场景融合
实时社区
Android 1.X

2024-08-30

实时社区概述

更新时间:2024-08-30

实时社区 Community Demo 是基于融云 IM 超级群能力实现的**开源Demo**，组件功能需要配合 业务服务器 共同完成。实时社区是一种新型的社交形态，用户可以建立或者加入自己感兴趣的社区，每个社区下可以创建独立的频道，频道中的消息独立，互不影响，并且每个社区对于人数不设定上限。

社区核心场景

1. **传统社区升级：** 国内的社交软件、论坛、基于兴趣、游戏、粉丝的社群等都有升级成实时社区形态的强烈需求，打造类Discord的实时社区
2. **游戏社区：** 用户可以加入感兴趣的游戏社区，通过社区中游戏资讯、攻略教学、赛事预告等主题频道获取不同的内容信息，可以在社区里交流心得、组队开黑、游戏实时语音交流等
3. **兴趣社区：** 用户基于共同话题与爱好构建社区，覆盖动漫漫画、音乐、科技、语言、电影、加密货币等话题，用户可以在不同频道中实时在线交流
4. **粉丝社区：** 基于明星、网红、KOL构建的粉丝社群场景，无成员上限，支持大规模用户的流畅沟通、实时互动

主要功能

创建&加入社区： 用户可创建自己的社区，或者从发现社区-点击加入 加入自己感兴趣的社区
文字频道： 可以在社区中发表文字、视频、图片消息
消息互动： 相比于传统社区，消息及时，相对于传统群组，人数无上限
社区成员管理： 可针对社区成员进行禁言，封禁，踢出等操作
发现模块： 可以查看自己感兴趣的社区列表
私聊互动： 可以和社区中兴趣相投的人一对一私聊
推送通知： 支持社区全局设置和单个频道设置
系统消息： 接收社区中的系统消息

功能列表

Community Demo 的功能包括但不限于以下内容：

功能	描述	支持版本
创建社区	可创建自己感兴趣的社区	1.0.0
创建分组	在社区下可创建属于自己的频道分组	1.0.0
创建频道	在社区下可创建属于自己的频道	1.0.0
社区通知管理	支持社区全局设置和单个频道设置	1.0.0
社区成员管理	可针对社区成员进行禁言，封禁，踢出等操作	1.0.0
修改用户社区昵称	可修改当前所有用户在当前社区的昵称	1.0.0
修改社区封面	可自主设置社区的封面	1.0.0

功能	描述	支持版本
修改社区头像	可自主设置社区的头像	1.0.0
成员验证	可修改用户申请加入社区的方式，例如：是否需要审核	1.0.0
分组管理	可编辑分组，移动分组，删除分组	1.0.0
频道管理	可编辑频道，移动频道，删除频道	1.0.0
发送消息	目前仅支持文字频道，可发送文字、表情、图片、视频消息	1.0.0
未读消息提示	可实时显示每个频道的未读消息数目	1.0.0
输入状态提示	支持实时显示当前频道中的输入状态	1.0.0
标注消息	支持消息标注，可快速定位到标注消息	1.0.0
快速回到底部	支持一键下滑到底部	1.0.0
重发消息	支持消息的重新发送	1.0.0
@用户	支持@当前社区的用户	1.0.0
消息撤回	支持对已经发送的消息进行撤回	1.0.0

实时社区 App 服务端

实时社区客户端 SDK 仅依赖融云提供的 IM 超级群的能力。App 服务端指您自己的 **App 业务服务器**（App server），即为您自身的业务提供接口的后端，您需要自行实现。

> 您可以使用 IM 服务端 API 所提供的全部能力构建您的实时社区 App 后台服务系统。了解如何使用 IM + RTC 全部服务端能力，请前往即时通讯服务端文档 · 实时音视频服务端文档 »

为协助您搭建 App 业务服务端，我们提供了 App 服务端示例项目（Demo server）。

> 前往实时社区服务端开发指南 »

资源与支持

- **RC RTC 应用**

  展示如何使用场景化 SDK 搭建语聊房、语音电台、视频直播、音视频通话、实时社区等互动社交场景，提供 Android 端和 iOS 端的源码。预置了融云 App Key 和对应的测试服务器 URL，无需部署服务器即可运行。

  - RC RTC Android 源码 (Github)
  - RC RTC Android 源码 (Gitee)

- **RC RTC 配套 App server**

  官网体验应用 RC RTC 配套的 App server，基于 java。

  - RC RTC 配套 App server (Github)
  - RC RTC 配套 App server (Gitee)

- **Community Demo 模块**

  您可以直接使用实时社区 Demo 提供的代码。您需要下载 RC RTC Android 源码 (Github)，然后在您的项目中导入 Demo 代码模块，具体步骤如下：

1. 下载 RC RTC 项目后，在 Android Studio 中导入模块，模块路径为：{RC RTC 项目所在路径}/rongcloud-scene-android-demo/business/community。

2. 打开您自己项目根目录下的 build.gradle 文件，从 rongcloud-scene-android-demo 项目根目录下的 build.gradle 中，复制 ext { ... } 相关配置自己的 build.gradle 中。

3. 打开您自己 App 工程下的 build.gradle 文件，引入实时社区模块：

```
implementation project(":business:community")
```

4. 在自己项目的 Application 中加入如下代码：

```
public class RCApplication extends Application {
@Override
public void onCreate() {
// 注册相关模块
ModuleManager.manager().register(new CommunityModule());
// 相关 key 可以配置到 build.gradle 中，也可以直接写在这里,不用的功能可以写空字符串
AppConfig.get().init(
BuildConfig.APP_KEY, // 融云 appKey
BuildConfig.UM_APP_KEY, // 友盟，可选
BuildConfig.BASE_SERVER_ADDRES, // 服务器地址
BuildConfig.BUSINESS_TOKEN, // 自己服务器可以去除businessToken验证,可选
"", // 渠道，可选
false,
null
);
}
}
```

超级群服务配置

更新时间:2024-08-30

您可以在控制台 的超级群服务页面控制是否为您的应用启用超级群服务，并修改默认服务配置。

修改服务配置

- **用户未加入超级群是否能获取历史消息**：默认用户不在超级群中时，不能拉取历史消息。开启后不在超级群中的用户也可以拉取历史消息。
- **新用户加入群后是否拉取加入群之前的消息**：默认用户加入超级群后，不能拉取之前的历史消息。开启后可拉取加入群之前的历史消息。
- **超级群消息云端存储**：默认存储 7 天。支持付费开通为 3 个月、6 个月、 1 年
- **超级群默认推送频率设置**：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。@ 消息不受此限制。

> 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，默认普通消息累计 2 条时才会触发推送。@ 消息无此限制。

运行示例项目（Demo）

更新时间:2024-08-30

场景化 RC RTC 开源 Demo 项目(Github · Gitee) 中已加入实时社区（RCSceneCommunity）模块，并实现了相关功能。您可以试运行 Demo，快速体验实时社区产品功能。

申请试用 Token

为帮助您快速体验，Demo 项目提供测试配置（App Key、测试 Token、测试服务器地址）。您无需自行创建应用及获取 App Key，即可完成体验。

测试配置中的 App Key、测试服务器地址已内置在 Demo 中。请前往以下地址申请测试所需的试用 Token。

https://rcrtc-api.rongcloud.net/code/

配置测试环境

请在项目 app 下 build.gradle 文件中指定 BUSINESS_TOKEN 字段为您申请的 Business Token。

测试环境下登录时，验证码输入任意6位数字即可。

```
productFlavors {
// 开发环境
develop {
dimension 'environment'
applicationId "cn.rongcloud.voiceroomdemo.dev" // 配置不同的包名，确保切换环境时不会因为缓存导致出现问题
buildConfigField("String", "APP_KEY", "\"pvxdm17jpw7ar\"")//
buildConfigField("String", "BASE_SERVER_ADDRES", "\"https://rcrtc-api.rongcloud.net/\"")
buildConfigField("String", "BUSINESS_TOKEN", "\"这里是测试服务器token，需要申请 https://rcrtc-
api.rongcloud.net/code\"")

buildConfigField("String", "SENSORS_URL", "\"神策地址\"")//可选
buildConfigField("String", "UM_APP_KEY", "\"这里是友盟AppKey\"")//可选
buildConfigField("String", "BUGLY_ID", "\"这里是Bugly的Id\"")//可选
buildConfigField("String", "MH_APP_KEY", "\"这里是美狐美颜SDK的appkey，要想体验美颜功能需要去美狐官网申请
http://www.facegl.com/\"")//可选
buildConfigField("Boolean", "INTERIAL", "false")// 区分国内/国际 国内环境不显示UI 选择地区
manifestPlaceholders = [
APP_NAME : "@string/app_name_test",
// hifive音乐服务的 appid和servercode
HIFIVE_APPID : "替换您hifive音乐服务的appid",//可选替换
HIFIVE_SERVERCODE: "替换您hifive音乐服务的servercode"//可选替换
]
}
```

