场景融合

实时社区

iOS 1.X

2024-05-17

实时社区概述

更新时间:2024-05-17

实时社区 RCSceneCommunity 是基于融云 IM 超级群能力实现的开源组件，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

RCSceneCommunity 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 iOS 源码 (Github)
  - RC RTC iOS 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)

- **RCSceneCommunity 模块**

  - RCSceneCommunity (GitHub)
  - RCSceneCommunity (Gitee)

超级群服务配置

更新时间:2024-05-17

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:2024-05-17

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/

加载配置

```
/// 实时社区初始化需要注入的配置
/// - Parameters:
/// - serviceHost: HTTP 数据接口需要的 URL Host
/// - businessToken: 为了快速运行您的 Demo， 融云提供相应的测试服务，
/// 如果使用测试服务需要申请对应的 Business Token
/// 申请地址：https://rcrtc-api.rongcloud.net/code/
public static func loadConfig(serviceHost: String, businessToken: String)

//初始化代码示例
RCSCConfig.loadConfig(serviceHost: host, businessToken: businessToken)
```

快速上手

更新时间:2024-05-17

> 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

- **Xcode**：确保与苹果官方同步更新
- **CocoaPods**：1.10.0 及以上
- **iOS**：11.0 及以上
- **Swift**：5.0 及以上

开通服务

超级群功能需要在控制台超级群服务 页面开通。配置说明详见超级群服务配置。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。

> 服务开通、关闭等设置完成 30 分钟后生效。

集成 RCSceneCommunity

实时社区支持使用 CocoaPods 导入并管理 RCSceneCommunity

> pod 版本必须为 1.10.0 或更新版本。具体请查看知识库文档：
> https://help.rongcloud.cn/t/topic/747

1. 在 podfile 中添加如下内容：

```
pod 'RCSceneCommunity', '~> 0.1.2'
```

2. 请在终端中运行以下命令：

```
pod install
```

> 如果出现找不到相关版本的问题，可先执行 `pod repo update`，再执行 `pod install`。

3. 上一步完成后，CocoaPods 会在您的工程根目录下生成一个 `xcworkspace` 文件，只需通过 XCode 打开该文件即可加载工程。

集成后的包大小

集成 `RCSceneCommunity` 后，增量大约 6 MB（包含 IMLib 依赖库）

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/IOS/5.X/noui/ultragroup/intro

初始化 融云 IM 初始化

更新时间:2024-05-17

RCSceneCommunity 的初始化依赖于融云 IMLib SDK 的初始化，具体可以根据当前项目的实际依赖选择使用。

> 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 IM 的初始化，您可以使用 IMLib 或 IMKit 初始化，具体可以根据当前项目的实际依赖选择使用。

- 方法（Objective-C）

```
/*!
初始化融云 SDK

@param appKey 从融云开发者平台创建应用后获取到的 App Key
@discussion 初始化后，SDK 会监听 app 生命周期， 用于判断应用处于前台、后台，根据前后台状态调整链接心跳
@discussion
您在使用融云 SDK 所有功能 ( 包括显示 SDK 中或者继承于 SDK 的 View ) 之前，您必须先调用此方法初始化 SDK。
在 App 整个生命周期中，您只需要执行一次初始化。

**升级说明:**
**从2.4.1版本开始，为了兼容 Swift 的风格与便于使用，将原有的 init: 方法升级为此方法，方法的功能和使用均不变。**

@warning 如果您使用 IMLibCore，请使用此方法初始化 SDK；
如果您使用 IMKit，请使用 RCIM 中的同名方法初始化，而不要使用此方法。

@remarks 连接
*/
- (void)initWithAppKey:(NSString *)appKey;
```

- 调用示例（Swift）

```
//初始化代码示例
RCCoreClient.shared().initWithAppKey(appKey)
```

登录

获取当前用户信息，注入当前用户信息到 RCSceneCommunity

```
/// 注入当前登录的用户信息
/// - Parameter user: 用户信息
public static func loadUser(user: RCSCUser?)

//初始化代码示例
RCSCUser.loadUser(user: user)
```

