直播 SDK

无 UI 集成

腾讯云

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100或95716。

文档目录

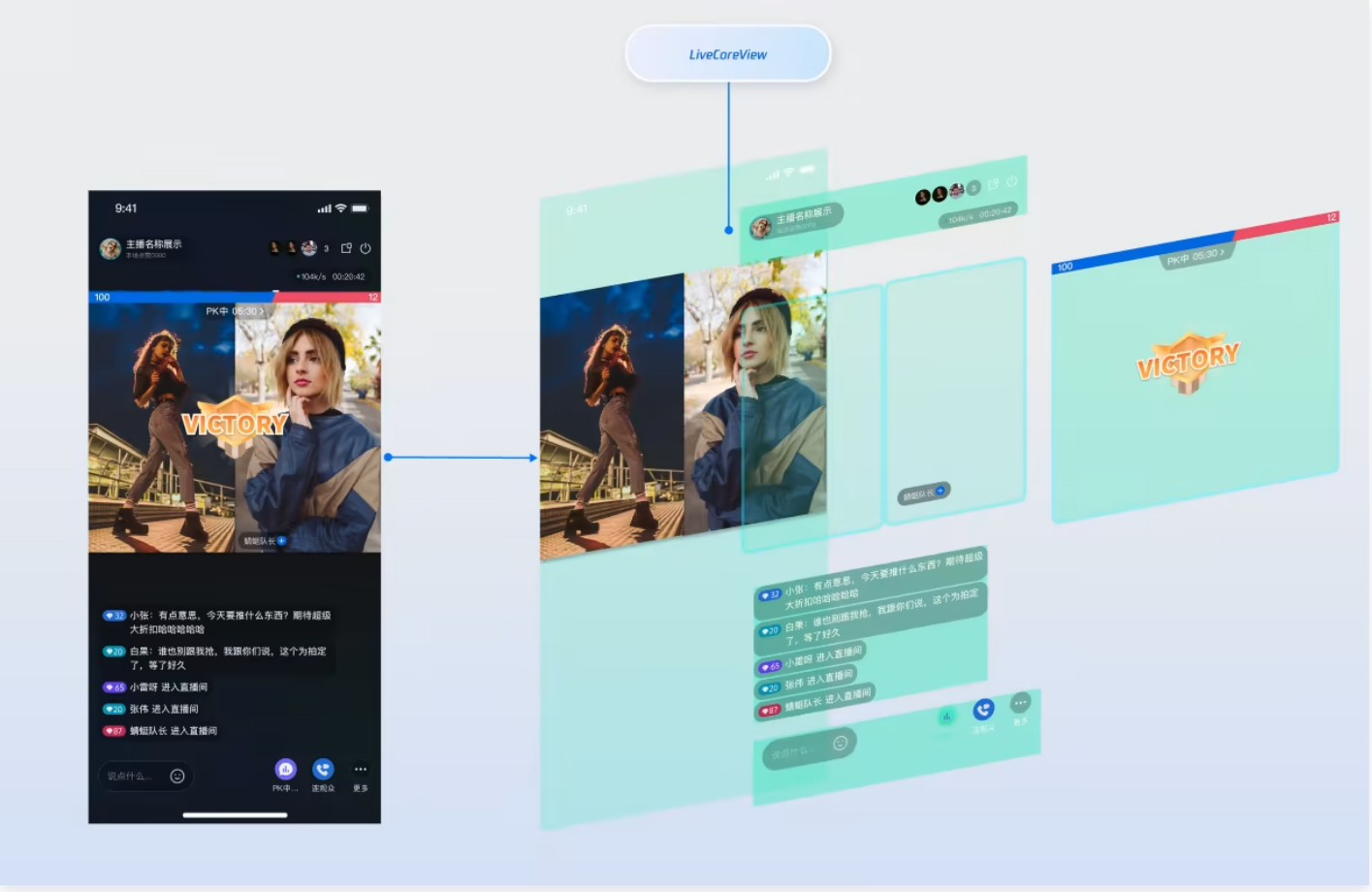
无 UI 集成
搭建视频直播
开始直播
开始直播（Android）

最近更新时间：2025-11-21 14:15:34

本文档将帮助您使用 AtomicXCore SDK的核心组件 LiveCoreView，快速构建一个包含主播开播和观众观看功能的基础直播 App。

核心功能

LiveCoreView 是一个专为直播场景设计的轻量级 `View` 组件，是您构建直播场景的核心，它封装了所有复杂的底层直播技术（如推拉流、连麦、音视频渲染）。您可以将 LiveCoreView 作为直播画面的"画布"，专注于上层 UI 与交互的开发。
通过下方的视图层级示意图，您可以直观了解 LiveCoreView 在直播界面中的位置和作用：



准备工作

步骤1：开通服务

请参见 开通服务，获取体验版或付费版 SDK。

步骤2：在当前项目中导入 AtomicXCore

安装组件：请在您的 build.gradle 文件中添加 `implementation 'com.tencent.atomicx:atomicxcore:latest'` 依赖，然后执行 Gradle Sync。

```
dependencies {
    implementation 'io.trtc.uikit:atomicx-core:latest.release'
    api "io.trtc.uikit:rtc_room_engine:3.4.0.1306"
    api "io.trtc.uikit:atomicx-core:3.4.0.1307"
    api "com.tencent.liteav:LiteAVSDK_Professional:12.8.0.19279"
    api "com.tencent.imsdk:imsdk-plus:8.7.7201"
    // 其他依赖...
}
```

步骤3：实现登录逻辑

在您的项目中调用 `LoginStore.shared.login` 完成登录，**这是使用 `AtomicXCore` 所有功能的关键前提。**

> ⚠ **重要：**
> 推荐在您 App 自身的用户账户登录成功后，再调用 LoginStore.shared.login，以确保登录业务逻辑的清晰和一致。

```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import io.trtc.tuikit.atomicxcore.api.login.LoginStore
import io.trtc.tuikit.atomicxcore.api.CompletionHandler
import android.util.Log

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        LoginStore.shared.login(
            this,               // context
            1400000001,         // 替换为您的 SDKAppID
            "test_001",         // 替换为您的 UserID
```

```
            "xxxxxxxxxx",      // 替换为您的 UserSig
            object : CompletionHandler {
                override fun onSuccess() {
                    // 登录成功处理
                    Log.d("Login", "login success");
                }

                override fun onFailure(code: Int, desc: String) {
                    // 登录失败处理
                    Log.e("Login", "login failed, code: $code, error:
$desc");
                }
            }
        )
    }
}
```

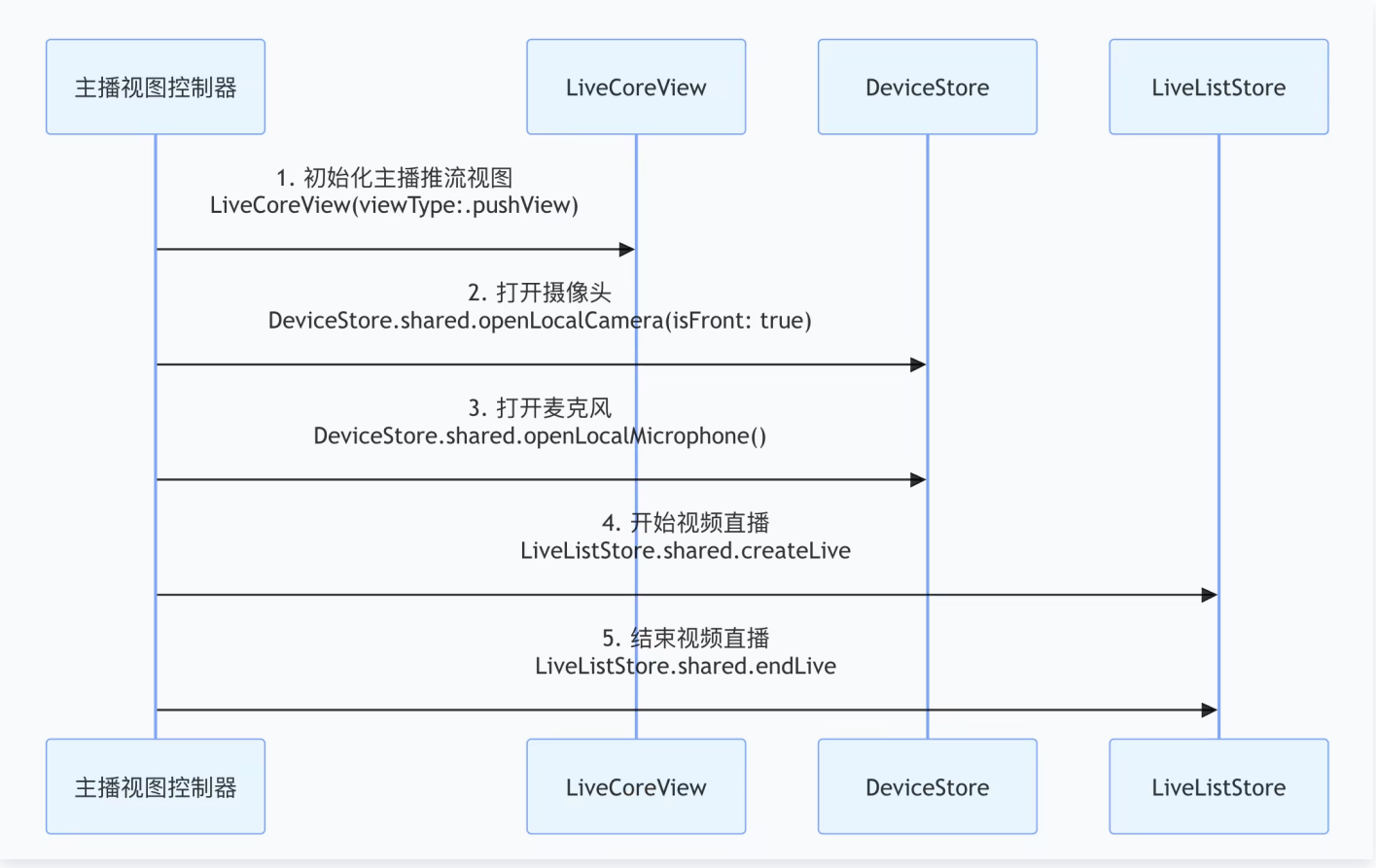
登录接口参数说明：

参数	类型	说明
sdkAppId	Int	从 控制台 获取，通常是以 `140` 或 `160` 开头的 10 位整数。
userID	String	当前用户的唯一 ID，仅包含英文字母、数字、连字符和下划线。为避免多端登录冲突，**请勿使用** `1` 、`123` **等简单 ID**。
userSig	String	用于腾讯云鉴权的票据。请注意： • 开发环境：您可以采用本地 `GenerateTestUserSig.genTestSig` 函数生成 userSig 或者 通过 UserSig 辅助工具 生成临时的 UserSig。 • 生产环境：为了防止密钥泄露，请务必采用服务端生成 UserSig 的方式。详细信息请参考 服务端生成 UserSig 。 更多信息请参见 如何计算及使用 UserSig 。

搭建基础直播间

步骤1：实现主播视频开播

主播开播流程如下，您只需执行以下几步操作，即可快速搭建主播视频直播。



提示：
主播开播推流的业务代码，您也可以参考 TUILiveKit 开源项目中的 VideoLiveAnchorActivity.java 文件来了解完整的实现逻辑。

1. 初始化主播推流的视图

在您的主播 Activity 中，创建一个 `LiveCoreView` 实例，并将 `viewType` 指定为 `PUSH_VIEW`（推流视图）。

```
import io.trtc.tuikit.atomicxcore.api.view.CoreViewType
import io.trtc.tuikit.atomicxcore.api.view.LiveCoreView
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.constraintlayout.widget.ConstraintLayout

// YourAnchorActivity 代表您的主播开播Activity
class YourAnchorActivity : AppCompatActivity() {

    // 1. 将 LiveCoreView 作为您Activity的一个属性
    private lateinit var coreView: LiveCoreView
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // 2. 初始化视图
        coreView = LiveCoreView(this, viewType =
CoreViewType.PUSH_VIEW)
        coreView.setLiveID("test_live_001")

        // 布局 UI
        setupUI()
    }

    private fun setupUI() {
        setContentView(R.layout.activity_anchor)

        // 3. 将主播推流页面加载到您的视图上
        val container = findViewById<ConstraintLayout>(R.id.container)
        container.addView(coreView)

        // 设置布局参数
        val params = ConstraintLayout.LayoutParams(
            ConstraintLayout.LayoutParams.MATCH_PARENT,
            ConstraintLayout.LayoutParams.MATCH_PARENT
        )
        params.topMargin = 36
        params.bottomMargin = 96
        coreView.layoutParams = params
    }
}
```

2. 打开摄像头和麦克风

通过调用 DeviceStore 的 `openLocalCamera`、`openLocalMicrophone` 接口打开摄像头和麦克风，您无需做额外操作，LiveCoreView 会自动预览当前摄像头的视频流，示例代码如下：

```
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import io.trtc.tuikit.atomicxcore.api.device.DeviceStore
```

```
class YourAnchorActivity : AppCompatActivity() {
    // ... 其他代码 ...

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // 布局 UI
        setupUI()
        // 打开设备
        openDevices()
    }

    private fun openDevices() {
        // 1. 打开前置摄像头
        DeviceStore.shared().openLocalCamera(true, completion = null)
        // 2. 打开麦克风
        DeviceStore.shared().openLocalMicrophone(completion = null)
    }
}
```

3. 开始直播

通过调用 LiveListStore 的 `createLive` 接口开始视频直播，完整示例代码如下：

```
import android.util.Log
import androidx.appcompat.app.AppCompatActivity
import io.trtc.tuikit.atomicxcore.api.live.LiveInfo
import io.trtc.tuikit.atomicxcore.api.live.LiveInfoCompletionHandler
import io.trtc.tuikit.atomicxcore.api.live.LiveListStore

class YourAnchorActivity : AppCompatActivity() {

    // ... 其他代码 ...

    // 调用开播接口开始直播
    private fun startLive() {
        val liveInfo = LiveInfo().apply {
            // 1. 设置直播的房间 id
            liveID = "test_live_001"
            // 2. 设置直播的房间名称
            liveName = "test 直播"
```

```
            // 3. 配置布局模板，默认：600 动态宫格布局
            seatLayoutTemplateID = 600
            // 4. 配置主播始终在麦位上
            keepOwnerOnSeat = true
        }
        // 5. 调用 LiveListStore.shared().createLive 开始直播
        LiveListStore.shared().createLive(liveInfo, object:
LiveInfoCompletionHandler {
            override fun onFailure(code: Int, desc: String) {
                Log.e("Live", "Response startLive onError: $desc")
            }

            override fun onSuccess(liveInfo: LiveInfo) {
                Log.d("Live", "Response startLive onSuccess")
            }
        })
    }
}
```

`LiveInfo` 参数说明

参数名	类型	属性	描述
`liveID`	`String`	必填	直播间的唯一标识符。
`liveName`	`String`	选填	直播间的标题。
`notice`	`String`	选填	直播间的公告信息。
`isMessageDisable`	`Boolean`	选填	是否禁言（`true`：是，`false`：否）。
`isPublicVisible`	`Boolean`	选填	是否公开可见（`true`：是，`false`：否）。
`isSeatEnabled`	`Boolean`	选填	是否启用麦位功能（`true`：是，`false`：否）。
`keepOwnerOnSeat`	`Boolean`	选填	是否保持房主在麦位上。
`maxSeatCount`	`Int`	必填	最大麦位数量。

seatMode	TakeSeatMode	选填	上麦模式（FREE：自由上麦，APPLY：申请上麦）。
seatLayoutTemplateID	Int	必填	麦位布局模板 ID。
coverURL	String	选填	直播间的封面图片地址。
backgroundURL	String	选填	直播间的背景图片地址。
categoryList	List<Int>	选填	直播间的分类标签列表。
activityStatus	Int	选填	直播活动状态。
isGiftEnabled	Boolean	选填	是否启用礼物功能（true：是，false：否）。

4. 结束直播

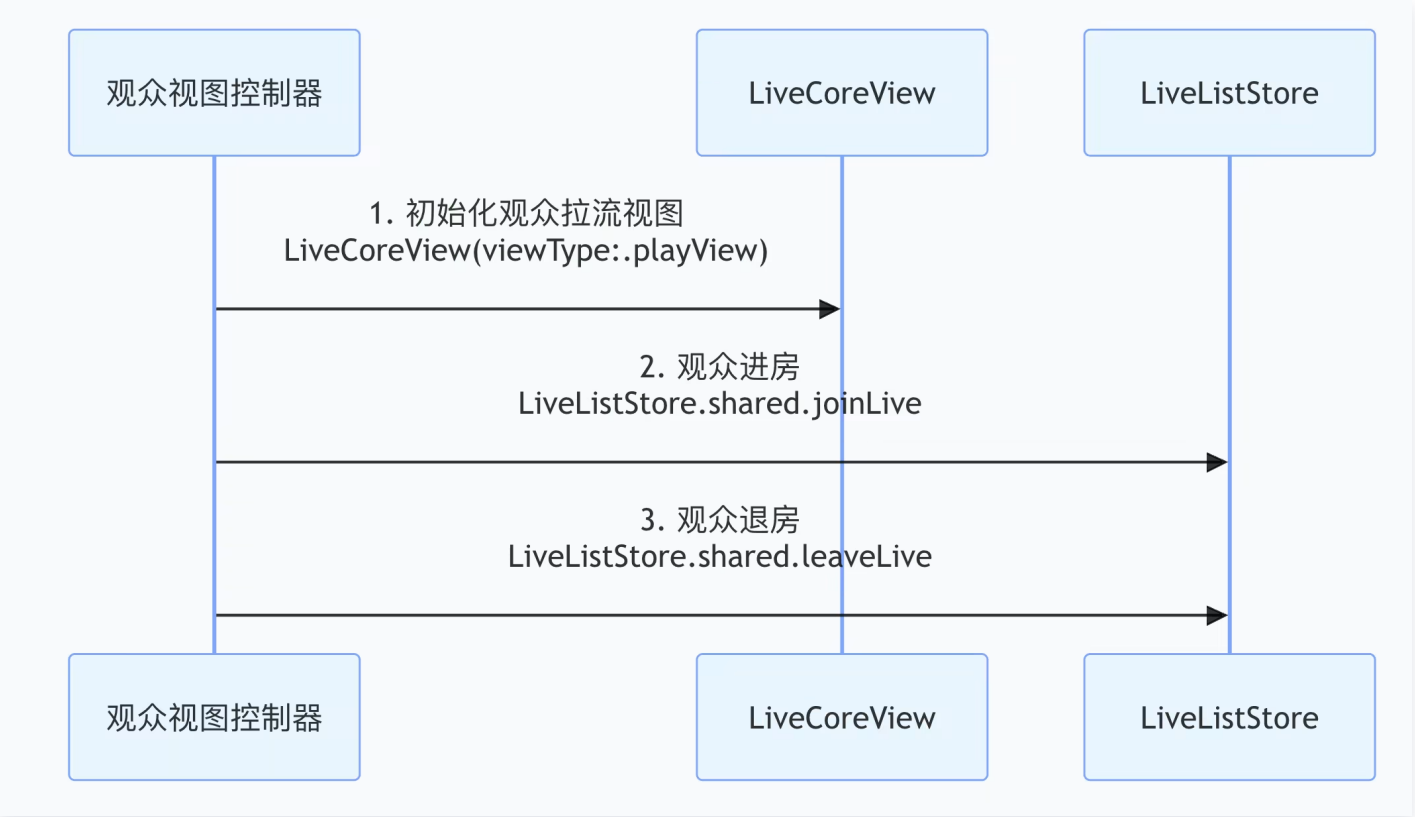
直播结束后，主播可以调用 LiveListStore 的 endLive 接口结束直播。SDK 会处理停止推流和销毁房间的逻辑。

```
import android.util.Log
import androidx.appcompat.app.AppCompatActivity
import com.tencent.cloud.tuikit.engine.extension.TUILiveListManager
import io.trtc.tuikit.atomicxcore.api.device.DeviceStore
import io.trtc.tuikit.atomicxcore.api.live.LiveListStore
import io.trtc.tuikit.atomicxcore.api.live.StopLiveCompletionHandler

class YourAnchorActivity : AppCompatActivity() {

    // ... 其他代码 ...

    // 结束直播
    private fun stopLive() {
        LiveListStore.shared().endLive(object :
StopLiveCompletionHandler {
            override fun onSuccess(statisticsData:
TUILiveListManager.LiveStatisticsData) {
                Log.d("Live", "endLive success, duration:
${statisticsData.liveDuration}, totalViewers:
${statisticsData.totalViewers}")
```

```
            }

            override fun onFailure(code: Int, desc: String) {
                Log.e("Live", "endLive error: $desc")
            }
        })
    }

    // 确保在 Activity 销毁时也调用结束直播
    override fun onDestroy() {
        super.onDestroy()
        stopLive()
        // 关闭本地设备
        DeviceStore.shared().closeLocalCamera()
        DeviceStore.shared().closeLocalMicrophone()
        Log.d("Live", "YourAnchorActivity onDestroy")
    }
}
```

步骤2：实现观众进房观看

观众观看流程如下，通过简单几步操作，即可实现观众观看直播。



> **提示：**
> 观众进房拉流的业务代码，您也可以参考 TUILiveKit 开源项目中的 VideoLiveAudienceActivity.java 文件来了解完整的实现逻辑。

1. 添加观众拉流页面

在您的观众 Activity 中，创建 `LiveCoreView` 实例，并将 `viewType` 指定为 `PLAY_VIEW`（拉流视图）。

```
import io.trtc.tuikit.atomicxcore.api.view.CoreViewType
import io.trtc.tuikit.atomicxcore.api.view.LiveCoreView
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.constraintlayout.widget.ConstraintLayout

// YourAudienceActivity 代表您的观众观看Activity
class YourAudienceActivity : AppCompatActivity() {

    // 1. 初始化观众拉流页面
    private lateinit var coreView: LiveCoreView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // 2. 初始化视图
        coreView = LiveCoreView(this, viewType =
CoreViewType.PLAY_VIEW)
        coreView.setLiveId("test_live_001")

        // UI 布局
        setupUI()
    }

    private fun setupUI() {
        setContentView(R.layout.activity_audience)

        // 3. 将观众拉流页面加载到您的视图上
        val container = findViewById<ConstraintLayout>(R.id.container)
        container.addView(coreView)
```

```
        // 设置布局参数
        val params = ConstraintLayout.LayoutParams(
            ConstraintLayout.LayoutParams.MATCH_PARENT,
            ConstraintLayout.LayoutParams.MATCH_PARENT
        )
        params.topMargin = 36
        params.bottomMargin = 96
        coreView.layoutParams = params
    }
}
```

2. **进入直播间观看**

通过调用 LiveListStore 的 `joinLive` 接口加入直播，您无需做额外操作，LiveCoreView 会自动播放当前房间的视频流，完整示例代码如下：

```
import io.trtc.tuikit.atomicxcore.api.live.LiveInfo
import io.trtc.tuikit.atomicxcore.api.live.LiveInfoCompletionHandler
import io.trtc.tuikit.atomicxcore.api.live.LiveListStore
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import android.util.Log

// YourAudienceActivity 代表您的观众观看Activity
class YourAudienceActivity : AppCompatActivity() {

    // ... 其他代码 ...

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setupUI()
        // 1. 调用 LiveListStore.shared().joinLive 进入直播间
        //        - liveID: 与主播开播同样的 liveID
        LiveListStore.shared().joinLive(liveID, object :
LiveInfoCompletionHandler {
            override fun onFailure(code: Int, desc: String) {
                Log.e("Live", "joinLive error: $desc")
            }

            override fun onSuccess(liveInfo: LiveInfo) {
```

```
                Log.d("Live", "joinLive success")
            }
        })
    }
}
```

3. 退出直播

观众退出直播间时，需要调用 LiveListStore 的 `leaveLive` 接口退出直播。SDK 会自动停止拉流并退出房间。

```
import io.trtc.tuikit.atomicxcore.api.live.LiveListStore
import io.trtc.tuikit.atomicxcore.api.CompletionHandler
import androidx.appcompat.app.AppCompatActivity
import android.util.Log

// YourAudienceActivity 代表您的观众观看Activity
class YourAudienceActivity : AppCompatActivity() {

    // ... 其他代码 ...

    // 退出直播
    private fun leaveLive() {
        LiveListStore.shared().leaveLive(object : CompletionHandler {
            override fun onSuccess() {
                Log.d("Live", "leaveLive success")
            }

            override fun onFailure(code: Int, desc: String) {
                Log.e("Live", "leaveLive error: $desc")
            }
        })
    }

    // 确保在 Activity 销毁时也调用退出直播
    override fun onDestroy() {
        super.onDestroy()
        leaveLive()
        Log.d("Live", "YourAudienceActivity onDestroy")
    }
```

```
}
```

步骤3：监听直播事件

在观众加入直播间后，您还需要处理一些房间内的“被动”事件。例如主播主动结束了直播，或者观众因为违规等原因被踢出房间。如果不监听这些事件，观众端 UI 可能会停留在黑屏页面，影响用户体验。
您可以通过实现 `LiveListListener` 监听器，并将其注册到 `LiveListStore` 来实现事件监听。

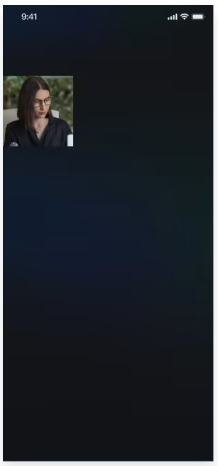
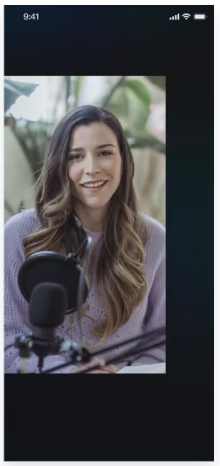
```
import io.trtc.tuikit.atomicxcore.api.live.LiveEndedReason
import io.trtc.tuikit.atomicxcore.api.live.LiveKickedOutReason
import io.trtc.tuikit.atomicxcore.api.live.LiveListListener
import io.trtc.tuikit.atomicxcore.api.live.LiveListStore
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import android.util.Log

// YourAudienceActivity 代表您的观众观看Activity
class YourAudienceActivity : AppCompatActivity() {

    // ... (coreView, liveId, setupUI, joinLive, leaveLive 等代码) ...

    // 2. 定义一个 LiveListListener 实例
    private val liveListListener = object : LiveListListener() {
        override fun onLiveEnded(liveID: String, reason:
LiveEndedReason, message: String) {
            // 监听到直播结束
            Log.d("Live", "Live ended. liveID: $liveID, reason: $reason,
message: $message")
            // 在此处处理退出直播间的逻辑，例如关闭当前 Activity
            // finish()
        }

        override fun onKickedOutOfLive(liveID: String, reason:
LiveKickedOutReason, message: String) {
            // 监听到被踢出直播
            Log.d("Live", "Kicked out of live. liveID: $liveID, reason:
$reason, message: $message")
            // 在此处处理退出直播间的逻辑
            // finish()
        }
```

```
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setupUI()

        // 3. 注册监听器
        LiveListStore.shared().addLiveListListener(liveListListener)

        // 4. 进入直播间
        joinLive()
    }

    // ... joinLive() 和 leaveLive() 方法 ...

    override fun onDestroy() {
        super.onDestroy()
        // 5. 确保在 Activity 销毁时移除监听器，防止内存泄漏
        LiveListStore.shared().removeLiveListListener(liveListListener)
        leaveLive()
        Log.d("Live", "YourAudienceActivity onDestroy")
    }
}
```

运行效果

集成 LiveCoreView 后，您将得到一个纯净的视频渲染视图，它已具备完整的直播业务能力，但没有任何交互 UI。您可以参考下一章节 丰富直播场景 来完善直播场景。

	动态宫格布局	浮动小窗布局	固定宫格布局	固定小窗布局
模板 ID	600	601	800	801
描述	默认布局，可根据连麦人数动态调整宫格大小。	连麦嘉宾以浮动小窗形式显示。	连麦人数固定，每个嘉宾占据一个固定宫格。	连麦人数固定，嘉宾以固定小窗形式显示。

运行效果				

丰富直播场景

当您完成了基础的直播功能后，您可以参考以下功能指南来为直播添加丰富的互动玩法。

直播功能	功能介绍	功能 Stores	实现指南
实现观众音视频连线	观众申请上麦，与主播进行实时视频互动。	CoGuestStore	观众连线
实现主播跨房连线 PK	两个不同房间的主播进行连线，实现互动或 PK。	CoHostStore BattleStore	主播连线 & PK
添加弹幕聊天功能	观众可以在直播间发送和接收实时文字消息。	BarrageStore	实现弹幕功能
构建礼物赠送系统	观众可以向主播赠送虚拟礼物，增加互动和趣味性。	GiftStore	实现礼物功能

API 文档

Store/Component	功能描述	API 文档
LiveCoreView	直播视频流展示与交互的核心视图组件：负责视频流渲染和视图挂件处理，支持主播直播、观众连麦、主播连线等场景。	API 文档
LiveListStore	直播间全生命周期管理：创建 / 加入 / 离开 / 销毁房间，查询房间列表，修改直播信息（名称、公告等），监听直播状态（如被	API 文档

	踢出、结束）。	
DeviceStore	音视频设备控制：麦克风（开关 / 音量）、摄像头（开关 / 切换 / 画质）、屏幕共享，设备状态实时监听。	API 文档
CoGuestStore	观众连麦管理：连麦申请 / 邀请 / 同意 / 拒绝，连麦成员权限控制（麦克风 / 摄像头），状态同步。	API 文档
CoHostStore	主播跨房连线：支持多布局模板（动态网格等），发起 / 接受 / 拒绝连线，连麦主播互动管理。	API 文档
BattleStore	主播 PK 对战：发起 PK（配置时长 / 对手），管理 PK 状态（开始 / 结束），同步分数，监听对战结果。	API 文档
GiftStore	礼物互动：获取礼物列表，发送 / 接收礼物，监听礼物事件（含发送者、礼物详情）。	API 文档
BarrageStore	弹幕功能：发送文本 / 自定义弹幕，维护弹幕列表，实时监听弹幕状态。	API 文档
LikeStore	点赞互动：发送点赞，监听点赞事件，同步总点赞数。	API 文档
LiveAudienceStore	观众管理：获取实时观众列表（ID / 名称 / 头像），统计观众数量，监听观众进出事件。	API 文档
AudioEffectStore	音频特效：变声（童声 / 男声）、混响（KTV 等）、耳返调节，实时切换特效。	API 文档
BaseBeautyStore	基础美颜：调节磨皮 / 美白 / 红润（0–100 级），重置美颜状态，同步效果参数。	API 文档

常见问题

主播调用 createLive 或 观众调用 joinLive 后为什么画面是黑的，没有视频画面?

- **检查 setLiveID**：请确保在调用开播或观看接口前，已经为 LiveCoreView 实例设置了正确的 liveID。
- **检查设备权限**：请确保 App 已获得摄像头和麦克风的系统使用权限。
- **检查主播端**：主播端是否正常调用 `DeviceStore.shared().openLocalCamera(true)` 打开了摄像头。
- **检查网络**：请检查设备网络连接是否正常。

主播端打开摄像头后，开播后可以看到本地视频预览画面，开播前视频预览是黑屏?

- **检查主播端**：请检查主播推流视图 `LiveCoreView` 的 `viewType` 是否配置为 `PUSH_VIEW` 。
- **检查观众端**：请检查观众拉流视图 `LiveCoreView` 的 `viewType` 是否配置为 `PLAY_VIEW` 。

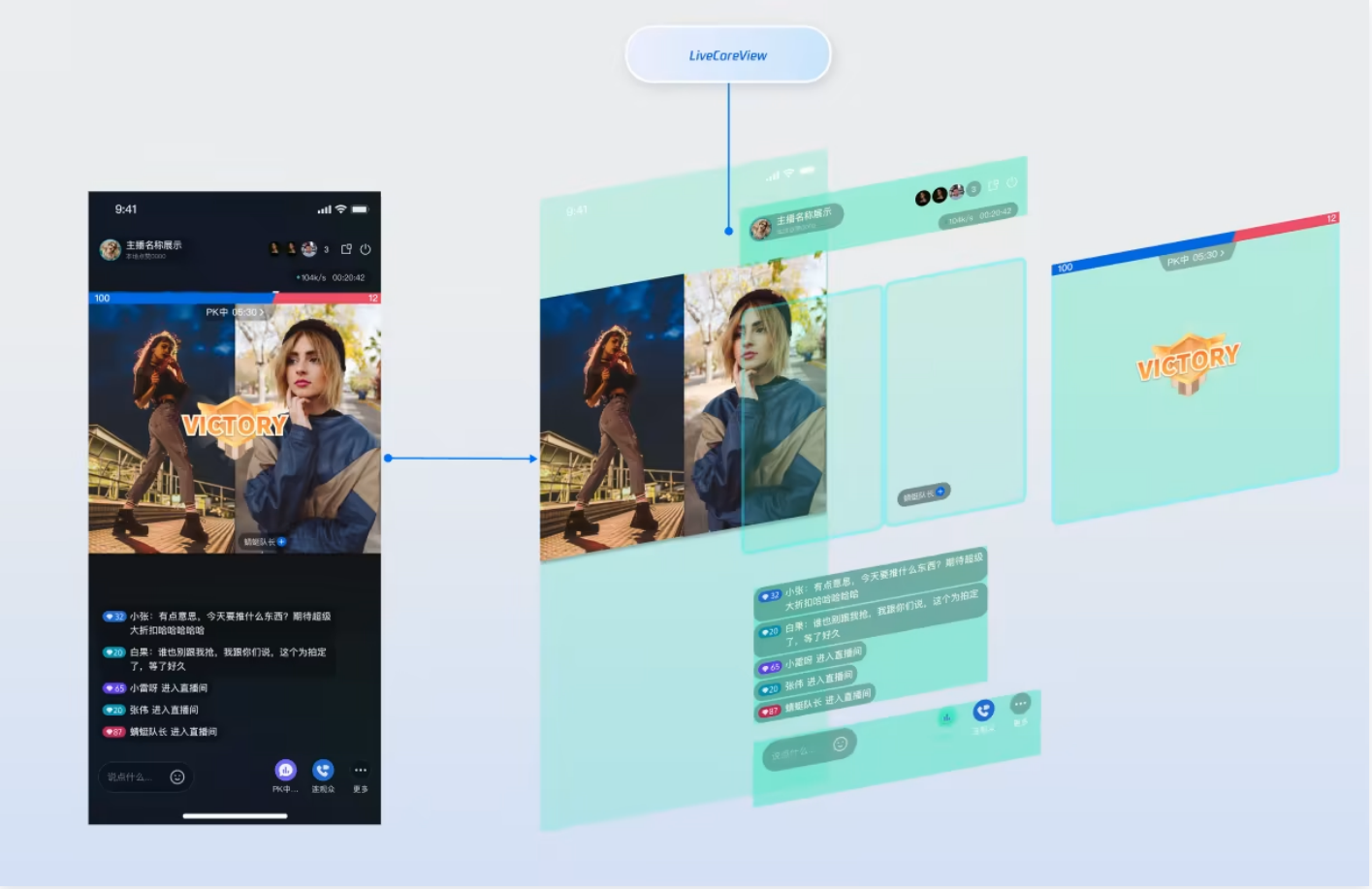
开始直播（iOS）

最近更新时间：2025-11-21 14:15:34

本文档将帮助您使用 AtomicXCore SDK 的核心组件 LiveCoreView，快速构建一个包含主播开播和观众观看功能的基础直播 App。

核心功能

LiveCoreView 是一个专为直播场景设计的轻量级 `UIView` 组件，是您构建直播场景的核心，它封装了所有复杂的底层直播技术（例如推拉流、连麦、音视频渲染）。您可以将 LiveCoreView 作为直播画面的“画布”，专注于上层 UI 与交互的开发。
通过下方的视图层级示意图，您可以直观了解 LiveCoreView 在直播界面中的位置和作用：



准备工作

步骤1：开通服务

请参见 开通服务，获取体验版或付费版 SDK。

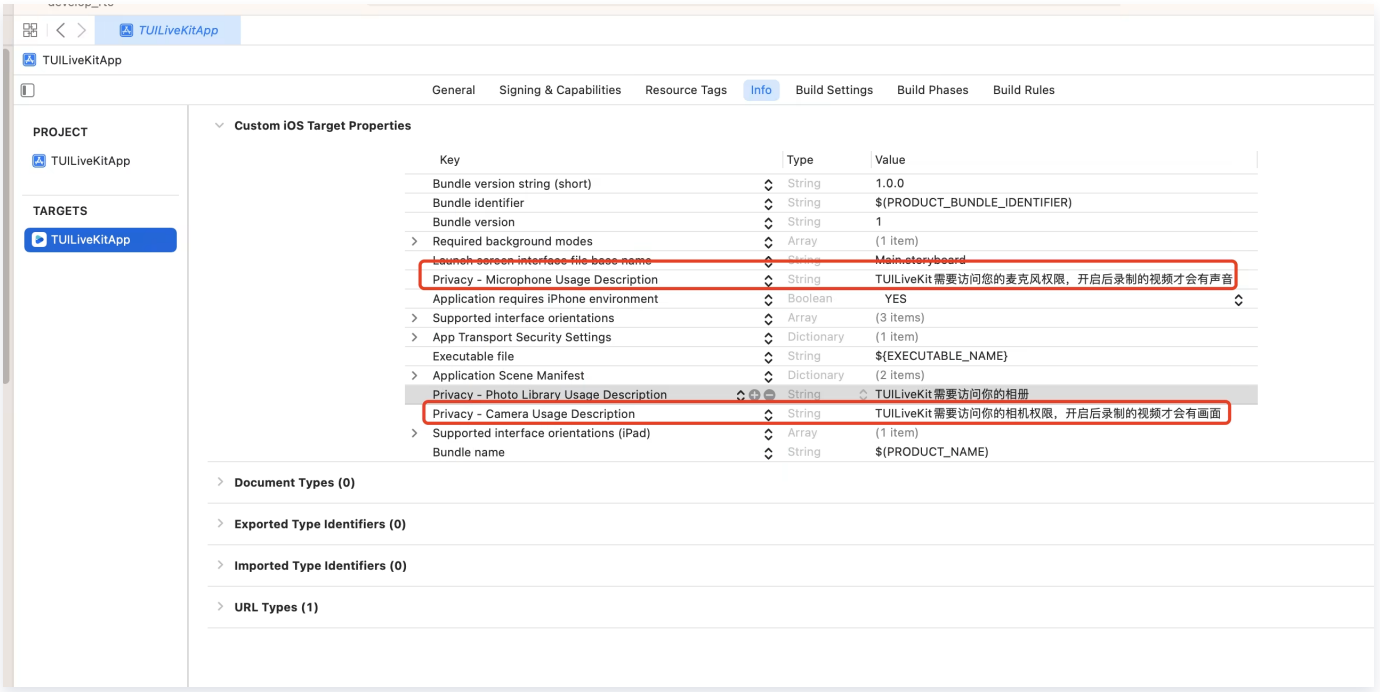
步骤2：在当前项目中导入 AtomicXCore

1. **安装组件**：请在您的 Podfile 文件中添加 `pod 'AtomicXCore'` 依赖，然后执行 `pod install`。

```
target 'xxxx' do
  pod 'AtomicXCore'
end
```

2. **配置工程权限：**请在应用的 `Info.plist` 文件中添加相机和麦克风的使用权限说明。

```
<key>NSCameraUsageDescription</key>
<string>TUILiveKit需要访问你的相机权限，开启后录制的视频才会有画面</string>
<key>NSMicrophoneUsageDescription</key>
<string>TUILiveKit需要访问您的麦克风权限，开启后录制的视频才会有声音</string>
```



