager.getInstance().registerExtensionModule(new RecognizeExtensionModule());
```

添加其他插件

要了解如何在扩展面板添加其他插件，请参阅以下文档：

- 语音消息
- 小视频消息
- 位置消息
- 名片消息
- 集成 CallKit 通话功能

动态配置扩展面板插件

IMKit 的上述插件模块均实现了 `IExtensionModule` 接口类，通过实现 `getPluginModules`，`getEmoticonTabs` 方法，并通过 `RongExtensionManager` 向 `RongExtension` 进行注册，即可向 IMKit 提供扩展面板插件（`IPluginModule` 实例）、自定义表情页（`IEmoticonTab` 实例）等。

如果应用程序需要动态添加、删除扩展面板插件，或者调整插件位置，您无需实现 `IExtensionModule` 接口类。建议通过创建自定义的扩展面板配置实现这些自定义需求。

1. 继承 `DefaultExtensionConfig`，创建自定义的扩展面板配置类 `MyExtensionConfig`，重写 `getPluginModules()` 方法。您可以增加或删除扩展项，也可以调整各插件的位置。

```
public class MyExtensionConfig extends DefaultExtensionConfig {
@Override
public List<IPluginModule> getPluginModules(Conversation.ConversationType conversationType, String targetId) {
List<IPluginModule> pluginModules = super.getPluginModules(conversationType,targetId);
ListIterator<IPluginModule> iterator = pluginModules.listIterator();

// 删除扩展项
while (iterator.hasNext()) {
IPluginModule integer = iterator.next();
// 以删除 FilePlugin 为例
if (integer instanceof FilePlugin ) {
iterator.remove();
}
}

// 增加扩展项，以添加自定义插件 MyConnectionPlugin 为例
pluginModules.add(new MyConnectionPlugin());
return pluginModules;
}
}
```

2. SDK 初始化之后，调用以下方法设置自定义的输入配置，SDK 会根据此配置展示扩展面板。

```
RongExtensionManager.getInstance().setExtensionConfig(new MyExtensionConfig());
```

自定义扩展面板插件

您可以在 IMKit 的扩展面板中添加自定义插件。自定义插件需要实现 IPluginModule 接口类。您可以参考 IMKit 源码中的 IPluginModule.java，以及具体的实现类。

此处以实现自定义插件 MyConnectionPlugin 为例。

```
public class MyPlugin implements IPluginModule {
/**
* 获取 plugin 图标
*
* @param context 上下文
* @return 图片的 Drawable
*/
@Override
public Drawable obtainDrawable(Context context) {
return context.getResources().getDrawable(R.drawable.my_plugin); //示例代码
}
/**
* 获取 plugin 标题
*
* @param context 上下文
* @return 标题的字符串
*/
@Override
public String obtainTitle(Context context) {
return context.getString(R.string.my_plugin);//示例代码
}
/**
* plugin 被点击时调用。
* 1. 如果需要 Extension 中的数据，可以调用 Extension 相应的方法获取。
* 2. 如果在点击后需要开启新的 activity，可以使用 {@link Activity#startActivityForResult(Intent, int)}
* 或者 {@link RongExtension#startActivityForPluginResult(Intent, int, IPluginModule)} 方式。
* <p/>
* 注意：不要长期持有 fragment 或者 extension 对象，否则会有内存泄露。
*
* @param currentFragment plugin 所关联的 fragment。
* @param extension RongExtension 对象
* @param index plugin 在 plugin 面板中的序号。
*/
@Override
public void onClick(Fragment currentFragment, RongExtension extension, int index) {
Intent intent = new Intent(currentFragment.getActivity(), MyPluginActivity.class);
extension.startActivityForPluginResult(intent, requestCode, MyPlugin.this);
}
/**
* activity 结束时返回数据结果。
* <p/>
* 在 {@link #onClick(Fragment, RongExtension, int)} 中，您可能会开启新的 activity，您有两种开启方式：
* <p/>
* 1. 使用系统中 {@link Activity#startActivityForResult(Intent, int)} 开启方法
* 这就需要自己在对应的 Activity 中接收处理 {@link Activity#onActivityResult(int, int, Intent)} 返回的结果。
* <p/>
* 2. 如果调用了 {@link RongExtension#startActivityForPluginResult(Intent, int, IPluginModule)} 开启方法
* 则在 ConversationFragment 中接收到 {@link Activity#onActivityResult(int, int, Intent)} 后，
* 必须调用 {@link RongExtension#onActivityPluginResult(int, int, Intent)} 方法，RongExtension 才会将数据结果
* 通过 IPluginModule 中 onActivityResult 方法返回。
* <p/>
*
* @param requestCode 开启 activity 时请求码，不会超过 255.
* @param resultCode activity 结束时返回的数据结果.
* @param data 返回的数据.
*/
@Override
public void onActivityResult(int requestCode, int resultCode, Intent data) {

}
}
```

自定义扩展面板插件制作完成后，请参考上文动态配置扩展面板插件，将自定义插件加入扩展面板中。

通过 XML 资源自定义 UI

您可以通过修改对应的布局文件来调整界面布局、字体大小、颜色及背景色等。请勿删减 SDK 默认控件，不要随意修改 View 的 ID。

IMKit 资源文件	位置	说明
rc_extension_board.xml	res/layout	输入框布局文件。它是整个输入框的容器。复制 IMKit 源码文件的全部内容，在项目下创建 `/res/layout` 下创建同名文件。
rc_extension_input_panel.xml	res/layout	输入栏布局文件。复制 IMKit 源码文件的全部内容，在项目下创建 `/res/layout` 下创建同名文件。

以上 XML 资源也可从 官网 SDK 下载 页面获取。

修改消息的展示样式　　修改消息的展示样式

更新时间:2024-08-30

IMKit SDK 通过「消息展示模板」控制会话页面中消息的展示样式。App 可以按需修改指定类型消息的展示模板，实现对消息展示样式的个性化配置。

- SDK 默认为会话页面中需要展示的内置消息类型（详见消息类型概述）提供了展示模板，App 可以按需创建模板，替换默认模板。
- App 创建的自定义消息类型（详见自定义消息类型）默认没有对应的消息展示模板。如果 App 需要在会话界面中展示该自定义类型的消息，则必须创建对应的消息展示模板，提供给 SDK。

> 提示
>
> App 如需在会话界面中展示自定义类型的消息，必须创建对应的消息展示模板，否则 SDK 无法正常展示该类型消息。

通过样式资源修改消息 UI

您可以通过替换 IMKit 自带样式资源，调整 SDK 内置消息的展示背景图，文字颜色和字体大小，适用于需要轻度自定义会话界面的场景。

替换消息背景图

IMKit 会话页面中每条消息都有气泡背景，蓝色气泡为发送的消息，白色气泡为接收的消息。

IMKit 在消息展示模版基类 `BaseMessageItemProvider` 中引用了两个 9-patch 文件。您可以在应用程序的 `res/drawable-xhdpi/` 目录下创建同名文件，替换默认的两个 `.9.png` 的图片资源来更改消息气泡的展示效果。

资源路径与名称	说明
res/drawable-xhdpi/rc_ic_bubble_right.9.png	右侧气泡，即发送消息的展示气泡
res/drawable-xhdpi/rc_ic_bubble_left.9.png	左侧气泡，即接受消息的展示气泡

修改内置文本消息的字体颜色或字体大小

您可以自定义 IMKit 中内置文本消息的字体颜色或字体大小。

从 IMKit 源码中复制 rc_translate_text_message_item.xml 文件到应用程序目录的 `res/layout` 目录下，修改该布局文件里的字体大小和颜色修改为自定义值。

> 提示
>
> 如果 SDK 版本 < 5.2.2，不能使用 `rc_translate_text_message_item.xml` 文件，请复制 rc_text_message_item.xml。

替换内置消息默认展示模板

IMKit 为每种类型的消息都封装了对应的消息展示模板。所有的消息展示模板都继承自 `BaseMessageItemProvider`。

如果您需要更改 SDK 内置消息的展示效果，且自定义需求较高，可以继承 `BaseMessageItemProvider`，创建新的消息展示模板，并将该自定义模板提供给 SDK，替换 SDK 默认模板。

内置默认消息展示模板

消息类型	模板类名（附源码链接）
文本消息 TextMessage	TextMessageItemProvider.class
图片消息 ImageMessage	ImageMessageItemProvider.class
语音消息 HQVoiceMessage	HQVoiceMessageItemProvider.class
文件消息 FileMessage	FileMessageItemProvider.class
位置消息 LocationMessage	LocationMessageItemProvider.class
实时位置共享消息 RealTimeLocationStartMessage	RealTimeLocationMessageItemProvider.class
小视频消息 SightMessage	SightMessageItemProvider.class
图文消息 RichContentMessage	RichContentMessageItemProvider.class
动图消息 GIFMessage	GIFMessageItemProvider.class

消息类型	模板类名（附源码链接）
合并转发消息 CombineMessage	CombineMessageItemProvider.class
小灰条消息 InformationNotificationMessage	InformationNotificationMessageItemProvider.class
群组通知消息 GroupNotificationMessage	GroupNotificationMessageItemProvider.class

步骤 1：创建自定义模板

此处以自定义文本消息的展示模板为例说明：

1. 创建自定义的文本消息展示布局文件 my_text_message.xml。

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/my_text"
style="@style/TextStyle.Alignment"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:padding="12dp"
android:textColor="@android:color/holo_green_dark"
android:textColorLink="@android:color/holo_red_dark"
android:fontFamily="sans-serif"
android:textSize="14sp"
android:maxWidth="223dp" />
```

2. 创建自定义模板 MyTextMessageProvider，继承 BaseMessageItemProvider，指定 <> 中的消息类型为 TextMessage，并根据 Android Studio 的提示，实现所有抽象方法。

```
public class MyTextMessageProvider extends BaseMessageItemProvider<TextMessage> {
public MyTextMessageProvider() {
mConfig.centerInHorizontal = true; // 配置展示属性，该消息居中显示。
}
/**
* 根据自定义布局文件生成 ViewHolder
*/
@Override
protected ViewHolder onCreateMessageContentViewHolder(ViewGroup viewGroup, int i) {
View textView = LayoutInflater.from(viewGroup.getContext()).inflate(R.layout.my_text_message, viewGroup, false);
return new ViewHolder(viewGroup.getContext(), textView);
}

/**
* 根据消息内容，设置布局文件里各控件的值。
*/
@Override
protected void bindMessageContentViewHolder(ViewHolder viewHolder, ViewHolder viewHolder1, TextMessage textMessage, UiMessage uiMessage, int i, List<UiMessage> list,
IViewProviderListener<UiMessage> iViewProviderListener) {
TextView textView = viewHolder.getView(R.id.my_text);
textView.setText(textMessage.getContent());
}

/**
* 自定义布局里各控件点击时会回调此方法，可以在这里实现点击逻辑。
* 此处忽略处理，没有处理点击事件。
*/
@Override
protected boolean onItemClick(ViewHolder viewHolder, TextMessage textMessage, UiMessage uiMessage, int i, List<UiMessage> list, IViewProviderListener<UiMessage>
iViewProviderListener) {
return false;
}

/**
* 根据消息类型，返回是否为本模板需要展示的消息类型。
* 此处示例代表本模板仅处理文本类型的消息。
*/
@Override
protected boolean isMessageViewType(MessageContent messageContent) {
return messageContent instanceof TextMessage;
}

/**
* 当该类型消息为会话最后一条消息时，需要在会话列表的会话里展示此消息的描述，该方法返回描述内容。
* 此处以返回文本消息的具体内容为例。
*/
@Override
public Spannable getSummarySpannable(Context context, TextMessage textMessage) {
return new SpannableString(AndroidEmoji.ensure(textMessage.getContent()));
}
}
```

步骤 2：替换默认模板

创建自定义模板后，您需要调用如下方法替换掉 SDK 默认的消息展示模板。替换后，SDK 在渲染消息时，会自动调用该类型消息的自定义模板进行渲染。

- **代码示例**

  此处以自定义模板 MyTextMessageProvider 为例。

```
RongConfigCenter.conversationConfig().replaceMessageProvider(TextMessageItemProvider.class, new MyTextMessageProvider())
```

- **替换方法：**

```
RongConfigCenter.conversationConfig().replaceMessageProvider(Class oldProviderClass, IMessageProvider provider)
```

参数	说明
oldProviderClass	SDK 默认消息展示模板的类名，参考内置默认消息展示模板。
provider	自定义的消息展示模板对象。

步骤 3：配置模板属性

IMKit 在消息展示模版基类 `BaseMessageItemProvider` 中使用了消息展示配置类 `MessageItemProviderConfig` 控制消息模板的部分样式属性。您可以在 `BaseMessageItemProvider` 子类模板的构造方法中，直接获取消息展示配置对象 `mConfig`，并修改 `mConfig` 中以下属性值：

消息展示配置属性	描述	默认值
showPortrait	是否显示头像。	true
centerInHorizontal	是否将消息内容横向居中。	false
showWarning	是否显示未发送成功警告。	true
showProgress	是否显示发送进度。	true
showSummaryWithName	是否在会话的内容体里显示发送者名字。	true
showReadState	单聊会话中是否支持在消息旁边显示已读回执状态。	false
showContentBubble	是否需要展示消息气泡。	true

例如，在以下示例中，我们可修改属性默认值，将使用该模板的消息设置为居中显示，并支持展示单聊已读回执状态图标：

```
public MyCustomMessageProvider() {
mConfig.showReadState = true; // 单聊会话中是否支持在消息旁边显示已读回执状态。 默认不支持。
mConfig.centerInHorizontal = true; // 配置展示属性，该消息居中显示。
}
```

页面事件监听 页面事件监听

更新时间:2024-08-30

IMKit 支持监听会话列表页面和会话页面上的点击、长按事件，应用程序可以相应的方法中拦截并实现自定义需求。

监听会话列表页面事件

IMKit 提供了会话列表监听器 ConversationListBehaviorListener，可监听会话列表中针对会话 item 和会话头像的长按与点击事件。

使用 RongIM 或 IMCenter 的 setConversationListBehaviorListener 方法设置监听器。

```
RongIM.setConversationListBehaviorListener(listener);
```

长按会话事件

在会话列表中长按会话 item 时触发以下方法。SDK 默认弹出菜单选项。

```
boolean onConversationLongClick(Context context, View view, BaseUiConversation conversation)
```

参数	类型	说明
context	Context	上下文
view	View	触发点击的 View
conversation	BaseUiConversation	长按的会话

如果需要自定义处理此长按事件，返回 true；否则返回 false，继续执行 SDK 默认逻辑。

点击会话事件

在会话列表中点击会话 item 时触发以下方法。触发该点击事件时，SDK 默认跳转逻辑如下：

- 如果是聚合会话，跳转到聚合会话列表页面。
- 如果是非聚合会话，跳转到会话页面。

```
boolean onConversationClick(Context context, View view, BaseUiConversation conversation)
```

参数	类型	说明
context	Context	上下文
view	View	触发点击的 View
conversation	BaseUiConversation	点击的会话

如果需要自定义处理此长按事件，返回 true；否则返回 false，继续执行 SDK 默认逻辑。

点击会话头像事件

点击会话头像（图标）时触发以下方法。SDK 没有默认不处理该事件。

```
boolean onConversationPortraitClick(Context context, Conversation.ConversationType conversationType, String targetId)
```

参数	类型	说明
context	Context	上下文
conversationType	Conversation.ConversationType	会话类型
targetId	String	会话 Id

如果需要自定义处理此点击事件，返回 true。否则返回 false，不处理该事件。

长按会话头像的事件

长按会话头像（图标）时触发以下方法。SDK 默认不处理该事件，直接将结果返回给系统。

```
boolean onConversationPortraitLongClick(Context context, Conversation.ConversationType conversationType, String targetId)
```

参数	类型	说明
context	Context	上下文
conversationType	Conversation.ConversationType	会话类型
targetId	String	会话 Id

如果需要自定义处理此长按事件，请返回 `true`。否则返回 `false`，不处理该事件。

监听会话页面事件

IMKit 提供了会话列表监听器 ConversationClickListener，可监听会话页面中针对消息 item 和消息头像的长按与点击事件。

使用 RongIM 或 IMCenter 的 `setConversationClickListener` 方法设置监听器。

```
IMCenter.setConversationClickListener(listener);
```

点击消息事件

```
boolean onMessageClick(Context context, View view, Message message);
```

参数	类型	说明
context	Context	上下文
view	View	触发点击的 View
message	Message	被点击的消息的实体信息

如果用户自己处理了点击后的逻辑处理，则返回 `true`；否则返回 `false`，继续执行 SDK 的默认逻辑。

长按消息事件

```
boolean onMessageLongClick(Context context, View view, Message message);
```

参数	类型	说明
context	Context	上下文
view	View	触发点击的 View
message	Message	被点击的消息的实体信息

如果用户自己处理了长按后的逻辑处理，则返回 `true`；否则返回 `false`，继续执行 SDK 默认逻辑。

点击消息上的用户头像事件

```
boolean onUserPortraitClick(
Context context,
Conversation.ConversationType conversationType,
UserInfo user,
String targetId);
```

参数	类型	说明
context	Context	上下文
conversationType	Conversation.ConversationType	会话类型
user	UserInfo	被点击的用户的信息
targetId	String	会话 Id

如果自己处理了点击后的逻辑处理，则返回 `true`；否则返回 `false`，继续执行 SDK 默认逻辑。

长按消息上的用户头像事件

长按消息上的用户头像（图标）时触发以下方法。SDK 默认跳转到 @功能的选择用户界面。

```
boolean onUserPortraitLongClick(
Context context,
Conversation.ConversationType conversationType,
UserInfo user,
String targetId);
```

参数	类型	说明
context	Context	上下文
conversationType	Conversation.ConversationType	会话类型
user	UserInfo	被点击的用户的信息

参数	类型	说明
targetId	String	会话 Id

如果用户自己处理了点击后的逻辑处理，则返回 true；否则返回 false，继续执行 SDK 的默认逻辑。

点击消息超链接事件

```
boolean onMessageLinkClick(Context context, String link, Message message);
```

参数	类型	说明
context	Context	上下文
link	String	被点击的链接
message	Message	被点击的消息的实体信息

如果用户自己处理了点击后的逻辑处理，则返回 true；否则返回 false，继续执行 SDK 默认逻辑。

点击已读回执状态事件

```
boolean onReadReceiptStateClick(Context context, Message message);
```

参数	类型	说明
context	Context	上下文
message	Message	被点击的消息的实体信息

如果用户自己处理了长按后的逻辑处理，则返回 true；否则返回 false，继续执行 SDK 默认逻辑。

点击常用语按钮事件

> 提示
>
> 要求 IMKit 版本 ≧ 5.6.3。

如果启用了 IMKit 快捷回复 功能，用户在会话页面点击常用语按钮后会弹出快捷回复。如需拦截该点击事件，返回 true，您可以自定义点击常用语按钮后的逻辑；否则返回 false，继续执行 SDK 默认逻辑。

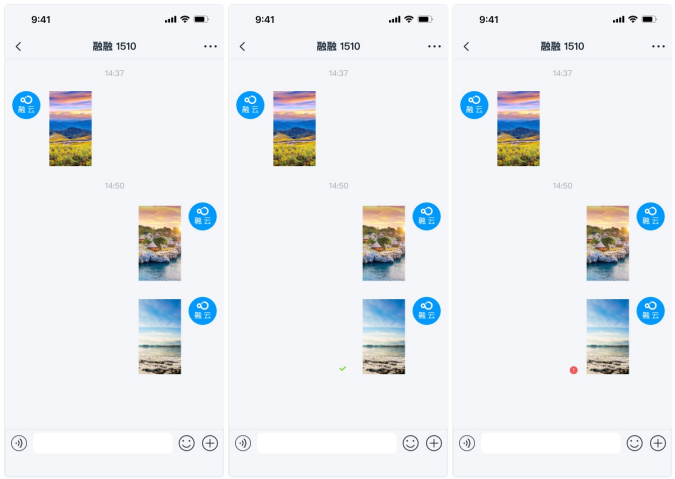
```
default boolean onQuickReplyClick(Context context) {
return false;
}
```

图片和 GIF 消息 图片和 GIF 消息

更新时间:2024-08-30

用户可以通过 IMKit 内置的图片插件发送图片消息和 GIF 消息。消息将出现在会话页面的消息列表组件中。SDK 默认发送消息包含以下消息内容对象：

- 图片消息内容类为 ImageMessage（类型标识：`RC:ImgMsg`）
- GIF 消息内容类为 GIFMessage（类型标识：`RC:GIFMsg`）



局限

- 仅支持发送本地图片和 GIF。
- GIF 文件大小上限为 2 MB。
- 图片消息和 GIF 消息中的文件默认会上传到融云的服务器。如需上传到自己的服务器，您需要拦截消息，自行上传。详见拦截消息。

用法

扩展面板里默认带有发送图片消息入口，由 IMKit 内置的 `ImagePlugin` 实现。用户点击输入栏右侧 + 号按钮可展开扩展面板，点击图片图标，即可打开本地相册，选择图片、GIF 文件进行发送。您可以参考 IMKit 源码中的 ImagePlugin.java。

定制化

修改默认文件保存位置

客户端接收图片消息和 GIF 消息后，在会话页面中长按保存时，SDK 默认保存到 `/RongCloud/Image/` 下。您可以在应用程序目录下创建 `res/values/rc_config` 文件，全局修改默认保存位置：

```
<string name="rc_image_default_saved_path">/RongCloud/Image/</string>
```

调整图片压缩质量

在发送前，图片会被压缩质量，以及生成缩略图，在聊天界面中展示。GIF 无缩略图，也不会被压缩。

- **图片消息的缩略图**：SDK 会以原图 30% 质量生成符合标准大小要求的大图后再上传和发送。压缩后最长边不超过 240 px。缩略图用于在聊天界面中展示。
- **图片**：发送消息时如未选择发送原图，SDK 会以原图 85% 质量生成符合标准大小要求的大图后再上传和发送。压缩后最长边不超过 1080 px。

一般情况下不建议修改 SDK 默认压缩配置。如需调整 SDK 压缩质量，详见知识库文档如何修改 SDK 默认的图片与视频压缩配置。

自定义图片、GIF 消息的 UI

图片消息与 GIF 消息默认使用以下模板展示在消息列表中。

- `ImageMessageItemProvider`
- `GIFMessageItemProvider`

如果需要调整内置消息样式，建议自行实现消息展示模板类，并将该自定义模板提供给 SDK。所有消息模板都继承自 `BaseMessageItemProvider<CustomMessage>`，自定义消息展示模板也需要继承 `BaseMessageItemProvider<CustomMessage>`。详见修改消息的展示样式。

您也可以直接替换消息展示模板中引用的样式资源、字符串资源和图标资源。详见 IMKit 源码 ImageMessageItemProvider.java 、GIFMessageItemProvider.java 。

隐藏扩展面板中的图片入口

IMKit 默认在扩展面板中启用了图片入口。如需动态修改，可创建自定义的扩展面板配置类 `MyExtensionConfig`，继承自 `DefaultExtensionConfig`，重写其中的 `getPluginModules()` 方法。详见输入区域。

小视频消息　　小视频消息

更新时间:2024-08-30

用户可以通过 IMKit 图库（本地相册）或小视频插件发送小视频消息。消息将出现在会话页面的消息列表组件中。插件默认发送的消息包含小视频消息内容对象 SightMessage（类型标识：RC:SightMsg）。

局限性

小视频功能目前存在以下限制：

- IMKit 仅单聊会话和群聊会话支持发送小视频消息。
- 如果使用小视频插件进行录制，支持录制长度不超过 10 秒的小视频。
- 如果从本地相册中选择视频文件，请注意服务端的默认视频时长上限为 2 分钟。如需调整上限，请联系商务。
- 仅支持 `H.264` + `AAC` 编码的视频文件，因为 IMKit 的短视频录制、播放只实现了该编码组合的支持。
- 如果 App Key 使用 **IM 旗舰版**或 **IM 尊享版**，文件存储时长默认为 180 天（不含小视频文件，小视频文件存储 7 天）。注意，**IM 商用版（已下线）默认存储 7 天。如需了解IM 旗舰版或 IM 尊享版**的具体功能与费用，请参见融云官方价格说明页面及计费说明。

用法

建议通过集成 IMKit 小视频插件使用小视频消息功能。

集成小视频插件

IMKit 的小视频插件实现了小视频消息的消息注册、录制、播放等功能。集成小视频模块后，在单聊、群组会话输入区域的扩展面板中自动出现发送小视频消息的入口。

1. 在应用的 `build.gradle` 文件中增加对小视频的依赖。

```
dependencies{
// 插件版本需要与主 SDK 版本保持一致。
api 'cn.rongcloud.sdk:sight:x.y.z'
}
```

> 提示
>
> 各个 SDK 的最新版本号可能不相同，还可能是 x.y.z.h，可前往 融云官网 SDK 下载页面 或 融云的 Maven 代码库 查询。

2. 通过 RongExtensionManager，向 IMKit 的输入区域 RongExtension 中注册小视频模块 SightExtensionModule。小视频模块会向 IMLib 注册 SightMessage，并向扩展面板中添加小视频插件 SightPlugin。

```
RongExtensionManager.getInstance().registerExtensionModule(new SightExtensionModule());
```

从本地相册选择小视频

> 提示

> 前提条件：集成 IMKit 小视频插件后，插件内部会向 IMLib 注册（SightMessage）类型的消息。请先集成 IMKit 小视频插件，再进行以下配置，否则 SDK 无法识别小视频消息。

通过 IMKit 输入区域中的图库插件 `ImagePlugin` 打开本地相册时，默认不包含视频文件，用户无法选择视频文件进行发送。

您可以修改全局配置，设置在本地相册中包含视频文件。

```
RongConfigCenter.featureConfig().rc_media_selector_contain_video = true;
```

如果通过 XML 资源修改 IMKit 默认配置，可在应用 `res/values` 目录下创建 `rc_config.xml` 文件，添加以下配置：

```
<bool name="rc_media_selector_contain_video">true</bool>
```

如果您的项目中不使用 IMKit 小视频插件，但仍希望支持从本地发送小视频文件，请自行向 IMLib 注册 `SightMessage`，否则 SDK 无法发送小视频消息。您还需要自行实现小视频录制、播放（在会话页面消息点击事件中处理播放）功能。

```
// 注册 SightMessage，必须在连接之前处理
RongIMClient.registerMessageType(SightMessage.class);
```

发送小视频消息

用户点击输入栏右侧 + 号按钮可展开扩展面板，点击**图片**或**小视频**图标，即可发送小视频消息。

定制化

修改小视频文件保存位置

客户端接收小视频消息后，在会话页面中长按保存时，SDK 默认保存到 `/RongCloud/Video/` 下。您可以在应用程序目录下创建 `res/values/rc_config` 文件，全局修改默认保存位置：

```
<string name="rc_file_default_saved_path">/RongCloud/Video/</string>
```

调整小视频压缩质量

小视频文件会被压缩为分辨率 `544 * 960` 的文件。小视频首帧画面会被用于生成缩略图，在聊天界面中展示。SDK 默认以原图 30% 质量生成符合标准大小要求的缩略图后再上传和发送，缩略图最长边不超过 240 px。

一般情况下不建议修改 SDK 默认压缩配置。如需调整 SDK 压缩质量，详见知识库文档如何修改 SDK 默认的图片与视频压缩配置。

自定义小视频消息的 UI

IMKit 默认生成和发送小视频消息（`RC:SightMsg`），使用 `SightMessageItemProvider` 模板展示在消息列表中。

所有消息展示模板都继承自 `BaseMessageItemProvider<CustomMessage>`，您可以继承 `BaseMessageItemProvider<CustomMessage>`，自行实现一个小视频消息展示模板类，并将该自定义模板提供给 SDK。

您也可以直接替换小视频消息展示模板中引用的样式资源、字符串资源和图标资源。详见 IMKit 源码 SightMessageItemProvider.java 中引用的资源。

例如：您可以复制 IMKit 源码中的 rc_item_sight_message.xml 的全部内容，您可以在项目下创建 `/res/layout/rc_item_sight_message.xml` 文件，修改其中定义的样式值。请勿删减 SDK 默认控件，不要随意修改 View 的 ID。

如果希望修改录制、播放 UI，可以参考 IMKit 小视频插件源码 Sight。

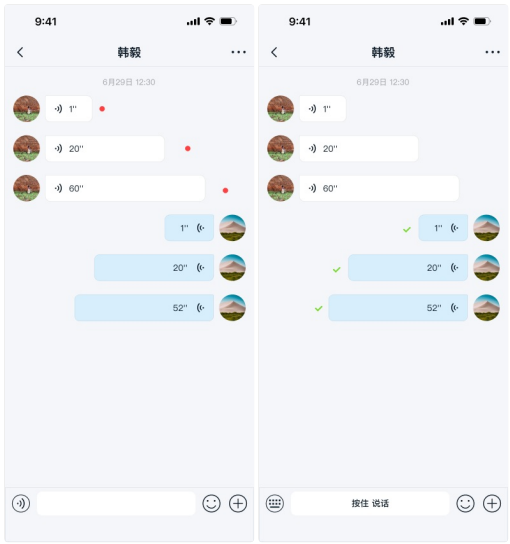
动态隐藏小视频入口

如需动态隐藏输入区域扩展面板中的小视频功能入口，可创建自定义的扩展面板配置类 `MyExtensionConfig`，继承自 `DefaultExtensionConfig`，重写其中的 `getPluginModules()` 方法。详见输入区域。

语音消息 语音消息

更新时间:2024-08-30

用户可以通过 IMKit 内置的输入组件录制并发送语音消息。消息将出现在会话页面的消息列表组件中。SDK 默认生成和发送的消息包含高清语音消息内容对象 HQVoiceMessage（类型标识：RC:HQVcMsg）。



局限性

语音输入功能目前存在以下限制：

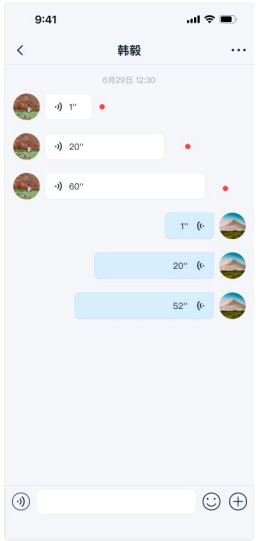
- IMKit 仅单聊会话和群聊会话支持发送语音消息。
- 用户必须录制至少为 1 秒的音频内容，且必须短于 60 秒钟。
- 用户在录制语音消息时无法暂停。
- 正在视频通话和语音通话中不能进行语音消息发送。

用法

IMKit 默认在输入栏（InputPanel）组件中启用语音消息输入功能。输入栏中默认带有切换语音输入按钮。

发送语音消息

为了发送语音消息，用户必须首先在 InputPanel 中通过 AudioRecordManager 录制消息。默认情况下，语音消息图标显示在输入字段的左侧。点击此图标后，就会出现录制按钮（“按住说话”）。用户可以通过点击录制按钮来录制语音消息。长度必须至少为一秒且短于 60 秒钟。如果在点击停止按钮之前消息不到一秒，则不会保存该消息。录制过程中可以上滑取消录制或放弃取消。一旦松开按钮，SDK 默认发送到目前为止录制的内容。不支持在发送语音消息之前预览。在播放语音消息中的音频文件时不可暂停。

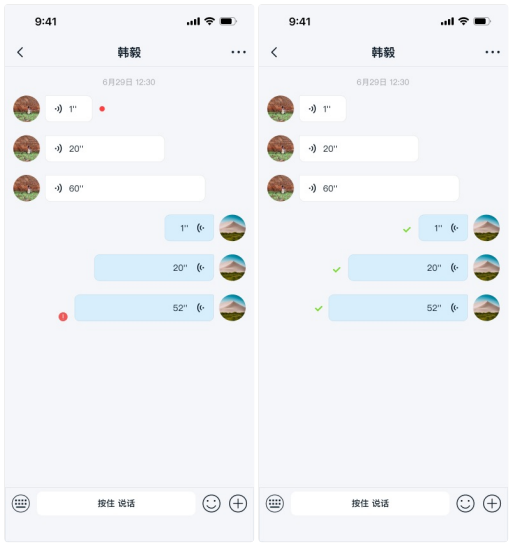


消息列表中的语音消息

您可以在单聊会话、群聊会话、系统会话中接收语音消息。语音消息显示在消息列表中。

用户可以通过点击播放按钮查看和播放语音消息。未播放的语音消息旁边会显示一个红点，消息可多次播放。但是，用户只能在客户端应用程序中收听语音消息，并且无法将其保存到自己的设备中。您

在频道中一次只能收听一条音频文件。如果您在收听消息时尝试播放另一条消息，则先播放的消息将暂停。



IMKit 默认下载高清语音消息，并且默认点击播放后连续播放消息下方未收听的语音消息。您可以修改全局配置，设置未听的语音消息不连续播放。

```
RongConfigCenter.featureConfig().rc_play_audio_continuous = false;
```

IMKit 在线时默认自动下载高清语音消息。您可以通过全局配置禁用该行为：

```
RongConfigCenter.featureConfig().rc_enable_automatic_download_voice_msg = false;
```

如果修改 IMKit 默认配置，您可以在应用 `res/values` 目录下创建 `rc_config.xml` 文件，添加以下配置项：

```
<bool name="rc_play_audio_continuous">true</bool>
<bool name="rc_enable_automatic_download_voice_msg">true</bool>
```

定制化

语音消息的定制化涉及到输入栏的语音输入切换图标、音频录制界面、和高清语音消息展示 UI。

自定义语音消息的 UI

IMKit 默认生成和发送高清语音消息（`RC:HQVoiceMsg`），使用 `HQVoiceMessageItemProvider` 模板展示在消息列表中。

所有消息展示模板都继承自 `BaseMessageItemProvider<CustomMessage>`，您可以继承 `BaseMessageItemProvider<CustomMessage>`，自行实现一个高清语音消息展示模板类，并将该自定义模板提供给 SDK。

您也可以直接替换高清语音消息展示模板中引用的样式资源、字符串资源和图标资源。详见 IMKit 源码 HQVoiceMessageItemProvider.java 中引用的资源。

例如：您可以复制 IMKit 源码中的 rc_item_hq_voice_message.xml 的全部内容，您可以在项目下创建 `/res/layout/rc_item_hq_voice_message.xml` 文件，修改其中定义的样式值。请勿删减 SDK 默认控件，不要随意修改 View 的 ID。

输入栏样式资源

要自定义输入栏样式，您可以复制 IMKit 源码中的 rc_extension_input_panel.xml 的全部内容，在项目下创建 `/res/layout/rc_extension_input_panel.xml` 文件，修改其中定义的样式值。请勿删减 SDK 默认控件，不要随意修改 View 的 ID。

录音弹窗样式资源

开始录制后，屏幕中央会出现录音弹窗。要自定义语音消息录音弹窗中的样式，您可以复制 IMKit 源码中的 rc_voice_record_popup.xml 的全部内容，您可以在项目下创建 `/res/layout/rc_voice_record_popup.xml` 文件，修改其中定义的样式值。请勿删减 SDK 默认控件，不要随意修改 View 的 ID。

图标资源

下表显示了可自定义的图标。

图标	图像	描述
rc_ext_toggle_voice		用作语音输入按钮的图标，用于在消息输入时切换至语音消息录制视图。
rc_voice_send_play3		本端用户发送的高清语音消息气泡中的图标。播放语音消息时，该图标与 `rc_voice_send_play1` 和 `rc_voice_send_play2` 连续播放形成动效。
rc_voice_receive_play3		本端用户接收的高清语音消息气泡中的图标。播放语音消息时，与 `rc_voice_receive_play1` 和 `rc_voice_receive_play2` 连续播放形成动效。
rc_voice_hq_message_download_error		下载语音消息出错时的图标。

隐藏切换语音输入按钮

如需全局修改，可通过修改全局默认输入模式实现。在应用程序项目下创建 res/values/rc_conversation_fragment.xml，找到 RongExtension 组件 app:RCStyle="SCE"，更改默认输入显示形式。

如需动态修改，需要通过继承 IMKit 的 ConversationFragment，通过在子类中重写 setInputPanelStyle() 方法实现。这种方式要求在 IMKit 中注册和使用自定义的会话页面 Activity，详见集成会话界面。

```
public class ChatFragment extends ConversationFragment {

@Override
public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
super.onViewCreated(view, savedInstanceState);
mRongExtension.getInputPanel().setInputPanelStyle(STYLE_CONTAINER_EXTENSION);

}
}
```

文件消息　　文件消息

更新时间:2024-08-30

用户可以通过 IMKit 内置的文件插件发送文件消息。消息将出现在会话页面的消息列表组件中。文件插件默认发送的消息包含文件消息内容对象 FileMessage（类型标识：RC:FileMsg）

局限

- 仅支持发送本地文件。
- 文件消息中的文件默认会上传到融云的服务器。如需上传到自己的服务器，您需要拦截消息，自行上传。详见拦截消息。
- 不支持在 IMKit 中预览文件，请在 UI 中选择用其他应用打开。

用法

IMKit 内置的 FilePlugin 实现了扩展面板中的文件消息功能。您也可以参考 IMKit 源码中的 FilePlugin.java。

发送文件消息

扩展面板里默认带有发送文件消息入口。用户点击输入栏右侧 + 号按钮可展开扩展面板，点击文件图标，即可发送文件消息。

定制化

修改默认文件保存位置

客户端接收文件消息后，在会话页面中长按保存时，SDK 默认保存到 /RongCloud/File/ 下。您可以在应用程序目录下创建 res/values/rc_config 文件，全局修改默认保存位置：

```
<string name="rc_file_default_saved_path">/RongCloud/File/</string>
```

替换文件消息默认的文件图标

文件消息（FileMessage）在会话界面中显示时，会根据消息携带的文件类型展示匹配的图标。SDK 默认为以下类型的文件提供了匹配的图标，如果为其他类型文件，则默认显示统一的默认图标。

- **图片类**：bmp、cod、gif、ief、jpe、jpeg、jpg、jfif、svg、tif、tiff、ras、ico、pbm、pgm、png、pnm、ppm、xbm、xpm、xwd、rgb
- **文本类**：txt、log、html、stm、uls、bas、c、h、rtx、sct、tsv、htt、htc、etx、vcf
- **视频类**：rmvb、avi、mp4、mp2、mpa、mpe、mpeg、mpg、mpv2、mov、qt、lsf、lsx、asf、asr、asx、avi、movie、wmv
- **音频类**：mp3、au、snd、mid、rmi、aif、aifc、aiff、m3u、ra、ram、wav、wma
- **Word 类**：doc、dot、docx
- **Excel 类**：xla、xlc、xlm、xls、xlt、xlw、xlsx

SDK 从 5.3.4 版本开始，支持 App 修改文件类型（扩展名）对应显示的图标。App 可以按需更新指定图标，替换全部图标，或增加文件类型（扩展名）及图标。

建议在初始化前调用。

```
RongConfigCenter.conversationConfig().registerFileSuffixTypes(map);
```

参数	类型	说明
map	HashMap<String, Integer>	文件消息图标配置。key 是不带 . 的文件扩展名（例：png、pdf）。value 是 Android 资源 ID，App 需要把图标添加到 drawable 资源目录中。如果要替换统一默认文件图标，key 使用 default 进行配置。

自定义文件消息的 UI

文件消息使用 FileMessageItemProvider 模板展示在消息列表中。如果需要调整内置消息样式，建议自行实现消息展示模板类，并将该自定义模板提供给 SDK。所有消息模板都继承自 BaseMessageItemProvider<CustomMessage>，自定义消息展示模板也需要继承 BaseMessageItemProvider<CustomMessage>。详见修改消息的展示样式。

您也可以直接替换文件消息展示模板中引用的样式资源、字符串资源和图标资源。详见 IMKit 源码 FileMessageItemProvider.java 中引用的资源。

隐藏文件消息入口

IMKit 默认在扩展面板中启用了文件消息入口。如需动态修改，可创建自定义的扩展面板配置类 MyExtensionConfig，继承自 DefaultExtensionConfig，重写其中的 getPluginModules() 方法。详见输入区域扩展面板。

位置消息　　位置消息

更新时间:2024-08-30

IMKit 基于高德地图 SDK 提供了位置消息，实现了应用内位置共享、位置缩略图与地图预览功能。

- SDK 默认发送的消息包含位置消息内容对象 `LocationMessage`（类型标识：`RC:LBSMsg`）。
- 实时位置共享也是基于消息实现的。SDK 默认使用类型标识为 `RC:RL`、`RC:RLStart`、`RC:RLJoin`、`RC:RLQuit` 的消息。

> 提示
> IMKit 默认会话页面未启用位置功能。如需要使用位置功能，可集成 IMKit 位置插件并配置您自己的高德地图 SDK 帐号。

局限

- IMKit 的位置插件目前存在仅支持高德地图 SDK。如需使用其他地图服务，您可以自定义插件，自行构造位置消息并发送。添加自定义插件的方法详见输入区域。

用法

IMKit 从 5.2.3 及之后开始支持 `locationKit` 插件。如果从低于 5.2.3 的 IMKit 版本升级，请参见下文升级旧版位置插件。

> 重要
> IMKit 5.2.3 - 5.6.6 版本存在已知问题，无法正常发送位置消息。建议您尽快升级到 5.6.7 及之后版本。如果暂时无法升级，可参考知识库文档中的修复方案，详见 解决内置高德地图 ApiKey 失效，导致无法正常发送位置消息的问题。

自行申请高德地图 API Key

IMKit 使用 `locationKit` 插件发送位置消息时，需要调用高德地图的创建静态图接口，因此您需要申请高德地图申请 Web 服务 API 密钥（Key）。在高德平台创建一个 Web 服务后，可生成 API Key。

集成位置插件

1. 集成位置插件库：

本地库文件：从官网下载高德位置相关的库文件。将下载包里 `LocationLib` 文件夹下的 `locationKit_<ver>.aar` 文件拷贝到项目的 `libs` 目录下，并在应用的 `build.gradle` 中添加依赖。


```
dependencies {
implementation fileTree(include: ['*.jar', '*.aar'], dir: 'libs')
...
}
```

Maven：

```
// 插件版本需要与主 SDK 版本保持一致。
implementation 'cn.rongcloud.sdk:locationKit:x.y.z'
```

> 提示
>
> 各个 SDK 的最新版本号可能不相同，还可能是 x.y.z.h，可前往 融云官网 SDK 下载页面 或 融云的 Maven 代码库 查询。

2. 打开应用的 `AndroidManifest.xml` 文件，把从高德官网获取的 ApiKey 添加到 application 标签的 `meta-data` 里。

```
<meta-data
android:name="com.amap.api.v2.apikey"
android:value="高德地图的 ApiKey" />
```

3. 添加高德隐私协议代码。

   集成 `locationKit` 需要遵守高德地图 隐私合规接口说明，否则会导致应用崩溃。

   请在您当前应用的隐私协议中添加高德平台隐私政策，在获取用户同意之后，并在第一次调用地图插件功能之前调用以下代码：

```
// 表示用户同意定位功能隐私协议
AMapLocationClient.updatePrivacyShow(this, true, true);
AMapLocationClient.updatePrivacyAgree(this, true);

// 表示用户同意地图功能隐私协议
MapsInitializer.updatePrivacyShow(this, true, true);
MapsInitializer.updatePrivacyAgree(this, true);

// 表示用户同意搜索功能隐私协议
ServiceSettings.updatePrivacyShow(this, true, true);
ServiceSettings.updatePrivacyAgree(this, true);
```

   通过以上步骤，即完成了高德 3D 地图的集成，在扩展面板里会自动生成位置插件。

4. （可选）配置代码混淆。详见混淆。

发送位置消息

位置插件集成完毕后，在扩展面板里会自动生成位置消息入口。用户点击输入栏右侧 + 号按钮可展开扩展面板，点击位置图标，即可发送位置消息。

集成位置插件后，默认能力如下：

- 单聊会话中点击位置插件时，会有 **发送位置**和**实时位置共享**两个功能选项。用户可以选择发送实时位置或发起位置共享。您也可以为单聊会话关闭实时位置共享。
- 群聊会话中默认没有开启 **实时位置共享**，仅可以发送位置。您可以自行为群聊会话启用该选项。

使用实时位置共享

您可以创建自定义的扩展面板配置，修改是否需要启用实时位置共享。

1. 创建自定义的扩展面板配置类 `MyExtensionConfig`，继承自 `DefaultExtensionConfig`，重写其中的 `getPluginModules()` 方法。

```
public class MyExtensionConfig extends DefaultExtensionConfig {
@Override
public List<IPluginModule> getPluginModules(Conversation.ConversationType conversationType, String targetId) {
List<IPluginModule> pluginModules = super.getPluginModules(conversationType,targetId);
ListIterator<IPluginModule> iterator = pluginModules.listIterator();

// 如果
while (iterator.hasNext()) {
IPluginModule integer = iterator.next();
// 删除不支持实时位置共享的 DefaultLocationPlugin
if (integer instanceof DefaultLocationPlugin) {
iterator.remove();
}
}

// 添加支持实时位置共享的 CombineLocationPlugin 扩展项
pluginModules.add(new CombineLocationPlugin());
return pluginModules;
}
}
```

2. SDK 初始化之后，调用以下方法设置自定义的输入配置，SDK 会根据此配置展示扩展面板。

```
RongExtensionManager.getInstance().setExtensionConfig(new MyExtensionConfig());
```

您也可以通过修改默认配置。在 res/values 下创建 IMLib 的配置文件 rc_configuration.xml 文件中，添加并调整以下配置，配置需要支持实时位置共享的会话类型。如果需要关闭位置共享功能，可以移除相应的 item。

```
<!--实时位置共享会话类型-->
<string-array name="rc_realtime_support_conversation_types" translatable="false">
<item>private</item>
<item>group</item>
</string-array>
```

升级旧版位置插件

> 提示
>
> IMKit 5.2.3 之前版本升级到 5.2.3 时，旧版位置插件即失效。

从 IMKit 5.2.3 版本开始，不再提供默认的高德地图开发账号。如开发者需要使用位置功能，可配置自己的高德地图 SDK 帐号集成使用。

如集成新版插件，请务必将原来集成的以下 Jar 包删除，否则会导致编译失败。

> 提示
>
> - `Android_Map3D_SDK_v6.9.3.jar`
> - `AMap_Search_v6.9.3.jar`
> - `MapApiLocationMini.jar`
> - `libAMapSDK_MAP_v6_9_3.so`
> - `libJni_wgs2gcj.so`

定制化

调整位置缩略图压缩比例

从客户端发送位置消息时，SDK 会自动生成预览地图图片，自动上传到文件服务器（默认上传到七牛云存储），并将云存储服务返回的图片的远程地址放入消息体对应字段后进行发送。SDK 会以宽度不超过 408 像素、高度不超过 240 进行压缩。

您可以在应用程序目录下创建 res/values/rc_configuration 文件，修改以下配置调整预览地图压缩比例与图片尺寸：

```
// 位置消息缩略图压缩比例
<integer name="rc_location_thumb_quality">70</integer>
// 位置消息缩略图压缩宽度
<integer name="rc_location_thumb_width">408</integer>
// 位置消息缩略图压缩高度
<integer name="rc_location_thumb_height">240</integer>
```

自定义位置消息的 UI

位置消息使用 LocationMessageItemProvider 模板和 RealTimeLocationMessageItemProvider 模板展示在消息列表中。

如果需要调整内置消息样式，建议自行实现消息展示模板类，并将该自定义模板提供给 SDK。所有消息模板都继承自 BaseMessageItemProvider<CustomMessage>，自定义消息展示模板也需要继

承 BaseMessageItemProvider<CustomMessage>。详见修改消息的展示样式。

名片消息　　名片消息

更新时间:2024-08-30

用户可以通过 IMKit 名片插件发送个人名片。消息将出现在会话页面的消息列表组件中。插件默认发送的消息中包含名片消息内容对象 ContactMessage（类型标识：RC:CardMsg）。

> 提示
> IMKit 默认会话页面未启用名片消息功能。如需要使用该功能，可集成 IMKit 名片插件，并提供需要展示和发送的数据。

用法

IMKit 的名片插件仅支持通过源码导入。

1. 下载融云开源仓库（GitHub · Gitee），将 contactcard 目录拷贝到您应用工程中。注意插件版本需要与当前 SDK 版本保持一致。

2. 在工程根目录下的 settings.gradle 中增加配置。

```
include ':contactcard'
```

3. 在应用的 build.gradle 中添加依赖。

```
implementation project(path: ':contactcard')
```

4. 在进入会话页之前，实现名片信息提供者，以及会话页面点击名片消息时回调方法。

```
//名片信息提供者
IContactCardInfoProvider contactCardInfoProvider = new IContactCardInfoProvider() {
@Override
public void getContactAllInfoProvider(IContactCardInfoCallback contactInfoCallback) {
//获取所有名片列表，并通过 contactInfoCallback 回调给名片模块
imInfoProvider.getAllContactUserInfo(contactInfoCallback); //伪代码，应用层实现。
}

@Override
public void getContactAppointedInfoProvider(String userId, String name, String portrait, IContactCardInfoCallback contactInfoCallback) {
//获取单一用户的名片信息
imInfoProvider.getContactUserInfo(userId, contactInfoCallback); //伪代码，应用层实现。
}
};

IContactCardClickListener contactCardClickListener = new IContactCardClickListener() {
//在会话页面点击名片消息时，回调此方法，应用层可以在此回调里实现跳转逻辑。
@Override
public void onContactCardClick(View view, ContactMessage content) {
//此处示例点击名片进入到个人详细界面
Context activityContext = view.getContext();
Intent intent = new Intent(activityContext, UserDetailActivity.class);
intent.putExtra(IntentExtra.STR_TARGET_ID, content.getId());
activityContext.startActivity(intent);
}
};
```

5. 通过 RongExtensionManager，向 IMKit 的输入区域 RongExtension 中注册名片模块 ContactCardExtensionModule。名片模块会向 IMLib 注册 ContactMessage，并向扩展面板中添加名片插件 ContactCardPlugin。以上步骤建议在应用生命周期内统一配置。

```
RongExtensionManager.getInstance().registerExtensionModule(new ContactCardExtensionModule(contactCardInfoProvider, contactCardClickListener));
```

发送名片消息

用户点击输入栏右侧 + 号按钮可展开扩展面板，点击个人名片图标，即可发送名片。

定制化

名片插件内部定义了名片消息内容类 ContactMessage.java，名片消息展示模板 ContactMessageItemProvider.java。

自定义名片消息的 UI

如果需要调整内置消息样式，建议自行实现消息展示模板类，并将该自定义模板提供给 SDK，替换默认的名片消息展示模板。

所有消息模板都继承自 BaseMessageItemProvider<CustomMessage>，自定义消息展示模板也需要继承 BaseMessageItemProvider<CustomMessage>。详见修改消息的展示样式。

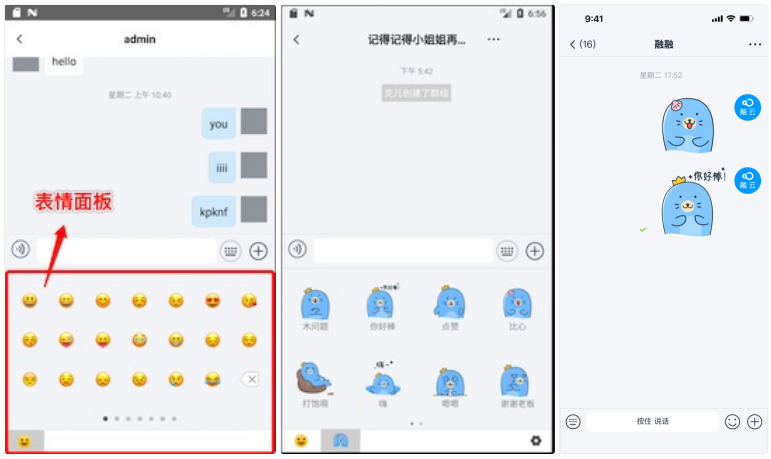
您也可以直接替换名片消息展示模板中引用的样式资源、字符串资源和图标资源。详见 IMKit 源码 ContactMessageItemProvider.java 中引用的资源。

Emoji 与贴纸表情 Emoji 与贴纸表情

更新时间:2024-08-30

用户可以 IMKit 输入区域发送 Emoji 表情、贴纸表情。点击输入栏的表情（☺）按钮，即可展开表情面板，支持发送 Emoji 表情、贴纸表情。表情消息将出现在会话页面的消息列表组件中。

> 提示
>
> IMKit 输入区域的表情面板中默认仅包含 emoji。图中的贴纸表情需要集成 `rcsticker` 库。支持添加自定义表情。



Emoji 符号表情

IMKit 输入区域的表情面板中默认展示内置 Emoji 符号表情。用户点击后可发送 Emoji 符号表情。

适配新式 Emoji 符号（Emoji 13.1）

> 提示
>
> IMKit 从 5.2.2 开始支持集成融云 Emoji 模块。

随着用户越来越多地使用各种应用中的表情符号，Unicode 每年都会更新标准表情符号集。根据 Android 表情符号政策的说明：

- 如果应用在 Android 12 或更高版本上运行，而且使用默认 Android 表情符号，没有任何自定义实现，便已使用最新版 Unicode 表情符号。
- 如果应用在 Android 11 或更低版本上运行，在设备已经正常安装、运行谷歌服务框架（可联网）的情况下，可自动支持 Emoji 13.1 或更新的表情符号。

但由于国内手机大多数没有安装谷歌服务框架，且因网络原因无法通过可下载字体支持 Emoji 13.1，因此需要将表情符号字体捆绑到应用中，才能在 Android 11 及更低版本上支持显示新式表情符号字体。

为简化开发者的 Emoji 适配工作，融云提供了 Emoji 模块。您仅需要在项目中集成融云提供的 Emoji 模块，即可适配新表情，无需额外代码。

> 提示
>
> 由于当前的 Emoji 字体大小超过 10 MB，建议您根据实际需要考虑是否集成融云 Emoji 模块。

```
// 如果您依赖的 androidx.appcompat:appcompat 版本为 1.4.0 及以上，仅需添加以下依赖
implementation 'cn.rongcloud.sdk:emoji-compat:5.2.2'

// 如果您依赖的 androidx.appcompat:appcompat 版本为 1.4.0 以下，需添加以下两个依赖
implementation 'cn.rongcloud.sdk:emoji-compat:5.2.2'
implementation "androidx.emoji2:emoji2:1.1.0"
```

禁用表情面板中的内置 Emoji 表情

> 提示
>
> SDK 从 5.2.3 开始支持禁用内置 Emoji 表情。

用户点击会话列表中的会话时，SDK 自动跳转带有内置 Emoji 的会话页面 Activity（`RongConversationActivity`）。如需隐藏融云内置 Emoji，您需要自行跳转到会话页面，同时设置隐藏 内置 Emoji 表情。禁用后，跳转的目标会话页面的表情面板中不会显示 Emoji 标签页。

```
String targetId = "userId";
ConversationIdentifier conversationIdentifier = new ConversationIdentifier(Conversation.ConversationType.PRIVATE, targetId);
Boolean disableSystemEmoji = true;

RouteUtils.routeToConversationActivity(context, conversationIdentifier, disableSystemEmoji, bundle)
```

参数	类型	说明
context	Context	Activity 上下文
conversationIdentifier	ConversationIdentifier	会话类型。其中的 `targetId` 为会话的目标 ID。单聊会话时使用对方用户 ID 为会话 ID，群聊会话时为群组 ID。
disableSystemEmoji	Boolean	是否隐藏 SDK 内置 Emoji 表情。`true` 为隐藏，`false` 为不隐藏。
bundle	Bundle	扩展参数

贴纸表情

IMKit 表情面板中可支持贴纸表情，您可以集成融云表情贴纸库 `rcsticker`，也可以添加自定义贴纸表情。

集成融云贴纸表情

IMKit 可集成融云表情贴纸库 `rcsticker`，其中包含一套“嗨豹”贴纸表情。

1. 下载 IMKit 源码，将 `rcsticker` 目录拷贝到您应用工程中。

2. 在工程根目录下的 `settings.gradle` 中增加以下配置：

```
include ':rcsticker'
```

3. 在应用的 `build.gradle`中添加以下依赖项：

```
implementation project(path: ':rcsticker')
```

集成成功后，`StickerExtensionModule` 模块会向表情面板中添加融云贴纸标签页。首次点击后请根据提示下载贴纸。下载成功后会在面板里展示所有贴纸。

自定义贴纸表情

> 提示
>
> **自定义表情的资源码不可与 rc_emoji.xml 中的资源码重复。**

IMKit 的 `rcsticker` 库中的 `StickerExtensionModule` 实现了 `IExtensionModule` 接口类，通过实现其中的 `getEmoticonTabs` 方法，向 IMKit 提供扩展面板中添加了自定义表情页（`IEmoticonTab` 实例）。

要添加自定义贴纸表情，有两种方式：

- 实现 `IExtensionModule` 接口类，实现 `getPluginModules()` 方法，添加 `IEmoticonTab` 实例。下文介绍这种方式。
- 继承 `DefaultExtensionConfig` 创建自定义扩展面板配置，重写 `getPluginModules()` 方法，添加 `IEmoticonTab` 实例。

以下步骤介绍了如何通过实现 `IExtensionModule`，将自定义贴纸表情加入表情面板。

1. 创建 `MyEmoticonTab` 实现 IEmoticonTab。

```
public class MyEmoticonTab implements IEmoticonTab {

public MyEmoticonTab() {
}
@Override
public Drawable obtainTabDrawable(final Context context) {
return context.getResources().getDrawable(R.drawable.u1f603);
}

@Override
public View obtainTabPager(Context context) {
//参考步骤 2
return initView(context);
}
@Override
public void onTableSelected(int i) {
}

@Override
public LiveData<String> getEditInfo() {
return null;
}
}
```

2. 在上方的 obtainTabPager 方法中添加想要展示在表情面板上的 View。下方提供了一个参考示例：

```
public View initView(Context context) {
View view = LayoutInflater.from(context).inflate(R.layout.view_emoji, null);
RecyclerView rv = view.findViewById(R.id.recycler_view);
//LinearLayoutManager是用来做列表布局，也就是单列的列表
GridLayoutManager mLayoutManager = new GridLayoutManager(context, 5, OrientationHelper.VERTICAL, false);
rv.setLayoutManager(mLayoutManager);

// Google 提供了一个默认的item删除添加的动画
rv.setItemAnimator(new DefaultItemAnimator());
rv.setHasFixedSize(true);

//模拟列表数据
ArrayList newsList = new ArrayList<>();
TypedArray array = context.getResources().obtainTypedArray(context.getResources().getIdentifier("rc_emoji_res", "array", context.getPackageName()));
int i = -1;
while (++i < array.length()) {
newsList.add(array.getResourceId(i, -1));
}
adapter = new NewsAdapter(newsList);
rv.setAdapter(adapter);
return view;
}
```

3. 创建 MyEmoticonTabExtModule 实现 IExtensionModule 接口类。在 getEmoticonTabs() 方法中返回本模块提供的表情页列表。您可以参考 IMKit 源码中的 IExtensionModule.java 及其他实现类。

```
public class MyEmoticonTabExtModule implements IExtensionModule {
@Override
public void onInit(Context context, String appKey) {

}

@Override
public void onAttachedToExtension(Fragment fragment, RongExtension extension) {

}

@Override
public void onDetachedFromExtension() {

}

@Override
public void onReceivedMessage(Message message) {

}

@Override
public List<IPluginModule> getPluginModules(Conversation.ConversationType conversationType) {
return null;
}

@Override
public List<IEmoticonTab> getEmoticonTabs() {
List<IEmoticonTab> emoticonTabs = new ArrayList<>();
emoticonTabs.add(myEmoticonTab);
return emoticonTabs;
}

@Override
public void onDisconnect() {

}
}
```

4. 在初始化之后，进入会话页面之前，通过 RongExtensionManager，向 IMKit 的输入区域 RongExtension 中注册自定义的 MyExtensionModule。

```
RongExtensionManager.getInstance().registerExtensionModule(new MyExtensionModule());
```

隐藏表情面板入口

> 提示
>
> IMKit SDK 从 5.3.2 版本开始提供该功能。

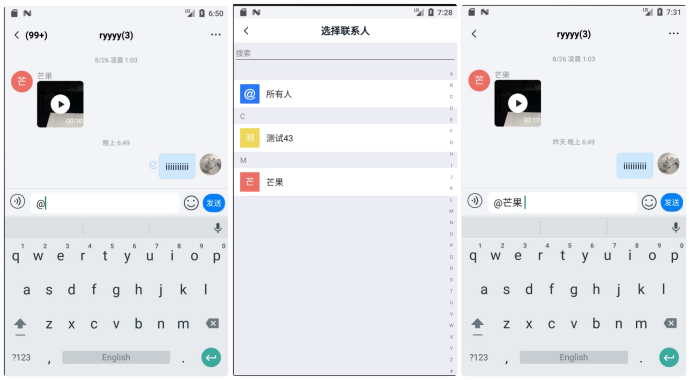
在 App 不需要提供表情输入功能时，可隐藏输入栏中的表情按钮，隐藏后用户无法展开表情面板。详见输入区域。

@ 消息 @ 消息

更新时间:2024-08-30

提及（@）是群聊会话中常见功能，允许用户可以会话中提及指定用户，或全部群成员，以增强消息的提示作用。使用 @ 功能后，消息内容中会额外携带 MentionedInfo 对象。IMKit 默认启用了 @ 功能。

> 提示
>
> IMKit 未实现 @所有人功能，图中「@所有人」仅作为设计参考。选择联系人页面的数据需要由应用程序提供，否则展示空列表。在使用 @ 功能前，请先实现 群组成员提供者。



局限

- 仅支持群聊会话。
- IMKit 默认仅实现了在发送文本消息、引用消息时使用 @ 功能。
- IMKit 未实现 @ 所有人功能。
- @ 消息可以被转发，但转发的只是纯文本，不再具备 @ 功能。

用法

> 提示
>
> 在使用 @ 功能前，请先实现群组成员提供者。

IMKit 默认在配置中启用了 @ 功能，用法如下：

- 在会话页面长按用户头像可触发消息编辑，提及（@）该用户。
- 在会话页面输入 @ 符号之后，IMKit 会跳转到成员列表选择页面。如果应用程序未实现群组成员提供者（`IGroupMembersProvider`），该页面会显示一个空列表。实现群组成员提供者（`IGroupMembersProvider`）后，IMKit 会通过 `IGroupMembersProvider` 对象的 `getGroupMembers` 方法取得群成员数据，并展示在该列表页面中。

定制化

自定义选择成员界面

打开 @ 功能开关后，直接输入 @ 字符，会弹出成员列表页面（MentionMemberSelectActivity），您可以替换该页面。

1. 调用 RongMentionManager 的方法设置监听器，监听 @ 字符输入。

```
RongMentionManager.getInstance().setMentionedInputListener(new IMentionedInputListener() {
/**
* 设置监听
* @param conversationType 会话类型. 群会话类型
* @param targetId 会话 id. 群 ID
*/
@Override
public boolean onMentionedInput(Conversation.ConversationType conversationType, String targetId) {
// 此处可跳转到您自定义的成员列表, 或进行其他操作.
// 返回 true, 拦截事件
return true;
}
});
```

2. 在监听器的回调 onMentionedInput() 里跳转到自定义的 @ 成员选择界面，并返回 true。

3. 在自定义的成员选择界面中选择成员后，调用 mentionMember 方法返回所选成员信息 UserInfo。

如果返回 UserInfo，IMKit 编辑框中会按照 UserInfo 中的昵称显示被 @ 成员的名称。

```
RongMentionManager.getInstance().mentionMember(userInfo);
```

如果使用以下方法。IMKit 会通过 userId 从您实现的群成员用户信息提供者（UserDataProvider.GroupUserInfoProvider）中获取需要显示的用户昵称。

```
String targetId = "群组 Id";
String userId = "所 @ 的人的用户 ID";

RongMentionManager.getInstance().mentionMember(ConversationType.GROUP, targetId, userId);
```

实现 @ 所有人

IMKit 未实现 @ 所有人功能的页面逻辑，您可以自行实现。

1. 新建 MentionedInfo，设置 MentionedType 为 MentionedType.ALL。

```
String targetId = "群组 Id";
MentionedInfo mentionedInfo = new MentionedInfo(MentionedInfo.MentionedType.ALL, null, null);
```

2. 将 MentionedInfo 对象设置到 MessageContent 中。

```
TextMessage messageContent = TextMessage.obtain(content);
messageContent.setMentionedInfo(mentionedInfo);
Message message = Message.obtain(targetId, ConversationType.GROUP, textMessage);
```

3. 调用 IMKit 核心类 IMCenter 的方法发送消息。

```
IMCenter.getInstance().sendMessage(message, null , null, new IRongCallback.ISendMessageCallback(){
@Override
public void onAttached(Message message) {

}

@Override
public void onSuccess(Message message) {

}

@Override
public void onError(Message message, RongIMClient.ErrorCode errorCode) {

}
});
```

关闭 @ 功能

IMKit 的 @ 功能默认开启。您可以通过修改全局配置关闭 @ 功能。

```
RongConfigCenter.conversationConfig().rc_enable_mentioned_message = false;
```

如果通过 XML 资源修改 IMKit 默认配置，可在应用 res/values 目录下创建 rc_config.xml 文件，添加以下配置：

```
<bool name="rc_enable_mentioned_message">false</bool>
```

输入状态　　输入状态

更新时间:2024-08-30

输入状态可让用户直观地了解其他用户是否正在键入消息。在对方用户键入内容时，标题栏会一直显示「对方正在输入」，直到用户发送消息或完全删除文本。如果用户停止打字超过 6 秒，该提示也会消失。SDK 在输入框中有内容变化时，默认向对端用户发送一条正在输入的状态消息，包含消息内容对象 TypingStatusMessage（类型标识：RC:TypSts）。

> 提示
>
> IMKit 在默认会话页面 Activity（RongConversationActivity）包含了标题栏显示正在输入状态的实现。如果基于 Fragment 构建会话页面，请自行实现标题栏输入状态的展示与更新。

局限

- 只支持单聊会话。
- 因无法确定用户的输入操作，该功能可能会产生大量状态消息，为防止消息发送频繁，默认在 6 秒钟内的多次状态变化，只产生一条输入状态消息。
- 该功能可能会导致大量状态消息，如不需要此功能建议关闭。

用法

如果使用 IMKit 默认会话页面 Activity（RongConversationActivity），该功能默认可用，无需额外处理。

在自定义会话页面监听输入状态

IMKit 的 ConversationFragment 不含标题栏实现。如果使用 Fragment 构建自定义会话页面，您需要自行实现输入状态的展示与更新。IMKit SDK 内部已经处理好逻辑，在输入框中有内容变化时，SDK 会向目标用户发送一条正在输入的状态消息。接收方可根据该状态消息，实现类似「对方正在输入」的提示。

应用程序仅需要在自定义会话页面 Activity 注册监听器，在收到回调通知时更新标题栏。

1. 在自定义会话页面 Activity 中注册输入状态的监听器。您可以在 Activity 的 onCreate() 里添加如下代码：

```
RongIMClient.setTypingStatusListener(new RongIMClient.TypingStatusListener() {
@Override
public void onTypingStatusChanged(Conversation.ConversationType type, String targetId, Collection<TypingStatus> typingStatusSet) {
//当输入状态的会话类型和targetID与当前会话一致时，才需要显示
if (type.equals(mConversationType) && targetId.equals(mTargetId)) {
//count表示当前会话中正在输入的用户数量，目前只支持单聊，所以判断大于0就可以给予显示了
int count = typingStatusSet.size();
if (count > 0) {
Iterator iterator = typingStatusSet.iterator();
TypingStatus status = (TypingStatus) iterator.next();
String objectName = status.getTypingContentType();

MessageTag textTag = TextMessage.class.getAnnotation(MessageTag.class);
MessageTag voiceTag = VoiceMessage.class.getAnnotation(MessageTag.class);
//匹配对方正在输入的是文本消息还是语音消息
if (objectName.equals(textTag.value())) {
//显示“对方正在输入”

} else if (objectName.equals(voiceTag.value())) {
//显示"对方正在讲话"

}
} else {
//当前会话没有用户正在输入，标题栏仍显示原来标题

}
}
}
});
```

2. 当对方正在输入时，本端会触发 onTypingStatusChanged()，回调里携带有正在输入的用户列表。当对方停止输入时，该监听还会触发一次，此时回调里的输入用户列表为空。这时您需要取消「正在输入」的显示，并显示原有的会话标题栏。

定制化

设置发送输入状态消息的默认时间间隔

如果修改 IMKit 默认配置，您可以在应用 res/values 目录下创建 rc_configuration.xml 文件，修改以下配置项的值，单位为毫秒。

```
<integer name="rc_disappear_interval">6000</integer>
```

关闭输入状态功能

IMKit 默认开启输入状态功能。您可以通过修改 XML 关闭输入状态功能。在应用 res/values 目录下创建 rc_configuration.xml 文件，修改以下配置项。

```
<bool name="rc_typing_status">false</bool>
```

已读回执　　已读回执

更新时间:2024-08-30

IMKit 提供了单聊、群聊的已读回执功能。App 用户通过已读回执获知对方是否阅读该消息。

在 IMKit 内置页面中已默认实现并启用已读回执功能。在单聊会话中，消息默认更新已读状态。在群聊会话中，消息发送者需要在页面上主动请求获取已读状态。

阅读回执开关

IMKit 中回执功能默认开启，在单聊和群聊中默认会展示消息回执。

您可以将 res下 rc_config.xml 中下面的变量设为 false 来关闭此功能。

```
<bool name="rc_read_receipt">false</bool>
```

上面变量配置为 false 后，单聊和群聊中的回执功能都会被关闭。

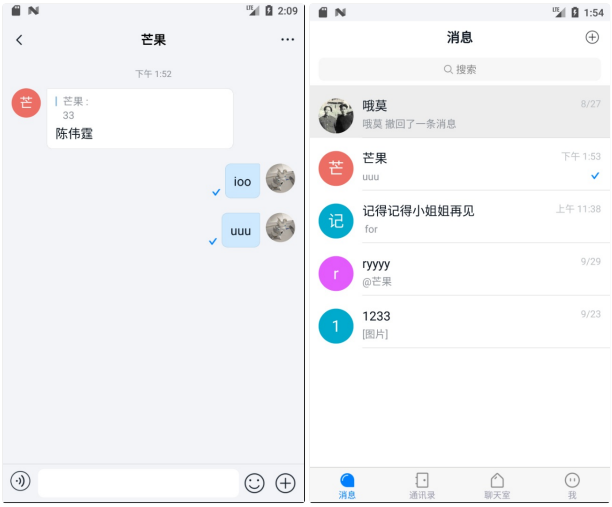
您也可以在进入会话列表页面之前，通过下面的代码动态配置回执功能支持的会话类型。

```
//下面的代码将阅读回执功能配置为仅在单聊中支持，群聊中将不启用阅读回执功能。
RongConfigCenter.conversationConfig().setSupportReadReceiptConversationType(Conversation.ConversationType.PRIVATE);
```

单聊阅读回执

在单聊会话中，发送方会实时收到消息的已读状态更新。在 IMKit 内置页面中，单聊已读状态显示在两处：

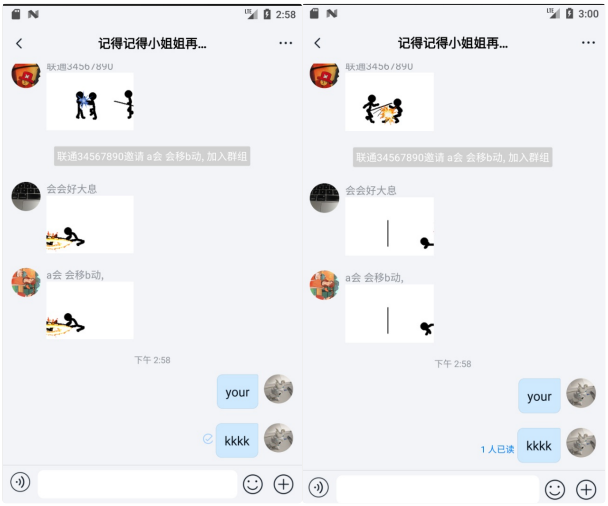
- **单聊会话页面（消息列表页面）**：在发送方的单聊会话页面，消息的左下角会显示对号，表示对方已读。
- **会话列表页面**：会话列表的每条会话会显示会话中的最后一条消息的预览。如果单聊会话最后一条消息被对方阅读，发送方的会话列表页中对应会话条目的右下角也会显示对号，表示对方已读。



群聊阅读回执

> 提示
>
> IMKit 的群聊已读回执功能仅支持文本消息类型。

在群聊会话中发送消息后 120 秒之内，发送者可以在会话页面上主动请求获取已读人数数据。在会话页面上，消息的左下角会显示对号按钮（请求阅读回执的按钮）。点击按钮后，IMKit 才会请求已读回执。IMKit 在收到已读回执结果后会刷新页面，显示为 “n 人已读”。



在群聊会话中发送消息后 120 秒之后请求阅读回执的按钮会自动消失。您可以在 res 下的 rc_config.xml 中修改回执请求按钮的有效时间。

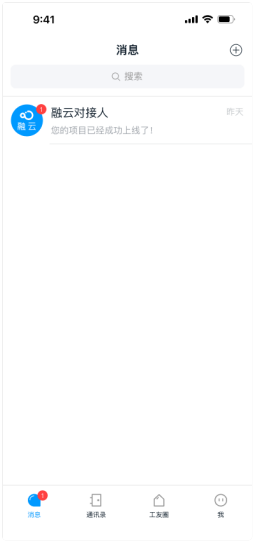
```
<integer name="rc_read_receipt_request_interval">120</integer>
```

消息未读数　　消息未读数

更新时间:2024-08-30

未读消息计数是 IMKit 默认提供的一项功能，可告知用户每个会话中未读消息的数量。未读消息计数显示在会话列表 ConversationListFragment 的会话条目上。每个会话的未读消息数显示在会话图标右上角。如果未读消息数超过 100 条，则会显示为 99+ 。

> 提示
>
> 为了使用未读消息计数功能，您必须首先构建会话列表页面。IMKit 默认未实现底部导航栏。



用法

IMKit SDK 默认已经实现了一整套会话未读消息数获取和展示逻辑，使用默认会话列表和会话页面时，不需要额外调用会话相关 API。

IMKit 会在用户进入单聊、群聊、系统会话页面时将会话未读数清零。在用户多端登录时，IMKit 会在设备间同步会话的阅读状态，您也可以按业务需求选择关闭该功能，详见下文多端同步阅读状态。

定制化

如果 IMKit 已有实现无法满足您的需求，可以使用 IMKit 或 IMLib SDK 中相关 API。

获取会话未读数

IMKit 未直接提供获取会话未读数的 API。如果您有类似以下自定义需求，可以调用 IMLib SDK 相关方法。

- 获取所有会话未读数（IMLib 方法）
- 获取指定会话未读数（IMLib 方法）
- 按会话类型获取未读数（IMLib 方法）

具体的核心类、API 与 使用方法，详见 IMLib 文档 处理会话未读消息数。

> 提示
>
> IMLib 方法并不提供页面刷新能力，您需要根据业务需求自定义通知机制进行页面刷新。

清除会话未读数

使用 IMCenter 的 `clearMessagesUnreadStatus()` 方法清除指定会话的全部未读数，该方法接受一个会话类型枚举值和一个会话 ID。IMKit 会自动刷新会话列表页面的未读展示。

支持的会话类型：

- 单聊（`ConversationType.PRIVATE`）
- 群聊（`ConversationType.GROUP`）
- 系统（`ConversationType.SYSTEM`）

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";

IMCenter.getInstance().clearMessagesUnreadStatus(conversationType, targetId, new ResultCallback<Boolean>() {

@Override
public void onSuccess() {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

提示

- IMKit 未提供根据时间戳清除未读数的方法，您可以调用 IMLib SDK 提供的方法（ `RongIMClient` 或 RongCoreClient 的 `clearMessagesUnreadStatus` 方法）。IMLib 方法并不提供页面刷新能力，您需要根据业务需求自定义通知机制进行页面刷新。

监听会话未读数变化

IMKit 提供了未读数变化监听机制。您可以设置监听器，监听指定一个或多个会话类型的会话未读数变化事件。

版本 5.6.7 及更早版本

在这些版本中，您可以通过 UnReadMessageManager 类的 addObserver 方法注册一个 IUnReadMessageObserver 监听器，同时传入您希望监听的会话类型的数组。当这些会话类型的未读消息数发生变化时，SDK 将自动触发 onCountChanged 方法，并通过该监听器将更新后的总未读消息数回调至您的应用程序。

```
UnReadMessageManager.getInstance().addObserver(conversationTypes,observer);
```

当您不再需要监听未读消息数变化时，可以通过调用 removeObserver 方法来注销已注册的监听器。

```
UnReadMessageManager.getInstance().removeObserver(observer);
```

如果您在 Activity 或 Fragment 中使用此功能，请在组件的生命周期结束时调用 removeObserver 方法来移除监听器，以避免潜在的内存泄漏问题。

版本 5.8.0 及更新版本

在 5.8.0 及以后的版本中，我们引入了 addForeverObserver 和 removeForeverObserver 接口。这些接口通过强引用实现，确保了监听器的持久性。如果您在 Activity 或 Fragment 中使用此功能，建议在组件生命周期结束时调用 removeForeverObserver 来移除监听器，以避免潜在的内存泄漏问题。

```
UnReadMessageManager.getInstance().addForeverObserver(conversationTypes, observer);
```

```
UnReadMessageManager.getInstance().removeForeverObserver(observer);
```

请注意，addObserver 和 removeObserver 接口在版本 5.8.0 及以后版本中通过弱引用实现，减少了内存泄漏的风险。

版本 5.6.10

这个版本存在 UnReadMessageManager 监听失效的问题，建议您升级到版本 5.8.0，并根据接口指南进行操作。

修改已读未读状态图标为文本

SDK 会话页面中文本消息已读的 UI 默认是一个“对勾”图标，如果希望修改为「已读」或「未读」，可以在消息展示时将 SDK 默认的图标移除。

修改步骤详见知识库文档修改已读未读状态图标为文字提示。

多端同步阅读状态

提示

如果 SDK 版本 ≦ 5.6.2，不支持多端同步 系统会话的已读、未读状态。

在即时通讯业务中，同一用户账号可能在多个设备上登录。仅在开通多设备消息同步服务后，融云会在多个设备之间同步消息数据，但设备上的会话中消息的已读/未读状态仅存储在本地。

IMKit SDK 已默认实现了会话多端阅读状态同步功能，在一端发起同步后，其他端可接收通知，按要求同步阅读状态。您可以按业务需求决定是否使用该功能，该功能默认开启。

```
RongConfigCenter.conversationConfig().setEnableMultiDeviceSync(true);
```

主动同步消息未读状态

IMKit SDK 默认在进入会话页面后会清理会话消息的未读状态。如需在未进入会话页面时同步未读状态，需要 App 主动同步消息未读状态。

IMCenter 提供 syncConversationReadStatus 方法，可在多端登录时，通知其它终端同步某个会话的消息未读状态。该方法同时按时间戳清除本端会话中传入时间之前的所有消息的未读状态。会话列表页面的会话未读数也会同步刷新。

```
IMCenter.getInstance().syncConversationReadStatus(conversationType, targetId, timestamp, callback);
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID
timestamp	String	该会话中已读的最后一条消息的发送时间戳
callback	RongIMClient.OperationCallback	回调接口

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";
String timestamp = "12222222";

IMCenter.getInstance().syncConversationReadStatus(conversationType, targetId, timestamp, new RongIMClient.OperationCallback() {
@Override
public void onSuccess() {
}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {
}
});
```

监听消息未读状态同步数据

> 提示
>
> IMKit 内置的会话列表页面已通过该监听实现了未读状态的同步逻辑，当其它端的消息变为已读时，本端通过监听会清除对应会话的消息未读数，您不需要额外处理。

IMCenter 中提供了 SyncConversationReadStatusListener 监听器。客户端设置该监听器后，才能接收来自其他同步的阅读状态数据。

```
IMCenter.getInstance().addSyncConversationReadStatusListener(listener);
```

在接收到阅读状态同步数据后，会触发监听器的以下方法。SDK 会将指定会话中早于等于 syncConversationReadStatus 传入时间戳的消息均置为已读：

```
onSyncConversationReadStatus(Conversation.ConversationType type, String targetId)
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID

未读消息气泡提醒

IMKit 支持在会话页面中显示未读消息气泡提醒。

是否显示未读消息数提醒

如果会话的未读消息数已超过 10，可在进入会话页面后在右下角显示提醒气泡，例如「99+」。用户点击提醒气泡后，页面会跳转到最开始的未读消息。

未读消息数提醒组件默认不显示，您可以调用如下方法打开此功能：

```
// 设置显示未读消息数目
RongIM.getInstance().enableUnreadMessageIcon(true);
// 或者使用以下方法
RongConfigCenter.conversationConfig().setShowHistoryMessageBar(true);
```

未读消息气泡在未读消息大于 **10** 条 即可展示，超过 **99** 条显示为 "99+ 条新消息"。控件样式可以在 `rc_conversation_fragment.xml` 中进行调整，具体控件为 `rc_new_message_number`。

启用未读消息数提醒后，默认未读消息气泡在未读消息大于 **10** 条 时展示。您可以配置未读消息为多少条时才展示气泡。

```
RongConfigCenter.conversationConfig().setConversationShowUnreadMessageCount(10);
```

是否显示未读 @ 消息数提醒

当一个群聊会话收到大量消息（超过一个屏幕能显示的内容），且收到的消息中有 @ 消息时，进入会话页面后，会话页面右上角会提示未读 @ 消息数。用户点击该提醒按钮，会跳转到最早的未读 @ 消息处，同时未读 @ 消息数量减 1。再次点击，未读 @ 消息数量会根据当前屏幕内看到的个数相应减少。

新消息提醒组件默认不显示，您可以调用如下方法打开此功能：

```
// 设置显示未读 @ 消息数提醒
RongConfigCenter.conversationConfig().setShowNewMentionMessageBar(true);
```

是否显示新消息提醒

如果用户在查看会话页面中的历史消息，且当前视图未显示会话最新消息，此时如果收到新消息，会话页面右下角可显示新消息气泡提醒，例如「15 条新消息」。用户点击提醒按钮，会滚动到会话最新消息处。

新消息提醒组件默认不显示，您可以调用如下方法打开此功能：

```
// 设置显示新消息提醒
RongIM.getInstance().enableNewComingMessageIcon(true);
// 或者使用以下方法
RongConfigCenter.conversationConfig().setShowHistoryMessageBar(true);
```

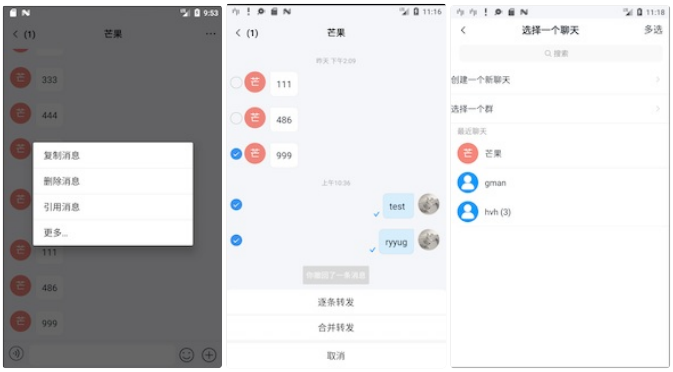
新消息气泡在新消息数量大于 **1** 条 即可展示，超过 **99** 条 显示为 **99+**。控件的样式可以在 `layout/rc_fr_messagelist.xml` 中进行调整。

转发消息　　转发消息

更新时间:2024-08-30

IMKit 支持对单条消息转发，和对多条消息的逐条转发/合并转发的功能，允许用户在聊天页面中将消息转发到其他会话中。转发后消息将出现在目标会话页面的消息列表组件中。

> 提示
>
> IMKit 默认未启用合并转发功能。您可以按需启用合并转发功能。



局限

- 并非所有消息类型均支持合并转发。
  - 支持的消息类型：文本、图片、图文、GIF、动态表情（`RC:StkMsg`）、名片、位置、小视频、文件、普通语音、高清语音、音视频通话（`RC:VCSummary`）。
  - 不支持的情况：未在支持列表中的消息类型，例如引用消息，以及未发送成功的消息等特殊情况不支持转发。自定义消息均不支持合并转发。
- 合并转发支持合并最多 100 条消息。

用法

IMKit 会话页面默认已启用转发功能。用户在会话页面长按消息，在弹框里选择更多，即可展示转发消息选项。

- **逐条转发**：默认开启。可转发单聊或多条消息至目标会话。
- **合并转发**：默认关闭，不展示。如果合并转发，SDK 会将选中的消息合并为一条合并转发消息，包含消息内容对象 `CombineMessage`（类型标识：`RC:CombineMsg`）。合并转发的消息默认折叠显示，可点击展开。

启用合并转发功能

IMKit 支持合并转发功能，该功能默认关闭。您可以修改全局配置，打开 IMKit 的合并转发功能。

```
RongConfigCenter.conversationConfig().rc_enable_send_combine_message = true;
```

如果通过 XML 资源修改 IMKit 默认配置，可在应用 `res/values` 目录下创建 `rc_config.xml` 文件，添加以下配置：

```
<bool name="rc_enable_send_combine_message">true</bool>
```

定制化

修改单次转发消息数量

转发消息数量受消息多选功能。IMKit 默认实现消息多选能力，您可以修改多选消息数量上限。修改方法详见会话页面。注意，合并转发功能支持最多合并 100 条，无法通过配置上调限制。

跳转自定义的转发会话列表

IMKit 中选中逐条转发或合并转发后，默认跳转的页面是 SDK 本地存储的最近会话列表。

如果需要 SDK 在转发选择会话时跳转应用程序自定义页面，调用 SDK 内置路由 `RouteUtils` 的以下方法将自定义 Activity 注册给 SDK 即可。

在进入会话页面之前配置：

```
//此处以自定义页面为 MyForwardSelectActivity 为例。
RouteUtils.registerActivity(RouteUtils.RongActivityType.ForwardSelectConversationActivity, MyForwardSelectActivity.class);
```

关闭转发功能

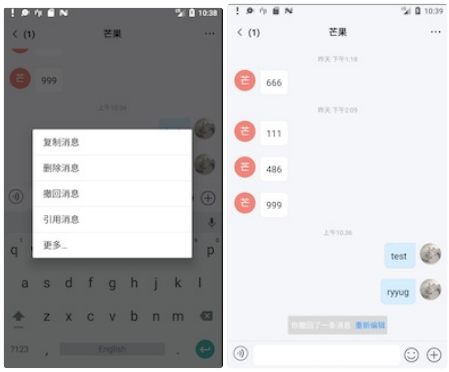
如果不需要转发功能，可以直接将长按消息弹窗中的「更多」选项隐藏。方法详见会话页面。

撤回消息　　撤回消息

更新时间:2024-08-30

用户通过 App 成功发送了一条消息之后，可能发现消息内容错误等情况，希望将消息撤回，同时从接收者的消息记录中移除该消息。IMKit 默认实现了消息撤回功能。

> 提示
>
> IMKit 在撤回消息后，会替换聊天记录中的原始消息为一条 objectName 为 RC:RcNtf 的撤回通知消息（RecallNotificationMessage），可参见服务端文档通知类消息格式。



用法

IMKit 默认启用撤回功能。用户在会话页面长按消息（已发送成功的消息）可打开弹窗，选择撤回。消息撤回后在一定时间内可以“重新编辑”。

定制化

限制撤回操作权限

默认情况下，融云对撤回消息的操作者不作限制。这意味着任何人都可以撤回他人发送的消息。如需限制，可考虑以下方案：

- App 客户端自行限制撤回消息的操作者。例如，不允许 App 业务中的普通用户撤回他人发送的消息，允许 App 业务中的管理员角色撤回他人发送的消息。
- 如需避免用户撤回非本人发送的消息，可以提交工单申请打开**IMLib SDK 只允许撤回自己发送的消息**。从融云服务端进行限制，禁止用户撤回非本人发送的消息。

修改消息可撤回的最大时间

IMKit 默认允许在消息发送后 120 秒内撤回。您可以通过全局配置调整该上限。

```
RongConfigCenter.conversationConfig().rc_message_recall_interval = 120;
```

如果通过 XML 资源修改 IMKit 默认配置，可在应用 res/values 目录下创建 rc_config.xml 文件，添加以下配置：

```
<integer name="rc_message_recall_interval">120</integer>
```

修复撤回后可重新编辑的时间

IMKit 默认允许在消息撤回后 300 秒内可点击**重新编辑**，仅文本消息支持撤回再编辑。您可以通过全局配置调整该上限。

```
RongConfigCenter.conversationConfig().rc_message_recall_edit_interval = 300;
```

如果通过 XML 资源修改 IMKit 默认配置，可在应用 res/values 目录下创建 rc_config.xml 文件，添加以下配置：

```
<integer name="rc_message_recall_edit_interval">300</integer>
```

其他定制化

IMKit SDK 默认已经实现了一套消息撤回和展示逻辑，不需要额外调用会话相关 API。如果已有实现无法满足您的需求，可以使用 IMCenter 中相关 API。详见撤回消息。

关闭撤回功能

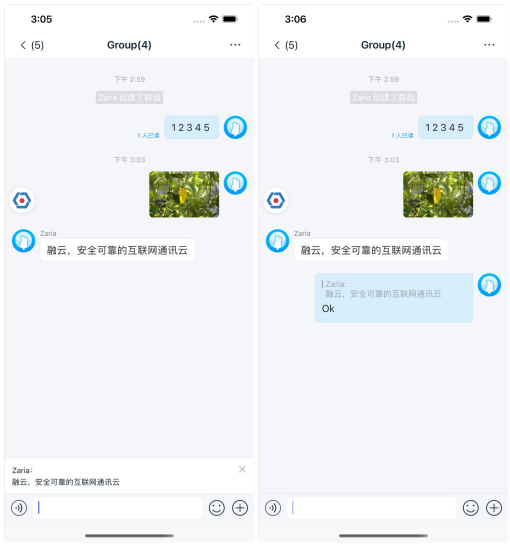
您可以通过 XML 修改 IMKit 默认配置。在应用 res/values 目录下创建 rc_config.xml 文件，添加以下配置项，设置为 false：

```
<bool name="rc_enable_message_recall">false</bool>
```

引用回复　　引用回复

更新时间:2024-08-30

IMKit 支持引用回复功能，允许用户在聊天页面中回复彼此的消息。消息将出现在会话页面的消息列表组件中。引用回复功能默认发送的消息包含引用消息内容对象 ReferenceMessage（类型标识：RC:ReferenceMsg）。



局限

引用回复功能目前有以下限制：

- 仅支持文本消息、文件消息、图文消息、图片消息、引用消息的引用。
- 引用深度仅支持一度，即只能引用回复原始消息。如果多重引用，只展示上一层被引消息内容。

用法

IMKit 会话页面默认已启用引用回复功能。用户在会话页面长按消息，在弹框里选择**引用消息**，即可引用该消息。在输入区添加消息内容后，SDK 默认会将输入内容与被引消息组合为 ReferenceMessage，并发送到会话中。

关闭引用回复功能

IMKit 5.X 版本不支持通过修改 XML 资源文件控制此功能。如需关闭引用回复功能，可以修改 IMKit 全局配置：

在 init 之后调用。

```
RongConfigCenter.featureConfig().enableReference(false);
```

自定义引用消息的 UI

引用回复功能默认发送的消息包含引用消息内容（RC:ReferenceMsg），使用 ReferenceMessageItemProvider 模板展示在消息列表中。

所有消息展示模板都继承自 BaseMessageItemProvider<CustomMessage>，您可以继承 BaseMessageItemProvider<CustomMessage>，自行实现一个引用消息展示模板类，并将该自定义模板提供给 SDK。

您也可以直接替换引用消息展示模板中引用的样式资源、字符串资源和图标资源。详见 IMKit 源码 ReferenceMessageItemProvider.java 中引用的资源。

例如：您可以复制 IMKit 源码中的 rc_item_reference_message.xml 的全部内容，您可以在项目下创建 /res/layout/rc_item_reference_message.xml 文件，修改其中定义的样式值。请勿删减 SDK 默认控件，不要随意修改 View 的 ID。

快捷回复 快捷回复

更新时间:2024-08-30

IMKit 支持为单聊、群聊会话页面设置常用回复语。

> 提示
>
> IMKit 默认未启用该功能。

局限

- 单条常用语最多 50 个字，分两行展示。超过 50 个字，默认情况下可能无法完全展示。如需修改，可以把 SDK 中 `rc_ext_quick_reply_list_item.xml` 复制到主 App 下，修改 XML 中 `TextView` 属性，覆盖 SDK 中默认配置。

用法

IMKit SDK 要求在初始化之前设置默认常用回复语，以开启常用回复语功能，否则该功能无法生效。

```
RongConfigCenter.featureConfig().enableQuickReply(new IQuickReplyProvider() {
@Override
public List<String> getPhraseList(Conversation.ConversationType type) {
List<String> phraseList = new ArrayList<>();
phraseList.add("您好！");
return phraseList;
}
});
```

如需再次设置常用回复语，新的常用语列表会覆盖已有常用语。如果 IMKit ≦ 5.6.2，请在启动会话之前设置，否则页面无法最新常用语。如果 IMKit ≧ 5.6.3，设置常用语后页面会立即刷新。

```
RongConfigCenter.featureConfig().enableQuickReply(new IQuickReplyProvider() {
@Override
public List<String> getPhraseList(Conversation.ConversationType type) {
List<String> phraseList = new ArrayList<>();
phraseList.add("您好！");
phraseList.add("您太客气了！");
phraseList.add("您吃饭了吗？");
return phraseList;
}
});
```

定制化

拦截点击常用语按钮事件

> 提示
>
> 要求 IMKit 版本 ≧ 5.6.3。

用户在会话页面点击常用语按钮后会弹出快捷回复。如需拦截该点击事件，返回 `true`，自定义点击常用语按钮后的逻辑；否则返回 `false`，继续执行 SDK 默认逻辑。

```
default boolean onQuickReplyClick(Context context) {
return false;
}
```

集成 CallKit 通话功能　　集成 CallKit 通话功能 语音/视频通话插件

更新时间:2024-08-30

语音通话、视频通话插件由音视频模块提供。成功集成音视频模块之后在扩展面板会自动展示这两个插件。您需要完成以下工作：

- 集成音视频通话（呼叫）SDK。详见 CallKit 文档实现音视频通话。
- 在控制台开通音视频服务。前往音视频通话服务 页面。

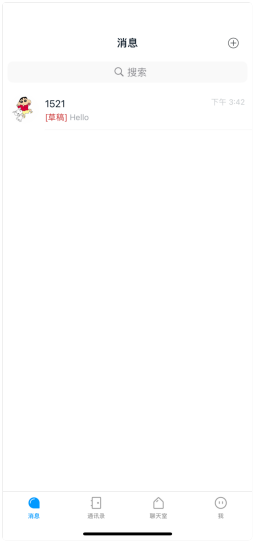
> 提示
>
> 扩展面板展示音/视频通话按钮，需要等待 IM 服务下发的配置生效以及客户端本地缓存更新，最多可能需要等待两小时。如长时间未展示，请尝试退出当前账号，清除缓存，卸载重装应用，然后查看是否解决问题。

会话草稿　　会话草稿

更新时间:2024-08-30

IMKit 支持会话草稿功能。

> 提示
>
> 用户在会话页面输入框中输入文本后没有发送，退出会话页面到会话列表，会话列表会显示草稿提示及草稿内容。



用法

IMKit 中默认已实现了获取会话草稿、删除草稿的功能和页面刷新，您不需要额外调用 API。

定制化

如果已有实现无法满足您的需求，可以使用 IMKit 和 IMLib 提供的以下 API。

保存会话草稿

使用 IMKit 核心类 IMCenter 的方法保存一条草稿内容至指定会话。保存草稿会更新会话 `sentTime`，触发会话列表重排序。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "会话 Id ";
String content = "草稿内容";

IMCenter.getInstance().saveTextMessageDraft(conversationType, targetId, content, new
ResultCallback<Boolean>() {

@Override
public void onSuccess(Boolean success) {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
})
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 Id
content	String	草稿的文字内容
callback	ResultCallback<Boolean>	回调接口

获取/删除会话草稿

IMKit 中未封装获取草稿、删除草稿的 API。如果您有自定义需求，可以调用 IMLib SDK 中获取草稿和删除草稿的方法。

- `RongIMClient#getTextMessasgeDraft()`：获取会话草稿
- `RongIMClient#clearTextMessageDraft()`：删除会话草稿

提示

详细使用方法请参见 IMLib 文档会话草稿。注意，IMLib 的方法并不提供页面刷新能力，您需要根据业务需求自定义通知机制进行页面刷新。

会话置顶 会话置顶

更新时间:2024-08-30

IMKit 提供设置会话置顶与展示置顶会话。

用法

设置会话置顶后，该状态将会被同步到服务端。融云会为用户自动在设备间同步会话置顶的状态数据。客户端可以通过监听器获取同步通知，也可以主动获取最新数据。

定制化

如果 IMKit 默认实现的功能不满足需求，您可以使用 IMKit 提供的 API。

设置会话置顶

设置会话置顶后，会话将在会话列表页面置顶显示。所有置顶会话按照会话时间降序排列。

```
IMCenter.getInstance().setConversationToTop(conversationType, targetId, isTop, needCreate, callback);
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 Id
isTop	boolean	是否置顶, true 为置顶, false 为取消置顶
needCreate	boolean	会话不存在时，是否创建会话
callback	ResultCallback<Boolean>	回调接口

客户端一般通过本地消息数据自动生成会话与会话列表。如果需要置顶的会话在本地尚不存在，您可以通过 needCreate 参数创建会话。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";
boolean isTop = true;
boolean needCreate = true;

IMCenter.getInstance().setConversationToTop(conversationType, targetId, isTop, needCreate, new
ResultCallback<Boolean>() {

@Override
public void onSuccess(Boolean success) {

}

@Override
public void onError(RongIMClient.ErrorCode ErrorCode) {

}
}) ;
```

监听置顶状态同步

即时通讯业务支持会话状态（置顶状态数据和免打扰状态数据）同步机制。设置会话状态同步监听器后，如果会话状态改变，可在本端收到通知。

会话的置顶和免打扰状态数据同步后，SDK 会触发 ConversationStatusListener 的 onStatusChanged 方法。详细说明可参见多端同步免打扰/置顶。

```
public interface ConversationStatusListener {
void onStatusChanged(ConversationStatus[] conversationStatus);
}
```

获取会话置顶状态与置顶会话

您可以从客户端主动获取会话置顶状态数据和置顶会话，但 IMKit SDK 未直接提供相关方法，您需要使用 IMLib 中提供的方法。

详见 IMLib 文档会话置顶 中的**获取会话置顶状态**与**获取置顶会话列表**。

发送消息　　发送消息

更新时间:2024-08-30

IMKit 内置会话页面已实现了发送各类型消息的功能和 UI（部分消息类型需要插件支持）。当您在自定义页面需要发送消息时，可使用 IMKit 核心类 IMCenter 或 RongIM 下发送消息的方法。这些方法除了提供发送消息的功能外，还会触发 IMKit 内置页面的更新。

IMKit 支持发送普通消息和媒体类消息（参考消息介绍），普通消息父类是 MessageContent，媒体消息父类是 MediaMessageContent。发送媒体消息和普通消息本质的区别为是否有上传数据过程。

重要

- 请务必使用 IMKit 核心类 IMCenter 或 RongIM 下发送消息的方法，否则不会触发页面刷新。发送普通消息使用 `sendMessage` 方法，发送媒体消息使用 `sendMediaMessage` 方法。
- 客户端 SDK 发送消息存在频率限制，每秒最多只能发送 5 条消息。

发送普通消息

发送消息前需要构造 Message 消息对象。消息的 content 字段中必须包含普通消息内容（即 MessageContent 的子类），如文本消息（TextMessage）。

调用 RongIM 或 IMCenter 的发送消息方法时，SDK 会触发内置的会话列表和会话页面的更新。

```
String targetId = "会话 Id";
ConversationType conversationType = ConversationType.PRIVATE;
TextMessage messageContent = TextMessage.obtain("消息内容");

Message message = Message.obtain(targetId, conversationType, messageContent);

IMCenter.getInstance().sendMessage(message, null, null, new IRongCallback.ISendMessageCallback() {

@Override
public void onAttached(Message message) {

}

@Override
public void onSuccess(Message message) {

}

@Override
public void onError(Message message, RongIMClient.ErrorCode errorCode) {

}
});
```

sendMessage 中直接提供了用于控制推送通知内容（pushContent）和推送附加信息（pushData）的参数。不过，如果您需要更精细地控制离线推送通知，例如标题、内容、图标、或其他第三方厂商个性化配置，请使用消息推送属性进行配置，详见自定义消息推送通知。请注意，聊天室会话不支持离线消息机制，因此也不支持离线消息转推送。

- 如果发送的消息属于 SDK 内置消息类型，例如 TextMessage，这两个参数可设置为 `null`。一旦消息触发离线推送通知时，融云服务端会使用各个内置消息类型默认的 `pushContent` 值。关于各类型消息的默认推送通知内容，详见用户内容类消息格式。
- 如果消息类型为自定义消息类型，且需要支持离线推送通知，则必须向融云提供 `pushContent` 字段，否则用户无法收到离线推送通知。
- 如果消息类型为自定义消息类型，但不需要支持远程推送通知（例如通过自定义消息类型实现的 App 业务层操作指令），可将 `pushContent` 字段留空。
- `Message` 的推送属性配置 `MessagePushConfig` 的 `pushContent` 和 `pushData` 会覆盖此处配置，并提供更多配置能力，例如自定义推送通知的标题。详见自定义消息推送通知。

参数	类型	说明
message	Message	要发送的消息体。详见消息介绍 中对 Message 的结构说明。
pushContent	String	修改或指定远程消息推送通知栏显示的内容。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见下文自定义消息推送通知。 • 如果希望使用融云默认推送通知内容，可以填 `null`。注意自定义消息类型无默认值。 • 如果要为该条消息指定个性化的离线推送通知，必须在以上任一处向融云提供 `pushContent` 字段内容。 • 如果您自定义的消息类型需要离线推送服务，必须在以上任一处向融云提供 `pushContent` 字段内容，否则用户无法收到离线推送通知。
pushData	String	远程推送附加信息。如果设置该字段，用户在收到远程推送消息时，能通过 io.rong.push.notification.PushNotificationMessage#getPushData()方法获取。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见下文自定义消息推送通知。
callback	ISendMessageCallback	发送消息的回调

发送媒体消息

媒体消息 Message 对象的 content 字段必须传入 MediaMessageContent 的子类对象，表示媒体消息内容。例如图片消息（ImageMessage）、GIF 消息（GIFMessage）等。其他内置媒体消息类型包括文件消息（FileMessage）、位置消息（LocationMessage）、高清语音消息（HQVoiceMessage）、小视频消息（SightMessage），建议先集成相应的 IMKit 插件。

图片消息内容（ImageMessage）支持设置为发送原图。

```
String targetId = "目标 ID";
ConversationType conversationType = ConversationType.PRIVATE;
Uri localUri = Uri.parse("file://图片的路径");//图片本地路径，接收方可以通过 getThumUri 获取自动生成的缩略图 Uri
boolean mIsFull = true; //是否发送原图
ImageMessage mediaMessageContent = ImageMessage.obtain(localUri, mIsFull);

Message message = Message.obtain(targetId, conversationType, mediaMessageContent);
```

发送媒体消息需要使用 sendMediaMessage 方法。SDK 会为图片、小视频等生成缩略图，根据默认压缩配置进行压缩，再将图片、小视频等媒体文件上传到融云默认的文件服务器（文件存储时长），上传成功之后再发送消息。图片消息如已设置为发送原图，则不会进行压缩。

调用 RongIM 或 IMCenter 的发送消息方法时，SDK 会根据发送状态同步更新内置的会话列表和会话页面。

```
IMCenter.getInstance().sendMediaMessage(message, null, null, new IRongCallback.ISendMediaMessageCallback() {
@Override
public void onProgress(Message message, int i) {

}

@Override
public void onCanceled(Message message) {

}

@Override
public void onAttached(Message message) {

}

@Override
public void onSuccess(Message message) {

}

@Override
public void onError(final Message message, final RongIMClient.ErrorCode errorCode) {

}
});
```

sendMediaMessage 中直接提供了用于控制推送通知内容（pushContent）和推送附加信息（pushData）的参数。不过，如果您需要更精细地控制离线推送通知，例如标题、内容、图标、或其他第三方厂商个性化配置，请使用消息推送属性进行配置，详见自定义消息推送通知。请注意，聊天室会话不支持离线消息机制，因此也不支持离线消息转推送。

- 如果发送的消息属于 SDK 内置的媒体消息类型，例如 ImageMessage，这两个参数可设置为 `null`。一旦消息触发离线推送通知时，融云服务端会使用各个内置消息类型默认的 `pushContent` 值。关于各类型消息的默认推送通知内容，详见用户内容类消息格式。
- 如果消息类型为自定义消息类型，且需要支持离线推送通知，则必须向融云提供 `pushContent` 字段，否则用户无法收到离线推送通知。
- 如果消息类型为自定义消息类型，但不需要支持远程推送通知（例如通过自定义消息类型实现的 App 业务层操作指令），可将 `pushContent` 字段留空。
- `Message` 的推送属性配置 `MessagePushConfig` 的 `pushContent` 和 `pushData` 会覆盖此处配置，并提供更多配置能力，例如自定义推送通知的标题。详见自定义消息推送通知。

参数	类型	说明
message	Message	要发送的消息体。消息的 content 字段必须为媒体消息内容 MediaMessageContent。详见消息介绍 中对 Message 的结构说明。
pushContent	String	修改或指定远程消息推送通知栏显示的内容。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见下文自定义消息推送通知。 • 如果希望使用融云默认推送通知内容，可以填 null。注意自定义消息类型无默认值。 • 如果要为该条消息指定个性化的离线推送通知，必须在以上任一处向融云提供 pushContent 字段内容。 • 如果您自定义的消息类型需要离线推送服务，必须在以上任一处向融云提供 pushContent 字段内容，否则用户无法收到离线推送通知。
pushData	String	远程推送附加信息。如果设置该字段，用户在收到远程推送消息时，能通过 io.rong.push.notification.PushNotificationMessage#getPushData()方法获取。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见下文自定义消息推送通知。
callback	ISendMediaMessageCallback	发送多媒体消息的回调

发送多媒体消息并且上传到自己的服务器

SDK 支持由 App 自行处理媒体文件上传逻辑（可上传到自己的服务器），然后再发送消息，同时 SDK 会更新 UI 状态。

您需要使用 sendMediaMessage 方法发送媒体消息，在回调接口 ISendMediaMessageCallbackWithUploader 的 onAttached 回调方法中自行实现媒体文件上传，并上传成功后通知 SDK，并提供回媒体文件的远端地址。

```
String path = "file://图片的路径";
Uri localUri = Uri.parse(path);

ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "会话 Id";
ImageMessage imageMessage = ImageMessage.obtain(localUri, localUri);

Message message = Message.obtain(targetId, conversationType, imageMessage);

RongIM.getInstance().sendMediaMessage(message, null, null, new IRongCallback.ISendMediaMessageCallbackWithUploader() {

@Override
public void onAttached(Message message, final MediaMessageUploader uploader) {
/*上传图片到自己的服务器*/
uploadImg(imgMsg.getPicFilePath(), new UploadListener() {
@Override
public void onSuccess(String url) {
// 上传成功，回调 SDK 的 success 方法，传递回图片的远端地址
uploader.success(Uri.parse(url));
}


@Override
public void onProgress(float progress) {
//刷新上传进度
uploader.update((int) progress);
}


@Override
public void onFail() {
// 上传图片失败，回调 error 方法。
uploader.error();
}
});
}


@Override
public void onError(Message message, RongIMClient.ErrorCode errorCode) {
//发送失败
}


@Override
public void onSuccess(Message message) {
//发送成功
}


@Override
public void onProgress(Message message, int progress) {
//发送进度
}
});
```

参数	类型	说明
message	Message	发送消息的实体
pushContent	String	修改或指定远程消息推送通知栏显示的内容。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见下文自定义消息推送通知。 • 如果希望使用融云默认推送通知内容，可以填 null。注意自定义消息类型无默认值。 • 如果要为该条消息指定个性化的离线推送通知，必须在以上任一处向融云提供 pushContent 字段内容。 • 如果您自定义的消息类型需要离线推送服务，必须在以上任一处向融云提供 pushContent 字段内容，否则用户无法收到离线推送通知。
pushData	String	远程推送附加信息。如果设置该字段，用户在收到远程推送消息时，能通过 io.rong.push.notification.PushNotificationMessage#getPushData()方法获取。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见下文自定义消息推送通知。
callback	ISendMediaMessageCallbackWithUploader	发送多媒体消息并上传到自己服务器的回调。IRongCallback 中的 MediaMessageUploader 可以直接参考 IRongCoreCallback.MediaMessageUploader。

发送定向普通消息

IMCenter 提供 sendDirectionalMessage 方法，支持向群组中特定的某些用户发送消息，会话中其他用户将不会收到此消息。

使用该方法发送普通定向消息前，不需要构造 Message 对象。请在 MessageContent 参数中直接传入 MessageContent 子类示例，例如文本消息（TextMessage）。

```
IMCenter.getInstance().sendDirectionalMessage(conversationType, targetId, messageContent, userIds, pushContent, pushData, callback);
```

参数	类型	说明
conversationType	ConversationType	会话类型，必须为 ConversationType.GROUP
targetId	String	会话 Id
messageContent	MessageContent	消息的具体内容。只支持普通类型消息，不支持多媒体类型消息。
userIds	String[]	会话中将会接收到此消息的用户列表.

参数	类型	说明
pushContent	String	修改或指定远程消息推送通知栏显示的内容。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见下文自定义消息推送通知。 • 如果希望使用融云默认推送通知内容，可以填 null。注意自定义消息类型无默认值。 • 如果要为该条消息指定个性化的离线推送通知，必须在以上任一处向融云提供 pushContent 字段内容。 • 如果您自定义的消息类型需要离线推送服务，必须在以上任一处向融云提供 pushContent 字段内容，否则用户无法收到离线推送通知。
pushData	String	push 附加信息。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见下文自定义消息推送通知。
callback	IRongCallback.ISendMessageCallback	发送消息的回调。

发送定向媒体消息

IMCenter 提供 sendDirectionalMediaMessage 方法，支持向群组中特定的某些用户发送消息，会话中其他用户将不会收到此消息。

使用该方法发送普通定向消息前，不需要构造 Message 对象。请在 MessageContent 参数中直接传入 MessageContent 子类示例，例如文本消息（ImageMessage）。

```
IMCenter.getInstance().sendDirectionalMessage(conversationType, targetId, messageContent, userIds, pushContent, pushData, callback);
```

参数	类型	说明
conversationType	ConversationType	会话类型。必须为 ConversationType.GROUP
targetId	String	会话 ID
messageContent	MediaMessageContent	媒体消息的具体内容。只支持多媒体型消息，不支持普通类型消息。
userIds	String[]	会话中将会接收到此消息的用户列表.
pushContent	String	修改或指定远程消息推送通知栏显示的内容。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见下文自定义消息推送通知。 • 如果希望使用融云默认推送通知内容，可以填 null。注意自定义消息类型无默认值。 • 如果要为该条消息指定个性化的离线推送通知，必须在以上任一处向融云提供 pushContent 字段内容。 • 如果您自定义的消息类型需要离线推送服务，必须在以上任一处向融云提供 pushContent 字段内容，否则用户无法收到离线推送通知。
pushData	String	push 附加信息。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见下文自定义消息推送通知。
callback	IRongCallback.ISendMediaMessageCallback	发送消息的回调。

自定义消息推送通知

IMKit 支持对单条消息的推送行为添加个性化配置（MessagePushConfig），但并未在 UI 上实现。如有需要，您可以在发送时拦截消息添加配置。拦截方式与为消息禁用推送通知一致。

在发送消息时，您可以通过设置消息的 MessagePushConfig 对象，对单条消息的推送行为进行个性化配置。例如：

- 自定义推送标题、推送通知内容
- 自定义通知栏图标
- 添加远程推送附加信息
- 越过接收客户端的配置，强制在推送通知内显示通知内容
- 其他 APNs 或 Android 推送通道支持的个性化配置

相对于发送消息方法输入参数中的 pushContent 和 pushData，MessagePushConfig 中的配置具有更高优先级。发送消息时，如已配置 MessagePushConfig，则优先使用 MessagePushConfig 中的配置。

```
Message message = Message.obtain(mTargetId, mConversationType, textMessage);
MessagePushConfig messagePushConfig = new MessagePushConfig.Builder().setPushTitle(title)
.setPushContent(content)
.setPushData(data)
.setForceShowDetailContent(forceDetail)
.setAndroidConfig(new AndroidConfig.Builder()
.setNotificationId(id)
.setChannelIdHW(hw)
.setCategoryHW("IM")
.setChannelIdMi(mi)
.setChannelIdOPPO(oppo)
.setTypeVivo(vivo ? AndroidConfig.SYSTEM : AndroidConfig.OPERATE)
.setCategoryVivo("IM")
.build())
.setIOSConfig(new IOSConfig(threadId, apnsId)).setTemplateId("")
build();
message.setMessagePushConfig(messagePushConfig);

// 请根据消息类型调用对应的发送方法
```

MessagePushConfig 属性说明

参数	类型	说明
disablePushTitle	boolean	是否屏蔽通知标题。此属性只针目标用户为 iOS 平台时有效，Android 第三方推送平台的通知标题为必填项，所以暂不支持。
pushTitle	String	推送标题，此处指定的推送标题优先级最高。如不设置，可参考 用户内容类消息格式 中对各内置消息类型默认推送通知标题与推送通知内容的说明。
pushContent	String	推送内容。此处指定的推送内容优先级最高。如不设置，可参考 用户内容类消息格式 中对各内置消息类型默认推送通知标题与推送通知内容的说明。
pushData	String	远程推送附加信息。如不设置，则使用消息发送参数中设置的 pushData 值。
forceShowDetailContent	boolean	是否越过目标客户端的配置，强制在推送通知内显示通知内容（pushContent）。 客户端设备可通过 setPushContentShowStatus 设置在接收推送通知时仅显示类似「您收到了一条通知」的提醒。发送消息时，可设置 forceShowDetailContent 为 1 越过该配置，强制目标客户端在此条消息的推送通知中显示推送内容。
iOSConfig	IOSConfig	iOS 平台推送通知配置。目标端设备为 iOS 平台设备时，适用该配置。详细说明见 iOSConfig 属性说明。
androidConfig	AndroidConfig	Android 平台推送通知配置。目标端设备为 Android 平台设备时，适用该配置。详细说明见 AndroidConfig 属性说明。
templateId	String	推送模板 ID。根据目标客户端用户通过 RongIMClient 中的 setPushLanguageCode 设置的语言环境，匹配模板 ID 所对应的模板中设置的语言内容进行推送。未匹配成功时使用默认内容进行推送。 模板内容在控制台 > 自定义推送文案中进行设置，具体操作请参见 配置和使用自定义多语言推送模板。

- iOSConfig 属性说明

参数	类型	说明
threadId	String	iOS 平台通知栏分组 ID，相同的 threadId 推送分为一组（iOS10 开始支持）。
apnsCollapseId	String	iOS 平台通知覆盖 ID，apnsCollapseId 相同时，新收到的通知会覆盖老的通知，最大 64 字节（iOS10 开始支持）。
richMediaUri	String	iOS 推送自定义的通知栏消息右侧图标 URL，需要 App 自行解析 richMediaUri 并实现展示。图片要求：大小 120 * 120px，格式为 png 或者 jpg 格式。SDK 从 5.2.4 版本开始支持携带该字段。
interruptionLevel	String	适用于 iOS 15 及之后的系统。取值为 passive，active（默认），time-sensitive，或 critical，取值说明详见对应的 APNs 的 interruption-level 字段。在 iOS 15 及以上版本中，系统的 “定时推送摘要”、“专注模式” 都可能导致重要的推送通知（例如余额变化）无法及时被用户感知的情况，可考虑设置该字段。SDK 5.6.7 及以上版本支持该字段。

- AndroidConfig 属性说明

参数	类型	说明
notificationId	String	Android 平台 Push 唯一标识，目前支持小米、华为推送平台，默认开发者不需要进行设置，当消息产生推送时，消息的 messageUId 作为 notificationId 使用。
channelIdMi	String	小米推送通知渠道的 ID。详细使用方式及创建方法请参见第三方文档 小米推送消息分类新规。
imageUrlMi	String	（由于小米官方已停止支持该能力，该字段已失效）小米通知类型的推送所使用的通知图片 url。图片要求：大小120 * 120px，格式为 png 或者 jpg 格式。 此属性 5.1.7 及以上版本支持。支持 MIUI 国内版（国内版要求为 MIUI 12 及以上）和国际版。
channelIdHW	String	华为推送通知渠道的 ID。详细使用方式及创建方法请参见第三方文档 华为自定义通知渠道。更多通知渠道信息，请参见 Android 官方文档。
imageUrlHW	String	华为推送通知中自定义的通知栏消息右侧小图片 URL，如果不设置，则不展示通知栏右侧图片。图标文件须小于 512 KB，图标建议规格大小：40dp x 40dp，弧角大小为 8dp，超出建议规格大小的图标会存在图片压缩或显示不全的情况。 此属性 5.1.7 及以上版本支持。
importanceHW	String	华为推送的消息提醒级别。LOW 表示通知栏消息预期的提醒方式为静默提醒，消息到达手机后，无铃声震动。NORMAL 表示通知栏消息预期的提醒方式为强提醒，消息到达手机后，以铃声、震动提醒用户。终端设备实际消息提醒方式将根据 categoryHW 字段取值、或者控制台配置的 category 字段取值，或者华为智能分类结果进行调整。SDK 5.1.3 及以上版本支持该字段。
categoryHW	String	华为推送通道的消息自分类标识，默认为空。category 取值必须为大写字母，例如 IM。App 根据华为要求完成华为自分类权益申请 或 申请特殊权限 后可传入该字段有效。详见华为推送官方文档华为消息分类标准。该字段优先级高于控制台为 App Key 下的应用标识配置的华为推送 Category。SDK 5.4.0 及以上版本支持该字段。
imageUrlHonor	String	荣耀推送通知中用户自定义的通知栏右侧大图标 URL，如果不设置，则不展示通知栏右侧图标。图标文件须小于 512 KB，图标建议规格大小：40dp x 40dp，弧角大小为 8dp，超出建议规格大小的图标会存在图片压缩或显示不全的情况。SDK 5.6.7 及以上版本支持该字段。
importanceHonor	String	荣耀推送的 Android 通知消息分类，决定用户设备消息通知行为。LOW 表示资讯营销类消息。NORMAL（默认值）表示服务与通讯类消息。SDK 5.6.7 及以上版本支持该字段。
typeVivo	String	VIVO 推送服务的消息类别。可选值 0（运营消息） 和 1（系统消息）。该参数对应 VIVO 推送服务的 classification 字段，详见 VIVO 推送消息分类说明。
categoryVivo	String	VIVO 推送服务的消息二级分类。例如 IM（即时消息）。该参数对应 VIVO 推送服务的 category 字段。详细的 category 取值请参见 VIVO 推送消息分类说明。如果指定二级分类 categoryVivo，必须同时指定 typeVivo（系统消息或运营消息）。请注意遵照 VIVO 官方要求，确保二级分类属于 VIVO 系统消息场景或运营消息场景下允许发送的内容。categoryVivo 字段优先级高于控制台为 App Key 下的应用标识配置的 VIVO 推送 Category。SDK 5.4.2 及以上版本支持该字段。
channelIdOPPO	String	OPPO 推送通知渠道的 ID。详细使用方式及创建方法请参见第三方文档 OPPO PUSH 通道升级说明。
channelIdFCM	String	FCM 推送通知渠道的 ID。应用程序必须先创建一个具有此频道 ID 的频道，然后才能收到具有此频道 ID 的任何通知。更多信息请参见安卓官方文档。
collapseKeyFCM	String	FCM 推送的通知分组 ID。SDK 5.1.3 及以上版本支持该字段。注意，如使用该字段，请确保控制台的 FCM 推送配置中推送方式为通知消息方式。
imageUrlFCM	String	FCM 推送的通知栏右侧图标 URL。如果不设置，则不展示通知栏右侧图标。SDK 5.1.3 及以上版本支持该字段。注意，如使用该字段，请确保控制台的 FCM 推送配置鉴权方式为证书，推送方式为通知消息方式。

为消息禁用推送通知

在接收者未上线时，融云服务端默认触发推送服务，将消息通过推送通道下发（服务端是否触发推送也会收到应用级别、用户级别免打扰设置的影响）。

有时 App 用户可能希望在发送消息时就指定该条消息不需要触发推送。IMKit 提供该能力，但并未在 UI 上实现。如有需要，可通过以下方式实现：

1. 设置消息拦截监听器 MessageInterceptor ，在 interceptOnSendMessage 回调中获取 Message 对象。返回 true 表示由 App 自行处理该消息。详见消息拦截。

2. 将 messageConfig 的 disableNotification 属性设置为 true 禁用该条消息的推送通知。接收方再次上线时会通过融云服务端的离线消息缓存（最多缓存 7 天）自动收取单聊、群聊、系统会话消息。

```
message.setMessageConfig(new MessageConfig.Builder().setDisableNotification(true).build());
```

3. 调用发送消息方法重新发送消息。

接收消息　　接收消息

更新时间:2024-08-30

开发者拦截 SDK 接收的消息，并进行相应的业务操作。

消息接收监听器说明

SDK 提供了消息接收监听器 OnReceiveMessageWrapperListener，可接收实时消息或离线消息。

当客户端连接成功后，服务端会将所有离线消息以消息包（Package）的形式下发给客户端，每个 Package 中最多含 200 条消息。客户端会解析 Package 中的消息，逐条上抛并通知应用。

SDK 接收到消息时会触发以下方法。

```
public boolean onReceived(Message message, int left, boolean hasPackage, boolean offline)
```

- `left` 为当前正在解析的消息包（Package）中还剩余的消息条数。
- `hasPackage` 表示当前是否在服务端还存在未下发的消息包（Package）。
- `offline` 表示当前消息是否为离线消息。

同时满足以下条件，表示离线消息已收取完毕：

- `hasPackage` 为 `false`：表示当前正在解析最后一包消息。
- `left` 为 0：表示最后一个消息包中最后一条消息已接收完毕。

从 5.2.3 版本开始，每次连接成功后，离线消息收取完毕时会触发以下回调方法。如果没有离线消息，连接成功后会立即触发。