快速上手

更新时间:2024-08-30

> 实时社区 Community Demo 是基于融云 IM 超级群能力实现的开源 Demo。客户端 Demo 的组件功能需要配合业务服务器共同完成

环境要求

- **Android Studio**：4.0+
- **Android**：5.0 及以上
- **Java**：java 11

集成IMLib SDK

- gradle 远程依赖

远端依赖

1. 在 project 的 build.gradle 添加融云仓库

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

2. module 的 build.gradle 文件中的 dependencies 节点添加如下代码：

```
dependencies {
......
// Demo 中部分 UI 使用了 IMKit 中的工具类。推荐引入 IMKit（已含 IMLib），并使用最新版本
implementation 'cn.rongcloud.sdk:im_kit:5.2.2'
// 如果希望自行实现 UI，可以仅依赖 IMLib，建议使用最新版本
//implementation 'cn.rongcloud.sdk:im_kit:5.2.2'
......
}
```

3. **gradle** 配置完成后，重新 build 项目。

混淆配置

- 根据实际项目的依赖情况，选项对应的混淆配置：
  1. 项目中依赖 imlib ，则添加 imlib 的混淆配置即可
  2. 项目中依赖 imkit ，则添加 imkit 的混淆配置即可

- 混淆配置有交叉重复的部分，可适当去重。

1. IMLib 的混淆配置

```
# imkit 的混淆配置
-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.**

# 当代码中有继承 PushMessageReceiver 的子类时，需 keep 所创建的子类广播
# 把 io.rong.app.DemoNotificationReceiver 改成子类广播的完整类路径即可.
-keep class io.rong.app.DemoNotificationReceiver {*;}

-ignorewarnings
```

2. IMKit 的混淆配置

```
# imkit 的混淆配置
-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.**

# 下方混淆使用了Location包时才需要配置，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.**{*;}
-keep class com.amap.api.services.**{*;}
-keep class com.autonavi.**{*;}

# 当代码中有继承 PushMessageReceiver 的子类时，需 keep 所创建的子类广播
# 把 io.rong.app.DemoNotificationReceiver 改成子类广播的完整类路径即可.
-keep class io.rong.app.DemoNotificationReceiver {*;}

-ignorewarnings
```

版本依赖说明

Community Demo 依赖融云 IMLib，依赖版本如下。

依赖组件	版本
IMLib	5.2.2 及以上

参考资源

实时社区 Demo 的相关功能基于融云 IMLib SDK。

> **更多关于超级群的内容可以参考 IMLib SDK 5.X 开发者文档。**
> https://doc.rongcloud.cn/im/Android/5.X/noui/ultragroup/intro

初始化

更新时间:2024-08-30

Community Demo 的初始化依赖于融云 IMLib 的初始化。

实时社区 初始化

> 在初始化前，请确保已完成以下操作：
>
> - 您已开通融云开发者账号，并申请了融云 App Key。
> - 建议在 Application 中初始化，实时社区初始化 依赖于 IMLib 的初始化。

```
/**
* 初始化 AppKey
* 注意：实时社区依赖于 IM 的初始化
*
* @param context 当前应用的 application
* @param appKey 开发者申请的 AppKey
*/
void init(Application context, String appKey);

// imlib 初始化代码示例
String process = UIKit.getCurrentProcessName();
if (!getPackageName().equals(process)) {
// 非主进程不初始化 避免过度初始化
return;
}
RongCoreClient.init(this, APP_KEY);
```

连接融云服务

```
// imlib 中 RCCoreClient 的连接接口说明
/**
* 连接融云服务器
* 注意：如果使用 RCCoreClient
*
* @param appToken 从服务器获取的 imToken
* @param connectCallback 结果回调
*/
RongCoreClient connect(String appToken, ConnectCallback connectCallback)

// 具体调用示例
RongCoreClient.connect(accout.getToken(), new IRongCoreCallback.ConnectCallback() {
@Override
public void onSuccess(String t) {
KToast.show("connect success");
}

@Override
public void onError(IRongCoreEnum.ConnectionErrorCode e) {
String info = "connect fail：\n【" + e.getValue() + "】" + e.name();
Log.e("ConnectActivity", info);
KToast.show(info);
setTitle(accout.getName() + " 连接失败");
}

@Override
public void onDatabaseOpened(IRongCoreEnum.DatabaseOpenStatus code) {

}
}
);
```

社区的创建和解散

更新时间:2024-08-30

本章主要介绍社区的创建流程

实时社区是使用融云即时通讯 IMLib SDK 构建的场景化业务，主要基于 IMLib 的超级群功能。IMLib SDK 负责提供即时通讯基础业务能力，本身并不直接提供创建社区的 API。

在实时社区 Demo 中，创建社区依赖于服务端接口实现，所以需要和后台协商具体的参数，这里主要介绍 Demo 中如何调用接口创建社区。

> **注意**
>
> - Demo中，服务端限制每个人最多只可创建三个社区
> - Demo中，服务端限制上传图片文件最大不可超过5M
> - Demo内置了10种默认的背景，会随机选择默认背景，如果没有自己手动选择背景，那么以默认背景为主，且社区的背景和头像默认是一致的

创建社区

创建社区接口，主要需要的参数为社区名、社区背景图片地址

上传图片到文件服务器

将本地图片上传到文件服务器，获取该图片的服务器地址

```
//file 图片文件
//ApiConfig.FILE_UPLOAD 为图片服务器上传接口
FileBody body = new FileBody("multipart/form-data", file);
OkApi.file(ApiConfig.FILE_UPLOAD, "file", body, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
///得到图片在服务器的地址
String url = result.getBody().getAsString();
}

@Override
public void onError(int code, String msg) {
super.onError(code, msg);
//TODO Code
}
});
```

创建接口

```
Map<String, Object> params = new HashMap<>(2);
params.put("name", name); //name 为社区名称
if (!TextUtils.isEmpty(url))
params.put("portrait", url);//url 为图片在服务器的地址
OkApi.post(CommunityAPI.Community_Create, params, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
if (result.ok()) {
//创建成功 TODO Code
} else {
//创建失败 TODO Code
}
}
});
```

解散社区

解散社区接口，主要需要的参数为社区ID

```
//CommunityAPI.Community_delete 解散社区接口
OkApi.post(CommunityAPI.Community_delete + uid, null, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
if (result.ok()) {
//解散成功 TODO
} else {
//解散失败 TODO
}
}
});
```