步骤3：实现登录逻辑

在您的项目中调用 `LoginStore.shared.login` 完成登录，**这是使用 `AtomicXCore` 所有功能的关键前提。**

> ⚠ **重要：**
> 推荐在您 App 自身的用户账户登录成功后，再调用 LoginStore.shared.login，以确保登录业务逻辑的清晰和一致。

```
import AtomicXCore

//  AppDelegate.swift
```

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    LoginStore.shared.login(sdkAppID: 1400000001,                    // 替
换为您的 SDKAppID
                            userID: "test_001",                      // 替
换为您的 UserID
                            userSig: "xxxxxxxxxxx") { result in      // 替
换为您的 UserSig
      switch result {
        case .success(let info):
        debugPrint("login success")
        case .failure(let error):
        debugPrint("login failed code:\(error.code), message:\
(error.message)")
      }
    }
    return true
}
```

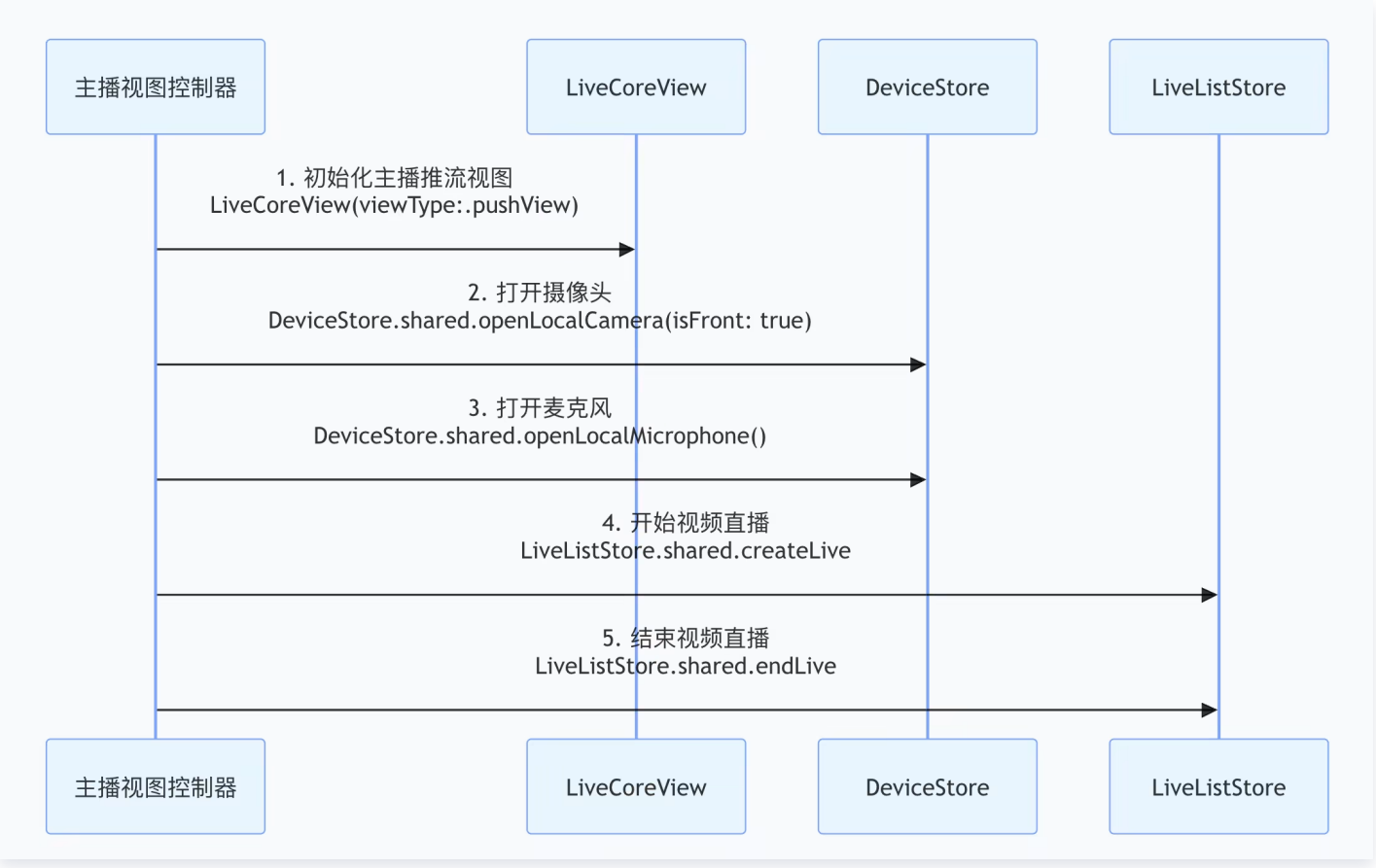
登录接口参数说明：

参数	类型	说明
sdkAppId	Int32	从 控制台 获取，通常是以 `140` 或 `160` 开头的 10 位整数。
userID	String	当前用户的唯一 ID，仅包含英文字母、数字、连字符和下划线。为避免多端登录冲突，**请勿使用** `1` 、`123` **等简单 ID**。
userSig	String	用于腾讯云鉴权的票据。请注意： • 开发环境：您可以采用本地 `GenerateTestUserSig.genTestSig` 函数生成 userSig 或者通过 UserSig 辅助工具 生成临时的 UserSig。 • 生产环境：为了防止密钥泄露，请务必采用服务端生成 UserSig 的方式。详细信息请参考 服务端生成 UserSig 。 更多信息请参见 如何计算及使用 UserSig 。

搭建基础直播间

步骤1：实现主播视频开播

主播开播流程如下，您只需执行以下几步操作，即可快速搭建主播视频直播。



提示：
主播开播推流的业务代码，您也可以参考 TUILiveKit 开源项目中的 TUILiveRoomAnchorViewController.swift 文件来了解完整的实现逻辑。

1. 初始化主播推流的视图

在您的主播视图控制器中，创建一个 `LiveCoreView` 实例，并将 `viewType` 指定为 `.pushView`（推流视图）。

```
import AtomicXCore

// YourAnchorViewController 代表您的主播开播视图控制器
class YourAnchorViewController: UIViewController {

    private let liveId: String
    // 1. 将 LiveCoreView 作为您视图控制器的一个属性
    private let coreView = LiveCoreView(viewType:.pushView, frame:
UIScreen.main.bounds)

    // 2. 新增便利构造函数完成实例初始化
    //     - liveId: 您需要开播的直播房间 id
```

```
    public init(liveId: String) {
        self.liveId = liveId
        super.init(nibName: nil, bundle: nil)
        self.coreView.setLiveID(liveId)
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    public override func viewDidLoad() {
        super.viewDidLoad()
        // 3. 将主播推流页面加载到您的视图上
        view.addSubview(coreView)
    }
}
```

2. **打开摄像头和麦克风**

通过调用 DeviceStore 的 `openLocalCamera`、`openLocalMicrophone` 接口打开摄像头和麦克风，您无需做额外操作，LiveCoreView 会自动预览当前摄像头的视频流，示例代码如下：

```
import AtomicXCore

// YourAnchorViewController 代表您的主播开播视图控制器
class YourAnchorViewController: UIViewController {
    // ... 其他代码 ...
    public override func viewDidLoad() {
        super.viewDidLoad()
        // 打开设备
        openDevices()
    }

    private func openDevices() {
        // 1. 打开前置摄像头
        DeviceStore.shared.openLocalCamera(isFront: true, completion:
nil)
        // 2. 打开麦克风
        DeviceStore.shared.openLocalMicrophone(completion: nil)
    }
```

```
}
```

3. 开始直播

通过调用 LiveListStore 的 `createLive` 接口开始视频直播，完整示例代码如下：

```
import AtomicXCore

// YourAnchorViewController 代表您的主播开播视图控制器
class YourAnchorViewController: UIViewController {

    // ... 其他代码 ...

    // 调用开播接口开始直播
    private func startLive() {
        var liveInfo = LiveInfo()
        // 1. 设置直播的房间 id
        liveInfo.liveID = self.liveId
        // 2. 设置直播的房间名称
        liveInfo.liveName = "test 直播"
        // 3. 配置布局模板，默认：600 动态宫格布局
        liveInfo.seatLayoutTemplateID = 600
        // 4. 配置主播始终在麦位上
        liveInfo.keepOwnerOnSeat = true
        // 5. 调用 LiveListStore.shared.createLive 开始直播
        LiveListStore.shared.createLive(liveInfo) { [weak self] result
in
            guard let self = self else { return }
            switch result {
            case .success(let info):
                debugPrint("startLive success")
            case .failure(let error):
                debugPrint("startLive error:\(error.message)")
            }
        }
    }
}
```

`LiveInfo` 参数说明：

参数名	类型	属性	描述
liveID	String	必填	直播间的唯一标识符。
liveName	String	选填	直播间的标题。
notice	String	选填	直播间的公告信息。
isMessageDisable	Bool	选填	是否禁言（true：是，false：否）。
isPublicVisible	Bool	选填	是否公开可见（true：是，false：否）。
isSeatEnabled	Bool	选填	是否启用麦位功能（true：是，false：否）。
keepOwnerOnSeat	Bool	选填	是否保持房主在麦位上。
maxSeatCount	Int	必填	最大麦位数量。
seatMode	TakeSeatMode	选填	上麦模式（.free：自由上麦，.apply：申请上麦）。
seatLayoutTemplateID	UInt	必填	麦位布局模板 ID。
coverURL	String	选填	直播间的封面图片地址。
backgroundURL	String	选填	直播间的背景图片地址。
categoryList	[NSNumber]	选填	直播间的分类标签列表。
activityStatus	Int	选填	直播活动状态。
isGiftEnabled	Bool	选填	是否启用礼物功能（true：是，false：否）。

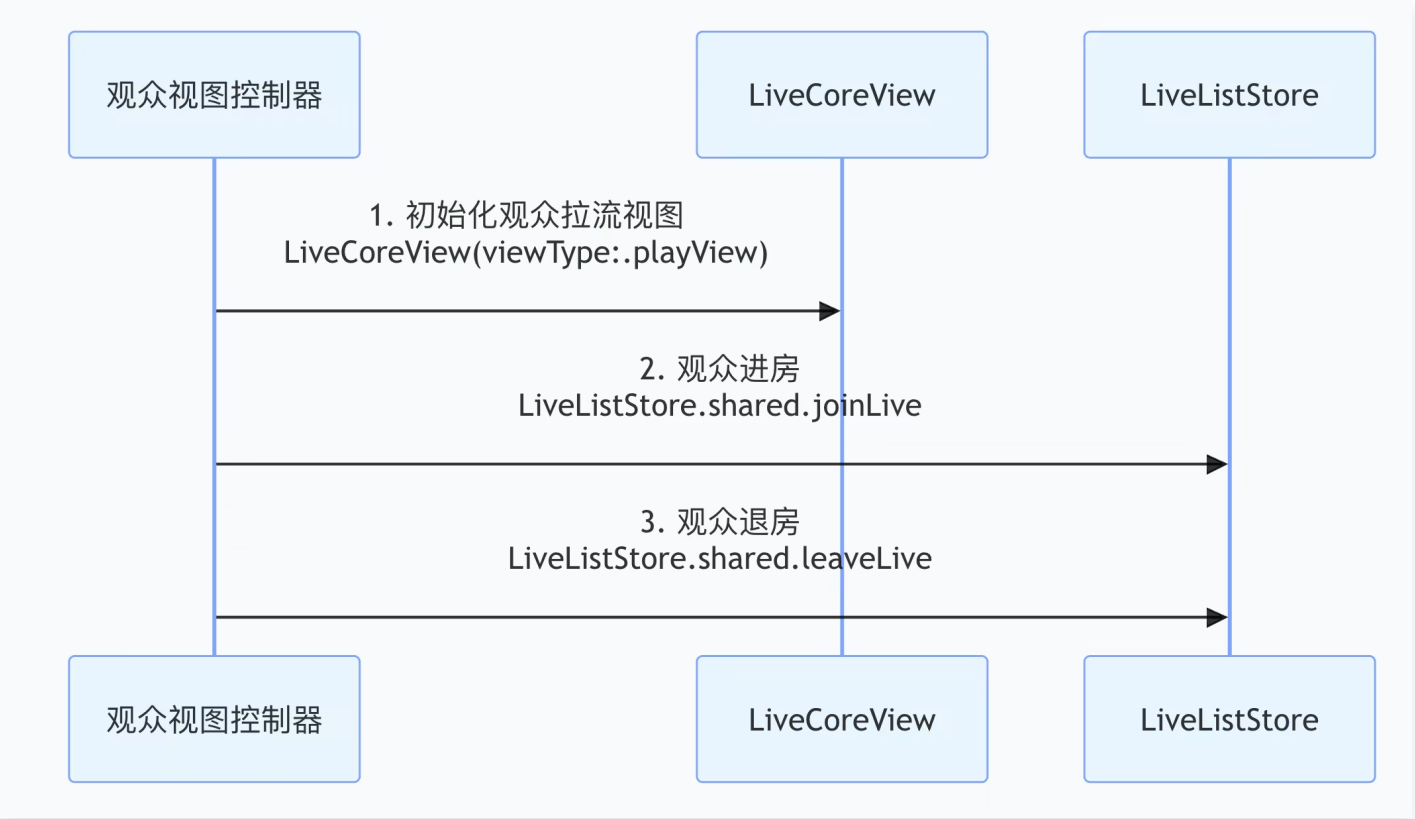
4. 结束直播

直播结束后，主播可以调用 LiveListStore 的 `endLive` 接口结束直播。SDK 会处理停止推流和销毁房间的逻辑。

```
import AtomicXCore

// YourAnchorViewController 代表您的主播开播视图控制器
class YourAnchorViewController: UIViewController {

    // ... 其他代码 ...

    // 结束直播
    private func stopLive() {
        LiveListStore.shared.endLive { [weak self] result in
            guard let self = self else { return }
            switch result {
            case .success(let data):
                debugPrint("endLive success")
            case .failure(let error):
                debugPrint("endLive error: \(error.message)")
            }
        }
    }
}
```

步骤2：实现观众进房观看

观众观看流程如下，通过简单几步操作，即可实现观众观看直播。



提示：

观众进房拉流的业务代码，您也可以参考 TUILiveKit 开源项目中的 TUILiveRoomAudienceViewController.swift 文件来了解完整的实现逻辑。

1. 实现观众拉流页面

在您的观众视图控制器中，创建 `LiveCoreView` 实例，并将 `viewType` 指定为 `.playView`（拉流视图）。

```
import AtomicXCore

// YourAudienceViewController 代表您的观众观看视图控制器
class YourAudienceViewController: UIViewController {

    // 1. 初始化观众拉流页面
    private let coreView = LiveCoreView(viewType:.playView, frame:
UIScreen.main.bounds)
    private let liveId: String

    public init(liveId: String) {
        self.liveId = liveId
        super.init(nibName: nil, bundle: nil)
```

```
        // 2. 绑定直播 id
        self.coreView.setLiveID(liveId)
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    public override func viewDidLoad() {
        super.viewDidLoad()
        // 3. 将主播推流页面加载到您的视图上
        view.addSubview(coreView)
    }
}
```

2. **进入直播间观看**

通过调用 LiveListStore 的 `joinLive` 接口加入直播，您无需做额外操作，LiveCoreView 会自动播放当前房间的视频流，完整示例代码如下：

```
import AtomicXCore

// YourAudienceViewController 代表您的观众观看视图控制器
class YourAudienceViewController: UIViewController {
    // ... 其他代码 ...

    public override func viewDidLoad() {
        super.viewDidLoad()
        // 3. 将主播推流页面加载到您的视图上
        view.addSubview(coreView)

        // 4. 进入直播间
        joinLive()
    }

    private func joinLive() {
         // 调用 LiveListStore.shared.joinLive 进入直播间
        // - liveId: 与主播开播同样的 liveId
        LiveListStore.shared.joinLive(liveID: liveId) { [weak self]
result in
```

```
            guard let self = self else { return }
            switch result {
            case .success(let info):
                debugPrint("joinLive success")
            case .failure(let error):
                debugPrint("joinLive error \(error.message)")
                // 进房失败，也需要退出页面
                // self.dismiss(animated: true)
            }
        }
    }
}
```

3. 退出直播

观众退出直播间时，需要调用 LiveListStore 的 `leaveLive` 接口退出直播。SDK 会自动停止拉流并退出房间。

```
import AtomicXCore

// YourAudienceViewController 代表您的观众观看视图控制器
class YourAudienceViewController: UIViewController {

    // ... 其他代码 ...

    // 退出直播
    private func leaveLive() {
        LiveListStore.shared.leaveLive { [weak self] result in
            guard let self = self else { return }
            switch result {
            case .success:
                debugPrint("leaveLive success")
            case .failure(let error):
                debugPrint("leaveLive error \(error.message)")
            }
        }
    }
}
```

步骤3：监听直播事件

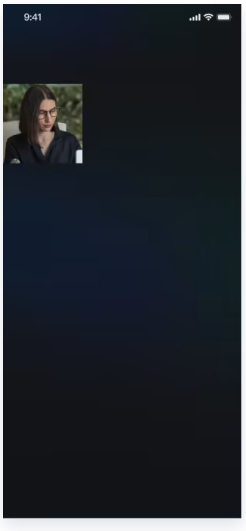
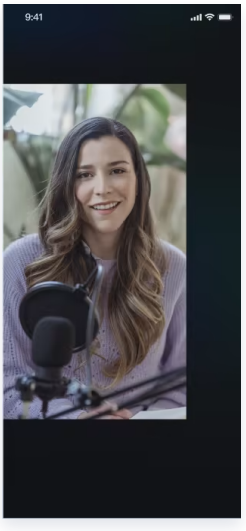
在观众加入直播间后，您还需要处理一些房间内的“被动”事件。例如，主播主动结束了直播，或者观众因为违规等原因被踢出房间。如果不监听这些事件，观众端 UI 可能会停留在黑屏页面，影响用户体验。
您可以通过订阅 `LiveListStore` 提供的 `liveListEventPublisher` 来实现事件监听。

```
import AtomicXCore
import Combine // 1. 导入 Combine 框架

// YourAudienceViewController 代表您的观众观看视图控制器
class YourAudienceViewController: UIViewController {

    // ... 其他代码 (coreView, liveId, init, deinit, joinLive, leaveLive
等) ...

    // 2. 定义 cancellableSet 来管理订阅生命周期
    private var cancellableSet: Set<AnyCancellable> = []

    public override func viewDidLoad() {
        super.viewDidLoad()
        view.addSubview(coreView)
        // 3. 监听直播事件
        setupLiveEventListener()
        // 4. 进入直播间
        joinLive()
    }

    // 5. 新增一个方法来设置事件监听
    private func setupLiveEventListener() {
        LiveListStore.shared.liveListEventPublisher
            .receive(on: RunLoop.main) // 确保在主线程处理 UI 更新
            .sink { [weak self] event in
                guard let self = self else { return }
                switch event {
                case .onLiveEnded(let liveID, let reason, let message):
                    // 监听到直播结束
                    debugPrint("Live ended. liveID: \(liveID), reason: \
(reason.rawValue), message: \(message)")
                    // 在此处处理退出直播间的逻辑，例如关闭当前页面
                    // self.dismiss(animated: true)
                case .onKickedOutOfLive(let liveID, let reason, let
message):
```

```
                    // 监听到被踢出直播
                    debugPrint("Kicked out of live. liveID: \(liveID),
reason: \(reason.rawValue), message: \(message)")
                    // 在此处处理退出直播间的逻辑
                    // self.dismiss(animated: true)
                }
            }
            .store(in: &cancellableSet) // 管理订阅
    }

    // ... joinLive() 和 leaveLive() 方法 ...
}
```

运行效果

集成 LiveCoreView 后，您将得到一个纯净的视频渲染视图，它已具备完整的直播业务能力，但没有任何交互 UI。您可以参考下一章节 丰富直播场景 来完善直播场景。

	动态宫格布局	浮动小窗布局	固定宫格布局	固定小窗布局
模板 ID	600	601	800	801
描述	默认布局，可根据连麦人数动态调整宫格大小。	连麦嘉宾以浮动小窗形式显示。	连麦人数固定，每个嘉宾占据一个固定宫格。	连麦人数固定，嘉宾以固定小窗形式显示。
运行效果				

丰富直播场景

当您完成了基础的直播功能后，您可以参考以下功能指南来为直播添加丰富的互动玩法。

直播功能	功能介绍	功能 Stores	实现指南
实现观众音视频连线	观众申请上麦，与主播进行实时视频互动。	CoGuestStore	实现观众连线
实现主播跨房连线 PK	两个不同房间的主播进行连线，实现互动或 PK。	CoHostStore BattleStore	实现主播连线 PK
添加弹幕聊天功能	观众可以在直播间发送和接收实时文字消息。	BarrageStore	实现弹幕功能
构建礼物赠送系统	观众可以向主播赠送虚拟礼物，增加互动和趣味性。	GiftStore	实现礼物功能

API 文档

Store/Component	功能描述	API 文档
LiveCoreView	直播视频流展示与交互的核心视图组件：负责视频流渲染和视图挂件处理，支持主播直播、观众连麦、主播连线等场景。	API 文档
LiveListStore	直播间全生命周期管理：创建 / 加入 / 离开 / 销毁房间，查询房间列表，修改直播信息（名称、公告等），监听直播状态（如被踢出、结束）。	API 文档
DeviceStore	音视频设备控制：麦克风（开关 / 音量）、摄像头（开关 / 切换 / 画质）、屏幕共享，设备状态实时监听。	API 文档
CoGuestStore	观众连麦管理：连麦申请 / 邀请 / 同意 / 拒绝，连麦成员权限控制（麦克风 / 摄像头），状态同步。	API 文档
CoHostStore	主播跨房连线：支持多布局模板（动态网格等），发起 / 接受 / 拒绝连线，连麦主播互动管理。	API 文档
BattleStore	主播 PK 对战：发起 PK（配置时长 / 对手），管理 PK 状态（开始 / 结束），同步分数，监听对战结果。	API 文档
GiftStore	礼物互动：获取礼物列表，发送 / 接收礼物，监听礼物事件（含发送者、礼物详情）。	API 文档
BarrageStore	弹幕功能：发送文本 / 自定义弹幕，维护弹幕列表，实时监听弹幕状态。	API 文档
LikeStore	点赞互动：发送点赞，监听点赞事件，同步总点赞数。	API 文档

LiveAudienceStore	观众管理：获取实时观众列表（ID / 名称 / 头像），统计观众数量，监听观众进出事件。	API 文档
AudioEffectStore	音频特效：变声（童声 / 男声）、混响（KTV 等）、耳返调节，实时切换特效。	API 文档
BaseBeautyStore	基础美颜：调节磨皮 / 美白 / 红润（0–100 级），重置美颜状态，同步效果参数。	API 文档

常见问题

主播调用 createLive 或 观众调用 joinLive 后为什么画面是黑的，没有视频画面?

- **检查 setLiveID**：请确保在调用开播或观看接口前，已经为 LiveCoreView 实例设置了正确的 liveID。
- **检查设备权限**：请确保 App 已获得摄像头和麦克风的系统使用权限。
- **检查主播端**：主播端是否正常调用 `DeviceStore.shared.openLocalCamera(isFront: true)` 打开了摄像头。
- **检查网络**：请检查设备网络连接是否正常。

主播端打开摄像头后，开播后可以看到本地视频预览画面，开播前视频预览是黑屏?

- **检查主播端**：请检查主播推流视图 `LiveCoreView` 的 `viewType` 是否配置为 `.pushView`。
- **检查观众端**：请检查观众拉流视图 `LiveCoreView` 的 `viewType` 是否配置为 `.playView`。

rsync 因权限不足导致依赖三方库资源拷贝失败

例如在使用 Xcode 集成 SnapKit 框架进行开发时，可能会遇到如下编译报错：

```
rsync(xxxx):1:1: SnapKit.framework/SnapKit_Privacy.bundle/: mkpathat:
Operation not permitted
```

问题原因

Xcode 的用户**脚本沙盒机制**限制了构建过程中 `rsync` 脚本对 `SnapKit.framework` 资源文件的写入权限，导致框架内的 `SnapKit_Privacy.bundle` 无法正常拷贝。

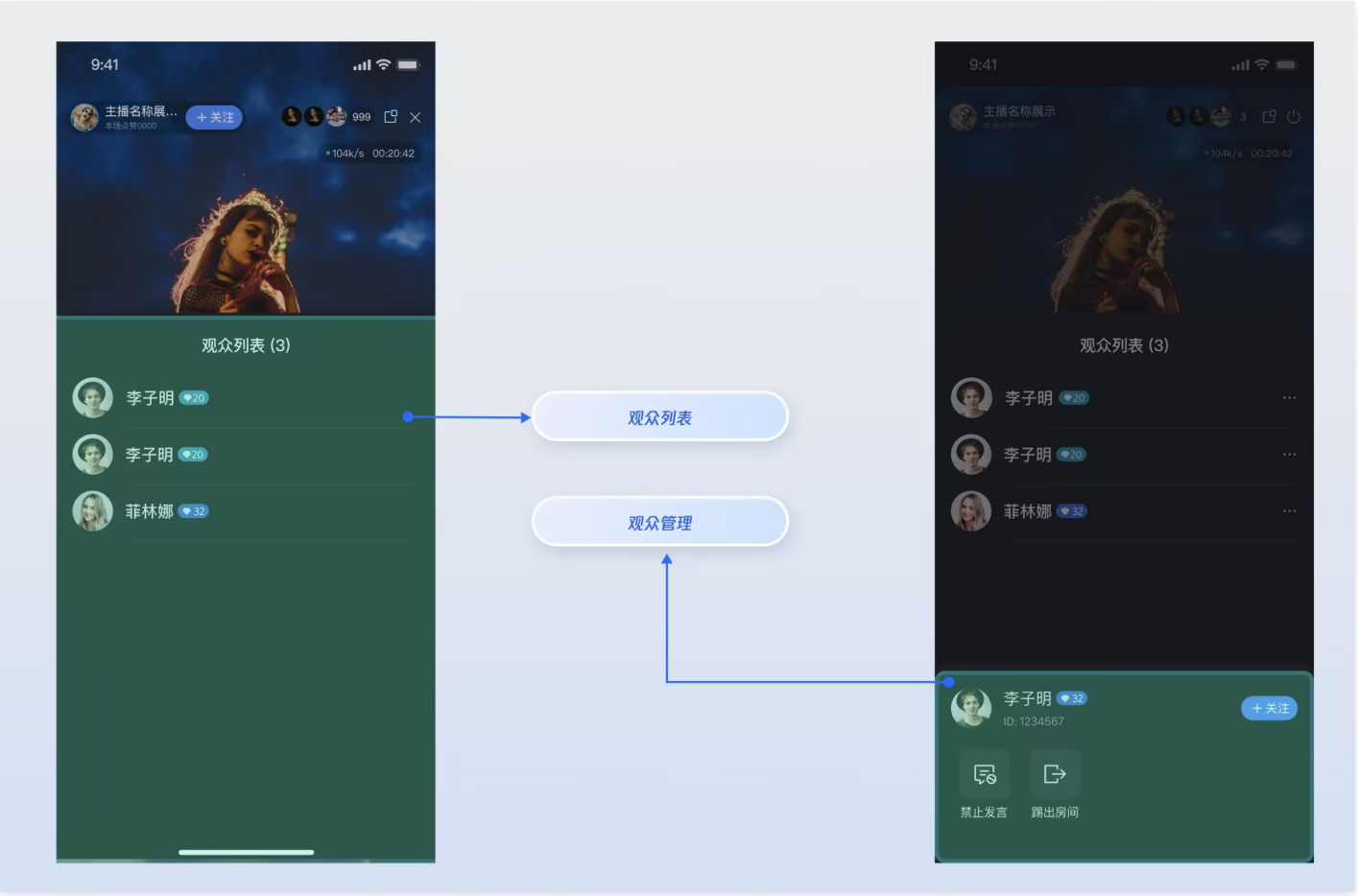
解决步骤

1. 关闭用户脚本沙盒打开 Xcode 项目，进入项目的 **Build Settings**，搜索 `User Script Sandboxing`（或标识 `ENABLE_USER_SCRIPT_SANDBOXING`），将其值从 `YES` 修改为 `NO`。
2. 清理并重新构建项目执行 **Product → Clean Build Folder** 清理项目缓存，然后重新编译运行即可。

观众列表
观众列表（Android）

最近更新时间：2025-11-21 14:15:34

`LiveAudienceStore` 是 `AtomicXCore` 中专门负责管理直播间观众信息的模块。通过 `LiveAudienceStore`，您可以为您的直播应用构建一套完整的观众列表及管理系统。



核心功能

- **实时观众列表**：获取并展示当前在直播间内的所有观众信息。
- **观众人数统计**：实时获取当前直播间的准确观众总数。
- **观众动态监听**：通过事件订阅，实时感知观众的加入和离开。
- **管理员权限**：主播可以将普通观众设为管理员，或撤销其管理员身份。
- **房间管理**：主播或管理员可以将任意普通观众请出（踢出）直播间。

核心概念

在开始集成之前，我们先通过下表了解一下 LiveAudienceStore 相关的几个核心概念：

核心概念	类型	核心职责与描述
`LiveUserInfo`	data class	代表一个观众（用户）的基础信息模型。它包含了用户的唯一标识（`userID`）、昵称（`userName`）和头像地址（`avatarURL`）。
`LiveAudienceState`	data class	代表观众模块的当前状态。其核心属性 `audienceList` 是一个 StateFlow，存储了观众列表的实时状态；`audienceCount` 则代表当前观众总数的实时状态。
`LiveAudienceListener`	abstract class	代表观众动态的实时事件。分为 `onAudienceJoined`（观众加入）和 `onAudienceLeft`（观众离开）两种，用于实现对观众列表的增量更新。
`LiveAudienceStore`	abstract class	这是与观众列表功能交互的核心管理类。通过它，您可以获取观众列表快照、执行管理操作，并通过订阅其 `LiveAudienceListener` 来接收实时动态。

实现指南

步骤1：组件集成

请参考 开始直播 集成 AtomicXCore，完成接入。

步骤2：初始化并获取观众列表

获取一个与当前直播间 liveID 绑定的 `LiveAudienceStore` 实例，并主动拉取一次当前的观众列表，用于首次展示。

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import io.trtc.tuikit.atomicxcore.api.CompletionHandler
import io.trtc.tuikit.atomicxcore.api.live.LiveAudienceStore
import io.trtc.tuikit.atomicxcore.api.live.LiveUserInfo

class AudienceManager(
    private val liveId: String
) {
    private val audienceStore: LiveAudienceStore =
LiveAudienceStore.create(liveId)
    private val scope = CoroutineScope(Dispatchers.Main)
```

```
    // 对外暴露【全量】观众列表的状态流，方便UI层订阅
    private val _audienceList = MutableStateFlow<List<LiveUserInfo>>
(emptyList())
    val audienceList: StateFlow<List<LiveUserInfo>> =
_audienceList.asStateFlow()

    // 对外暴露观众总数的状态流
    private val _audienceCount = MutableStateFlow(0)
    val audienceCount: StateFlow<Int> = _audienceCount.asStateFlow()

    init {
        // 1. 订阅状态和事件，实现见下一节
        subscribeToAudienceState()
        subscribeToAudienceEvents()

        // 2. 主动拉取首屏数据
        fetchInitialAudienceList()
    }

    /// 主动获取一次观众列表快照
    private fun fetchInitialAudienceList() {
        audienceStore.fetchAudienceList(object : CompletionHandler {
            override fun onSuccess() {
                println("首次拉取观众列表成功")
                // 成功后，数据会通过下面的 state 订阅通道自动更新
            }

            override fun onFailure(code: Int, desc: String) {
                println("首次拉取观众列表失败: $desc")
            }
        })
    }

    // ... 后续代码
}
```

步骤3：监听观众列表状态与实时动态

订阅 `audienceStore` 的 `liveAudienceState` 和添加 `LiveAudienceListener`，以接收全量列表快照和实时的观众进出事件，从而驱动UI更新。

```
class AudienceManager {
    /// 订阅 state，用于获取观众总数和列表快照
    private fun subscribeToAudienceState() {
        scope.launch {
            // audienceList 是一个全量快照
            audienceStore.liveAudienceState.audienceList.collect {
audienceList ->
                _audienceList.value = audienceList
            }
        }

        scope.launch {
            // audienceCount 是实时总数
            audienceStore.liveAudienceState.audienceCount.collect {
count ->
                _audienceCount.value = count
            }
        }
    }

    /// 订阅 event，用于处理观众的实时进出事件
    private fun subscribeToAudienceEvents() {
        audienceStore.addLiveAudienceListener(object :
LiveAudienceListener() {
            override fun onAudienceJoined(newAudience: LiveUserInfo) {
                println("观众 ${newAudience.userName} 加入了直播间")
                // 增量更新：在当前列表末尾添加新观众
                val currentList = _audienceList.value.toMutableList()
                if (!currentList.any { it.userID == newAudience.userID
}) {
                    currentList.add(newAudience)
                    _audienceList.value = currentList
                }
            }

            override fun onAudienceLeft(departedAudience: LiveUserInfo)
{
                println("观众 ${departedAudience.userName} 离开了直播间")
                // 增量更新：从当前列表中移除离开的观众
```

```
                val currentList = _audienceList.value.toMutableList()
                currentList.removeAll { it.userID ==
departedAudience.userID }
                _audienceList.value = currentList
            }
        })
    }
}
```

步骤4：管理观众（踢人与设置管理员）

作为主播或管理员，对直播间内的观众进行管理操作。

4.1 将观众踢出直播间

调用 `kickUserOutOfRoom` 接口可以将指定用户请出直播间。

```
class AudienceManager {
    fun kick(userId: String) {
        audienceStore.kickUserOutOfRoom(userId, object :
CompletionHandler {
            override fun onSuccess() {
                println("成功将用户 $userId 踢出房间")
                // 踢出成功后，您会收到 onAudienceLeft 事件
            }

            override fun onFailure(code: Int, desc: String) {
                println("踢出用户 $userId 失败: $desc")
            }
        })
    }
}
```

4.2 设置/撤销管理员

通过 `setAdministrator` 和 `revokeAdministrator` 接口可以管理用户的管理员身份。

```
class AudienceManager {
    /// 将用户设为管理员
    fun promoteToAdmin(userId: String) {
```

```
        audienceStore.setAdministrator(userId, object :
CompletionHandler {
            override fun onSuccess() {
                println("成功将用户 $userId 设为管理员")
            }

            override fun onFailure(code: Int, desc: String) {
                println("设置管理员失败: $desc")
            }
        })
    }

    /// 撤销用户的管理员身份
    fun revokeAdmin(userId: String) {
        audienceStore.revokeAdministrator(userId, object :
CompletionHandler {
            override fun onSuccess() {
                println("成功撤销用户 $userId 的管理员身份")
            }

            override fun onFailure(code: Int, desc: String) {
                println("撤销管理员失败: $desc")
            }
        })
    }
}
```

功能进阶

在弹幕区展示观众入场欢迎语

当有新观众进入直播间时，弹幕/聊天区域会在本地自动显示一条欢迎消息，例如："欢迎 [用户昵称] 进入直播间"。

实现方式

我们通过添加 `LiveAudienceStore` 的观众加入事件监听器（ `LiveAudienceListener.onAudienceJoined` ）来获取新观众加入的实时通知。一旦事件触发，我们便提取出新观众的昵称信息，然后立即调用 `BarrageStore` 的本地插入接口 `appendLocalTip` 。

```
import io.trtc.tuikit.atomicxcore.api.live.LiveAudienceListener
import io.trtc.tuikit.atomicxcore.api.live.LiveAudienceStore
```

```
import io.trtc.tuikit.atomicxcore.api.live.LiveUserInfo
import io.trtc.tuikit.atomicxcore.api.barrage.Barrage
import io.trtc.tuikit.atomicxcore.api.barrage.BarrageStore
import io.trtc.tuikit.atomicxcore.api.barrage.BarrageType

class LiveRoomManager(
    private val liveId: String
) {

    init {
        // 初始化两个核心管理器
        setupWelcomeMessageFlow()
    }

    private fun setupWelcomeMessageFlow() {
        // 1. 获取 LiveAudienceStore 的实例
        val audienceStore = LiveAudienceStore.create(liveId)

        // 2. 获取 BarrageStore 的实例 (得益于内部机制，这会是同一个实例)
        val barrageStore = BarrageStore.create(liveId)

        // 3. 订阅观众事件
        audienceStore.addLiveAudienceListener(object :
LiveAudienceListener() {
            override fun onAudienceJoined(audience: LiveUserInfo) {
                // 4. 创建一条本地提示消息
                val welcomeTip = Barrage().apply {
                    liveID = liveId
                    messageType = BarrageType.TEXT
                    textContent = "欢迎 ${audience.userName} 进入直播间！"
                }

                // 5. 调用 BarrageStore 的接口将其插入弹幕列表
                barrageStore.appendLocalTip(welcomeTip)
            }
        })
    }
}
```

API 文档

关于 LiveAudienceStore 及其相关类的所有公开接口、属性和方法的详细信息，请参阅随 AtomicX Core 框架的官方 API 文档。本文档使用到的相关 Store 如下：

Store/Component	功能描述	API 文档
LiveCoreView	直播视频流展示与交互的核心视图组件：负责视频流渲染和视图挂件处理，支持主播直播、观众连麦、主播连线等场景。	API 文档
LiveAudienceStore	观众管理：获取实时观众列表（ID / 名称 / 头像），统计观众数量，监听观众进出事件。	API 文档
BarrageStore	弹幕功能：发送文本 / 自定义弹幕，维护弹幕列表，实时监听弹幕状态。	API 文档

常见问题

`LiveAudienceState` 中的在线人数（`audienceCount`）是如何更新的？时机和频率是怎样的？

`audienceCount` 的更新并非总是实时的，其机制如下：

- **主动进出房间:** 当用户主动加入或正常退出直播间时，在线人数的变更通知会即时触发。您会很快在 `LiveAudienceState` 中观察到 `audienceCount` 的变化。
- **异常掉线:** 当用户因网络问题、应用崩溃等原因异常掉线时，系统需要通过心跳机制来判断其真实状态。只有当该用户连续 90 秒没有心跳后，系统才会判定其为离线，并触发人数变更通知。
- **更新机制与频率控制:**
  - 无论是即时触发还是延迟触发，所有的人数变更通知都是以消息的形式在房间内广播的。
  - 房间内每秒的消息总量有上限，单房间消息频控是**每秒 40 条消息**。
  - **关键点: 在"弹幕风暴"或礼物刷屏等消息流量极高的场景下，如果房间内的消息速率超过了 40条/秒 的阈值，为了保证核心消息（例如弹幕）的送达，人数变更的消息可能会被频控策略丢弃。**

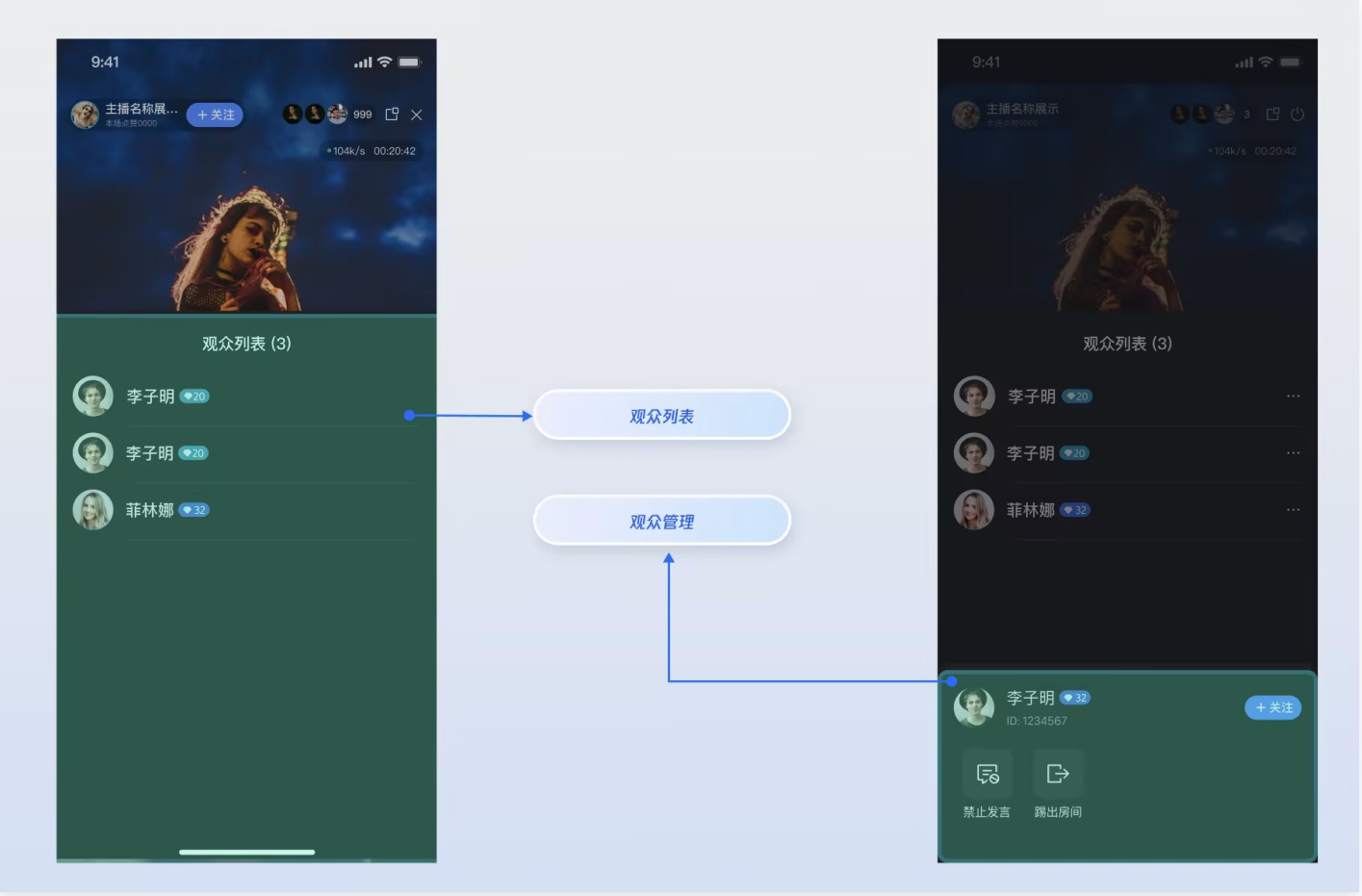
    这对开发者意味着什么?