连接融云服务

- 方法（Objective-C）

```
// IMLib 中 RCCoreClient 的连接接口说明
/*!
与融云服务器建立连接

@param token 从您服务器端获取的 token（用户身份令牌）
@param dbOpenedBlock 本地消息数据库打开的回调
@param successBlock 连接建立成功的回调 [ userId: 当前连接成功所用的用户 ID]
@param errorBlock 连接建立失败的回调，触发该回调代表 SDK 无法继续重连 [errorCode: 连接失败的错误码]

@discussion 调用该接口，SDK 会在连接失败之后尝试重连，直到连接成功或者出现 SDK 无法处理的错误（如 token 非法）。
如果您不想一直进行重连，可以使用 connectWithToken:timeLimit:dbOpened:success:error: 接口并设置连接超时时间
timeLimit。

@discussion 连接成功后，SDK 将接管所有的重连处理。当因为网络原因断线的情况下，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。

对于 errorBlock 需要特定关心 tokenIncorrect 的情况：
一是 token 错误，请您检查客户端初始化使用的 AppKey 和您服务器获取 token 使用的 AppKey 是否一致；
二是 token 过期，是因为您在控制台设置了 token 过期时间，您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。
在此种情况下，您需要请求您的服务器重新获取 token 并建立连接，但是注意避免无限循环，以免影响 App 用户体验。

@warning 如果您使用 IMLibCore，请使用此方法建立与融云服务器的连接；
如果您使用 IMKit，请使用 RCIM 中的同名方法建立与融云服务器的连接，而不要使用此方法。

此方法的回调并非为原调用线程，您如果需要进行 UI 操作，请注意切换到主线程。
*/
- (void)connectWithToken:(NSString *)token
dbOpened:(void (^)(RCDBErrorCode code))dbOpenedBlock
success:(void (^)(NSString *userId))successBlock
error:(void (^)(RCConnectErrorCode errorCode))errorBlock;
```

- 调用示例（Swift）

```
//初始化代码示例
RCCoreClient.shared().connect(withToken: token) { dbErrorCode in
//数据库状态
} success: { userId in
//链接成功
} error: { errorCode in
//链接失败
}
```

社区的创建和解散

更新时间:2024-05-17

本章主要介绍社区的创建流程

社区功能依赖于IMLib的超级群业务，IMLib本身并不提供创建社区的API，创建社区依赖于服务端接口，所以需要和后台协商具体的参数，这里主要介绍RCSceneCommunity中如何调用接口创建社区

> **注意**
>
> - RCSceneCommunity中，服务端限制每个人最多只可创建三个社区
> - RCSceneCommunity中，服务端限制上传图片文件最大不可超过5M
> - RCSceneCommunity中，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

```
//上传图片到服务端
// data 图片数据
RCSCUploadApi().uploadImage(data: data).success { path in
//获取到图片上传的完成后的URI
}
```

创建接口

```
//创建社区
//name 社区名称
//portrait 社区头像
RCSCCommunityCreateApi(name: name, portrait: portrait).create().success { _ in
//创建成功
}.failed { error in
//创建失败
}
```

解散社区

解散社区接口，主要需要的参数为社区ID

```
//解散社区
//communityId 当前社区ID
RCSCCommunityDeleteApi(communityId: communityId).delete().success { _ in
//社区解散成功
}.failed { error in
//社区解散失败
}
```

分组频道管理

更新时间:2024-05-17

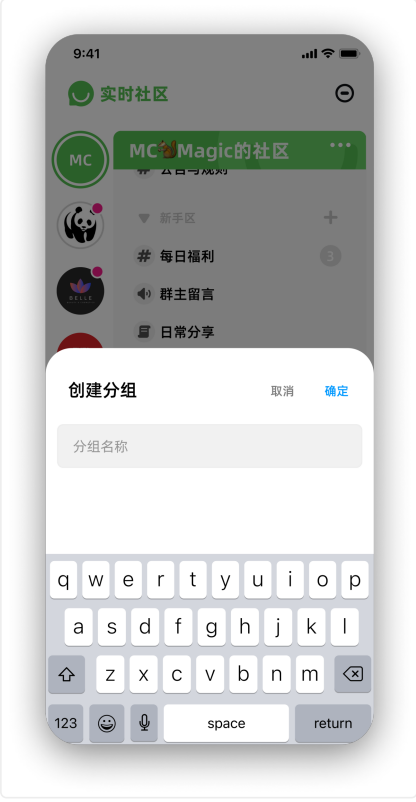
本章主要介绍社区的分组和频道的管理

- 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RCSCConversationMessageManager。

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



```
//创建分组
//communityId 社区ID
//name 分组名称
RCSCGroupCreateApi(communityId: communityId, name: name).create().success { object in
//创建分组成功
}.failed { error in
//创建分组失败
}
```