分组频道管理

更新时间:2024-08-30

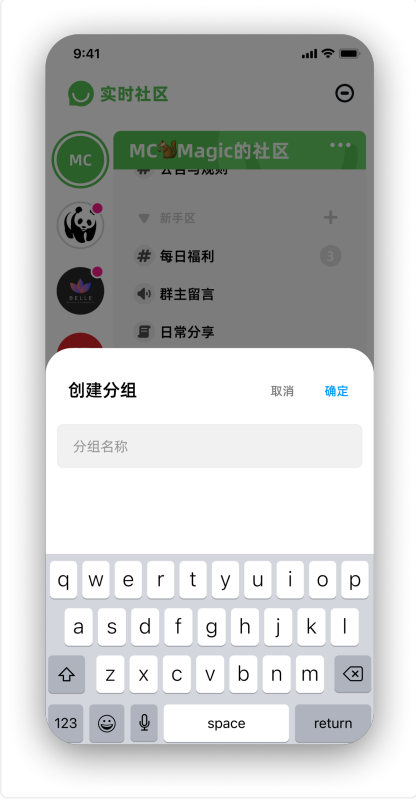
本章主要介绍社区的分组和频道的管理

- 社区的分组和频道的管理主要是依赖于业务服务器，所以分组和频道的管理都是通过业务服务器接口来完成的
- 分组和频道信息发生了变更，会收到相关回调，然后可以根据回调取刷新UI，可以参考关键类 UltraCenter

分组管理

分组管理功能可以针对社区的分组做出：创建、重命名、调整顺序、删除等操作

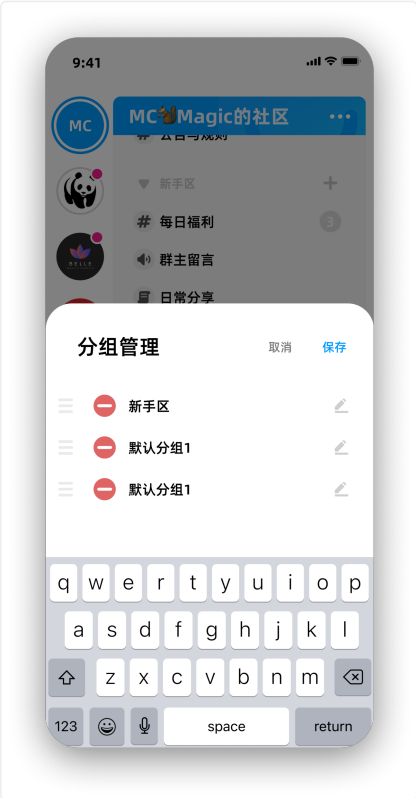
创建分组



```
//CommunityAPI.Community_create_group 为创建分组的接口地址
Map<String, Object> params = new HashMap<>();
params.put("communityUid", uid);//社区ID
params.put("name", groupName);//分组名称
OkApi.post(CommunityAPI.Community_create_group, params, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
if (result.ok()) {
//创建成功 TODO CODE
} else {
//创建失败 TODO CODE
}
}
});
```

分组重命名和调整顺序

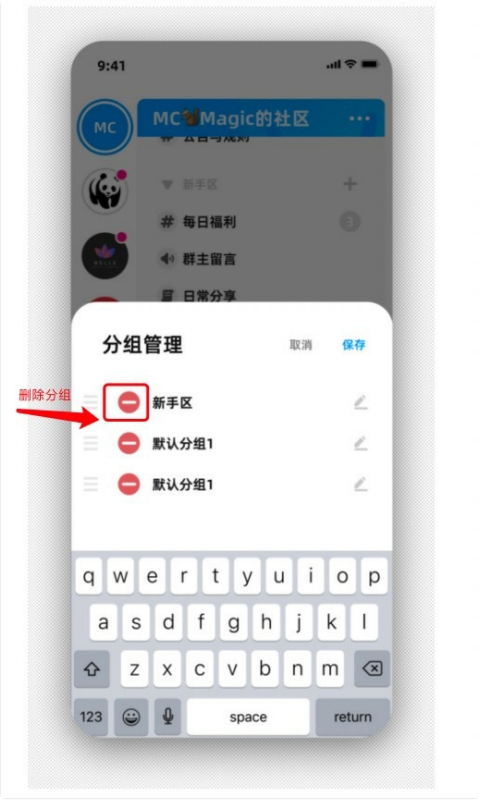
分组信息的调整，包括重命名和顺序调整都是通过调用保存社区详情接口，进行全量更新的



```
//communityDetailsBean 当前操作的社区的详情
//CommunityAPI.Community_save_all 为保存社区详情，这里也作为全量更新社区信息
communityDetailsBean.setGroupList(groupBeanList);//如果更新分组，那么设置当前的调整完以后的分组信息给社区详情对象就可以了
communityDetailsBean.setUpdateType(Constants.UpdateType.UPDATE_TYPE_GROUP.getUpdateTypeCode());//更新社区详情的类型设置为更新分组
String jsonStr = GsonUtil.obj2Json(communityDetailsBean);
Map<String, Object> params = GsonUtil.json2Map(jsonStr, new TypeToken<HashMap<String, Object>>()
{}.getType());
OkApi.post(CommunityAPI.Community_save_all, params, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
if (result.ok()) {
//更新社区详情成功
} else {
//更新社区详情失败
}
}
});
```

删除分组

删除分组和创建分组一样，也是通过业务服务器接口来实现的



```
//CommunityAPI.Community_delete_group 删除社区分组接口
//groupId 分组ID
OkApi.post(CommunityAPI.Community_delete_group + groupId, null, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
if (result.ok()) {
//删除成功
} else {
//删除失败
}
}
});
```

频道管理

频道管理功能和分组管理功能类似，功能分别为：创建频道、删除频道、调整频道所在分组、重命名

创建频道

```
//CommunityAPI.Community_create_channel 为创建频道的接口地址
Map<String, Object> params = new HashMap<>(3);
params.put("communityUid", uid);//社区id
if (null != selectedGroup)
params.put("groupUid", selectedGroup.uid);//频道所属的分组id
params.put("name", channelName);//频道名称
OkApi.post(CommunityAPI.Community_create_channel, params, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
if (result.ok()) {
//创建频道成功 TODO Code
} else {
//创建频道失败 TODO Code
}
}
});
```

频道重命名和调整顺序

频道信息的调整，包括重命名和顺序调整都是通过调用保存社区详情接口，进行全量更新的

```
//communityDetailsBean 当前操作的社区的详情
//CommunityAPI.Community_save_all 为保存社区详情，这里也作为全量更新社区信息
communityDetailsBean.setGroupList(groupBeanList);//那么设置当前的调整完以后的分组信息给社区详情对象就可以了
communityDetailsBean.setChannelList(channelBeanArrayList);////如果更新频道，那么设置当前的调整完以后的频道信息给社区详情对象就可以了
communityDetailsBean.setUpdateType(Constants.UpdateType.UPDATE_TYPE_CHANNEL.getUpdateTypeCode());//更新社区详情的类型设置为更新频道
String jsonStr = GsonUtil.obj2Json(communityDetailsBean);
Map<String, Object> params = GsonUtil.json2Map(jsonStr, new TypeToken<HashMap<String, Object>>()
{}.getType());
OkApi.post(CommunityAPI.Community_save_all, params, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
if (result.ok()) {
//更新社区详情成功
} else {
//更新社区详情失败
}
}
});
```

删除分组

删除频道和创建频道一样，也是通过业务服务器接口来实现的

```
//CommunityAPI.Community_delete_channel 删除社区分组接口
//groupId 分组ID
OkApi.post(CommunityAPI.Community_delete_channel + channelUid, null, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
if (result.ok()) {
//频道删除成功
} else {
//频道删除失败
}
}
});
```

社区资料编辑

更新时间:2024-08-30

本章主要介绍社区的资料编辑

社区资料编辑分为主要是2种，分别为：

- 业务服务器管理的社区的信息：头像、封面、社区名称、简介、接收系统消息的频道、默认进入的频道
- 社区默认免打扰模式，Demo 中会调用 IMLib SDK 接口进行管理

社区信息编辑

通过业务服务器接口，修改社区信息

```
/**
* 社区详情的更新是通过调用保存社区详情接口，进行全量更新的
*/
communityDetailsBean.setUpdateType(Constants.UpdateType.UPDATE_TYPE_ALL.getUpdateTypeCode());//设置更新类
型为全量更新
communityDetailsBean.setPortrait(url);//更新头像
communityDetailsBean.setName(name);//更新社区名称
String jsonStr = GsonUtil.obj2Json(communityDetailsBean);
Map<String, Object> params = GsonUtil.json2Map(jsonStr, new TypeToken<HashMap<String, Object>>()
{}.getType());
OkApi.post(CommunityAPI.Community_save_all, params, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
if (result.ok()) {
//修改成功 TODO Code
} else {
//修改失败 TODO Code
}

}
});
```