    `audienceCount` 是一个非常接近实时的**高精度估算值**，但在极端高并发场景下，它可能存在短暂的延迟或数据丢失。因此，我们建议您将其用于**UI展示**，而不应作为计费、统计等需要绝对精准场景的唯一依据。

观众列表（iOS）

最近更新时间：2025-11-21 14:15:34

`LiveAudienceStore` 是 `AtomicXCore` 中专门负责管理直播间观众信息的模块。通过 `LiveAudienceStore`，您可以为您的直播应用构建一套完整的观众列表及管理系统。



核心功能

- **实时观众列表**：获取并展示当前在直播间内的所有观众信息。
- **观众人数统计**：实时获取当前直播间的准确观众总数。
- **观众动态监听**：通过事件订阅，实时感知观众的加入和离开。
- **管理员权限**：主播可以将普通观众设为管理员，或撤销其管理员身份。
- **房间管理**：主播或管理员可以将任意普通观众请出（踢出）直播间。

核心概念

在开始集成之前，我们先通过下表了解一下 LiveAudienceStore 相关的几个核心概念：

核心概念	类型	核心职责与描述

`LiveUserInfo`	struct	代表一个观众（用户）的基础信息模型。它包含了用户的唯一标识（`userID`）、昵称（`userName`）和头像地址（`avatarURL`）。
`LiveAudienceState`	struct	代表观众模块的当前状态。其核心属性 `audienceList` 是一个 `[LiveUserInfo]` 数组，存储了观众列表的快照；`audienceCount` 则代表当前观众总数。
`LiveAudienceEvent`	enum	代表观众动态的实时事件。分为 `.onAudienceJoined`（观众加入）和 `.onAudienceLeft`（观众离开）两种，用于实现对观众列表的增量更新。
`LiveAudienceStore`	class	这是与观众列表功能交互的核心管理类。通过它，您可以获取观众列表快照、执行管理操作，并通过订阅其 `liveAudienceEventPublisher` 来接收实时动态。

实现步骤

步骤1：组件集成

请参考 开始直播 集成 AtomicXCore，完成接入。

步骤2：初始化并获取观众列表

获取一个与当前直播间 liveId 绑定的 `LiveAudienceStore` 实例，并主动拉取一次当前的观众列表，用于首次展示。

```
import Foundation
import AtomicXCore
import Combine

class AudienceManager {
    private let liveId: String
    private let audienceStore: LiveAudienceStore
    private var cancellables = Set<AnyCancellable>()

    // 对外暴露【全量】观众列表的发布者，方便UI层订阅
    let audienceListPublisher = CurrentValueSubject<[LiveUserInfo],
Never>([])
    // 对外暴露观众总数的发布者
    let audienceCountPublisher = PassthroughSubject<UInt, Never>()

    init(liveId: String) {
```

```
        self.liveId = liveId
        // 1. 通过 liveId 获取 LiveAudienceStore 的实例
        self.audienceStore = LiveAudienceStore.create(liveID: liveId)

        // 2. 订阅状态和事件
        subscribeToAudienceState()
        subscribeToAudienceEvents()

        // 3. 主动拉取首屏数据
        fetchInitialAudienceList()
    }

    /// 主动获取一次观众列表快照
    func fetchInitialAudienceList() {
        audienceStore.fetchAudienceList { [weak self] result in
            switch result {
            case .success:
                print("首次拉取观众列表成功")
                // 成功后，数据会通过下面的 state 订阅通道自动更新
            case .failure(let error):
                print("首次拉取观众列表失败: \(error.localizedDescription)")
            }
        }
    }

    // ... 后续代码
}
```

步骤3：监听观众列表状态与实时动态

订阅 `audienceStore` 的 `state` 和 `event`，以接收全量列表快照和实时的观众进出事件，从而驱动 UI 更新。

```
extension AudienceManager {
    /// 订阅 state，用于获取观众总数和列表快照
    private func subscribeToAudienceState() {
        audienceStore.state
            .subscribe()
            .receive(on: DispatchQueue.main)
            .sink { [weak self] state in
```

```
                // state.audienceList 是一个全量快照
                self?.audienceListPublisher.send(state.audienceList)
                // state.audienceCount 是实时总数
                self?.audienceCountPublisher.send(state.audienceCount)
            }
            .store(in: &cancellables)
    }

    /// 订阅 event，用于处理观众的实时进出
    private func subscribeToAudienceEvents() {
        audienceStore.liveAudienceEventPublisher
            .receive(on: DispatchQueue.main)
            .sink { [weak self] event in
                guard let self = self else { return }

                var currentList = self.audienceListPublisher.value

                switch event {
                case .onAudienceJoined(let newAudience):
                    print("观众 \(newAudience.userName) 加入了直播间")
                    // 增量更新：在当前列表末尾添加新观众
                    if !currentList.contains(where: { $0.userID ==
newAudience.userID }) {
                        currentList.append(newAudience)
                        self.audienceListPublisher.send(currentList)
                    }

                case .onAudienceLeft(let departedAudience):
                    print("观众 \(departedAudience.userName) 离开了直播间")
                    // 增量更新：从当前列表中移除离开的观众
                    currentList.removeAll { $0.userID ==
departedAudience.userID }
                    self.audienceListPublisher.send(currentList)
                }
            }
            .store(in: &cancellables)
    }
}
```

步骤4：管理观众（踢人与设置管理员）

作为主播或管理员，对直播间内的观众进行管理操作。

4.1 将观众踢出直播间

调用 `kickUserOutOfRoom` 接口可以将指定用户请出直播间。

```
extension AudienceManager {
    func kick(userId: String) {
        audienceStore.kickUserOutOfRoom(userID: userId) { result in
            switch result {
            case .success:
                print("成功将用户 \(userId) 踢出房间")
                // 踢出成功后，您会收到 onAudienceLeft 事件
            case .failure(let error):
                print("踢出用户 \(userId) 失败: \
(error.localizedDescription)")
            }
        }
    }
}
```

4.2 设置/撤销管理员

通过 `setAdministrator` 和 `revokeAdministrator` 接口可以管理用户的管理员身份。

```
extension AudienceManager {
    /// 将用户设为管理员
    func promoteToAdmin(userId: String) {
        audienceStore.setAdministrator(userID: userId) { result in
            if case .success = result {
                print("成功将用户 \(userId) 设为管理员")
            }
        }
    }

    /// 撤销用户的管理员身份
    func revokeAdmin(userId: String) {
        audienceStore.revokeAdministrator(userID: userId) { result in
            if case .success = result {
```

```
                print("成功撤销用户 \(userId) 的管理员身份")
            }
        }
    }
}
```

功能进阶

在弹幕区展示观众入场欢迎语

当有新观众进入直播间时，弹幕/聊天区域会在本地自动显示一条欢迎消息，例如：“欢迎 [用户昵称] 进入直播间”。

实现方式

我们通过订阅 `LiveAudienceStore` 的观众加入事件（`LiveAudienceEvent.onAudienceJoined`）来获取新观众加入的实时通知。一旦事件触发，我们便提取出新观众的昵称信息，然后立即调用 `BarrageStore` 的本地插入接口 `appendLocalTip`。

```
import Foundation
import AtomicXCore
import Combine

class LiveRoomManager {
    private let liveId: String
    private let audienceManager: AudienceManager
    private let barrageManager: BarrageManager // 假设您已实现了弹幕文档中的
BarrageManager
    private var cancellables = Set<AnyCancellable>()

    init(liveId: String) {
        self.liveId = liveId
        // 初始化两个核心管理器
        self.audienceManager = AudienceManager(liveId: liveId)
        self.barrageManager = BarrageManager(liveId: liveId)

        // 设置联动
        setupWelcomeMessageFlow()
    }

    private func setupWelcomeMessageFlow() {
```

```
        // 1. 获取 LiveAudienceStore 的实例
        let audienceStore = LiveAudienceStore.create(liveID:
self.liveId)

        // 2. 获取 BarrageStore 的实例 （得益于内部机制，这会是同一个实例）
        let barrageStore = BarrageStore.create(liveID: self.liveId)

        // 3. 订阅观众事件
        audienceStore.liveAudienceEventPublisher
            .receive(on: DispatchQueue.main)
            .sink { event in
                // 4. 只处理观众加入事件
                guard case .onAudienceJoined(let newAudience) = event
else {
                    return
                }

                // 5. 创建一条本地提示消息
                var welcomeTip = Barrage()
                welcomeTip.messageType = .text
                welcomeTip.textContent = "欢迎 \(newAudience.userName) 进
入直播间！"

                // 6. 调用 BarrageStore 的接口将其插入弹幕列表
                barrageStore.appendLocalTip(message: welcomeTip)
            }
            .store(in: &cancellables)
    }
}
```

API 文档

关于 LiveAudienceStore 及其相关类的所有公开接口、属性和方法的详细信息，请参阅随 AtomicXCore 框架的官方 API 文档。本文档使用到的相关 Store 如下：

Store/Component	功能描述	API 文档
LiveCoreView	直播视频流展示与交互的核心视图组件：负责视频流渲染和视图挂件处理，支持主播直播、观众连麦、主播连线等场景。	API 文档

LiveAudienceStore	观众管理：获取实时观众列表（ID / 名称 / 头像），统计观众数量，监听观众进出事件。	API 文档
BarrageStore	弹幕功能：发送文本 / 自定义弹幕，维护弹幕列表，实时监听弹幕状态。	API 文档

常见问题

`LiveAudienceState` 中的在线人数（`audienceCount`）是如何更新的？时机和频率是怎样的？

`audienceCount` 的更新并非总是实时的，其机制如下：

- **主动进出房间**：当用户主动加入或正常退出直播间时，在线人数的变更通知会即时触发。您会很快在 `LiveAudienceState` 中观察到 `audienceCount` 的变化。
- **异常掉线**：当用户因网络问题、应用崩溃等原因异常掉线时，系统需要通过心跳机制来判断其真实状态。只有当该用户连续 90 秒没有心跳后，系统才会判定其为离线，并触发人数变更通知。
- **更新机制与频率控制**：
  - 无论是即时触发还是延迟触发，所有的人数变更通知都是以消息的形式在房间内广播的。
  - 房间内每秒的消息总量有上限，单房间消息频控是**每秒 40 条消息**。
  - **关键点**：在“弹幕风暴”或礼物刷屏等消息流量极高的场景下，如果房间内的消息速率超过了 40条/秒 的阈值，为了保证核心消息（例如弹幕）的送达，**人数变更的消息可能会被频控策略丢弃**。

这对开发者意味着什么？

`audienceCount` 是一个非常接近实时的**高精度估算值**，但在极端高并发场景下，它可能存在短暂的延迟或数据丢失。因此，我们建议您将其用于 **UI 展示**，而不应作为计费、统计等需要绝对精准场景的唯一依据。

观众连线
观众连线（Android）

最近更新时间：2025-11-21 14:15:34

AtomicXCore 提供了 CoGuestStore 模块，专门用于管理观众连麦的完整业务流程。您无需关心复杂的状态同步和信令交互，只需调用几个简单的方法，即可为您的直播添加强大的观众与主播音视频互动功能。

核心场景

`CoGuestStore` 支持以下两种最主流的连麦场景：

- **观众申请上麦**：观众主动发起连麦请求，主播在收到请求后进行同意或拒绝。
- **主播邀请上麦**：主播可以主动向直播间内的任意一位观众发起连麦邀请。

实现步骤

步骤1：组件集成

请参考 开始直播 集成 AtomicXCore，并完成 LiveCoreView 的接入。

步骤2：实现观众申请上麦

观众端实现

作为观众，您的核心任务是**发起申请、接收结果**和**主动下麦**。

1. **发起连麦申请**

   当用户点击界面上的"申请连麦"按钮时，调用 `applyForSeat` 方法。

```
import io.trtc.tuikit.atomicxcore.api.live.CoGuestStore
import io.trtc.tuikit.atomicxcore.api.CompletionHandler

val liveId = "房间ID"
val guestStore = CoGuestStore.create(liveId)

// 用户点击"申请连麦"
fun requestToConnect() {
    // timeout: 请求超时时间，例如 30 秒
    guestStore.applyForSeat(
        seatIndex = 0, // 麦位索引
        timeout = 30,
        extraInfo = null,
```

```
        completion = object : CompletionHandler {
            override fun onSuccess() {
                println("连麦申请已发送，等待主播处理...")
            }

            override fun onFailure(code: Int, desc: String) {
                println("申请发送失败: $desc")
            }
        }
    )
}
```

2. 监听主播的响应

通过添加 `GuestListener`，您可以接收到主播的处理结果。

```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import io.trtc.tuikit.atomicxcore.api.device.DeviceStore
import io.trtc.tuikit.atomicxcore.api.live.CoGuestStore
import io.trtc.tuikit.atomicxcore.api.live.GuestListener
import io.trtc.tuikit.atomicxcore.api.live.LiveUserInfo

// 在您的Activity或Fragment中实现监听
class YourActivity : AppCompatActivity() {
    val liveId = "房间ID"
    val guestStore = CoGuestStore.create(liveId)
    val deviceStore = DeviceStore.shared()
    private val guestListener = object : GuestListener() {
        override fun onGuestApplicationResponded(isAccept: Boolean,
hostUser: LiveUserInfo) {
            if (isAccept) {
                println("主播 ${hostUser.userName} 同意了你的申请，准备上
麦")
                // 上麦申请被同意，进行上麦后的操作
                // 1. 打开摄像头、麦克风
                DeviceStore.shared().openLocalCamera(true, completion
= null)
                DeviceStore.shared().openLocalMicrophone(completion =
null)
```

```
                // 2. 在此更新 UI，例如关闭申请按钮，显示连麦中的状态
            } else {
                println("主播 ${hostUser.userName} 拒绝了你的申请")
                // 上麦申请被拒绝，弹窗提示
            }
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // 添加监听器
        guestStore.addGuestListener(guestListener)
    }

    override fun onDestroy() {
        super.onDestroy()
        // 移除监听器
        guestStore.removeGuestListener(guestListener)
    }
}
```

3. 主动下麦

当连麦观众想结束互动时，调用 `disconnect` 方法即可返回普通观众状态。

```
// 用户点击"下麦"按钮
fun leaveSeat() {
    guestStore.disconnect(object : CompletionHandler {
        override fun onSuccess() {
            println("已成功下麦")
        }

        override fun onFailure(code: Int, desc: String) {
            println("下麦失败: $desc")
        }
    })
}
```

4. (可选) 取消申请

如果观众在主播处理前想撤回申请，可以调用 `cancelApplication` 。

```
// 用户在等待时，点击"取消申请"
fun cancelRequest() {
    guestStore.cancelApplication(object : CompletionHandler {
        override fun onSuccess() {
            println("申请已取消")
        }

        override fun onFailure(code: Int, desc: String) {
            println("申请取消失败: $desc")
        }
    })
}
```

主播端实现

作为主播，您的核心任务是**接收申请、展示申请列表**和**处理申请**。

1. 监听新的连麦申请

通过添加 `HostListener`，您可以在有新观众申请时立即收到通知，并给出提示。

```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import io.trtc.tuikit.atomicxcore.api.live.CoGuestStore
import io.trtc.tuikit.atomicxcore.api.live.HostListener
import io.trtc.tuikit.atomicxcore.api.live.LiveUserInfo

class YourActivity : AppCompatActivity() {
    val liveId = "房间ID"
    val guestStore = CoGuestStore.create(liveId)
    val hostListener = object : HostListener() {
        override fun onGuestApplicationReceived(guestUser:
LiveUserInfo) {
            println("收到观众 ${guestUser.userName} 的连麦申请")
            // 在此更新 UI，例如在"申请列表"按钮上显示红点
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // 添加监听器
```

```
        guestStore.addHostListener(hostListener)
    }

    override fun onDestroy() {
        super.onDestroy()
        // 移除监听器
        guestStore.removeHostListener(hostListener)
    }
}
```

2. 展示申请列表

`CoGuestStore` 的 state 会实时维护当前的申请者列表，您可以订阅它来刷新您的 UI。

```
class YourActivity : AppCompatActivity() {
    val liveId = "房间ID"
    val guestStore = CoGuestStore.create(liveId)

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // 订阅申请者列表变化
        lifecycleScope.launch {
            guestStore.coGuestState.applicants.collect { applicants ->
                println("当前申请人数: ${applicants.size}")
                // 在此刷新您的"申请者列表"UI
                // updateApplicantListView(applicants)
            }
        }
    }
}
```

3. 处理连麦申请

当您在列表中选择一位观众并点击"同意"或"拒绝"时，调用相应的方法。

```
// 主播点击"同意"按钮，传入申请者的 userID
fun accept(userId: String) {
    guestStore.acceptApplication(userId, object : CompletionHandler {
        override fun onSuccess() {
            println("已同意 $userId 的申请，对方正在上麦")
```

```
        }

        override fun onFailure(code: Int, desc: String) {
            println("同意申请失败: $desc")
        }
    })
}

// 主播点击"拒绝"按钮
fun reject(userId: String) {
    guestStore.rejectApplication(userId, object : CompletionHandler {
        override fun onSuccess() {
            println("已拒绝 $userId 的申请")
        }

        override fun onFailure(code: Int, desc: String) {
            println("拒绝申请失败: $desc")
        }
    })
}
```

步骤3：实现主播邀请上麦

主播端实现

1. 向观众发起邀请

当主播在观众列表中选择某人并点击"邀请连麦"时，调用 `inviteToSeat` 方法。

```
// 主播选择观众并发起邀请
fun invite(userId: String) {
    // timeout: 邀请超时时间
    guestStore.inviteToSeat(
        inviteeID = userId,
        seatIndex = 0,
        timeout = 30,
        extraInfo = null,
        completion = object : CompletionHandler {
            override fun onSuccess() {
                println("已向 $userId 发出邀请，等待对方回应...")
            }
```

```
            override fun onFailure(code: Int, desc: String) {
                println("发送邀请失败: $desc")
            }
        }
    )
}
```

2. **监听观众的回应**

通过 `HostListener` 监听 `onHostInvitationResponded` 事件。

```
// 在 HostListener 的实现中添加
override fun onHostInvitationResponded(isAccept: Boolean, guestUser:
LiveUserInfo) {
    if (isAccept) {
        println("观众 ${guestUser.userName} 接受了你的邀请")
    } else {
        println("观众 ${guestUser.userName} 拒绝了你的邀请")
    }
}
```

观众端实现

1. **接收主播的邀请**

通过 `GuestListener` 监听 `onHostInvitationReceived` 事件。

```
// 在 GuestListener 的实现中添加
override fun onHostInvitationReceived(hostUser: LiveUserInfo) {
    println("收到主播 ${hostUser.userName} 的连麦邀请")
    // 在此弹出一个对话框，让用户选择"接受"或"拒绝"
    // showInvitationDialog(hostUser)
}
```

2. **响应邀请**

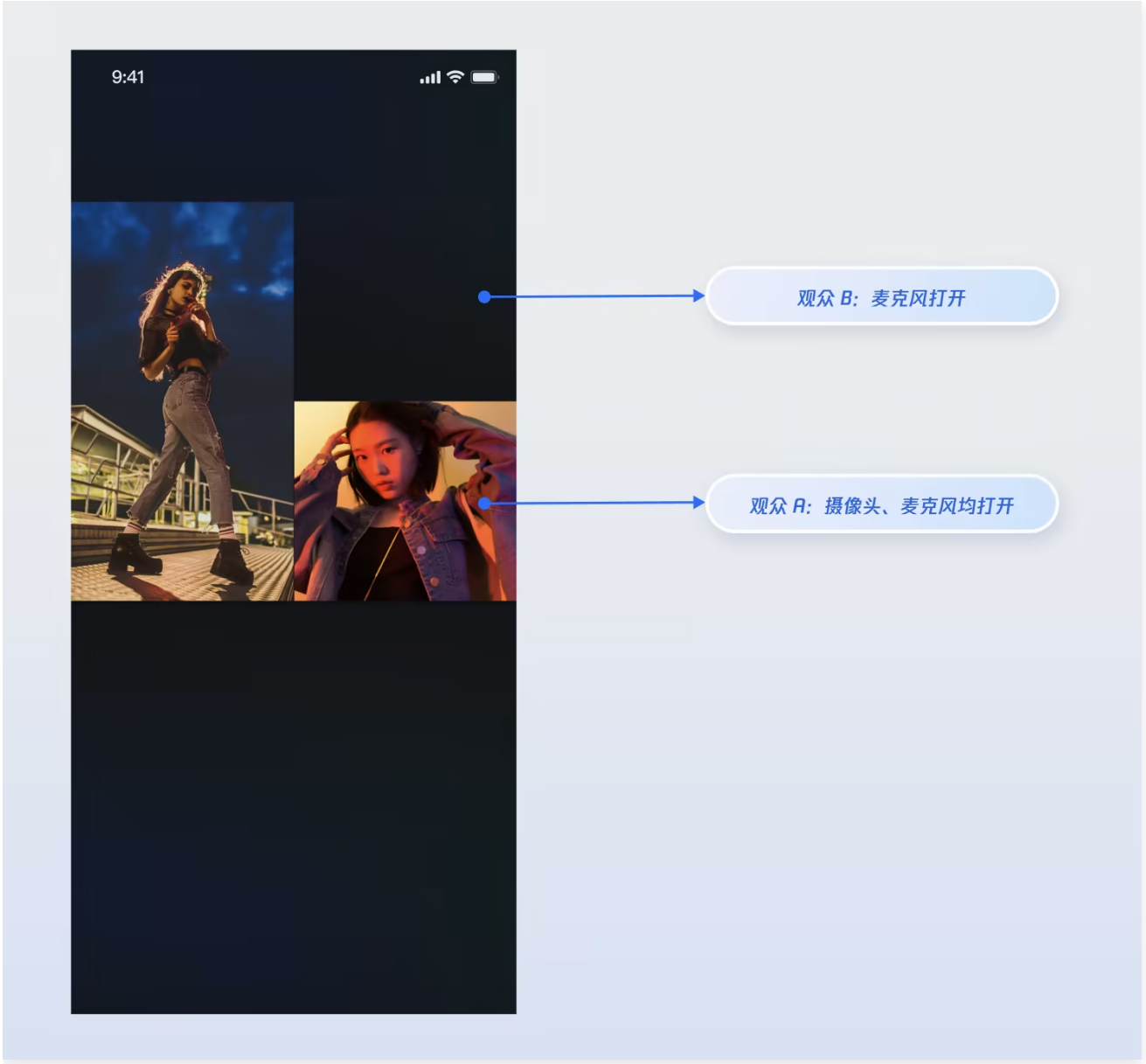
当用户在弹出的对话框中做出选择后，调用相应的方法。

```
val inviterId = "发起邀请的主播ID" // 从 onHostInvitationReceived 事件中获
取
```

```
// 用户点击"接受"
fun accept() {
    guestStore.acceptInvitation(inviterId, object : CompletionHandler
{
        override fun onSuccess() {
            // 2. 打开摄像头、麦克风
            DeviceStore.shared().openLocalCamera(true, completion =
null)
            DeviceStore.shared().openLocalMicrophone(completion =
null)
        }

        override fun onFailure(code: Int, desc: String) {
            println("接受邀请失败: $desc")
        }
    })
}

// 用户点击"拒绝"
fun reject() {
    guestStore.rejectInvitation(inviterId, object : CompletionHandler
{
        override fun onSuccess() {
            println("已拒绝邀请")
        }

        override fun onFailure(code: Int, desc: String) {
            println("拒绝邀请失败: $desc")
        }
    })
}
```

运行效果

当您集成以上功能实现后，请分别使用两个观众与主播进行连麦操作，观众 A 同时打开摄像头和麦克风，观众 B 只打开麦克风，运行效果如下，您可以参考下一章节 完善 UI 细节 来定制您想要的 UI 逻辑。



完善 UI 细节

您可以通过 `VideoViewAdapter` 接口提供的"插槽"能力，在观众连麦的视频流画面上添加自定义视图，用于显示昵称、头像等信息，或在他们关闭摄像头时提供占位图，以优化视觉体验。

实现视频流画面的昵称显示

实现效果

实现方式

> **提示：**
> 您也可以参考 TUILiveKit 开源项目中的 widgets 目录下文件来了解完整的实现逻辑。

- **步骤 1：创建前景视图（CustomSeatView），该视图用于在视频流上方显示用户信息。**

```
import android.content.Context
import android.graphics.Color
import android.view.Gravity
import android.view.ViewGroup
import android.widget.LinearLayout
import android.widget.TextView

// 自定义的用户信息悬浮视图（前景）
class CustomSeatView(context: Context) : LinearLayout(context) {
    private val nameLabel: TextView

    init {
        orientation = VERTICAL
        gravity = Gravity.BOTTOM or Gravity.START
        setBackgroundColor(Color.parseColor("#80000000")) // 半透明黑色
背景
```

```
        nameLabel = TextView(context).apply {
            setTextColor(Color.WHITE)
            textSize = 14f
        }

        addView(nameLabel, LayoutParams(
            ViewGroup.LayoutParams.WRAP_CONTENT,
            ViewGroup.LayoutParams.WRAP_CONTENT
        ).apply {
            setMargins(20, 0, 0, 20) // 左边距20，下边距20
        })
    }

    fun setUserName(userName: String) {
        nameLabel.text = userName
    }
}
```

- **步骤 2：创建背景视图（CustomAvatarView），该视图用于在用户无视频流时作为占位图显示。**

```
import android.content.Context
import android.graphics.Color
import android.view.Gravity
import android.view.ViewGroup
import android.widget.ImageView
import android.widget.LinearLayout

// 自定义的头像占位视图（背景）
class CustomAvatarView(context: Context) : LinearLayout(context) {
    private val avatarImageView: ImageView

    init {
        orientation = VERTICAL
        gravity = Gravity.CENTER
        setBackgroundColor(Color.TRANSPARENT)

        avatarImageView = ImageView(context).apply {
            setColorFilter(Color.GRAY)
```

```
            scaleType = ImageView.ScaleType.CENTER_CROP
        }

        addView(avatarImageView, LayoutParams(120, 120)) // 60dp * 2 =
120px
    }

    fun setUserAvatar(avatarUrl: String) {
        // 在这里加载用户头像，可以使用Glide等图片加载库
        // Glide.with(context).load(avatarUrl).into(avatarImageView)
    }
}
```

- **步骤 3：实现 `VideoViewAdapter.createCoGuestView` 接口，根据 `viewLayer` 的值返回对应的视图。**

```
import android.os.Bundle
import android.view.View
import androidx.appcompat.app.AppCompatActivity
import com.tencent.cloud.tuikit.engine.room.TUIRoomDefine
import io.trtc.tuikit.atomicxcore.api.view.VideoViewAdapter
import io.trtc.tuikit.atomicxcore.api.view.ViewLayer

// 1. 在您的Activity中，实现 VideoViewAdapter 接口
class YourActivity : AppCompatActivity(), VideoViewAdapter {

    // ... 其他代码 ...

    // 2. 完整实现接口方法，处理两种 viewLayer
    override fun createCoGuestView(userInfo:
TUIRoomDefine.SeatFullInfo?, viewLayer: ViewLayer?): View? {
        userInfo ?: return null
        val userId = userInfo.userId
        if (userId.isNullOrEmpty()) return null

        return when (viewLayer) {
            ViewLayer.FOREGROUND -> {
                // 用户摄像头开启时，显示前景视图
                val seatView = CustomSeatView(this)
```

```
                seatView.setUserName(userInfo.userName ?: "")
                seatView
            }
            ViewLayer.BACKGROUND -> {
                // 用户摄像头关闭时，显示背景视图
                val avatarView = CustomAvatarView(this)
                // 您可以在这里通过 userInfo.userAvatar 加载用户真实头像
                userInfo.userAvatar?.let {
avatarView.setUserAvatar(it) }
                avatarView
            }
            else -> null
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // 设置适配器
        liveCoreView.setVideoViewAdapter(this)
    }
}
```

参数说明：

参数	类型	说明
`seatInfo`	`SeatFullInfo?`	麦位信息对象，包含麦上用户的详细信息
`seatInfo.userId`	`String`	麦上用户的 ID
`seatInfo.userName`	`String`	麦上用户的昵称
`seatInfo.userAvatar`	`String`	麦上用户的头像 URL
`seatInfo.userMicrophoneStatus`	`DeviceStatus`	麦上用户的麦克风状态
`seatInfo.userCameraStatus`	`DeviceStatus`	麦上用户的摄像头状态
`viewLayer`	`ViewLayer`	视图层级枚举

		• `FOREGROUND` 表示前景挂件视图，始终显示在视频画面的最上层 • `BACKGROUND` 表示背景挂件视图，位于前景视图下层，仅在对应用户没有视频流（例如未开摄像头）的情况下显示，通常用于展示用户的默认头像或占位图

API 文档

关于 CoGuestStore 及其相关类的所有公开接口、属性和方法的详细信息，请参阅随 AtomicXCore 框架的官方 API 文档。本指南使用到的相关 Store 如下：

Store/Component	功能描述	API 文档
LiveCoreView	直播视频流展示与交互的核心视图组件：负责视频流渲染和视图挂件处理，支持主播直播、观众连麦、主播连线等场景。	API 文档
DeviceStore	音视频设备控制：麦克风（开关 / 音量）、摄像头（开关 / 切换 / 画质）、屏幕共享，设备状态实时监听。	API 文档
CoGuestStore	观众连麦管理：连麦申请 / 邀请 / 同意 / 拒绝，连麦成员权限控制（麦克风 / 摄像头），状态同步。	API 文档

常见问题

如何管理通过 `VideoViewAdapter` 添加的自定义视图的生命周期和事件?

LiveCoreView 会自动管理您通过适配器方法返回的视图的添加和移除，您无需手动处理。如果需要在自定义视图中处理用户交互（例如点击事件），请在创建视图时为其添加相应的事件即可。

`VideoViewAdapter` 中的 viewLayer 参数有什么作用?

`viewLayer` 用于区分前景和背景挂件：

- `FOREGROUND`：前景层，始终显示在视频画面的最上层。
- `BACKGROUND`：背景层，仅在对应用户没有视频流（例如未开摄像头）的情况下显示，通常用于展示用户的默认头像或占位图。

我的自定义视图为什么没有显示?

- **检查适配器设置**：请确认已调用 `liveCoreView.setVideoViewAdapter(this)` 并成功设置了适配器。
- **检查实现方法**：请检查是否正确实现了对应的适配器方法（例如 `createCoGuestView`）。

- **检查返回值**：确保您的适配器方法在正确的时机返回了一个有效的 `View` 实例，而不是 `null`。您可以在适配器方法中添加日志进行调试。

观众连线（iOS）

最近更新时间：2025-11-21 14:15:34

AtomicXCore 提供了 CoGuestStore 模块，专门用于管理观众连麦的完整业务流程。您无需关心复杂的状态同步和信令交互，只需调用几个简单的方法，即可为您的直播添加强大的观众与主播音视频互动功能。

核心场景

`CoGuestStore` 支持以下两种最主流的连麦场景：

- **观众申请上麦**：观众主动发起连麦请求，主播在收到请求后进行同意或拒绝。
- **主播邀请上麦**：主播可以主动向直播间内的任意一位观众发起连麦邀请。

实现步骤

步骤1：组件集成

请参考 开始直播 集成 AtomicXCore，并完成 LiveCoreView 的接入。

步骤2：实现观众申请上麦

观众端实现

作为观众，您的核心任务是**发起申请、接收结果**和**主动下麦**。

1. **发起连麦申请**

   当用户点击界面上的“申请连麦”按钮时，调用 `applyForSeat` 方法。

```
import AtomicXCore

let liveId = "房间ID"
let guestStore = CoGuestStore.create(liveID: liveId)

// 用户点击“申请连麦”
func requestToConnect() {
    // timeout: 请求超时时间，例如 30 秒
    guestStore.applyForSeat(timeout: 30.0, extraInfo: nil) { result in
      switch result {
      case .success():
          print("连麦申请已发送，等待主播处理...")
      case .failure(let error):
          print("申请发送失败: \(error.message)")
      }
```

```
    }
}
```

2. 监听主播的响应

通过订阅 `guestEventPublisher`，您可以接收到主播的处理结果。

```
// 在您的视图控制器初始化时订阅事件
func subscribeGuestEvents() {
    guestStore.guestEventPublisher
      .sink { [weak self] event in
          if case let .onGuestApplicationResponded(isAccept, hostUser)
= event {
              if isAccept {
                  print("主播 \(hostUser.userName) 同意了你的申请，准备上
麦")
                  // 1. 打开摄像头、麦克风
                  DeviceStore.shared.openLocalCamera(isFront: true,
completion: nil)
                  DeviceStore.shared.openLocalMicrophone(completion:
nil)
                  // 2. 在此更新 UI，例如关闭申请按钮，显示连麦中的状态
              } else {
                  print("主播 \(hostUser.userName) 拒绝了你的申请")
                  // 弹窗提示用户申请被拒绝
              }
          }
      }
      .store(in: &cancellables) // 管理订阅生命周期
}
```

3. 主动下麦

当连麦观众想结束互动时，调用 `disConnect` 方法即可返回普通观众状态。

```
// 用户点击“下麦”按钮
func leaveSeat() {
    guestStore.disConnect { result in
      switch result {
      case .success():
          print("已成功下麦")
```

```
        case .failure(let error):
            print("下麦失败: \(error.message)")
        }
    }
}
```

4. **(可选) 取消申请**

   如果观众在主播处理前想撤回申请，可以调用 `cancelApplication` 。

```
// 用户在等待时，点击“取消申请”
func cancelRequest() {
    guestStore.cancelApplication { result in
        switch result {
        case .success():
            print("申请已取消")
        case .failure(let error):
            print("申请取消失败: \(error.message)")
        }
    }
}
```

主播端实现

作为主播，您的核心任务是**接收申请、展示申请列表**和**处理申请**。

1. **监听新的连麦申请**

   通过订阅 `hostEventPublisher` ，您可以在有新观众申请时立即收到通知，并给出提示。

```
import AtomicXCore

let liveId = "房间ID"
let guestStore = CoGuestStore.create(liveID: liveId)

// 订阅主播事件
guestStore.hostEventPublisher
    .sink { [weak self] event in
        if case let .onGuestApplicationReceived(guestUser) = event {
            print("收到观众 \(guestUser.userName) 的连麦申请")
            // 在此更新 UI，例如在“申请列表”按钮上显示红点
        }
```

```
    }
    .store(in: &cancellables)
```

2. 展示申请列表

`CoGuestStore` 的 state 会实时维护当前的申请者列表，您可以订阅它来刷新您的 UI。

```
// 订阅状态变更
guestStore.state
    .subscribe(StatePublisherSelector(keyPath:
\CoGuestState.applicants)) // 只关心申请者列表的变化
    .removeDuplicates()
    .sink { applicants in
        print("当前申请人数：\(applicants.count)")
        // 在此刷新您的“申请者列表”UI
        // self.applicantListView.update(with: applicants)
    }
    .store(in: &cancellables)
```

3. 处理连麦申请

当您在列表中选择一位观众并点击“同意”或“拒绝”时，调用相应的方法。

```
// 主播点击“同意”按钮，传入申请者的 userID
func accept(userId: String) {
    guestStore.acceptApplication(userID: userId) { result in
        if case .success = result {
            print("已同意 \(userId) 的申请，对方正在上麦")
        }
    }
}

// 主播点击“拒绝”按钮
func reject(userId: String) {
    guestStore.rejectApplication(userID: userId) { result in
        if case .success = result {
            print("已拒绝 \(userId) 的申请")
        }
    }
}
```

步骤3：实现主播邀请上麦

主播端实现

1. **向观众发起邀请**

   当主播在观众列表中选择某人并点击“邀请连麦”时，调用 `inviteToSeat` 方法。

```
 // 主播选择观众并发起邀请
func invite(userId: String) {
    // timeout: 邀请超时时间
    guestStore.inviteToSeat(userID: userId, timeout: 30.0, extraInfo:
nil) { result in
        if case .success = result {
            print("已向 \(userId) 发出邀请，等待对方回应...")
        }
    }
}
```

2. **监听观众的回应**

   通过 `hostEventPublisher` 监听 `onHostInvitationResponded` 事件。

```
// 在 hostEventPublisher 的订阅中添加
if case let .onHostInvitationResponded(isAccept, guestUser) = event {
    if isAccept {
        print("观众 \(guestUser.userName) 接受了你的邀请")
    } else {
        print("观众 \(guestUser.userName) 拒绝了你的邀请")
    }
}
```

观众端实现

1. **接收主播的邀请**

   通过 `guestEventPublisher` 监听 `onHostInvitationReceived` 事件。

```
// 在 guestEventPublisher 的订阅中添加
if case let .onHostInvitationReceived(hostUser) = event {
    print("收到主播 \(hostUser.userName) 的连麦邀请")
    // 在此弹出一个对话框，让用户选择“接受”或“拒绝”
    // self.showInvitationDialog(from: hostUser)
```

```
}
```