```
public void onOfflineMessageSyncCompleted() {
//
}
```

设置消息接收监听器

IMKit SDK 在 IMCenter 类中提供了 addOnReceiveMessageListener/ removeOnReceiveMessageListener 方法，支持设置多个消息接收监听器。

添加消息监听器

设置消息接收监听器。所有接收到的消息都会在此接口方法中回调。建议在应用生命周期内注册消息监听。请注意不要重复添加，避免内存泄露。

调用 IMCenter 类 addOnReceiveMessageListener 设置消息接收监听器，回调线程的情况如下：

- 如果 SDK 版本 = 5.3.3，`onReceivedMessage` 回调在非主线程中。
- 如果 SDK 版本 ≦ 5.3.2 或 ≧ 5.3.4，`onReceivedMessage` 回调在主线程中。

```
IMCenter.getInstance().addOnReceiveMessageListener(
new RongIMClient.OnReceiveMessageWrapperListener() {
@Override
public boolean onReceived(Message message, int left, boolean hasPackage, boolean offline) {
return false;
}
});
```

移除消息监听器

SDK 支持移除监听器。为了避免内存泄露，请在不需要监听时将监听器移除。

```
IMCenter.getInstance().removeOnReceiveMessageListener(listener);
```

设置异步消息接收监听器

> 提示
>
> IMKit 从 5.3.4 开始提供以下接口，并且 IMKit SDK 默认使用 addAsyncOnReceiveMessageListener。

IMKit SDK 在 IMCenter 类中提供了 addAsyncOnReceiveMessageListener/ removeAsyncOnReceiveMessageListener 方法，支持设置多个消息接收监听器。

添加异步消息监听器

如果 SDK 版本 ≧ 5.3.4，可调用 IMCenter 类的 addAsyncOnReceiveMessageListener 设置消息接收监听器，onReceivedMessage 回调在非主线程中。所有接收到的消息都会在此接口方法中回

调。建议在应用生命周期内注册消息监听。请注意不要重复添加，避免内存泄露。

```
IMCenter.getInstance().addAsyncOnReceiveMessageListener(
new RongIMClient.OnReceiveMessageWrapperListener() {
@Override
public boolean onReceived(Message message, int left, boolean hasPackage, boolean offline) {
return false;
}
});
```

移除异步消息监听器

如果 SDK 版本 ≥ 5.3.4，可调用 IMCenter 类 removeAsyncOnReceiveMessageListener 移除监听器。为了避免内存泄露，请在不需要监听时将监听器移除。

```
IMCenter.getInstance().removeAsyncOnReceiveMessageListener(listener);
```

禁用排重机制

消息排重机制会在 SDK 接收单聊、群聊、系统消息时自动去除内容重复消息。当 App 本地存在大量消息，SDK 默认的排重机制可能会因性能问题导致收消息卡顿。因此在接收消息发生卡顿问题时，App 可以尝试禁用 SDK 的排重机制。

为什么接收消息可能出现消息重复

发送端处于弱网情况下可能出现该问题。A 向 B 发送消息后，消息成功到达服务端，并成功下发到接收者 B。但 A 由于网络等原因可能未收到服务端返回的 ack，导致 A 认为没有发送成功。此时如果 A 重发消息（IMKit 默认具有失败自动重发机制，建议同时关闭），此时 B 就会收到与之前重复的消息（消息内容相同，但 Message UID 不同）。

关闭消息排重机制

单聊、群聊、系统消息使用 IMLib 的核心类 RongCoreClient 中的 setCheckDuplicateMessage 方法（要求 SDK 版本 ≥ 5.3.4），禁用消息排重行为。请在 SDK 初始化之后，建立 IM 连接之前完成以下配置：

```
boolean enableCheck = false // 关闭消息排重
RongCoreClient.getInstance().setCheckDuplicateMessage(enableCheck)
```

> 聊天室消息从 5.8.2 版本开始支持关闭消息排重。在 RongChatRoomClient 中提供。

请在 SDK 初始化之后，建立 IM 连接之前调用。多次调用以最后一次为准。

```
boolean enableCheck = false // 关闭消息排重
RongChatRoomClient.getInstance().setCheckChatRoomDuplicateMessage(enableCheck)
```

关闭失败重发机制

IMKit SDK 默认启用了消息失败自动重发机制。在禁用消息排重机制后，为避免收到 UID 重复的消息，建议同时在 App 中禁用 IMKit 的失败重发机制。

请在 SDK 初始化之后，建立 IM 连接之前完成以下配置：

```
RongConfigCenter.conversationConfig().rc_enable_resend_message = false;
```

获取历史消息 获取历史消息

更新时间:2024-08-30

IMKit SDK 默认已经实现了一整套消息的获取和展示逻辑，使用默认会话列表和会话页面时，不需要额外调用获取消息相关 API。

如果您有自定义需求，可以调用 IMLib SDK 中获取消息相关的 API。

请注意，IMLib 方法并不提供页面展示和刷新能力，您需要根据业务需求自定义实现页面展示和刷新。

- 获取本地数据库中的历史消息。

  IMLib 方法：getHistoryMessages()

- 获取远端服务器的历史消息。

  IMLib 方法：getRemoteHistoryMessages()

拦截消息　　拦截消息

更新时间:2024-08-30

IMKit 支持设置消息拦截器 `MessageInterceptor`，可在消息发送前、发送后、接收时、插入本地数据库前进行拦截，方便应用程序进行自定义处理。

消息拦截器说明

`MessageInterceptor` 是一个接口类，包含以下方法。

方法	说明
interceptOnSendMessage	拦截待发送的消息。该方法可拦截 Message 对象。
interceptOnSentMessage	拦截发送成功的消息。该方法可拦截 Message 对象。
interceptOnInsertOutgoingMessage	interceptOnInsertOutgoingMessage 拦截待插入本地消息数据库的发送方向的消息的消息内容，提供两个重载方法： • 支持 `ResultCallback<Message> callback` 的方法用于在图片消息、GIF 消息被插入本地数据库前进行拦截。详见下文**如何拦截 IMKit 内置会话页面发送的图片和 GIF 消息**。要求 SDK ≧ 5.2.4。 • 不支持 `callback` 参数的方法用于拦截待插入本地消息数据库的发送方向的消息的消息内容。**注意**，不可用于拦截 IMKit 内置会话页面发送的图片消息和 GIF 消息。
interceptOnInsertIncomingMessage	拦截待插入本地消息数据库的接收方向的消息的消息内容。
interceptReceivedMessage	拦截接收的消息。SDK 连接后将先接收离线消息。`hasPackage` 标识是否还有剩余的离线消息包，`left` 表示当前正在接收的离线消息包剩余多少条。`hasPackage` 为 `false` 且 `left` 为 0 表示离线消息已接收完毕，当前拦截的消息为实时消息。该方法可拦截 Message 对象。

在上述所有方法中返回 `false` 表示 App 仅需要拦截处理，处理完毕后由 SDK 完成后续流程。返回 `true` 表示 App 需要在拦截后自行处理后续流程，例如 App 希望在处理完毕后自行发送修改后的消息，可返回 `true`。

设置消息拦截器

使用 IMCenter 的 `setMessageInterceptor` 设置消息拦截器。代码示例说明如下：

- 在 `interceptOnSendMessage` 方法中拦截并修改消息推送内容
- 在 `interceptOnInsertOutgoingMessage` 方法中拦截并修改图片和 GIF 消息扩展

```
private void initInterceptor() {
IMCenter.getInstance().setMessageInterceptor(
new MessageInterceptor() {
@Override
public boolean interceptReceivedMessage(
Message message,
int left,
boolean hasPackage,
boolean offline) {
return false;
}

@Override
public boolean interceptOnSendMessage(Message message) {
MessageContent messageContent = message.getContent();
String language = mContext.getResources().getConfiguration().locale.getLanguage();
if (language.endsWith("en")) { // 手机系统语言为英文
if (messageContent instanceof VoiceMessage) {
MessagePushConfig messagePushConfig = message.getMessagePushConfig();
if (messagePushConfig == null){
messagePushConfig = new MessagePushConfig();
message.setMessagePushConfig(messagePushConfig);
}
messagePushConfig.setPushContent("[voice]");// 语音消息时推送显示为 "\[voice\]"
}
}
return false;

}

@Override
public boolean interceptOnSentMessage(Message message) {
if (message != null) {
message.setMessageConfig(null);
}

return false;
}


@Override
public boolean interceptOnInsertOutgoingMessage(
Conversation.ConversationType type,
String targetId,
Message.SentStatus sentStatus,
MessageContent content,
long time) {
return false;
}


@Override
public boolean interceptOnInsertOutgoingMessage(Conversation.ConversationType type, String targetId, Message.SentStatus sentStatus,
MessageContent content, long time, RongIMClient.ResultCallback<Message> callback) {
//含 UI 集成设置图片消息可扩展时，此方法 return true，构建 message 将可扩展属性设置为 true，并调用 callback.onSuccess(message);
Message message=Message.obtain(targetId,type,content);
message.setCanIncludeExpansion(true);
callback.onSuccess(message);
return true;
}

@Override
public boolean interceptOnInsertIncomingMessage(
Conversation.ConversationType type,
String targetId,
String senderId,
Message.ReceivedStatus receivedStatus,
MessageContent content,
long time) {
return false;
}
});

}
```

如何拦截 IMKit 内置会话页面发送的图片和 GIF 消息

> ⓘ 提示
>
> 要求 SDK 版本 ≧ 5.2.4。

使用 IMKit 内置会话页面发送图片和 GIF 消息时，SDK 会先在内部调用插入消息方法将消息插入本地消息数据库，再调用发送消息方法。如果应用程序有以下需求：

- 对图片消息和 GIF 消息设置消息扩展，需要打开图片消息和 GIF 消息的可扩展属性（`setCanIncludeExpansion(true)`）。
- 修改图片消息和 GIF 消息的部分数据，例如缩略图、或者消息内容体内的 `extra` 字段等。
- 彻底拦截图片消息和 GIF 消息，由应用程序自定义处理。

必须在消息插入本地数据库前拦截图片、GIF 消息，使用以下方法进行拦截：

```
default boolean interceptOnInsertOutgoingMessage(
Conversation.ConversationType type,
String targetId,
Message.SentStatus sentStatus,
MessageContent content,
long time,
RongIMClient.ResultCallback<Message> callback)
```

应用程序处理完毕后：

- 如需 SDK 继续发送被拦截的消息，需要调用 `callback.onSuccess(message)` 表示交给 SDK 继续发送该消息，**此时必须在** `interceptOnInsertOutgoingMessage` **方法中返回** `true`，**否则会发出两条内容重复的消息。**
- 如需彻底拦截，则调用 `callback.onError(IRongCoreEnum.CoreErrorCode.PARAMETER_ERROR)`，并在 `interceptOnInsertOutgoingMessage` 中返回 `true`。

删除消息　　删除消息

更新时间:2024-08-30

单聊会话、群聊会话的参与者可删除会话中的消息。IMKit 会话页面默认已实现了长按删除消息的功能，支持仅删除单条本地消息，或同步删除本地和远端的单条消息。

您可以修改 IMKit 会话页面长按消息菜单中删除按钮的行为，详见会话页面。如果 IMKit 的已有实现无法满足您的需求，可以直接使用 IMKit SDK 提供的删除消息接口。调用 IMKit 的删除 API 会同时触发会话列表和会话页面的刷新。

> 提示
>
> - App 用户的单聊会话、群聊会话、系统会话的消息默认仅存储在本地数据库中，仅支持从本地删除。如果 App（App Key/环境）已开通 单群聊消息云端存储，该用户的消息还会保存在融云服务端（默认 6 个月），可从远端历史消息记录中删除消息。
> - 针对单聊会话、群聊会话，如果通过任何接口以传入时间戳的方式删除远端消息，服务端默认不会删除对应的离线消息补偿（该机制仅会在打开多设备消息同步开关后生效）。此时如果换设备登录或卸载重装，仍会因为消息补偿机制获取到已被删除的历史消息。如需彻底删除消息补偿，请提交工单，申请开通删除服务端历史消息时同时删除多端补偿的离线消息。如果以传入消息对象的方式删除远端消息，则服务端一定会删除消息补偿中的对应消息。

同时删除本地与远端消息

使用以下接口可以同时删除本地数据库和服务端消息：

- 使用 `deleteMessages` 方法，并将是否删除远端消息的参数（`cleanRemote`）置为 `true`。详见「删除本地消息」下的通过时间戳删除。
- 使用 `deleteRemoteMessages` 方法，详见「删除服务端消息」下的 通过消息删除。

删除本地消息

消息送达后会直接存入本地数据库，您可以调用接口从本地数据库中删除消息。删除消息同时触发会话列表和会话页面的刷新。

> 提示
>
> 如果已经开通单群聊历史消息云端存储服务，从本地删除消息对服务端存储的历史消息无影响，客户端仍然可以从服务端拉取到本地已删除的历史消息。

如果希望仅删除本地消息，可使用 IMKit SDK 提供的以下能力：

- 通过指定消息 ID 等参数，删除指定单个会话在本地的消息记录中的一条或一组消息。
- 通过指定会话 ID 等参数，删除指定单个会话在本地的所有消息。
- 通过时间戳、会话 ID 等参数，删除指定单个会话在本地消息记录中早于指定时间点的消息。**注意**：通过时间戳删除消息的接口提供一个删除远端消息的开关（`cleanRemote`），支持同时删除服务端的历史消息。

通过消息 ID 删除

使用 IMCenter 或 RongIM 的以下方法可通过消息 ID 删除指定会话在本地消息数据中的一条或一组消息，删除成功后会刷新会话和会话列表页面。请确保所提供的消息 ID 均属于同一会话。

```
IMCenter.getInstance().deleteMessages(conversationType, targetId, messageIds，callback);
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 Id
messageIds	int[]	待删除的消息 ID 数组
callback	IRongCallback.ResultCallback<Boolean>	接口回调

通过会话删除

使用 IMCenter 或 RongIM 的以下方法可删除指定单个会话在本地数据库中的全部消息。删除成功后会刷新会话和会话列表页面。

```
ConversationType conversationType = ConversationType.PRIVATE; //此处以单聊会话为例
String targetId = "会话 Id";

IMCenter.getInstance().deleteMessages(conversationType, targetId, new RongIMClient.ResultCallback<Boolean>() {
/**
* 删除消息成功回调
*/
@Override
public void onSuccess(Boolean bool) {

}
/**
* 删除消息失败回调
* @param errorCode 错误码
*/
@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 Id
callback	IRongCallback.ResultCallback<Boolean>	接口回调

通过时间戳删除

使用 IMCenter 的以下方法可通过时间戳、会话 ID 等参数，删除指定单个会话在本地消息记录中早于指定时间点的消息。删除成功后，会话列表页会刷新并删除此会话。

您需要提供一个时间戳，用于指定所需删除消息的时间范围。所有发送时间小于等于该时间戳的消息将被删除。如果这个时间戳值为 0，则删除该会话中的全部消息。

> 提示
>
> 该方式同时提供一个开关（cleanRemote），可以用来设置是否需要同时删除服务端的对应消息。您也可以使用同时删除本地与远端消息中介绍的其他方法。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "会话 Id";
//此处以删除当前时间戳之前的消息为例，您也可以传入会话中最后一条消息的发送时间来进行删除操作。
long recordTime = System.currentTimeMillis();

IMCenter.getInstance().cleanHistoryMessages(conversationType, targetId, recordTime, false, new RongIMClient.OperationCallback() {
/**
* 删除消息成功回调
*/
@Override
public void onSuccess() {

}
/**
* 删除消息失败回调
* @param errorCode 错误码
*/
@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 Id
recordTime	long	时间戳。 传 0 清除所有消息；非 0 则清除小于等于 recordTime 的消息
cleanRemote	boolean	是否删除服务器端消息。传 `true` 删除服务端消息。传 `false` 不删除。
callback	OperationCallback	接口回调

删除服务端消息

IMKit SDK 删除服务端消息的接口会**同时从服务端与本地删除消息**。如果希望仅删除服务端消息，请另行参考以下方法：

- 使用即时通讯服务端 API。详见即时通讯服务端 API 文档消息清除。
- 直接使用 IMLib 中提供的方法，删除指定会话中早于某个时间点的消息。参见通过时间戳删除。注意，IMLib 方法并不提供页面刷新能力，您需要根据业务需求自定义通知机制进行页面刷新。

如果希望 IMKit 同时删除本地与服务端消息，可使用以下方法：

- 使用上文描述的 `deleteMessages` 方法，传入时间戳、会话 ID 等参数，设置同时删除远端（`cleanRemote` 为 `true`），删除指定单个会话中早于指定时间点的消息。详见上文通过消息 ID 删除。
- 使用 `deleteRemoteMessages`，通过指定消息对象等参数，删除指定单个会话在本地与远端的消息记录中的一条或一组消息。详见下文通过消息删除。

通过消息删除

使用 IMCenter 的以下方法可通过指定会话类型、会话 ID 和消息对象数组，同时删除服务端与本地数据库中单个会话内的一条或者一组消息。删除成功后会触发页面刷新，会话或会话列表页面会同时

移除被删消息。请确保所提供的消息 ID 均属于同一会话。

```
ConversationType conversationType = ConversationType.PRIVATE;//此处以单聊会话为例。
String targetId = "会话 Id";
Message[] messages = {message1, message2};

IMCenter.getInstance().deleteRemoteMessages(conversationType, targetId, messages, new RongIMClient.OperationCallback() {
/**
* 删除消息成功回调
*/
@Override
public void onSuccess() {

}
/**
* 删除消息失败回调
* @param errorCode 错误码
*/
@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型。不支持聊天室。
targetId	String	会话 Id
messages	Message[]	要删除的消息数组
callback	IRongCallback.ResultCallback<Boolean>	接口回调

出于安全与性能考虑，我们对接口做出了一些限制。

- 首先，从客户端接口删除服务端数据的操作具有一定的危险性，因此客户端接口一次删除操作仅能删除单个会话内的消息。
- 其次，出于性能考虑，对于 Android 端，一次最多删除 100 条消息。

撤回消息　　撤回消息

更新时间:2024-08-30

IMKit SDK 默认已经实现了一套消息撤回和展示逻辑，不需要额外调用会话相关 API。如果已有实现无法满足您的需求，可以使用 IMCenter 中相关 API。

撤回消息

您可以在自定义页面调用以下方法撤回消息，该方法会同时触发会话列表和会话页面的刷新。

```
IMCenter.getInstance().recallMessage(message, pushContent, callback)
```

参数	类型	说明
message	Message	要撤回的消息。
pushContent	String	消息被撤回时，通知栏显示的内容。
callback	ResultCallback	撤回消息的结果回调。onSuccess 中会返回替换后撤回提示小灰条消息（RecallNotificationMessage ），您可以根据需要在界面展示。

监听他人撤回消息事件

您可以添加监听器，监听已接收的消息被撤回的事件。

```
private RongIMClient.OnRecallMessageListener mRecallMessageListener =
new RongIMClient.OnRecallMessageListener() {
@Override
public boolean onMessageRecalled(
Message message, RecallNotificationMessage recallNotificationMessage) { }
};

IMCenter.getInstance().addOnRecallMessageListener(mRecallMessageListener);

// 不需要时可移除
IMCenter.getInstance().removeOnRecallMessageListener(mRecallMessageListener);
```

插入消息　　插入消息 功能描述

更新时间:2024-08-30

开发者可通过下面方法在会话页面内插入一条消息。通过此方法插入的消息，会将消息实体对应的内容插入数据源中，并更新 UI。

插入发送消息

向本地会话中插入一条发送方向的消息。此消息必须为客户端会存储的消息类型，参考消息介绍文档的「了解消息注解」。

该方法只是将消息存储在 SDK 本地数据库中，不会实际发送给服务器和对方。

插入消息后，会自动刷新 UI 界面。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "user1";
SentStatus sentStatus = SentStatus.SENT;
TextMessage content = TextMessage.obtain("这里是消息内容");
String sentTime = System.currentTimeMillis();

IMCenter.getInstance().insertOutgoingMessage(conversationType, targetId, sentStatus, content, sentTime, new RongIMClient.ResultCallback<Message>() {

/**
* 成功回调
* @param message 插入的消息
*/
@Override
public void onSuccess(Message message) {

}

/**
* 失败回调
* @param errorCode 错误码
*/
@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID。详见消息介绍中关于 Target ID 的说明。
sentStatus	Message.SentStatus	发送状态
content	MessageContent	消息内容
sentTime	long	消息的发送时间。 消息列表页面会按照此时间排序并展示消息。
callback	IRongCallback.ResultCallback<Message>	回调接口

插入接收消息

向本地会话中插入一条接收方向的消息，此消息属性必须为存储类型，参考消息属性说明。

该方法只是将消息存储在 SDK 本地数据库中，不会发送给服务器。

插入消息后，会自动刷新 UI 界面。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "user1";
String senderUserId = "模拟发送方的 ID";
ReceivedStatus receivedStatus = new ReceivedStatus(0x1);
TextMessage content = TextMessage.obtain("这是一条插入数据");
String sentTime = System.currentTimeMillis();

IMCenter.getInstance().insertIncomingMessage(conversationType, targetId, senderUserId, receivedStatus, content, sentTime, new RongIMClient.ResultCallback<Message>() {
/**
* 成功回调
* @param message 插入的消息
*/
@Override
public void onSuccess(Message message) {

}

/**
* 失败回调
* @param errorCode 错误码
*/
@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	必填	说明
conversationType	ConversationType	是	会话类型.
targetId	String	是	Target ID 用于标识会话，称为目标 ID 或会话 ID。注意，因为单聊业务中始终使用对端用户 ID 作为标识本端会话的 Target ID，因此在单聊会话中插入本端接收的消息时，Target ID 始终是单聊对端用户 ID。群聊、超级群的会话 ID 分别为群组 ID、超级群 ID。详见消息介绍中关于 Target ID 的说明。
senderUserId	String	是	发送方 ID。
receivedStatus	Message.ReceivedStatus	是	接收状态。
content	MessageContent	是	消息内容
sentTime	long	是	消息的发送时间。 消息列表页面会按照此时间排序并展示消息。
callback	IRongCallback.ResultCallback<Message>	是	回调接口

搜索消息 搜索消息

更新时间:2024-08-30

您可以通过 IMLib 中提供的消息搜索相关 API 实现消息搜索功能。

IMKit 中没有提供消息搜索的功能入口及展示页面，您需要根据业务需求在应用层实现消息搜索的 UI 展示。

IMLib 提供三种搜索消息的方法：

- 根据关键字搜索消息

  IMLib 方法：searchMessages()

- 根据用户 Id 搜索

  IMLib 方法：searchMessagesByUser()

- 根据关键字，搜索指定会话中指定时间段内的消息

  IMLib 方法：searchMessages()

自定义消息类型　　自定义消息类型

更新时间:2024-08-30

IMKit 支持自定义的消息类型（区别于内置消息类型），并支持修改内置消息类型与自定义消息类型在 IMKit SDK 会话页面的展示形式。

创建自定义消息类型

除了使用 SDK 内置消息类型外，还可以根据自己的业务需求自定义消息。

> 提示
>
> 关于如何创建自定义消息类型，详见 IMLib SDK 的自定义消息类型。

仅当自定义消息的 `persistentFlag` 为以下值时，可在 IMKit 的会话页面中展示：

- `MessageTag.ISCOUNTED`
- `MessageTag.ISPERSISTED`

如果自定义消息类型带有以上属性，则必须为该自定义消息创建展示模板，IMKit SDK 会调用此模板进行消息的展示。否则自定义消息将被展示为 “当前版本不支持查看此消息”。

为自定义消息创建和注册展示模版

如果您创建了自定义消息类型，且需要将消息展示在会话界面中，必须创建对应的消息展示模板，否则 SDK 无法正常展示该类型消息。以下步骤与修改融云预置消息的展示样式的步骤大致相同，如有需要，您也可以同时参考替换内置消息默认展示模板，了解内置的默认消息类型的消息展示模板的实现，以及如何创建自定义消息展示模板。

步骤一：创建自定义的消息模版

所有的消息展示模板都继承自 `BaseMessageItemProvider`。App 需要继承 `BaseMessageItemProvider`，创建一个消息展示模板类。

1. 创建一个继承自 `BaseMessageItemProvider<CustomMessage>` 的子类，例如 `CustomMessageProvider`。

   以下是创建消息展示模板 `CustomMessageProvider` 的完整示例：

```
public class CustomMessageProvider extends BaseMessageItemProvider<CustomMessage> {

public CustomMessageProvider(){
mConfig.showReadState = true; // 修改模板属性，此处为该模板启用了在单聊会话中启用消息已读回执状态显示功能。
mConfig.xxx = ...; //此处省略对模板其它属性的配置
}
/**
* 创建 ViewHolder
* @param parent 父 ViewGroup
* @param viewType 视图类型
* @return ViewHolder
*/

@Override
protected ViewHolder onCreateMessageContentViewHolder(ViewGroup parent, int viewType) {
View view =
LayoutInflater.from(parent.getContext())
.inflate(R.layout.xxx, parent, false);
return new ViewHolder(parent.getContext(), view);
}

/**
* 设置消息视图里各 view 的值
* @param holder ViewHolder
* @param parentHolder 父布局的 ViewHolder
* @param t 此展示模板对应的消息
* @param uiMessage {@link UiMessage}
* @param position 消息位置
* @param list 列表
* @param listener ViewModel 的点击事件监听器。如果某个子 view 的点击事件需要 ViewModel 处理，可通过此监听器回调。
*/
@Override
protected void bindMessageContentViewHolder(ViewHolder holder, ViewHolder parentHolder, CustomMessage customMessage, UiMessage uiMessage, int position,
List<UiMessage> list, IViewProviderListener<UiMessage> listener) {

}

/**
* 处理会话页面上的消息点击事件
* @param holder ViewHolder
* @param t 自定义消息
* @param uiMessage {@link UiMessage}
* @param position 位置
* @param list 列表数据
* @param listener ViewModel 的点击事件监听器。如果某个子 view 的点击事件需要 ViewModel 处理，可通过此监听器回调。
* @return 点击事件是否被消费
*/
@Override
protected boolean onItemClick(ViewHolder holder, CustomMessage customMessage, UiMessage uiMessage, int position, List<UiMessage> list,
IViewProviderListener<UiMessage> listener) {
return false;
}
/** 处理会话页面上的消息长按事件 */
@Override
protected boolean onItemLongClick(ViewHolder holder, CustomMessage customMessage, UiMessage uiMessage, int position, List<UiMessage> list,
IViewProviderListener<UiMessage> listener) {
return false;
}
/**
* 根据消息内容，判断是否为本模板需要展示的消息类型
*
* @param messageContent 消息内容
* @return 本模板是否处理。
*/
@Override
protected boolean isMessageViewType(MessageContent messageContent) {
return messageContent instanceof CustomMessage;
}

/**
* 在会话列表页某条会话最后一条消息为该类型消息时，会话里需要展示的内容。
* 比如：图片消息在会话里需要展示为"图片"，那返回对应的字符串资源即可。
* @param context 上下文
* @param t 消息内容
* @return 会话里需要展示的字符串资源
*/
@Override
public Spannable getSummarySpannable(Context context, CustomMessage customMessage) {
return new SpannableString("会话列表content位置展示内容");
}
}
```

2. SDK 提供了一个消息展示配置类 `MessageItemProviderConfig`，包含以下属性配置：

消息展示模板属性	默认值	描述
showPortrait	true	是否显示头像。
centerInHorizontal	false	消息内容是否横向居中。
showWarning	true	是否显示未发送成功警告。
showProgress	true	是否显示发送进度。
showSummaryWithName	true	是否在会话的内容体里显示发送者名字。
showReadState	false	单聊会话中是否在消息旁边显示已读回执状态。

消息展示模板属性	默认值	描述
showContentBubble	true	是否需要展示气泡。

在自定义模板的构造方法中，可以获取到基类 MessageItemProviderConfig 变量 mConfig，从而修改模板属性。如下所示：

```
public CustomMessageProvider(){
mConfig.showReadState = true; //修改模板属性，此处为该模板启用了在单聊会话中启用消息已读回执状态显示功能。
mConfig.xxx = ...; //此处省略对模板其它属性的配置
}
```

3. 必须实现 isMessageViewType ，传入需要展示的消息类型的 MessageContent，该模板才能与需要展示的消息类型进行绑定。
4. 如需处理消息点击、长按事件，请实现 onItemClick 与 onItemLongClick。App 可以通过会话页面上的 onMessageClick 与 onMessageLongClick 监听消息点击与长按事件。详见页面事件监听。

步骤 2：注册展示模板

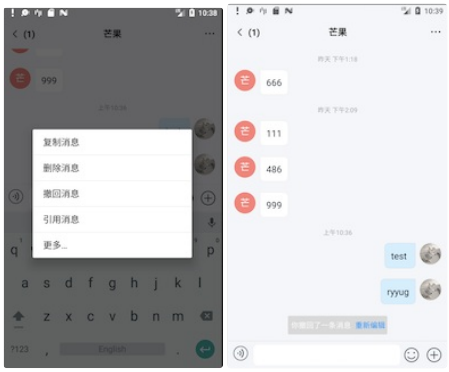
将新建的自定义消息的展示模板提供给 SDK。

```
RongConfigCenter.conversationConfig().addMessageProvider(new CustomMessageProvider());
```

自定义长按消息菜单 自定义长按消息菜单

更新时间:2024-08-30

用户在会话页面长按消息可打开弹窗，根据当前消息类型、会话类型提供不同选项。您可以在自定义菜单选项的显示名称、顺序、以及自行增删菜单选项。



自定义长按消息弹窗的菜单选项

1. 监听会话页面的长按消息事件（onMessageLongClick）。

```
IMCenter.setConversationClickListener(
new ConversationClickListener() {
@Override
public boolean onUserPortraitClick(
Context context,
Conversation.ConversationType conversationType,
UserInfo user,
String targetId) {
return false;
}

@Override
public boolean onUserPortraitLongClick(
Context context,
Conversation.ConversationType conversationType,
UserInfo user,
String targetId) {
return false;
}

@Override
public boolean onMessageClick(
Context context, View view, Message message) {
return false;
}

@Override
public boolean onMessageLongClick(
Context context, View view, Message message) {
return false;
}

@Override
public boolean onMessageLinkClick(
Context context, String link, Message message) {
return false;
}

@Override
public boolean onReadReceiptStateClick(
Context context, Message message) {
return false;
}
});
```

2. 通过 MessageItemLongClickActionManager 修改会话页面中的长按消息的操作选项（MessageItemLongClickAction）。您可以自行增删菜单选项。长按消息的操作选项，或调整菜单选项的显示名称、顺序。

   - MessageItemLongClickActionManager 在初始化方法 initCommonMessageItemLongClickActions 中添加了部分操作选项。您可以使用 addMessageItemLongClickAction 添加选项。可参考 IMKit 源码中的 MessageItemLongClickActionManager.java。
   - 如需删除已有选项时，需先获取到要删除选项的对象，然后调删除接口 removeMessageItemLongClickAction。以去除默认已经添加的「删除消息」选项为例：

```
List<MessageItemLongClickAction> clickActions = MessageItemLongClickActionManager
.getInstance().getMessageItemLongClickActions();
Iterator<MessageItemLongClickAction> iterator = clickActions.iterator();
String delActionTitle = getString(R.string.rc_dialog_item_message_delete);
while (iterator.hasNext()) {
MessageItemLongClickAction clickAction = iterator.next();
boolean isDelAction = delActionTitle.equals(clickAction.getTitle(this));
if (isDelAction) {
iterator.remove();
break;
}
}
```

MessageItemLongClickAction 类属性如下表所示。

属性	类型	描述
title	String	显示名称。
listener	MessageItemLongClickListener	长按消息监听器。
priority	int	优先级越高，排在越前面（由上到下的顺序）。默认全是 0，按添加顺序排列。
filter	Filter	控制是否会被显示出来的过滤器。

自定义消息多选操作菜单

在长按消息弹窗的菜单选项选择更多后，SDK 进入消息多选模式，多选模式下默认提供了转发和删除按钮。您可以增删已有按钮、添加自定义按钮。

1. 实现 IClickActions 接口。

```
public class CustomClickActions implements IClickActions {
/**
* 获取点击按钮的图标
*
* @param context 上下文
* @return 图片的Drawable, 如需高亮或者置灰，则返回类型为selector, 分别显示enable或者disable状态下的drawable
*/
@Override
public Drawable obtainDrawable(Context context) {
return null;
}
/**
* 图标按钮点击事件
*
* @param curFragment 当前 Fragment, 请注意不要持有该 fragment, 否则容易引起内存泄露。
*/
@Override
public void onClick(Fragment curFragment) {

}
/**
* 是否过滤。可以根据业务场景，设置某些会话下过滤掉此按钮，即不进行显示。
* @param message 消息
* @return 返回true，表示过滤掉，页面上不显示此 item。
*/
@Override
public boolean filter(UiMessage message) {
return false;
}
}
```

2. 将自定义按钮配置到 SDK 中。

```
RongConfigCenter.conversationConfig().addMoreClickAction(0, new CustomClickAction());
```

3. 根据业务需要，彻底移除某内置按钮。以下示例以移除转发按钮为例。

```
List<IClickActions> clickActions = RongConfigCenter.conversationConfig().getMoreClickActions();
List<IClickActions> removeActions = new ArrayList<>();
for (IClickActions action : clickActions){
if (action instanceof ForwardClickActions){
RongConfigCenter.conversationConfig().removeMoreClickAction(action);
}
}
```

> 提示
>
> 移除“转发”按钮仅在 IMKit 合并转发功能关闭的情况下生效。

自行控制多选状态

您可以自行控制多选状态，或获取多选状态下的数据。

进入多选状态：

```
MessageViewModel messageViewModel = ViewModelProviders.of(fragment).get(MessageViewModel.class);
messageViewModel.enterEditState();
```

关闭多选状态：

```
MessageViewModel messageViewModel = ViewModelProviders.of(fragment).get(MessageViewModel.class);
messageViewModel.quitEditMode();
```

获取选中的信息：

```
MessageViewModel messageViewModel = ViewModelProviders.of(fragment).get(MessageViewModel.class);
List<UiMessage> selectedMessages = messageViewModel.getSelectedUiMessages();
```

监听多选状态变化

通过 MessageViewModel 获取多选状态的 LiveData, 可以监听多选状态的变化。

```
MessageViewModel messageViewModel = ViewModelProviders.of(fragment).get(MessageViewModel.class);
messageViewModel.getEditStatusLiveData().observe(this, new Observer<Boolean>() {
@Override
public void onChanged(Boolean aBoolean) {
//根据业务需要，进行 UI 更新。
}
});
```

删除会话　　删除会话

更新时间:2024-08-30

IMKit 默认在长按会话时显示以下弹窗，实现了删除会话功能。

如果已有实现无法满足您的需求，可以使用 IMCenter 提供的以下 API：

删除指定会话

从会话列表移除会话项目，但不删除会话内的历史消息。该方法会自动触发会话列表页面刷新。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "会话 Id";

IMCenter.getInstance().removeConversation(conversationType, targetId, new ResultCallback<Boolean>() {

@Override
public void onSuccess(Boolean success) {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 Id
callback	ResultCallback<Boolean>	回调接口

> 提示
>
> 该方法仅从会话列表移除会话项目，但不会删除会话内的历史消息。如果会话内再来一条消息，该会话会重新出现在列表中。如果需要移除会话并删除会话内的消息，必须同时调用消息的 API，您需要同时删除本地与远端的历史消息。详见删除消息。

按类型删除会话

从本地数据库中删除指定会话类型的所有会话，并删除这些会话内的消息。IMKit 没有清除全部会话方法，您可以通过传入所有需要删除的类型来实现该效果。该方法会自动触发会话列表页面刷新。

```
Conversation.ConversationType[] mConversationTypes = {
Conversation.ConversationType.PRIVATE,
Conversation.ConversationType.GROUP
};

IMCenter.getInstance().clearConversations(new RongIMClient.ResultCallback() {

@Override
public void onSuccess(Object object) {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
}, mConversationTypes);
```

参数	类型	说明
callback	ResultCallback	移除会话是否成功的回调
conversationTypes	ConversationType ...	需要清空的会话类型列表

获取会话 获取会话

更新时间:2024-08-30

IMKit SDK 默认已经实现了一整套会话获取和展示逻辑，使用默认会话列表和会话页面时，不需要额外调用会话相关 API。如果已有实现无法满足您的需求，可以使用 IMLib SDK 中相关 API，例如：

- `RongIMClient#getConversationListByPage()`：获取会话列表
- `RongCoreClient#getUnreadConversationList()`：获取未读会话列表
- `RongIMClient#getConversation()`：获取指定会话

> 提示
>
> 具体使用方法请参见 IMLib 文档 获取会话。注意，IMLib 的方法并不提供页面刷新能力，您需要根据业务需求自定义通知机制进行页面刷新。

多端同步免打扰/置顶 多端同步免打扰/置顶

更新时间:2024-08-30

SDK 提供了会话状态（置顶或免打扰）同步机制，通过设置会话状态同步监听器，当在其它端修改会话状态时，可在本端实时监听到会话状态的改变。

监听器说明

RongIMClient 中提供了 ConversationStatusListener 监听器。设置监听后，在会话的状态（置顶和免打扰）改变时，会触发以下方法：

```
public interface ConversationStatusListener {
void onStatusChanged(ConversationStatus[] conversationStatus);
}
```

onStatusChanged 方法返回 conversationStatus 的列表，参数如下：

参数	类型	描述
conversationType	ConversationType	会话类型。
targetId	String	会话 ID。
isTop	boolean	会话是否被设置为置顶。
level	PushNotificationLevel	会话的免打扰级别。SDK 从 5.2.5 版本支持多端同步免打扰级别。具体级别说明详见免打扰功能概述。
notificationStatus	ConversationNotificationStatus	会话提醒状态。仅支持提醒（1）或免打扰（0）两种状态。

设置监听器

IMKit SDK 需要使用 IMCenter 类的 addConversationStatusListener 方法设置会话状态（置顶和免打扰）多端同步监听器。

```
//同步监听器
RongIMClient.ConversationStatusListener listener = new RongIMClient.ConversationStatusListener() {
@Override
public void onStatusChanged(ConversationStatus[] conversationStatus) {
if (conversationStatus == null) {
return;
}
for (ConversationStatus status : conversationStatus) {
Conversation.ConversationType conversationType = status.getConversationType(); //获取会话类型
String targetId = status.getTargetId(); //获取会话 Id
boolean isTop = status.isTop(); //获取该会话当前状态是否置顶
IRongCoreEnum.PushNotificationLevel level = status.getNotificationLevel();// 获取PushNotificationLevel (5.2.5新增)
Conversation.ConversationNotificationStatus notificationStatus = status.getNotifyStatus(); //获取该会话当前的免打扰状态。
}
}
};
IMCenter.getInstance().addConversationStatusListener(listener); //设置监听器
```

自定义处理会话列表数据 自定义处理会话列表数据

更新时间:2024-08-30

数据处理机制支持对会话列表数据进行以下自定义处理：

- 按会话类型过滤会话，被过滤掉的会话不在会话列表展示。
- 按自定义规则过滤会话，会话列表页面仅展示过滤后的数据。
- 设置某类型会话聚合显示。

数据处理器（抽象类）

数据处理器通过抽象类的形式提供，支持的版本为 5.1.3.x 系列中的稳定版、5.1.5 及之后所有版本。其它版本的数据处理机制请参考数据处理接口部分的说明。

数据处理器说明

名称：`BaseDataProcessor`

该抽象类默认提供了以下三个方法，开发者可以根据业务需要覆写任意方法达到自定义需求。

返回值	方法
Conversation.ConversationType[]	supportedTypes()
List<T>	filtered(List<T> data)
boolean	isGathered(Conversation.ConversationType type)
boolean	isGathered(ConversationIdentifier identifier)

方法说明

- `supportedTypes()`

  会话列表页支持的会话类型数组，默认支持所有类型的会话。

- `filtered(List<T> data)`

  对会话数据进行过滤，当从数据库批量拉取到会话和在线收到消息产生新会话时会回调此方法。

参数	类型	说明
data	List<T>	待过滤的数据

- `isGathered(Conversation.ConversationType type)`

  设置某一会话类型是否聚合显示，如果需要聚合显示，返回 true，否则返回 false。

  返回 true 后，该类型的所有会话在会话列表将显示为一条，点击此聚合会话将跳转到聚合会话列表，此时才展示该类型的所有会话，详情参考按类型聚合会话。

- `isGathered(ConversationIdentifier identifier)`

参数	类型	说明
identifier	ConversationIdentifier	(SDK 5.3.0 版本新增默认方法) 包含 Conversation.ConversationType、targetId(String) 。

  （DataProcessor 接口的 default 方法）设置某一会话类型是否聚合显示，如果需要聚合显示，返回 true，否则返回 false。

  返回 true 后，该类型的所有会话在会话列表将显示为一条，点击此聚合会话将跳转到聚合会话列表，此时才展示该类型的所有会话，详情参考按类型聚合会话。

自定义数据处理

1. 自定义数据处理器。声明自定义的数据处理器类，继承 SDK 的 `BaseDataProcessor`，复写需要自定义的方法。

   下面以自定义数据处理类 `MyDataProcessor` 为例说明：

```
public class MyDataProcessor extends BaseDataProcessor<Conversation> {
/**
* 自定义会话列表页面支持的会话类型，此处设置为仅支持单聊和群组会话。
*/
@Override
public Conversation.ConversationType[] supportedTypes() {
Conversation.ConversationType[] types = {Conversation.ConversationType.PRIVATE, Conversation.ConversationType.GROUP};
return types;
}

/**
* 自定义需要聚合显示的会话。此处设置系统会话聚合显示。
*/
@Override
public boolean isGathered(Conversation.ConversationType type) {
if (type.equals(Conversation.ConversationType.SYSTEM)) {
return true;
} else {
return false;
}
}

/**
* 自定义需要聚合显示的会话。此处设置系统会话聚合显示。
*/
@Override
public boolean isGathered(ConversationIdentifier identifier) {
if (identifier.getType().equals(Conversation.ConversationType.SYSTEM)) {
return true;
} else {
return false;
}
}
}
```

2. 设置数据处理器。在打开会话列表页面之前，调用下面方法设置自定义的数据处理器。

```
RongConfigCenter.conversationListConfig().setDataProcessor(new MyDataProcessor());
```

上面示例代码里的 MyDataProcessor 为第 1 步的自定义数据处理器。

数据处理接口

SDK 5.1.5 之前的开发版，可通过数据处理接口进行会话拦截。

通过 Android Studio 实现数据处理接口时，部分方法的默认返回值为 null，会导致会话列表的数据全部被过滤，显示空白会话列表。请务必注意修改返回值为自定义处理后的数据或者原始数据。

强烈建议您升级到稳定版，通过数据处理器实现。

接口说明

数据处理接口提供以下三种方法：

```
public interface DataProcessor<T> {
/**
* 设置会话列表页支持的会话类型
* @return 所支持的会话类型
*/
Conversation.ConversationType[] supportedTypes();
/**
* 对会话数据进行过滤。
* <p>从数据库批量拉取到会话列表时和在线收到消息产生新会话时都会回调此方法</p>
* @param data 待过滤的数据
* @return 过滤完的数据。
*/
List<T> filtered(List<T> data);

/**
* 某一会话类型是否聚合状态显示。
* @param type 会话类型
*/
boolean isGathered(Conversation.ConversationType type);


/**
* 某一会话类型是否聚合状态显示。
*
* @param identifier 会话标识符
*/
default boolean isGathered(ConversationIdentifier identifier) {
if (identifier == null || identifier.getType() == null) {
return false;
}
return isGathered(identifier.getType());
}
}
```

自定义数据处理

1. 声明自定义的数据处理类，实现 DataProcessor 接口，按照自定义需求实现各方法。

```
private Conversation.ConversationType[] supportedTypes = {Conversation.ConversationType.PRIVATE,
Conversation.ConversationType.GROUP, Conversation.ConversationType.SYSTEM};

public class MyDataProcessor implements DataProcessor<Conversation> {
@Override
public Conversation.ConversationType[] supportedTypes() {
//返回自定义的会话列表所支持的会话类型。
return supportedTypes;
}

@Override
public List<Conversation> filtered(List<Conversation> data) {
//返回过滤后的数据，此处示例没有过滤，返回原始数据。
return data; //请注意不要返回 null !!!!
}

@Override
public boolean isGathered(Conversation.ConversationType type) {
//自定义系统会话聚合显示，其它会话非聚合。
if (type.equals(Conversation.ConversationType.SYSTEM)) {
return true;
}
return false;
}

@Override
public boolean isGathered(ConversationIdentifier identifier) {
//自定义系统会话聚合显示，其它会话非聚合。
if (identifier.getType().equals(Conversation.ConversationType.SYSTEM)) {
return true;
}
return false;
}
}
```

2. 通过下面的方法将自定义的数据处理类设置到 SDK 里。

```
RongConfigCenter.conversationListConfig().setDataProcessor(new MyDataProcessor());
```

自定义处理权限请求 自定义处理权限请求

更新时间:2024-08-30

IMKit SDK 的会话页面提供了发送图片、语音、文件、发起通话等功能。在调用涉及敏感权限的 API 时，IMKit 均会向用户申请权限，在权限请求通过后，才会继续执行任务。

IMKit 并未开放与权限申请相关的 UI。如果应用程序需要自定义权限申请的 UI 界面，可以拦截 IMKit 的权限申请。例如，应用程序可以在 IMKit 申请敏感权限时，在自定义页面上同步告知用户申请该敏感权限的目的，满足上架应用市场的合规要求。

拦截权限请求

> 提示
>
> 以下方式在小于 5.6.3 版本的开发版 IMKit 上无法拦截到相册中拍照时的权限请求。建议您使用最新的开发版。稳定版 SDK 存在同样问题，具体是否修复请关注 5.5.2 之后版本的更新日志。

建议在进入会话页面前，使用 PermissionCheckUtil 的 setRequestPermissionListListener 方法，在 onRequestPermissionList 回调中您来自定义处理权限请求。

权限处理完成后，通过 IPermissionEventCallback 告知 IMKit 权限请求结果。confirmed 代表权限通过，cancelled 代表权限不通过。

```
PermissionCheckUtil.setRequestPermissionListListener(
new PermissionCheckUtil.IRequestPermissionListListener() {
@Override
public void onRequestPermissionList(
Context activity,
List<String> permissionsNotGranted,
PermissionCheckUtil.IPermissionEventCallback callback) {
AlertDialog dialog =
new AlertDialog.Builder(
activity,
android.R
.style
.Theme_DeviceDefault_Light_Dialog_Alert)
.setMessage("向用户说明申请权限")
.setPositiveButton(
"去申请",
new DialogInterface.OnClickListener() {
@Override
public void onClick(
DialogInterface dialog, int which) {
dialog.dismiss();
callback.confirmed();
}
})
.setNegativeButton(
"取消",
new DialogInterface.OnClickListener() {
@Override
public void onClick(
DialogInterface dialog, int which) {
dialog.dismiss();
callback.cancelled();
}
})
.show();
}
});
```

参考资源

- 如果需要描述 IMKit 申请敏感权限的目的，可参考 SDK 隐私政策。

设置会话免打扰　　设置会话免打扰

更新时间:2024-08-30

即时通讯服务支持会话免打扰设置。IMKit SDK 可根据会话类型、会话 ID 设置消息提醒状态为「免打扰」。设置后如果客户端在后台运行时，会话中有新的消息，将不会进行通知提醒，可以收到消息内容。如果客户端为离线状态，将不会收到远程通知提醒。

会话的免打扰状态将会被同步到服务端。融云会为用户自动在设备间同步会话免打扰状态数据。客户端可以通过监听器获取同步通知，也可以主动获取最新数据。

设置会话的免打扰状态

IMCenter 类提供 setConversationNotificationStatus 方法，可根据会话类型、会话 ID 设置消息提醒状态为「免打扰」。设置成功后，客户端在后台运行时或处于用户离线状态时，均不会收到该会话的新消息通知。在会话列表页该会话的右下角将展示一个灰色小铃铛图标。

```
IMCenter.getInstance().setConversationNotificationStatus(conversationType, targetId, notificationStatus, callback);
```

参数	类型	说明
conversationType	ConversationType	会话类型。不支持聊天室类型，因为聊天室默认不支持消息提醒。
targetId	String	会话 Id
notificationStatus	ConversationNotificationStatus	消息状态设置。DO_NOT_DISTURB 为免打扰状态。NOTIFY 为提醒状态。
callback	ResultCallback<ConversationNotificationStatus>	回调接口

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";
// 消息免打扰
ConversationNotificationStatus notificationStatus = ConversationNotificationStatus.DO_NOT_DISTURB;

IMCenter.getInstance().setConversationNotificationStatus(conversationType, targetId, notificationStatus, new
RongIMClient.ResultCallback<Conversation.ConversationNotificationStatus>() {
@Override
public void onSuccess(Conversation.ConversationNotificationStatus status) {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

监听会话的免打扰状态同步

即时通讯业务支持会话状态（置顶状态数据和免打扰状态数据）同步机制。设置会话状态同步监听器后，如果会话状态改变，可在本端收到通知。

会话的置顶和免打扰状态数据同步后，SDK 会触发 ConversationStatusListener 的 onStatusChanged 方法。详细说明可参见多端同步免打扰/置顶。

```
public interface ConversationStatusListener {
void onStatusChanged(ConversationStatus[] conversationStatus);
}
```

获取会话的免打扰状态

客户端可以主动获取最新的会话免打扰状态数据，但 IMKit SDK 未直接提供该方法，您需要使用 IMLib 中获取会话免打扰状态的方法。

```
RongIMClient.getInstance().getConversationNotificationStatus(conversationType, targetId, callback);
```

详见 IMLib 文档按会话设置免打扰 中的**获取指定会话的免打扰状态**。

设置全局免打扰　　设置全局免打扰

更新时间:2024-08-30

IMKit SDK 支持设置通知静默时段，以实现全局免打扰的效果。

- 该接口会设置一个从任意时间点（`HH:MM:SS`）开始的免打扰时间窗口。在再次设置或删除用户免打扰时间段之前，当次设置的免打扰时间窗口会每日重复生效。例如，App 用户希望设置永久全天免打扰，可设置 `startTime` 为 `00:00:00`，`period` 为 `1439`。
- 单个用户仅支持设置一个时间段，重复设置会覆盖该用户之前设置的时间窗口。

> 提示
>
> 从 5.2.2 版本开始，推荐集成 IMKit 的客户直接使用 IMLib 的 ChannelClient 类中可同时配置「免打扰级别」的全局免打扰接口。详见 IMLib 文档全局免打扰。

在经 SDK 设置的全局免打扰时段内：

- 如果客户端在后台运行时，会话中有新的消息，将不会进行通知提醒，可以收到消息内容。
- 如果客户端为离线状态，将不会收到远程通知提醒。
- 例外情况：@ 消息属于高优先级消息，不受全局免打扰逻辑控制，会始终进行通知提醒。

设置全局免打扰时段（< 5.2.2）

设置全局免打扰时段，以屏蔽所有通知，包括本地通知以及远程推送通知。

```
RongNotificationManager.getInstance().setNotificationQuietHours(startTime, spanMinutes, callback)
```

参数	类型	说明
startTime	String	开始时间，精确到秒。格式为 HH:MM:SS，例如 01:31:17。
spanMinutes	int	免打扰时间窗口大小，单位为分钟。支持范围为 [1-1439]。
callback	RongIMClient.OperationCallback	操作回调

获取全局免打扰时段（< 5.2.2）

通过以下方法，可以获取当前应用设置的静默时间。

```
RongNotificationManager.getInstance().getNotificationQuietHours(callback);
```

参数	类型	说明
callback	GetNotificationQuietHoursCallback	获取全局免打扰时间的回调。

GetNotificationQuietHoursCallback 提供两个回调方法：

- 获取通知静默时间成功时的回调方法

```
/**
* 获取通知静默时间成功时的回调方法。
*
* @param startTime 起始时间 格式 HH:MM:SS。
* @param spanMinutes 间隔分钟数 0 < spanMinutes < 1440。
*/
void onSuccess(String startTime, int spanMinutes);
```

- 获取通知静默时间出错时的回调方法

```
/**
* 获取通知静默时间出错时的回调方法。
*
* @param errorCode 获取通知静默时间错误代码。
*/
void onError(ErrorCode errorCode);
```

取消全局免打扰时段（< 5.2.2）

通过以下方法移除之前设置的通知静默时间，移除成功后，会正常收到本地通知或推送通知。

```
RongNotificationManager.getInstance().removeNotificationQuietHours(callback);
```

本地通知 本地通知

更新时间:2024-08-30

IMKit 已实现本地通知的创建、弹出与跳转行为，方便开发者快速构建应用。

什么是本地通知

本地通知指应用在前台或后台运行时，由 IMKit 或应用客户端直接调用系统接口创建并发送的通知（Notification）。IMKit SDK 内部已经实现了本地通知功能 (Notification)，当应用处于后台接收到新消息时，IMKit 默认会在通知面板弹出通知提醒，即本地通知。

IMKit 的本地通知已支持以下场景：

- 当 App 刚进入后台时（仍处于后台活跃状态），IMKit 仍可通过长连接通道接收到新消息（撤回消息也会产生撤回信令消息）。接到新消息后，IMKit 默认会创建并弹出通知，即本地通知。点击此通知会跳转到会话页面。

> 提示
>
> App 在后台进入非活跃状态后（例如，被系统杀死），IMKit 与服务端断开连接。如果应用已集成第三方厂商推送服务（或融云自建推送服务），此时可通过接收推送通知。来自第三方厂商的离线推送通知一般由推送厂商直接创建并弹出，不属于本文所述的本地通知。

- 当 App 处于前台，且未打开任何会话页面时（未与任何人聊天），接收新消息后默认会响铃并震动，但不弹通知。可通过 `setForegroundOtherPageAction` 修改为静默或仅弹出通知，参考 NotificationConfig.java。

设置本地通知分类

华为从 2023 年 9 从2023年9月15日开始，华为推送服务将基于《华为消息分类标准》对本地通知进行灰度管控，主要包括对应用发送的本地通知进行分类管理，以及对资讯营销消息统一进行频次管控。应用在未申请华为自分类权益的情况下，发送的本地通知将全部按照资讯营销消息处理（限频2条或5条/天）。

为避免 IMKit 直接调用华为设备系统接口发送的本地通知无法弹出，应用应确保自身已向华为申请所需自分类权益。IMKit 从 5.6.4 版本开始，内部弹出本地通知的 `category` 默认设置为 `CATEGORY_MESSAGE`，应用程序不需要额外调用 API 进行设置。

您可以在初始化之后，使用 IMKit 全局配置修改默认的本地通知分类：

```
RongConfigCenter.notificationConfig().setCategoryNotification(Notification.CATEGORY_MESSAGE);
```

> 提示
>
> 如果您集成的 IMKit < 5.6.4，需要应用程序自行适配，建议参考知识库文档关于华为限制本地通知频次及分类的适配流程。

设置前台本地通知铃声与震动

> 提示
>
> SDK 从 5.2.3 版本开始支持该功能。

从 5.2.3 开始，IMKit 增加了控制应用在前台非会话页面时，接收到新消息是否响铃提醒、是否震动提醒的独立控制开关。

如果修改 App 默认行为，您可以在应用 `res/values` 目录下创建 `rc_config.xml` 文件，修改以下配置项的值，默认开启。

```
//在前台非会话页面时，接收到新消息是否响铃
<bool name="rc_sound_in_foreground">true</bool>

//在前台非会话页面时，接收到新消息是否震动
<bool name="rc_vibrate_in_foreground">true</bool>
```

针对本地通知响铃、震动行为控制，可通过 IMKit 全局配置动态修改。

```
/** 是否震动 */
RongConfigCenter.featureConfig().setVibrateInForeground(true)
/** 是否有铃声 */
RongConfigCenter.featureConfig().setSoundInForeground(false)
```

拦截本地通知

应用在本地通知显示前对通知进行拦截。IMKit 支持在拦截修改 `Message` 后继续弹出本地通知。您可以参考 IMKit 源码中的以下文件：

- DefaultInterceptor.java
- RongNotificationManager.java

- NotificationUtil.java

如果需要更多自定义效果，建议完全拦截，由应用程序自行接管并弹出本地通知。

拦截本地通知

> 提示
>
> 从 SDK 版本 5.1.4 开始，请使用以下方法设置拦截器。

您可以拦截 SDK 的本地通知。设置拦截器后，可阻止 SDK 弹出本地通知，并按需自定义处理。请在初始化之后，建立 IM 连接之前设置监听器。

```
public class MyNotificationInterceptor extends DefaultInterceptor {
/**
* 是否拦截此本地通知，一般用于自定义本地通知的显示。
*
* @param message 本地通知对应的消息
* @return 是否拦截。true 拦截本地通知，SDK 不弹出通知，需要用户自己处理。false 不拦截，由 SDK 展示本地通知。
*/
@Override
public boolean isNotificationIntercepted(Message message) {
...
return true; //true, 拦截本地通知，SDK 不再处理本地通知的逻辑。
}

/**
* 是否为高优先级消息。高优先级消息不受全局静默时间和会话免打扰控制，比如 @ 消息。
*
* @param message 接收到的消息
* @return 是否为高优先级消息
*/
@Override
public boolean isHighPriorityMessage(Message message) {
return false;
}
/**
* 注册默认 channel 之前的回调。可以通过此方法拦截并修改默认 channel 里的配置，将修改后的 channel 返回。
*
* @param defaultChannel 默认通知频道
* @return 修改后的通知频道。
*/
@Override
public NotificationChannel onRegisterChannel(NotificationChannel defaultChannel) {
return defaultChannel;
}

/**
* 设置本地通知 PendingIntent 时的回调。
* 应用层可通过此方法更改 PendingIntent 里的设置，以便自定义本地通知的点击行为。
* 点击本地通知时，SDK 默认跳转到对应会话页面。
* @param pendingIntent SDK 默认 PendingIntent
* @param intent pendingIntent 里携带的 intent。
* 可通过 intent 获取以下信息:
* intent.getStringExtra(RouteUtils.CONVERSATION_TYPE);
* intent.getStringExtra(RouteUtils.TARGET_ID);
* intent.getIntExtra(RouteUtils.MESSAGE_ID, -1);;
* @return 本地通知里需配置的 PendingIntent.
*/
@Override
public PendingIntent onPendingIntent(PendingIntent pendingIntent, Intent intent) {
Intent intentNew = new Intent(getApplicationContext(), MainActivity.class); //自定义跳转到会话列表
PendingIntent pendingIntent1 = PendingIntent.getActivity(getApplicationContext(), 1, intentNew, PendingIntent.FLAG_UPDATE_CURRENT);
return pendingIntent1;
}

}

RongConfigCenter.notificationConfig().setInterceptor(new MyNotificationInterceptor());
```

拦截本地通知（< 5.1.4）

> 提示
>
> 从 SDK 版本 5.1.4 开始，废弃以下方法。

您可以拦截 SDK 的本地通知。设置拦截器后，可阻止 SDK 弹出本地通知，并按需自定义处理。请在初始化之后，建立 IM 连接之前设置监听器。

```
RongConfigCenter.notificationConfig().setInterceptor(new NotificationConfig.Interceptor() {
/**
* 是否拦截此本地通知，一般用于自定义本地通知的显示。
*
* @param message 本地通知对应的消息
* @return 是否拦截。true 拦截本地通知，SDK 不弹出通知，需要用户自己处理。false 不拦截，由 SDK 展示本地通知。
*/
@Override
public boolean isNotificationIntercepted(Message message) {
...
return true; //true, 拦截本地通知，SDK 不再处理本地通知的逻辑。
}

/**
* 是否为高优先级消息。高优先级消息不受全局静默时间和会话免打扰控制，比如 @ 消息。
*
* @param message 接收到的消息
* @return 是否为高优先级消息
*/
@Override
public boolean isHighPriorityMessage(Message message) {
return false;
}
/**
* 注册默认 channel 之前的回调。可以通过此方法拦截并修改默认 channel 里的配置，将修改后的 channel 返回。
*
* @param defaultChannel 默认通知频道
* @return 修改后的通知频道。
*/
@Override
public NotificationChannel onRegisterChannel(NotificationChannel defaultChannel) {
return defaultChannel;
}

/**
* 设置本地通知 PendingIntent 时的回调。
* 应用层可通过此方法更改 PendingIntent 里的设置，以便自定义本地通知的点击行为。
* 点击本地通知时，SDK 默认跳转到对应会话页面。
* @param pendingIntent SDK 默认 PendingIntent
* @param intent pendingIntent 里携带的 intent。
* 可通过 intent 获取以下信息:
* intent.getStringExtra(RouteUtils.CONVERSATION_TYPE);
* intent.getStringExtra(RouteUtils.TARGET_ID);
* intent.getIntExtra(RouteUtils.MESSAGE_ID, -1);;
* @return 本地通知里需配置的 PendingIntent.
*/
@Override
public PendingIntent onPendingIntent(PendingIntent pendingIntent, Intent intent) {
Intent intentNew = new Intent(getApplicationContext(), MainActivity.class); //自定义跳转到会话列表
PendingIntent pendingIntent1 = PendingIntent.getActivity(getApplicationContext(), 1, intentNew, PendingIntent.FLAG_UPDATE_CURRENT);
return pendingIntent1;
}

});
```

自定义通知点击

点击本地通知时，SDK 默认跳转到通知对应的会话页面。您可以自定义点击时的跳转事件。在上文拦截通知的 onPendingIntent 回调中使用。

```
@Override
public PendingIntent onPendingIntent(PendingIntent pendingIntent, Intent intent) {
Intent intentNew = new Intent(getApplicationContext(), MainActivity.class); //自定义跳转到会话列表
PendingIntent pendingIntent1 = PendingIntent.getActivity(getApplicationContext(), 1, intentNew, PendingIntent.FLAG_UPDATE_CURRENT);
return pendingIntent1;
}
```

拦截消息框　　拦截消息框

更新时间:2024-08-30

IMKit 通过 Android 的 Toast 消息框弹出提醒或消息反馈，一般用来显示操作结果，或者应用状态的改变。从 5.6.3 版本开始，您可以设置 Toast 拦截器，方便统一修改。

拦截 Toast 消息框

> 提示
>
> 要求 IMKit 版本 ≧ 5.6.3。

ToastInterceptor 的willToast 方法会在 UI 线程被调用。在 willToast 方法中返回 false 表示应用程序需要拦截 Toast。

```
import io.rong.imkit.utils.ToastUtils;


ToastUtils.setInterceptor(
new ToastUtils.ToastInterceptor() {
@Override
public boolean willToast(
@NonNull Context context, @NonNull CharSequence text, int duration) {
String s = "这是被拦截的 toast：" + text;
Toast.makeText(context, s, Toast.LENGTH_SHORT).show();
return false;
}
});
```

如果想取消拦截器，设置拦截器为 null 即可。

混淆　　混淆 混淆脚本

更新时间:2024-08-30

融云 SDK 相关混淆脚本参考如下配置：

```
-keepattributes Exceptions,InnerClasses

-keepattributes Signature

-keep class io.rong.** {*;}
-keep class cn.rongcloud.** {*;}
-keep class * implements io.rong.imlib.model.MessageContent {*;}
-dontwarn io.rong.push.**
-dontnote com.xiaomi.**
-dontnote com.google.android.gms.gcm.**
-dontnote io.rong.**

# 下方混淆使用了融云 IMKit 提供的 locationKit 位置插件时才需要配置，可参考高德官网的混淆方式:https://lbs.amap.com/api/android-sdk/guide/create-project/dev-attention
-keep class com.amap.api.maps.**{*;}
-keep class com.autonavi.**{*;}
-keep class com.amap.api.trace.**{*;}
-keep class com.amap.api.location.**{*;}
-keep class com.amap.api.fence.**{*;}
-keep class com.loc.**{*;}
-keep class com.autonavi.aps.amapapi.model.**{*;}
-keep class com.amap.api.services.**{*;}

-ignorewarnings
```

当代码中有继承 PushMessageReceiver 的子类时，需 keep 所创建的子类广播。

示例：

```
-keep class io.rong.app.DemoNotificationReceiver {*;}
```

把 io.rong.app.DemoNotificationReceiver 改成子类广播的完整类路径即可.

快速上手 快速上手

更新时间:2024-08-30

本教程是为了让新手快速了解融云即时通讯能力库（IMLib）。在本教程中，您可以体验集成 SDK 的基本流程和 IMLib 的基础通信能力。

前置条件

- 注册开发者账号。注册成功后，控制台会默认自动创建您的首个应用，默认生成开发环境下的 App Key，使用国内数据中心。
- 获取开发环境的应用 App Key。如不使用默认应用，请参考 如何创建应用，并获取对应环境 App Key 和 App Secret。

> 提示
>
> 每个应用具有两个不同的 App Key，分别对应开发环境与生产环境，两个环境之间数据隔离。在您的应用正式上线前，可切换到使用生产环境的 App Key，以便上线前进行测试和最终发布。

环境要求

- （SDK ≧ 5.6.3）使用 Android 5.0（API 21）或更高版本
- （SDK < 5.6.3）使用 Android 4.4（API 19）或更高版本

开始集成

IMLib 支持通过 Maven、本地 Android 库模块（Module）的方式集成。请提前在融云官网 SDK 下载页面或融云的 Maven 仓库查询最新版本。

导入 SDK

本教程以在 Gradle 中添加远程依赖项为例，将 IMLib SDK 导入到您的应用工程中。请注意使用 融云的 Maven 仓库。

1. 打开根目录下的 `build.gradle`（**Project** 视图下），声明融云的 Maven 代码库。

```
allprojects {
repositories {
...
//融云 maven 仓库地址
maven {url "https://maven.rongcloud.cn/repository/maven-releases/"}
}
}
```

2. 在应用的 `build.gradle` 中，添加融云即时通讯能力库（IMLib）为远程依赖项。

```
dependencies {
...
//此处以集成 IMLib 为例
api 'cn.rongcloud.sdk:im_lib:x.y.z'
}
```

> 提示
>
> 各个 SDK 的最新版本号可能不相同，还可能是 x.y.z.h，可前往 融云官网 SDK 下载页面 或 融云的 Maven 代码库 查询。

其他导入 SDK 方式请参见导入 SDK。

初始化 SDK

融云即时通讯客户端 SDK 核心类为 RongCoreClient 和 `RongIMClient`。在 Application 的 `onCreate()` 方法中，调用 RongCoreClient 的初始化方法，传入生产或开发环境的 App Key。

如果 SDK 版本 ≧ 5.4.2，请使用以下初始化方法。

```
String appKey = "Your_AppKey"; // example: bos9p5rlcm2ba
InitOption initOption = new InitOption.Builder().build();

RongCoreClient.init(getApplicationContext(), appKey, initOption);
```

初始化配置（`InitOption`）中封装了区域码（AreaCode），导航服务地址（`naviServer`）、文件服务地址（`fileServer`）、数据统计服务地址（`statisticServer`）配置，以及是否开启推送的开关（`enablePush`）和主进程开关（`isMainProcess`）。不传入任何配置表示全部使用默认配置。SDK 默认连接北京数据中心。

如果 App Key 不属于中国（北京）数据中心，则必须传入有效的初始化配置。初始化详细说明参见初始化。

获取用户 Token

用户 Token 是与用户 ID 对应的身份验证令牌，是应用程序的用户在融云的唯一身份标识。应用客户端在使用融云即时通讯功能前必须与融云建立 IM 连接，连接时必须传入 Token。

在实际业务运行过程中，应用客户端需要通过应用的服务端调用 IM Server API 申请取得 Token。详见 Server API 文档 注册用户 。

在本教程中，为了快速体验和测试 SDK，我们将使用控制台「北极星」开发者工具箱，从 API 调试页面调用 获取 Token 接口，获取到 **userId** 为 1 的用户的 Token。提交后，可在返回正文中取得 Token 字符串。

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"code":200,"userId":"1","token":"gxld6GHx3t1eDxof1qtxxYrQcjkbhl1V@sgyu.cn.example.com;sgyu.cn.example.com"}
```

建立 IM 连接

1. 监听 IM 连接状态的变化，以便 UI 上给用户以提示，提高体验。建议在应用生命周期内设置。为了避免内存泄露，请在不需要监听时，将设置的监听器移除。详见连接状态监听。

```
private IRongCoreListener.ConnectionStatusListener connectionStatusListener = new IRongCoreListener.ConnectionStatusListener() {
@Override
public void onChanged(ConnectionStatus status) {
//开发者需要根据连接状态码，进行不同业务处理
}
};
public void setIMStatusListener() {
// 添加连接状态监听器 since 5.1.6
RongCoreClient.addConnectionStatusListener(ConnectionStatusListener);
}
```

2. 使用以上步骤中获取的 Token, 模拟 userId 为 1 的用户连接到融云服务器。

```
String token = "用户Token";
RongCoreClient.connect(token, new IRongCoreCallback.ConnectCallback() {
/**
* 数据库回调.
* @param code 数据库打开状态. DATABASE_OPEN_SUCCESS 数据库打开成功; DATABASE_OPEN_ERROR 数据库打开失败
*/
@Override
public void OnDatabaseOpened(IRongCoreEnum.DatabaseOpenStatus code) {

}
/**
* 成功回调
* @param userId 当前用户 ID
*/
@Override
public void onSuccess(String userId) {

}
/**
* 错误回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.ConnectionErrorCode errorCode) {

}
});
```

SDK 已实现自动重连机制，请参见连接。

监听消息

设置消息接收监听器，用于接收所有类型的实时或者离线消息。

```
RongCoreClient.addOnReceiveMessageListener(
new io.rong.imlib.listener.OnReceiveMessageWrapperListener() {
@Override
public void onReceivedMessage(Message message, ReceivedProfile profile) {
// 针对接收离线消息时，服务端会将 200 条消息打成一个包发到客户端，客户端对这包数据进行解析。该参数表示每个数据包数据逐条上抛后，还剩余的条数
int left = profile.getLeft();
// 消息是否离线消息
boolean isOffline = profile.isOffline();
// 是否在服务端还存在未下发的消息包
boolean hasPackage = profile.hasPackage();
}
}
);
```

发送消息

向 userId 为 2 的用户发送一条消息。

```
String content = “您好，这是从用户1发出的消息”;
String targetId = "2";
ConversationType conversationType = ConversationType.PRIVATE;
// 构建消息
TextMessage messageContent = TextMessage.obtain(content);
Message message = Message.obtain(targetId, conversationType, messageContent);

// 发送消息
RongCoreClient.getInstance().sendMessage(message, null, null, new IRongCoreCallback.ISendMessageCallback() {
@Override
public void onAttached(Message message) {

}

@Override
public void onSuccess(Message message) {

}

@Override
public void onError(Message message, IRongCoreEnum.ErrorCode errorCode) {

}
});
```

导入 SDK 导入 SDK

更新时间:2024-08-30

利用 Android Studio 中的 Gradle 构建系统，您可以轻松地将融云即时通讯能力库（IMLib）作为依赖项添加到您的构建中。

您可以使用 Gradle 直接导入远程依赖项，或者以 Android 本地库模块导入本地依赖项。

环境要求

- （SDK ≧ 5.6.3）使用 Android 5.0（API 21）或更高版本
- （SDK < 5.6.3）使用 Android 4.4（API 19）或更高版本

检查版本

在导入 SDK 前，您可以前往融云官网 SDK 下载页面 确认当前最新版本号。

Gradle

使用 Gradle，添加融云即时通讯能力库（IMLib）为远程依赖项。请注意使用 融云的 Maven 仓库 。Android Studio 的配置在 Gradle 插件 7.0 以下版本、7.0 版本、和 7.1 及以上版本有所不同。请根据您当前的 Gradle 插件版本进行配置。本文以使用 Gradle 插件 7.0 以下版本为例。

1. 声明融云的 Maven 代码库 ，以使用 Gradle 插件 7.0 以下版本为例。打开根目录下的 `build.gradle`（**Project** 视图下）：

```
allprojects {
repositories {
...
//融云 maven 仓库地址
maven {url "https://maven.rongcloud.cn/repository/maven-releases/"}
}
}
```

2. 在应用的 `build.gradle` 中，添加融云即时通讯能力库（IMLib）为远程依赖项。

```
dependencies {
...
//此处以集成 IMLib 为例
api 'cn.rongcloud.sdk:im_lib:x.y.z'
}
```

> 提示
>
> 各个 SDK 的最新版本号可能不相同，还可能是 x.y.z.h，可前往 融云官网 SDK 下载页面 或 融云的 Maven 代码库 查询。

Android 本地库 (Module)

在导入 SDK 前，您需要前往融云官网 SDK 下载页面 ，将 **即时通讯能力库 IMLib** 下载到本地。

1. 在 Android Studio 中打开工程后，依次点击 **File** > **New** > **Import Module**，找到下载的 Module 组件并导入。

2. 如果导入的内容中包含有插件的 aar 包，请移至 `app/libs` 目录下。

3. 打开根目录下的 `settings.gradle`（**Project** 视图下），添加 IMLib 本地库模块。

```
include ':IMLib'
...
```

4. 在应用的 `build.gradle` 中，添加 IMLib 为本地库模块依赖项。

```
dependencies {
...
api project(':IMLib')
...
}
```

5. (可选) 以 Android 本地库模块导入 SDK 时默认不带 Javadoc。建议自行从融云的 Maven 代码库 下载 Javadoc 并导入，以便于在 Android Studio 中即时查看。

如需指导，请参见以下知识库链接：

https://help.rongcloud.cn/t/topic/727

初始化 初始化

更新时间:2024-08-30

在使用 SDK 其它功能前，必须先进行初始化。本文中将详细说明 IMLib SDK 初始化的方法。

首次使用融云的用户，我们建议先阅读 IMLib SDK 快速上手，以完成开发者账号注册等工作。

推送

推送是常见的基础功能。IMLib SDK 已集成融云自有推送，以及多家第三方推送，且会在 SDK 初始化后触发。因此，客户端的推送配置必须在初始化之前提供。详细说明请参见启用推送。

海外数据中心

- 如果您使用海外数据中心，且使用 5.4.2 及更新版本的开发版（dev）SDK，请注意在初始化配置中传入正确的区域码（AreaCode）。
- 如果您使用海外数据中心，且使用稳定版（stable） SDK，或使用早于 5.4.2 版本的开发版（dev）SDK，必须在初始化之前修改 IMLib SDK 连接的服务地址为海外数据中心地址。否则 SDK 默认连接中国国内数据中心服务地址。详细说明请参见配置海外数据中心服务地址。

进程

> 提示
>
> 从 5.3.0 版本开始，IMLib SDK 支持单进程。

SDK 可支持多进程和单进程机制。

如果 SDK 版本 < 5.3.3 或 ≥ 5.3.5，默认采用多进程机制，初始化之后，应用会启动以下进程：

1. 应用的主进程
2. `<应用包名>:ipc`。此进程是 IM 通信的核心进程，和主进程任务相互隔离。
3. `io.rong.push`：融云默认推送进程。该进程是否启动由推送通道的启用策略决定。详细说明可参考启用推送。

如果 SDK 版本 ≥ 5.3.3 且 < 5.3.5，默认采用单进程机制。

从 5.3.0 版本开始，可以在初始化前通过 RongCoreClient 的 `enableSingleProcess` 接口开启或关闭单进程。请注意，开启单进程后融云会占用主进程的部分内存。

```
//开启单进程
RongCoreClient.getInstance().enableSingleProcess(true);
```

初始化 SDK

> 提示
>
> 以下初始化方法要求 SDK 版本 ≥ 5.4.2。您可前往 融云官网 SDK 下载页面 查询最新开发版（Dev）和稳定版（Stable）的版本号。

请在 Application 的 `onCreate()` 方法中初始化 SDK，传入**生产或开发环境的** App Key。

```
String appKey = "YourAppKey"; // example: bos9p5rlcm2ba
InitOption initOption = new InitOption.Builder().build();

RongCoreClient.init(this, appKey, initOption);
```

初始化配置（`InitOption`）中封装了区域码（AreaCode）配置。SDK 将通过区域码获取有效的导航服务地址、文件服务地址、数据统计服务地址、和日志服务地址等配置。

- 如果 App Key 属于中国（北京）数据中心，您无需传入任何配置，SDK 会使用默认配置。
- 如果 App Key 属于海外数据中心，则必须传入有效的区域码（AreaCode）配置。请务必在控制台核验当前 App Key 所属海外数据中心后，找到 AreaCode 中对应的枚举值进行配置。

例如，使用新加坡数据中心的应用的**生产或开发环境的** App Key：

```
String appKey = "Singapore_dev_AppKey";
AreaCode areaCode = AreaCode.SG;

InitOption initOption = new InitOption.Builder()
.setAreaCode(areaCode)
.build();

RongCoreClient.init(context, appKey, initOption);
```

除区域码外，初始化配置（`InitOption`）中还封装了以下配置：

- 是否开启推送的开关（`enablePush`）：是否整体禁用推送。

- 主进程开关（`isMainProcess`）：是否为主进程。默认情况下由 SDK 判断进程。
- 导航服务地址（`naviServer`）：一般情况下不建议单独配置。SDK 内部默认使用与区域码对应的地址。
- 文件服务地址（`fileServer`）：仅限私有云使用。
- 数据统计服务地址（`statisticServer`）：一般情况下不建议单独配置。SDK 内部默认使用与区域码对应的地址。

```
String appKey = "Your_AppKey";
AreaCode areaCode = AreaCode.BJ;

InitOption initOption = new InitOption.Builder()
.setAreaCode(areaCode)
.enablePush(true)
.setFileServer("http(s)://fileServer")
.setNaviServer("http(s)://naviServer")
.setStatisticServer("http(s)://StatisticServer")
.build();

RongCoreClient.init(context, appKey, initOption);
```

初始化 SDK（旧版）

> 提示
>
> 如果您使用的开发版 SDK 版本号小于 5.4.2，或者稳定版 SDK 版本号小于等于 5.3.8，只能使用以下方式初始化。建议您及时升级 SDK 到最新版本。

请在 Application 的 `onCreate()` 方法中初始化 SDK，传入**生产**或**开发**环境的 App Key。您可以使用 RongCoreClient 或 `RongIMClient` 的初始化方法。

```
RongCoreClient.init(context, appKey, enablePush);
```

参数	类型	说明
context	Context	应用上下文。
appKey	String	应用的**开发**环境或**生产**环境 AppKey，请勿混淆。如果您选择在 `AndroidManifest.xml` 设置 appKey，此处可不传。
enablePush	boolean	是否启用推送。默认 `true`，即开启推送功能。

仅为保持向后兼容，IMLib SDK 5x 仍支持在 `AndroidManifest.xml` 中配置 App Key。但考虑到 App Key 安全性，融云已不再建议使用这种配置方式。

建议在用户接受隐私协议后，再进行初始化。

```
/**
* 应用启动时，判断用户是否已接受隐私协议，如果已接受，正常初始化；否则跳转到隐私授权页面请求用户授权。
*/
public class App extends Application {
@Override
public void onCreate() {
super.onCreate();

//伪代码，从 sp 里读取用户是否已接受隐私协议
boolean isPrivacyAccepted = getPrivacyStateFromSp();
//用户已接受隐私协议，进行初始化
if (isPrivacyAccepted) {
String appKey = "控制台创建的应用的 AppKey";
//第一个参数必须传应用上下文
RongIMClient.init(this.getApplicationContext(), appKey);
} else {
//用户未接受隐私协议，跳转到隐私授权页面。
goToPrivacyActivity();
}
...
}
}

/**
* 该类为隐私授权页面，示范如何在用户接受隐私协议后进行 IM 初始化。
*/
public class PrivacyActivity extends Activity implements View.OnClickListener {
...
@Override
public void onClick(View v) {
switch (v.getId()) {
case R.id.accept_privacy:
//伪代码，保存到 sp
savePrivacyStateToSp();

String appKey = "控制台创建的应用的 AppKey";
//第一个参数必须传应用上下文
RongIMClient.init(this.getApplicationContext(), appKey);
break;
default:
...
}
}
}
```

监听连接状态　　监听连接状态

更新时间:2024-08-30

客户端 SDK 为 App 提供了 IM 连接状态监听器 ConnectionStatusListener。通过监听 IM 连接状态的变化，App 可以进行不同业务处理，或在页面上给出提示。

连接状态监听器说明

ConnectionStatusListener 接口定义如下：

```
public interface ConnectionStatusListener {
void onChanged(ConnectionStatus status);
}
```

当连接状态发生变化时，SDK 会通过 onChanged() 方法，将当前的连接状态回调给开发者。各状态的具体说明请参考下表。

状态名称	状态值	说明
NETWORK_UNAVAILABLE	-1	网络不可用
CONNECTED	0	连接成功
CONNECTING	1	连接中
UNCONNECTED	2	未连接状态，即应用没有调用过连接方法
KICKED_OFFLINE_BY_OTHER_CLIENT	3	用户账号在其它设备登录，此设备被踢下线
TOKEN_INCORRECT	4	Token 过期时触发此状态
CONN_USER_BLOCKED	6	用户被控制台封禁
SIGN_OUT	12	用户主动断开连接的状态，见断开连接
SUSPEND	13	连接暂时挂起（多是由于网络问题导致），SDK 会在合适时机进行自动重连
TIMEOUT	14	连接超时，SDK 将停止连接，用户需要做超时处理，再自行调用连接接口进行连接

添加或移除连接状态监听器

SDK 支持设置多个监听器。建议在应用生命周期内设置。

为了避免内存泄露，请在不需要监听时，将设置的监听器移除。

```
// 添加连接状态监听器 since 5.1.6
RongCoreClient.addConnectionStatusListener(listener);
// 移除连接状态监听器 since 5.1.6
RongCoreClient.removeConnectionStatusListener(listener);
```

连接 连接

更新时间:2024-08-30

应用客户端成功连接到融云服务器后，才能使用融云即时通讯 SDK 的收发消息功能。

融云服务端在收到客户端发起的连接请求后，会根据连接请求里携带的用户身份验证令牌（Token 参数），判断是否允许用户连接。

前置条件

- 通过服务端 API 注册用户（获取 Token）。融云客户端 SDK 不提供获取 Token 方法。应用程序可以调用自身服务端，从融云服务端获取 Token。
  - 取得 Token 后，客户端可以按需保存 Token，供后续连接时使用。具体保存位置取决于应用程序客户端设计。如果 Token 未失效，就不必再向融云请求 Token。
  - Token 有效期可在控制台进行配置，默认为永久有效。即使重新生成了一个新 Token，未过期的旧 Token 仍然有效。Token 失效后，需要重新获取 Token。如有需要，可以主动调用服务端 API 作废 Token。
- 建议应用程序在连接之前设置连接状态监听。
- SDK 已完成初始化。

> 提示
>
> **请不要在客户端直接调用服务端 API 获取 Token。获取 Token 需要提供应用的 App Key 和 App Secret。客户端如果保存这些凭证，一旦被反编译，会导致应用的 App Key 和 App Secret 泄露。所以，请务必确保在应用服务端获取 Token。**

连接聊天服务器

请根据应用的业务需求与设计，自行决定合适的时机（登陆、注册、或其他时机以免无法进入应用主页），向融云聊天服务器发起连接请求。

请注意以下几点：

- 必须在 SDK 初始化之后，调用 `connect` 方法进行连接。否则可能没有回调。
- SDK 本身有重连机制，在一个应用生命周期内不要多次调用 `connect` 。否则可能触发多个回调，触发导致回调被清除。
- 在应用主进程内调用一次即可。

接口（可设置超时）

客户端用户首次连接聊天服务器时，建议调用带连接超时时间（timeLimit）的接口，并设置超时秒数。在网络较差等导致连接超时的情况下，您可以利用接口的超时错误回调，并在 UI 上提醒用户，例如建议客户端用户等待网络正常的时候再次连接。

一旦连接成功，SDK 的重连机制将立即开始生效，并接管所有的重连处理。当因为网络原因断线时，SDK 会不停重连直到连接成功为止，不需要您做额外的连接操作。应用主动断开连接，将退出重连机制，详见断开连接。其他情况请参见重连机制与重连互踢。

如果后续离线登录，建议将 timeLimit 参数设置为 0，可以在回调数据库打开（onDatabaseOpened）后就进行页面跳转，优先展示本地历史数据。连接逻辑则完全托管给 SDK。

接口原型

```
RongIMClient.connect(token, timeLimit, connectCallback);
```

参数说明

参数	类型	说明
token	String	从服务端获取的 Token。
timeLimit	int	超时时间（秒）。超时后不再重连。取值 <=0 则将一直连接，直到连接成功或者发生业务错误。
connectCallback	ConnectCallback	连接回调。详见下文连接回调方法说明。

- `timeLimit` **参数详细说明**：
  - `timeLimit` <= 0，IM 将一直连接，直到连接成功或者发生业务错误（如 token 非法）。
  - `timeLimit` > 0，IM 将最多连接 timeLimit 秒。如果在 timeLimit 秒内无法连接成功则不再进行重连，通过回调 onError 告知连接超时，您需要再自行调用 connect 接口。

示例代码

```
public connectIM(String token){
boolean isCachedLogin == getLoginStatusFromSP(context); //伪代码，从 sharedpreference 里读取该用户是否为免密登录
int timeLimit = 0;
if(!isCachedLogin) {
timeLimit = 5;
}
RongIMClient.connect("用户Token", timeLimit, new RongIMClient.ConnectCallback() {
@Override
public void onDatabaseOpened(RongIMClient.DatabaseOpenStatus code) {
if(RongIMClient.DatabaseOpenStatus.DATABASE_OPEN_SUCCESS.equals(code)) {
//本地数据库打开，跳转到会话列表页面
} else {
//数据库打开失败，可以弹出 toast 提示。
}
}

@Override
public void onSuccess(String s) {
//连接成功，如果 onDatabaseOpened() 时没有页面跳转，也可在此时进行跳转。
}

@Override
public void onError(RongIMClient.ConnectionErrorCode errorCode) {
if(errorCode.equals(RongIMClient.ConnectionErrorCode.RC_CONN_TOKEN_EXPIRE)) {
//从 APP 服务请求新 token，获取到新 token 后重新 connect()
} else if (errorCode.equals(RongIMClient.ConnectionErrorCode.RC_CONNECT_TIMEOUT)) {
//连接超时，弹出提示，可以引导用户等待网络正常的时候再次点击进行连接
} else {
//其它业务错误码，请根据相应的错误码作出对应处理。
}
}
})
}
```

接口（无超时设置）

如果客户端用户已成功登录过您的应用，且连接过融云聊天服务器，后续离线登录时建议使用此接口。

此时因用户已有历史数据，并不需要强依赖于连接成功。可以在回调数据库打开（onDatabaseOpened）后就进行页面跳转，优先展示本地历史数据。连接逻辑完全托管给 SDK 即可。

> 提示
>
> 当网络较差时，有可能长时间不会回调。

调用此接口后，SDK 的重连机制将立即开始生效，并接管所有的重连处理。SDK 会不停重连直到连接成功为止，不需要您做额外的连接操作。应用主动断开连接，将退出重连机制，详见断开连接。其他情况请参见重连机制与重连互踢。

接口原型

```
RongIMClient.connect(token, connectCallback);
```

参数说明

参数	类型	说明
token	String	从服务端获取的用户身份令牌
connectCallback	ConnectCallback	连接回调。详见下文连接回调方法说明。

连接回调方法说明

连接回调 connectCallback 提供了以下三个回调方法：

- onDatabaseOpened(DatabaseOpenStatus code)

  本地数据库打开状态回调。当回调 DATABASE_OPEN_SUCCESS 时，说明本地数据库打开，此时可以拉取本地历史会话及消息，适用于离线登录场景。

- onSuccess(String userId)

  连接成功的回调，返回当前连接的用户 ID。

- onError(ConnectionErrorCode errorCode)

  连接失败并返回对应的连接错误码，开发者需要参考连接状态码进行不同业务处理。

  常见错误如下：

  - SDK 没有初始化即调用 connect()方法。
  - 应用客户端与应用服务器的 App Key 不一致，导致 TOKEN_INCORRECT 错误。

    融云的每个应用都提供用于隔离生产和开发环境的两套独立 App Key / Secret。由测试环境切换到生产环境时，很容易发生应用客户端与应用服务端 App Key 不一致的问题。

  - Token 已过期。参见[获取 Token]。

连接状态码

下表列出了连接相关错误码。

名称	值	说明
IPC_DISCONNECT	-2	IPC 进程意外终止。 可能原因： 1. 手机系统策略，导致 IPC 进程被回收或被解绑，应用层调用 IM 接口时会触发此问题，SDK 会做好自动重连，应用不需要额外处理。 2. 找不到对应 CPU 架构的 libRongIMLib.so 或 libsqlite.so，请确保集成了对应架构的 so
RC_CONN_ID_REJECT	31002	AppKey 错误，请检查您使用的 AppKey 是否正确
RC_CONN_TOKEN_INCORRECT	31004	Token 无效 请检查客户端初始化使用的 AppKey 和您服务器获取 token 时使用的 AppKey 是否一致。
RC_CONN_NOT_AUTHRORIZED	31005	App 校验未通过（开通了 App 校验功能，但是校验未通过）
RC_CONN_APP_BLOCKED_OR_DELETED	31008	应用被封禁或已删除。请检查您使用的 AppKey 是否被封禁或已删除。
RC_CONN_USER_BLOCKED	31009	连接失败，一般因为用户已被封禁。请检查您使用的 Token 是否正确，以及对应的 UserId 是否被封禁
RC_CONN_TOKEN_EXPIRE	31020	Token 过期 一般是因为在控制台设置了 token 过期时间，需要请求您的服务器重新获取 token 并再次用新的 token 建立连接。
RC_CONN_PROXY_UNAVAILABLE	31028	代理服务不可访问。
RC_CLIENT_NOT_INIT	33001	SDK 没有初始化。在使用 SDK 任何功能之前，必须先初始化。
RC_CONNECTION_EXIST	34001	连接已存在，或正在重连中。
RC_CONNECT_TIMEOUT	34006	SDK 内部连接超时，调用可设置超时的连接接口，并设置有效的 timeLimit 值时会出现该错误 SDK 不会继续重连，需要 APP 手动调用 connect 接口进行连接。
UNKNOWN	-1	未知错误。

断开连接　　断开连接

更新时间:2024-08-30

在 App 需要执行切换用户登录、注销登录的操作时，需要断开与融云的 IM 连接。SDK 支持设置断开 IM 连接之后是否允许向用户发送消息推送通知。

> 提示
>
> SDK 在前后台切换或者网络出现异常都会自动重连，会保证连接的可靠性。除非 App 逻辑需要登出，否则不需要调用此方法进行手动断开。

断开连接（允许推送）

主动断开与融云服务端的 IM 连接，并设置断开连接后允许融云服务端进行远程推送。以下两种方式效果一致：

```
RongIMClient.getInstance().disconnect()
```

或者：

```
RongIMClient.getInstance().disconnect(true)
```

参数	类型	说明
isReceivePush	Boolean	断开连接后，是否允许融云服务端进行远程推送。`true` 表示接收远程推送。`false` 表示不接收远程推送。

如果融云服务端发现 App 客户端不在线（默认要求全部设备已下线），在接收新消息时，融云服务端会为该用户记录一条离线消息[7]，并触发融云服务端的推送服务。融云服务端会通过推送通道下发一条提醒到客户端 SDK。该提醒一般以通知形式展示在通知面板，提示用户有离线消息。

断开连接（不允许推送）

主动断开与融云服务端的 IM 连接，并设置断开连接后不允许融云服务端进行远程推送。

需要注销登录（登出）或切换 App 用户账号时，推荐使用以下方法，效果一致：

```
RongIMClient.getInstance().logout()
```

或者：

```
RongIMClient.getInstance().disconnect(false)
```

参数	类型	说明
isReceivePush	Boolean	断开连接后，是否允许融云服务端进行远程推送。`true` 表示接收远程推送。`false` 表示不接收远程推送。

断开连接且不允许推送的情况下，融云服务端仅记录离线消息，但不会为当前设备触发推送服务。如果用户登录了多个设备，则在其他设备中最后一个登录的设备上可正常接收推送。在多设备场景下，App 可能需要保证设备间消息记录一致，可通过开启**多设备消息同步**实现。详见多设备消息同步。

重连机制与重连互踢　　重连机制与重连互踢 自动重连机制

更新时间:2024-08-30

SDK 内已实现自动重连机制，一旦连接成功，SDK 的重连机制将立即开始生效，并接管所有的重连处理。当因为网络原因断线时，SDK 内部会尝试重新建立连接，不需要您做额外的连接操作。

可能导致 SDK 断线重连的异常情况如下：

- **弱网环境**：可能出现 SDK 不停重连的情况。因为客户端 SDK 和融云服务端之间存在连接保活机制，一旦因如果网络太差导致心跳超时，SDK 就会触发重连操作，尝试重连直到连接成功。
- **无网环境**：SDK 的重连机制会暂停。一旦网络恢复，SDK 会进行重连操作。

> 提示
> 一旦触发连接错误的回调，SDK 将退出重连机制。请根据具体的状态码自行处理。

重连时间间隔

SDK 尝试重连时，时间间隔逐次变大，分别是 0.05s（5.6.2 之前为 0s）, 0.25s, 0.5s, 1s, 2s, 4s, 8s, 16s, 32s。之后每 64s 重试一次。

当 APP 切换到前台或者网络状态发生变化，重连时间会按照上面的时间间隔从头开始，保证这种情况下能尽快的连接成功。

主动退出重连机制

应用主动断开连接后，SDK 将退出重连机制，不再尝试重连。

重连互踢策略

重连互踢策略用于控制 SDK 自动重连成功时是否需要下线的设备。

即时通讯业务默认仅允许同一用户账号在单台移动端设备上登录。后登录的移动端设备一旦连接成功，则自动踢出之前登录的设备。在部分情况下，SDK 的重连机制可能会导致后登录设备无法正常在线。

例如，默认的重连互踢策略可能导致以下情况：

1. 用户张三尝试在移动设备 A 上登录，但因 A 设备网络不稳定导致未连接成功，触发了 SDK 的自动重连机制。
2. 用户此时尝试换到移动设备 B 上登录。B 设备连接成功，且用户可通过 B 设备正常使用即时通讯业务。
3. A 设备网络稳定之后，SDK 重连成功。因此时 A 设备为后上线设备，导致 B 设备被踢出。

修改 App 用户的重连互踢策略

如果 App 用户希望在以上场景中将重连成功的 A 设备下线，同时保持 B 设备登录，可以修改当前用户（User ID）的重连互踢策略。

> 提示
> 使用该接口要求该 App Key 已启用登录时判断用户其他端登录状态提示（移动端）。如需启用该服务，请提交工单。

设置断线重连时是否踢出重连设备。此方法需要在 **init** 之前调用。

```
RongIMClient.getInstance().setReconnectKickEnable(enable);
```

参数	类型	说明
enable	boolean	是否踢出正在重连的设备。详见下方 `enable` 参数详细说明。

`enable` 参数详细说明：

- 设置 `enable` 为 `true`：

  如果重连时发现已有别的移动端设备在线，将不再重连，不影响已正常登录的移动端设备。

- 设置 `enable` 为 `false`：

  如果重连时发现已有别的移动端设备在线，将踢出已在线的移动端设备，使当前设备上线。

多端同时在线

更新时间:2024-08-30

多端同时在线是指同一用户账号从多个平台同时连接到融云即时通讯服务的功能。默认情况下，融云即支持多端设备同时在线。该功能无需开通即可使用。

> 默认多端设备之间不会进行消息同步。如有需要，请开通多设备消息同步服务。

多端登录限制说明

默认的情况下，同一用户账号可在移动端、Web 端、桌面端、小程序端最多一个设备上同时在线。

平台类别	限制	IM SDK 支持平台列表
移动端	默认仅支持一个移动端设备连接。如需支持移动端多设备登录，请提交工单申请开通移动多端服务。	Android、iOS、Flutter、React Native、uni-app、Unity
桌面端	默认仅支持一个桌面端设备连接。	Electron 框架（通过 Web 端 SDK 的 Electron 模块支持）
Web 端	默认仅支持一个 Web 页面连接（每个浏览器标签页认为是一个连接）。在控制台自助开通多设备消息同步服务后，自动支持多 Web 页面连接。	Web
小程序端	默认仅支持一个小程序连接。如需支持小程序多设备登录，请提交工单申请开通小程序多端服务。	小程序

多设备消息同步

更新时间:2024-08-30

在即时通讯业务中，同一用户账号可能在多个设备上登录。**多设备消息同步**是融云服务端提供的一项服务，可用于同一用户账号的多个设备之间同步收发消息。

默认情况下，融云不会在设备之间同步消息。新消息被某一端设备收取后，其他端无法收取该消息。

适用场景

在融云即时通讯业务中，多设备消息同步适用于以下情况：

- 同一用户账号在多设备上同时在线（无论是否为同一端），希望同步收发消息。例如，用户可能拥有多个移动端设备，如两个 Android 设备、一个 iOS 设备。

> 注意：融云默认已支持多端同时在线，同一用户账号可在移动端、Web 端、桌面端、小程序端最多一个设备上同时在线。但是如果需要允许 App 用户同时在多个移动端设备或多个小程序端上在线，需要分别提交工单申请，详见多端同时在线。

- 同一用户账号换设备登录（无论是否曾在该设备登录过），希望同步收发的消息记录。例如用户从 Android 设备下线后，换到另一个设备从 Web 端登录。
- 同一用户账号在当前设备卸载重装 App，希望同步收发消息记录。

支持在多设备间同步的消息

并非所有消息均支持多设备消息同步。**状态消息**仅支持在多设备同时在线时同步接收，不在线的设备无法通过**多设备消息同步**收到消息。

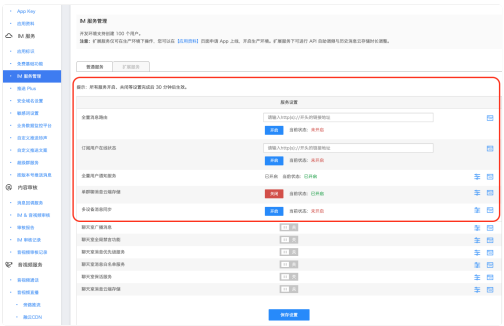
以下情况均属于状态消息：

- 融云内置消息类型中定义为状态消息类型的消息。内置状态类型消息的具体包括：正在输入状态消息（`RC:TypSts`）
- 自定义的状态消息类型的消息。详见各个客户端「自定义消息」文档。
- 使用服务端 API 状态消息接口发送的所有消息（不区分消息类型）均不支持同步。具体的 API 接口为发送单聊状态消息（`/statusmessage/private/publish.json`）、发送群聊状态消息（`/message/group/publish.json`）。

开通服务

请前往控制台，在 IM 服务管理 页面的**普通服务**标签下开通**多设备消息同步**服务。该服务在开发环境免费使用，默认为关闭状态。生产环境预存费用后才可开通服务。

> 服务开启、关闭设置完成后 30 分钟内生效。



对其他功能或业务的影响

多设备消息同步服务的状态对即时通讯业务中的离线补偿、撤回消息、聊天室业务等有影响。

对离线补偿的影响

控制台的**多设备消息同步**服务包含了融云服务端离线补偿机制的开关。

开通**多设备消息同步**服务后，融云服务端自动为 App 启用离线补偿机制。离线补偿机制会在以下场景触发：

- **换设备登录**。用户在新设备上登录后（无论是否曾在该设备登录过），SDK 可获取最近指定天数（默认离线补偿天数为 1 个自然日）在其他终端上发送和接收过的消息。
- **应用卸载重装**。消息与会话列表是存储在本地数据库，用户卸载 App 时会删除本地数据库。用户重新安装 App 后并再次连接时，会触发融云服务端离线补偿机制，SDK 可获取最近指定天数（默认离线补偿天数为 1 个自然日）在其他终端上发送和接收过的消息。

> 注意：在 **换设备登录** 或 **应用卸载重装** 场景下，离线补偿机制仅可获取到最近（默认离线补偿天数为 1 天，最大 7 天）的会话。早于该天数的会话无法通过离线补偿机制获取。因此，离线补偿后的会话列表可能与原设备上或卸载前的会话列表并不一致（您可能会有丢失部分会话的错觉）。

如需修改离线消息补偿的天数，可提交工单。建议谨慎设置离线补偿天数，当单用户消息量超大时，可能会因为补偿消息量过大，造成端上处理压力较大。

对 Web 平台连接数的影响

开通多设备消息同步服务后，可额外支持多 Web 页面连接（每个浏览器标签页也认为是一个连接），最多 10 个。

对撤回消息的影响

- 未开通多设备消息同步服务时，多端之间无法同步撤回的消息。
- 开通多设备消息同步服务后，消息发送端一旦撤回消息，如果用户在其他端在线，则其他端同步撤回该条已发送消息。如果用户在其他端不在线时，则在用户登录后同步撤回已发送的消息。

对服务端 API 发送消息的影响

通过服务端 API 发送消息时，部分接口可通过 `isIncludeSender` 指定消息发送者可否在客户端接收该已发消息。

- 未开通多设备消息同步服务时，仅在发送者已登录客户端（在线）的情况下，通过 Server API 发送的消息可即时同步到发送者的在线客户端，无法同步到离线的客户端。
- 开通多设备消息同步服务后，如果发送者未登录客户端（离线），通过 Server API 发送的消息可在再次上线时同步到发送者的在线客户端。

用户概述 用户概述

更新时间:2024-08-30

App 用户需要接入融云服务，才能使用即时通讯服务。对于融云来说，用户是指持有由融云分发的有效 Token，接入并使用即时通讯服务的 App 用户。

注册用户

应用服务端（App Server）应向融云服务端提供 App 用户的用户 ID（`userId`），以向融云换取唯一用户 Token。对融云来说，这个以 `userId` 获取 Token 的步骤即注册用户，且必须通过调用 Server API 来完成。

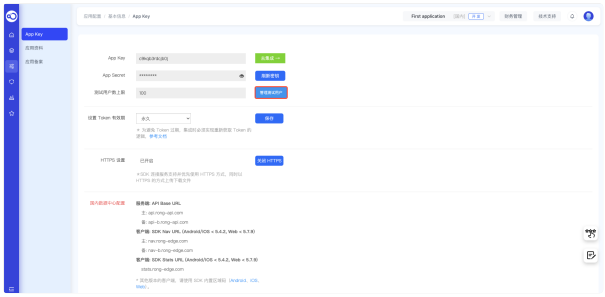
应用客户端必须持有有效 Token，才能成功连接到融云服务端，使用融云即时通讯服务。当 App 客户端用户向服务器发送登录请求时，服务器会查询数据库以检查连接请求是否匹配。

注册用户数限制

- **开发环境**中的注册用户数上限为 100 个。
- 在 **生产环境**中，升级为 **IM 旗舰版**或 **IM 尊享版**后不限制注册用户数。

删除用户

删除用户是指在应用的**开发环境**中，通过控制台删除已注册的测试用户，以控制开发环境中的测试用户总数。**生产环境**不支持该操作。



注销用户

注销用户是指在融云服务中删除用户数据。App 可使用该能力实现自身的用户销户功能，满足 App 上架或合规要求。

融云返回注销成功结果后，与用户 ID 相关数据即被删除。您可以向融云查询所有已注销用户的 ID。如有需要，您可以重新激活已被注销的用户 ID（注意，用户个人数据无法被恢复）。

仅 IM Server API 提供上述能力。

用户信息

用户信息泛指用户的昵称、头像，以及群组的群昵称、群头像等数据。融云服务端不提供用户信息托管维护服务。

IMLib SDK 中未提供与用户信息相关的客户端接口。

好友关系

App 用户之间的好友关系需要由应用服务器（App Server）自行维护。融云不会同步或保存 App 端的好友关系数据。

如果需要对客户端用户之间的消息收发行为进行限制（例如，App 的所有 `userId` 泄漏，导致某个恶意用户可越过好友关系向任意用户发送消息），可以考虑使用用户白名单服务。用户一旦开启并设置白名单，则仅可接收来自该白名单中用户的消息。

用户管理接口

功能分类	功能描述	客户端 API	服务端 API
注册用户	使用 App 用户的用户 ID 向融云换取 Token。	不提供该 API	注册用户
删除用户	参见上文删除用户。	不提供该 API	不提供该 API
废弃 Token	废弃在特定时间点之前获取的 Token。	不提供该 API	作废 Token
注销用户	注销用户是指在融云服务中停用用户 ID，并删除用户个人数据。	不提供该 API	注销用户
查询已注销用户	获取已注销的用户 ID 列表。	不提供该 API	查询已注销用户
重新激活用户 ID	在融云服务中重新启用已注销用户的 ID。	不提供该 API	重新激活用户 ID
设置融云服务端的用户信息	设置在融云推送服务中使用的用户名称与头像。	不提供该 API	未提供单独的设置接口。在注册用户时必须提供用户信息。
获取融云服务端的用户信息	获取用户在融云注册的信息，包括用户创建时间和服务端的推送服务使用的用户名称、头像 URL。	不提供该 API	获取信息
修改融云服务端的用户信息	修改在融云推送服务中使用的用户名称与头像。	不提供该 API	修改信息

功能分类	功能描述	客户端 API	服务端 API
封禁用户	禁止用户连接到融云即时通讯服务，并立即断开连接。可按时长解封或主动解封。查询被封禁用户的用户 ID、封禁结束时间。	不提供该 API	添加封禁用户、解除封禁用户、查询封禁用户
查询用户在线状态	查询某用户的在线状态。	不提供该 API	查询在线状态
加入黑名单	在用户的黑名单列表中添加用户。在 A 用户黑名单的用户无法向 A 发送消息。	加入黑名单	加入黑名单
移出黑名单	在用户的黑名单中移除用户。	移出黑名单	移出黑名单
查询黑名单	查询某用户（userId）是否已被当前用户加入黑名单。	查询用户是否在黑名单中	不提供该 API
获取黑名单列表	获取用户的黑名单列表。	获取黑名单列表	查询黑名单
用户白名单	用户一旦开启并设置白名单，则仅可接收来自该白名单中用户的消息。	不提供该 API	开启用户白名单、用户白名单状态查询、添加白名单、移出白名单、查询白名单

黑名单管理　　黑名单管理

更新时间:2024-08-30

将用户加入黑名单之后，将不再收到对方发来的任何单聊消息。

- 加入黑名单为单向操作，例如：用户 A 拉黑用户 B，代表 B 无法给 A 发消息（错误码 `405`）。但 A 向 B 发消息，B 仍然能正常接收。
- 单个用户的黑名单总人数存在上限，具体与计费套餐有关。IM 旗舰版与 IM 尊享版上限为 3000 人，其他套餐详见功能对照表中的服务限制。
- 调用服务端 API 发送单聊消息默认不受黑名单限制。如需启用限制，请在调用 API 时设置 `verifyBlacklist` 为 `1`。

加入黑名单

将指定用户（userId）加入当前用户的黑名单。操作成功后，当前用户仍可向被拉黑的用户发送消息，但被拉黑的用户无法给当前用户发送单聊消息。

```
RongIMClient.getInstance().addToBlacklist(userId, callback);
```

参数	类型	说明
userId	String	用户 ID
callback	OperationCallback	回调接口

移出黑名单

将指定用户（userId）从当前用户的黑名单中移出。

```
RongIMClient.getInstance().removeFromBlacklist(userId, callback);
```

参数	类型	说明
userId	String	用户 ID
callback	OperationCallback	回调接口

查询用户是否在黑名单中

根据用户 ID（userId）查询指定用户是否在当前用户的黑名单中。

```
RongIMClient.getInstance().getBlacklistStatus(userId,
new ResultCallback<BlacklistStatus>() {
@Override
public void onSuccess(BlacklistStatus blacklistStatus) {
}

@Override
public void onError(ErrorCode e) {
}
});
```

`BlacklistStatus` 枚举值：

- 在黑名单中：`IN_BLACK_LIST(0)`
- 不在黑名单中：`NOT_IN_BLACK_LIST(1)`

获取黑名单列表

获取当前用户的黑名单列表。

```
RongIMClient.getInstance()
.getBlacklist(
new GetBlacklistCallback() {
@Override
public void onSuccess(String[] strings) {
}

@Override
public void onError(ErrorCode e) {
}
});
```

订阅用户在线状态

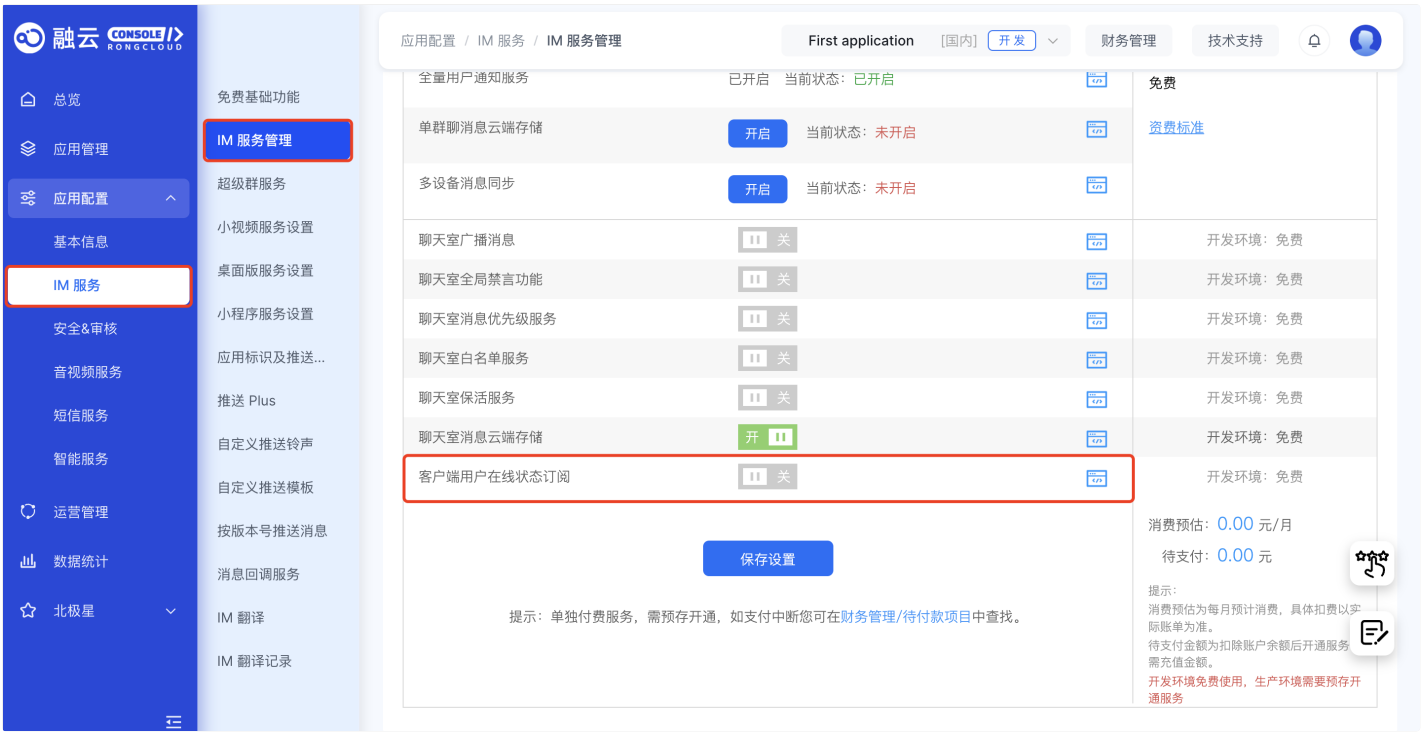
更新时间:2024-08-30

本文档旨在指导开发者如何在融云即时通讯 Android 客户端 SDK 中实现用户在线状态的订阅、查询和监听。通过本文档，Android 开发者将了解如何获取和跟踪用户在线状态,以及如何在状态变更时接收通知。

> 提示
>
> 此功能在 5.8.1 版本开始支持。

开通服务

您可以通过控制台开通服务。在融云控制台，选择 **IM 服务 > IM 服务管理 > 客户端用户在线状态订阅** ，开启服务。



订阅用户在线状态

为了跟踪特定用户的在线状态，您需要使用 `subscribeEvent` 方法进行订阅。以下是具体的操作步骤和示例代码：

1. 定义订阅用户 ID 列表：创建一个包含您希望订阅用户 ID 的 `ArrayList`，即单聊的 `targetId`。一次订阅用户的上限为 200 个，订阅的用户数最多为 1000 个，一个用户最多可以被 5000 个用户订阅。
2. 设置订阅时间：定义一个整数值,该值表示订阅的持续时间,订阅时长的范围为 60 秒到 2592000 秒。
3. 指定订阅类型：使用 `SubscribeEvent.SubscribeType.ONLINE_STATUS` 来指定订阅的是用户在线状态。
4. 使用 `subscribeEvent` 方法执行订阅操作。

示例代码

```
//设置订阅类型。
SubscribeEvent.SubscribeType type= SubscribeEvent.SubscribeType.ONLINE_STATUS;
//设置订阅时间，取值范围为[60,2592000]（单位:秒）。
int expiry=180000;
//订阅用户userId，即单聊的 targetId（一次最多订阅 200 个）。
List<String> userList=new ArrayList<>();
userList.add("user1");
userList.add("user2");
userList.add("user3");
SubscribeEventRequest request = new SubscribeEventRequest(type,expiry,userList);
RongCoreClient.getInstance().subscribeEvent(request, new IRongCoreCallback.SubscribeEventCallback<List<String>>() {
@Override
public void onSuccess() {
//订阅成功。
}

@Override
public void onError(int errorCode, List<String> strings) {
//订阅失败。
}
});
```

取消订阅用户在线状态

当您不再需要跟踪用户的在线状态时,可以使用 unSubscribeEvent 方法取消订阅。

```
//设置取消订阅的类型。
SubscribeEvent.SubscribeType type= SubscribeEvent.SubscribeType.ONLINE_STATUS;
//取消订阅用户userId，即单聊的 targetId，一次最多取消订阅 200 个。
List<String> userList=new ArrayList<>();
userList.add("user1");
userList.add("user2");
userList.add("user3");
SubscribeEventRequest request = new SubscribeEventRequest(type,userList);
RongCoreClient.getInstance().unSubscribeEvent(request, new IRongCoreCallback.SubscribeEventCallback<List<String>>() {
@Override
public void onSuccess() {
//取消订阅成功。
}

@Override
public void onError(int errorCode, List<String> strings) {
//取消订阅失败。
}
});
```

指定用户查询订阅状态信息

您可以使用 querySubscribeEvent 方法查询指定用户和订阅类型的状态信息。一次最多查询200个用户的状态信息。

```
//设置查询类型。
SubscribeEvent.SubscribeType type= SubscribeEvent.SubscribeType.ONLINE_STATUS;
//查询用户在线状态。
List<String> userList=new ArrayList<>();
userList.add("user1");
userList.add("user2");
userList.add("user3");
SubscribeEventRequest request = new SubscribeEventRequest(type,userList);
RongCoreClient.getInstance().querySubscribeEvent(request, new IRongCoreCallback.ResultCallback<List<SubscribeInfoEvent>>() {
@Override
public void onSuccess(List<SubscribeInfoEvent> subscribeInfoEvents) {
//查询成功，返回查询用户信息。
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {
//查询失败。
}
});
```

分页查询已订阅用户的状态信息

如果您需要分页获取已订阅的所有事件状态信息，可以使用 querySubscribeEvent 方法，并指定分页大小和起始索引。

- Parameter pageSize: 分页大小[1~200]。
- Parameter startIndex: 第一页传 0， 下一页取返回所有数据的数组数量（比如 pageSize = 20，第二页传 20，第三页传40）。

```
//查询类型
SubscribeEvent.SubscribeType type= SubscribeEvent.SubscribeType.ONLINE_STATUS;
SubscribeEventRequest request = new SubscribeEventRequest(type);
//分页大小，取值范围为[1,200]。
int pageSize=20;
//分页起始索引，第一页传 0， 下一页取返回所有数据的数组数量，比如 pageSize = 20，则第二页传 20，第三页传 40。
int startIndex=0;
RongCoreClient.getInstance().querySubscribeEvent(request,pageSize,startIndex, new IRongCoreCallback.ResultCallback<List<SubscribeInfoEvent>>() {
@Override
public void onSuccess(List<SubscribeInfoEvent> subscribeInfoEvents) {
//查询成功，返回查询用户信息。
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {
//查询失败。
}
});
```

监听订阅事件

为了接收订阅事件的变更通知,您需要设置订阅监听器。监听器需要在连接之前调用。

> 提示
>
> 从5.10.0版本开始，订阅事件包含了在线状态订阅、用户信息托管这2种类型，订阅事件的变更，需要根据SubscribeType订阅类型来处理对应的业务。

```
RongCoreClient.getInstance().addSubscribeEventListener(new OnSubscribeEventListener() {
/**
* @param subscribeEvents 订阅事件的列表，包含所有发生变化的事件。
* 被订阅者发生状态变更时，RCSubscribeEvent.operationType 无值。
* 订阅过期没有通知， 开发者需自行关注过期时间。
* 注意：需要判断 SubscribeInfoEvent 的 SubscribeType， 等于 ONLINE_STATUS 时代表在线状态
*/
@Override
public void onEventChange(List<SubscribeInfoEvent> subscribeEvents) {

}

/**
* 订阅数据同步完成。
* 该方法在订阅数据成功同步到设备或系统后调用，用于执行后续处理。
* @deprecated 已废弃。请使用 {@link #onSubscriptionSyncCompleted(SubscribeEvent.SubscribeType)}
*/
@Override
public void onSubscriptionSyncCompleted() {

}

/**
* 标记订阅数据同步完成。 该方法在订阅数据成功同步到设备或系统后调用，用于执行后续处理。
* 注意：需要判断 SubscribeType， 等于 ONLINE_STATUS 时代表在线状态
*
* @param type 同步完成的类型。需要通过根据类型来判断具体是哪种业务同步完成。
* @since 5.10.0
*/
@Override
public void onSubscriptionSyncCompleted(SubscribeEvent.SubscribeType type) {

}

/**
* 当用户在其他设备上的订阅信息发生变更时调用此方法。
* 这可以用于更新当前设备上的用户状态，确保订阅信息的一致性。
* 注意：需要判断 SubscribeInfoEvent 的 SubscribeType， 等于 ONLINE_STATUS 时代表在线状态
* @param subscribeEvents 订阅事件的列表
*/
@Override
public void onSubscriptionChangedOnOtherDevices(List<SubscribeEvent> subscribeEvents) {

}
});
```

用户信息托管

更新时间:2024-08-30

本文档旨在指导开发者如何在融云即时通讯 Android 客户端 SDK 中实现用户信息订阅、查询和监听，同时支持用户信息与权限的修改、查询。
通过本文档，Android 开发者将了解如何获取和跟踪用户信息，以及如何在用户信息变更、订阅状态变更时接收通知。

> 提示
> 此功能在 5.10.0 版本开始支持。

开通服务

使用此功能前，您须在控制台开通信息托管服务。

用户信息管理

可以修改或查询自己的用户信息，批量查询指定多个用户的用户信息。

用户信息设置

用户在应用中使用 `updateMyUserProfile` 可以修改自己的用户信息。

下表描述 UserProfile 类的属性，也可参考 API 文档。

属性名	类型	描述
name	String	昵称，长度不超过 32 个字符。
portraitUri	String	头像地址，长度不超过 128 个字符。
uniqueId	String	用户应用号，支持大小写字母、数字，长度不超过 32 个字符。**请注意** SDK不支持设置此字段。
email	String	Email，长度不超过 128 个字符。
birthday	String	生日，长度不超过 32 个字符
gender	int	性别，未知 0 、男 1、女 2。
location	String	所在地，长度不超过 32 个字符。
role	int	角色，支持 0~100 以内数字。
level	int	级别，支持 0~100 以内数字。
userExtProfile	long	自定义扩展信息，默认最多可以设置 20 个用户信息（以 Key、Value 方式设置，扩展用户信息通过开发者后台进行设置） 1. Key：支持大小写字母、数字，长度不超过 32 个字符，需要保障 AppKey 下唯一。Key 不存在时，设置不成功返回错误提示。 2. Value：字符串，不超过 256 个字符。

```
// 更新自己的用户信息
UserProfile userProfile = new UserProfile();
RongCoreClient.getInstance().updateMyUserProfile(userProfile, new IRongCoreCallback.UpdateUserProfileCallback() {
@Override
public void onSuccess() {
// 更新成功
}

@Override
public void onError(int errorCode, String errorKey) {
// 更新失败
}
});
```

批量获取用户信息

您可以使用 getUserProfiles 方法查询指定用户的用户信息。一次最多查询20个用户的用户信息。

```
// 设置查询用户ID列表
List<String> userIdList = new ArrayList<>();
userIdList.add("user1");
userIdList.add("user2");
userIdList.add("user3");
RongCoreClient.getInstance().getUserProfiles(userIdList, new IRongCoreCallback.ResultCallback<List<UserProfile>>() {
@Override
public void onSuccess(List<UserProfile> userProfiles) {
// 查询成功，返回查询用户信息。
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {
// 查询失败
}
});
```

获取当前用户信息

您可以使用 getMyUserProfile 方法获取当前用户信息。

```
RongCoreClient.getInstance().getMyUserProfile(new IRongCoreCallback.ResultCallback<UserProfile>() {
@Override
public void onSuccess(UserProfile userProfile) {
// 查询成功，返回自己的用户信息。
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {
// 查询失败
}
});
```

用户权限

客户端 SDK 提供了用户权限的设置和获取接口，通过 UserProfileVisibility 枚举来表示用户权限，各枚举值代表的含义参考下表：

枚举值	用户权限
UserProfileVisibility.NotSet	未设置：以 AppKey 权限设置为准，默认是此状态。
UserProfileVisibility.Invisible	都不可见：任何人都不能搜索到我的用户信息，名称、头像除外。
UserProfileVisibility.Everyone	所有人：应用中任何用户都可以查看到我的用户信息。

用户权限设置

如果您需要设置用户权限，可以使用 updateMyUserProfileVisibility 方法。

```
// 用户信息权限
UserProfileVisibility visibility = UserProfileVisibility.Everyone;
RongCoreClient.getInstance().updateMyUserProfileVisibility(visibility, new IRongCoreCallback.ResultCallback<Boolean>() {
@Override
public void onSuccess(Boolean result) {
// 设置成功
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {
// 设置失败
}
});
```

用户权限获取

如果您需要获取用户权限，可以使用 getMyUserProfileVisibility 方法。