默认免打扰模式设置

```
/**
* 设置指定的超级群的默认通知级别
* 默认免打扰逻辑对所有群成员生效，由超级群的管理员进行设置
*
* @param targetId 社区 ID
* @param pushNotificationLevel 通知级别
*/
public void setUltraGroupConversationDefaultNotificationLevel(String targetId,
IRongCoreEnum.PushNotificationLevel pushNotificationLevel, IResultBack<Boolean> resultBack) {
ChannelClient.getInstance().setUltraGroupConversationDefaultNotificationLevel(targetId,
pushNotificationLevel, new IRongCoreCallback.OperationCallback() {
@Override
public void onSuccess() {
Logger.d("setConversationNotificationLevel：targetId:" + "targetId:" + pushNotificationLevel);
if (resultBack != null) {
resultBack.onResult(true);
}
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {
if (resultBack != null) {
resultBack.onResult(false);
KToast.show(coreErrorCode.getMessage());
}
}
});
}
```

社区通知管理

更新时间:2024-08-30

本文主要介绍社区的通知管理方式

> **注意**
>
> - 用户加入社区以后，社区的通知方式为社区的默认通知方式
> - 社区创建者修改社区默认通知方式，只对修改以后才加入社区的用户有影响，已经加入的用户不受影响
> - 通知优先级排名为:指定频道的推送级别>指定社区的推送级别>社区默认通知方式

社区通知

设置当前社区的默认通知方式

```
/**
* 设置指定的超级群的通知级别
*
* @param targetId
*/
public void setUltraGroupNotificationLevel(String targetId, IRongCoreEnum.PushNotificationLevel
pushNotificationLevel) {
ChannelClient.getInstance().setConversationNotificationLevel(Conversation.ConversationType.ULTRA_GROUP,
targetId, pushNotificationLevel, new IRongCoreCallback.OperationCallback() {
@Override
public void onSuccess() {
Logger.d("setConversationNotificationLevel：targetId:" + "targetId:" + pushNotificationLevel);
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {

}
});
}
```

频道通知

设置频道的默认通知方式

```
/**
* 设置超级群指定频道的推送级别
*
* @param targetId
* @param channelId
* @param pushNotificationLevel
*/
public void setChannelNotificationLevel(String targetId, String channelId,
IRongCoreEnum.PushNotificationLevel pushNotificationLevel, IResultBack<Boolean> resultBack) {
ChannelClient.getInstance().setConversationChannelNotificationLevel(Conversation.ConversationType.ULTRA_
GROUP, targetId, channelId, pushNotificationLevel, new IRongCoreCallback.OperationCallback() {
@Override
public void onSuccess() {
Logger.d("setChannelNotificationLevel：targetId:" + "targetId:" + pushNotificationLevel);
if (resultBack != null) {
resultBack.onResult(true);
}
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {
if (resultBack != null) {
resultBack.onResult(false);
KToast.show(coreErrorCode.getMessage());
}
}
});
}
```

社区成员管理

更新时间:2024-08-30

本文主要介绍创建者如何对于成员进行管理

创建者对于用户的管理主要通过业务服务器接口去更新用户信息，目前支持的能力为:

- 修改昵称
- 禁言
- 踢出

成员管理

通过业务服务器，可以拿到当前社区的成员列表，根据在线状态分为在线和离线两种，创建者可以对社区内的成员进行 ：修改社区昵称、禁言、踢出等操作

```
/// 社区用户修改接口 "mic/community/user/update"
Map<String, Object> params = new HashMap<>();
params.put("communityUid", uid);//社区ID
params.put("userUid", userId);//用户 ID
params.put(key, value);//key value 操作用户信息键值队 例如:key(name) value("Tom")
OkApi.post(CommunityAPI.Community_update_user_info, params, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
//TODO
}
});
```

社区审核模式

更新时间:2024-08-30

本文主要介绍如何通过业务服务器修改加入社区方式。

加入社区的方式一共分为 2 种，主要通过业务服务器接口进行管理：

- 审核后加入
- 无需审核即可加入

设置审核模式

这里修改社区的审核模式是依赖于社区的详情接口，所以直接修改审核字段，然后全量更新社区详情接口就可以了

```
CommunityDetailsBean communityDetailsBean =
CommunityHelper.getInstance().getCommunityDetailsBean().clone();
communityDetailsBean.setUpdateType(Constants.UpdateType.UPDATE_TYPE_ALL.getUpdateTypeCode());//设置更新模
式为全量更新
communityDetailsBean.setNeedAudit(s.verify ? Constants.NeedAuditType.NEED_AUDIT_TYPE.getNeedAuditCode()
: Constants.NeedAuditType.NOT_NEED_AUDIT_TYPE.getNeedAuditCode());//设置审核模式
String jsonStr = GsonUtil.obj2Json(communityDetailsBean);
Map<String, Object> params = GsonUtil.json2Map(jsonStr, new TypeToken<HashMap<String, Object>>()
{}.getType());
OkApi.post(CommunityAPI.Community_save_all, params, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
if (result.ok()) {
//修改审核模式成功 TODO Code
} else {
//修改审核模式失败 TODO Code
}

}
});
```

加入社区

更新时间:2024-08-30

本文主要介绍如何加入社区

业务服务器提供了申请加入社区的API接口，如果被申请加入的社区的审核模式为无须审核，那么直接加入，如果审核模式为需要审核，那么等待创建者的审核结果，审核结果的监听可以参考UlrtaGroupCenter

申请加入社区

申请加入社区接口，主要需要的参数为社区ID

```
//CommunityAPI.Community_Join 为申请加入社区的接口
String communityUid = CommunityHelper.getInstance().getCommunityUid();
OkApi.post(CommunityAPI.Community_Join + communityUid, null, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
//如果不需要审核，那么是直接加入成功的
if (result.ok() && result.getBody() != null) {
int asInt = result.getBody().getAsInt();
//根据asInt判断加入结果 TODO Code
} else {
//TODO Code
}
}
});
```

发送消息

更新时间:2024-08-30

本文介绍Demo是如何发送消息的。Demo中将IMLib的消息发送方法封装到消息管理类中MessageManager发送消息可供参考，如果需要自己实现，可以直接参考IMLib的超级群文档

- 如果使用MessageManager中的方法，那么前置条件必须调用了

```
MessageManager.get().attachChannel(targetId, channelId);
```

发送普通消息

接口说明

发送普通类型消息，即不需要上传多媒体文件的消息。

当调用该方法时，发送成功以后会自动插入本地适配器中并刷新会话列表。

所属类 : MessageManager

方法: sendMessage()

调用示例:

```
MessageManager.get().sendMessage(messageContent, editText, callback);
```

参数说明:

参数	类型	说明
messageContent	MessageContent	消息对象
editText	EditText	文字输入框，这里需要传入是为了用到IMkit中的@消息
callback	SendMessageCallback	发送消息的回调

代码示例：

```
TextMessage textMessage = TextMessage.obtain(editText.getText().toString());
MessageManager.get().sendMessage(textMessage, mView.getEditText(), new SendMessageCallback() {
@Override
public void onSuccess(Message message) {

}

@Override
public void onError(Message message, int code, String reason) {
Logger.e("onClickSend", code + reason);
KToast.show(reason);
}
});
```

发送图片消息

接口说明

当调用该方法时，发送成功以后会自动插入本地适配器中并刷新会话列表。

所属类 : `MessageManager`

方法: `sendImageMessage()`

调用示例:

```
MessageManager.get().sendImageMessage(messageContent, callback);
```

参数说明:

参数	类型	说明
messageContent	MessageContent	消息对象
callback	SendMessageCallback	发送消息的回调

代码示例：

```
MessageContent content;
if (PictureMimeType.isGif(mimeType)) {
content = GIFMessage.obtain(uri);
} else {
content = ImageMessage.obtain(uri, uri, sendOrigin);
}
MessageManager.get().sendImageMessage(content, new SendMessageCallback() {
@Override
public void onSuccess(Message message) {
Logger.d("ImagePlugin", "上传图片" + message.getContent() + "成功");
}

@Override
public void onError(Message message, int code, String reason) {
Logger.d("ImagePlugin", "上传图片" + message.getContent() + "失败");
}
});
```

发送媒体消息

接口说明

当调用该方法时，发送成功以后会自动插入本地适配器中并刷新会话列表。

所属类 : MessageManager

方法: sendMediaMessage()

调用示例:

```
MessageManager.get().sendMediaMessage(messageContent, callback);
```

参数说明:

参数	类型	说明
messageContent	MessageContent	消息对象
callback	SendMessageCallback	发送消息的回调

代码示例：

```
SightMessage sightMessage = SightMessage.obtain(activity, mediaUri, (int) duration / 1000);
if (DestructManager.isActive()) {
sightMessage.setDestruct(true);
sightMessage.setDestructTime((long) DestructManager.SIGHT_DESTRUCT_TIME);
}
MessageManager.get().sendImageMessage(sightMessage, new SendMessageCallback() {
@Override
public void onSuccess(Message message) {
//TODO Code
}

@Override
public void onError(Message message, int code, String reason) {
//TODO Code
}
});
```

删除消息

更新时间:2024-08-30

本文介绍Demo是如何发送消息的。Demo中将IMLib的消息删除方法封装到消息管理类中MessageManager发送消息可供参考，如果需要自己实现，可以直接参考IMLib的超级群文档

- MessageManager 中有一个重要的类 `WrapperMessage`,这个只是IMLib中`Message`的包装类，在SDK中是不存在的，是DEMO中自定义的

删除本地消息

接口说明

删除本地消息，删除成功以后会刷新会话列表。

所属类 : `MessageManager`

方法: `deleteMessage()`

调用示例:

```
MessageManager.get().deleteMessage(message);
```

参数说明:

参数	类型	说明
message	WrapperMessage	消息对象`Message`的包装类

代码示例：

```
///iMessage 是从适配器中拿到的item项
MessageManager.get().deleteMessage(iMessage);
```

撤销消息

更新时间:2024-08-30

本文介绍 Demo 是如何撤回消息的。Demo中将 IMLib 的消息撤回方法封装到消息管理类 MessageManager 中。如果需要自己实现，可以直接参考 IMLib 的超级群文档。

- MessageManager 中有一个重要的类 `WrapperMessage`,这个只是IMLib中`Message`的包装类，在SDK中是不存在的，是DEMO中自定义的

撤销消息

接口说明

当调用该方法时，撤销成功以后会刷新会话列表，其他用户会触发`IRongCoreListener.UltraGroupMessageChangeListener.onUltraGroupMessageRecalled`。

所属类 : `MessageManager`

方法: `recallMessage()`

调用示例:

```
MessageManager.get().recallMessage(message);
```

参数说明:

参数	类型	说明
message	WrapperMessage	消息对象`Message`的包装类

代码示例：

```
///iMessage 是从适配器中拿到的item项
MessageManager.get().recallMessage(iMessage);
```

编辑消息

更新时间:2024-08-30

本文介绍Demo是如何发送消息的。Demo中将IMLib的消息重新编辑方法封装到消息管理类中MessageManager发送消息可供参考，如果需要自己实现，可以直接参考IMLib的超级群文档

- MessageManager 中有一个重要的类 `WrapperMessage`,这个只是IMLib中`Message`的包装类，在SDK中是不存在的，是DEMO中自定义的

删除消息

接口说明

调用该方法编辑当前用户发送的消息。编辑成功以后会刷新会话列表。

所属类 :`MessageManager`

方法: `editMessage()`

调用示例:

```
MessageManager.get().editMessage(message，callBack);
```

参数说明:

参数	类型	说明
message	WrapperMessage	消息对象Message的包装类
callback	IResultBack<Boolean>	编辑消息的回调

代码示例：

```
///iMessage 是从适配器中拿到的item项
MessageManager.get().editMessage(message, new IResultBack<Boolean>() {
@Override
public void onResult(Boolean aBoolean) {
//更新适配器

}
});
```

标记消息

更新时间:2024-08-30

本文主要介绍demo中的标注消息

标注消息

- 标注消息并不是一种消息类型，而是Demo中的一种功能逻辑，类似于有些APP中的置顶消息或收藏消息，方便快速查看和定位重要消息,该消息逻辑主要依赖于业务服务器来完成的
- 标注消息的添加和移除权限，在Demo中是限定为创建者才有的权限，具体的逻辑可以根据自己的产品逻辑

获取标注消息列表

获取标注消息列表。目前该接口可返回消息 ID 集合。建议再通过 IMLib SDK 提供的 API 接口去获取对应的消息。

```
/// channelUid 当前频道的ID
//pageNum 页码
//pageSize 每一页的数量
//CommunityAPI.CHANNEL_MARK_MSG demo中的标注消息列表的API接口
Map<String, Object> params = new HashMap<>(4);
params.put("channelUid", getChannelUid());
params.put("pageNum", 0);
params.put("pageSize", 100);
OkApi.post(CommunityAPI.CHANNEL_MARK_MSG, params, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
//TODO Code
}
});
```

获取标注消息详情

在Demo中，该接口主要用来判断该消息的标注情况，是否已经标注了

```
//messageUid 消息Uid
//CommunityAPI.MARK_MSG_DETAILS 标注消息详情接口
OkApi.post(CommunityAPI.MARK_MSG_DETAILS + messageUid, null, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
if (result.ok()) {
//该消息已经被标注了
} else {
//该消息未被标注
}
}
});
```

添加标注消息

在Demo中调用标注消息接口，标注成功以后，会触发 IUltraGroupChannelListener.onAddMarkMessage

```
//添加频道标记信息
//channelUid 消息所在的频道ID
//messageUid 消息Uid
HashMap map = new HashMap();
map.put("channelUid", channelUid);
map.put("messageUid", messageUid);
OkApi.post(CommunityAPI.MARK_MSG, map, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
if (result.ok()) {
//TODO Code
} else {
//TODO Code
}
}
});
```

取消标注消息

在Demo中调用标注消息接口，移除标注消息成功以后，会触发
IUltraGroupChannelListener.onRemoveMarkMessage

```
//添加频道标记信息
//messageUid 消息Uid
OkApi.post(CommunityAPI.REMOVE_MARK_MSG + messageUid, null, new WrapperCallBack() {
@Override
public void onResult(Wrapper result) {
if (result.ok()) {
//TODO Code
} else {
//TODO Code
}
}
});
```

未读消息

更新时间:2024-08-30

本文介绍Demo是如何获取未读消息的。Demo中将IMLib的未读消息相关API封装到消息管理类中MessageManager可供参考，也可以直接参考IMLib的超级群文档

- 如果使用MessageManager中的方法，那么前置条件必须调用了

```
//targetId 社区ID
//channelId 频道ID
MessageManager.get().attachChannel(targetId, channelId);
```

获取社区的未读消息数量

接口说明

调用该方法可直接获得绑定好的社区(targetId)的未读数

所属类 : MessageManager

方法: getUltraGroupUnreadCount()

调用示例:

```
MessageManager.get().getUltraGroupUnreadCount(callback);
```

参数说明:

参数	类型	说明
callback	IResultBack<Integer>	获取未读数的回调

代码示例：

```
MessageManager.get().getUltraGroupUnreadCount(new IResultBack<Integer>() {
@Override
public void onResult(Integer integer) {
//TODO Code
}
});
```

获取社区某个频道的未读消息数量

接口说明

调用该方法可直接获得绑定好的频道(channelId)的未读数

所属类 : MessageManager

方法: getChannelUnreadCount()