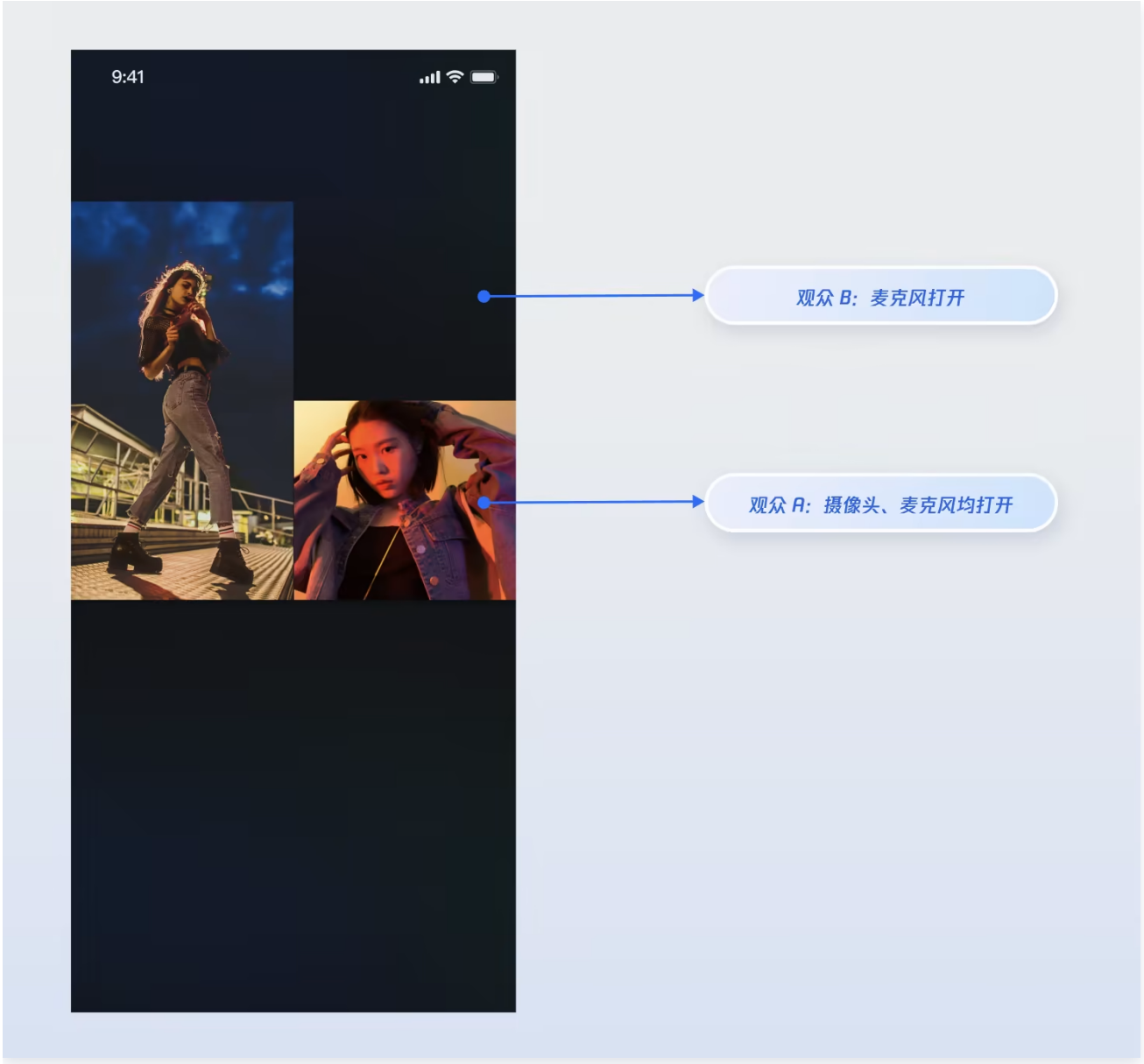
2. **响应邀请**

当用户在弹出的对话框中做出选择后，调用相应的方法。

```
let inviterId = "发起邀请的主播ID" // 从 onHostInvitationReceived 事件中获
取

// 用户点击“接受”
func accept() {
    guestStore.acceptInvitation(inviterID: inviterId) { result in
        if case .success = result {
            // 2. 打开摄像头、麦克风
            DeviceStore.shared.openLocalCamera(isFront: true,
completion: nil)
            DeviceStore.shared.openLocalMicrophone(completion: nil)
        }
    }
}

// 用户点击“拒绝”
func reject() {
    guestStore.rejectInvitation(inviterID: inviterId) { result in
        // ...
    }
}
```

运行效果

当您集成以上功能实现后，请分别使用两个观众与主播进行连麦操作，观众 A 同时打开摄像头和麦克风，观众 B 只打开麦克风，运行效果如下，您可以参考下一章节 完善 UI 细节 来定制您想要的 UI 逻辑。



完善 UI 细节

您可以通过 `LiveCoreView.VideoViewDelegate` 协议提供的“插槽”能力，在观众连麦的视频流画面上添加自定义视图，用于显示昵称、头像等信息，或在他们关闭摄像头时提供占位图，以优化视觉体验。

实现视频流画面的昵称显示

实现效果

实现方式

> **提示：**
> 您也可以参考 TUILiveKit 开源项目中的 AnchorCoGuestView.swift 和 AnchorEmptySeatView.swift 文件来了解完整的实现逻辑。

- **步骤 1：创建前景视图（CustomSeatView），该视图用于在视频流上方显示用户信息。**

```
import UIKit

// 自定义的用户信息悬浮视图（前景）
class CustomSeatView: UIView {
    lazy var nameLabel: UILabel = {
        let label = UILabel()
        label.textColor = .white
        label.font = .systemFont(ofSize: 14)
        return label
    }()

    override init(frame: CGRect) {
        super.init(frame: frame)
        backgroundColor = UIColor.black.withAlphaComponent(0.5)
        addSubview(nameLabel)
```

```
        nameLabel.snp.makeConstraints { make in
            make.bottom.equalToSuperview().offset(-5)
            make.leading.equalToSuperview().offset(5)
        }
    }
}
```

- **步骤 2：创建背景视图（CustomAvatarView），该视图用于在用户无视频流时作为占位图显示。**

```
import UIKit

// 自定义的头像占位视图（背景）
class CustomAvatarView: UIView {
    lazy var avatarImageView: UIImageView = {
        let imageView = UIImageView()
        imageView.tintColor = .gray
        return imageView
    }()

    override init(frame: CGRect) {
        super.init(frame: frame)
        backgroundColor = .clear
        layer.cornerRadius = 30
        addSubview(avatarImageView)
        avatarImageView.snp.makeConstraints { make in
            make.center.equalToSuperview()
            make.width.height.equalTo(60)
        }
    }
}
```

- **步骤 3：实现 `VideoViewDelegate.createCoGuestView` 协议，根据 `viewLayer` 的值返回对应的视图。**

```
import AtomicXCore

// 1. 在您的视图控制器中，遵守 VideoViewDelegate 协议
class YourViewController: UIViewController, VideoViewDelegate {
```

```
    // ... 其他代码 ...

    // 2. 完整实现协议方法，处理两种 viewLayer
    func createCoGuestView(seatInfo: TUISeatFullInfo, viewLayer:
ViewLayer) -> UIView? {
        guard let userId = seatInfo.userID,!userId.isEmpty else {
            return nil
        }
        if viewLayer == .foreground {
            // 用户摄像头开启时，显示前景视图
            let seatView = CustomSeatView()
            seatView.nameLabel.text = seatInfo.userName
            return seatView
        } else { // viewLayer == .background
            // 用户摄像头关闭时，显示背景视图
            let avatarView = CustomAvatarView()
            // 您可以在这里通过 seatInfo.userAvatar 加载用户真实头像
            return avatarView
        }
    }
}
```

参数说明：

参数	类型	说明
`seatInfo`	`TUISeatFullInfo`	麦位信息对象，包含麦上用户的详细信息。
`seatInfo.userID`	`String?`	麦上用户的 ID。
`seatInfo.userName`	`String?`	麦上用户的昵称。
`seatInfo.userAvatar`	`String?`	麦上用户的头像 URL。
`seatInfo.userMicrophoneStatus`	`TUIDeviceStatus`	麦上用户的麦克风状态。
`seatInfo.userCameraStatus`	`TUIDeviceStatus`	麦上用户的摄像头状态。

`viewLayer`	`ViewLayer`	视图层级枚举。 `.foreground` 表示前景挂件视图，始终显示在视频画面的最上层。 `.background` 表示背景挂件视图，位于前景视图下层，仅在对应用户没有视频流（例如未开摄像头）的情况下显示，通常用于展示用户的默认头像或占位图。

API 文档

关于 CoGuestStore 及其相关类的所有公开接口、属性和方法的详细信息，请参阅 AtomicXCore 框架的官方 API 文档。本指南使用到的相关 Store 如下：

Store/Component	功能描述	API 文档
LiveCoreView	直播视频流展示与交互的核心视图组件：负责视频流渲染和视图挂件处理，支持主播直播、观众连麦、主播连线等场景。	API 文档
DeviceStore	音视频设备控制：麦克风（开关 / 音量）、摄像头（开关 / 切换 / 画质）、屏幕共享，设备状态实时监听。	API 文档
CoGuestStore	观众连麦管理：连麦申请 / 邀请 / 同意 / 拒绝，连麦成员权限控制（麦克风 / 摄像头），状态同步。	API 文档

常见问题

如何管理通过 `VideoViewDelegate` 添加的自定义视图的生命周期和事件？

LiveCoreView 会自动管理您通过代理方法返回的视图的添加和移除，您无需手动处理。如果需要在自定义视图中处理用户交互（例如点击事件），请在创建视图时为其添加相应的事件即可。

`VideoViewDelegate` 中的 viewLayer 参数有什么作用？

`viewLayer` 用于区分前景和背景挂件：

- `.foreground`：前景层，始终显示在视频画面的最上层。
- `.background`：背景层，仅在对应用户没有视频流（例如未开摄像头）的情况下显示，通常用于展示用户的默认头像或占位图。

我的自定义视图为什么没有显示？

- **检查代理设置**：请确认已调用 `coreView.videoViewDelegate = self` 并成功设置了代理。
- **检查实现方法**：请检查是否正确实现了对应的代理方法（例如 `createCoGuestView`）。

- **检查返回值**：确保您的代理方法在正确的时机返回了一个有效的 `UIView` 实例，而不是 `nil` 。您可以在代理方法中添加断点或日志进行调试。

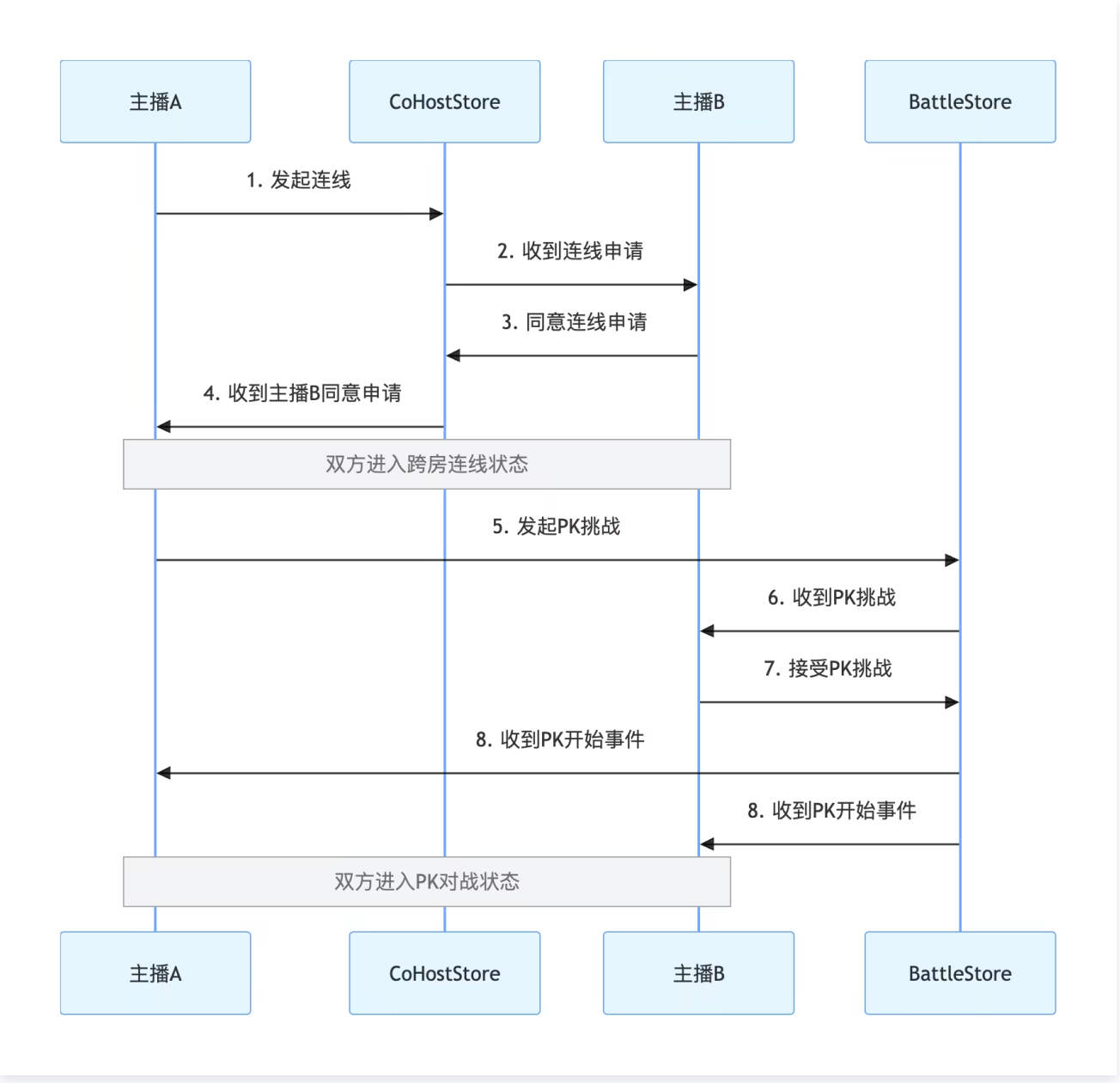
主播连线和 PK
主播连线和 PK（Android）

最近更新时间：2025-11-21 14:15:34

AtomicXCore 提供了 CoHostStore 和 BattleStore 两个核心模块，分别用于处理跨房连线和 PK 对战。本文档将指导您如何组合使用这两个工具，来完成直播场景下连线到 PK 的完整流程。

核心场景

一次完整的“主播连线 PK”通常包含三个核心阶段，其整体流程如下：



实现步骤

步骤1：组件集成

请参考 开始直播 集成 AtomicXCore，并完成 LiveCoreView 的接入。

步骤2：实现跨房连线

此步骤的目标是让两个主播的画面出现在同一个视图中，我们将使用 CoHostStore 来完成。

邀请方（主播 A）实现

1. **发起连线邀请**

   当主播A在界面上选择目标主播B并发起连线时，调用 `requestHostConnection` 方法。

```
import androidx.appcompat.app.AppCompatActivity
import io.trtc.tuikit.atomicxcore.api.CompletionHandler
import io.trtc.tuikit.atomicxcore.api.live.CoHostLayoutTemplate
import io.trtc.tuikit.atomicxcore.api.live.CoHostStore

// 主播A的Activity
class AnchorAActivity : AppCompatActivity() {
    private val liveId = "主播A的房间ID"
    private val coHostStore = CoHostStore.create(liveId)

    // 用户点击"连线"按钮，并选择了主播B
    fun inviteHostB(targetHostLiveId: String) {
        val layout = CoHostLayoutTemplate.HOST_DYNAMIC_GRID // 选择一个
布局模板
        val timeout = 30 // 邀请超时时间（秒）

        coHostStore.requestHostConnection(
            targetHostLiveID = targetHostLiveId,
            layoutTemplate = layout,
            timeout = timeout,
            extraInfo = null,
            completion = object : CompletionHandler {
                override fun onSuccess() {
                    println("连线邀请已发送，等待对方处理...")
                }

                override fun onFailure(code: Int, desc: String) {
                    println("邀请发送失败: $desc")
```

```
                }
            }
        )
    }
}
```

2. 监听邀请结果

通过 `CoHostListener` 的回调方法，您可以接收到主播 B 的处理结果。

```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import io.trtc.tuikit.atomicxcore.api.live.CoHostListener
import io.trtc.tuikit.atomicxcore.api.live.CoHostStore
import io.trtc.tuikit.atomicxcore.api.live.SeatUserInfo

class AnchorAActivity : AppCompatActivity() {
    private val liveId = "主播A的房间ID"
    private val coHostStore = CoHostStore.create(liveId)
    private val coHostListener = object : CoHostListener() {
        override fun onCoHostRequestAccepted(invitee: SeatUserInfo) {
            println("主播 ${invitee.userName} 同意了你的连线邀请")
        }

        override fun onCoHostRequestRejected(invitee: SeatUserInfo) {
            println("主播 ${invitee.userName} 拒绝了你的邀请")
        }

        override fun onCoHostRequestTimeout(inviter: SeatUserInfo,
invitee: SeatUserInfo) {
            println("邀请超时，对方未回应")
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_anchor_a)

        // 初始化CoHostStore
        coHostStore.addCoHostListener(coHostListener)
```

```
    }
}
```

受邀方（主播 B）实现

1. 接收连线邀请

通过 `CoHostListener`，主播B可以监听到来自主播A的邀请。

```
import androidx.appcompat.app.AppCompatActivity
import io.trtc.tuikit.atomicxcore.api.live.CoHostListener
import io.trtc.tuikit.atomicxcore.api.live.CoHostStore
import io.trtc.tuikit.atomicxcore.api.live.SeatUserInfo

// 主播B的Activity
class AnchorBActivity : AppCompatActivity() {
    private val liveId = "主播B的房间ID"
    private val coHostStore = CoHostStore.create(liveId)

    private val coHostListener = object : CoHostListener() {
        override fun onCoHostRequestReceived(inviter: SeatUserInfo,
extensionInfo: String) {
            println("收到主播 ${inviter.userName} 的连线邀请")
            // 显示邀请对话框，响应联线邀请
            // showInvitationDialog(inviter)
        }
    }
}
```

2. 响应连线邀请

当主播B在弹出的对话框中做出选择后，调用相应的方法。

```
// AnchorBActivity 的一部分
fun acceptInvitation(fromHostLiveId: String) {
    coHostStore.acceptHostConnection(fromHostLiveID = fromHostLiveId,
completion = null)
}

fun rejectInvitation(fromHostLiveId: String) {
```

```
    coHostStore.rejectHostConnection(fromHostLiveID = fromHostLiveId,
completion = null)
}
```

步骤3：实现主播 PK

连线成功后，任意一方都可以发起 PK，此步骤我们将使用 BattleStore 来实现主播 PK。

挑战方（例如主播 A）实现

1. **发起 PK 挑战**

   当主播 A 点击 PK 按钮时，调用 `requestBattle` 方法。

```
// AnchorAActivity 的一部分
class AnchorAActivity : AppCompatActivity() {
    private val liveId = "主播A的房间ID"
    private val battleStore = BattleStore.create(liveId)

    fun startPK(opponentUserId: String) {
        val config = BattleConfig(duration = 300) // PK持续5分钟
        battleStore.requestBattle(
            config = config,
            userIDList = listOf(opponentUserId),
            timeout = 30,
            completion = null
        )
    }
}
```

2. **监听 PK 状态**

   通过 `BattleListener` 监听 PK 的开始、结束等关键事件。

```
 // AnchorAActivity 的一部分
class AnchorAActivity : AppCompatActivity() {
    private val liveId = "主播A的房间ID"
    private val battleStore = BattleStore.create(liveId)

    private val battleListener = object : BattleListener() {
        override fun onBattleStarted(
            battleInfo: BattleInfo,
```

```
            inviter: SeatUserInfo,
            invitees: List<SeatUserInfo>
        ) {
            println("PK 开始")
        }

        override fun onBattleEnded(battleInfo: BattleInfo, reason:
BattleEndedReason?) {
            println("PK 结束")
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // ... 其他初始化代码 ...
        battleStore.addBattleListener(battleListener)
    }
}
```

应战方（主播 B）实现

1. 接收 PK 挑战

通过 `BattleListener` 监听到 PK 邀请。

```
// 在 AnchorBActivity 中添加
class AnchorBActivity : AppCompatActivity() {
    private val liveId = "主播B的房间ID"
    private val battleStore = BattleStore.create(liveId)

    private val battleListener = object : BattleListener() {
        override fun onBattleRequestReceived(battleID: String,
inviter: SeatUserInfo, invitee: SeatUserInfo) {
            println("收到主播 ${inviter.userName} 的PK挑战")
            // 弹出对话框，让主播B选择"接受"或"拒绝"
            // showPKChallengeDialog(battleID, inviter)
            }
        }

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
```

```
            // ... 其他初始化代码 ...
            battleStore.addBattleListener(battleListener)
        }
    }
```

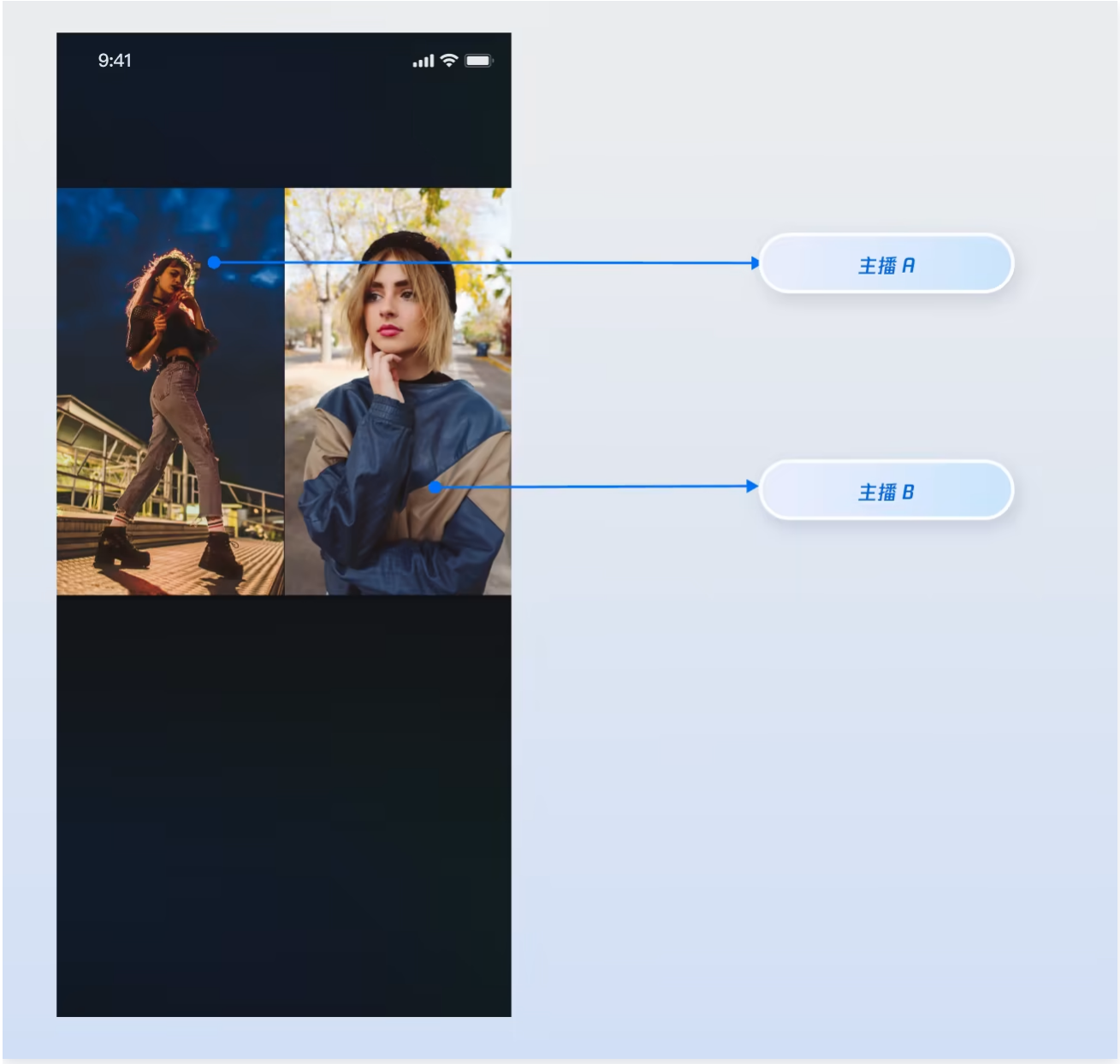
2. **响应 PK 挑战**

当主播 B 做出选择后，调用相应的方法。

```
// AnchorBActivity 的一部分
// 用户点击"接受挑战"
fun acceptPK(battleId: String) {
    battleStore.acceptBattle(battleID = battleId, completion = null)
}

// 用户点击"拒绝挑战"
fun rejectPK(battleId: String) {
    battleStore.rejectBattle(battleID = battleId, completion = null)
}
```

运行效果

当您集成以上功能实现后，请分别使用主播 A 和主播 B 进行对应操作，运行效果如下，您可以参考下一章节 完善 UI 细节 来定制您想要的 UI 逻辑。

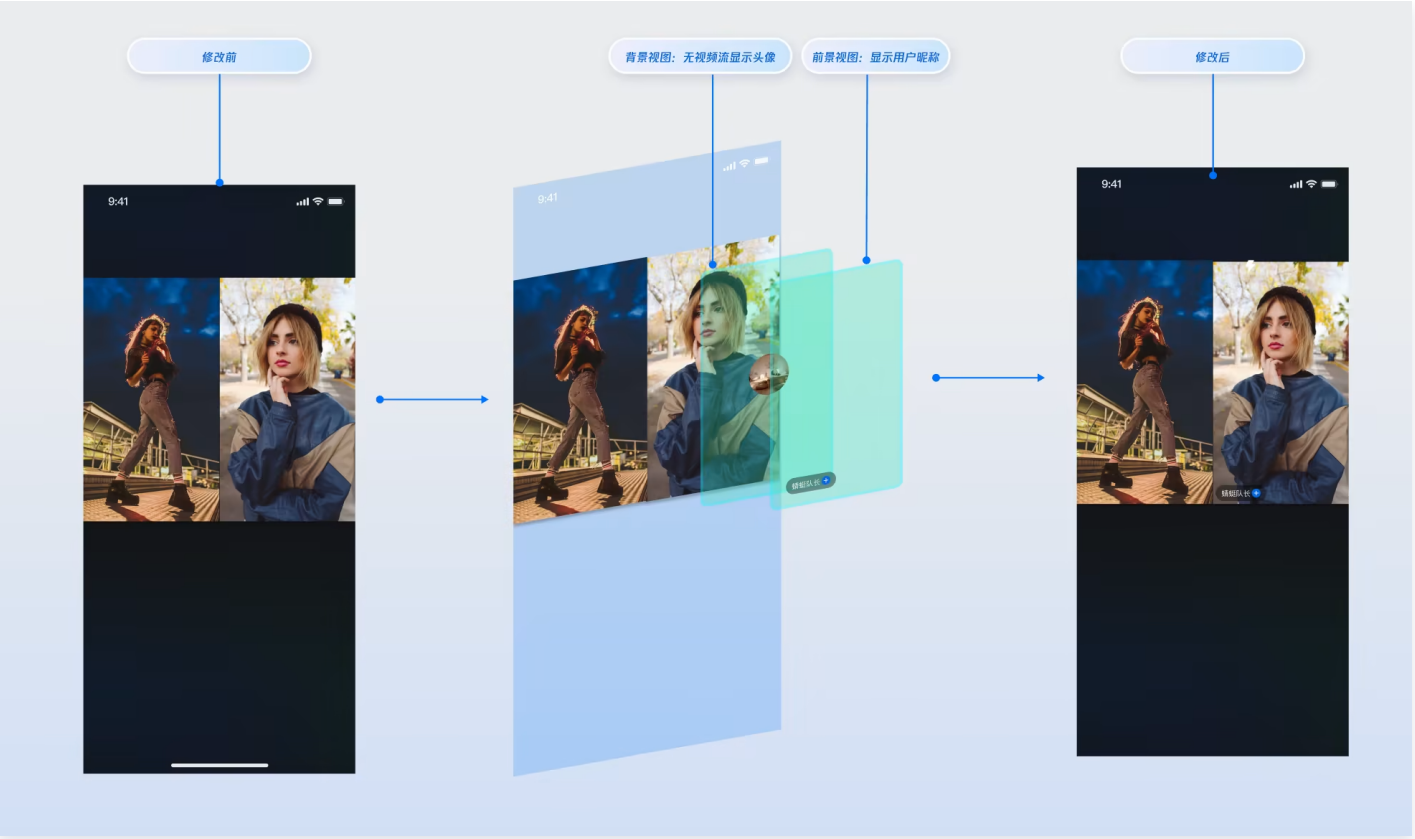


完善 UI 细节

您可以通过 `VideoViewAdapter` 接口提供的"插槽"能力，在视频流画面上添加自定义视图，用于显示昵称、头像、PK 进度条等信息，或在他们关闭摄像头时提供占位图，以优化视觉体验。

实现视频流画面的昵称显示

实现效果



实现方式

- **步骤1：创建前景视图（CustomSeatView），该视图用于在视频流上方显示用户信息。**

> **提示：**
> 您也可以参考 TUILiveKit 开源项目中的 widgets 目录下文件来了解完整的实现逻辑。

```
import android.content.Context
import android.graphics.Color
import android.view.Gravity
import android.widget.LinearLayout
import android.widget.TextView

// 自定义的用户信息悬浮视图（前景）
class CustomSeatView(context: Context) : LinearLayout(context) {
    private val nameLabel: TextView

    init {
        orientation = VERTICAL
        setBackgroundColor(Color.parseColor("#80000000")) // 半透明黑色
背景
```

```
        nameLabel = TextView(context).apply {
            setTextColor(Color.WHITE)
            textSize = 14f
            gravity = Gravity.CENTER
        }

        addView(nameLabel)
        val layoutParams = nameLabel.layoutParams as LayoutParams
        layoutParams.setMargins(5, 0, 5, 5)
    }

    fun setUserName(userName: String) {
        nameLabel.text = userName
    }
}
```

- **步骤2：创建背景视图（CustomAvatarView），该视图用于在用户无视频流时作为占位图显示。**

```
import android.content.Context
import android.graphics.Color
import android.view.Gravity
import android.widget.ImageView
import android.widget.LinearLayout

// 自定义的头像占位视图（背景）
class CustomAvatarView(context: Context) : LinearLayout(context) {
    private val avatarImageView: ImageView

    init {
        orientation = VERTICAL
        gravity = Gravity.CENTER
        setBackgroundColor(Color.TRANSPARENT)

        avatarImageView = ImageView(context).apply {
            setColorFilter(Color.GRAY)
            scaleType = ImageView.ScaleType.CENTER_CROP
        }
```

```
        addView(avatarImageView)
        val layoutParams = avatarImageView.layoutParams as
LayoutParams
        layoutParams.width = 120 // 60dp
        layoutParams.height = 120 // 60dp
    }
}
```

- **步骤3：实现 `VideoViewAdapter.createCoHostView` 协议，根据 `viewLayer` 的值返回对应的视图。**

```
import com.tencent.cloud.tuikit.engine.room.TUIRoomDefine
import android.view.View
import androidx.appcompat.app.AppCompatActivity
import io.trtc.tuikit.atomicxcore.api.view.VideoViewAdapter
import io.trtc.tuikit.atomicxcore.api.view.ViewLayer

// 1. 在您的Activity中，实现 VideoViewAdapter 接口
class YourActivity : AppCompatActivity(), VideoViewAdapter {

    // ... 其他代码 ...

    // 2. 完整实现接口方法，处理两种 viewLayer
    override fun createCoHostView(coHostUser:
TUIRoomDefine.SeatFullInfo?, viewLayer: ViewLayer?): View? {
        val seatInfo = coHostUser ?: return null
        val userId = seatInfo.userId
        if (userId.isNullOrEmpty()) {
            return null
        }

        return when (viewLayer) {
            ViewLayer.FOREGROUND -> {
                val seatView = CustomSeatView(this)
                seatView.setUserName(seatInfo.userName ?: "")
                seatView
            }
            ViewLayer.BACKGROUND -> {
                val avatarView = CustomAvatarView(this)
                // 您可以在这里通过 seatInfo.userAvatar 加载用户真实头像
```

```
                    avatarView
                }
                null -> null
            }
        }
    }
```

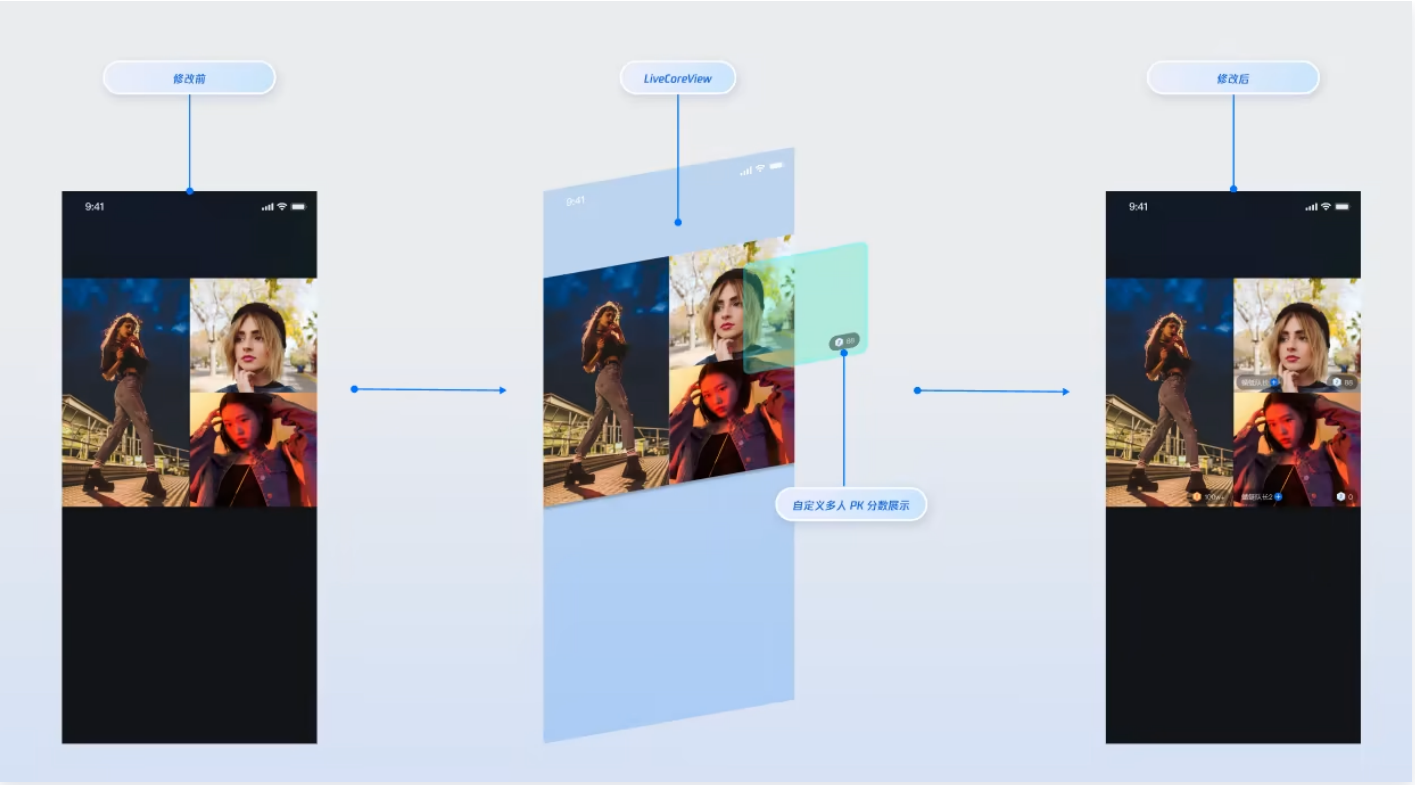
参数说明：

参数	类型	说明
`seatInfo`	`SeatFullInfo?`	麦位信息对象，包含麦上用户的详细信息
`seatInfo.userId`	`String`	麦上用户的 ID
`seatInfo.userName`	`String`	麦上用户的昵称
`seatInfo.userAvatar`	`String`	麦上用户的头像 URL
`seatInfo.userMicrophoneStatus`	`DeviceStatus`	麦上用户的麦克风状态
`seatInfo.userCameraStatus`	`DeviceStatus`	麦上用户的摄像头状态
`viewLayer`	`ViewLayer`	视图层级枚举 • `FOREGROUND` 表示前景挂件视图，始终显示在视频画面的最上层 • `BACKGROUND` 表示背景挂件视图，位于前景视图下层，仅在对应用户没有视频流（例如未开摄像头）的情况下显示，通常用于展示用户的默认头像或占位图

实现 PK 用户视图的分数展示

当主播开始 PK 后，可以在对方主播的视频画面上挂载自定义视图，通常用于展示该主播收到的礼物价值或其它 PK 相关信息。

实现效果



实现方式

- **步骤1：创建自定义 PK 用户视图，您可以参考 TUILiveKit 开源项目中的 BattleMemberInfoView.java 文件来了解完整的实现逻辑。**

```
import android.content.Context
import android.graphics.Color
import android.view.Gravity
import android.widget.LinearLayout
import android.widget.TextView
import
com.tencent.cloud.tuikit.engine.extension.TUILiveBattleManager.BattleU
ser
import io.trtc.tuikit.atomicxcore.api.live.BattleStore
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

// 自定义PK 用户视图
class CustomBattleUserView(
    context: Context,
    private val liveId: String,
    private val battleUser: BattleUser
```

```
) : LinearLayout(context) {

    private lateinit var scoreView: LinearLayout
    private lateinit var scoreLabel: TextView
    private val battleStore: BattleStore

    init {
        orientation = VERTICAL
        gravity = Gravity.BOTTOM or Gravity.END
        setBackgroundColor(Color.TRANSPARENT)
        isClickable = false

        battleStore = BattleStore.create(liveId)

        // UI 布局
        setupUI()
        // 订阅分数变化
        subscribeBattleState()
    }

    private fun setupUI() {
        scoreView = LinearLayout(context).apply {
            setBackgroundColor(Color.parseColor("#66000000"))
            orientation = VERTICAL
            gravity = Gravity.CENTER
        }

        scoreLabel = TextView(context).apply {
            setTextColor(Color.WHITE)
            textSize = 14f
            gravity = Gravity.CENTER
        }

        scoreView.addView(scoreLabel)
        addView(scoreView)

        val layoutParams = scoreView.layoutParams as LayoutParams
        layoutParams.width = LayoutParams.WRAP_CONTENT
        layoutParams.height = 48 // 24dp
        layoutParams.setMargins(0, 0, 10, 10)
```

```
    }

    // 订阅 PK 分数变化
    private fun subscribeBattleState() {
        CoroutineScope(Dispatchers.Main).launch {
            battleStore.battleState.battleScore.collect { battleScore
->
                val score = battleScore[battleUser.userId] ?: 0
                // 更新 UI
                scoreLabel.text = score.toString()
            }
        }
    }
}
```

- **步骤2：实现 `VideoViewAdapter.createBattleView` 协议**

```
  // 1. 让您的Activity实现 VideoViewAdapter 接口
class YourActivity : AppCompatActivity(), VideoViewAdapter {

    override fun createBattleView(battleUser: BattleUser?): View? {
        battleUser ?: return null
        // CustomBattleUserView 是您自定义的PK用户信息视图
        return CustomBattleUserView(this, liveId, battleUser)
    }

}
```