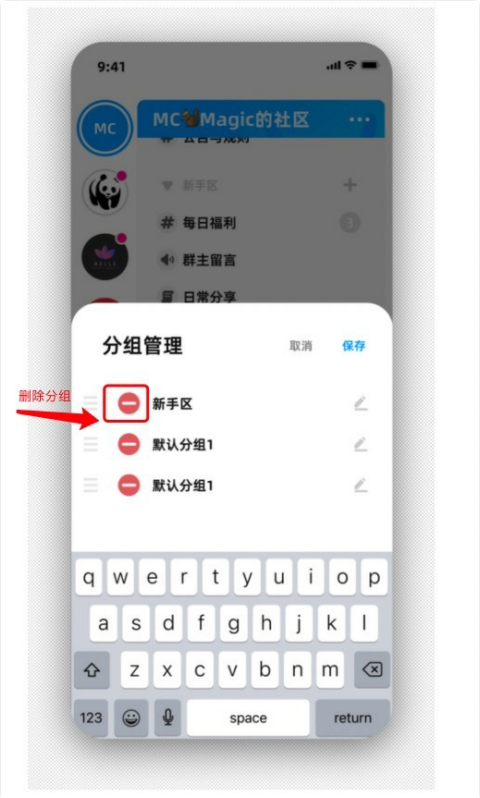
分组重命名和调整顺序

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


```
//保存社区详情信息
//communityDetailDictionary 当前操作的社区的详情
RCSCCommunityEditDetailApi(communityData: communityDetailDictionary, updateType:
updateType).fetch().success {object in
//保存社区详情信息成功
}.failed { error in
//保存社区详情信息失败
}
```

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




```
//删除分组
//groupId 被删除的分组ID
RCSCGroupDeleteApi(groupUid: groupId).fetch().success { msg in
//删除分组成功
}.failed { error in
//删除分组失败
}
```

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

```
//创建频道
//communityId 社区ID
//groupId 分组ID
RCSCChannelCreateApi(communityId: communityId, groupId: groupId, name: name).create().success { object
in
//创建频道成功
}.failed { error in
//创建频道失败
}
```

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

```
//保存社区详情信息
//communityDetailDictionary 当前操作的社区的详情
RCSCCommunityEditDetailApi(communityData: communityDetailDictionary, updateType:
updateType).fetch().success {object in
//保存社区详情信息成功
}.failed { error in
//保存社区详情信息失败
}
```

删除频道

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

```
//删除频道
//channelId 频道ID
RCSCChannelDeleteApi(channelUid: channelId).fetch().success { _ in
//删除频道成功
}.failed { error in
//删除频道失败
}
```

社区资料编辑

更新时间:2024-05-17

本章主要介绍社区的资料编辑

社区资料编辑主要分为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
- IMLib管理的社区默认免打扰模式

社区信息编辑

通过业务服务器接口，修改社区信息

```
//更新社区信息
//communityId 社区ID
//修改的社区信息内容
/* 社区各个信息对应的编辑Key
{
"coverUrl": "",//社区封面
"joinChannelUid": "",//默认进入的频道
"msgChannelUid": "",//默认通知频道
"name": "",//社区名称
"needAudit": 0,//是否需要审核
"noticeType": 0,//通知类型
"portrait": "",//社区头像
"remark": "",//社区简介
}
*/
RCSCCommunityUpdateInfoApi(communityId: communityId, param: param).fetch().success { _ in
//编辑成功
}.failed { error in
//编辑失败
}
```

默认免打扰模式设置

```
/// @class RCSCRemoteNotificationHandler
/// 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
/// - Parameters:
/// - communityId: 社区ID
/// - channelId: 频道ID
/// - level: 通知等级
/// - success: 成功回调
/// - error: 失败回调
func setChannelDefaultNotificationType(communityId: String, channelId: String, level:
RCPushNotificationLevel, success: @escaping (() -> Void), error: @escaping ((RCErrorCode) -> Void))
```

社区通知管理

更新时间:2024-05-17

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

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - level: 社区通知等级
/// - success: 成功回调
/// - error: 失败回调
func setCommunityDefaultNotificationType(communityId: String,
level: RCPushNotificationLevel,
success: @escaping (() -> Void),
error: @escaping ((RCErrorCode) -> Void))
```

频道通知

设置频道的默认通知方式


```
/// 设置社区指定频道的推送级别
/// - Parameters:
/// - communityId: 社区ID
/// - channelId: 频道ID
/// - success: 成功回调
/// - error: 失败回调
func getChannelDefaultNotificationType(communityId: String, channelId: String,
success: @escaping ((RCPushNotificationLevel) -> Void),
error: @escaping ((RCErrorCode) -> Void))
```

社区成员管理

更新时间:2024-05-17

本文主要介绍创建者如果对于成员进行管理

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

```
/*
{
"communityUid": "",//必传
"userUid": "",//必传
"freezeFlag": 0,
"nickName": "",
"noticeType": 0,
"shutUp": 0,
"status": 0
}
*/
//更新用户信息接口，相关参数如上，communityUid，userUid为必传
RCSCUpdateUserInfoApi.init(userData: param).fetch().success { [weak self] object in
//用户信息更新成功
}.failed { error in
//用户信息更新失败
}
```

社区审核模式

更新时间:2024-05-17

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