```
// 获取自己的用户权限
RongCoreClient.getInstance().getMyUserProfileVisibility(new IRongCoreCallback.ResultCallback<UserProfileVisibility>() {
@Override
public void onSuccess(UserProfileVisibility userProfileVisibility) {
// 获取成功
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {
// 获取失败
}
});
```

用户权限说明

AppKey 默认用户信息权限为 **都不可见**，用户级别默认为 **未设置**，此状态默认可以查看用户名称和头像。
下面列举了 AppKey 级权限与 用户 级权限综合说明：

AppKey 权限	用户级权限	结果
都不可见、仅好友可见、所有人可见	未设置	以 AppKey 权限设置为准
都不可见、仅好友可见、所有人可见	设置为（都不可见、所有人可见）	以用户权限设置为准

用户搜索

按用户应用号精确搜索

支持按用户应用号搜索用户信息，可以使用 searchUserProfileByUniqueId 方法。

```
// 使用应用号查询用户信息
String uniqueId = "uniqueId";
RongCoreClient.getInstance().searchUserProfileByUniqueId(uniqueId, new IRongCoreCallback.ResultCallback<UserProfile>() {
@Override
public void onSuccess(UserProfile userProfile) {
// 查询成功
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {
// 查询失败
}
});
```

用户信息&权限变更订阅

订阅用户信息&权限变更

为了跟踪特定用户的用户信息, 您需要使用 subscribeEvent 方法进行订阅。以下是具体的操作步骤和示例代码:

1. 定义订阅用户 ID 列表: 创建一个包含您希望订阅用户 ID 的 `ArrayList` ，即单聊的 `targetId` 。一次订阅用户的上限为 200 个, 订阅的用户数最多为 1000 个，一个用户最多可以被 5000 个用户订阅。
2. 设置订阅时间：定义一个整数值,该值表示订阅的持续时间,订阅时长的范围为 60 秒到 2592000 秒。
3. 指定订阅类型: 使用 `SubscribeEvent.SubscribeType.USER_PROFILE` 来指定订阅的是用户信息。
4. 使用 `subscribeEvent` 方法执行订阅操作。

示例代码

```
//设置订阅类型。
SubscribeEvent.SubscribeType type= SubscribeEvent.SubscribeType.USER_PROFILE;
//设置订阅时间，取值范围为[60,2592000]（单位:秒）。
int expiry=180000;
//订阅用户userId，即单聊的 targetId（一次最多订阅 200 个）。
List<String> userList=new ArrayList<>();
userList.add("user1");
userList.add("user2");
userList.add("user3");
SubscribeEventRequest request = new SubscribeEventRequest(type,expiry,userList);
RongCoreClient.getInstance().subscribeEvent(request, new IRongCoreCallback.SubscribeEventCallback<List<String>>() {
@Override
public void onSuccess() {
//订阅成功。
}

@Override
public void onError(int errorCode, List<String> strings) {
//订阅失败。
}
});
```

取消用户信息&权限订阅

当您不再需要跟踪用户的用户信息时,可以使用 unSubscribeEvent 方法取消订阅。

```
//设置取消订阅的类型。
SubscribeEvent.SubscribeType type= SubscribeEvent.SubscribeType.USER_PROFILE;
//取消订阅用户userId，即单聊的 targetId，一次最多取消订阅 200 个。
List<String> userList=new ArrayList<>();
userList.add("user1");
userList.add("user2");
userList.add("user3");
SubscribeEventRequest request = new SubscribeEventRequest(type,userList);
RongCoreClient.getInstance().unSubscribeEvent(request, new IRongCoreCallback.SubscribeEventCallback<List<String>>() {
@Override
public void onSuccess() {
//取消订阅成功。
}

@Override
public void onError(int errorCode, List<String> strings) {
//取消订阅失败。
}
});
```

指定用户查询用户信息托管的订阅状态信息

您可以使用 querySubscribeEvent 方法查询指定用户和订阅类型的状态信息。一次最多查询200个用户的订阅状态信息。

```
//设置查询类型。
SubscribeEvent.SubscribeType type= SubscribeEvent.SubscribeType.USER_PROFILE;
//查询用户在线状态。
List<String> userList=new ArrayList<>();
userList.add("user1");
userList.add("user2");
userList.add("user3");
SubscribeEventRequest request = new SubscribeEventRequest(type,userList);
RongCoreClient.getInstance().querySubscribeEvent(request, new IRongCoreCallback.ResultCallback<List<SubscribeInfoEvent>>() {
@Override
public void onSuccess(List<SubscribeInfoEvent> subscribeInfoEvents) {
//查询成功，返回查询用户信息。
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {
//查询失败。
}
});
```

分页查询已订阅用户信息托管的用户的状态信息

如果您需要分页获取已订阅的所有事件状态信息，可以使用 querySubscribeEvent 方法，并指定分页大小和起始索引。

- Parameter pageSize: 分页大小 [1~200]。
- Parameter startIndex: 第一页传 0， 下一页取返回所有数据的数组数量（比如 pageSize = 20，第二页传 20，第三页传40）。

```
//查询类型
SubscribeEvent.SubscribeType type= SubscribeEvent.SubscribeType.USER_PROFILE;
SubscribeEventRequest request = new SubscribeEventRequest(type);
//分页大小，取值范围为[1,200]。
int pageSize=20;
//分页起始索引，第一页传 0， 下一页取返回所有数据的数组数量，比如 pageSize = 20，则第二页传 20，第三页传 40。
int startIndex=0;
RongCoreClient.getInstance().querySubscribeEvent(request,pageSize,startIndex, new IRongCoreCallback.ResultCallback<List<SubscribeInfoEvent>>() {
@Override
public void onSuccess(List<SubscribeInfoEvent> subscribeInfoEvents) {
//查询成功，返回查询用户信息。
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {
//查询失败。
}
});
```

监听订阅事件

为了接收订阅事件的变更通知,您需要设置订阅监听器。监听器需要在连接之前调用。

注意：从5.10.0版本开始，订阅事件包含了在线状态订阅、用户信息托管这2种类型，订阅事件的变更，需要根据SubscribeType订阅类型来处理对应的业务。

```
RongCoreClient.getInstance().addSubscribeEventListener(new OnSubscribeEventListener() {
/**
* @param subscribeEvents 订阅事件的列表，包含所有发生变化的事件。
* 被订阅者发生状态变更时，RCSubscribeEvent.operationType 无值。
* 订阅过期没有通知， 开发者需自行关注过期时间。
* 注意：需要判断 SubscribeInfoEvent 的 SubscribeType， 等于 USER_PROFILE 时代表用户信息托管
*/
@Override
public void onEventChange(List<SubscribeInfoEvent> subscribeEvents) {

}

/**
* 标记订阅数据同步完成。 该方法在订阅数据成功同步到设备或系统后调用，用于执行后续处理。
* 注意：需要判断 SubscribeType， 等于 USER_PROFILE 时代表用户信息托管
*
* @param type 同步完成的类型。需要通过根据类型来判断具体是哪种业务同步完成。
* @since 5.10.0
*/
@Override
public void onSubscriptionSyncCompleted(SubscribeEvent.SubscribeType type) {

}

/**
* 当用户在其他设备上的订阅信息发生变更时调用此方法。
* 这可以用于更新当前设备上的用户状态，确保订阅信息的一致性。
* 注意：需要判断 SubscribeInfoEvent 的 SubscribeType， 等于 USER_PROFILE 时代表用户信息托管
* @param subscribeEvents 订阅事件的列表
*/
@Override
public void onSubscriptionChangedOnOtherDevices(List<SubscribeEvent> subscribeEvents) {

}
});
```

消息介绍 消息介绍

更新时间:2024-08-30

IMLib SDK 定义了 Message 类，用于进行消息传输和管理。

Message 模型

Message 类中封装了以下关键数据：

- **用于消息传输的属性**：发送者 ID、接收者 ID、所属会话类型等。
- **消息内容体**： 用于封装一条消息携带的的具体内容，分为普通消息内容体和媒体消息内容体。例如，文本消息内容（TextMessage）属于普通消息内容体，图片消息内容（ImageMessage）属于媒体消息内容体。消息内容体的类型，常称为「消息类型」，决定了使用发送普通消息还是发送媒体的接口。

> 提示
>
> 文档里出现的「消息」，如文本消息、语音消息等，有时指继承自 `MessageContent` 或 `MediaMessageContent` 的消息具体内容。

下表描述 Message 类的关键属性，完整的属性列表可参考 API 文档。

属性名	类型	描述
ObjectName	String	消息类型的标识名，由消息内容体中的 `io.rong.imlib.MessageTag` 注解中的 value 值决定。
Content	MessageContent	消息携带的具体内容，必须为 MessageContent 或 MediaMessageContent 的子类对象，其中封装了不同类型消息的具体数据字段。即时通讯服务已提供预定义的消息类型，并规定了跨平台一致的消息内容体结构（参见消息类型概述），如文本、语音、图片、GIF 等。您也可以通过自定义消息内容体创建自定义消息类型。
ConversationType	ConversationType	会话类型枚举，例如单聊、群聊、聊天室、超级群、系统会话等。参考会话介绍。
SenderUserId	String	消息发送者的用户 ID。
TargetId	String	会话 ID（或称目标 ID），用于标识会话对端。 1. 针对单聊会话，用对端用户的 ID 作为 Target ID，因此发送与接收的所有消息的 Target ID 一定为对端的用户 ID。请注意，本端用户所接收的消息在本地消息数据中存储的 Target ID 也不是当前用户 ID，而是对端用户（消息发送者）的用户 ID。 2. 在群组、聊天室、超级群会话中，Target ID 为对应的群组、聊天室、超级群 ID。 3. 针对系统会话，因客户端用户无法回复系统会话消息，因此不需要 Target ID。
ChannelId	String	超级群频道 ID。
MessageId	int	消息 ID，消息在本地存储中的唯一 ID（数据库索引唯一值）。
UId	String	消息 UID，在同一个 App Key 下全局唯一。只有发送成功的消息才有唯一 ID。
MessageDirection	MessageDirection	消息方向枚举，分为发送和接收。
SentTime	long	消息发送时间。发送时间为消息从发送客户端到达服务器时服务器的本地时间。使用 UNIX 时间戳，单位为毫秒。
ReceivedTime	long	消息接收时间。接收时间为消息到达接收客户端时客户端的本地时间。使用 UNIX 时间戳，单位为毫秒。
SentStatus	SentStatus	发送消息的状态。如发送中、发送成功、发送失败、取消发送。
ReceivedStatus	ReceivedStatus	接收到的消息的状态，如是否已读、是否下载等。详见消息接收状态。
ReadReceiptInfo	ReadReceiptInfo	群聊消息已读回执信息。详细请参考群聊消息回执。
isCanIncludeExpansion	boolean	是否允许消息扩展。详情请参考 消息扩展。
Expansion	Map<String, String>	消息扩展信息。详情请参考 消息扩展。
Extra	String	消息的附加信息，该字段为本地操作的字段，不会发往服务端。请区别于 `MessageContent.extra`，后者会随消息一并发送到对端。
isOffline	boolean	是否是离线消息，只在接收消息的回调方法中有效，如果消息为离线消息，则为 true ，其他情况均为 false。该属性在 SDK 5.4.4 版本开始支持。

MessageContent

`Message` 类中封装了 `content` 属性，代表一条消息携带的具体内容。消息内容体的类型必须继承以下基类：

- MessageContent - 即普通消息内容体。例如，SDK 内置消息类型中的文本消息内容体（TextMessage）即继承自 `MessageContent`。
- MediaMessageContent - 即媒体消息内容体，继承自 `MessageContent` 基类，并在其基础上增加了对媒体文件的处理逻辑。。例如，SDK 内置消息类型中的文本消息内容体（ImageMessage）即继承自 `MediaMessageContent`。在发送和接收消息时，SDK 会判断消息类型是否为媒体类型消息，如果是媒体类型，则会触发上传或下载媒体文件流程。

即时通讯服务已提供预定义的、跨平台一致的消息内容体结构（参见消息类型概述），如文本、语音、图片、GIF 等。如果您需要实现自定义消息，可以继承 MessageContent 或 MediaMessageContent，创建自定义消息内容体。

消息存储策略

客户端 SDK 和即时通讯服务端通过消息注解（`MessageTag`）识别消息的类型、在本地和服务端的存储策略、是否计入未读消息数等属性。`MessageTag` 是基于 Java annotation 实现的、对消息内容类添加的注释。继承自 `MessageContent` 或 `MediaMessageContent` 的消息具体内容都必须带有 `MessageTag` 注解。

如果发送 SDK 内置消息类型的消息，则 `MessageTag` 默认自动添加，无需额外操作。

如果您需要创建自定义消息类型，则需要了解学习如何正确创建消息注解。详见以下文档：

- 自定义消息类型
- 自定义消息类型（旧版）

发送消息　　发送消息

更新时间:2024-08-30

本文主要描述了如何使用 IMLib SDK 向单聊会话、群聊会话、聊天室会话中发送消息。

> 如发送超级群消息，请参见超级群文档 收发消息。

消息内容类型简介

IMLib SDK 定义的 Message 对象的 content 属性中可包含两大类消息内容：普通消息内容和媒体消息内容。普通消息内容父类是 MessageContent，媒体消息内容父类是 MediaMessageContent。发送媒体消息和普通消息本质的区别为是否有上传数据过程。

功能	消息内容的类型	父类	描述
文本消息	TextMessage	MessageContent	文本消息的内容。
引用回复	ReferenceMessage	MessageContent	引用消息的内容，用于实现引用回复功能。
图片消息	ImageMessage	MediaMessageContent	图片消息的内容，支持发送原图。
GIF 消息	GIFMessage	MediaMessageContent	GIF 消息的内容。
文件消息	FileMessage	MediaMessageContent	文件消息的内容。
语音消息	HQVoiceMessage	MediaMessageContent	高清语音消息的内容。
提及他人（@ 消息）	不适用	不适用	@消息并非预定义的消息类型。详见如何发送 @ 消息。

以上为 IMLib SDK 内置的部分消息内容类型。您还可以创建自定义的消息内容类型，并使用 sendMessage 方法或sendMediaMessage 方法发送。详见自定义消息类型。

> 重要
>
> - 发送普通消息使用 `sendMessage` 方法，发送媒体消息使用 `sendMediaMessage` 方法。
> - 客户端 SDK 发送消息存在频率限制，每秒最多只能发送 5 条消息。

本文使用 IMLib SDK 核心类 RongCoreClient（也可以用 RongIMClient）的发送消息方法。

普通消息

普通消息指文本消息、引用消息等不涉及媒体文件上传的消息。普通消息的消息内容为 MessageContent 的子类的消息，例如文本消息内容（TextMessage），或自定义类型的普通消息内容。

构造普通消息

在发送前，需要先构造 Message 对象。conversationType 字段为会话类型。targetId 表示会话的目标 ID。以下示例中构造了一条包含文本消息内容（TextMessage）的消息对象。

```
String targetId = "Target ID";
ConversationType conversationType = Conversation.ConversationType.PRIVATE;

TextMessage messageContent = TextMessage.obtain("消息内容");

Message message = Message.obtain(targetId, conversationType, messageContent);
```

发送普通消息

如果 Message 对象包含普通消息内容，使用 IMLib SDK 核心类 RongCoreClient（也可以用 RongIMClient）的 sendMessage 方法发送消息。

```
String pushContent = null;
String pushData = null;

RongCoreClient.getInstance().sendMessage(message, pushContent, pushData, new IRongCoreCallback.ISendMessageCallback() {

@Override
public void onAttached(Message message) {

}

@Override
public void onSuccess(Message message) {

}

@Override
public void onError(Message message, IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

sendMessage 方法中直接提供了用于控制推送通知内容（pushContent）和推送附加信息（pushData）的参数。另一种方式是使用消息推送属性配置 MessagePushConfig，其中包含了 pushContent 和 pushData，并提供更多推送通知配置能力，例如标题、内容、图标、或其他第三方厂商个性化配置。详见远程推送通知。

关于是否需要配置输入参数或 Message 的 messagePushConfig 属性中 pushContent，请参考以下内容：

- 如果消息类型为即时通讯服务预定义消息类型中的用户内容类消息格式，例如 TextMessage，pushContent 可设置为 null。一旦消息触发离线推送通知时，远程推送通知默认使用服务端预置的推送通知内容。关于各类型消息的默认推送通知内容，详见用户内容类消息格式。
- 如果消息类型为即时通讯服务预定义消息类型中通知类、信令类（"撤回命令消息"除外），且需要支持远程推送通知，则必须填写 pushContent，否则收件人不在线时无法收到远程推送通知。如无需触发远程推送，可不填该字段。
- 如果消息类型为自定义消息类型，请参考自定义消息如何支持远程推送。
- 请注意，聊天室会话不支持离线消息机制，因此也不支持离线消息转推送。

参数	类型	说明
message	Message	要发送的消息体。必填属性包括会话类型（conversationType），会话 ID（targetId），消息内容（content）。详见消息介绍 中对 Message 的结构说明。
pushContent	String	修改或指定远程消息推送通知栏显示的内容。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见下文远程推送通知。
pushData	String	远程推送附加信息。对端收到远程推送消息时，可通过以下方法获取该字段内容：io.rong.push.notification.PushNotificationMessage#getPushData()。 您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见下文远程推送通知。
callback	ISendMessageCallback	发送消息的回调。

通过 RongCoreClient 的 sendMessage 方法的回调处理程序 ISendMessageCallback，融云服务器始终会通知您的消息是否已发送成功。当因任何问题导致发送失败时，可通过回调方法返回异常。

> **注意：**
>
> - 关于如何个性化配置接收方离线时收到的远程推送通知，详见下文远程推送通知。
> - 自定义消息类型默认不支持离线消息转推送机制。如需支持，详见下文自定义消息如何支持远程推送。

媒体消息

媒体消息指图片、GIF、文件等需要处理媒体文件上传的消息。媒体消息的消息内容类型为 MeidaMessageContent 的子类，例如高清语音消息内容（HQVoiceMessage），或您自定义类型的媒体消息内容。

构造媒体消息

在发送前，需要先构造 Message 对象。媒体消息 Message 对象的 content 字段必须传入 MediaMessageContent 的子类对象，表示媒体消息内容。例如图片消息内容（ImageMessage）、GIF 消息内容（GIFMessage），或继承自 MediaMessageContent 的自定义媒体消息内容。

图片消息内容（ImageMessage）支持设置为发送原图。

```
String targetId = "Target ID";
ConversationType conversationType = Conversation.ConversationType.ULTRA_GROUP;

Uri localUri = Uri.parse("file://图片的路径");//图片本地路径，接收方可以通过 getThumUri 获取自动生成的缩略图 Uri
boolean mIsFull = true; //是否发送原图
ImageMessage mediaMessageContent = ImageMessage.obtain(localUri, mIsFull);

Message message = Message.obtain(targetId, conversationType, mediaMessageContent);
```

在发送前，图片会被压缩质量，以及生成缩略图，在聊天界面中展示。GIF 无缩略图，也不会被压缩。

- **图片消息的缩略图**：SDK 会以原图 30% 质量生成符合标准大小要求的大图后再上传和发送。压缩后最长边不超过 240 px。缩略图用于在聊天界面中展示。
- **图片**：发送消息时如未选择发送原图，SDK 会以原图 85% 质量生成符合标准大小要求的大图后再上传和发送。压缩后最长边不超过 1080 px。

一般情况下不建议修改 SDK 默认压缩配置。如需调整 SDK 压缩质量，详见知识库文档如何修改 SDK 默认的图片与视频压缩配置。

发送媒体消息

发送媒体消息需要使用 sendMediaMessage 方法。SDK 会为图片、小视频等生成缩略图，根据默认压缩配置进行压缩，再将图片、小视频等媒体文件上传到融云默认的文件服务器（文件存储时长），上传成功之后再发送消息。图片消息如已设置为发送原图，则不会进行压缩。

```
String pushContent = null;
String pushData = null;

RongCoreClient.getInstance().sendMediaMessage(message, pushContent, pushData, new IRongCoreCallback.ISendMediaMessageCallback() {
@Override
public void onProgress(Message message, int i) {

}

@Override
public void onCanceled(Message message) {

}

@Override
public void onAttached(Message message) {

}

@Override
public void onSuccess(Message message) {

}

@Override
public void onError(final Message message, final IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

sendMediaMessage 方法中直接提供了用于控制推送通知内容（pushContent）和推送附加信息（pushData）的参数。另一种方式是使用消息推送属性配置 MessagePushConfig，其中包含了 pushContent 和 pushData，并提供更多推送通知配置能力，例如标题、内容、图标、或其他第三方厂商个性化配置。详见远程推送通知。

关于是否需要配置输入参数或 Message 的 messagePushConfig 属性中 pushContent，请参考以下内容：

- 如果消息类型为即时通讯服务预定义消息类型中的用户内容类消息格式，例如 HQVoiceMessage，`pushContent` 可设置为 `null`。一旦消息触发离线推送通知时，远程推送通知默认使用服务端预置的推送通知内容。关于各类型消息的默认推送通知内容，详见用户内容类消息格式。
- 如果消息类型为即时通讯服务预定义消息类型中通知类、信令类（"撤回命令消息" 除外），且需要支持远程推送通知，则必须填写 `pushContent`，否则收件人不在线时无法收到远程推送通知。如无需触发远程推送，可不填该字段。
- 如果消息类型为自定义消息类型，请参考自定义消息如何支持远程推送。
- 请注意，聊天室会话不支持离线消息机制，因此也不支持离线消息转推送。

参数	类型	说明
message	Message	要发送的消息体。必填属性包括会话类型（conversationType），会话 ID（targetId），消息内容（content）。详见消息介绍 中对 Message 的结构说明。
pushContent	String	修改或指定远程消息推送通知栏显示的内容。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见下文远程推送通知。
pushData	String	远程推送附加信息。对端收到远程推送消息时，可通过以下方法获取该字段内容：io.rong.push.notification.PushNotificationMessage#getPushData()。 您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见下文远程推送通知。
callback	ISendMediaMessageCallback	发送媒体消息的回调

通过 RongCoreClient 的 sendMediaMessage 方法的回调处理程序 ISendMediaMessageCallback，融云服务器始终会通知媒体文件上传进度，以及您的消息是否已发送成功。当因任何问题导致发送失败时，可通过回调方法返回异常。

发送媒体消息的方法默认将媒体文件上传到融云的文件服务器，您可以在客户端应用程序 下载媒体消息文件。融云对上传媒体文件大小进行了限制，GIF 大小限制为 2 MB，文件上传限制为 100 MB。如果您需要提高上限，可提交工单。

注意：

- 关于如何个性化配置接收方离线时收到的远程推送通知，详见下文远程推送通知。
- 自定义消息类型默认不支持离线消息转推送机制。如需支持，详见下文自定义消息如何支持远程推送。

发送媒体消息并且上传到自己的服务器

您可以直接发送您服务器上托管的文件。将媒体文件的 URL（表示其位置）作为参数，在构建媒体消息内容时传入。在这种情况下，您的文件不会托管在融云服务器上。当您发送带有远程 URL 的文件消息时，文件大小没有限制，您可以直接使用 sendMessage 方法发送消息。

如果您希望 SDK 在您上传成功后发送消息，您可以使用 sendMediaMessage 方法，在回调接口 ISendMediaMessageCallbackWithUploader 的 onAttached 回调方法中自行实现媒体文件上传，并在上传成功后通知 SDK，提供回媒体文件的远端地址。SDK 会在收到上传成功的通知后发送消息。

```
String path = "file://图片的路径";
Uri localUri = Uri.parse(path);

ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "目标 ID";
ImageMessage imageMessage = ImageMessage.obtain(localUri, localUri);

Message message = Message.obtain(targetId, conversationType, imageMessage);

IRongCoreCallback.ISendMediaMessageCallbackWithUploader sendMediaMessageCallbackWithUploader =
new IRongCoreCallback.ISendMediaMessageCallbackWithUploader() {
@Override
public void onAttached(
Message message, IRongCoreCallback.MediaMessageUploader uploader) {
/*上传图片到自己的服务器*/
uploadImg(imgMsg.getPicFilePath(), new UploadListener() {
@Override
public void onSuccess(String url) {
// 上传成功，回调 SDK 的 success 方法，传递回图片的远端地址
uploader.success(Uri.parse(url));
}


@Override
public void onProgress(float progress) {
//刷新上传进度
uploader.update((int) progress);
}


@Override
public void onFail() {
// 上传图片失败，回调 error 方法。
uploader.error();
}
});

}

@Override
public void onError(
final Message message, final IRongCoreEnum.CoreErrorCode coreErrorCode) {
//发送失败
}

@Override
public void onProgress(Message message, int progress) {
//发送进度
}

@Override
public void onSuccess(Message message) {
//发送成功
}

@Override
public void onCanceled(Message message) {
//发送取消
}
};

RongCoreClient.getInstance().sendMediaMessage(
message, pushContent, pushData, sendMediaMessageCallbackWithUploader);
```

如何发送 @ 消息

@消息在融云即时通讯服务中不属于一种预定义的消息类型（详见服务端文档 消息类型概述 ）。融云通过在消息中携带 mentionedInfo 数据，帮助 App 实现提及他人（@）功能。

消息的 MessageContent 中的 MentionedInfo 字段存储了携带所 @ 人员的信息。无论是 SDK 内置消息类型，或者您自定义的消息类型，都直接或间接继承了 MessageContent 类。

融云支持在向群组和超级群中发消息时，在消息内容中添加 mentionedInfo。消息发送前，您可以构造 MentionedInfo，并设置到消息的 MessageContent 中。

```
List<String> userIdList = new ArrayList<>();
userIdList.add("userId1");
userIdList.add("userId2");
MentionedInfo mentionedInfo = new MentionedInfo(MentionedInfo.MentionedType.PART, userIdList, null);
TextMessage messageContent = TextMessage.obtain("文本消息");
messageContent.setMentionedInfo(mentionedInfo);
```

MentionedInfo 参数：

参数	类型	说明
type	MentionedType	（必填）指定 MentionedInfo 的类型。MentionedType.ALL 表示需要提及（@）所有人。MentionedType.PART 表示需要 @ 部分人，被提及的人员需要在 userIdList 中指定。
userIdList	List<String>	被提及（@）用户的 ID 集合。当 type 为 MentionedType.PART 时必填；从 5.3.1 版本开始，支持当 type 为 MentionedType.ALL 时同时在 userIdList 中提及部分人。接收端可通过 mentionedInfo 获取 userIdList 数据。
mentionedContent	String	触发离线消息推送时，通知栏显示的内容。如果是 NULL，则显示默认提示内容（“有人 @ 你”）。@消息携带的 mentionedContent 优先级最高，会覆盖所有默认或自定义的 pushContent 数据。

以下示例展示了发送一条提及部分用户的文本消息，该消息发往一个群聊会话。

```
List<String> userIdList = new ArrayList<>();
userIdList.add("userId1");
userIdList.add("userId2");
MentionedInfo mentionedInfo = new MentionedInfo(MentionedInfo.MentionedType.PART, userIdList, null);
TextMessage messageContent = TextMessage.obtain("文本消息");
messageContent.setMentionedInfo(mentionedInfo);
Message message = Message.obtain(mTargetId, ConversationType.GROUP, messageContent);
RongCoreClient.getInstance().sendMessage(message, null , null, new IRongCoreCallback.ISendMessageCallback(){
/**
* 消息发送前回调，回调时消息已存储数据库
* @param message 已存库的消息体
*/
@Override
public void onAttached(Message message) {

}
/**
* 消息发送成功。
* @param message 发送成功后的消息体
*/
@Override
public void onSuccess(Message message) {

}

/**
* 消息发送失败
* @param message 发送失败的消息体
* @param errorCode 具体的错误
*/
@Override
public void onError(Message message, IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

IMLib SDK 接收消息后，您需要处理 @ 消息中的数据。您可以在获取 Message(消息对象)后，通过以下方法获取到消息对象携带的 MentionedInfo 对象。

```
MentionedInfo mentionedInfo = message.getContent().getMentionedInfo();
```

远程推送通知

> 提示
>
> 聊天室会话不支持离线消息机制，因此也不支持离线消息转推送。

如果您的应用已经在融云配置第三方推送，在消息接收方离线时，融云服务端会根据消息类型、接收方支持的推送通道、接收方的免打扰设置等，决定是否触发远程推送。

远程推送通知一般会展现在系统的通知栏。融云内置的消息类型默认会在通知栏展现通知标题和通知内容。关于各类型消息的默认推送通知内容，详见用户内容类消息格式。

如果您需要个性化的离线推送通知，可以通过以下方式，修改或指定远程推送的通知标题、通知内容其他属性。

- sendMessage 和 sendMediaMessage 方法的输入参数中直接提供了用于控制推送通知内容（`pushContent`）参数。
- 如果您需要控制离线推送通知的更多属性，例如标题、内容、图标、或根据第三方厂商通道作个性化配置，请使用 Message 的推送属性（`setMessagePushConfig`）方法进行配置。消息推送属性中的 `pushContent` 配置会覆盖发送消息接口中的 `pushContent`。

配置消息推送属性

在发送消息时，您可以通过设置消息的 MessagePushConfig 对象，对单条消息的推送行为进行个性化配置。例如：

- 自定义推送标题、推送通知内容
- 自定义通知栏图标
- 添加远程推送附加信息
- 越过接收客户端的配置，强制在推送通知内显示通知内容
- 其他 APNs 或 Android 推送通道支持的个性化配置

相对于发送消息方法输入参数中的 pushContent 和 pushData，MessagePushConfig 中的配置具有更高优先级。发送消息时，如已配置 MessagePushConfig，则优先使用 MessagePushConfig 中的配置。

```
Message message = Message.obtain(mTargetId, mConversationType, textMessage);
MessagePushConfig messagePushConfig = new MessagePushConfig.Builder().setPushTitle(title)
.setPushContent(content)
.setPushData(data)
.setForceShowDetailContent(forceDetail)
.setAndroidConfig(new AndroidConfig.Builder()
.setNotificationId(id)
.setChannelIdHW(hw)
.setCategoryHW("IM")
.setChannelIdMi(mi)
.setChannelIdOPPO(oppo)
.setTypeVivo(vivo ? AndroidConfig.SYSTEM : AndroidConfig.OPERATE)
.setCategoryVivo("IM")
.build())
.setIOSConfig(new IOSConfig(threadId, apnsId)).setTemplateId("")
build();
message.setMessagePushConfig(messagePushConfig);

// 请根据消息类型调用对应的发送方法
```

`MessagePushConfig` 属性说明如下：

参数	类型	说明
disablePushTitle	boolean	是否屏蔽通知标题。此属性只针目标用户为 iOS 平台时有效，Android 第三方推送平台的通知标题为必填项，所以暂不支持。
pushTitle	String	推送标题，此处指定的推送标题优先级最高。如不设置，可参考 用户内容类消息格式 中对各内置消息类型默认推送通知标题与推送通知内容的说明。
pushContent	String	推送内容。此处指定的推送内容优先级最高。如不设置，可参考 用户内容类消息格式 中对各内置消息类型默认推送通知标题与推送通知内容的说明。
pushData	String	远程推送附加信息。如不设置，则使用消息发送参数中设置的 `pushData` 值。对端收到远程推送通知时，可通过以下方法获取该字段内容：`io.rong.push.notification.PushNotificationMessage#getPushData()`。
forceShowDetailContent	boolean	是否越过目标客户端的配置，强制在推送通知内显示通知内容（`pushContent`）。 客户端设备可通过 `setPushContentShowStatus` 设置在接收推送通知时仅显示类似「您收到了一条通知」的提醒。发送消息时，可设置 `forceShowDetailContent` 为 1 越过该配置，强制目标客户端在此条消息的推送通知中显示推送内容。
iOSConfig	IOSConfig	iOS 平台推送通知配置。目标端设备为 iOS 平台设备时，适用该配置。详细说明见 `iOSConfig` 属性说明。
androidConfig	AndroidConfig	Android 平台推送通知配置。目标端设备为 Android 平台设备时，适用该配置。详细说明见 `AndroidConfig` 属性说明。
templateId	String	推送模板 ID。根据目标客户端用户通过 `RongIMClient` 中的 `setPushLanguageCode` 设置的语言环境，匹配模板 ID 所对应的模板中设置的语言内容进行推送。未匹配成功时使用默认内容进行推送。 模板内容在控制台 > 自定义推送文案中进行设置，具体操作请参见 配置和使用自定义多语言推送模板。

- `iOSConfig` 属性说明

参数	类型	说明
threadId	String	iOS 平台通知栏分组 ID，相同的 `threadId` 推送分为一组（iOS10 开始支持）。
apnsCollapseId	String	iOS 平台通知覆盖 ID，`apnsCollapseId` 相同时，新收到的通知会覆盖老的通知，最大 64 字节（iOS10 开始支持）。
richMediaUri	String	iOS 推送自定义的通知栏消息右侧图标 URL，需要 App 自行解析 `richMediaUri` 并实现展示。图片要求：大小 120 * 120px，格式为 png 或者 jpg 格式。SDK 从 5.2.4 版本开始支持携带该字段。
interruptionLevel	String	适用于 iOS 15 及之后的系统。取值为 `passive`，`active`（默认），`time-sensitive`，或 `critical`，取值说明详见对应的 APNs 的 interruption-level 字段。在 iOS 15 及以上版本中，系统的“定时推送摘要”、“专注模式” 都可能导致重要的推送通知（例如余额变化）无法及时被用户感知的情况，可考虑设置该字段。SDK 5.6.7 及以上版本支持该字段。

- `AndroidConfig` 属性说明

参数	类型	说明
notificationId	String	Android 平台 Push 唯一标识，目前支持小米、华为推送平台，默认开发者不需要进行设置，当消息产生推送时，消息的 `messageUId` 作为 `notificationId` 使用。
channelIdMi	String	小米推送通知渠道的 ID。详细使用方式及创建方法请参见第三方文档 小米推送消息分类新规。
imageUrlMi	String	**（由于小米官方已停止支持该能力，该字段已失效）**小米通知类型的推送所使用的通知图片 url。图片要求：大小120 * 120px，格式为 png 或者 jpg 格式。 此属性 5.1.7 及以上版本支持。支持 MIUI 国内版（国内版要求为 MIUI 12 及以上）和国际版。
channelIdHW	String	华为推送通知渠道的 ID。详细使用方式及创建方法请参见第三方文档 华为自定义通知渠道。更多通知渠道信息，请参见 Android 官方文档。
imageUrlHW	String	华为推送通知中自定义的通知栏消息右侧小图片 URL，如果不设置，则不展示通知栏右侧图片。图标文件须小于 512 KB，图标建议规格大小：40dp x 40dp，弧角大小为 8dp，超出建议规格大小的图标会存在图片压缩或显示不全的情况。 此属性 5.1.7 及以上版本支持。
importanceHW	String	华为推送的消息提醒级别。`LOW` 表示通知栏消息预期的提醒方式为静默提醒，消息到达手机后，无铃声震动。`NORMAL` 表示通知栏消息预期的提醒方式为强提醒，消息到达手机后，以铃声、震动提醒用户。终端设备实际消息提醒方式将根据 `categoryHW` 字段取值、或者控制台配置的 `category` 字段取值，或者华为智能分类结果进行调整。SDK 5.1.3 及以上版本支持该字段。
categoryHW	String	华为推送通道的消息自分类标识，默认为空。category 取值必须为大写字母，例如 `IM`。App 根据华为要求完成华为自分类权益申请 或 申请特殊权限 后可传入该字段有效。详见华为推送官方文档华为消息分类标准。该字段优先级高于控制台为 App Key 下的应用标识配置的华为推送 Category。SDK 5.4.0 及以上版本支持该字段。
imageUrlHonor	String	荣耀推送通知中用户自定义的通知栏右侧大图标 URL，如果不设置，则不展示通知栏右侧图标。图标文件须小于 512 KB，图标建议规格大小：40dp x 40dp，弧角大小为 8dp，超出建议规格大小的图标会存在图片压缩或显示不全的情况。SDK 5.6.7 及以上版本支持该字段。
importanceHonor	String	荣耀推送的 Android 通知消息分类，决定用户设备消息通知行为。`LOW` 表示资讯营销类消息。`NORMAL`（默认值）表示服务与通讯类消息。SDK 5.6.7 及以上版本支持该字段。

参数	类型	说明
typeVivo	String	VIVO 推送服务的消息类别。可选值 `0`（运营消息）和 `1`（系统消息）。该参数对应 VIVO 推送服务的 `classification` 字段，详见 VIVO 推送消息分类说明。
categoryVivo	String	VIVO 推送服务的消息二级分类。例如 `IM`（即时消息）。该参数对应 VIVO 推送服务的 `category` 字段。详细的 category 取值请参见 VIVO 推送消息分类说明。如果指定二级分类 `categoryVivo`，必须同时指定 `typeVivo`（系统消息或运营消息）。请注意遵照 VIVO 官方要求，确保二级分类属于 VIVO 系统消息场景或运营消息场景下允许发送的内容。`categoryVivo` 字段优先级高于控制台为 App Key 下的**应用标识**配置的 VIVO 推送 Category。SDK 5.4.2 及以上版本支持该字段。
channelIdOPPO	String	OPPO 推送通知渠道的 ID。详细使用方式及创建方法请参见第三方文档 OPPO PUSH 通道升级说明。
channelIdFCM	String	FCM 推送通知渠道的 ID。应用程序必须先创建一个具有此频道 ID 的频道，然后才能收到具有此频道 ID 的任何通知。更多信息请参见安卓官方文档。
collapseKeyFCM	String	FCM 推送的通知分组 ID。SDK 5.1.3 及以上版本支持该字段。**注意**，如使用该字段，请确保控制台的 FCM 推送配置中推送方式为**通知消息方式**。
imageUrlFCM	String	FCM 推送的通知栏右侧图标 URL。如果不设置，则不展示通知栏右侧图标。SDK 5.1.3 及以上版本支持该字段。**注意**，如使用该字段，请确保控制台的 FCM 推送配置鉴权方式为**证书**，推送方式为**通知消息方式**。

自定义消息如何支持远程推送

融云为内置的消息类型默认支提供了远程通知标题和通知内容（详见用户内容类消息格式）。不过，如果您发送的是自定义类型的消息，则需要您自行提供 pushContent 字段，否则用户无法收到离线推送通知。具体如下：

- 如果消息类型为自定义消息类型，且需要支持离线推送通知，则必须向融云提供 `pushContent` 字段，否则用户无法收到离线推送通知。
- 如果消息类型为自定义消息类型，但不需要支持远程推送通知（例如通过自定义消息类型实现的 App 业务层操作指令），可将 `pushContent` 字段留空。

> 提示
>
> 如果要自定义消息启用离线推送通知能力，请务必发送消息的入参或消息推送属性向融云提供 pushContent 字段内容，否则接收方用户无法收到离线推送通知。

为消息禁用推送通知

在接收者未上线时，融云服务端默认触发推送服务，将消息通过推送通道下发（服务端是否触发推送也会收到应用级别、用户级别免打扰设置的影响）。

您的 App 用户可能希望在发送消息时就指定该条消息不需要触发推送，该需求可通过 Message 对象的 MessageConfig 配置实现。

以下示例中，我们将 messageConfig 的 disableNotification 设置为 true 禁用该条消息的推送通知。接收方再次上线时会通过融云服务端的离线消息缓存（最多缓存 7 天）自动收取单聊、群聊、系统会话消息。超级群消息因不支持离线消息机制，需要主动拉取。

```
String targetId = "目标 ID";
ConversationType conversationType = ConversationType.PRIVATE;
TextMessage messageContent = TextMessage.obtain("消息内容");
Message message = Message.obtain(targetId, conversationType, messageContent);

message.setMessageConfig(new MessageConfig.Builder().setDisableNotification(true).build());
```

如何透传自定义数据

如果应用需要将自定义数据透传到对端，可通过以下方式实现：

- 消息内容 MessageContent 的附加信息字段，**该字段会随即时消息一并发送到对端**。接收方在线接收消息后，可从消息内容中获取该字段。请区别于 `Message.extra`，`Message.extra` 为本地操作的字段，不会被发往服务端。
- 远程推送附加信息 `pushData`。您可以在 sendMessage 和 sendMediaMessage 的输入参数中直接设置 `pushData`，也可以使用消息推送属性中的同名字段，后者中的 `pushData` 会覆盖前者。pushData **仅会在消息触发离线远程推送时下发到对端设备**。对端收到远程推送通知时，可通过以下方法获取该字段内容：`io.rong.push.notification.PushNotificationMessage#getPushData()`。

如何处理消息发送失败

对于客户端本地会存储的消息类型（参见消息类型概述），如果触发（onAttached）回调，表示此时该消息已存入本地数据库，并已进入本地消息列表。

- 如果入库失败，您可以检查参数是否合法，设备存储是否可正常写入。
- 如果入库成功，但消息发送失败（`onError`），App 可以在 UI 临时展示这条发送失败的消息，并缓存 `onError` 抛出的 `Message` 实例，在合适的时机重新调用 `sendMessage` / `sendMediaMessage` 发送。请注意，如果不复用该消息实例，而是重发相同内容的新消息，本地消息列表中会存储内容重复的两条消息。

对于客户端本地不存储的消息，如果发送失败（onError），App 可缓存消息实例再重试发送，或直接新建消息实例进行发送。

如何实现转发、群发消息

转发消息：IMLib SDK 未提供转发消息接口。App 实现转发消息效果时，可调用发送消息接口发送。

群发消息：群发消息是指向多个用户发送消息。更准确地说，是向多个 Target ID 发送消息，一个 Target ID 可能代表一个用户，一个群组，或一个聊天室。客户端 SDK 未提供直接实现群发消息效果的 API。您可以考虑以下实现方式：

- App 循环调用客户端的发送消息 API。请注意，客户端 SDK 限制发送消息的频率上限为每秒 5 条。
- App 在业务服务端接入即时通讯服务端 API（IM Server API），由 App 业务服务端群发消息。IM Server API 的 发送单聊消息 接口支持单次向多个用户发送单聊消息。您也可以通过 发送群聊消息、发送聊天室消息接口实现单次向多个群组或多个聊天室发送消息。

接收消息　　接收消息

更新时间:2024-08-30

开发者拦截 SDK 接收的消息，并进行相应的业务操作。

监听消息接收

应用程序可以通过 addOnReceiveMessageListener 方法设置多个消息接收监听器。所有接收到的消息都会在 OnReceiveMessageWrapperListener 监听器的方法中回调。建议在应用生命周期内注册消息监听。

```
RongCoreClient.addOnReceiveMessageListener(
new io.rong.imlib.listener.OnReceiveMessageWrapperListener() {
@Override
public boolean onReceivedMessage(Message message, ReceivedProfile profile) {
int left = profile.getLeft();
boolean isOffline = profile.isOffline();
boolean hasPackage = profile.hasPackage();
}
});
```

消息接收监听器 OnReceiveMessageWrapperListener，可接收实时消息或离线消息。

当客户端连接成功后，服务端会将所有离线消息[?]以消息包（Package）的形式下发给客户端，每个 Package 中最多含 200 条消息。客户端会解析 Package 中的消息，逐条上抛并通知应用。

SDK 接收到消息时会触发以下方法。

```
public abstract void onReceivedMessage(Message message, ReceivedProfile profile);
```

ReceivedProfile 中封装了当前与接收消息相关的数据：

- `ReceivedProfile` 中的 `left` 为当前正在解析的消息包（Package）中还剩余的消息条数。
- `ReceivedProfile` 中的 `hasPackage` 表示当前是否在服务端还存在未下发的消息包（Package）。
- `ReceivedProfile` 中的 `offline` 表示当前消息是否为离线消息。

同时满足以下条件，表示离线消息已收取完毕：

- `hasPackage` 为 `false`：表示当前正在解析最后一包消息。
- `left` 为 0：表示最后一个消息包中最后一条消息已接收完毕。

从 5.2.3 版本开始，每次连接成功后，离线消息收取完毕时会触发以下回调方法。如果没有离线消息，连接成功后会立即触发。

```
public void onOfflineMessageSyncCompleted() {
//
}
```

为了避免内存泄露，请在不需要监听时调用 removeOnReceiveMessageListener 移除监听器。

```
RongCoreClient.removeOnReceiveMessageListener(listener);
```

消息接收状态

Message 类中封装了 ReceivedStatus，使用了以下状态表示接收到的消息的状态。

状态	描述
isRead()	是否已读。如果消息在当前设备上被阅读，该状态会变为已读。SDK 5.6.8 版本开始，只要在其他设备上阅读过该消息，当前设备的该状态值会也变为已读。
isListened()	是否已被收听，仅用于语音消息。
isDownload()	是否已被下载，仅适用于媒体消息。
isRetrieved()	该消息是否已被同时在线或之前登录的其他设备接收。只要其他设备先收到该消息，该状态值都会变为已接收。

禁用消息排重机制

消息排重机制会在 SDK 接收单聊、群聊、系统消息、聊天室时自动去除内容重复消息。当 App 本地存在大量消息，SDK 默认的排重机制可能会因性能问题导致收消息卡顿。因此在接收消息发生卡顿问题时，可尝试关闭 SDK 的排重机制。

为什么接收消息可能出现消息重复

发送端处于弱网情况下可能出现该问题。A 向 B 发送消息后，消息成功到达服务端，并成功下发到接收者 B。但 A 由于网络等原因可能未收到服务端返回的 ack，导致 A 认为没有发送成功。此时如果 A 重发消息，此时 B 就会收到与之前重复的消息（消息内容相同，但 Message UID 不同）。

关闭消息排重

> 提示
>
> SDK 从 5.3.4 版本开始支持关闭消息排重。仅在 RongCoreClient 中提供该接口。

请在 SDK 初始化之后，建立 IM 连接之前调用。多次调用以最后一次为准。

```
boolean enableCheck = false // 关闭消息排重
RongCoreClient.getInstance().setCheckDuplicateMessage(enableCheck)
```

> 聊天室消息从 5.8.2 版本开始支持关闭消息排重。在 RongChatRoomClient 中提供。

请在 SDK 初始化之后，建立 IM 连接之前调用。多次调用以最后一次为准。

```
boolean enableCheck = false // 关闭消息排重
RongChatRoomClient.getInstance().setCheckChatRoomDuplicateMessage(enableCheck)
```

获取历史消息　　获取历史消息

更新时间:2024-08-30

获取历史消息可以仅从本地数据中获取，仅从远端获取，和同时从本地与远端获取。

开通服务

从远端获取单群聊历史消息是指从融云服务端获取历史消息，该功能要求 App Key 已启用融云提供的**单群聊消息云端存储**服务。您可以在控制台 IM 服务管理 页面为当前使用的 App Key 开启服务。如果使用生产环境的 App Key，请注意仅 **IM 旗舰版**或 **IM 尊享版**可开通该服务。具体功能与费用以融云官方价格说明 页面及计费说明 文档为准。

提示：请注意区分历史消息记录与离线消息[7]。融云针对单聊、群聊、系统消息默认提供最多 7 天（可调整）的离线消息缓存服务。客户端上线时 SDK 会自动收取离线期间的消息，无需 App 层调用 API。详见管理离线消息存储配置。

从本地数据库中获取消息

使用 `getHistoryMessages` 方法可分页查询指定会话存储在本地数据库中的历史消息，并返回消息对象列表。列表中的消息按发送时间从新到旧排列。

获取会话中所有类型的消息

```
RongIMClient.getInstance().getHistoryMessages(conversationType, targetId, oldestMessageId, count,callback);
```

`count` 参数表示返回列表中应包含多少消息。`oldestMessageId` 参数用于控制分页的边界。每次调用 `getHistoryMessages` 方法时，SDK 会以 `oldestMessageId` 参数指向的消息为界，继续在下一页返回指定数量的消息。如果需要获取会话中最新的 `count` 条消息，可以将 `oldestMessageId` 设置为 -1。

建议获取返回结果中最早一条消息的 ID，并在下一次调用时作为 `oldestMessageId` 传入，以便遍历整个会话的消息历史记录。

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID
oldestMessageId	long	最后一条消息的 ID。如需查询本地数据库中最新的消息，设置为 –1。
count	int	每页消息的数量。
callback	ResultCallback<List<Message>>	获取历史消息的回调，按照时间顺序从新到旧排列

```
Conversation.ConversationType conversationType = Conversation.ConversationType.PRIVATE;
String targetId = "会话 Id";
int oldestMessageId = -1;
int count = 10;

RongIMClient.getInstance().getHistoryMessages(conversationType, targetId, oldestMessageId, count,
new RongIMClient.ResultCallback<List<Message>>() {
/**
* 成功时回调
* @param messages 获取的消息列表
*/
@Override
public void onSuccess(List<Message> messages) {
}

/**
* 错误时回调。
* @param e 错误码
*/
@Override
public void onError(RongIMClient.ErrorCode e) {
}
});
```

获取会话中指定类型的消息

```
RongIMClient.getInstance().getHistoryMessages(conversationType, targetId, objectName, oldestMessageId, count, callback);
```

`count` 参数表示返回列表中应包含多少消息。`oldestMessageId` 参数用于控制分页的边界。每次调用 `getHistoryMessages` 方法时，SDK 会以 `oldestMessageId` 参数指向的消息为界，继续在下一页返回指定数量的消息。如果需要获取会话中最新的 `count` 条消息，可以将 `oldestMessageId` 设置为 -1。`objectName` 参数指定需要获取的消息类型。

建议获取返回结果中最早一条消息的 ID，并在下一次调用时作为 `oldestMessageId` 传入，以便遍历整个会话的消息历史记录。

参数	类型	说明
conversationType	ConversationType	会话类型。
targetId	String	会话 ID。
objectName	String	消息类型标识。内置消息类型的标识可参见消息类型概述。
oldestMessageId	long	最后一条消息的 ID。如需查询本地数据库中最新的消息，设置为 –1。
count	int	每页消息的数量。

参数	类型	说明
callback	ResultCallback<List<Message>>	获取历史消息的回调，按照时间顺序从新到旧排列。

获取会话中指定时间戳前后的 N 条消息

您可以通过 getHistoryMessages 方法，获取指定时间戳前后的 N 条本地消息。如果指定的消息数量大于实际的消息数，则返回实际数量的消息。

```
RongIMClient.getInstance().getHistoryMessages(conversationType, targetId, sentTime, before, after, callback);
```

在此方法中，before 参数用于指定您希望获取的、发送时间早于 sentTime 的消息数量；after 参数则用于指定您希望获取的、发送时间晚于 sentTime 的消息数量。sentTime 参数是确定消息检索范围的关键时间点。
每次调用 getHistoryMessages 方法时，将获取到会话中发送时间为 sentTime 的消息，以及该消息之前 before 条和之后 after 条的消息。

参数	类型	说明
conversationType	ConversationType	会话类型。
targetId	String	会话 ID。
sentTime	long	消息的发送时间戳。
before	int	发送时间早于 sentTime 的消息数量。
after	int	发送时间晚于 sentTime 的消息数量。
callback	ResultCallback<List<Message>>	获取历史消息的回调，返回的历史消息按照时间顺序从新到旧排列。

通过消息 UID 获取消息

> 提示
> - SDK 从 5.2.5.2 版本开始支持通过 UID 批量获取消息的接口。仅在 ChannelClient 类中提供。支持的会话类型包括单聊、群聊、聊天室、超级群。
> - 如果 SDK 版本低于 5.2.5.2，可以使用 RongCoreClient 类中的 getMessageByUId 方法，该方法一次仅支持传入一个 UID。

消息的 UID 是由融云服务端生成的全局唯一 ID。消息存入本地数据库后，App 可能需要再次提取特定消息。例如，您的用户希望收藏聊天记录中的部分消息，App 可以先记录消息的 UID，在需要展示时调用 getBatchLocalMessages，传入收藏消息的 UID，从本地数据库中提取消息。

```
ChannelClient.getInstance().getBatchLocalMessages(conversationType, targetId, channelId, messageUIDs, callback);
```

只要持有消息 UID（messageUIDs），并且本地数据库中已存有消息，即可以使用该方法从本地数据库提取消息。单次仅可从一个会话（targetId）或超级群频道（channelId）中提取消息。

在会话类型为**超级群**或**聊天室**时，请注意以下情况：

- 超级群会话默认只同步会话最后一条消息。如果您直接调用该方法（例如您传入了通过融云服务端回调**全量消息路由**获取的 UID），可能 SDK 无法在本地找到对应消息。建议先调用 getBatchRemoteUltraGroupMessages 从服务端获取消息。
- 聊天室会话自动在用户退出清空本地消息。如果用户退出聊天室后再调用该接口，则无法取得本地消息。

参数	类型	说明
conversationType	ConversationType	会话类型，支持单聊、群聊、聊天室、超级群。
targetId	String	会话 ID
channelId	String	超级群的频道 ID。非超级群会话类型时，传入 null。
messageUIDs	List<String>	消息的 UID，即由融云服务端生成的全局唯一 ID。必须传入有效的 UID，最多支持 20 条。
callback	IRongCoreCallback.IGetMessagesByUIDsCallback	获取历史消息的回调。获取成功时会返回消息列表，和提取失败的消息的 UID 列表。

```
Conversation.ConversationType conversationType = Conversation.ConversationType.PRIVATE;
String targetId = "会话 Id";
String sampleMessageUID = "C3GC-8VAA-LJQ4-TPPM";
List<String> messageUIDs = new ArrayList<>();
messageUIDs.add(sampleMessageUID);

ChannelClient.getInstance().getBatchLocalMessages(conversationType, targetId, null, messageUIDs,
new IRongCoreCallback.IGetMessagesByUIDsCallback() {
/**
* 成功时回调
* @param messageList 获取的消息列表
* @param mismatchList 失败的消息 UID 列表
*/
@Override
public void onSuccess(List<Message> messages, List<String> mismatchList) {

}

/**
* 错误时回调。
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode errorCode) {

}
});
```

获取远端历史消息

使用 getRemoteHistoryMessages 方法可直接查询指定会话存储在**单群聊消息云端存储**中的历史消息。

> 提示
>
> 用户是否可以获取在加入群组之前的群聊历史消息取决于 App 在控制台的设置。您可以在控制台的免费基础功能 页面，启用**新用户获取加入群组前历史消息**。启用此选项后，新入群用户可以获取在他们加入群组之前发送的所有群聊消息。如不启用，新入群用户只能看到他们入群后的群聊消息。

获取会话的远端历史消息

SDK 按照指定条件直接查询并获取**单群聊消息云端存储**中的满足查询条件的历史消息。查询结果与本地数据库对比，排除重复的消息后，返回消息对象列表。返回的消息列表中的消息按发送时间从新到旧排列。

因为默认该接口返回的消息会跟本地消息排重后返回，建议先使用 getHistoryMessages，在本地数据库消息全部获取完之后，再获取远端历史消息。否则可能会获取不到指定的部分或全部消息。

```
RongIMClient.getInstance().getRemoteHistoryMessages(conversationType, targetId, dateTime, count,callback);
```

count 参数表示返回列表中应包含多少消息。dateTime 参数用于控制分页的边界。每次调用 getRemoteHistoryMessages 方法时，SDK 会以 dateTime 为界，继续在下一页返回指定数量的消息。如果需要获取会话中最新的 count 条消息，可以将 dateTime 设置为 0。

建议获取返回结果中最早一条消息的 sentTime，并在下一次调用时作为 dateTime 的值传入，以便遍历整个会话的消息历史记录。

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 Id
dateTime	long	时间戳，获取发送时间早于 dateTime 的历史消息。传 0 表示获取最新 count 条消息。
count	int	要获取的消息数量。如果 SDK < 5.4.1，范围为 [2-20]；如果 SDK ≥ 5.4.1，范围为 [2-100]。
callback	ResultCallback<List<Message>>	获取历史消息的回调，按照时间顺序从新到旧排列。

```
Conversation.ConversationType conversationType = Conversation.ConversationType.PRIVATE;
String targetId = "会话 Id";
long dateTime = 0;
int count = 20;

RongIMClient.getInstance().getRemoteHistoryMessages(conversationType, targetId, dateTime, count,
new RongIMClient.ResultCallback<List<Message>>() {
/**
* 成功时回调
* @param messages 获取的消息列表
*/
@Override
public void onSuccess(List<Message> messages) {

}

/**
* 错误时回调。
* @param e 错误码
*/
@Override
public void onError(RongIMClient.ErrorCode e) {

}
});
```

自定义获取会话的远端历史消息

通过 RemoteHistoryMsgOption 可以自定义 getRemoteHistoryMessages 方法获取远端历史消息的行为。SDK 会按照指定条件直接查询**单群聊消息云端存储**中的满足查询条件的历史消息。

```
RongIMClient.getInstance().getRemoteHistoryMessages(conversationType, targetId, remoteHistoryMsgOption, callback);
```

remoteHistoryMsgOption 中包含多个配置项，其中 count 与 dateTime 参数分别时获取历史消息的数量与分页查询时间戳。pullOrder 参数用于控制获取历史消息的时间方向，可选择获取早于或者晚于给定的 dateTime 的消息。includeLocalExistMessage 参数控制返回消息列表中是否需要包含本地数据库已存在的消息。

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID
remoteHistoryMsgOption	RemoteHistoryMsgOption	获取远端历史消息的配置选项。
callback	ResultCallback<List<Message>>	获取历史消息的回调。

- RemoteHistoryMsgOption **说明**：

参数	说明
dataTime	时间戳，用于控制分页查询消息的边界。默认值为 0。

参数	说明
count	要获取的消息数量。如果 SDK < 5.4.1，范围为 [2-20]；如果 SDK ≥ 5.4.1，范围为 [2-100]；默认值为 5。
pullOrder	拉取顺序。DESCEND：降序，按消息发送时间递减的顺序，获取发送时间早于 dataTime 的消息，返回的列表中的消息按发送时间从新到旧排列。ASCEND： 升序，按消息发送时间递增的顺序，获取发送时间晚于 dataTime 的消息，返回的列表中的消息按发送时间从旧到新排列。默认值为 1。
includeLocalExistMessage	是否包含本地数据库中的已有消息。true：包含，查询结果不与本地数据库排除重复，直接返回。false：不包含，查询结果先与本地数据库排除重复，只返回本地数据库中不存在的消息。默认值为 false。

RemoteHistoryMsgOption 默认按消息发送时间降序查询会话中的消息，且默认会并将查询结果与本地数据库对比，排除重复的消息后，再返回消息对象列表。在 includeLocalExistMessage 设置为 false 的情况下，建议 App 层先使用 getHistoryMessages，在本地数据库消息全部获取完之后，再获取远端历史消息，否则可能会获取不到指定的部分或全部消息。

如需按消息发送时间升序查询会话中的消息，建议获取返回结果中最新一条消息的 sentTime，并在下一次调用时作为 dateTime 的值传入，以便遍历整个会话的消息历史记录。升序查询消息一般可用于跳转到会话页面指定消息位置后需要查询更新的历史消息的场景。

```
Conversation.ConversationType conversationType= Conversation.ConversationType.PRIVATE;
String targetId="会话 Id";
RemoteHistoryMsgOption remoteHistoryMsgOption=new RemoteHistoryMsgOption();
remoteHistoryMsgOption.setDataTime(1662542712112L);//2022-09-07 17:25:12:112
remoteHistoryMsgOption.setOrder(HistoryMessageOption.PullOrder.DESCEND);
remoteHistoryMsgOption.setCount(20);

RongIMClient.getInstance().getRemoteHistoryMessages(conversationType, targetId, remoteHistoryMsgOption,
new RongIMClient.ResultCallback<List<Message>>() {
/**
* 成功时回调
* @param messages 获取的消息列表
*/
@Override
public void onSuccess(List<Message> messages) {

}

/**
* 错误时回调。
* @param e 错误码
*/
@Override
public void onError(RongIMClient.ErrorCode e) {

}
});
```

获取本地与远端历史消息

> 提示
>
> 该接口在 RongCoreClient 类中提供。注意：用户是否可以获取在加入群组之前的群聊历史消息取决于 App 在控制台的设置。您可以在控制台的免费基础功能页面，启用新用户获取加入群组前历史消息。启用此选项后，新入群用户可以获取在他们加入群组之前发送的所有群聊消息。如不启用，新入群用户只能看到他们入群后的群聊消息。

getMessages 方法与 getRemoteHistoryMessages 的区别是 getMessages 会先查询指定会话存储本地数据库的消息，当本地消息无法满足查询条件时，再查询在单群聊消息云端存储中的历史消息，以返回连续且相邻的消息对象列表。

```
RongCoreClient.getInstance().getMessages(conversationType, targetId, historyMessageOption, callback);
```

通过 historyMessageOption 可以控制 getMessages 方法获取历史消息的行为。count 参数表示返回列表中应包含多少消息。dateTime 参数用于控制分页的边界。每次调用 getRemoteHistoryMessages 方法时，SDK 会以 dateTime 为界，继续在下一页返回指定数量的消息。如果需要获取会话中最新的 count 条消息，可以将 dateTime 设置为 0。pullOrder 参数用于控制获取历史消息的时间方向，可选择获取早于或者晚于给定的 dateTime 的消息。

建议获取返回结果中最早一条消息的 sentTime，并在下一次调用时作为 dateTime 的值传入，以便遍历整个会话的消息历史记录。

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID
historyMsgOption	HistoryMessageOption	获取历史消息的配置选项。
callback	IRongCoreCallback.IGetMessageCallback<List<Message>>	获取历史消息的回调。

- HistoryMessageOption 说明：

参数	说明
dataTime	时间戳，用于控制分页查询消息的边界。默认值为 0。
count	要获取的消息数量。如果 SDK < 5.4.1，范围为 [2-20]；如果 SDK ≥ 5.4.1，范围为 [2-100]；默认值为 5。
pullOrder	拉取顺序。DESCEND：降序，按消息发送时间递减的顺序，获取发送时间早于 dataTime 的消息，返回的列表中的消息按发送时间从新到旧排列。ASCEND： 升序，按消息发送时间递增的顺序，获取发送时间晚于 dataTime 的消息，返回的列表中的消息按发送时间从旧到新排列。

HistoryMessageOption 默认按消息发送时间升序查询会话中的消息。升序查询消息一般可用于跳转到会话页面指定消息位置后需要查询更新的历史消息的场景。

如需按消息发送时间降序查询会话中的消息，建议获取返回结果中最早一条消息的 sentTime，并在下一次调用时作为 dateTime 的值传入，以便遍历整个会话的消息历史记录。

```
Conversation.ConversationType conversationType= Conversation.ConversationType.PRIVATE;
String targetId="会话 Id";
HistoryMessageOption historyMessageOption=new HistoryMessageOption();
historyMessageOption.setDataTime(1662542712112L);//2022-09-07 17:25:12:112
historyMessageOption.setOrder(HistoryMessageOption.PullOrder.ASCEND);
historyMessageOption.setCount(20);
RongCoreClient.getInstance().getMessages(conversationType, targetId, historyMessageOption, new IRongCoreCallback.IGetMessageCallback() {
@Override
public void onComplete(List<Message> list, IRongCoreEnum.CoreErrorCode coreErrorCode) {

Log.e("tag","list---"+list.size());
}
});
```

下载媒体消息文件　　下载媒体消息文件

更新时间:2024-08-30

SDK 提供多媒体文件的下载功能，支持通过媒体消息中的地址下载文件。

下载媒体消息中的媒体文件

如果消息 Message 对象中包含媒体消息内容（指 Message#getContent() 返回媒体消息内容（FileMessage，SightMessage，ImageMessage，GIFMessage，HQVoiceMessage 等），其中携带了媒体文件地址），可以使用 downloadMediaMessage 下载媒体文件。

```
RongCoreClient.getInstance().downloadMediaMessage(message, new IRongCallback.IDownloadMediaMessageCallback() {
@Override
public void onSuccess(Message message) {

}

@Override
public void onProgress(Message message, int progress) {

}

@Override
public void onError(Message message, IRongCoreEnum.CoreErrorCode code) {

}

@Override
public void onCanceled(Message message) {

}
});
```

获取当前下载的文件信息

在调用 downloadMediaMessage 下载多媒体文件的过程中，可调用 getDownloadInfo 获取下载文件总大小、存储路径等信息。该接口仅在下载过程中调用时会返回正确信息。下载完成后调用该接口会返回 null。

tag 是文件唯一识别标志，可以使用 messageId 字符串。

```
String tag = message.getMessageId();

RongCoreClient.getInstance().getDownloadInfo(tag, new IRongCoreCallback.ResultCallback<DownloadInfo>() {
@Override
public void onSuccess(DownloadInfo downloadInfo) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode code) {

}
});
```

取消下载多媒体文件

使用 cancelDownloadMediaMessage 取消下载多媒体文件，需要传入当前正在下载的 Message 对象。

```
RongCoreClient.getInstance().cancelDownloadMediaMessage(message, new IRongCoreCallback.OperationCallback() {
@Override
public void onSuccess() {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode code) {

}
});
```

暂停下载多媒体文件

暂停多媒体消息下载。

```
RongCoreClient.getInstance().pauseDownloadMediaMessage(message, new IRongCoreCallback.OperationCallback() {
@Override
public void onSuccess() {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode code) {

}
});
```

插入消息　　插入消息

更新时间:2024-08-30

SDK 支持在本地数据库中插入消息。本地插入的消息不会实际发送给服务器和对方。

> 提示
>
> 所有插入消息的 MessageTag 必须设置为 MessageTag.ISPERSISTED，否则报参数错误异常。

插入单条发送消息

在本地会话中插入一条消息，方向为发送。消息进入本地数据库，但不会发送给服务器和对方。不支持聊天室会话类型。以下示例使用 ChannelClient 下的 insertOutgoingMessage 方法。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "会话 Id";
SentStatus sentStatus = SentStatus.SENT;
TextMessage content = TextMessage.obtain("这里是消息内容");
long sentTime = System.currentTimeMillis();

ChannelClient.getInstance().insertOutgoingMessage(conversationType, targetId, "", true, sentStatus, content, sentTime, new IRongCoreCallback.ResultCallback<Message>() {

/**
* 成功回调
* @param message 插入的消息
*/
@Override
public void onSuccess(Message message) {

}

/**
* 失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode errorCode) {

}
});
```

插入本地数据库的消息如需支持消息扩展功能，必须使用 ChannelClient 下的插入方法，否则无法打开消息的可扩展属性（canIncludeExpansion）。本地插入消息时不支持设置消息扩展信息数据。App 可以选择合适的时机为消息设置扩展数据。详见消息扩展。

参数	类型	说明
type	ConversationType	会话类型。
targetId	String	会话 ID。
channelId	String	超级群频道 ID。会话类型为单聊、群聊时传空字符串即可。
canIncludeExpansion	boolean	是否支持消息扩展。true 表示可扩展；false 表示不可扩展。
sentStatus	Message.SentStatus	发送状态。
content	MessageContent	消息内容。
sentTime	long	消息的发送时间，Unix 时间戳（毫秒）。传 0 会按照本地时间插入。
callback	IRongCoreCallback.ResultCallback<Message>	回调接口。

插入单条接收消息

本地会话中插入一条消息，方向为接收。消息不会实际发送给服务器和对方。不支持聊天室会话类型。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "会话 Id";
String senderUserId = "模拟发送方的 ID";
ReceivedStatus receivedStatus = new ReceivedStatus(0x1);
TextMessage content = TextMessage.obtain("这是一条插入数据");
long sentTime = System.currentTimeMillis();

RongCoreClient.getInstance().insertIncomingMessage(conversationType, targetId, senderUserId, receivedStatus, content, sentTime, new
IRongCoreCallback.ResultCallback<Message>() {
/**
* 成功回调
* @param message 插入的消息
*/
@Override
public void onSuccess(Message message) {

}

/**
* 失败回调
* @param errorCode 错误码
*/
@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型。
targetId	String	Target ID 用于标识会话，称为目标 ID 或会话 ID。注意，因为单聊业务中始终使用对端用户 ID 作为标识本端会话的 Target ID，因此在单聊会话中插入本端接收的消息时，Target ID 始终是单聊对端用户 ID。群聊、超级群的会话 ID 分别为群组 ID、超级群 ID。详见消息介绍中关于 Target ID 的说明。
senderUserId	String	发送方的用户 ID。
receivedStatus	Message.ReceivedStatus	接收状态。
content	MessageContent	消息内容。
sentTime	long	消息的发送时间，Unix 时间戳（毫秒）。传 0 会按照本地时间插入。
callback	IRongCoreCallback.ResultCallback<Message)>	回调接口。

批量插入消息

在本地数据库批量插入消息。由于批量插入失败会导致整体插入失败，建议分批插入。该接口不支持聊天室会话类型，不支持超级群会话类型。

Message 的下列属性会被入库，其余属性会被抛弃：

- `UId`：消息全局唯一 ID，消息成功收发后会带有由服务端生成的全局唯一 ID。SDK 从 5.3.5 开始支持入库该属性，该字段一般用于迁移数据。
- `conversationType`：会话类型。
- `targetId`：会话 ID。
- `messageDirection`：消息方向。
- `senderUserId`：发送者 ID。
- `receivedStatus`：接收状态；如果消息方向为接收方且未调用 `message.getReceivedStatus().setRead()`，该条消息为未读消息。未读消息数会累加到会话的未读消息数上。
- `sentStatus`：发送状态。
- `content`：消息的内容。
- `sentTime`：消息发送的 Unix 时间戳，单位为毫秒，会影响消息排序。
- `extra`：消息的附加信息

```
RongCoreClient.getInstance().batchInsertMessage(messages, new IRongCoreCallback.ResultCallback<Message>() {
/**
* 成功回调
*/
@Override
public void onSuccess(Boolean success) {


}

/**
* 失败回调
* @param errorCode 错误码
*/
@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
messages	List<Message>	要插入的消息数组。单次操作最多插入 500 条消息，建议单次插入 10 条。 注意，message 中必须包含正确有效的 sentTime（消息发送的 Unix 时间戳，单位为毫秒），否则无法通过获取历史消息接口从数据库中获取该消息。
callback	IRongCoreCallback.ResultCallback<Boolean>	回调接口

批量插入消息提供以下重载方法，设置 enableCheck 为 true 后，SDK 会在插入时检查当前 UID 是否与数据库中 UID 重复，并移除重复项。注意，App 应自行保证插入的消息数组内部无重复的

UID 。

```
public abstract void batchInsertMessage(
final List<Message> messages,
boolean enableCheck,
final IRongCoreCallback.ResultCallback<Boolean> callback);
```

删除消息　　删除消息

更新时间:2024-08-30

针对单聊会话、群聊会话、系统会话，融云支持 App 用户通过客户端 SDK 删除自己的历史消息，支持仅从本地数据库删除消息、仅从融云服务端删除自己的历史消息、或从两处同时删除。

SDK 的删除消息操作（下表中的 API）均指从当前登录用户的历史消息记录中删除一条或一组消息，不影响会话中其他用户的历史消息记录。

功能	本地/服务端	API
仅从本地删除指定消息（消息 ID）	仅从本地删除（重载方法）	deleteMessages
仅从本地删除会话全部历史消息	仅从本地删除（重载方法）	deleteMessages
删除会话内指定消息（消息对象）	同时从本地和服务端删除消息	deleteRemoteMessages
删除会话历史消息（时间戳）	可选仅本地删除、或者同时从本地和服务端删除消息	cleanHistoryMessages
仅从服务端删除会话历史消息（时间戳）	仅从服务端删除	cleanRemoteHistoryMessages

提示

- App 用户的单聊会话、群聊会话、系统会话的消息默认仅存储在本地数据库中，仅支持从本地删除。如果 App（App Key/环境）已开通单群聊消息云端存储，该用户的消息还会保存在融云服务端（默认 6 个月），可从远端历史消息记录中删除消息。
- 针对单聊会话、群聊会话，如果通过任何接口以传入时间戳的方式删除远端消息，服务端默认不会删除对应的离线消息补偿（该机制仅会在打开多设备消息同步开关后生效）。此时如果换设备登录或卸载重装，仍会因为消息补偿机制获取到已被删除的历史消息。如需彻底删除消息补偿，请提交工单，申请开通删除服务端历史消息时同时删除多端补偿的离线消息。如果以传入消息对象的方式删除远端消息，则服务端一定会删除消息补偿中的对应消息。
- 针对单聊会话、群聊会话，如果 App 的管理员或者某普通用户希望在所有会话参与者的历史记录中彻底删除一条消息，应使用撤回消息功能。消息成功撤回后，原始消息内容会在所有用户的本地与服务端历史消息记录中删除。
- 客户端 SDK 不提供删除聊天室消息的接口。当前用户的聊天室本地消息在退出聊天室时会被自动删除。开通聊天室消息云端存储服务后，如需清除全部用户的聊天室历史消息，可使用服务端 API 清除消息。
- 客户端 SDK 提供删除超级群会话的消息的 API，详见「超级群管理」下的删除消息。

仅从本地删除指定消息（消息 ID）

App 用户可以按消息 ID 删除存储在本地数据库内的消息。待删除消息可以属于不同会话。一次最多删除 100 条消息。

如果 App 已经开通单群聊历史消息云存储服务，服务端保存的该用户的历史消息记录不受影响。如果该用户从服务端获取历史消息，可能会获取到在本地已删除的消息。

```
int[] messageIds = {12, 13};
RongCoreClient.getInstance().deleteMessages(messageIds, new IRongCoreCallback.ResultCallback<Boolean>() {
/**
* 删除消息成功回调
*/
@Override
public void onSuccess(Boolean bool) {

}
/**
* 删除消息失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

删除时需要提供待删除消息 ID 数组。

参数	类型	说明
messageIds	int[]	消息的 ID 数组。详见消息介绍中的 MessageId 属性。
callback	IRongCoreCallback.ResultCallback<Boolean>	接口回调

仅从本地删除会话全部历史消息

App 用户可能希望在设备本地清空自己的单聊、群聊或系统会话的历史记录，可以删除指定会话保存在本地数据库中的全部消息。

如果 App 已经开通单群聊历史消息云存储服务，服务端保存的该用户的历史消息记录不受影响。如果该用户从服务端获取历史消息，可能会获取到在本地已删除的消息。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "会话 Id";

RongCoreClient.getInstance().deleteMessages(conversationType, targetId, new IRongCoreCallback.ResultCallback<Boolean>() {
/**
* 删除消息成功回调
*/
@Override
public void onSuccess(Boolean bool) {

}
/**
* 删除消息失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

该接口一次仅允许删除指定的单个会话的消息。会话由会话类型和会话 ID 参数指定。

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID
callback	IRongCoreCallback.ResultCallback<Boolean>	接口回调

删除会话内指定消息（消息对象）

> 提示
>
> 如果 App 已经开通单群聊历史消息云存储服务，您可以调用以下接口删除 App 用户在融云服务端的历史消息记录。该接口同时从本地和服务端删除消息。

如果 App 用户希望从自己的单聊、群聊或系统会话的历史记录中彻底删除一条消息，可以使用 deleteRemoteMessages 从本地和服务端同时删除消息。删除成功后，该用户无法从本地数据库获取消息。如果从服务端获取历史消息，也无法获取到已删除的消息。

该接口允许一次删除指定的单个会话内的一条或一组消息。请确保所提供的消息均属于同一会话（由会话类型和会话 ID 指定）。一次最多删除 100 条消息。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "会话 Id";
Message[] messages = {message1, message2};

RongCoreClient.getInstance().deleteRemoteMessages(conversationType, targetId, messages, new IRongCoreCallback.OperationCallback() {
/**
* 删除消息成功回调
*/
@Override
public void onSuccess() {

}
/**
* 删除消息失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型。不支持聊天室。
targetId	String	会话 ID
messages	Message[]	要删除的消息对象 Message 数组。请确保所提供的消息均属于同一会话。
callback	IRongCoreCallback.ResultCallback<Boolean>	接口回调

删除会话历史消息（时间戳）

> 提示
>
> 该如果 App 已经开通单群聊历史消息云存储服务，您可以调用以下接口并通过参数指定需要同时删除 App 用户在融云服务端的历史消息记录。

如果 App 用户希望从自己的单聊、群聊或系统会话的历史记录中删除一段历史消息记录，可以使用 cleanHistoryMessages 删除会话中早于某个时间点（recordTime）的消息。cleanRemote 参数控制是否同时删除服务端对应的历史消息。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "会话 Id";
long recordTime = 0;
Boolean cleanRemote = true; // 同时从服务端删除对应的消息历史记录

RongCoreClient.getInstance().cleanHistoryMessages(conversationType, targetId, recordTime, cleanRemote,
new IRongCoreCallback.OperationCallback() {
/**
* 删除消息成功回调
*/
@Override
public void onSuccess() {

}
/**
* 删除消息失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

如果 cleanRemote 参数设置为 true，表示需要同时删除服务端对应消息，该接口会从本地和服务端同时删除早于 （recordTime）的消息。服务端消息删除后，该用户无法再从服务端获取到已删除的消息。该接口一次仅允许删除指定的单个会话的消息，会话由会话类型和会话 ID 参数指定。

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID
recordTime	long	时间戳。默认删除小于等于 recordTime 的消息。如果传 0，则删除所有消息。
cleanRemote	boolean	是否删除服务器端消息。true：同时删除服务端对应消息。false：不删除服务端对应消息。
callback	OperationCallback	接口回调

仅从服务端删除会话历史消息（时间戳）

> 提示
>
> - 如果 App 已经开通单群聊历史消息云存储服务，您可以调用以下接口删除 App 用户在融云服务端的历史消息记录。该接口仅从服务端删除消息。
> - 使用融云即时通讯服务端 API 也可以直接删除服务端消息。具体操作请参阅服务端文档消息清除。

App 用户可以仅删除指定会话保存在服务端的历史消息。该接口提供时间戳参数（recordTime），支持删除早于指定时间的消息。如果 recordTime 设置为 0 则删除该会话保存在服务端的全部历史消息。

```
ConversationType conversationType = ConversationType.PRIVATE
String targetId = "会话 Id";
long recordTime = 0;

RongCoreClient.getInstance().cleanRemoteHistoryMessages(conversationType, targetId, recordTime, new IRongCoreCallback.OperationCallback() {
/**
* 删除消息成功回调
*/
@Override
public void onSuccess() {

}
/**
* 删除消息失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID
recordTime	long	时间戳。默认删除所有发送时间小于等于该时间戳的消息。传 0 表示清除会话中所有消息。
callback	OperationCallback	接口回调

撤回消息　　撤回消息

更新时间:2024-08-30

您的用户通过 App 成功发送了一条消息之后，可能发现消息内容错误等情况，希望将消息撤回，同时从接收者的消息记录中移除该消息。您可以使用撤回消息功能实现该需求。

recallMessage 会替换聊天记录中的原始消息为一条 objectName 为 RC:RcNtf 的撤回通知消息（RecallNotificationMessage ）。同时消息接收方可通过 OnRecallMessageListener 监听器获取服务端通知，在收到对方已撤回通知时进行相应操作并刷新界面。

撤回通知消息的结构说明可参见服务端文档通知类消息格式 。

默认情况下，融云对撤回消息的操作者不作限制。如需限制，可考虑以下方案：

- App 客户端自行限制撤回消息的操作者。例如，不允许 App 业务中的普通用户撤回他人发送的消息，允许 App 业务中的管理员角色撤回他人发送的消息。
- 如需避免用户撤回非本人发送的消息，可以提交工单 申请打开**IMLib SDK 只允许撤回自己发送的消息**。从融云服务端进行限制，禁止用户撤回非本人发送的消息。

IMLib 对于撤回消息并没有做时间限制，现在主流的社交软件都会进行撤回时间限制，建议开发者自行做撤回时间限制。

撤回消息

撤回指定消息，只有已发送成功的消息可被撤回。撤回的结果通过 callback 回调。onSuccess 中会返回替换后撤回提示小灰条消息（RecallNotificationMessage），您可以根据需要在界面展示。

```
RongIMClient.getInstance().recallMessage(message, pushContent, callback);
```

参数	类型	说明
message	Message	待撤回的消息对象
pushContent	String	Push 消息时，在通知栏里会显示这个字段。 如果发送的是自定义消息，该字段必须填写，否则无法收到 push 消息。sdk 中默认的消息类型，例如 RC:TxtMsg, RC:VcMsg, RC:ImgMsg，则不需要填写，默认已经指定
callback	IRongCallback.ResultCallback<RecallNotificationMessage>	接口回调

```
Message message = Message.obtain("123", ConversationType.GROUP, "12345");

RongIMClient.getInstance().recallMessage(message, " ", new RongIMClient.ResultCallback<RecallNotificationMessage>() {
/**
* 成功回调
*/
@Override
public void onSuccess(RecallNotificationMessage recallNotificationMessage) {

}

/**
* 失败回调
* @param errorCode 错误码
*/
@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

如果需要通过消息 ID （数据库索引唯一值）或者 UID（服务器消息唯一 ID）撤回消息，可先通过以下方法获取待撤回的消息，再使用 recallMessage 撤回消息。

```
/**
* 根据消息 id 获取消息体（数据库索引唯一值）。
*
* @param messageId 消息 id。
* @param callback 回调。
*/
public abstract void getMessage(final int messageId, final ResultCallback<Message> callback);

/**
* 通过全局唯一 id 获取消息实体。
*
* @param uid 全局唯一 id（服务器消息唯一 id）。
* @param callback 获取消息的回调。
*/
public abstract void getMessageByUid(final String uid, final ResultCallback<Message> callback);
```

撤回消息时携带用户信息（userInfo）和附加信息（extra）

某些情况下，您可能希望在撤回消息的同时附加一些用户信息和其他的一些额外信息。 为了实现这个功能，您可以在调用撤回消息的 API 时，为消息内容中的 senderUserInfo 和 extra 字段设置相应的值。以下是如何在 Java 代码中实现上述功能的示例：

```
// 获取消息实例。
RongIMClient.getInstance().getMessage(messageId, new ResultCallback<Message>() {
@Override
public void onSuccess(Message message) {
// 设置用户信息和额外信息。
message.getContent().setUserInfo(new UserInfo("id", "name", null));
message.getContent().setExtra("the user's extra information");
// 执行撤回操作。
RongIMClient.getInstance().recallMessage(message, "", new ResultCallback<RecallNotificationMessage>() {
@Override
public void onSuccess(RecallNotificationMessage recallNotificationMessage) {

}

@Override
public void onError(ErrorCode e) {
// 撤回成功处理逻辑。
}
});
}

@Override
public void onError(ErrorCode e) {
// 撤回失败处理逻辑。
}
});
```

撤回操作完成后，消息的 senderUserInfo 和 extra 字段将更新为撤回时所附带的信息。

监听消息被撤回事件

消息发送方调用 recallMessage 后，消息接收方可通过 OnRecallMessageListener 监听消息被撤回的事件，并进行相应处理。

消息接收方需要使用 setOnRecallMessageListener 设置一个监听器，用于监听已接收的消息被撤回的事件。当接收到的某条消息被撤回时，会通过此监听器回调。

```
RongIMClient.setOnRecallMessageListener(listener);
```

其中 listener 参数为消息被撤回监听器（OnRecallMessageListener），提供一个 onMessageRecalled 方法，如下：

返回值	方法	
boolean	onMessageRecalled(Message message, RecallNotificationMessage recallNotificationMessage)	接收到的某条消息被撤回时，会触发此回调。

搜索消息　　搜索消息

更新时间:2024-08-30

SDK 提供了本地消息搜索功能，允许 App 用户通过关键词、用户 ID 等条件搜索指定的单个会话中的消息，支持按时间段搜索。消息搜索仅查询本地数据库中的消息，返回包含指定关键字或符合全部搜索条件的消息列表。

并非所有消息类型均支持关键字搜索：

- 内置的消息类型中文本消息（TextMessage），文件消息（FileMessage），和图文消息 `RichContentMessage` 类型默认实现了 `MessageContent#getSearchableWord` 方法。
- 自定义消息类型也可以支持关键字搜索，需要您参考文档自行实现。详见自定义消息类型。

如何实现基于关键字的全局搜索：

1. 根据关键字搜索本地存储的全部会话，获取包含关键字的会话列表。
2. 根据搜索会话返回的会话列表数据，调用搜索单个会话的方法，搜索符合条件的消息。

搜索全部会话

按关键字搜索本地存储的所有会话，获取符合条件的会话列表 SearchConversationResult。

```
String keyword = "搜索的关键字";

Conversation.ConversationTypes[] conversationTypes = {ConversationTypes.PRIVATE, ConversationTypes.GROUP};
String[] messageTypeObjectNames = {"RC:TxtMsg"};

RongIMClient.getInstance().searchConversations(keyword, conversationTypes, messageTypeObjectNames,
new RongIMClient.ResultCallback<List<SearchConversationResult>>() {

@Override
public void onSuccess(List<SearchConversationResult> searchConversationResults) {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
keyword	String	搜索的关键字
conversationTypes	Conversation.ConversationType[]	会话类型列表，包含 ConversationType。
objectNames	String[]	消息类型列表，默认仅支持内置类型 `RC:TxtMsg`（文本消息）、`RC:FileMsg`（文件消息）、`RC:ImgTextMsg`（图文消息）。
callback	RongIMClient.ResultCallback<List<SearchConversationResult>>	回调。搜索结果为 SearchConversationResult 列表。

在指定单个会话中搜索

获取包含关键词的会话列表后，可以搜索指定单个会话中符合条件的消息。

根据关键字搜索消息

在本地存储中根据关键字搜索指定会话中的消息，支持搜索指定时间点之前的历史消息记录。回调中分页返回包含指定关键字的消息列表。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";
String keyword = "搜索的关键字";
int count = 20;
long beginTime = 122323344;

RongIMClient.getInstance().searchMessages(conversationType, targetId, keyword, count, beginTime,
new RongIMClient.ResultCallback<List<Message>>() {

@Override
public void onSuccess(List<Message> messages) {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型。

参数	类型	说明
targetId	String	会话 ID。
keyword	String	搜索的关键字。
count	int	每页的数量，每页数量建议最多 100 条。传 0 时返回所有搜索到的消息。
beginTime	long	查询记录的起始时间。传 0 时从最新消息开始搜索。非 0 时从该时间往前搜索。
callback	RongIMClient.ResultCallback<List<Message>>	回调。搜索结果为 Message 列表。

根据关键字搜索指定时间段的消息

> 提示
>
> RongIMClient 中不提供该方法。请使用 RongCoreClient。

SDK 支持将关键字搜索的范围限制在指定时间段内。回调中分页返回包含指定关键字和时间段要求的消息列表。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";
String keyword = "123";
long startTime = 0;
long endTime = 1585815113;
int offset = 0;
int limit = 80;

RongCoreClient.getInstance().searchMessages(conversationType, targetId, keyword, startTime, endTime, offset, limit,
new IRongCoreCallback.ResultCallback<List<Message>>(){

@Override
public void onSuccess(List<Messag> messages) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {

}
});
```

limit 参数控制返回的搜索结果数量，取值范围为 [1-100]。超过 100 默认使用最大值 100。

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 Id
keyword	String	搜索的关键字
startTime	long	开始时间
endTime	long	结束时间
offset	int	偏移量
limit	int	返回的搜索结果数量，limit 需大于 0。最大值为 100。超过 100 时默认返回 100 条。
callback	IRongCallback.ResultCallback<List<Message>>	回调。搜索结果为 Message 列表。

根据用户 ID 搜索消息

在本地存储中根据搜索来自指定用户 ID 的消息，支持搜索指定时间点之前的历史消息记录。回调中分页返回包含符合条件的消息列表。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";
String userId = "查询的用户 Id";
int count = 20;
long beginTime = 1585815113;

RongIMClient.getInstance().searchMessagesByUser(conversationType, targetId, userId, count, beginTime,
new RongIMClient.ResultCallback<Message>() {

@Override
public void onSuccess(List<Messag> messages) {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID。
userId	String	要查询的用户 ID。
count	int	返回的搜索结果数量。最大值为 100。超过 100 时默认返回 100 条。
beginTime	long	查询记录的起始时间。传 0 时从最新消息开始搜索。非 0 时从该时间往前搜索。

参数	类型	说明
callback	IRongCallback.ResultCallback<List<Message>>	回调。搜索结果为 Message 列表。

在指定会话中的指定消息类型中搜索

您可以在指定会话中，按关键字搜索指定消息类型的历史消息，搜索结果将分页展示。

参数	类型	说明
conversationIdentifier	ConversationIdentifier	会话标识，用于指定要搜索的会话类型和会话ID的组合。
keyword	String	搜索的关键字，不能为空。
objectNameList	String[]	消息类型数组，支持传入多个。例如文本消息为 "RC:TxtMsg"。
limit	int	最大的搜索数量，最大为 100，超过则使用 100。
startTime	long	搜索起始时间点（毫秒），传 0 表示从最新消息开始搜索。
resultCallback	IRongCoreCallback.ResultCallback<List>	搜索结果的回调接口。成功时返回消息列表，失败时返回错误信息。

示例代码

```
RongCoreClient coreClient = RongCoreClient.getInstance();

// 准备搜索参数。
ConversationIdentifier conversationIdentifier = ...; // 填充会话标识。
String keyword = "match"; // 定义搜索的关键字。
String[] objectNameList = {
"RC:TxtMsg", // 文本消息类型
"RC:ImgTextMsg", // 图文消息类型
"RC:FileMsg", // 文件消息类型
"RC:ReferenceMsg" // 引用消息类型
};
int limit = 50; // 定义最大搜索数量上限，这里设置为 50 条。
long startTime = 0L; // 定义搜索起始时间点，0 表示从最新消息开始搜索。

// 调用搜索消息的方法
coreClient.searchMessages(
conversationIdentifier,
keyword,
objectNameList,
limit,
startTime,
new IRongCoreCallback.ResultCallback<List<Message>>() {
@Override
public void onSuccess(List<Message> messages) {
// 搜索成功时的回调，messages 包含了匹配搜索条件的消息列表。
// 在这里处理搜索到的消息，例如更新 UI 显示结果等。
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode errorCode) {
// 搜索失败时的回调，errorCode 提供了错误的原因。
// 在这里处理错误情况，例如给用户显示错误信息等。
}
}
);
```

单群聊已读回执　　单群聊已读回执

更新时间:2024-08-30

已读回执功能可以让发送方知道消息是否被对方已读。

- 在单聊场景下，当接收方已读消息后，可以主动发送已读回执给发送方，发送方通过监听已读回执消息来获得已读通知。
- 在群聊场景下，当一条消息发送到群组后，发送方可以在消息发送完成后主动发起已读回执请求。群组中的其他成员在读到这条消息后，可以对请求进行响应。发送方通过监听已读回执响应结果来获知哪些群成员已读了这条消息。

SDK 提供了一个已读回执监听器 ReadReceiptListener ，用于处理单聊、群聊已读回执相关的事件通知。

单聊已读回执

单聊已读回执功能基于用户在 1 对 1 会话中上次阅读位置的消息的时间戳。通过将该时间戳传递给对方，对方可获得已发送的消息的阅读进度。在单聊会话中，本端发出的消息的发送状态属性（SentStatus ）中记录了该条消息是否已被对方阅读。

发送单聊已读回执

消息接收方在阅读过某条消息后，需要主动发送已读回执给发送方。调用 sendReadReceiptMessage 方法时需要传入一个时间戳。可以传入指定消息的发送时间（message.getSentTime()），或者传入会话最后一条消息的发送时间（conversation.getSentTime()）。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "接收回执方的用户 ID ";
long timestamp = message.getSentTime();// 接收的消息的发送时间

RongCoreClient.getInstance().sendReadReceiptMessage(conversationType, targetId, timestamp, new IRongCoreCallback.ISendMessageCallback() {

@Override
public void onAttached(Message message) {

}

@Override
public void onSuccess(Message message) {

}

@Override
public void onError(Message message, IRongCoreEnum.CoreErrorCode CoreErrorCode) {

}
});
```

单聊已读回执实际是一条已读通知消息，由 SDK 内部构建并发出。回调中返回的 Message 对象的content 属性包含类型为 ReadReceiptMessage 的消息内容体对象。

接收单聊已读回执

发送方需要使用 setReadReceiptListener 设置已读回执监听器，才能在收到单聊已读回执时收到通知。SDK 收到单聊消息的已读回执后，会读取其中携带的时间戳，将本地消息数据库中该单聊会话的早于该时间戳的所有消息的 SentStatus 属性改为 READ，同时触发 ReadReceiptListener 的 onReadReceiptReceived 方法。

```
RongCoreClient.setReadReceiptListener(new IRongCoreListener.ReadReceiptListener() {

@Override
public void onReadReceiptReceived(final Message message) {
// 此处为单聊消息回执回调方法，可在这做回执消息处理
Long timestamp = message.getContent().getLastMessageSendTime();
}
@Override
public void onMessageReceiptRequest(Conversation.ConversationType type, String targetId, String messageUId) {
// 此处为群聊，消息回执请求消息回调方法
}

@Override
public void onMessageReceiptResponse(Conversation.ConversationType type, String targetId, String messageUId, HashMap<String, Long> respondUserIdList) {
// 此处为群聊，消息回执响应消息回调方法
}
});
```

应用程序可以从 onReadReceiptReceived 方法中获取 Message 对象，并从其 content 属性中的消息内容体对象 ReadReceiptMessage 中获取其携带的时间戳。在 UI 展示本端已发送的消息时，可以将早于该时间戳的消息均展示为对方已读。

群聊已读回执

群聊已读回执功能基于消息的全局唯一 ID（UID）。通过在群聊会话中传递消息的 UID，消息发送者可获得一条指定消息的阅读进度和已读用户列表。在群聊会话中，本端发出的消息的 ReadReceiptInfo 属性中记录了该条消息的群组已读回执数据。

群聊已读回执的工作流程如下：

1. 群成员 Alice 往群组发送了一条消息。消息将到达即时通讯服务端，并由即时通讯服务端发送到目标群组中。
2. Alice 希望得知已阅读该条消息的群成员列表。作为该条消息的发送者，她可以发起群组消息已读回执请求。
3. 群组中其他成员监听到 Alice 发起的群消息已读回执请求。
4. 群成员 Bob 阅读了 Alice 发送的群消息，于是在群组中响应 Alice 的已读回执请求。
5. 作为消息发送者，Alice 可以通过监听接收该条消息的已读回执响应。收到 Bob 在群组中发送的响应后，即可调整消息已读数。

发起群聊已读回执请求

发送者向群组中发送消息后，必须先主动发起已读回执请求，获取群成员的主动响应，才能获取该条消息的阅读状态。在一个群组会话中，只有消息发送者可以发起群聊已读回执请求。

消息发送完成后，消息发送者使用 sendReadReceiptRequest 方法，传入已发送的 Message 对象，对这条消息发起已读回执请求，SDK 内部会构建并发出一条群聊已读回执请求消息。

```
RongCoreClient.getInstance().sendReadReceiptRequest(message, new IRongCoreCallback.OperationCallback(){
@Override
public void onSuccess() {
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode errorCode) {
}
});
```

接收群聊已读回执请求

群组成员均需要使用 setReadReceiptListener 设置已读回执监听器，才能接收到群组内的已读回执请求。在任意群成员发起已读回执请求后，其他群成员会通过 ReadReceiptListener 的 onMessageReceiptRequest 回调收到已读回执请求。

```
@Override
public void onMessageReceiptRequest(Conversation.ConversationType type, String targetId, String messageUId) {
// 收到群聊消息已读回执请求
}
```

onMessageReceiptRequest 方法中会返回消息的 UID，您可以通过 getMessageByUid 方法获取消息对象，用于后续响应已读回执请求。如果需要批量获取消息，可以使用 ChannelClient 下的 getBatchLocalMessages 方法。

响应群聊已读回执请求

应用程序可以根据用户的阅读进度，对已读回执请求进行响应。如果一次接收到多个请求，可以批量响应已读回执请求，但一次只能批量响应同一个会话中的已读回执请求。

使用 sendReadReceiptResponse 方法，传入会话类型、群组 ID 和 Message 对象列表，发起已读回执响应。注意会话类型一定是 ConversationType.GROUP。

```
Conversation.ConversationType conversationType = ConversationType.GROUP;
String targetId = "群 ID";
List<Message> messageList = new ArrayList<>();
messageList.add(message1);

RongCoreClient.getInstance().sendReadReceiptResponse(conversationType, targetId, messageList, new IRongCoreCallback.OperationCallback() {

public void onSuccess() {
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {
}
});
```

接收群聊已读回执响应

群组内有用户响应成功后，请求发起方会通过 ReadReceiptListener 的 onMessageReceiptResponse 收到通知及响应结果。注意只有请求发起者可以收到响应。

```
@Override
public void onMessageReceiptResponse(Conversation.ConversationType type, String targetId, String messageUId, HashMap<String, Long> respondUserIdList) {
// 收到对群聊已读回执请求的响应

}
```

应用程序可以从 onMessageReceiptResponse 方法中获取消息的 UID 和所有已响应的用户列表。在 UI 展示本端发送的该条消息时，可以展示已读人数和具体的已读用户列表。SDK 会在消息的（ReadReceiptInfo）属性中存储该条群聊消息的已读回执数据。

消息扩展　　消息扩展

更新时间:2024-08-30

消息扩展功能可为消息对象（Message）增加基于 Key/Value 的状态标识。消息的扩展信息可在发送前、后设置或更新，可用于实现消息评论、礼物领取、订单状态变化等业务需求。

一条消息是否可携带或可设置扩展信息，由发送消息时 Message 的可扩展（canIncludeExpansion）属性决定，该属性必须在发送前设置，发送后无法修改。单条消息单次最多可设置 20 个扩展信息 KV 对，总计不可超过 300 个扩展信息 KV 对。在并发情况下如出现设置超过 300 个的情况，超出部分会被丢弃。

为 Message 消息对象添加的 Key、Value 扩展信息会被存储。如已开通历史消息云存储功能，从服务端获取的历史消息也会携带已设置的扩展信息。

> 提示
>
> - 消息扩展仅支持单聊、群聊、超级群会话类型。不支持聊天室和系统会话。
> - 4.x SDK 从 4.0.3 版本开始支持消息扩展功能。

实现思路

以**订单状态变化**为例，可通过消息扩展改变消息显示状态。以订单确认为例：

1. 当用户购买指定产品下单后，商家需要向用户发送订单确认信息。可在发送消息时，将消息对象中的 `canIncludeExpansion` 属性设置为可扩展，同时设置用于标识订单状态的 Key 和 Value。例如，在用户未确认前，可用一对 Key/Value 表示该订单状态为「未确认」。
2. 用户点击确认（或其他确认操作）该订单消息后，订单消息状态需要变更为「已确认」。此时，可通过 `updateMessageExpansion` 更新此条消息的扩展信息，标识为已确认状态，同时更改本地显示的消息样式。
3. 发送方通过消息扩展状态监听，获取指定消息的状态变化，根据最新扩展信息显示最新的订单状态。

消息评论、礼物领取可参照以上实现思路：

- **礼物领取**：可通过消息扩展改变消息显示状态实现。例如，向用户发送礼物，默认为未领取状态，用户点击后可设置消息扩展为已领取状态。
- **消息评论**：可通过设置原始消息扩展信息的方式添加评论信息。

打开消息的可扩展属性（仅发送前）

构建新消息后，调用 Message 的 `setCanIncludeExpansion()` 方法打开或关闭某条消息的可扩展属性。必须在发送消息前设置该属性。

```
TextMessage textMessage = TextMessage.obtain("content");
Message message = Message.obtain("userId", Conversation.ConversationType.PRIVATE, textMessage);
message.setCanIncludeExpansion(true);
```

参数	类型	说明
canIncludeExpansion	boolean	该条消息是否可扩展。`true`：该消息允许设置扩展信息。`false`：该消息不允许设置扩展信息。消息发送之后不允许再更改该开关状态。

如果 App 先在本地插入消息，再发送本地已插入的消息（例如 App 业务要求在发送前审核消息内容），则必须在插入消息时设置此属性（详见插入消息）。本地插入成功后返回的消息不支持再通过 `setCanIncludeExpansion` 修改可扩展属性开关的状态。

设置消息扩展数据（仅发送前）

如果消息已打开可扩展属性，可调用 Message 的 `setExpansion` 方法设置扩展数据。该接口仅可在消息发送前调用。

```
/** 为消息设置扩展数据 */
HashMap expansionKey = new HashMap();
expansionKey.put("key", "value");
message.setExpansion(expansionKey);

/** 发送消息 */
RongIMClient.getInstance().sendMessage(message, null, null, new IRongCallback.ISendMessageCallback() {
@Override
public void onAttached(io.rong.imlib.model.Message message) {

}

@Override
public void onSuccess(io.rong.imlib.model.Message message) {

}

@Override
public void onError(io.rong.imlib.model.Message message, RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
expansionMap	HashMap<String, String>	消息扩展信息。单次最大设置扩展信息键值对 20 对。单条消息可设置最多 300 个扩展信息。 • Key 支持大小写英文字母、数字、 特殊字符 `+ = - _` 的组合方式，不支持汉字。最大 32 个字符。 • （SDK < 5.2.0）Value 最大 64 个字符 • （SDK ≧ 5.2.0）Value 最大 4096 个字符

上例中发送的消息会携带 setExpansion 方法设置的扩展数据。消息发送成功后，SDK 会将消息及扩展数据存入本地数据库。

如果发送的是本地数据库中已存在的消息（详见插入消息），请注意：

- （SDK ≧ 5.3.4）发送成功后 SDK 会刷新本地数据库中消息的扩展数据。
- （SDK < 5.3.4）发送成功后 SDK 无法刷新本地数据库中消息的扩展数据。建议 App 先发送消息，再通过调用 `updateMessageExpansion` 更新本地与远端的扩展信息。对端可通过监听收到扩展数据更新。

更新扩展数据

调用 RongIMClient 的 updateMessageExpansion() 方法更新消息扩展信息。仅支持已打开可扩展属性的消息。更新消息扩展信息后，更新发起者应在成功回调里处理 UI 数据刷新，会话对端可通过消息扩展监听器收到更新的数据。

> 提示
> - 每次更新（或删除）消息扩展时，SDK 内部将向会话对端发送一条类型标识为 `RC:MsgExMsg` 的消息扩展信令消息。因此，频繁更新消息扩展会导致产生大量消息。
> - 每个终端在设置扩展信息时，如未达到上限都可以进行设置；在并发情况下，会出现设置超过 300 的情况，超出部分会被丢弃。

```
RongIMClient.getInstance()
.updateMessageExpansion(
expansionMap,
messageUId,
new RongIMClient.OperationCallback() {
@Override
public void onSuccess() {
// 更新发起者在这里处理更新扩展后的 UI 数据刷新

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {
Toast.makeText(
getApplicationContext(),
"设置失败，ErrorCode : " + errorCode.getValue(),
Toast.LENGTH_LONG)
.show();
}
});
```

参数	类型	说明
expansion	Map	要更新的消息扩展信息键值对，类型是 HashMap。 • Key 支持大小写英文字母、数字、 特殊字符 `+ = - _` 的组合方式，不支持汉字。最大 32 个字符。 • （SDK < 5.2.0）Value 最大 64 个字符 • （SDK ≧ 5.2.0）Value 最大 4096 个字符
messageUId	String	消息唯一 Id，通过 Message 对象的 getUId()方法获取。
callback	OperationCallback	更新扩展消息回调

删除扩展数据

消息发送后调用 RongIMClient 的 removeMessageExpansion 方法删除消息扩展信息中特定的键值对。仅支持已打开可扩展属性的消息。删除消息扩展信息后，发起者应在成功回调里处理删除后的数据刷新，会话对端可通过消息扩展监听器收到通知。

```
RongIMClient.getInstance().removeMessageExpansion(keyArray, messageUId, callback);
```

参数	类型	说明
keyArray	List	消息扩展信息中待删除的 key 的列表，类型是 ArrayList。
messageUId	String	消息唯一 Id，通过 Message 对象的 getUId() 方法获取。
callback	OperationCallback	删除扩展消息的回调

监听消息扩展数据变更

消息扩展变更的发起方调用 API 更新、删除扩展数据后，SDK 内部将向会话对端发送一条类型标识为 RC:MsgExMsg 的消息扩展信令消息。MessageExpansionListener 监听器会在 SDK 接收该信令消息后触发相应的回调方法。

调用 RongIMClient 的 setMessageExpansionListener 方法设置消息扩展监听器。

```
RongIMClient.getInstance().setMessageExpansionListener(listener);
```

参数	类型	说明
listener	MessageExpansionListener	消息扩展监听器。详见下方 `MessageExpansionListener` 说明。

- `MessageExpansionListener` 说明

返回值	方法	触发时机
void	onMessageExpansionUpdate()	消息扩展信息更改时
void	onMessageExpansionRemove()	消息扩展信息删除时

- onMessageExpansionUpdate(Map<String, String> expansion, Message message)

  触发时机：消息扩展信息更改时

参数	类型	说明
expansion	Map<String, String>	消息扩展信息中被更新的键值对。只包含更新的键值对，不是全部数据。如果想获取全部的键值对，请使用 Message 对象的 expansion 属性。
message	Message	消息对象

- onMessageExpansionRemove(List<String> keyArray, Message message)

  触发时机：消息扩展信息删除时

参数	类型	说明
keyArray	List<String>	消息扩展信息中被删除的键值对 key 列表
message	Message	消息对象

自定义消息类型　　自定义消息类型

更新时间:2024-08-30

> 重要
>
> 本文适用于 SDK 5.6.7 及之后版本。如果您的 SDK 版本 < 5.6.7，请使用 自定义消息（旧版）。

> 提示
>
> SDK 5.6.7 版本开始，MessageContent.java 中封装了 user（用户信息），mentionedInfo（@人信息），extra（附加信息）字段的编解码及序列化方法，实现自定义消息时可直接调用父类的解析方法，无需开发者自行解析。新旧版本自定义消息可兼容互通。已使用旧版自定义消息客户，如需新增自定义消息类型，推荐使用新版自定义消息。

客户端 SDK 使用 Message 对象表示收发的消息。Message 类中封装了 MessageContent 对象，代表消息的具体内容。

消息类型的概念是通过 MessageContent 的子类实现的，默认实现了文本、图片、语音、视频、文件等基本消息类型（参见内置消息类型）。如果内置消息类型满足不了您的需求，您可以自定义消息类型。

实现自定义消息必须继承自以下抽象类中的一个：

- MessageContent：即普通类型消息内容。例如在 SDK 内置消息类型中的文本消息和位置消息。
- MediaMessageContent：即媒体类型消息。媒体类型消息内容继承自 MessageContent，并在其基础上增加了对多媒体文件的处理逻辑。在发送和接收消息时，SDK 会判断消息类型是否为多媒体类型消息，如果是多媒体类型，则会触发上传或下载多媒体文件流程。

> 提示
>
> 自定义消息的类型、消息结构需要确保多端一致，否则将出现无法互通的问题。

创建自定义消息

SDK 不负责定义和解析自定义消息的具体内容，您需要自行实现。

自定义消息类型的 @MessageTag（MessageTag）决定该消息类型的唯一标识（objectname），以及是否存储、是否展示、是否计入消息未读数等属性。

自定义消息示例代码：普通消息

以下为自定义普通消息的完整示例代码。

```
// 1. 自定义消息实现 MessageTag 注解
@MessageTag(value = "app:txtcontent", flag = MessageTag.ISCOUNTED)
public class MyTextContent extends MessageContent {
// <editor-fold desc="* 2. 自身内部变量，可以有多个">
private static final String TAG = "appTextContent";
private String content;
// </editor-fold>

// <editor-fold desc="* 3. 对外构造方法">
private MyTextContent() {}

// 快速构建消息对象方法
public static MyTextContent obtain(String content) {
MyTextContent msg = new MyTextContent();
msg.content = content;
return msg;
}
// </editor-fold>

// <editor-fold desc="* 4. 二进制 Encode & decode 编解码方法">

/**
* 将本地消息对象序列化为消息数据。
*
* @return 消息数据。
*/
@Override
public byte[] encode() {
//此处以需要携带 “用户信息” 或者 “@ 人” 信息为例
JSONObject jsonObj = super.getBaseJsonObject();
try {
// 将所有自定义消息的内容，都序列化至 json 对象中
jsonObj.put("content", this.content);
} catch (JSONException e) {
Log.e(TAG, "JSONException " + e.getMessage());
}

try {
return jsonObj.toString().getBytes("UTF-8");
} catch (UnsupportedEncodingException e) {
Log.e(TAG, "UnsupportedEncodingException ", e);
}
return null;
}
```

```
/** 创建 MyTextContent(byte[] data) 带有 byte[] 的构造方法用于解析消息内容. */
public MyTextContent(byte[] data) {
if (data == null) {
return;
}
String jsonStr = null;
try {
jsonStr = new String(data, "UTF-8");
} catch (UnsupportedEncodingException e) {
Log.e(TAG, "UnsupportedEncodingException ", e);
}
if (jsonStr == null) {
Log.e(TAG, "jsonStr is null ");
return;
}

try {
JSONObject jsonObj = new JSONObject(jsonStr);
//此处以需要携带 “用户信息” 或者 “@ 人” 信息为例
super.parseBaseJsonObject(jsonObj);
// 将所有自定义变量从收到的 json 解析并赋值
if (jsonObj.has("content")) {
content = jsonObj.optString("content");
}
} catch (JSONException e) {
Log.e(TAG, "JSONException " + e.getMessage());
}
}

// </editor-fold>

// <editor-fold desc="* 5. Parcel 的序列化方法">

@Override
public void writeToParcel(Parcel dest, int i) {
// 对消息属性进行序列化，将类的数据写入外部提供的 Parcel 中，此处以需要序列化"用户信息"，"@信息" 等为例
super.writeToBaseInfoParcel(dest);
ParcelUtils.writeToParcel(dest, content);
}
/**
* 构造函数。
*
* @param in 初始化传入的 Parcel。
*/
public MyTextContent(Parcel in) {
// 此处以需要序列化"用户信息"，"@信息" 等为例
super.readFromBaseInfoParcel(in);
setContent(ParcelUtils.readFromParcel(in));
}

public static final Creator<MyTextContent> CREATOR =
new Creator<MyTextContent>() {
public MyTextContent createFromParcel(Parcel source) {
return new MyTextContent(source);
}

public MyTextContent[] newArray(int size) {
return new MyTextContent[size];
}
};
@Override
public int describeContents() {
return 0;
}
// </editor-fold>

// <editor-fold desc="* 6. get & set 方法">
/**
* 设置文字消息的内容。
*
* @param content 文字消息的内容。
*/
public void setContent(String content) {
this.content = content;
}

public String getContent() {
return content;
}
// </editor-fold>

}
```

详解示例代码

以下讲解自定义普通消息的示例代码。创建一个 MessageContent 的子类。

1. 自定义消息类型，必须使用 @MessageTag 添加消息注解。

   @MessageTag 中需要指定以下参数：

参数	类型	描述
value	String	（必要参数）消息类型唯一标识，例如 app:txtcontent。为尽量减少对消息体积的影响，建议控制在 16 字符以内。**注意**，请不要以 RC: 开头（RC: 为官方保留前缀），否则会和融云内置消息冲突。
flag	int	（必要参数）消息的存储与计数属性。传入值详见下方 **Flag 说明**。**注意**，客户端与服务端的存储行为均会受该属性的影响。

参数	类型	描述
messageHandler	Class<? extends MessageHandler>	（可选参数）默认使用 SDK 默认的 `messageHandler`。在自定义媒体消息类型的情况下，如果 `encode` 后的大小超过 128 KB，您需要使用自定义的 `messageHandler`。

`flag` 参数影响客户端与服务端的存储等行为，具体说明如下：

flag 取值	适用场景
`MessageTag.NONE`	此类消息不会保存到客户端本地消息数据库，也不会记录未读数。支持离线消息机制。一般用作命令消息等不需要展示的消息。例如，运营平台向终端发送的指令消息，通知端上执行一个动作。
`MessageTag.ISPERSISTED`	此类消息会保存到客户端本地消息数据库，但是不记录未读数。支持离线消息机制，且存入服务端历史消息。常用于小灰条类型的消息，需要 UI 展示，但不需要增加未读数。
`MessageTag.ISCOUNTED`	此类消息会保存到客户端本地消息数据库，并且增加未读数。支持离线消息机制，且存入服务端历史消息。如文本，图片等消息均为此类。
`MessageTag.STATUS`	状态消息在客户端与服务端均不会存储，也不会记录未读数，仅用于传递即时状态的消息，例如发送输入状态。对方在线能收到该消息；对方不在线，服务器会直接丢弃该消息，不会进行推送。因此如果接收方不在线，则无法再收到该状态消息；卸载重装后也不会重新下发。

2. 按需增加内部变量，可以有多个。示例代码中的 `content` 字段即为自定义消息的内部变量，用于存放消息内容。

```
private String content;
```

3. 实现对外构造方法，提供给其他类调用。

```
// 快速构建消息对象方法
public static MyTextContent obtain(String content) {
MyTextContent msg = new MyTextContent();
msg.content = content;
return msg;
}
```

4. 实现二进制编解码方法，用于二进制数据和消息对象的相互转换，一共有两个方法（Encode 和 Decode）。

> 提示
>
> 父类 `MessageContent.java` 中封装了 `user`（用户信息），`mentionedInfo`（@人信息），`extra`（附加信息）字段的编解码及序列化方法。自定义消息如需携带以上信息，可参考示例代码注释调用父类方法进行解析。

1. 重写父类的 `encode` 方法，将消息对象转为 `byte[]`。

   具体流程：消息对象 => JSONObject => JSON String => byte[]

```
/**
* 将本地消息对象序列化为消息数据。
*
* @return 消息数据。
*/
@Override
public byte[] encode() {

// 调用父类接口将基础字段转为 JSONObject
JSONObject jsonObj = super.getBaseJsonObject();
try {
// 将所有自定义消息的内部变量，都序列化至 json 对象中
jsonObj.put("content", this.content);
} catch (JSONException e) {
Log.e(TAG, "JSONException " + e.getMessage());
}

try {
// 将 jsonObj 转为二进制
return jsonObj.toString().getBytes("UTF-8");
} catch (UnsupportedEncodingException e) {
Log.e(TAG, "UnsupportedEncodingException ", e);
}
return null;
}
```

2. 重写父类的 `MessageContent(byte[] data)` 构造方法，以实现对自定义消息内容的解析。

   解析流程（Decode）与编码流程完全相反，具体流程为：byte[] => JSON String => JSONObject => 消息对象

```
/** 创建 MyMyTextContent(byte[] data) 带有 byte[] 的构造方法用于解析消息内容。 */
public MyTextContent(byte[] data) {
if (data == null) {
return;
}
String jsonStr = null;
try {
jsonStr = new String(data, "UTF-8");
} catch (UnsupportedEncodingException e) {
Log.e(TAG, "UnsupportedEncodingException ", e);
}
if (jsonStr == null) {
Log.e(TAG, "jsonStr is null ");
return;
}

try {
JSONObject jsonObj = new JSONObject(jsonStr);

//需要调用父类的 parseBaseJsonObjec() 方法，将基础字段解析出来
super.parseBaseJsonObject(jsonObj);

// 在将所有自定义变量从 json 解析并赋值
if (jsonObj.has("content")) {
content = jsonObj.optString("content");
}
} catch (JSONException e) {
Log.e(TAG, "JSONException " + e.getMessage());
}
}
```

5. 实现 Parcel 的序列化方法，这一组方法用于消息对象跨进程传输，总共有四个方法。

使用 SDK 的 ParcelUtils 工具类实现 MyTextContent 的序列化与反序列化。

> 重要
>
> - 父类 `MessageContent.java` 中封装了 `user`（用户信息），`mentionedInfo`（@人信息），`extra`（附加信息）字段的编解码及序列化方法。自定义消息如需携带以上信息，可参考示例代码注释调用父类方法进行序列化。
> - 序列化中的 Parcel 的读写个数和顺序一定要一一对应。

```
/**
* 将类的数据写入外部提供的 Parcel 中。
*
* @param dest 对象被写入的 Parcel。
* @param flags 对象如何被写入的附加标志，可能是 0 或 PARCELABLE_WRITE_RETURN_VALUE。
*/
@Override
public void writeToParcel(Parcel dest, int flags) {
// 将父类变量写入
super.writeToBaseInfoParcel(dest);
// 将内部变量写入
ParcelUtils.writeToParcel(dest, content);
}

/**
* 构造函数。
*
* @param in 初始化传入的 Parcel。
*/
public MyTextContent(Parcel in) {
// 读取父类变量
super.readFromBaseInfoParcel(in);
// 读取内部变量
setContent(ParcelUtils.readFromParcel(in));
}

/**
* 描述了包含在 Parcelable 对象排列信息中的特殊对象的类型。
*
* @return 一个标志位，表明 Parcelable 对象特殊对象类型集合的排列。
*/
public int describeContents() {
return 0;
}

/** 读取接口，目的是要从 Parcel 中构造一个实现了 Parcelable 的类的实例处理。 */
public static final Creator<MyTextContent> CREATOR =
new Creator<MyTextContent>() {

@Override
public TextMessage createFromParcel(Parcel source) {
return new MyTextContent(source);
}

@Override
public TextMessage[] newArray(int size) {
return new MyTextContent[size];
}
};
```

6. 实现自定义内部变量的 Getter 和 Setter 方法。

```
/**
* 设置文字消息的内容。
*
* @param content 文字消息的内容。
*/
public void setContent(String content) {
this.content = content;
}

public String getContent() {
return content;
}
```

自定义消息示例代码：媒体消息

> 提示
>
> 除非不使用 SDK 内置上传逻辑，否则 JSON 的 localPath 属性必须有值。

```
// 1. 自定义消息实现 MessageTag 注解
@MessageTag(value = "app:mediacontent", flag = MessageTag.ISCOUNTED)
public class MyMediaMessageContent extends MediaMessageContent {
// <editor-fold desc="* 2. 自身内部变量，可以有多个，媒体类消息可以直接用 MediaMessageContent 的 localPath ">

// </editor-fold>
// <editor-fold desc="* 3. 对外构造方法">
public MyMediaMessageContent(Uri localUri) {
setLocalPath(localUri);
}

/**
* 生成 MyMediaMessageContent 对象。
*
* @param localUri 媒体文件地址。
* @return MyMediaMessageContent 对象实例。
*/
public static MyMediaMessageContent obtain(Uri localUri) {
return new MyMediaMessageContent(localUri);
}
// </editor-fold>

// <editor-fold desc="* 4. 二进制 Encode & decode 编解码方法">

@Override
public byte[] encode() {
//此处以需要携带 “用户信息” 或者 “@ 人” 信息为例
JSONObject jsonObj = super.getBaseJsonObject();

try {

if (getLocalUri() != null) {
/** 除非不使用 SDK 内置上传逻辑，否则 JSON 的 `localPath` 属性必须有值。 */
jsonObj.put("localPath", getLocalUri().toString());
}
} catch (JSONException e) {
Log.e("JSONException", e.getMessage());
}
return jsonObj.toString().getBytes();
}

public MyMediaMessageContent(byte[] data) {
String jsonStr = new String(data);

try {
JSONObject jsonObj = new JSONObject(jsonStr);
//此处以需要携带 “用户信息” 或者 “@ 人” 信息为例
super.parseBaseJsonObject(jsonObj);
if (jsonObj.has("localPath")) {
setLocalPath(Uri.parse(jsonObj.optString("localPath")));
}

} catch (JSONException e) {
Log.e("JSONException", e.getMessage());
}
}
// </editor-fold>

// <editor-fold desc="* 5. Parcel 的序列化方法">

/**
* 将类的数据写入外部提供的 Parcel 中。
*
* @param dest 对象被写入的 Parcel。
* @param flags 对象如何被写入的附加标志，可能是 0 或 PARCELABLE_WRITE_RETURN_VALUE。
*/
@Override
public void writeToParcel(Parcel dest, int flags) {
// 对消息属性进行序列化，将类的数据写入外部提供的 Parcel 中，此处以需要序列化"用户信息"，"@信息" 等为例
super.writeToBaseInfoParcel(dest);
ParcelUtils.writeToParcel(dest, getLocalPath());
```

```
}

/**
* 构造函数。
*
* @param in 初始化传入的 Parcel。
*/
public MyMediaMessageContent(Parcel in) {
// 此处以需要序列化"用户信息"，"@信息" 等为例
super.readFromBaseInfoParcel(in);
setLocalPath(ParcelUtils.readFromParcel(in, Uri.class));
}

/** 读取接口，目的是要从 Parcel 中构造一个实现了 Parcelable 的类的实例处理。 */
public static final Creator<MyMediaMessageContent> CREATOR =
new Creator<MyMediaMessageContent>() {

@Override
public MyMediaMessageContent createFromParcel(Parcel source) {
return new MyMediaMessageContent(source);
}

@Override
public MyMediaMessageContent[] newArray(int size) {
return new MyMediaMessageContent[size];
}
};

/**
* 描述了包含在 Parcelable 对象排列信息中的特殊对象的类型。
*
* @return 一个标志位，表明Parcelable对象特殊对象类型集合的排列。
*/
@Override
public int describeContents() {
return 0;
}

// </editor-fold>

// <editor-fold desc="* 6. get & set 方法">

/**
* 获取本地图片地址 (file:///) 。
*
* @return 本地图片地址 (file:///) 。
*/
public Uri getLocalUri() {
return getLocalPath();
}

/**
* 设置本地图片地址 (file:///) 。
*
* @param localUri 本地图片地址 (file:///) .
*/
public void setLocalUri(Uri localUri) {
setLocalPath(localUri);
}
// </editor-fold>
}
```

注册自定义消息

您需要在建立 IM 连接之前调用 registerMessageType 注册自定义消息类型，SDK 才能识别该类消息。否则消息将无法识别，SDK 会按照 UnknownMessage 处理。

> 提示
>
> 必须在连接之前注册自定义消息。建议在应用生命周期内调用。

以下以注册自定义消息 MyTextContent.class、MyMessage.class 为例：