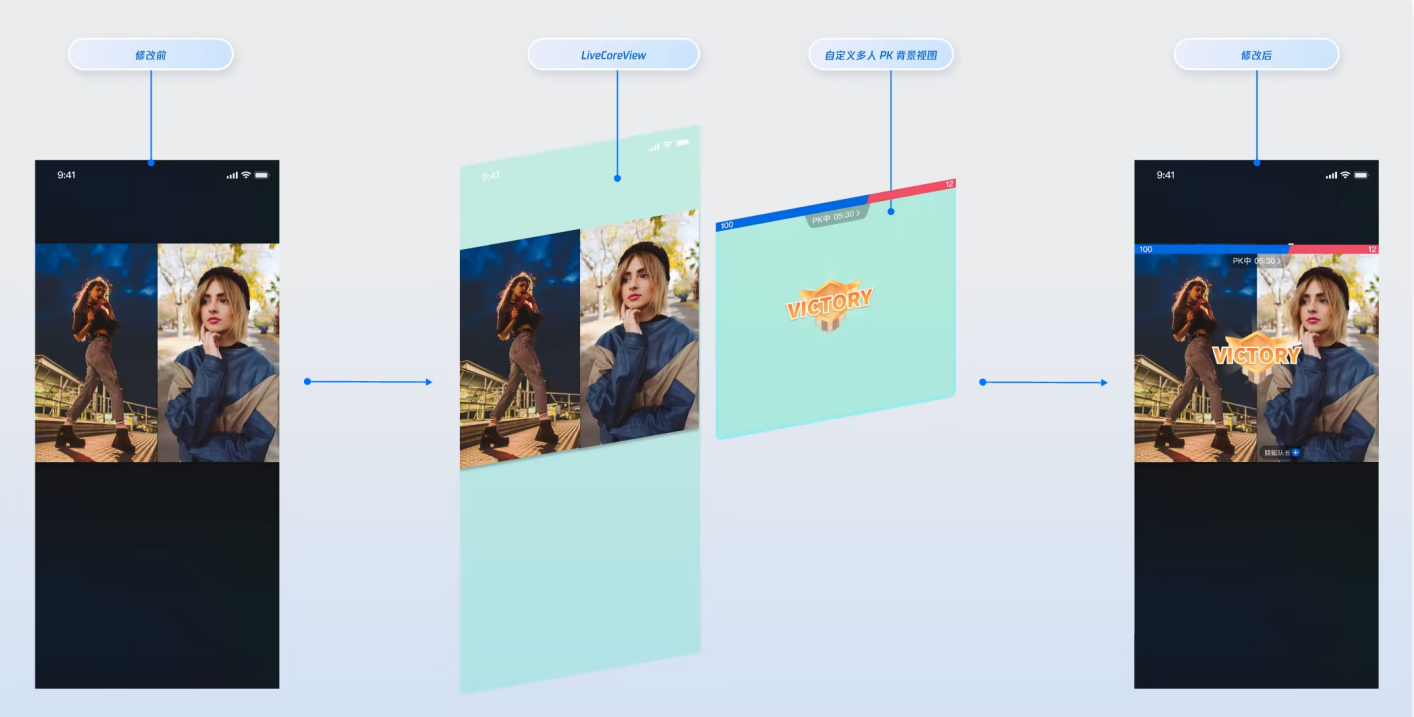
参数说明：

参数	类型	说明
`battleUser`	`BattleUser?`	PK 用户信息对象
`battleUser.roomId`	`String`	PK 的房间 ID
`battleUser.userId`	`String`	PK 用户 ID
`battleUser.userName`	`String`	PK 用户昵称

`battleUser.avatarUrl`	`String`	PK 用户头像地址
`battleUser.score`	`int`	PK 分数

实现视频流画面上的 PK 状态显示

实现效果



实现方式

- **步骤1：** 创建自定义 PK 全局视图 **CustomBattleContainerView**，您可以参考 TUILiveKit 开源项目中的 BattleInfoView.java 文件来实现，即可实现同样的效果。
- **步骤2：** 实现 `VideoViewAdapter.createBattleContainerView` 协议。

```
// 让您的Activity实现 VideoViewAdapter 接口并设置适配器
class YourActivity : AppCompatActivity(), VideoViewAdapter {
    override fun createBattleContainerView(): View? {
        return CustomBattleContainerView(this)
    }
}
```

功能进阶

通过 REST API 实现 PK 分数更新

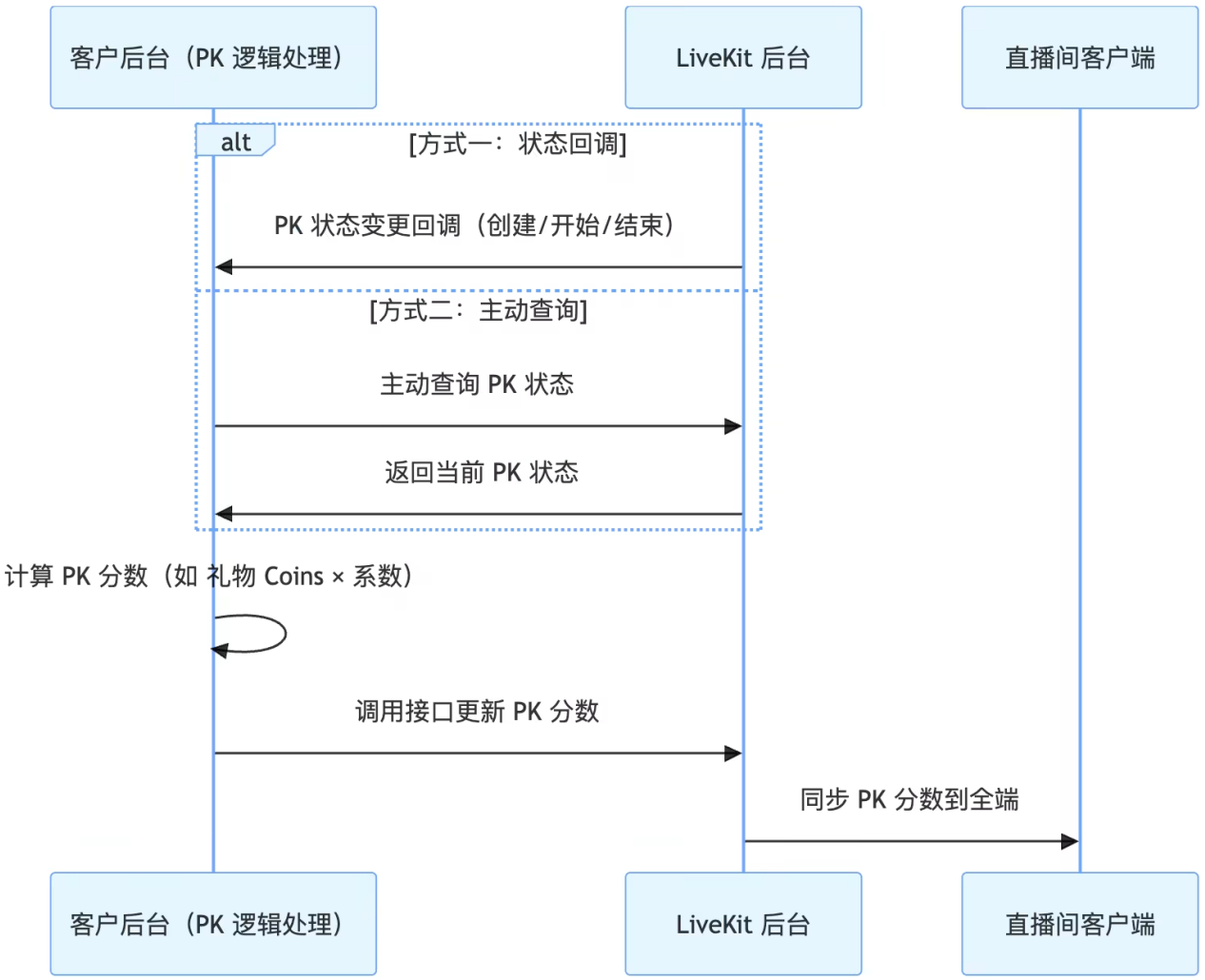
通常在**直播主播 PK 场景**下，会将主播收到的礼物价值与 PK 数值挂钩（例如：观众送 "火箭" 礼物，主播 PK 分数增加 500 分），您可以通过我们的 REST API，轻松实现直播 PK 场景下的分数实时更新。

重要说明：

LiveKit 后台的 PK 分数系统采用纯数值计算和累加机制，所以您需要根据自身的运营策略和业务需求，调用更新接口前完成 PK 分数的计算，您可以参考如下的 PK 分数计算示例：

礼物类型	分数计算规则	示例
基础礼物	礼物价值 × 5	10元礼物 → 50分
中级礼物	礼物价值 × 8	50元礼物 → 400分
高级礼物	礼物价值 × 12	100元礼物 → 1200分
特效礼物	固定高分数	520元礼物 → 1314分

REST API 调用流程



关键流程说明

1. **获取 PK 状态：**
   - **回调配置：您可以通过配置 PK 状态回调，由 LiveKit 后台在 PK 开始、结束时，主动通知您的系统 PK 状态。**
   - **主动查询：您的后台服务可主动调用 PK 状态查询 接口，随时查询当前 PK 状态。**
2. **PK分数计算：您的后台服务根据业务规则（如上述示例），计算 PK 分数增量。**
3. **PK分数更新：您的后台服务调用 修改 PK 分数 接口，向 LiveKit 后台更新 PK 分数。**
4. **LiveKit后台 同步客户端：LiveKit 后台自动将更新后的 PK 分数同步到所有客户端。**

涉及的 REST API 接口

接口	功能描述	请求示例

主动接口 – 查询 PK 状态	可根据此接口查询当前房间是否在 PK	请求示例
主动接口 – 修改 PK 分数	将计算后的 PK 数值通过此接口更新	请求示例
回调配置 – PK 开始时回调	客户后台可以通过该回调及时知晓 PK 开启	回调示例
回调配置 – PK 结束时回调	客户后台可以通过该回调及时知晓 PK 结束	回调示例

API 文档

关于 CoHostStore 及其相关类的所有公开接口、属性和方法的详细信息，请参阅随 AtomicXCore 框架的官方 API 文档。本指南使用到的相关 Store 如下：

Store/Component	功能描述	API 文档
LiveCoreView	直播视频流展示与交互的核心视图组件：负责视频流渲染和视图挂件处理，支持主播直播、观众连麦、主播连线等场景。	API 文档
DeviceStore	音视频设备控制：麦克风（开关 / 音量）、摄像头（开关 / 切换 / 画质）、屏幕共享，设备状态实时监听。	API 文档
CoHostStore	主播跨房连线：支持多布局模板（动态网格等），发起 / 接受 / 拒绝连线，连麦主播互动管理。	API 文档
BattleStore	主播 PK 对战：发起 PK（配置时长 / 对手），管理 PK 状态（开始 / 结束），同步分数，监听对战结果。	API 文档

常见问题

为什么发起了连线邀请，对方却没收到？

- 请检查 targetHostLiveId 是否正确，并且对方直播间处于正常开播状态。
- 检查网络连接是否通畅，邀请信令有30秒的默认超时时间。

连线或PK过程中，一方主播网络断开或App崩溃了怎么办？

CoHostStore 和 BattleStore 内部都有心跳和超时检测机制。如果一方异常退出，另一方会通过 onCoHostUserLeft 或 onUserExitBattle 等事件收到通知，您可以根据这些事件来处理UI，例如提示"对方已掉线"并结束互动。

为什么 PK 分数只能通过 REST API 更新？

因为 REST API 能同时满足 PK 分数的**安全性、实时性、扩展性**需求：

- **防篡改保公平**：需鉴权 + 数据校验，每笔更新可追溯来源（例如礼物行为），杜绝手动改分、刷分，保障竞技公平；
- **多端实时同步**：用标准化格式（例如 JSON）快速对接礼物、PK、展示系统，确保主播 / 观众 / 后台分数实时一致，无延迟；
- **灵活适配规则**：后端改配置（例如调整礼物对应分数、加成分数）即可适配业务变化，无需改前端，降低迭代成本。

如何管理通过 `VideoViewAdapter` 添加的自定义视图的生命周期和事件？

LiveCoreView 会自动管理您通过适配器方法返回视图的添加和移除，您无需手动处理。如果需要在自定义视图中处理用户交互（例如点击事件），请在创建视图时为其添加相应的事件即可。

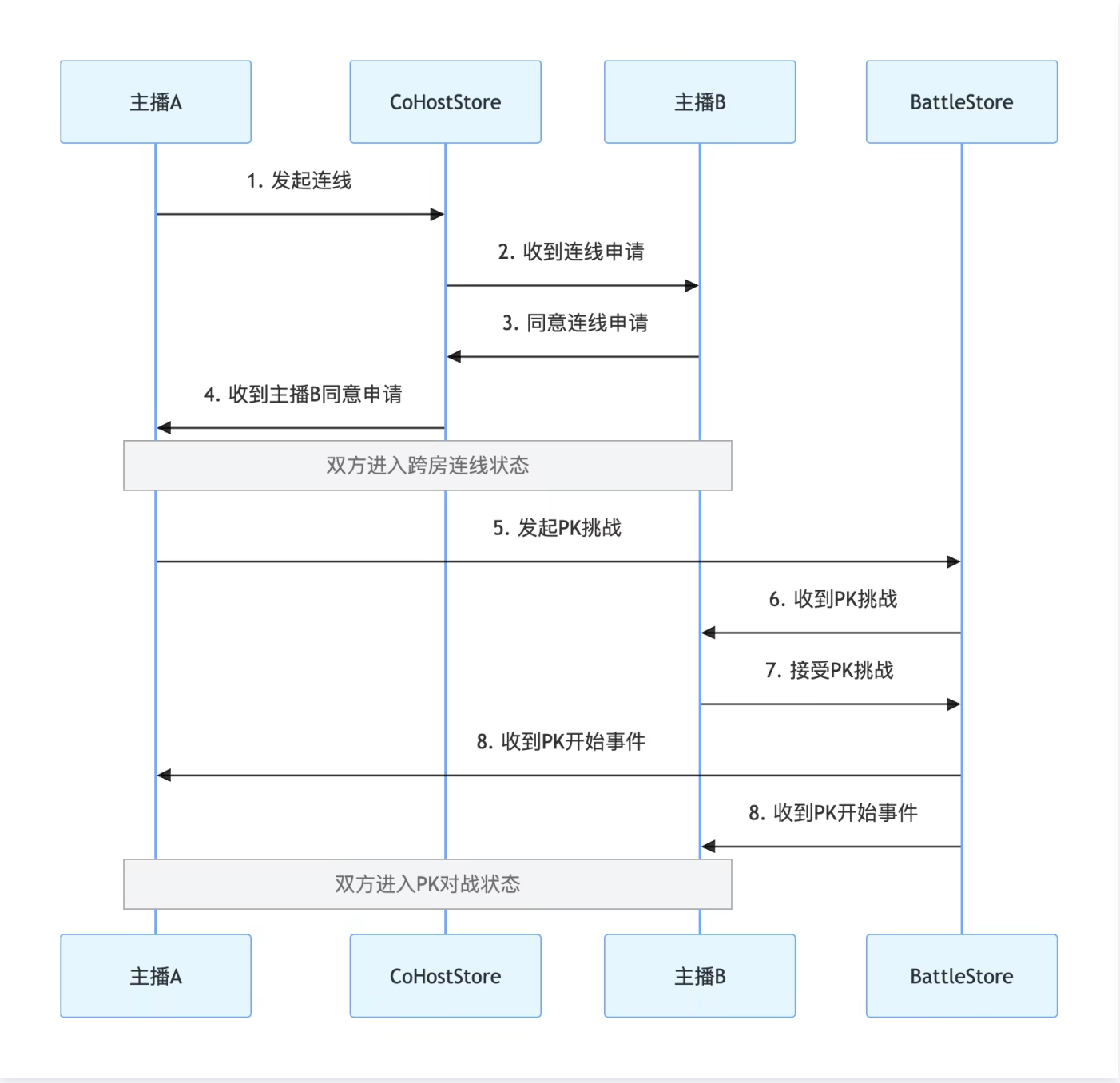
主播连线和 PK（iOS）

最近更新时间：2025-11-21 14:15:34

AtomicXCore 提供了 CoHostStore 和 BattleStore 两个核心模块，分别用于处理跨房连线和 PK 对战。本文档将指导您如何组合使用这两个工具，来完成直播场景下连线到 PK 的完整流程。

核心场景

一次完整的“主播连线 PK”通常包含三个核心阶段，其整体流程如下：



实现步骤

步骤1：组件集成

请参考 开始直播 集成 AtomicXCore，并完成 LiveCoreView 的接入。

步骤2：实现跨房连线

此步骤的目标是让两个主播的画面出现在同一个视图中，我们将使用 CoHostStore 来完成。

邀请方（主播 A）实现

1. 发起连线邀请

   当主播A在界面上选择目标主播 B 并发起连线时，调用 `requestHostConnection` 方法。

```
import AtomicXCore
import Combine

// 主播A的视图控制器
class AnchorAViewController {
    private let liveId = "主播A的房间ID"
    private var cancellables: Set<AnyCancellable> = []
    private lazy var coHostStore: CoHostStore = {
        return CoHostStore.create(liveID: self.liveId)
    }()

    // 用户点击“连线”按钮，并选择了主播B
    func inviteHostB(targetHostLiveId: String) {
        let layout: CoHostLayoutTemplate = .hostDynamicGrid // 选择一个
布局模板
        let timeout: TimeInterval = 30.0 // 邀请超时时间

        coHostStore.requestHostConnection(targetHost:
targetHostLiveId,
                                          layoutTemplate: layout,
                                          timeout: timeout) { result
in
            switch result {
            case .success():
                print("连线邀请已发送，等待对方处理...")
            case .failure(let error):
                print("邀请发送失败: \(error.message)")
            }
```

```
        }
    }
}
```

2. 监听邀请结果

通过订阅 `coHostEventPublisher`，您可以接收到主播 B 的处理结果。

```
// 在 AnchorAViewController 初始化时设置监听
func setupListeners() {
    coHostStore.coHostEventPublisher
        .sink { [weak self] event in
            switch event {
            case .onCoHostRequestAccepted(let invitee): //
                print("主播 \(invitee.userName) 同意了你的连线邀请")
            case .onCoHostRequestRejected(let invitee): //
                print("主播 \(invitee.userName) 拒绝了你的邀请")
            case .onCoHostRequestTimeout: //
                print("邀请超时，对方未回应")
            default:
                break
            }
        }
        .store(in: &cancellables)
}
```

受邀方（主播 B）实现

1. 接收连线邀请

通过 `coHostEventPublisher`，主播B可以监听到来自主播 A 的邀请。

```
import AtomicXCore
import Combine

// 主播B的视图控制器
class AnchorBViewController {
    // ... coHostStore 和 cancellables 初始化 ...

    // 在初始化时设置监听
    func setupListeners() {
```

```
        coHostStore.coHostEventPublisher
            .sink { [weak self] event in
                if case let .onCoHostRequestReceived(inviter, _) =
event { //
                    print("收到主播 \(inviter.userName) 的连线邀请")
                    // self?.showInvitationDialog(from: inviter)
                }
            }
            .store(in: &cancellables)
    }
}
```

2. **响应连线邀请**

   当主播 B 在弹出的对话框中做出选择后，调用相应的方法。

```
// AnchorBViewController 的一部分
func acceptInvitation(fromHostLiveId: String) {
    coHostStore.acceptHostConnection(fromHostLiveID: fromHostLiveId,
completion: nil) //
}

func rejectInvitation(fromHostLiveId: String) {
    coHostStore.rejectHostConnection(fromHostLiveID: fromHostLiveId,
completion: nil) //
}
```

步骤3：实现主播 PK

连线成功后，任意一方都可以发起 PK，此步骤我们将使用 BattleStore 来实现主播 PK。

挑战方（例如主播 A）实现

1. **发起 PK 挑战**

   当主播 A 点击“PK”按钮时，调用 `requestBattle` 方法。

```
// AnchorAViewController 的一部分
private lazy var battleStore: BattleStore = BattleStore.create(liveID:
self.liveId)
```

```
func startPK(with opponentUserId: String) {
    var config = BattleConfig(duration: 300) // PK持续5分钟
    battleStore.requestBattle(config: config, userIDList:
[opponentUserId], timeout: 30.0, completion: nil)
}
```

2. 监听 PK 状态

通过 `battleEventPublisher` 监听 PK 的开始、结束等关键事件。

```
// 在 AnchorAViewController 的 setupListeners 方法中添加
battleStore.battleEventPublisher
    .sink { [weak self] event in
        switch event {
        case .onBattleStarted: //
            print("PK 开始")
        case .onBattleEnded: //
            print("PK 结束")
        default:
            break
        }
    }
    .store(in: &cancellables)
```

应战方（主播 B）实现

1. 接收 PK 挑战

通过 `battleEventPublisher` 监听到 PK 邀请。

```
// 在 AnchorBViewController 的 setupListeners 方法中添加
battleStore.battleEventPublisher
    .sink { [weak self] event in
        if case let .onBattleRequestReceived(battleId, inviter, _) =
event {
            print("收到主播 \(inviter.userName) 的PK挑战")
            // 弹出对话框，让主播B选择“接受”或“拒绝”
            // self?.showPKChallengeDialog(battleId: battleId)
        }
    }
```

```
    .store(in: &cancellables)
```

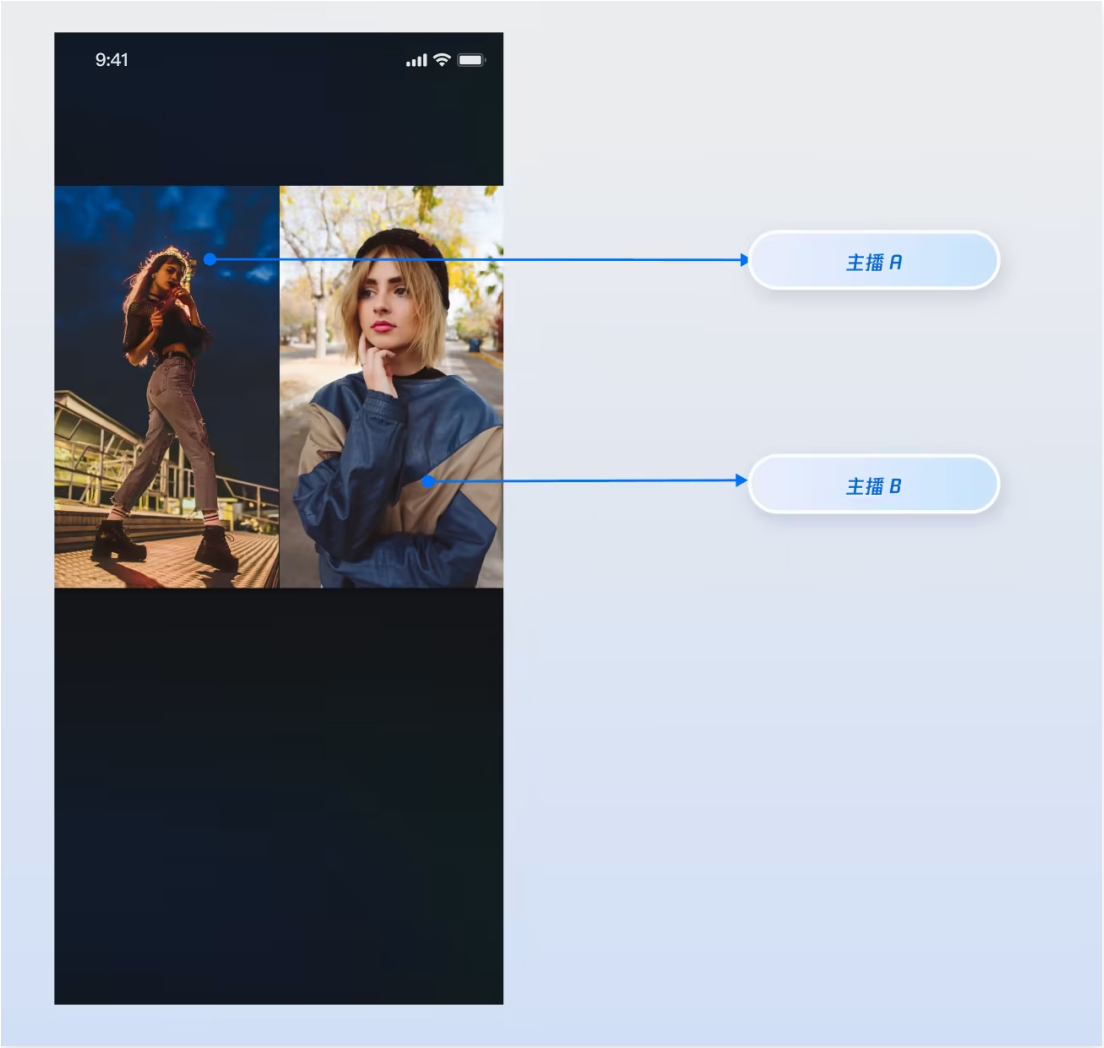
2. **响应 PK 挑战**

当主播 B 做出选择后，调用相应的方法。

```
// AnchorBViewController 的一部分
// 用户点击“接受挑战”
func acceptPK(battleId: String) {
    battleStore.acceptBattle(battleID: battleId) { result in
        // ...
    }
}

// 用户点击“拒绝挑战”
func rejectPK(battleId: String) {
    battleStore.rejectBattle(battleID: battleId) { result in
        // ...
    }
}
```

运行效果

当您集成以上功能实现后，请分别使用主播 A 和主播 B 进行对应操作，运行效果如下，您可以参考下一章节 完善 UI 细节 来定制您想要的 UI 逻辑。

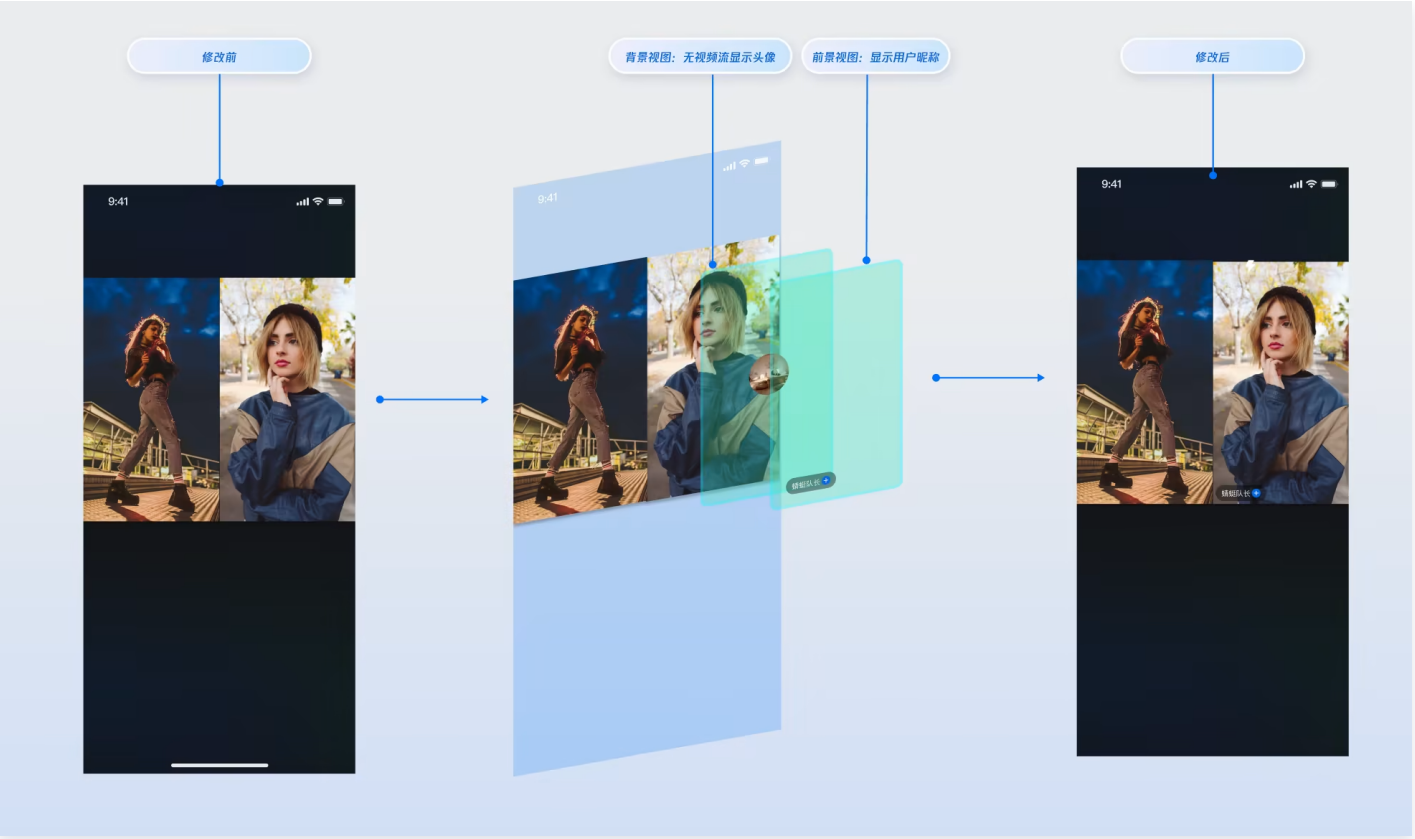


完善 UI 细节

您可以通过 `LiveCoreView.VideoViewDelegate` 协议提供的“插槽”能力，在视频流画面上添加自定义视图，用于显示昵称、头像、PK 进度条等信息，或在他们关闭摄像头时提供占位图，以优化视觉体验。

实现视频流画面的昵称显示

实现效果



实现方式

- **步骤1：创建前景视图（CustomSeatView），该视图用于在视频流上方显示用户信息。**

> **提示：**
> 您也可以参考 TUILiveKit 开源项目中的 AnchorCoHostView.swift 和 AnchorEmptySeatView.swift 文件来了解完整的实现逻辑。

```
import UIKit
import SnapKit

// 自定义的用户信息悬浮视图（前景）
class CustomSeatView: UIView {
    lazy var nameLabel: UILabel = {
        let label = UILabel()
        label.textColor = .white
        label.font = .systemFont(ofSize: 14)
        return label
    }()

    override init(frame: CGRect) {
```

```
        super.init(frame: frame)
        backgroundColor = UIColor.black.withAlphaComponent(0.5)
        addSubview(nameLabel)
        nameLabel.snp.makeConstraints { make in
            make.bottom.equalToSuperview().offset(-5)
            make.leading.equalToSuperview().offset(5)
        }
    }
}
```

- **步骤2：创建背景视图（CustomAvatarView），该视图用于在用户无视频流时作为占位图显示。**

```
import UIKit
import SnapKit

// 自定义的头像占位视图（背景）
class CustomAvatarView: UIView {
    lazy var avatarImageView: UIImageView = {
        let imageView = UIImageView()
        imageView.tintColor = .gray
        return imageView
    }()

    override init(frame: CGRect) {
        super.init(frame: frame)
        backgroundColor = .clear
        layer.cornerRadius = 30
        addSubview(avatarImageView)
        avatarImageView.snp.makeConstraints { make in
            make.center.equalToSuperview()
            make.width.height.equalTo(60)
        }
    }
}
```

- **步骤3：实现 `VideoViewDelegate.createCoHostView` 协议，根据 `viewLayer` 的值返回对应的视图。**

```
import AtomicXCore
import RTCRoomEngine
```

```
// 1. 在您的视图控制器中，遵守 VideoViewDelegate 协议
class YourViewController: UIViewController, VideoViewDelegate {

    // ... 其他代码 ...

    // 2. 完整实现协议方法，处理两种 viewLayer
    func createCoHostView(seatInfo: TUISeatFullInfo, viewLayer:
ViewLayer) -> UIView? {
        guard let userId = seatInfo.userId, !userId.isEmpty else {
            return nil
        }
        if viewLayer == .foreground {
            let seatView = CustomSeatView()
            seatView.nameLabel.text = seatInfo.userName
            return seatView
        } else { // viewLayer == .background
            let avatarView = CustomAvatarView()
            // 您可以在这里通过 seatInfo.userAvatar 加载用户真实头像
            return avatarView
        }
    }
}
```

参数说明：

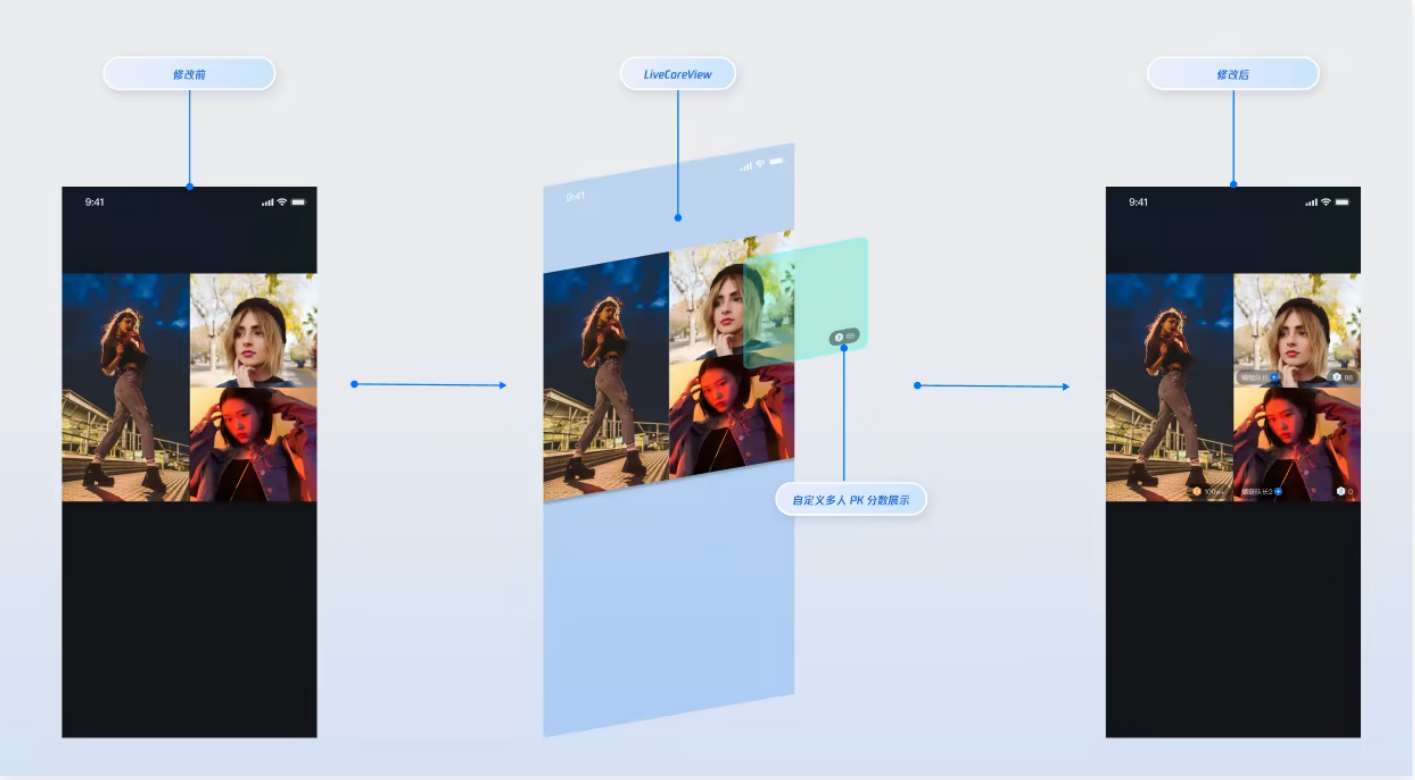
参数	类型	说明
`seatInfo`	`TUISeatFullInfo`	麦位信息对象，包含麦上用户的详细信息
`seatInfo.userId`	`String?`	麦上用户的 ID
`seatInfo.userName`	`String?`	麦上用户的昵称
`seatInfo.userAvatar`	`String?`	麦上用户的头像 URL
`seatInfo.userMicrophoneStatus`	`TUIDeviceStatus`	麦上用户的麦克风状态

`seatInfo.userCameraStatus`	`TUIDeviceStatus`	麦上用户的摄像头状态
`viewLayer`	`ViewLayer`	视图层级枚举 `.foreground` 表示前景挂件视图，始终显示在视频画面的最上层 `.background` 表示背景挂件视图，位于前景视图下层，仅在对应用户没有视频流（例如未开摄像头）的情况下显示，通常用于展示用户的默认头像或占位图

实现 PK 用户视图的分数展示

当主播开始 PK 后，可以在对方主播的视频画面上挂载自定义视图，通常用于展示该主播收到的礼物价值或其它 PK 相关信息。

实现效果



实现方式

- **步骤1：**创建自定义 PK 用户视图，您可以参考 TUILiveKit 开源项目中的 AnchorBattleMemberInfoView.swift 文件来了解完整的实现逻辑。

```
import AtomicXCore
import RTCRoomEngine
```

```
import SnapKit

// 自定义PK 用户视图
class CustomBattleUserView: UIView {
    private let scoreView: UIView = {
        let view = UIView()
        view.backgroundColor = .black.withAlphaComponent(0.4)
        view.layer.cornerRadius = 12
        return view
    }()

    private lazy var scoreLabel: UILabel = {
        let label = UILabel()
        label.textColor = .white
        label.font = .systemFont(ofSize: 14, weight: .bold)
        return label
    }()

    private var userId: String
    private let battleStore: BattleStore
    private var cancellableSet: Set<AnyCancellable> = []

    init(liveId: String, battleUser: TUIBattleUser) {
        self.userId = battleUser.userId
        self.battleStore = BattleStore.create(liveID: liveId)
        super.init(frame: .zero)
        backgroundColor = .clear
        isUserInteractionEnabled = false
        // UI 布局
        setupUI()
        // 订阅分数变化
        subscribeBattleState()
    }

    private func setupUI() {
        addSubView(scoreView)
        scoreView.addSubview(scoreLabel)
        scoreLabel.snp.makeConstraints { make in
            make.leading.trailing.equalToSuperview().inset(5)
        }
```

```
        scoreView.snp.makeConstraints { make in
            make.height.equalTo(24)
            make.bottom.equalToSuperview().offset(-5)
            make.trailing.equalToSuperview().offset(-5)
        }
    }

    // 订阅 PK 分数变化
    private func subscribeBattleState() {
        battleStore.state
            .subscribe(StatePublisherSelector(keyPath:
\BattleState.battleScore))
            .removeDuplicates()
            .receive(on: RunLoop.main)
            .sink { battleUsers in
                guard let self = self else { return }
                guard let score = battleScore[self.userId] else {
return }
                // 更新 UI
                self.scoreLabel.text = "\(battleUser.score)"
            }
            .store(in: &cancellableSet)
    }
}
```

- **步骤2：实现 `VideoViewDelegate.createBattleView` 协议**

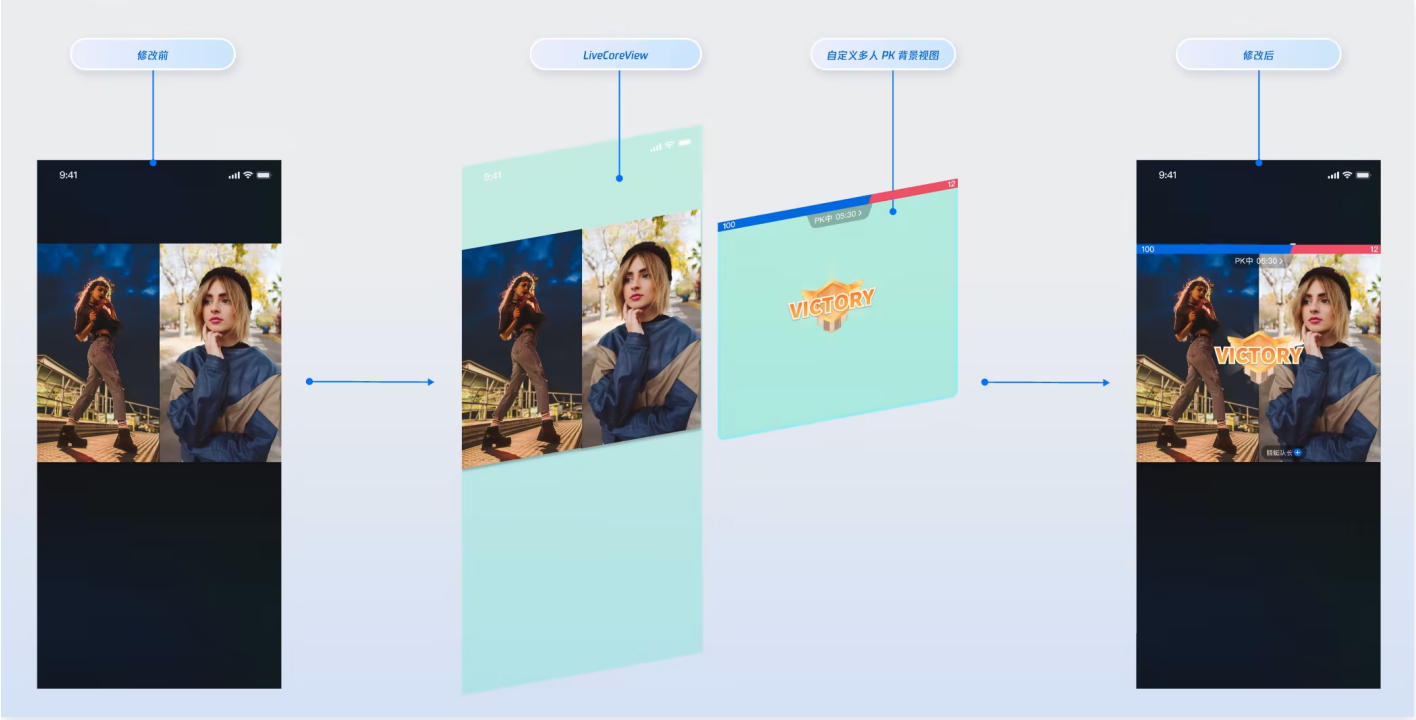
```
  // 1. 让您的视图控制器遵守 VideoViewDelegate 协议
extension YourViewController: VideoViewDelegate {

    public func createBattleView(battleUser: TUIBattleUser) -> UIView?
{
        // CustomBattleUserView 是您自定义的PK用户信息视图
        let customView = CustomBattleUserView(liveId:liveId,
battleUser:battleUser)
        return customView
    }

}
```