```
//修改社区信息
//communityId 社区ID
//param = ["needAudit":0]不需要审核
// ["needAudit":1]需要审核
RCSCCommunityUpdateInfoApi(communityId: communityId, param: param).fetch().success { _ in
//修改成功
}.failed { error in
//修改失败
}
```

加入社区

更新时间:2024-05-17

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RCSCConversationMessageManager。

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

```
//加入社区
//communityId 社区ID
RCSCCommunityJoinApi(communityId: communityId).fetch().success {[weak self] _ in
//加入社区成功
}.failed { error in
//加入社区失败
}
```

发送消息

更新时间:2024-05-17

本文介绍RCSceneCommunity是如何发送消息的。RCSceneCommunity中将IMLib的消息发送方法封装到RCSCSendMessageHandler中，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，

发送文本消息

接口说明

当调用该方法时，发送以后会自动插入本地数据源中并刷新会话列表。

```
/// 发送文本消息
/// - Parameters:
/// - text: 文本内容
/// - communityId: 社区ID
/// - channelId: 频道ID
/// - atUsers: 需要@的用户
/// - pushContent: 远程推送的相关信息

func sendTextMessage(text: String,
communityId: String,
channelId: String,
atUsers: Array<RCSCCommunityUser>?,
pushContent: RCSCPushContent)
```

发送图片消息

接口说明

当调用该方法时，发送以后会自动插入本地数据源中并刷新会话列表。

```
/// 发送图片消息
/// - Parameters:
/// - image: 需要发送的图片
/// - communityId: 社区ID
/// - channelId: 频道ID
/// - pushContent: 远程推送的相关信息

func sendImageMessage(image: UIImage,
communityId: String,
channelId: String,
pushContent: RCSCPushContent)
```

发送视频消息

接口说明

当调用该方法时，发送以后会自动插入本地数据源中并刷新会话列表。

```
/// 发送视频消息
/// - Parameters:
/// - videoPath: 视频本地路径
/// - thumbnail: 封面图片
/// - duration: 视频时长
/// - communityId: 社区ID
/// - channelId: 频道ID
/// - pushContent: 远程推送的相关信息

func sendVideoMessage(videoPath: String,
thumbnail: UIImage,
duration: UInt,
communityId: String,
channelId: String,
pushContent: RCSCPushContent)
```

删除消息

更新时间:2024-05-17

本文介绍RCSceneCommunity是如何删除消息的。RCSceneCommunity中将IMLib的消息删除方法封装到RCSCModifyMessageHandler中，删除消息可供参考，如果需要自己实现，可以直接参考IMLib的超级群文档

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。

```
/// 删除消息
/// - Parameter messageId: 当前消息的消息ID
/// - Returns: 是否删除成功
func deleteMessage(messageId: Int64) -> Bool
```

撤销消息

更新时间:2024-05-17

本文介绍RCSceneCommunity是如何撤销已发送的消息。RCSceneCommunity中将IMLib的消息撤回方法封装到RCSCModifyMessageHandler中，撤回消息可供参考，如果需要自己实现，可以直接参考IMLib的超级群文档

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。

```
/// 撤回消息
/// - Parameter message: IMLib 消息实例
func recallMessage(message: RCMessage)
```

编辑消息

更新时间:2024-05-17

本文介绍RCSceneCommunity是如何编辑已发送消息的。RCSceneCommunity中将IMLib的编辑已发送消息方法封装到RCSCModifyMessageHandler中，编辑已发送消息可供参考，如果需要自己实现，可以直接参考IMLib的超级群文档，

编辑消息

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。

```
/// 编辑已发送消息
/// - Parameters:
/// - messageUid: 消息的唯一ID
/// - text: 编辑过的文本
/// - atUsers: 需要@的用户
/// - completion: 操作完成的回调
func modifyMessage(messageUid: String,
text: String,
atUsers: Array<RCSCCommunityUser>?,
completion: @escaping RCSCSendMessageCompletion)
```

标记消息

更新时间:2024-05-17

本文主要介绍RCSceneCommunity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`RCSceneCommunity`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`RCSceneCommunity`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实施

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。

```
//获取标注消息列表
//channelUid 频道ID
//pageNum 分页的页码
//pageSize 分页的容量
RCSCCommunityMarkMessageListApi(channelUid: channelUid, pageNum: pageNum, pageSize:
pageSize).fetch().success {[weak self] object in
//获取标注消息成功
}.failed { error in
//获取标注消息失败
}
```

获取标注消息详情

在RCSceneCommunity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

```
//获取标注消息详情
//messageUid 消息唯一ID
RCSCMarkMessageDetailApi(messageUid: message.messageUId).fetch().success { object in
//获取标注详情成功
}.failed { error in
//获取标注详情失败
}
```

添加标注消息

在RCSceneCommunity中调用标注消息接口，标注成功以后，会刷新会话列表