```
ArrayList<Class<? extends MessageContent>> myMessages = new ArrayList<>();
myMessages.add(MyTextContent.class);
myMessages.add(MyMessage.class);
RongIMClient.registerMessageType(myMessages);
```

参数	类型	说明
messageContentClassList	List<Class<? extends MessageContent>>	自定义消息的类列表。

发送自定义消息

自定义消息类型可直接使用发送内置消息类型的方法。请注意根据当前使用的 SDK、业务、消息类型选择合适的核心类与方法：

- 如果自定义消息类型继承 `MessageContent`，请使用发送普通消息的接口发送。
- 如果自定义消息类型继承 `MediaMessageContent`，请使用发送媒体消息的接口发送。

如果自定义消息类型需要支持推送，必须在发送自定义消息时额外指定推送内容（pushContent）。推送内容在接收方收到推送时显示在通知栏中。

- 在发送消息时，可直接通过 `pushContent` 参数指定推送内容。

- 您也可以通过设置 Message 的 `MessagePushConfig` 中的 `pushContent` 及其他字段，对消息的推送进行个性化配置。优先使用 `MessagePushConfig` 中的配置。

发送消息的具体方法与配置方式，请参考以下文档：

- **App 仅集成 IMLib SDK**：发送消息（单聊、群聊、聊天室）、收发消息（超级群）
- **App 集成 IMKit SDK**：发送消息（单聊、群聊、聊天室）

> 提示
>
> - 如果融云服务端无法获取自定义消息的 `pushContent`，则无法触发消息推送。例如，在接收方在离线等情况无法收到消息推送通知。
> - 如果自定义的消息类型为**状态消息**，则无法支持推送，不需要额外指定推送内容。

自定义消息类型（旧版）

自定义消息类型（旧版）

更新时间:2024-08-30

> 重要
>
> 本文适用于 SDK < 5.6.7 版本。如果您的 SDK 版本 ≧ 5.6.7，推荐使用新版 自定义消息。

客户端 SDK 使用 Message 对象表示收发的消息。Message 类中封装了 MessageContent 对象，代表消息的具体内容。

消息类型的概念是通过 MessageContent 的子类实现的，默认实现了文本、图片、语音、视频、文件等基本消息类型（参见内置消息类型）。如果内置消息类型满足不了您的需求，您可以自定义消息类型。

实现自定义消息必须继承自以下抽象类中的一个：

- MessageContent：即普通类型消息内容。例如在 SDK 内置消息类型中的文本消息和位置消息。
- MediaMessageContent：即媒体类型消息。媒体类型消息内容继承自 MessageContent，并在其基础上增加了对多媒体文件的处理逻辑。在发送和接收消息时，SDK 会判断消息类型是否为多媒体类型消息，如果是多媒体类型，则会触发上传或下载多媒体文件流程。

> 提示
>
> 自定义消息的类型、消息结构需要确保多端一致，否则将出现无法互通的问题。

创建自定义消息

SDK 不负责定义和解析自定义消息的具体内容，您需要自行实现。

自定义消息类型的 @MessageTag（MessageTag）决定该消息类型的唯一标识（objectname），以及是否存储、是否展示、是否计入消息未读数等属性。

1. 创建一个 MessageContent 的子类。如果自定义媒体消息，需要继承 MediaMessageContent。以下示例中创建了一个 MyTextContent 类：

```
@MessageTag(value = "app:txtcontent", flag = MessageTag.ISCOUNTED)
public class MyTextContent extends MessageContent {

private static final String TAG = "MyTextContent";
// 自定义消息变量，可以有多个
private String content;

private MyTextContent() {}

/**
* 设置文字消息的内容。
*
* @param content 文字消息的内容。
*/
public void setContent(String content) {
this.content = content;
}

}
```

   继承自 MessageContent 或 MediaMessageContent 的子类都必须使用 @MessageTag 添加消息注解。以上是一个非媒体消息的示例，其中的 MessageTag 指定了消息类型的唯一标识为 app:txtcontent、需要在客户端存入数据库、且需要计入未读消息数。

   MessageTag 字段说明与详细用法参见下文如何添加消息注解。

2. 重写父类的 encode 方法，将 MyTextContent 的属性写入 JSON，转为 JSON 字符串，最后编码为字节序列（Bytes 数组）。

```
/**
* 将本地消息对象序列化为消息数据。
*
* @return 消息数据。
*/
@Override
public byte[] encode() {
JSONObject jsonObj = new JSONObject();
try {
// 消息携带用户信息时，自定义消息需添加下面代码
if (getJSONUserInfo() != null) {
jsonObj.putOpt("user", getJSONUserInfo());
}
// 用于群组聊天，消息携带 @ 人信息时，自定义消息需添加下面代码
if (getJsonMentionInfo() != null) {
jsonObj.putOpt("mentionedInfo", getJsonMentionInfo());
}
// 将所有自定义消息的内容，都序列化至 json 对象中
jsonObj.put("content", this.content);
} catch (JSONException e) {
Log.e(TAG, "JSONException " + e.getMessage());
}

try {
return jsonObj.toString().getBytes("UTF-8");
} catch (UnsupportedEncodingException e) {
Log.e(TAG, "UnsupportedEncodingException ", e);
}
return null;
}
```

3. 重写父类的 MessageContent(byte[] data) 构造方法，以实现对自定义消息内容的解析。先由字节序列（Bytes）解码成 JSON 字符串，再构造 JSON，并将内容取出赋给 MyTextContent 的属性。

```
/** 创建 MyMyTextContent(byte[] data) 带有 byte[] 的构造方法用于解析消息内容。 */
public MyTextContent(byte[] data) {
if (data == null) {
return;
}
String jsonStr = null;
try {
jsonStr = new String(data, "UTF-8");
} catch (UnsupportedEncodingException e) {
Log.e(TAG, "UnsupportedEncodingException ", e);
}
if (jsonStr == null) {
Log.e(TAG, "jsonStr is null ");
return;
}

try {
JSONObject jsonObj = new JSONObject(jsonStr);
// 消息携带用户信息时，自定义消息需添加下面代码
if (jsonObj.has("user")) {
setUserInfo(parseJsonToUserInfo(jsonObj.getJSONObject("user")));
}
// 用于群组聊天，消息携带 @ 人信息时，自定义消息需添加下面代码
if (jsonObj.has("mentionedInfo")) {
setMentionedInfo(parseJsonToMentionInfo(jsonObj.getJSONObject("mentionedInfo")));
}
// 将所有自定义变量从收到的 json 解析并赋值
if (jsonObj.has("content")) {
content = jsonObj.optString("content");
}
} catch (JSONException e) {
Log.e(TAG, "JSONException " + e.getMessage());
}
}
```

4. 使用 ParcelUtils 工具类实现 MyTextContent 的序列化与反序列化。

> 提示
>
> 序列化中的 Parcel 的读写个数和顺序一定要一一对应。

```
/**
* 描述了包含在 Parcelable 对象排列信息中的特殊对象的类型。
*
* @return 一个标志位，表明 Parcelable 对象特殊对象类型集合的排列。
*/
public int describeContents() {
return 0;
}

/**
* 将类的数据写入外部提供的 Parcel 中。
*
* @param dest 对象被写入的 Parcel。
* @param flags 对象如何被写入的附加标志，可能是 0 或 PARCELABLE_WRITE_RETURN_VALUE。
*/
@Override
public void writeToParcel(Parcel dest, int flags) {
ParcelUtils.writeToParcel(dest, getExtra());
ParcelUtils.writeToParcel(dest, content);
}

/**
* 构造函数。
*
* @param in 初始化传入的 Parcel。
*/
public MyTextContent(Parcel in) {
setExtra(ParcelUtils.readFromParcel(in));
setContent(ParcelUtils.readFromParcel(in));
}
```

您已成功创建自定义消息类型。后续步骤如下：

1. 完善消息注解 `@MessageTag`。如果需要实现对消息内容的自定义处理，您需要创建自定义的 `messageHandler`。
2. 注册自定义消息类型，SDK 才能识别并收发该类消息。

如何添加消息注解

SDK 内置消息类型默认已带有 MessageTag。如果创建自定义消息类型，必须使用 @MessageTag 添加消息注解。

@MessageTag 中需要指定以下参数：

参数	类型	描述
value	String	（必要参数）消息类型唯一标识，例如 "app:txtcontent。为尽量减少对消息体积的影响，建议控制在 16 字符以内。**注意**，请不要以 RC: 开头（RC: 为官方保留前缀），否则会和融云内置消息冲突。
flag	int	（必要参数）消息的存储与计数属性。传入值详见下方 **Flag 说明**。**注意**，客户端与服务端的存储行为均会受该属性的影响。
messageHandler	Class<? extends MessageHandler>	（可选参数）默认使用 SDK 默认的 `messageHandler`。在自定义媒体消息类型的情况下，如果 encode 后的大小超过 128 KB，您需要使用自定义的 `messageHandler`。

- `flag` **参数说明**：

存储计数属性	客户端是否存储	服务端是否存储	适用场景
MessageTag.NONE	客户端不存储、不计入未读消息数	支持离线消息[?]机制	不需要展示的消息。例如，运营平台向终端发送的指令消息。
MessageTag.ISCOUNTED	客户端存储、计入未读消息数	支持离线消息[?]机制，且存入服务端历史消息	---
MessageTag.ISPERSISTED	客户端存储，不计入未读消息数	支持离线消息[?]机制，且存入服务端历史消息	---
MessageTag.STATUS	客户端不存储，不计入未读消息数	服务端不存储	用于传递即时状态的消息，无法支持消息推送。

> 提示
>
> - `MessageTag.NONE` 一般用于需要确保收到，但不需要展示的消息，例如运营平台向终端发送的指令信息。如果消息接收方不在线，再次上线时可通过离线消息收到。
> - `MessageTag.STATUS` 用于状态消息。状态消息表示的是即时的状态，例如输入状态。因为状态消息在客户端与服务端均不会存储，如果接收方不在线，则无法再收到该状态消息。

- **代码示例**

```
@MessageTag(value = "appx:MyTextContent", flag = MessageTag.ISCOUNTED, )
class MyTextContent extend MessageContent {

}
```

自定义媒体消息类型的处理方式

MessageTag 中定义了 messageHandler，支持自定义处理消息内容，例如文件的压缩等操作。

- 如果不指定 `messageHandler`，SDK 会默认使用 `DefaultMessageHandler`。一般情况下，您不需要指定 `messageHandler`，
- 如果希望自定义处理消息内容，您需要创建 `messageHandler` 并继承 MessageHandler。在自定义媒体消息类型的情况下，如果 `encode` 后的大小超过 128 KB，您需要使用自定义的 `messageHandler`。

以下示例中自定义的 MyMediaHandler 继承了 MessageHandler。其中 MyMediaMessageContent 为自定义的媒体消息内容。

```
public class MyMediaHandler extends MessageHandler<MyMediaMessageContent> {

public MyMediaHandler(Context context) {
super(context);
}

/**
* 解码 MessageContent 到 Message 中。
*
* @param message 用于存放 MessageContent 的消息实体。
* @param content 将要被解码的 MessageContent。
*/
@Override
public void decodeMessage(Message message, MyMediaMessageContent model) {

}

/**
* 对 Message 编码。
*
* @param message 将要被编码的 Message 实体。
*/
@Override
public void encodeMessage(Message message) {

}
}
```

注册自定义消息

您需要在建立 IM 连接之前调用 registerMessageType 注册自定义消息类型，SDK 才能识别该类消息。否则消息将无法识别，SDK 会按照 UnknownMessage 处理。

> 提示
>
> 必须在连接之前注册自定义消息。建议在应用生命周期内调用。

以注册自定义消息 MyTextContent.class、MyMessage.class 为例：

```
ArrayList<Class<? extends MessageContent>> myMessages = new ArrayList<>();
myMessages.add(MyTextContent.class);
myMessages.add(MyMessage.class);
RongIMClient.registerMessageType(myMessages);
```

参数	类型	说明
messageContentClassList	List<Class<? extends MessageContent>>	自定义消息的类列表。

发送自定义消息

自定义消息类型可直接使用发送内置消息类型的方法。请注意根据当前使用的 SDK、业务、消息类型选择合适的核心类与方法：

- 如果自定义消息类型继承 `MessageContent`，请使用发送普通消息的接口发送。
- 如果自定义消息类型继承 `MediaMessageContent`，请使用发送媒体消息的接口发送。

如果自定义消息类型需要支持推送，必须在发送自定义消息时额外指定推送内容（pushContent）。推送内容在接收方收到推送时显示在通知栏中。

- 在发送消息时，可直接通过 `pushContent` 参数指定推送内容。
- 您也可以通过设置 Message 的 `MessagePushConfig` 中的 `pushContent` 及其他字段，对消息的推送进行个性化配置。优先使用 `MessagePushConfig` 中的配置。

发送消息的具体方法与配置方式，请参考以下文档：

- **App 仅集成 IMLib SDK**：发送消息（单聊、群聊、聊天室）、收发消息（超级群）
- **App 集成 IMKit SDK**：发送消息（单聊、群聊、聊天室）

> 提示
>
> - 如果融云服务端无法获取自定义消息的 `pushContent`，则无法触发消息推送。例如，在接收方在离线等情况无法收到消息推送通知。
> - 如果自定义的消息类型为**状态消息**（见如何添加消息注解），则无法支持推送，不需要额外指定推送内容。

代码示例

示例：自定义普通消息类型

```
@MessageTag(value = "app:txtcontent", flag = MessageTag.ISCOUNTED)
```

```
public class MyTextContent extends MessageContent {

private static final String TAG = "MyTextContent";
// 自定义消息变量，可以有多个
private String content;

private MyTextContent() {}

/**
* 设置文字消息的内容。
*
* @param content 文字消息的内容。
*/
public void setContent(String content) {
this.content = content;
}

/**
* 构造函数。
*
* @param in 初始化传入的 Parcel。
*/
public MyTextContent(Parcel in) {
setExtra(ParcelUtils.readFromParcel(in));
setContent(ParcelUtils.readFromParcel(in));
}

// 快速构建消息对象方法
public static MyTextContent obtain(String content) {
MyTextContent msg = new MyTextContent();
msg.content = content;
return msg;
}

/** 创建 MyTextContent(byte[] data) 带有 byte[] 的构造方法用于解析消息内容. */
public MyTextContent(byte[] data) {
if (data == null) {
return;
}
String jsonStr = null;
try {
jsonStr = new String(data, "UTF-8");
} catch (UnsupportedEncodingException e) {
Log.e(TAG, "UnsupportedEncodingException ", e);
}
if (jsonStr == null) {
Log.e(TAG, "jsonStr is null ");
return;
}

try {
JSONObject jsonObj = new JSONObject(jsonStr);
// 消息携带用户信息时，自定义消息需添加下面代码
if (jsonObj.has("user")) {
setUserInfo(parseJsonToUserInfo(jsonObj.getJSONObject("user")));
}
// 用于群组聊天，消息携带 @ 人信息时，自定义消息需添加下面代码
if (jsonObj.has("mentionedInfo")) {
setMentionedInfo(parseJsonToMentionInfo(jsonObj.getJSONObject("mentionedInfo")));
}
// 将所有自定义变量从收到的 json 解析并赋值
if (jsonObj.has("content")) {
content = jsonObj.optString("content");
}
} catch (JSONException e) {
Log.e(TAG, "JSONException " + e.getMessage());
}
}

public String getContent() {
return content;
}

/**
* 将本地消息对象序列化为消息数据。
*
* @return 消息数据。
*/
@Override
public byte[] encode() {
JSONObject jsonObj = new JSONObject();
try {
// 消息携带用户信息时，自定义消息需添加下面代码
if (getJSONUserInfo() != null) {
jsonObj.putOpt("user", getJSONUserInfo());
}
// 用于群组聊天，消息携带 @ 人信息时，自定义消息需添加下面代码
if (getJsonMentionInfo() != null) {
jsonObj.putOpt("mentionedInfo", getJsonMentionInfo());
}
// 将所有自定义消息的内容，都序列化至 json 对象中
jsonObj.put("content", this.content);
} catch (JSONException e) {
Log.e(TAG, "JSONException " + e.getMessage());
}

try {
return jsonObj.toString().getBytes("UTF-8");
} catch (UnsupportedEncodingException e) {
Log.e(TAG, "UnsupportedEncodingException ", e);
}
return null;
}
```

```
@Override
public int describeContents() {
return 0;
}

@Override
public void writeToParcel(Parcel dest, int i) {
// 对消息属性进行序列化，将类的数据写入外部提供的 Parcel 中
ParcelUtils.writeToParcel(dest, getExtra());
ParcelUtils.writeToParcel(dest, content);
}

public static final Creator<MyTextContent> CREATOR =
new Creator<MyTextContent>() {
public MyTextContent createFromParcel(Parcel source) {
return new MyTextContent(source);
}

public MyTextContent[] newArray(int size) {
return new MyTextContent[size];
}
};
}
```

示例：自定义媒体消息类型

> 提示
>
> 除非不使用 SDK 内置上传逻辑，否则 JSON 的 localPath 属性必须有值。

```
@MessageTag(value = "app:mediacontent", flag = MessageTag.ISCOUNTED)
public class MyMediaMessageContent extends MediaMessageContent {

/** 读取接口，目的是要从 Parcel 中构造一个实现了 Parcelable 的类的实例处理。 */
public static final Creator<MyMedia> CREATOR =
new Creator<MyMediaMessage>() {

@Override
public MyMediaMessageContent createFromParcel(Parcel source) {
return new MyMediaMessageContent(source);
}

@Override
public MyMediaMessageContent[] newArray(int size) {
return new MyMediaMessageContent[size];
}
};


public MyMediaMessageContent(byte[] data) {
String jsonStr = new String(data);

try {
JSONObject jsonObj = new JSONObject(jsonStr);

if (jsonObj.has("localPath")) {
setLocalPath(Uri.parse(jsonObj.optString("localPath")));
}

} catch (JSONException e) {
Log.e("JSONException", e.getMessage());
}
}


/**
* 构造函数。
*
* @param in 初始化传入的 Parcel。
*/
public MyMediaMessageContent(Parcel in) {
setLocalPath(ParcelUtils.readFromParcel(in, Uri.class));
}


public MyMediaMessageContent(Uri localUri) {
setLocalPath(localUri);
}

/**
* 生成 MyMediaMessageContent 对象。
*
* @param localUri 媒体文件地址。
* @return MyMediaMessageContent 对象实例。
*/
public static MyMediaMessageContent obtain(Uri localUri) {
return new MyMediaMessageContent(localUri);
}

@Override
public byte[] encode() {
JSONObject jsonObj = new JSONObject();

try {

if (getLocalUri() != null) {
/** 除非不使用 SDK 内置上传逻辑，否则 JSON 的 `localPath` 属性必须有值。 */
jsonObj.put("localPath", getLocalUri().toString());
}
} catch (JSONException e) {
```

```
RLog.e("JSONException", e.getMessage());
}
return jsonObj.toString().getBytes();
}

/**
* 获取本地图片地址 (file:///) 。
*
* @return 本地图片地址 (file:///) 。
*/
public Uri getLocalUri() {
return getLocalPath();
}

/**
* 设置本地图片地址 (file:///) 。
*
* @param localUri 本地图片地址 (file:///) .
*/
public void setLocalUri(Uri localUri) {
setLocalPath(localUri);
}

/**
* 描述了包含在 Parcelable 对象排列信息中的特殊对象的类型。
*
* @return 一个标志位，表明Parcelable对象特殊对象类型集合的排列。
*/
@Override
public int describeContents() {
return 0;
}

/**
* 将类的数据写入外部提供的 Parcel 中。
*
* @param dest 对象被写入的 Parcel。
* @param flags 对象如何被写入的附加标志，可能是 0 或 PARCELABLE_WRITE_RETURN_VALUE。
*/
@Override
public void writeToParcel(Parcel dest, int flags) {
ParcelUtils.writeToParcel(dest, getLocalPath());
}
}
```

管理离线消息存储配置 管理离线消息存储配置

更新时间:2024-08-30

即时通讯业务支持修改 App 级别与用户级别的离线消息配置。

> 提示
>
> 离线消息配置仅适用于单聊、群聊。聊天室、超级群因业务特性不支持离线消息，因此无离线消息配置。

了解离线消息

离线消息[7]是指当用户不在线时收到的消息。融云服务端会自动为用户保留离线期间接收的消息，默认的离线消息保留时长为 7 天。7 天内客户端如果上线，服务端会直接将离线消息发送到该接收端。如果 7 天内客户端都没有上线，服务端将抛弃掉过期的消息。

即时通讯业务下并非所有会话类型都支持离线消息：

- **支持离线消息**：单聊、群聊、系统消息
- **不支持离线消息**：聊天室、超级群

App 级别离线消息配置

如需修改 App 级别设置，请提交工单。

App 级别的离线消息配置如下：

- 单聊离线消息存储时长：默认存储 7 天。设置范围为 1 - 7 天。配置修改将影响 App 下所有单聊会话。
- 群聊离线消息存储时长：默认存储 7 天。设置范围为 1 - 7 天。配置修改将影响 App 下所有群聊会话。
- 群组离线消息存储数量：默认存储 7 天内的所有群消息。配置修改将影响 App 下所有群聊会话。

用户级别离线消息配置

> 提示
>
> 设置、获取用户的离线消息存储时长功能均要求已开通用户级别功能设置。如需开通，请提交工单。

即时通讯业务支持用户级别的离线消息配置，仅支持修改离线消息存储时长。未修改的情况下，用户的离线消息存储时长为 7 天。设置范围为 1 - 7 天。

App Key 开通**用户级别功能设置**功能后，客户端 SDK 支持修改当前登录用户的离线消息存储时长。

设置用户的离线消息存储时长

设置当前用户的离线消息存储时长，以天为单位。

```
int duration = 3;

RongIMClient.getInstance().setOfflineMessageDuration(duration, new RongIMClient.ResultCallback<Long>() {
@Override
public void onSuccess(Long aLong) {

}

@Override
public void onError(RongIMClient.ErrorCode e) {

}
});
```

参数	类型	说明
duration	int	离线消息存储时长，范围为 1 - 7 天。
callback	RongIMClient.ResultCallback<Long>	回调接口。

获取用户的离线消息存储时长

获取当前用户的离线消息存储时长，以天为单位。

```
RongIMClient.getInstance().getOfflineMessageDuration(new RongIMClient.ResultCallback<String>() {
@Override
public void onSuccess(String s) {

}

@Override
public void onError(RongIMClient.ErrorCode e) {

}
});
```

参数	类型	说明
callback	RongIMClient.ResultCallback<String>	回调接口。

会话介绍 会话介绍

更新时间:2024-08-30

会话是指融云 SDK 根据每条消息的发送方、接收方以及会话类型等信息，自动建立并维护的逻辑关系，是一种抽象概念。

会话类型

融云支持多种会话类型，以满足不同业务场景需求。客户端 SDK 通过 `ConversationType` 枚举来表示各类型会话，各枚举值代表的含义参考下表：

枚举值	会话类型
ConversationType.PRIVATE	单聊会话
ConversationType.GROUP	群组会话
ConversationType.ULTRA_GROUP	超级群会话
ConversationType.CHATROOM	聊天室会话
ConversationType.SYSTEM	系统会话

`ConversationType` 枚举中还定义了其它会话类型，目前已废弃，不再维护。

单聊会话

指两个用户一对一进行聊天，两个用户间可以是好友也可以是陌生人，融云不对用户的关系进行维护管理，会话关系由融云负责建立并保持。

单聊类型会话里的消息会保存在客户端本地数据库中。

群组会话

群组指两个以上用户一起进行聊天，群组成员信息由 App 提供并进行维系，融云只负责将消息传达给群组中的所有用户。每个群最大人数上限为 3000 人，App 内的群组数量没有限制。

群组类型会话里的消息会保存在客户端本地数据库中。

超级群会话

超级群会话指无成员上限的多人聊天服务，海量消息并发即时到达，支持消息推送服务。群组成员信息由 App 提供并维系，融云负责将消息传达给群成员。App 内超级群数量没有限制，超级群无人数上限，一个用户可加入 100 个超级群。

超级群类型会话里的消息会保存在客户端本地数据库中，更多内容请参见超级群概述。

聊天室会话

聊天室成员不设用户上限，海量消息并发即时到达，用户退出聊天室后不会再接收到任何聊天室中的消息，没有推送通知功能。会话关系由融云负责建立并保持连接，通过 SDK 相关接口，可以让用户加入或者退出聊天室。

SDK 不保存聊天室消息，在退出聊天室时会清空此聊天室所有数据，更多内容请参见聊天室概述。

系统会话

系统会话是指利用系统帐号向用户发送消息从而建立的会话关系，此类型会话可以是通过调用广播接口发送广播来建立，也可以是加好友等单条通知消息而建立的会话。

会话实体类

客户端 SDK 中封装的会话实体类是 `Conversation`，所有会话相关的信息都从该实体类中获取。

下表列出了 `Conversation` 中提供的主要属性：

属性名	类型	描述
targetId	String	会话 ID（或目标 ID），用于标识会话对端。 • 单聊时，会话 ID 直接使用对方的用户 ID。 • 在群组、聊天室、超级群中，为对应的群组、聊天室、超级群 ID。 • 系统会话中，为开发者指定的系统账号 ID。
channelId	String	消息所属会话的业务标识，仅适用于超级群。
portraitUrl	String	会话中头像的 URL。
unreadMessageCount	String	会话中未读消息数。
isTop	boolean	会话是否置顶。
conversationTitle	String	会话标题。
conversationType	ConversationType	会话类型，参考上文详细描述。
latestMessage	MessageContent	会话中在客户端本地存储的最后一条消息的消息内容。关于消息的存储属性请参考 消息介绍 中关于 MessageTag 的说明。
latestMessageId	int	会话中最后一条在客户端本地存储的消息的 ID。

属性名	类型	描述
draft	String	会话里保存的草稿信息，参考草稿详细说明。
receivedTime	long	会话中最后一条消息的接收时间。 1. 返回值为 Unix 时间戳，单位毫秒。 2. 接收时间为消息到达接收端时客户端的本地时间。
sentTime	long	会话中最后一条消息的发送时间，为 Unix 时间戳，单位毫秒。 1. 当会话里最后一条消息为发送成功或者接收到的消息时，返回该消息到达融云服务器的时间。 2. 当会话里最后一条消息为发送失败的消息时，返回此条消息的本地发送时间。 3. 当会话有草稿信息，且草稿保存时间大于最后一条消息时间时，返回草稿保存时间。
receivedStatus	Message.ReceivedStatus	会话中最后一条消息的接收状态。
sentStatus	Message.SentStatus	会话中最后一条消息的发送状态。
objectName	String	会话中最后一条消息的类型名，与消息内容体对应。预定义消息类型的 objectName 参见消息类型概述。自定义消息类型的 objectName 为您自行指定的值。
senderUserId	String	会话中最后一条消息发送者 ID。
senderUserName	String	发送者名称。该字段仅供 SDK 内部使用。
notificationStatus	ConversationNotificationStatus	会话的免打扰状态。
mentionedCount	int	本会话里自己被 @ 的消息数量。
latestMessageDirection	Message.MessageDirection	会话中最后一条消息的方向，分为发送和接收。
latestMessageExtra	String	会话中最后一条消息的附加信息。
latestMessageUId	String	会话中最后一条消息的唯一 ID。 1. 只有发送成功的消息才有唯一 ID。 2. 在同一个 Appkey 下全局唯一。
latestMessageReadReceiptInfo	ReadReceiptInfo	会话中最后一条消息的阅读回执状态，仅适用于群聊。
latestMessageConfig	MessageConfig	会话中最后一条消息的配置信息。
latestCanIncludeExpansion	boolean	会话中最后一条消息是否允许消息扩展。 1. 该属性在消息发送时确定，发送之后不能再做修改； 2. 扩展信息只支持单聊、群组和超级群，其它会话类型不能设置扩展信息。
latestExpansion	Map<String, String>	会话中最后一条消息的扩展信息。详情请参考消息扩展。
pushNotificationLevel	int	会话免打扰级别，详见免打扰功能概述。
channelType	IRongCoreEnum.UltraGroupChannelType	从 5.2.4 开始，支持超级群频道类型，包含公有频道和私有频道。
firstUnreadMsgSendTime	long	从 5.2.5 版本开始，支持会话中第一条未读消息的时间戳属性，仅对超级群生效。

获取会话　　获取会话

更新时间:2024-08-30

客户端 SDK 会根据用户收发的消息，在本地数据库中生成对应会话，并维护会话列表。应用程序可以获取本地数据库中的会话列表。

获取指定单个会话

获取某个会话的详细信息。

```
String conversationType = ConversationType.PRIVATE;
String targetId = "会话 Id";

RongCoreClient.getInstance().getConversation(conversationType, targetId, new IRongCoreCallback.ResultCallback<Conversation>() {

@Override
public void onSuccess(Conversation conversation) {
// 成功并返回会话信息
}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID
callback	ResultCallback<Conversation>	回调接口

批量获取会话信息

除了单个获取会话信息外，客户端 SDK 还支持批量获取会话的详细信息。

> **注意：**
> 客户端 SDK 从 5.8.2 版本开始支持批量获取会话信息。
> 支持的会话类型：单聊、群聊、系统。

使用 `getConversations()` 方法查询多个会话的详细信息。

```
// 假设我们有会话标识 conIden1 和 conIden2，它们代表想要查询的会话。

List<ConversationIdentifier> conversationIdentifiers = new ArrayList<>();
conversationIdentifiers.add(ConversationIdentifier.obtain(ConversationType.PRIVATE, "tId1", ""));
conversationIdentifiers.add(ConversationIdentifier.obtain(ConversationType.PRIVATE, "tId2", ""));

RongCoreClient.getInstance().getConversations(conversationIdentifiers, new IRongCoreCallback.ResultCallback<List<Conversation>>() {
@Override
public void onSuccess(List<Conversation> conversations) {
// 成功并返回会话信息
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {

}
});
```

参数	类型	说明
conversationIdentifiers	List	会话类型
callback	ResultCallback<Conversation>	回调接口

获取会话列表

会话列表是 SDK 在本地生成和维护的。如果未发生卸载重载或换设备登录，您可以获取本地设备上存储的所有历史消息生成的会话列表。

分页获取会话列表

分页获取 SDK 在本地数据库生成的会话列表。返回的会话列表按照时间倒序排列。如果返回结果含有被设置为置顶状态的会话，则置顶会话默认排在最前。

使用 getConversationListByPage 方法，时间戳（`startTime`）首次可传 `0`，后续可以使用返回的 Conversation 对象的 `sentTime` 或 `operationTime` 属性值为下一次查询的 `startTime`。推荐使用 `operationTime`，该属性仅在 5.6.8 及之后版本提供。

```
long timeStamp = 0;
int count = 10;
Conversation.ConversationType[] conversationTypes = {ConversationType.PRIVATE, ConversationType.GROUP};

RongCoreClient.getInstance().getConversationListByPage(new IRongCoreCallback.
ResultCallback<List<Conversation>>() {

@Override
public void onSuccess(List<Conversation> conversations) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode ErrorCode) {

}
},timeStamp, count, conversationTypes);
```

如果希望返回的会话列表严格按照时间倒序排列，请使用带 topPriority 参数的重载方法，并将该参数设置为 false。该重载方法仅在 5.6.9 及之后版本提供。

```
long timeStamp = 0;
int count = 10;
boolean topPriority = false;

Conversation.ConversationType[] conversationTypes = {ConversationType.PRIVATE, ConversationType.GROUP};

RongCoreClient.getInstance().getConversationListByPage(new IRongCoreCallback.
ResultCallback<List<Conversation>>() {

@Override
public void onSuccess(List<Conversation> conversations) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode ErrorCode) {

}
},timeStamp, count, topPriority, conversationTypes);
```

参数	类型	说明
callback	ResultCallback<List<Conversation>>	方法回调。
timeStamp	long	时间戳，以获取早于这个时间戳的会话列表。首次可传 0，表示从最新开始获取。后续使用真实时间戳。
count	int	取回的会话数量。建议此数值不要超过 10 个，当一次性获取的会话数过大时，会导致跨进程通信崩溃，引发获取会话列表失败及通信连接被中断。 当实际取回的会话数量小于 count 值时，表明已取完数据。
topPriority	boolean	是否优先显示置顶消息。要求 SDK 版本 ≥ 5.6.9。
conversationTypes	Array of ConversationType	选择要获取的会话类型。可设置多个会话类型。

获取未读的会话列表

> 提示
>
> SDK 从 5.3.2 版本开始提供该接口。

使用 RongCoreClient 的 getUnreadConversationList 方法获取指定类型的含有未读消息的会话列表，支持单聊、群聊、系统会话。获取到的会话列表按照时间倒序排列，置顶会话会排在最前。

```
ConversationType[] conversationTypes = {ConversationType.PRIVATE, ConversationType.GROUP, ConversationType.SYSTEM};

RongCoreClient.getInstance().getUnreadConversationList(new IRongCoreCallback.ResultCallback<List<Conversation>>() {

@Override
public void onSuccess(List<Conversation> conversations) {
// 成功并返回会话信息
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {

}
}, conversationTypes);
```

参数	类型	说明
callback	ResultCallback<List<Conversation>>	回调接口
conversationTypes	Array of ConversationType	会话类型数组

卸载重装或换设备登录后的处理方案

如果您的用户卸载重装或换设备登录，可能会发现会话列表为空，或者有部分会话丢失的错觉。

原因如下：

- 在卸载的时候会删除本地数据库，本地没有任何历史消息，导致重新安装后会话列表为空。

- 如果换设备登录，可能本地没有历史消息数据，导致会话列表为空。
- 如果您的 App Key 开启了多设备消息同步 功能，服务端会同时启用**离线消息补偿**功能。服务端会在 SDK 连接成功后自动同步当天 0 点后的消息，客户端 SDK 接收到服务端补偿的消息后，可生成部分会话和会话列表。与卸载前或换设备前比较，可能会有部分会话丢失的错觉。

如果您希望在卸载重装或换设备登录后，获取到之前的会话列表，可以参考如下方案：

- 申请增加离线消息补偿的天数，最大可修改为 7 天。**注意，设置时间过长，当单用户消息量超大时，可能会因为补偿消息量过大，造成端上处理压力的问题**。如有需要，请提交工单 。
- 在您的服务器中自行维护会话列表，并通过 API 向服务端获取需要展示的历史消息。

处理会话未读消息数　　处理会话未读消息数

更新时间:2024-08-30

即时通讯客户端常常需要对会话进行未读消息计数。

您可以使用 IMLib SDK 提供的接口直接获取会话中的未读消息数。具体能力如下：

- 获取所有会话（不含聊天室）中的未读消息总数（`getTotalUnreadCount`）
- 获取指定会话中的总未读消息数，或指定会话中指定消息类型的总未读消息数，或按会话类型获取总未读消息总数（`getUnreadCount`）

在用户使用您的 App 时，UI 上未读计数可能需要发生变化，此时您可以清除会话中的未读数（`clearMessagesUnreadStatus`）。

获取所有会话总未读消息数

您可以使用 `getTotalUnreadCount` 获取所有类型会话（不含聊天室）中未读消息的总数。

```
RongIMClient.getInstance().getTotalUnreadCount(callback);
```

获取成功后，`callback` 中会返回未读消息数（`unReadCount`）。

```
RongIMClient.getInstance().getTotalUnreadCount(new ResultCallback<Integer>() {

@Override
public void onSuccess(Integer unReadCount) {

}

@Override
public void onError(RongIMClient.ErrorCode ErrorCode) {

}
});
```

获取指定会话的总未读消息数

获取指定会话中的未读消息总数。

```
RongIMClient.getInstance().getUnreadCount(conversationType, targetId, callback);
```

参数	类型	说明
conversationType	ConversationType	会话类型。不适用于聊天室、超级群。
targetId	String	会话 ID
callback	ResultCallback<Integer>	回调接口

获取成功后，`callback` 中会返回未读消息数（`unReadCount`）。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";

RongIMClient.getInstance().getUnreadCount(conversationType, targetId,
new ResultCallback<Integer>() {

@Override
public void onSuccess(Integer unReadCount) {

}

@Override
public void onError(RongIMClient.ErrorCode ErrorCode) {

}
});
```

获取指定会话中指定消息类型的总消息未读数

> 提示
>
> 该方法在 SDK 5.1.5 版本引入。该方法仅在 RongCoreClient 中提供。

获取指定会话内指定的某一个或多个消息类型的未读数。

```
RongCoreClient.getInstance().getUnreadCount(targetId, conversationType, objectNames, callback)
```

参数	类型	说明
targetId	String	会话 ID。
conversationType	ConversationType	会话类型。不适用于聊天室、超级群。
objectNames	String[]	消息类型标识数组。内置消息类型的标识可参见消息类型概述。
callback	IRongCoreCallback.ResultCallback	回调结果。

获取成功后，callback 中会返回未读消息数（unReadCount）。

按会话类型获取总未读消息数

获取多个指定会话类型的未读数。

```
RongIMClient.getInstance().getUnreadCount(conversationTypes, containBlocked, callback);
```

参数	类型	说明
conversationTypes	ConversationType []	会话类型数组。不适用于聊天室、超级群。
containBlocked	boolean	是否包含消息免打扰的未读消息数。true：包含。false：不包含。
callback	ResultCallback<Integer>	回调接口

获取成功后，callback 中会返回未读消息数（unReadCount）。

```
ConversationTypes[] conversationTypes = {ConversationTypes.PRIVATE, ConversationTypes.GROUP};
boolean containBlocked = true;

RongIMClient.getInstance().getUnreadCount(conversationTypes, containBlocked,
new ResultCallback<Integer>() {

@Override
public void onSuccess(Integer unReadCount) {

}

@Override
public void onError(RongIMClient.ErrorCode ErrorCode) {

}
});
```

按会话免打扰级别获取总未读消息数

> 提示
>
> SDK 从 5.2.5 版本开始支持该接口。仅在 ChannelClient 中提供。

获取已设置指定免打扰级别的会话的总未读消息数。SDK 将按照传入的免打扰级别配置查找会话，再返回这些所有会话的总未读消息数。

```
ChannelClient.getInstance().getUnreadCount(conversationTypes, levels, callback)
```

参数	类型	必填
conversationTypes	ConversationType []	会话类型数组。不适用于聊天室。
levels	[PushNotificationLevel] []	免打扰类型数组。详见免打扰功能概述。
callback	ResultCallback<Integer>	回调接口

获取成功后，callback 中会返回未读消息数（unReadCount）。

```
ConversationTypes[] conversationTypes = {ConversationTypes.PRIVATE, ConversationTypes.GROUP};
PushNotificationLevel[] levels = {PushNotificationLevel.PUSH_NOTIFICATION_LEVEL_MENTION}

ChannelClient.getInstance().getUnreadCount(conversationTypes, levels,

new ResultCallback<Integer>() {

@Override
public void onSuccess(Integer unReadCount) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode ErrorCode) {

}
});
```

按会话免打扰级别获取总未读 @ 消息数

> 提示
>
> SDK 从 5.2.5 版本开始支持该接口。仅在 ChannelClient 中提供。

获取已设置指定免打扰级别的会话的总未读 @ 消息数。SDK 将按照传入的免打扰级别配置查找会话，再返回这些所有会话的总未读 @ 消息数。

```
ChannelClient.getInstance().getUnreadMentionedCount(conversationTypes, levels, callback)
```

参数	类型	必填
conversationTypes	ConversationType []	会话类型数组。不适用于聊天室。
levels	[PushNotificationLevel] []	免打扰类型数组。详见免打扰功能概述。
callback	ResultCallback<Integer>	回调接口

获取成功后，callback 中会返回未读 @ 消息数（unReadCount）。

```
ConversationTypes[] conversationTypes = {ConversationTypes.PRIVATE, ConversationTypes.GROUP};
PushNotificationLevel[] levels = {PushNotificationLevel.PUSH_NOTIFICATION_LEVEL_MENTION}

ChannelClient.getInstance().getUnreadMentionedCount(conversationTypes, levels,

new ResultCallback<Integer>() {

@Override
public void onSuccess(Integer unReadCount) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode ErrorCode) {

}
});
```

清除指定会话未读数

按时间戳清除指定会话的未读数。SDK 会将该时间戳之前的消息的未读状态全部清除。

```
RongIMClient.getInstance().clearMessagesUnreadStatus(conversationType, targetId, timestamp, callback);
```

参数	类型	说明
conversationType	ConversationType	会话类型。不适用于聊天室、超级群。
targetId	String	会话 ID
timestamp	long	时间戳。此时间戳之前的未读消息都清除。
callback	OperationCallback	回调接口

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";
long timestamp = 1585811571;

RongIMClient.getInstance().clearMessagesUnreadStatus(conversationType, targetId, timestamp, new OperationCallback() {

@Override
public void onSuccess() {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

获取会话未读消息 获取会话未读消息

更新时间:2024-08-30

IMLib SDK 支持从指定会话中获取未读消息，可满足 App 跳转到第一条未读消息、展示全部未读 @ 消息的需求。

获取会话中第一条未读消息

获取会话中的最早一条未读消息。

```
RongCoreClient.getInstance().getTheFirstUnreadMessage(conversationType, targetId, callback);
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 Id
callback	IRongCoreCallback.ResultCallback<Message>	回调接口

获取成功后，callback 中会返回消息对象（Message）。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";


RongCoreClient.getInstance().getTheFirstUnreadMessage(conversationType, targetId, new ResultCallback<Message>() {
@Override
public void onSuccess(Message message) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {

}
});
```

获取会话中未读的 @ 消息

提示

- 低于 5.2.5 版本仅提供不带 `count` 与 `desc` 参数的 `getUnreadMentionedMessages` 方法，每次最多返回 10 条数据。
- 从 5.2.5 版本开始，`getUnreadMentionedMessages` 支持 `count` 与 `desc` 参数。

获取会话中最早或最新的未读 @ 消息，最多返回 100 条。

```
RongCoreClient.getInstance().getUnreadMentionedMessages(conversationType, targetId, count, desc, callback);
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 Id
count	int	消息条数。最大 100 条。
desc	boolean	`true`：拉取最新的 `count` 条数据。`false`：拉取最旧的 `count` 条数据。
callback	IRongCoreCallback.ResultCallback<List<Message>>	回调接口

获取成功后，callback 中会返回消息对象（Message）列表。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";
int count = 100;
boolean desc = true;

RongCoreClient.getInstance().getUnreadMentionedMessages(conversationType, targetId, count, desc, new ResultCallback<List<Message>>() {
@Override
public void onSuccess(List<Message> messageList) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {

}
});
```

删除会话　　删除会话

更新时间:2024-08-30

App 用户可能需要从会话列表中删除一个会话或多个会话，可以通过 SDK 删除会话功能实现。客户端的会话列表是根据本地消息生成的，删除会话操作指的是删除本地会话。

删除指定会话

调用 removeConversation 可实现软删除的效果。该接口实际不会删除会话内的消息，仅将该会话项目从 SDK 的会话列表中移除。成功删除会话后，App 可以刷新 UI，不再向用户展示该会话项目。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "会话 Id";

RongIMClient.getInstance().removeConversation(conversationType, targetId, new ResultCallback<Boolean>() {

@Override
public void onSuccess(Boolean success) {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID
callback	ResultCallback<Boolean>	回调接口

如果会话内再来一条消息，该会话会重新出现在会话列表中，App 用户可查看会话内的历史消息和最新消息。

SDK 未提供同时删除指定会话项目和会话历史消息的接口。如果需要同时删除会话内的消息，您可以在删除指定会话时同时调用删除消息的接口。详见删除消息。

按会话类型删除会话

App 用户可能需要清空某一类型的所有会话，例如清空所有群聊会话。SDK 支持按指定会话类型清空所有会话及会话信息，一次支持清空多个类型的会话。

```
Conversation.ConversationType[] mConversationTypes = {
Conversation.ConversationType.PRIVATE,
Conversation.ConversationType.GROUP
};

RongIMClient.getInstance().clearConversations(new RongIMClient.ResultCallback() {

@Override
public void onSuccess(Object object) {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}, mConversationTypes);
```

参数	类型	说明
callback	ResultCallback	移除会话是否成功的回调
conversationTypes	ConversationType ...	需要清空的会话类型列表。不支持超级群。

该接口仅清空会话在当前用户设备本地数据库内的消息。如果 App 已经开通**单群聊历史消息云存储**服务，服务端保存的该用户的历史消息记录不受影响。如果该用户从服务端获取历史消息，可能会获取到在本地已删除的消息。如果需要同时删除服务端的会话历史消息，您可以在删除会话后同时调用删除消息的接口。详见删除消息。

删除全部会话

SDK 内部没有清除全部会话的方法，如有需要可通过以下任一方式实现：

- 如果 App 不涉及超级群业务，您可使用 按会话类型删除会话，传入所有会话类型。这种方式会清除本地消息。
- 先获取会话列表，循环删除指定会话。

会话草稿　　会话草稿 保存草稿

更新时间:2024-08-30

- 保存一条草稿内容至指定会话。
- 保存草稿会更新会话 `sentTime`，该会话会排在列表前部。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";
String content = "草稿内容";

RongIMClient.getInstance().saveTextMessageDraft(conversationType, targetId, content, new
ResultCallback<Boolean>() {

@Override
public void onSuccess(Boolean success) {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
})
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 Id
content	String	草稿的文字内容
callback	ResultCallback<Boolean>	回调接口

获取草稿

获取草稿内容。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";

RongIMClient.getInstance().getTextMessageDraft(conversationType, targetId, new
ResultCallback<String>() {

@Override
public void onSuccess(String draft) {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 Id
callback	ResultCallback<String>	回调接口

删除草稿

删除草稿。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = "会话 Id";

RongIMClient.getInstance().clearTextMessageDraft(conversationType, targetId, new ResultCallback<Boolean>() {

@Override
public void onSuccess(Boolean success) {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 Id
callback	ResultCallback<Boolean>	回调接口

输入状态　　输入状态

更新时间:2024-08-30

应用程序可以在单聊会话中发送当前用户输入状态。对端收到通知后可以在 UI 展示 “xxx 正在输入”。

发送输入状态消息

在当前用户输入文本时调用 sendTypingStatus，发送当前用户输入状态。

```
RongIMClient.getInstance().sendTypingStatus(conversationType, targetId, typingContentType);
```

参数	类型	说明
conversationType	ConversationType	会话类型。该接口仅支持单聊会话类型。
targetId	String	会话 ID。
typingContentType	String	正在输入的消息的类型名。如文本消息，应该传类型名 "RC:TxtMsg"。

监听输入状态

应用程序可以使用 setTypingStatusListener 设置 TypingStatusListener，监听在单聊类型的会话收到的输入状态通知。在收到对端发来的输入状态通知时，SDK 会通过 onTypingStatusChanged 回调方法返回当前正在输入的用户列表和消息类型。

```
RongIMClient.setTypingStatusListener(new RongIMClient.TypingStatusListener() {
@Override
public void onTypingStatusChanged(Conversation.ConversationType type, String targetId, Collection<TypingStatus> typingStatusSet) {
//当输入状态的会话类型和 targetID 与当前会话一致时，才需要显示
if (type.equals(mConversationType) && targetId.equals(mTargetId)) {
// count 表示当前会话中正在输入的用户数量，目前只支持单聊
int count = typingStatusSet.size();
if (count > 0) {
Iterator iterator = typingStatusSet.iterator();
TypingStatus status = (TypingStatus) iterator.next();
String objectName = status.getTypingContentType();

MessageTag textTag = TextMessage.class.getAnnotation(MessageTag.class);
MessageTag voiceTag = VoiceMessage.class.getAnnotation(MessageTag.class);
//匹配对方正在输入的是文本消息还是语音消息
if (objectName.equals(textTag.value())) {
//显示 “对方正在输入”
mHandler.sendEmptyMessage(SET_TEXT_TYPING_TITLE);
} else if (objectName.equals(voiceTag.value())) {
//显示 "对方正在讲话"
mHandler.sendEmptyMessage(SET_VOICE_TYPING_TITLE);
}
} else {
//当前会话没有用户正在输入，标题栏仍显示原来标题
mHandler.sendEmptyMessage(SET_TARGETID_TITLE);
}
}
}
});
```

管理标签信息数据 管理标签信息数据

更新时间:2024-08-30

SDK 从 5.1.1 版本开始支持创建标签。

本文描述如何 App 如何使用 RongCoreClient 下的接口创建和管理标签信息数据。客户端 SDK 支持用户创建标签信息（TagInfo），用于对会话进行标记分组。每个用户最多可以创建 20 个标签。App 用户创建的标签信息数据会同步融云服务端。

标签信息（TagInfo）的定义如下：

参数	类型	说明
tagId	String	标签唯一标识，字符型，长度不超过 10 个字。
tagName	String	长度不超过 15 个字，标签名称可以重复。
count	int	匹配的会话个数。
timestamp	long	时间戳由 SDK 内部协议栈提供。

提示

本文仅描述如何管理标签信息数据。关于如何为会话设置标签、以及如何按标签获取会话数据，请参见设置与使用会话标签。

创建标签信息

创建标签，每个用户最多可以创建 20 个标签。

```
RongCoreClient.getInstance().addTag(tagInfo, new IRongCoreCallback.OperationCallback() {

@Override
public void onSuccess() {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

移除标签信息

移除标签。移除标签信息时只需要传入 TagInfo 中的 tagId。

```
RongCoreClient.getInstance().removeTag(tagId, new IRongCoreCallback.OperationCallback() {

@Override
public void onSuccess() {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

编辑标签信息

更新标签信息。您可以修改 TagInfo 中的标签名称（tagName）字段。

```
RongCoreClient.getInstance().updateTag(tagInfo, new IRongCoreCallback.OperationCallback() {

@Override
public void onSuccess() {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

获取标签信息列表

获取当前用户已创建的标签信息。成功回调中会返回 TagInfo 列表。

```
RongCoreClient.getInstance().getTags(new IRongCoreCallback.ResultCallback<List<TagInfo>>() {

/**
* 成功时回调
* @param messages 获取的消息列表
*/
@Override
public void onSuccess(List<TagInfo> tagInfos) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

多端同步标签信息修改

即时通讯支持同一用户账号在多端登录。如果您的 App 用户在当前设备上修改了标签信息，SDK 会负责通知该用户的其他设备。其他设备收到通知后，需要调用 getTags 从融云服务端获取最新标签信息。

设置标签信息更改监听器 IRongCoreListener.TagListener 后可在当前设备上接收到来自其他设备的标签信息修改通知。

> 提示
>
> - 请在初始化之后，连接之前调用该方法。
> - 在当前设备上修改标签信息不会触发该回调方法。服务端仅会通知 SDK 在同一用户账号登录的其他设备上触发回调。

```
RongCoreClient.getInstance().setTagListener(new IRongCoreListener.TagListener{
@Override
public void onTagChanged() {
}
});
```

当用户在其它端添加、移除、编辑标签时，TagListener 会触发 onTagChanged 回调。请在收到通知后调用 getTags 从融云服务端获取最新标签信息。

设置与使用会话标签 设置与使用会话标签

更新时间:2024-08-30

- SDK 从 5.1.1 版本开始支持会话标签功能，相关接口仅在 RongCoreClient 中提供。
- 在为会话设置标签前，请确保已创建标签信息。详见管理标签信息数据。
- 本功能不适用于聊天室、超级群。

每个用户最多可以创建 20 个标签，每个标签下最多可以添加 1000 个会话。如果标签下已添加 1000 个会话，继续在该标签下添加会话仍会成功，但会导致最早添加标签的会话被移除标签。

场景描述

会话标签常实现 App 用户对会话进行分组的需求。创建标签信息（TagInfo）后，App 用户可以为会话设置一个或多个标签。

设置标签后，可以利用会话的标签数据实现会话的分组获取、展示、删除等特性。还可以获取指定标签下所有会话的消息未读数，或在特定标签下设置某个会话置顶。

- **场景 1**：对会话列表中的每个会话打 tag，类似企业微信会话列表中的外部群，部门群，个人群等 tag。
- **场景 2**：通讯录根据 tag 来分组，类似 QQ 好友列表中的家人，朋友，同事分组等。
- **场景 3**：前两个场景的结合，按照 tag 来进行会话列表分组，类似 Telegram 的会话列表分组。

使用标签标记会话

在创建标签信息（TagInfo）后，App 用户可以使用标签标记会话。SDK 将用标签标记会话的操作视为将会话添加到标签中。

支持以下操作：

- 标记会话，即将一个或多个会话添加到指定标签
- 从标签中移除一个或多个会话
- 为指定会话移除一个或多个标签

将一个或多个会话添加到指定标签

SDK 将用标签标记会话的操作视为将会话添加到标签中。您可以将多个会话添加到一个标签。指定标签时只需要传入 TagInfo 中的 `tagId`。

```
RongCoreClient.getInstance().addConversationsToTag(tagId, conversationIdentifierList,
new IRongCoreCallback.OperationCallback() {
/**
* 成功回调
*/
@Override
public void onSuccess() {

}

/**
* 失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

参数	类型	说明
tagId	String	标签 ID
conversationIdentifierList	List<ConversationIdentifier>	会话标识列表。每个 ConversationIdentifier 中需要指定会话类型（ConversationType）和 Target ID。
callback	OperationCallback	回调

从指定标签下移除会话

App 用户可能需要携带指定标签的会话中移除一个或多个会话。例如，在所有添加了「培训班」标签的会话中移除与「Tom」的私聊会话。SDK 将该操作视为从指定标签中移除会话。移除成功后，会话仍然存在，但不再携带该标签。

```
RongCoreClient.getInstance().removeConversationsFromTag(tagId, conversationIdentifierList, callback);
```

参数	类型	说明
tagId	String	标签 ID
conversationIdentifierList	List<ConversationIdentifier>	会话标识列表。每个 ConversationIdentifier 中需要指定会话类型（ConversationType）和 Target ID。
callback	OperationCallback	回调

为指定会话中移除标签

App 用户可能为指定会话中添加了多个标签。SDK 支持一次移除单个或多个标签。移除时需要传入所有待移除 TagInfo 的 tagId 列表。

```
RongCoreClient.getInstance(). removeTagsFromConversation(conversationIdentifier, tagIds,
new IRongCoreCallback.OperationCallback() {
/**
* 成功回调
*/
@Override
public void onSuccess() {

}

/**
* 失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

获取指定会话的所有标签

获取指定会话携带的所有标签。获取成功后，回调中返回 ConversationTagInfo 的列表。每个 ConversationTagInfo 中包含对应的标签信息 TagInfo 和置顶状态信息（会话是否在携带该标签信息的所有会话中置顶）。

```
RongCoreClient.getInstance().getTagsFromConversation(conversationIdentifier,
new IRongCoreCallback.ResultCallback<List<ConversationTagInfo>>() {
/**
* 成功回调
*/
@Override
public void onSuccess(List<ConversationTagInfo> conversationTagInfos) {

}

/**
* 失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

参数	类型	说明
conversationIdentifier	ConversationIdentifier	会话标识，需要指定会话类型（ConversationType）和 Target ID。
callback	ResultCallback<List<ConversationTagInfo>>	结果回调，返回会话携带的所有标签。

多端同步会话标签修改

即时通讯支持同一用户账号在多端登录。如果您的 App 用户在当前设备上修改了会话标签，SDK 会负责通知该用户的其他设备。其他设备收到通知后，可以调用 getTagsFromConversation 从融云服务端获取指定会话的最新标签数据。

设置会话标变更监听器 IRongCoreListener.ConversationTagListener 后可在当前设备上接收到来自其他设备的会话标签修改通知。

> 提示
>
> - 请在初始化之后，连接之前调用该方法。
> - 在当前设备上修改标签信息不会触发该回调方法。服务端仅会通知 SDK 在同一用户账号登录的其他设备上触发回调。

```
RongCoreClient.getInstance().setConversationTagListener(new IRongCoreListener.ConversationTagListener{
@Override
public void onConversationTagChanged() {
}
});
```

当 App 用户在其它端添加、移除、编辑会话上的标签时，ConversationTagListener 会触发 onConversationTagChanged 回调。请在收到通知后调用 getTagsFromConversation 从融云服务端获取指定会话的最新标签数据。

按标签操作会话数据

SDK 支持对携带指定标签的会话进行操作。App 用户为会话添加标签后，可以实现以下操作：

- 配合使用会话置顶功能，可以实现在携带指定标签的会话中置顶会话
- 按标签获取会话列表，即获取携带指定标签的所有会话
- 按标签获取未读消息数

- 清除标签对应会话的未读消息数
- 删除标签对应的会话

在携带指定标签的会话中置顶

App 可以使用会话标签按照业务需求对会话进行分类和展示。如果需要在同一类会话（携带同一标签的所有会话）中置顶显示会话，可以使用会话置顶功能。

详细实现方式请参见会话置顶的**在标签下置顶会话。

分页获取本地指定标签下会话列表

以会话中最后一条消息时间戳为界，分页获取本地指定标签下会话列表。该方法仅从本地数据库中获取数据。从 5.6.4 版本开始，该接口返回的 `Conversation` 对象新增 `isTopForTag` 属性，如果为 `true`，表示该会话在当前 `tagId` 下为置顶会话。

```
RongCoreClient.getInstance().getConversationsFromTagByPage(tagId, ts, count, callback);
```

参数	类型	说明
tagId	String	标签 ID
ts	long	会话的时间戳。获取这个时间戳之前的会话列表。首次可传 `0`，后续可以使用返回的 Conversation 对象的 `sentTime` 或 `operationTime` 属性值，作为下一次查询的 `startTime`。推荐使用 `operationTime`（该属性仅在 5.6.8 及之后版本提供）。
count	int	获取数量(20<= count <=100)
callback	ResultCallback<List<Conversation>>	返回携带指定 tagId 的会话列表。

按标签获取未读消息数

获取携带指定标签的所有会话的未读消息数。

```
RongCoreClient.getInstance().getUnreadCountByTag(tagId, containBlocked, callback);
```

参数	类型	说明
tagId	String	标签 ID
containBlocked	boolean	是否包含免打扰消息
callback	ResultCallback<Integer>	按标签获取未读消息数回调

清除标签对应会话的未读消息数

> 提示
>
> SDK 从 5.1.5 开始提供该接口。

清除携带指定标签的所有会话的未读消息数。

```
RongCoreClient.getInstance().clearMessagesUnreadStatusByTag(tagId, callback)
```

参数	类型	说明
tagId	String	标签 ID
callback	IRongCoreCallback.ResultCallback	操作结果回调

删除标签对应的会话

> 提示
>
> SDK 从 5.1.5 开始提供该接口。

删除指定标签下的全部会话，同时解除这些会话和标签的绑定关系。删除成功后，会话不再携带指定的标签。这些会话收到新消息时，会产生新的会话。

```
RongCoreClient.getInstance().clearConversationsByTag(tagId, deleteMessage, callback)
```

参数	类型	说明
tagId	String	标签 ID。
deleteMessage	boolean	是否清除该标签下所有会话的本地历史消息。
callback	IRongCoreCallback.ResultCallback	操作结果回调。

您可以通过 `deleteMessage` 参数配置是否同时清除这些会话对应的本地消息。

- 如果不删除会话对应的本地消息，再接收到新消息时，可以看到历史聊天记录。
- 如果删除会话对应的本地消息，再接收到新消息时，无法看到历史聊天记录。如果开通了**单群聊消息云存储服务**，服务端仍保存有消息历史。如需删除，请使用删除服务端历史消息接口。

会话置顶　　会话置顶

更新时间:2024-08-30

会话置顶功能提供以下能力：

- 在会话列表中置顶会话：通过会话（`Conversation`）的置顶（`isTop`）属性控制。
- 在携带同一标签的会话中置顶（需配合使用会话标签功能）：通过 ConversationTagInfo 类的 `isTop` 属性控制。

在会话列表中置顶会话

设置指定会话在会话列表中置顶后，SDK 将修改 Conversation 的 isTop 字段，该状态将会被同步到服务端。融云会在为用户自动同步会话置顶的状态数据。客户端可以主动获取或通过监听器获取到最新数据。

设置会话置顶

使用 setConversationToTop 设置会话置顶。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";
boolean isTop = true;
boolean needCreate = true;

RongIMClient.getInstance().setConversationToTop(conversationType, targetId, isTop, new
ResultCallback<Boolean>() {

@Override
public void onSuccess(Boolean success) {

}

@Override
public void onError(RongIMClient.ErrorCode ErrorCode) {

}
}) ;
```

参数	类型	说明
conversationType	ConversationType	会话类型，支持单聊、群聊、系统会话。
targetId	String	会话 ID
isTop	boolean	是否置顶。`true` 为置顶，`false` 为取消置顶。
callback	ResultCallback<Boolean>	回调接口

客户端通过本地消息数据自动生成会话与会话列表，并会在用户登录的多个设备之间同步置顶状态。如果在调用该 API 时，会话尚未生成，或者已被移除，SDK 的处理方式如下：

- 如果 SDK 版本 ≧ 5.6.8，当需要置顶的会话在本地或该用户登录的其他设备上不存在时，SDK 会自动创建会话并置顶。
- 如果 SDK 版本 < 5.6.8，必须使用支持 `needCreate` 参数的重载方法，并设置该参数为 `true`，SDK 才会自动创建会话并置顶。详见 API 参考 setConversationToTop。

设置会话置顶可选择是否更新会话时间（SDK 版本 ≧ 5.8.2）

使用 setConversationToTop 设置会话置顶。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";
boolean isTop = true;
boolean needCreate = true;
boolean needUpdateTime = true;

RongCoreClient.getInstance().setConversationToTop(conversationType, targetId, isTop, needCreate, needUpdateTime, new
IRongCoreCallback.ResultCallback<Boolean>() {

@Override
public void onSuccess(Boolean success) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode ErrorCode) {

}
}) ;
```

参数	类型	说明
conversationType	ConversationType	会话类型，支持单聊、群聊、系统会话。
targetId	String	会话 ID
isTop	boolean	是否置顶。`true` 为置顶，`false` 为取消置顶。

参数	类型	说明
needCreate	boolean	是否创建。`true` 为创建，`false` 为不创建。
needUpdateTime	boolean	是否更新时间。`true` 为更新，`false` 为不更新。
callback	ResultCallback<Boolean>	回调接口

监听置顶状态同步

SDK 提供了会话状态（置顶状态数据和免打扰状态数据）同步机制。设置会话状态同步监听器后，如果会话状态改变，可在本端收到通知。

会话的置顶和免打扰状态数据同步后，触发 `ConversationStatusListener` 的 `onStatusChanged` 方法。详细说明可参见多端同步免打扰/置顶。

```
public interface ConversationStatusListener {
void onStatusChanged(ConversationStatus[] conversationStatus);
}
```

获取会话置顶状态

> 提示
>
> SDK 从 5.1.5 版本开始支持该功能。

主动获取指定会话的置顶状态。

```
RongCoreClient.getInstance().getConversationTopStatus(targetId, conversationType, callback)
```

参数	类型	说明
targetId	String	会话 ID
conversationType	ConversationType	会话类型
callback	IRongCoreCallback.ResultCallback	回调结果

获取置顶会话列表

主动获取指定会话类型的所有置顶会话。

```
Conversation.ConversationType[] conversationTypes = {Conversation.ConversationType.PRIVATE};

RongIMClient.getInstance().getTopConversationList(new ResultCallback<List<Conversation>>() {
@Override
public void onSuccess(List<Conversation> conversations) {

}

@Override
public void onError(ErrorCode e) {

}
}, conversationTypes);
```

参数	类型	说明
callback	ResultCallback<List<Conversation>>	回调接口
conversationTypes	Conversation.ConversationType...	会话类型数组

在携带标签的所有会话中置顶会话

> 提示
>
> 该功能相关接口仅在 RongCoreClient 中提供。在标签标记的所有会话中置顶是通过修改 `ConversationTagInfo.isTop` 字段实现的，不影响 Conversation 的 `isTop` 字段。

如果 App 实现了会话标签功能，App 用户可能会使用同一个标签标记多个会话，并且需要将其中一个会话置顶。SDK 在 ConversationTagInfo 中提供 `isTop` 属性，用于控制会话是否需要在携带同样标签的会话中置顶。

在标签下置顶会话

在携带指定标签的所有会话中设置指定会话置顶。例如，在所有添加了「培训班」标签的会话中将与「Tom」的私聊会话置顶。

```
String targetId = "useridoftom";
ConversationIdentifier conversationIdentifier = new ConversationIdentifier(Conversation.ConversationType.PRIVATE, targetId);
String tagId = "peixunban";

RongCoreClient.getInstance().setConversationToTopInTag(tagId, conversationIdentifier, isTop,
new IRongCoreCallback.OperationCallback() {
/**
* 成功回调
*/
@Override
public void onSuccess() {

}

/**
* 失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

参数	类型	说明
tagId	String	标签 ID
conversationIdentifier	ConversationIdentifier	会话标识，需要指定会话类型（ConversationType）和 Target ID。
isTop	boolean	是否置顶
callback	OperationCallback	操作回调

获取会话在标签下的置顶状态

查询指定会话是否在携带同一标签的所有会话中置顶。获取成功后会返回是否已置顶。

```
RongCoreClient.getInstance().getConversationTopStatusInTag(conversationIdentifier, tagId,
new IRongCoreCallback.ResultCallback<Boolean>() {
/**
* 获取成功回调
*/
@Override
public void onSuccess(Boolean bool) {

}
/**
* 获取失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

是否开启同步空置顶会话

> 此功能在 5.10.0 版本起开始支持。

在您卸载并重新安装应用或更换设备登录时，融云允许您决定是否希望保留会话列表中置顶的会话，包括那些尚未包含任何消息的空会话。默认情况下，这些置顶状态将被保留。

您可以在 SDK 初始化过程中选择是否启用同步置顶会话的功能。通过调用 `InitOption` 类的 `enableSyncEmptyTopConversation` 方法，并传入 `true` 或 `false` 作为参数，来启用或禁用此功能。默认情况下，此功能是关闭的。

示例代码

```
InitOption initOption = new InitOption.Builder()
.enableSyncEmptyTopConversation(true)
.build();
RongCoreClient.init(context, appKey, initOption);
```

免打扰功能概述 免打扰功能概述

更新时间:2024-08-30

「免打扰功能」用于控制用户在客户端设备离线时，是否可针对离线消息接收推送通知。

- **客户端为离线状态**：会话中有新离线消息时，用户默认通过推送通道收到消息且默认弹出通知。设置免打扰后，融云服务端不会为相关消息触发推送。
- **客户端在后台运行**：会话中有新消息时，用户直接收到消息。如果使用 IMLib，您需要自行判断 App 是否在后台运行，并根据业务需求自行实现本地通知弹窗。

> 提示
>
> 如果不需要接收推送，可以通过设置 SDK 的初始化配置中的 enablePush 参数为 false，向融云服务申请禁用推送服务（当前设备）。您也可以在断开连接时设置不接收推送（当前设备）。

免打扰设置维度

客户端 SDK 支持对单聊、群聊、系统会话业务进行以下多个维度的免打扰设置：

- App 的免打扰设置
- 按指定会话类型设置免打扰级别
- 按会话设置免打扰级别
- 全局免打扰

App 的免打扰设置

以 App Key 为单位，设置整个应用所有用户的默认免打扰级别。默认未设置，等同于全部消息都接收通知。该级别的配置暂未在控制台开放，如有需要，请提交工单。

- **全部消息均通知**：当前 App 下的用户可针对任何消息接收推送通知。
- **未设置**：默认全部消息都通知。
- **仅 @ 消息通知**：当前 App 下，仅针对提及（@）指定用户和群组全体成员的消息向离线用户发送推送通知。
- **仅 @ 指定用户通知**：当前 App 下，用户仅针对提及（@）当前用户的消息接收推送通知。例如：仅张三会接收且仅接收 “@张三 Hello” 的消息的通知。
- **仅 @群全员通知**：当前 App 下，用户仅针对提及（@）群组全体成员的消息接收推送通知。
- **都不接收通知**：当前 App 下，用户不针对任何消息接收推送通知，即任何离线消息都不会触发推送通知。
- **除 @ 消息外群聊消息不发推送**：当前 App 下，用户针对单聊消息、提及（@）指定用户的消息、和提及（@）群组全体成员的消息接收推送通知。

融云服务端判断是否需要推送时，App 级别的免打扰配置的优先级最低。如果存在以下任何一种用户级别的免打扰配置，以用户级别配置为准：

- 按指定会话类型设置免打扰级别
- 按会话设置免打扰级别
- 全局免打扰

按会话类型设置免打扰级别

> 提示
>
> 客户端 SDK 从 5.2.2.1 开始支持该功能。该功能属于用户级别设置。

客户端 SDK 提供 PushNotificationLevel，允许用户为会话类型（单聊、群聊、超级群、系统会话）配置触发推送通知的消息类别，或完全关闭通知。提供以下六个级别：

枚举值	数值	说明
PUSH_NOTIFICATION_LEVEL_ALL_MESSAGE	-1	与融云服务端断开连接后，当前用户可针对指定类型会话中的所有消息接收通知。
PUSH_NOTIFICATION_LEVEL_DEFAULT	0	未设置。未设置时均为此初始状态。
PUSH_NOTIFICATION_LEVEL_MENTION	1	与融云服务端断开连接后，当前用户仅针对指定类型的会话中提及（@）当前用户和全体群成员的消息接收通知。
PUSH_NOTIFICATION_LEVEL_MENTION_USERS	2	与融云服务端断开连接后，当前用户仅针对指定类型的会话中提及（@）当前用户的消息接收通知。例如：张三只会接收 “@张三 Hello” 的消息的通知。
PUSH_NOTIFICATION_LEVEL_MENTION_ALL	4	与融云服务端断开连接后，当前用户仅针对指定类型的会话中提及（@）全部群成员的消息接收通知。
PUSH_NOTIFICATION_LEVEL_BLOCKED	5	当前用户针对指定类型的会话中的任何消息都不接收推送通知。

具体设置方法详见按会话类型设置免打扰。

融云服务端判断是否需要为用户发送推送通知时，如果同时存在以下任何一种用户级别的免打扰配置，以下列配置为准：

- 按会话设置免打扰级别
- 全局免打扰

按会话设置免打扰级别

> 提示

客户端 SDK 从 5.2.2 开始支持该功能。该功能属于用户级别设置。

客户端 SDK 提供 PushNotificationLevel，允许用户为会话配置触发通知的消息类别，或完全关闭通知。提供以下六个级别：

枚举值	数值	说明
PUSH_NOTIFICATION_LEVEL_ALL_MESSAGE	-1	与融云服务端断开连接后，当前用户可针对指定类型会话中的所有消息接收通知。
PUSH_NOTIFICATION_LEVEL_DEFAULT	0	未设置。未设置时均为此初始状态。
PUSH_NOTIFICATION_LEVEL_MENTION	1	与融云服务端断开连接后，当前用户仅针对指定会话中提及（@）当前用户和全体群成员的消息接收通知。
PUSH_NOTIFICATION_LEVEL_MENTION_USERS	2	与融云服务端断开连接后，当前用户仅针对接收指定会话中提及（@）当前用户的消息接收通知。例如：张三只会接收 “@张三 Hello” 的消息的通知。
PUSH_NOTIFICATION_LEVEL_MENTION_ALL	4	与融云服务端断开连接后，当前用户仅针对指定会话中提及（@）全部群成员的消息接收通知。
PUSH_NOTIFICATION_LEVEL_BLOCKED	5	当前用户针对指定会话中的任何消息都不接收推送通知。

具体设置方法详见按会话设置免打扰。

融云服务端判断是否需要为用户发送推送通知时，如果该用户已配置全局免打扰，则已全局免打扰的配置细节为准。

全局免打扰

客户端 SDK 从 5.2.2 开始提供 PushNotificationQuietHoursLevel，允许用户配置何时接收通知以及触发通知的消息类别。提供了以下三个级别：

枚举值	数值	说明
PUSH_NOTIFICATION_QUIET_HOURS_LEVEL_DEFAULT	0	未设置。如未设置，SDK 会依次查询消息所属群的用户级别免打扰设置及其他非用户级别设置，再判断是否需要推送通知。
PUSH_NOTIFICATION_QUIET_HOURS_LEVEL_MENTION_MESSAGE	1	与融云服务端断开连接后，当前用户仅在指定时段内针对指定会话中提及（@）当前用户和全体群成员的消息接收通知。
PUSH_NOTIFICATION_QUIET_HOURS_LEVEL_BLOCKED	5	当前用户在指定时段内针对任何消息都不接收推送通知。

具体设置方法详见全局免打扰。

早于 5.2.2 的 SDK 版本不支持设置触发通知的消息类别，仅支持设置为接收或不接收推送通知。

免打扰设置的优先级

针对单聊、群聊、系统会话、融云服务端会遵照以下顺序搜索免打扰配置。优先级从左至右依次降低，以优先级最高的配置为准判断是否需要触发推送：

全局免打扰设置（用户级）> 指定会话类型的免打扰设置（用户级）> 指定会话的免打扰设置（用户级）> App 级的免打扰设置

API 接口列表

下表描述了适用于单聊、群聊、系统会话的免打扰配置 API 接口。

免打扰配置维度	客户端 API	服务端 API
设置指定时段内，应用全局的免打扰级别。	详见全局免打扰。	详见 设置用户免打扰时段
设置指定类型会话的免打扰级别	详见按会话类型设置免打扰。	详见设置会话类型免打扰 。
设置指定会话的免打扰级别	详见按会话设置免打扰。	详见设置会话免打扰 。
设置 App 级免打扰级别	客户端 SDK 不提供 API。	服务端不提供该 API。

按会话设置免打扰 按会话设置免打扰

更新时间:2024-08-30

本文描述如何为指定会话（targetId）设置免打扰级别。

> 提示
>
> 即时通讯客户端 SDK 支持多维度、多级别的免打扰设置。
>
> - App 开发者可实现从 App Key、指定细分业务（仅超级群）、用户级别多个维度的免打扰功能配置。在融云服务端决定是否触发推送通知时，不同维度的优先级如下：用户级别设置 > 指定超级群频道的默认配置（仅超级群支持）> 指定超级群会话的默认配置（仅超级群支持）> App Key 级设置。
> - 用户级别设置下包含多个细分维度。在融云服务端决定是否触发推送通知时，如存在用户级别配置，不同细分维度的优先级如下：全局免打扰 > 按频道设置的免打扰 > 按会话设置的免打扰 > 按会话类型设置的免打扰。详见免打扰功能概述。

支持的免打扰级别

免打扰级别提供了针对不同 @ 消息的免打扰控制。从 SDK 5.2.2 开始，指定会话的免打扰配置支持以下级别：

枚举值	数值	说明
PUSH_NOTIFICATION_LEVEL_ALL_MESSAGE	-1	所有消息均可进行通知。
PUSH_NOTIFICATION_LEVEL_DEFAULT	0	未设置。未设置时均为此初始状态。 注意：在此状态下，如果超级群与群频道均为未设置，则认为超级群与频道的默认免打扰级别为全部消息都通知。
PUSH_NOTIFICATION_LEVEL_MENTION	1	仅针对 @ 消息进行通知，包括 @指定用户 和 @所有人
PUSH_NOTIFICATION_LEVEL_MENTION_USERS	2	仅针对 @ 指定用户消息进行通知，且仅针对被 @ 的指定的用户进行通知。 如：@张三，则张三可以收到推送； @所有人不会触发推送通知。
PUSH_NOTIFICATION_LEVEL_MENTION_ALL	4	仅针对 @群全员进行通知，即只接收 @所有人的推送信息。
PUSH_NOTIFICATION_LEVEL_BLOCKED	5	不接收通知，即使为 @ 消息也不推送通知。

早于 5.2.2 的 SDK 版本仅支持设置为免打扰状态（不接收推送通知）或提醒状态（接收推送通知）。

管理会话的免打扰设置

即时通讯业务用户（userId）为指定会话（targetId）设置免打扰级别，支持单聊、群聊、超级群会话。

设置指定会话免打扰级别 （SDK >= 5.2.2）

> 提示
>
> 该接口在 ChannelClient 中，从 5.2.2 版本开始支持。

为当前用户设置指定会话的（targetId）免打扰级别。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";

ChannelClient.getInstance().setConversationNotificationLevel(conversationType,targetId,
IRongCoreEnum.PushNotificationLevel.PUSH_NOTIFICATION_LEVEL_DEFAULT,new IRongCoreCallback.OperationCallback() {
@Override
public void onSuccess() {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型。请注意以下限制： • 超级群会话类型：如在 2022.09.01 之前开通超级群业务，默认不支持为单个超级群会话所有消息设置免打扰级别（“所有消息”指所有频道中的消息和不属于任何频道的消息）。该接口仅设置指定超级群会话（targetId）中不属于任何频道的消息的免打扰状态级别。如需修改请提交工单。 • 聊天室会话类型：不支持，因为聊天室消息默认不支持消息推送提醒。
targetId	String	会话 ID

参数	类型	说明
lev	PushNotificationLevel	• `-1`：全部消息通知 • `0`：未设置（用户未设置情况下，默认以群 或者 APP级别的默认设置为准，如未设置则全部消息都通知） • `1`：仅针对 @ 消息进行通知 • `2`：仅针对 @ 指定用户进行通知 如：@张三 则张三可以收到推送，@所有人 时不会收到推送。 • `4`：仅针对 @ 群全员进行通知，只接收 @所有人 的推送信息。 • `5`：不接收通知
callback	OperationCallback	回调接口

移除指定会话免打扰级别 （SDK >= 5.2.2）

如需移除指定会话类型的免打扰设置，请调用设置接口，并将 lev 参数传入 0。

查询指定会话免打扰级别 （SDK >= 5.2.2）

> 提示
>
> 该接口在 ChannelClient 中，从 5.2.2 版本开始支持。

查询当前用户为指定会话（targetId）设置的免打扰级别。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";

ChannelClient.getInstance().getConversationChannelNotificationLevel(conversationType, targetId, new IRongCoreCallback.ResultCallback<
IRongCoreEnum.PushNotificationLevel>() {
@Override
public void onSuccess(IRongCoreEnum.PushNotificationLevel level) {

}
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {

});
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID
callback	ResultCallback<IRongCoreEnum.PushNotificationLevel>	回调接口

获取免打扰状态列表

> 提示
>
> 该接口在 RongIMClient 中。

获取所有设置了免打扰的会话。返回会话列表不包含具体免打扰级别信息，不包含频道信息。

```
Conversation.ConversationType[] conversationTypes = {ConversationType.PRIVATE, ConversationType.GROUP};

RongIMClient.getInstance().getBlockedConversationList(new RongIMClient.ResultCallback<List<Conversation>>() {

@Override
public void onSuccess(List<Conversation> conversations) {

}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
}, conversationTypes);
```

参数	类型	说明
callback	ResultCallback<List<Conversation>>	回调接口
conversationTypes	ConversationType ...	会话类型数组，可设置多个会话类型(不支持聊天室)。

多端同步免打扰状态

SDK 提供了会话状态（置顶或免打扰）同步机制，通过设置会话状态同步监听器，当在其它端修改会话状态时，可在本端实时监听到会话状态的改变。详见多端同步免打扰/置顶。

按频道设置免打扰 按频道设置免打扰

更新时间:2024-08-30

本文描述如何为超级群业务的指定频道（channelId）设置免打扰级别。

> 提示
>
> 即时通讯客户端 SDK 支持多维度、多级别的免打扰设置。
>
> - App 开发者可实现从 App Key、指定细分业务（仅超级群）、用户级别多个维度的免打扰功能配置。在融云服务端决定是否触发推送通知时，不同维度的优先级如下：用户级别设置 > 指定超级群频道的默认配置（仅超级群支持）> 指定超级群会话的默认配置（仅超级群支持）> **App Key** 级设置。
> - 用户级别设置下包含多个细分维度。在融云服务端决定是否触发推送通知时，如存在用户级别配置，不同细分维度的优先级如下：全局免打扰 > 按频道设置的免打扰 > 按会话设置的免打扰 > 按会话类型设置的免打扰。详见免打扰功能概述。

支持的免打扰级别

免打扰级别提供了针对不同 @ 消息的免打扰控制。从 SDK 5.2.2 开始，指定频道的免打扰配置支持以下级别：

枚举值	数值	说明
PUSH_NOTIFICATION_LEVEL_ALL_MESSAGE	-1	所有消息均可进行通知。
PUSH_NOTIFICATION_LEVEL_DEFAULT	0	未设置。未设置时均为此初始状态。 注意：在此状态下，如果超级群与群频道均为未设置，则认为超级群与频道的默认免打扰级别为全部消息都通知。
PUSH_NOTIFICATION_LEVEL_MENTION	1	仅针对 @ 消息进行通知，包括 @指定用户 和 @所有人
PUSH_NOTIFICATION_LEVEL_MENTION_USERS	2	仅针对 @ 指定用户消息进行通知，且仅针对被 @ 的指定的用户进行通知。 如：@张三，则张三可以收到推送； @所有人不会触发推送通知。
PUSH_NOTIFICATION_LEVEL_MENTION_ALL	4	仅针对 @群全员进行通知，即只接收 @所有人的推送信息。
PUSH_NOTIFICATION_LEVEL_BLOCKED	5	不接收通知，即使为 @ 消息也不推送通知。

早于 5.2.2 的 SDK 版本仅支持设置为免打扰状态（不接收推送通知）或提醒状态（接收推送通知）。

管理频道的免打扰设置

超级群（UltraGroup）支持在超级群的会话下创建独立的频道（channel），对消息数据（会话、消息、未读数）和群组成员分频道进行聚合。SDK 支持在超级群业务中的用户（userId）设置指定群频道（channelId）的免打扰级别。

设置指定频道免打扰级别 （SDK ≧ 5.2.2）

> 提示
>
> 该接口在 ChannelClient 中，从 5.2.2 版本开始支持。

为当前用户设置超级群频道中的消息的免打扰级别。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";
String channelId = " 频道 Id ";

ChannelClient.getInstance().setConversationChannelNotificationLevel(conversationType,targetId,channelId,
IRongCoreEnum.PushNotificationLevel.PUSH_NOTIFICATION_LEVEL_DEFAULT,new IRongCoreCallback.OperationCallback() {
@Override
public void onSuccess() {
if (callback != null) {
callback.onSuccess(notificationStatus);
}
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {
if (callback != null) {
callback.onError(coreErrorCode);
}
}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型。
targetId	String	会话 ID

参数	类型	说明
channelId	String	超级群的会话频道 ID。 • 如果传入频道 ID，则针对该指定频道设置消息免打扰级别。如果不指定频道 ID，则对所有超级群消息生效。 • **注意**：2022.09.01 之前开通超级群业务的客户，如果不指定频道 ID，则默认传 "" 空字符串，即仅针对指定超级群会话（`targetId`）中不属于任何频道的消息设置免打扰状态级别。如需修改请提交工单。
lev	PushNotificationLevel	• `-1`：全部消息通知 • `0`：未设置（用户未设置情况下，默认以群 或者 APP级别的默认设置为准，如未设置则全部消息都通知） • `1`：仅针对 @ 消息进行通知 • `2`：仅针对 @ 指定用户进行通知 如：@张三 则张三可以收到推送，@所有人 时不会收到推送。 • `4`：仅针对 @ 群全员进行通知，只接收 @所有人 的推送信息。 • `5`：不接收通知
callback	OperationCallback	回调接口

获取指定频道免打扰级别 （SDK ≧ 5.2.2）

获取为当前用户设置的超级群频道免打扰级别。

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";
String channelId = " 频道 Id ";

ChannelClient.getInstance().getConversationChannelNotificationLevel(conversationType,targetId,channelId,new IRongCoreCallback.ResultCallback<
IRongCoreEnum.PushNotificationLevel>() {
@Override
public void onSuccess(IRongCoreEnum.PushNotificationLevel level) {

}


@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 Id
channelId	String	超级群的会话频道 ID，获取会话指定频道的设置。
callback	ResultCallback<IRongCoreEnum.PushNotificationLevel>	回调接口

多端同步免打扰状态

SDK 提供了会话状态（置顶或免打扰）同步机制，通过设置会话状态同步监听器，当在其它端修改会话状态时，可在本端实时监听到会话状态的改变。详见多端同步免打扰/置顶。

按会话类型设置免打扰 按会话类型设置免打扰

更新时间:2024-08-30

本文描述如何为指定类型（conversationType）的会话设置免打扰级别。

> 提示
>
> 即时通讯客户端 SDK 支持多维度、多级别的免打扰设置。
>
> - App 开发者可实现从 App Key、指定细分业务（仅超级群）、用户级别多个维度的免打扰功能配置。在融云服务端决定是否触发推送通知时，不同维度的优先级如下：用户级别设置 > 指定超级群频道的默认配置（仅超级群支持）> 指定超级群会话的默认配置（仅超级群支持）> App Key 级设置。
> - 用户级别设置下包含多个细分维度。在融云服务端决定是否触发推送通知时，如存在用户级别配置，不同细分维度的优先级如下：全局免打扰 > 按频道设置的免打扰 > 按会话设置的免打扰 > 按会话类型设置的免打扰。详见免打扰功能概述。

支持的免打扰级别

免打扰级别提供了针对不同 @ 消息的免打扰控制。从 SDK 5.2.2.1 开始，指定会话类型的免打扰配置支持以下级别：

枚举值	数值	说明
PUSH_NOTIFICATION_LEVEL_ALL_MESSAGE	-1	所有消息均可进行通知。
PUSH_NOTIFICATION_LEVEL_DEFAULT	0	未设置。未设置时均为此初始状态。 注意：在此状态下，如果超级群与群频道均为未设置，则认为超级群与频道的默认免打扰级别为全部消息都通知。
PUSH_NOTIFICATION_LEVEL_MENTION	1	仅针对 @ 消息进行通知，包括 @指定用户 和 @所有人
PUSH_NOTIFICATION_LEVEL_MENTION_USERS	2	仅针对 @ 指定用户消息进行通知，且仅针对被 @ 的指定的用户进行通知。 如：@张三，则张三可以收到推送； @所有人不会触发推送通知。
PUSH_NOTIFICATION_LEVEL_MENTION_ALL	4	仅针对 @群全员进行通知，即只接收 @所有人的推送信息。
PUSH_NOTIFICATION_LEVEL_BLOCKED	5	不接收通知，即使为 @ 消息也不推送通知。

早于 5.2.2.1 的 SDK 版本仅支持设置为免打扰状态（不接收推送通知）或提醒状态（接收推送通知）。

管理会话类型的免打扰级别

从 SDK 5.2.2.1 版本开始，支持在即时通讯业务中由用户（userId）为指定类型的会话（conversationType）设置免打扰级别，支持单聊、群聊、超级群会话。

设置指定会话类型免打扰级别

> 提示
>
> 该接口在 ChannelClient 中，从 5.2.2.1 版本开始支持。

为用户设置指定会话类型（conversationType）的免打扰级别，支持单聊、群聊、超级群会话。

```
ConversationType conversationType = ConversationType.PRIVATE;

ChannelClient.getInstance().setConversationTypeNotificationLevel(conversationType,
IRongCoreEnum.PushNotificationLevel.PUSH_NOTIFICATION_LEVEL_DEFAULT,new IRongCoreCallback.OperationCallback() {
@Override
public void onSuccess() {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型。不支持聊天室类型，因为聊天室默认就是不接受消息提醒的。

参数	类型	说明
lev	PushNotificationLevel	• `-1`：全部消息通知 • `0`：未设置（用户未设置情况下，默认以群 或者 APP级别的默认设置为准，如未设置则全部消息都通知） • `1`：仅针对 @ 消息进行通知 • `2`：仅针对 @ 指定用户进行通知 如：@张三 则张三可以收到推送，@所有人 时不会收到推送。 • `4`：仅针对 @ 群全员进行通知，只接收 @所有人 的推送信息。 • `5`：不接收通知
callback	OperationCallback	回调接口

移除指定会话类型的免打扰级别

如需移除指定会话类型的免打扰级别设置，请调用设置接口，并将 lev 参数传入 0。

查询指定会话类型的免打扰级别

> 提示
>
> 该接口在 ChannelClient 中，从 5.2.2.1 版本开始支持。

查询当前用户为指定会话类型（conversationType）设置的免打扰级别。

```
ConversationType conversationType = ConversationType.PRIVATE;

ChannelClient.getInstance().getConversationTypeNotificationLevel(conversationType, new IRongCoreCallback.ResultCallback<
IRongCoreEnum.PushNotificationLevel>() {
@Override
public void onSuccess(IRongCoreEnum.PushNotificationLevel level) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型
callback	ResultCallback<IRongCoreEnum.PushNotificationLevel>	回调接口

多端同步免打扰状态

SDK 提供了会话状态（置顶或免打扰）同步机制，通过设置会话状态同步监听器，当在其它端修改会话状态时，可在本端实时监听到会话状态的改变。详见多端同步免打扰/置顶。

多端同步免打扰/置顶 多端同步免打扰/置顶

更新时间:2024-08-30

SDK 提供了会话状态（置顶或免打扰）同步机制，通过设置会话状态同步监听器，当在其它端修改会话状态时，可在本端实时监听到会话状态的改变。

监听器说明

RongIMClient 中提供了 ConversationStatusListener 监听器。设置监听后，在会话的状态（置顶和免打扰）改变时，会触发以下方法：

```
public interface ConversationStatusListener {
void onStatusChanged(ConversationStatus[] conversationStatus);
}
```

onStatusChanged 方法返回 conversationStatus 的列表，参数如下：

参数	类型	描述
conversationType	ConversationType	会话类型。
targetId	String	会话 ID。
isTop	boolean	会话是否被设置为置顶。
level	PushNotificationLevel	会话的免打扰级别。SDK 从 5.2.5 版本支持多端同步免打扰级别。具体级别说明详见免打扰功能概述。
notificationStatus	ConversationNotificationStatus	会话提醒状态。仅支持提醒（1）或免打扰（0）两种状态。

设置监听器

设置会话状态（置顶和免打扰）多端同步监听器。

```
//同步监听器
RongIMClient.ConversationStatusListener listener = new RongIMClient.ConversationStatusListener() {
@Override
public void onStatusChanged(ConversationStatus[] conversationStatus) {
if (conversationStatus == null) {
return;
}
for (ConversationStatus status : conversationStatus) {
Conversation.ConversationType conversationType = status.getConversationType(); //获取会话类型
String targetId = status.getTargetId(); //获取会话 Id
boolean isTop = status.isTop(); //获取该会话当前状态是否置顶
IRongCoreEnum.PushNotificationLevel level = status.getNotificationLevel();// 获取PushNotificationLevel（5.2.5新增）
Conversation.ConversationNotificationStatus notificationStatus = status.getNotifyStatus(); //获取该会话当前的免打扰状态。
}
}
};
RongIMClient.getInstance().setConversationStatusListener(listener); //设置监听器
```

多端同步阅读状态 多端同步阅读状态

更新时间:2024-08-30

在即时通讯业务中，同一用户账号可能在多个设备上登录。仅在开通多设备消息同步服务后，融云会在多个设备之间同步消息数据，但设备上的会话中消息的已读/未读状态仅存储在本地。因此，应用程序可能希望将用户当前登录设备上指定会话的已读/未读状态同步给其他终端。

SDK 设计了多端同步单聊和群聊会话消息未读状态的机制。在一端主动调用同步消息未读状态接口 syncConversationReadStatus，其他端可通过会话状态同步监听器监听到会话中消息未读状态的最新数据。

> 提示
> - 如果 SDK 版本 ≤ 5.6.2，不支持多端同步系统会话的阅读状态。
> - 超级群业务采用不同多端消息未读状态同步机制。详见超级群文档清除消息未读状态。

主动同步消息未读状态

多端登录时，通知其它终端同步某个会话的消息未读状态。

```
RongIMClient.getInstance().syncConversationReadStatus(conversationType, targetId, timestamp, callback);
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID
timestamp	String	该会话中已读的最后一条消息的发送时间戳
callback	RongIMClient.OperationCallback	回调接口

```
ConversationType conversationType = ConversationType.PRIVATE;
String targetId = " 会话 Id ";
String timestamp = "12222222";

RongIMClient.getInstance().syncConversationReadStatus(conversationType, targetId, timestamp, new RongIMClient.OperationCallback() {
@Override
public void onSuccess() {
}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {
}
});
```

监听消息未读状态同步数据

RongIMClient 中提供了 SyncConversationReadStatusListener 监听器。客户端设置该监听器后，才能接收来自其他同步的阅读状态数据。

```
RongIMClient.getInstance().setSyncConversationReadStatusListener(listener);
```

在接收到阅读状态同步数据后，会触发监听器的以下方法。SDK 会将指定会话中早于等于 syncConversationReadStatus 传入时间戳的消息均置为已读：

```
onSyncConversationReadStatus(Conversation.ConversationType type, String targetId)
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID

群组业务概述 群组业务概述

更新时间:2024-08-30

群聊是即时通讯类应用中常见的多人通讯方式，一般包含两个及以上的用户。融云的群组业务支持丰富的群组成员管理、禁言管理等特性，支持离线消息推送和历史消息记录漫游，可用于兴趣群、办公群、客服服务沟通等。

群组业务要点如下：

- 融云只负责将消息传达给群组中的所有用户，不维护群组成员的资料（头像、名称、群成员名片等），需要由开发者应用服务器维护。
- 创建/解散/加入/退出群组等群组管理操作，必须由 App 服务器请求融云服务端 API 实现。融云客户端 SDK 不提供相应方法。详见下方群组管理功能。
- App Key 下可创建的群组数量没有限制，单个群组默认成员上限为 3000 人。可提交工单修改群成员人数上限。
- 单个用户可加入的群组数量无限制。
- 从控制台 IM 服务管理页面为 App Key 开启**单群聊消息云端存储**服务后，可使用融云提供的消息存储服务，实现消息历史记录漫游。**IM 旗舰版**或**IM 尊享版**可开通该服务。具体功能与费用以融云官方价格说明页面及计费说明文档为准。

服务配置

客户端 SDK 默认支持群组业务，不需要申请开通。

群组业务的部分基础功能与增值服务可以在控制台的免费基础功能和 IM 服务管理页面进行开通和配置。

客户端 SDK 使用须知

> 提示
>
> - 仅 IMKit 提供开箱即用的群聊会话 UI 组件。IMLib 不提供开箱即用的群聊会话 UI 组件。IMKit 依赖 IMLib，因此 IMKit 具备 IMLib 的全部能力。
> - 客户端不提供群组管理的 API。群组管理需要由 App 服务端调用相应的融云服务端 API（Server API）接口完成。

群组管理功能

融云不会托管用户，也不管理群组的业务逻辑，因此群的业务逻辑全部需要在 App 服务器进行实现。

> 提示
>
> 群主、群管理员、群公告、邀请入群、群号搜索等均为群组业务逻辑，需在 App 侧自行实现。

对于客户端开发人员来说，创建群组、解散等基础管理操作只需要与 App 自身的业务服务端交互即可，由 App 服务端负责调用相应的融云服务端 API（Server API）接口完成相关操作。

下表列出了融云服务端提供的群组基础管理接口。注意，客户端不提供群组管理的 API。

功能分类	功能描述	融云服务端 API
创建、解散群组	提供创建者用户 ID、群组 ID、和群名称，向融云服务端申请建群。如解散群组，则群成员关系不复存在。	创建群组、解散群组
加入、退出群组	加入群组后，默认可查看入群以后产生的新消息。退出群组后，不再接收该群的新消息。	加入群组、退出群组
修改融云服务端的群组信息	修改在融云推送服务中使用的群组信息。	刷新群组信息
查询群组成员	查询指定群组所有成员的用户 ID 信息。	查询群组成员
查询用户所在群组	根据用户 ID 查询该用户加入的所有群组，返回群组 ID 及群组名称。融云不存储群组资料信息，群组资料及群成员信息需要开发者在应用服务器自行维护，如应用服务端维护的用户群组关系有缺失时，可通过此接口来核对校验。	查询用户所在群组
同步用户所在群组	向融云服务端同步指定用户当前所加入的所有群组，防止应用中的用户群组信息与融云服务端的用户所属群信息不一致。如果在集成融云服务前 App Server 上已有群组及成员数据，第一次连接融云服务器时，可使用此接口向融云同步已有的用户与群组对应关系。	同步用户所在群组
群组单人禁言	在指定的单个群组中或全部群组中，禁言一个或多个用户。被禁言用户可以接收查看群组中其他用户消息，但不能通过客户端 SDK 发送消息。	单人禁言
群组全体禁言	将群组全体成员禁言。被禁言群组的所有成员均不能发送消息，需要某些用户可以发言时，可将此用户加入到群禁言用户白名单中。	全体禁言
群组禁言用户白名单	群组被整体禁言后，禁言白名单中用户可以发送群消息。	全体成员禁言白名单

群聊消息功能

群聊消息功能与单聊业务类似，共用部分 API 及配置。

功能	描述	客户端 API	融云服务端 API
发送消息	可发送普通消息与媒体消息，例如文本、图片、GIF 等，或自定义消息。支持在发送消息时添加 @ 信息。	发送消息	发送群聊消息
发送群聊定向消息	可发送普通消息与媒体消息给群组中的指定的一个或多个成员，其他成员不会收到该消息。	发送群定向消息	发送群聊定向消息
接收消息	监听并实时接收消息，或在客户端上线时接收离线消息。	接收消息	不适用
群聊已读回执	发送群消息后如需要查看消息的阅读状态，需要先发送回执请求，在通过接受者的响应获取已读数据。	群聊已读回执	不适用

功能	描述	客户端 API	融云服务端 API
离线消息	支持离线消息存储，存储时间可设置（1 ～ 7 天），默认存储 7 天内的所有群消息，支持调整存储时长与存储的群消息数量。	管理离线消息存储配置	不提供该 API
离线消息推送	离线状态下，群组中有新消息时，支持 Push 通知。	Android 推送开发指南	不提供该 API
撤回消息	消息发送成功后可撤回该条消息。	撤回消息	撤回消息
本地搜索消息	消息存储在本地（移动端），支持按关键字或用户搜索本地指定会话的消息内容。	搜索消息	不提供该 API
获取历史消息	从本地数据库或远端获取历史消息。注意，从远端获取历史消息需要开通**单群聊消息云存储**服务，默认存储时长为 6 个月。	获取历史消息	不提供该 API
获取历史消息日志	融云服务端可以保存 APP 内所有会话的历史消息记录，历史消息记录以日志文件方式提供，并已经过压缩。您可以使用服务端 API 获取、删除指定 App 的历史消息日志	不提供该 API	获取历史消息日志
本地插入消息	在本地数据库中插入消息。本地插入的消息不会实际发送给服务器和对方。	插入消息	不适用
删除消息	支持按会话删除本地和存储在服务器的指定消息或会话中全部历史消息。	删除消息	消息清除
单群聊消息扩展	为原始消息增加状态标识（扩展数据为 KV 键值对），提供添加、删除、查询扩展信息的接口。	消息扩展	单/群聊消息扩展
自定义消息类型	如果内置消息类型满足不了您的需求，可以自定义消息类型。支持自定义普通消息类型与自定义媒体消息类型。	自定义消息类型	不适用

默认新入群的成员的用户仅可接收加入群组后产生的消息。如果需要查看入群之前的历史消息，请为 App Key 开启以下两项服务（请注意区分开发/生产环境）：

从控制台 IM 服务管理 页面开启**单群聊消息云端存储**服务。开启该服务后，可使用融云提供的消息存储服务，实现消息历史记录漫游。**IM 旗舰版**或**IM 尊享版**可开通该服务。具体功能与费用以融云官方价格说明 页面及计费说明 文档为准。

- 从控制台免费基础功能 页面开启**新用户获取加入群组前历史消息**。

群聊会话功能

群聊会话功能与单聊业务类似，共用部分 API 及配置。

功能	描述	客户端 API	融云服务端 API
获取会话	SDK 会根据收发的消息在本地数据库中生成对应会话。您可以从本地数据库获取 SDK 生成的会话列表。	获取会话	不提供该 API
获取会话未读消息	从指定会话中获取未读消息，可满足 App 跳转到第一条未读消息、展示全部未读 @ 消息的需求。	获取会话未读消息	不提供该 API
处理会话未读消息数	获取或清除会话中的未读消息数，可用于 UI 展示。	处理会话未读消息数	不提供该 API
删除会话	从 SDK 生成的会话列表中删除一个会话或多个会话	删除会话	不提供该 API
会话草稿	保存一条草稿内容至指定会话。	会话草稿	不提供该 API
输入状态	可设置指定的群聊会话，收到新的消息后是否进行提醒，默认进行新消息提醒。	输入状态	不提供该 API
管理会话标签	创建和管理标签信息数据，用于对会话进行标记分组。每个用户最多可以创建 20 个标签。App 用户创建的标签信息数据会同步融云服务端。	管理标签信息数据	不提供该 API
设置与使用会话标签	使用会话标签对会话进行分组。	设置与使用会话标签	不提供该 API
会话置顶	在会话列表中将指定会话置顶。	会话置顶	会话置顶
会话免打扰	控制用户在客户端设备离线时，是否可针对离线消息接收推送通知。支持按照会话或按会话类型设置免打扰。	免打扰功能概述	免打扰功能概述
多端同步会话免打扰/置顶状态	SDK 提供了会话状态（置顶或免打扰）同步机制，通过设置会话状态同步监听器，当在其它端修改会话状态时，可在本端实时监听到会话状态的改变。	多端同步免打扰/置顶	不适用
多端同步阅读状态	在同一用户账户的多个设备间主动同步会话的阅读状态。	多端同步阅读状态	不适用

与聊天室和超级群的区别

您可以通过以下文档了解业务类型之间的区别及所有功能：

- 即时通讯开发指导 · 业务类型介绍
- IM 尊享版、IM 旗舰版功能对照表

发送群定向消息　　发送群定向消息

更新时间:2024-08-30

可发送普通消息与媒体消息给群组中的指定的一个或多个成员，其他成员不会收到该消息。

开通服务

使用发送群组定向消息功能无需开通服务。注意，如需将群组定向消息存入服务端历史消息记录，需要开通以下服务：

- 单群聊历史消息云存储服务，可前往控制台 IM 服务管理 页面为当前使用的 App Key 开启服务。IM 旗舰版或IM 尊享版可开通该服务。具体功能与费用以融云官方价格说明 页面及计费说明 文档为准。
- 群定向消息云存储服务，需要提交工单 申请开通。

默认情况下，客户端发送与接收的群定向消息默认都不会存入历史消息服务，因此客户端调用获取历史消息的 API 时，从融云服务端返回的结果中不会包含当前用户发送、接收的群组定向消息。

发送群组定向普通消息

在群组中发送普通消息给群组中的指定用户。不在接收列表的其它用户不会收到这条消息。注意，Message 中仅保存群组 ID（Target ID），不会保存接收用户 ID 列表。

```
Conversation.ConversationType type = Conversation.ConversationType.GROUP;
String targetId = "123";
TextMessage content = TextMessage.obtain("定向消息文本内容");
Message message = Message.obtain(targetId, conversationType, messageContent);

String[] userIds = new String[]{"id_01", "id_02"};

RongCoreClient.getInstance().sendDirectionalMessage(message, userIds, null, null, new IRongCoreCallback.ISendMessageCallback() {
@Override
public void onAttached(Message message) {

}

@Override
public void onSuccess(Message message) {

}

@Override
public void onError(final Message message, IRongCoreEnum.CoreErrorCode errorCode) {

}
});
```

sendDirectionalMessage 中直接提供了用于控制推送通知内容（pushContent）和推送附加信息（pushData）的参数。不过，如果您需要更精细地控制离线推送通知，例如标题、内容、图标、或其他第三方厂商个性化配置，请使用消息推送属性进行配置，详见自定义消息推送通知。

- 如果发送的消息属于 SDK 内置的普通消息类型，例如 TextMessage，这两个参数可设置为 `null`。一旦消息触发离线推送通知时，融云服务端会使用各个内置消息类型默认的 `pushContent` 值。关于各类型消息的默认推送通知内容，详见用户内容类消息格式。
- 如果消息类型为自定义消息类型，且需要支持离线推送通知，则必须向融云提供 `pushContent` 字段，否则用户无法收到离线推送通知。
- 如果消息类型为自定义消息类型，但不需要支持远程推送通知（例如通过自定义消息类型实现的 App 业务层操作指令），可将 `pushContent` 字段留空。
- `Message` 的推送属性配置 `MessagePushConfig` 的 `pushContent` 和 `pushData` 会覆盖此处配置，并提供更多配置能力，例如自定义推送通知的标题。详见自定义消息推送通知。

参数	类型	说明
message	Message	要发送的消息体。必填属性包括会话类型（conversationType），会话 ID（targetId），消息内容（content）。注意，群组定向消息要求会话类型为 Conversation.ConversationType.GROUP。
userIds	String[]	需要接收消息的用户列表。
pushContent	String	修改或指定远程消息推送通知栏显示的内容。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见自定义消息推送通知。 • 如果希望使用融云默认推送通知内容，可以填 `null`。注意自定义消息类型无默认值。 • 如果要为该条消息指定个性化的离线推送通知，必须在以上任一处向融云提供 `pushContent` 字段内容。 • 如果您自定义的消息类型需要离线推送服务，必须在以上任一处向融云提供 `pushContent` 字段内容，否则用户无法收到离线推送通知。
pushData	String	远程推送附加信息。对端收到远程推送消息时，可通过以下方法获取该字段内容： io.rong.push.notification.PushNotificationMessage#getPushData() 。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见自定义消息推送通知。
callback	ISendMessageCallback	发送消息的回调。

发送群组定向媒体消息

在群聊中发送多媒体消息给指定的单个或多个用户。注意，Message 中仅保存群组 ID（Target ID），不会保存接收用户 ID 列表。

```
String targetId = "目标 ID";
ConversationType conversationType = ConversationType.PRIVATE;
Uri localUri = Uri.parse("file://图片的路径");//图片本地路径，接收方可以通过 getThumUri 获取自动生成的缩略图 Uri
boolean mIsFull = true; //是否发送原图
ImageMessage mediaMessageContent = ImageMessage.obtain(localUri, mIsFull);

Message message = Message.obtain(targetId, conversationType, mediaMessageContent);

String[] userIds = new String[]{"id_01", "id_02"};

RongCoreClient.getInstance().sendDirectionalMediaMessage(message, userIds, null, null, new IRongCoreCallback.ISendMediaMessageCallback() {
@Override
public void onProgress(Message message, int i) {

}

@Override
public void onCanceled(Message message) {

}

@Override
public void onAttached(Message message) {

}

@Override
public void onSuccess(Message message) {

}

@Override
public void onError(final Message message, final IRongCoreEnum.CoreErrorCode errorCode) {

}
});
```

sendDirectionalMediaMessage 中直接提供了用于控制推送通知内容（pushContent）和推送附加信息（pushData）的参数。不过，如果您需要更精细地控制离线推送通知，例如标题、内容、图标、或其他第三方厂商个性化配置，请使用消息推送属性进行配置，详见自定义消息推送通知。

- 如果发送的消息属于 SDK 内置的媒体消息类型，例如 ImageMessage，这两个参数可设置为 `null`。一旦消息触发离线推送通知时，融云服务端会使用各个内置消息类型默认的 `pushContent` 值。关于各类型消息的默认推送通知内容，详见用户内容类消息格式。
- 如果消息类型为自定义消息类型，且需要支持离线推送通知，则必须向融云提供 `pushContent` 字段，否则用户无法收到离线推送通知。
- 如果消息类型为自定义消息类型，但不需要支持远程推送通知（例如通过自定义消息类型实现的 App 业务层操作指令），可将 `pushContent` 字段留空。
- `Message` 的推送属性配置 `MessagePushConfig` 的 `pushContent` 和 `pushData` 会覆盖此处配置，并提供更多配置能力，例如自定义推送通知的标题。详见自定义消息推送通知。

参数	类型	说明
message	Message	要发送的消息体。必填属性包括会话类型（conversationType），会话 ID（targetId），消息内容（content）。**注意**，群组定向消息要求会话类型为 Conversation.ConversationType.GROUP。
userIds	String[]	需要接收消息的用户列表。
pushContent	String	修改或指定远程消息推送通知栏显示的内容。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见自定义消息推送通知。 • 如果希望使用融云默认推送通知内容，可以填 `null`。注意自定义消息类型无默认值。 • 如果要为该条消息指定个性化的离线推送通知，必须在以上任一处向融云提供 `pushContent` 字段内容。 • 如果您自定义的消息类型需要离线推送服务，必须在以上任一处向融云提供 `pushContent` 字段内容，否则用户无法收到离线推送通知。
pushData	String	远程推送附加信息。对端收到远程推送消息时，可通过以下方法获取该字段内容： io.rong.push.notification.PushNotificationMessage#getPushData() 。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见自定义消息推送通知。
callback	ISendMediaMessageCallback	发送媒体消息的回调

获取定向消息的目标用户列表

> 提示
>
> 从 5.8.0 版本起，SDK 支持获取定向消息的目标用户列表。此功能仅适用于普通群和超级群消息。

通过检查消息 Message 对象的 directedUserIds 属性，您可以获取定向消息的目标用户列表。如果该列表为空，表示此消息不是定向消息。

```
//获取定向消息列表
List directedUserIds = message.getDirectedUserIds();
```

聊天室概述 聊天室概述

更新时间:2024-08-30

聊天室（Chatroom）提供了一种不设用户上限，支持高并发消息处理的业务形态，可用于直播、社区、游戏、广场交友、兴趣讨论等场景。聊天室业务要点如下：

- App Key 下可创建的聊天室数量没有限制，单个聊天室成员数量没有限制。
- 聊天室具有自动销毁机制，默认情况下所有聊天室会在不活跃（连续时间段内无成员进出且无新消息）达到 1 小时后踢出所有成员并自动销毁，可延长该时间，也可配置为定时自动销毁。详见服务端文档聊天室销毁机制。
- 聊天室具有离线成员自动退出机制。满足默认预设条件时，融云服务端会踢出聊天室成员，详见退出聊天室。
- 聊天室本地消息会在退出聊天室时删除。**IM 旗舰版** 与 **IM 尊享版**客户可选择启用**聊天室消息云端存储**功能，将消息存储在融云服务端。具体功能与费用以融云官方价格说明页面及计费说明文档为准。
- 聊天室不具备离线消息转推送功能，只有在线的聊天室成员可接收聊天室消息。

客户端 UI 框架参考设计

聊天室产品暂不提供聊天室会话专用的 UI 组件。您可以参考以下 UI 框架设计了解聊天室的设计思路。

- 下图**聊天室**标签中为聊天室消息列表。
- 下图**聊天室管理**窗口中展示了聊天室支持的部分能力，如禁言、封禁、白名单等。

服务配置

客户端 SDK 默认支持聊天室，不需要申请开通。

聊天室的部分基础功能与增值服务可以在控制台的免费基础功能和 IM 服务管理页面进行开通和配置。

客户端 SDK 使用须知

> 提示
>
> IMKit 依赖 IMLib，因此 IMKit 具备 IMLib 的全部能力，包括聊天室。但请注意 IMKit 不提供开箱即用的聊天室会话 UI 组件。

聊天室功能接口

聊天室会话关系由融云负责建立并保持连接。SDK 提供加入、退出等部分聊天室管理接口。更多聊天室管理功能需要配合使用即时通讯服务端 API。下表描述了融云聊天室主要的功能接口。

功能分类	功能描述	客户端 API	融云服务端 API
创建与销毁聊天室	手动创建聊天室，或手动销毁聊天室。**注意**：客户端 SDK 无单独的创建聊天室 API。客户端不提供手动销毁聊天室 API。	不提供该 API	创建房间、销毁房间
加入与退出聊天室	加入已存在的聊天室，请确保聊天室 ID 已存在。加入与退出聊天室仅客户端提供 API。**注意**：客户端有废弃接口可支持在聊天室不存在时创建聊天室再加入，但已不推荐使用。	加入聊天室、退出聊天室	不提供该 API
查询聊天室房间与用户信息	• 查询聊天室房间的基础信息，包括聊天室 ID、名称、创建时间。 • 查询聊天室成员信息，支持获取聊天室成员用户 ID、加入时间，最多返回 500 个成员信息，支持按加入时间排序。	查询聊天室信息	查询房间信息
聊天室保活	添加一个或多个聊天室到聊天室保活列表。在保活列表中的聊天室不会被融云服务端自动销毁。	不提供该 API	保活房间

功能分类	功能描述	客户端 API	融云服务端 API
聊天室属性管理	在指定聊天室中设置自定义属性。比如在语音直播聊天室场景中，利用此功能记录聊天室中各麦位的属性；或在狼人杀等卡牌类游戏场景中记录用户的角色和牌局状态等。 聊天室属性以 Key-Value 的方式进行存储，支持设置、删除与查询属性，支持批量和强制操作。	聊天室属性	属性管理（KV）
封禁/解封聊天室用户	封禁一个或多个聊天室成员。被封禁成员将被踢出指定聊天室，并在封禁时间内不能再进入此聊天室中。	不提供该 API	成员封禁
聊天室用户白名单	需在 IM 服务管理 页面**普通服务**下开通后使用。**IM 旗舰版**或**IM 尊享版**可开通该服务。具体功能与费用以融云官方价格说明 页面及计费说明 文档为准。 用户被加入某个聊天室的白名单后，在该聊天室消息量较大的情况下，该用户发送的消息不会被丢弃；并且用户也不会被融云服务端自动踢出该聊天室。	不提供该 API	聊天室白名单服务
发送聊天室消息	发送聊天室消息。	发送消息	发送聊天室消息
撤回聊天室消息	撤回聊天室消息。	撤回消息	消息撤回
获取聊天室历史消息	获取聊天室历史消息。	获取聊天室历史消息	历史消息日志
聊天室低级别消息	需在 IM 服务管理 页面**普通服务**下开通后使用。**IM 旗舰版**或**IM 尊享版**可开通该服务。具体功能与费用以融云官方价格说明 页面及计费说明 文档为准。 如果消息类型在低级别消息列表中，该类型的消息全部视为低级别消息。当服务器负载高时，高级别的消息优先保留，低级别消息则优先丢弃。默认情况下，所有消息均为高级别消息。	不提供该 API	聊天室消息优先级服务
聊天室消息白名单	需在 IM 服务管理 页面**普通服务**下开通后使用。**IM 旗舰版**或**IM 尊享版**可开通该服务。具体功能与费用以融云官方价格说明 页面及计费说明 文档为准。 如果消息类型在聊天室消息白名单中，该类型的消息全部受到保护，在聊天室消息量较大的情况下也不会被丢弃。	不提供该 API	聊天室白名单服务
聊天室成员禁言	在指定的某个聊天室中，禁言一个或多个成员。聊天室成员被禁言后，可以接收并查看聊天室中用户聊天信息，但不能通过往该聊天室内发送消息。	不提供该 API	单人禁言
全体成员禁言	设置某一聊天室全部成员禁言，或取消指定聊天室全部成员禁言状态。设置全体群成员禁言后，该聊天室的所有成员均不能通过客户端 SDK 往该群组内发送消息。	不提供该 API	全体禁言
全体禁言白名单	添加一个或多个群成员到聊天室全体成员禁言白名单。聊天室成员被添加到白名单后，即使该聊天室处于全体成员禁言状态，该成员仍可通过客户端 SDK 往该聊天室发送消息。	不提供该 API	全体禁言
全局禁言聊天室成员	需在 IM 服务管理 页面**普通服务**下开通后使用。**IM 旗舰版**或**IM 尊享版**可开通该服务。具体功能与费用以融云官方价格说明 页面及计费说明 文档为准。 添加一个或多个用户到聊天室全局禁言列表中，列表中的用户在应用下的所有聊天室中都无法发送消息。	不提供该 API	全局禁言

与群组和超级群的区别

您可以通过以下文档了解业务类型之间的区别及所有功能：

- 即时通讯开发指导 · 业务类型介绍
- IM 尊享版、IM 旗舰版功能对照表

聊天室服务配置

更新时间:2024-08-30

聊天室业务本身不需要单独申请开通，但部分聊天室服务需要在控制台开通与配置，例如**聊天室广播消息**、**聊天室消息云端存储**、以及与聊天室相关的回调地址等。

聊天室服务配置主要在**免费基础功能**和 **IM 服务管理**页面。

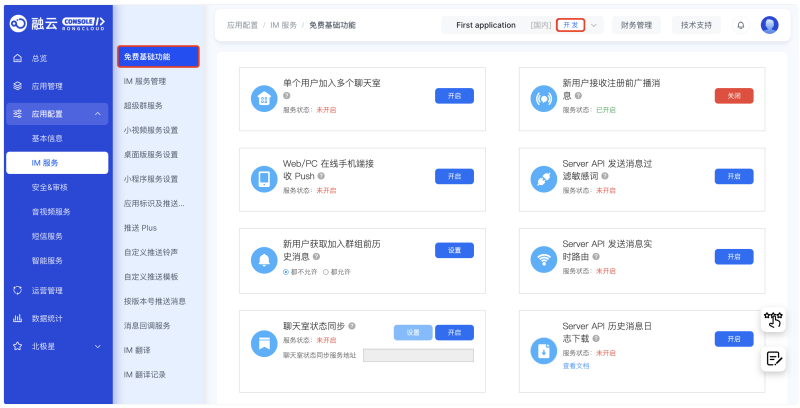
免费基础功能

以下是聊天室业务提供的免费基础功能：

- **单个用户加入多个聊天室**：默认一个用户只能加入一个聊天室中，开启后一个用户可以同时加入到多个聊天室中。
- **聊天室状态同步**：聊天室状态同步是融云提供的服务端回调服务，需同时提供可正常访问的回调地址。配置成功后，在应用下聊天室发生状态变化时，将实时同步到开发者的应用服务器地址，目前支持的同步状态包括：创建、销毁、成员加入、成员退出聊天室。详见 IM 服务端文档「聊天室管理」下的聊天室状态同步。
- **加入聊天室获取指定消息设置**：默认加入聊天室时可最多获取全部消息类型的最近 50 条消息，开启后可设置指定消息类型获取。
- **聊天室销毁等待时间**：
  1. 可以支持配置不活跃聊天室销毁的等待时间，默认等待时间为1小时，即超过1小时不活跃即被销毁，客户可以按需调整这个时间，最长可设置24小时。
  2. 聊天室销毁时，会向聊天室成员发送聊天室销毁通知，以方便客户可以在聊天室销毁后，在终端自定义一些操作(依赖 5.1.1 及以后版本)。
  3. 聊天室增加 sessionid，在聊天室生存周期内保持不变，聊天室重建后重新生成。用于使用相同聊天室ID，多次开播时，客户端能区分出来。
- **聊天室属性自定义设置**：可在指定聊天室中设置自定义属性，用于语音直播聊天室场景的会场属性同步或狼人杀等卡牌类游戏场景中记录用户的角色和牌局状态等，详见「聊天室业务」下的**聊天室属性管理（KV）**文档。如果 App 业务服务端需要融云提供聊天室属性变更数据同步，需要提供可正常访问的回调地址，配置成功后，自动开启融云提供的服务端回调服务，详见 IM 服务端文档「聊天室管理」下的聊天室属性同步。

修改服务配置

访问控制台免费基础功能页面，可调整聊天室业务相关的免费基础功能配置。



IM 旗舰版/尊享版功能

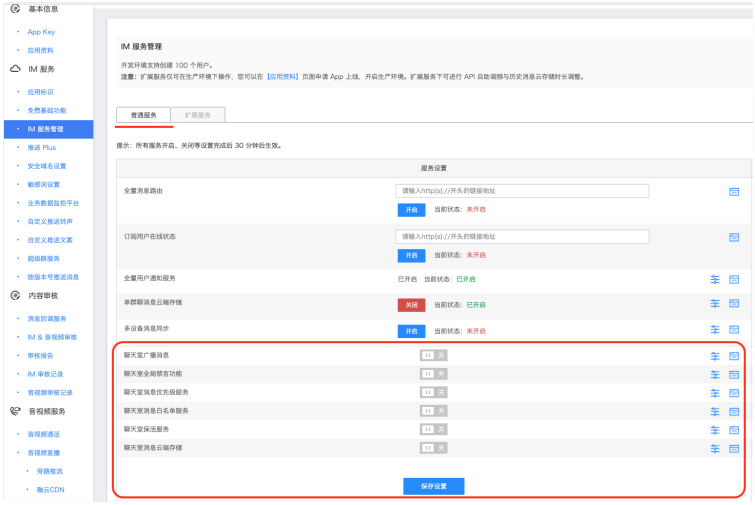
下图显示了控制台 IM 服务管理 页面与聊天室业务相关的**普通服务**配置。

> **开发环境下可以免费使用。生产环境下，IM 旗舰版或 IM 尊享版才能使用以下服务。**

- **聊天室广播消息**：向应用中的所有聊天室发送一条消息，单条消息最大 128k。详见 IM 服务端文档「消息管理」下的发送全体聊天室广播消息。
- **聊天室全局禁言功能**：当不想让某一用户在所有聊天室中发言时，可将此用户添加到聊天室全局禁言中，被禁言用户可接收查看聊天室中用户聊天信息，但不能发送消息。详见 IM 服务端文档「聊天室用户管理」下的全局禁言用户。
- **聊天室消息优先级服务**：在指定聊天室中设置指定类型的消息为低级别消息。当服务器负载高时低级别消息优先被丢弃，这样可以确保重要的消息不被丢弃。详见 IM 服务端文档「聊天室消息优先级服务」下的添加低级别消息。
- **聊天室白名单服务**：开通后，可以使用以下功能对应的 Server API：
  - 聊天室用户白名单：可用于保护指定聊天室中的重要用户，支持按聊天室设置白名单用户。例如，App 业务中指定聊天室中的管理员、主播等重要角色的用户。
  - 聊天室消息白名单：可用于保护 App 下所有聊天室中的指定消息类型。例如 App 业务中自定义的红包消息。
- **聊天室保活服务**：当聊天室中 1 小时无人说话，同时没有人加入聊天室时，融云服务端会自动把聊天室内所有成员踢出聊天室并销毁聊天室。保活的聊天室不会被自动销毁，可以调用 API 接口销毁聊天室。详见 IM 服务端文档「聊天室管理」下的保活聊天室。
- **聊天室消息云端存储**：聊天消息保存在云端，用户进入聊天室后，可以查看聊天室中以前的消息，历史消息默认保存 2 个月。
- **加入聊天室获取指定消息配置**：加入聊天室时只返回指定类型的消息，不返回其他类型的消息。

修改服务配置

访问开发后台 IM 服务管理页面，切换到**普通服务**标签下，可启用以下聊天室服务配置开关。



您还可以在**扩展服务**标签下对部分服务的具体配置进行调整。

查询聊天室房间信息 查询聊天室房间信息

更新时间:2024-08-30

获取聊天室的信息，可返回以下数据：

- 聊天室成员总数
- 指定数量（最多 20 个）的聊天室成员的列表，包括该成员的用户 ID 以及加入聊天室的时间

> 提示
>
> **频率限制：单个设备每秒钟支持调用一次，每分钟单个设备最多调用 20 次。**

您可以使用 RongChatRoomClient 或 RongIMClient 下的 getChatRoomInfo 方法：

```
String chatroomId = "Chatroom Target ID";
int defMemberCount = 10;

RongChatRoomClient.getInstance().getChatRoomInfo(chatroomId, defMemberCount, ChatRoomMemberOrder.RC_CHAT_ROOM_MEMBER_ASC, new
IRongCoreCallback.ResultCallback<ChatRoomInfo>() {

@Override
public void onSuccess(ChatRoomInfo chatRoomInfo) {
// Get ChatRoomInfo properties

String chatRoomId = chatRoomInfo.getChatRoomId();
int totalMemberCount = chatRoomInfo.getTotalMemberCount();

// Get ChatRoomMemberInfo properties
List<ChatRoomMemberInfo> memberInfoList = chatRoomInfo.getMemberInfo();
if (memberInfoList != null) {
for (ChatRoomMemberInfo memberInfo : memberInfoList) {
String MemberId = memberInfo.getUserId();
long JoinTime = memberInfo.getJoinTime();
}
}
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {
// Handle error
}
});
```