参数说明：

参数	类型	说明
`battleUser`	`TUIBattleUser`	PK 用户信息对象
`battleUser.roomId`	`String`	PK 的房间 ID
`battleUser.userId`	`String`	PK 用户 ID
`battleUser.userName`	`String`	PK 用户昵称
`battleUser.avatarUrl`	`String`	PK 用户头像地址
`battleUser.score`	`UInt`	PK 分数

实现视频流画面上的 PK 状态显示

实现效果



实现方式

- **步骤1：**创建自定义 PK 全局视图 **CustomBattleContainerView**，您可以参考 TUILiveKit 开源项目中的 AnchorBattleInfoView.swift 文件来实现，即可实现同样的效果。
- **步骤2：**实现 `VideoViewDelegate.createBattleContainerView` 协议。

```
// 让您的视图控制器遵守 VideoViewDelegate 协议并设置代理
extension YourViewController: VideoViewDelegate {
    func createBattleContainerView() -> UIView? {
        return CustomBattleContainerView()
    }
}
```

功能进阶

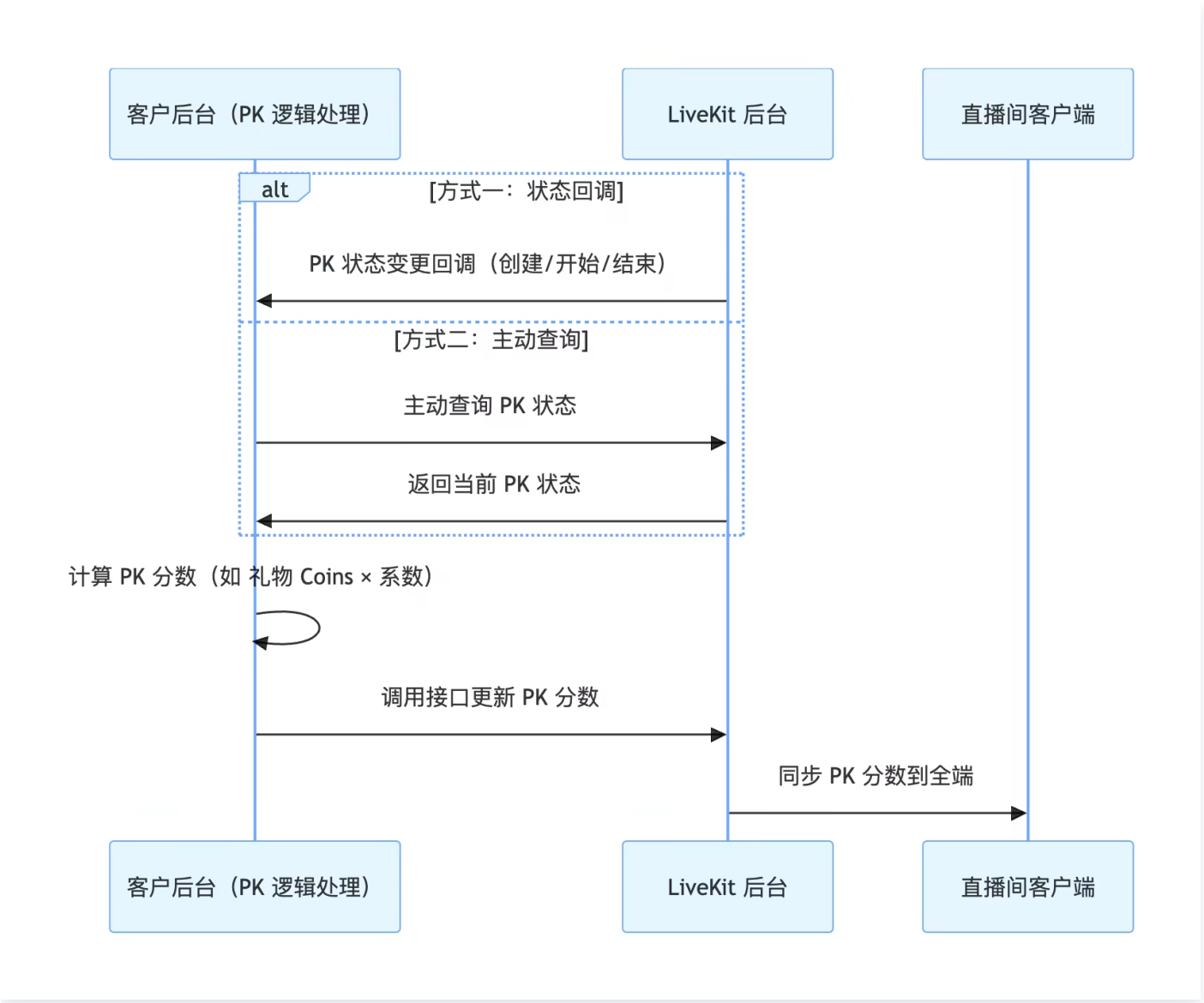
通过 REST API 实现 PK 分数更新

通常在**直播主播 PK 场景**下，会将主播收到的礼物价值与 PK 数值挂钩（例如：观众送 “火箭” 礼物，主播 PK 分数增加 500 分），您可以通过我们的 REST API，轻松实现直播 PK 场景下的分数实时更新。

> ⚠ **重要说明：**
> LiveKit 后台的 PK 分数系统采用纯数值计算和累加机制，所以您需要根据自身的运营策略和业务需求，调用更新接口前完成 PK 分数的计算，您可以参考如下的 PK 分数计算示例：

礼物类型	分数计算规则	示例
基础礼物	礼物价值 × 5	10元礼物 → 50分
中级礼物	礼物价值 × 8	50元礼物 → 400分
高级礼物	礼物价值 × 12	100元礼物 → 1200分
特效礼物	固定高分数	520元礼物 → 1314分

REST API 调用流程



关键流程说明

1. **获取 PK 状态：**
   - 回调配置：您可以通过配置 PK 状态回调，由 LiveKit 后台在 PK 开始、结束时，主动通知您的系统 PK 状态。
   - 主动查询：您的后台服务可主动调用 PK 状态查询 接口，随时查询当前 PK 状态。
2. **PK分数计算**：您的后台服务根据业务规则（如上述示例），计算 PK 分数增量。
3. **PK分数更新**：您的后台服务调用 修改 PK 分数 接口，向 LiveKit 后台更新 PK 分数。
4. **LiveKit 后台 同步客户端**：LiveKit 后台自动将更新后的 PK 分数同步到所有客户端。

涉及的 REST API 接口

接口	功能描述	请求示例
主动接口 – 查询 PK 状态	可根据此接口查询当前房间是否在 PK	请求示例

主动接口 – 修改 PK 分数	将计算后的 PK 数值通过此接口更新	请求示例
回调配置 – PK 开始时回调	客户后台可以通过该回调及时知晓 PK 开启	回调示例
回调配置 – PK 结束时回调	客户后台可以通过该回调及时知晓 PK 结束	回调示例

API 文档

关于 CoHostStore 及其相关类的所有公开接口、属性和方法的详细信息，请参阅随 AtomicXCore 框架的官方 API 文档。本指南使用到的相关 Store 如下：

Store/Component	功能描述	API 文档
LiveCoreView	直播视频流展示与交互的核心视图组件：负责视频流渲染和视图挂件处理，支持主播直播、观众连麦、主播连线等场景。	API 文档
DeviceStore	音视频设备控制：麦克风（开关 / 音量）、摄像头（开关 / 切换 / 画质）、屏幕共享，设备状态实时监听。	API 文档
CoHostStore	主播跨房连线：支持多布局模板（动态网格等），发起 / 接受 / 拒绝连线，连麦主播互动管理。	API 文档
BattleStore	主播 PK 对战：发起 PK（配置时长 / 对手），管理 PK 状态（开始 / 结束），同步分数，监听对战结果。	API 文档

常见问题

为什么发起了连线邀请，对方却没收到？

- 请检查 `targetHostLiveId` 是否正确，并且对方直播间处于正常开播状态。
- 检查网络连接是否通畅，邀请信令有30秒的默认超时时间。

连线或 PK 过程中，一方主播网络断开或 App 崩溃了怎么办？

CoHostStore 和 BattleStore 内部都有心跳和超时检测机制。如果一方异常退出，另一方会通过 onCoHostUserLeft 或 onUserExitBattle 等事件收到通知，您可以根据这些事件来处理UI，例如提示“对方已掉线”并结束互动。

为什么 PK 分数只能通过 REST API 更新?

因为 REST API 能同时满足 PK 分数的**安全性、实时性、扩展性**需求：

- **防篡改保公平**：需鉴权 + 数据校验，每笔更新可追溯来源（例如礼物行为），杜绝手动改分、刷分，保障竞技公平；

- **多端实时同步**：用标准化格式（例如 JSON）快速对接礼物、PK、展示系统，确保主播 / 观众 / 后台分数实时一致，无延迟；
- **灵活适配规则**：后端改配置（例如调整礼物对应分数、加成分数）即可适配业务变化，无需改前端，降低迭代成本。

如何管理通过 `VideoViewDelegate` 添加的自定义视图的生命周期和事件？

LiveCoreView 会自动管理您通过代理方法返回视图的添加和移除，您无需手动处理。如果需要在自定义视图中处理用户交互（例如点击事件），请在创建视图时为其添加相应的事件即可。

弹幕
弹幕（Android）

最近更新时间：2025-11-21 14:15:34

本篇文档旨在指导 Android 开发者如何使用 `AtomicXCore` 框架中的 `BarrageStore` 模块，为您的直播应用快速集成功能丰富、性能卓越的弹幕系统。

核心功能

`BarrageStore` 为您的直播应用提供了一套完整的弹幕解决方案，核心功能包括：

- 接收并展示直播间弹幕消息。
- 发送文本弹幕与观众互动。
- 发送自定义业务弹幕，以支持礼物、点赞等复杂场景。
- 在本地消息列表中插入系统提示（例如，"欢迎 XX 进入直播间"）。

核心概念解析

在开始集成之前，我们先通过下表了解一下 BarrageStore 相关的几个核心概念：

核心概念	类型	核心职责与描述
Barrage	`data class`	代表一条弹幕消息的数据模型。它包含了发送者信息（`sender`）、消息内容（`textContent` 或 `data`）、消息类型（`messageType`）等所有关键信息。
Barrage State	`data class`	代表弹幕模块的当前状态。其核心属性 `messageList` 是一个 `StateFlow`，按时间顺序存储了当前直播间的所有弹幕消息，是UI渲染的数据源。
Barrage Store	`abstract class`	这是与弹幕功能交互的核心管理类。通过它，您可以发送消息（`sendTextMessage`，`sendCustomMessage`），并通过订阅其 `state` 属性来接收和监听所有弹幕消息的更新。

实现步骤

步骤1：组件集成

请参考 开始直播 集成 AtomicXCore，完成接入。

步骤2：初始化并监听弹幕

获取一个与当前直播间 `liveId` 绑定的 `BarrageStore` 实例，并设置一个订阅者来实时接收最新的全量弹幕消息列表。

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import io.trtc.tuikit.atomicxcore.api.barrage.Barrage
import io.trtc.tuikit.atomicxcore.api.barrage.BarrageStore

class BarrageManager(
    private val liveId: String
) {
    private val barrageStore: BarrageStore = BarrageStore.create(liveId)
    private val scope = CoroutineScope(Dispatchers.Main)

    // 对外暴露【全量】消息列表的Flow，方便UI层订阅
    private val _messagesFlow = MutableStateFlow<List<Barrage>>
(emptyList())
    val messagesFlow: StateFlow<List<Barrage>> =
_messagesFlow.asStateFlow()

    init {
        // 初始化后立即开始监听弹幕消息
        subscribeToBarrageUpdates()
    }

    private fun subscribeToBarrageUpdates() {
        scope.launch {
            barrageStore.barrageState.messageList
                .collect { messageList ->
                    // 当 messageList 更新时，通过 Flow 将新列表传递给UI层
                    // 关键点：这里接收到的是包含所有历史消息的【完整列表】
                    _messagesFlow.value = messageList
                }
        }
    }
}
```

步骤3：发送文本弹幕

调用 `sendTextMessage` 方法向直播间内的所有用户广播一条纯文本消息。

```
import io.trtc.tuikit.atomicxcore.api.CompletionHandler
import io.trtc.tuikit.atomicxcore.api.barrage.BarrageStore

class BarrageManager(
    private val liveId: String
) {
    private val barrageStore: BarrageStore = BarrageStore.create(liveId)

    // 发送一条文本弹幕
    fun sendTextMessage(text: String) {
        // 建议：增加非空校验，避免发送无效消息
        if (text.isEmpty()) {
            return
        }

        // 调用核心API发送消息
        barrageStore.sendTextMessage(
            text,
            mapOf(
                "custom key" to "custom value",
            ),
            object : CompletionHandler {
                override fun onSuccess() {
                    println("文本弹幕 '$text' 发送成功")
                }

                override fun onFailure(code: Int, desc: String) {
                    println("文本弹幕发送失败: $desc")
                }
            }
        )
    }
}
```

接口参数

参数	类型	描述

`text`	`String?`	要发送的文本内容。
`extensionInfo`	`Map<String, String>?`	附加的扩展信息，可用于业务自定义。
`completion`	`CompletionHandler?`	发送完成后的回调，包含成功或失败的结果。

步骤4：发送自定义弹幕

发送一条包含自定义业务逻辑的消息，例如礼物、点赞或游戏化互动指令。这条消息对其他客户端来说是透明的，需要业务层自行解析。

```
class BarrageManager(
    private val liveId: String
) {
    // 发送一条自定义弹幕，例如用于发送礼物
    fun sendGiftMessage(giftId: String, giftCount: Int) {
        // 1. 定义一个能识别业务的ID
        val businessId = "live_gift"

        // 2. 将业务数据编码为 JSON 字符串
        val giftData = mapOf(
            "gift_id" to giftId,
            "gift_count" to giftCount
        )

        val jsonString = try {
            // 使用 Gson 或其他 JSON 库进行序列化
            // 这里假设使用 Gson
            Gson().toJson(giftData)
        } catch (e: Exception) {
            return
        }

        // 3. 调用核心API发送自定义消息
        barrageStore.sendCustomMessage(
            businessId,
            jsonString,
            object : CompletionHandler {
                override fun onSuccess() {
```

```
                    println("礼物消息（自定义弹幕）发送成功")
                }

                override fun onFailure(code: Int, desc: String) {
                    println("礼物消息发送失败: $desc")
                }
            }
        )
    }
}
```

接口参数

参数	类型	描述
`businessId`	`String`	业务唯一标识符，例如 "live_gift"，用于接收端区分不同的自定义消息。
`data`	`String`	业务数据，通常为 JSON 格式的字符串。
`completion`	`CompletionHandler?`	发送完成后的回调。

步骤5：在本地插入提示消息

在当前用户的消息列表中插入一条本地消息，这条消息不会被发送到直播间的其他用户。这非常适合用来显示系统欢迎、警告或操作提示等信息。

```
class BarrageManager(
    private val liveId: String
) {
    // 在本地消息列表中插入一条欢迎提示
    fun showWelcomeMessage(user: LiveUserInfo) {
        // 1. 创建一条 Barrage 消息
        val welcomeTip = Barrage(
            liveID = liveId,
            messageType = BarrageType.TEXT, // 可以复用 text 类型来显示
            textContent = "欢迎 ${user.userName} 进入直播间！",
            sender = LiveUserInfo() // sender 可以留空或设置为一个系统用户的标
识
```

```
        )

        // 2. 调用API将其插入本地列表
        barrageStore.appendLocalTip(welcomeTip)
    }
}
```

接口参数

参数	类型	描述
`message`	`Barrage`	要在本地插入的消息对象。SDK 会将此消息对象追加到 BarrageState 的 messageList 中。

步骤6：管理用户发言（禁言与解禁）

作为主播或管理员，您可以对直播间内的用户发言权限进行管理，维护健康的社区氛围。

禁止/解禁单个用户发言

通过 `LiveAudienceStore` 中的 `disableSendMessage` 接口来实现对指定用户的禁言或解禁。此状态会被持久化，即使用户重新进入直播间，禁言状态依然有效。

```
import io.trtc.tuikit.atomicxcore.api.*
import io.trtc.tuikit.atomicxcore.api.live.LiveAudienceStore

// 1. 获取与当前直播间绑定的 LiveAudienceStore 实例
val audienceStore = LiveAudienceStore.create(liveId)

// 2. 定义要操作的用户ID和禁言状态
val userIdToMute = "user_id_to_be_muted"
val shouldDisable = true // true为禁言, false为解禁

// 3. 调用接口执行操作
audienceStore.disableSendMessage(
    userIdToMute,
    shouldDisable,
    object : CompletionHandler {
        override fun onSuccess() {
```

```
            println("${if (shouldDisable) "禁言" else "解禁"}用户
$userIdToMute 成功")
        }

        override fun onFailure(code: Int, desc: String) {
            println("操作失败: $desc")
        }
    }
)
```

开启/关闭全体禁言

要对直播间内所有用户（通常不包括主播自己）进行禁言，您需要通过 LiveListStore 更新直播间信息来实现。

```
import io.trtc.tuikit.atomicxcore.api.CompletionHandler
import io.trtc.tuikit.atomicxcore.api.live.LiveInfo
import io.trtc.tuikit.atomicxcore.api.live.LiveListStore

// 1. 获取 LiveListStore 单例
val liveListStore = LiveListStore.shared()

// 2. 获取当前直播间信息，并修改全体禁言状态
val currentLiveInfo = liveListStore.liveState.currentLive.value.copy(
    isMessageDisable = true // true为全体禁言, false为关闭
)

// 3. 调用更新接口，并指定修改的标志位
liveListStore.updateLiveInfo(
    currentLiveInfo,
    listOf(LiveInfo.ModifyFlag.IS_MESSAGE_DISABLE),
    object : CompletionHandler {
        override fun onSuccess() {
            println("全体禁言状态更新成功")
        }

        override fun onFailure(code: Int, desc: String) {
            println("操作失败: $desc")
        }
    }
```

```
)
```

功能进阶：高并发场景下的性能优化

当您使用 `BarrageStore` 构建弹幕功能后，本章将指导您如何处理更复杂的业务场景，确保它能在真实、复杂的高并发直播场景下，依然为用户提供流畅、稳定的体验。本章将围绕三个核心业务场景，为您提供明确的优化方案和代码示例。

场景一：应对热门直播间的"弹幕风暴"

场景描述

在一场热门活动中，直播间涌入大量观众，弹幕以每秒数十条的频率刷新。

技术挑战

SDK会以极高频率返回完整的弹幕列表。如果每次都调用 `adapter.notifyDataSetChanged()`，主线程会被密集的 UI 布局和渲染操作阻塞，导致界面卡顿。

优化方案：批处理与流量削峰 (Batch Processing & Debouncing)

不必响应每一次的数据更新，而是设定一个时间阈值（例如300毫秒）。只在距离上次UI刷新超过这个阈值后，才执行下一次刷新操作。这能将每秒数十次的 `notifyDataSetChanged()` 调用，降低到每秒3-4次，极大提升流畅度。

代码示例

创建一个 `BarrageUIManager` 类，它内置一个缓冲区和一个定时器，专门负责将数据批量更新到 `RecyclerView`。

```
import android.os.Handler
import androidx.recyclerview.widget.RecyclerView
import io.trtc.tuikit.atomicxcore.api.barrage.Barrage

private const val UPDATE_DURATION_MS = 300L

class BarrageUIManager() {
    private var timestampOnLastUpdate = 0L

    private var dataSource: ArrayList<Barrage> = ArrayList()
    private val updateViewTask = Runnable { notifyDataSetChanged() }
    private val handler = Handler()
    private val adapter = BarrageAdapter(dataSource)
```

```
    // 外部高频调用此方法，传入最新的全量列表
    fun update(newList: List<Barrage>) {
        handler.removeCallbacks(updateViewTask)
        dataSource.clear()
        dataSource.addAll(newList)
        // 刷新频率小于 0.3 秒，不做刷新
        if (System.currentTimeMillis() - timestampOnLastUpdate <
UPDATE_DURATION_MS) {
            handler.postDelayed(updateViewTask, UPDATE_DURATION_MS)
            return
        }
    }

    private fun notifyDataSetChanged() {
        adapter.notifyDataSetChanged()
        timestampOnLastUpdate = System.currentTimeMillis()
    }
}

class BarrageAdapter(dataSource: ArrayList<Barrage>):
RecyclerView.Adapter<RecyclerView.ViewHolder>() {
    //这里实现adapter能力
}
```

场景二：保障长时间直播的内存稳定性

场景描述

您的应用需要支持数小时乃至全天候的"不间断直播"，例如游戏直播或慢直播。在此期间，App 必须保持稳定运行，不能因为长时间运行而意外退出。

技术挑战

SDK 返回的全量 `messageList` 会在长时间直播中无限增长，即使 UI 层做了节流，数据层持有的这个巨大数组也会持续侵占内存，最终导致应用闪退。

优化方案：固定容量的循环数组 (Circular Buffer)

只让您自己的数据源（DataSource）持有有限数量的消息。无论 SDK 返回的全量列表有多大，您的应用只截取其中最新的部分用于显示。

代码示例

在接收到 SDK 的全量列表后，只取最新的500条（或您定义的其他数量）来更新 UI。

```
class BarrageUIManager(
    private val recyclerView: RecyclerView
) {
    private val capacity: Int = 500 // 客户端只保留最新的500条消息
    // ...（其他代码同上）...

    private fun refreshUIIfNeeded() {
        val fullList = this.latestMessageList ?: return
        val adapter = recyclerView.adapter ?: return
        this.latestMessageList = null

        // 关键点：截取最新的N条消息
        val cappedList = fullList.takeLast(capacity)

        dataSource.clear()
        dataSource.addAll(cappedList)
        adapter.notifyDataSetChanged()
    }
}
```

场景三：渲染包含用户等级、徽章的复杂弹幕样式

场景描述

为了增强直播氛围和付费用户荣誉感，弹幕需要展示丰富的视觉元素，例如混排用户名、用户等级图标、粉丝徽章和消息内容。

技术挑战

渲染包含多图文、自定义字体或复杂布局的视图，比渲染纯文本耗时更长。在列表中高频渲染这些复杂视图，会加重主线程的计算压力，导致列表滚动时出现卡顿。

优化方案：异步绘制 (Asynchronous Drawing)

将视图内容的绘制过程从主线程剥离，放到后台线程去完成。主线程只负责最终的"贴图"操作，即显示已经绘制好的位图。这极大地减轻了主线程的计算压力。

代码示例

在您的自定义 `RecyclerView.ViewHolder` 中，通过使用 `AsyncLayoutInflater` 或预加载机制，来指示系统尽可能地在后台线程完成绘制任务。

```
import android.view.LayoutInflater
import androidx.asynclayoutinflater.view.AsyncLayoutInflater
import androidx.recyclerview.widget.RecyclerView

// 在你的自定义弹幕ViewHolder中
class BarrageViewHolder(
    itemView: View
) : RecyclerView.ViewHolder(itemView) {

    // ... (TextView, ImageView等子视图的声明)

    companion object {
        fun create(parent: ViewGroup): BarrageViewHolder {
            // 使用异步布局加载器来减少主线程阻塞
            val inflater = AsyncLayoutInflater(parent.context)
            val view = LayoutInflater.from(parent.context)
                .inflate(R.layout.item_barrage, parent, false)
            return BarrageViewHolder(view)
        }
    }

    fun bind(viewModel: BarrageViewModel) {
        // 在这里设置你的TextView文本、ImageView图片等
        // 由于使用了异步布局加载，这些内容的最终渲染会尽量在后台完成
    }

    // 追求极致性能的进阶方案：完全手动绘制
    // 如果需要更精细的控制，可以使用Canvas手动绘制文本和图片
    // 结合后台线程生成Bitmap，然后在主线程的onDraw方法中绘制它，可以实现完全的异步
渲染
}
```

API 文档

关于 BarrageStore 及其相关类的所有公开接口、属性和方法的详细信息，请参阅随 AtomicXCore 框架的官方 API 文档。

常见问题

除了基础的文本弹幕，我们还希望实现"彩色弹幕"、"礼物弹幕"等更丰富的样式，该如何实现？

这是通过自定义消息 `sendCustomMessage` 来实现的。`BarrageStore` 不会限制您的业务想象力。

实现思路

1. **定义数据结构：**与您的客户端和服务器团队共同定义好自定义消息的 JSON 结构。例如，一条彩色弹幕可以这样定义：

```
{  "type": "colored_text",  "text": "这是一条彩色弹幕!",  "color": "#FF5733" }
```

2. **发送端：**在发送时，将这个 JSON 结构转换为字符串，并通过 `sendCustomMessage` 的 `data` 参数发送出去。`businessID` 可以设置为一个能代表您业务的唯一标识，例如 "`barrage_style_v1`"。
3. **接收端：**在接收到弹幕消息后，检查其 `messageType` 是否为 `BarrageType.CUSTOM` 以及 `businessID` 是否匹配。如果匹配，则解析 `data` 字符串（通常是解析 JSON），根据解析出的数据（例如 `color`、`text`）来渲染您的自定义UI样式。

我在不同的类、不同的文件中都调用了 `BarrageStore.create(liveID = "some_id")`，这会创建出多个实例导致混乱吗？

完全不会。`AtomicXCore` 内部机制会确保只要您传入的 `liveID` 相同，获取到的永远是同一个与该直播间绑定的 `BarrageStore` 实例。您可以在需要的地方随用随取，无需手动管理单例。

为什么我调用了 sendTextMessage，但是在消息列表中看不到我发送的消息？

请按以下步骤排查：

1. **检查 completion 回调**：sendTextMessage 方法有一个完成回调。请检查回调返回的结果是成功还是失败。如果失败，错误信息会明确指出问题所在（例如"您已被禁言"、"网络错误"等）。
2. **确认订阅时机**：确保您对 barrageStore.barrageState.messageList 的订阅发生在该 liveID 对应的直播开始之后。如果在加入直播房间之前就开始监听，可能会错过部分消息。
3. **检查 liveID**：确认您在创建 BarrageStore 实例、加入直播房间、以及发送消息时使用的 liveID 完全一致，包括大小写。
4. **网络问题**：检查设备当前的网络连接是否正常。消息发送依赖于网络。

新观众进入直播间时，如何让他们看到加入前的历史弹幕消息？

`AtomicXCore` 支持拉取历史弹幕消息，但这需要您在**服务端控制台**进行一项简单的配置。配置完成后，SDK会自动处理后续的一切，您无需编写额外的代码。

步骤1：在 IM 控制台进行配置

1. 登录您的 即时通讯 IM 控制台 。

2. 在左侧导航栏，按照路径**消息服务 Chat > 功能配置 > 群组配置 > 群功能配置 > 直播群新成员查看入群前消息量配置**进行导航。

3. 修改"新成员可查看最近消息数"，最大支持 50 条。

步骤2：客户端无感获取

完成上述配置后，您的客户端代码无需做任何改动。

当新用户加入直播间时，`AtomicXCore` 的底层会自动拉取您配置的历史消息数量。这些历史消息会和实时消息一样，通过您已实现的 `BarrageStore.barrageState.messageList` 订阅通道推送给您的 UI 层。您的应用会像接收实时弹幕一样，自然地接收并展示这些历史弹幕。

弹幕（iOS）

最近更新时间：2025-11-21 14:15:34

本篇文档旨在指导 iOS 开发者如何使用 `AtomicXCore` 框架中的 `BarrageStore` 模块，为您的直播应用快速集成功能丰富、性能卓越的弹幕系统。

核心功能

`BarrageStore` **为您的直播应用提供了一套完整的弹幕解决方案，核心功能包括：**

- **接收并展示直播间弹幕消息。**
- **发送文本弹幕与观众互动。**
- **发送自定义业务弹幕，以支持礼物、点赞等复杂场景。**
- **在本地消息列表中插入系统提示（例如，“欢迎 XX 进入直播间”）。**

核心概念解析

在开始集成之前，我们先通过下表了解一下 BarrageStore 相关的几个核心概念：

核心概念	类型	核心职责与描述
Barrage	`struct`	代表一条弹幕消息的数据模型。它包含了发送者信息（`sender`）、消息内容（`textContent` 或 `data`）、消息类型（`messageType`）等所有关键信息。
BarrageState	`struct`	代表弹幕模块的当前状态。其核心属性 `messageList` 是一个 `[Barrage]` 数组，按时间顺序存储了当前直播间的所有弹幕消息，是UI渲染的数据源。
BarrageStore	`class`	这是与弹幕功能交互的核心管理类。通过它，您可以发送消息（`sendTextMessage`，`sendCustomMessage`），并通过订阅其 `state` 属性来接收和监听所有弹幕消息的更新。

实现步骤

步骤1：组件集成

请参考 开始直播 集成 AtomicXCore，完成接入。

步骤2：初始化并监听弹幕

获取一个与当前直播间 `liveId` 绑定的 `BarrageStore` 实例，并设置一个订阅者来实时接收最新的全量弹幕消息列表。

```
import Foundation
import AtomicXCore // 确保导入核心库
import Combine     // 用于处理响应式编程

class BarrageManager {

    private let liveId: String
    private let barrageStore: BarrageStore
    private var cancellables = Set<AnyCancellable>()

    // 对外暴露【全量】消息列表的发布者，方便UI层订阅
    let messagesPublisher = PassthroughSubject<[Barrage], Never>()

    init(liveId: String) {
        self.liveId = liveId
        // 1. 通过 liveId 获取 BarrageStore 的单例
```

```
        self.barrageStore = BarrageStore.create(liveID: liveId)

        // 2. 初始化后立即开始监听弹幕消息
        subscribeToBarrageUpdates()
    }

    private func subscribeToBarrageUpdates() {
        barrageStore.state
            .subscribe() // 订阅 BarrageState 的所有变化
            .receive(on: DispatchQueue.main) // 确保在主线程处理数据，保证UI
安全
            .sink { [weak self] barrageState in
                // 3. 当 messageList 更新时，通过 publisher 将新列表传递给UI
层
                // 关键点：这里接收到的是包含所有历史消息的【完整列表】
                self?.messagesPublisher.send(barrageState.messageList)
            }
            .store(in: &cancellables) // 管理订阅的生命周期
    }
}
```

步骤3：发送文本弹幕

调用 `sendTextMessage` 方法向直播间内的所有用户广播一条纯文本消息。

```
extension BarrageManager {

    /// 发送一条文本弹幕
    func sendTextMessage(text: String) {
        // 建议：增加非空校验，避免发送无效消息
        guard !text.isEmpty else {
            print("弹幕内容不能为空")
            return
        }

        // 调用核心API发送消息
        barrageStore.sendTextMessage(text: text, extensionInfo: nil) {
result in
            // 在回调中处理发送结果，例如给用户提示
            switch result {
```

```
            case .success:
                print("文本弹幕 '\(text)' 发送成功")
            case .failure(let error):
                print("文本弹幕发送失败: \(error.localizedDescription)")
            }
        }
    }
}
```

接口参数

参数	类型	描述
`text`	`String`	要发送的文本内容。
`extensionInfo`	`[String: String]?`	附加的扩展信息，可用于业务自定义。
`completion`	`CompletionClosure?`	发送完成后的回调，包含成功或失败的结果。

步骤4：发送自定义弹幕

发送一条包含自定义业务逻辑的消息，例如礼物、点赞或游戏化互动指令。这条消息对其他客户端来说是透明的，需要业务层自行解析。

```
extension BarrageManager {

    /// 发送一条自定义弹幕，例如用于发送礼物
    func sendGiftMessage(giftId: String, giftCount: Int) {
        // 1. 定义一个能识别业务的ID
        let businessId = "live_gift"

        // 2. 将业务数据编码为 JSON 字符串
        let giftData: [String: Any] = ["gift_id": giftId, "gift_count":
giftCount]
        guard let jsonData = try? JSONSerialization.data(withJSONObject:
giftData),
              let jsonString = String(data: jsonData, encoding: .utf8)
else {
            print("无法将礼物数据编码为JSON")
```

```
            return
        }

        // 3. 调用核心API发送自定义消息
        barrageStore.sendCustomMessage(businessID: businessId, data:
jsonString) { result in
            switch result {
            case .success:
                print("礼物消息（自定义弹幕）发送成功")
            case .failure(let error):
                print("礼物消息发送失败: \(error.localizedDescription)")
            }
        }
    }
}
```

接口参数

参数	类型	描述
`businessId`	`String`	业务唯一标识符，例如 "live_gift"，用于接收端区分不同的自定义消息。
`data`	`String`	业务数据，通常为 JSON 格式的字符串。
`completion`	`CompletionClosure?`	发送完成后的回调。

步骤5：在本地插入提示消息

在当前用户的消息列表中插入一条本地消息，这条消息不会被发送到直播间的其他用户。这非常适合用来显示系统欢迎、警告或操作提示等信息。

```
extension BarrageManager {

    /// 在本地消息列表中插入一条欢迎提示
    func showWelcomeMessage(for user: LiveUserInfo) {
        // 1. 创建一条 Barrage 消息
        var welcomeTip = Barrage()
        welcomeTip.messageType = .text // 可以复用 text 类型来显示
        welcomeTip.textContent = "欢迎 \(user.userName) 进入直播间！"
```

```
        // sender 可以留空或设置为一个系统用户的标识

        // 2. 调用API将其插入本地列表
        barrageStore.appendLocalTip(message: welcomeTip)
    }
}
```

接口参数

参数	类型	描述
`message`	`Barrage`	要在本地插入的消息对象。SDK 会将此消息对象追加到 BarrageState 的 messageList 中。

步骤6：管理用户发言（禁言与解禁）

作为主播或管理员，您可以对直播间内的用户发言权限进行管理，维护健康的社区氛围。

禁止/解禁单个用户发言

通过 `LiveAudienceStore` 中的 `disableSendMessage` 接口来实现对指定用户的禁言或解禁。此状态会被持久化，即使用户重新进入直播间，禁言状态依然有效。

```
import AtomicXCore

// 1. 获取与当前直播间绑定的 LiveAudienceStore 实例
let audienceStore = LiveAudienceStore.create(liveID: "your_live_id")

// 2. 定义要操作的用户ID和禁言状态
let userIdToMute = "user_id_to_be_muted"
let shouldDisable = true // true为禁言, false为解禁

// 3. 调用接口执行操作
audienceStore.disableSendMessage(userID: userIdToMute, isDisable:
shouldDisable) { [weak self] result in
    guard let self = self else { return }
    switch result {
    case .success:
        print("\(shouldDisable ? "禁言" : "解禁")用户 \(userIdToMute) 成
功")
    case .failure(let error):
```

```
        print("操作失败: \(error.localizedDescription)")
    }
}
```

开启/关闭全体禁言

要对直播间内所有用户（通常不包括主播自己）进行禁言，您需要通过 LiveListStore 更新直播间信息来实现。

```
import AtomicXCore

// 1. 获取 LiveListStore 单例
let liveListStore = LiveListStore.shared

// 2. 获取当前直播间信息，并修改全体禁言状态
var currentLiveInfo = liveListStore.state.value.currentLive
currentLiveInfo.isMessageDisable = true // true为全体禁言, false为关闭

// 3. 调用更新接口，并指定修改的标志位
liveListStore.updateLiveInfo(currentLiveInfo, modifyFlag:
.isMessageDisable) { result in
    switch result {
    case .success:
        print("全体禁言状态更新成功")
    case .failure(let error):
        print("操作失败: \(error.localizedDescription)")
    }
}
```

功能进阶：高并发场景下的性能优化

当您使用 `BarrageStore` 构建弹幕功能后，本章将指导您如何处理更复杂的业务场景，确保它能在真实、复杂的高并发直播场景下，依然为用户提供流畅、稳定的体验。本章将围绕三个核心业务场景，为您提供明确的优化方案和代码示例。

场景一：应对热门直播间的“弹幕风暴”

场景描述

在一场热门活动中，直播间涌入大量观众，弹幕以每秒数十条的频率刷新。

技术挑战

SDK 会以极高频率返回完整的弹幕列表。如果每次都调用 `tableView.reloadData()`，主线程会被密集的UI布局和渲染操作阻塞，导致界面卡顿。

优化方案：批处理与流量削峰 (Batch Processing & Debouncing)

不必响应每一次的数据更新，而是设定一个时间阈值（例如300毫秒）。只在距离上次 UI 刷新超过这个阈值后，才执行下一次刷新操作。这能将每秒数十次的 `reloadData()` 调用，降低到每秒3-4次，极大提升流畅度。

代码示例

创建一个 `BarrageUIManager` 类，它内置一个缓冲区和一个定时器，专门负责将数据批量更新到 `UITableVie w` 。

```
class BarrageUIManager {
    private var latestMessageList: [BarrageViewModel]?
    private var refreshTimer: Timer?
    private weak var tableView: UITableView?
    private var dataSource: [BarrageViewModel] = []

    init(tableView: UITableView) {
        self.tableView = tableView
        // 每 0.3 秒检查一次是否需要刷新
        self.refreshTimer = Timer.scheduledTimer(withTimeInterval: 0.3,
repeats: true) { [weak self] _ in
            self?.refreshUIIfNeeded()
        }
    }

    /// 外部高频调用此方法，传入最新的全量列表
    func update(with fullList: [BarrageViewModel]) {
        self.latestMessageList = fullList
    }

    private func refreshUIIfNeeded() {
        // 检查是否有新的数据待刷新
        guard let newList = self.latestMessageList, let tableView =
self.tableView else { return }

        self.latestMessageList = nil // 清空标志位，避免重复刷新

        // 更新数据源并刷新UI
```

```
            self.dataSource = newList
            tableView.reloadData()
        }

        deinit {
            refreshTimer?.invalidate()
        }
    }
```

场景二：保障长时间直播的内存稳定性

场景描述

您的应用需要支持数小时乃至全天候的“不间断直播”，例如游戏直播或慢直播。在此期间，App 必须保持稳定运行，不能因为长时间运行而意外退出。

技术挑战

SDK 返回的全量 `messageList` 会在长时间直播中无限增长，即使 UI 层做了节流，数据层持有的这个巨大数组也会持续侵占内存，最终导致应用闪退。

优化方案：固定容量的循环数组 (Circular Buffer)

只让您自己的数据源（DataSource）持有有限数量的消息。无论 SDK 返回的全量列表有多大，您的应用只截取其中最新的部分用于显示。

代码示例

在接收到 SDK 的全量列表后，只取最新的500条（或您定义的其他数量）来更新 UI。