```
//添加频道标记信息
//channelUid 频道ID
//messageUid 消息唯一ID
RCSCCommunityMarkApi(channelUid: channelId, messageUid: messageUid).mark().success { _ in
//标注消息成功
}.failed { error in
//标注消息失败
}
```

取消标注消息

在RCSceneCommunity中调用标注消息接口，移除标注消息成功以后，会刷新会话列表

```
//删除频道标记信息
//messageUid 消息唯一ID
RCSCCommunityDeleteMarkApi(messageUid: markMessage.messageUid).deleteMark().success { object in
//删除标记消息成功
}.failed { error in
//删除标记消息失败
}
```

未读消息

更新时间:2024-05-17

本文介绍RCSceneCommunity是如何查询未读消息的。RCSceneCommunity中将IMLib的消息未读数查询方法封装到RCSCUnreadCountHandler中，查询消息未读数可供参考，如果需要自己实现，可以直接参考IMLib的超级群文档，

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

```
/// 获取社区消息未读数
/// - Parameter communityId: 社区ID
/// - Returns: 消息未读数
func fetchCommunityUnreadCount(communityId: String) -> Int32
```

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

```
/// 获取频道消息未读数
/// - Parameters:
/// - communityId: 社区ID
/// - channelId: 频道ID
/// - Returns: 频道消息未读数
func fetchChannelUnreadCount(communityId: String,
channelId: String) -> Int32
```

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)的最早一条未读消息。

```
/// 获取频道内第一条未读消息
/// - Parameters:
/// - communityId: 社区ID
/// - channelId: 频道ID
/// - Returns: 第一条未读消息实例
func fetchFirstUnreadMessage(communityId: String,
channelId: String) -> RCMessage?
```

系统消息

更新时间:2024-05-17

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

> - 如果使用RCSCSystemMessageManager中的方法，那么前置条件必须遵守RCIMClientReceiveMessageDelegate

```
RCCoreClient.shared().add(self)
```

获取社区的系统消息

接口说明

调用该方法续跟后台绑定好的社区(targetId)的系统消息

> RCSceneCommunity 中系统消息默认targetId: "*SYSTEM*"

所属类 : RCSCSystemMessageManager

首次获取方法:

```
/// 获取初始化历史系统消息
/// - Parameter callBackMsgArr: 成功回调
func fetchInitializedHistoryMessage(_ callBackMsgArr:RCSCSysInitMsgClosure?)
```

调用示例:

```
RCSCSystemMessageManager().fetchInitializedHistoryMessage({ [weak self] msgArr in
guard let self = self else { return }
if let msgArr = msgArr , msgArr.count > 0 {
for msg in msgArr {
//社区系统消息的RCMic:CommunitySysNotice表示
if msg.objectName == "RCMic:CommunitySysNotice" {
self.dataSourceArray.append(msg)
}
}
//TODO: your code
}
})
```

获取更多方法:

```
/// 获取更多历史消息
/// - Parameters:
/// - messageId: 哨兵消息ID
/// - callBackMsgArr: 成功回调
func fetchSystemHistoryMessage(_ messageId:Int,
callBackMsgArr:RCSCSysInitMsgClosure?)
```

调用示例:

```
let msg = self.dataSourceArray.last {
let lastMsgId = msg.messageId
RCSCSystemMessageManager().fetchSystemHistoryMessage(lastMsgId){ [weak self] msgArr in
guard let self = self else { return }
if let msgArr = msgArr , msgArr.count > 0 {
for msg in msgArr {
//社区系统消息的RCMic:CommunitySysNotice表示
if msg.objectName == "RCMic:CommunitySysNotice" {
self.dataSourceArray.append(msg)
}
}
//TODO: your code
}
}
```

捕获新增的系统消息

> 实现RCIMClientReceiveMessageDelegate代理onReceived方法即可即可

示例代码

```
extension RCSCSystemMessageTableListView: RCIMClientReceiveMessageDelegate{
func onReceived(_ message: RCMessage!, left nLeft: Int32, object: Any!) {
if message.conversationType == .ConversationType_SYSTEM {
//TODO: 捕获新的系统消息
//insertMsg(message)
}
}