参数	类型	说明
chatRoomId	String	聊天室 ID
defMemberCount	int	需要获取的聊天室成员数量。范围 0-20。因为聊天室一般成员数量巨大，权衡效率和用户体验，获取的聊天室成员数上限为 20。如果 count 为 0，则返回的聊天室信息仅包含成员总数，不包含具体的成员列表。
order	ChatRoomMemberOrder	按照何种顺序返回聊天室成员信息。RC_CHAT_ROOM_MEMBER_ASC（升序），表示从最早加入聊天室成员开始，按加入时间递增的顺序获取，返回最早加入的成员列表。RC_CHAT_ROOM_MEMBER_DESC（降序），表示从最晚加入聊天室成员开始，按加入时间递减的顺序获取，返回最晚加入的成员列表。
callback	ResultCallback<ChatRoomInfo>	回调接口。在成功回调中返回 ChatRoomInfo，其中包含按要求获取的聊天室成员列表结果。列表元素为聊天室成员对象 ChatRoomMemberInfo，内部包含用户 ID（userId）和Unix 时间戳格式的加入时间（joinTime），单位为毫秒。 列表中的成员按加入时间从旧到新排列。

加入聊天室　　加入聊天室

更新时间:2024-08-30

加入聊天室分为以下几种情况：

- （推荐）应用程序后端通过即时通讯服务端 API 创建聊天室，将聊天室 ID 下分发至客户端。客户端获取聊天室 ID 后可加入聊天室。
- （已废弃）应用程序通过直接调用客户端 SDK 加入聊天室方法，创建并加入该聊天室。这种方式已不推荐，相关 API 已于 5.6.3 版本废弃，未来可能会删除。
- SDK 断网重连后会自动重新加入聊天室，不需要应用程序处理。

应用程序后端可以通过提前注册服务端回调聊天室状态同步，收取聊天室创建成功与成员加入等事件通知。客户端可以通过监听聊天室事件收取相关通知。

局限

- 默认同一用户不能同时加入多个聊天室。用户加入新的聊天室后会自动退出之前的聊天室。您可以在控制台启用单个用户加入多个聊天室。
- 客户端的加入聊天室方法允许在加入时获取最新的历史消息（默认 10 条，最多 50 条），但不支持指定消息类型。您可以在控制台启用加入聊天室获取指定消息配置，限制加入聊天室时只获取指定类型的消息。

加入已存在的聊天室

> 提示
>
> IMLib 从 5.6.3 开始新增重载方法，支持在回调中返回 JoinChatRoomResponse，其中包含聊天室的创建时间、成员数量、聊天室禁言状态、用户禁言状态等信息。

如果 IMLib 版本 ≥ 5.6.3，可使用 RongChatRoomClient 的 joinExistChatRoom 方法加入一个已存在的聊天室。

```
String chatroomId = "聊天室 ID";
int defMessageCount = 50;

RongChatRoomClient.getInstance().joinExistChatRoom(chatRoomId, defMessageCount, new RongCoreCallback.ResultCallback<JoinChatRoomResponse>() {
@Override
public void onSuccess(JoinChatRoomResponse response) {
// 处理成功加入聊天室的情况
System.out.println("成功加入聊天室");

// 解析每个返回值
long createTime = response.getCreateTime(); // 聊天室创建时间（毫秒时间戳）
int memberCount = response.getMemberCount(); // 成员数量
boolean isAllChatRoomBanned = response.isAllChatRoomBanned(); // 是否全局禁言
boolean isCurrentUserBanned = response.isCurrentUserBanned(); // 当前用户是否被禁言
boolean isCurrentChatRoomBanned = response.isCurrentChatRoomBanned(); // 当前用户是否在此聊天室被禁言
boolean isCurrentChatRoomInWhitelist = response.isCurrentChatRoomInWhitelist(); // 当前用户是否在此聊天室的白名单中

// 可以根据需要对解析的值进行进一步处理
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {
// 处理加入聊天室错误的情况
System.out.println("加入聊天室失败。错误代码: " + e.getValue());
// 可以根据错误代码进行适当的处理和向用户提供反馈
}
});
```

加入成功的回调中会返回 JoinChatRoomResponse，其中包含聊天室的创建时间、成员数量、聊天室禁言状态、用户禁言状态等信息。

参数	类型	说明
chatRoomId	String	聊天室会话 ID，最大长度为 64 个字符。
defMessageCount	int	进入聊天室时获取历史消息的数量，数量范围：1-50。如果传 `-1`，表示不获取任何历史消息。如果传 `0`，表示使用 SDK 默认设置（默认为获取 10 条）。
callback	ResultCallback<JoinChatRoomResponse>	回调接口。

如果 IMLib 版本 < 5.6.3，joinExistChatRoom 方法的成功回调中不携带额外信息。该方法在 5.6.3 版本上已废弃。

```
String chatroomId = "聊天室 ID";
int defMessageCount = 50;

RongChatRoomClient.getInstance().joinExistChatRoom(chatroomId, defMessageCount, new IRongCoreCallback.OperationCallback() {

@Override
public void onSuccess() {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

加入聊天室

> 提示
>
> IMLib 从 5.6.3 开始废弃该方法。

joinChatRoom 接口会创建并加入聊天室。如果聊天室已存在，则直接加入。如果您使用了服务端回调聊天室状态同步，融云会将聊天室创建成功的通知发送到您指定的服务器地址。

```
String chatroomId = "聊天室 ID";
int defMessageCount = 50;

RongChatRoomClient.getInstance().joinChatRoom(chatroomId, defMessageCount, new IRongCoreCallback.OperationCallback() {

@Override
public void onSuccess() {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

参数	类型	说明
chatRoomId	String	聊天室会话 ID，最大长度为 64 个字符。
defMessageCount	int	进入聊天室时获取历史消息的数量，数量范围：1-50。如果传 `-1`，表示不获取任何历史消息。如果传 `0`，表示使用 SDK 默认设置（默认为获取 10 条）。
callback	OperationCallback	回调接口

断线重连后重新加入聊天室

SDK 具备断线重连机制。重连成功后，如果当前登录用户曾经加入过聊天室，且没有退出，则 SDK 会自动重新加入聊天室，不需要 App 处理。App 可以通过监听聊天室状态收到通知。

> 提示
>
> 断网重连的场景下，一旦 SDK 重新加入聊天室成功，会自动收取一定数量的聊天室消息。
>
> - 如果 SDK 版本 < 5.3.1，SDK 不会拉取消息。
> - 如果 SDK 版本 ≧ 5.3.1，SDK 会按照加入聊天室时传入的 `defMessageCount` 拉取固定数量的消息。拉取的消息有可能在本地已存在，App 可能需要进行排重后再显示。

获取聊天室历史消息　　获取聊天室历史消息

更新时间:2024-08-30

客户端 SDK 支持获取聊天室历史消息。具体能力如下：

- 通过 getHistoryMessages 获取聊天室保存在本地的历史消息。请注意，聊天室业务默认在用户退出聊天室时会清除聊天室保存在本地的历史消息。
- 通过 getChatroomHistoryMessages 获取聊天室保存在服务端的历史消息。请注意，该功能依赖聊天室消息云端存储服务。

开通服务

使用 getChatroomHistoryMessages 要求开通 聊天室消息云端存储服务。使用前请确认已开通服务。开通后聊天室历史消息保存在云端，默认保存 2 个月。

获取聊天室远端历史消息

您可以通过 getChatroomHistoryMessages 获取聊天室远端历史记录。如果本地数据库存在相同时段内的历史消息，那么调用这个接口会返回一个 size 为 0 的数组。因此建议的调用顺序如下：

1. 先调用 RongCoreClient 的 getHistoryMessages，传入会话类型，聊天室 ID、最后一条消息的 ID，并指定要获取的消息数量等参数，从本地消息数据库中获取的历史消息。
2. 如果为空的话，再调用 RongChatRoomClient 的 getChatroomHistoryMessages 方法，传入聊天室 ID、消息的发送时间戳、查询方向，并指定要获取的消息数量等参数，从聊天室历史消息云存储中获取历史消息。在查询方向不变的情况下，当次返回的 `syncTime` 的值可以作为下次拉取时的 `recordTime` 传入，方便连续拉取。

```
Conversation.ConversationType conversationType = Conversation.ConversationType.CHATROOM;
int oldestMessageId = -1;
final int count = 10;

final String targetId = "聊天室 ID";
final long recordTime = 0;

RongCoreClient.getInstance().getHistoryMessages(conversationType, targetId, oldestMessageId, count,
new IRongCoreCallback.ResultCallback<List<Message>>() {

@Override
public void onSuccess(List<Message> messages) {
if (messages == null || messages.isEmpty()) {
RongChatRoomClient.getInstance().getChatroomHistoryMessages(targetId, recordTime, count, IRongCoreEnum.TimestampOrder.RC_TIMESTAMP_ASC,
new IRongCoreCallback.IChatRoomHistoryMessageCallback() {

@Override
public void onSuccess(List<Message> messages, long syncTime) {

}


@Override
public void onError(IRongCoreEnum.CoreErrorCode code) {

}
});

}
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {

}
});
```

getChatroomHistoryMessages 的成功回调中会返回一个 `syncTime`。`syncTime` 的具体意义与 `getChatroomHistoryMessages` 的 `order` 参数取值有关。

- 如果拉取顺序为 `RC_Timestamp_Desc`，表示查询方向为降序，即按消息发送时间递减的顺序，获取发送时间早于 `recordTime` 的消息。`syncTime` 为结果中最早一条消息的时间戳（即最小的时间戳）。
- 如果拉取顺序为 `RC_Timestamp_Asc`，表示查询方向为升序，即按消息发送时间递增的顺序，获取发送时间晚于 `recordTime` 的消息。`syncTime` 为结中最晚消息的时间戳（即最大的时间戳）。

监听聊天室事件　　监听聊天室事件

更新时间:2024-08-30

聊天室业务为客户端 App 提供三种类型的事件监听器：聊天室状态监听器、聊天室成员变化监听器、聊天室事件通知监听器。

通过 SDK 提供的以上监听器，客户端 App 可以从融云服务端获取聊天室销毁状态、用户在当前及其他客户端加入退出聊天室的状态、聊天室中成员进出的事件通知、以及聊天室中成员禁言、封禁相关的信息。

监听器	名称	说明
ChatRoomAdvancedActionListener	聊天室状态监听器	接收在当前客户端登录的用户加入、退出聊天室的事件与聊天室销毁状态的信息。
ChatRoomMemberActionListener	聊天室成员变化监听器	接收当前用户所在聊天室中其他用户加入、退出聊天室的信息。
ChatRoomNotifyEventListener	聊天室事件通知监听器	接收当前所在聊天室中成员禁言、封禁相关的信息；接收当前用户在其他端加入、退出聊天室相关的信息。

监听聊天室状态

应用程序可以监听当前客户端上用户加入、退出聊天室的事件与聊天室销毁状态变更。

使用 RongChatRoomClient 的 addChatRoomAdvanceActionListener setChatRoomAdvancedActionListener 添加或设置 ChatRoomAdvancedActionListener 监听器。

```
// addChatRoomAdvanceActionListener，since 5.2.5
RongChatRoomClient.addChatRoomAdvanceActionListener(
new ChatRoomAdvancedActionListener() {
@Override
public void onJoining(String chatRoomId) {
// default implementation ignored
}

@Override
public void onJoined(String chatRoomId) {
// 加入聊天室成功，已废弃
// default implementation ignored
}

@Override
public void onJoined(String chatRoomId, JoinChatRoomResponse joinChatRoomResponse) {
// 加入聊天室成功 since 5.6.3
// default implementation ignored
}

@Override
public void onReset(String chatRoomId) {
// default implementation ignored
}

@Override
public void onQuited(String chatRoomId) {
// default implementation ignored
}

@Override
public void onDestroyed(String chatRoomId, ChatRoomDestroyType type) {
data.setFirstAndNotify(Pair.create(chatRoomId, type));
}

@Override
public void onError(String chatRoomId, CoreErrorCode code) {
// default implementation ignored
}
});


// setChatRoomAdvanceActionListener
RongChatRoomClient.setChatRoomAdvanceActionListener(listener);
```

移除监听器：

```
// removeChatRoomAdvanceActionListener，since 5.2.5
RongChatRoomClient.addChatRoomAdvanceActionListener(listener);
// setChatRoomAdvanceActionListener
RongChatRoomClient.setChatRoomAdvanceActionListener(null);
```

聊天室状态事件

下方列出了聊天室操作监听器 ChatRoomAdvancedActionListener 提供的事件列表及回调方法。

回调方法	触发时机	说明
onJoining(String chatRoomId)	当前用户正在加入聊天室	返回数据包括：聊天室 ID
onJoined(String chatRoomId, JoinChatRoomResponse joinChatRoomResponse)	当前用户已加入聊天室（5.6.3 开始提供）。	返回数据包括：聊天室 ID，聊天室房间信息、用户在聊天室的状态。

回调方法	触发时机	说明
onJoined(String chatRoomId)	当前用户已加入聊天室（从 5.6.3 开始废弃）	返回数据包括：聊天室 ID
onReset(String chatRoomId)	用户所在聊天室被重置。	返回数据包括：聊天室 ID。注意，加入聊天室成功，但是聊天室被重置，接收到此回调后，还会立刻收到 onJoining 和 onJoined 回调。
onQuited(String chatRoomId)	当前用户退出聊天室	返回数据包括：聊天室 ID
onDestroyed(String chatRoomId, IRongCoreEnum.ChatRoomDestroyType type)	当前用户在线，且所在聊天室被销毁	返回数据包括：聊天室 ID、聊天室销毁类型。聊天室销毁类型为 ChatRoomDestroyType.MANUAL，表示主动调用 IM Server API 销毁聊天室。ChatRoomDestroyType.AUTO 表示聊天室自动销毁。
onError(String chatRoomId, IRongCoreEnum.CoreErrorCode code)	聊天室操作异常	返回数据包括：聊天室 ID、错误码。

监听聊天室成员变化

> 提示
>
> SDK 从 5.1.4 版本开始支持监听聊天室成员变化。

从 5.1.4 版本，SDK 开始在 RongChatRoomClient 中提供 ChatRoomMemberActionListener，支持当前登录用户监听所在聊天室中其它成员的加入、退出行为。

开通服务

此功能需要开通服务后方可使用。如有需求，请提交工单，申请开通聊天室成员变化监听。

设置聊天室成员变化监听器

设置聊天室成员变化监听器 ChatRoomMemberActionListener。从 5.6.7 版本 开始，新增返回 ChatRoomMemberActionModel 对象的回调方法，其中增加了当前聊天室人数。

```
RongChatRoomClient.setChatRoomMemberListener(new RongChatRoomClient.ChatRoomMemberActionListener() {
/**
* @param chatRoomMemberActions
* @param roomId
*/
@Override
public void onMemberChange(List<ChatRoomMemberAction> chatRoomMemberActions, String chatRoomId) {
// Handle the member change with the list of ChatRoomMemberAction objects and the chatRoomId
}

/**
* @since 5.6.7
*/
@Override
public void onMemberChange(ChatRoomMemberActionModel model) {
// Handle the member change with the ChatRoomMemberActionModel object
String chatroomId = model.getRoomId();
List<ChatRoomMemberAction> chatRoomMemberActions = model.getChatRoomMemberActions();
int memberCount = model.getMemberCount();

for (ChatRoomMemberAction action : chatRoomMemberActions) {
// Access the properties of ChatRoomMemberAction object
String userId = action.getUserId();
ChatRoomMemberAction.ChatRoomMemberActionType actionType = action.getChatRoomMemberAction();

if (actionType == ChatRoomMemberAction.ChatRoomMemberActionType.CHAT_ROOM_MEMBER_JOIN) {
System.out.println("Action Type: Join");
} else if (actionType == ChatRoomMemberAction.ChatRoomMemberActionType.CHAT_ROOM_MEMBER_QUIT) {
System.out.println("Action Type: Leave");
}
}
}
});
```

聊天室中有其他用户加入或退出时，SDK 会同时触发两个 onMemberChange 回调方法，都在 UI 线程回调。ChatRoomMemberAction 列表中包含了当前聊天室中加入或退出聊天室的成员信息。ChatRoomMemberActionModel 额外封装了聊天室当前人数（指已加入未退出的人数）。推荐使用返回 ChatRoomMemberActionModel 对象的 onMemberChange 方法。

如果当前用户由于网络原因和聊天室断开连接，则无法监听到断开连接期间的其他成员的加入、退出行为。

监听聊天室事件通知

> 提示
>
> SDK 从 5.4.5 版本开始支持监听聊天室事件通知。

应用程序可以监听当前聊天室中成员禁言、封禁相关的通知，以及当前用户在其他端加入、退出聊天室相关的通知。

使用 RongChatRoomClient 的 addChatRoomNotifyEventListener 方法添加一个ChatRoomNotifyEventListener 监听器。SDK 从 5.4.5 版本开始支持 RCChatRoomNotifyEventListener 监听器。

```
// Since 5.4.5
RongChatRoomClient.addChatRoomNotifyEventListener(
new RongChatRoomClient.ChatRoomNotifyEventListener() {
@Override
public void onChatRoomNotifyMultiLoginSync(ChatRoomSyncEvent event) {

}

@Override
public void onChatRoomNotifyBlock(ChatRoomMemberBlockEvent event) {

}

@Override
public void onChatRoomNotifyBan(ChatRoomMemberBanEvent event) {

}
});

// Remove
RongChatRoomClient.removeChatRoomNotifyEventListener(listener);
```

回调方法	触发时机	说明
onChatRoomNotifyMultiLoginSync(ChatRoomSyncEvent event)	在用户在多端登录情况下，在其他客户端加入、退出聊天室	接收在当前客户端上用户加入、退出聊天室的事件
onChatRoomNotifyBlock(ChatRoomMemberBlockEvent event)	用户所在聊天室中发生了禁言、解除禁言相关的事件	是否接收通知取决于调用服务端封禁、解封的相关 API 时是否指定了 needNotify 为 true。
onChatRoomNotifyBan(ChatRoomMemberBanEvent event)	用户所在聊天室发生了封禁用户、解除封禁相关的事件	是否接收通知取决于调用服务端封禁、解封的相关 API 时是否指定了 needNotify 为 true。

多端登录事件通知

在用户在多端登录情况下，在其他客户端加入、退出聊天室时，触发 onChatRoomNotifyMultiLoginSync(ChatRoomSyncEvent event) 方法。

触发场景	通知范围	说明
用户多端登录，在一台设备上加入聊天室	当前加入的用户	当前用户在一端加入聊天室时，其他端的在线设备上会接收通知，不在线的设备不会通知。返回数据包括：聊天室 ID、加入时间。
用户多端登录，在一台设备上退出聊天室	当前退出的用户	当前用户在一端退出聊天室时，其他端的在线设备上会接收通知，不在线的设备不会通知。 返回数据包括：聊天室 ID、退出时间。
用户多端登录且已在聊天室中，加入一个新的聊天室导致被踢出上一个聊天室	当前用户，和被踢出的聊天室所有成员	如果在控制台开通了单个用户加入多个聊天室，则不会通知。返回数据包括：聊天室 ID、被踢出的时间。

ChatRoomSyncEvent 具体属性定义如下：

属性	类型	说明
chatroomId	String	聊天室 ID。
status	ChatRoomSyncStatus	变更的状态，Quit(0)：离开。Join(1)：加入。
reason	ChatRoomSyncStatusReason	如果 status 是 Quit 的情况，区分离开类型：LeaveOnMyOwn(1) 表示自己主动离开。OtherDeviceLogin(2) 表示多端加入互踢导致离开。如果 status 为 Join 的情况，无需关心 reason 值。
time	long	用户加入/退出/被踢的时间戳，精确到毫秒。
extra	String	附加信息。

封禁相关事件通知

用户所在聊天室发生了封禁、解封的事件，且调用相关 Server API 时指定了需要通知（指定 needNotify 为 true），触发 onChatRoomNotifyBlock(ChatRoomMemberBlockEvent event) 方法。

触发场景	通知范围	说明
封禁聊天室用户	聊天室中所有成员	返回数据包括：聊天室 ID、被封禁成员用户 ID 列表、封禁时间与持续时长、附加信息。
解除封禁聊天室用户	被解除封禁的成员	返回数据包括：聊天室 ID、当前用户 ID、附加信息。

ChatRoomMemberBlockEvent 具体属性定义如下：

属性	类型	说明
chatroomId	String	聊天室 ID。
operateType	ChatRoomOperateType	封禁类型。Deblock(0)：解封。Blocked(1)：封禁。
durationTime	long	封禁时长，单位为毫秒，最大值为 43200分钟（1个月），最小值 1分钟。 operateType 为 Blocked 时，该字段有效。
operateTime	long	操作时间戳（毫秒）。
userIdList	List<String>	被封禁的用户 ID 列表。解封时为当前用户 ID。
extra	String	附加信息。

禁言相关事件通知

用户所在聊天室发生了禁言、解除禁言相关的事件，且调用相关 Server API 时指定了需要通知（指定 needNotify 为 true），触发 onChatRoomNotifyBan(ChatRoomMemberBanEvent event) 方法。

触发场景	通知范围	说明
禁言指定聊天室用户	聊天室中所有成员	返回数据包括：聊天室 ID、被封禁成员用户 ID 列表、禁言时间与持续时长、附加信息。

触发场景	通知范围	说明
取消禁言指定聊天室用户	聊天室中所有成员	返回数据包括：聊天室 ID、被封禁成员用户 ID 列表、解除禁言时间、附加信息。
设置聊天室全体禁言	聊天室中所有成员	返回数据包括：聊天室 ID、禁言时间、附加信息。
取消聊天室全体禁言	聊天室中所有成员	返回数据包括：聊天室 ID、解除禁言时间、附加信息。
加入聊天室全体禁言白名单	聊天室中所有成员	返回数据包括：聊天室 ID、被添加白名单的用户 ID 列表、设置白名单时间、附加信息。
移出聊天室全体禁言白名单	聊天室中所有成员	返回数据包括：聊天室 ID、被移出白名单的用户 ID 列表、移出白名单时间、附加信息。
全局禁言用户	被全局禁言的用户	返回数据包括：聊天室 ID、禁言时间与持续时长、附加信息。
取消全局禁言用户	被解除全局禁言的用户	返回数据包括：聊天室 ID、解除禁言时间、附加信息。

`ChatRoomMemberBanEvent` 具体属性定义如下：

属性	类型	说明
`chatroomId`	String	聊天室 ID。
`banType`	ChatRoomMemberBanType	聊天室禁言/解除禁言操作类型枚举
`durationTime`	long	禁言时长，单位为毫秒：最大值为 43200 分钟（1个月），最小值1分钟。 `banType` 为禁言类型时，该字段有效。
`operateTime`	long	操作时间戳（毫秒）。
`userIdList`	List<String>	被封禁的用户 ID 列表。解封时为当前用户 ID。
`extra`	String	附加信息。

聊天室属性管理（KV）　　聊天室属性管理（KV）

更新时间:2024-08-30

SDK 在聊天室业务核心类 RongChatRoomClient 中提供了聊天室属性（KV）管理接口（也可以使用 RongIMClient），用于在指定聊天室中设置自定义属性。

在语音直播聊天室场景中，可利用此功能记录聊天室中各麦位的属性；或在狼人杀等卡牌类游戏场景中记录用户的角色和牌局状态等。

开通服务

使用聊天室属性（KV）接口要求开通**聊天室属性自定义设置**服务。您可以前往控制台的免费基础功能页面开启服务。

聊天室业务还提供服务端回调功能，支持由融云服务端将应用下的全部聊天室属性变化（设置，删除，全部删除等操作）同步到您指定的地址，方便 App 业务服务端了解聊天室属性变化。详见服务端文档聊天室属性同步（KV）。

功能局限

提示

- 聊天室销毁后，聊天室中的自定义属性同时销毁。
- 每个聊天室中，最多允许设置 **100** 个属性信息，以 `Key-Value` 的方式进行存储。
- 客户端 SDK 未针对聊天室属性 KV 的操作频率进行限制。建议每个聊天室，每秒钟操作 `Key-Value` 频率保持在 **100** 次及以下（一秒内单次操作 100 个 KV 等同于操作 100 次）。

监听聊天室属性变化

SDK 在聊天室业务核心类 RongChatRoomClient 中提供了 KVStatusListener，用于监听聊天室属性（KV）变化。您也可以使用 RongIMClient 下的同名接口类。

返回值	方法	说明
void	onChatRoomKVSync(String roomId)	聊天室 KV 列表同步完成时触发。
void	onChatRoomKVUpdate(String roomId, Map<String, String> chatRoomKvMap)	聊天室 KV 列表更新完成时触发，首次同步 KV 时返回全量 KV，后续触发时仅返回新增、修改的 KV。
void	onChatRoomKVRemove(String roomId, Map<String, String> chatRoomKvMap)	KV 被删除时触发。

- `onChatRoomKVSync(String roomId)`

  触发时机：加入聊天室成功后，SDK 默认从服务端同步 KV 列表，同步完成后触发。

参数	类型	说明
roomId	String	聊天室 Id

- `onChatRoomKVUpdate(String roomId, Map<String, String> chatRoomKvMap)`

  触发时机：聊天室 KV 更新时。

  如果刚进入聊天室时存在 KV，会通过此回调将所有 KV 返回，再次回调时为其他人设置或者修改 KV 的增量数据。

参数	类型	说明
roomId	String	聊天室 Id
chatRoomKvMap	Map<String, String>	发生变化的聊天室 KV

- `onChatRoomKVRemove(String roomId, Map<String, String> chatRoomKvMap)`

  触发时机：KV 被删除时触发。

参数	类型	说明
roomId	String	聊天室 Id
chatRoomKvMap	Map<String, String>	被删除的聊天室 KV

添加聊天室 KV 监听器

使用 addKVStatusListener 方法，添加聊天室 KV 监听器 KVStatusListener。支持设置多个监听器。为了避免内存泄露，请在不需要监听时将监听器移除。

```
RongChatRoomClient.getInstance().addKVStatusListener(listener);
```

移除聊天室 KV 监听器

SDK 支持移除监听器。为了避免内存泄露，请在不需要监听时将监听器移除。

```
RongChatRoomClient.getInstance().removeKVStatusListener(listener);
```

获取单个属性

通过属性的 Key 获取指定聊天室中的单个属性 KV。

```
String chatRoomId = "聊天室 ID";
String key = "name";

RongChatRoomClient.getInstance().getChatRoomEntry(chatRoomId, key, new IRongCoreCallback.ResultCallback<Map<String, String>>() {
@Override
public void onSuccess(Map<String, String> stringStringMap) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {

}
});
```

参数	类型	说明
chatRoomId	String	聊天室 ID
key	String	聊天室属性名称,Key 支持大小写英文字母、数字、部分特殊符号 + = - _ 的组合方式，最大长度 128 个字符
callback	ResultCallback<Map<String, String>>	回调接口

获取所有属性

获取指定聊天室中所有属性 KV 对。

```
String chatRoomId = "聊天室 ID";

RongChatRoomClient.getInstance().getAllChatRoomEntries(chatRoomId, new IRongCoreCallback.ResultCallback<Map<String, String>>() {
@Override
public void onSuccess(Map<String, String> stringStringMap) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {

}
});
```

参数	类型	说明
chatRoomId	String	聊天室 ID
callback	ResultCallback<Map<String, String>>	回调接口

设置单个属性

在指定单个聊天室中设置单个属性 KV 对。注意，该接口不支持更新已存在的属性。

```
String chatRoomId = "聊天室 ID";
String key = "name";
String value = "融融";
boolean sendNotification = true;
boolean isAutoDel = false;
String notificationExtra = "通知消息扩展";

RongChatRoomClient.getInstance().setChatRoomEntry(chatRoomId, key, value, sendNotification, isAutoDel, notificationExtra, new IRongCoreCallback.OperationCallback() {

@Override
public void onSuccess() {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

在设置 KV 时，可指定 SDK 发送一条聊天室属性通知消息（ChatRoomKVNotiMessage）。消息内容结构说明可参见通知类消息格式。

参数	类型	说明
chatRoomId	String	聊天室 ID
key	String	聊天室属性名称，Key 支持大小写英文字母、数字、部分特殊符号 + = - _ 的组合方式，最大长度 128 个字符

参数	类型	说明
value	String	聊天室属性对应的值，最大长度 4096 个字符
sendNotification	boolean	`true` 发送通知; `false` 不发送。如果发送通知，SDK 会接收到类型标识为 `RC:chrmKVNotiMsg` 的聊天室属性通知消息（`ChatRoomKVNotiMessage`），并且消息内容中包含 K，V
autoDelete	boolean	退出后是否删除
notificationExtra	String	通知的自定义字段，RC:chrmKVNotiMsg(ChatRoomKVNotiMessage), 通知消息中会包含此字段，最大长度 2048 个字符
callback	OperationCallback	回调接口

强制设置单个属性

强制设置聊天室属性。以 `key = value` 的形式存储。当 key 不存在时，代表增加属性。当 key 已经存在时，代表更新属性的值。使用强制设置可修改他人创建的属性值。

```
String chatRoomId = "聊天室 ID";
String key = "name";
String value = "融融";
boolean sendNotification = true;
boolean isAutoDel = false;
String notificationExtra = "通知消息扩展";

RongChatRoomClient.getInstance().forceSetChatRoomEntry(chatRoomId, key, value, sendNotification, isAutoDel, notificationExtra, new IRongCoreCallback.OperationCallback()
{

@Override
public void onSuccess() {

}


@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

在设置 KV 时，可指定 SDK 发送一条聊天室属性通知消息（`ChatRoomKVNotiMessage`）。消息内容结构说明可参见通知类消息格式。

参数	类型	说明
chatRoomId	String	聊天室 ID
key	String	聊天室属性名称,Key 支持大小写英文字母、数字、部分特殊符号 + = - _ 的组合方式，最大长度 128 个字符
value	String	聊天室属性对应的值，最大长度 4096 个字符
sendNotification	boolean	true 发送通知; false 不发送. 如果发送通知，SDK 会接收到 RC:chrmKVNotiMsg 通知消息 ChatRoomKVNotiMessage，并且消息内容中包含 K，V
autoDelete	boolean	退出后是否删除
notificationExtra	String	通知的自定义字段，RC:chrmKVNotiMsg{@link ChatRoomKVNotiMessage} 通知消息中会包含此字段，最大长度 2048 个字符
callback	OperationCallback	回调接口

批量设置属性

批量设置聊天室属性。通过 `isForce` 参数可控制是否强制覆盖他人设置的属性。

```
RongChatRoomClient.getInstance().setChatRoomEntries(chatRoomId, chatRoomEntryMap, isAutoDel, isForce, new IRongCoreCallback.SetChatRoomKVCallback() {
/**
* 成功回调
*/
@Override
public void onSuccess() {

}

/**
* 失败回调
* @param errorCode
* @param map
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode errorCode, Map<String, IRongCoreEnum.CoreErrorCode> map) {

}
});
```

参数	类型	说明
chatRoomId	String	聊天室 ID
chatRoomEntryMap	Map	chatRoomEntryMap集合size最大限制为10,超过限制返回错误码KV_STORE_OUT_OF_LIMIT (23429)}
isAutoDel	boolean	用户掉线或退出时，是否自动删除该 Key、Value 值
isForce	boolean	是否强制覆盖
callback	SetChatRoomKVCallback	回调接口

如果部分失败，SDK 会触发 SetChatRoomKVCallback 的 `onError` 回调方法：

回调参数	回调类型	说明
errorCode	CoreErrorCode	错误码

回调参数	回调类型	说明
map	Map	当 errorCode 为 KV_STORE_NOT_ALL_SUCCESS（23428）时,map 才会有值（key 为设置失败的 key，value 为该 key 对应的错误码）

删除单个属性

删除指定单个聊天室的单个属性。该接口仅支持删除当前用户所创建的属性。

```
String chatRoomId = "聊天室 ID";
String key = "name";
boolean sendNotification = true;
String notificationExtra = "通知消息扩展";

RongChatRoomClient.getInstance().removeChatRoomEntry(chatRoomId, key, sendNotification, notificationExtra, new IRongCoreCallback.OperationCallback() {

@Override
public void onSuccess() {

}


@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

在删除 KV 时，可指定 SDK 发送一条聊天室属性通知消息（ChatRoomKVNotiMessage）。消息内容结构说明可参见通知类消息格式 。

参数	类型	说明
chatRoomId	String	聊天室 ID
key	String	聊天室属性名称,Key 支持大小写英文字母、数字、部分特殊符号 + = - _ 的组合方式，最大长度 128 个字符
sendNotification	boolean	true 发送通知; false 不发送. 如果发送通知，SDK 会接收到 RC:chrmKVNotiMsg 通知消息 ChatRoomKVNotiMessage，并且消息内容中包含 K，V
notificationExtra	String	通知的自定义字段，RC:chrmKVNotiMsg{@link ChatRoomKVNotiMessage} 通知消息中会包含此字段，最大长度 2048 个字符
callback	OperationCallback	回调接口

强制删除单个属性

强制删除指定单个聊天室的单个属性。可删除他人所创建的属性。

```
String chatRoomId = "聊天室 ID";
String key = "name";
boolean sendNotification = true;
String notificationExtra = "通知消息扩展";

RongChatRoomClient.getInstance().forceRemoveChatRoomEntry(chatRoomId, key, sendNotification, notificationExtra, new IRongCoreCallback.OperationCallback() {

@Override
public void onSuccess() {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

在强制删除 KV 时，可指定 SDK 发送一条聊天室属性通知消息（ChatRoomKVNotiMessage）。消息内容结构说明可参见通知类消息格式 。

参数	类型	说明
chatRoomId	String	聊天室 ID
key	String	聊天室属性名称，Key 支持大小写英文字母、数字、部分特殊符号 + = - _ 的组合方式，最大长度 128 个字符。
sendNotification	boolean	true 发送通知; false 不发送. 如果发送通知，SDK 会接收到 RC:chrmKVNotiMsg 通知消息 ChatRoomKVNotiMessage，并且消息内容中包含 KV。
notificationExtra	String	通知的自定义字段，RC:chrmKVNotiMsg{@link ChatRoomKVNotiMessage} 通知消息中会包含此字段，最大长度 2048 个字符。
callback	OperationCallback	回调接口

批量删除属性

从指定单个聊天室中批量删除属性。单次最多删除 10 个属性。通过 isForce 参数可控制是否强制删除他人设置的属性。

```
RongChatRoomClient.getInstance().deleteChatRoomEntries(chatRoomId, keys, isForce, new IRongCoreCallback.SetChatRoomKVCallback() {
/**
* 成功回调
*/
@Override
public void onSuccess() {

}

/**
* 失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode errorCode, Map<String, IRongCoreEnum.CoreErrorCode> map) {

}
});
```

参数	类型	必填	说明
chatRoomId	String	是	聊天室 ID
keys	List	是	聊天室属性名称集合，集合 size 最大限制 10
isForce	boolean	是	是否强制覆盖
callback	SetChatRoomKVCallback	是	回调接口

如果部分失败，SDK 会触发 SetChatRoomKVCallback 的 onError 回调方法：

回调参数	回调类型	说明
errorCode	CoreErrorCode	错误码
map	Map<String, IRongCoreEnum.CoreErrorCode>	当 errorCode 为 KV_STORE_NOT_ALL_SUCCESS（23428）时,map 才会有值（key 为设置失败的 key，value 为该 key 对应的错误码）

绑定音视频房间　　绑定音视频房间

更新时间:2024-08-30

聊天室与音视频房间绑定成功后，只要音视频房间仍存在，则阻止聊天室自动销毁。

适用场景：

聊天室具有自动销毁机制。在使用融云 RTC 业务的 App 中，可能配合使用 IM SDK 的聊天室业务实现直播聊天、弹幕、属性记录等功能。这种情况下，可以考虑将聊天室与音视频房间绑定，确保聊天室不会在语聊、直播结束前销毁，以免丢失关键数据。

在单独使用聊天室业务情况的下，无需调用该接口。

> 提示
>
> **关于聊天室自动销毁逻辑的说明：**
>
> 聊天室具有自动销毁机制，默认情况下所有聊天室会在不活跃（连续时间段内无成员进出且无新消息）达到 1 小时后踢出所有成员并自动销毁，可延长该时间，也可配置为定时自动销毁。详见服务端文档聊天室销毁机制 。

绑定音视频房间

> 提示
>
> - 客户端 SDK 5.2.1 版本开始支持绑定音视频房间接口。
> - 该接口在 `RongChatRoomClient` 中。

绑定音视频房间后，当聊天室达到预设的自动销毁条件时，服务端会先检测已绑定的音视频房间（RTCRoomId）是否仍存在：

- 如果绑定的音视频房间仍存在，则聊天室不会销毁。
- 如果绑定的音视频房间已销毁，则直接销毁聊天室。

该接口仅创建从聊天室到音视频房间的**单向绑定关系**。因此在绑定音视频房间后，音视频房间的主动销毁或自动销毁，并不会直接触发聊天室房间的销毁。关于音视频房间的销毁机制，详见音视频房间销毁机制 。

接口说明

调用该接口必须传入聊天室 ID，因此必须在聊天室房间已创建成功之后调用。客户端调用加入聊天室接口时会自动完成创建与加入动作。

该接口不具备用户权限控制功能，建议由业务侧的房主或者主播角色用户加入聊天室房间成功后调用一次，其他用户无需调用。

```
public abstract void bindChatRoomWithRTCRoom(
String chatRoomId, String rtcRoomId, IRongCoreCallback.OperationCallback callback)
```

参数说明

参数	类型	说明
chatRoomId	String	聊天室 ID，非空。聊天室 ID 必须已存在。需要 App 传入。
rtcRoomId	String	音视频房间 ID，非空。需要 App 传入。
callback	IRongCoreCallback.OperationCallback	事件监听回调

退出聊天室　　退出聊天室

更新时间:2024-08-30

退出聊天室支持以下几种情况：

- **被动退出聊天室**：聊天室具有离线成员自动踢出机制。该机制被触发时，融云服务端会将用户踢出聊天室。用户如被封禁，也会被踢出聊天室。
- **主动退出聊天室**：客户端提供 API，支持由用户主动退出聊天室。

聊天室离线成员自动退出机制

聊天室具有离线成员自动退出机制。用户离线后，如满足以下默认预设条件，融云服务端会自动将该用户踢出聊天室：

- 从用户离线开始 30 秒内，聊天室中产生第 31 条消息时，触发自动踢出。
- 或用户已离线 30 秒后，聊天室有新消息产生时，触发自动踢出。

> 提示
>
> - 默认预设条件均要求聊天室中必须要有新消息产生，否则无法触发踢出动作。如果聊天室中没有消息产生，则无法将异常用户踢出聊天室。
> - 如需修改默认行为对新消息的依赖，请提交工单申请开通**聊天室成员异常掉线实时踢出**。开通该服务后，服务端会通过 SDK 行为（要求 Android/iOS IMLib SDK 版本 ≧ 5.1.6，Web IMLib 版本 ≧ 5.3.2）判断用户是否处于异常状态，最迟 5 分钟可以将异常用户踢出聊天室。
> - 如需保护特定用户，即不自动踢出指定用户（如某些应用场景下可能希望用户驻留聊天室），可使用 Server API 提供的聊天室用户白名单功能。

主动退出聊天室

客户端用户可主动退出聊天室。

```
String chatroomId = "聊天室 ID";

RongChatRoomClient.getInstance().quitChatRoom(chatroomId, new IRongCoreCallback.OperationCallback() {

public void onSuccess() {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

参数	类型	说明
chatRoomId	String	聊天室 ID
callback	OperationCallback	回调接口

超级群概述 超级群概述

更新时间:2024-08-30

- 客户端 IMLib SDK 从 SDK 5.2.0 开始支持超级群。IMKit 暂不支持超级群业务。
- 除部分接口另有说明外，超级群 Android 客户端接口大部分均在 ChannelClient 中。

融云超级群（UltraGroup）提供了一种新的群组业务形态。超级群不设置群成员人数上限，允许用户在超级社群中建立社交关系、在海量信息中聚焦自己感兴趣的内容，帮助开发者打造高用户黏性的群体。超级群组成员最多可加入 100 个超级群，每个超级群下的不同频道之间共享一份超级群成员关系。App 内的超级群数量没有限制。

超级群业务的会话类型（ConversationType）为 ConversationType.ULTRA_GROUP，用 targetId 表示超级群 ID，channelId 表示超级群频道 ID。除部分接口另有说明外，超级群 Android 客户端接口大部分均在 ChannelClient 中。

开通服务

超级群功能需要在控制台超级群服务页面开通。仅 **IM 尊享版**支持开通超级群服务。具体功能与费用以融云官方价格说明页面及计费说明文档为准。

如何使用频道

超级群支持在群会话中创建独立的频道（客户端由 channelId 指定、对应服务端的 busChannel），超级群的会话、消息、未读数等消息数据和群组成员支持分频道进行聚合，各个频道之间消息独立。

频道按类型区分为公有频道与私有频道。公有频道对所有超级群成员开放（无需加入）。该超级群的所有成员都会接收公有频道下的消息。私有频道仅对该频道成员列表上的用户开放。有关私有频道的详细介绍，可参见超级群私有频道概述。

超级群业务提供一个 ID 为 RCDefault 的默认频道。RCDefault 频道对所有超级群成员开放，不可转为私有频道。

对于 App 业务来说，如果仅需实现类似群聊的业务，可以利用超级群无成员上限的特性构建大于 3000 人的超大群。这种场景下，可以让所有消息都在 RCDefault 默认频道中进行收发。建议在调用客户端、服务端 API 时指定频道 ID 为 RCDefault。

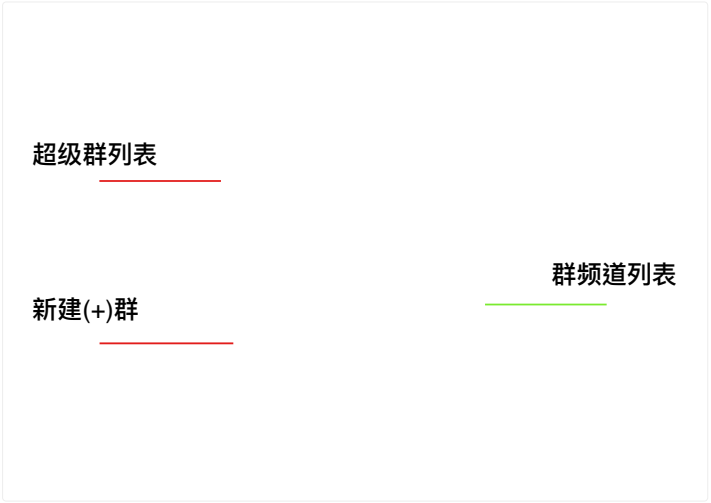
如果仅需实现类似 Discord 类业务，通过超级群频道功能构建子社区，推荐全部使用您自行创建的频道实现您的业务特性。默认频道（RCDefault）与自建频道的行为存在差异，全部使用自建频道可避免这种差异在实现 App 业务逻辑时造成限制。

> 提示
>
> 如果您的 App / 环境在 2022.10.13 日之前开通超级群服务，则您的超级群服务中不存在 RCDefault 频道。在调用客户端、服务端 API 时如果不指定频道 ID，一般仅作用于不属于任何频道的消息，具体行为需参见各功能文档。融云支持客户调整服务至最新行为。该行为调整将影响客户端、服务端收发消息、获取会话、清除历史消息、禁言等多个功能。如有需要，请提交工单咨询详细方案。

客户端 UI 框架参考设计

超级群产品暂不提供 UI 组件。您可以参考以下 UI 框架设计了解超级群 App 的设计思路。

- 下图左侧红框中为超级群列表，即当前登录用户的超级群列表。红框底部的加号（+）按钮代表新建超级群。
- 上图绿框中为超级群频道（Channel）列表。超级群的每一个频道由频道 ID（`channelId`）指定。



超级群管理接口

融云不会托管用户，也不管理群组的业务逻辑，因此超级群的业务逻辑全部需要在 App 服务器进行实现。

对于客户端开发人员来说，创建群组、频道等基础管理操作只需要与 App 服务端交互即可。App 服务端需要调用相应的融云服务端 API（Server API）接口相关接口创建超级群、创建频道等其他管理操作。

下表列出了融云提供的超级群基础管理接口。

> 提示
>
> Server API 还提供超级群全体禁言、超级群用户禁言等更多管理接口。具体请参见 融云服务端超级群文档 。

功能分类	功能描述	客户端 API	融云服务端 API
创建、解散超级群	提供创建者用户 ID、超级群 ID、和群名称，向融云服务端申请建群。如解散超级群，则群成员关系不复存在。	不提供该 API	创建超级群 、解散超级群
加入、退出超级群	加入超级群后，默认可查看入群以后产生的新消息。退出超级群后，不再接受该群的新消息。	不提供该 API	加入超级群 、退出超级群
修改融云服务端的超级群信息	修改在融云推送服务中使用的超级群信息。	不提供该 API	更新超级群信息
创建、删除群频道	在超级群会话中创建独立沟通的频道。如删除频道，将无法在频道中发送消息。	不提供该 API	创建频道 、删除频道
查询群频道列表	加入超级群后，默认可查看入群以后产生的新消息。退出超级群后，不再接受该群的新消息。	不提供该 API	查询频道列表
变更群频道类型	超级群频道可以随时切换为公有频道或私有频道。	不提供该 API	变更频道类型
添加、删除私有频道成员	将超级群成员加入或移出指定频道的私有频道成员列表。在频道类型为私有频道时启用该成员列表的数据。	不提供该 API	添加私有频道成员 、删除私有频道成员

私有频道概述 私有频道概述

更新时间:2024-08-30

超级群服务支持创建私有频道，满足社区场景中只有指定用户可以在频道中沟通的业务需求。

在使用私有频道功能前，建议先阅读超级群概述，了解在 App 业务中**如何使用频道**。

已知限制

> 提示
>
> **私有频道功能目前有以下限制：**
>
> **用户即使不在私有频道成员列表中，发送消息会失败，但消息仍会进入本地数据库。如果用户未在私有频道成员列表中，但仍往频道中发送消息，此时消息无法成功发送到服务端，但会进入本地数据库，并可生成会话。**
>
> **建议解决方案：开发者在 App 业务侧维护一份私有频道成员列表数据，如果用户未在私有频道成员列表中，禁止用户往私有频道发送消息。**

了解私有频道

> 提示
>
> 客户端不提供创建私有频道、变更频道类型的 API。请通过服务端 API 实现。

超级群的频道按类型区分为公有频道与私有频道。公有频道对所有超级群成员开放（无需加入）。该超级群的所有成员都会接收公有频道下的消息。私有频道仅对该频道的成员列表开放，仅频道成员可在该频道中收发消息、接收频道状态变化的通知。私有频道可以随时切换为公有频道，公有频道也可以切换为私有频道。

例如，管理员在社区中新创建一个频道，并定义为私有频道。接着邀请社区中部分成员加入私有频道。只有加入频道的成员才可以浏览此频道，并在频道中接收、发送消息。

频道变更会影响该频道下服务端历史消息的拉取权限，具体如下：

- 公有频道转换为私有频道：仅当前私有频道成员列表中的用户可以拉取该频道下的历史消息（含原公有频道消息）；
- 私有频道转换为公有频道：所有超级群用户均可拉取该频道下的历史消息（含原私有频道消息）。

您可以通过以下方式管理私有频道：

功能描述	客户端 API	融云服务端 API
创建私有频道	不提供该 API	创建频道
删除私有频道	不提供该 API	删除频道
查询频道列表	不提供该 API	查询频道列表
变更频道类型	不提供该 API	变更频道类型

了解私有频道成员列表

> 提示
>
> 客户端 SDK 不提供管理私有频道成员列表的 API。您可以调用服务端 API 查询私有频道成员列表、或将超级群成员加入、移出成员列表。

频道类型为私有频道时，只有该列表中用户可在频道中的收发消息、接收频道内状态通知。

私有频道转为公有频道时，频道变更为对所有超级群成员开放。但该列表数据不会被删除。假设频道类型再次变更为私有，则启用该成员列表。

您也可以为公有频道创建私有频道成员列表。一旦该公有频道变更类型为私有频道，该列表即生效。

> 提示
>
> - 删除频道时，服务端会清除该频道的私有频道成员列表。
> - 一旦超级群成员退群，服务端会自动将用户从私有频道成员列表移除。

您可以通过以下方式管理私有频道成员列表：

功能描述	客户端 API	融云服务端 API
加入私有频道成员列表	不提供该 API	添加私有频道成员
移出私有频道成员列表	不提供该 API	删除私有频道成员
查询私有频道成员列表	不提供该 API	查询私有频道成员列表

用户组概述 用户组概述

更新时间:2024-08-30

超级群用户组（User Group）功能是融云超级群业务提供的群成员管理工具，结合**超级群私有频道**功能，可以帮助 App 实现更高效的超级群成员管理，沟通管理，和更精细的用户通知能力。

前提条件

> 提示
>
> 客户端 SDK 从 5.4.0 版本开始支持超级群用户组功能。

超级群用户组功能主要通过与**私有频道**的绑定操作，提供了对用户在社区私有频道中沟通权限（收发消息、通知）的批量管理能力，提升了 App 集成效率与对超级群的运营管理能力。

在了解与使用超级群用户组功能前，需要先了解超级群私有频道功能。请参见超级群私有频道概述。

如何使用超级群用户组

App 服务端可以通过调用融云服务端 API，在超级群中创建最多 50 个用户组（`userGroup`），每个用户组成员最多由 100 个超级群成员组成。单个用户可以存在于多个用户组中。

用户组创建后，可以与超级群频道绑定。如果用户组绑定了一个或多个私有频道，该用户组的所有成员即具有在绑定的私有频道中收发消息、接收通知的能力。

- App 将用户组绑定私有频道后，可认为组中所有用户均加入了该私有频道。与该私有频道成员列表中的用户类似，只有加入频道的成员才可以浏览此频道，并在频道中接收消息，发送消息，和接收通知。
- App 将用户组绑定公有频道后，不会影响组中用户可收发消息的范围。但一旦该公有频道转为私有频道，该组用户将具有在该私有频道中收发消息、获取通知的能力。
- App 可以在一个频道上绑定最多 10 个用户组，一个用户组可以与多个频道绑定（一个超级群最多 50 个频道）。

> 提示
>
> 超级群业务默认提供 `RCDefault` 频道，对所有超级群成员开放，不可转为私有频道。建议不要将用户组绑定到 `RCDefault` 频道。

App 客户端无法进行用户组管理操作，仅可通过 SDK 提供的回调方法监听用户组相关变更的通知，具体包括：

- **删除用户组**：用户组下所有用户均可收到客户端回调
- **用户组成员变更**：被加入或移出用户组的用户可收到客户端回调
- **频道与用户组绑定关系变更**：用户组下所有用户均可收到客户端回调

混合使用用户组与私有频道成员列表

只要 App 用户在指定私有频道绑定的任意一个用户组中，或者在有该私有频道成员列表中，该用户就能在私有频道中收发消息接收通知。

如果混合使用私有频道成员列表与用户组，在 App 业务中可能存在以下情况：

- 私有频道配置了私有成员列表，并添加了多位用户。
- 该私有频道绑定了多个用户组。

在上述使用场景中，某个用户可能既在私有频道成员列表中，又同时在该频道绑定的多个用户组中。如果该用户不再使用该私有频道，App 进行以下操作，确保该用户无法继续在私有频道收发消息：

- 从该私有频道的成员列表中移除该用户。
- 检查该私有频道绑定的所有用户组，从绑定的所有用户组中移除该用户，或解绑用户组。

用户组管理接口

即时通讯（IM）服务端提供了超级群用户组的基础管理接口。用户组的业务逻辑需要 App 服务端自行实现，例如申请加入用户组、审核用户组加入申请等。建议 App 服务端同时维护一份用户、用户组、频道之间对应关系的数据。

对于客户端开发人员来说，创建用户组等基础管理操作只需要与 App 服务端交互即可。App 服务端需要调用相应的融云服务端 API（Server API）接口相关接口创建用户组等其他管理操作。

下表列出了超级群用户组基础管理接口。更多相关接口可参见 IM 服务端文档 API 接口列表。

功能分类	功能描述	融云服务端 API
创建、删除用户组	在指定超级群下创建用户组。如删除用户组，则用户、用户组、频道之间的关系不复存在。	创建用户组、删除用户组
查询用户组列表	分页查询指定超级群下的用户组，返回用户组 ID 列表。	查询用户组列表
添加、删除用户	在超级群用户组中添加、删除用户。	添加用户、移出用户
查询用户所属用户组	分页查询指定单个用户在超级群下所属的用户组列表，返回用户组 ID 列表。	查询用户所属用户组
绑定频道与用户组	将指定单个超级群频道与用户组绑定，可绑定多个用户组，单个频道最多支持与 10 个用户组绑定。	绑定频道与用户组
解绑频道与用户组	将指定单个超级群频道与用户组解除绑定，单次请求可解绑最多 10 个用户组。	解绑频道与用户组
查询频道绑定的用户组	分页查询指定的超级群频道绑定的用户组列表，返回用户组 ID 列表。	查询频道绑定的用户组
查询用户组绑定的频道	分页查询指定单个用户组绑定的超级群频道列表，返回超级群频道 ID 列表。	查询用户组绑定的频道

创建超级群与频道 创建超级群与频道

更新时间:2024-08-30

客户端 SDK 不提供创建超级群与创建群频道的接口。请使用融云服务端 API（Server API）的相关接口创建超级群、群频道，或进行其他管理操作。

> 提示
>
> 单个用户最多可以加入 100 个超级群。单个用户在每个群中最多可以加入或者创建 50 个频道。

本文仅简单介绍创建超级群与创建频道的基本流程。

创建超级群

创建超级群必须使用融云服务端 API（Server API），具体接口使用方法请参见服务端文档创建超级群。

基本流程

1. App 客户端请求 App 服务端（AppServer）创建超级群。
2. App 服务端调用融云 Server API 接口创建超级群，群组 ID 由 App 服务器自行生成
3. 超级群创建成功后，由 App 服务端返回给 App 客户端。

如何处理与展示超级群列表

AppServer 需要保存当前用户的超级群列表，并下发给 App，然后进行展示。

> 考虑到 App 由于自身业务需求，需要知晓当前用户的超级群列表（例如用户所在的超级群有等级权重等排序规则），而融云的超级群列表是通过消息产生的，因而两个列表可能不完全一样，建议由 App 按照自身业务保存用户的超级群列表。

创建群频道

创建超级群必须使用融云服务端 API（Server API），具体接口使用方法请参见服务端文档创建频道。

基本流程

如何处理与展示超级群频道列表

为方便在同一超级群下的不同用户按需看到自己需要的列表（例如用户对特定频道标星需要特别展示，APP 服务则需要按用户记录），APP 服务应采取更为灵活的方式维护超级群的频道列表。

APP 服务端也可以按照需求将超级群分组（如**超级群概述**中的 UI 框架设计所示）。

获取频道列表 获取频道列表

更新时间:2024-08-30

App 可按需使用客户端 SDK 与融云服务端 API 提供的能力，采取灵活的方式维护超级群的频道列表。

> 提示
>
> - 如果您的应用/环境在 2022.10.13 日及以后开通超级群服务，超级群业务中会包含一个 ID 为 `RCDefault` 的默认频道。如果发消息时不指定频道 ID，则该消息会发送到 `RCDefault` 频道中。在获取 `RCDefault` 频道的历史消息时，需要传入该频道 ID。
> - 如果您的应用/环境在 2022.10.13 日前已开通超级群服务，在发送消息时如果不指定频道 ID，则该消息不属于任何频道。获取历史消息时，如果不传入频道 ID，可获取不属于任何频道的消息。融云支持客户调整服务至最新行为。该行为调整将影响客户端、服务端收发消息、获取会话、清除历史消息、禁言等多个功能。如有需要，请提交工单咨询详细方案。

如何获取超级群全部频道列表

超级群下全部频道的列表可通过融云服务端 API (`/ultragroup/channel/get.json`) 获取。

注意，对单个用户来说，最多可以加入 100 个超级群，在每个超级群中最多可以加入或者创建 50 个频道。

> 提示
>
> 客户端 SDK 会通过频道中收发的消息在本地数据库中生成一个超级群频道列表。该列表仅包含了已在本地产生消息的频道，可能并非该超级群下全部频道的列表。

建议从 App 业务服务端维护超级群的频道列表。App 业务一般需要知晓当前用户的所加入的超级群，以及超级群下的频道列表，才能实现特定的业务功能。假设同一超级群下的不同用户需要展示个性化的频道列表（例如 App 需要在 UI 上展示用户标星的特定频道），则需要为用户保存超级群频道列表。

APP 服务端也可以按照需求将超级群分组（如**超级群概述**中的 UI 框架设计所示）。

获取本地指定超级群下的频道列表

客户端 SDK 会根据频道中收发的消息在本地数据库中生成对应频道的会话。您可以从本地数据库获取 SDK 生成的频道列表。获取到的频道列表按照时间倒序排列。

```
ChannelClient.getInstance()
.getConversationListForAllChannel(
Conversation.ConversationType.ULTRA_GROUP,
groupId,
new IRongCoreCallback.ResultCallback<List<Conversation>>() {
@Override
public void onSuccess(List<Conversation> conversations) {
if (conversations == null) return;
for (Conversation conversation : conversations) {
//获取会话的未读消息数
int unreadMessageCount = conversation.getUnreadMessageCount();
//获取会话的 @ 未读消息数
int unreadMentionedCount =
conversation.getUnreadMentionedCount();
//获取会话的 @ 我的未读消息数 @since 5.4.5
int mentionedMeCount = conversation.getUnreadMentionedMeCount();

}
//该回调在非 UI 线程返回，如果需要做 UI 操作，请切换至 UI 线程
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {}
});
```

获取本地指定超级群特定类型频道列表

> 提示
>
> SDK 从 5.2.4 开始支持按类型获取频道列表。

客户端 SDK 会根据频道中收发的消息在本地数据库中生成对应频道的会话。您可以从本地数据库获取 SDK 生成的频道列表。获取到的频道列表按照时间倒序排列。

频道类型（`channelType`）支持超级群公有频道或私有频道。

```
/**
* 获取超级群频道列表
*
* @param targetId 超级群 id
* @param channelType 频道类型
* @param callback 结果回调
*/
public abstract void getUltraGroupChannelList(
String targetId,
IRongCoreEnum.UltraGroupChannelType channelType,
IRongCoreCallback.ResultCallback<List<Conversation>> callback);
```

- **参数说明**：
  - **频道类型**（channelType）

```
/** 超级群频道类型 Added from version 5.2.4 */
public enum UltraGroupChannelType {
/** 超级群共有频道 */
ULTRA_GROUP_CHANNEL_TYPE_PUBLIC,

/** 超级群私有频道 */
ULTRA_GROUP_CHANNEL_TYPE_PRIVATE;
}
```

监听频道状态变更　　监听频道状态变更

更新时间:2024-08-30

客户端可设置监听，在超级群频道发生类型变更（仅适用于私有频道）、成员变更、频道删除等情况时收到取对应通知。

- 由于频道类型、用户是否在私有频道成员列表等差异，通知用户的范围会有差异。
- 客户端 SDK 接收到频道删除（解散）、私有频道成员列表变更通知后，会根据具体变化清理本地数据。

设置频道状态变化通知

1. 需要首先实现 IRongCoreListener.UltraGroupChannelListener 接口，包括私有频道成员变更通知，频道类型变更通知，频道解散通知。

```
/** 私有频道时，私有频道成员列表用户被踢通知。 此时私有频道成员列表内用户才能收到此事件。公有频道时，不会触发此回调 */
void ultraGroupChannelUserDidKicked(List<UltraGroupChannelUserKickedInfo> infoList);

/** 频道属性改变时有频道时的回调。私有变公有时，私有频道成员列表内会收到此事件。公有变私有时，所有成员都能收到此事件 */
void ultraGroupChannelTypeDidChanged(List<UltraGroupChannelChangeTypeInfo> infoList);

/** 频道被解散通知 公有频道时，删除频道通知频道中所有人。私有频道时，删除频道通知私有频道成员列表中所有人。*/
void ultraGroupChannelDidDisbanded(List<UltraGroupChannelDisbandedInfo> infoList);
```

2. 通过调用 io.rong.imlib.ChannelClient#setUltraGroupChannelListener 方法, 添加频道变更监听:

```
ChannelClient.getInstance().setUltraGroupChannelListener();
```

频道类型变更的通知

超级群的频道按类型区分为公有频道与私有频道。超级群频道类型变更仅可通过 App 服务端调用服务端 API 实现。客户端 SDK 不提供接口。

频道类型发生变更时，SDK 通过以下方法通知 App：

```
void ultraGroupChannelTypeDidChanged(List<UltraGroupChannelChangeTypeInfo> infoList);
```

- 参数说明

返回值	返回类型	说明
infoList	UltraGroupChannelChangeTypeInfo	频道变更信息

- 频道变更类型说明

枚举值	说明
ULTRA_GROUP_CHANNEL_CHANGE_TYPE_PUBLIC_TO_PRIVATE	超级群公有频道变成了私有频道
ULTRA_GROUP_CHANNEL_CHANGE_TYPE_PRIVATE_TO_PUBLIC	超级群私有频道变成了公有频道
ULTRA_GROUP_CHANNEL_CHANGE_TYPE_PUBLIC_TO_PRIVATE_USER_NOT_IN	超级群公有频道变成了私有频道，但是当前用户不在该私有频道中

频道类型的变更的通知范围

- **公有频道变私有频道**：公有频道变私有频道时，所有用户都会收到通知。但根据用户是否在私有频道成员列表中，收到的通知有差异：
  - 在私有频道成员列表内的用户，收到的变更类型是 `ULTRA_GROUP_CHANNEL_CHANGE_TYPE_PUBLIC_TO_PRIVATE`。
  - 不在私有频道成员列表的用户，收到的变更类型为 `ULTRA_GROUP_CHANNEL_CHANGE_TYPE_PUBLIC_TO_PRIVATE_USER_NOT_IN`。

  如有需要，App 可在公有频道变私有频道前，提前指定用户加入私有频道成员列表。
- **私有频道变公有频道**：私有频道变公有频道时，仅在私有频道成员列表的用户会收到通知，变更类型为 ULTRA_GROUP_CHANNEL_CHANGE_TYPE_PRIVATE_TO_PUBLIC。

频道已删除（解散）的通知

超级群频道删除（解散）仅可通过 App 服务端调用服务端 API 实现。客户端 SDK 不提供接口。

删除频道时，SDK 通过以下方法通知 App：

```
void ultraGroupChannelDidDisbanded(List<UltraGroupChannelDisbandedInfo> infoList);
```

- 参数说明

返回值	返回类型	说明
infoList	UltraGroupChannelDisbandedInfo	频道变更信息

频道已删除的通知范围

- **删除公有频道的通知**：所有用户会收到通知。
- **删除私有频道的通知**：仅在私有频道成员列表的用户会收到通知。

如何清理本地数据

客户端 SDK 收到通知后会清理会删除用户本地会话，但会保留本地会话的消息。

私有频道成员列表变更的通知

超级群私有频道成员列表仅可通过服务端 API 变更。客户端 SDK 不提供接口。

私有频道成员列表发生变化时，SDK 通过以下方法通知 App：

```
void ultraGroupChannelUserDidKicked(List<UltraGroupChannelUserKickedInfo> infoList);
```

- 参数说明：

返回值	返回类型	说明
infoList	UltraGroupChannelUserKickedInfo	频道变更信息

私有频道成员列表变更的通知范围

- 如果频道类型为私有频道，将用户从私有频道成员列表移除时，仅通知被移除的用户
- 如果频道类型为公有频道，将用户从私有频道成员名单移除时，不发送通知。注意，该列表在该公有频道变更类型为私有频道时才会生效。

如何清理本地数据

- **当前登录用户被移出私有频道成员列表**
  - （SDK < 5.4.0）：客户端 SDK 收到通知后会从会话列表中删除本地会话，但会保留本地会话的消息。
  - （SDK ≧ 5.4.0）：客户端 SDK 收到通知后会不从会话列表中删除本地会话，App 可以自行决定是否需要删除会话及会话中的消息。
- **其他情况**：如果当被移出列表的用户非本端登录用户，客户端 SDK 收到通知后不做任何处理。

监听用户组状态变更 监听用户组状态变更

更新时间:2024-08-30

> SDK 从 5.4.0 版本开始支持超级群用户组功能。

客户端可设置监听，在超级群用户组发生变更时收到取对应通知。

客户端 SDK 针对以下操作提供回调。回调的通知范围与回调数据有差异具体如下：

- **用户加入用户组**：被加入用户组的用户可收到通知，通知中携带超级群 ID、用户组 ID。
- **用户被移出用户组**：被移出的用户可收到通知，通知中携带超级群 ID、用户组 ID。
- **频道与用户组绑定**：用户组下所有用户均可收到通知，通知中携带超级群 ID、频道 ID、频道类型、用户组 ID。
- **频道与用户组解除绑定**：用户组下所有用户均可收到通知，通知中携带超级群 ID、频道 ID、频道类型、用户组 ID。
- **删除用户组**：用户组下所有用户均可收到通知，通知中携带超级群 ID、用户组 ID。

创建用户组时，SDK 不会收到回调。

监听用户组变更通知

SDK 在 IRongCoreListener.UserGroupStatusListener 接口类中提供了与用户组变更相关的回调方法。您可以从返回的 ConversationIdentifier 中获取超级群 ID（targetId）、频道 ID（channelId）数据。

```
interface UserGroupStatusListener {
/**
* 当前用户收到超级群下的用户组中解散通知
*
* @param identifier 会话标识
* @param userGroupIds 用户组ID列表
*/
void userGroupDisbandFrom(ConversationIdentifier identifier, String[] userGroupIds);

/**
* 当前用户被添加到超级群下的用户组
*
* @param identifier 会话标识
* @param userGroupIds 用户组ID列表
*/
void userAddedTo(ConversationIdentifier identifier, String[] userGroupIds);

/**
* 当前用户从到超级群下的用户组中被移除
*
* @param identifier 会话标识
* @param userGroupIds 用户组ID列表
*/
void userRemovedFrom(ConversationIdentifier identifier, String[] userGroupIds);

/**
* 频道中绑定用户组回调
*
* @param identifier 频道标识
* @param channelType 频道类型
* @param userGroupIds 用户组ID列表
*/
void userGroupBindTo(
ConversationIdentifier identifier,
IRongCoreEnum.UltraGroupChannelType channelType,
String[] userGroupIds);

/**
* 频道解绑用户组回调
*
* @param identifier 频道标识
* @param channelType 频道类型
* @param userGroupIds 用户组ID列表
*/
void userGroupUnbindFrom(
ConversationIdentifier identifier,
IRongCoreEnum.UltraGroupChannelType channelType,
String[] userGroupIds);
}
```

App 可通过以下方法设置用户组变更监听：

```
ChannelClient.getInstance().setUserGroupStatusListener(
new IRongCoreListener.UserGroupStatusListener() {
@Override
public void userGroupDisbandFrom(
ConversationIdentifier identifier, String[] userGroupIds) {
}

@Override
public void userAddedTo(
ConversationIdentifier identifier, String[] userGroupIds) {
}

@Override
public void userRemovedFrom(
ConversationIdentifier identifier, String[] userGroupIds) {
}

@Override
public void userGroupBindTo(
ConversationIdentifier identifier,
IRongCoreEnum.UltraGroupChannelType channelType,
String[] userGroupIds) {
}

@Override
public void userGroupUnbindFrom(
ConversationIdentifier identifier,
IRongCoreEnum.UltraGroupChannelType channelType,
String[] userGroupIds) {

}
}
);
```

关于更新 UI 的提示

考虑到同一用户既在私有频道成员列表中，又在私有频道绑定的（多个）用户组中，App 可以在收到回调时向 App 自身业务服务端查询当前用户是否仍可继续访问该私有频道，并根据该结果刷新 UI。

对于未在通知范围内的用户，可以在用户进入相应页面时向 App 自身业务服务端查询数据，并决定是否需要刷新 UI。

监听会话同步状态　　监听会话同步状态

更新时间:2024-08-30

在每一次连接 IM 服务时，客户端 SDK 都会自动从服务端拉取超级群会话列表与消息。

通过设置超级群会话同步监听器，您可以获取超级群会话列表与会话最后一条消息同步完成的通知，从而进行不同业务处理，例如刷新 UI 界面。

建议在应用生命周期内设置。

设置超级群会话同步监听器

> 提示
>
> 从 SDK 5.2.2 版本开始支持该回调接口。

```
// 超级群会话监听器
interface UltraGroupConversationListener {
// 超级群会话列表与会话最后一条消息同步完成
void ultraGroupConversationListDidSync();
}

public void setUltraGroupConversationListener(IRongCoreListener.UltraGroupConversationListener listener) {
```

收发消息 收发消息

更新时间:2024-08-30

本文介绍了如何从客户端发送超级群消息。超级群收发消息需要使用 RongCoreClient 下的方法。

前置条件

建议先阅读超级群概述和超级群私有频道概述，了解在 App 业务中**如何使用频道**和超级群频道功能特性。

- 通过服务端 API 创建超级群
- 通过服务端 API 创建频道，或使用默认频道 ID `RCDefault`
- 通过服务端 API 将发件人加入超级群
- 如不确定发件人是否在超级群中，请通过服务端 API 查询用户是否为群成员
- 如向超级群私有频道中发送消息，请确认已通过服务端 API 添加私有频道成员

> 提示
>
> 当前频道聊天页面发送与接收消息，需要同时检查超级群 ID 和频道 ID。如果超级群 ID 和频道 ID 和当前频道聊天页面对应，才能在当前频道页面进行展示处理，否则就不处理。如果消息出现在其他聊天页面，一般是因为超级群 ID 或者频道 ID 发生错误。

构造消息对象

在发送消息前，需要构造 Message 对象，消息的 `conversationType` 字段必须填写超级群业务的会话类型 `ConversationType.ULTRA_GROUP`。消息的 `targetId` 字段表示超级群 ID，`channelId` 表示超级群频道 ID。

`Message` 对象中可包含普通消息内容或媒体消息内容。普通消息内容指 `MessageContent` 的子类，例如文本消息（TextMessage）。

```
String targetId = "超级群 ID";
ConversationType conversationType = Conversation.ConversationType.ULTRA_GROUP;
String channelId = "超级群频道 ID";

TextMessage messageContent = TextMessage.obtain("测试超级群");

Message message = Message.obtain(targetId, conversationType, channelId, messageContent);
```

媒体消息内容指 `MediaMessageContent` 的子类，例如图片消息（ImageMessage）、GIF 消息（GIFMessage）等。

```
String targetId = "超级群 ID";
ConversationType conversationType = Conversation.ConversationType.ULTRA_GROUP;
String channelId = "超级群频道 ID";

Uri localUri = Uri.parse("file://图片的路径");//图片本地路径，接收方可以通过 getThumUri 获取自动生成的缩略图 Uri
boolean mIsFull = true; //是否发送原图
ImageMessage mediaMessageContent = ImageMessage.obtain(localUri, mIsFull);

Message message = Message.obtain(targetId, conversationType, channelId, mediaMessageContent);
```

> 提示
>
> - 如果您的应用/环境在 2022.10.13 日及以后开通超级群服务，超级群业务中会包含一个 ID 为 `RCDefault` 的默认频道。如果发消息时不指定频道 ID，则该消息会发送到 `RCDefault` 频道中。
> - 如果您的应用/环境在 2022.10.13 日前已开通超级群服务，在发送消息时如果不指定频道 ID，则该消息不属于任何频道。融云支持客户调整服务至最新行为。该行为调整将影响客户端、服务端收发消息、获取会话、清除历史消息、禁言等多个功能。如有需要，请提交工单咨询详细方案。

发送普通消息

发送超级群普通消息需要使用 RongCoreClient 中的提供的 `sendMessage` 方法。

```
RongCoreClient.getInstance().sendMessage(message, null, null, new IRongCoreCallback.ISendMessageCallback() {

@Override
public void onAttached(Message message) {

}

@Override
public void onSuccess(Message message) {

}

@Override
public void onError(Message message, IRongCoreEnum.CoreErrorCode errorCode) {

}
});
```

sendMessage 中直接提供了用于控制推送通知内容（pushContent）和推送附加信息（pushData）的参数。不过，如果您需要更精细地控制离线推送通知，例如标题、内容、图标、或其他第三方厂商个性化配置，请使用消息推送属性进行配置，详见自定义消息推送通知。

- 如果发送的消息属于 SDK 内置的普通消息类型，例如 TextMessage，这两个参数可设置为 `null`。一旦消息触发离线推送通知时，融云服务端会使用各个内置消息类型默认的 `pushContent` 值。关于各类型消息的默认推送通知内容，详见用户内容类消息格式。
- 如果消息类型为自定义消息类型，且需要支持离线推送通知，则必须向融云提供 `pushContent` 字段，否则用户无法收到离线推送通知。
- 如果消息类型为自定义消息类型，但不需要支持远程推送通知（例如通过自定义消息类型实现的 App 业务层操作指令），可将 `pushContent` 字段留空。
- `Message` 的推送属性配置 `MessagePushConfig` 的 `pushContent` 和 `pushData` 会覆盖此处配置，并提供更多配置能力，例如自定义推送通知的标题。详见自定义消息推送通知。

参数	类型	说明
message	Message	要发送的消息实体，在消息实体中必须指定会话类型（`conversationType`），目标 ID（`targetId`），消息内容（`messageContent`）。详见消息介绍。
pushContent	String	修改或指定远程消息推送通知栏显示的内容。您也可以在 `Message` 的推送属性（`MessagePushConfig`）中配置，会覆盖此处配置，详见自定义消息推送通知。 • 如果希望使用融云默认推送通知内容，可以填 `nil`。注意自定义消息类型无默认值。 • 如果要为该条消息指定个性化的离线推送通知，必须在以上任一处向融云提供 `pushContent` 字段内容。 • 如果您自定义的消息类型需要离线推送服务，必须在以上任一处向融云提供 `pushContent` 字段内容，否则用户无法收到离线推送通知。
pushData	String	远程推送附加信息。对端收到远程推送消息时，可通过以下方法获取该字段内容： `io.rong.push.notification.PushNotificationMessage#getPushData()` 。您也可以在 `Message` 的推送属性（`MessagePushConfig`）中配置，会覆盖此处配置，详见自定义消息推送通知。
callback	ISendMessageCallback	发送消息的回调

发送媒体消息

媒体消息 `Message` 对象的 `content` 字段必须传入 MediaMessageContent 的子类对象，表示媒体消息内容。例如图片消息内容（ImageMessage）、GIF 消息内容（GIFMessage），或继承自 MediaMessageContent 的自定义媒体消息内容。

图片消息（ImageMessage）支持设置为发送原图。

```
String targetId = "目标 ID";
ConversationType conversationType = ConversationType.ULTRA_GROUP;
String channelId = "目标频道 ID";
Uri localUri = Uri.parse("file://图片的路径");//图片本地路径，接收方可以通过 getThumUri 获取自动生成的缩略图 Uri
boolean mIsFull = true; //是否发送原图
ImageMessage mediaMessageContent = ImageMessage.obtain(localUri, mIsFull);

Message message = Message.obtain(targetId, conversationType, channelId, mediaMessageContent);
```

向超级群中发送媒体消息需要使用 RongCoreClient 中的 sendMediaMessage 方法。SDK 会为图片、小视频等生成缩略图，根据默认压缩配置进行压缩，再将图片、小视频等媒体文件上传到融云默认的文件服务器（文件存储时长），上传成功之后再发送消息。图片消息如已设置为发送原图，则不会进行压缩。

```
RongCoreClient.getInstance().sendMediaMessage(message, null, null, new IRongCoreCallback.ISendMediaMessageCallback() {
@Override
public void onProgress(Message message, int i) {

}

@Override
public void onCanceled(Message message) {

}

@Override
public void onAttached(Message message) {

}

@Override
public void onSuccess(Message message) {

}

@Override
public void onError(final Message message, final IRongCoreEnum.CoreErrorCode errorCode) {

}
});
```

sendMediaMessage 中直接提供了用于控制推送通知内容（pushContent）和推送附加信息（pushData）的参数。不过，如果您需要更精细地控制离线推送通知，例如标题、内容、图标、或其他第三方厂商个性化配置，请使用消息推送属性进行配置，详见自定义消息推送通知。

- 如果发送的消息属于 SDK 内置的媒体消息类型，例如 ImageMessage，这两个参数可设置为 `null`。一旦消息触发离线推送通知时，融云服务端会使用各个内置消息类型默认的 `pushContent` 值。关于各类型消息的默认推送通知内容，详见用户内容类消息格式。
- 如果消息类型为自定义消息类型，且需要支持离线推送通知，则必须向融云提供 `pushContent` 字段，否则用户无法收到离线推送通知。
- 如果消息类型为自定义消息类型，但不需要支持远程推送通知（例如通过自定义消息类型实现的 App 业务层操作指令），可将 `pushContent` 字段留空。
- `Message` 的推送属性配置 `MessagePushConfig` 的 `pushContent` 和 `pushData` 会覆盖此处配置，并提供更多配置能力，例如自定义推送通知的标题。详见自定义消息推送通知。

参数	类型	说明
message	Message	要发送的消息实体，在消息实体中必须指定会话类型（conversationType），目标 ID（targetId），消息内容（messageContent）。详见消息介绍。
pushContent	String	修改或指定远程消息推送通知栏显示的内容。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见自定义消息推送通知。 • 如果希望使用融云默认推送通知内容，可以填 `nil`。注意自定义消息类型无默认值。 • 如果要为该条消息指定个性化的离线推送通知，必须在以上任一处向融云提供 `pushContent` 字段内容。 • 如果您自定义的消息类型需要离线推送服务，必须在以上任一处向融云提供 `pushContent` 字段内容，否则用户无法收到离线推送通知。
pushData	String	远程推送附加信息。对端收到远程推送消息时，可通过以下方法获取该字段内容： io.rong.push.notification.PushNotificationMessage#getPushData() 。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见自定义消息推送通知。
callback	ISendMediaMessageCallback	发送媒体消息的回调

接收消息

> 提示
>
> 消息接收监听的设置在 RongCoreClient 中。

设置监听

```
public static boolean addOnReceiveMessageListener(OnReceiveMessageWrapperListener listener)
```

实现监听代理方法

```
public boolean onReceived(Message message, int left)
```

如何发送 @ 消息

@消息在融云即时通讯服务中不属于一种预定义的消息类型（详见服务端文档 消息类型概述）。融云通过在消息中携带 mentionedInfo 数据，帮助 App 实现提及他人（@）功能。

消息的 MessageContent 中的 MentionedInfo 字段存储了携带所 @ 人员的信息。无论是 SDK 内置消息类型，或者您自定义的消息类型，都直接或间接继承了 MessageContent 类。

融云支持在向群组和超级群中发消息时，在消息内容中添加 mentionedInfo。消息发送前，您可以构造 MentionedInfo，并设置到消息的 MessageContent 中。

```
List<String> userIdList = new ArrayList<>();
userIdList.add("userId1");
userIdList.add("userId2");
MentionedInfo mentionedInfo = new MentionedInfo(MentionedInfo.MentionedType.PART, userIdList, null);
TextMessage messageContent = TextMessage.obtain("文本消息");
messageContent.setMentionedInfo(mentionedInfo);
```

MentionedInfo 参数：

参数	类型	说明
type	MentionedType	（必填）指定 MentionedInfo 的类型。MentionedType.ALL 表示需要提及（@）所有人。MentionedType.PART 表示需要 @ 部分人，被提及的人员需要在 userIdList 中指定。
userIdList	List<String>	被提及（@）用户的 ID 集合。当 type 为 MentionedType.PART 时必填。
mentionedContent	String	触发离线消息推送时，通知栏显示的内容。如果是 NULL，则显示默认提示内容（“有人 @ 你”）。@消息携带的 mentionedContent 优先级最高，会覆盖所有默认或自定义的 pushContent 数据。

IMLib SDK 接收消息后，您需要处理 @ 消息中的数据。您可以在获取 Message(消息对象)后，通过以下方法获取到消息对象携带的 MentionedInfo 对象。

```
MentionedInfo mentionedInfo = message.getContent().getMentionedInfo();
```

自定义消息推送通知

您可以在发送消息时提供 MessagePushConfig 配置，对单条消息的推送行为进行个性化配置。例如：

- 自定义推送标题、推送通知内容
- 自定义通知栏图标
- 添加远程推送附加信息
- 越过接收客户端的配置，强制在推送通知内显示通知内容
- 其他 APNs 或 Android 推送通道支持的个性化配置

关于 MessagePushConfig 的配置方式详见自定义消息推送通知。

为消息禁用推送通知

在接收者未上线时，融云服务端默认触发推送服务，将消息通过推送通道下发（服务端是否触发推送也会收到应用级别、用户级别免打扰设置的影响）。

您的 App 用户可能希望在发送消息时就指定该条消息不需要触发推送，该需求可通过 Message 对象的 MessageConfig 配置实现。

以下示例中，我们将 messageConfig 的 disableNotification 设置为 true 禁用该条消息的推送通知。接收方再次上线时需要主动拉取。

```
String targetId = "目标 ID";
String channelId = "目标频道 ID";
ConversationType conversationType = ConversationType.ULTRA_GROUP;
TextMessage messageContent = TextMessage.obtain("消息内容");
Message message = Message.obtain(targetId, conversationType, channelId, messageContent);

message.setMessageConfig(new MessageConfig.Builder().setDisableNotification(true).build());
```

如何处理消息发送失败

对于客户端本地会存储的消息类型（参见消息类型概述），如果触发（onAttached）回调，表示此时该消息已存入本地数据库，并已进入本地消息列表。

- 如果入库失败，您可以检查参数是否合法，设备存储是否可正常写入。
- 如果入库成功，但消息发送失败（`onError`），App 可以在 UI 临时展示这条发送失败的消息，并缓存 `onError` 抛出的 `Message` 实例，在合适的时机重新调用 `sendMessage` / `sendMediaMessage` 发送。请注意，如果不复用该消息实例，而是重发相同内容的新消息，本地消息列表中会存储内容重复的两条消息。

对于客户端本地不存储的消息，如果发送失败（onError），App 可缓存消息实例再重试发送，或直接新建消息实例进行发送。

发送超级群定向消息 发送超级群定向消息

更新时间:2024-08-30

可发送普通消息与媒体消息给超级群频道中的指定的一个或多个成员，其他成员不会收到该消息。

局限

- 5.6.9 版本开始支持该能力。
- 定向目标用户上限 300 个用户 ID。
- 如果为 @ 消息，不支持 @所有人。

发送定向普通消息

使用 sendDirectionalMessage 在超级群中发送普通消息给指定频道中的指定用户。不在接收列表的用户不会收到这条消息。

请在消息对象中设置超级群会话类型、超级群 ID 和 频道 ID。频道 ID 为空时默认向 RCDefault 频道发送消息。注意，Message 中不会保存接收用户 ID 列表。

```
Conversation.ConversationType type = Conversation.ConversationType.GROUP;
String targetId = "123";
TextMessage content = TextMessage.obtain("定向消息文本内容");
Message message = Message.obtain(targetId, conversationType, channelId, messageContent);

String[] userIds = new String[]{"id_01", "id_02"};

RongCoreClient.getInstance().sendDirectionalMessage(message, userIds, null, null, new IRongCoreCallback.ISendMessageCallback() {
@Override
public void onAttached(Message message) {

}

@Override
public void onSuccess(Message message) {

}

@Override
public void onError(final Message message, IRongCoreEnum.CoreErrorCode errorCode) {

}
});
```

sendDirectionalMessage 中直接提供了用于控制推送通知内容（pushContent）和推送附加信息（pushData）的参数。不过，如果您需要更精细地控制离线推送通知，例如标题、内容、图标、或其他第三方厂商个性化配置，请使用消息推送属性进行配置，详见自定义消息推送通知。

- 如果发送的消息属于 SDK 内置的普通消息类型，例如 TextMessage，这两个参数可设置为 `null`。一旦消息触发离线推送通知时，融云服务端会使用各个内置消息类型默认的 `pushContent` 值。关于各类型消息的默认推送通知内容，详见用户内容类消息格式。
- 如果消息类型为自定义消息类型，且需要支持离线推送通知，则必须向融云提供 `pushContent` 字段，否则用户无法收到离线推送通知。
- 如果消息类型为自定义消息类型，但不需要支持远程推送通知（例如通过自定义消息类型实现的 App 业务层操作指令），可将 `pushContent` 字段留空。
- `Message` 的推送属性配置 `MessagePushConfig` 的 `pushContent` 和 `pushData` 会覆盖此处配置，并提供更多配置能力，例如自定义推送通知的标题。详见自定义消息推送通知。

参数	类型	说明
message	Message	要发送的消息体。必填属性包括会话类型（conversationType），频道 ID（channelId）会话 ID（targetId），消息内容（content）。
userIds	String[]	需要接收消息的用户列表。
pushContent	String	修改或指定远程消息推送通知栏显示的内容。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见自定义消息推送通知。 • 如果希望使用融云默认推送通知内容，可以填 `null`。注意自定义消息类型无默认值。 • 如果要为该条消息指定个性化的离线推送通知，必须在以上任一处向融云提供 `pushContent` 字段内容。 • 如果您自定义的消息类型需要离线推送服务，必须在以上任一处向融云提供 `pushContent` 字段内容，否则用户无法收到离线推送通知。
pushData	String	远程推送附加信息。对端收到远程推送消息时，可通过以下方法获取该字段内容： io.rong.push.notification.PushNotificationMessage#getPushData() 。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见自定义消息推送通知。
callback	ISendMessageCallback	发送消息的回调。

发送定向媒体消息

使用 sendDirectionalMediaMessage 在超级群中发送多媒体消息给指定的频道中的单个或多个用户。

请在消息对象中设置超级群会话类型、超级群 ID 和 频道 ID。频道 ID 为空时默认向 RCDefault 频道发送消息。注意，Message 中不会保存接收用户 ID 列表。

```
String targetId = "目标 ID";
ConversationType conversationType = ConversationType.ULTRA_GROUP;
Uri localUri = Uri.parse("file://图片的路径");//图片本地路径，接收方可以通过 getThumUri 获取自动生成的缩略图 Uri
boolean mIsFull = true; //是否发送原图
ImageMessage mediaMessageContent = ImageMessage.obtain(localUri, mIsFull);
Message message = Message.obtain(targetId, conversationType, channelId, mediaMessageContent);

String[] userIds = new String[]{"id_01", "id_02"};

RongCoreClient.getInstance().sendDirectionalMediaMessage(message, userIds, null, null, new IRongCoreCallback.ISendMediaMessageCallback() {
@Override
public void onProgress(Message message, int i) {

}

@Override
public void onCanceled(Message message) {

}

@Override
public void onAttached(Message message) {

}

@Override
public void onSuccess(Message message) {

}

@Override
public void onError(final Message message, final IRongCoreEnum.CoreErrorCode errorCode) {

}
});
```

sendDirectionalMediaMessage 中直接提供了用于控制推送通知内容（pushContent）和推送附加信息（pushData）的参数。不过，如果您需要更精细地控制离线推送通知，例如标题、内容、图标、或其他第三方厂商个性化配置，请使用消息推送属性进行配置，详见自定义消息推送通知。

- 如果发送的消息属于 SDK 内置的媒体消息类型，例如 ImageMessage，这两个参数可设置为 `null`。一旦消息触发离线推送通知时，融云服务端会使用各个内置消息类型默认的 `pushContent` 值。关于各类型消息的默认推送通知内容，详见用户内容类消息格式。
- 如果消息类型为自定义消息类型，且需要支持离线推送通知，则必须向融云提供 `pushContent` 字段，否则用户无法收到离线推送通知。
- 如果消息类型为自定义消息类型，但不需要支持远程推送通知（例如通过自定义消息类型实现的 App 业务层操作指令），可将 `pushContent` 字段留空。
- `Message` 的推送属性配置 `MessagePushConfig` 的 `pushContent` 和 `pushData` 会覆盖此处配置，并提供更多配置能力，例如自定义推送通知的标题。详见自定义消息推送通知。

参数	类型	说明
message	Message	要发送的消息体。必填属性包括会话类型（conversationType），会话 ID（targetId），消息内容（content）。**注意**，超级群定向消息要求会话类型为 Conversation.ConversationType.GROUP。
userIds	String[]	需要接收消息的用户列表。
pushContent	String	修改或指定远程消息推送通知栏显示的内容。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见自定义消息推送通知。 • 如果希望使用融云默认推送通知内容，可以填 `null`。注意自定义消息类型无默认值。 • 如果要为该条消息指定个性化的离线推送通知，必须在以上任一处向融云提供 `pushContent` 字段内容。 • 如果您自定义的消息类型需要离线推送服务，必须在以上任一处向融云提供 `pushContent` 字段内容，否则用户无法收到离线推送通知。
pushData	String	远程推送附加信息。对端收到远程推送消息时，可通过以下方法获取该字段内容： `io.rong.push.notification.PushNotificationMessage#getPushData()` 。您也可以在 Message 的推送属性（MessagePushConfig）中配置，会覆盖此处配置，详见自定义消息推送通知。
callback	ISendMediaMessageCallback	发送媒体消息的回调

获取历史消息　　获取历史消息

更新时间:2024-08-30

- 如果您的应用/环境在 2022.10.13 日及以后开通超级群服务，超级群业务中会包含一个 ID 为 `RCDefault` 的默认频道。如果发消息时不指定频道 ID，则该消息会发送到 `RCDefault` 频道中。在获取 `RCDefault` 频道的历史消息时，需要传入该频道 ID。
- 如果您的应用/环境在 2022.10.13 日前已开通超级群服务，在发送消息时如果不指定频道 ID，则该消息不属于任何频道。获取历史消息时，如果不传入频道 ID，可获取不属于任何频道的消息。融云支持客户调整服务至最新行为。该行为调整将影响客户端、服务端收发消息、获取会话、清除历史消息、禁言等多个功能。如有需要，请提交工单咨询详细方案。

获取本地与远端历史消息

getMessages 方法先从本地获取历史消息，本地有缺失的情况下会从服务端同步缺失的部分。当本地没有更多消息的时候，会从服务端拉取。

```
interface IGetMessageCallbackEx{
void onComplete(List<Message> messageList, long syncTimestamp, boolean hasMoreMsg, IRongCoreEnum.CoreErrorCode errorCode);
void onFail(IRongCoreEnum.CoreErrorCode errorCode);
}

public void getMessages(final Conversation.ConversationType conversationType, final String targetId, final String channelId, final HistoryMessageOption
historyMessageOption, final IRongCoreCallback.IGetMessageCallbackEx callback)
```

参数	类型	说明
conversationType	ConversationType	会话类型
targetId	String	会话 ID
channel	String	超级群频道 ID
historyMsgOption	HistoryMessageOption	获取历史消息的配置选项。
callback	IRongCoreCallback.IGetMessageCallback<List<Message>>	获取历史消息的回调。

- `HistoryMessageOption` **说明**：

参数	说明
dataTime	时间戳，用于控制分页查询消息的边界。默认值为 `0`。
count	要获取的消息数量。如果 SDK < 5.4.1，范围为 [2-20]；如果 SDK ≧ 5.4.1，范围为 [2-100]；默认值为 `5`。
pullOrder	拉取顺序。`DESCEND`：降序，按消息发送时间递减的顺序，获取发送时间早于 `dataTime` 的消息，返回的列表中的消息按发送时间从新到旧排列。`ASCEND`： 升序，按消息发送时间递增的顺序，获取发送时间晚于 `dataTime` 的消息，返回的列表中的消息按发送时间从旧到新排列。

从本地数据库中获取消息

使用 `getHistoryMessages` 方法可分页查询指定会话存储在本地数据库中的历史消息，并返回消息对象列表。列表中的消息按发送时间从新到旧排列。

获取指定消息 ID 前的消息

异步获取会话中，从指定消息之前、指定数量的最新消息实体，返回消息实体 `Message` 对象列表。

```
public abstract void getHistoryMessages(
final Conversation.ConversationType conversationType,
final String targetId,
final int oldestMessageId,
final int count,
final String channelId,
final IRongCoreCallback.ResultCallback<List<Message>> callback);
```

`count` 参数表示返回列表中应包含多少消息。`oldestMessageId` 参数用于控制分页的边界。每次调用 `getHistoryMessages` 方法时，SDK 会以 `oldestMessageId` 参数指向的消息为界，继续在下一页返回指定数量的消息。如果需要获取会话中最新的 `count` 条消息，可以将 `oldestMessageId` 设置为 -1。

建议获取返回结果中最早一条消息的 ID，并在下一次调用时作为 `oldestMessageId` 传入，以便遍历整个会话的消息历史记录。

参数	类型	说明
conversationType	ConversationType	会话类型。
targetId	String	会话 ID。
oldestMessageId	long	最后一条消息的 ID。如需查询本地数据库中最新的消息，设置为 `-1`。
count	int	每页消息的数量。
channelId	String	超级群频道 ID。
callback	ResultCallback<List<Message>>	获取历史消息的回调，按照时间顺序从新到旧排列。

获取指定时间戳前后的消息

> 提示
>
> 超级群业务从 5.4.5 开始支持该功能。

获取会话中指定时间戳之前之后、指定数量的最新消息实体，返回消息实体 Message 对象列表。

```
public abstract void getHistoryMessages(
final Conversation.ConversationType conversationType,
final String targetId,
final String channelId,
final long sentTime,
final int before,
final int after,
final IRongCoreCallback.ResultCallback<List<Message>> resultCallback);
```

参数	类型	说明
conversationType	ConversationType	会话类型。
targetId	String	会话 ID。
channelId	String	超级群频道 ID。
timestamp	long	以此时间戳为界，获取早于或晚于该时间的历史消息。
before	int	需要获取的发送时间早于指定时间戳的消息数量。
after	int	需要获取的发送时间晚于指定时间戳的消息数量。
callback	ResultCallback<List<Message>>	获取历史消息的回调，按照时间顺序从新到旧排列。

获取指定单个消息类型的历史消息

以下方法从指定消息之前的特定类型的最新消息实体，异步返回消息实体 Message 对象列表。

```
public abstract void getHistoryMessages(
final Conversation.ConversationType conversationType,
final String targetId,
final String channelId,
final String objectName,
final int oldestMessageId,
final int count,
final IRongCoreCallback.ResultCallback<List<Message>> callback);
```

count 参数表示返回列表中应包含多少消息。oldestMessageId 参数用于控制分页的边界。每次调用 getHistoryMessages 方法时，SDK 会以 oldestMessageId 参数指向的消息为界，继续在下一页返回指定数量的消息。如果需要获取会话中最新的 count 条消息，可以将 oldestMessageId 设置为 -1。

建议获取返回结果中最早一条消息的 ID，并在下一次调用时作为 oldestMessageId 传入，以便遍历整个会话的消息历史记录。

参数	类型	说明
conversationType	ConversationType	会话类型。
targetId	String	会话 ID。
objectName	String	消息类型标识。内置消息类型的标识可参见消息类型概述。
oldestMessageId	long	最后一条消息的 ID。如需查询本地数据库中最新的消息，设置为 -1。
count	int	每页消息的数量。
channelId	String	超级群频道 ID。
callback	ResultCallback<List<Message>>	获取历史消息的回调，按照时间顺序从新到旧排列。

如果需要获取早于或晚于指定消息的历史消息，可以使用以下带 direction 参数的方法：

```
public abstract void getHistoryMessages(
final Conversation.ConversationType conversationType,
final String targetId,
final String channelId,
final String objectName,
final int baseMessageId,
final int count,
final RongCommonDefine.GetMessageDirection direction,
final IRongCoreCallback.ResultCallback<List<Message>> callback);
```

参数	类型	说明
conversationType	ConversationType	会话类型。
targetId	String	会话 ID。
channelId	String	超级群频道 ID。
objectName	String	消息类型标识。内置消息类型的标识可参见消息类型概述。
baseMessageId	int	以此消息 ID 为界，获取早于或晚于该消息的历史消息。
count	int	每页消息的数量。

参数	类型	说明
channelId	String	超级群频道 ID。
direction	RongCommonDefine.GetMessageDirection	FRONT表示获取早于传入时间戳的消息。BEHIND 表示获取晚于传入时间戳的消息。
callback	ResultCallback<List<Message>>	获取历史消息的回调，按照时间顺序从新到旧排列。

获取指定多个消息类型的历史消息

获取会话中，从指定消息之前或之后、指定数量的、多个消息类型的最新消息实体，异步返回消息实体 Message 对象列表。

```
public abstract void getHistoryMessages(
final Conversation.ConversationType conversationType,
final String targetId,
final String channelId,
final List<String> objectNames,
final long timestamp,
final int count,
final RongCommonDefine.GetMessageDirection direction,
final IRongCoreCallback.ResultCallback<List<Message>> callback);
```

参数	类型	说明
conversationType	ConversationType	会话类型。
targetId	String	会话 ID。
channelId	String	超级群频道 ID。
objectNames	List<String>	消息类型标识。内置消息类型的标识可参见消息类型概述。
timestamp	long	以此时间戳为界，获取早于或晚于该时间的历史消息。
count	int	每页消息的数量。
direction	RongCommonDefine.GetMessageDirection	FRONT表示获取早于传入时间戳的消息。BEHIND 表示获取晚于传入时间戳的消息。
callback	ResultCallback<List<Message>>	获取历史消息的回调，按照时间顺序从新到旧排列。

获取远端历史消息

使用 ChannelClient 的 getRemoteHistoryMessages 方法直接查询指定会话存储在**超级群消息云端存储**中的历史消息。

> 提示
>
> 用户是否可以获取在加入超级群之前的群聊历史消息取决于 App 在控制台的设置。默认新入群用户只能看到他们入群后的群聊消息。配置方法详见 开通超级群服务。

获取会话的远端历史消息

SDK 按照指定条件直接查询并获取**超级群消息云端存储**中的满足查询条件的历史消息。查询结果与本地数据库对比，排除重复的消息后，返回消息对象列表。返回的消息列表中的消息按发送时间从新到旧排列。因为默认该接口返回的消息会跟本地消息排重后返回，建议先使用 getHistoryMessages，在本地数据库消息全部获取完之后，再获取远端历史消息。否则可能会获取不到指定的部分或全部消息。

remoteHistoryMsgOption 中包含多个配置项，其中 count 与 dateTime 参数分别时获取历史消息的数量与分页查询时间戳。pullOrder 参数用于控制获取历史消息的时间方向，可选择获取早于或者晚于给定的 dateTime 的消息。includeLocalExistMessage 参数控制返回消息列表中是否需要包含本地数据库已存在的消息。

参数	类型	说明
conversationType	ConversationType	会话类型
channelId	String	频道 ID
targetId	String	会话 ID
remoteHistoryMsgOption	RemoteHistoryMsgOption	获取远端历史消息的配置选项。
callback	ResultCallback<List<Message>>	获取历史消息的回调。

- RemoteHistoryMsgOption **说明**：

参数	说明
dataTime	时间戳，用于控制分页查询消息的边界。默认值为 0。
count	要获取的消息数量。如果 SDK < 5.4.1，范围为 [2-20]；如果 SDK ≥ 5.4.1，范围为 [2-100]；默认值为 5。
pullOrder	拉取顺序。DESCEND：降序，按消息发送时间递减的顺序，获取发送时间早于 dataTime 的消息，返回的列表中的消息按发送时间从新到旧排列。ASCEND： 升序，按消息发送时间递增的顺序，获取发送时间晚于 dataTime 的消息，返回的列表中的消息按发送时间从旧到新排列。默认值为 1。
includeLocalExistMessage	是否包含本地数据库中的已有消息。true：包含，查询结果不与本地数据库排除重复，直接返回。false：不包含，查询结果先与本地数据库排除重复，只返回本地数据库中不存在的消息。默认值为 false。

RemoteHistoryMsgOption 默认按消息发送时间降序查询会话中的消息，且默认会并将查询结果与本地数据库对比，排除重复的消息后，再返回消息对象列表。在 includeLocalExistMessage 设置为 false 的情况下，建议 App 层先使用 getHistoryMessages，在本地数据库消息全部获取完之后，再获取远端历史消息，否则可能会获取不到指定的部分或全部消息。

如需按消息发送时间升序查询会话中的消息，建议获取返回结果中最新一条消息的 sentTime，并在下一次调用时作为 dateTime 的值传入，以便遍历整个会话的消息历史记录。升序查询消息一般可用于跳转到会话页面指定消息位置后需要查询更新的历史消息的场景。

```
Conversation.ConversationType conversationType= Conversation.ConversationType.ULTRA_GROUP;
String targetId="会话 Id";
String channelId="频道 Id";
RemoteHistoryMsgOption remoteHistoryMsgOption=new RemoteHistoryMsgOption();
remoteHistoryMsgOption.setDataTime(1662542712112L);//2022-09-07 17:25:12:112
remoteHistoryMsgOption.setOrder(HistoryMessageOption.PullOrder.DESCEND);
remoteHistoryMsgOption.setCount(20);

ChannelClient.getInstance().getRemoteHistoryMessages(conversationType, targetId, remoteHistoryMsgOption,
new IRongCoreCallback.ResultCallback<List<Message>>() {
/**
* 成功时回调
* @param messages 获取的消息列表
*/
@Override
public void onSuccess(List<Message> messages) {

}

/**
* 错误时回调。
* @param e 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {

}
});
```

强制从服务端获取特定批量消息

getBatchRemoteUltraGroupMessages 接口会强制从服务端获取对应的消息。

1. 0 < messages 个数 ≦ 20
2. messages 所有数据必须是超级群类型且为同一个会话
3. message 有效值为 ConversationType，targetId，channelId，messageUid，sentTime

```
matchedMsgList 从服务获取的消息列表
notMatchMsgList 非法参数或者从服务没有拿到对应消息

interface IGetBatchRemoteUltraGroupMessageCallback {
void onSuccess(List<Message> matchedMsgList, List<Message> notMatchedMsgList);

void onError(IRongCoreEnum.CoreErrorCode errorCode);
}

public void getBatchRemoteUltraGroupMessages(final List<Message> msgList,final IRongCoreCallback.IGetBatchRemoteUltraGroupMessageCallback callback)
```

超级群还支持通过消息 UID 从本地数据库中提取消息。详见获取历史消息 中的**通过消息 UID 获取消息**。

统计本地历史消息数量

> 提示
>
> ChannelClient 从 5.6.4 开始支持该能力。

您可以获取指定超级群在指定时间范围内的消息数量。该方法支持传入一个频道列表，如果不指定任何频道，则返回全部频道在本地的历史消息数量。

```
public abstract void getUltraGroupMessageCountByTimeRange(
String targetId,
String[] channelIds,
long startTime,
long endTime,
IRongCoreCallback.ResultCallback<Integer> callback);
```

搜索本地消息　　搜索本地消息

更新时间:2024-08-30

> SDK 从 5.3.4 版本 开始支持超级群本地消息搜索功能。

SDK 允许 App 通过关键词、用户 ID 等条件搜索指定超级群频道或多个频道中已拉取到本地的消息，支持按时间段搜索。

并非所有消息类型均支持关键字搜索：

- 内置的消息类型中文本消息（TextMessage ），文件消息（FileMessage ），和图文消息 `RichContentMessage` 类型默认实现了 `MessageContent#getSearchableWord` 方法。
- 自定义消息类型也可以支持关键字搜索，需要您参考文档自行实现。详见自定义消息类型。

请注意，消息搜索仅查询本地数据库。超级群默认每次连接时只同步频道最后一条消息，因此**超级群业务本地消息记录可能不完整**，需要 App 自行将消息拉取到本地（一般在进入会话页面时主动获取历史消息）。

> 提示
>
> 融云提供搜索超级群历史消息记录的 IM Server API。详见服务端文档搜索超级群消息 。

搜索会话

按关键字搜索本地所有会话。返回符合条件的会话列表 SearchConversationResult 。请注意，超级群业务中单个会话（conversation）仅对应单个超级群频道。

```
String keyword = "搜索的关键字";
Conversation.ConversationTypes[] conversationTypes = {ConversationType.ULTRA_GROUP};
String[] messageTypeObjectNames = {"RC:TxtMsg"};

ChannelClient.getInstance().searchConversationForAllChannel(keyword, conversationTypes, messageTypeObjectNames,
new IRongCoreCallback.ResultCallback<List<SearchConversationResult>>() {

@Override
public void onSuccess(List<SearchConversationResult> searchConversationResults) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode errorCode) {

}
});
```

如果搜索条件与搜索结果包含多个会话类型，您可以通过返回结果中 Conversation 对象的会话类型（ConversationType）字段进行分类或筛选。

参数	类型	说明
keyword	String	搜索的关键字
conversationTypes	Conversation.ConversationType[]	会话类型列表，包含 ConversationType ，例如 `ConversationType.ULTRA_GROUP`、`ConversationTypes.PRIVATE`，`ConversationTypes.GROUP`。
objName	String[]	消息类型列表，默认仅支持内置类型 `RC:TxtMsg`（文本消息）、`RC:FileMsg`（文件消息）、`RC:ImgTextMsg`（图文消息）。
callback	IRongCoreCallback.ResultCallback<List<SearchConversationResult>>	回调。搜索结果为 SearchConversationResult 列表。请注意，针对超级群，单个会话（conversation）对应单个超级群频道。

搜索单个频道的消息

获取包含关键词的会话列表后，可以搜索指定单个会话中符合条件的消息。在实现超级群消息搜索时，App 可以先通过搜索会话获取符合条件的超级群会话列表，再搜索指定频道中符合条件的消息。

搜索单个频道的消息需要 App 同时提供超级群 ID（targetId） 与 频道 ID（channelId）。

根据关键字搜索指定频道的消息

在本地存储中根据关键字搜索指定频道会话中的消息，支持搜索指定时间点之前的历史消息记录。回调中分页返回包含指定关键字的消息列表。

```
ConversationType conversationType = ConversationType.ULTRA_GROUP;
String targetId = "超级群 ID";
String channelId = "超级群频道 ID";
String keyword = "搜索的关键字";
int count = 20;
long beginTime = 122323344;

ChannelClient.getInstance().searchMessages(conversationType, targetId, channelId, keyword, count, beginTime,
new IRongCoreCallback.ResultCallback<List<Message>>() {

@Override
public void onSuccess(List<Message> messages) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode errorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型。超级群会话传入 `ConversationType.ULTRA_GROUP`。
targetId	String	超级群 ID。
channelId	String	超级群频道 ID。
keyword	String	搜索的关键字。
count	int	每页的数量，每页数量建议最多 100 条。传 `0` 时返回所有搜索到的消息。
beginTime	long	查询记录的起始时间。传 `0` 时从最新消息开始搜索。非 `0` 时从该时间往前搜索。
callback	IRongCoreCallback.ResultCallback<List<Message>>	回调。搜索结果为 Message 列表。

根据关键字搜索指定时间段内指定频道的消息

SDK 支持将关键字搜索的范围限制在指定时间段内。回调中分页返回包含指定关键字和时间段要求的消息列表。

```
ConversationType conversationType = ConversationType.ULTRA_GROUP;
String targetId = "超级群 ID";
String channelId = "超级群频道 ID";
String keyword = "123";
long startTime = 0;
long endTime = 1585815113;
int offset = 0;
int limit = 80;

ChannelClient.getInstance().searchMessages(conversationType, targetId, channelId, keyword, startTime, endTime, offset, limit,
new IRongCoreCallback.ResultCallback<List<Message>>(){

@Override
public void onSuccess(List<Message> messages) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {

}
});
```

`limit` 参数控制返回的搜索结果数量，取值范围为 [1-100]。超过 100 默认使用最大值 100。

参数	类型	说明
conversationType	ConversationType	会话类型。超级群会话传入 `ConversationType.ULTRA_GROUP`。
targetId	String	超级群 ID
channelId	String	超级群频道 ID。
keyword	String	搜索的关键字
startTime	long	开始时间
endTime	long	结束时间
offset	int	偏移量
limit	int	返回的搜索结果数量，limit 需大于 0。最大值为 100。超过 100 时默认返回 100 条。
callback	IRongCallback.ResultCallback<List<Message>>	回调。搜索结果为 Message 列表。

根据用户 ID 搜索指定频道的消息

在本地存储中根据搜索来自指定用户 ID 的消息，支持搜索指定时间点之前的历史消息记录。回调中分页返回包含符合条件的消息列表。

```
ConversationType conversationType = ConversationType.ULTRA_GROUP;
String targetId = "超级群 ID";
String channelId = "超级群频道 ID";
String userId = "查询的用户 Id";
int count = 20;
long beginTime = 1585815113;

ChannelClient.getInstance().searchMessagesByUser(conversationType, targetId, channelId, userId, count, beginTime,
new IRongCoreCallback.ResultCallback<Message>() {

@Override
public void onSuccess(List<Message> messages) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode errorCode) {

}
});
```

参数	类型	说明
conversationType	ConversationType	会话类型。超级群会话传入 `ConversationType.ULTRA_GROUP`。
targetId	String	超级群 ID。
channelId	String	超级群频道 ID。
userId	String	要查询的用户 ID。
count	int	返回的搜索结果数量。最大值为 100。超过 100 时默认返回 100 条。
beginTime	long	查询记录的起始时间。传 `0` 时从最新消息开始搜索。非 `0` 时从该时间往前搜索。
callback	IRongCallback.ResultCallback<List<Message>>	回调。搜索结果为 Message 列表。

搜索多个频道的消息

App 如需搜索指定超级群的消息，并已持有超级群 ID（targetId），可以直接使用以下接口搜索本地已接收、已存储的消息。符合搜索条件的消息可能来自多个频道。

根据关键字搜索指定超级群本地所有频道的消息

搜索指定超级群存储在本地的消息，支持搜索指定时间点之前的历史消息记录。回调中分页返回包含指定关键字的消息列表。

```
String targetId = "超级群 ID";
ConversationType conversationType = ConversationType.ULTRA_GROUP;
String keyword = "搜索的关键字";
int count = 20;
long timestamp = "122323344";

ChannelClient.getInstance().searchMessageForAllChannel(targetId, conversationType, keyword, count, timestamp,
new IRongCoreCallback.ResultCallback<List<Message>>() {

@Override
public void onSuccess(List<Message> messages) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode errorCode) {

}
});
```

参数	类型	说明
targetId	String	超级群 ID。
conversationType	ConversationType	会话类型。超级群会话传入 `ConversationType.ULTRA_GROUP`。
keyword	String	搜索的关键字。
count	int	每页的数量，每页数量建议最多 100 条。传 `0` 时返回所有搜索到的消息。
beginTime	long	查询记录的起始时间。传 `0` 时从最新消息开始搜索。非 `0` 时从该时间往前搜索。
callback	IRongCoreCallback.ResultCallback<List<Message>>	回调。搜索结果为 Message 列表。

根据关键字搜索指定超级群指定时间段内本地所有频道的消息

在搜索指定超级群存储在本地的消息时，SDK 支持将关键字搜索的范围限制在指定时间段内。回调中分页返回包含指定关键字和时间段要求的消息列表。

```
String targetId = "超级群 ID";
ConversationType conversationType = ConversationType.ULTRA_GROUP;
String keyword = "123";
long startTime = 0;
long endTime = 1585815113;
int offset = 0;
int limit = 80;

ChannelClient.getInstance().searchMessageByTimestampForAllChannel(conversationType, targetId, keyword, startTime, endTime, offset, limit,
new IRongCoreCallback.ResultCallback<List<Message>>(){

@Override
public void onSuccess(List<Message> messages) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {

}
});
```

limit 参数控制返回的搜索结果数量，取值范围为 [1-100]。超过 100 默认使用最大值 100。

参数	类型	说明
targetId	String	超级群 ID
conversationType	ConversationType	会话类型。超级群会话传入 ConversationType.ULTRA_GROUP。
keyword	String	搜索的关键字
startTime	long	开始时间
endTime	long	结束时间
offset	int	偏移量
limit	int	返回的搜索结果数量，limit 需大于 0。最大值为 100。超过 100 时默认返回 100 条。
callback	IRongCallback.ResultCallback<List<Message>>	回调。搜索结果为 Message 列表。

根据关键字搜索指定超级群下多个频道的本地历史消息

> 提示
>
> SDK 从 5.6.2 版本开始提供该方法。

请确保传入合法的 Channel ID 数组。Channel ID 数组不可为空，ID 数量不可超过 50 个，数组中的 Channel ID 均必须为合法有效的值（大小写英文字母与数字），不支持 Channel ID 为空字符串。支持从最新消息开始搜索，或者搜索早于 startTime 的消息。每次搜索最大 100 条，如果还有更多的消息，可以将 startTime 设为获取到的消息数组最后一条消息的时间戳，然后继续调用接口。

```
/**
* 根据关键字搜索指定超级群下多个频道的本地历史消息
*
* <p>注意：如果需要自定义消息也能被搜索到，需要在自定义消息中实现 {@link MessageContent#getSearchableWord()} 方法。
*
* @param conversationType 指定的会话类型。
* @param targetId 指定的会话 id。
* @param channelIds 消息所属会话的业务标识。(0 < channelIds的数量 <= 50, 不符合则返回 {@link
* IRongCoreEnum.CoreErrorCode#RC_INVALID_PARAMETER_CHANNEL_ID})
* @param keyword 关键词。(0<长度<=1000)
* @param startTime 查询记录的起始时间， 传 0 时从最新消息开始搜索，从该时间往前搜索。
* @param limit 返回的搜索结果数量 {@code 0 < limit <= 100}，如果 {@code limit > 100}，则返回 100。
* @param resultCallback 搜索结果回调。
* @since 5.6.2
*/
public abstract void searchMessagesForChannels(
final Conversation.ConversationType conversationType,
final String targetId,
final String[] channelIds,
final String keyword,
final long startTime,
final int limit,
final IRongCoreCallback.ResultCallback<List<Message>> resultCallback);
```

根据发送者用户 ID 搜索指定超级群下多个频道的本地历史消息

> 提示
>
> SDK 从 5.6.2 版本开始提供该方法。

请确保传入合法的 Channel ID 数组。Channel ID 数组不可为空，ID 数量不可超过 50 个，数组中的 Channel ID 均必须为合法有效的值（大小写英文字母与数字），不支持 Channel ID 为空字符串。支持从最新消息开始搜索，或者搜索早于 startTime 的消息。每次搜索最大 100 条，如果还有更多的消息，可以将 startTime 设为获取到的消息数组最后一条消息的时间戳，然后继续调用接口。

```
/**
* 根据会话id、会话业务标识、用户ID等搜索指定会话中的消息。
*
* <p>注意：如果需要自定义消息也能被搜索到，需要在自定义消息中实现 {@link MessageContent#getSearchableWord()} 方法。
*
* @param conversationType 指定的会话类型。
* @param targetId 指定的会话 id。
* @param channelIds 消息所属会话的业务标识。(0 < channelIds的数量 <= 50, 不符合则返回 {@link
* IRongCoreEnum.CoreErrorCode#RC_INVALID_PARAMETER_CHANNEL_ID})
* @param userId 用户ID。
* @param startTime 查询记录的起始时间， 传 0 时从最新消息开始搜索，从该时间往前搜索。
* @param limit 返回的搜索结果数量 {@code 0 < limit <= 100}，如果 {@code limit > 100}，则返回 100。
* @param resultCallback 搜索结果回调。
* @since 5.6.2
*/
public abstract void searchMessagesByUserForChannels(
final Conversation.ConversationType conversationType,
final String targetId,
final String[] channelIds,
final String userId,
final long startTime,
final int limit,
final IRongCoreCallback.ResultCallback<List<Message>> resultCallback);
```

根据发送者用户 ID 搜索指定超级群下所有频道的本地历史消息

> 提示
>
> SDK 从 5.6.2 版本开始提供该方法。

支持从最新消息开始搜索，或者搜索早于 startTime 的消息。每次搜索最大 100 条，如果还有更多的消息，可以将 startTime 设为获取到的消息数组最后一条消息的时间戳，然后继续调用接口。

```
/**
* 根据会话id、用户id等搜索指定会话中的消息。(包含所有的 ChannelId)
*
* <p>注意：如果需要自定义消息也能被搜索到，需要在自定义消息中实现 {@link MessageContent#getSearchableWord()} 方法。
*
* @param conversationType 指定的会话类型。
* @param targetId 指定的会话 id。
* @param userId 用户ID。
* @param startTime 查询记录的起始时间， 传 0 时从最新消息开始搜索，从该时间往前搜索。
* @param limit 返回的搜索结果数量 {@code 0 < limit <= 100}，如果 {@code limit > 100}，则返回 100。
* @param resultCallback 搜索结果回调。
* @since 5.6.2
*/
public abstract void searchMessagesByUserForAllChannel(
final Conversation.ConversationType conversationType,
final String targetId,
final String userId,
final long startTime,
final int limit,
final IRongCoreCallback.ResultCallback<List<Message>> resultCallback);
```

获取未读消息数 获取未读消息数

更新时间:2024-08-30

超级群业务支持从客户端 SDK 获取未读消息数，具体如下：

- 当前用户加入的所有超级群、指定超级群、或指定频道未读消息数。
- 当前用户加入的所有超级群、指定超级群、或指定频道的未读 @ 消息数。
- 按免打扰级别获取总未读消息数

返回的未读数最大值为 999。如果实际未读数超过 999，接口仍返回 999。

获取多个超级群的未读消息数

SDK 支持获取当前用户加入的所有超级群中未读消息数与未读 @ 消息数。

批量获取当前用户的超级群的未读消息数

> 提示
>
> SDK 从 5.4.6 版本开始支持该接口。仅在 ChannelClient 中提供。

在社群应用场景中，App 可能需要实时显示用户所在的多个超级群下所有频道的最新未读消息数据，可以使用 `getUltraGroupConversationUnreadInfoList` 一次获取最多 20 个超级群下所有频道的未读数据。具体包含：

- 超级群频道的未读消息数
- 超级群频道的未读 @ 消息数
- 超级群频道中仅 @ 当前用户的未读 @ 消息数
- 超级群频道的免打扰级别

```
ChannelClient.getInstance().getUltraGroupConversationUnreadInfoList(targetIds,

new IRongCoreCallback.ResultCallback<List<ConversationUnreadInfo>>() {

@Override
public void onSuccess(List<ConversationUnreadInfo> ConversationUnreadInfos) {
if (ConversationUnreadInfos == null) return;
for (ConversationUnreadInfo unreadInfo : ConversationUnreadInfos) {
// 获取超级群会话类型
Conversation.ConversationType type = unreadInfo.getType();
String targetId = unreadInfo.getTargetId();
// 获取超级群频道 ID
String channelId = unreadInfo.getChannelId();
//获取频道内的未读消息数
int unreadMessageCount = unreadInfo.getUnreadMessageCount();
//获取频道内的 @ 未读消息数
int unreadMentionedCount =
unreadInfo.getUnreadMentionedCount();
//获取频道内仅自己被 @ 的未读消息数（@ 我）
int mentionedMeCount = unreadInfo.getUnreadMentionedMeCount();
//获取频道的免打扰级别设置
IRongCoreEnum.PushNotificationLevel pushNotificationLevel = getPushNotificationLevel();

}
//该回调在非 UI 线程返回，如果需要做 UI 操作，请切换至 UI 线程

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode ErrorCode) {

}
});
```

获取当前用户的超级群未读消息数总和

获取当前登录用户加入的超级群所有会话的未读消息数的总和。

```
class ResultCallback {
onSuccess(Integer count)
onError(ErrorCode code)
}

public void getUltraGroupAllUnreadCount(final IRongCoreCallback.ResultCallback<Integer> callback)
```

获取当前用户的超级群未读 @ 消息数总和

获取当前登录用户加入的超级群所有会话的未读 @ 消息数的总和。

```
class ResultCallback {
onSuccess(Integer count)
onError(ErrorCode code)
}

public void getUltraGroupAllUnreadMentionedCount(IRongCoreCallback.ResultCallback<Integer> callback) {}
```

获取单个超级群的未读消息数

SDK 支持获取指定超级群或频道中的未读消息数和未读 @ 消息数。

获取指定单个超级群的未读消息数

获取当前用户在指定超级群中的未读消息数。

```
class ResultCallback {
onSuccess(Integer count)
onError(ErrorCode code)
}

public void getUltraGroupUnreadCount(String targetId, IRongCoreCallback.ResultCallback<Integer> callback) {}
```

获取指定单个超级群的未读 @ 消息数

获取当前用户在指定超级群中的未读 @ 消息数。

```
class ResultCallback {
onSuccess(Integer count)
onError(ErrorCode code)
}

public void getUltraGroupUnreadMentionedCount(final String targetId, final IRongCoreCallback.ResultCallback<Integer> callback)
```

获取指定频道的未读消息数

获取当前用户在超级群会话指定的频道中的未读消息数。

```
class ResultCallback {
onSuccess(Integer count)
onError(ErrorCode code)
}

public void getUnreadCount(final Conversation.ConversationType conversationType,
final String targetId,
final String channelId,
final IRongCoreCallback.ResultCallback<Integer> callback)
```

获取指定频道的未读 @ 消息数

App 直接从 Conversation 对象上获取该频道未读 @ 消息数。

- `getUnreadMentionedCount()`：当前频道中“@所有人”与“@当前用户”的未读消息数之和。
- `getUnreadMentionedMeCount()`：当前频道中“@当前用户”的未读消息数。要求 SDK 版本 ≥ 5.4.5。

具体示例可参见获取频道列表。

按免打扰级别获取超级群的总未读消息数

> 提示
>
> SDK 从 5.2.5 版本开始支持该接口。仅在 ChannelClient 中提供。

获取已设置指定免打扰级别的会话及频道的总未读消息数。SDK 将按照传入的免打扰级别配置查找，再返回该会话下符合设置的频道的总未读消息数。

```
PushNotificationLevel[] levels = {PushNotificationLevel.PUSH_NOTIFICATION_LEVEL_MENTION}

ChannelClient.getInstance().getUltraGroupUnreadCount(targetId, levels,

new ResultCallback<Integer>() {

@Override
public void onSuccess(Integer unReadCount) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode ErrorCode) {

}
});
```

参数	类型	说明
targetId	String	会话 ID
levels	PushNotificationLevel[]	免打扰类型数组。详见免打扰功能概述。
callback	ResultCallback<Integer>	回调接口

获取成功后，callback 中会返回未读消息数（unReadCount）。

按免打扰级别获取超级群的总未读 @ 消息数

> 提示
>
> SDK 从 5.2.5 版本开始支持该接口。仅在 ChannelClient 中提供。

获取已设置指定免打扰级别的会话及频道的总未读 @ 消息数。SDK 将按照传入的免打扰级别配置查找，再返回该会话下符合设置的频道的总未读 @ 消息数。

```
PushNotificationLevel[] levels = {PushNotificationLevel.PUSH_NOTIFICATION_LEVEL_MENTION}

ChannelClient.getInstance().getUltraGroupUnreadMentionedCount(targetId, levels,

new ResultCallback<Integer>() {

@Override
public void onSuccess(Integer unReadCount) {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode ErrorCode) {

}
});
```