```
class BarrageUIManager {
    private let capacity: Int = 500 // 客户端只保留最新的500条消息
    // ...（其他代码同上）...

    private func refreshUIIfNeeded() {
        guard let fullList = self.latestMessageList, let tableView =
self.tableView else { return }
        self.latestMessageList = nil

        // 关键点：截取最新的N条消息
        let cappedList = Array(fullList.suffix(self.capacity))

        self.dataSource = cappedList
```

```
            tableView.reloadData()
        }
    }
```

场景三：渲染包含用户等级、徽章的复杂弹幕样式

场景描述

为了增强直播氛围和付费用户荣誉感，弹幕需要展示丰富的视觉元素，例如混排用户名、用户等级图标、粉丝徽章和消息内容。

技术挑战

渲染包含多图文、自定义字体或复杂布局的视图，比渲染纯文本耗时更长。在列表中高频渲染这些复杂视图，会加重主线程的计算压力，导致列表滚动时出现卡顿。

优化方案：异步绘制 (Asynchronous Drawing)

将视图内容的绘制过程从主线程剥离，放到后台线程去完成。主线程只负责最终的“贴图”操作，即显示已经绘制好的位图。这极大地减轻了主线程的计算压力。

代码示例

在您的自定义 `UITableViewCell` 中，通过开启 layer 的 `drawsAsynchronously` 属性，来指示系统尽可能地在后台线程完成绘制任务。

```
import UIKit

// 在你的自定义弹幕Cell中
class BarrageCell: UITableViewCell {

    // ... (UILabel, UIImageView等子视图的声明)

    override init(style: UITableViewCell.CellStyle, reuseIdentifier:
String?) {
        super.init(style: style, reuseIdentifier: reuseIdentifier)

        // 开启异步绘制，这是最简单的实现方式
        // 系统会在合适的时机将这个Cell的绘制任务放到后台线程
        self.layer.drawsAsynchronously = true
    }

    required init?(coder: NSCoder) {
```

```
        fatalError("init(coder:) has not been implemented")
    }

    func configure(with viewModel: BarrageViewModel) {
        // 在这里设置你的Label文本、Image图片等
        // 由于开启了异步绘制，这些内容的最终渲染会尽量在后台完成
    }

    // 追求极致性能的进阶方案：完全手动绘制
    override func draw(_ rect: CGRect) {
        // 如果需要更精细的控制，可以在此使用Core Graphics手动绘制文本和图片
        // 结合后台线程生成UIImage，然后在主线程的draw方法中绘制它，可以实现完全的
异步渲染
    }
}
```

API 文档

关于 BarrageStore 及其相关类的所有公开接口、属性和方法的详细信息，请参阅随 AtomicXCore 框架的官方 API 文档。

常见问题

除了基础的文本弹幕，我们还希望实现“彩色弹幕”、“礼物弹幕”等更丰富的样式，该如何实现？

这是通过自定义消息 `sendCustomMessage` 来实现的。`BarrageStore` 不会限制您的业务想象力。

实现思路

1. **定义数据结构：** 与您的客户端和服务器团队共同定义好自定义消息的 JSON 结构。例如，一条彩色弹幕可以这样定义：

   ```
   {  "type": "colored_text",  "text": "这是一条彩色弹幕!",  "color":
   "#FF5733" }
   ```

2. **发送端：** 在发送时，将这个 JSON 结构转换为字符串，并通过 `sendCustomMessage` 的 `data` 参数发送出去。`businessId` 可以设置为一个能代表您业务的唯一标识，例如 "`barrage_style_v1`"。
3. **接收端：** 在接收到弹幕消息后，检查其 `messageType` 是否为 `.custom` 以及 `businessId` 是否匹配。如果匹配，则解析 `data` 字符串（通常是解析JSON），根据解析出的数据（例如 `color`、`text`）来渲染您的自定义UI样式。

我在不同的类、不同的文件中都调用了 `BarrageStore.create(liveID: "some_id")`，这会创建出多个实例导致混乱吗？

完全不会。`AtomicXCore` 内部机制会确保只要您传入的 `liveId` 相同，获取到的永远是同一个与该直播间绑定的 `BarrageStore` 实例。您可以在需要的地方随用随取，无需手动管理单例。

为什么我调用了 sendTextMessage，但是在消息列表中看不到我发送的消息？

请按以下步骤排查：

1. **检查 completion 回调**：sendTextMessage 方法有一个完成回调。请检查回调返回的结果是成功还是失败。如果失败，错误信息会明确指出问题所在（例如“您已被禁言”、“网络错误”等）。
2. **确认订阅时机**：确保您对 barrageStore.state 的订阅发生在该 liveId 对应的直播开始之后。如果在加入直播房间之前就开始监听，可能会错过部分消息。
3. **检查 liveId**：确认您在创建 BarrageStore 实例、加入直播房间、以及发送消息时使用的 liveId 完全一致，包括大小写。
4. **网络问题**：检查设备当前的网络连接是否正常。消息发送依赖于网络。

新观众进入直播间时，如何让他们看到加入前的历史弹幕消息？

`AtomicXCore` 支持拉取历史弹幕消息，但这需要您在**服务端控制台**进行一项简单的配置。配置完成后，SDK 会自动处理后续的一切，您无需编写额外的代码。

步骤1：在 IM 控制台进行配置

1. 登录您的 即时通讯 IM 控制台 。
2. 在左侧导航栏按照路径：**消息服务 Chat > 功能配置 > 群组配置 > 群功能配置 > 直播群新成员查看入群前消息量配置**进行导航。

3. 修改“新成员可查看最近消息数”，最大支持 50 条。

步骤2：客户端无感获取

完成上述配置后，您的客户端代码**无需做任何改动**。

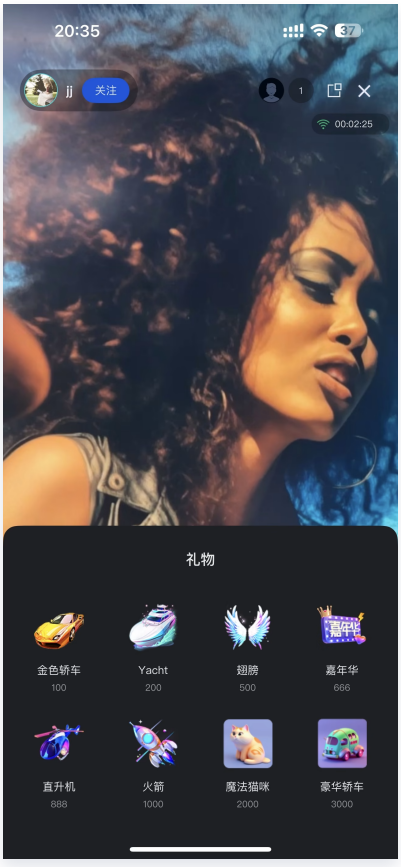
当新用户加入直播间时， `AtomicXCore` 的底层 会自动拉取您配置的历史消息数量。这些历史消息会和实时消息一样，通过您已实现的 `BarrageStore.state` 订阅通道推送给您的UI层。您的应用会像接收实时弹幕一样，自然地接收并展示这些历史弹幕。

礼物

礼物（Android）

最近更新时间：2025-11-21 14:15:34

`GiftStore` 是 `AtomicXCore` 中专门负责管理直播间礼物功能的模块。通过它，您可以为您的直播应用构建一套完整的礼物系统，实现丰富的营收和互动场景。

礼物面板	弹幕礼物	全屏礼物
		

核心功能

- **礼物列表获取**：从服务端拉取礼物面板所需的数据，包括礼物分类和礼物详情。
- **发送礼物**：观众可以向主播发送选定的礼物，并附带数量。
- **礼物事件广播**：实时接收房间内发生的礼物赠送事件，用于展示礼物动画和弹幕通知。

核心概念

在开始集成之前，我们先通过下表了解一下 `GiftStore` 相关的几个核心概念：

核心概念	类型	核心职责与描述
`Gift`	`data class`	代表一个具体的礼物数据模型。包含了礼物的 ID、名称、图标地址、动画资源地址 (resourceURL) 和价格 (coins) 等。
`GiftCategory`	`data class`	代表一个礼物分类，例如“热门”、“豪华”等。它包含了分类的名称以及该分类下的 Gift 列表。
`GiftState`	`data class`	代表礼物模块的当前状态。其核心属性 usableGifts 是一个 `StateFlow`，存储了从服务端拉取到的完整礼物列表。
`GiftEvent`	`abstract class`	代表直播间内发生的礼物事件监听器。目前只有 onReceiveGift，当有任何人（包括自己）发送礼物时，房间内所有人都会收到此事件。
`GiftStore`	`abstract class`	这是与礼物功能交互的核心管理类。通过它，您可以拉取礼物列表、发送礼物，并通过添加监听器来接收礼物事件。

实现步骤

步骤1：组件集成

请参考 开始直播 集成 AtomicXCore，完成接入。

步骤2：初始化并监听礼物事件

获取 GiftStore 实例，并设置监听器以接收礼物事件和礼物列表更新。

实现方式:

1. **获取实例**：使用 `GiftStore.create(liveID)` 获取与当前直播间绑定的 `GiftStore` 实例。
2. **添加监听器**：添加 `GiftListener` 以接收 `onReceiveGift` 事件。
3. **订阅状态**：订阅 `giftStore.giftState.usableGifts` 以获取礼物列表。

代码示例:

```
import io.trtc.tuikit.atomicxcore.api.gift.Gift
import io.trtc.tuikit.atomicxcore.api.gift.GiftCategory
import io.trtc.tuikit.atomicxcore.api.gift.GiftListener
import io.trtc.tuikit.atomicxcore.api.gift.GiftStore
import io.trtc.tuikit.atomicxcore.api.live.LiveUserInfo
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.asStateFlow
```

```
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

class GiftManager(
    private val liveId: String
) {
    private val giftStore: GiftStore = GiftStore.create(liveId)
    private val coroutineScope = CoroutineScope(Dispatchers.Main)
    // 对外暴露礼物列表
    private val _giftList = MutableStateFlow<List<GiftCategory>>
(emptyList())
    val giftList: StateFlow<List<GiftCategory>> =
_giftList.asStateFlow()

    init {
        setupGiftListener()
        subscribeToGiftState()
    }

    /// 2. 订阅礼物事件
    private fun setupGiftListener() {
        giftStore.addGiftListener(object : GiftListener() {
            override fun onReceiveGift(liveID: String, gift: Gift,
count: Int, sender: LiveUserInfo) {
                // 收到事件后，将其转发给UI层处理
            }
        })
    }

    /// 3. 订阅礼物列表状态
    private fun subscribeToGiftState() {
        coroutineScope.launch(Dispatchers.Main) {
            giftStore.giftState.usableGifts.collect { giftList ->
                // 仅在列表内容变化时才更新
                _giftList.value = giftList
            }
        }
    }
```

```
}
```

礼物列表结构体参数

- `GiftCategory` 参数说明

参数	类型	描述
`categoryID`	`String`	礼物分类的唯一 ID。
`name`	`String`	礼物分类的显示名称。
`desc`	`String`	礼物分类的描述信息。
`extensionInfo`	`Map<String, String>`	扩展信息字段。
`giftList`	`List`	该分类下包含的 Gift 礼物对象数组。

- `Gift` 参数说明

参数	类型	描述
`giftID`	`String`	礼物的唯一 ID。
`name`	`String`	礼物的显示名称。
`desc`	`String`	礼物的描述信息。
`iconURL`	`String`	礼物图标 URL。
`resourceURL`	`String`	礼物动画资源 URL。
`level`	`Long`	礼物等级。
`coins`	`Long`	礼物价格。
`extensionInfo`	`Map<String, String>`	扩展信息字段。

步骤3：获取礼物列表

调用 `refreshUsableGifts` 方法，从服务端拉取礼物数据。

实现方式:

1. **调用接口：**在合适的时机（如进入直播间后）调用 `giftStore.refreshUsableGifts()` 。

2. **处理回调**：可选地处理 completion 回调以获知拉取结果。
3. **接收数据**：拉取成功后，通过步骤一中对 giftStore.giftState.usableGifts 的订阅，自动接收到更新后的 usableGifts 列表。

代码示例:

```
class GiftManager(
    private val liveId: String
) {
    private val giftStore: GiftStore = GiftStore.create(liveId)

    /// 从服务端刷新礼物列表
    fun fetchGiftList() {
        giftStore.refreshUsableGifts(object : CompletionHandler {
            override fun onSuccess() {
                println("礼物列表拉取成功")
                // 成功后，数据会通过 giftState 订阅通道自动更新
            }

            override fun onFailure(code: Int, desc: String) {
                println("礼物列表拉取失败: $desc")
            }
        })
    }
}
```

步骤4：发送礼物

当用户在礼物面板选择一个礼物并点击发送时，调用 `sendGift` 接口将礼物发送出去。

实现方式:

1. **获取参数**：从UI获取用户选择的 giftID 和发送数量 count。
2. **调用接口**：调用 `giftStore.sendGift(giftID, count, completion)` 。
3. **处理回调**：在 `completion` 回调中处理发送失败的情况（例如余额不足提示）；发送成功后的 UI 更新（动画、弹幕）应由 `onReceiveGift` 事件驱动。

代码示例:

```
class GiftManager(
    private val liveId: String
```

```
) {
    private val giftStore: GiftStore = GiftStore.create(liveId)

    /// 用户发送一个礼物
    fun sendGift(giftID: String, count: Int) {
        giftStore.sendGift(giftID, count, object : CompletionHandler {
            override fun onSuccess() {
                println("礼物 $giftID 发送成功")
                // 发送成功后，包括发送者在内的所有人都会收到 onReceiveGift 事件
            }

            override fun onFailure(code: Int, desc: String) {
                println("礼物发送失败: $desc")
                // 可以在此提示用户，例如"余额不足"或"网络错误"
            }
        })
    }
}
```

`sendGift` 接口参数

参数名	类型	描述
`giftID`	`String`	要发送的礼物的唯一 ID。
`count`	`Int`	发送的数量。
`completion`	`CompletionHandler?`	发送完成后的回调。

功能进阶

`GiftStore` 的功能高度依赖于您的业务后台服务。本章将指导您如何通过服务端配置和客户端实现，构建功能丰富、体验卓越的礼物互动系统。

礼物素材配置

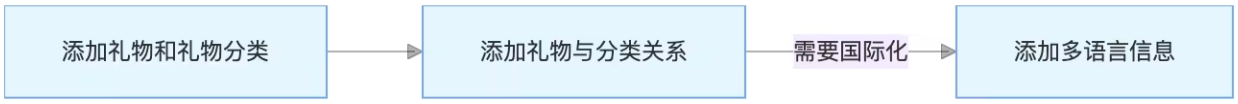
您需要自定义直播间可用的礼物种类、分类、名称、图标、价格以及动画效果，以满足运营需求和品牌特色。

实现方式

1. **服务端配置**：使用 LiveKit 服务端 REST API 管理礼物信息、分类、多语言等。请参考 礼物配置指引文档 。
2. **客户端拉取**：在客户端调用 `refreshUsableGifts` 获取配置数据。

3. UI 展示：使用拉取到的 `List<GiftCategory>` 数据填充礼物面板。

配置时序图



涉及 REST API 接口一览

接口分类	接口	请求示例
礼物管理	添加礼物信息	请求示例
	删除礼物信息	请求示例
	查询礼物信息	请求示例
礼物分类管理	添加礼物分类信息	请求示例
	删除指定礼物分类信息	请求示例
	获取指定礼物分类信息	请求示例
礼物关系管理	添加指定礼物分类和礼物间的关系	请求示例
	删除指定礼物分类和礼物间的关系	请求示例
	获取指定礼物分类下的礼物关系	请求示例
礼物多语言管理	添加礼物多语言信息	请求示例
	删除指定礼物多语言信息	请求示例
	获取礼物多语言信息	请求示例
	添加礼物分类多语言信息	请求示例
	删除指定礼物分类多语言信息	请求示例
	获取礼物分类多语言信息	请求示例

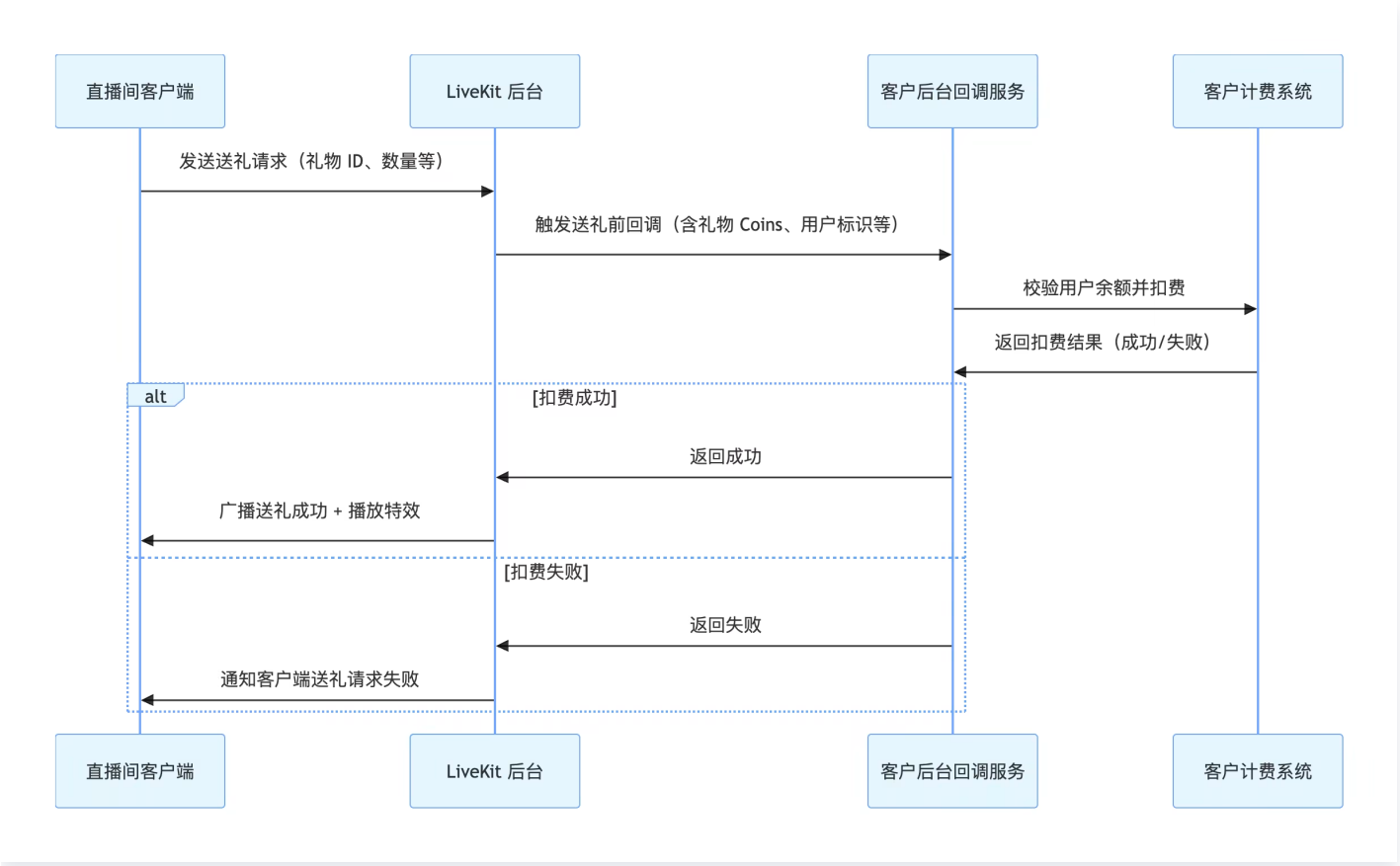
计费与送礼扣费流程

当观众赠送礼物时，需要确保其账户余额充足，并完成实际的扣费操作，然后才能触发礼物特效的播放和广播。

实现方式

1. **后台配置回调**：在 LiveKit 后台配置您的自建计费系统的回调 URL。参考 回调配置文档 。
2. **客户端发送**：客户端调用 `sendGift` 。
3. **后台交互**：LiveKit 后台调用您的回调 URL，您的计费系统执行扣费并返回结果。
4. **结果同步**：扣费成功，AtomicXCore 广播 `onReceiveGift` 事件；扣费失败， `sendGift` 的 `completion` 收到错误。

调用流程图



涉及 REST API 接口一览

接口	说明	请求示例
回调配置 – 发送礼物之前回调	客户后台可以通过该回调决定是否通过送礼前校验等场景	调用示例

实现全屏礼物动画播放

当直播间有用户（包括自己）发送了"火箭"、"嘉年华"等豪华礼物时，全屏播放一个酷炫的礼物动画（例如 SVGA 动画），营造热烈的氛围。

实现方式

`AtomicXCore` 本身不包含礼物动画播放器，您需要根据业务需求选择并集成第三方库。以下是两种方案的对比：

对比项	基础方案 (SVGAPlayer-Android)	高级方案 (TCEffectPlayerKit)
计费	免费 (开源库)	付费 (需购买 License)，请参见 付费指引
集成方式	需手动集成 SVGAPlayer 库 (例如通过 Gradle)	需额外集成 TCEffectPlayerKit 并进行鉴权
动画格式支持	仅支持 SVGA	SVGA、PAG、WebP、Lottie、MP4 等多种格式
性能	建议 SVGA 文件 ≤ 10MB	支持更大的动画文件，复杂特效/多动画并发/低端机表现更优
推荐场景	礼物动画格式统一为 SVGA，且文件大小可控	需要支持多种动画格式，或对动画性能/设备兼容性有更高要求

本章节主要演示基础方案的集成。如果您选择高级方案，请参见 TCEffectPlayer 集成指引 。

基础方案实现：使用 SVGAPlayer

1. **集成 SVGAPlayer:** 在您的 `build.gradle` 文件中，添加 `SVGAPlayer` 的依赖并同步项目。

```
dependencies {
    // ... other dependencies
    implementation 'com.github.yyued:SVGAPlayer-Android:2.6.1'
}
```

2. **监听礼物事件:** 订阅 `GiftStore` 的 `GiftListener` 。
3. **解析并播放:** 当收到 `onReceiveGift` 事件时，检查 `gift.resourceURL` 是否有效且指向一个 SVGA 文件。如果是，则使用 `SVGAParser` 解析 URL，并将解析后的 `SVGAVideoEntity` 交给 `SVGAPlayer` 实例播放。

代码示例

```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import com.opensource.svgaplayer.SVGAParser
import com.opensource.svgaplayer.SVGAVideoEntity
import com.opensource.svgaplayer.SVGAImageView
import io.trtc.tuikit.atomicxcore.api.gift.Gift
```

```
import io.trtc.tuikit.atomicxcore.api.gift.GiftListener
import io.trtc.tuikit.atomicxcore.api.gift.GiftStore
import io.trtc.tuikit.atomicxcore.api.live.LiveUserInfo
import java.net.URL

class LiveRoomActivity : AppCompatActivity() {
    private lateinit var giftStore: GiftStore
    private lateinit var svgaImageView: SVGAImageView
    private lateinit var svgaParser: SVGAParser

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_live_room)
        setupSVGAPlayer()
    }


    private fun setupSVGAPlayer() {
        svgaImageView = findViewById(R.id.svga_player)
        svgaParser = SVGAParser(this)
        svgaImageView.visibility = android.view.View.GONE
    }

    fun setupGiftSubscription(liveId: String) {
        giftStore = GiftStore.create(liveId)

        giftStore.addGiftListener(object : GiftListener() {
            override fun onReceiveGift(liveID: String, gift: Gift,
count: Int, sender: LiveUserInfo) {
                if (gift.resourceURL.isNotEmpty()) {
                    try {
                        val url = URL(gift.resourceURL)
                        playAnimation(url)
                    } catch (e: Exception) {
                        println("无效的动画URL: ${gift.resourceURL}")
                    }
                }
            }
        })
    }
```

```
    private fun playAnimation(url: URL) {
        // 4. 解析播放
        svgaParser.decodeFromURL(url, object :
SVGAParser.ParseCompletion {
            override fun onComplete(videoItem: SVGAVideoEntity) {
                runOnUiThread {
                    svgaImageView.setVideoItem(videoItem)
                    svgaImageView.visibility = android.view.View.VISIBLE
                    svgaImageView.startAnimation()
                }
            }

            override fun onError() {
                println("SVGA 动画解析失败")
                // 可以在此添加失败重试或上报逻辑
            }
        })
    }
}
```

在弹幕区展示礼物赠送消息

当有用户发送礼物时，不仅播放动画，同时在公屏弹幕区域显示一条系统消息，例如：“【观众昵称】送出了【礼物名称】x【数量】”，让所有观众都能看到。

实现方式

1. **监听事件：**订阅 `giftStore` 的 `GiftListener`。
2. **获取信息：**收到 `onReceiveGift` 事件后，提取发送者 sender、礼物 gift 和数量 count。
3. **获取弹幕 Store：**使用 `BarrageStore.create(liveID)` 获取与当前房间绑定的实例。
4. **拼接消息：**创建一条 `Barrage` 结构体，设置 `messageType = BarrageType.TEXT`，textContent 为拼接好的字符串（例如 “[`sender.userName`] 送出了 [`gift.name`] x [`count`]”）。
5. **本地插入：**调用 `barrageStore.appendLocalTip(message: giftTip)` 将消息插入本地列表。

代码示例

```
import io.trtc.tuikit.atomicxcore.api.barrage.Barrage
import io.trtc.tuikit.atomicxcore.api.barrage.BarrageStore
```

```
import io.trtc.tuikit.atomicxcore.api.barrage.BarrageType
import io.trtc.tuikit.atomicxcore.api.gift.Gift
import io.trtc.tuikit.atomicxcore.api.gift.GiftListener
import io.trtc.tuikit.atomicxcore.api.gift.GiftStore
import io.trtc.tuikit.atomicxcore.api.live.LiveUserInfo

// 在 LiveRoomManager 或类似的顶层管理器中
class LiveRoomManager(
    private val liveId: String
) {
    private val giftStore: GiftStore = GiftStore.create(liveId)
    private val barrageStore: BarrageStore = BarrageStore.create(liveId)

    private fun setupGiftToBarrageFlow() {
        giftStore.addGiftListener(object : GiftListener() {
            override fun onReceiveGift(liveID: String, gift: Gift,
count: Int, sender: LiveUserInfo) {
                // 1. 监听事件 2. 获取信息

                // 4. 拼接消息
                val tipText = "${sender.userName} 送出了 ${gift.name} x
$count"
                val giftTip = Barrage(
                    liveID = liveID,
                    messageType = BarrageType.TEXT,
                    textContent = tipText
                    // 可选：设置一个特殊的sender
                )

                // 5. 本地插入
                barrageStore.appendLocalTip(giftTip)
            }
        })
    }
}
```

API 文档

关于 GiftStore 及其相关类的所有公开接口、属性和方法的详细信息，请参阅随 AtomicXCore 框架的官方 API 文档。本指南使用到的相关 Store 如下：

Store/Component	功能描述	API 文档
GiftStore	礼物互动：获取礼物列表，发送 / 接收礼物，监听礼物事件（含发送者、礼物详情）。	API 文档
BarrageStore	弹幕功能：发送文本 / 自定义弹幕，维护弹幕列表，实时监听弹幕状态。	API 文档

常见问题

`GiftStore` 的礼物列表是空的，我该怎么办？

您必须主动调用 `refreshUsableGifts(completion)` 来从您的业务后台拉取礼物数据。这些礼物数据需要在您的业务后台通过服务端 REST API 进行配置。

如何实现礼物的多语言展示（例如中文、英文）？

`GiftStore` 提供了 `setLanguage(language: String)` 接口。您可以在 `refreshUsableGifts` 之前调用此方法，传入目标语言代码（例如 "en" 或 "zh-CN"）。服务端会根据此语言代码，返回对应语言的礼物名称和描述。

我调用 sendGift 发送了礼物，礼物动画为什么重复播放了两次？

`onReceiveGift` 事件是对房间内所有成员的广播（包括发送者自己）。如果您在 `sendGift` 的 `completion` 成功回调里执行了一次UI操作，同时又在 `onReceiveGift` 订阅中执行了相同的UI操作，就会造成重复。

- **实践教程**：UI 的更新（例如播放动画、弹幕提示）只在 `onReceiveGift` 事件的订阅中处理。`sendGift` 的 `completion` 回调仅用于处理发送失败的逻辑（例如提示用户"发送失败"或"余额不足"）。

礼物扣费逻辑在哪里实现？

礼物扣费逻辑完全由您的自建计费系统负责。`AtomicXCore` 通过后台回调机制与您的计费系统对接。客户端调用 `sendGift` 触发回调，您的后台服务完成扣费后，将结果返回给 `AtomicXCore` 后台，从而决定礼物事件是否广播。

送礼通知会被禁言或频控拦截吗？

不会。送礼通知（`onReceiveGift` 事件）不受禁言或消息频控影响，确保可靠投递。

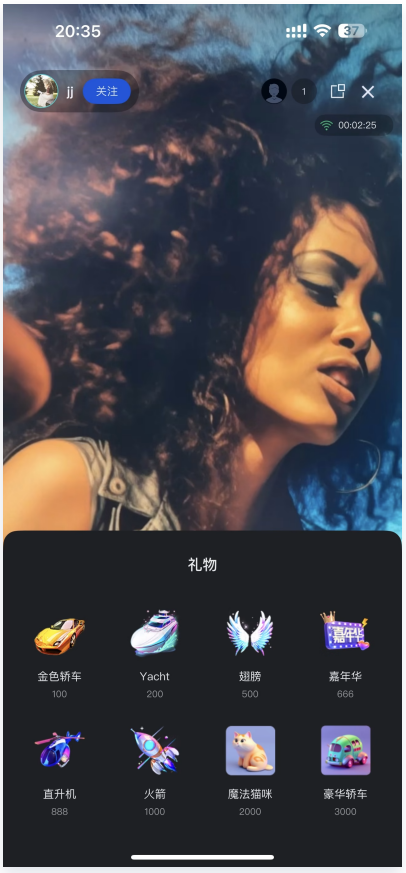
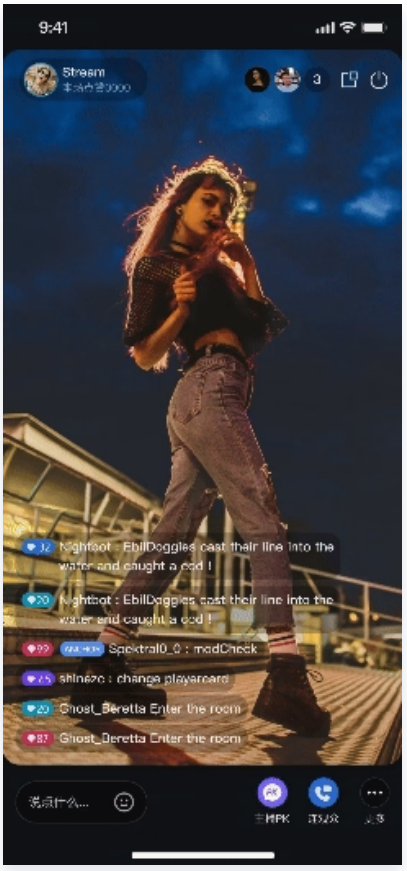
礼物动画播放卡顿怎么办？

请检查您的 SVGA 文件大小，基础播放器建议不要超过 10MB。如果文件过大或动画复杂，您可以考虑集成 TUILiveKit 提供的高级特效播放器，以获得更优的性能表现。

礼物（iOS）

最近更新时间：2025-11-21 14:15:34

`GiftStore` 是 `AtomicXCore` 中专门负责管理直播间礼物功能的模块。通过它，您可以为您的直播应用构建一套完整的礼物系统，实现丰富的营收和互动场景。

礼物面板	弹幕礼物	全屏礼物
		

核心功能

- **礼物列表获取**：从服务端拉取礼物面板所需的数据，包括礼物分类和礼物详情。
- **发送礼物**：观众可以向主播发送选定的礼物，并附带数量。
- **礼物事件广播**：实时接收房间内发生的礼物赠送事件，用于展示礼物动画和弹幕通知。

核心概念

在开始集成之前，我们先通过下表了解一下 `GiftStore` 相关的几个核心概念：

核心概念	类型	核心职责与描述

`Gift`	`struct`	代表一个具体的礼物数据模型。包含了礼物的 ID、名称、图标地址、动画资源地址 (resourceURL) 和价格 (coins) 等。
`GiftCategory`	`struct`	代表一个礼物分类，例如“热门”、“豪华”等。它包含了分类的名称以及该分类下的 Gift 列表。
`GiftState`	`struct`	代表礼物模块的当前状态。其核心属性 usableGifts 是一个 [GiftCategory] 数组，存储了从服务端拉取到的完整礼物列表。
`GiftEvent`	`enum`	代表直播间内发生的礼物事件。目前只有 .onReceiveGift，当有任何人（包括自己）发送礼物时，房间内所有人都会收到此事件。
`GiftStore`	`class`	这是与礼物功能交互的核心管理类。通过它，您可以拉取礼物列表、发送礼物，并通过订阅其 giftEventPublisher 来接收礼物事件。

实现步骤

步骤1：组件集成

请参考 开始直播 集成 AtomicXCore，完成接入。

步骤2：初始化并监听礼物事件

获取 GiftStore 实例，并设置订阅者以接收礼物事件和礼物列表更新。

实现方式:

1. **获取实例**：使用 `GiftStore.create(liveID:)` 获取与当前直播间绑定的 `GiftStore` 实例。
2. **订阅事件**：订阅 `giftStore.giftEventPublisher` 以接收 `.onReceiveGift` 事件。
3. **订阅状态**：订阅 `giftStore.state` 以获取 `usableGifts` 礼物列表。

代码示例:

```
import Foundation
import AtomicXCore
import Combine

class GiftManager {
    private let liveId: String
    private let giftStore: GiftStore
    private var cancellables = Set<AnyCancellable>()

    // 对外暴露礼物事件，方便UI层订阅以播放动画
```

```
    let giftEventPublisher = PassthroughSubject<GiftEvent, Never>()
    // 对外暴露礼物列表
    let giftListPublisher = CurrentValueSubject<[GiftCategory], Never>
([])

    init(liveId: String) {
        self.liveId = liveId
        // 1. 通过 liveId 获取 GiftStore 的实例
        self.giftStore = GiftStore.create(liveID: liveId)

        subscribeToGiftState()
        subscribeToGiftEvents()
    }

    /// 2. 订阅礼物事件
    private func subscribeToGiftEvents() {
        giftStore.giftEventPublisher
            .receive(on: DispatchQueue.main)
            .sink { [weak self] event in
                // 收到事件后，将其转发给UI层处理
                self?.giftEventPublisher.send(event)
            }
            .store(in: &cancellables)
    }

    /// 3. 订阅礼物列表状态
    private func subscribeToGiftState() {
        giftStore.state
            .subscribe(StatePublisherSelector(keyPath:
\GiftState.usableGifts))
            .receive(on: DispatchQueue.main)
            .assign(to: \.value, on: giftListPublisher)
            .store(in: &cancellables)
    }
}
```

礼物列表结构体参数

- `GiftCategory` 参数说明

参数	类型	描述
`categoryID`	`String`	礼物分类的唯一 ID。
`name`	`String`	礼物分类的显示名称。
`desc`	`String`	礼物分类的描述信息。
`extensionInfo`	`[String: String]`	扩展信息字段。
`giftList`	`[ Gift ]`	该分类下包含的 Gift 礼物对象数组。

- **`Gift` 参数说明**

参数	类型	描述
`giftID`	`String`	礼物的唯一 ID。
`name`	`String`	礼物的显示名称。
`desc`	`String`	礼物的描述信息。
`iconURL`	`String`	礼物图标 URL。
`resourceURL`	`String`	礼物动画资源 URL。
`level`	`UInt`	礼物等级。
`coins`	`UInt`	礼物价格。
`extensionInfo`	`[String: String]`	扩展信息字段。

步骤3：获取礼物列表

调用 `refreshUsableGifts` 方法，从服务端拉取礼物数据。

实现方式:

1. **调用接口**：在合适的时机（例如进入直播间后）调用 `giftStore.refreshUsableGifts(completion:)` 。
2. **处理回调**：可选地处理 completion 回调以获知拉取结果。
3. **接收数据**：拉取成功后，通过步骤2中对 giftStore.state 的订阅，自动接收到更新后的 usableGifts 列表。

代码示例:

```
extension GiftManager {
    /// 从服务端刷新礼物列表
    func fetchGiftList() {
        giftStore.refreshUsableGifts { result in
            switch result {
            case .success:
                print("礼物列表拉取成功")
                // 成功后，数据会通过 state 订阅通道自动更新
            case .failure(let error):
                print("礼物列表拉取失败: \(error.localizedDescription)")
            }
        }
    }
}
```

步骤4：发送礼物

当用户在礼物面板选择一个礼物并点击发送时，调用 `sendGift` 接口将礼物发送出去。

实现方式:

1. **获取参数**：从 UI 获取用户选择的 giftID 和发送数量 count。
2. **调用接口**：调用 `giftStore.sendGift(giftID:count:completion:)` 。
3. **处理回调**：在 `completion` 回调中处理发送失败的情况（例如余额不足提示）；发送成功后的 UI 更新（动画、弹幕）应由 `.onReceiveGift` 事件驱动。

代码示例:

```
extension GiftManager {
    /// 用户发送一个礼物
    func sendGift(giftID: String, count: UInt) {
        giftStore.sendGift(giftID: giftID, count: count) { result in
            switch result {
            case .success:
                print("礼物 \(giftID) 发送成功")
                // 发送成功后，包括发送者在内的所有人都会收到 onReceiveGift 事件
            case .failure(let error):
                print("礼物发送失败: \(error.localizedDescription)")
                // 可以在此提示用户，例如“余额不足”或“网络错误”
            }
```