调用示例:

```
MessageManager.get().getChannelUnreadCount(callback);
```

参数说明:

参数	类型	说明
callback	IResultBack<Integer>	获取未读数的回调

代码示例：

```
MessageManager.get().getChannelUnreadCount(new IResultBack<Integer>() {
@Override
public void onResult(Integer integer) {
//TODO Code
}
});
```

获取第一条未读消息

接口说明

调用该方法可直接获得绑定好的社区的频道(channelId)的第一条未读消息

所属类 : MessageManager

方法: getFirstUnReadMessage()

调用示例:

```
MessageManager.get().getFirstUnReadMessage(callback);
```

参数说明:

参数	类型	说明
callback	IResultBack<Message>	获取第一条未读消息的回调

代码示例：

```
MessageManager.get().getTheFirstUnreadMessage(new IResultBack<Message>() {
@Override
public void onResult(Message message) {
//TODO Code
}
});
```

监听未读消息

Demo 内部封装了工具类 UltraUnReadMessageManager，用于监听社区和频道的未读消息数量。

系统消息

更新时间:2024-08-30

本文介绍 Demo 是如何获取系统消息的。Demo中将 IMLib的系统消息相关 API 封装到消息管理类 MessageManager 中。如需自行实现，您也可以直接参考 IMLib 的超级群文档。

- 如果使用MessageManager中的方法，那么前置条件必须调用了

```
//targetId 社区ID
//channelId 频道ID
MessageManager.get().attachChannel(targetId, channelId);
```

获取社区的系统消息

接口说明

调用该方法可直接获得绑定好的社区(targetId)的系统消息

所属类 : MessageManager

方法: getSystemMessage(time,objectNames,callback)

调用示例:

```
MessageManager.get().getSystemMessage(lastMessageTime, OBJ_TAGS, callback);
```

参数说明:

参数	类型	说明
lastMessageTime	long	以这个时间为节点
OBJ_TAGS	String[]	获取Message对象的ObjectName字段在这个数组范围内的
callback	IResultBack<List<Message>>	获取的系统消息的回调

代码示例：

```
MessageManager.get().getSystemMessage(lastMessageTime, OBJ_TAGS, new IResultBack<List<Message>>() {
@Override
public void onResult(List<Message> messages) {
//TODO Code
});
```

社区事件监听管理类

更新时间:2024-08-30

UltraGroupCenter 超级群消息监听类，用于做消息监听的扩展，不处理具体逻辑

超级群消息改变监听

设置超级群消息改变监听的方法为 addUltraGroupMessageChangeListener，所属类为 UltraGroupCenter，您也可通过 ChannelClient 调用。

```
UltraGroupCenter.getInstance().addUltraGroupMessageChangeListener(listener);
```

参数	类型	说明
listener	IRongCoreListener.UltraGroupMessageChangeListener	超级群消息改变监听

返回值	方法	触发时机
void	onUltraGroupMessageExpansionUpdated	超级群消息扩展字段被更新
void	onUltraGroupMessageModified	超级群消息被编辑
void	onUltraGroupMessageRecalled	被撤销的消息集合

onUltraGroupMessageExpansionUpdated

触发时机：超级群消息扩展字段被更新。

```
void onUltraGroupMessageExpansionUpdated(List<Message> messages)
```

参数	类型	说明
messages	List<Message>	扩展字段被更新的消息集合

onUltraGroupMessageModified

触发时机：超级群消息被编辑。

```
void onUltraGroupMessageModified(List<Message> messages)
```

参数	类型	说明
messages	List<Message>	被编辑的消息集合

onUltraGroupMessageRecalled

触发时机：超级群消息被撤回。

```
void onUltraGroupMessageRecalled(List<Message> messages)
```

参数	类型	说明
messages	List<Message>	被撤销的消息集合

超级群消息已读时间监听

设置超级群消息已读时间监听的方法为 addUltraGroupMessageChangeListener，所属类为 UltraGroupCenter，您也可通过 ChannelClient 调用。

```
UltraGroupCenter.getInstance().addUltraGroupReadTimeListener(listener);
```

参数	类型	说明
listener	IRongCoreListener.UltraGroupReadTimeListener	超级群被阅读时间监听

返回值	方法	触发时机
void	onUltraGroupReadTimeReceived	超级群已读时间同步

onUltraGroupReadTimeReceived

触发时机：超级群已读时间同步。

```
void onUltraGroupReadTimeReceived(String targetId, String channelId, long time)
```

参数	类型	说明
targetId	String	超级群ID
channelId	String	频道ID
time	long	已读时间（服务端将未读数清零的时间）

超级群输入状态监听

设置超级群输入状态监听的方法为 addUltraGroupTypingStatusListener，所属类为 UltraGroupCenter，您也可通过 ChannelClient 调用。

```
UltraGroupCenter.getInstance().addUltraGroupTypingStatusListener( listener);
```

参数	类型	说明
listener	IRongCoreListener.UltraGroupTypingStatusListener	超级群输入状态监听

返回值	方法	触发时机
void	onUltraGroupTypingStatusChanged	超级群输入状态发生改变

onUltraGroupTypingStatusChanged

超级群输入状态发生改变

```
void onUltraGroupTypingStatusChanged(List<UltraGroupTypingStatusInfo> infoList)
```

参数	类型	说明
infoList	List<UltraGroupTypingStatusInfo>	超级群输入状态信息集合

超级群消息接收监听

设置超级群消息接收监听的方法为 addReceiveMessageWrapperListener，所属类为 UltraGroupCenter，您也可通过 RongCoreClient 调用。

```
UltraGroupCenter.getInstance().addReceiveMessageWrapperListener(listener);
```

参数	类型	说明
listener	OnReceiveMessageWrapperListener	超级群消息接收监听

返回值	方法	触发时机
void	onReceivedMessage	接收到超级群消息

onReceivedMessage

接收到超级群消息

```
void onReceivedMessage(Message message, ReceivedProfile profile)
```

参数	类型	说明
message	Message	接收到的消息对象
profile	ReceivedProfile	配置信息

监听用户在超级群中的动作

监听用户在超级群中的动作为 addIUltraGroupUserEventListener，所属类为 UltraGroupCenter。

```
UltraGroupCenter.getInstance().addIUltraGroupUserEventListener(listener);
```

参数	类型	说明
listener	IUltraGroupUserEventListener	监听用户在超级群中的动作

返回值	方法	触发时机
void	onKickOut	被踢出社区

返回值	方法	触发时机
void	onJoined	加入了某个社区
void	onRejected	被拒绝加入了某个社区
void	onLeft	离开了某个社区
void	onRequestJoin	申请加入某个社区
void	onBeForbidden	被某个社区禁言
void	onCancelForbidden	被某个社区取消禁言

onKickOut

被踢出社区

```
void onKickOut(String targetId, String fromUserId, String hint)
```

参数	类型	说明
targetId	String	社区 ID
fromUserId	String	操作人的ID
hint	String	提示

onJoined

加入了某个社区

```
void onJoined(String targetId, String fromUserId, String hint)
```

参数	类型	说明
targetId	String	社区 ID
fromUserId	String	操作人的ID
hint	String	提示

onRejected

被拒绝加入了某个社区

```
void onRejected(String targetId, String fromUserId, String hint)
```

参数	类型	说明
targetId	String	社区 ID
fromUserId	String	操作人的ID
hint	String	提示

onLeft

离开了某个社区

```
void onLeft(String targetId, String fromUserId, String hint)
```

参数	类型	说明
targetId	String	社区 ID
fromUserId	String	操作人的ID
hint	String	提示

onRequestJoin

申请加入某个社区

```
void onRequestJoin(String targetId, String fromUserId, String hint)
```

参数	类型	说明
targetId	String	社区 ID
fromUserId	String	操作人的ID
hint	String	提示

onBeForbidden

被某个社区禁言

```
void onBeForbidden(String targetId, String fromUserId, String toUserId, String hint)
```