}
```

社区消息管理类

更新时间:2024-05-17

RCSCConversationMessageManager 社区消息管理类，用于做消息监听的扩展，消息相关功能的处理。

> RCSCConversationMessageManager 只对外提供统一的功能接口和回调，具体的实现由相应的类去完成
>
> - 消息的接收和处理由 RCSCMessageReceiveHandler 完成。
> - 消息的发送由 RCSCSendMessageHandler 完成。
> - 消息被编辑同步由 RCSCMessageChangeHandler 完成。
> - 消息编辑由 RCSCModifyMessageHandler 完成。
> - 历史消息获取由 RCSCHistoryMessageHandler 完成。
> - 已读状态同步由 RCSCReadTimeHandler 完成。
> - 未读数获取由 RCSCUnreadCountHandler 完成。
> - 输入状态同步由 RCSCTypingStatusHandler 完成。
> - 本地通知发送由 RCSCLocalNotificationCenter 完成。
> - 推送通知等级设置由 RCSCRemoteNotificationHandler 完成。

初始化

设置Handler代理

handler提供功能的具体实现，和 SDK 的回调监听，回调通过 RCSCConversationMessageManager 转发到相应的实例对象

```
private func setHandlerDelegate() {
readTimeHandler.delegate = self
typingStatusHandler.delegate = self
messageChangeHandler.delegate = self
messageReceiveHandler.delegate = self
sendMessageHandler.delegate = self
historyMessageHandler.delegate = self
modifyMessageHandler.delegate = self
}
```

注册自定义消息

IMLib除内置消息外，还支持自定义消息。RCSceneCommunity 中服务端信令基于自定义消息完成。

```
//系统通知消息
RCCoreClient.shared().registerMessageType(RCSCSystemMessage.self)
//用户信息更新消息
RCCoreClient.shared().registerMessageType(RCSCUserInfoUpdateMessage.self)
//社区信息更新消息
RCCoreClient.shared().registerMessageType(RCSCChangeMessage.self)
//社区系统消息
RCCoreClient.shared().registerMessageType(RCSCChannelNoticeMessage.self)
//图片消息
RCCoreClient.shared().registerMessageType(RCImageMessage.self)
//视频消息
RCCoreClient.shared().registerMessageType(RCSightMessage.self)
//引用消息
RCCoreClient.shared().registerMessageType(RCReferenceMessage.self)
```

注册已读状态回调代理

```
RCChannelClient.sharedChannelManager().setRCUltraGroupReadTimeDelegate(readTimeHandler)
```

注册输入状态代理回调

```
RCChannelClient.sharedChannelManager().setRCUltraGroupTypingStatusDelegate(typingStatusHandler)
```

注册消息被编辑状态同步代理

```
RCChannelClient.sharedChannelManager().setRCUltraGroupMessageChangeDelegate(messageChangeHandler)
```

注册 IM 消息接收代理

```
RCCoreClient.shared().add(messageReceiveHandler)
```

注册RCSCConversationMessageManagerDelegate代理

在需要监听代理方法的类里设置delegate

```
RCSCConversationMessageManager.setDelegate(delegate: self)
```

代理回调

同步超级群已读消息时间

```
//多端同步已读时间
//targetId 超级群ID
//channelId 频道ID
//readTime 已读时间
func onUltraGroupReadTimeReceived(_ targetId: String!,
channelId: String!,
readTime: Int64)
```

同步远端超级群输入状态

```
//用于显示 `XXX 正在输入`
//[RCUltraGroupTypingStatusInfo] 当前正在输入的用户信息
func onUltraGroupTypingStatusChanged(_ infoArr: [RCUltraGroupTypingStatusInfo]!)
```

同步扩展信息更新过的超级群消息

```
//当消息的扩展信息有更新时，该代理方法会被调用
//[RCMessage] 被更新消息的合集
func onUltraGroupMessageExpansionUpdated(_ messages: [RCMessage]!)
```

接收IM消息

```
//当收到IM消息时，该代理方法会被调用
//message 收到的IM消息
//left 剩余未接收的消息数量
//object 消息监听设置的key值
func onReceived(_ message: RCMessage!, left nLeft: Int32, object: Any!)
```

接收到系统消息

```
//当收到系统消息时，该代理方法会被回调，同时接收IM的消息也会被调用
//message 收到的系统消息
//left 剩余未接收的消息数量包含非系统消息
//object 消息监听设置的key值
func onReceivedSystem(_ message: RCMessage!, left nLeft: Int32, object: Any!)
```

同步已发送的消息

```
//当发送消息时，该方法会被调用，此时的消息状态为发送在途。
//message 被发送的消息实例
func onMessageSend(_ message: RCMessage?)
```

同步历史消息

```
//当调用获取历史消息方法后，如获取成功该方法会被调用，并返回历史消息集合
//Array<RCMessage> 历史消息集合
//RCSCFetchMessageStrategy 获取历史消息记录的策略，在某个时间点之后或者某个时间点之前
func onFetchHistoryMessagesSuccess(messages:Array<RCMessage>, strategy: RCSCFetchMessageStrategy)
```

同步重新编辑过的超级群消息