参数	类型	说明
targetId	String	会话 ID
levels	PushNotificationLevel[]	免打扰类型数组。详见免打扰功能概述。
callback	ResultCallback<Integer>	回调接口

获取成功后，callback 中会返回未读消息数（unReadCount）。

如果您希望获取多个类型会话的未读消息总数，可参见处理会话未读消息数中介绍的方法。

处理未读 @ 消息 处理未读 @ 消息

更新时间:2024-08-30

本页面描述了在超级群业务如何实现获取全部未读 @ 消息、跳转到指定未读 @ 消息等功能。

> 提示
>
> 本页面功能依赖融云服务端保存的超级群会话「未读 @ 消息摘要」数据。「@所有人」消息的摘要数据固定存储 7 天（不支持调整）。其他类型 @ 消息的摘要数据与超级群历史消息存储时长一致。

获取全部未读 @ 消息

> 提示
>
> 要求 SDK 版本 ≧ 5.2.5。

您需要先获取未读 @ 消息的摘要信息（MessageDigestInfo）列表，再从远端提取未读的 @ 消息的完整内容。

例如，App 希望获取仅展示未读 @ 消息的场景，步骤如下：

1. 获取超级群频道下的未读 @ 消息摘要，最多可获取 50 条（count 取值范围为 [1-50]）。成功回调中会返回 MessageDigestInfo 的列表。每个 MessageDigestInfo 对象包含一条未读消息的摘要信息。使用 MessageDigestInfo 中的摘要信息构造 Message 列表。每个消息对象需包含从 MessageDigestInfo 中获取的 ConversationType，targetId，channelId，messageUid，sentTime。

```
String targetId = "会话 Id";
String channelId = "频道 ID";
long sendTime = 1662542712112L;
int count = 20;

ChannelClient.getInstance().getUltraGroupUnreadMentionedDigests(targetId, channelId, sendTime, count,
new IRongCoreCallback.ResultCallback<List<MessageDigestInfo>>() {
@Override
public void onSuccess(List<MessageDigestInfo> messageDigestInfos) {
List<Message> msgList = new ArrayList<>();
for (MessageDigestInfo info : messageDigestInfos) {
// 可使用 MessageDigestInfo 的属性筛选 @ 消息摘要
Message message = new Message();
message.setConversationType(info.getConversationType());
message.setTargetId(info.getTargetId());
message.channelId = info.getChannelId();
message.setUId(info.getMessageUid());
message.setSentTime(info.getSentTime());
// 从 5.6.0 开始，MessageDigestInfo 携带消息类型的唯一标识 ObjectName
message.setObjectName(info.getObjectName());
msgList.add(message);
}}

@Override
public void onError(IRongCoreEnum.CoreErrorCode errorCode) {

}});
```

2. 您可以使用 getBatchRemoteUltraGroupMessages(msgList, callback) 方法，传入上一步构建的消息列表，从远端批量提取 @ 消息的完整内容。

```
ChannelClient.getInstance().getBatchRemoteUltraGroupMessages(msgList,
new IRongCoreCallback.IGetBatchRemoteUltraGroupMessageCallback(
@Override
public void onSuccess(List<Message> matchedMsgList, List<Message> notMatchedMsgList) {}

@Override
public void onError(IRongCoreEnum.CoreErrorCode errorCode) {}

));
```

定位到指定未读 @ 消息

> 提示
>
> 要求 SDK 版本 ≧ 5.2.5。

利用获取超级群会话未读 @ 消息摘要列表接口（getUltraGroupUnreadMentionedDigests），可以实现从会话列表进入会话页面后定位到特定 @ 消息的位置。

1. 获取超级群频道下的未读 @ 消息摘要，最多可获取 50 条（count 取值范围为 [1-50]）。每个 MessageDigestInfo 对象包含一条未读 @ 消息的摘要信息。

```
public void getUltraGroupUnreadMentionedDigests(
String targetId,
String channelId,
long sendTime,
int count,
IRongCoreCallback.ResultCallback<List<MessageDigestInfo>> callback);
```

2. 构造 HistoryMessageOption 对象用于获取历史消息。从 MessageDigestInfo 中取出 sendTime，作为 HistoryMessageOption 对象的 DataTime。

```
HistoryMessageOption option = new HistoryMessageOption();
option.setOrder(HistoryMessageOption.PullOrder.DESCEND);
option.setCount(count);
option.setDataTime(sendTime);
```

3. （可选）修正 HistoryMessageOption 的 dataTime。getMessages 的查询结果中不会包含 message.sentTime == dataTime 的消息。如有需要，可根据 PullOrder 分别对 dataTime 做以下修正，以保证查询结果包含您想要的消息：

   - HistoryMessageOption.PullOrder.DESCEND: dataTime = message.sentTime + 1
   - HistoryMessageOption.PullOrder.ASCEND : dataTime = message.sentTime - 1

4. 使用超级群 ChannelClient 的获取历史消息接口 getMessages，可以获取特定 @ 消息前、后的历史消息。

```
public void getMessages(
final Conversation.ConversationType conversationType,
final String targetId,
final String channelId,
final HistoryMessageOption historyMessageOption,
final IRongCoreCallback.IGetMessageCallbackEx getMessageCallback);
```

何时清空未读 @ 消息

未读数需要在会话页面清除:

- 会话页面即将消失时
- 处于会话页面, app返回前台时
- 已经加载到最新消息时

> 提示
>
> 当调用 syncUltraGroupReadStatus 成功后，客户端本地 Conversation 的 firstUnreadMsgSendTime 会变为 0。服务端保存的第一条未读时间会变为 0，保存的未读 @ 消息摘要列表也会清零。

清除消息未读状态　　清除消息未读状态

更新时间:2024-08-30

超级群业务可在多个客户端之间同步消息阅读状态。

同步消息已读状态

调用同步已读状态接口会同时清除本地与服务端记录的消息的未读状态，同时服务端会将最新状态同步给同一用户账号的其他客户端。

- 如果指定了频道 ID（`channelId`），则标记该频道所有消息为全部已读，并同步其他客户端。
- 如果频道 ID 为空，则标记该超级群会话下所有不属于任何频道的消息为全部已读，并同步其他客户端。

> 提示
>
> 超级群暂不支持按时间戳同步已读状态。调用 syncUltraGroupReadStatus 会按指定参数的要求标记全部消息为已读。时间戳参数（timestamp）未使用，可传入任意数字。

```
class OperationCallback {
void onSuccess()
void onError(ErrorCode code)
}

public void syncUltraGroupReadStatus(final String targetId, final String channelId, final long timestamp, final IRongCoreCallback.OperationCallback callback)
```

监听消息已读时间

```
interface UltraGroupReadTimeListener {
/**
超级群已读时间同步

@param targetId 会话 ID
@param channelId 频道 ID
@param readTime 已读时间（服务端将未读数清零的时间）
*/
void onUltraGroupReadTimeReceived(String targetId, String channelId, long time);
}

public void setUltraGroupReadTimeListener(IRongCoreListener.UltraGroupReadTimeListener listener)
```

删除消息　　删除消息

更新时间:2024-08-30

超级群会话消息存储在服务端（免费存储 7 天）和用户设备本地数据库。App 用户通过客户端 SDK 删除自己的历史消息，支持仅从本地数据库删除消息、或仅从融云服务端删除消息。

提示

- 客户端的删除消息的操作均指从当前登录用户的历史消息记录中删除消息，不影响会话中其他用户的历史消息记录。
- 如果 App 的管理员或者某普通用户希望在该 App 中彻底删除一条消息，例如在所有超级群成员的聊天记录中删除一条消息，应使用客户端或服务端的 撤回消息功能。消息成功撤回后，原始消息内容会在所有用户的本地与服务端历史消息记录中删除。

功能	本地/服务端	API
从本地删除全部频道的消息（时间戳）	仅从本地删除	deleteUltraGroupMessagesForAllChannel
从本地删除指定频道的消息（时间戳）	仅从本地删除	deleteUltraGroupMessages
从服务端删除指定频道的消息（时间戳）	仅从服务端删除	deleteRemoteUltraGroupMessages
从本地和远端删除消息（消息对象）	同时从本地和服务端删除	deleteRemoteMessage

从本地删除全部频道的消息（时间戳）

删除本地数据库删除所有频道指定时间戳之前的历史消息。需提供 Unix 时间戳，精确到毫秒。单次操作仅针对单个超级群，不支持一次删除多个超级群中的消息。

如果 App 用户希望从超级群在当前设备上存储的所有历史记录中删除一段时间的记录，可以使用 deleteUltraGroupMessagesForAllChannel 删除所有频道中早于某个时间点（timestamp）的消息。时间戳为 0 表示使用当前时间戳，会从本地清除全部消息。服务端保存的该用户的历史消息记录不受影响。如果该用户从服务端获取历史消息，可能会获取到在本地已删除的消息。

```
String targetId = "会话 Id";
String timestamp = 0;

ChannelClient.getInstance().deleteUltraGroupMessagesForAllChannel(targetId, timestamp,
new IRongCoreCallback.ResultCallback<Boolean>() {
/**
* 成功回调
*/
@Override
public void onSuccess(Boolean bool) {

}
/**
* 失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

从本地删除指定频道的消息（时间戳）

删除本地数据库删除指定单个频道指定时间戳之前的历史消息。需提供 Unix 时间戳，精确到毫秒。单次操作仅针对单个超级群，不支持一次删除多个超级群中的消息。

如果 App 用户希望从超级群在当前设备上存储的频道消息历史记录中删除一段时间的记录，可以使用 deleteUltraGroupMessages 删除早于某个时间点（timestamp）的消息。时间戳为 0 表示使用当前时间戳，会从本地清除指定频道中的全部消息。服务端保存的该用户的历史消息记录不受影响。如果该用户从服务端获取历史消息，可能会获取到在本地已删除的消息。

```
String targetId = "超级群 ID";
String channelId = "频道 ID";
String recordTime = 0;

ChannelClient.getInstance().deleteUltraGroupMessages(targetId, channelId, timestamp,
new IRongCoreCallback.ResultCallback<Boolean>() {
/**
* 获取成功回调
*/
@Override
public void onSuccess(Boolean bool) {

}
/**
* 失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

从服务端删除指定频道的消息（时间戳）

App 用户从本地删除消息后，如果再从服务端获取历史消息，可能会获取到在本地已删除的消息。如果希望删除服务端历史消息记录，可以使用以下接口，从服务端历史消息记录中删除指定单个频道删除早于某个时间点（timestamp）的消息的历史消息。

timestamp 为 Unix 时间戳，精确到毫秒。时间戳为 0 表示使用当前时间戳，会清除指定频道中的全部消息。单次操作仅针对单个超级群，不支持一次删除多个超级群中的消息。该接口仅清除当前用户在服务端历史消息，本地保存的用户历史消息记录不受影响。

> 提示
>
> 启用超级群服务后，默认自动启用超级群历史消息存储，免费存储 7 天内的超级群消息。

```
String targetId = "超级群 ID";
String channelId = "频道 ID";
String timestamp = 0;

ChannelClient.getInstance().deleteRemoteUltraGroupMessages(targetId, channelId, timestamp,
new IRongCoreCallback.ResultCallback<Boolean>() {
/**
* 获取成功回调
*/
@Override
public void onSuccess(Boolean bool) {

}
/**
* 失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

从本地和远端删除指定频道的消息（消息对象）

> 提示
>
> 启用超级群服务后，默认自动启用超级群历史消息存储，免费存储 7 天内的超级群消息。

如果 App 用户希望从自己的超级群会话的历史记录中删除一组消息，可以使用以下方法，传入需要被删除的消息对象 Message 列表。请确保所提供的消息均属于同一超级群频道（Message#channelId）。一次最多删除 100 条消息。

删除成功后，该用户无法从本地数据库获取消息。如果从服务端获取历史消息，也无法获取到已删除的消息。

```
ConversationType conversationType = ConversationType.ULTRA_GROUP;
String targetId = "会话 Id";
Message[] messages = {message1, message2};

RongCoreClient.getInstance().deleteRemoteMessages(conversationType, targetId, messages, new IRongCoreCallback.OperationCallback() {
/**
* 删除消息成功回调
*/
@Override
public void onSuccess() {

}
/**
* 删除消息失败回调
* @param errorCode 错误码
*/
@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

修改消息　　修改消息

更新时间:2024-08-30

用户在成功发送超级群消息后，可以主动修改已发送消息的消息内容。

修改本端用户已发消息的内容

本端用户发送消息成功后，服务端会返回消息的 UID。如果需要修改消息内容，可以使用 modifyUltraGroupMessage ，传入待修改消息的 Message UID 和新的消息内容进行修改。消息被修改后，Message#isHasChanged() 会返回 true。**注意**：消息类型无法修改。如果改前为文本消息，则传入的新消息内容必须为 TextMessage 类型。无法修改他人发送的消息。

```
TextMessage newContent = TextMessage.obtain("修改后的文本消息内容");

ChannelClient.getInstance().modifyUltraGroupMessage(msgUid, newContent,

new IRongCoreCallback.OperationCallback() {

@Override
public void onSuccess() { }

@Override
public void onError(IRongCoreEnum.CoreErrorCode ErrorCode) { }
});
```

监听远端用户的消息变更

使用 setUltraGroupMessageChangeListener 设置 UltraGroupMessageChangeListener 监听器。远端用户修改消息时，应用程序可以通过 onUltraGroupMessageModified 方法收到通知。消息被修改后，Message#isHasChanged() 返回 true。

```
//超级群消息变化通知
interface UltraGroupMessageChangeListener {
//消息扩展更新，删除
void onUltraGroupMessageExpansionUpdated(List<Message> messages);
//消息内容发生变更
void onUltraGroupMessageModified(List<Message> messages);
//消息撤回
void onUltraGroupMessageRecalled(List<Message> messages);
}

//设置超级群消息变化监听
public void setUltraGroupMessageChangeListener(IRongCoreListener.UltraGroupMessageChangeListener listener)
```

撤回消息　　撤回消息

更新时间:2024-08-30

超级群业务中，消息发送方可撤回已发送成功的消息。撤回成功后，服务端即删除原始消息。

默认情况下，融云对撤回消息的操作者不作限制。如需限制，可考虑以下方案：

- App 客户端自行限制撤回消息的操作者。例如，不允许 App 业务中的普通用户撤回他人发送的消息，允许 App 业务中的管理员角色撤回他人发送的消息。
- 如需避免用户撤回非本人发送的消息，可以提交工单 申请打开**IMLib SDK 只允许撤回自己发送的消息**。从融云服务端进行限制，禁止用户撤回非本人发送的消息。

撤回指定消息

撤回指定消息，只有已发送成功的消息可被撤回。撤回的结果通过 `callback` 回调。

方法说明

```
class OperationCallback {
onSuccess()
onError(ErrorCode code)
}
//撤回消息
public void recallUltraGroupMessage(final Message message, final IRongCoreCallback.ResultCallback<RecallNotificationMessage> callback)
```

参数说明

参数	类型	说明
message	Message	消息对象
callback	IRongCoreCallback.ResultCallback<RecallNotificationMessage>	接口回调

撤回指定消息并删除原始消息数据

撤回指定消息，并删除移动端发送方、接收方的原始消息数据。撤回的结果通过 `callback` 回调。

方法说明

```
class OperationCallback {
onSuccess()
onError(ErrorCode code)
}
//撤回消息
public void recallUltraGroupMessage(final Message message, final boolean isDelete, final IRongCoreCallback.ResultCallback<RecallNotificationMessage> callback)
```

参数说明

参数	类型	说明
message	Message	消息对象
isDelete	Boolean	指定移动端发送方与接收方是否需要从本地删除原始消息记录。为 `false` 时，移动端不会删除原始消息记录，会将消息内容替换为撤回提示（小灰条通知）。为 `true` 时，移动端会删除原始消息记录，不显示撤回提示（小灰条通知）。
callback	IRongCoreCallback.ResultCallback<RecallNotificationMessage>	接口回调

监听消息处理

```
//超级群消息变化通知
interface UltraGroupMessageChangeListener {
//消息扩展更新，删除
void onUltraGroupMessageExpansionUpdated(List<Message> messages);
//消息内容发生变更
void onUltraGroupMessageModified(List<Message> messages);
//消息撤回
void onUltraGroupMessageRecalled(List<Message> messages);
}

//设置超级群消息变化监听
public void setUltraGroupMessageChangeListener(IRongCoreListener.UltraGroupMessageChangeListener listener)
```

扩展消息　　扩展消息

更新时间:2024-08-30

已发送的超级群消息可增加、修改、删除扩展信息。

适用场景：

原始消息增加状态标识的需求，都可使用消息扩展。

- 消息评论需求，可通过设置原始消息扩展信息的方式添加评论信息。
- 礼物领取、订单状态变化需求，通过此功能改变消息显示状态。 例如：向用户发送礼物，默认为未领取状态，用户点击后可设置消息扩展为已领取状态。

> 提示
> - 每次设置消息扩展将会产生内置通知消息，频繁设置扩展会产生大量消息。
> - 仅当发送消息时指定 `canIncludeExpansion` 值为 `true`，才可对消息进行扩展。

设置、更新消息扩展信息

> 提示
> 消息扩展功能要求在发送消息前设置该消息为可扩展消息。关于如何修改消息的的可扩展属性，请参考消息扩展中的对 `setCanIncludeExpansion()` 的说明。

```
class OperationCallback {
onSuccess()
onError(ErrorCode code)
}

//更新消息扩展
public void updateUltraGroupMessageExpansion(final Map<String, String> expansion, final String messageUId, final IRongCoreCallback.OperationCallback callback)

//删除消息扩展
public void removeUltraGroupMessageExpansion(final String messageUId, final List<String> keyArray, final IRongCoreCallback.OperationCallback callback)
```

监听消息处理

```
//超级群消息变化通知
interface UltraGroupMessageChangeListener {
//消息扩展更新，删除
void onUltraGroupMessageExpansionUpdated(List<Message> messages);
//消息内容发生变更
void onUltraGroupMessageModified(List<Message> messages);
//消息撤回
void onUltraGroupMessageRecalled(List<Message> messages);
}

//设置超级群消息变化监听
public void setUltraGroupMessageChangeListener(IRongCoreListener.UltraGroupMessageChangeListener listener)
```

输入状态 输入状态

更新时间:2024-08-30

应用程序可以向超级群中发送当前用户输入状态。超级群内收到通知的用户可以在 UI 展示 “ xxx 正在输入”。

> 提示
>
> 为保证最佳体验，建议在仅在人数小于 10,000 的超级群中使用该功能。

发送输入状态

应用程序可以在当前用户输入文本时调用 sendUltraGroupTypingStatus，发送当前用户输入状态。

```
enum UltraGroupTypingStatus{
//正在输入文本
UltraGroupTypingStatusText = 0
}

class OperationCallback {
onSuccess()
onError(ErrorCode code)
}
public void sendUltraGroupTypingStatus(final String targetId, final String channelId, final IRongCoreEnum.UltraGroupTypingStatus typingStatus, final
IRongCoreCallback.OperationCallback callback)
```

监听输入状态

为了减少服务端和客户端的压力，服务端会将一段时间内（例如 5 秒）用户输入事件汇总之后统一批量下发，所以回调以数组形式提供。应用程序收到通知时可以在 UI 展示 “ xxx 正在输入”。

```
class UltraGroupTypingStatusInfo {
String targetId
String channelId
String userId
UltraGroupTypingStatus status
long timestamp //服务端收到用户操作的上行时间.
}

interface UltraGroupTypingStatusListener {
void onUltraGroupTypingStatusChanged(List<UltraGroupTypingStatusInfo> infoList);
}

public void setUltraGroupTypingStatusListener(IRongCoreListener.UltraGroupTypingStatusListener listener)
```

超级群免打扰功能概述 超级群免打扰功能概述

更新时间:2024-08-30

「免打扰功能」用于控制用户在客户端设备离线时，是否可针对离线消息接收推送通知。

- **客户端为离线状态**：会话中有新离线消息时，用户默认通过推送通道收到消息且默认弹出通知。设置免打扰后，融云服务端不会为相关消息触发推送。
- **客户端在后台运行**：会话中有新消息时，用户直接收到消息。如果使用 IMLib，您需要自行判断 App 是否在后台运行，并根据业务需求自行实现本地通知弹窗。

> 提示
>
> 如果不需要接收推送，可以通过设置 SDK 的初始化配置中的 `enablePush` 参数为 `false`，向融云服务申请禁用推送服务（当前设备）。您也可以在断开连接时设置不接收推送（当前设备）。

免打扰设置维度

客户端 SDK 支持超级群业务进行以下多个维度的免打扰设置：

- App 的免打扰设置
- 按超级群或频道设置默认免打扰级别
- 按指定会话类型设置免打扰级别
- 按会话设置免打扰级别
- 按超级群频道设置免打扰级别
- 全局免打扰

> 提示
>
> 超级群离线消息推送通知还会收到控制台超级群默认推送频率设置影响。详见开通超级群服务。

App 的免打扰设置

以 App Key 为单位，设置整个应用所有用户的默认免打扰级别。默认未设置，等同于全部消息都接收通知。该级别的配置暂未在控制台开放，如有需要，请提交工单。

- **全部消息均通知**：当前 App 下的用户可针对任何消息接收推送通知。
- **未设置**：默认全部消息都通知。
- **仅 @ 消息通知**：当前 App 下的用户仅针对提及（@）当前用户和提及所在群组全体成员的消息接收推送通知。
- **仅 @ 指定用户通知**：当前 App 下，用户仅针对提及（@）当前用户的消息接收推送通知。例如：仅张三会接收且仅接收 “@张三 Hello” 的消息的通知。
- **仅 @群全员通知**：当前 App 下，用户仅针对提及（@）群组全体成员的消息接收推送通知。
- **都不接收通知**：当前 App 下，用户不针对任何消息接收推送通知，即任何离线消息都不会触发推送通知。
- **除 @ 消息外群聊消息不发推送**：当前 App 下，用户针对单聊消息、提及（@）指定用户的消息、和提及（@）群组全体成员的消息接收推送通知。

融云服务端判断是否需要推送时，如果存在以下任何一种用户级别的免打扰配置，以用户级别配置为准：

- 按指定会话类型设置免打扰级别
- 按会话设置免打扰级别
- 全局免打扰

如果不存在用户级别配置，则以消息所属超级群/频道的默认免打扰级别为准。App 级别的免打扰配置的优先级最低。

按超级群/频道设置默认免打扰级别

客户端 SDK 支持为**指定超级群下的所有成员**配置触发推送通知的消息类别，或完全关闭通知。客户端 SDK 提供 `PushNotificationLevel`，支持以下六个级别：

枚举值	数值	说明
PUSH_NOTIFICATION_LEVEL_ALL_MESSAGE	-1	与融云服务端断开连接后，当前超级群的所有用户可针对指定超级群（或频道）中的所有消息接收通知。
PUSH_NOTIFICATION_LEVEL_DEFAULT	0	未设置。未设置时均为此初始状态。在此状态下，如果超级群与群频道均为未设置，则认为超级群与频道的默认免打扰级别为全部消息都通知。
PUSH_NOTIFICATION_LEVEL_MENTION	1	与融云服务端断开连接后，当前超级群的所有用户仅针对指定超级群（或频道）中提及（@）当前用户和全体群成员的消息接收通知。
PUSH_NOTIFICATION_LEVEL_MENTION_USERS	2	与融云服务端断开连接后，当前用户仅针对指定超级群（或频道）中提及（@）当前用户的消息接收通知。例如：张三只会接收 “@张三 Hello” 的消息的通知。
PUSH_NOTIFICATION_LEVEL_MENTION_ALL	4	与融云服务端断开连接后，当前用户仅针对指定超级群（或频道）中提及（@）全部群成员的消息接收通知。
PUSH_NOTIFICATION_LEVEL_BLOCKED	5	当前用户针对指定超级群（或频道）中的任何消息都不接收推送通知。

具体设置方法详见设置群/频道默认免打扰。

为指定的超级群设置的默认免打扰逻辑，自动适用于群下的所有频道。如果针对频道另行设置了默认免打扰逻辑，则以该频道的默认设置为准。融云服务端判断是否需要为用户发送推送通知时，如果同时存在以下任何一种用户级别的免打扰配置，以下列配置为准：

- 按会话类型设置免打扰级别

- 按会话设置免打扰级别
- 按超级群频道设置免打扰级别
- 全局免打扰

按会话类型设置免打扰级别

> 提示
>
> 客户端 SDK 从 5.2.2.1 开始支持该功能。该功能属于用户级别设置。

客户端 SDK 提供 PushNotificationLevel，允许用户为会话类型（单聊、群聊、超级群、系统会话）配置触发推送通知的消息类别，或完全关闭通知。提供以下六个级别：

枚举值	数值	说明
PUSH_NOTIFICATION_LEVEL_ALL_MESSAGE	-1	与融云服务端断开连接后，当前用户可针对指定类型会话中的所有消息接收通知。
PUSH_NOTIFICATION_LEVEL_DEFAULT	0	未设置。未设置时均为此初始状态。 注意：在此状态下，如果超级群与群频道均为未设置，则认为超级群与频道的默认免打扰级别为全部消息都通知。
PUSH_NOTIFICATION_LEVEL_MENTION	1	与融云服务端断开连接后，当前用户仅针对指定类型的会话中提及（@）当前用户和全体群成员的消息接收通知。
PUSH_NOTIFICATION_LEVEL_MENTION_USERS	2	与融云服务端断开连接后，当前用户仅针对指定类型的会话中提及（@）当前用户的消息接收通知。例如：张三只会接收 “@张三 Hello” 的消息的通知。
PUSH_NOTIFICATION_LEVEL_MENTION_ALL	4	与融云服务端断开连接后，当前用户仅针对指定类型的会话中提及（@）全部群成员的消息接收通知。
PUSH_NOTIFICATION_LEVEL_BLOCKED	5	当前用户针对指定类型的会话中的任何消息都不接收推送通知。

具体设置方法详见按会话类型设置免打扰。

融云服务端判断是否需要为用户发送推送通知时，如果同时存在以下任何一种用户级别的免打扰配置，以下列配置为准：

- 按会话设置免打扰级别
- 按超级群频道设置免打扰级别
- 全局免打扰

按会话设置免打扰级别

> 提示
>
> 客户端 SDK 从 5.2.2 开始支持该功能。该功能属于用户级别设置。

客户端 SDK 提供 PushNotificationLevel，允许用户为会话配置触发通知的消息类别，或完全关闭通知。提供以下六个级别：

枚举值	数值	说明
PUSH_NOTIFICATION_LEVEL_ALL_MESSAGE	-1	与融云服务端断开连接后，当前用户可针对指定类型会话中的所有消息接收通知。
PUSH_NOTIFICATION_LEVEL_DEFAULT	0	未设置。未设置时均为此初始状态。 注意：在此状态下，如果超级群与群频道均为未设置，则认为超级群与频道的默认免打扰级别为全部消息都通知。
PUSH_NOTIFICATION_LEVEL_MENTION	1	与融云服务端断开连接后，当前用户仅针对指定会话中提及（@）当前用户和全体群成员的消息接收通知。
PUSH_NOTIFICATION_LEVEL_MENTION_USERS	2	与融云服务端断开连接后，当前用户仅针对接收指定会话中提及（@）当前用户的消息接收通知。例如：张三只会接收 “@张三 Hello” 的消息的通知。
PUSH_NOTIFICATION_LEVEL_MENTION_ALL	4	与融云服务端断开连接后，当前用户仅针对指定会话中提及（@）全部群成员的消息接收通知。
PUSH_NOTIFICATION_LEVEL_BLOCKED	5	当前用户针对指定会话中的任何消息都不接收推送通知。

具体设置方法详见按会话设置免打扰。

从 2022.09.01 开始，为指定的超级群设置的免打扰级别，自动适用于群下的所有频道。融云服务端判断是否需要为用户发送推送通知时，如果同时存在以下任何一种用户级别的免打扰配置，以下列配置为准：

- 按超级群频道设置免打扰级别
- 全局免打扰

按超级群频道设置免打扰级别

> 提示
>
> 客户端 SDK 从 5.2.2 开始支持该功能。该功能属于用户级别设置。

客户端 SDK 提供 PushNotificationLevel，允许用户为指定超级群频道配置触发通知的消息类别，或完全关闭通知。提供以下六个级别：

枚举值	数值	说明
PUSH_NOTIFICATION_LEVEL_ALL_MESSAGE	-1	与融云服务端断开连接后，当前用户可针对指定超级群频道中的所有消息接收通知。
PUSH_NOTIFICATION_LEVEL_DEFAULT	0	未设置。未设置时均为此初始状态。 注意：在此状态下，如果超级群与群频道均为未设置，则认为超级群与频道的默认免打扰级别为全部消息都通知。
PUSH_NOTIFICATION_LEVEL_MENTION	1	与融云服务端断开连接后，当前用户可针对指定超级群频道中提及（@）当前用户和全体群成员的消息接收通知。

枚举值	数值	说明
PUSH_NOTIFICATION_LEVEL_MENTION_USERS	2	与融云服务端断开连接后，当前用户可针对指定超级群频道中提及（@）当前用户的消息接收通知。例如：张三只会接收 “@张三 Hello” 的消息的通知。
PUSH_NOTIFICATION_LEVEL_MENTION_ALL	4	与融云服务端断开连接后，当前用户可针对指定超级群频道中提及（@）全部群成员的消息接收通知。
PUSH_NOTIFICATION_LEVEL_BLOCKED	5	当前用户针对指定超级群频道中的任何消息都不接收推送通知。

具体设置方法详见按频道设置免打扰。

融云服务端判断是否需要为用户发送推送通知时，如果该用户已配置全局免打扰，则已全局免打扰的配置细节为准。

全局免打扰

客户端 SDK 从 5.2.2 开始提供 `PushNotificationQuietHoursLevel`，允许用户配置何时接收通知以及触发通知的消息类别。提供了以下三个级别：

枚举值	数值	说明
PUSH_NOTIFICATION_QUIET_HOURS_LEVEL_DEFAULT	0	未设置。如未设置，SDK 会依次查询消息所属群的用户级别免打扰设置及其他非用户级别设置，再判断是否需要推送通知。
PUSH_NOTIFICATION_QUIET_HOURS_LEVEL_MENTION_MESSAGE	1	与融云服务端断开连接后，当前用户仅在指定时段内针对指定会话中提及（@）当前用户和全体群成员的消息接收通知。
PUSH_NOTIFICATION_QUIET_HOURS_LEVEL_BLOCKED	5	当前用户在指定时段内针对任何消息都不接收推送通知。

具体设置方法详见全局免打扰。

早于 5.2.2 的 SDK 版本不支持设置触发通知的消息类别，仅支持设置为接收或不接收推送通知。

免打扰设置的优先级

针对超级群会话，融云服务端会遵照以下顺序搜索免打扰配置。优先级从左至右依次降低，以优先级最高的配置为准判断是否需要触发推送：

全局免打扰设置（用户级）> 指定超级群频道的免打扰设置（用户级）> 指定会话的免打扰设置（用户级）> 指定会话类型的免打扰设置（用户级）> 指定超级群频道的默认免打扰设置（超级群全员）> 指定超级群的免打扰设置（超级群全员）> App 级的免打扰设置

API 接口列表

下表描述了适用于超级群会话的免打扰配置 API 接口。

免打扰配置维度	客户端 API	服务端 API
设置指定时段内，应用全局的免打扰级别（用户级）	详见全局免打扰。	详见 设置用户免打扰时段
设置指定超级群频道的免打扰级别（用户级）	详见按频道设置免打扰。	详见设置会话免打扰 。
设置指定会话的免打扰级别（用户级）	详见按会话设置免打扰。	详见设置会话免打扰 。
设置指定类型会话的免打扰级别（用户级）	详见按会话类型设置免打扰。	详见设置会话类型免打扰 。
设置指定超级群/频道的默认免打扰设置（超级群全体成员）	详见「超级群管理」下的设置群/频道默认免打扰。	详见「超级群管理」下的设置群/频道默认免打扰
设置 App 级免打扰级别	客户端 SDK 不提供 API。	服务端不提供该 API。

设置群/频道默认免打扰 设置群/频道默认免打扰

更新时间:2024-08-30

超级群业务支持为指定的群，或群频道设置默认免打扰逻辑。默认免打扰逻辑对所有群成员生效，一般由超级群的管理员进行设置。

如果您希望从 App 服务端控制指定超级群，或指定群频道默认免打扰逻辑，可参考服务端 API 文档设置超级群/频道默认免打扰 。

注意事项

- 在融云服务端判断是否需要推送超级群消息时，指定的超级群，或群频道的默认免打扰配置优先级均低于用户级别配置。如果存在任何用户级别的免打扰配置，则优先以用户级别免打扰配置为准进行判断。

> 提示
>
> 即时通讯业务免打扰功能的 用户级别设置 支持控制指定的单聊会话、群聊会话、超级群会话、超级群频道的免打扰级别，并可设置全局免打扰的时间段与级别。用户级别设置优先级如下：全局免打扰 > 按频道设置的免打扰 > 按会话设置的免打扰 > 按会话类型设置的免打扰。详见超级群免打扰功能概述。

- 为指定的超级群设置的默认免打扰逻辑，自动适用于群下的所有频道。如果针对频道另行设置了默认免打扰逻辑，则以该频道的默认设置为准。

支持的免打扰级别

指定超级群或群频道的默认免打扰级别可设置为以下任一级别：

枚举值	数值	说明
PUSH_NOTIFICATION_LEVEL_ALL_MESSAGE	-1	所有消息均可进行通知。
PUSH_NOTIFICATION_LEVEL_DEFAULT	0	未设置。未设置时均为此初始状态。 注意：在此状态下，如果超级群与群频道均为未设置，则认为超级群与频道的默认免打扰级别为全部消息都通知。
PUSH_NOTIFICATION_LEVEL_MENTION	1	仅针对 @ 消息进行通知，包括 @指定用户 和 @所有人
PUSH_NOTIFICATION_LEVEL_MENTION_USERS	2	仅针对 @ 指定用户消息进行通知，且仅针对被 @ 的指定的用户进行通知。 如：@张三，则张三可以收到推送； @所有人不会触发推送通知。
PUSH_NOTIFICATION_LEVEL_MENTION_ALL	4	仅针对 @群全员进行通知，即只接收 @所有人的推送信息。
PUSH_NOTIFICATION_LEVEL_BLOCKED	5	不接收通知，即使为 @ 消息也不推送通知。

```
public enum PushNotificationLevel {
NONE(-100),
/** 全部消息通知（接收全部消息通知 -- 显示指定关闭免打扰功能） */
PUSH_NOTIFICATION_LEVEL_ALL_MESSAGE(-1),

/** 未设置（向上查询群或者APP级别设置）//存量数据中0表示未设置 */
PUSH_NOTIFICATION_LEVEL_DEFAULT(0),

/** 群聊，超级群 @所有人 或者 @成员列表有自己 时通知；单聊代表消息不通知 */
PUSH_NOTIFICATION_LEVEL_MENTION(1),

/** 群聊，超级群 @成员列表有自己时通知，@所有人不通知；单聊代表消息不通知 */
PUSH_NOTIFICATION_LEVEL_MENTION_USERS(2),

/** 群聊，超级群 @所有人通知，其他情况都不通知；单聊代表消息不通知 */
PUSH_NOTIFICATION_LEVEL_MENTION_ALL(4),

/** 消息通知被屏蔽，即不接收消息通知 */
PUSH_NOTIFICATION_LEVEL_BLOCKED(5);
}
```

设置指定超级群的默认免打扰级别

```
class OperationCallback {
onSuccess()
onError(ErrorCode code)
}
public void setUltraGroupConversationDefaultNotificationLevel(
final String targetId,
final IRongCoreEnum.PushNotificationLevel level,
final IRongCoreCallback.OperationCallback callback);
```

查询指定超级群的默认免打扰级别

```
class ResultCallback {
onSuccess(PushNotificationLevel level)
onError(ErrorCode code)
}

public void getUltraGroupConversationDefaultNotificationLevel(
final String targetId,
final IRongCoreCallback.ResultCallback<IRongCoreEnum.PushNotificationLevel> callback);
```

设置指定群频道的默认免打扰级别

```
class OperationCallback {
onSuccess()
onError(ErrorCode code)
}

public void setUltraGroupConversationChannelDefaultNotificationLevel(
final String targetId,
final String channelId,
final IRongCoreEnum.PushNotificationLevel level,
final IRongCoreCallback.OperationCallback callback);
```

查询指定群频道的默认免打扰级别

```
class ResultCallback {
onSuccess(RCPushNotificationLevel level)
onError(ErrorCode code)
}

public void getUltraGroupConversationChannelDefaultNotificationLevel(
final String targetId,
final String channelId,
final IRongCoreCallback.ResultCallback<IRongCoreEnum.PushNotificationLevel> callback);
```

关于本地通知 关于本地通知

更新时间:2024-08-30

IMLib 未实现本地通知功能，需要您的 App 自行实现本地通知。

> 提示
>
> - 当 App 刚进入后台时， App 处于后台活跃状态，SDK 通过长连接通道接收消息，此时 SDK 收到消息会触发消息接收监听。您可以在收到消息监听通知时弹出本地通知。关于消息监听器的说明，详见接收消息。
> - App 进入后台如果被系统杀死，SDK 会在断开连接后通过推送通道推送通知。

由于超级群产品本身的业务特性，App 可能需要配合**免打扰级别**（详见超级群免打扰功能概述）功能，实现精细化的本地通知控制策略。当前 IMLib SDK 获取免打扰的 API 是异步的，每次都从数据库中获取，可能会造成一定延迟，无法满足部分 App 对本地通知处理的要求。如果您希望实现从内存中获取免打扰级别数据，可以参考融云 IMKit SDK 中的实现方案。

IMKit 提供一个获取免打扰级别核心类 `io.rong.imkit.notification/MessageNotificationHelper`，该类负责将免打扰级别存储在内存中，避免每次从数据库中获取的开销。

以下简述了 IMKit SDK 的实现方案，相关代码可见 **IMKit 开源工程**（GitHub · Gitee）。我们建议您参考 IMKit 的方案，自行实现相关业务逻辑。

1. IMKit 在初始化时，内部会调用 `setPushNotifyLevelListener`，设置通知级别监听器，监听通知级别变化。

```
MessageNotificationHelper.setPushNotifyLevelListener();
```

2. IMKit 在初始化时，内部调用 `setNotifyListener(notifyListener)` 决定本地通知是否需要通知。

```
void setNotifyListener(NotifyListener notifyListener);

interface NotifyListener {
void onPreToNotify(Message message);
}
```

3. 设置会话状态同步监听。在会话状态改变时，调用 `MessageNotificationHelper.updateLevelMap` 更新会话的免打扰级别，保存在内存中。

```
//同步监听器
IRongCoreListener.ConversationStatusListener listener = new RongIMClient.ConversationStatusListener() {
@Override
public void onStatusChanged(ConversationStatus[] conversationStatuses) {
if (conversationStatuses == null) {
return;
}

for (ConversationStatus status : conversationStatuses) {
Conversation.ConversationType conversationType = status.getConversationType(); //获取会话类型
String targetId = status.getTargetId(); //获取会话 Id
// TODO 从本地获取 conversation
// 保存会话的免打扰级别到内存中
MessageNotificationHelper.updateLevelMap(conversation);

}
}
};
RongCoreClient.getInstance().setConversationStatusListener(listener); //设置监听器
```

4. 调用 `MessageNotificationHelper.getNotificationQuietHoursLevel(callback)` 查询用户应用全局免打扰设置。IMKit 内部会按照用户应用全局 （2） > 用户指定群组频道（3） > 用户指定群 （4） > 用户会话类型（5） > 超级群默认配置 (6) 依次进行查询，是否需要通知，示例代码如下：

```
MessageNotificationHelper.getNotificationQuietHoursLevel(
new RongIMClient.GetNotificationQuietHoursCallback() {
@Override
public void onSuccess(String startTime, int spanMinutes) {
if (callback != null) {
callback.onSuccess(startTime, spanMinutes);
}
}

@Override
public void onError(RongIMClient.ErrorCode errorCode) {
if (callback != null) {
callback.onError(errorCode);
}
}
});
```

5. 在 NotifyListener 触发 onPreToNotify 时决定是否需要弹出本地通知。

```
interface NotifyListener {
void onPreToNotify(Message message);
}
```

本地通知判断逻辑：消息配置(1) > 用户应用全局 （2） > 用户指定群组频道（3） > 用户指定群 （4）> 用户会话类型（5）> 超级群默认配置 (6)

全局免打扰　　全局免打扰

更新时间:2024-08-30

SDK 支持为当前用户设置全局免打扰时段与免打扰级别。

- 该接口会设置一个从任意时间点（`HH:MM:SS`）开始的免打扰时间窗口。在再次设置或删除用户免打扰时间段之前，当次设置的免打扰时间窗口会每日重复生效。例如，App 用户希望设置永久全天免打扰，可设置 `startTime` 为 `00:00:00`，`period` 为 `1439`。
- 单个用户仅支持设置一个时间段，重复设置会覆盖该用户之前设置的时间窗口。
- 如果 SDK 版本 < 5.2.2，仅支持设置免打扰时段，不支持同时设置免打扰级别。建议您尽快升级到最新稳定版或开发版。

> 提示
>
> 在经 SDK 设置的全局免打扰时段内：
>
> - 如果客户端处于离线状态，融云服务端将不会进行推送通知。
> - 「全局免打扰时段」为用户级别的免打扰设置，且具有最高优先级。在用户设置了「全局免打扰时段」时，均以此设置的免打扰级别为准。

（推荐）在 App 自行实现本地通知处理时，如果检测到客户端 App 已转至后台运行，可通过 SDK 提供的全局免打扰接口决定是否弹出本地通知，以实现全局免打扰的效果。

设置免打扰时段与级别（SDK >= 5.2.2）

从 SDK 5.2.2 开始，为当前用户设置免打扰时间段时，可使用以下免打扰级别：

枚举值	数值	说明
PUSH_NOTIFICATION_QUIET_HOURS_LEVEL_DEFAULT	0	未设置。如未设置，SDK 会依次查询消息所属群的用户级别免打扰设置及其他非用户级别设置，再判断是否需要推送通知。
PUSH_NOTIFICATION_QUIET_HOURS_LEVEL_MENTION_MESSAGE	1	仅针对 @ 消息进行通知，包括 @指定用户 和 @所有人的消息。
PUSH_NOTIFICATION_QUIET_HOURS_LEVEL_BLOCKED	5	不接收通知，即使为 @ 消息也不推送通知。

设置免打扰时段与级别

> 提示
>
> 该接口在 ChannelClient 中，从 5.2.2 版本开始支持。

设置消息通知免打扰时间。在免打扰时间内接收到消息时，会根据该接口设置的免打扰级别判断是否需要推送消息通知。

```
String startTime = "00:00:00";
int spanMinutes = 1439;

ChannelClient.getInstance().setNotificationQuietHoursLevel(startTime, spanMinutes,
IRongCoreEnum.PushNotificationQuietHoursLevel.PUSH_NOTIFICATION_QUIET_HOURS_LEVEL_DEFAULT,
new IRongCoreCallback.OperationCallback() {
@Override
public void onSuccess() {

}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
```

参数	类型	说明
startTime	String	开始时间，精确到秒。格式为 HH:MM:SS，例如 01:31:17。
spanMinutes	int	免打扰时间窗口大小，单位为分钟。范围为 [1-1439]。
lev	PushNotificationQuietHoursLevel	• 1：仅针对 @ 消息进行通知，包括 @指定用户 和 @所有人的消息。如果消息所属会话类型为单聊，则代表不通知。 • 0：未设置。如未设置，SDK 会依次查询消息所属群的用户级别免打扰设置及其他非用户级别设置，再判断是否需要推送通知。 • 5：不接收通知，即使为 @ 消息也不推送通知。
callback	ResultCallback<List<Conversation>>	回调接口

移除免打扰时段与级别

> 提示
>
> 该接口在 ChannelClient 中，从 5.2.2 版本开始支持。

您可以调用以下方法将免打扰时间段设置移除。

```
ChannelClient.getInstance().removeNotificationQuietHours(callback);
```

获取免打扰时段与级别

> 提示
>
> 该接口在 ChannelClient 中，从 5.2.2 版本开始支持。

您可以通过以下方法获取免打扰时间段设置。在免打扰时间内接收到消息时，会根据当前免打扰级别判断是否需要推送消息通知。

```
ChannelClient.getInstance().getNotificationQuietHoursLevel(callback)
```

参数	类型	说明
callback	GetNotificationQuietHoursCallbackEx	获取免打扰时间的回调。获取成功时会返回 `startTime`、`spanMinutes`、`level`。

混淆　　混淆 混淆脚本

更新时间:2024-08-30

在项目目录下的 proguard-rules.pro 文件中添加混淆配置。

```
-keepattributes Exceptions,InnerClasses

-keepattributes Signature

-keep class io.rong.** {*;}
-keep class cn.rongcloud.** {*;}
-keep class * implements io.rong.imlib.model.MessageContent {*;}
-dontwarn io.rong.push.**
-dontnote com.xiaomi.**
-dontnote com.google.android.gms.gcm.**
-dontnote io.rong.**

-ignorewarnings
```

当代码中有继承 PushMessageReceiver 的子类时，需 keep 所创建的子类广播。

示例：

```
-keep class io.rong.app.DemoNotificationReceiver {*;}
```

把 io.rong.app.DemoNotificationReceiver 改成子类广播的完整类路径即可。

推送 2.0 集成概述

更新时间:2024-08-30

融云 Push 2.0 推送集成方案（**要求 SDK ≧ 5.6.0**）将第三方厂商推送通道的 SDK 封装成插件，方便开发者快速集成与配置，适用于 IMLib、IMKit 或其他依赖 IMLib 的融云 Android 客户端 SDK。

> 提示
> 如果您的应用需要自行集成第三方厂商推送客户端，或需要使用推送集成商的 SDK（例如个推、极光等），则本集成方案不适用。详见解决推送客户端冲突。

推送服务能力

推送支持「离线消息推送」和「不落地通知」两种场景。

离线消息推送通知

假设用户仅在一台设备上登录，如果主动断开连接（`disconnect()`）或者应用程序已被用户或系统杀死，融云会认为用户在该客户端离线。用户离线状态下，支持将收到的单聊消息、群聊消息、系统消息、超级群消息通过第三方推送厂商或融云自建的推送服务通知客户端。

- 如果由第三方厂商推送服务发送提醒，该提醒一般由系统直接弹出，以通知形式展示在通知面板，提示用户收到消息。
- 如果由融云自建推送通道（RongPush）发送提醒，该提醒一般由 SDK 调用系统 API 构建通知后弹出。注意，RongPush 在国内大部分机型上无法存活。建议应用程序集成第三方厂商的推送服务。

用户点击推送通知后再次与融云服务端建立 IM 连接后，SDK 会有如下行为：

- 自动收到离线期间的单聊、群聊离线消息[7]。服务端默认缓存 7 天未收取的离线消息。
- 自动收到离线期间超级群会话中最后一条消息，应用程序需要自行拉取离线期间的历史消息。

> 提示
> 应用程序处于后台且活跃时，用户仍处于在线状态，SDK 仍可实时收到会话消息，消息送达过程中不会使用任何推送服务，因此用户设备不会收到来自任何推送服务的通知。如果使用 IMLib，应用程序需要自行调用系统 API 创建并弹出本地通知。如果使用 IMKit，SDK 默认会调用系统 API 创建并弹出本地通知。

不落地通知

融云支持直接通过服务端 API 向客户端发送远程推送通知，称为不落地通知[7]。不落地通知中不包含任何会话消息，无论客户端 App 是否在前台，所有通知内容始终仅会以通知形式展示在系统通知栏中，用户无法在任何聊天会话中看到不落地通知的内容。

不落地通知始终通过推送通道下发数据，因此依赖应用程序集成第三方厂商推送服务，或者在客户端启用 RongPush。

- 如果由第三方厂商推送服务发送提醒，该提醒一般由系统直接弹出，以通知形式展示在通知面板，提示用户收到消息。
- 如果由融云自建推送通道（RongPush）发送提醒，该提醒一般由 SDK 调用系统 API 构建通知后弹出。注意，RongPush 在国内大部分机型上无法存活。建议应用程序集成第三方厂商的推送服务。

不落地通知仅支持通过服务端 API 发送，例如：

- 单个用户不落地通知
- 全量用户不落地通知

目前不支持通过控制台发送不落地通知（仅部分旧账号仍保留该能力）。

无法推送以及推送受限的情况

- 因聊天室业务设计特点，仅当聊天室中的用户在线时才会收到聊天室会话中的消息，因此**聊天室消息不支持离线消息推送**。
- 客户端调用了 `logout` 方法，或在 `disconnect` 时设置了不允许推送，或通过设置 SDK 的初始化配置中的 `enablePush` 参数为 `false`，向融云服务申请禁用推送服务（当前设备），导致彻底注销用户在融云服务端的登录信息。这种情况下，用户无法通过任何推送通道收到通知。
- 即使用户的所有移动端设备均已离线，只要用户仍在 Web/PC 端在线，此时融云认为用户在线，默认不会给移动端发送推送通知。如有需要，您可以在控制台免费基础功能页面调整 **Web/PC 在线手机端接收 Push** 开关设置。
- 即使用户的移动端应收到推送通知，融云服务端不会向所有已登录过移动端设备均发送推送，仅会向最后一个登录的设备发送推送通知。
- 已触发第三方厂商推送服务的频率、数量限制。为改善终端用户推送体验，部分第三方推送服务（例如华为、vivo）已对推送消息的分类进行数量和频率管控。建议您充分了解第三方的管理细则。
- 因超级群业务中普通消息的数量较大，为控制离线推送频率，默认每分钟针对单个用户的单个超级群，每个频道最多产生 1 条推送。默认普通消息累计 2 条时才会触发推送。@ 消息不受此限制。如需调整，详见开通超级群服务。

> 提示
> 用户必须至少在设备上连接成功过一次，该设备才能接收推送。

集成第三方推送

目前融云 Push 2.0 集成方案（**要求 SDK ≧ 5.6.0**）已适配了小米、华为、荣耀、魅族、OPPO、vivo、FCM 推送服务。由于国内手机厂商的限制，融云自建推送通道（RongPush）在国内大部分机型上无法

存活。您可以优先选择集成第三方推送。

- 集成小米推送
- 集成华为推送
- 集成荣耀推送
- 集成魅族推送
- 集成 OPPO 推送
- 集成 vivo 推送
- 集成 FCM 推送

在控制台配置 ApplicationId

融云服务端在向第三方推送通道发送推送数据时，需要使用 App 的应用标识[?]（Android 应用 ID）。您需要在控制台进行配置。

1. 访问控制台应用标识页面（也可从**服务管理**页面左侧 **IM 服务**下访问）。
   1. 如您有多个融云应用，请确保在页面顶部**应用**一栏切换到正确的应用。
   2. 新建的应用默认拥有一个应用标识，您可以创建更多应用表示，最多 5 个。每个应用标识均可设置推送，应用标识之间不共享推送配置。
2. 在应用标识旁，点击**设置推送**，并在 **Android** > **ApplicationId** 一栏填写您的 Android 应用 ID。

   每个 Android 应用均有一个唯一的应用 ID（applicationId），像 Java 软件包名称一样，例如 com.example.myapp。此 ID 可以作为您的应用在设备上和 Google Play 商店中的唯一标识。

   应用 ID 由模块的 build.gradle 文件中的 applicationId 属性定义，如下所示：

```
android {
defaultConfig {
applicationId "com.example.myapp"
minSdkVersion 21
targetSdkVersion 24
versionCode 1
versionName "1.0"
}
...
}
```

   如果您没有在 build.gradle 中配置 applicationId，则 applicationId 默认为应用的包名。如果您有更多关于应用 ID 与包名的疑问，可参见 Android 官方文档。

解决第三方推送客户端冲突

如果您的应用或应用依赖的其他 SDK 已集成第三方推送客户端，此时无法再按上述步骤集成融云 Push 2.0 SDK，否则会发生冲突。

在参照集成第三方推送完成控制台的的配置后，您需要自行将第三方厂商推送服务的 Token 上报给融云服务端。详细步骤参见解决推送客户端冲突。

集成融云自建推送通道

融云自建推送通道（RongPush）是融云客户端 SDK 与融云推送服务之间维护的一条稳定可靠的长连接通道。属于 SDK 默认推送，不需要额外集成其它三方库即可拥有的基础推送能力。

SDK 初始化启用推送功能后，则自动启用融云自建推送通道（RongPush），App 即具备了基本推送能力，由于国内手机厂商的限制，RongPush 在国内大部分机型上无法存活。建议同时集成第三方推送通道。

配置 build.gradle。

```
android {
defaultConfig {
//...
manifestPlaceholders = [
//启用 rongpush，如不需要 RongPush 也可以禁用
RONG_PUSH_ENABLE : "true"
]
}
}
dependencies {
// x.y.z 为当前 IM SDK 版本号
implementation 'cn.rongcloud.sdk.push:rong:x.y.z'
}
```

在 SDK init 之前，调用下面代码，初始化 RongPushPlugin 模块。

```
RongPushPlugin.init(getContext());
```

> 提示
>
> 由于融云默认推送通道属于应用级别的推送，会受系统各种权限限制，建议提示用户打开对应权限，以提高推送到达率。

推送通道选择策略

在应用配置了多通道的情况下，为了提高用户体验，提高推送到达率，融云客户端 SDK 会根据应用配置，智能选择最优推送通道。

详细启用策略如下：

推送通道	配置要求	ROM 要求
华为	应用配置了华为推送	当前设备为华为 ROM
荣耀	应用配置了荣耀推送	当前设备为荣耀 ROM，且为 2023 年 11 月 30 日后发售的设备。
OPPO	应用配置了 OPPO 推送	当前设备为以下 ROM 中的一种： OPPO、realme、OnePlus
vivo	应用配置了 vivo 推送	当前设备为 vivo ROM
魅族	应用配置了魅族推送	当前设备为魅族 ROM
FCM	应用配置了 FCM 推送	客户端出访 IP 在国外
RongPush	默认推送通道	不满足其它通道启用策略，且 RongPush 可用时，默认使用 RongPush 推送通道

提示

关于 FCM 的补充说明

系统原生推送和 FCM 同时配置时，具体使用哪种推送通道，取决于应用层的配置顺序。比如在小米手机上，同时配置了小米推送和 FCM 推送，该用户在国外时：

- 如果应用层配置时的顺序是小米推送在 FCM 之前，则会启用小米推送。
- 如果应用层配置时的顺序是 FCM 在小米推送之前，则会启用 FCM 推送。

禁用当前设备的推送功能

IM SDK 默认启用推送功能。如需在当前设备上彻底禁用推送，可以在断开连接时调用 `logout` 方法，或通过设置 SDK 的初始化配置中的 `enablePush` 参数为 `false`，向融云服务申请禁用推送服务（当前设备），彻底注销用户在融云服务端的登录信息。这种情况下，当前设备将无法通过任何推送通道收到通知。

集成小米推送 集成小米推送

更新时间:2024-08-30

按照本指南集成小米 Mi Push 国内版或海外版，让融云 SDK 支持小米推送。

在集成第三方推送前，请确保已在控制台配置 **Android 应用 ID**。详见推送集成概述。

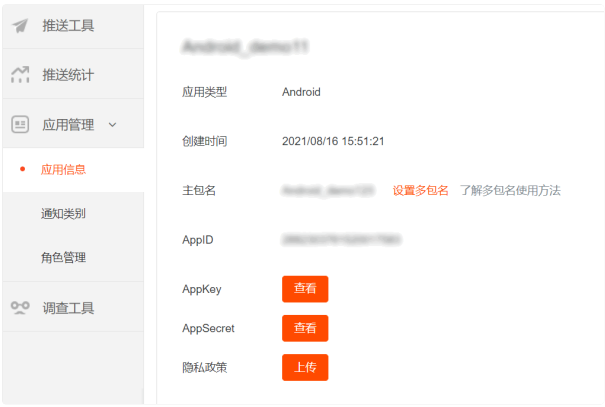
> 提示
>
> IMLib SDK 从 5.6.8 开始支持小米国际推送服务。

在控制台配置小米推送

如果想通过小米推送通道从融云服务端接收推送通知，您需要在控制台上提供您的小米推送应用的详细信息。

1. 前往小米开放平台，选择您当前的项目所对应的小米应用，点击**应用信息**，并记录下应用的 **AppID**、**AppKey**、**AppSecret**。

> **注意：**
>
> **如果没有小米开发者账号，或尚未创建应用，参考小米推送文档：**
>
> - **中国大陆地区 - 推送服务启用指南**
> - **海外版 - Push Service Activation Guide 。目前，小米在印度孟买、德国法兰克福、俄罗斯莫斯科和新加坡设有数据中心，请选择合适的地域创建应用。**



其中 AppSecret 是小米推送服务器端的身份标识，在使用小米推送服务端 SDK 向客户端发送消息时使用，需要在控制台的小米推送配置中提供给融云。AppId 和 AppKey 是小米推送客户端的身份标识，后续在启用小米推送服务时需要提供给融云 SDK，用于初始化小米推送客户端 SDK。

2. 打开控制台，前往在应用标识页面。点击**设置推送**，找到 **Android > 小米推送**，填入上一步获取的 **AppSecret**。

3. （可选）配置**小米推送通知标题**。设置默认的推送通知标题。一般情况下客户端发送消息转 Push 时不使用此标题设置。在调用融云服务端 API `/push.json`、`/push/user.json`、`/push/custom.json` 接口推送通知时，如未传入通知标题，则使用该处设置的标题。从服务端发消息时，如果发送用户 ID 在融云服务端没有用户名，也会使用此“推送通知

标题"。

4. **选择推送通道类型**。小米推送的消息类型，分为普通消息和通知消息，查看小米推送消息限制明文档。

   - 普通消息，融云默认使用的小米推送通道，有限制。
   - 通知消息，无限制。需要填写小米后台申请的 channelId。

5. 保存设置。所有设置 30 分钟后生效。

您已完成需要控制台小米推送配置的全部内容。现在可以设置客户端集成。

配置客户端接收小米推送

> 重要
>
> 建议通过 gradle.properties 配置小米的 App ID 与 App Key。如果希望在 App 的 build.gradle 中直接写入配置，请务必额外添加转义符，示例如下：
>
> - `XIAOMI_APP_ID: "\"1882303761517473625\""`
> - `XIAOMI_APP_KEY : "\"1451747338625\""` 。

您可以通过以下任意一种方案集成小米推送：

- **方案一**：通过 gradle.properties 配置小米的 App ID 与 App Key。
  在 build.gradle 文件 manifestPlaceholders 下 XIAOMI_APP_ID、XIAOMI_APP_KEY 字段中引用 gradle.properties 属性文件中配置。

  gradle.properties 属性文件配置示例：

```
MI_PUSH_APPID="9882303761517473625"
MI_PUSH_APPKEY="9451747338625"
```

build.gradle 配置示例：

```
```gradle
android {
    defaultConfig {
        manifestPlaceholders = [
                // 小米相关应用参数
                XIAOMI_APP_ID  : "${`MI_PUSH_APPID`}",
                XIAOMI_APP_KEY : "${`MI_PUSH_APPKEY`}",
        ]
    }
    // ...其他配置
}
```
```

- **方案二**：在 App 的 build.gradle 中添加依赖。

> 提示
>
> 如果希望在 App 的 build.gradle 中直接写入配置，请务必额外添加转义符，如下所示：
>
> ```
> android {
> defaultConfig {
> manifestPlaceholders = [
> // 小米相关应用参数
> XIAOMI_APP_ID : "\"9882303761517473625\""
> XIAOMI_APP_KEY : "\"9451747338625\"",
> ]
> }
> // ...其他配置
> }
> ```

根据小米推送服务的地域完成相应的配置：

- **中国大陆地区**：集成适用于中国大陆地区的小米推送客户端，并配置小米的 XIAOMI_APP_ID 和 XIAOMI_APP_KEY。

```
android {
defaultConfig {
//...
manifestPlaceholders = [
XIAOMI_APP_ID : "xxxxxxxx",
XIAOMI_APP_KEY: "xxxxxxxx"
]
}
}
dependencies {
// x.y.z 为当前 IM SDK 版本号
implementation 'cn.rongcloud.sdk.push:xiaomi:x.y.z'

}
```

- **海外地区**：小米在印度孟买、德国法兰克福、俄罗斯莫斯科和新加坡设有数据中心。如果您使用小米海外推送服务，必须集成适用于海外地区的小米推送客户端，并配置小米的 XIAOMI_APP_ID 和 XIAOMI_APP_KEY，和地域。

```
android {
defaultConfig {
//...
manifestPlaceholders = [
XIAOMI_APP_ID : "xxxxxxxx",
XIAOMI_APP_KEY: "xxxxxxxx",
XIAOMI_APP_REGION : "" // Global, Europe, Russis, India
]
}
}
dependencies {
// x.y.z 为当前 IM SDK 版本号
implementation 'cn.rongcloud.sdk.xiaomi_global:x.y.z'

}
```

启用小米推送服务

在 SDK init 之前，调用下面代码，初始化 RongPushPlugin 模块。

```
RongPushPlugin.init(getContext());
```

如果找不到 RongPushPlugin 模块，请检查是否已经集成融云自建推送通道。

混淆配置

如果您的应用使用了混淆，您可以使用下面的代码混淆配置：

```
-dontwarn com.xiaomi.mipush.sdk.**
-keep public class com.xiaomi.mipush.sdk.* {*; }
```

处理推送通知的点击事件

- **自定义推送通知点击事件**：介绍如何实现 SDK 的默认跳转行为，以及如何自定义处理点击事件。详见自定义推送通知点击事件。
- **自定义推送通知样式**：SDK 接收到其他第三方厂商的推送后，弹出的通知是系统通知，由手机系统底层直接弹出通知，所以不支持自定义。

集成华为推送 集成华为推送

更新时间:2024-08-30

按照本指南集成 华为推送服务（Push Kit），让融云 SDK 支持从华为推送服务获取推送通知。

在集成第三方推送前，请确保已在控制台配置 **Android 应用 ID**。详见推送集成概述。

在控制台配置华为推送

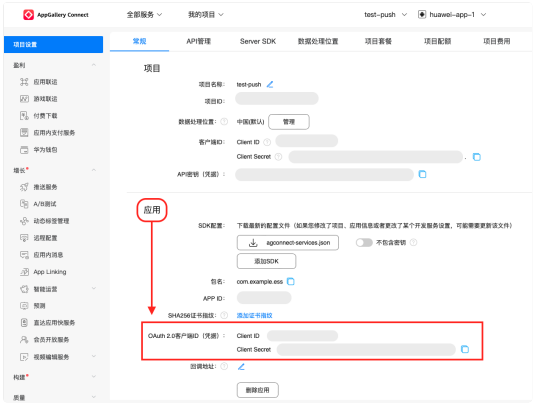
如果想通过华为推送接收融云即时通讯服务的推送通知，您需要在控制台上提供您的华为推送应用的详细信息。

1. 前往华为 AppGallery Connect 网站，点击**我的项目**，在项目列表中找到您的项目，上方导航栏选择需要查看信息的应用。

   如下图所示，test-push 是项目名称，huawei-app-1 是已关联到该项目的应用。

> 提示
>
> 如果没有华为开发者账号，或尚未创建项目和应用，请先创建账号、项目和应用。详见华为开发者文档创建账号、创建应用。请确保项目下已关联了应用，启用了推送服务，并配置签名证书指纹。如您对华为控制台上配置推送服务的流程有任何疑问，可参见华为官方开发者文档配置 AppGallery Connect。

   您需要记录下应用下的 **Client ID**（同 App ID） 和 **Client Secret**。



2. 打开控制台，在应用标识页面点击**设置推送**，找到 **Android** > **华为推送**，填入上一步获取的 **Client ID**、**Client Secret**。

3. （可选）配置自定义点击消息动作的 **Intent**。详见华为官方开发者文档自定义点击消息动作。

   - 注意 intent 的格式必须要以 `end` 结尾。
   - 自定义 intent 后，需按照定义 intent 在 `AndroidManifest.xml` 的 Activity 中配置如下 `intent-filter`。

4. （可选）配置推送角标数。详见华为官方开发者文档桌面角标。

   - **badgeAddNum**：应用角标累加数字非应用角标实际显示数字，为大于0小于100的整数。例如，某应用当前有N条未读消息，若 add_num 设置为3，则每发一次消息，应用角标显示的数字累加3，为 N+3。

- **Activity**：应用入口 Activity 类全路径。样例：com.example.hmstest.MainActivity

5. （可选）配置默认的华为推送通道的消息自分类标识，例如 IM。App 根据华为要求完成自分类权益申请 或 申请特殊权限 后配置字段有效。详见华为推送官方文档消息分类标准 。配置成功后，当前包名接收的华为推送通知默认均会携带该字段。注意，如果客户端或服务端发送消息时配置了华为推送 Category，则使用发消息时指定的配置。

6. 配置**推送通知标题**。设置默认的推送通知标题。一般情况下客户端发送消息转 Push 时不使用此标题设置。在调用融云服务端 API `/push.json`、`/push/user.json`、`/push/custom.json` 接口推送通知时，如未传入通知标题，则使用该处设置的标题。从服务端发消息时，如果发送用户 ID 在融云服务端没有用户名，也会使用此“推送通知标题”。

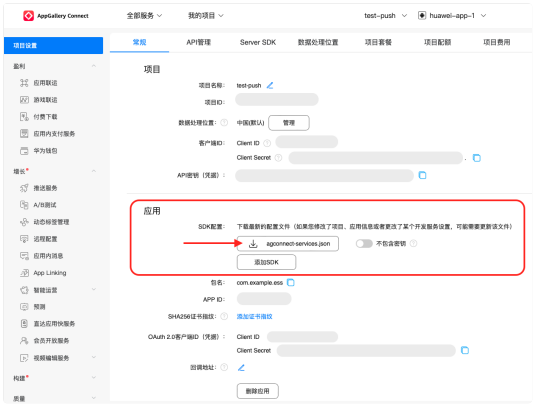
7. 保存设置。所有设置 30 分钟后生效。

您已完成需要控制台华为推送配置的全部内容。现在可以设置客户端集成。

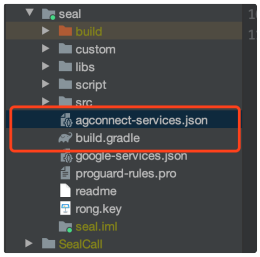
配置客户端接收华为推送

首先，需要将华为推送客户端 SDK 添加到您的 Android 项目。

根据华为开发者文档集成 HMS Core SDK ，您需要将“agconnect-services.json”文件添加到您的 App 中。点击 **agconnect-services.json** 下载配置文件。



将下载好的 `agconnect-servics.json` 文件放到 app 模块下的根目录，如下图：



导入华为推送 SDK

华为推送客户端 SDK 需要从华为 Maven 仓库获取。Android Studio的代码库配置在 Gradle 插件 7.0 以下版本、7.0 版本和 7.1 及以上版本有所不同。详见华为官方开发者文档集成 HMS Core SDK 中的「配置 HMS Core SDK 的 Maven 仓库地址」。

本步骤中以 Gradle 插件 7.0 以下版本为例。打开在 project 的 `build.gradle` 中添加如下内容。

```
allprojects {
repositories {
//Add Huawei Maven
maven {url 'http://developer.huawei.com/repo/'}
}
}

buildscript{
repositories {
//Add Huawei Maven
maven { url 'http://developer.huawei.com/repo/' }
}
dependencies {
// Add this line
classpath 'com.huawei.agconnect:agcp:1.6.1.300'
}
}
```

> 提示
>
> “buildscript > dependencies” 下需要添加 AGC 插件配置，请您参见华为官方开发者文档 AGC 插件依赖关系 选择合适的 AGC 插件版本。

添加华为 Maven 仓库后，您可以在 App 的 `build.gradle` 中添加依赖，直接引入华为推送客户端 SDK。建议您集成的SDK使用最新版本号，版本号索引请参见华为官方开发者文档推送服务 SDK 版本更新说明 。

在 app 模块的 `build.gradle` 添加如下内容：

```
android {
defaultConfig {
//...
manifestPlaceholders = [
HW_PUSH_ENABLE : "true"
]
}
}
dependencies {
// x.y.z 为当前 IM SDK 版本号
implementation 'cn.rongcloud.sdk.push:huawei:x.y.z'
}
...
// Add to the bottom of the file
apply plugin: 'com.huawei.agconnect'
```

> 提示
>
> **华为 Push SDK 依赖说明**：最新版本 Push SDK 需要终端设备上安装HMS Core（APK）4.0.0.300 及以上版本；如果用户手机没有安装，您的应用调用 HMS Core 时，会自动引导用户提示安装。
>
> 默认支持 HMS Core（APK）的手机包括：部分 EMUI 4.0 和 4.1 的手机，以及 EMUI 5.0 及之后的华为手机。

启用华为推送服务

请在初始化融云 SDK 之前启用华为推送服务。融云 SDK 将向华为推送服务注册、并将获取的华为推送 Token 上报给融云服务端。

在 SDK init 之前，调用下面代码，初始化 RongPushPlugin 模块。

```
RongPushPlugin.init(getContext());
```

如果找不到 RongPushPlugin 模块，请检查是否已经集成融云自建推送通道。

混淆配置

请参见华为推送官方文档配置混淆脚本。

处理推送通知的点击事件

- **自定义推送通知点击事件**：介绍如何实现 SDK 的默认跳转行为，以及如何自定义处理点击事件。详见自定义推送通知点击事件。
- **自定义推送通知样式**：SDK 接收到其他第三方厂商的推送后，弹出的通知是系统通知，由手机系统底层直接弹出通知，所以不支持自定义。

角标未读数

融云不维护应用角标数量，融云客户端 SDK 不支持控制角标展示。如需了解与厂商推送通知相关的角标控制实现，可参考知识库文档推送角标。

集成荣耀推送 集成荣耀推送

更新时间:2024-08-30

按照本指南集成 荣耀推送服务，让融云 SDK 支持从荣耀推送服务获取推送通知。

在集成第三方推送前，请确保已在控制台配置 **Android 应用 ID**。详见推送集成概述。

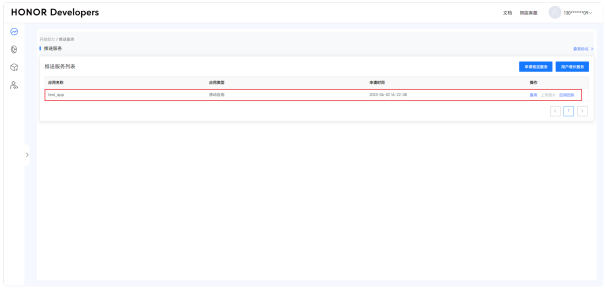
> 提示
> IMLib SDK（开发版）从 5.6.7 版本开始支持荣耀推送。

> 重要
> 如果您的项目集成或升级到 IM SDK（开发版）5.6.7 版本，必须同时集成荣耀推送，否则荣耀 Magic OS 8.0 及之后系统版本的设备可能无法接收推送通知。

在控制台配置荣耀推送

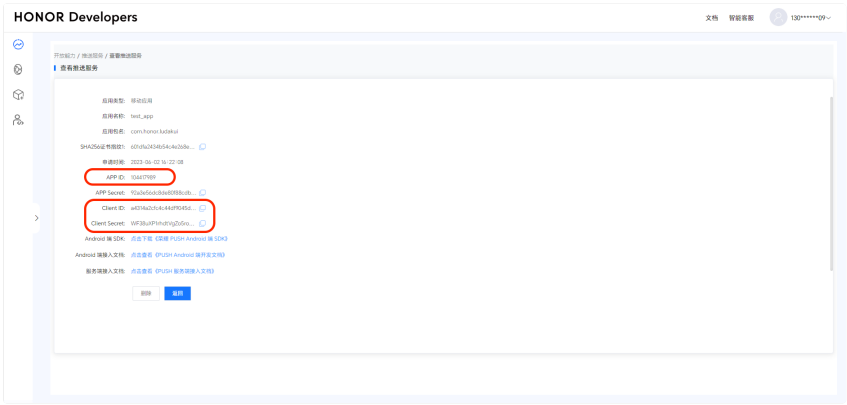
如果想通过荣耀推送接收融云即时通讯服务的推送通知，您需要在控制台上提供您的荣耀推送应用的详细信息。

1. 前往荣耀开发者服务平台，在 推送服务 页面，选择您创建的应用。



> 提示
> - 如果没有荣耀开发者账号，或尚未创建项目和应用，请先创建账号、项目和应用，并开通推送服务。详见荣耀开发者文档创建账号、创建应用、申请开通推送服务。
> - 在继续完成以下步骤前，请确保已在荣耀开发者服务平台为应用开通了推送服务，并配置签名证书指纹。如您对荣耀控制台上配置推送服务的流程有任何疑问，可参见荣耀官方开发者文档上架申请指南。

您需要记录下应用下的 **App ID**、**Client ID** 和 **Client Secret**。



2. 打开控制台，在应用标识页面点击**设置推送**，找到 **Android > 荣耀推送**，填入上一步获取的 **Client ID**、**Client Secret**。

3. （可选）配置自定义点击消息动作的 **Intent**，用于打开应用自定义页面。该字段对应荣耀官方下行消息接口中 `ClickAction.type` 为 1 时的 `ClickAction.action`。如有疑问，详见荣耀开发者文档消息推送。自定义 intent 后，需在 `AndroidManifest.xml` 的 Activity 中配置 `intent-filter`，接收自定义的 intent。

4. （可选）配置推送角标。

   - **badgeAddNum**：应用角标累加数字非应用角标实际显示数字，为大于 0 小于 100 的整数。例如，某应用当前有N条未读消息，若 add_num 设置为3，则每发一次消息，应用角标显示的数字累加3，为N+3。该字段对应荣耀官方下行消息接口中 `BadgeNotification.addNum`。如有疑问，详见荣耀开发者文档消息推送。
   - **Activity**：应用入口 Activity 类全路径。样例：com.example.honortest.MainActivity。该字段对应荣耀官方下行消息接口中 `BadgeNotification.badgeClass`。如有疑问，详见荣耀开发者文档消息推送。

5. 配置**推送通知标题**。设置默认的推送通知标题。一般情况下客户端发送消息转 Push 时不使用此标题设置。在调用融云服务端 API `/push.json`、`/push/user.json`、`/push/custom.json` 接口推送通知时，如未传入通知标题，则使用该处设置的标题。从服务端发消息时，如果发送用户 ID 在融云服务端没有用户名，也会使用此“推送通知标题”。

6. 保存设置。所有设置 30 分钟后生效。

您已完成需要控制台荣耀推送配置的全部内容。现在可以设置客户端集成。

配置客户端接收荣耀推送

首先，需要将荣耀推送客户端 SDK 添加到您的 Android 项目。

下载荣耀服务配置文件

以下步骤来自荣耀开发者文档添加应用配置文件：

1. 登录荣耀开发者服务平台，单击**应用管理**，在应用列表中找到目标应用，单击**应用详情**。

2. 在应用基础信息查看页面的 SDK 配置区域，下载 `mcs-services.json` 配置文件。

3. 将下载好的 `mcs-servics.json` 文件拷贝到应用级根目录下。

导入荣耀推送 SDK

> 重要
>
> 荣耀推送 SDK Maven 仓库的配置在 Gradle 插件 7.0 以下版本、7.0 版本和 7.1 及以上版本有所不同。本步骤中以荣耀官方文档中 Gradle 插件 7.0 以下版本为例。其他 Gradle 版本配置方法，详见荣耀官方文档 集成 SDK。

1. 配置荣耀官方 Maven 仓库，以 Gradle 7.0 以下版本为例。

   打开在 project 的 `build.gradle`：

   - 在 buildscript > repositories 中配置 SDK 的 Maven 仓地址。
   - 在 allprojects > repositories 中配置 SDK 的 Maven 仓地址。
   - 如果 App 中添加了 mcs-services.json 文件则需要在 buildscript > dependencies 中增加 asplugin 插件配置。

```
buildscript {
repositories {
google()
jcenter()
// 配置SDK的Maven仓地址。
maven {url 'https://developer.hihonor.com/repo'}
}
dependencies {
...
// 增加asplugin插件配置，推荐您使用最新版本。
classpath 'com.hihonor.mcs:asplugin:2.0.0'
// 增加gradle插件配置，根据gradle版本选择对应的插件版本号
classpath 'com.android.tools.build:gradle:4.1.2'
}
}

allprojects {
repositories {
google()
jcenter()
// 配置SDK的Maven仓地址。
maven {url 'https://developer.hihonor.com/repo'}
}
}
```

2. 添加荣耀 asplugin 插件配置。以 Gradle 7.0 以下版本为例，在应用级别的 `build.gradle` 文件头部声明下一行添加如下配置：

```
apply plugin: 'com.hihonor.mcs.asplugin'
```

> 提示
>
> 引入 asplugin 插件需要在 Android studio 中配置 gradle 的 jdk 版本为 11 以上。详见荣耀官方文档 配置开发环境。

3. 如果您的应用 `targerSdkVersion` 大于等于30，需要在 `AndroidManifest.xml` 中添加标签，指定了应用可以处理的 intent 的 action。

```
<queries>
<intent>
<action android:name="com.hihonor.push.action.BIND_PUSH_SERVICE" />
</intent>
</queries>
```

4. 在应用级别的 `build.gradle` 添加如下编译依赖，集成融云推送 2.0 SDK。

```
android {
defaultConfig {
//...
manifestPlaceholders = [
HONOR_APP_ID : "xxxxxxxxxx"
]
}
}
dependencies {
// x.y.z 为当前 IM SDK 版本号
implementation 'cn.rongcloud.sdk.push:honor:x.y.z'
}
```

`HONOR_APP_ID` 对应荣耀应用的 APP ID。

配置签名

参考荣耀官方文档，将生成签名证书指纹步骤中生成的签名文件拷贝到工程的 App 目录下，在 `build.gradle` 文件中配置签名。

```
android {
signingConfigs {
config {
// 根据您实际的签名信息，替换以下参数中的xxxx
keyAlias 'xxxx'
keyPassword 'xxxx'
storeFile file('xxxx.jks')
storePassword 'xxxx'
}
}
buildTypes {
debug {
signingConfig signingConfigs.config
}
release {
signingConfig signingConfigs.config
minifyEnabled true
proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
}
}
}
```

启用荣耀推送服务

> 提示
>
> SDK 从 5.6.7 版本开始支持荣耀推送，并向融云服务端上报荣耀推送的 Token。

请在初始化融云 SDK 之前启用荣耀推送服务。融云 SDK 将向荣耀推送服务注册设备，并将从荣耀推送服务端获取的荣耀推送 Token 上报给融云推送服务端。

在 SDK init 之前，调用下面代码，初始化 RongPushPlugin 模块。

```
RongPushPlugin.init(getContext());
```

如果找不到 RongPushPlugin 模块，请检查是否已经集成融云自建推送通道。

混淆配置

请参见荣耀推送官方文档配置混淆脚本。

处理推送通知的点击事件

- **自定义推送通知点击事件**：介绍如何实现 SDK 的默认跳转行为，以及如何自定义处理点击事件。详见自定义推送通知点击事件。
- **自定义推送通知样式**：SDK 接收到其他第三方厂商的推送后，弹出的通知是系统通知，由手机系统底层直接弹出通知，所以不支持自定义。

角标未读数

融云不维护应用角标数量，融云客户端 SDK 不支持控制角标展示。如需了解与厂商推送通知相关的角标控制实现，可参考知识库文档推送角标。

集成魅族推送 集成魅族推送

更新时间:2024-08-30

按照本指南集成 魅族 Flyme 推送客户端，让融云 SDK 支持从魅族推送服务获取推送通知。

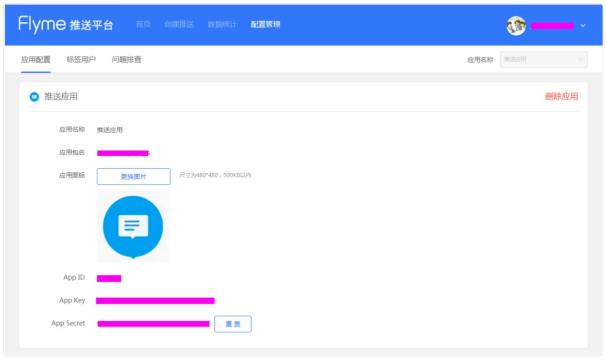
在集成第三方推送前，请确保已在控制台配置 **Android 应用 ID**。详见推送集成概述。

在控制台配置魅族推送

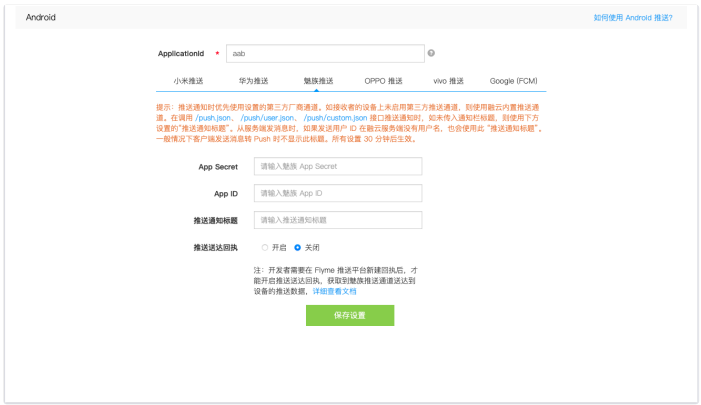
如果想通过魅族推送接收融云即时通讯服务的推送通知，您需要在控制台上提供您的魅族推送应用的详细信息。

1. 前往 魅族推送平台，并记录下应用的 **AppID**、**AppKey**、**AppSecret**。

> 提示
> 如果没有魅族 Flyme 开发者账号，或尚未创建应用，参考魅族 Flyme 推送接入文档。Flyme 开发者账号通过认证后可创建应用。



2. 打开控制台，在应用标识页面点击**设置推送**，找到 **Android** > **魅族推送**，填入上一步获取的 **AppID**、**AppSecret**。



3. （可选）配置**推送通知标题**。设置默认的推送通知标题。一般情况下客户端发送消息转 Push 时不使用此标题设置。在调用融云服务端 API `/push.json`、`/push/user.json`、`/push/custom.json` 接口推送通知时，如未传入通知标题，则使用该处设置的标题。从服务端发消息时，如果发送用户 ID 在融云服务端没有用户名，也会使用此“推送通知标题”。

4. 是否开启推送回执。您需要在魅族推送平台中新建回执，并在此启用后，才能获取到魅族通道送达数据。具体配置流程详见上报推送数据。

5. 保存设置。所有设置 30 分钟后生效。

您已完成需要控制台魅族推送配置的全部内容。现在可以设置客户端集成。

配置客户端接收魅族推送

在 App 的 `build.gradle` 中添加依赖，并配置魅族的 `MEIZU_APP_ID` 和 `MEIZU_APP_KEY`：

```
android {
defaultConfig {
//...
manifestPlaceholders = [
MEIZU_APP_ID : "xxxxxxxx",
MEIZU_APP_KEY: "xxxxxxxx"
]
}
dependencies {
// x.y.z 为当前 IM SDK 版本号
implementation 'cn.rongcloud.sdk.push:meizu:x.y.z'
}
}
```

启用魅族 Flyme 推送服务

在 SDK init 之前，调用下面代码，初始化 RongPushPlugin 模块。

```
RongPushPlugin.init(getContext());
```

如果找不到 RongPushPlugin 模块，请检查是否已经集成融云自建推送通道。

混淆配置

魅族的厂商推送客户端使用了 aar 包方式，因此已经处理好了一些通用的权限配置和代码混淆。App 接入融云推送插件时不需要再对魅族推送 SDK 进行额外的配置。如需了解细节或遇到相关问题，请参考魅族 Flyme 推送接入文档 。

处理推送通知的点击事件

- **自定义推送通知点击事件**：介绍如何实现 SDK 的默认跳转行为，以及如何自定义处理点击事件。详见自定义推送通知点击事件。
- **自定义推送通知样式**：SDK 接收到其他第三方厂商的推送后，弹出的通知是系统通知，由手机系统底层直接弹出通知，所以不支持自定义。

集成 OPPO 推送 集成 OPPO 推送

更新时间:2024-08-30

按照本指南集成OPPO 推送客户端，融云 SDK 支持从 OPPO 推送服务收取推送。

> 提示
> - 并非所有的 OPPO 手机都支持 OPPO 推送。OPPO 推送目前支持 ColorOS 3.1 及以上的系统的OPPO的机型，一加5/5t及以上机型，realme 所有机型。详见 OPPO 推送服务文档 业务功能使用问题。
> - 在不支持 OPPO 推送的 OPPO 手机上会使用融云默认推送。

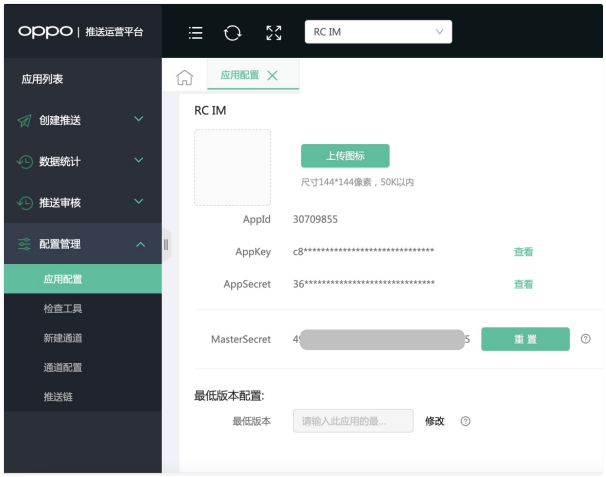
在集成第三方推送前，请确保已在控制台配置 **Android 应用 ID**。详见推送集成概述。

在控制台配置 OPPO 推送

如果想通过 OPPO 推送接收融云即时通讯服务的推送通知，您需要在控制台上提供您的 OPPO 推送应用的详细信息。

1. 前往 OPPO 推送运营平台 的**配置管理 > 应用配置**中获取推送凭证，包括 **AppKey**、**AppSecret**、 **MasterSecret**。

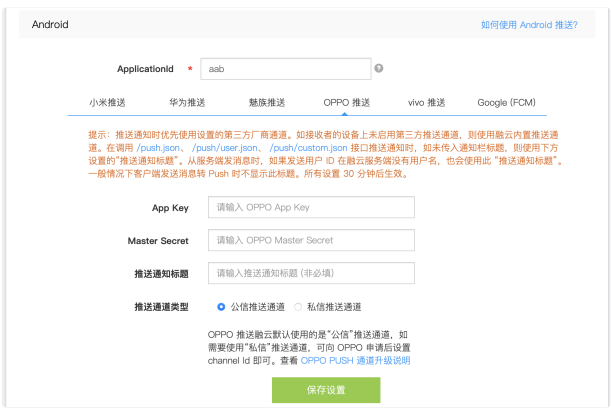
> 提示
> - 如果没有 OPPO 开发者账号，或尚未创建应用，或尚未开启推送服务权限，请先在 OPPO 开发者平台上创建账号、应用并开启推送服务权限。详见 OPPO 开发者文档 推送服务开启指南。
> - OPPO 要求开启推送服务需要注册成为 OPPO 企业开发者，详情请参考 OPPO企业开发者帐号注册流程。



OPPO 推送凭证用途如下：

- **AppKey**：OPPO 应用的身份标识。客户端以及在控制台配置 OPPO 推送时均需使用。
- **AppSecret**：客户端集成 OPPO 推送 SDK 时使用。
- **MasterSecret**（即 AppServerSecret）：在控制台配置 OPPO 推送服务凭证时使用。

2. 打开控制台，在应用标识页面点击**设置推送**，找到 **Android > OPPO 推送**，填入上一步获取的 **App Key**、**Master Secret**。



3. （可选）**配置推送通知标题**。设置默认的推送通知标题。一般情况下客户端发送消息转 Push 时不使用此标题设置。在调用融云服务端 API `/push.json`、`/push/user.json`、`/push/custom.json` 接口推送通知时，如未传入通知标题，则使用该处设置的标题。从服务端发消息时，如果发送用户 ID 在融云服务端没有用户名，也会使用此“推送通知

标题"。

4. **配置推送通道类型**。OPPO 官方开发者文档说明如下：为了改善终端用户的通知体验，营造良好可持续的推送生态，OPPO推送对各个APP的push消息数量进行了限量管控；同时针对即时聊天/系统提醒等push需求可以申请走私信通道（该通道不限量）。

> 提示
>
> 融云服务端默认使用的是 OPPO 公信推送通道，单日推送消息次数受限，总的推送次数用完后，当日将无法再给用户推送内容。目前单日推送数量为：累计注册用户数*2。如需使用不受推送数量限制的私信推送通道，可参考 OPPO 文档推送私信通道申请。申请私信通道完成后，请返回控制台填入私信推送通道的 channel Id。

5. 保存设置。所有设置 30 分钟后生效。

您已完成需要控制台 OPPO 推送配置的全部内容。现在可以设置客户端集成。

配置客户端接收 OPPO 推送

在 App 的 `build.gradle` 中添加依赖，并配置 oppo 的 `OPPO_APP_KEY` 和 `OPPO_APP_SECRET`。

```
android {
defaultConfig {
//...
manifestPlaceholders = [
OPPO_APP_KEY : "xxxxxxxx",
OPPO_APP_SECRET: "xxxxxxxx"
]
}
}
dependencies {
// x.y.z 为当前 IM SDK 版本号
implementation 'cn.rongcloud.sdk.push:oppo:x.y.z'
}
```

关于权限要求的说明，可参见 OPPO 官方开发者文档 SDK 数据安全说明。

启用 OPPO 推送服务

在 SDK init 之前，调用下面代码，初始化 `RongPushPlugin` 模块。

```
RongPushPlugin.init(getContext());
```

如果找不到 `RongPushPlugin` 模块，请检查是否已经集成融云自建推送通道。

混淆配置

如果您的应用使用了混淆，您可以使用下面的代码混淆配置：

```
-keep public class * extends android.app.Service
-keep class com.heytap.msp.** { *;}
```

处理推送通知的点击事件

- **自定义推送通知点击事件**：介绍如何实现 SDK 的默认跳转行为，以及如何自定义处理点击事件。详见自定义推送通知点击事件。
- **自定义推送通知样式**：SDK 接收到其他第三方厂商的推送后，弹出的通知是系统通知，由手机系统底层直接弹出通知，所以不支持自定义。

常见问题

接收 OPPO 推送需要用户的 OPPO 手机开启应用的通知权限。由于融云推送属于应用级别的推送，会受系统各种权限限制，我们建议您在使用时，在设置里打开自启动权限和通知权限，或者勾选"信任此应用"等，以提高推送到达率。

OPPO推送暂时只支持通知栏消息的推送。消息下发到 OS 系统模块并由系统通知模块展示，在用户点击通知前，不启动应用。

由于并非所有的 OPPO 手机都支持 OPPO 推送，所以仅在支持 OPPO 推送的 OPPO 手机上使用 OPPO 推送，在不支持 OPPO 推送的 OPPO 手机上使用融云默认推送。

关于 OPPO 推送服务的其他问题，建议参考 OPPO 官方开发者文档「常见 FAQ」中技术问题和业务功能使用问题。

集成 vivo 推送 集成 vivo 推送

更新时间:2024-08-30

按照本指南集成vivo 推送客户端，融云 SDK 支持从 vivo 推送服务收取推送。

在集成第三方推送前，请确保已在控制台配置 **Android 应用 ID**。详见推送集成概述。

在控制台配置 vivo 推送

如果想通过 vivo 推送接收融云即时通讯服务的推送通知，您需要在控制台上提供您的 vivo 推送应用的详细信息。

1. 前往 vivo 开发者平台-推送服务器平台，并记录下 vivo 应用的 **AppID**、**AppKey**、**AppSecret**。

> 提示
> 如果没有 vivo 开发者账号，或尚未创建应用，参考 vivo 文档 vivo 推送接入流程。

2. 打开控制台，在应用标识页面点击**设置推送**，找到 **Android > vivo 推送**，填入上一步获取的 **AppID**、**AppKey**、**AppSecret**。

3. （可选）配置**推送通知标题**。设置默认的推送通知标题。一般情况下客户端发送消息转 Push 时不使用此标题设置。在调用融云服务端 API `/push.json`、`/push/user.json`、`/push/custom.json` 接口推送通知时，如未传入通知标题，则使用该处设置的标题。从服务端发消息时，如果发送用户 ID 在融云服务端没有用户名，也会使用此“推送通知标题”。

4. 配置**推送模式**。关于正式推送与测试推送的区别，请参考 vivo 文档vivo 推送使用指南。

5. （可选）配置**推送通道类型**与 **Category**（消息二级分类）。如果调用客户端或服务端 API 发送消息或推送通知时未传值，默认使用此处配置的值。请参考 vivo 文档推送消息分类说明进行配置。

6. 保存设置。所有设置 30 分钟后生效。

您已完成需要控制台 vivo 推送配置的全部内容。现在可以设置客户端集成。

配置客户端接收 vivo 推送

在 App 的 `build.gradle` 中添加依赖，并配置 vivo 的 `VIVO_APP_ID` 和 `VIVO_APP_KEY`。

```
android {
defaultConfig {
//...
manifestPlaceholders = [
VIVO_APP_ID : "xxxxxxxxxx",
VIVO_APP_KEY: "xxxxxxxxxx"
]
}
}
dependencies {
// x.y.z 为当前 IM SDK 版本号
implementation 'cn.rongcloud.sdk.push:vivo:x.y.z'
}
```

启用 vivo 推送服务

在 SDK init 之前，调用下面代码，初始化 RongPushPlugin 模块。

```
RongPushPlugin.init(getContext());
```

如果找不到 RongPushPlugin 模块，请检查是否已经集成融云自建推送通道。

混淆配置

请在混淆文件中添加以下说明，防止 SDK 内容被二次混淆，自定义回调类切勿混淆。

```
-dontwarn com.vivo.push.**
-keep class com.vivo.push.**{*; }
-keep class com.vivo.vms.**{*; }
-keep class    xxx.xxx.xxx.PushMessageReceiverImpl{*;}
```

处理推送通知的点击事件

- **自定义推送通知点击事件**：介绍如何实现 SDK 的默认跳转行为，以及如何自定义处理点击事件。详见自定义推送通知点击事件。
- **自定义推送通知样式**：SDK 接收到其他第三方厂商的推送后，弹出的通知是系统通知，由手机系统底层直接弹出通知，所以不支持自定义。

设置应用自增角标

vivo 提供以下桌面角标的技术方案，以帮助您实现应用自增角标的功能。

- **系统 API 接入**。
  通过此方案，您可以利用 vivo 提供的系统 API 来设置桌面角标。此方案仅在应用存活时有效。具体接入方式，请参考桌面角标设置方法 - 系统API 。
- **Vpush 接入**。
  由于融云已经集成 Vpush 接入方案，您可以向融云提交工单开通此功能，并按照桌面角标设置方法 - Vpush接入方法 接入。

集成 FCM 推送 集成 FCM 推送

更新时间:2024-08-30

- FCM 推送通道适用于海外正式发售的 Android 设备（内置 Google GMS 服务），且会在海外网络环境下启用。
- 建议您在集成后根据测试 FCM 推送 中描述的条件与步骤进行测试。

融云服务端已集成与 FCM 后端通信的功能组件。在消息接收者设备上运行的 App 被杀进程，或者在后台被挂起，或者在后台存活超过 2 分钟的情况下，IM SDK 长连接通道会断开。此时如有消息需要送达，融云服务端会向 FCM 后端发送消息请求，然后由 FCM 后端再将消息发送到用户设备上运行的客户端应用。

Android 项目集成 FCM

本节内容将遵照 Google 推荐的设置工作流，描述如何通过 Firebase 控制台 将 Firebase 添加到您的 Android 项目。在此过程中，您必须手动将插件和配置文件添加到您的项目。

为帮助您快速以下步骤已经简化。如需详细步骤，您可以参考 Google 文档，或 Firebase 中文文档。

前提条件

- 安装最新版本的 Android Studio，或将其更新为最新版本。
  - 确保您的项目满足以下要求：
  - （SDK ≧ 5.6.3）使用 Android 5.0（API 21）或更高版本；
  - （SDK < 5.6.3）使用 Android 4.4（API 19）或更高版本；
  - 使用 Jetpack (AndroidX)，这需要满足以下版本要求：
    - `com.android.tools.build:gradle 3.2.1` 或更高版本
    - `compileSdkVersion 28` 或更高版本
- 设置一台实体设备或使用模拟器运行您的应用。
  请注意，FCM 客户端属于 依赖于 Google Play 服务的 Firebase SDK，需要在设备或模拟器上安装 Google Play 服务。
- 使用您的 Google 帐号登录 Firebase。

如要将 Firebase 添加到您的应用，您需要在 Firebase 控制台和打开的 Android 项目中执行若干任务（例如，从控制台下载 Firebase 配置文件，然后将配置文件移动到 Android 项目中）。

第 1 步：创建 Firebase 项目

1. 在Firebase 控制台中，点击**添加项目**。
   - 如需创建新项目，请输入要使用的项目名称。您也可以视需要修改项目名称下方显示的项目 ID。
   - 如需将 Firebase 资源添加到现有 Google Cloud 项目，请输入该项目的名称或从下拉菜单中选择该项目。
2. 点击**继续**。最后，点击**创建项目**（如果使用现有的 Google Cloud 项目，则点击**添加 Firebase**）。

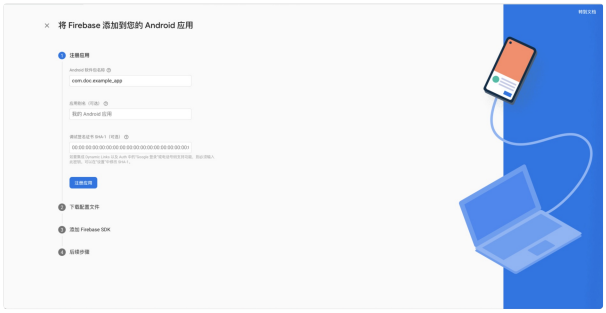
Firebase 会自动为您的 Firebase 项目预配资源。完成此过程后，您将进入 Firebase 控制台中 Firebase 项目的概览页面。

Firebase 项目实际上只是一个启用了额外的 Firebase 特定配置和服务的 Google Cloud 项目。在创建新的 Firebase 项目时，您实际上是在幕后创建 Google Cloud 项目。详情可参考 Firebase 中文文档 Firebase 项目与 Google Cloud 之间的关系。

第 2 步：在 Firebase 中注册您的 Android 应用

如需在 Android 应用中使用 Firebase，您需要向 Firebase 项目注册您的应用。注册应用的过程通常称为将应用“添加”到项目中。

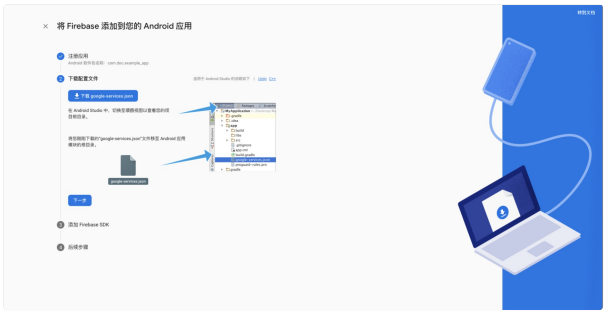
1. 前往 Firebase 控制台。
2. 在项目概览页面的中心位置，点击 **Android** 图标或添加应用，启动设置工作流。
3. 在 **Android 软件包名称**字段中输入应用的软件包名称。
   - 软件包名称是您的应用在设备上和 Google Play 商店中的唯一标识符。
   - 软件包名称通常称为应用 ID。
   - 在模块（应用级）Gradle 文件（通常是 app/build.gradle）中查找应用的软件包名称（示例软件包名称：com.yourcompany.yourproject）。
   - 请注意，软件包名称值区分大小写，并且当您在 Firebase 项目中注册此 Firebase Android 应用后，将无法更改其软件包名称。
4. 如有需要，可完成其他可选配置。然后点击**注册应用**。



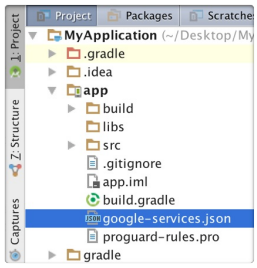
第 3 步：在 Android Studio 项目中添加 Firebase 配置文件

本步骤描述如何将 Firebase Android 配置文件添加到您的应用。

1. 注册应用后，点击**下载 google-services.json** 以获取 Firebase Android 配置文件 (`google-services.json`)。



2. 将配置文件移到应用的模块（应用级）目录中。



- 如需详细了解此配置文件，请访问了解 Firebase 项目。
- 您可以随时再次下载 Firebase 配置文件。
- 请确保该配置文件名未附加其他字符，如 (2)。

3. 如需在应用中启用 Firebase 产品，请将 `google-services` 插件添加到 Gradle 文件中。

> 注意，所有 Firebase SDK 都使用 google-services Gradle 插件（`google-services`），但该插件与 Google Play 服务没有任何关系）。

- **方法 1**：使用 Gradle 的 `buildscript` 块添加插件。Google 文档 与 Firebase 中文文档 均使用该方法。

1. 修改根级（项目级）Gradle 文件 (`build.gradle`)，在 buildscript 块中添加规则，以导入 Google 服务 Gradle 插件。**请确认您添加了 Google 的 Maven 代码库。**

```
buildscript {

repositories {
// 请检查是否有此行（如果没有，请添加）：
google() // Google's Maven repository
}

dependencies {
// ...

// 请添加以下配置：
classpath 'com.google.gms:google-services:4.3.13' // Google Services plugin
}
}

allprojects {
// ...

repositories {
// 请检查是否有此行（如果没有，请添加）：
google() // Google's Maven repository
// ...
}
}
```

2. 在您的模块（应用级）Gradle 文件（通常是 app/build.gradle）中，应用 Google 服务 Gradle 插件：

```
apply plugin: 'com.android.application'
// Add the following line:
apply plugin: 'com.google.gms.google-services' // Google Services plugin

android {
// ...
}
```

- **方法 2**：使用 Gradle 的 plugins DSL。如果您熟悉 Gradle 的 plugins DSL 语法，且已在项目中使用，可使用该方法。

1. 默认情况下 plugins DSL 仅支持已发布在 Gradle Plugin Portal 的核心插件。因为 google-services 插件不在该仓库中，所以您需要先添加 Google 的插件仓库地址。

   在根级（项目级）Gradle 设置文件 (settings.gradle) 中声明 Google 的 Maven 代码库。

```
pluginManagement {
// （可选）在声明仓库时即可引入 google-services plugin 并声明版本
//plugins {
// id 'com.google.gms.google-services' version '4.3.13'
//}
repositories {
google()
gradlePluginPortal()
}
}
rootProject.name='exampleProject'
include ':app'
```

2. 在您的模块（应用级）Gradle 文件（通常是 app/build.gradle）中，应用 Google 服务 Gradle 插件google-services 插件，并声明版本：

```
plugins {
id 'com.google.gms.google-services' version '4.3.13'
}
```

   注意：如果在 settings.gradle 的 pluginManagement {} 块中已使用 plugins DSL 引入了 google-services 插件，并且声明了版本，则可直接在模块（应用级）app/build.gradle 文件中应用插件，无需再声明版本：

```
plugins {
id 'com.google.gms.google-services'
}
```

第 4 步：将 Firebase SDK 添加到您的 Android 应用

1.在模块（应用级）Gradle 文件（通常为 app/build.gradle）中声明依赖项,并开启 fcm 推送。

```
android {
defaultConfig {
//...
manifestPlaceholders = [
FCM_PUSH_ENABLE : "true"
]
}
}
dependencies {
// x.y.z 为当前 IM SDK 版本号
implementation 'cn.rongcloud.sdk.push:fcm:x.y.z'
}
```

如需了解更多细节，可参考 Firebase 中文文档 设置 Firebase Cloud Messaging 客户端应用 (Android)。

至此，您已经成功将 Firebase 集成到 Android 项目中。

请继续完成后续步骤，**在控制台授权 FCM 请求**。

在控制台授权 FCM 请求

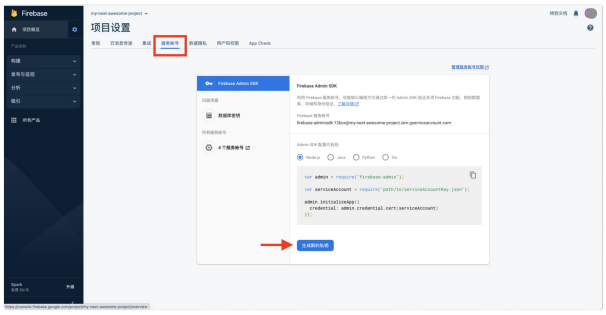
从融云发送到 FCM 的请求必须经过授权。因此，您需要向融云提供 FCM 项目相关的服务账号授权凭证。您需要手动将该凭证上传到控制台。

第 1 步：获取 Google 服务账号凭证

Firebase 项目支持 Google 服务帐号 。您需要从您的 Google 服务帐号获取凭据，然后授权融云服务器这些帐号凭据调用 Firebase 服务器 API。

如需对服务帐号进行身份验证并授予其访问 Firebase 服务的权限，您必须生成 JSON 格式的私钥文件。**如需为您的服务帐号生成 JSON 格式的私钥文件，请执行以下操作：**

1. 在 Firebase 控制台中，打开设置 > **服务帐号**。
2. 点击生成新的私钥，然后点击生成密钥进行确认。
3. 妥善存储包含密钥的 JSON 文件。稍后需要在控制台上传该文件。



第 2 步：在控制台上传 FCM 凭证

前往控制台，在**应用标识** > **Android** > **Google（FCM）** 标签页中，填写上一步在获取的凭证。

- **选择鉴权方式**：
  - **证书（推荐）**：上传从您的服务帐号生成的私钥 JSON 文件。
  - **API 密钥**（使用旧版 Server Key 授权方式和旧版 FCM API）：填入从您的服务帐号获取的服务器密钥。

> ⚠ 警告
> 如果您的项目中配置的鉴权方式中为 **API 密钥**，请注意自 2023 年 6 月 20 日起，使用 FCM XMPP 和 HTTP 旧版 API 发送消息（包括上游消息）已被 FCM 官方弃用，并将

> 于 2024 年 6 月移除，融云的推送服务后台届时将同步移除该能力。因此，您需要尽快迁移到使用证书（FCM 的 JSON 格式私钥文件）的鉴权方式，详细方法请参考上文获取 Google 服务账号凭证。

- **推送方式**（详见 Firebase 中文文档 FCM 消息类型 ）：
  - **透传消息方式**：由 Android OS 直接在通知面板弹出通知。融云仅下发对应 FCM「数据消息」类型的数据。
  - **通知消息方式**：IMLib SDK 接收到透传数据后，进行数据解析并弹出通知。开发者也可以自定义处理。融云下发包含 FCM「数据消息」和「通知消息」的数据。
- **intent**：自定义用户点击通知相关联的操作，对应 FCM 的 `click_action` 字段。如果指定，当用户单击通知时，默认将启动匹配该 Intent 的 Activity。最好将应用的软件包名称用作前缀，以确保唯一性。

  例如，在控制台指定 Intent 为：

  `com.yourapp.demo.ExampleActivity`

  在您 App 的 `AndroidManifest.xml` 中需要指定可以匹配该 Intent 的 Activity。

```
<activity android: name = ".ExampleActivity">
<intent-filter>
<action android: name = "com.yourapp.demo.ExampleActivity" />
...
</intent-filter>
</activity>
```

- **优先级**（详见 Firebase 中文文档 设置消息的优先级 ）
  - **普通**：应用在前台运行时，普通优先级消息会被立即传递。当应用在后台运行时，消息传递可能会延迟。如果是对时间不太敏感的消息（例如新电子邮件通知、使界面保持同步或在后台同步应用数据），建议您选择普通传递优先级。
  - **高**：即使设备处于低电耗模式，FCM 也会立即尝试传递高优先级消息。 高优先级消息适用于对时间敏感的用户可见内容。
- **推送通知标题**：在调用服务端 API 接口 `/push.json`、`/push/user.json`、 `/push/custom.json` 接口推送通知时，如未传入通知栏标题，则使用下方设置的“推送通知标题”。从服务端发消息时，如果发送用户 ID 在融云服务端没有用户名，也会使用此 “推送通知标题”。一般情况下客户端发送消息转 Push 时不显示此标题。
- **推送通道类型**：GCM 推送融云默认使用的是**默认**推送通道，如需要使用**私有**推送通道，可选择私有通道并设置 Channel ID 即可。

客户端启用 FCM 推送服务

1. 在 SDK init 之前，初始化 `RongPushPlugin` 模块。

```
RongPushPlugin.init(getContext());
```

   如果找不到 `RongPushPlugin` 模块，请检查是否已经集成融云自建推送通道。

2. （可选）启用 FCM 推送后，如需重新启用 Analytics 数据收集，请调用 `FirebaseAnalytics` 类的 `setAnalyticsCollectionEnabled()` 方法。例如：

```
setAnalyticsCollectionEnabled(true);
```

> 提示
>
> 如果不需要接收推送，可以通过设置 SDK 的初始化配置中的 `enablePush` 参数为 `false`，向融云服务申请禁用推送服务（当前设备）。您也可以在断开连接时设置不接收推送（当前设备）。

测试 FCM 推送

在完成上述步骤之后，可直接测试推送是否集成成功。

设备条件

建议直接使用在海外正式发售的 Android 设备测试 FCM 推送通知的接收。针对测试设备的详细要求如下：

> 提示
>
> 1. 测试设备必须使用真机。模拟器收不到远程推送。
> 2. 测试设备必须支持 Google GMS 服务（Google Mobile Services）。您可以直接使用在海外正式发售的 Android 设备。注意，在中国大陆地区发售的设备一般均未预装 GMS 服务。部分品牌的设备因无法获得授权，无法自行安装 GMS 服务。

网络条件

建议在模拟的海外网络环境下进行测试。详细条件如下：

> 提示
>
> 1. 如果在中国大陆地区进行测试，要求测试设备必须使用国外 IP 地址与融云建立 IM 连接，并使用海外 IP 访问融云服务。如果使用国内 IP，融云服务端认为该设备处于国内，且不会启用 FCM 通道。
> 2. 测试设备所处的网络环境必须可以正常访问 Google 网络服务。否则即使触发 FCM 推送，该设备也无法从 Google 的 FCM 服务收取推送。

测试步骤

假设在开发环境下进行测试。App 使用测试环境的 App Key。

具体步骤如下：

1. APP 连接融云成功之后杀掉 APP 进程。
2. 访问控制台的 IM Server API 调试 页面，切换到 App 的开发环境，找到**消息 > 消息服务 > 发送单聊消息**，直接发送一条单聊消息内容。
3. 查看手机是否收到本 APP 的推送。

故障排除

- 如果测试使用的 App 属于融云的海外数据中心（北美或新加坡），请检查是否已在 SDK 初始化之前执行 `setServerInfo` 设置海外数据中心的导航域名。详见海外数据中心。
- 从 Google FCM 服务后台直接发送消息到设备。如果未收到，请先根据上述文档自查是否正确集成。
- 检查 App 是否被设备为后台受限应用。在 Android P 或更高版本 上，FCM 将不会向被用户添加了后台限制（例如，通过“设置”->“应用和通知”->“[appname]”->“电池”实施后台限制）的应用发送推送通知。详见 Android 开发者文档 后台受限应用 。
- 不要频繁发同一内容的消息到同一台手机，有可能会被 FCM 服务端屏蔽导致无法收到推送。
- 检查测试设备是否为国行手机的 ROM。部分品牌已不支持自行安装 GMS 服务。建议您替换为海外发售的设备，或更新测试设备的系统。

如果问题无法解决，可提交工单 并提供您的消息 ID。

混淆脚本

```
-keep public class com.google.firebase.* {*;}
```

处理推送通知的点击事件

- **自定义推送通知点击事件**：介绍如何实现 SDK 默认跳转行为，以及如何自定义处理点击事件。详见自定义推送通知点击事件。
- **自定义推送通知样式**：仅在控制台 FCM 的消息推送方式为**透传消息方式**时支持。详见自定义推送通知样式。

针对 CallKit SDK 客户的说明

在控制台 FCM 的消息推送方式为**透传消息方式**时，IMKit/IMLib 默认不会为音视频信令消息（呼叫邀请、挂断等）弹出通知。可通过以下方式解决：

IM SDK 接收的 FCM 透传消息会拉起应用，可在 Application 的 `onCreate` 方法中即调用 IM 的连接方法。连接成功后 SDK 会主动弹出通知。

针对 CallLib SDK 客户的说明

在控制台 FCM 的消息推送方式为**透传消息方式**时，IMKit/IMLib 默认不会为音视频信令（呼叫邀请、挂断等）弹出通知。您可以通过**自定义推送通知样式**，重写 `PushEventListener` 的 `preNotificationMessageArrived` 方法，在该方法自行实现通知弹出逻辑。

解决推送客户端冲突 解决推送客户端冲突

更新时间:2024-08-30

本文描述了如何避免或解决在应用程序中可能出现的推送客户端冲突的问题。

出现推送客户端冲突，可能由以下情况导致：

- 应用程序自身已集成第三方推送客户端，与融云推送 2.0 SDK 中的第三方客户端冲突
- 应用程序中依赖的其他 SDK 已集成第三方推送客户端，与融云推送 2.0 SDK 中的第三方客户端冲突

如果存在上述情况，您可以选择是否继续使用融云推送 2.0 SDK。

不再使用融云推送 2.0 SDK

如果您不需要使用融云推送 2.0 SDK，请按如下步骤进行处理：

1. 参照推送 2.0 集成文档，完成控制台的第三方推送配置。

2. 在 SDK init 之前，必须创建 PushConfig，根据您使用的厂商启用推送通道，并将配置提供给 RongPushClient。

```
PushConfig config = new PushConfig.Builder()
.enableFCM(true)
.enableHWPush(true)
.enableVivoPush(true)
.enableMiPush("小米 appId", "小米 appKey") //小米推送配置
.enableMeiZuPush("魅族 appId", "魅族 appKey") // 魅族推送配置
.enableOppoPush("OPPO_App_KEY", "OPPO_App_Secret") // OPPO 推送配置
.build();
RongPushClient.setPushConfig(config); //将推送相关配置设置到 SDK
```

3. 从第三方推送 SDK 获取厂商 token。具体方式请您查看第三方推送 SDK 文档。

4. 由于 RongPushClient 会在 SDK 初始化异步执行，且暂不提供初始化完成的通知，建议您延时 3 秒左右调用 onReceiveToken 方法向融云上报推送第三方 Token，否则 SDK 无法正常上报推送相关统计数据。

```
PushManager.getInstance().onReceiveToken(context,pushType,token);
```

5. 通过日志查看是否上报成功，搜索关键字 L-push-config-report-token-R，提示 report token success，上报成功集成成功。

继续使用融云推送 2.0 SDK

如果您希望继续使用融云推送 2.0 SDK，可移除应用自身集成的推送 SDK，从融云推送 2.0 SDK 的回调中获取厂商推送 Token，上报给第三方推送厂商，或第三方推送集成商。

您可以在 Application 的 onCreate 方法中设置 PushEventListener，通过 onTokenReceived(PushType pushType, String token); 回调方法获取厂商推送 Token。

```
RongPushClient.setPushEventListener(
new PushEventListener() {

@Override
public void onTokenReceived(
PushType pushType,
String token) {
}

/// 此处省略其他方法

});
```

自定义推送通知样式 自定义推送通知样式

更新时间:2024-08-30

接收到推送消息后，系统会弹出通知，显示推送消息。

- **FCM 类型推送**：针对 FCM 类型推送方式，如果在控制台设置的**推送方式**为**透传消息方式**，可通过以下方式自定义通知样式。详见集成 FCM。
- **融云自建推送**：可通过以下方式自定义通知样式。但因到达率较低，已不推荐使用。
- **其他第三方类型推送**：SDK 接收到其他第三方厂商的推送后，弹出的通知是系统通知，由手机系统底层直接弹出通知，所以不支持自定义。

自定义通知样式

如果您使用 FCM 推送，且 FCM 后台配置消息**推送方式**为**透传消息方式**时，可通过以下方式自定义通知样式。

使用 PushEventListener

您可以设置 PushEventListener，覆写 preNotificationMessageArrived 方法，拦截通知事件并自定义显示。

> 提示
> - SDK 从 5.1.0 版本开始支持该功能。
> - 请在 Application 的 `onCreate` 方法中注册监听器。
> - 由于 Android 12 通知 trampoline 限制的原因，当您的 App 的 targetVersion >= 31 时，建议直接在 `RongPushClient.setPushEventListener` 中的回调中直接启动 Activity。请勿再通过广播或者服务再进行消息分发之后再启动 Activity。

```
RongPushClient.setPushEventListener(
new PushEventListener() {
@Override
public boolean preNotificationMessageArrived(
Context context,
PushType pushType,
PushNotificationMessage notificationMessage) {
// 该回调仅在通知类型为透传消息时生效。返回 true 表示拦截，false 为不拦截
return false;
}

@Override
public void afterNotificationMessageArrived(
Context context,
PushType pushType,
PushNotificationMessage notificationMessage) {
// 该回调仅在通知类型为透传消息时生效
}

@Override
public boolean onNotificationMessageClicked(
Context context,
PushType pushType,
PushNotificationMessage notificationMessage) {
// 用户可以在此定义自己的通知的点击事件业务，返回 true 表示拦截，false 为不拦截
return false;
}

@Override
public void onThirdPartyPushState(
PushType pushType, String action, long resultCode) {}
});
```

使用 PushMessageReceiver

您也可以通过 PushMessageReceiver 的 onNotificationMessageArrived 方法自定义 FCM 通知样式。该方式优先级低于 PushEventListener。

> 提示
> 由于 Android 12 通知 trampoline 限制的原因，当您的 App 的 targetVersion >= 31 时，建议直接在 `PushMessageReceiver` 的回调中直接启动 Activity。请勿再通过广播或者服务再进行消息分发之后再启动 Activity。

1. 创建自定义 YourCustomPushMessageReceiver，并继承 PushMessageReceiver。在 onNotificationMessageArrived（通知到达事件）中可收到推送内容。

```
public class YourCustomMessageReceiver extends PushMessageReceiver {

@Override
public boolean onNotificationMessageClicked(
Context context, PushType pushType, PushNotificationMessage message) {
if (pushType.equals(PushType.GOOGLE_FCM)){

// TODO
}
// 返回 true 表示拦截，false 为不拦截
return true;
}
}
```

2. 在主工程的 AndroidManifest.xml 中注册您自定义的推送广播接收组件。请注意替换 AndroidManifest.xml 中已添加的 PushMessageReceiver 注册内容。

```
<receiver
android:name="xxx.YourCustomPushMessageReceiver"
android:exported="true">
<intent-filter>
<action android:name="io.rong.push.intent.MESSAGE_ARRIVED" />
<action android:name="io.rong.push.intent.MESSAGE_CLICKED" />
<action android:name="io.rong.push.intent.THIRD_PARTY_PUSH_STATE" />
</intent-filter>
</receiver>
```

自定义推送点击事件 自定义推送点击事件

更新时间:2024-08-30

本文描述了 IMLib 和 IMKit 处理用户点击远程通知（Remote notifications）的默认行为，以及应用程序如何自行处理远程推送通知的点击事件。

> 提示
>
> 本文仅描述远程推送通知点击事件的处理方案。如果您需要处理 IMKit 的本地通知的点击事件，请参考本地通知点击事件处理。如果您不清楚如何区分本地通知与远程推送，可阅读知识库文档 如何理解即时通讯业务中的实时消息、本地通知和远程推送（Android）。

实现 SDK 默认跳转行为

IMLib 和 IMKit 默认处理远程推送通知的点击事件。用户点击通知时，SDK 会发出对应的隐式 Intent。您需要在 App 的应用清单文件（`AndroidManifest.xml`）配置添加 Intent-filter，以接收 Intent，完成默认跳转动作。

- **来自一个联系人的通知**：用户点击来自一个联系人发来一条或多条通知时，SDK 默认跳转到会话 Activity。
- **来自多个联系人的折叠通知**：多个联系人发来多条通知时，这些通知会折叠显示。用户点击来自多个联系人的通知时，SDK 默认跳转到会话列表 Activity。
- **不落地推送的通知**：当用户点击不落地推送的通知时，SDK 也会发出隐式 Intent。您可以自行决定接收该 Intent 的 Activity。

跳转到与消息对应的会话页面

用户点击来自一个联系人发来一条或多条通知时，SDK 默认发出跳转到会话 Activity 的 Intent。

在 `AndroidManifest.xml` 的会话 Activity 中配置如下 `intent-filter` 即可跳转到会话 Activity。如果自定义了会话 Activity，需要替换 会话 `activity` 为自定义 Activity 的类名。

```
<activity
android:name="RongConversationActivity"
android:exported="true">
<intent-filter>
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />
<data
android:host="${`applicationId`}"
android:pathPrefix="/conversation"
android:scheme="rong" />
</intent-filter>
</activity>
```

跳转到会话列表页面

多个联系人发来多条通知时，这些通知会折叠显示。用户点击来自多个联系人的通知时，SDK 默认发出跳转到会话列表 Activity 的 Intent。

在 `AndroidManifest.xml` 的会话列表 Activity 中配置如下 `intent-filter` 即可跳转到当前 Activity。如果自定义了会话列表 Activity，需要替换 `RongConversationListActivity` 为自定义 Activity 的类名。

```
<activity
android:name="RongConversationListActivity"
android:exported="true">
<intent-filter>
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />
<data
android:host="${`applicationId`}"
android:pathPrefix="/conversationlist"
android:scheme="rong" />
</intent-filter>
</activity>
```

处理不落地通知点击跳转

通过融云发起的不落地通知会独立显示。用户点击不落地通知时，SDK 默认发出 Intent。

在 `AndroidManifest.xml` 进行如下配置，即可跳转到您指定的 Activity。

```
<activity
android:name="自定义 activity"
android:exported="true"
android:launchMode="singleTask"
android:screenOrientation="portrait">

<intent-filter>
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />

<data
android:host="${`applicationId`}"
android:pathPrefix="/push_message"
android:scheme="rong" />
</intent-filter>
</activity>
```

自定义通知点击跳转行为

如果默认跳转行为无法满足需求，您可以通过以下任一方式，修改点击推送通知的跳转行为。

- 设置 `PushEventListener`，重写 `onNotificationMessageClicked` 方法（推荐）。
- 继承 `PushMessageReceiver`，重写 `onNotificationMessageClicked` 方法。