参数	类型	说明
targetId	String	社区 ID
fromUserId	String	操作人的ID
toUserId	String	被操作人的ID
hint	String	提示

onCancelForbidden

被某个社区取消禁言

```
void onCancelForbidden(String targetId, String fromUserId, String toUserId, String hint)
```

参数	类型	说明
targetId	String	社区 ID
fromUserId	String	操作人的ID
toUserId	String	被操作人的ID
hint	String	提示

超级群频道系统消息监听

设置超级群频道系统消息监听的方法为 `addIUltraGroupChannelListener`，所属类为 `UltraGroupCenter`。

```
UltraGroupCenter.getInstance().addIUltraGroupChannelListener(listener);
```

参数	类型	说明
listener	IUltraGroupChannelListener	超级群频道系统消息监听

返回值	方法	触发时机
void	onAddMarkMessage	新增了标注消息
void	onRemoveMarkMessage	标注消息被删除
void	onUserJoined	当有用户加入了社区

onAddMarkMessage

新增了标注消息

```
void onAddMarkMessage(String targetId, String channelId)
```

参数	类型	说明
targetId	String	超级群ID
channelId	String	频道ID

onRemoveMarkMessage

标注消息被删除

```
void onRemoveMarkMessage(String targetId, String channelId)
```

参数	类型	说明
targetId	String	超级群ID
channelId	String	频道ID

onUserJoined

当有用户加入了社区

```
void onUserJoined(String targetId, String fromUserId, String toUserId, String hint)
```

参数	类型	说明
targetId	String	超级群ID
fromUserId	String	操作人ID
toUserId	String	被操作人ID
hint	String	提示

超级群改变监听

设置超级群改变监听的方法为 `addIUltraGroupChangeListener`，所属类为 `UltraGroupCenter`。

```
UltraGroupCenter.getInstance().addIUltraGroupChangeListener(listener);
```

参数	类型	说明
listener	IUltraGroupChangeListener	超级群改变监听

返回值	方法	触发时机
void	onUltraGroupChanged	社区发生了改变
void	onChannelDeleted	频道被删除
void	onUltraGroupDelete	社区被解散了

onUltraGroupChanged

社区发生了改变

```
void onUltraGroupChanged(String targetId)
```

参数	类型	说明
targetId	String	超级群ID

onChannelDeleted

频道被删除

```
void onChannelDeleted(String[] channelIds)
```

参数	类型	说明
channelIds	String[]	被删除的频道ID集合

onUltraGroupDelete

当有用户加入了社区

```
void onUltraGroupDelete(String targetId)
```

参数	类型	说明
targetId	String	超级群ID

超级群用户信息变更监听

设置超级群用户信息变更监听的方法为 addIUltraGroupUserInfoUpdateListener，所属类为 UltraGroupCenter。

```
UltraGroupCenter.getInstance().addIUltraGroupUserInfoUpdateListener(listener);
```

参数	类型	说明
listener	`IUltraGroupUserInfoUpdateListener`	超级群用户信息变更监听

返回值	方法	触发时机
void	onUpdateUserInfo	社区的某个人的个人信息发生了变化

onUpdateUserInfo

社区的某个人的个人信息发生了变化

```
void onUpdateUserInfo(String userId, String userName, String portrait)
```

参数	类型	说明
userId	String	用户ID
userName	String	用户名称
portrait	String	用户头像

未读消息管理类

更新时间:2024-08-30

UltraUnReadMessageManager 为超级群社区消息未读数量监听工具类

设置超级群消息未读数量监听器

设置超级群消息未读数量监听器的方法为 addObserver，所属类为 UltraUnReadMessageManager。

```
UltraUnReadMessageManager.getInstance().addObserver(observer);
```

参数	类型	说明
observer	UltraUnReadMessageManager.IUnReadMessageObserver	超级群消息未读数量监听器

监听器方法说明

监听器名称：UltraUnReadMessageManager.IUnReadMessageObserver

返回值	方法	触发时机
void	onChannelUnReadChanged	社区的某个频道的消息未读数量变化
void	onUltraGroupUnReadChanged	社区的消息未读数发生变化

onChannelUnReadChanged

触发时机：社区的某个频道的消息未读数量变化。

```
void onChannelUnReadChanged(String targetId, String channelId, int count)
```

参数	类型	说明
targetId	String	社区ID
channelId	String	频道ID
count	int	未读数量

onUltraGroupUnReadChanged

触发时机：社区的消息未读数发生变化。

```
void onUltraGroupUnReadChanged(String targetId, int count)
```

参数	类型	说明
targetId	String	社区ID
count	int	未读数量

社区用户信息管理类

更新时间:2024-08-30

UltraGroupUserManager 为社区用户信息缓存管理类 ，key为用户ID+社区ID，保持每个社区的唯一性

异步获取用户社区信息

异步获取用户社区信息的方法为 getAsyn，所属类为 UltraGroupUserManager。

```
UltraGroupUserManager.getInstance().getAsyn(userId,back);
```

参数	类型	说明
userId	String	用户ID
back	IResultBack<UltraGroupUserBean>	用户信息回调接口

用户信息回调接口方法说明

监听器名称：IResultBack<UltraGroupUserBean>

返回值	方法	触发时机
void	onResult	获取到用户信息

onResult

触发时机：获取到用户信息。

```
void onResult(UltraGroupUserBean ultraGroupUserBean)
```

参数	类型	说明
ultraGroupUserBean	UltraGroupUserBean	用户社区信息

同步获取用户社区信息

同步获取用户社区信息的方法为 getSync，所属类为 UltraGroupUserManager。

```
UltraGroupUserManager.getInstance().getSync(userId);
```

返回值	参数	类型	说明

返回值	参数	类型	说明
UltraGroupUserBean	userId	String	用户ID

更新用户社区信息

更新用户社区信息方法为 update，所属类为 UltraGroupUserManager。

```
UltraGroupUserManager.getInstance().update(userBean)
```

参数	类型	说明
userBean	UltraGroupUserBean	用户信息

监听单个用户社区信息

监听单个用户信息方法为observeSingle，所属类为 UltraGroupUserManager。

```
UltraGroupUserManager.getInstance().observeSingle(userId,resultBack);
```

参数	类型	说明
userId	String	用户ID
resultBack	IResultBack<UltraGroupUserBean>	用户信息回调接口

用户信息回调接口方法说明

监听器名称：IResultBack<UltraGroupUserBean>

返回值	方法	触发时机
void	onResult	用户信息发生变化的时候

强制从网络获取用户社区信息

强制从网络获取用户社区信息的方法为 batchGetAsyn，所属类为 UltraGroupUserManager。

```
UltraGroupUserManager.getInstance().batchGetAsyn(userIds,back);
```

参数	类型	说明
userIds	List<String>	用户ID集合
back	IResultBack<List<UltraGroupUserBean>>	用户信息回调接口

用户信息回调接口方法说明

监听器名称：IResultBack<List<UltraGroupUserBean>>

返回值	方法	触发时机
void	onResult	获取到用户信息集合

视频图片播放类

更新时间:2024-08-30

MediaPlayUtils 显示大图预览或者小视频

播放小视频

播放小视频的方法为 playSightMessage，所属类为 MediaPlayUtils 。

```
MediaPlayUtils.playSightMessage(message);
```

参数	类型	说明
message	Message	消息对象(支持SightMessage或ReferenceMessage)

显示大图预览

显示大图预览的方法为 showImage，所属类为 MediaPlayUtils 。

```
MediaPlayUtils.showImage(message);
```

参数	类型	说明
message	Message	消息对象(支持ImageMessage或ReferenceMessage)

社区通知管理类

更新时间:2024-08-30

UltraGroupNotificationLeaveManager 是实时社区Demo用来做社区通知的管理类

设置社区默认通知级别

设置指定的超级群的默认通知级别的方法为setUltraGroupConversationDefaultNotificationLevel,所属类为UltraGroupNotificationLeaveManager

```
void setUltraGroupConversationDefaultNotificationLevel(targetId, pushNotificationLevel, resultBack);
```