```
//当社区用户重新编辑了消息，该代理方法会被调用并同步相应的消息
//[RCMessage] 被编辑过的消息集合
func onMessageModified(_ messages: [RCMessage]!)
```

同步撤回的超级群消息

```
//当社区用户撤回了消息，该代理方法会被调用并同步相应的消息
//[RCMessage] 被撤回的消息集合
func onMessageRecall(_ messages: [RCMessage]!)
```

同步标记消息列表

```
//当调用获取标注消息列表成功后，会调用该代理方法
//[RCSCMarkMessage] 标记消息的集合
//loadMore 是否还有标记消息
func onFetchMarkMessages(_ messages: [RCSCMarkMessage]!, _ loadMore: Bool)
```

同步消息已读状态

```
//已读消息是否同步成功，当调用消息已读接方法后，会调用改代理方法
//success 是否同步成功
//errorCode 错误码
func syncCommunityReadStatus(_ success: Bool, _ errorCode: RCErrorCode?)
```

功能函数

同步频道已读的时间

```
/// - Parameters:
/// - communityId: 当前的社区ID
/// - channelId: 当前的频道ID
/// - time: 当前时间
static func syncCommunityReadStatus(communityId: String,
channelId: String,
time: Int64)
```

同步本端输入状态

```
/// - Parameters:
/// - communityId: 当前的社区ID
/// - channelId: 当前的频道ID
static func sendTypingMessage(communityId: String,
channelId: String)
```

发送消息

发送文本消息

```
/// - Parameters:
/// - text: 发送的文本内容
/// - communityId: 当前社区ID
/// - channelId: 当前频道ID
/// - atUsers: 被@的用户
/// - pushContent: 推送相关的信息
static func sendTextMessage(text: String,
communityId: String,
channelId: String,
atUsers: Array<RCSCCommunityUser>?,
pushContent: RCSCPushContent)
```

发送图片消息

```
/// - Parameters:
/// - image: 图片
/// - communityId: 当前的社区ID
/// - channelId: 当前的频道ID
/// - pushContent: 推送相关的信息
static func sendImageMessage(image: UIImage,
communityId: String,
channelId: String,
pushContent: RCSCPushContent)
```

发送视频消息

```
/// - Parameters:
/// - videoPath: 视频资源路径
/// - thumbnail: 封面图片
/// - duration: 视频时常
/// - communityId: 当前的社区ID
/// - channelId: 当前的频道ID
/// - pushContent: 推送相关的信息
static func sendVideoMessage(videoPath: String,
thumbnail: UIImage,
duration: UInt,
communityId: String,
channelId: String,
pushContent: RCSCPushContent)
```

消息重新发送

```
/// - Parameters:
/// - message: 重新发送的消息实例
/// - type: 被重新发送的消息类型（文本、图片、视频等等）
/// - pushContent: 推送相关的信息
static func resendMessage(message: RCMessage,
type: RCSCMessageType,
pushContent: RCSCPushContent)
```

发送引用消息

```
/// - Parameters:
/// - quoteMessage: 被引用的消息实例
/// - text: 引用相关的文本内容
/// - atUsers: 被@的用户
/// - pushContent: 推送相关的信息
static func sendQuoteMessage(quoteMessage: RCMessage,
text: String,
atUsers: Array<RCSCCommunityUser>?,
pushContent: RCSCPushContent)
```

获取消息

获取历史消息（频道第一次获取历史消息调用该方法）

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - channelId: 当前频道ID
/// - conversation: 当前频道会话对象
static func fetchConversationInitializedHistoryMessage(communityId: String,
channelId: String,
conversation: RCConversation?) conversation: RCConversation?)
```

分页加载历史消息

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - channelId: 当前频道ID
/// - sendTime: 最后一条消息的发送时间
/// - strategy: 返回的消息集合排序策略
/// - fix: 当前不使用该参数
static func fetchHistoryMessage(communityId: String,
channelId: String,
sendTime: Int64,
strategy: RCSCFetchMessageStrategy,
fix: Bool = true)
```

根据消息Uid获取消息

```
/// - Parameter messageUid: 消息唯一ID
/// - Returns: 消息实例
static func fetchMessageByMessageUid(messageUid: String) -> RCMessage?
```

获取标注消息列表

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - channelId: 当前频道ID
/// - pageNum: 分页页码
/// - pageSize: 分页容量
/// - loadMore: 是否还有数据
static func fetchConversationMarkHistoryMessage(communityId: String,
channelId: String,
pageNum: Int,
pageSize: Int,
loadMore: Bool = false)
```

获取频道会话对象

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - channelId: 当前频道ID
/// - Returns: 会话对象
static func getConversation(communityId: String,
channelId: String) -> RCConversation?
```

消息修改

删除消息

```
/// - Parameter messageId: 消息的数据库ID
/// - Returns: 是否删除成功
static func deleteMessage(messageId: Int64) -> Bool
```

修改消息