设置 PushEventListener

> 提示
>
> - 开发版 SDK 从 5.2.1 版本开始，所有推送类型均支持使用 `PushEventListener` 自定义点击通知跳转行为。
> - 稳定版 SDK 从 5.1.9.2 版本开始，所有推送类型均支持使用 `PushEventListener` 自定义点击通知跳转行为。
> - 由于 Android 12 通知 trampoline 限制的原因，当您的 App 的 targetVersion >= 31 时，建议直接在 `RongPushClient.setPushEventListener` 中的回调中直接启动 Activity。请勿再通过广播或者服务再进行消息分发之后再启动 Activity。

您可以在 Application 的 `onCreate` 方法中设置 `PushEventListener`，覆写 `onNotificationMessageClicked` 方法，监听并拦截推送通知点击事件。

```
RongPushClient.setPushEventListener(
new PushEventListener() {
@Override
public boolean preNotificationMessageArrived(
Context context,
PushType pushType,
PushNotificationMessage notificationMessage) {
// 该回调仅在通知类型为透传消息时生效。返回 true 表示拦截，false 为不拦截
return false;
}

@Override
public void afterNotificationMessageArrived(
Context context,
PushType pushType,
PushNotificationMessage notificationMessage) {
// 该回调仅在通知类型为透传消息时生效
}

@Override
public boolean onNotificationMessageClicked(
Context context,
PushType pushType,
PushNotificationMessage notificationMessage) {
// 用户可以在此定义自己的通知的点击事件业务，返回 true 表示拦截，false 为不拦截
return false;
}

@Override
public void onThirdPartyPushState(
PushType pushType, String action, long resultCode) {}
});
```

使用 PushMessageReceiver

您也可以通过 `PushMessageReceiver` 的 `onNotificationMessageClicked` 方法自定义点击事件。该方式优先级低于 `PushEventListener`。

> 提示
>
> - 华为 和 Oppo 的推送点击事件无法通过 `PushMessageReceiver` 方式进行自定义。
> - 由于 Android 12 通知 trampoline 限制的原因，当您的 App 的 targetVersion >= 31 时，建议直接在 `PushMessageReceiver` 中的回调中直接启动 Activity。请勿再通过广播或者服务再进行消息分发之后再启动 Activity。

1. 创建 `CustomPushMessageReceiver` 继承 `PushMessageReceiver` 类。

```
class CustomPushMessageReceiver extends PushMessageReceiver {
...
}
```

2. 在主工程的 AndroidManifest.xml 中注册推送广播接收组件。请注意替换 AndroidManifest.xml 中已添加的 PushMessageReceiver 注册内容。

```
<receiver
android:name="xxx.CustomPushMessageReceiver"
android:exported="true">
<intent-filter>
<action android:name="io.rong.push.intent.MESSAGE_ARRIVED" />
<action android:name="io.rong.push.intent.MESSAGE_CLICKED" />
<action android:name="io.rong.push.intent.THIRD_PARTY_PUSH_STATE" />
</intent-filter>
</receiver>
```

3. 注册完成广播后，可在创建的 CustomPushMessageReceiver 类中复写 onNotificationMessageClicked 方法。在此方法中做拦截，然后实现跳转逻辑。

参数说明

参数	类型	必填	说明
context	Context	是	上下文对象
pushType	PushType	是	推送类型
notificationMessage	PushNotificationMessage	是	推送消息

```
class CustomPushMessageReceiver extends PushMessageReceiver {
@Override
public boolean onNotificationMessageClicked(Context context, PushType pushType, PushNotificationMessage notificationMessage) {
// 可通过 pushType 判断 Push 的类型
// 华为 和 Oppo的推送点击事件不会回调到该方法，需要按照各自的配置方法在相应的activity解析intent获得信息.

if (pushType == PushType.XIAOMI) {
//实现您自定义的通知点击跳转逻辑
return true; // 此处返回 true. 代表不触发 SDK 默认实现，您自定义处理通知点击跳转事件。
} else if (pushType == PushType.MEIZU) {
//实现您自定义的通知点击跳转逻辑
return true; // 此处返回 true. 代表不触发 SDK 默认实现，您自定义处理通知点击跳转事件。
} else if (pushType == PushType.VIVO) {
//实现您自定义的通知点击跳转逻辑
return true; // 此处返回 true. 代表不触发 SDK 默认实现，您自定义处理通知点击跳转事件。
} else if (pushType == PushType.GOOGLE_FCM) {
//实现您自定义的通知点击跳转逻辑
return true; // 此处返回 true. 代表不触发 SDK 默认实现，您自定义处理通知点击跳转事件。
} else if (pushType == PushType.RONG) {
//实现您自定义的通知点击跳转逻辑
return true; // 此处返回 true. 代表不触发 SDK 默认实现，您自定义处理通知点击跳转事件。
}
return false;
}
}
```

配置推送铃声 配置推送铃声

更新时间:2024-08-30

远程推送铃声是指收到来自推送通道的通知时播放的铃声。App 仅会在关闭状态下收到离线推送通知。默认使用手机系统设置的声音与振动提示状态。

> 提示
>
> 在 App 退到后台且并未被系统回收的情况下不会触发远程推送。如果您集成 IMKit，仅会触发 IMKit 的本地通知，此时播放本地通知的铃声。如果您集成 IMLib，则没有本地通知。

融云目前支持为指定的消息类型配置自定义铃声，已适配**小米推送**、**华为推送**、**FCM 推送**。

选择消息类型

首先，请确认需要自定义铃声的消息内容类型唯一标识（Object Name）。如需查询，参见消息类型概述。

准备铃声资源文件

您需要将自定义声音资源的文件名已打包到应用程序中，用户接收该类型的推送消息时，自动读取设置的文件进行声音提醒。以下是各厂商对文件的路径要求：

- **小米推送**：`android.resource://<package-name>/raw/<filename>`，其中 `<package-name>` 需要替换为您的应用包名。`<filename>` 需要替换为不带后缀名的铃声文件名（存储的声音文件需要有扩展名，但是不要把扩展名写在 uri 中）。
- **华为推送**：`/res/raw/`。
- **FCM 推送**：`/res/raw/`

创建 Channel ID

创建自定义通知渠道。在控制台配置自定义推送铃声时需要配置该 Channel ID。

小米推送

> 提示
>
> Android 8.0 以上小米手机需要设置 Channel 后，才能自定义推送铃声。详见 小米推送开发文档。Channel 一旦创建并发送了带有 Channel 的消息，设备上即会生成这个 Channel，不能删除也不能修改，所以请谨慎创建 Channel。

请参照小米文档创建 Channel ID：

- 使用小米开放平台上创建 Channel ID
- 使用小米服务端 API 创建 Channel ID
- 使用小米推送服务端 SDK 创建 Channel ID（此处仅提供小米 Push Server Java SDK 文档链接，其他语言请参照小米推送官方文档。）

华为推送

> 提示
>
> 受华为平台限制，在华为平台设置数据处理位置为中国时，不支持自定义渠道，无法使用自定义铃声。其他地区注册 Channel ID 后支持设置自定义铃声。

App 端调用 Android SDK 接口，创建应用专有的推送通知渠道。可在 MainActivity 进行初始化，或者 Application 进行初始化，具体代码示例如下：

```
private void initNotificationChannel() {
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
//增加铃声渠道,可选择资源文件里面的音频文件
String channelIdKanong = "NotificationKanong";
String channelNameKanong = "铃声Kanong";
createNotificationChannel(channelIdKanong, channelNameKanong, importance, R.raw.kanong);
}

//创建通知渠道
@RequiresApi(api = Build.VERSION_CODES.O)
private void createNotificationChannel(String channelId, String channelName, int importance, int rawSource) {
NotificationChannel channel = new NotificationChannel(channelId, channelName, importance);
if (rawSource != 0) {
// 存储的声音文件需要有扩展名，但是不要把扩展名写在 uri 中
String uriStr = "android.resource://" + this.getPackageName() + "/" + rawSource;
Uri uri = Uri.parse(uriStr);
channel.setSound(uri, Notification.AUDIO_ATTRIBUTES_DEFAULT);
}
NotificationManager notificationManager = (NotificationManager) getSystemService(
NOTIFICATION_SERVICE);
notificationManager.createNotificationChannel(channel);
}
}
```

FCM 推送

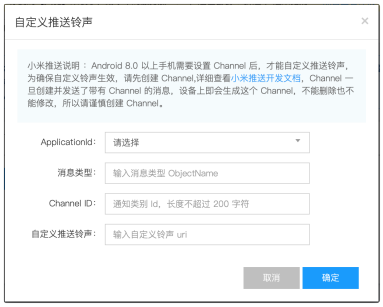
详见 Android 官方文档通知渠道 和 FCM 推送开发文档 。

配置消息类型对应的铃声

确认应用中已打包铃声资源文件后，您需要在控制台按消息类型、Channel ID 配置对应的自定义推送铃声。目前控制台已支持设置小米推送、华为推送、FCM 推送。每个推送厂商下最多设置 5 个自定义铃声，设置 30 分钟后生效。

1. 访问控制台 自定义推送铃声 页面，在 **小米**、**华为**、**FCM** 标签下点击**添加**。

2. 选择 ApplicationID，输入 Channel ID、消息类型名称、以及自定义铃声文件资源地址。以配置小米自定义推送铃声为例：



设置成功后，所有小米手机用户离线状态下接收的该类型消息，都会使用设置的 Channel ID 进行推送，提醒铃声为注册 Channel 时设置的自定义铃声。

用户级推送配置 用户级推送配置

更新时间:2024-08-30

用户级别推送配置是指针对 App 当前登录用户的推送配置。

> 提示
> - 用户级别推送配置区别于 App Key 级别推送配置。App Key 级别的推送配置针对 App 下所有用户。您可以在控制台调整部分 App Key 级别的推送服务配置。
> - 用户级别推送配置要求 App Key 已开通用户级别功能设置。如需开通，请提交工单。

以下配置适用于 IMKit 或 IMLib，或其他依赖 IMLib/IMKit 的客户端 SDK。

设置用户推送语言偏好

为当前登录用户设置推送通知的展示语言偏好。在用户未设置偏好的情况下，使用 App Key 级别的 **Push** 语言设置。

融云内置消息类型的默认推送内容中含有部分格式文本字符串。例如，默认情况下用户收到单聊会话的文件消息推送时，推送通知内容中将显示简体中文字符串“[文件]”。如果用户将自己的推送语言偏好修改为美国英语 en_US，则再接收到文件消息时，通知内容中的格式文本字符串将为“[File]”。

上例中的 “[文件]” “[File]” 即格式文本字符串。目前融云服务端为内置消息类型的推送内容提供了格式文本字符串，支持简体中文 zh_CN、美国英语 en_US、阿拉伯语 ar_SA。

```
RongIMClient.getInstance().setPushLanguageCode(languageCode,
new RongIMClient.OperationCallback() {

/**
* 成功回调
*/
@Override
public void onSuccess() {

}

/**
* 错误回调
* @param errorCode 错误码
*/
@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

设置成功后，当前用户接收内置消息类型的推送通知时，推送内容中的格式文本字符串会根据对应语种进行调整。

参数	类型	说明
languageCode	String	设置推送通知显示的语言。目前融云支持的内置推送语言为 zh_CN、en_US、ar_SA。自定义推送语言请与 控制台 > 自定义推送文案 中的语言标识保持一致。
callback	OperationCallback	操作回调

目前融云支持的内置推送语言为简体中文 zh_CN、美国英语 en_US、阿拉伯语 ar_SA。App 可以配合使用自定义多语言推送模板功能，可以实现在一条推送通知中支持更多推送语言。

您可以修改 App Key 在融云的默认推送语言配置。如有需要，请提交工单申请更改 App Key 级别的 **Push** 语言。

设置用户推送通知详情偏好

在用户未设置偏好的情况下，默认会展示推送通知内容。该功能支持当前登录用户设置自己的推送通知中是否需要展示推送通知的内容详情。

```
boolean showStatus = false; // 不显示推送通知详情

RongIMClient.getInstance().setPushContentShowStatus(showStatus,
new RongIMClient.OperationCallback() {

/**
* 成功回调
*/
@Override
public void onSuccess() {

}

/**
* 错误回调
* @param errorCode 错误码
*/
@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

如果设置为不显示详情，推送通知将显示为格式文本字符串"您收到了一条通知"（该格式文本字符串支持简体中文 zh_CN、美国英语 en_US、阿拉伯语 ar_SA）。

参数	类型	说明
showStatus	boolean	是否显示推送详情。`true`：显示详情。`false`：不显示详情。
callback	OperationCallback	操作回调

请注意，发送消息时可以指定 `forceShowDetailContent` 强制越过消息接收者的该项配置，强制显示推送通知内容详情。以下列出了部分平台的配置：

- Android：`Message` 的 MessagePushConfig 属性。参见 Android 端「发送消息」文档中的 **MessagePushConfig 属性说明**。
- iOS：`RCMessage` 的 RCMessagePushConfig 属性。参见 iOS 端「APNs 推送开发指南」下的**配置消息的推送属性**。
- Web：IPushConfig
- IM Server API：若接口提供 `forceShowPushContent` 参数，则支持该功能。

您也可以修改 App Key 在融云的默认配置。如有需要，请提交工单申请更改 App Key 级别的 **推送通知详情**。关闭后，所有推送通知均默认不显示推送内容详情。

设置用户多端时接收推送偏好

为当前登录用户设置在 Web 端或 PC 端在线时，离线的移动端设备是否需要接收推送通知。请注意，该接口仅在 App Key 已开通 **Web/PC 在线手机端接收 Push** 服务后可用。

您可以前往控制台的免费基础功能页面修改 App Key 级别配置。

- 如果 App Key 未开通 **Web/PC 在线手机端接收 Push**，则所有 App 用户在 Web 端或 PC 端在线时，不在线的移动端不会收到推送。不支持 App 用户修改自己的偏好。
- 如果 App Key 已开通 **Web/PC 在线手机端接收 Push**，当前登录用户可自行关闭或开启该行为。

```
boolean receiveStatus = false; // 当前用户希望 Web/PC 在线时不接收推送通知

RongIMClient.getInstance().setPushReceiveStatus(receiveStatus,
new RongIMClient.OperationCallback() {

/**
* 成功回调
*/
@Override
public void onSuccess() {

}

/**
* 错误回调
* @param errorCode 错误码
*/
@Override
public void onError(RongIMClient.ErrorCode errorCode) {

}
});
```

成功设置为 `false` 后，当前用户如果在 Web/PC 端在线，则移动端不会接收推送通知。

参数	类型	说明
receiveStatus	boolean	其它端在线时，移动端是否接受推送。`true`：接收推送。`false`：不接收推送。
callback	OperationCallback	操作回调

上报推送数据 上报推送数据

更新时间:2024-08-30

融云提供推送统计数据功能，您可以在控制台推送成功率统计 页面查看推送服务的统计数据。

> 提示
>
> 推送数据统计功能仅针对已上线应用的生产环境。如果是海外数据中心的应用，请根据 SDK 版本完成配置，确保数据上报到正确的数据中心，详见知识库文档融云海外数据中心使用指南 。

推送数据统计包括：

- **推送总量**：表示实际需要推送的通知条数，包括厂商推送和融云推送。
- **推送成功总量**：表示成功推送到第三方 Push 厂商、融云 Push 服务的通知数，推送成功并不代表实际已推送到了目标设备上。
- **推送失败总量**：表示推送到第三方 Push 厂商、融云 Push 服务失败的通知数。
- **推送到达总量（需配置）**：表示手机设备实际已经收到的通知数，部分系统手机需要终端主动上报达到数据。暂仅支持上报单聊、群聊会话类型的推送到达数据，不支持上报超级群推送和不落地推送到达数据。
- **推送点击总量（需配置）**：表示终端用户点击通知总数，部分系统手机需要终端主动上报点击数据。

因为不同手机厂商推送服务设计差异，融云无法直接获取部分厂商的推送到达和推送点击数据，需要您在第三方推送平台进行配置或者客户端自行上报。

检查 SDK 初始化时机

如果用户点击推送通知时融云 SDK 尚未完成初始化，SDK 将无法上报推送数据。

建议您在集成时检查 Application 中需要初始化的组件，将融云 SDK 初始化尽量提前，确保融云可准确上报推送相关数据。

采集推送到达数据

推送到达是指通知已发送到第三方厂商推送通道或融云推送服务后，推送通知成功下发到目标设备。

华为、魅族推送到达数据依赖手机厂商推送通道提供的「送达回执」服务。Google FCM 仅支持采集透传消息方式的推送到达数据。详细支持情况参见下表：

推送平台	提供推送到达数据	是否需要配置
华为	厂商支持	需要，参见配置华为推送回执
荣耀	厂商支持	需要，参见配置荣耀推送回执
魅族	厂商支持	需要，参见配置魅族推送回执
小米	厂商支持	无需配置
Vivo	厂商支持	无需配置
OPPO（包括 Realme、OnePlus）	厂商支持	无需配置
Google FCM	厂商不支持，由融云 SDK 提供	详见采集 FCM 推送到达数据

配置华为推送回执

> 提示
>
> 在配置之前，请确保已完成集成华为推送。

完成华为推送通道配置集成后，需要开发者在华为开放平台开通并配置消息回执，才能获取到华为推送送达设备数据。详细可参考华为开放平台消息回执，配置流程如下：

1. 登录华为开放平台，在 Push 开发服务中选择对应的应用，进入推送配置页面。

2. 进入配置界面后，开通应用回执状态。

3. 点击**新建回执**后，需要配置如下参数。

- 回执名称：填写自定义名称。
- 回调地址：请务必根据应用所属数据中心配置对应的推送回执回调地址。
  - 国内（北京）数据中心（推荐）：`https://callback.rong-edge.com/push/callback/huawei`
  - 国内（北京）数据中心（旧）：`https://callback-bj.ronghub.com/push/callback/huawei`
  - 新加坡数据中心回调地址：`https://callback.sg-light-edge.com/push/callback/huawei`
  - 新加坡 B 企业合作数据中心：`https://callback.sg-b-light-edge.com/push/callback/huawei`
  - 北美数据中心回执地址：`https://callback.us-light-edge.com/push/callback/huawei`
  - 沙特数据中心回执地址：`https://callback.sau-light-edge.com/push/callback/huawei`
- 回调用户名：请留空
- 回调密钥：请留空
- 支持版本：V1、V2 版本均已适配，建议选择 V2 版本。已在华为平台配置为 V1 版本的客户，建议尽早修改为 V2 版本。

4. 配置完成后，点击**测试回执**，提示成功后可点**提交**配置完成。

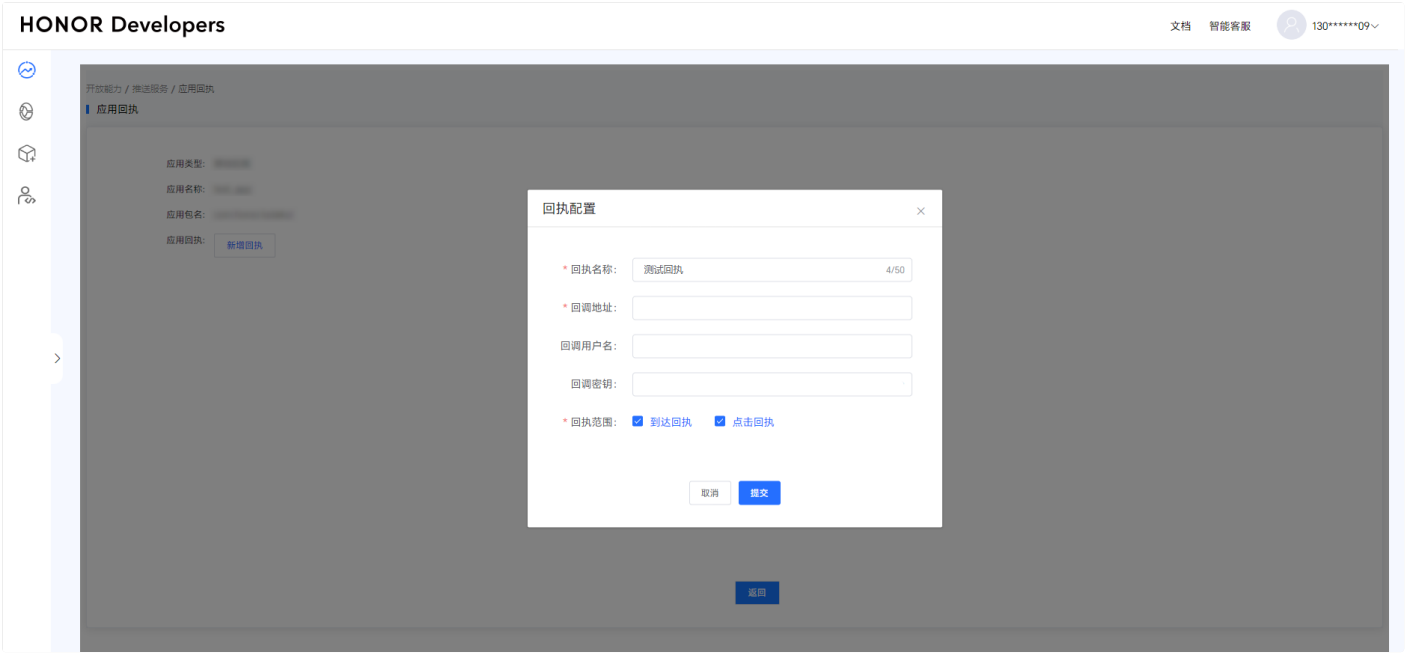
配置荣耀推送回执

> 提示
> 在配置之前，请确保已完成集成荣耀推送。

完成荣耀推送通道配置集成后，需要开发者在荣耀开放平台开通并配置消息回执，才能获取到荣耀推送送达设备数据。详细可参考荣耀推送服务开通消息回执指南，配置流程如下：

1. 登录荣耀开发者服务平台，选择 **管理中心** > **生态服务** > **开发服务** > **推送服务**，进入推送服务页面。选择需要配置回执的应用，点击**应用回执** ，进入应用回执页面。

2. 点击 **新增回执** ，进入回执配置页面。



配置信息如下：

- 回执名称：您自定义的名称，长度不能超过 50
- 回调地址：请务必根据应用所属数据中心配置对应的推送回执回调地址。
  - 国内（北京）数据中心（推荐）：`https://callback.rong-edge.com/push/callback/honor`
  - 国内（北京）数据中心（旧）：`https://callback-bj.ronghub.com/push/callback/honor`
  - 新加坡数据中心回调地址：`https://callback.sg-light-edge.com/push/callback/honor`
  - 新加坡 B 企业合作数据中心：`https://callback.sg-b-light-edge.com/push/callback/honor`
  - 北美数据中心回执地址：`https://callback.us-light-edge.com/push/callback/honor`
  - 沙特数据中心回执地址：`https://callback.sau-light-edge.com/push/callback/honor`
- 回调用户名：请留空
- 回调密钥：请留空
- 回执范围：请勾选到达回执和点击回执

配置魅族推送回执

> 提示
>
> 在配置之前，请确保已完成集成魅族推送。

完成魅族推送通道集成后，需要开发者先在**魅族推送平台**中新建回执，再在**控制台**开启“魅族推送送达回执”后，才能获取到魅族通道送达数据，配置流程如下：

1. 登录魅族推送平台，选择需要配置回执的应用，单击**打开应用**。

2. 进入应用后，选择**配置管理** > **回执管理**，新增**回执地址**。

北京数据中心回执地址：`https://callback.rong-edge.com/push/callback/meizu`

3. 魅族平台设置完回执地址后，需要在控制台**魅族推送配置**中开启**推送送达回执**，才能获取到魅族通道送达数据。

采集 FCM 推送到达数据

> 提示
> - 在配置之前，请确保已完成集成 FCM 推送。
> - 请确认 FCM 推送方式配置为**透传消息方式**。融云不支持在**通知消息方式**采集推送送达到数据。

仅在控制台 FCM 推送方式配置为**透传消息方式**时，支持采集 FCM 推送到达数据。

App 需要手动上报 FCM 推送到达数据，方法如下：

调用 `RongPushClient.setPushEventListener` 设置推送事件监听器，在接口方法 `preNotificationMessageArrived` 中调用 `RongPushClient.recordPushArriveEvent` 上报推送到达数据。

```
RongPushClient.setPushEventListener(
new PushEventListener() {
@Override
public boolean preNotificationMessageArrived(
Context context,
PushType pushType,
PushNotificationMessage notificationMessage) {
RongPushClient.recordPushArriveEvent(
context, pushType, notificationMessage);
return false;
}

...
});
```

采集推送点击数据

推送点击数据是指表示终端用户点击通知的总数。

融云可直接获取大部分厂商的推送通道的点击数据。如果融云客户端 SDK 版本低于 5.2.3，华为推送通道下发的推送通知点击数据需要您在客户端手动上报。

详细支持情况参见下表：

推送平台	推送点击事件上报	描述
RongPush	支持	融云 SDK 默认实现上报逻辑
小米	支持	融云 SDK 默认实现上报逻辑
华为	支持（要求 SDK ≧ 5.1.4）	从 IMLib 5.2.3 版本开始，融云 SDK 默认实现上报逻辑（低于该版本则需要调用融云 SDK 手动上报点击事件）
荣耀	支持	融云 SDK 默认实现上报逻辑
Vivo	支持	融云 SDK 默认实现上报逻辑
魅族	支持	融云 SDK 默认实现上报逻辑
OPPO	支持	融云 SDK 默认实现上报逻辑
Google FCM	支持	融云 SDK 默认实现上报逻辑

采集华为推送点击数据

> 提示
>
> 在配置之前，请确保已完成集成华为推送。从 SDK 5.1.4 版本开始，支持采集华为推送点击数据，低于该版本无法采集推送点击数据。

从 SDK 5.2.3 版本开始，融云 SDK 内部默认上报华为推送点击数据，无需您进行处理。

如果融云客户端 SDK 版本低于 5.2.3，则需要您手动上报华为推送通知点击数据，方法如下：

1. 控制台华为推送 intent 使用默认值，没有自定义时，在会话列表页面 activity 的 onCreate() 中调用如下方法：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
if(intent != null && intent.getData() != null
&& intent.getData().getScheme() != null
&& intent.getData().getScheme().equals("rong")
&& intent.getData().getQueryParameter("isFromPush") != null
&& intent.getData().getQueryParameter("isFromPush").equals("true")) {
RongPushClient.recordHWNotificationEvent(intent); //上报点击事件
}
...
}
```

2. 自定义了华为推送 intent, 需要在自定义配置的 activity 里拦截 intent, 并调用以下上报方法。

```
RongPushClient.recordHWNotificationEvent(intent);
```

附录：融云回调服务 HTTPS 证书

如有推送平台要求提供回调服务器的 HTTPS 证书，您可以使用以下证书。

- （推荐）更新北京数据中心回执地址为 `https://callback.rong-edge.com/push/callback/huawei`，对应 HTTPS 证书如下（有效期至 2024-11-28）：

```
-----BEGIN CERTIFICATE-----
MIIHFDCCBXygAwIBAgIQD7iyosOKQGwk/WonYz+eQDANBgkqhkiG9w0BAQwFADBZ
MQswCQYDVQQGEwJDTjElMCMGA1UEChMcVHJ1c3RBc2lhIFRlY2hub2xvZ2llcywg
SW5jLjEjMCEGA1UEAxMaVHJ1c3RBc2lhIFJTQSBPViBUTFMgQ0EgRzIwHhcNMjMw
NDI2MDAwMDAwWhcNMjQwNDI1MjM1OTU5WjBwMQswCQYDVQQGEwJDTjESMBAGA1UE
CAwJ5YyX5Lqs5biCMTMwMQYDVQQKDCrljJfkuqzkupHkuK3ono3kv6HnvZHnu5zn
p5HmioDmnInpmZDlhazlj7gxGDAWBgNVBAMMDyoucm9uZyllZGdlLmNvbTCCASIw
DQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAJ+eS+CUXovisSlSEYeP+jz3Ufr1
XkMnkFCyZ8nHoitVgrzyUUQFrcrFW4vRaeo0FeqPs38C/KrW8fut78LEOnyoIu6w
hVgPACpA+p6mlBEhLzYwEIvI9cmJrr0IuXjZiVzcgQysCCUdrWo+E0T3kPFGdCu1
DrVT2qVMywBwfxokOvZKZKCAt3PPy8jxBxq/sprFghUvCAMAFtWCD507N216W2Vf
143NxpOWvARriA7n2UnzmcY1Qr/4rZre1cqX3LC/BK37bCrHVUIGcEr8JI8zdF7G
YPIlxtn0stbxjuriRJO938rvxH9pD/ynP5VejDRRmran9s4StYQu73QNyZECAwEA
AaOCAz8wggM7MB8GA1UdIwQYMBaAFMoxITcM3jm30istVZpcqalcUPomMB0GA1Ud
DgQWBBT8RqJ49OUDOqSVbFY6bxgiFOxrqzAOBgNVHQ8BAf8EBAMCBaAwDAYDVR0T
AQH/BAIwADAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIwSQYDVR0gBEIw
QDA0BgsrBgEEAbIxAQICMTAlMCMGCCsGAQUFBwIBFhdodHRwczovL3NlY3RpZ28u
Y29tL0NQUzAIBgZngQwBAgIwRwYDVR0fBEAwPjA8oDqgOIY2aHR0cDovL2NybC50
cnVzdC1wcm92aWRlci5jbi9UcnVzdEFzaWFSU0FPVlRMU0NBRzIuY3JsMH0GCCsG
AQUFBwEBBHEwbzBCBggrBgEFBQcwAoY2aHR0cDovL2NydC50cnVzdC1wcm92aWRl
ci5jbi9UcnVzdEFzaWFSU0FPVlRMU0NBRzIuY3J0MCkGCCsGAQUFBzABhh1odHRw
Oi8vb2NzcC50cnVzdC1wcm92aWRlci5jbjApBgNVHREEIjAggg8qLnJvbmctZWRn
ZS5jb22CDXJvbmctZWRnZS5jb20wggF8BgorBgEEAdZ5AgQCBIIBbASCAWgBZgB1
AHb/iD8KtvuVUcJhzPWHujS0pM27KdxoQgqf5mdMWjp0AAABh7s7gZAAAAQDAEYw
RAIgN0nPOOHS1Sn6Fot7rzhFDv8vIz3iBCK+ju4Gu4TPqV0CIHQtkwAMuiwVumvT
7FSch3+RKjA3Rt9CWAfawmFgH0FQAHUA2ra/az+1tiKfm8K7XGvocJFxbLtRhIU0
vaQ9MEjX+6sAAAGHuzuB7AAABAMARjBEAiAecnn4wh/hbLT0/be0ueJGRGK6ZA06
HG0ylDL5K6NgCwIgDvgnFB19naHWLMchaajAvCWV3xupAYeCEdnSpzX/2uYAdgDu
zdBk1dsazsVct520zROiModGfLzs3sNRSFlGcR+1mwAAAYe7O4H8AAAEAwBHMEUC
IQDqlr+hSMGYcxO1pMqgpYHXt+g8VkrF6ZF1IeGAaV1mIgIgLqn1vQoeuzLgWrZ2
BKnYtTes3DHqUg3iSN53/rGTsT4wDQYJKoZIhvcNAQEMBQADggGBALFCFyIGc7rZ
24kuAsU9fUkH/98RnVOkbmBaXqotGoSGm6iPE4sM0Ow88SDP2saeu41r5DKYp7eL
yY+3EuOpuFbaoG43xt2JxLXTFdypxRPFj48dVW31y64M2Jxi2kCjvExXrrRAoSTk
HMbPc4klOGWHvMglaKTdc7oiN1xF9gL8ZQB4Y6S8Dj4aQS7+WqvhQmkX9NKpdnxU
6rpA8xLloCamAiwB414ZIFbALFCW6CyQ6rgJoPBPw/9/+vdMX0fcbHjl6yk3jAbh
mk/VKFjOHLvr7hAUanOBWrroy2d0eCz7kO6AbWiSU0WBPfpFSXMGx3FFhY0ZaJn
bAaP1vgoRC9+cdiqVfDCdZLWBoZ7E85UQwYwCXN5UrXhc2Hapc1dmCz0jKwdeC7t
0AMepPLbswJ2jpMBsTMm6i83fd+qxGtWjaXmt4n+/1uq4CO5Dv4L1EGgaOQ3JOhO
zzRPXRBxo+NcUcN2u7e82wEuP5eRPxSnb2kaTGWYh+FxJ8IRfUoEqw==
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFBzCCA++gAwIBAgIRAIjIlPomNnQ0gajkW/aGHc4wDQYJKoZIhvcNAQELBQAw
ezELMAkGA1UEBhMCR0IxGzAZBgNVBAgMEkdyZWF0ZXIgTWFuY2hlc3RlcjEQMA4G
A1UEBwwHU2FsZm9yZDEaMBgGA1UECgwRQ29tb2RvIENBIExpbWl0ZWQxITAfBgNV
BAMMGEFBQSBDZXJ0aWZpY2F0ZSBTZXJ2aWNlczAeFw0yMjAxMTAwMDAwMDBaFw0y
ODEyMzEyMzU5NTlaMFkxCzAJBgNVBAYTAkNOMSUwIwYDVQQKExxUcnVzdEFzaWEg
VGVjaG5vbG9naWVzLCBJbmMuMSMwIQYDVQQDExpUcnVzdEFzaWEgUlNBIE9WIFRM
UyBDQSBHMjCCAaIwDQYJKoZIhvcNAQEBBQADggGPADCCAYoCggGBAMMsv9Ae9mba
03+pEjRkDcT5/psaxSnRb1ml1UkOo55ptSHNvD6ksa6h1ABuw/0n90bmj3aMJspG
1IQXkRBoVnB2BUmDGMUfscHFmOsnI0qJxKpwe6YAxyFJbzIjq8D7DShACNUKzypR
poBCoJKC3nSAqk+bkZ4qnmmd/FWl/K1EEcZHmJSFRkNm2V5Wiolithqeoulz9YCF
Uml/D4pwgheY5yp+SeiNF9lwM6eILUQ2Rdot/nx2vQ0WypVfrz0TgR/VBPtSln6Y
LVO2awsQWYW2QjKzUVmk1wCHiTGQ8s6MlV35UcmycMJhS+lFxe6qPduxWrtbMGwJ
PyzPBEPt2mItBkQGQmJ6suT2w092ask0cpotSR6zPFLpiawsE5k9KXl2yOCGAHH1
WtiDAqu1hyW/isrGDlVIYvXNrsDBngog97kl4gHQASDM1HuiG3j9d2Bao2BP3q4u
DI3GCJd468paeBDdkrQux+4tSO7eCoVbGnPFjUspEtMuMuNy5XD8JwIDAQABo4IB
JjCCASIwHwYDVR0jBBgwFoAUoBEKIz6W8Qfs4q8p74Klf9AwpLQwHQYDVR0OBBYE
FMoxITcM3jm30istVZpcqalcUPomMA4GA1UdDwEB/wQEAwIBhjASBgNVHRMBAf8E
CDAGAQH/AgEAMB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcDAjAiBgNVHSAE
GzAZMA0GCysGAQQBsjEBAgIxMAgGBmeBDAECAjBDBgNVHR8EPDA6MDigNqA0hjJo
dHRwOi8vY3JsLmNvbW9kb2NhLmNvbS9BQUFDZXJ0aWZpY2F0ZVNlcnZpY2VzLmNy
bDA0BggrBgEFBQcBAQQoMCYwJAYIKwYBBQUHMAGGGGh0dHA6Ly9vY3NwLmNvbW9k
b2NhLmNvbTANBgkqhkiG9w0BAQsFAAOCAQEAdI/xWnHciaMYplkXi+20Ec5OLR1h
UwczrHpDCvWAob2Hl+NKlIpwO3y3ALjCXvFTPAEkNI/RFbNoxz2UiFQh946m4ggB
n77Dj35xu6TWgCeYnhc2R87EDLYS3UHxfg0fZY8Pvo4wm+Bqabx23aFESYDlk6fO
/VgvUzZ/GewYo78DGPNd/dcqjMSR0Ga0Q7BD+SeFbmzmPD1oRfIPKog4kDum95Ro
nt7m3K8cEjhEWZ6h3ZQlWrdi6oGdxMixhnE4b8X9tIflmVSoIdzSbsrIcaOvlkzv
nc9IKm8XQvqOewdx5x3Gfd8pzGr5JjGQEBzDXtArfKt4pbBk2j5b9FAjCA==
-----END CERTIFICATE-----
```

- 如果继续使用旧的北京数据中心回执地址 `https://callback-bj.ronghub.com/push/callback/huawei`，您需要及时更新为最新的 HTTPS 证书（有效期至 2024-11-28）：

```
-----BEGIN CERTIFICATE-----
MIIHDjCCBXagAwIBAgIQfetRWzHz7gPOiYIly7MnNjANBgkqhkiG9w0BAQwFADBZ
MQswCQYDVQQGEwJDTjElMCMGA1UEChMcVHJ1c3RBc2lhIFRlY2hub2xvZ2llcywg
SW5jLjEjMCEGA1UEAxMaVHJ1c3RBc2lhIFJTQSBPViBUTFMgQ0EgRzIwHhcNMjMx
MDMwMDAwMDAwWhcNMjQxMTI3MjM1OTU5WjBuMQswCQYDVQQGEwJDTjESMBAGA1UE
CAwJ5YyX5Lqs5biCMTMwMQYDVQQKDCrljJfkuqzkupHkuK3ono3kv6HnvZHnu5zn
p5HmioDmnInpmZDlhazlj7gxFjAUBgNVBAMMDSoucm9uZ2h1Yi5jb20wggEiMA0G
CSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDDWDfF76DKkXyv2PDYdXiULJHEDYGe
shac538yjCW3ObKD1B5H6A+OqbwocUimki1HQziPxP7LLVbYsW/yoCiCIF8vCooT
Mz6S+e7OitZwAgAYiqBjCjDYDqYsGh+IpX1bgXttL8/ggMUc9+Uirk3fEtv5KXUZ
1bxg28krAgZwq/VyCPBYM6Sy/GLM5v7McDDUjiVzbh8M11/frJ2UeGPf5t0xlfUG
aAn8MG9haIWawpmHtGjuU8FOFgGZjh9xM5yjXydOgOH/j+k1BgT4E5qtuDQNpn0D
jNlB9QYgFHWH0Z5kSXjTrcAenLL50xFM5khwnN4bC3PKjlA9E47L87lJAgMBAAGj
ggM7MIIDNzAfBgNVHSMEGDAWgBTKMSE3DN45t9IrLVWaXKmpXFD6JjAdBgNVHQ4E
FgQUPmfjcxjPKk3d8nMrwumxBoDKgqYwDgYDVR0PAQH/BAQDAgWgMAwGA1UdEwEB
/wQCMAAwHQYDVR0lBBYwFAYIKwYBBQUHAwEGCCsGAQUFBwMCMEkGA1UdIARCMEAw
NAYLKwYBBAGyMQECAjEwJTAjBggrBgEFBQcCARYXaHR0cHM6Ly9zZWN0aWdvLmNv
bS9DUFMwCAYGZ4EMAQICMEcGA1UdHwRAMD4wPKA6oDiGNmh0dHA6Ly9jcmwudHJ1
c3QtcHJvdmlkZXIuY24vVHJ1c3RBc2lhUlNBT1ZUTFNDQUcyLmNybDB9BggrBgEF
BQcBAQRxMG8wQgYIKwYBBQUHMAKGNmh0dHA6Ly9jcnQudHJ1c3QtcHJvdmlkZXIu
Y24vVHJ1c3RBc2lhUlNBT1ZUTFNDQUcyLmNydDApBggrBgEFBQcwAYYdaHR0cDov
L29jc3AudHJ1c3QtcHJvdmlkZXIuY24wJQYDVR0RBB4wHIINKi5yb25naHViLmNv
bYILcm9uZ2h1Yi5jb20wggF8BgorBgEEAdZ5AgQCBIIBbASCAWgBZgB1AHb/iD8K
tvuVUcJhzPWHujS0pM27KdxoQgqf5mdMWjp0AAABi35X8rgAAAQDAEYwRAIgNKTK
ek69iQ9myOQlOI6I3pFdtG8WOF/j2oXze+rm4hICIDeEvwtw6NV3yo7n1Wd7mmM8
4n3d3xm/8JSCso444R3cAHUA2ra/az+ltiKfm8K7XGvocJFxbLtRhIU0vaQ9MEjX
+6sAAAGLflfzFAAABAMARjBEAiAMsNftxXkgy/o+R4WZWLuRI6t/LBwaSMglR1Xz
VaOCNwIgWrnysdL7Tp2E4dX+ADbNmUziJ3JbQV9KX+e1WUWbCN0AdgDuzdBk1dsa
zsVct520zROiModGfLzs3sNRSFlGcR+1mwAAAYt+V/MZAAAEAwBHMEUCIBY4qYAp
5cSNPuIBwHMWXpOOarYiebtZFu+zOtl6puQtAiEA1ahkA+YsAyjoJ0mHjm2rNJPu
NC/5cQbTrf4AwviYTNIwDQYJKoZIhvcNAQEMBQADggGBAANNMqwddxkG7Soa0ryu
X16rnZ6GXk0pV16rB8MzNs01bgFpxc7PI0c6l8L3IOKqN3cl/3PGWuHgfZs/zmPX
LSgYYJFyMZiJlN1kxlLpFQ5EC16ihl9HKmYrFRNb8ROI0fjjXVafZ4qfA3jVuWAE
JiV09cXU2DkkAy40I9JS/rGCo/iBxRM8JYbCEA7JPj/c73jjHbBzCmUXOM7LlxWM
/Rvz+rvGYn+O8qPAPNBlUaSySsdmDLezJc2hbWb11PJkge8pIqrrMAamEiokpBOD
EaMmslX+Cw9pg3L/UbmPg4yJdEx4aKgfTvz9RcJKTYhDKhgK3wNjYq9acbpmpTeG
raC6jagqmlNtAf26mshWTrp9OuekkVahnAhaujwIQ214e2naWJWIDgKnlZ6LWpWw
d2Y4+FKqFrSDyUKs7c6I0QY2k7PlwLIlMCN82pn2Ssbchy1KjCyGagtNpcVfTsJr
sAu+ws7kGsTjFjyEfihH0PlBJwXvj9bks2nubR4waGwpBQ==
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFBzCCA++gAwIBAgIRAIjIlPomNnQ0gajkW/aGHc4wDQYJKoZIhvcNAQELBQAw
ezELMAkGA1UEBhMCR0IxGzAZBgNVBAgMEkdyZWF0ZXIgTWFuY2hlc3RlcjEQMA4G
A1UEBwwHU2FsZm9yZDEaMBgGA1UECgwRQ29tb2RvIENBIExpbWl0ZWQxITAfBgNV
BAMMGEFBQSBDZXJ0aWZpY2F0ZSBTZXJ2aWNlczAeFw0yMjAxMTAwMDAwMDBaFw0y
ODEyMzEyMzU5NTlaMFkxCzAJBgNVBAYTAkNOMSUwIwYDVQQKExxUcnVzdEFzaWEg
VGVjaG5vbG9naWVzLCBJbmMuMSMwIQYDVQQDExpUcnVzdEFzaWEgUlNBIE9WIFRM
UyBDQSBHMjCCAaIwDQYJKoZIhvcNAQEBBQADggGPADCCAYoCggGBAMMsv9Ae9mba
03+pEjRkDcT5/psaxSnRb1ml1UkOo55ptSHNvD6ksa6h1ABuw/0n90bmj3aMJspG
1IQXkRBoVnB2BUmDGMUfscHFmOsnI0qJxKpwe6YAxyFJbzIjq8D7DShACNUKzypR
poBCoJKC3nSAqk+bkZ4qnmmd/FWl/K1EEcZHmJSFRkNm2V5Wiolithqeoulz9YCF
Uml/D4pwgheY5yp+SeiNF9lwM6eILUQ2Rdot/nx2vQ0WypVfrz0TgR/VBPtSln6Y
LVO2awsQWYW2QjKzUVmk1wCHiTGQ8s6MlV35UcmycMJhS+lFxe6qPduxWrtbMGwJ
PyzPBEPt2mItBkQGQmJ6suT2w092ask0cpotSR6zPFLpiawsE5k9KXl2yOCGAHH1
WtiDAqu1hyW/isrGDlVIYvXNrsDBngog97kl4gHQASDM1HuiG3j9d2Bao2BP3q4u
DI3GCJd468paeBDdkrQux+4tSO7eCoVbGnPFjUspEtMuMuNy5XD8JwIDAQABo4IB
JjCCASIwHwYDVR0jBBgwFoAUoBEKIz6W8Qfs4q8p74Klf9AwpLQwHQYDVR0OBBYE
FMoxITcM3jm30istVZpcqalcUPomMA4GA1UdDwEB/wQEAwIBhjASBgNVHRMBAf8E
CDAGAQH/AgEAMB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcDAjAiBgNVHSAE
GzAZMA0GCysGAQQBsjEBAgIxMAgGBmeBDAECAjBDBgNVHR8EPDA6MDigNqA0hjJo
dHRwOi8vY3JsLmNvbW9kb2NhLmNvbS9BQUFDZXJ0aWZpY2F0ZVNlcnZpY2VzLmNy
bDA0BggrBgEFBQcBAQQoMCYwJAYIKwYBBQUHMAGGGGh0dHA6Ly9vY3NwLmNvbW9k
b2NhLmNvbTANBgkqhkiG9w0BAQsFAAOCAQEAdI/xWnHciaMYplkXi+20Ec5OLR1h
UwczrHpDCvWAob2Hl+NKlIpwO3y3ALjCXvFTPAEkNI/RFbNoxz2UiFQh946m4ggB
n77Dj35xu6TWgCeYnhc2R87EDLYS3UHxfg0fZY8Pvo4wm+Bqabx23aFESYDlk6fO
/VgvUzZ/GewYo78DGPNd/dcqjMSR0Ga0Q7BD+SeFbmzmPD1oRfIPKog4kDum95Ro
nt7m3K8cEjhEWZ6h3ZQlWrdi6oGdxMixhnE4b8X9tIflmVSoIdzSbsrIcaOvlkzv
nc9IKm8XQvqOewdx5x3Gfd8pzGr5JjGQEBzDXtArfKt4pbBk2j5b9FAjCA==
-----END CERTIFICATE-----
```

自定义 token 优先级上报 自定义 token 优先级上报

更新时间:2024-08-30

手机接收到多个推送 token 时（如：FCM + 华为），开发者可以配置推送优先级，融云服务端会选择优先级最高的通道下发推送通知。此功能仅适用于 5.8.0 之后的版本。

优先级设置

1. 开发者通过 `setPushEventListener` 接口来调整推送的优先级，SDK 会根据调整后的顺序上报。SDK 在准备上报时推送 token 时回调 `onStartTokenReport` 方法。

```
RongPushClient.setPushEventListener(new PushEventListener() {
/**

*
* @param tokenList SDK 默认上报顺序，注册优先级，优先级高。
* @return 返回优先级列表。
*/
@Override
public List<TokenBean> onStartTokenReport(List<TokenBean> tokenList) {
//todo token 的注册顺序，index 靠前的优先级高。
return tokenList;
}

/**
* token 上报回调
*
* @param reportType 上报类型，兼容之前接口，默认为 Null。
* @param code 服务器状态码。
* @param finalType 最终上报成功的优先级最高的 token 对应的推送类型。
* @param finalToken 最终上报成功的优先级最高的 token。
* @param message 服务器返回的信息。
* @param failMap 服务器注册失败的推送类型。
*/
@Override
public void onTokenReportResult(PushType reportType, int code, PushType finalType, String finalToken, String message, Map<String, String> failMap) {
PushEventListener.super.onTokenReportResult(reportType, code, finalType, finalToken, message, failMap);
}
});
```

2. 上报 token 成功后，可以在 `onTokenReportResult` 接口查看最终结果。

推送集成概述

更新时间:2024-08-30

融云 SDK 支持集成三方推送通道与融云自建推送通道（RongPush）。

> 提示
> 本文适用于 **5.6.0** 版本之前的 IMLib、IMKit 或其他依赖 IMLib 的融云客户端 SDK。从 5.6.0 版本开始，融云推出集成方式更简单的 Push 2.0 集成方案，欢迎使用。

推送服务能力

推送支持「离线消息推送」和「不落地通知」两种场景。

离线消息推送通知

假设用户仅在一台设备上登录，如果主动断开连接（disconnect()）或者应用程序已被用户或系统杀死，融云会认为用户在该客户端离线。用户离线状态下，支持将收到的单聊消息、群聊消息、系统消息、超级群消息通过第三方推送厂商或融云自建的推送服务通知客户端。

- 如果由第三方厂商推送服务发送提醒，该提醒一般由系统直接弹出，以通知形式展示在通知面板，提示用户收到消息。
- 如果由融云自建推送通道（RongPush）发送提醒，该提醒一般由 SDK 调用系统 API 构建通知后弹出。注意，RongPush 在国内大部分机型上无法存活。建议应用程序集成第三方厂商的推送服务。

用户点击推送通知后再次与融云服务端建立 IM 连接后，SDK 会有如下行为：

- 自动收到离线期间的单聊、群聊离线消息[7]。服务端默认缓存 7 天未收取的离线消息。
- 自动收到离线期间超级群会话中最后一条消息，应用程序需要自行拉取离线期间的历史消息。

> 提示
> 应用程序处于后台且活跃时，用户仍处于在线状态，SDK 仍可实时收到会话消息，消息送达过程中不会使用任何推送服务，因此用户设备不会收到来自任何推送服务的通知。如果使用 IMLib，应用程序需要自行调用系统 API 创建并弹出本地通知。如果使用 IMKit，SDK 默认会调用系统 API 创建并弹出本地通知。

不落地通知

融云支持直接通过服务端 API 向客户端发送远程推送通知，称为不落地通知[7]。不落地通知中不包含任何会话消息，无论客户端 App 是否在前台，所有通知内容始终仅会以通知形式展示在系统通知栏中，用户无法在任何聊天会话中看到不落地通知的内容。

不落地通知始终通过推送通道下发数据，因此依赖应用程序集成第三方厂商推送服务，或者在客户端启用 RongPush。

- 如果由第三方厂商推送服务发送提醒，该提醒一般由系统直接弹出，以通知形式展示在通知面板，提示用户收到消息。
- 如果由融云自建推送通道（RongPush）发送提醒，该提醒一般由 SDK 调用系统 API 构建通知后弹出。注意，RongPush 在国内大部分机型上无法存活。建议应用程序集成第三方厂商的推送服务。

不落地通知仅支持通过服务端 API 发送，例如：

- 单个用户不落地通知
- 全量用户不落地通知

目前不支持通过控制台发送不落地通知（仅部分旧账号仍保留该能力）。

无法推送以及推送受限的情况

- 因聊天室业务设计特点，仅当聊天室中的用户在线时才会收到聊天室会话中的消息，因此**聊天室消息不支持离线消息推送**。
- 客户端调用了 `logout` 方法，或在 `disconnect` 时设置了不允许推送，或通过设置 SDK 的初始化配置中的 `enablePush` 参数为 `false`，向融云服务申请禁用推送服务（当前设备），导致彻底注销用户在融云服务端的登录信息。这种情况下，用户无法通过任何推送通道收到通知。
- 即使用户的所有移动端设备均已离线，只要用户仍在 Web/PC 端在线，此时融云认为用户在线，默认不会给移动端发送推送通知。如有需要，您可以在控制台免费基础功能页面调整 **Web/PC 在线手机端接收 Push** 开关设置。
- 即使用户的移动端应收到推送通知，融云服务端不会向所有已登录过移动端设备均发送推送，仅会向最后一个登录的设备发送推送通知。
- 已触发第三方厂商推送服务的频率、数量限制。为改善终端用户推送体验，部分第三方推送服务（例如华为、vivo）已对推送消息的分类进行数量和频率管控。建议您充分了解第三方的管理细则。
- 因超级群业务中普通消息的数量较大，为控制离线推送频率，默认每分钟针对单个用户的单个超级群，每个频道最多产生 1 条推送。默认普通消息累计 2 条时才会触发推送。@ 消息不受此限制。如需调整，详见开通超级群服务。

> 提示
> 用户必须至少在设备上连接成功过一次，该设备才能接收推送。

注册推送广播接收组件

在主工程的 `AndroidManifest.xml` 中注册 `PushMessageReceiver`，用于接收推送相关的事件广播。

```
<receiver
android:name="io.rong.push.notification.PushMessageReceiver"
android:exported="true">
<intent-filter>
<action android:name="io.rong.push.intent.MESSAGE_ARRIVED" />
<action android:name="io.rong.push.intent.MESSAGE_CLICKED" />
<action android:name="io.rong.push.intent.THIRD_PARTY_PUSH_STATE" />
</intent-filter>
</receiver>
```

集成第三方推送

融云支持第三方推送通道。由于国内手机厂商的限制，融云自建推送通道（RongPush）在国内大部分机型上无法存活。您可以优先选择集成第三方推送。第三方推送通道指各手机厂商从系统层维护的长连接通道，您可以与融云提供的推送服务结合使用。

目前融云已适配了小米、华为、荣耀、魅族、OPPO、vivo、FCM 推送服务。

- 集成小米推送
- 集成华为推送
- 集成荣耀推送
- 集成魅族推送
- 集成 OPPO 推送
- 集成 vivo 推送
- 集成 FCM 推送

在控制台配置 ApplicationId

融云服务端在向第三方推送通道发送推送数据时，需要使用 App 的应用标识（Android 应用 ID）。您需要在控制台进行配置。

1. 访问控制台应用标识页面（也可从**服务管理**页面左侧 **IM 服务**下访问）。
   1. 如您有多个融云应用，请确保在页面顶部**应用**一栏切换到正确的应用。
   2. 新建的应用默认拥有一个应用标识，您可以创建更多应用表示，最多 5 个。每个应用标识均可设置推送，应用标识之间不共享推送配置。
2. 在应用标识旁，点击**设置推送**，并在 **Android** > **ApplicationId** 一栏填写您的 Android 应用 ID。

   每个 Android 应用均有一个唯一的应用 ID（applicationId），像 Java 软件包名称一样，例如 com.example.myapp。此 ID 可以作为您的应用在设备上和 Google Play 商店中的唯一标识。

   应用 ID 由模块的 build.gradle 文件中的 applicationId 属性定义，如下所示：

```
android {
defaultConfig {
applicationId "com.example.myapp"
minSdkVersion 21
targetSdkVersion 24
versionCode 1
versionName "1.0"
}
...
}
```

   如果您没有在 build.gradle 中配置 applicationId，则 applicationId 默认为应用的包名。如果您有更多关于应用 ID 与包名的疑问，可参见 Android 官方文档。

集成融云自建推送通道

融云自建推送通道（RongPush）是融云客户端 SDK 与融云推送服务之间维护的一条稳定可靠的长连接通道。属于 SDK 默认推送，不需要额外集成其它三方库即可拥有的基础推送能力。

SDK 初始化启用推送功能后，则自动启用融云自建推送通道（RongPush），App 即具备了基本推送能力，由于国内手机厂商的限制，RongPush 在国内大部分机型上无法存活。建议同时集成第三方推送通道。

从 SDK 5.3.4 版本开始，支持禁用 RongPush，您需要在初始化融云 SDK 之前进行传入推送配置：

```
PushConfig config = new PushConfig.Builder()
.enableRongPush(false) // 禁用融云自建推送通道
.XXX //此处忽略其它三方推送配置
.build();
RongPushClient.setPushConfig(config); //将推送相关配置设置到 SDK
```

注意，用户必须至少在设备上连接成功过一次，该设备才能接收推送。

客户端权限配置

由于融云默认推送通道属于应用级别的推送，会受系统各种权限限制，建议提示用户打开对应权限，以提高推送到达率。

推送通道选择策略

在应用配置了多通道的情况下，为了提高用户体验，提高推送到达率，融云客户端 SDK 会根据应用配置，智能选择最优推送通道。

详细启用策略如下：

推送通道	配置要求	ROM 要求
华为	应用配置了华为推送	当前设备为华为 ROM
荣耀	应用配置了荣耀推送	当前设备为荣耀 ROM，且为 2023 年 11 月 30 日后发售的设备。
OPPO	应用配置了 OPPO 推送	当前设备为以下 ROM 中的一种： OPPO、realme、OnePlus
vivo	应用配置了 vivo 推送	当前设备为 vivo ROM
魅族	应用配置了魅族推送	当前设备为魅族 ROM
FCM	应用配置了 FCM 推送	客户端出访 IP 在国外
RongPush	默认推送通道	不满足其它通道启用策略，且 RongPush 可用时，默认使用 RongPush 推送通道

提示

关于 FCM 的补充说明

系统原生推送和 FCM 同时配置时，具体使用哪种推送通道，取决于应用层的配置顺序。比如在小米手机上，同时配置了小米推送和 FCM 推送，该用户在国外时：

- 如果应用层配置时的顺序是小米推送在 FCM 之前，则会启用小米推送。
- 如果应用层配置时的顺序是 FCM 在小米推送之前，则会启用 FCM 推送。

集成小米推送　　集成小米推送

更新时间:2024-08-30

按照本指南集成小米 Mi Push 国内版或海外版 ，让融云 SDK 支持小米推送。

在集成第三方推送前，请确保已在控制台配置 **Android 应用 ID**。详见推送集成概述。

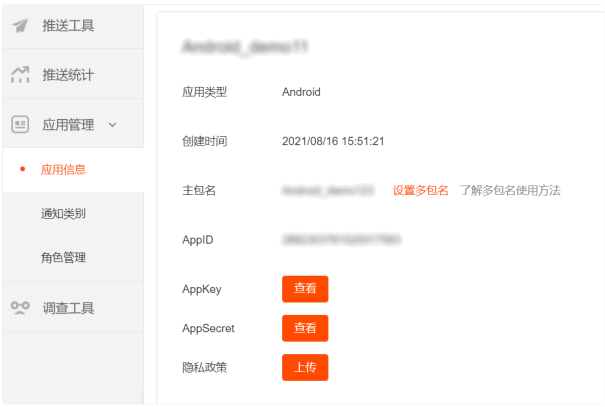
> 提示
>
> IMLib SDK 从 5.6.8 开始支持小米海外推送服务。

在控制台配置小米推送

如果想通过小米推送通道从融云服务端接收推送通知，您需要在控制台上提供您的小米推送应用的详细信息。

1. 前往小米开放平台，选择您当前的项目所对应的小米应用，点击**应用信息**，并记录下应用的 **AppID**、**AppKey**、**AppSecret**。

> **注意：**
>
> **如果没有小米开发者账号，或尚未创建应用，参考小米推送文档：**
>
> - **中国大陆地区 - 推送服务启用指南**
> - **海外版 - Push Service Activation Guide 。目前，小米在印度孟买、德国法兰克福、俄罗斯莫斯科和新加坡设有数据中心，请选择合适的地域创建应用。**



其中 AppSecret 是小米推送服务器端的身份标识，在使用小米推送服务端 SDK 向客户端发送消息时使用，需要在控制台的小米推送配置中提供给融云。AppId 和 AppKey 是小米推送客户端的身份标识，后续在启用小米推送服务时需要提供给融云 SDK，用于初始化小米推送客户端 SDK。

2. 打开控制台，前往在应用标识页面。点击**设置推送**，找到 **Android > 小米推送**，填入上一步获取的 **AppSecret**。

3. （可选）配置**小米推送通知标题**。设置默认的推送通知标题。一般情况下客户端发送消息转 Push 时不使用此标题设置。在调用融云服务端 API `/push.json`、`/push/user.json`、`/push/custom.json` 接口推送通知时，如未传入通知标题，则使用该处设置的标题。从服务端发消息时，如果发送用户 ID 在融云服务端没有用户名，也会使用此 “推送通知

标题"。

4. 选择推送通道类型。小米推送的消息类型，分为普通消息和通知消息，查看小米推送消息限制明文档。

- 普通消息，融云默认使用的小米推送通道，有限制。
- 通知消息，无限制。需要填写小米后台申请的 channelId。

5. 保存设置。所有设置 30 分钟后生效。

您已完成需要控制台小米推送配置的全部内容。现在可以设置客户端集成。

配置客户端接收小米推送

1. 首先，需要将小米推送客户端 SDK 添加到您的 Android 项目。您需要手动下载小米推送客户端 SDK 并加到项目中。

- 融云官网 SDK 下载页提供小米推送客户端 SDK 下载。在页面勾选第三方推送后，下载 zip 文件。在解压后的文件目录中找到 pushlibs 下的小米推送客户端 SDK 文件（MiPush_SDK_Client_xxx.jar，4.9.1）。拷贝 .jar 文件到 app 的 libs 目录下。

- 如需获取最新版小米推送客户端 SDK，可以从小米直接获取。MiPush Android 客户端 SDK 从 5.0.1 版本开始，提供 AAR 包接入方式。
  - 参考小米文档 Android 客户端 SDK 集成指南（JAR版）
  - 参考小米文档 Android 客户端 SDK 集成指南（AAR版）（中国大陆）
  - 参考小米文档 Android 客户端 SDK 集成指南（AAR版）（非中国大陆）

> 提示
> - 如果您项目使用的 IMLib/IMKit SDK 版本大于等于 5.2.1，必须使用小米推送客户端 SDK 4.9.1 及以上版本。
> - 由于 Android 12 通知 trampoline 限制，对于 targetSdkVersion >= 31 的应用，需要接入SDK 4.9.1及以上版本，否则会出现点击通知无法正常跳转的情况。
> - 小米推送客户端 SDK 区分中国大陆地区与海外地区，请从对应小米官方文档中下载正确的版本。

2. 在 App 的 build.gradle 中添加依赖：

- 如果集成 jar 文件：

```
dependencies {
implementation files('libs/MiPush_SDK_Client_xxx.jar')
}
```

- 如果集成 aar 文件：

```
dependencies {
implementation (name: 'MiPush_SDK_Client_xxx', ext: 'aar')
}
```

处理小米推送的权限、服务与广播接收器（JAR版）

如果使用小米 Android 客户端 SDK 集成指南（JAR版），需要完成以下配置。AAR 版无需完成以下配置。

在主工程的 AndroidManifest.xml 中增加如下权限配置：

```
<permission
android:name="${`applicationId`}.permission.MIPUSH_RECEIVE"
android:protectionLevel="signature" />
<uses-permission android:name="${`applicationId`}.permission.MIPUSH_RECEIVE" />
```

如果应用 targetSdkVersion >= 31（**Android 12**），且接入的小米推送客户端 SDK 版本为 4.9.1 及以上，则需配置以下 Activity ，否则会出现点击通知 activity 无法启动的情况。

```
<activity
android:name="com.xiaomi.mipush.sdk.NotificationClickedActivity"
android:theme="@android:style/Theme.Translucent.NoTitleBar"
android:launchMode="singleInstance"
android:exported="true"
android:enabled="true">
</activity>
```

在主工程的 AndroidManifest.xml 中添加小米推送配置需要的 service 和 receiver：

```
<service android:name="com.xiaomi.push.service.XMPushService" android:enabled="true" />
<service android:name="com.xiaomi.mipush.sdk.PushMessageHandler" android:enabled="true" android:exported="true" />
<service android:name="com.xiaomi.mipush.sdk.MessageHandleService" android:enabled="true" />

<service
android:name="com.xiaomi.push.service.XMJobService"
android:enabled="true"
android:exported="false"
android:permission="android.permission.BIND_JOB_SERVICE" />

<receiver android:name="com.xiaomi.push.service.receivers.PingReceiver" android:exported="false">
<intent-filter>
<action android:name="com.xiaomi.push.PING_TIMER" />
</intent-filter>
</receiver>
```

以上内容来自小米。如需了解更多细节，可参考小米推送客户端文档 Android 客户端 SDK 集成指南（JAR版）。

使用融云 SDK 提供的广播接收器

为了接收小米推送消息，必须实现一个继承自小米 PushMessageReceiver 类的广播接收器（Broadcast Receiver）。融云 SDK 已经实现了该广播接收器，您只需要将融云提供的 MiMessageReceiver 注册到AndroidManifest.xml 文件中，注册内容如下：

```
<receiver android:name="io.rong.push.platform.mi.MiMessageReceiver" android:exported="true">
<intent-filter>
<action android:name="com.xiaomi.mipush.RECEIVE_MESSAGE" />
</intent-filter>
<intent-filter>
<action android:name="com.xiaomi.mipush.MESSAGE_ARRIVED" />
</intent-filter>
<intent-filter>
<action android:name="com.xiaomi.mipush.ERROR" />
</intent-filter>
</receiver>
```

使用应用自定义的广播接收器

如果您的应用因自身业务需求已集成小米推送客户端，您将需要用 App 自定义的 Broadcast Receiver 替换融云 SDK 提供的 MiMessageReceiver。否则两者会发生冲突。

请如下步骤进行处理：

1. 在 AndroidManifest.xml 中，移除融云 SDK 实现的小米广播接收器（MiMessageReceiver）：

```
<receiver android:name="io.rong.push.platform.mi.MiMessageReceiver" android:exported="true">
<intent-filter>
<action android:name="com.xiaomi.mipush.RECEIVE_MESSAGE" />
</intent-filter>
<intent-filter>
<action android:name="com.xiaomi.mipush.MESSAGE_ARRIVED" />
</intent-filter>
<intent-filter>
<action android:name="com.xiaomi.mipush.ERROR" />
</intent-filter>
</receiver>
```

2. 在应用自行实现的广播接收器中，增加针对融云业务的推送处理代码。以下示例中 MyMiPushMessageReceiver 为应用实现的继承自小米 PushMessageReceiver 的广播接收器。

```
/**
* 该类为应用实现的继承自小米 PushMessageReceiver 的广播接收器。
**/
public class MyMiPushMessageReceiver extends PushMessageReceiver {
...
@Override
public void onNotificationMessageClicked(Context context, MiPushMessage message) {
if(isRongPush()) { //此处为伪代码，应用需要根据业务自己实现。
//增加如下代码，处理融云业务对应推送。
PushNotificationMessage pushNotificationMessage = PushUtils.transformToPushMessage(message.getContent());
PushManager.getInstance().onNotificationMessageClicked(context, PushType.XIAOMI, pushNotificationMessage);
} else {
... //应用自身推送业务处理
}
}

@Override
public void onNotificationMessageArrived(Context context, MiPushMessage message) {
if(isRongPush()) { //此处为伪代码，应用需要根据业务自己实现。
PushNotificationMessage pushNotificationMessage = PushUtils.transformToPushMessage(message.getContent());
if (pushNotificationMessage != null) {
PushManager.getInstance().onNotificationMessageArrived(context, PushType.XIAOMI, pushNotificationMessage);
}
} else {
... //应用自身推送业务处理
}
}

@Override
public void onReceiveRegisterResult(Context context, MiPushCommandMessage message) {
String command = message.getCommand();
List<String> arguments = message.getCommandArguments();
String cmdArg1 = ((arguments != null && arguments.size() > 0) ? arguments.get(0) : null);
String cmdArg2 = ((arguments != null && arguments.size() > 1) ? arguments.get(1) : null);
if (MiPushClient.COMMAND_REGISTER.equals(command)) {
if (message.getResultCode() == ErrorCode.SUCCESS) {
if(isRongPush()){ //如果是融云业务，调用如下代码处理
PushManager.getInstance().onReceiveToken(context, PushType.XIAOMI, cmdArg1);
} else {
... //应用自身推送业务处理
}
} else {
if(isRongPush()){ //如果是融云业务，调用如下代码处理
PushManager.getInstance().onErrorResponse(context, PushType.XIAOMI, PushConst.PUSH_ACTION_REQUEST_TOKEN, message.getResultCode());
} else {
... //应用自身推送业务处理
}
}
}
}
}
```

混淆配置

如果您的应用使用了混淆，您可以使用下面的代码混淆配置：

```
-dontwarn com.xiaomi.mipush.sdk.**
-keep public class com.xiaomi.mipush.sdk.* {*; }
```

启用小米推送服务

请在初始化融云 SDK 之前，将小米的 **AppID**、**AppKey** 添加到 PushConfig，并提供给 SDK。融云 SDK 将向小米推送服务注册、并将获取的小米推送 Token 上报给融云服务端。

PushConfig 为所有推送配置相关的入口类。

```
PushConfig config = new PushConfig.Builder()
.enableMiPush("小米 appId", "小米 appKey") //小米推送配置
.XXX //此处忽略其它三方推送配置
.build();
RongPushClient.setPushConfig(config); //将推送相关配置设置到 SDK
```

如果您使用小米海外推送服务，还需要调用小米推送客户端的 API，报告小米推送服务的地域。

```
MiPushClient.setRegion(region); // Global, Europe, Russia, India
```

自定义铃声设置

> ① 提示
>
> Android 8.0 以上手机需要设置 Channel 后，才能自定义推送铃声。

创建 Channel ID

配置自定义铃声前，请先参照小米文档创建 Channel ID：

- 使用小米开放平台上创建 Channel ID
- 使用小米服务端 API 创建 Channel ID
- 使用小米推送服务端 SDK 创建 Channel ID（此处仅提供小米 Push Server Java SDK 文档链接，其他语言请参照小米推送官方文档。）

配置自定义推送铃声

Channel ID 创建成功后，可以前往控制台进行配置。设置成功后，所有小米手机用户离线状态下接收的该类型消息，都会使用设置的 Channel ID 进行推送，提醒铃声为注册 Channel 时设置的自定义铃声。

1. 前往控制台 > 自定义推送铃声 页面。
2. 切换至**小米**，点击**添加**。
3. 选择 **ApplicationId**，为**消息类型**指定对应的 Channel ID 和自定义推送铃声文件路径。

设置成功后，所有小米手机用户离线状态下接收的该类型消息，都会使用设置的 Channel ID 进行推送，提醒铃声为注册 Channel 时设置的自定义铃声。

> 提示
>
> 音频文件需要打包到应用中，存放在应用 raw 目录下，存储的声音文件需要有扩展名，但是不要把扩展名写在 uri 中。当小米手机接收音视频推送消息时，自动读取设置的文件进行声音提醒。

集成华为推送　　集成华为推送

更新时间:2024-08-30

按照本指南集成 华为推送服务（Push Kit），让融云 SDK 支持从华为推送服务获取推送通知。

在集成第三方推送前，请确保已在控制台配置 **Android 应用 ID**。详见推送集成概述。

在控制台配置华为推送

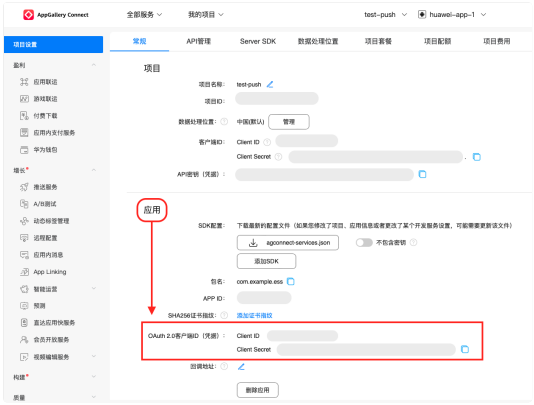
如果想通过华为推送接收融云即时通讯服务的推送通知，您需要在控制台上提供您的华为推送应用的详细信息。

1. 前往华为 AppGallery Connect 网站，点击**我的项目**，在项目列表中找到您的项目，上方导航栏选择需要查看信息的应用。

   如下图所示，test-push 是项目名称，huawei-app-1 是已关联到该项目的应用。

> 提示
>
> 如果没有华为开发者账号，或尚未创建项目和应用，请先创建账号、项目和应用。详见华为开发者文档创建账号、创建应用。请确保项目下已关联了应用，启用了推送服务，并配置签名证书指纹。如您对华为控制台上配置推送服务的流程有任何疑问，可参见华为官方开发者文档配置 AppGallery Connect。

   您需要记录下应用下的 **Client ID**（同 App ID） 和 **Client Secret**。



2. 打开控制台，在应用标识页面点击**设置推送**，找到 **Android** > **华为推送**，填入上一步获取的 **Client ID**、**Client Secret**。

3. （可选）配置自定义点击消息动作的 **Intent**。详见华为官方开发者文档自定义点击消息动作。

- 注意 intent 的格式必须要以 `end` 结尾。
- 自定义 intent 后，需按照定义 intent 在 `AndroidManifest.xml` 的 Activity 中配置如下 `intent-filter`。

4. （可选）配置推送角标数。详见华为官方开发者文档桌面角标。

- **badgeAddNum**：应用角标累加数字非应用角标实际显示数字，为大于0小于100的整数。例如，某应用当前有N条未读消息，若 add_num 设置为3，则每发一次消息，应用角标显示的数字累加3，为 N+3。

- **Activity**：应用入口 Activity 类全路径。样例：com.example.hmstest.MainActivity

5. （可选）配置默认的华为推送通道的消息自分类标识，例如 IM。App 根据华为要求完成自分类权益申请 或 申请特殊权限 后配置字段有效。详见华为推送官方文档消息分类标准 。配置成功后，当前包名接收的华为推送通知默认均会携带该字段。注意，如果客户端或服务端发送消息时配置了华为推送 Category，则使用发消息时指定的配置。

6. 配置**推送通知标题**。设置默认的推送通知标题。一般情况下客户端发送消息转 Push 时不使用此标题设置。在调用融云服务端 API `/push.json`、`/push/user.json`、`/push/custom.json` 接口推送通知时，如未传入通知标题，则使用该处设置的标题。从服务端发消息时，如果发送用户 ID 在融云服务端没有用户名，也会使用此“推送通知标题”。

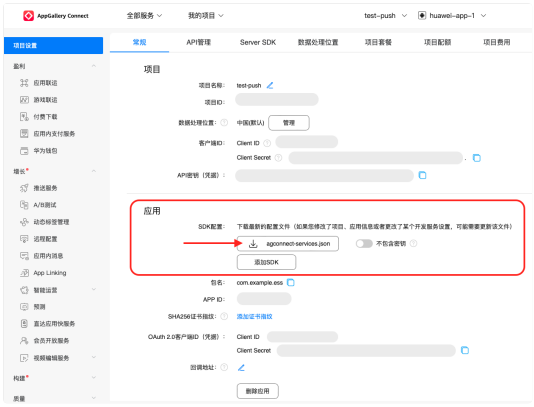
7. 保存设置。所有设置 30 分钟后生效。

您已完成需要控制台华为推送配置的全部内容。现在可以设置客户端集成。

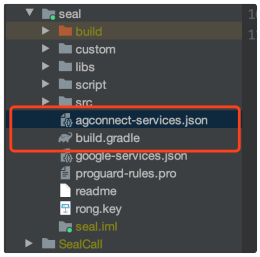
配置客户端接收华为推送

首先，需要将华为推送客户端 SDK 添加到您的 Android 项目。

根据华为开发者文档集成 HMS Core SDK ，您需要将“agconnect-services.json”文件添加到您的 App 中。点击 **agconnect-services.json** 下载配置文件。



将下载好的 `agconnect-servics.json` 文件放到 app 模块下的根目录，如下图：



导入华为推送 SDK

华为推送客户端 SDK 需要从华为 Maven 仓库获取。Android Studio的代码库配置在 Gradle 插件 7.0 以下版本、7.0 版本和 7.1 及以上版本有所不同。详见华为官方开发者文档集成 HMS Core SDK 中的「配置 HMS Core SDK 的 Maven 仓库地址」。

本步骤中以 Gradle 插件 7.0 以下版本为例。打开在 project 的 `build.gradle` 中添加如下内容。

```
allprojects {
repositories {
//Add Huawei Maven
maven {url 'http://developer.huawei.com/repo/'}
}
}

buildscript{
repositories {
//Add Huawei Maven
maven { url 'http://developer.huawei.com/repo/' }
}
dependencies {
// Add this line
classpath 'com.huawei.agconnect:agcp:1.6.1.300'
}
}
```

> 提示
>
> “buildscript > dependencies” 下需要添加 AGC 插件配置，请您参见华为官方开发者文档 AGC 插件依赖关系 选择合适的 AGC 插件版本。

添加华为 Maven 仓库后，您可以在 App 的 `build.gradle` 中添加依赖，直接引入华为推送客户端 SDK。建议您集成的SDK使用最新版本号，版本号索引请参见华为官方开发者文档推送服务 SDK 版本更新说明 。

在 app 模块的 `build.gradle` 添加如下内容：

```
dependencies {
// Add this line
implementation 'com.huawei.hms:push:6.7.0.300'
}
...
// Add to the bottom of the file
apply plugin: 'com.huawei.agconnect'
```

> 提示
>
> **华为 Push SDK 依赖说明**：最新版本 Push SDK 需要终端设备上安装HMS Core（APK）4.0.0.300 及以上版本；如果用户手机没有安装，您的应用调用 HMS Core 时，会自动引导用户提示安装。
>
> 默认支持 HMS Core（APK）的手机包括：部分 EMUI 4.0 和 4.1 的手机，以及 EMUI 5.0 及之后的华为手机。

使用融云 SDK 提供的华为推送 Service

为了接收华为推送透传消息和获取推送 Token，必须实现一个继承自华为 Push Kit 基础类 HmsMessageService 的 service。融云 SDK 已经实现了该 service，您只需要将融云提供的 `HMSPushService` 注册到 AndroidManifest.xml 文件中，注册内容如下：

在 `AndroidManifest.xml` 中添加如下配置信息

```
<service
android:name="io.rong.push.platform.hms.HMSPushService"
android:exported="false">
<intent-filter>
<action android:name="com.huawei.push.action.MESSAGING_EVENT" />
</intent-filter>
</service>
```

以上步骤也可参考华为官方开发者文档配置 Manifest 文件。

启用华为推送服务

请在初始化融云 SDK 之前启用华为推送服务。融云 SDK 将向华为推送服务注册、并将获取的华为推送 Token 上报给融云服务端。

PushConfig 为所有推送配置相关的入口类。

```
PushConfig config = new PushConfig.Builder()
.enableHWPush(true)
.XXX //此处忽略其它三方推送配置
.build();
RongPushClient.setPushConfig(config); //将推送相关配置设置到 SDK
```

拦截并解析推送点击 Intent

华为通知栏推送方式下，当用户点击通知时，华为系统会通过隐式调用的方式发出 intent，可通过拦截解析 intent，获取用户点击行为以及携带的数据。

1. 默认传递的 intent 如下：

```
intent://您的包名/conversationlist?isFromPush=true#Intent;scheme=rong;launchFlags=0x4000000;end
```

intent 会同时携带一些附加数据，比如在调用 `RongIMClient.getInstance.sendMessage()` 时传递的参数 pushData；使用控制台的广播推送功能时，自定义的键值对等。

使用默认 intent 跳转，在主工程中的 `AndroidManifest.xml` 的 Activity 中配置如下 `intent-filter`，点击通知即可跳转到此 Activity.

```
<activity
android:name="您的会话列表 activity"
android:exported="true">
<intent-filter>
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />
<data
android:host="${`applicationId`}"
android:pathPrefix="/conversationlist"
android:scheme="rong" />
</intent-filter>
</activity>
```

> 提示
>
> 设置的 intent 即为要隐式跳转的页面设置。详见华为官方开发者文档自定义点击消息动作。可在控制台，在 服务管理 -> 应用标识 -> Android -> 华为推送设置 栏目中自定义设置。

2. 在跳转界面获取并解析 intent 携带数据。

```
Intent intent = getIntent();
if (intent.getData().getScheme().equals("rong") && intent.getData().getQueryParameter("isFromPush") != null) {
if (intent.getData().getQueryParameter("isFromPush").equals("true")) {
String options = getIntent().getStringExtra("options"); // 获取 intent 里携带的附加数据
NLog.d(TAG, "options:", options);
try{
JSONObject jsonObject = new JSONObject(options);
if(jsonObject.has("appData")) { // appData 对应的是客户端 sendMessage() 时的参数 pushData
NLog.d(TAG, "pushData:", jsonObject.getString("appData"));
}
if(jsonObject.has("rc")) {
JSONObject rc = jsonObject.getJSONObject("rc");
NLog.d(TAG, "rc:", rc);
String targetId = rc.getString("tId"); // 该推送通知对应的目标 id.
String pushId = rc.getString("id"); // 控制台使用广播推送功能发出的推送通知，会有该字段，代表该推送的唯一 id， 需要调用下面接口，上传用户打开事件。
if(!TextUtils.isEmpty(pushId)) {
RongPushClient.recordNotificationEvent(pushId); // 上传用户打开事件，以便进行推送打开率的统计。
NLog.d(TAG, "pushId:", pushId);
}
if (rc.has("ext") && rc.getJSONObject("ext") != null) {
String ext = rc.getJSONObject("ext").toString(); // 使用控制台的广播推送功能时，填充的自定义键值对。
NLog.d(TAG, "ext:", ext);
}
}
} catch (JSONException e) {

}
enterActivity();
}
}
```

自定义铃声设置

Android 8.0 及以上手机支持设置通知渠道功能，在此基础上同时支持设置通道的自定义铃声功能。详细说明参见配置推送铃声

> 提示
>
> 音频文件需要打包到应用中，存放在应用 /res/raw/ 目录，存储的声音文件需要有扩展名，但是不要把扩展名写在 uri 中。HMS 相关错误码说明可参考： 错误码说明

角标未读数

融云不维护应用角标数量，融云客户端 SDK 不支持控制角标展示。如需了解与厂商推送通知相关的角标控制实现，可参考知识库文档推送角标。

常见问题

处理透传消息

融云 SDK 没有对华为的透传消息进行处理，如果用户通过华为推送运营平台使用了华为透传消息推送，可以继承 HMSPushService 实现 onMessageReceived 方法，并替换清单文件的 io.rong.push.platform.hms.HMSPushService。

无法获取 token 的异常情况

由于华为 4.0 推送需要华为手机 HMS Core 应用升级到 3.0 以上，2.0 的手机无法正常获取到 token 需要用户主动升级。下面是引导用户升级的代码示例，开发者可根据自身需求集成。

继承 PushMessageReceiver 重写 onThirdPartyPushState

```
public class SealNotificationReceiver extends PushMessageReceiver {
public static boolean needUpdate = false;

@Override
public boolean onNotificationMessageArrived(Context context, PushType pushType, PushNotificationMessage message) {
return false;
}

@Override
public boolean onNotificationMessageClicked(Context context, PushType pushType, PushNotificationMessage message) {
return false;
}
//华为获取 token 异常回调此方法
@Override
public void onThirdPartyPushState(PushType pushType, String action, long resultCode) {
super.onThirdPartyPushState(pushType, action, resultCode);
if (pushType.equals(PushType.HUAWEI)) {
if (resultCode == CommonCode.ErrorCode.CLIENT_API_INVALID) {
//设置标记位，引导用户升级
needUpdate = true;
}
}
}
}
```

注册监听

```
<!-- push start-->
<receiver
android:name=".push.SealNotificationReceiver"
android:exported="false">
<intent-filter>
<action android:name="io.rong.push.intent.MESSAGE_ARRIVED" />
<action android:name="io.rong.push.intent.MESSAGE_CLICKED" />
<action android:name="io.rong.push.intent.THIRD_PARTY_PUSH_STATE" />
</intent-filter>
</receiver>
```

当需要更新 HMS Core ，主动调用 RongPushClient.resolveHMSCoreUpdate() 方法，注意要传递 activity

```
if (SealNotificationReceiver.needUpdate) {
//重置标记位，防止多次弹窗提醒
SealNotificationReceiver.needUpdate = false;
RongPushClient.resolveHMSCoreUpdate(this);
}
```

删除配置（如配置过 2.x 版本及更早版本的华为推送）

如果您是首次接入华为推送，您无需删除配置。

如果您配置过 2.x 版本及更早版本的华为推送，请参见如下示例，删除从 "HMS 配置开始" 到 "HMS配置结束" 注释间的全部配置。

```
///HMS 配置开始
<meta-data
android:name="com.huawei.hms.client.appid"
android:value="10535759" />
<meta-data
android:name="firebase_messaging_auto_init_enabled"
android:value="false" />
<meta-data
android:name="firebase_analytics_collection_enabled"
android:value="false" />

<!-- BridgeActivity定义了HMS-SDK中一些跳转所需要的透明页面-->
<activity
android:name="com.huawei.hms.activity.BridgeActivity"
android:configChanges="orientation|locale|screenSize|layoutDirection|fontScale"
android:excludeFromRecents="true"
android:exported="false"
android:hardwareAccelerated="true"
android:theme="@android:style/Theme.Translucent">
<meta-data
android:name="hwc-theme"
android:value="androidhwext:style/Theme.Emui.Translucent" />
</activity>

<!-- 解决华为移动服务升级问题的透明界面（必须声明）-->
<activity
android:name="io.rong.push.platform.hms.common.HMSAgentActivity"
android:configChanges="orientation|locale|screenSize|layoutDirection|fontScale"
android:excludeFromRecents="true"
android:exported="false"
android:hardwareAccelerated="true"
android:theme="@android:style/Theme.Translucent">
<meta-data
android:name="hwc-theme"
android:value="androidhwext:style/Theme.Emui.Translucent" />
</activity>

<provider
android:name="com.huawei.hms.update.provider.UpdateProvider"
android:authorities="cn.rongcloud.im.hms.update.provider"
android:exported="false"
android:grantUriPermissions="true">
</provider>
<!-- 第三方相关 :接收Push消息（注册、Push消息、Push连接状态）广播 -->
<receiver android:name="io.rong.push.platform.hms.HMSReceiver">
<intent-filter>
<!-- 必须,用于接收token -->
<action android:name="com.huawei.android.push.intent.REGISTRATION" />
<!-- 必须，用于接收消息 -->
<action android:name="com.huawei.android.push.intent.RECEIVE" />
<!-- 可选，用于点击通知栏或通知栏上的按钮后触发onEvent回调 -->
<action android:name="com.huawei.android.push.intent.CLICK" />
<!-- 可选，查看push通道是否连接，不查看则不需要 -->
<action android:name="com.huawei.intent.action.PUSH_STATE" />
</intent-filter>
</receiver>
<receiver android:name="com.huawei.hms.support.api.push.PushEventReceiver">
<intent-filter>
<!-- 接收通道发来的通知栏消息，兼容老版本Push -->
<action android:name="com.huawei.intent.action.PUSH" />
</intent-filter>
</receiver>
///HMS 配置结束
```

移动端推送注册的 log 表明成功，查询消息流转也成功下发，但还是收不到推送

这个问题是因为调起 activity 的 intent 所指向的 activity，没有在 manifest 配属性导致的，偶现与一些 EMUI 版本。请参考如下步骤：

1. 如果没有自定义点击跳转的 intent，那么需要在您的会话列表界面所在的 activity 在 manifest的配置中加 android:exported="true" 这个flag

2. 如果配置了自定义跳转的 intent，需要在相应要跳入的 activity 在 manifest 的配置中加 android:exported="true" 这个 flag

集成荣耀推送 集成荣耀推送

更新时间:2024-08-30

按照本指南集成 荣耀推送服务，让融云 SDK 支持从荣耀推送服务获取推送通知。

在集成第三方推送前，请确保已在控制台配置 **Android 应用 ID**。详见推送集成概述。

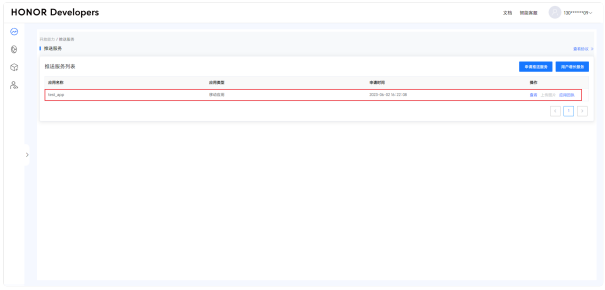
> 提示
> IMLib SDK（开发版）从 5.6.7 版本开始支持荣耀推送。

> 重要
> 如果您的项目集成或升级到 IM SDK（开发版）5.6.7 版本，必须同时集成荣耀推送，否则荣耀 Magic OS 8.0 及之后系统版本的设备可能无法接收推送通知。

在控制台配置荣耀推送

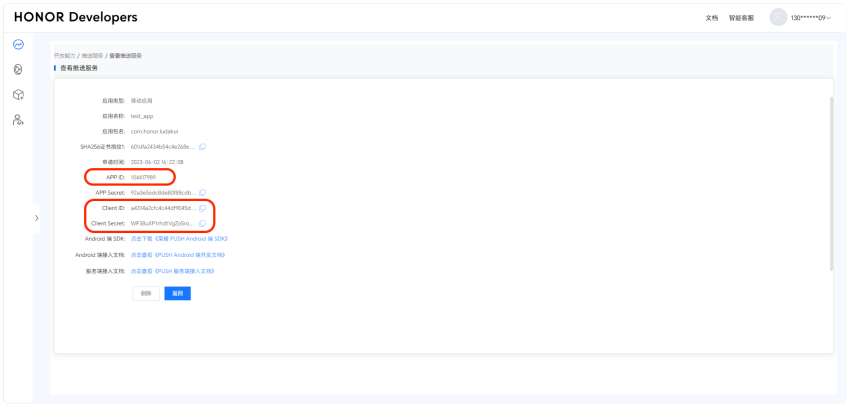
如果想通过荣耀推送接收融云即时通讯服务的推送通知，您需要在控制台上提供您的荣耀推送应用的详细信息。

1. 前往荣耀开发者服务平台，在 推送服务 页面，选择您创建的应用。



> 提示
> - 如果没有荣耀开发者账号，或尚未创建项目和应用，请先创建账号、项目和应用，并开通推送服务。详见荣耀开发者文档创建账号、创建应用、申请开通推送服务。
> - 在继续完成以下步骤前，请确保已在荣耀开发者服务平台为应用开通了推送服务，并配置签名证书指纹。如您对荣耀控制台上配置推送服务的流程有任何疑问，可参见荣耀官方开发者文档上架申请指南。

您需要记录下应用下的 **App ID**、**Client ID** 和 **Client Secret**。



2. 打开控制台，在应用标识页面点击设置推送，找到 **Android > 荣耀推送**，填入上一步获取的 **Client ID**、**Client Secret**。

3. （可选）配置自定义点击消息动作的 **Intent**，用于打开应用自定义页面。该字段对应荣耀官方下行消息接口中 `ClickAction.type` 为 1 时的 `ClickAction.action`。如有疑问，详见荣耀开发者文档消息推送。自定义 intent 后，需在 `AndroidManifest.xml` 的 Activity 中配置 `intent-filter`，接收自定义的 intent。

4. （可选）配置推送角标。

   - **badgeAddNum**：应用角标累加数字非应用角标实际显示数字，为大于 0 小于 100 的整数。例如，某应用当前有N条未读消息，若 add_num 设置为3，则每发一次消息，应用角标显示的数字累加3，为N+3。该字段对应荣耀官方下行消息接口中 `BadgeNotification.addNum`。如有疑问，详见荣耀开发者文档消息推送。
   - **Activity**：应用入口 Activity 类全路径。样例：com.example.honortest.MainActivity。该字段对应荣耀官方下行消息接口中 `BadgeNotification.badgeClass`。如有疑问，详见荣耀开发者文档消息推送。

5. 配置**推送通知标题**。设置默认的推送通知标题。一般情况下客户端发送消息转 Push 时不使用此标题设置。在调用融云服务端 API `/push.json`、`/push/user.json`、`/push/custom.json` 接口推送通知时，如未传入通知标题，则使用该处设置的标题。从服务端发消息时，如果发送用户 ID 在融云服务端没有用户名，也会使用此“推送通知标题”。

6. 保存设置。所有设置 30 分钟后生效。

您已完成需要控制台荣耀推送配置的全部内容。现在可以设置客户端集成。

配置客户端接收荣耀推送

首先，需要将荣耀推送客户端 SDK 添加到您的 Android 项目。

下载荣耀服务配置文件

以下步骤来自荣耀开发者文档添加应用配置文件：

1. 登录荣耀开发者服务平台，单击**应用管理**，在应用列表中找到目标应用，单击**应用详情**。

2. 在应用基础信息查看页面的 SDK 配置区域，下载 `mcs-services.json` 配置文件。

3. 将下载好的 `mcs-servics.json` 文件拷贝到应用级根目录下。

导入荣耀推送 SDK

> 重要
>
> 荣耀推送 SDK Maven 仓库的配置在 Gradle 插件 7.0 以下版本、7.0 版本和 7.1 及以上版本有所不同。本步骤中以荣耀官方文档中 Gradle 插件 7.0 以下版本为例。其他 Gradle 版本配置方法，详见荣耀官方文档 集成 SDK。

1. 配置荣耀官方 Maven 仓库，以 Gradle 7.0 以下版本为例。

   打开在 project 的 `build.gradle`：

   - 在 buildscript > repositories 中配置 SDK 的 Maven 仓地址。
   - 在 allprojects > repositories 中配置 SDK 的 Maven 仓地址。
   - 如果 App 中添加了 mcs-services.json 文件则需要在 buildscript > dependencies 中增加 asplugin 插件配置。

```
buildscript {
repositories {
google()
jcenter()
// 配置SDK的Maven仓地址。
maven {url 'https://developer.hihonor.com/repo'}
}
dependencies {
...
// 增加asplugin插件配置，推荐您使用最新版本。
classpath 'com.hihonor.mcs:asplugin:2.0.0'
// 增加gradle插件配置，根据gradle版本选择对应的插件版本号
classpath 'com.android.tools.build:gradle:4.1.2'
}
}

allprojects {
repositories {
google()
jcenter()
// 配置SDK的Maven仓地址。
maven {url 'https://developer.hihonor.com/repo'}
}
}
```

2. 在 dependencies 中添加如下编译依赖，集成荣耀官方推送 SDK。最新版本号可请参见荣耀推送 SDK 版本信息。

```
dependencies {
// 添加如下配置，推荐您使用最新版本。
implementation 'com.hihonor.mcs:push:7.0.61.303'
}
```

3. 添加荣耀 asplugin 插件配置。以 Gradle 7.0 以下版本为例，在应用级别的 build.gradle 文件头部声明下一行添加如下配置：

```
apply plugin: 'com.hihonor.mcs.asplugin'
```

> 提示
>
> 引入 asplugin 插件需要在 Android studio 中配置 gradle 的 jdk 版本为 11 以上。详见荣耀官方文档 配置开发环境。

4. 配置 AndroidManifest.xml 文件。

   1. 添加如下 meta-data 标签，配置荣耀推送服务的 App ID。

```
<manifest>
<application>
<meta-data
android:name="com.hihonor.push.app_id"
android:value="您的AppId" />
</application>
</manifest>
```

   2. 融云 SDK 已经创建了荣耀推送接收服务，您需要在 application 标签下注册该服务。exported 属性需要设置为 false，限制其他应用的组件唤醒该 service。

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="io.rong.push.plugin.honor">

<application>
<service
android:name="io.rong.push.platform.honor.HonorPushService"
android:exported="false">
<intent-filter>
<action android:name="com.hihonor.push.action.MESSAGING_EVENT" />
</intent-filter>
</service>
</application>
</manifest>
```

5. 如果您的应用 targerSdkVersion 大于等于 30，需要在 AndroidManifest.xml 中添加标签，指定了应用可以处理的 intent 的 action。

```
<queries>
<intent>
<action android:name="com.hihonor.push.action.BIND_PUSH_SERVICE" />
</intent>
</queries>
```

配置签名

参考荣耀官方文档，将生成签名证书指纹步骤中生成的签名文件拷贝到工程的 App 目录下，在 build.gradle 文件中配置签名。

```
android {
signingConfigs {
config {
// 根据您实际的签名信息，替换以下参数中的xxxx
keyAlias 'xxxx'
keyPassword 'xxxx'
storeFile file('xxxx.jks')
storePassword 'xxxx'
}
}
buildTypes {
debug {
signingConfig signingConfigs.config
}
release {
signingConfig signingConfigs.config
minifyEnabled true
proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
}
}
}
```

启用荣耀推送服务

> 提示
>
> SDK 从 5.6.7 版本开始支持荣耀推送，并向融云服务端上报荣耀推送的 Token。

请在初始化融云 SDK 之前启用荣耀推送服务。融云 SDK 将向荣耀推送服务注册设备，并将从荣耀推送服务端获取的荣耀推送 Token 上报给融云推送服务端。

PushConfig 为所有推送配置相关的入口类。

```
PushConfig config = new PushConfig.Builder()
.enableHonorPush(true)
.XXX //此处忽略其它三方推送配置
.build();
RongPushClient.setPushConfig(config); //将推送相关配置设置到 SDK
```

混淆配置

请参见荣耀推送官方文档配置混淆脚本。

处理推送通知的点击事件

- **自定义推送通知点击事件**：介绍如何实现 SDK 的默认跳转行为，以及如何自定义处理点击事件。详见自定义推送通知点击事件。
- **自定义推送通知样式**：SDK 接收到其他第三方厂商的推送后，弹出的通知是系统通知，由手机系统底层直接弹出通知，所以不支持自定义。

角标未读数

融云不维护应用角标数量，融云客户端 SDK 不支持控制角标展示。如需了解与厂商推送通知相关的角标控制实现，可参考知识库文档推送角标。

集成魅族推送 集成魅族推送

更新时间:2024-08-30

按照本指南集成 魅族 Flyme 推送客户端，让融云 SDK 支持从魅族推送服务获取推送通知。

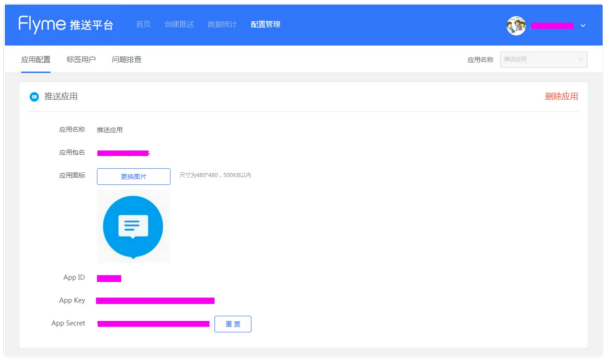
在集成第三方推送前，请确保已在控制台配置 **Android 应用 ID**。详见推送集成概述。

在控制台配置魅族推送

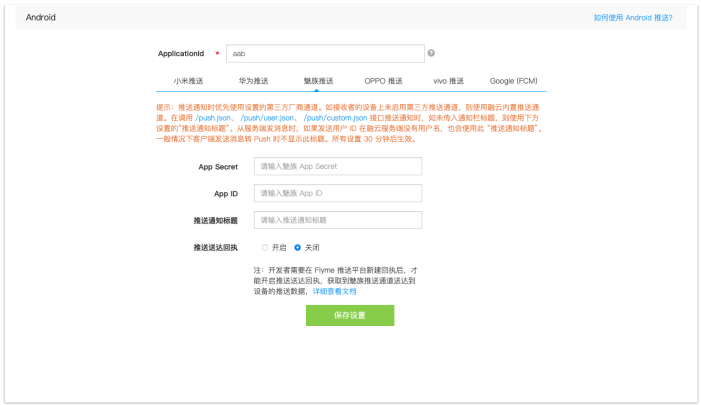
如果想通过魅族推送接收融云即时通讯服务的推送通知，您需要在控制台上提供您的魅族推送应用的详细信息。

1. 前往 魅族推送平台，并记录下应用的 **AppID**、**AppKey**、**AppSecret**。

> 提示
> 如果没有魅族 Flyme 开发者账号，或尚未创建应用，参考魅族 Flyme 推送接入文档。Flyme 开发者账号通过认证后可创建应用。



2. 打开控制台，在应用标识页面点击**设置推送**，找到 **Android** > **魅族推送**，填入上一步获取的 **AppID**、**AppSecret**。



3. （可选）配置**推送通知标题**。设置默认的推送通知标题。一般情况下客户端发送消息转 Push 时不使用此标题设置。在调用融云服务端 API `/push.json`、`/push/user.json`、`/push/custom.json` 接口推送通知时，如未传入通知标题，则使用该处设置的标题。从服务端发消息时，如果发送用户 ID 在融云服务端没有用户名，也会使用此“推送通知标题”。

4. 是否开启推送回执。您需要在魅族推送平台中新建回执，并在此启用后，才能获取到魅族通道送达数据。具体配置流程详见上报推送数据。

5. 保存设置。所有设置 30 分钟后生效。

您已完成需要控制台魅族推送配置的全部内容。现在可以设置客户端集成。

配置客户端接收魅族推送

首先，需要将魅族推送客户端 SDK 添加到您的 Android 项目。

魅族已将 4.1.0 之后的推送客户端 SDK 发布到 mavenCentral()。确保您的项目已经添加 mavenCentral() 后，您可以在 App 的 build.gradle 中添加依赖，直接引入魅族推送客户端 SDK：

```
dependencies {
implementation 'com.meizu.flyme.internet:push-internal:4.1.4@aar'
}
```

您也可以直接从魅族下载最新版 AAR 包进行手动集成（Flyme 推送 SDK 下载）或从融云下载。融云官网 SDK 下载页提供魅族推送客户端 SDK 下载。在页面勾选第三方推送后，下载 zip 文件。在解压后的文件目录中找到 pushlibs 下的魅族推送客户端 SDK 文件（meizu-push-xxx.aar），拷贝到 app 的 libs 目录下。


```
dependencies {
implementation (name: 'meizu-push-xxx', ext: 'aar')
}
```

配置魅族推送客户端的权限

在主工程中的 AndroidManifest.xml 里增加进行以下权限声明：

```
<!-- MeiZu 配置开始 -->
<!-- 兼容 flyme5.0 以下版本，魅族内部集成 pushSDK 必填，不然无法收到 消息-->
<uses-permission
android:name="com.meizu.flyme.push.permission.RECEIVE"/>
<permission
android:name="您的包名.push.permission.MESSAGE"
android:protectionLevel="signature"/>
<uses-permission android:name="您的包名.push.permission.MESSAGE"/>
<!-- 兼容 flyme3.0 配置权限-->
<uses-permission android:name="com.meizu.c2dm.permission.RECEIVE" />
<!-- MeiZu 配置结束 -->
```

使用融云 SDK 提供的广播接收器

为了接收魅族推送消息，必须实现一个继承自魅族 MzPushMessageReceiver 类的广播接收器（Broadcast Receiver）。融云 SDK 已经实现了该广播接收器，您只需要将融云提供的 MeiZuReceiver 注册到 AndroidManifest.xml 文件中，注册内容如下：

```
<application>
<receiver
android:name="io.rong.push.platform.meizu.MeiZuReceiver" android:exported="true"> <!-- 由于 Android 12 的要求，存在 intent-filter 的组件 exported 必须设置为 true -->
<intent-filter>
<!-- 接收 push 消息 -->
<action android:name="com.meizu.flyme.push.intent.MESSAGE"/>
<!-- 接收 register 消息 -->
<action
android:name="com.meizu.flyme.push.intent.REGISTER.FEEDBACK" />
<!-- 接收 unregister 消息-->
<action
android:name="com.meizu.flyme.push.intent.UNREGISTER.FEEDBACK"/>
<!-- 兼容低版本 Flyme3 推送服务配置 -->
<action android:name="com.meizu.c2dm.intent.REGISTRATION"/>
<action android:name="com.meizu.c2dm.intent.RECEIVE" />
<category android:name="您的包名"/>
</intent-filter>
</receiver>
</application>
```

启用魅族 Flyme 推送服务

请在初始化融云 SDK 之前，将魅族的 **AppID**、**AppKey** 添加到 PushConfig，并提供给 SDK。融云 SDK 将向魅族 Flyme 推送服务注册、并将获取的 Flyme 推送 Token 上报给融云服务端。

PushConfig 为所有推送配置相关的入口类。

```
PushConfig config = new PushConfig.Builder()
.enableMeiZuPush("魅族 appId", "魅族 appKey")
.build();
RongPushClient.setPushConfig(config);
```

解决与已有 Flyme 推送集成的冲突

如果您的应用或应用依赖的其他 SDK 已集成魅族 Flyme 推送客户端，此时无法再按上述步骤集成魅族推送，否则会发生冲突。

请如下步骤进行处理：

1. 创建一个自定义的 Broadcast Receiver，例如 `MZPushReceiver`，继承融云 SDK 提供的 `MeiZuReceiver`，并覆写父类方法。

2. 使用新创建的 `MZPushReceiver` 替换应用中原有的 Flyme 推送的 Broadcast Receiver。
3. 在 `AndroidManifest.xml` 中注册新创建的 `MZPushReceiver`。

集成 OPPO 推送 集成 OPPO 推送

更新时间:2024-08-30

按照本指南集成OPPO 推送客户端，融云 SDK 支持从 OPPO 推送服务收取推送。

> 提示
> - 并非所有的 OPPO 手机都支持 OPPO 推送。OPPO 推送目前支持 ColorOS 3.1 及以上的系统的OPPO的机型，一加5/5t及以上机型，realme 所有机型。详见 OPPO 推送服务文档 业务功能使用问题。
> - 在不支持 OPPO 推送的 OPPO 手机上会使用融云默认推送。

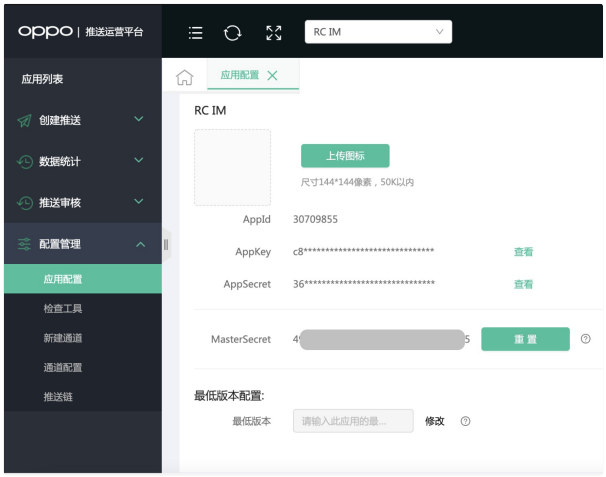
在集成第三方推送前，请确保已在控制台配置 **Android 应用 ID**。详见推送集成概述。

在控制台配置 OPPO 推送

如果想通过 OPPO 推送接收融云即时通讯服务的推送通知，您需要在控制台上提供您的 OPPO 推送应用的详细信息。

1. 前往 OPPO 推送运营平台 的**配置管理 > 应用配置**中获取推送凭证，包括 **AppKey**、**AppSecret**、 **MasterSecret**。

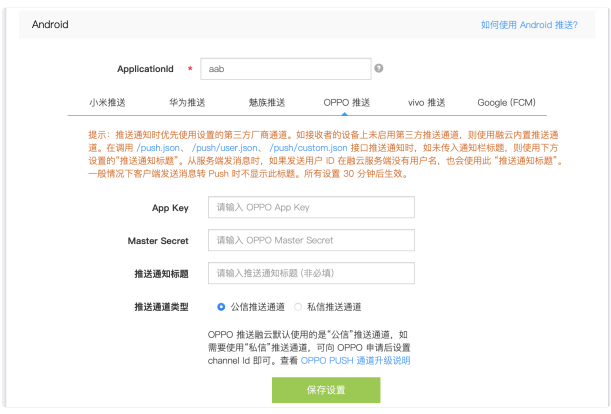
> 提示
> - 如果没有 OPPO 开发者账号，或尚未创建应用，或尚未开启推送服务权限，请先在 OPPO 开发者平台上创建账号、应用并开启推送服务权限。详见 OPPO 开发者文档 推送服务开启指南。
> - OPPO 要求开启推送服务需要注册成为 OPPO 企业开发者，详情请参考 OPPO企业开发者帐号注册流程。



OPPO 推送凭证用途如下：

- **AppKey**：OPPO 应用的身份标识。客户端以及在控制台配置 OPPO 推送时均需使用。
- **AppSecret**：客户端集成 OPPO 推送 SDK 时使用。
- **MasterSecret**（即 AppServerSecret）：在控制台配置 OPPO 推送服务凭证时使用。

2. 打开控制台，在应用标识页面点击**设置推送**，找到 **Android > OPPO 推送**，填入上一步获取的 **App Key**、**Master Secret**。



3. （可选）配置**推送通知标题**。设置默认的推送通知标题。一般情况下客户端发送消息转 Push 时不使用此标题设置。在调用融云服务端 API `/push.json`、`/push/user.json`、`/push/custom.json` 接口推送通知时，如未传入通知标题，则使用该处设置的标题。从服务端发消息时，如果发送用户 ID 在融云服务端没有用户名，也会使用此“推送通知

标题”。

4. 配置推送通道类型。OPPO 官方开发者文档说明如下：为了改善终端用户的通知体验，营造良好可持续的推送生态，OPPO推送对各个APP的push消息数量进行了限量管控；同时针对即时聊天/系统提醒等push需求可以申请走私信通道（该通道不限量）。

> 提示
> 融云服务端默认使用的是 OPPO 公信推送通道，单日推送消息次数受限，总的推送次数用完后，当日将无法再给用户推送内容。目前单日推送数量为：累计注册用户数*2。如需使用不受推送数量限制的私信推送通道，可参考 OPPO 文档推送私信通道申请。申请私信通道完成后，请返回控制台填入私信推送通道的 channel Id。

5. 保存设置。所有设置 30 分钟后生效。

您已完成需要控制台 OPPO 推送配置的全部内容。现在可以设置客户端集成。

配置客户端接收 OPPO 推送

1. 首先，需要将 OPPO 推送客户端 SDK 添加到您的 Android 项目。

> 重要
> - 如果您项目使用的 IMLib/IMKit SDK 版本大于等于 5.2.1，必须使用 OPPO 推送客户端 SDK 3.0.0 或更新版本。
> - 如果您项目使用的 IMLib/IMKit SDK 版本大于等于 5.6.8，必须使用 OPPO 推送客户端 SDK 3.4.0 或更新版本。

- 融云官网 SDK 下载页提供 OPPO 推送客户端 SDK 下载。在页面勾选第三方推送后，下载 zip 文件。在解压后的文件目录中找到 pushlibs 下的 OPPO 推送客户端 SDK 文件（`push-xxx.aar`，3.4.0）。拷贝 .aar 文件到 app 的 `libs` 目录下。

- 如需获取最新版 OPPO 推送客户端 SDK，可以从 OPPO 直接获取 aar 文件。参见 OPPO 官方开发者文档 Android SDK 集成。

2. 在项目中导入 OPPO 推送 aar 包。在 App 的 `build.gradle` 中添加依赖：

```
dependencies {
...
implementation(name: 'push-3.4.0', ext: 'aar')
//以下依赖都需要添加
implementation 'com.google.code.gson:gson:2.6.2'
implementation 'commons-codec:commons-codec:1.6'
implementation 'androidx.annotation:annotation:1.1.0'
...
}
```

注册 OPPO 推送的服务

在主工程的 AndroidManifest.xml 中添加 OPPO 推送配置需要的 service：

```
<service
android:name="com.heytap.msp.push.service.CompatibleDataMessageCallbackService"
android:exported="true"
android:permission="com.coloros.mcs.permission.SEND_MCS_MESSAGE" >
<intent-filter>
<action android:name="com.coloros.mcs.action.RECEIVE_MCS_MESSAGE" />
</intent-filter>
</service>
<service
android:name="com.heytap.msp.push.service.DataMessageCallbackService"
android:exported="true"
android:permission="com.heytap.mcs.permission.SEND_PUSH_MESSAGE" >
<intent-filter>
<action android:name="com.heytap.mcs.action.RECEIVE_MCS_MESSAGE" />
<action android:name="com.heytap.msp.push.RECEIVE_MCS_MESSAGE" />
</intent-filter>
</service>
```

关于权限要求的说明，可参见 OPPO 官方开发者文档 SDK 数据安全说明。

混淆配置

如果您的应用使用了混淆，您可以使用下面的代码混淆配置：

```
-keep public class * extends android.app.Service
-keep class com.heytap.msp.** { *;}
```

启用 OPPO 推送服务

请在初始化融云 SDK 之前，将 OPPO 的 **App Key**、**App Secret** 添加到 PushConfig，并提供给 SDK。融云 SDK 将向 OPPO 推送服务注册、并将获取的 OPPO 推送 Token 上报给融云服务端。

PushConfig 为所有推送配置相关的入口类。

```
PushConfig.Builder builder = new PushConfig.Builder();
builder.enableOppoPush(OPPO_App_KEY, OPPO_App_Secret);
RongPushClient.setPushConfig(builder.build());
```

> 提示
>
> 若同时使用其他推送请在 builder 中启用其他推送，若使用旧版本的 RongPushClient 的 register 方式启用的推送，请改用此 RongPushClient#setPushConfig 方式由 builder 启用各推送。

常见问题

接收 OPPO 推送需要用户的 OPPO 手机开启应用的通知权限。由于融云推送属于应用级别的推送，会受系统各种权限限制，我们建议您在使用时，在设置里打开自启动权限和通知权限，或者勾选"信任此应用"等，以提高推送到达率。

OPPO推送暂时只支持通知栏消息的推送。消息下发到 OS 系统模块并由系统通知模块展示，在用户点击通知前，不启动应用。

由于并非所有的 OPPO 手机都支持 OPPO 推送，所以仅在支持 OPPO 推送的 OPPO 手机上使用 OPPO 推送，在不支持 OPPO 推送的 OPPO 手机上使用融云默认推送。

关于 OPPO 推送服务的其他问题，建议参考 OPPO 官方开发者文档「常见 FAQ」中技术问题和业务功能使用问题。

集成 vivo 推送 集成 vivo 推送

更新时间:2024-08-30

按照本指南集成vivo 推送客户端，融云 SDK 支持从 vivo 推送服务收取推送。

在集成第三方推送前，请确保已在控制台配置 **Android 应用 ID**。详见推送集成概述。

在控制台配置 vivo 推送

如果想通过 vivo 推送接收融云即时通讯服务的推送通知，您需要在控制台上提供您的 vivo 推送应用的详细信息。

1. 前往 vivo 开发者平台-推送服务器平台，并记录下 vivo 应用的 **AppID**、**AppKey**、**AppSecret**。

> 提示
> 如果没有 vivo 开发者账号，或尚未创建应用，参考 vivo 文档 vivo 推送接入流程。

2. 打开控制台，在应用标识页面点击**设置推送**，找到 **Android > vivo 推送**，填入上一步获取的 **AppID**、**AppKey**、**AppSecret**。

3. （可选）配置**推送通知标题**。设置默认的推送通知标题。一般情况下客户端发送消息转 Push 时不使用此标题设置。在调用融云服务端 API `/push.json`、`/push/user.json`、`/push/custom.json` 接口推送通知时，如未传入通知标题，则使用该处设置的标题。从服务端发消息时，如果发送用户 ID 在融云服务端没有用户名，也会使用此“推送通知标题”。

4. 配置**推送模式**。关于正式推送与测试推送的区别，请参考 vivo 文档vivo 推送使用指南。

5. （可选）配置**推送通道类型**与 **Category**（消息二级分类）。如果调用客户端或服务端 API 发送消息或推送通知时未传值，默认使用此处配置的值。请参考 vivo 文档推送消息分类说明进行配置。

6. 保存设置。所有设置 30 分钟后生效。

您已完成需要控制台 vivo 推送配置的全部内容。现在可以设置客户端集成。

配置客户端接收 vivo 推送

首先，需要将 vivo 客户端 SDK 添加到您的 Android 项目。

重要

- 如果您项目使用的 IMLib/IMKit SDK 版本大于等于 5.2.1，则 vivo 推送客户端 SDK 必须集成或升级至 3.0.0.4 及之后版本。
- 如果您项目使用的 IMLib/IMKit SDK 版本大于等于 5.6.8，必须使用 vivo 推送客户端 SDK 3.0.0.7 或更新版本。

您也可以直接从 vivo 下载最新版 AAR 包进行手动集成（参考 vivo 的Android PUSH-SDK 集成指南）或从融云下载。融云官网 SDK 下载页提供 vivo 推送客户端 SDK 下载。在页面勾选第三方推送后，下载 zip 文件。在解压后的文件目录中找到 pushlibs 下的 vivo 推送客户端 SDK 文件（vivo-pushSDK-xxx.aar），拷贝到 app 的 libs 目录下。

在项目中导入 vivo 推送 aar 包。在 App 的 build.gradle 中添加依赖：

```
dependencies {
implementation (name: 'vivo-pushSDK-xxx', ext: 'aar')
}
```

处理 vivo 推送的权限、服务与广播接收器

在主工程中的 AndroidManifest.xml 中增加权限声明，集成 vivo 推送客户端 SDK 只需要配置网络权限：

```
<!-- 推送需要的权限列表 -->
<uses-permission android:name="android.permission.INTERNET"/>
```

vivo 推送客户端 SDK 需要配置对应的 AppID 、AppKey 信息（从 vivo 开发者平台-推送服务器平台 获取），请在 AndroidManifest.xml 中添加以下配置：

```
<!-- vivo 推送配置项 -->
<meta-data
android:name="com.vivo.push.api_key"
android:value="您的 vivo 推送平台生成 AppKey"/>
<meta-data
android:name="com.vivo.push.app_id"
android:value="您的 vivo 推送平台生成 AppID"/>
```

接入 vivo 推送客户端 SDK 注册以下 service、activity。

```
<!-- vivo 推送服务需要配置的 service、activity -->
<service
android:name="com.vivo.push.sdk.service.CommandClientService"
android:exported="true"/>
<activity
android:name="com.vivo.push.sdk.LinkProxyClientActivity"
android:exported="false"
android:screenOrientation="portrait"
android:theme="@android:style/Theme.Translucent.NoTitleBar"/>
```

以上内容来自 vivo。如需了解更多细节，请参考 vivo 文档 Android PUSH-SDK 集成指南。

使用融云 SDK 提供的广播接收器

为了接收 vivo 推送消息，必须实现一个继承自 vivo OpenClientPushMessageReceiver 类的广播接收器（Broadcast Receiver）。融云 SDK 已经实现了该广播接收器，您只需要将融云提供的 VivoPushMessageReceiver 注册到 AndroidManifest.xml 文件中，注册内容如下：

```
<!-- vivo push 推送 receiver 声明 -->
<receiver android:name="io.rong.push.platform.vivo.VivoPushMessageReceiver"
android:exported="true">
<intent-filter>
<!-- 接收 vivo push 消息 -->
<action android:name="com.vivo.pushclient.action.RECEIVE" />
</intent-filter>
</receiver>
```

启用 vivo 推送服务

请在初始化融云 SDK 之前启用 vivo 推送服务。融云 SDK 将向 vivo 推送服务注册、并将获取的 vivo 推送 Token 上报给融云服务端。

`PushConfig` 为所有推送配置相关的入口类。

```
PushConfig config = new PushConfig.Builder()
.enableVivoPush(true)
.build();
RongPushClient.setPushConfig(config);
```

解决与已有 vivo 推送集成的冲突

如果您的应用或应用依赖的其他 SDK 已集成 vivo 推送客户端，此时无法再按上述步骤集成 vivo 推送，否则会发生冲突。

请如下步骤进行处理：

1. 创建一个自定义的 Broadcast Receiver，例如 `VivoPushReceiver`，继承融云 SDK 提供的 `VivoPushMessageReceiver`，并覆写父类方法。
2. 使用新创建的 `VivoPushReceiver` 替换应用中原有的 vivo 推送的 Broadcast Receiver。
3. 在 `AndroidManifest.xml` 中注册新创建的 `VivoPushReceiver`。

集成 FCM 推送 集成 FCM 推送

更新时间:2024-08-30

- FCM 推送通道适用于海外正式发售的 Android 设备（内置 Google GMS 服务），且会在海外网络环境下启用。
- 建议您在集成后根据测试 FCM 推送 中描述的条件与步骤进行测试。

融云服务端已集成与 FCM 后端通信的功能组件。在消息接收者设备上运行的 App 被杀进程，或者在后台被挂起，或者在后台存活超过 2 分钟的情况下，IM SDK 长连接通道会断开。此时如有消息需要送达，融云服务端会向 FCM 后端发送消息请求，然后由 FCM 后端再将消息发送到用户设备上运行的客户端应用。

Android 项目集成 FCM

本节内容将遵照 Google 推荐的设置工作流，描述如何通过 Firebase 控制台 将 Firebase 添加到您的 Android 项目。在此过程中，您必须手动将插件和配置文件添加到您的项目。

为帮助您快速以下步骤已经简化。如需详细步骤，您可以参考 Google 文档，或 Firebase 中文文档。

前提条件

- 安装最新版本的 Android Studio，或将其更新为最新版本。
  - 确保您的项目满足以下要求：
  - （SDK ≧ 5.6.3）使用 Android 5.0（API 21）或更高版本；
  - （SDK < 5.6.3）使用 Android 4.4（API 19）或更高版本；
  - 使用 Jetpack (AndroidX)，这需要满足以下版本要求：
    - `com.android.tools.build:gradle 3.2.1` 或更高版本
    - `compileSdkVersion 28` 或更高版本
- 设置一台实体设备或使用模拟器运行您的应用。
  请注意，FCM 客户端属于 依赖于 Google Play 服务的 Firebase SDK，需要在设备或模拟器上安装 Google Play 服务。
- 使用您的 Google 帐号登录 Firebase。

如要将 Firebase 添加到您的应用，您需要在 Firebase 控制台和打开的 Android 项目中执行若干任务（例如，从控制台下载 Firebase 配置文件，然后将配置文件移动到 Android 项目中）。

第 1 步：创建 Firebase 项目

1. 在Firebase 控制台中，点击**添加项目**。
   - 如需创建新项目，请输入要使用的项目名称。您也可以视需要修改项目名称下方显示的项目 ID。
   - 如需将 Firebase 资源添加到现有 Google Cloud 项目，请输入该项目的名称或从下拉菜单中选择该项目。
2. 点击**继续**。最后，点击**创建项目**（如果使用现有的 Google Cloud 项目，则点击**添加 Firebase**）。

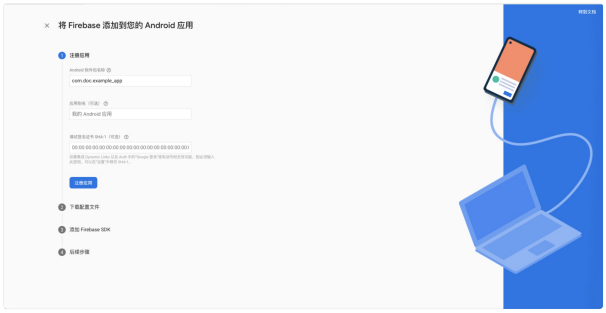
Firebase 会自动为您的 Firebase 项目预配资源。完成此过程后，您将进入 Firebase 控制台中 Firebase 项目的概览页面。

Firebase 项目实际上只是一个启用了额外的 Firebase 特定配置和服务的 Google Cloud 项目。在创建新的 Firebase 项目时，您实际上是在幕后创建 Google Cloud 项目。详情可参考 Firebase 中文文档 Firebase 项目与 Google Cloud 之间的关系。