参数	类型	说明
targetId	String	指定的社区ID
pushNotificationLevel	IRongCoreEnum.PushNotificationLevel	社区的默认通知级别
resultBack	IResultBack<Boolean>	设置默认通知的回调

代码示例

```
/**
* 设置指定的超级群的默认通知级别
* 默认免打扰逻辑对所有群成员生效，由超级群的管理员进行设置
*
* @param targetId
* @param pushNotificationLevel
*/
public void setUltraGroupConversationDefaultNotificationLevel(String targetId,
IRongCoreEnum.PushNotificationLevel pushNotificationLevel, IResultBack<Boolean> resultBack) {
ChannelClient.getInstance().setUltraGroupConversationDefaultNotificationLevel(targetId,
pushNotificationLevel, new IRongCoreCallback.OperationCallback() {
@Override
public void onSuccess() {
Logger.d("setConversationNotificationLevel：targetId:" + "targetId:" + pushNotificationLevel);
//TODO Code
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {
//TODO Code
}
});
}
```

获取社区默认通知级别

获取指定的超级群的默认通知级别的方法为getUltraGroupConversationDefaultNotificationLevel,所属类为UltraGroupNotificationLeaveManager

```
void getUltraGroupConversationDefaultNotificationLevel(targetId, resultBack);
```

参数	类型	说明
targetId	String	指定的社区ID
resultBack	IResultBack<IRongCoreEnum.PushNotificationLevel>	获取默认通知的回调

代码示例

```
/**
* 获取超级群的默认免打扰级别
* 如果没有设置默认的，那么就是全部消息都接受
*
* @param targetId
*/
public void getUltraGroupConversationDefaultNotificationLevel(String targetId,
IResultBack<IRongCoreEnum.PushNotificationLevel> resultBack) {
ChannelClient.getInstance().getUltraGroupConversationDefaultNotificationLevel(targetId, new
IRongCoreCallback.ResultCallback<IRongCoreEnum.PushNotificationLevel>() {
@Override
public void onSuccess(IRongCoreEnum.PushNotificationLevel pushNotificationLevel) {
//TODO Code
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {
//TODO Code
}
});
}
```

设置指定的超级群的通知级别

设置指定的超级群的通知级别的方法为setUltraGroupNotificationLevel,所属类为UltraGroupNotificationLeaveManager

```
void setUltraGroupNotificationLevel(targetId, pushNotificationLevel);
```

参数	类型	说明
targetId	String	指定的社区ID
pushNotificationLevel	IRongCoreEnum.PushNotificationLevel	超级群的通知级别

代码示例

```
/**
* 设置指定的超级群的通知级别
*
* @param targetId
*/
public void setUltraGroupNotificationLevel(String targetId, IRongCoreEnum.PushNotificationLevel
pushNotificationLevel) {
ChannelClient.getInstance().setConversationNotificationLevel(Conversation.ConversationType.ULTRA_GROUP,
targetId, pushNotificationLevel, new IRongCoreCallback.OperationCallbac(){
@Override
public void onSuccess() {
//TODO Code
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {
//TODO Code
}
});
}
```

获取指定的超级群的通知级别

获取指定的超级群的通知级别的方法为getUltraGroupNotificationLevel,所属类为UltraGroupNotificationLeaveManager

```
void getUltraGroupNotificationLevel(targetId, resultBack);
```

参数	类型	说明
targetId	String	指定的社区ID
resultBack	IResultBack<IRongCoreEnum.PushNotificationLevel>	获取超级群通知级别的回调

代码示例

```
/**
* 获取指定的超级群的通知级别，如果没有设置过，那么以超级群的默认通知级别为准
* 如果自己没有设置过，此时会接受全部消息，那么将默认的通知级别设置给自己，那么以后自己就不再受默认配置的影响了
*
* @param targetId
*/
public void getUltraGroupNotificationLevel(String targetId,
IResultBack<IRongCoreEnum.PushNotificationLevel> resultBack) {
ChannelClient.getInstance().getConversationNotificationLevel(Conversation.ConversationType.ULTRA_GROUP,
targetId, new IRongCoreCallback.ResultCallback<IRongCoreEnum.PushNotificationLevel>() {
@Override
public void onSuccess(IRongCoreEnum.PushNotificationLevel pushNotificationLevel) {
//TODO Code
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {
//TODO Code
}
});
}
```

设置超级群指定频道的推送级别

设置超级群指定频道的推送级别的方法为`setChannelNotificationLevel`,所属类为`UltraGroupNotificationLeaveManager`

```
void setChannelNotificationLevel(targetId, channelId, pushNotificationLevel, resultBack);
```

参数	类型	说明
targetId	`String`	指定的社区ID
channelId	`String`	指定的频道ID
pushNotificationLevel	`IRongCoreEnum.PushNotificationLevel`	指定的超级群指定频道的通知级别
resultBack	`IResultBack<Boolean>`	设置超级群指定频道的通知级别回调

代码示例

```
/**
* 设置超级群指定频道的推送级别
*
* @param targetId
* @param channelId
* @param pushNotificationLevel
*/
public void setChannelNotificationLevel(String targetId, String channelId,
IRongCoreEnum.PushNotificationLevel pushNotificationLevel, IResultBack<Boolean> resultBack) {
ChannelClient.getInstance().setConversationChannelNotificationLevel(Conversation.ConversationType.ULTRA_
GROUP, targetId, channelId, pushNotificationLevel, new IRongCoreCallback.OperationCallback() {
@Override
public void onSuccess() {
//TODO Code
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode coreErrorCode) {
//TODO Code
}
});
}
```

获取超级群指定频道的推送级别

获取超级群指定频道的推送级别的方法为getChannelNotificationLevel,所属类为UltraGroupNotificationLeaveManager

```
void getChannelNotificationLevel(targetId, channelId, resultBack);
```

参数	类型	说明
targetId	String	指定的社区ID
channelId	String	指定的频道ID
resultBack	IResultBack<IRongCoreEnum.PushNotificationLevel>	获取超级群指定频道的通知级别回调

代码示例

```
/**
* 获取超级群指定频道的推送级别
* 如果没有特定的设置过，那么默认是跟随社区的设置
*
* @param targetId
*/
public void getChannelNotificationLevel(String targetId, String channelId,
IResultBack<IRongCoreEnum.PushNotificationLevel> resultBack) {
ChannelClient.getInstance().getConversationChannelNotificationLevel(Conversation.ConversationType.ULTRA_
GROUP, targetId, channelId, new IRongCoreCallback.ResultCallback<IRongCoreEnum.PushNotificationLevel>()
{
@Override
public void onSuccess(IRongCoreEnum.PushNotificationLevel pushNotificationLevel) {
//TODO Code
}

@Override
public void onError(IRongCoreEnum.CoreErrorCode e) {
//TODO Code
}
});
}
```

更新日志 1.0.0

更新时间:2024-08-30

发布日期：2022/07/12

1. 支持 创建、解散、加入、退出社区
2. 支持社区的通知管理
3. 支持社区的成员管理
4. 支持发送图片，视频，文字等消息
5. 支持社区的未读消息管理

常见问题 为什么获取第一条未读消息有时候拿不到？

更新时间:2024-08-30

因为目前超级群API只支持从本地数据库获取未读消息，当程序被杀死的情况下，群内的大量未读消息是没有插入到本地数据库的，这个时候去通过超级群的获取第一条未读消息API去获取消息是拿不到的，这个需要等待后面支持。

谁有权限去管理社区，我看社区中只有创建者和用户，是否可以设置管理员？

在实时社区中其实是没有权限的概念的，当然可以设置管理员，只需要和您的业务服务端的开发商量好管理员字段，就可以按照自己的需求去设定管理员，并且设置管理员权限。

我看说的是社区是依赖的IMLib,但是为什么我只依赖IMLib会报错？

因为社区Demo在开发过程中用到了IMKit中的相关工具类方法，您如果觉得界面还是想自己实现，也可以不用管这个，而是自己来绘制界面。