```
/// - Parameters:
/// - messageUid: 消息唯一ID
/// - text: 修改之后的文本
/// - atUsers: 被@的用户
/// - completion: 修改完成回调
static func modifyMessage(messageUid: String,
text: String,
atUsers: Array<RCSCCommunityUser>?,
completion: @escaping RCSCSendMessageCompletion)
```

撤回消息

```
/// - Parameter message: 消息实例
static func recallMessage(message: RCMessage)
```

未读管理

频道是否需要展示小红点

```
/// 社区是否需要展示小红点
/// - Parameter communityId: 单签社区ID
/// - Returns: 是否需要展示
static func isShowRedDot(communityId: String) -> Bool
```

清除未读数

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - channelId: 当前频道ID
/// - time: 当前时间
/// - Returns: 是否清除成功
static func clearUnreadCount(communityId: String,
channelId: String,
time: Int64) -> Bool
```

获取频道第一条未读消息实例

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - channelId: 当前频道ID
/// - Returns: 消息实例
static func fetchFirstUnreadMessage(communityId: String,
channelId: String) -> RCMessage?
```

修改通知等级

修改社区默认通知等级

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - level: 通知等级
/// - success: 成功回调
/// - error: 失败回调
static func setCommunityDefaultNotificationType(communityId: String,
level: RCPushNotificationLevel,
success: @escaping (() -> Void),
error: @escaping ((RCErrorCode) -> Void))
```

获取社区的默认通知等级

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - success: 获取成功回调
/// - error: 获取失败回调
static func getCommunityDefaultNotificationType(communityId: String,
success: @escaping ((RCPushNotificationLevel) -> Void),
error: @escaping ((RCErrorCode) -> Void))
```

修改频道默认通知等级

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - channelId: 当前频道ID
/// - level: 通知等级
/// - success: 修改成功回调
/// - error: 修改失败回调
static func setChannelDefaultNotificationType(communityId: String,
channelId: String,
level: RCPushNotificationLevel,
success: @escaping (() -> Void),
error: @escaping ((RCErrorCode) -> Void))
```

获取频道默认通知等级

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - channelId: 当前频道ID
/// - success: 获取成功回调
/// - error: 获取失败回调
static func getChannelDefaultNotificationType(communityId: String,
channelId: String,
success: @escaping ((RCPushNotificationLevel) -> Void),
error: @escaping ((RCErrorCode) -> Void))
```

修改社区通知等级

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - level: 通知等级
/// - success: 修改成功回调
/// - error: 修改失败回调
static func setCommunityNotificationType(communityId: String,
level: RCPushNotificationLevel,
success: @escaping (() -> Void),
error: @escaping ((RCErrorCode) -> Void))
```

获取社区通知等级

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - success: 获取成功回调
/// - error: 获取失败回调
static func getCommunityNotificationType(communityId: String,
success: @escaping ((RCPushNotificationLevel) -> Void),
error: @escaping ((RCErrorCode) -> Void))
```

设置频道通知等级

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - channelId: 当前频道ID
/// - level: 通知等级
/// - success: 修改成功回调
/// - error: 修改失败回调
static func setChannelNotificationType(communityId: String,
channelId: String,
level: RCPushNotificationLevel,
success: @escaping (() -> Void),
error: @escaping ((RCErrorCode) -> Void))
```

获取频道通知等级

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - channelId: 当前频道ID
/// - success: 获取成功回调
/// - error: 获取失败回调
static func getChannelNotificationType(communityId: String,
channelId: String,
success: @escaping ((RCPushNotificationLevel) -> Void),
error: @escaping ((RCErrorCode) -> Void))
```

社区用户信息管理类

更新时间:2024-05-17

RCSCUserInfoCacheManager 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

获取用户社区信息

```
/// - Parameters:
/// - communityId: 当前社区ID
/// - userId: 当前用户ID
/// - completion: 异步获取用户信息完成的回调
/// - Returns: 同步获取的用户信息，如果本地没有会返回nil
static func getUserInfo(with communityId: String,
userId: String,
completion: ((RCSCCommunityUserInfo?) -> Void)? = nil) -> RCSCCommunityUserInfo?
```

更新用户社区信息

```
/// - Parameters:
/// - communityId: 当前的社区ID
/// - userId: 当前的用户ID
/// - userInfo: 用户信息
static func setUserInfo(with communityId: String,
userId: String,
userInfo: RCSCCommunityUserInfo)
```

更新日志 1.0.0

更新时间:2024-05-17

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:2024-05-17

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。

我看说的是社区需要依赖 IMKit？

因为社区Demo在开发过程中用到了 IMKit 中的相关工具类方法，您可以自行修改。