第 2 步：在 Firebase 中注册您的 Android 应用

如需在 Android 应用中使用 Firebase，您需要向 Firebase 项目注册您的应用。注册应用的过程通常称为将应用“添加”到项目中。

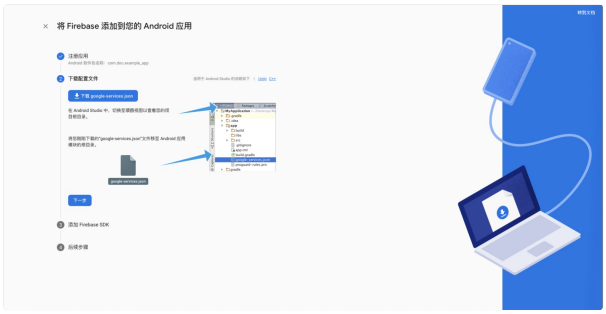
1. 前往 Firebase 控制台。
2. 在项目概览页面的中心位置，点击 **Android** 图标或添加应用，启动设置工作流。
3. 在 **Android 软件包名称**字段中输入应用的软件包名称。
   - 软件包名称是您的应用在设备上和 Google Play 商店中的唯一标识符。
   - 软件包名称通常称为应用 ID。
   - 在模块（应用级）Gradle 文件（通常是 app/build.gradle）中查找应用的软件包名称（示例软件包名称：com.yourcompany.yourproject）。
   - 请注意，软件包名称值区分大小写，并且当您在 Firebase 项目中注册此 Firebase Android 应用后，将无法更改其软件包名称。
4. 如有需要，可完成其他可选配置。然后点击**注册应用**。



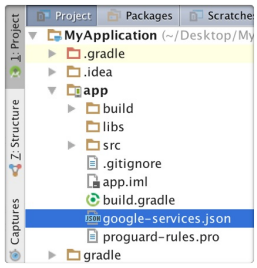
第 3 步：在 Android Studio 项目中添加 Firebase 配置文件

本步骤描述如何将 Firebase Android 配置文件添加到您的应用。

1. 注册应用后，点击**下载 google-services.json** 以获取 Firebase Android 配置文件 (`google-services.json`)。



2. 将配置文件移到应用的模块（应用级）目录中。



- 如需详细了解此配置文件，请访问了解 Firebase 项目。
- 您可以随时再次下载 Firebase 配置文件。
- 请确保该配置文件名未附加其他字符，如 (2)。

3. 如需在应用中启用 Firebase 产品，请将 `google-services` 插件添加到 Gradle 文件中。

> 注意，所有 Firebase SDK 都使用 google-services Gradle 插件（`google-services`），但该插件与 Google Play 服务没有任何关系）。

- **方法 1**：使用 Gradle 的 `buildscript` 块添加插件。Google 文档 与 Firebase 中文文档 均使用该方法。

1. 修改根级（项目级）Gradle 文件 (`build.gradle`)，在 buildscript 块中添加规则，以导入 Google 服务 Gradle 插件。**请确认您添加了 Google 的 Maven 代码库。**

```
buildscript {

repositories {
// 请检查是否有此行（如果没有，请添加）：
google() // Google's Maven repository
}

dependencies {
// ...

// 请添加以下配置：
classpath 'com.google.gms:google-services:4.3.13' // Google Services plugin
}
}

allprojects {
// ...

repositories {
// 请检查是否有此行（如果没有，请添加）：
google() // Google's Maven repository
// ...
}
}
```

2. 在您的模块（应用级）Gradle 文件（通常是 app/build.gradle）中，应用 Google 服务 Gradle 插件：

```
apply plugin: 'com.android.application'
// Add the following line:
apply plugin: 'com.google.gms.google-services' // Google Services plugin

android {
// ...
}
```

- **方法 2**：使用 Gradle 的 plugins DSL。如果您熟悉 Gradle 的 plugins DSL 语法，且已在项目中使用，可使用该方法。

1. 默认情况下 plugins DSL 仅支持已发布在 Gradle Plugin Portal 的核心插件。因为 google-services 插件不在该仓库中，所以您需要先添加 Google 的插件仓库地址。

   在根级（项目级）Gradle 设置文件（settings.gradle）中声明 Google 的 Maven 代码库。

```
pluginManagement {
// （可选）在声明仓库时即可引入 google-services plugin 并声明版本
//plugins {
// id 'com.google.gms.google-services' version '4.3.13'
//}
repositories {
google()
gradlePluginPortal()
}
}
rootProject.name='exampleProject'
include ':app'
```

2. 在您的模块（应用级）Gradle 文件（通常是 app/build.gradle）中，应用 Google 服务 Gradle 插件google-services 插件，并声明版本：

```
plugins {
id 'com.google.gms.google-services' version '4.3.13'
}
```

   注意：如果在 settings.gradle 的 pluginManagement {} 块中已使用 plugins DSL 引入了 google-services 插件，并且声明了版本，则可直接在模块（应用级）app/build.gradle 文件中应用插件，无需再声明版本：

```
plugins {
id 'com.google.gms.google-services'
}
```

第 4 步：将 Firebase SDK 添加到您的 Android 应用

1. 在模块（应用级）Gradle 文件（通常为 app/build.gradle）中声明依赖项。

- 如果您使用 Gradle 5.0+，推荐配合使用 Firebase Android BoM 声明您要在应用中使用的 Firebase 产品（可用的库）的依赖项（此处为 firebase-messaging）。

```
dependencies {
// ...

// 导入 Firebase BoM。后续声明 Firebase 库依赖项时，您无需指定具体的库版本。
implementation platform('com.google.firebase:firebase-bom:30.1.0')

// firebase-messaging
implementation 'com.google.firebase:firebase-messaging'
// 根据自身需求，导入 其他 Firebase 产品
// implementation 'com.google.firebase:firebase-auth'
// implementation 'com.google.firebase:firebase-firestore'

// Google Play Service
implementation 'com.google.android.gms:play-services-gcm:17.0.0'
}
```

  使用 Firebase Android BoM 可确保您的应用使用的多个 Firebase 产品的 Android 库之间始终保持兼容。注意，BOM 仅管理 Firebase 库版本之间的兼容关系，不会将任何 Firebase 库自动添加到您的应用。如需添加库，请为其添加单独的依赖项行。

- 如果不想使用 Firebase Android BoM，您可以直接指定依赖项，并声明依赖项版本。

```
dependencies {
// firebase-messaging
implementation 'com.google.firebase:firebase-messaging:22.0.0'
// Google Play Service
implementation 'com.google.android.gms:play-services-gcm:17.0.0'
}
```

> 提示
>
> **为什么 firebase-messaging 需要 Google Play 服务?**
>
> - 详见 Firebase 中文文档 Firebase Android SDK 对 Google Play 服务的依赖 。
> - 您可以自行确定所需要的 Google Play 服务的版本。

2. 同步您的应用以确保所有依赖项都具有所需的版本。

第 5 步：修改 Android 的应用清单（AndroidManifest.xml）

在主工程中的 `AndroidManifest.xml` 文件下增加如下两段配置：

- 阻止 FCM 自动初始化。

```
<meta-data
android:name="firebase_messaging_auto_init_enabled"
android:value="false" />
<meta-data
android:name="firebase_analytics_collection_enabled"
android:value="false" />
```

- 扩展 FirebaseMessagingService 的服务。除了接收通知外，如果您还希望在后台应用中进行消息处理，则必须添加此服务。如需在前台应用中接收通知、接收数据载荷以及发送上行消息等，您必须添加此扩展服务。

```
<service
android:name="io.rong.push.platform.google.RongFirebaseMessagingService"
android:stopWithTask="false"
android:exported="false">
<intent-filter>
<action android:name="com.google.firebase.MESSAGING_EVENT"/>
</intent-filter>
</service>
```

如需了解更多细节，可参考 Firebase 中文文档 设置 Firebase Cloud Messaging 客户端应用 (Android) 。

至此，您已经成功将 Firebase 集成到 Android 项目中。

请继续完成后续步骤，**在控制台授权 FCM 请求**。

在控制台授权 FCM 请求

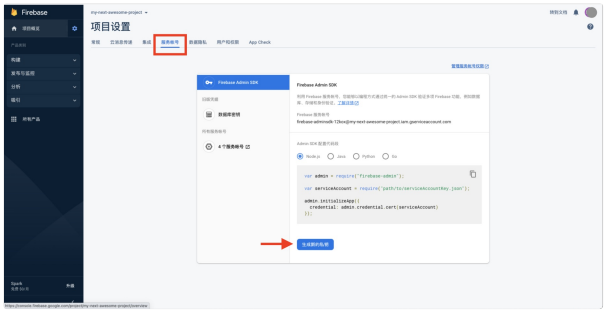
从融云发送到 FCM 的请求必须经过授权。因此，您需要向融云提供 FCM 项目相关的服务账号授权凭证。您需要手动将该凭证上传到控制台。

第 1 步：获取 Google 服务账号凭证

Firebase 项目支持 Google 服务帐号 。您需要从您的 Google 服务帐号获取凭据，然后授权融云服务器这些帐号凭据调用 Firebase 服务器 API。

如需对服务帐号进行身份验证并授予其访问 Firebase 服务的权限，您必须生成 JSON 格式的私钥文件。**如需为您的服务帐号生成 JSON 格式的私钥文件，请执行以下操作：**

1. 在 Firebase 控制台中，打开设置 > **服务帐号**。
2. 点击生成新的私钥，然后点击生成密钥进行确认。
3. 妥善存储包含密钥的 JSON 文件。稍后需要在控制台上传该文件。



第 2 步：在控制台上传 FCM 凭证

前往控制台，在**应用标识 > Android > Google（FCM）** 标签页中，填写上一步在获取的凭证。

- **选择鉴权方式**：
  - **证书（推荐）**：上传从您的服务帐号生成的私钥 JSON 文件。
  - **API 密钥**（使用旧版 Server Key 授权方式和旧版 FCM API）：填入从您的服务帐号获取的服务器密钥。

> ⚠ 警告
>
> 如果您的项目中配置的鉴权方式中为 **API** 密钥，请注意自 2023 年 6 月 20 日起，使用 FCM XMPP 和 HTTP 旧版 API 发送消息（包括上游消息）已被 FCM 官方弃用，并将于 2024 年 6 月移除，融云的推送服务后台届时将同步移除该能力。因此，您需要尽快迁移到使用证书（FCM 的 JSON 格式私钥文件）的鉴权方式，详细方法请参考上文获取 Google 服务账号凭证。

- **推送方式**（详见 Firebase 中文文档 FCM 消息类型）：
  - **透传消息方式**：由 Android OS 直接在通知面板弹出通知。融云仅下发对应 FCM「数据消息」类型的数据。
  - **通知消息方式**：IMLib SDK 接收到透传数据后，进行数据解析并弹出通知。开发者也可以自定义处理。融云下发包含 FCM「数据消息」和「通知消息」的数据。

- **intent**：自定义用户点击通知相关联的操作，对应 FCM 的 `click_action` 字段。如果指定，当用户单击通知时，默认将启动匹配该 Intent 的 Activity。最好将应用的软件包名称用作前缀，以确保唯一性。

  例如，在控制台指定 Intent 为：

  `com.yourapp.demo.ExampleActivity`

  在您 App 的 `AndroidManifest.xml` 中需要指定可以匹配该 Intent 的 Activity。

```
<activity android: name = ".ExampleActivity">
<intent-filter>
<action android: name = "com.yourapp.demo.ExampleActivity" />
...
</intent-filter>
</activity>
```

- **优先级**（详见 Firebase 中文文档 设置消息的优先级）
  - **普通**：应用在前台运行时，普通优先级消息会被立即传递。当应用在后台运行时，消息传递可能会延迟。如果是对时间不太敏感的消息（例如新电子邮件通知、使界面保持同步或在后台同步应用数据），建议您选择普通传递优先级。
  - **高**：即使设备处于低电耗模式，FCM 也会立即尝试传递高优先级消息。 高优先级消息适用于对时间敏感的用户可见内容。
- **推送通知标题**：在调用服务端 API 接口 `/push.json`、`/push/user.json`、 `/push/custom.json` 接口推送通知时，如未传入通知栏标题，则使用下方设置的“推送通知标题”。从服务端发消息时，如果发送用户 ID 在融云服务端没有用户名，也会使用此“推送通知标题”。一般情况下客户端发送消息转 Push 时不显示此标题。
- **推送通道类型**：GCM 推送融云默认使用的是默认推送通道，如需要使用私有推送通道，可选择私有通道并设置 Channel ID 即可。

启用 FCM 推送服务

1. 在 SDK init 之前，配置使用 FCM 推送。

```
PushConfig config = new PushConfig.Builder()
.enableFCM(true)
.build();
RongPushClient.setPushConfig(config);
```

2. （可选）启用 FCM 推送后，如需重新启用 Analytics 数据收集，请调用 `FirebaseAnalytics` 类的 `setAnalyticsCollectionEnabled()` 方法。例如：

```
setAnalyticsCollectionEnabled(true);
```

> 提示
>
> 如果不需要接收推送，可以通过设置 SDK 的初始化配置中的 `enablePush` 参数为 `false`，向融云服务申请禁用推送服务（当前设备）。您也可以在断开连接时设置不接收推送（当前设备）。

测试 FCM 推送

在完成上述步骤之后，可直接测试推送是否集成成功。

设备条件

建议直接使用在海外正式发售的 Android 设备测试 FCM 推送通知的接收。针对测试设备的详细要求如下：

> 提示
>
> 1. 测试设备必须使用真机。模拟器收不到远程推送。
> 2. 测试设备必须支持 Google GMS 服务（Google Mobile Services）。您可以直接使用在海外正式发售的 Android 设备。注意，在中国大陆地区发售的设备一般均未预装 GMS 服务。部分品牌的设备因无法获得授权，无法自行安装 GMS 服务。

网络条件

建议在模拟的海外网络环境下进行测试。详细条件如下：

> 提示
>
> 1. 如果在中国大陆地区进行测试，要求测试设备必须使用国外 IP 地址与融云建立 IM 连接，并使用海外 IP 访问融云服务。如果使用国内 IP，融云服务端认为该设备处于国内，且不会启用 FCM 通道。
> 2. 测试设备所处的网络环境必须可以正常访问 Google 网络服务。否则即使触发 FCM 推送，该设备也无法从 Google 的 FCM 服务收取推送。

测试步骤

假设在开发环境下进行测试。App 使用测试环境的 App Key。

具体步骤如下：

1. APP 连接融云成功之后杀掉 APP 进程。
2. 访问控制台的 IM Server API 调试 页面，切换到 App 的开发环境，找到**消息 > 消息服务 > 发送单聊消息**，直接发送一条单聊消息内容。
3. 查看手机是否收到本 APP 的推送。

故障排除

- 如果测试使用的 App 属于融云的海外数据中心（北美或新加坡），请检查是否已在 SDK 初始化之前执行 `setServerInfo` 设置海外数据中心的导航域名。详见海外数据中心。
- 从 Google FCM 服务后台直接发送消息到设备。如果未收到，请先根据上述文档自查是否正确集成。
- 检查 App 是否被设备为后台受限应用。在 Android P 或更高版本 上，FCM 将不会向被用户添加了后台限制（例如，通过“设置”->“应用和通知”->“[appname]”->“电池”实施后台限制）的应用发送推送通知。详见 Android 开发者文档 后台受限应用 。
- 不要频繁发同一内容的消息到同一台手机，有可能会被 FCM 服务端屏蔽导致无法收到推送。
- 检查测试设备是否为国行手机的 ROM。部分品牌已不支持自行安装 GMS 服务。建议您替换为海外发售的设备，或更新测试设备的系统。

如果问题无法解决，可提交工单 并提供您的消息 ID。

自定义推送通知

- **自定义推送通知样式**：仅在控制台 FCM 的消息推送方式为**透传消息方式**时支持。详见自定义推送通知样式。
- **自定义推送通知点击事件**：详见自定义推送通知点击事件。

针对 CallKit SDK 客户的说明

在控制台 FCM 的消息推送方式为**透传消息方式**时，IMKit/IMLib 默认不会为音视频信令消息（呼叫邀请、挂断等）弹出通知。可通过以下方式解决：

IM SDK 接收的 FCM 透传消息会拉起应用，可在 Application 的 `onCreate` 方法中即调用 IM 的连接方法。连接成功后 SDK 会主动弹出通知。

针对 CallLib SDK 客户的说明

在控制台 FCM 的消息推送方式为**透传消息方式**时，IMKit/IMLib 默认不会为音视频信令（呼叫邀请、挂断等）弹出通知。您可以通过**自定义推送通知样式**，重写 `PushEventListener` 的 `preNotificationMessageArrived` 方法，在该方法自行实现通知弹出逻辑。

混淆脚本

```
-keep public class com.google.firebase.* {*;}
```

内容审核概述 内容审核概述

更新时间:2024-08-30

即时通讯支持对 IM 内容进行审核。

- 即时通讯（IM）服务已内置敏感词机制。注意，敏感词机制仅是一种基础保护机制，且仅限于文本内容（默认最多 50 个敏感词），不可替代专业内容审核服务。
- 融云的内容审核服务产品中的 IM 审核服务，可为 IM 内容提供全面的保障与支持，支持审核文本、图片、语音片段、小视频，精准识别敏感信息。
- 如需自行实现审核或对接第三方审核服务，可以使用**消息回调服务**。

如果消息因被判定违规导致无法下发收件人，默认情况下消息发送者不会收到通知。如果 App 希望通知消息发送者消息已被拦截，可提交工单开通**含敏感词消息屏蔽状态回调发送端**，并在客户端设置监听（要求 Android/iOS SDK 版本 ≧ 5.1.4，Web ≧ 5.0.2）。详见敏感信息拦截回调。

敏感词机制

> 提示
> - 客户端不提供针对该功能的管理接口，仅提供回调接口，可在消息被判定为不下发时通知消息发送方。详见敏感信息拦截回调。
> - 如需审核文本（支持语义检测）、图片、语音片段、小视频，建议使用融云提供的内容审核服务产品。

敏感词机制是一种基础保护机制，仅支持对文本消息内容中的敏感词进行识别与过滤。对命中敏感词的消息，您可以选择进行屏蔽该消息（不会下发给接收方），或按指定规则替换消息中的敏感词后再进行下发。

目前支持的敏感词过滤语言包括：中文、英文、日语、德语、俄语、韩语、阿拉伯语。

您可以通过以下方式管理 App Key 下开发环境或生产环境的敏感词：

功能描述	客户端 API	融云服务端 API	控制台
添加敏感词，支持设置替换内容	不提供该 API	添加敏感词	敏感词设置页面
移除敏感词	不提供该 API	移除敏感词	敏感词设置页面
批量移除敏感词	不提供该 API	批量移除敏感词	敏感词设置页面
获取敏感词列表，支持获取设置的替换内容	不提供该 API	获取敏感词列表	敏感词设置页面

默认行为

- 默认最多设置 50 个敏感词。
- 默认仅针对从客户端 SDK 发送的消息生效。
- 默认仅支持识别官方内置的文本消息类型（消息标识为 `RC:TxtMsg`）中的敏感词。支持单聊、群聊、聊天室、超级群会话。超级群中文本消息修改后的内容默认也会敏感词识别、拦截或过滤。

调整配置

- **IM 旗舰版**或 **IM 尊享版**可以在控制台 IM 服务管理 页面的**扩展服务**标签下自行调整敏感词上限数。具体功能与费用以融云官方价格说明页面及计费说明文档为准。
- 如果您对使用服务端 API 发送的消息进行敏感词过滤，可以在控制台的免费基础功能界面打开 **Server API 发送消息过滤敏感词**开关。
- 如果您需要对自定义消息类型启用敏感词机制，可以在敏感词设置页面点击**设置自定义消息**。提供自定义消息的消息类型的 `ObjectName`，及该消息类型下内容（Content）JSON 结构中对应的键值 Key，即可对该 Key 所对应的 Value 值进行敏感词过滤处理。

IM 内容审核服务

> 提示
> 客户端不提供针对该功能的管理接口，仅提供回调接口，可在消息被判定为不下发时通知消息发送方。详见敏感信息拦截回调。

如果您希望全面审核 IM 内容，可以使用融云的内容审核服务产品，该产品提供 **IM 审核**服务与**音视频审核**服务。

IM 审核针对即时通讯业务，具体可提供以下能力：

- 审核文本内容
- 审核图片
- 审核语音片段
- 审核小视频
- 审核自定义消息类型（需要提交工单申请）
- 审核超级群业务中的消息修改
- 从控制台查看审核报告
- 从控制台查询 IM 审核记录
- 审核结果回调

您可以在控制台的 IM & 音视频审核 页面开通 **IM 审核**服务，配置接收审核结果回调的地址。详见服务端文档审核结果回调。

IM 内容审核计费

内容审核服务为付费服务，开发环境可免费体验，生产环境下需预存才能使用服务。具体计费说明详见资费标准 · IM 审核 。

消息回调服务

如果您希望对接自己的审核系统或其他第三方内容审核服务，可以使用消息回调服务 。

消息回调服务（原模版路由）提供一种消息过滤机制。您可以根据发送用户 ID、接收用户 ID、消息类型、会话类型等参数，将相应的消息同步到您指定的服务器。超级群业务中，修改消息内容、更新消息扩展也支持通过消息回调同步到您指定的服务器。

消息同步到您指定的服务器后，可以使用您自己的审核系统执行内容审核，也可以对接其他第三方审核系统。融云服务端会根据您应用服务器返回的响应结果，决定是否将消息下发、是否替换消息中的内容，以及如何进行内容替换。

您可以通过控制台的消息回调服务 页面管理 App Key 下开发环境或生产环境的消息回调服务状态和路由规则。

关于如何创建路由规则，以及回调参数的具体说明，请参见消息回调服务 文档。

消息回调服务计费

费用以融云官方价格说明 页面及计费说明 文档为准。

敏感信息拦截回调 敏感信息拦截回调

更新时间:2024-08-30

> SDK 从 5.1.4 版本开始支持敏感信息拦截回调。

默认情况下，消息发送方无法感知消息是否已被融云审核服务拦截。如果 App 希望在消息因触发审核规则而无法下发时通知消息发送方，可开通使用敏感信息拦截回调服务。

融云的内容审核服务（包括消息敏感词、IM 审核服务、消息回调服务），可能在以下情况下拦截消息：

- 文本消息内容命中了融云内置的消息敏感词，导致消息不下发给接收方。
- 文本消息内容命中了您自定义的消息敏感词（屏蔽敏感词），导致消息不下发给接收方。
- 消息命中了 **IM 审核服务**，或**消息回调服务**设置的审核规则，导致消息不下发给接收方。

开通服务

如有需求，请提交工单，申请开通**含敏感词消息屏蔽状态回调发送端**。客户端 SDK 5.1.4 及之后版本支持该回调服务。

敏感信息拦截监听器说明

在发出的消息被拦截时，SDK 会触发 `MessageBlockListener` 的以下方法：

```
void onMessageBlock(BlockedMessageInfo info)
```

`BlockedMessageInfo` 里包含了被拦截消息的相关信息，您可以通过下表列出的方法获取：

方法名称	说明
getConversationType（）	获取被拦截消息所在会话的会话类型
getTargetId()	获取被拦截消息所在的会话 Id
getChannelId()	获取被拦截消息所在的超级群频道 ID。要求 Android 端 SDK ≧ 5.2.4
getBlockMsgUId()	获取被拦截消息的唯一 Id
getType()	获取消息被拦截的原因，详见下方 `MessageBlockType` **说明**。
getExtra()	获取被拦截消息的附加信息。要求 Android 端 SDK 版本 ≧ 5.2.5（例外：因历史遗留问题，在 5.2.3.2 及之后的 **5.2.3.x** 系列的维护版本上也可用）。
getSourceType()	获取被拦截的超级群消息的源类型。`0`：原始消息触发了拦截（默认）。`1`：消息扩展触发了拦截。`2`：修改消息后的消息内容触发了拦截。Android 端 SDK 5.2.5 及以后版本支持（仅支持超级群）。
getSourceContent()	获取被拦截的超级群消息或扩展的内容 JSON 字符串。`sourceType` 字段为 1 时表示扩展内容。`sourceType` 为 2 时表示修改后的消息内容。详见下方 `sourceContent` **说明**。Android 端 SDK 5.2.5 及以后版本支持（仅支持超级群）。

- `MessageBlockType` **说明**

```
public enum MessageBlockType {

/**
* 未知类型
*/
UNKNOWN(0),

/**
* 全局敏感词：命中了融云内置的全局敏感词
*/
BLOCK_GLOBAL(1),

/**
* 自定义敏感词拦截：命中了客户在融云自定义的敏感词
*/
BLOCK_CUSTOM(2),

/**
* 第三方审核拦截：命中了第三方（数美）或消息回调服务（原模板路由服务）决定不下发的状态
*/
BLOCK_THIRD_PATY(3);
}
```

- `sourceContent` **说明**

  - `sourceType` 为 `0` 时，sourceContent 为空。
  - `sourceType` 为 `1` 时，sourceContent 是消息扩展内容，示例 {"mid":"xxx-xxx-xxx-xxx","put":{"key":"敏感词"}}。mid 为通知信息的 ID。
  - `sourceType` 为 `2` 时，sourceContent 是修改后的消息内容，示例 {"content":"含有敏感信息的文字"}。内置消息类型的消息内容格式消息类型概述。

设置敏感信息拦截监听器

> 提示
>
> 该接口在 RongIMClient 中。

您可以通过下面的方法设置敏感词拦截监听器，监听到被拦截的消息以及拦截原因。

```
RongIMClient.getInstance().setMessageBlockListener(listener)
```

IMLib 2.X 升级到 5.X IMLib 2.X 升级到 5.X

更新时间:2024-08-30

本文描述 IMLib SDK 从 2.X 到 5.X 版本的升级步骤。

5.x 相较于 2.x 版本，主要有以下变更：

- IMLib 拆分
- 连接接口变更
- 删除了部分废弃接口

IMLib 拆分说明

5.x 版本对 IMLib SDK 进行了拆分，拆分成如下六个模块：

模块名称	功能说明
LibCore	核心通讯模块
chatroom	聊天室
customservice	客服
disscussion	讨论组
location	实时位置
publicservice	公众号

5.x 版本开始会提供两种形式的 SDK：完整包和拆分包

- 如果使用完整包的话，直接正常升级就行。
- 如果使用拆分包，首先 **必需依赖 LibCore 库**，然后按需集成即可。比如您除了核心模块，还使用到聊天室功能，那让项目同时依赖 LibCore 模块和 chatroom 模块，依次类推使用到哪个模块就需要依赖相关模块。

连接接口变更

连接接口 1

```
connect(final String token, final RongIMClient.ConnectCallback connectCallback)
```

用户调用接口之后，如果因为网络原因暂时连接不上，SDK 会一直尝试重连，直到连接成功或者出现 SDK 无法处理的错误（如 token 无效，用户封禁等）。

```
RongIMClient.connect(token, new RongIMClient.ConnectCallback() {
@Override
public void onDatabaseOpened(RongIMClient.DatabaseOpenStatus code) {
//消息数据库打开，可以进入到主页面
}

@Override
public void onSuccess(String s) {
//连接成功
}

@Override
public void onError(RongIMClient.ConnectionErrorCode errorCode) {
if(errorCode.equals(RongIMClient.ConnectionErrorCode.RC_CONN_TOKEN_INCORRECT)) {
//从 APP 服务获取新 token，并重连
}else {
//无法连接到 IM 服务器，请根据相应的错误码作出对应处理
}
}
})
```

连接接口 2

```
connect(final String token, final int timeLimit, final ConnectCallback connectCallback)
```

用户调用接口之后，SDK 会在 timeLimit 秒内尝试连接，超过时间将会返回超时并停止连接，timeLimit <= 0 行为和没有 timeLimit 的接口一样。

```
RongIMClient.connect(token,5, new RongIMClient.ConnectCallback() {
@Override
public void onDatabaseOpened(RongIMClient.DatabaseOpenStatus code) {
//消息数据库打开，可以进入到主页面
}

@Override
public void onSuccess(String s) {
//连接成功
}

@Override
public void onError(RongIMClient.ConnectionErrorCode errorCode) {
if(errorCode.equals(RongIMClient.ConnectionErrorCode.RC_CONN_TOKEN_INCORRECT)) {
//从 APP 服务获取新 token，并重连
} else if (errorCode.equals(RongIMClient.ConnectionErrorCode.RC_CONNECT_TIMEOUT)) {
//连接超时，弹出提示，可以引导用户等待网络正常的时候再次点击进行连接
} else {
//无法连接 IM 服务器，请根据相应的错误码作出对应处理
}
}
})
```

错误回调变更

onError 回调参数以及回调机制变化如下：

版本	参数类型	回调时机	是否继续重连
2.x 版本	RongIMClient.ErrorCode	连接过程中发生的任何异常都会回调	业务错误时不再重连，其它情况 SDK 会继续重连。
4.x & 5.x 版本	RongIMClient.ConnectionErrorCode	连接过程出现业务错误时回调	否

连接错误信息说明

ConnectionErrorCode 说明如下：

```
31004：Token 无效。
处理方案：从 APP 服务器获取新的 token ，再调用 connect 接口进行连接。

31010： 用户被踢下线。
处理法案：退回到登录页面，给用户提示被踢掉线。

31023：用户在其它设备上登录。
处理方案：退回到登录页面，给用户提示其他设备登录了当前账号。

31009：用户被封禁。
处理方案：退回到登录页面，给用户提示被封禁。

34006：自动重连超时(发生在 timeLimit 为有效值并且网络极差的情况下)。
处理方案：重新调用 connect 接口进行连接。

31008：Appkey 被封禁。
处理方案：请检查您使用的 AppKey 是否被封禁或已删除。

33001：SDK 没有初始化。
处理方案：在使用 SDK 任何功能之前，必须先 Init。

33003：开发者接口调用时传入的参数错误。
处理方案：请检查接口调用时传入的参数类型和值。

33002：数据库错误。
处理方案：检查用户 userId 是否包含特殊字符，SDK userId支持大小写英文字母与数字的组合，最大长度 64 字节。
```

5. 连接错误解决方案：

1. connect() 的 onError 回调中判断是否是 token 无效和连接超时。

2. 连接状态监听中判断连接状态是否是 token 无效、踢掉线、封禁、自动重连超时等业务错误，并按照上述处理方案处理。

```
//连接状态监听设置
//其中 this 最好为单例类，以此保证在整个 APP 生命周期，该类都能够检测到 SDK 连接状态的变更
RongIMClient.setConnectionStatusListener(this);

public void onChanged(ConnectionStatus connectionStatus) {
if (connectionStatus.equals(ConnectionStatus.KICKED_OFFLINE_BY_OTHER_CLIENT)) {
//当前用户账号在其他端登录，请提示用户并做出对应处理
} else if (connectionStatus.equals(ConnectionStatus.CONN_USER_BLOCKED)) {
//用户被封禁，请提示用户并做出对应处理
}
}
```

移除接口说明

4.0.0 版本开始移除了以下废弃方法：

接口名称	替代方法
setUserPolicy()	无
List<> getConversationList()	getConversationList(ResultCallback)

接口名称	替代方法
List<> getConversationList(Conversation.ConversationType...)	getConversationList(ResultCallback)
Conversation getConversation(Conversation.ConversationType , String)	getConversation(final Conversation.ConversationType , String , ResultCallback<> callback)
boolean removeConversation()	removeConversation(final Conversation.ConversationType, String, ResultCallback<> callback)
boolean setConversationToTop()	setConversationToTop(final Conversation.ConversationType, String, boolean, ResultCallback<> callback)
int getTotalUnreadCount()	getTotalUnreadCount(ResultCallback<> callback)
int getTotalUnreadCount(ConversationType type)	getUnreadCount(Conversation.ConversationType, String, ResultCallback)
int getUnreadCount(Conversation.ConversationType…)	getUnreadCount(final Conversation.ConversationType[], ResultCallback<> callback)
List<> getLatestMessages(Conversation.ConversationType, String, int)	getLatestMessages(Conversation.ConversationType, String, int, ResultCallback)
List<> getHistoryMessagesByMessageId(Conversation.ConversationType, String, int, int)	getHistoryMessages(Conversation.ConversationType, String, int, int, ResultCallback)
List<Message> getHistoryMessagesByObjectNames(Conversation.ConversationType, String, List<>, long, int, RongCommonDefine.GetMessageDirection)	getHistoryMessagesByObjectNamesSync(final Conversation.ConversationType conversationType, final String targetId,List<String> objectNames, final long timestamp, final int count, final RongCommonDefine.GetMessageDirection direction)
getUserOnlineStatus()	无
setSubscribeStatusListener()	无
setUserOnlineStatus()	无
getHistoryMessagesOneWay()	无
boolean deleteMessages(int[])	deleteMessages(int[], ResultCallback)
boolean clearMessages(Conversation.ConversationType , String)	clearMessages(Conversation.ConversationType, String, ResultCallback)
boolean clearMessagesUnreadStatus(Conversation.ConversationType, String)	clearMessagesUnreadStatus(Conversation.ConversationType, String, ResultCallback)
boolean setMessageExtra(int messageId, String value)	setMessageExtra(int, String, ResultCallback)
boolean setMessageSentStatus(int, Message.SentStatus)	setMessageReceivedStatus(int, Message.ReceivedStatus, ResultCallback)
void setMessageSentStatus(final int, Message.SentStatus, ResultCallback<>)	setMessageSentStatus(Message, ResultCallback)
String getTextMessageDraft()	getTextMessageDraft(Conversation.ConversationType, String, ResultCallback)
boolean saveTextMessageDraft(Conversation.ConversationType, String, String)	saveTextMessageDraft(Conversation.ConversationType, String, String, ResultCallback)
boolean clearTextMessageDraft(Conversation.ConversationType, String)	clearTextMessageDraft(Conversation.ConversationType, String, ResultCallback)
void insertMessage(final Conversation.ConversationType, String, String, MessageContent, long , ResultCallback<>)	insertIncomingMessage(Conversation.ConversationType, String, String, Message.ReceivedStatus, MessageContent, long, ResultCallback) insertOutgoingMessage(Conversation.ConversationType, String, Message.SentStatus, MessageContent, long, ResultCallback)
void insertMessage(final Conversation.ConversationType, String, String, MessageContent, ResultCallback<>)	同上
Message sendMessage(Conversation.ConversationType, String, MessageContent, String, String, SendMessageCallback)	sendMessage(Conversation.ConversationType, String, MessageContent, String, String, IRongCallback.ISendMessageCallback)
void sendMessage(Conversation.ConversationType , String, MessageContent, String, String, SendMessageCallback, ResultCallback<Message> resultCallback)	sendMessage(Message, String, String, IRongCallback.ISendMessageCallback)
Message sendMessage(Message, pushContent, String, SendMessageCallback)	sendMessage(Message, String, String, IRongCallback.ISendMessageCallback)
syncGroup()	无
joinGroup()	无
quitGroup()	无
clearConversations()	clearConversations(ResultCallback, Conversation.ConversationType...)
syncUserData()	无
updateRealTimeLocationStatus()	无
RecallMessageListener()	无
setRecallMessageListener()	无
setPushNotificationListener()	无
syncConversationNotificationStatus()	无

旧版本快速兼容方案

说明：以下是旧版本快速兼容方案，但是我们依然建议您参考上面的详细建议进行处理；如果您是直接使用 4.0.0 新版本建议参见上面的详细处理流程。

1. 将 connect() 的回调更改为新的 callback。
2. 在 connect() 的 onError() 回调中，判断 ConnectionErrorCode 为 RC_CONN_TOKEN_INCORRECT()时，将 原 onTokenIncorrect() 中的处理逻辑拷贝过来。

```
RongIMClient.connect(token, new RongIMClient.ConnectCallback() {
@Override
public void onDatabaseOpened(RongIMClient.DatabaseOpenStatus code) {
//如果消息数据库打开，可以进入到主页面
}

@Override
public void onSuccess(String s) {
//连接成功
}

@Override
public void onError(RongIMClient.ConnectionErrorCode e) {
if(e.equals(RongIMClient.ConnectionErrorCode.RC_CONN_TOKEN_INCORRECT)) {
//将旧版本 token 无效的回调处理代码写到这里
//从 APP 服务获取新 token，并重连
}else {
//无法连接 im 服务器，请根据相应的错误码作出对应处理
}
}
});
```

常见问题

1.为什么一部分无法重连错误码的处理逻辑并没有在示例代码中写明？

都是开发阶段的问题，不需要代码兼容处理。

`31008`：Appkey 被封禁

当发生这个问题的时候，您可以自行在控制台查看您的 appkey 使用状态，大多情况是 appkey 被自行删除或者欠费。

`33001`：SDK 没有初始化

这个错误只会发生在开发阶段，只要您保证先 init 后 connect 就不会有这个问题。

`33003`：开发者接口调用时传入的参数错误

这个错误只会发生在开发阶段，很可能是传入的 token 为空，只要保证 connect 传入正确合法的 token 就不会有这个问题。

`33002`：数据库错误

这个问题只会发生在开发阶段，很可能是您的用户 id 体系和我们 SDK 的不一致，一般该情况很少发生。

2.旧版本连接过程中一旦出现中间错误码就会立即触发 error 回调，新版本中间错误码不会触发 error 回调了，可能会等很长时间没有任何回调，这要怎么处理？

以下的建议，择一选取即可。

建议1.

用户第一次登录，设置 timeLimit 为有效值，网络极差情况下超时回调 error。

用户后续登录，调用没有 timeLimit 的接口，SDK 就会保持旧版本的自动重连。

建议2.

设置 SDK 的连接状态监听，APP 自行做超时的记录，如果超时了，APP 可以自动断开连接再调用 connect 进行连接。

IMLib 4.X 升级到 5.X IMLib 4.X 升级到 5.X

更新时间:2024-08-30

本文描述 IMLib SDK 从 4.X 到 5.X 版本的升级步骤。

升级说明

5.x 相较于 4.x 版本，对即时通讯基础组件 IMLib 进行了拆分。IMLib 分成了 6 个模块：

> 提示
>
> LibCore（基础核心功能），chatroom（聊天室），customservice（客服），
>
> discussion（讨论组），location（实时位置），publicserivce（公众号）

5.x 版本会提供两种形式的 SDK，`完整包`和`拆分包`

1. 如果使用`完整包`，直接正常升级就行，从 4.x 能完美升级兼容
2. 如果使用'拆分包'，首先必需依赖 LibCore 模块，其它模块根据业务需求选择集成。比如您使用到聊天室的功能的话，除了依赖 LibCore 模块，还需要让项目依赖 chatroom 模块，依次类推使用到哪个模块就依赖相关模块

常见问题

1.为什么一部分无法重连错误码的处理逻辑并没有在示例代码中写明？

都是开发阶段的问题，不需要代码兼容处理。

`31008`：Appkey 被封禁

当发生这个问题的时候，您可以自行在控制台查看您的 appkey 使用状态，大多情况是 appkey 被自行删除或者欠费。

`33001`：SDK 没有初始化

这个错误只会发生在开发阶段，只要您保证先 init 后 connect 就不会有这个问题。

`33003`：开发者接口调用时传入的参数错误

这个错误只会发生在开发阶段，很可能是传入的 token 为空，只要保证 connect 传入正确合法的 token 就不会有这个问题。

`33002`：数据库错误

这个问题只会发生在开发阶段，很可能是您的用户 id 体系和我们 SDK 的不一致，一般该情况很少发生。

2.旧版本连接过程中一旦出现中间错误码就会立即触发 error 回调，新版本中间错误码不会触发 error 回调了，可能会等很长时间没有任何回调，这要怎么处理？

以下的建议，择一选取即可。

建议1.

用户第一次登录，设置 timeLimit 为有效值，网络极差情况下超时回调 error。

用户后续登录，调用没有 timeLimit 的接口，SDK 就会保持旧版本的自动重连。

建议2.

设置 SDK 的连接状态监听，APP 自行做超时的记录，如果超时了，APP 可以自动断开连接再调用 connect 进行连接。

IMKit 2.X 升级到 5.X IMKit 2.X 升级到 5.X

更新时间:2024-08-30

本文描述 IMKit SDK 从 2.X 到 5.X 版本的升级步骤。

变更说明

IMKit SDK 5.x 版本正式开源，此开源版本基于 MVVM 架构重构完成，因此和 2.x、4.x 版本不兼容，无法平滑升级。

详细变更说明如下：

1. 依赖包更改

Google 2018 IO 大会推出了扩展库 AndroidX，AndroidX 是对 android.support.xxx 包的整理后产物，Google 官方早已明确声明不再维护 support 包。因此融云 IMKit SDK 5.X 及以后版本更改为通过 AndroidX 实现，不再支持 support 包！

> 提示
>
> 如果您应用依赖于 support 包，可参考 AndroidX 迁移文档 将依赖转换为 AndroidX 后，再集成 IMKit SDK 5.X版本。

2. 类路径调整

IMKit SDK 5.X 版本重构了 SDK 的架构，通过 MVVM 框架实现，导致很多类路径发生了更改，下表列出了 5.X 之前版本对外公开类的路径变更：

2.x 、4.x 版本路径	5.x 版本路径
io.rong.imkit.fragment.ConversationListFragment	io.rong.imkit.conversationlist.ConversationListFragment
io.rong.imkit.fragment.ConversationFragment	io.rong.imkit.conversation.ConversationFragment
io.rong.imkit.RongExtension	无法直接替换。请按照 IMKit 5.x 的文档说明重新集成插件。
io.rong.imkit.userInfoCache.RongUserInfoManager	io.rong.imkit.userinfo.RongUserInfoManager
io.rong.imkit.mention.RongMentionManager	io.rong.imkit.feature.mention.RongMentionManager
io.rong.imkit.manager.IUnReadMessageObserver	io.rong.imkit.manager.UnReadMessageManager.IUnReadMessageObserver
io.rong.imkit.RongConfigurationManager	io.rong.imkit.utils.language.RongConfigurationManager

> 关于路径调整的说明
>
> 1. 参照上表，在 AndroidStudio 中 command+shift+R 全局搜索旧的类路径，替换为新路径。
> 2. `io.rong.imkit.RongExtension` 无法用替换的方式升级。请按照 IMKit 5.x 的文档说明重新集成插件。
> 3. 如果深度定制化了 IMKit SDK，部分类路径的调整可能不在上述列表中，升级以后 AndroidStudio 会有红色报错。删除报错类里红色未识别的导入路径，鼠标停留到报错的地方，根据 AndroidStudio 的提示导入新路径即可。如果有多个地方使用了该路径，可以参考第一步里的方法，全局搜索并替换。

3. 集成方式变更

页面跳转时由原先的隐式调用方式更改为显示调用， 因此集成步骤有所简化和更改。您需要先移除旧版本 AndroidManifest.xml 文件中关于 IMKit SDK 的配置，参考官网文档重新集成。

4. RongIM 接口变化

- 连接接口返回值变动。

  connect() 方法返回值由 RongIM 更改为 void.

- 接口名称更改

2.x、4.x 接口	5.x 接口
无	removeOnReceiveMessageListener
无	removeConnectionStatusListener
setSendMessageListener	setMessageInterceptor
sendImageMessage	**sendMediaMessage**
setMaxVoiceDurationg	setMaxVoiceDuration
startConversationList	无
startPublicServiceProfile	无

- 接口迁移。用户信息相关接口由 RongIM 迁移到了单独类 UserDataProvider 中。

2.x、4.x 接口	5.x 接口
RongIM.UserInfoProvider	UserDataProvider.UserInfoProvider
RongIM.GroupInfoProvider	UserDataProvider.GroupInfoProvider
RongIM.GroupUserInfoProvider	UserDataProvider.GroupUserInfoProvider

- RongIM 移除了以下和 UI 无关的接口，可改为调用 IMLib 核心类 RongIMClient 里的同名方法。

移除接口
saveTextMessageDraft
setMessageExtra
setMessageReceivedStatus
setServerInfo
setStatisticDomain
supportResumeBrokenTransfer
searchPublicService
searchPublicServiceByType
subscribePublicService
unsubscribePublicService
downloadMedia

- 删除了以下接口，不再支持。

接口名称
refreshDiscussionCache
createDiscussion
createDiscussionChat
getDiscussion
quitDiscussion
addMemberToDiscussion
recordNotificationEvent
isNotificationQuiteHoursConfigured
getEncryptedSessionStatus
getAllEncryptedConversations
quitEncryptedSession
isRegisted
getHistoryMessages() 废弃方法
disconnect() 废弃方法

- 删除了旧版本里的废弃接口

5. 消息展示模板变更

列表页由 ListView 更改为效率更高的 RecycleView，因此自定义消息的展示模板需要对应调整。

- 将自定义消息的展示模板更改为继承 BaseMessageItemProvider<>, 并实现基类方法。
- 移除展示模板里原先的注解，原注解属性可在新模板构建类里通过调用基类成员 mConfig 的各个方法进行配置

```
import android.content.Context;
import android.text.Spannable;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

import java.util.List;

import io.rong.imkit.R;
import io.rong.imkit.model.UiMessage;
import io.rong.imkit.widget.adapter.IViewProviderListener;
import io.rong.imkit.widget.adapter.ViewHolder;
import io.rong.imlib.model.MessageContent;

public class CustomMessageItemProvider extends BaseMessageItemProvider<CustomMessage> {

public CustomMessageItemProvider(){
mConfig.showSummaryWithName = false; // 配置模板基本属性。
}
/**
* 创建 ViewHolder
* @param parent 父 ViewGroup
* @param viewType 视图类型
* @return ViewHolder
*/
@Override
protected ViewHolder onCreateMessageContentViewHolder(ViewGroup parent, int viewType) {
View rootView = LayoutInflater.from(parent.getContext()).inflate(R.layout.custom_message_item, parent, false);
return new ViewHolder(parent.getContext(), rootView);
}

/**
* 设置消息视图里各 view 的值
* @param holder ViewHolder
* @param parentHolder 父布局的 ViewHolder
* @param customMessage 此展示模板对应的消息
* @param uiMessage {@link UiMessage}
* @param position 消息位置
* @param list 列表
* @param listener ViewModel 的点击事件监听器。如果某个子 view 的点击事件需要 ViewModel 处理，可通过此监听器回调。
*/
@Override
protected void bindMessageContentViewHolder(ViewHolder holder, ViewHolder parentHolder, CustomMessage customMessage, UiMessage uiMessage, int position, List<UiMessage>
list, IViewProviderListener<UiMessage> listener) {

}

/**
* @param holder ViewHolder
* @param customMessage 自定义消息
* @param uiMessage {@link UiMessage}
* @param position 位置
* @param list 列表数据
* @param listener ViewModel 的点击事件监听器。如果某个子 view 的点击事件需要 ViewModel 处理，可通过此监听器回调。
* @return 点击事件是否被消费
*/
@Override
protected boolean onItemClick(ViewHolder holder, CustomMessage customMessage, UiMessage uiMessage, int position, List<UiMessage> list, IViewProviderListener<UiMessage>
listener) {
return false;
}

/**
* 根据消息内容，判断是否为本模板需要展示的消息类型
*
* @param messageContent 消息内容
* @return 本模板是否处理。
*/
@Override
protected boolean isMessageViewType(MessageContent messageContent) {
return messageContent instanceof CustomMessage ;
}

/**
* 在会话列表页某条会话最后一条消息为该类型消息时，会话里需要展示的内容。
* 比如：图片消息在会话里需要展示为"图片"，那返回对应的字符串资源即可。
* @param context 上下文
* @param customMessage 自定义消息
* @return 会话里需要展示的字符串资源
*/
@Override
public Spannable getSummarySpannable(Context context, CustomMessage customMessage) {
return null;
}
}
```

6. 扩展区域自定义方式变更

输入区域配置方式变更，提供了 DefaultExtensionConfig 类，该类提供了 SDK 默认插件列表和表情列表。可以通过复写此类调整插件的位置或进行增减。

7. 本地通知变更

当应用退到后台后，同一个人发来多条消息时，本地通知由之前的折叠显示方式更改为分条显示。

8. 内部实现变更

- 资源名称变更。IMKit SDK 5.x 版本重新统一了资源名称的命名，如果您自定义了部分资源文件，可参考官网自定义文档，重新替换对应名称的资源。
- 移除了 EventBus 。IMKit 5.x 版本不再依赖于 EventBus 进行事件分发，更改为通过观察者模式实现。如果您应用里使用了旧版本 SDK 里的 EventBus, 升级后需要您从应用层自己引入依赖。

- 移除 RongContext 类。如果您应用里有调用 RongContext 作为上下文使用的地方，请更改为调用您自己的应用上下文。
- 由于更改为 MVVM 框架，原 fragment 里很多业务相关接口被移除，您可参考官网 5.x 版本文档重新进行页面自定义。

2.X 版本升级指导

1. 依赖 Androidx

IMKit 5.x 版本不再支持 support 包，开发者需要参考 AndroidX 迁移文档 将应用的依赖转换为 Androidx.

2. 连接接口变更

连接接口的返回值由 RongIM 更改为 void 类型即可。

3. 适配会话列表 ConversationListFragment

- 删除 ConversationListFragment 集成时对 setUri() 的调用，原 setUri() 的参数改为下面方法配置：

```
RongConfigCenter.conversationListConfig().setDataProcessor(new DataProcessor<Conversation>() {
@Override
public Conversation.ConversationType[] supportedTypes() {
return supportedTypes; //此处返回会话列表支持的会话类型
}

@Override
public List<Conversation> filtered(List<Conversation> data) {
return data;
}

@Override
public boolean isGathered(Conversation.ConversationType type) {
if (type.equals(Conversation.ConversationType.SYSTEM)) {
return true; //需要聚合显示的会话，返回 true.
} else {
return false;
}
}
});
```

SDK 默认支持所有会话类型，都非聚合显示。如果您没有自定义需求，可以不进行上述设置，直接去掉 setUri() 的调用即可。

- 将会话列表 activity 注册到 SDK。

```
RouteUtils.registerActivity(RouteUtils.RongActivityType.ConversationListActivity, CustomConversationListActivity.class);
```

- 更改会话列表页面启动方式，去掉老版本如下代码：

```
RongIM.getInstance().startConversationList(Context , supportedConversation);
```

改为调用如下方法：

```
RouteUtils.routeToConversationListActivity(context,title);
```

4. 适配会话页面 ConversationFragment

- 将会话 activity 注册到 SDK

```
RouteUtils.registerActivity(RouteUtils.RongActivityType.ConversationActivity, CustomConversationListActivity.class);
```

5. 设置用户信息

用户信息相关接口所属类由 RongIM 迁移到 UserDataProvider 类中，删除原用户信息接口指定的父类 'RongIM.'，AndroidStudio 会引导 import 新的路径，根据提示操作即可。

6. 适配新的自定义方式

由于架构的变更，原 fragment 里很多业务相关接口被移除，开发者需要根据原自定义需求参考 5.x 文档自定义部分重新适配。

IMKit 4.X 升级到 5.X

IMKit 4.X 升级到 5.X

更新时间:2024-08-30

本文描述 IMKit SDK 从 4.X 到 5.X 版本的升级步骤。

变更说明

> 提示
>
> IMKit SDK 5.x 版本正式开源，此开源版本基于 MVVM 架构重构完成，因此和 2.x、4.x 版本不兼容，无法平滑升级。

详细变更说明如下：

1. 依赖包更改

Google 2018 IO 大会推出了扩展库 AndroidX，AndroidX 是对 android.support.xxx 包的整理后产物，Google 官方早已明确声明不再维护 support 包。因此融云 IMKit SDK 5.X 及以后版本更改为通过 AndroidX 实现，不再支持 support 包！

> 提示
>
> 如果您应用依赖于 support 包，可参考 AndroidX 迁移文档 将依赖转换为 AndroidX 后，再集成 IMKit SDK 5.X版本。

2. 类路径调整

IMKit SDK 5.X 版本重构了 SDK 的架构，通过 MVVM 框架实现，导致很多类路径发生了更改，下表列出了 5.X 之前版本对外公开类的路径变更：

2.x 、4.x 版本路径	5.x 版本路径
io.rong.imkit.fragment.ConversationListFragment	io.rong.imkit.conversationlist.ConversationListFragment
io.rong.imkit.fragment.ConversationFragment	io.rong.imkit.conversation.ConversationFragment
io.rong.imkit.RongExtension	无法直接替换。请按照 IMKit 5.x 的文档说明重新集成插件。
io.rong.imkit.userInfoCache.RongUserInfoManager	io.rong.imkit.userinfo.RongUserInfoManager
io.rong.imkit.mention.RongMentionManager	io.rong.imkit.feature.mention.RongMentionManager
io.rong.imkit.manager.IUnReadMessageObserver	io.rong.imkit.manager.UnReadMessageManager.IUnReadMessageObserver
io.rong.imkit.RongConfigurationManager	io.rong.imkit.utils.language.RongConfigurationManager

> 关于路径调整的说明
>
> 1. 参照上表，在 AndroidStudio 中 command+shift+R 全局搜索旧的类路径，替换为新路径。
> 2. `io.rong.imkit.RongExtension` 无法用替换的方式升级。请按照 IMKit 5.x 的文档说明重新集成插件。
> 3. 如果深度定制化了 IMKit SDK，部分类路径的调整可能不在上述列表中，升级以后 AndroidStudio 会有红色报错。删除报错类里红色未识别的导入路径，鼠标停留到报错的地方，根据 AndroidStudio 的提示导入新路径即可。如果有多个地方使用了该路径，可以参考第一步里的方法，全局搜索并替换。

3. 集成方式变更

页面跳转时由原先的隐式调用方式更改为显示调用， 因此集成步骤有所简化和更改。您需要先移除旧版本 AndroidManifest.xml 文件中关于 IMKit SDK 的配置，参考官网文档重新集成。

4. RongIM 接口变化

- 连接接口返回值变动。

  connect() 方法返回值由 RongIM 更改为 void.

- 接口名称更改

2.x、4.x 接口	5.x 接口
无	removeOnReceiveMessageListener
无	removeConnectionStatusListener
setSendMessageListener	setMessageInterceptor
sendImageMessage	**sendMediaMessage**
setMaxVoiceDurationg	setMaxVoiceDuration
startConversationList	无
startPublicServiceProfile	无

- **接口迁移。用户信息相关接口由** RongIM **迁移到了单独类** UserDataProvider **中。**

2.x、4.x 接口	5.x 接口
RongIM.UserInfoProvider	UserDataProvider.UserInfoProvider
RongIM.GroupInfoProvider	UserDataProvider.GroupInfoProvider
RongIM.GroupUserInfoProvider	UserDataProvider.GroupUserInfoProvider

- RongIM **移除了以下和** UI **无关的接口，可改为调用 IMLib 核心类 RongIMClient 里的同名方法。**

移除接口
saveTextMessageDraft
setMessageExtra
setMessageReceivedStatus
setServerInfo
setStatisticDomain
supportResumeBrokenTransfer
searchPublicService
searchPublicServiceByType
subscribePublicService
unsubscribePublicService
downloadMedia

- **删除了以下接口，不再支持。**

接口名称
refreshDiscussionCache
createDiscussion
createDiscussionChat
getDiscussion
quitDiscussion
addMemberToDiscussion
recordNotificationEvent
isNotificationQuiteHoursConfigured
getEncryptedSessionStatus
getAllEncryptedConversations
quitEncryptedSession
isRegisted
getHistoryMessages() **废弃方法**
disconnect() **废弃方法**

- **删除了旧版本里的废弃接口**

5. 消息展示模板变更

列表页由 ListView **更改为效率更高的** RecycleView**，因此自定义消息的展示模板需要对应调整。**

- **将自定义消息的展示模板更改为继承** BaseMessageItemProvider<>, **并实现基类方法。**
- **移除展示模板里原先的注解，原注解属性可在新模板构建类里通过调用基类成员** mConfig **的各个方法进行配置**

```
import android.content.Context;
import android.text.Spannable;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

import java.util.List;

import io.rong.imkit.R;
import io.rong.imkit.model.UiMessage;
import io.rong.imkit.widget.adapter.IViewProviderListener;
import io.rong.imkit.widget.adapter.ViewHolder;
import io.rong.imlib.model.MessageContent;

public class CustomMessageItemProvider extends BaseMessageItemProvider<CustomMessage> {

public CustomMessageItemProvider(){
mConfig.showSummaryWithName = false; // 配置模板基本属性。
}
/**
* 创建 ViewHolder
* @param parent 父 ViewGroup
* @param viewType 视图类型
* @return ViewHolder
*/
@Override
protected ViewHolder onCreateMessageContentViewHolder(ViewGroup parent, int viewType) {
View rootView = LayoutInflater.from(parent.getContext()).inflate(R.layout.custom_message_item, parent, false);
return new ViewHolder(parent.getContext(), rootView);
}

/**
* 设置消息视图里各 view 的值
* @param holder ViewHolder
* @param parentHolder 父布局的 ViewHolder
* @param customMessage 此展示模板对应的消息
* @param uiMessage {@link UiMessage}
* @param position 消息位置
* @param list 列表
* @param listener ViewModel 的点击事件监听器。如果某个子 view 的点击事件需要 ViewModel 处理，可通过此监听器回调。
*/
@Override
protected void bindMessageContentViewHolder(ViewHolder holder, ViewHolder parentHolder, CustomMessage customMessage, UiMessage uiMessage, int position, List<UiMessage>
list, IViewProviderListener<UiMessage> listener) {

}

/**
* @param holder ViewHolder
* @param customMessage 自定义消息
* @param uiMessage {@link UiMessage}
* @param position 位置
* @param list 列表数据
* @param listener ViewModel 的点击事件监听器。如果某个子 view 的点击事件需要 ViewModel 处理，可通过此监听器回调。
* @return 点击事件是否被消费
*/
@Override
protected boolean onItemClick(ViewHolder holder, CustomMessage customMessage, UiMessage uiMessage, int position, List<UiMessage> list, IViewProviderListener<UiMessage>
listener) {
return false;
}

/**
* 根据消息内容，判断是否为本模板需要展示的消息类型
*
* @param messageContent 消息内容
* @return 本模板是否处理。
*/
@Override
protected boolean isMessageViewType(MessageContent messageContent) {
return messageContent instanceof CustomMessage ;
}

/**
* 在会话列表页某条会话最后一条消息为该类型消息时，会话里需要展示的内容。
* 比如：图片消息在会话里需要展示为"图片"，那返回对应的字符串资源即可。
* @param context 上下文
* @param customMessage 自定义消息
* @return 会话里需要展示的字符串资源
*/
@Override
public Spannable getSummarySpannable(Context context, CustomMessage customMessage) {
return null;
}
}
```

6. 扩展区域自定义方式变更

输入区域配置方式变更，提供了 DefaultExtensionConfig 类，该类提供了 SDK 默认插件列表和表情列表。可以通过复写此类调整插件的位置或进行增减。

7. 本地通知变更

当应用退到后台后，同一个人发来多条消息时，本地通知由之前的折叠显示方式更改为分条显示。

8. 内部实现变更

- 资源名称变更。IMKit SDK 5.x 版本重新统一了资源名称的命名，如果您自定义了部分资源文件，可参考官网自定义文档，重新替换对应名称的资源。
- 移除了 EventBus 。IMKit 5.x 版本不再依赖于 EventBus 进行事件分发，更改为通过观察者模式实现。如果您应用里使用了旧版本 SDK 里的 EventBus, 升级后需要您从应用层自己引入依赖。

- 移除 RongContext 类。如果您应用里有调用 RongContext 作为上下文使用的地方，请更改为调用您自己的应用上下文。
- 由于更改为 MVVM 框架，原 fragment 里很多业务相关接口被移除，您可参考官网 5.x 版本文档重新进行页面自定义。

4.X 版本升级指导

1. 依赖 Androidx

IMKit 5.x 版本不再支持 support 包，开发者需要参考 AndroidX 迁移文档 将应用的依赖转换为 Androidx.

2. 连接接口变更

连接接口的返回值由 RongIM 更改为 void 类型即可。

3. 适配会话列表 ConversationListFragment

- 删除 ConversationListFragment 集成时对 setUri() 的调用，原 setUri() 的参数改为下面方法配置：

```
RongConfigCenter.conversationListConfig().setDataProcessor(new DataProcessor<Conversation>() {
@Override
public Conversation.ConversationType[] supportedTypes() {
return supportedTypes; //此处返回会话列表支持的会话类型
}

@Override
public List<Conversation> filtered(List<Conversation> data) {
return data;
}

@Override
public boolean isGathered(Conversation.ConversationType type) {
if (type.equals(Conversation.ConversationType.SYSTEM)) {
return true; //需要聚合显示的会话，返回 true.
} else {
return false;
}
}
});
```

SDK 默认支持所有会话类型，都非聚合显示。如果您没有自定义需求，可以不进行上述设置，直接去掉 setUri() 的调用即可。

- 将会话列表 activity 注册到 SDK。

```
RouteUtils.registerActivity(RouteUtils.RongActivityType.ConversationListActivity, CustomConversationListActivity.class);
```

- 更改会话列表页面启动方式，去掉老版本如下代码：

```
RongIM.getInstance().startConversationList(Context , supportedConversation);
```

改为调用如下方法：

```
RouteUtils.routeToConversationListActivity(context,title);
```

4. 适配会话页面 ConversationFragment

- 将会话 activity 注册到 SDK

```
RouteUtils.registerActivity(RouteUtils.RongActivityType.ConversationActivity, CustomConversationListActivity.class);
```

5. 设置用户信息

用户信息相关接口所属类由 RongIM 迁移到 UserDataProvider 类中，删除原用户信息接口指定的父类 'RongIM.'，AndroidStudio 会引导 import 新的路径，根据提示操作即可。

6. 适配新的自定义方式

由于架构的变更，原 fragment 里很多业务相关接口被移除，开发者需要根据原自定义需求参考 5.x 文档自定义部分重新适配。

IM 翻译插件 IM 翻译插件

更新时间:2024-08-30

- IMLib 与 IMKit 从 5.2.2 版本开始支持翻译插件。
- 该插件暂仅适用于使用新加坡数据中心的应用。详见海外数据中心。

融云即时通讯业务提供翻译插件，可为 IMLib 与 IMKit SDK 快速接入外部翻译服务，由融云服务端负责对接外部翻译服务供应商的鉴权、API 调用、账号管理、计费等流程。翻译插件支持翻译文本。IMKit SDK 提供翻译 UI。

目前已支持接入 Google 翻译服务。

翻译流程

服务开通

该功能为付费增值服务。如有需求，请前往控制台 IM 翻译 页面开通服务。

关于 IM 翻译服务费用，详见 IM 翻译计费说明 。

客户端鉴权

客户端需要持有有效的 JWT Token，才能向融云请求翻译结果。

您的 App 服务端需要调用融云服务端 API 接口获取 JWT Token，然后返回给客户端。详见服务端文档获取 JWT Token 。

> 提示
> 翻译插件鉴权专用的 JWT Token 不同于 IM 用户连接 IM 服务的 Token，请注意区分。

JWT

JWT 全称 JSON web Token，是为了在网络应用环境间传递声明而执行的一种基于 JSON 的开放标准。 JWT 包含 header、payload、signature 三部分。通过解析 payload 部分可获取到 Token 有效期和 UserId 等信息。

获取和刷新 JWT Token 流程图

集成翻译插件

翻译插件以 aar 包方式提供。请首先在融云官网 下载 IMLib/IMKit SDK 。

1. 解压下载的 SDK 压缩包后，打开 IMLib 文件夹，将其中的 translation.aar 文件拷贝到项目 libs 目录下。
2. 在项目的 build.gradle 文件中添加以下依赖：

```
dependencies {
...
implementation files('libs/translation.aar')
}
```

IMKit 使用翻译插件

从 IMKit 5.2.2 版本开始，IMKit 集成了翻译功能。在开始使用翻译插件之前，请确认已开通翻译服务。

1. App 需要向自身的应用服务器发起请求，由应用服务器调用融云服务端 API 获取 JWT Token。App 获取 JWT Token 后，通过 updateAuthToken 接口设置到 SDK 中。

> 提示
> IMKit 只处理翻译结果，不会维护 JWT Token 的有效性。App 开发者需自行更新 JWT Token。并利用翻译插件提供的回调，在 JWT Token 失效时，重新获取并设置 JWT Token。

示例代码

```
TranslationClient.getInstance()
.addTranslationResultListener(
new TranslationResultListener() {
@Override
public void onTranslationResult(int code, RCTranslationResult result) {
if (code == IRongCoreEnum.CoreErrorCode.RC_TRANSLATION_CODE_INVALID_AUTH_TOKEN.code
|| code == IRongCoreEnum.CoreErrorCode.RC_TRANSLATION_CODE_AUTH_FAILED.code
|| code == IRongCoreEnum.CoreErrorCode.RC_TRANSLATION_CODE_SERVER_AUTH_FAILED.code) {
updateJwtToken();
}
}
});
```

2. （可选）IMKit 已设置默认语言。可通过以下方法修改：

示例代码

```
RongConfigCenter.featureConfig().rc_translation_src_language = "zh_CN";
RongConfigCenter.featureConfig().rc_translation_target_language = "en";
```

IMLib 使用翻译插件

1. **在用户登录成功之后，需先判断当前是否开通翻译服务。**

示例代码

```
RongCoreClient.getInstance().isTextTranslationSupported()
```

2. **在确认支持翻译服务之后，可以开始进行客户端鉴权。**App 需要向自身的应用服务器发起请求，由应用服务器调用融云服务端 API 获取 JWT Token。App 获取 JWT Token 后，通过 `updateAuthToken` 接口设置到 SDK 中。

示例代码

```
TranslationClient.getInstance().updateAuthToken("token");
```

3. 通过 `addTranslationResultListener` 添加翻译结果回调。

示例代码

```
// 添加监听
TranslationClient.getInstance()
.addTranslationResultListener(
new TranslationResultListener(){
@Override
public void onTranslationResult(int code,RCTranslationResult result){
// handle result
}
});

// 不需要时，及时移除监听，否则可能导致内存泄漏
TranslationClient.getInstance().removeTranslationResultListener(listener);
```

`RCTranslationResult` **参数说明**

参数名	类型	描述
messageId	Int	消息 ID。
srcText	String	源文本。
translatedText	String	翻译之后文本。
srcLanguage	String	源语言类型。
targetLanguage	String	目标语言类型。

4. 调用 `translate` 翻译文本。

示例代码

```
TranslationClient.getInstance()
.translate(
message.getMessageId(),
content,
"zh_CN", // 源语言类型
"en"); // 目标类型
```

参数说明

参数名	类型	描述
messageId	Int	消息 ID。`messageId` 大于 0 时 SDK 会在本地缓存翻译结果（推荐）。如果无需缓存，可传入小于等于 0 的 `messageId`。
content	String	要翻译的文本。
srcLanguage	String	源语言类型。
targetLanguage	String	目标语言类型。

> 提示
>
> 接入 Google 翻译时，`srcLanguage` 可传任意值。Google 翻译服务会自动识别待翻译文本的源语言，并仅以识别结果为准。

支持的语言类型

翻译插件支持的语言可参见文档中列出的语言列表。

Google 翻译服务

翻译插件已支持通过 Google Cloud Translation 服务翻译以下语言。更多细节，您可以直接参考 Google Cloud Translation 官方文档：语言列表 。

语言	枚举值（已替换 ISO-639 语言代码中的 - 为 _）
南非荷兰语	af
阿尔巴尼亚语	sq
阿姆哈拉语	am
阿拉伯语	ar
亚美尼亚文	hy
阿萨姆语	as
艾马拉语	ay
阿塞拜疆语	az
班巴拉语	bm
巴斯克语	eu
白俄罗斯语	be
孟加拉文	bn
博杰普尔语	bho
波斯尼亚语	bs
保加利亚语	bg
加泰罗尼亚语	ca
宿务语	ceb
中文（简体）	zh_CN 或 zh
中文（繁体）	zh_TW
科西嘉语	co
克罗地亚语	hr
捷克语	cs
丹麦语	da
迪维希语	dv
多格来语	doi
荷兰语	nl
英语	en
世界语	eo
爱沙尼亚语	et
埃维语	ee
菲律宾语（塔加拉语）	fil
芬兰语	fi
法语	fr
弗里斯兰语	fy
加利西亚语	gl
格鲁吉亚语	ka

语言	枚举值（已替换 ISO-639 语言代码中的 - 为 _）
德语	de
希腊文	el
瓜拉尼人	gn
古吉拉特文	gu
海地克里奥尔语	ht
豪萨语	ha
夏威夷语	haw
希伯来语	he 或 iw
印地语	hi
苗语	hmn
匈牙利语	hu
冰岛语	is
伊博语	ig
伊洛卡诺语	ilo
印度尼西亚语	id
爱尔兰语	ga
意大利语	it
日语	ja
爪哇语	jv 或 jw
卡纳达文	kn
哈萨克语	kk
高棉语	km
卢旺达语	rw
贡根语	gom
韩语	ko
克里奥尔语	kri
库尔德语	ku
库尔德语（索拉尼）	ckb
吉尔吉斯语	ky
老挝语	lo
拉丁文	la
拉脱维亚语	lv
林格拉语	ln
立陶宛语	lt
卢干达语	lg
卢森堡语	lb
马其顿语	mk
迈蒂利语	mai
马尔加什语	mg
马来语	ms
马拉雅拉姆文	ml
马耳他语	mt
毛利语	mi
马拉地语	mr
梅泰语（曼尼普尔语）	mni_Mtei
米佐语	lus
蒙古文	mn
缅甸语	my
尼泊尔语	ne
挪威语	no
尼杨扎语（齐切瓦语）	ny
奥里亚语（奥里亚）	or
奥罗莫语	om
普什图语	ps
波斯语	fa
波兰语	pl
葡萄牙语（葡萄牙、巴西）	pt
旁遮普语	pa
克丘亚语	qu
罗马尼亚语	ro
俄语	ru
萨摩亚语	sm

语言	枚举值（已替换 ISO-639 语言代码中的 - 为 _）
梵语	sa
苏格兰盖尔语	gd
塞佩蒂语	nso
塞尔维亚语	sr
塞索托语	st
修纳语	sn
信德语	sd
僧伽罗语	si
斯洛伐克语	sk
斯洛文尼亚语	sl
索马里语	so
西班牙语	es
巽他语	su
斯瓦希里语	sw
瑞典语	sv
塔加路语（菲律宾语）	tl
塔吉克语	tg
泰米尔语	ta
鞑靼语	tt
泰卢固语	te
泰语	th
蒂格尼亚语	ti
宗加语	ts
土耳其语	tr
土库曼语	tk
契维语（阿坎语）	ak
乌克兰语	uk
乌尔都语	ur
维吾尔语	ug
乌兹别克语	uz
越南语	vi
威尔士语	cy
班图语	xh
意第绪语	yi
约鲁巴语	yo
祖鲁语	zu

状态码

状态码	原因
26200	翻译成功
26201	翻译失败，融云鉴权失败 鉴权失败或者 token 过期
26202	翻译失败，翻译功能服务商鉴权失败 融云服务器的原因，token 无效
26203	翻译失败，翻译功能服务商返回失败 具体服务商失败码信息
26204	翻译失败，翻译功能未在融云开启
26205	翻译失败，融云限流
26206	翻译失败，Server 没有鉴权 token 的 secret 需要在控制台开启
26208	短语音转录结果为空, 请核对编解码类型
26209	语言设置错误
26210	编码格式设置错误

状态码	原因
34100	没有设置 authToken 或者 authToken 为 空串
34101	待翻译文本内容为空
34102	目标语言为空
34103	源语言为空
34104	翻译服务器地址为空
34105	app key 为空
34106	服务器返回数据无效

适配 Android OS 版本　　适配 Android OS 版本 Android 13 应用兼容性适配指导

更新时间:2024-08-30

谷歌于 2022 年第三季度发布 Android 13 与 33 级框架 API。

融云的 IMLib/IMKit SDK 已在 5.3.0 版本（开发版）与 5.1.9.5 版本（稳定版）适配了 Android 13，但部分特性可能需要您的 App 进行改动。我们在下面列出了受影响的项目：

1. 为确保推送功能正常工作，请使用各个厂商已适配 Android 13 的推送 SDK，具体版本如下：
   - 小米推送 SDK 版本：5.1.0+
   - 华为推送 SDK 版本：6.7.0.300+
   - oppo 推送 SDK 版本：3.1.0+
   - vivo 推送 SDK 版本（官方暂未适配 Android 13）：3.0.0.4+
2. Android 13 新增了推送权限，该权限融云 SDK 内部不会申请，需要用户 APP 自行申请。
3. 对于使用源码集成 IMKit 的用户，请将 compileSdkVersion 修改为 33，否则无法通过编译。

状态码 SDK 状态码

更新时间:2024-08-30

状态码	说明
-4	应用没有调用 connect() 方法，即调用业务。
-3	参数异常, 请确认参数是否填写正确且有效。
-2	IPC 进程意外终止。当 libRongIMLib.so 或 libsqlite.so 找不到或出现崩溃时也会触发此错误。如果是系统进行了资源回收后调用接口会触发此问题，SDK 会做好自动重连。
-1	未知错误。
0	连接成功。
405	已被对方加入黑名单，消息发送失败。
407	未在对方的白名单中，消息发送失败。
20106	该用户处于单聊禁言状态，禁止发送单聊消息。
20604	发送消息频率过高，1 秒钟最多只允许发送 5 条消息，详细请联系商务，电话：13161856839。
20605	信令被封禁。如遇到该错误，请提交工单。
20607	调用超过频率限制，请稍后再试。
22201	消息扩展/修改，但是原始消息不存在。
22202	消息扩展/修改，但是原始消息不支持扩展。
22203	消息扩展/修改，扩展内容格式错误。
22204	消息扩展/修改，无操作权限。
22406	当前用户不在群组中。
22408	当前用户在群组中已被禁言。
23406	当前用户不在聊天室中，请加入聊天室再调用和聊天室相关方法。
23408	当前用户在该聊天室中已被禁言。
23409	当前用户已被踢出并禁止加入聊天室。被禁止的时间取决于服务端调用踢出接口时传入的时间。
23410	聊天室不存在。
23411	聊天室成员超限，开发者可以提交工单 申请聊天室人数限制变更。
23412	聊天室接口参数不正确。
23414	聊天室云存储业务未开通。
23423	聊天室的属性个数超限，单个聊天室默认上限为 100 个。
23424	没有权限修改聊天室中已存在的属性值。

状态码	说明
23425	聊天室中属性设置频率超限，单个聊天室每秒上限 100 次。
23426	未开通聊天室属性自定义设置。请在控制台**免费基础功能**页面开通**聊天室属性自定义设置**。
23427	聊天室属性不存在。
23431	多端操作聊天室同一属性时属性设置失败。
24401	超级群功能没有开通。
24402	超级群服务异常。
24403	超级群参数错误。
24404	超级群未知异常。
24406	当前用户不在超级群中。
24408	当前用户在超级群中已被禁言。
24410	超级群不存在。
24411	超级群成员超限制。
24412	用户加入超级群数量超限。
24413	创建超级群频道，频道数超限。
24414	超级群频道 ID 不存在。
24416	用户不在超级群私有频道成员列表中
25101	撤回参数不正确。
25102	未开通历史消息云存储。请开通单群聊**消息云端存储**或**聊天室消息云端存储**服务。
25103	清除历史消息时，传递的时间戳大于当前系统时间。
25105	清除历史消息时遇到内部异常，请提交工单确认原因。
25107	撤回他人消息失败。服务端可控制消息是否可由他人（非发送者本人）撤回。如果服务端已设置为仅限发送者本人撤回，则在撤回他人消息时报这个错误。
26001	Push 参数不正确。
26002	向服务端同步时出现问题，有可能是操作过于频繁所致。请稍后再试。
30001	连接已被释放,连接相关的错误码，SDK 会做好自动重连，开发者无须处理。
30002	连接不可用, 连接相关的错误码，SDK 会做好自动重连，开发者无须处理。
30003	客户端发送消息请求，融云服务端响应超时。
30004	导航 HTTP 发送失败。
30005	请求连接导航地址失败。
30006	请求连接导航地址后，接收数据失败。

状态码	说明
30007	导航 HTTP 请求失败。
30008	导航 HTTP 返回数据格式错误。
30009	导航数据解析后，其中不存在有效 IP 地址。
30010	创建 Socket 连接失败,连接相关的错误码，SDK 会做好自动重连，开发者无须处理。
30011	Socket 断开。
30012	PING 失败。
30013	PING 超时。
30014	信令发送失败。
30015	连接过于频繁。
30016	消息大小超限，消息体最大 128 KB。
31000	连接ACK超时。
31002	初始化时填写的 AppKey 不正确，请在控制台获取。
31004	Token 无效。
31005	App 校验未通过（开通了 App 校验功能，但是校验未通过）
31006	连接重定向。连接相关的错误码，SDK 会做好自动重连，开发者无须处理。
31007	ApplicationId 与后台注册信息不一致。
31008	AppKey 被封禁或已删除。
31009	连接失败，一般因为用户已被封禁。
31010	当前用户在其他移动设备上登录，此设备被踢下线。
31011	用户在线时被封禁导致连接断开。
31020	Token过期。一般是因为在控制台设置了token 过期时间，需要请求您的服务器重新获取 token 并再次用新的 token 建立连接。
31023	重连过程中当前用户在其它移动设备上登录。
31028	配置了 SDK 通过代理连接融云，但代理地址不可用。
32061	连接被拒绝. SDK 会自动重连，开发者无须处理。
32054	TCP 连接被重置，可能原因是运营商认为此链接非法或无效。SDK 会自动触发重连，App 侧无需处理。
33000	将消息存储到本地数据时失败。发送或插入消息时，消息需要存储到本地数据库，当存库失败时，会回调此错误码。
33001	未调用 SDK 的初始化（`init`）方法。
33002	数据库错误。
33003	调用接口时传入的参数不正确。

状态码	说明
33007	未开通历史消息云存储服务，参见控制台文档开通单群聊历史消息云存储服务
34001	连接已存在，或正在重连中。
34002	小视频时间长度超出限制.默认小视频时长上限为 2 分钟。
34003	GIF 消息文件大小超出限制.默认 GIF 文件大小上限是 2 MB。
34004	聊天室状态未同步完成，加入聊天室时立即调用获取聊天室属性接口，极限情况下会存在本地数据和服务器未同步完成的情况。您可以设置聊天室属性回调，SDK 同步完成时会在属性回调中通知，您可根据回调状态进行获取。
34005	连接环境不正确。
34006	连接超时。
34007	查询的公共服务信息不存在。
34008	消息不能被扩展。消息在发送时，`Message` 对象的属性 `canIncludeExpansion` 置为 `true` 才能进行扩展。
34009	消息扩展失败。一般是网络原因导致的，请确保网络状态良好，并且融云 SDK 连接正常。
34010	消息扩展大小超出限制， 默认消息扩展字典 key 长度不超过 32 个字符，value 长度不超过 4096 个字符（ SDK < 5.2.0 时，value 长度限制最大 64 个字符），设置的 Expansion 键值对不超过 300 个。
34021	消息未注册。发送或者插入自定义消息之前，请确保注册了该类型的消息。
34022	该接口不支持超级群会话。
34023	超级群功能未开通。
34024	超级群频道不存在。
34004	聊天室状态未同步完成，加入聊天室时立即调用获取聊天室属性接口，极限情况下会存在本地数据和服务器未同步完成的情况，开发者可以设置聊天室属性回调，SDK 同步完成时会在属性回调中通知开发者，开发者可根据回调状态进行获取。
34005	连接环境不正确。2.10.4 公有云 SDK 将不再支持连接私有云. 私有云要使用新版 SDK 可联系商务。
34209	传入的 `ConversationType` 非法。
34210	传入的 `targetId` 非法。
34211	传入的 `channelId` 非法。
34212	传入的 `tagId` 非法。
34213	传入的 `tagName` 非法。
34214	传入的 `userId` 非法。
34215	传入的 `userIdList` 非法。
34238	非法的代理配置。请检查代理是否为空或者是否传入了非法参数。
34239	传入的代理测试服务非法。
34240	代理地址或 testHost 地址无法连通。
34241	超级群撤回了不支持的消息类型，请开发者判断当前 MessageContent 类型是否支持被撤回。
40006	RTC 房间操作时传入参数错误。

更新日志（开发版） 更新日志（开发版）

更新时间:2024-08-30

> 提示
> 仅 Android/iOS 平台的 IM SDK 存在开发版、稳定版区分。开发版（Dev）SDK 首推新功能，同时会得到最快的 bug 修复。

5.10.1 Dev

发布日期：2024/07/02

问题修复

- 是否同步置顶空会话的开关默认值改为 NO，默认不同步置顶的空会话。

5.10.0 Dev

发布日期：2024/06/28

新增功能

- 新增了用户信息托管功能，支持修改、查询、订阅托管的用户信息。
- 新增了一个置顶空会话的开关配置，开发者可以选择是否同步置顶的空会话。
- 支持了心跳间隔时间的配置。
- 优化了百度 DoH 的使用

问题修复

- 修复了安卓 FileProvider 存在的漏洞问题。
- 修复了开启多设备消息同步功能后，多端同步阅读状态消息失败，导致卸载重装或者多端登录无法同步消息未读数的问题。
- 修复了 IMKit SDK 转发选择联系人界面背景是黑色的问题。

5.8.2 Dev

发布日期：2024/06/05

新增功能

- 新增了 IMKit 会话页面消息全部拉取完的回调。
- 新增了批量获取会话信息的 API。
- 新增了会话置顶操作是否更新操作时间开关。
- 新增了聊天室消息排重开关。

优化功能

- 优化了合并转发消息的内容显示格式。
- IMKit 和 IMLib 中更新了高德地图版本，高德新版本优化安全合规。

问题修复

- 修复了引用消息原文件已下载，但点击引用处的文件依然显示开始下载的问题。
- 修复了同时使用两个不同 token 连接，偶现 第二个 token 连接成功。
- 修复了 获取远端历史消息，content 为空的问题。
- 修复了 IM 误判没有网路状态，导致连接挂起的问题。

5.8.1 Dev

发布日期：2024/04/29

新增功能

- 增加了客户端订阅用户在线状态的功能。
- IMKit 和 IMLib 中更新了高德地图版本，不再支持 x86 架构。
- 更新了荣耀推送。
- 支持了在指定会话中，对指定消息类型的历史消息，按关键字进行搜索的功能。

优化功能

- 优化了消息撤回功能，现在撤回消息时会同时撤回命令消息中携带的用户信息（`UserInfo`）和额外信息（`extra`）。

问题修复

- 修复了获取时间差时如果为0，下次获取真实值需要60s的问题。
- 修复了 `StatisticsProcessor` 空指针异常的问题。
- 修复了 FCM 透传消息没有赋值 `ChannelId` 的问题。
- 修复了 channelId 为 null 时，创建 `NotificationChannel` 失败崩溃的问题。
- 修复了 5.6.10 版本 `ConversationListViewModel` 内存泄漏的问题。

5.8.0 Dev

发布日期：2024/03/29

新增功能

- 支持获取定向消息的目标用户列表。此功能仅适用于普通群和超级群消息。
- 新增了错误码 34296，针对发送定向消息，当会话类型不是群聊、超级群，且定向消息目标用户列表为空时，返回此错误。
- 新增了协议栈的数据错误码。

优化功能

- 支持配置上报 token 的优先级。

问题修复

- 修复了 IMKit 鸿蒙 4.0 系统引用消息的气泡没有固定长度的问题。
- 修复了 IMKit 退出登录后没有清空上一个账号缓存的用户信息的问题。
- 修复了 `RongCoreClientImpl.connectServer` 偶现 ANR 的问题。

5.6.10 Dev

发布日期：2024/01/31

问题修复

- 修复了 5.6.8/5.6.9 版本存在的设计缺陷。

5.6.9 Dev

发布日期：2024/01/31

> ⚠ 警告
> 该版本 SDK 已被召回，请勿使用。如已使用，建议尽快升级至 5.6.10 版本。

新增功能：

- 超级群支持发送定向消息，可给指定频道中的指定用户发送消息，频道中其他用户不会收到该条消息。
- 超级群支持同时从本地和远端删除用户的历史消息。

优化功能：

- 设置会话置顶后，如果本地会话列表中不存在该会话（尚未创建或已被删除），SDK 会自动创建该会话，并将其置顶。
- 获取会话列表时，支持通过参数指定返回结果忽略置顶状态，严格按照时间排序返回会话列表。

问题修复

- 修复部分机型上设置重连互踢策略（`setReconnectKickEnable`）后不生效的问题。

5.6.8 Dev

发布日期：2023/12/29

> ⚠ 警告
> 该版本 SDK 已被召回，请勿使用。如已使用，建议尽快升级至 5.6.10 版本。

优化功能：

- 支持小米海外推送服务，需要集成小米国际版推送客户端。
- 推送功能依赖的 vivo 推送 SDK 从 3.0.0.4 升至 3.0.0.7。如果使用推送（旧版）集成方案，请注意同时升级 vivo 推送 SDK。官网下载包内附带的 vivo 推送 SDK 已更新。
- 推送功能依赖的 OPPO 推送 SDK 从 3.1.0 升至 3.4.0。如果使用推送（旧版）集成方案，请注意同时升级 OPPO 推送 SDK。官网下载包内附带的 OPPO 推送 SDK 已更新。
- 优化接收消息的状态处理。接收消息后，无论是否已被同时在线或之前登录的其他设备接收。只要其他设备先收到该消息，该状态值都会变为已接收。如果在其他设备已被阅读，同时还会变为已阅读。
- 会话（`Conversation`）新增操作时间（`operationTime`）属性，可在分页获取会话列表时作为传入的时间戳。

问题修复

- 修复主动调用断开连接方法后，`ConnectionStatusListener` 会重复触发 `SIGN_OUT` 状态回调的问题。

5.6.7 Dev

发布日期：2023/11/23

新增功能：

- 支持集成荣耀推送，消息推送属性中新增荣耀推送配置参数。
- 消息推送属性中新增 iOS Time Sensitive 时效性配置。
- IMLib 聊天室成员变更功能支持返回当前聊天室人数。
- IMLib 支持新的自定义消息流程。
- IMKit 支持复制引用消息中的被引用内容。

问题修复：

- 修复 `CSConversationUIRenderer` 空指针的问题。
- 修复 `TextMessage` 的 `content` 内容为空时，发出的消息缺少 `content` 字段的问题。

5.6.6 Dev

发布日期：2023/10/27

优化功能：

- 优化 SDK 日志上传机制。

问题修复：

- 修复 SDK 内部问题。

5.6.5 Dev

发布日期：2023/10/12

优化功能：

- 优化 SDK 内部日志。

问题修复：

- 修复 5.6.4 版本注册自定义消息失效的问题。

5.6.4 Dev

发布日期：2023/09/25

新增功能：

- 超级群业务可以使用 `ChannelClient` 下的 `getRemoteHistoryMessages` 方法获取远端历史消息。
- 超级群业务支持使用 `getUltraGroupMessageCountByTimeRange` 统计本地历史消息数量。
- IMKit 本地通知默认 category 类型设置为 `CATEGORY_MESSAGE`。

优化功能：

- 获取本地指定标签下的会话（`getConversationsFromTagByPage`）返回的 `Conversation` 新增 `isTopForTag` 属性，用于标识会话在当前标签下是否已置顶。
- 安全性改进。

问题修复：

- 修复 IMLib 无法连接的问题。
- 修复 IMKit `ConversationListViewModel` 内存泄露的问题。
- 修复 IMKit `MessageViewModel.onScrolled` 数组越界崩溃的问题。
- 修复 `UniqueIdUtils.getUniqueId` 崩溃的问题。
- 修复 IMKit 会话列表页面加载问题。

5.6.3 Dev

发布日期：2023/08/31

新增功能：

- IMLib 加入聊天室可返回聊天室当前状态（是否禁言、是否在禁言白名单中、聊天室人数等）信息。聊天室房间事件监听器中新增对应的回调方法。

- IMKit 提供 `ToastUtils` 类，支持统一拦截 Toast 消息框。
- IMKit 会话页面支持拦截点击常用语按钮的事件。

优化功能：

- IMKit 会话页面的长按删除消息功能改为默认同步删除远端历史消息。
- 支持的 Android 最低版本提升为 5.0，要求 `minSdkVersion` 最低为 21。

问题修复：

- 修复 IMKit 图片选择页面（PictureSelectorActivity）申请权限没有提供给用户回调的问题。
- 修复 IMKit 点击输入区域扩展面板图片入口后，错误弹出权限提示的问题。
- 修复 IMKit 媒体消息中的视频文件地址中存在特殊字符导致播放视频消息黑屏的问题。
- 修复 IMKit 字体太大导致语音消息倒计时在 UI 上显示不全的问题。
- 修复 IMKit 跳转到指定消息定位不准确的问题。
- 修复 IMKit 中 `VideoSlimEncoder` 的视频压缩错误的问题。
- 修复 PushDeliveryUpload 相关的 ANR 问题。

5.6.2 Dev

发布日期：2023/08/11

新增功能：

- IMKit/IMLib 支持多端同步系统会话阅读状态，新增错误码 20109。
- IMLib 超级群支持搜索本地数据库中指定用户 ID 发送的消息，支持通过关键词搜索所有频道的消息。
- IMKit 适配 Android 14 新增的照片和视频权限 `< uses-permission android:name = "android.permission.READ_MEDIA_VISUAL_USER_SELECTED" />`。

优化功能：

- 优化小米推送 SDK 注册失效，无法获取Token 的情况。增加一次重试获取 Token。
- 调整 SDK 重连时间间隔为 0.05s, 0.25s, 0.5s, 1s, 2s, 4s, 8s, 16s, 32s。之后每 64s 重试一次。

问题修复：

- 修复 IMKit 会话列表的下拉加载框经常出现一直转无法关闭的情况。
- 修复 IMKit 聚合会话未读数不准确的问题
- 修复 IMKit `PicturePagerActivity#java.lang.IllegalStateException: Cannot obtain size for recycled Bitmap` 问题。

5.6.1 Dev

发布日期：2023/07/14

优化功能：

- 支持在融云的 `libc++_shared.so` 不满足项目要求时剔除融云 SDK 中的 `libc++_shared.so`。详见知识库文档 Libc++_shared.so 标准库常见问题处理方案。
- 优化 IMKit 单聊、群聊会话页面消息加载速度。

问题修复：

- 修复在华为 9.0 系统中偶现单聊发送位置消息崩溃的问题
- 修复内存泄漏的问题。
- 修复 IMKit 初始化 `InitOption` 配置海外区域码无效的问题。
- 修复调用 `startConversation` 且 `fixedMsgSentTime` 为首条消息的时间导致会话页面空白的问题。
- 修复 5.4.6 版本开始收不到不落地通知的问题。

5.6.0 Dev

发布日期：2023/07/03

新增功能：

- 融云 Push 2.0 推送集成方案将第三方厂商推送通道的 SDK 封装成插件，方便开发者快速集成与配置，适用于 IMLib、IMKit 或其他依赖 IMLib 的融云 Android 客户端 SDK。
- 超级群业务中，获取未读 @ 消息的摘要信息 `getUltraGroupUnreadMentionedDigests` 接口返回的 `MessageDigestInfo` 中新增消息类型标识，可用于筛选数据。

问题修复：

- 修复推送注册死循环问题
- 修复拉取超过 7 天未更新的会话，会话列表最后一条消息显示 null 的问题。
- 修复 `getBlacklist` 方法在没有黑名单用户时超时的问题。
- 修复多线程操作 LinkedList 集合导致的空指针异常。
- 修复 `getConversationList` 无回调，多进程情况下线程被阻塞问题。

5.4.7 Dev

发布日期：2023/06/20

问题修复：

- 修复超级群业务的回调方法 `onUltraGroupMessageExpansionUpdated` 返回的 `RCMessage` 的消息 ID 为 -1 的问题。

5.4.6 Dev

发布日期：2023/06/15

新增功能：

- 新增批量获取当前用户的超级群的未读消息数接口 `getUltraGroupConversationUnreadInfoList` 一次获取最多 20 个超级群下所有频道的未读数据。

问题修复：

- 修复合并转发消息内邮箱地址不会识别的问题
- 修复 5.4.0 及之后版本上设置消息排重接口 `setCheckDuplicateMessage` 不生效的问题
- 修复 卸载重装第一次启动，`getConversationListByPage` 会延迟5秒才加载的问题
- 修复 MessageViewModel 内存泄漏问题
- 修复调用超级群修改消息内容接口 `ModifyUltraGroupMessage` 后未更新搜索索引的问题

优化功能：

- 优化后台切换回前台时重连耗时长的问题
- 优化获取指定时间戳前或后消息接口，以实际传入时间戳为准，SDK 内部不做时间戳 +1 或 -1 处理

5.4.5 Dev

发布日期：2023/05/29

新增功能：

1. 新增聊天室事件通知监听器 `ChatRoomNotifyEventListener`，支持在聊天室中执行成员封禁、禁言等操作时接收通知（封禁、禁言时需要指定 `needNotify` 为 `true`），支持在用户多端加入/退出接收通知。
2. 获取超级群获取频道列表时，支持通过 `conversation` 对象获取 @我 的未读消息数。
3. 超级群获取本地历史消息功能支持获取会话中指定时间戳前后、指定数量的消息。

问题修复：

1. 修复超级群未读消息数的问题。当前用户在连接状态下，超级群中有人撤回消息时（包括普通消息和 @消息），如消息在当前用户端为未读状态，未读消息及未读 @消息数没有修改。问题修复后，会对未读消息数做 -1 处理。
2. 修复超级群撤回消息小灰条提示重复的问题。超级群中撤回一条消息后，如本地没有找到原始消息，会插入小灰条消息。在特定情况下，可能出现小灰条消息重复的问题。问题修复后，小灰条消息会携带原始消息 ID，以进行排重。
3. 修复 IMKit SDK 初始化时传入空 `initOption` 导致 SDK 崩溃的问题。
4. 修复 获取远端历史消息的结果中小视频消息缩略图都相同的问题。
5. 修复 IMKit SDK 在特定手机上聊天中选择图片 > 拍照后，按钮没反应的问题
6. 修复 `getActiveNetworkInfo` ANR 问题
7. 修复 会话列表点击事件越界崩溃的问题
8. 修复 ipc 进程偶现的连接回调丢失的问题

5.4.4 Dev

发布日期：2023/05/11

新增功能：

1. 超级群业务新增获取本地超级群消息的方法 `getHistoryMessages`。
2. 发送媒体消息方法 `sendMediaMessage` 接口支持 VoIP 配置。

优化功能：

- `Message` 中增加是否为离线消息字段（`isOffline`），对齐 iOS 端。仅在接收消息的回调中有效。

5.4.3 Dev

发布日期：2023/04/21

问题修复：

1. 紧急修复 5.4.2 版本中与导航服务地址相关的问题。

5.4.2 Dev

发布日期：2023/04/20

新增功能

1. IMLib/IMKit SDK 支持在初始化配置 `InitOption` 中指定区域码。配置成功后，SDK 将使用与区域码对应的服务地址。
2. IMLib/IMKit SDK 支持在消息推送属性配置中指定 vivo 推送 category 参数。

优化功能

1. 优化 SDK 内置 IPluginModule 的稳定性
2. 为聊天室属性相关方法 `forceRemoveChatRoomEntry`、`removeChatRoomEntry`、`forceSetChatRoomEntry`、`setChatRoomEntry` 的 `notificationExtra` 字段增加长度校验

问题修复：

1. 修复 rc_picture_original_image_size 对应的阿拉伯语字符串
2. 修复 PicturePagerActivity 数组越界的问题
3. 修复频繁切换用户 token 进行连接时，登陆回调 userId 与 Token 会混淆的问题

5.4.1 Dev

发布日期：2023/03/31

优化功能

1. IMLib/IMKit SDK 新增支持 `InitOption` 的初始化接口。
2. IMKit SDK 预览小视频时支持左右滑动调整进度。
3. IMLib SDK 获取远端历史消息数量上限提升至 100 条。

问题修复：

1. 语音消息强制使用 AAC_ADTS 编码，修复部分机型发送的消息无法在 iOS、MacOS 上无法播放的问题
2. 去除 IMLib 层对 AndroidX 的使用
3. 修复部分导致崩溃的问题

5.4.0 Dev

发布日期：2023/03/03

新增功能

1. 发送消息时可在消息推送属性配置中设置华为推送通道的 Category 参数
2. IMLib SDK 新增超级群用户组功能

问题修复：

1. 修复 IMLib SDK 撤回超级群消息后，onUltraGroupMessageRecalled 回调的 RecallNotificationMessage 的 mOriginalObjectName 和 mOriginalMessageContent正常，但重新查询后的 Message 的 mOriginalObjectName 和 mOriginalMessageContent 不正常的问题
2. 修复 IMLib SDK 发送 FileMessage时，某些文件或者某些机型上读取 mimeType 报错的问题
3. 修复 IMKit SDK 线程问题导致的下标越界
4. 修复 消息发送成功回调时 ANR 的问题
5. 修复 IMLib SDK 超级群未读数不展示修改
6. 修复 IMLib SDK 合并转发下载特殊字符的文件，每次都要重复下载的 bug
7. 修复 IMKit SDK 接收到消息撤回处理时ANR问题处理
8. 修复 IMKit SDK 单聊个别消息不展示已读回执的问题
9. 修复 IMKit SDK 发送端下载媒体文件失败时候，红色叹号出现的问题
10. 修复 IMKit SDK 打开会话页面已读回执消息发送失败，计入消息重发列表的问题
11. 修复 IMKit SDK 群组消息在其他界面应该不发已读回执的问题
12. 修复 IMKit SDK 调整群已读 ui 位置
13. 修复 IMKit SDK 修复合并转发消息头像变形问题
14. 修复 IMKit SDK 输出视频时宽高错误导致编码器选择失败，导致小视频无法发送的问题
15. 修复 IMKit SDK 小视频录制页面，图像拉伸问题
16. 修复 IMKit SDK 小视频，图片，撤回时不弹框的问题
17. 修复 IMKit SDK 在 oppo 机型上录制小视频发送失败的问题
18. 修复 IMKit SDK 在对端登录接收群聊会话 @消息, 一端查看后另一端依然显示红色「有人@我」的问题
19. 修复 IMKit SDK 断开网络发送小视频提示"不能发送损坏的小视频"的问题

5.3.5 Dev

发布日期：2023/02/07

新增功能

1. IMLib SDK 本地批量插入消息接口支持将消息唯一标识 Message UID 存入数据库，支持针对 UID 进行排重

优化功能

1. 升级 xcrash 崩溃采集模块，增加符号表信息
2. 修复 小米 12 设备隐式跳转时无法携带 Parcelable 数据，引用消息中暂时改为显式跳转
3. IMKit SDK 新增 ConversationListAdapterSupportStrongerTouchEvent，可用于解决会话列表频繁刷新导致 Item 长按失效的问题
4. IMLib SDK 改为默认使用多进程模式
5. 优化合规问题，SDK 不再调用 TelephonyManager.getNetworkOperator

问题修复：

1. 修复 IMKit SDK 在 Android 13 上因权限问题导致图片选择不显示的问题
2. 修复 IMKit SDK 合并转发地图消息点击无响应的问题
3. 修复 IMKit SDK 下载 xml、txt 文件时候，开启 gzip 传输，获取文件大小异常的问题
4. 修复 IMKit SDK 未开启群组实时位置共享时，聊天页面收到邀请的加入通知的问题
5. 修复 IMKit SDK 在查看引用的文本消息时，对方撤回后未及时更新界面的问题
6. 修复 IMKit SDK 在发送时长超限制的小视频后，错误提示“不能分享超过 0 分钟的视频” 的问题。
7. 修复 IMLib SDK 在应用内存在其他功能录音时，发送语音消息会崩溃的问题
8. 修复 IMKit SDK 输入框在输入特定表情时光标会移动到最后的问题
9. 修复 IMKit SDK 在文本消息字数较多时，会话页面聊天消息错乱的问题

5.3.4 Dev

发布日期：2023/01/10

新增功能

1. IMKit SDK 支持配置文件消息的文件图标
2. IMLib SDK 超级群支持搜索本地消息

优化功能

1. 优化 IMKit SDK 发送消息，发送已插入本地的消息同时更新内容、扩展、状态
2. 优化 IMLib SDK 接收消息，在 IMLibCore 中添加了禁用消息排重机制的开关
3. 优化 IMKit SDK 录音，添加回退策略，不支持 HE_AAC 时回退为 AAC
4. 优化 IMKit SDK IMCenter 接收消息接口，增加 Async 类型 `OnReceiveMessageWrapperListener` 监听
5. 优化推送配置，支持禁用融云自建推送（RongPush）

问题修复：

1. 修复 IMKit SDK 特定情况下位置共享人数错误的问题
2. 修复 IMKit SDK 在 Android 13 设备上，从会话页面进入相册不显示相册照片，点击相机胶卷查看相册目录无响应的问题
3. 修复 IMKit SDK 点击合并转发消息中的位置消息无响应的问题
4. 修复翻译线程池核心线程一直占用的问题
5. 修复 SDK 5.3.2 / 5.3.3 版本调用 `removeChatRoomEntry` 后，其他人错误地触发 `onChatRoomKVUpdate` 回调的问题。修复后，其他人正常触发 `onChatRoomKVRemove` 回调。

5.3.3 Dev

发布日期：2022/12/22

问题修复：

1. 修复了 IMKit SDK 消息被删除之后重新拉取，会话列表没及时刷新问题
2. 修复了 IMKit SDK 在 isDelete 字段为 true 时还显示小灰条的问题

功能优化：

1. 优化 IMKit 在接收大量离线消息后 UI 的流畅性
2. 去掉 PushReceiver 涉及自启动的 intent-filter，避免引起应用上架问题
3. SDK 默认开启单进程

5.3.2 Dev

发布日期：2022/12/02

新增功能：

1. IMLib SDK 支持获取指定类型的所有未读会话的列表 `getUnreadConversationList`，支持单聊、群聊、系统会话

功能优化

1. IMLib SDK 用户断网重新连接后自动加入聊天室的逻辑中，增加用户是否被聊天室封禁的判断，如用户被封禁时，不再执行自动加入聊天室逻辑
2. IMKit SDK 更新合并转发模版
3. IMKit SDK 添加表情按钮隐藏配置开关，允许禁用表情面板

问题修复：

1. 修复部分场景下 SDK 导致 App ANR 问题
2. 修复部分场景下 IMKit 空指针问题
3. 修复 IMKit 初始化时报 UnsupportedOperationException 的问题
4. 修复 IMLib 超级群业务在多端同步已读消息时间戳时，未清除第一条未读消息时间戳（`firstUnreadMsgSendTime`）的问题。问题修复后，多端同步阅读状态时 `firstUnreadMsgSendTime` 会被置为 0。

5.3.1 Dev

发布日期：2022/11/18

新增功能：

1. IMLib SDK 加入聊天室后，断网重连场景下，重新加入聊天室成功后获取聊天室消息条数与断网前加入聊天室获取的消息条数一致
2. IMLib SDK 接收消息中同时提及（@）所有人和提及部分人时，支持获取 @ 部分人列表
3. IMKit SDK 发送本地通知时，从内存中获取免打扰级别信息

问题修复：

1. 修复 Glide 类冲突问题
2. 修复切换账号导致的位置共享人数不正确问题
3. 修复播放接收到的语音消息时，本端发送的语音消息异常播放问题
4. 修复位置共享时多线程导致的空指针问题
5. 修复在非主进程开启 CountDownTimer 造成的异常
6. 修复特定版本 SDK 升级导致的发送媒体失败问题

5.3.0 Dev

发布日期：2022/11/04

新增功能：

1. IMLib SDK 支持开启单进程
2. IMLib SDK Android 13 适配兼容
3. IMLib SDK 支持用户未加入聊天室时拉取聊天室的历史消息
4. IMKit SDK 群信息和群成员信息提供者添加 `extra` 字段

功能优化：

1. IMLib SDK 优化因坐标系不同，导致的和 IOS 定位有偏差的问题
2. IMLib SDK 移除部分不必要的权限，解决小米市场上架合规问题
3. IMLib SDK 超级群撤回消息时禁止撤回不支持的消息类型，新增错误码 34241
4. IMLib SDK 在 conversation 对象上提供 `getUnreadMentionedCount()` 方法，废弃原有 `getMentionedCount()` 方法
5. IMKit SDK 优化 `DataProcessor` 接口，新增重载方法

问题修复：

1. 修复偶发的 ANR 异常
2. IMLib SDK 修复集成第三方推送时，自行上报 PushToken 时类型错误问题
3. IMLib SDK 修复报 PushAdapter 空指针问题
4. IMLib SDK `getMessages` 接口未检验 `count` 参数的问题，合法取值范围为 2-20
5. IMKit SDK 修复聊天页面中，输入框有草稿时切换语音后再切换回来时发送按钮状态不正确问题

5.2.5 Dev

发布日期：2022/09/09

新增功能：

1. 初始化增加重载方法，由用户判断进程信息
2. Android 切换前台后探测连接是否存在
3. 按会话免打扰级别，获取未读消息数
4. 多端会话状态同步支持返回会话上设置的免打扰级别
5. 超级群撤回消息时，即时本地不存在原始消息，自动插入一条撤回小灰条消息
6. 超级群获取未读 @消息列表
7. 含敏感词消息回调信息中增加 sourceType，sourceContent 字段，只针对超级群会话

8. 超级群 `getUnreadMentionedMessages` 支持传入消息数量，拉取顺序参数
9. 聊天室房间状态监听支持多代理

问题修复：

1. 修复 getBatchRemoteUltraGroupMessages 回调多次的问题
2. 修复收集用户个人信息的频率超过合规范围的问题
3. 修复已读回执开关对会话列表不生效的问题
4. 修复 IPC 进程设置日志等级失效的问题,减少跨进程回调
5. 修复已读回执图标不显示的问题
6. 修复 push 模块重复收到 token 重复上报的问题

5.2.4 Dev

发布日期：2022/07/22

新增功能：

1. IMLib SDK 含敏感词消息回调信息中增加频道 ID 字段，只针对超级群会话。
2. IMLib SDK 超级群会话支持了私有频道功能。通过 Server API 创建私有频道并设置私有频道成员列表。只有在私有频道成员列表中的用户可以在私有频道中收发消息。
3. IMLib SDK 增加 25107 错误码。服务端可控制消息是否支持他人（非发送者本人）撤回。如果服务端已设置为仅限发送者本人撤回，则在他人尝试撤回消息时报这个错误。

问题修复：

1. 修复了 `RongCoreClient#getMessages` 接口因调用了废弃接口，当出现错误时没有回调的问题。
2. 修复了快速切换账号时，异步关闭数据库可能导致的事务异常。
3. 修复了 IMKit SDK 在通知栏仅可展示有限数量通知时通知顺序错乱的问题。
4. 修复了接收消息 `onReceived` 方法的 `haspackage` 与 `isOffline` 参数错误的问题。
5. 修复了 IMKit SDK 媒体消息不支持消息扩展的问题。

5.2.3.1 Dev

发布日期：2022/06/15

功能优化：

1. IMKit SDK 多选删除消息行为优化为同时删除本地与远端消息。

问题修复：

1. 修复 5.2.3 版本上 so 引用不正确的问题。

5.2.3 Dev

发布日期：2022/06/08

新增功能：

1. IMKit SDK 支持关闭本地通知的提示音或震动。
2. IMKit SDK 支持关闭表情面板中内置的 Emoji表情。
3. IMKit SDK 高德地图模块重构，推出新 `locationKit` 插件。**注意：**旧版 SDK 升级后，原有地图插件即失效，请重新集成地图插件。
4. IMKit 支持修改会话页面删除消息操作的默认行为。支持配置为在删除消息时同时删除本地与服务端的消息。
5. IMLib SDK 超级群消息撤回支持消息时支持通过设置 `isDelete` 参数同时删除发送端与接收端的原始消息数据。

功能优化：

1. 录制语音消息与小视频消息时，其他会话有新消息时默认不提醒。

问题修复：

1. 修复了若干 BUG。

5.2.2 Dev

发布日期：2022/05/05

新增功能：

1. 增加了新的全局免打扰功能接口 setNotificationQuietHoursLevel，原接口 setNotificationQuietHours 废弃仍然可以正常使用。
2. 新增了会话免打扰枚举 RCPushNotificationLevel，设置项包括：所有消息都通知、未设置（默认为所有消息都通知）、@消息通知、@指定用户通知、@所有人通知、所有消息都不通知，其中未设置、@消息通知设置项为老版本支持逻辑可兼容老版本，其他设置项需要升级到此版本后才能支持。
3. 针对超级群会话增加了默认免打扰状态设置接口 setUltraGroupConversationDefaultNotificationLevel
4. 针对超级群会话增加了查询免打扰默认状态接口 getUltraGroupConversationDefaultNotificationLevel
5. 针对超级群下指定频道增加了默认免打扰状态设置接口 setUltraGroupConversationChannelDefaultNotificationLevel

6. 针对超级群下指定频道增加了查询免打扰默认状态接口 getUltraGroupConversationChannelDefaultNotificationLevel
7. 增加了连接 IM SDK 后超级群会话信息同步完成的回调功能 setUltraGroupConversationListener
8. 增加了获取指定超级群下所有频道的未读消息总数接口 getUltraGroupUnreadCount
9. 增加了获取超级群会话类型的所有未读消息数接口 getUltraGroupAllUnreadCount
10. 增加了获取超级群会话类型的@消息未读数接口 getUltraGroupAllUnreadMentionedCount

问题修复：

1. 修复了若干 BUG

5.2.1 Dev

发布日期：2022/03/25

新增功能：

1. 超级群消息结构中增加消息已被修改标识。
2. 调整了超级群获取服务端历史消息接口 getMessages 获取历史消息条数的上限。调整后最多可获取 100 条。
3. 增加 IM 聊天室与 RTC 音视频房间绑定接口。创建绑定关系后，如果 RTC 房间仍存在，则服务端会阻止 IM 聊天室房间自动销毁。
4. 适配了 Android 12 系统，详见 Android 端推送集成文档。

问题修复：

1. 修复了超级群撤回消息的回调结果中 operatorId（撤回该条消息的操作用户）可能出现错误的问题。
2. 修复了 FCM 推送偶尔注册成 RongPush 的问题。
3. 修复了敏感词回调不带 channelid 的问题。

5.2.0 Dev

发布日期：2022/03/01

新增功能：

1. 新增了融云超级群会话，支持无成员上限的群组聊天
2. 新增了超级群频道功能，可在超级群会话下创建多个频道，成员可随意在不同群频道中发送消息，但不同频道间的消息相互隔离。
3. 消息扩展功能，可设置的 Value 值长度改为 4096 个字符。

5.1.8 Dev

发布日期：2022/01/20

新增功能：

1. 适配 Emoji 13.1，支持表情组合
2. 针对多端操作聊天室同一属性时，偶现属性设置失败的问题，增加单独错误码 23431

问题修复：

1. 防止 IPC 进程重启后，IRTCHeartbeatListener 回调事件丢失
2. 修复了偶现的空指针问题
3. 升级 IMKit 使用的 androidx.room 至 2.4.0，解决 arm64 架构编译问题
4. 修复了会话页面当没有和会话绑定时，有可能触发刷新事件导致 mProcessor 为 null 的问题

5.1.7 Dev

发布日期：2021/12/14

新增功能：

1. 针对小米、华为推送通道，在发送单条消息时，可设置推送时通知栏右侧显示的图片内容
2. 针对 FCM 推送通道通知消息方式，支持设置推送标题、通知栏右侧图片内容及推送 ChannelId

问题修复：

1. 修复了若干 BUG

5.1.6 Dev

发布日期：2021/11/05

问题修复：

1. 优化了重连触发机制
2. 规避了 Google Play 检测出来的“不安全的加密模式”问题
3. 修复了在聊天室同步消息过程中，断开连接，再次连接并加入聊天室后，同步状态没有复位，导致聊天室消息一直没有同步的问题。

4. 修复了捕获 AudioManager setMode 方法，调用动态代理异常的崩溃

5.1.5 Dev

发布日期：2021/09/24

新增功能：

1. 新增了清除标签对应会话的未读数接口 clearMessagesUnreadStatusByTag
2. 新增删除标签对应的会话接口 clearConversationsByTag
3. 新增获取会话的置顶状态接口 getConversationTopStatus
4. 新增获取某个会话内的指定消息类型未读消息数接口 getUnreadCount

问题修复：

1. 修复了 PushManager 内 mConfigCenter 偶现空指针的问题。
2. 修复了 saveTextMessageDraft 和 clearTextMessageDraft 返回值错误的问题
3. 废弃了设置语音消息最大时长的方法，避免超过消息大小限制后导致连接断开的问题
4. 修复了预览大图时候，由于 copy bitmap 导致的内存溢出的问题
5. 修复了媒体文件下载，没有 Handler.encode 导致 SightMessage 的 base64 入库前被置空的问题
6. 修复了部分 LG 手机调用 TelephonyManager 的 getNetworkOperatorName，getNetworkOperatorName 内部报空指针的问题
7. 修复了 HQVoiceMessageHandler.encodeMessage 时，localPath 无效导致空指针异常
8. 修复了会话页面加载 GIF 失败后显示空白的问题

5.1.4 Dev

发布日期：2021/08/11

新增功能：

1. 新增了批量设置和删除聊天室属性能力
2. 新增了用户未加入聊天室时，支持获取聊天室属性信息
3. 新增了用户加入、退出聊天室回调能力，需要客户开通后支持，可提交工单申请开通

问题修复：

1. 修复了实时位置共享时无法使用 2D 地图的问题

5.1.3 Dev

发布日期：2021/06/25

新增功能：

1. MessageContent 类中增加 extra
2. 下载媒体文件时增加 messageId 唯一标识
3. ImageMessage 增加 name 映射，如果用户设置了图片的名称，会按照用户的名称存储，否则自动生成名称
4. FCM 推送 SDK 版本升级到 22.0.0 版本
5. 发送单条消息时，针对华为推送通道，支持设置 LOW、NORMAL 级别消息
6. 优化引用消息结构，增加被引用消息 ID 属性 ReferMsgUid

问题修复：

1. IMKit SDK 会话列表和会话页面，头像显示失败时，增加默认显示逻辑
2. IMKit SDK 拦截不存储不计数的消息，废弃 MessageHandler 接口
3. 修复了推送服务开发者没有配置时，默认使用 RongPush

5.1.2 Dev

发布日期：2021/05/21

新增功能：

1. IMLib SDK 支持了媒体消息中文件分片下载功能
2. 升级华为推送到 5.1.1.301，魅族推送到 4.0.7 版本
3. IMLib SDK 增加了按时间搜索本地会话中历史消息功能

问题修复：

1. 修复了视频下载时缩略图显示比例不对的问题
2. 修复了点击视频播放界面无法关闭的问题
3. 语音消息录音时，增加是否正在通话的判断，如正在通话则不进行录音

5.1.1 Dev

发布日期：2021/04/09

新增功能：

1. 新增了会话标签设置功能
2. 新增了批量导入本地消息数据接口
3. 群会话中有 @我消息时，进入群会话界面支持点击跳转到 @ 消息功能
4. IMLib SDK 新增了图片缩略图尺寸设置能力

问题修复：

1. 发送横屏的小视频消息，压缩后保存系统相册显示异常问题
2. 修复了引用的链接消息点击无法打开的问题
3. 修复了 RongCoreClient 中，registerModule 可能出现的空指针问题，抛出异常改为 Exception
4. 修复了 IMKit 动态集成 ConversationFragment 时，加号 ICON 无法显示的问题

5.1.0 Dev

发布日期：2021/03/05

问题修复：

1. SDK 中日志信息修改成英文提示

5.0.0 Dev

发布日期：2021/01/19

新增功能：

1. 发送消息，支持设置推送模板 ID，模板 ID 及模板中内容在"控制台-自定义推送文案"中进行创建。设置后根据目标用户通过 RongIMClient 中的 setPushLanguageCode 设置的语言环境，匹配模板中设置的语言内容进行推送，未匹配成功时使用融云默认内容进行推送。
2. 消息撤回功能支持推送属性设置 MessagePushConfig 和 isDisableNotification

功能优化：

1. 对 IMKit SDK 进行了重构，提升了 UI 品质及用户体验
2. IMKit SDK UI 界面适配了阿拉伯语
3. IMLib SDK 按模块进行代码拆分，提升了初始化速度，减少了不必要的内存占用
4. 对 Android 11 进行了兼容适配
5. 对 so 文件进行了加固处理
6. 默认使用 AndroidX 架构

更新日志（稳定版） 更新日志（稳定版）

更新时间:2024-08-30

> 提示
> 仅 Android/iOS 平台的 IM SDK 存在开发版、稳定版区分。

设计原则

Android 平台提供稳定版 IMLib SDK 和 IMKit。

- SDK 的稳定版本在线上运行时长、稳定性、使用量等方面满足一定的指标要求。
- 更强调稳定性，而非引入新功能。

发布周期与版本号规则

IM SDK 在 5.4.X 版本前后版本号规则不同。5.4.X 后更方便区分开发版、稳定版。

- 从 5.4.X 版本及以后，Stable 版本占用第二位版本号。第二位为偶数均为开发版，第二位为奇数均为稳定版。例如，5.5.X 为稳定版 SDK 使用的版本号。
- 在 5.4.X 之前，稳定版版本号规则不固定。
- 融云会监控 Stable 版本客户的使用状况，定期更新稳定版，最长更新周期为两个月。

维护说明

1. 如果融云正在积极开发的大版本号（当前为 5.X）下发布了新的 Stable 版本，我们建议使用 Stable 版本的客户升级到新的 Stable 版本。新的 Stable 版本发布后，历史稳定版维护力度相应降低。
2. 针对已不再积极开发的历史大版本（2.X、4.X）SDK，融云仅维护一个 Stable 版本。请仍使用 2.X、4.X 版本的客户尽快升级到相应的 Stable 版本，或者考虑升级到 5.X 系列的 SDK。

> 5.7.x 系列是基于 5.8.0 Dev 版本推出的稳定版本

5.7.0 Stable

发布日期：2024/05/23

优化功能：

- 小米海外的推送模块停止发布。
- 更新了荣耀推送。
- 更新了高德地图的版本，不再支持 x86 架构。

问题修复：

- 修复了获取本地时间与服务器时间的差值如果为 0，下次获取真实值需要 60s 的问题。
- 修复了 FCM 透传消息没有赋值 `ChannelId` 的问题。
- 修复了相册预览页面中提示语的错误。
- 增加了图库 gif 图过大提示语，告知用户无法发送过大的 gif 文件。
- 增加了一个会话消息删除失败的弹窗提示。当用户在没有网络连接的情况下尝试删除会话消息时，系统会弹出此弹窗，告知用户删除操作失败。
- 修复了键盘输入消息后，点击文字语音切换按钮，右边没有显示更多选择按钮的问题。
- 修复了断开网络后，会话聊天页面显示空的问题。

> 5.5.X 系列是基于 5.4.7 Dev 版本推出的稳定版本。

5.5.4 Stable

发布日期：2024/07/10

优化功能：

- 适配了 Android 14 广播注册。

问题修复：

- 修复了翻译模块偶现的 `RejectedExecutionException` 的问题。
- 修复了单进程模式下，初始化后立即在子线程内调用连接方法无回调的问题。

5.5.3 Stable

发布日期：2024/02/29

优化功能：

- 推送功能依赖的 vivo 推送 SDK 从 3.0.0.4 升至 3.0.0.7。如果使用推送（旧版）集成方案，请注意同时升级 vivo 推送 SDK。官网下载包内附带的 vivo 推送 SDK 已更新。
- 推送功能依赖的 OPPO 推送 SDK 从 3.1.0 升至 3.4.0。如果使用推送（旧版）集成方案，请注意同时升级 OPPO 推送 SDK。官网下载包内附带的 OPPO 推送 SDK 已更新。

问题修复：

- 修复会话页面内存泄漏的问题。
- 修复极端情况下，多线程导致的频繁回调 `SUSPEND` 连接状态的问题。
- 修复重复创建 `NaviObserver` 导致 SDK 重连多次的问题。

5.5.2 Stable

发布日期：2023/12/08

优化功能：

- 消息推送属性（`MessagePushConfig`）中的 AndroidConfig 增加荣耀推送配置。
- 补全初始化配置 `InitOption` 中区域码（AreaCode）枚举值。

问题修复：

- 修复在部分机型上点击 FCM 透传方式推送通知后，通知未自动取消的问题。
- 修复 `getConversationList` 无回调，多进程情况下线程被阻塞问题。

5.5.1 Stable

发布日期：2023/10/20

问题修复：

- 修复数据库加密的问题。

5.5.0 Stable

发布日期：2023/09/08

优化功能：

- 支持的 Android 最低版本提升为 5.0，要求 `minSdkVersion` 最低为 21。

问题修复：

- 修复 IMKit 会话列表的下拉加载框经常出现一直转无法关闭的情况。
- 修复 IMLib 在处于后台一段时间再回前台偶先连接不上的问题。

> 5.3.X 系列是基于 5.3.5 Dev 版本推出的稳定版本。5.3.X 系列稳定版本现已过时，请尽快升级到最新 Stable 版本，或最新 dev 版本。

5.3.8 Stable

发布日期：2023/07/07

问题修复

- 优化获取指定时间戳前或后消息接口，以实际传入时间戳为准，SDK 内部不做时间戳 +1 或-1 处理
- 修复 `getConversationList` 无回调，多进程情况下线程被阻塞问题
- 修复会话草稿在杀掉 App 进程后再启动的情况下后未恢复的问题
- 修复会话中存在草稿的情况下杀死 App 后重新进入应用，该条会话用户信息为空的问题
- 修复通过通讯录选择好友发消息后，返回消息列表，刚发的消息变成了草稿的问题

5.3.7 Stable

发布日期：2023/07/07

5.3.7 Stable 版本号已废弃，请勿使用。

5.3.6 Stable

发布日期：2023/05/05

新增功能

1. IMLib/IMKit SDK 支持在消息推送属性配置中指定 vivo 推送 category 参数。
2. IMLib/IMKit SDK 支持在消息推送属性配置中指定华为推送 category 参数。

> 5.1.9 稳定版是 5.X 系列首个稳定版本。5.1.9 Stable 版本已过时，请尽快升级到最新 Stable 版本，或最新 dev 版本。

5.1.9 Stable

发布日期：2022/08/22

新增功能：

1. 适配 Android 12
2. 增加获取@未读消息列表接口

客户端 API 参考 IMKit

更新时间:2024-08-30

以下是 IMKit 5x 的 API 参考文档（javadoc）：

- IMKit（界面库）

IMLib

即时通讯基础能力库 IMLib 5.X 分为 6 个子模块：

LibCore（基础核心功能），chatroom（聊天室），customservice（客服），discussion（讨论组），locationlib（实时位置），publicserivce（公众号）

IMLib 核心类 `RongIMClient` 直接调用各个子模块的接口，内部没有任何具体实现。

以下是 IMLib 5x 的 API 参考文档（javadoc）：

- IMLib（通讯库）
- IMLibCore（通讯核心库）
- ChatRoom（聊天室库）

下列 IMLib API 不再推荐使用，保留文档仅作为作参考：

- CustomService（客服）
- PublicService（公众号）
- Discussion（讨论组）