```
            }
        }
    }
```

`sendGift` 接口参数

参数名	类型	描述
`giftID`	`String`	要发送的礼物的唯一 ID。
`count`	`UInt`	发送的数量。
`completion`	`CompletionClosure?`	发送完成后的回调。

功能进阶

`GiftStore` 的功能高度依赖于您的业务后台服务。本章将指导您如何通过服务端配置和客户端实现，构建功能丰富、体验卓越的礼物互动系统。

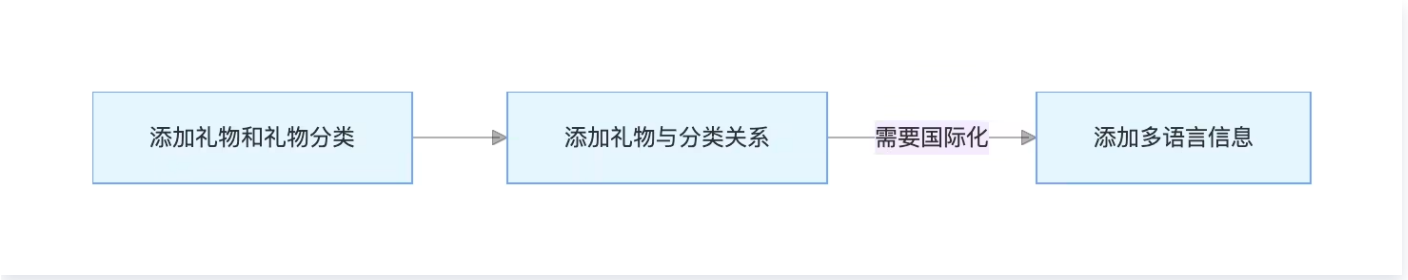
礼物素材配置

您需要自定义直播间可用的礼物种类、分类、名称、图标、价格以及动画效果，以满足运营需求和品牌特色。

实现方式

1. **服务端配置**：使用 LiveKit 服务端 REST API 管理礼物信息、分类、多语言等。请参考 礼物配置指引文档 。
2. **客户端拉取**：在客户端调用 `refreshUsableGifts` 获取配置数据。
3. **UI 展示**：使用拉取到的 `[GiftCategory]` 数据填充礼物面板。

配置时序图



涉及 REST API 接口一览

接口分类	接口	请求示例
礼物管理	添加礼物信息	请求示例
	删除礼物信息	请求示例

	查询礼物信息	请求示例
礼物分类管理	添加礼物分类信息	请求示例
	删除指定礼物分类信息	请求示例
	获取指定礼物分类信息	请求示例
礼物关系管理	添加指定礼物分类和礼物间的关系	请求示例
	删除指定礼物分类和礼物间的关系	请求示例
	获取指定礼物分类下的礼物关系	请求示例
礼物多语言管理	添加礼物多语言信息	请求示例
	删除指定礼物多语言信息	请求示例
	获取礼物多语言信息	请求示例
	添加礼物分类多语言信息	请求示例
	删除指定礼物分类多语言信息	请求示例
	获取礼物分类多语言信息	请求示例

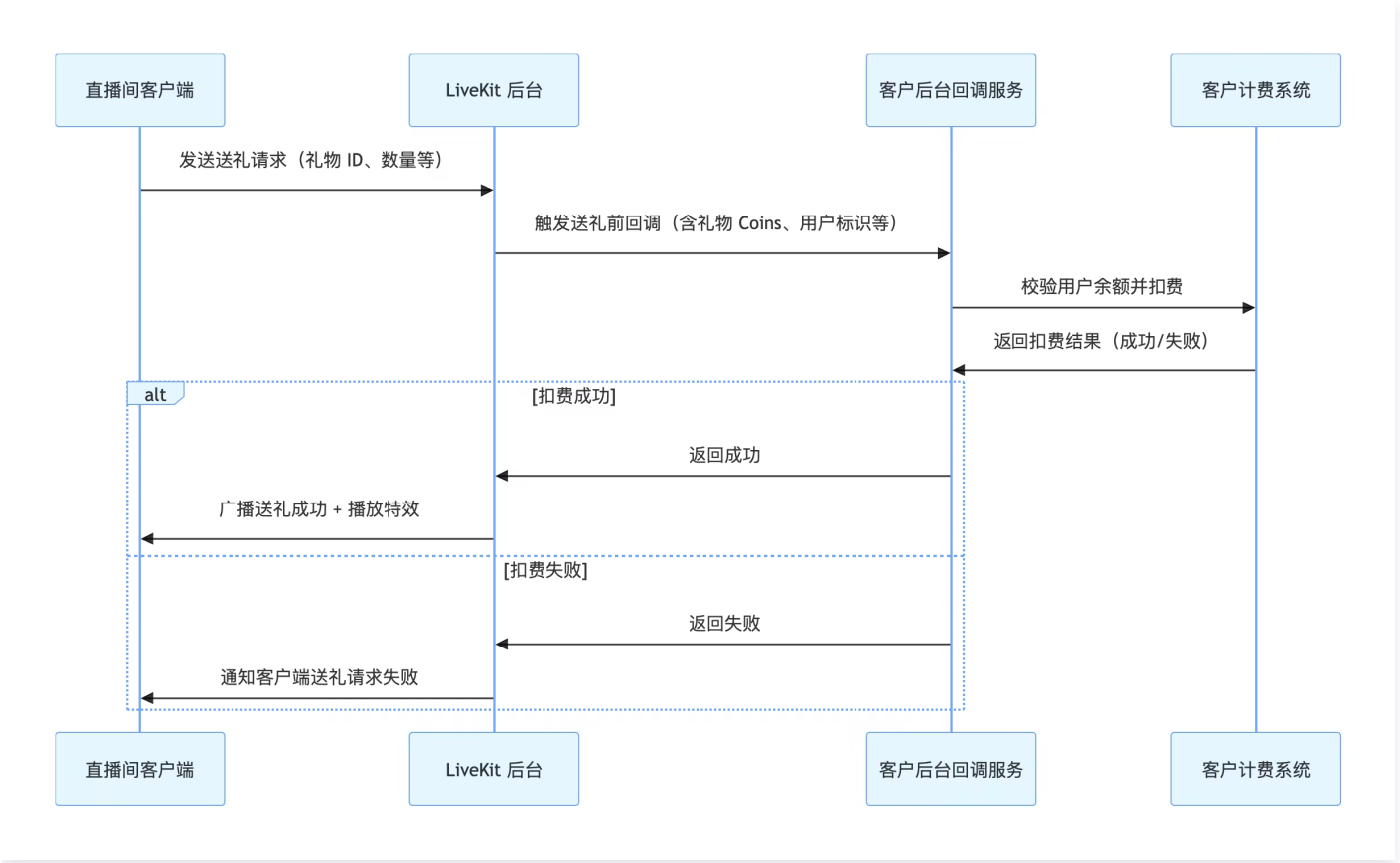
计费与送礼扣费流程

当观众赠送礼物时，需要确保其账户余额充足，并完成实际的扣费操作，然后才能触发礼物特效的播放和广播。

实现方式

1. **后台配置回调**：在 LiveKit 后台配置您的自建计费系统的回调 URL。参考 回调配置文档 。
2. **客户端发送**：客户端调用 `sendGift` 。
3. **后台交互**：LiveKit 后台调用您的回调 URL，您的计费系统执行扣费并返回结果。
4. **结果同步**：扣费成功，AtomicXCore 广播 `.onReceiveGift` 事件。扣费失败， `sendGift` 的 `completion` 收到错误。

调用流程图



涉及 REST API 接口一览

接口	说明	请求示例
回调配置 – 发送礼物之前回调	客户后台可以通过该回调决定是否通过送礼前校验等场景	调用示例

实现全屏礼物动画播放

当直播间有用户（包括自己）发送了“火箭”、“嘉年华”等豪华礼物时，全屏播放一个酷炫的礼物动画（例如 SVGA 动画），营造热烈的氛围。

实现方式

`AtomicXCore` 本身不包含礼物动画播放器，您需要根据业务需求选择并集成第三方库。以下是两种方案的对比：

对比项	基础方案 (SVGAPlayer-iOS)	高级方案 (TCEffectPlayerKit)
计费	免费 (开源库)	付费 (需购买 License)，请参见 付费指引
集成方式	需手动集成 SVGAPlayer 库 (例如通过 CocoaPods)	需额外集成 TCEffectPlayerKit 并进行鉴权
动画格式支	仅支持 SVGA	SVGA、PAG、WebP、Lottie、MP4 等

持		多种格式
性能	建议 SVGA 文件 ≤ 10MB	支持更大的动画文件，复杂特效/多动画并发/低端机表现更优
推荐场景	礼物动画格式统一为 SVGA，且文件大小可控	需要支持多种动画格式，或对动画性能/设备兼容性有更高要求

本章节主要演示基础方案的集成。如果您选择高级方案，请参考 TCEffectPlayer 集成指引 。

基础方案实现：使用 SVGAPlayer

1. **集成 SVGAPlayer**：在您的 `Podfile` 文件中，添加 `SVGAPlayer` 的依赖并在终端中运行 `pod install` 。

```
target 'YourApp' do
  # ... other pods
  pod 'SVGAPlayer'
end
```

2. **监听礼物事件**：订阅 `GiftStore` 的 `giftEventPublisher` 。
3. **解析并播放**：当收到 `.onReceiveGift` 事件时，检查 `gift.resourceURL` 是否有效且指向一个 SVGA 文件。如果是，则使用 `SVGAParser` 解析 URL，并将解析后的 `SVGAVideoEntity` 交给 `SVGAPlayer` 实例播放。

代码示例

```
import UIKit
import AtomicXCore
import Combine
import SVGAPlayer // 1. 导入

class LiveRoomViewController: UIViewController, SVGAPlayerDelegate {
    private var giftManager: GiftManager!
    private var cancellables = Set<AnyCancellable>()
    private let svgaPlayer = SVGAPlayer() // 2. 准备实例
    private let svgaParser = SVGAParser()

    override func viewDidLoad() {
        super.viewDidLoad()
        setupSVGAPlayer()
    }
```

```
    private func setupSVGAPlayer() {
        svgaPlayer.delegate = self
        svgaPlayer.loops = 1 // 默认播放1次
        svgaPlayer.clearsAfterStop = true // 播放完毕后自动清除
        view.addSubview(svgaPlayer)
        svgaPlayer.frame = view.bounds // 默认全屏
        svgaPlayer.isHidden = true
    }

    func setupGiftSubscription(liveId: String) {
        self.giftManager = GiftManager(liveId: liveId)

        giftManager.giftEventPublisher
            .sink { [weak self] event in // 3. 监听事件
                guard case .onReceiveGift(_, let gift, _, _) = event
else { return }

                if !gift.resourceURL.isEmpty, let url = URL(string:
gift.resourceURL) { // 4. 解析播放
                    self?.playAnimation(from: url)
                }
            }
            .store(in: &cancellables)
    }

    private func playAnimation(from url: URL) {
        svgaParser.parse(with: url, completionBlock: { [weak self]
videoItem in
            guard let self = self, let videoItem = videoItem else {
return }
            DispatchQueue.main.async {
                self.svgaPlayer.videoItem = videoItem
                self.svgaPlayer.isHidden = false
                self.svgaPlayer.startAnimation()
            }
        }, failureBlock: { error in
            print("SVGA 动画解析失败: \(error?.localizedDescription ??
"unknown error")")
            // 可以在此添加失败重试或上报逻辑
```

```
        })
    }

    func svgaPlayerDidFinishedAnimation(_ player: SVGAPlayer!) { /* ...
播放结束处理 ... */ }
}
```

在弹幕区展示礼物赠送消息

当有用户发送礼物时，不仅播放动画，同时在公屏弹幕区域显示一条系统消息，例如：“【观众昵称】送出了【礼物名称】x【数量】”，让所有观众都能看到。

实现方式

1. **监听事件**：订阅 `giftStore.giftEventPublisher`。
2. **获取信息**：收到 `.onReceiveGift` 事件后，提取发送者 sender、礼物 gift 和数量 count。
3. **获取弹幕 Store**：使用 `BarrageStore.create(liveID:)` 获取与当前房间绑定的实例。
4. **拼接消息**：创建一条 `Barrage` 结构体，设置 `messageType = .text`，textContent 为拼接好的字符串（例如 “[`sender.userName`] 送出了 [`gift.name`] x [`count`]”）。
5. **本地插入**：调用 `barrageStore.appendLocalTip(message: giftTip)` 将消息插入本地列表。

代码示例

```
// 在 LiveRoomManager 或类似的顶层管理器中
private func setupGiftToBarrageFlow() {
    giftStore.giftEventPublisher
        .receive(on: DispatchQueue.main)
        .sink { [weak self] event in // 1. 监听事件
            guard case .onReceiveGift(_, let gift, let count, let
sender) = event else { return } // 2. 获取信息

            guard let self = self else { return }
            // 3. 获取弹幕Store（假设 barrageStore 已初始化）

            // 4. 拼接消息
            let tipText = "\(sender.userName) 送出了 \(gift.name) x \
(count)"
            var giftTip = Barrage()
            giftTip.messageType = .text
            giftTip.textContent = tipText
```

```
            // 可选：设置一个特殊的sender

            // 5. 本地插入
            self.barrageStore.appendLocalTip(message: giftTip)
        }
        .store(in: &cancellables)
}
```

API 文档

关于 GiftStore 及其相关类的所有公开接口、属性和方法的详细信息，请参阅随 AtomicXCore 框架的官方 API 文档。本指南使用到的相关 Store 如下：

Store/Component	功能描述	API 文档
GiftStore	礼物互动：获取礼物列表，发送 / 接收礼物，监听礼物事件（含发送者、礼物详情）。	API 文档
BarrageStore	弹幕功能：发送文本 / 自定义弹幕，维护弹幕列表，实时监听弹幕状态。	API 文档

常见问题

`GiftStore` 的礼物列表是空的，我该怎么办？

您必须主动调用 `refreshUsableGifts(completion:)` 来从您的业务后台拉取礼物数据。这些礼物数据需要在您的业务后台通过服务端 REST API 进行配置。

如何实现礼物的多语言展示（例如中文、英文）？

`GiftStore` 提供了 `setLanguage(_ language: String)` 接口。您可以在 `refreshUsableGifts` 之前调用此方法，传入目标语言代码（例如 "en" 或 "zh-CN"）。服务端会根据此语言代码，返回对应语言的礼物名称和描述。

我调用 sendGift 发送了礼物，礼物动画为什么重复播放了两次？

`onReceiveGift` 事件是对房间内所有成员的广播（包括发送者自己）。如果您在 `sendGift` 的 `completion` 成功回调里执行了一次UI操作，同时又在 `onReceiveGift` 订阅中执行了相同的 UI 操作，就会造成重复。

- **实践教程：** UI 的更新（例如播放动画、弹幕提示）只在 `onReceiveGift` 事件的订阅中处理。`sendGift` 的 `completion` 回调仅用于处理发送失败的逻辑（例如提示用户“发送失败”或“余额不足”）。

礼物扣费逻辑在哪里实现？

礼物扣费逻辑完全由您的自建计费系统负责。`AtomicXCore` 通过后台回调机制与您的计费系统对接。客户端调用 `sendGift` 触发回调，您的后台服务完成扣费后，将结果返回给 `AtomicXCore` 后台，从而决定礼物事件是否广播。

送礼通知会被禁言或频控拦截吗?

不会。送礼通知（`.onReceiveGift` 事件）不受禁言或消息频控影响，确保可靠投递。

礼物动画播放卡顿怎么办?

请检查您的 SVGA 文件大小，基础播放器建议不要超过 10MB。如果文件过大或动画复杂，您可以考虑集成 TUILiveKit 提供的高级特效播放器，以获得更优的性能表现。

美颜
美颜（Android）

最近更新时间：2025-11-21 14:15:34

`BaseBeautyStore` 是 `AtomicXCore` 中负责管理人像基础美颜效果的模块。通过它，您可以轻松为您的直播或通话应用添加自然的美颜效果。

核心功能

- **磨皮效果调节**：设置磨皮强度 (0-9)。
- **美白效果调节**：设置美白强度 (0-9)。
- **红润效果调节**：设置红润强度 (0-9)。
- **效果重置**：一键恢复所有美颜参数至默认值。
- **状态监听**：实时获取当前生效的美颜参数。

核心概念

在开始集成之前，我们先通过下表了解一下 BaseBeautyStore 相关的核心概念：

核心概念	类型	核心职责与描述
`BaseBeautyState`	`data class`	代表基础美颜模块的当前状态。包含了当前生效的磨皮（`smoothLevel`）、美白（`whitenessLevel`）和红润（`ruddyLevel`）的强度值。
`BaseBeautyStore`	`abstract class`	这是与基础美颜功能交互的核心管理类。它是一个全局单例（`shared`），负责所有基础美颜参数的设置、重置和状态同步。

实现步骤

步骤1：组件集成

请参考 开始直播 集成 AtomicXCore，并完成 LiveCoreView 的接入。

步骤2：获取实例并监听状态

获取 `BaseBeautyStore` 的全局单例，并设置订阅者以实时获取当前的美颜参数状态。

实现方式:

1. **获取单例:** 直接使用 `BaseBeautyStore.shared()` 获取全局唯一的 `BaseBeautyStore` 实例。
2. **订阅状态:** 订阅 `baseBeautyStore.baseBeautyState` 以实时获取 `BaseBeautyState` 的更新。

代码示例:

```
import io.trtc.tuikit.atomicxcore.api.device.BaseBeautyState
import io.trtc.tuikit.atomicxcore.api.device.BaseBeautyStore
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import kotlinx.coroutines.flow.MutableStateFlow

class BeautyManager() {
    // 1. 获取单例
    private val baseBeautyStore = BaseBeautyStore.shared()
    private val coroutineScope = CoroutineScope(Dispatchers.Main)

    private val _smoothLevel = MutableStateFlow(0f)
    private val _whitenessLevel = MutableStateFlow(0f)
    private val _ruddyLevel = MutableStateFlow(0f)

    // 对外暴露美颜状态
    val baseBeautyState = BaseBeautyState(
        smoothLevel = _smoothLevel,
        whitenessLevel = _whitenessLevel,
        ruddyLevel = _ruddyLevel
    )

    init {
        // 2. 订阅状态
        subscribeToBeautyState()
    }

    private fun subscribeToBeautyState() {
        coroutineScope.launch(Dispatchers.Main) {
            baseBeautyStore.baseBeautyState.smoothLevel.collect {
smoothLevel ->
                _smoothLevel.value = smoothLevel
            }
        }

        coroutineScope.launch(Dispatchers.Main) {
```

```
            baseBeautyStore.baseBeautyState.whitenessLevel.collect {
whitenessLevel ->
                _whitenessLevel.value = whitenessLevel
            }
        }

        coroutineScope.launch(Dispatchers.Main) {
            baseBeautyStore.baseBeautyState.ruddyLevel.collect {
ruddyLevel ->
                _ruddyLevel.value = ruddyLevel
            }
        }
    }
    // ... 后续方法
}
```

步骤3：设置美颜参数

当用户拖动美颜滑块或点击预设按钮时，调用相应的接口设置美颜强度。

实现方式:

1. **获取强度值**：从 UI 控件（例如 `SeekBar`）获取用户设定的强度值。请注意，SDK 接口接收的参数范围是 `[0, 9]`，其中 `0` 表示关闭效果，`9` 表示效果最明显。您需要将 UI 控件的值（例如 SeekBar 的 `0 - 100`）映射到 `0 - 9` 的范围。
2. **调用接口**：分别调用 `setSmoothLevel(smoothLevel:)`、`setWhitenessLevel(whitenessLevel:)`、`setRuddyLevel(ruddyLevel:)` 来设置磨皮、美白、红润的强度。

代码示例:

```
class BeautyManager() {
    private val baseBeautyStore = BaseBeautyStore.shared()

    /// 设置磨皮等级 (输入范围 0 ~ 100, 内部转换为 0 ~ 9)
    fun updateSmoothLevel(uiLevel: Int) {
        // 将 UI 的 0 ~ 100 映射到 SDK 的 0 ~ 9
        val sdkLevel = (uiLevel / 100.0f * 9.0f)
        baseBeautyStore.setSmoothLevel(sdkLevel)
        println("设置磨皮等级: UI=$uiLevel, SDK=$sdkLevel")
    }
```

```
    /// 设置美白等级（输入范围 0 ~ 100，内部转换为 0 ~ 9）
    fun updateWhitenessLevel(uiLevel: Int) {
        val sdkLevel = (uiLevel / 100.0f * 9.0f)
        baseBeautyStore.setWhitenessLevel(sdkLevel)
        println("设置美白等级：UI=$uiLevel, SDK=$sdkLevel")
    }

    /// 设置红润等级（输入范围 0 ~ 100，内部转换为 0 ~ 9）
    fun updateRuddyLevel(uiLevel: Int) {
        val sdkLevel = (uiLevel / 100.0f * 9.0f)
        baseBeautyStore.setRuddyLevel(sdkLevel)
        println("设置红润等级：UI=$uiLevel, SDK=$sdkLevel")
    }
}
```

步骤4：重置美颜效果

当用户点击"重置"或"关闭美颜"按钮时，将所有美颜参数恢复到默认值（通常是0）。

实现方式

调用接口：直接调用 baseBeautyStore.reset() 方法。

代码示例

```
class BeautyManager() {
    private val baseBeautyStore = BaseBeautyStore.shared()

    /// 重置所有基础美颜效果
    fun resetBeautyEffects() {
        baseBeautyStore.reset()
    }
}
```

功能进阶

基础美颜与高级美颜对比

AtomicXCore 也提供了高级美颜支持，以满足不同场景的需求：

对比项	基础美颜 (BaseBeautyStore)	高级美颜 (TEBeautyKit 需额外集成)
核心功能	磨皮、美白、红润	包含基础美颜，并增加 V脸、眼距、瘦鼻、3D 贴纸、滤镜、美妆等丰富效果
计费	免费 (包含在 AtomicXCore 授权内)	付费（需要额外购买腾讯特效 SDK License）
集成方式	默认内置，直接使用 `BaseBeautyStore.shared()`	需要额外集成 `TEBeautyKit` 组件并进行鉴权
推荐场景	对美颜要求不高，需要快速实现基础美颜功能的场景	对美颜效果有较高要求，需要丰富的美型、贴纸、滤镜等高级功能的场景

集成高级美颜

如果您需要使用高级美颜功能，请参考 高级美颜 文档进行集成。集成并成功鉴权 `TEBeautyKit` 后，您可以通过 `TEBeautyKit` 提供的接口来控制所有美颜效果。

API 文档

关于 BaseBeautyStore 及其相关类的所有公开接口、属性和方法的详细信息，请参阅随 AtomicXCore 框架的官方 API 文档。本指南使用到的相关 Store 如下：

Store/Component	功能描述	API 文档
BaseBeautyStore	基础美颜：调节磨皮 / 美白 / 红润（0-9 级），重置美颜状态，同步效果参数。	API 文档
DeviceStore	音视频设备控制：麦克风（开关 / 音量）、摄像头（开关 / 切换 / 画质）、屏幕共享，设备状态实时监听。	API 文档

常见问题

我设置了美颜参数，为什么没有效果?

请检查以下几点：

1. **摄像头是否开启**：必须先成功打开摄像头（例如通过 `DeviceStore.shared().openLocalCamera`），美颜效果才能应用到视频流上。
2. **是否使用了高级美颜**：如果您集成了 `TEBeautyKit`（高级美颜），请确保您使用的是 `TEBeautyKit` 提供的接口来调节美颜。
3. **参数范围**：如果您使用的是基础美颜功能，请确认您传入的强度值在有效的范围内（ `0` 到 `9` 之间的 Float 值）。

美颜（iOS）

最近更新时间：2025-11-21 14:15:34

`BaseBeautyStore` 是 `AtomicXCore` 中负责管理人像基础美颜效果的模块。通过它，您可以轻松为您的直播或通话应用添加自然的美颜效果。

核心功能

- **磨皮效果调节**：设置磨皮强度 (0-9)。
- **美白效果调节**：设置美白强度 (0-9)。
- **红润效果调节**：设置红润强度 (0-9)。
- **效果重置**：一键恢复所有美颜参数至默认值。
- **状态监听**：实时获取当前生效的美颜参数。

核心概念

在开始集成之前，我们先通过下表了解一下 BaseBeautyStore 相关的核心概念：

核心概念	类型	核心职责与描述
`BaseBeautyState`	`struct`	代表基础美颜模块的当前状态。包含了当前生效的磨皮（`smoothLevel`）、美白（`whitenessLevel`）和红润（`ruddyLevel`）的强度值。
`BaseBeautyStore`	`class`	这是与基础美颜功能交互的核心管理类。它是一个全局单例（`shared`），负责所有基础美颜参数的设置、重置和状态同步。

实现步骤

步骤1：组件集成

请参考 开始直播 集成 AtomicXCore，并完成 LiveCoreView 的接入。

步骤2：获取实例并监听状态

获取 `BaseBeautyStore` 的全局单例，并设置订阅者以实时获取当前的美颜参数状态。

实现方式

1. **获取单例**：直接使用 `BaseBeautyStore.shared` 获取全局唯一的 `BaseBeautyStore` 实例。
2. **订阅状态**：订阅 `baseBeautyStore.state` 以实时获取 `BaseBeautyState` 的更新。

代码示例

```
import Foundation
import AtomicXCore
import Combine

class BeautyManager {
    // 1. 获取单例
    private let baseBeautyStore = BaseBeautyStore.shared //
    private var cancellables = Set<AnyCancellable>()

    // 对外暴露美颜状态
    let beautyStatePublisher = CurrentValueSubject<BaseBeautyState,
Never>(BaseBeautyState())

    init() {
        // 2. 订阅状态
        subscribeToBeautyState()
    }

    private func subscribeToBeautyState() {
        baseBeautyStore.state //
            .subscribe()
            .receive(on: DispatchQueue.main)
            .assign(to: \.value, on: beautyStatePublisher)
            .store(in: &cancellables)
    }

    // ... 后续方法
}
```

步骤3：设置美颜参数

当用户拖动美颜滑块或点击预设按钮时，调用相应的接口设置美颜强度。

实现方式

1. **获取强度值**：从 UI 控件（如 `UISlider`）获取用户设定的强度值。请注意，SDK 接口接收的参数范围是 `[0, 9]`，其中 `0` 表示关闭效果，`9` 表示效果最明显。您需要将 UI 控件的值（例如 UISlider 的 `0.0 - 1.0`）映射到 `0 - 9` 的范围。
2. **调用接口**：分别调用 `setSmoothLevel(smoothLevel:)`、`setWhitenessLevel(whitenessLevel:)`、`setRuddyLevel(ruddyLevel:)` 来设置磨皮、美白、红润的强度。

代码示例

```
extension BeautyManager {
    /// 设置磨皮等级（输入范围 0.0 ~ 1.0，内部转换为 0 ~ 9）
    func updateSmoothLevel(uiLevel: Float) {
        // 将 UI 的 0.0 ~ 1.0 映射到 SDK 的 0 ~ 9
        let sdkLevel = uiLevel * 9.0
        baseBeautyStore.setSmoothLevel(smoothLevel: sdkLevel) //
    }

    /// 设置美白等级（输入范围 0.0 ~ 1.0，内部转换为 0 ~ 9）
    func updateWhitenessLevel(uiLevel: Float) {
        let sdkLevel = uiLevel * 9.0
        baseBeautyStore.setWhitenessLevel(whitenessLevel: sdkLevel) //
    }

    /// 设置红润等级（输入范围 0.0 ~ 1.0，内部转换为 0 ~ 9）
    func updateRuddyLevel(uiLevel: Float) {
        let sdkLevel = uiLevel * 9.0
        baseBeautyStore.setRuddyLevel(ruddyLevel: sdkLevel) //
    }
}
```

步骤4：重置美颜效果

当用户点击“重置”或“关闭美颜”按钮时，将所有美颜参数恢复到默认值（通常是0）。

实现方式

调用接口：直接调用 baseBeautyStore.reset() 方法。

代码示例

```
extension BeautyManager {
    /// 重置所有基础美颜效果
    func resetBeautyEffects() {
        baseBeautyStore.reset() //
    }
}
```

功能进阶

基础美颜与高级美颜对比

AtomicXCore 也提供了高级美颜支持，以满足不同场景的需求

对比项	基础美颜 (BaseBeautyStore)	高级美颜 (TEBeautyKit 需额外集成)
核心功能	磨皮、美白、红润	包含基础美颜，并增加 V 脸、眼距、瘦鼻、3D 贴纸、滤镜、美妆等丰富效果
计费	免费 (包含在 AtomicXCore 授权内)	付费（需要额外购买腾讯特效 SDK License）
集成方式	默认内置，直接使用 `BaseBeautyStore.shared`	需要额外集成 `TEBeautyKit` 组件并进行鉴权
推荐场景	对美颜要求不高，需要快速实现基础美颜功能的场景	对美颜效果有较高要求，需要丰富的美型、贴纸、滤镜等高级功能的场景

集成高级美颜

如果您需要使用高级美颜功能，请参考 高级美颜 文档进行集成。集成并成功鉴权 `TEBeautyKit` 后，您可以通过 `TEBeautyKit` 提供的接口来控制所有美颜效果。

API 文档

关于 BaseBeautyStore 及其相关类的所有公开接口、属性和方法的详细信息，请参阅随 AtomicXCore 框架的官方 API 文档。本指南使用到的相关 Store 如下：

Store/Component	功能描述	API 文档
BaseBeautyStore	基础美颜：调节磨皮 / 美白 / 红润（0–9 级），重置美颜状态，同步效果参数。	API 文档
DeviceStore	音视频设备控制：麦克风（开关 / 音量）、摄像头（开关 / 切换 / 画质）、屏幕共享，设备状态实时监听。	API 文档

常见问题

我设置了美颜参数，为什么没有效果?

请检查以下几点：

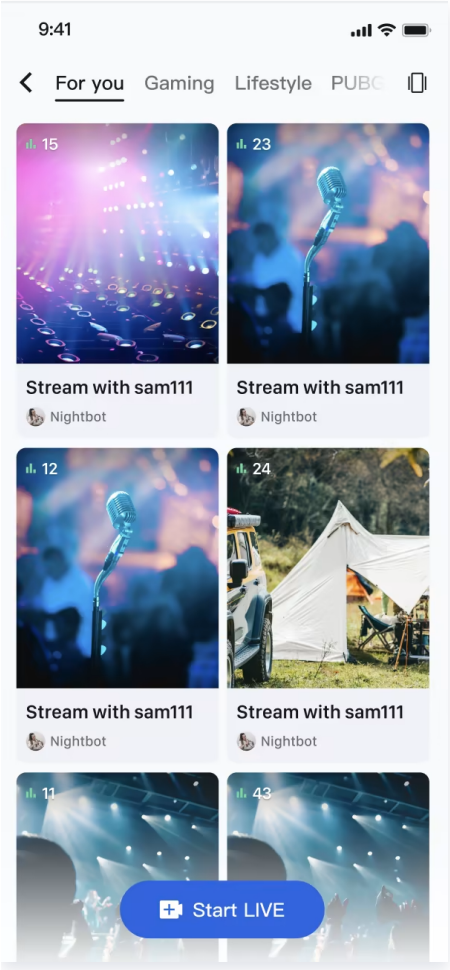
1. **摄像头是否开启**：必须先成功打开摄像头（例如通过 `DeviceStore.shared.openLocalCamera`），美颜效果才能应用到视频流上。

2. **是否使用了高级美颜**：如果您集成了 `TEBeautyKit`（高级美颜），请确保您使用的是 `TEBeautyKit` 提供的接口来调节美颜。
3. **参数范围**：如果您使用的基础美颜，请确认您传入的强度值在有效的范围内（`0` 到 `9` 之间的 Float 值）。

直播间列表
直播间列表（Android）

最近更新时间：2025-11-21 14:15:34

`LiveListStore` 是 `AtomicXCore` 中负责管理直播房间列表、创建、加入以及维护房间状态的核心模块。通过 `LiveListStore`，您可以为您的应用构建完整的直播生命周期管理。



核心功能

- **直播列表拉取**：获取当前所有公开的直播间列表，支持分页加载。
- **直播生命周期管理**：提供从创建、开播、加入、离开到结束直播的全套流程接口。
- **直播信息更新**：主播可以随时更新直播间的公开信息，例如封面、公告等。
- **实时事件监听**：监听直播结束、用户被踢出等关键事件。
- **自定义业务数据**：强大的自定义元数据（`MetaData`）能力，允许您在房间内存储和同步任何业务相关的信息，例如直播状态、音乐信息、自定义角色等 。

核心概念

在开始集成之前，我们先通过下表了解一下 `LiveListStore` 相关的几个核心概念：

核心概念	类型	核心职责与描述
`LiveInfo`	`data class`	代表一个直播间的所有信息模型。包含了直播ID（`liveID`）、名称（`liveName`）、房主信息（`liveOwner`）以及自定义元数据（`metaData`）等。
`LiveListState`	`data class`	代表直播列表模块的当前状态。其核心属性 liveList 是一个 `StateFlow`，存储了拉取到的直播列表；`currentLive` 则代表用户当前所在的直播间信息。
`LiveListListener`	`abstract class`	代表直播房间的全局事件。分为 `onLiveEnded`（直播结束）和 `onKickedOutOfLive`（被踢出直播）两种，用于处理关键的房间状态变更。
`LiveListStore`	`abstract class`	代表直播房间的全局事件。分为 `.onLiveEnded`（直播结束）和 `.onKickedOutOfLive`（被踢出直播）两种，用于处理关键的房间状态变更。

实现步骤

步骤1：组件集成

请参考 开始直播 集成 AtomicXCore，并完成 主播开播 和 观众观看 的功能实现。

步骤2：实现观众从直播列表进入直播间

创建一个展示直播列表的页面，该页面使用 `RecyclerView` 来布局直播间卡片。当用户点击某个卡片时，获取该直播间的 `liveId`，并跳转到观众观看页面。

```
import android.content.Intent
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.GridLayoutManager
import androidx.recyclerview.widget.RecyclerView
import io.trtc.tuikit.atomicxcore.api.CompletionHandler
import io.trtc.tuikit.atomicxcore.api.live.LiveInfo
import io.trtc.tuikit.atomicxcore.api.live.LiveListStore
import kotlinx.coroutines.*
```

```
import kotlinx.coroutines.flow.*

class LiveListActivity : AppCompatActivity() {
    private val liveListStore = LiveListStore.shared()
    private var liveList: List<LiveInfo> = emptyList()
    private val coroutineScope = CoroutineScope(Dispatchers.Main)

    private lateinit var recyclerView: RecyclerView
    private lateinit var adapter: LiveListAdapter

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_live_list)
        setupUI()
        bindStore()
        fetchLiveList()
    }

    private fun bindStore() {
        // 订阅 state，自动接收列表更新
        coroutineScope.launch {
            liveListStore.liveState.liveList.collect { fetchedList ->
                liveList = fetchedList
                adapter.notifyDataSetChanged()
            }
        }
    }

    private fun fetchLiveList() {
        liveListStore.fetchLiveList(
            cursor = "",
            count = 20,
            completion = object : CompletionHandler {
                override fun onSuccess() {
                    println("直播列表拉取成功")
                }

                override fun onFailure(code: Int, desc: String) {
                    println("直播列表拉取失败: $desc")
                }
```

```
            }
        )
    }

    // 当用户点击列表中的某个Item时
    private fun onLiveItemClick(liveInfo: LiveInfo) {
        // 创建观众观看页面，并将 liveId 传入
        val intent = Intent(this, YourAudienceActivity::class.java)
        intent.putExtra("liveId", liveInfo.liveID)
        startActivity(intent)
    }


    // --- RecyclerView 相关方法 ---
    private fun setupUI() {
        recyclerView = findViewById(R.id.recyclerView)
        adapter = LiveListAdapter(liveList) { liveInfo ->
            onLiveItemClick(liveInfo)
        }
        recyclerView.layoutManager = GridLayoutManager(this, 2)
        recyclerView.adapter = adapter
    }

    override fun onDestroy() {
        super.onDestroy()
        coroutineScope.cancel()
    }
}

// RecyclerView Adapter
class LiveListAdapter(
    private var liveList: List<LiveInfo>,
    private val onItemClick: (LiveInfo) -> Unit
) : RecyclerView.Adapter<LiveListAdapter.LiveViewHolder>() {

    class LiveViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
        // 定义您的ViewHolder视图组件
        // val titleTextView: TextView =
itemView.findViewById(R.id.titleTextView)
```

```
        // val coverImageView: ImageView =
itemView.findViewById(R.id.coverImageView)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
LiveViewHolder {
        val view = LayoutInflater.from(parent.context)
            .inflate(R.layout.item_live_card, parent, false)
        return LiveViewHolder(view)
    }

    override fun onBindViewHolder(holder: LiveViewHolder, position: Int)
{
        val liveInfo = liveList[position]
        // 设置数据到ViewHolder
        // holder.titleTextView.text = liveInfo.liveName
        // 加载封面图片等

        holder.itemView.setOnClickListener {
            onItemClick(liveInfo)
        }
    }

    override fun getItemCount(): Int = liveList.size
}
```

LiveInfo 参数说明

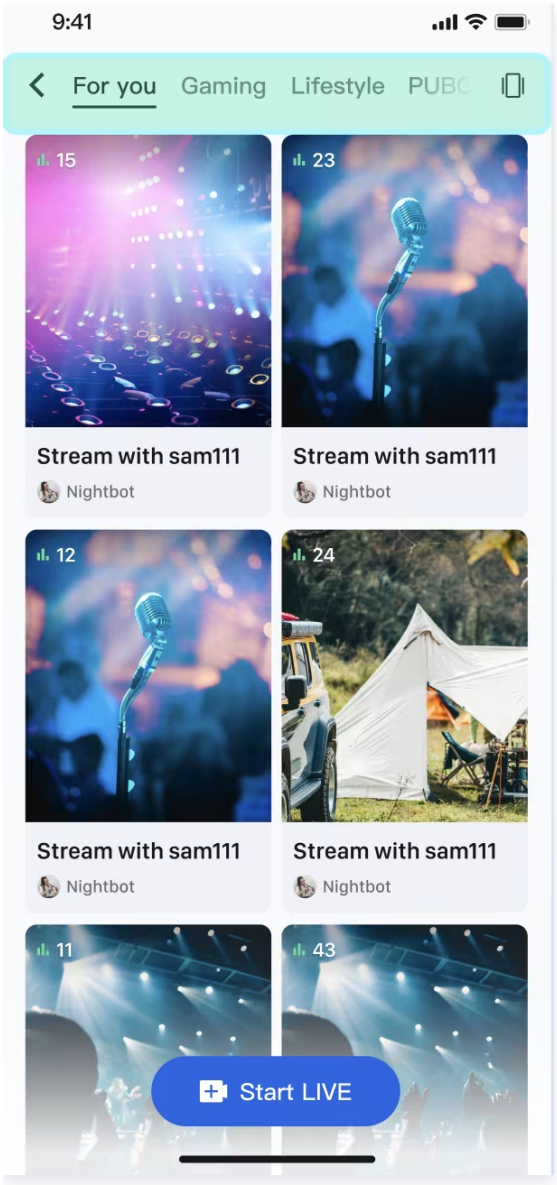
参数名	类型	描述
`liveID`	`String`	直播间的唯一标识符
`liveName`	`String`	直播间的标题
`coverURL`	`String`	直播间的封面图片地址
`liveOwner`	LiveUserInfo	房主的个人信息
`totalViewerCount`	`Int`	直播间的总观看人数

`categoryList`	`List<Int>`	直播间的分类标签列表
`notice`	`String`	直播间的公告信息
`metaData`	`Map<String: String>`	开发者自定义的元数据，用于实现复杂的业务场景

功能进阶

场景一：实现直播列表的分类展示

在 App 的直播广场页，顶部设有"热门"、"音乐"、"游戏"等分类标签。用户点击不同的标签后，下方的直播列表会动态筛选，只展示对应分类的直播间，从而帮助用户快速发现感兴趣的内容。



实现方式

核心是利用 `LiveInfo` 模型中的 `categoryList` 属性。当主播开播设置分类后，`fetchLiveList` 返回的 `LiveInfo` 对象中就会包含这些分类信息。您的 App 在获取到完整的直播列表后，只需在客户端根据用户选择的分类，对这个列表进行一次简单的筛选，然后刷新 UI 即可。

代码示例

以下示例展示了如何在 `LiveListActivity` 中扩展一个 `LiveListManager` 来处理数据和筛选逻辑。

```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.RecyclerView
import io.trtc.tuikit.atomicxcore.api.live.LiveInfo
import io.trtc.tuikit.atomicxcore.api.live.LiveListStore
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

// 1. 创建一个数据管理器来封装数据获取和筛选逻辑
class LiveListManager {
    private val liveListStore = LiveListStore.shared()
    private var fullLiveList: List<LiveInfo> = emptyList()

    // 对外暴露最终的直播列表
    private val _filteredLiveList = MutableStateFlow<List<LiveInfo>>
(emptyList())
    val filteredLiveList: StateFlow<List<LiveInfo>> = _filteredLiveList

    init {
        // 监听完整列表变化
        CoroutineScope(Dispatchers.Main).launch {
            liveListStore.liveState.liveList.collect { fetchedList ->
                fullLiveList = fetchedList
                // 默认将完整列表发布出去
                _filteredLiveList.value = fetchedList
            }
        }
    }

    fun fetchFirstPage() {
        liveListStore.fetchLiveList(cursor = "", count = 20, completion
= null)
```

```
    }

    /// 根据分类筛选直播列表
    fun filterLiveList(categoryId: Int?) {
        if (categoryId == null) {
            // 如果 categoryId 为 null，则显示完整列表
            _filteredLiveList.value = fullLiveList
            return
        }

        val filteredList = fullLiveList.filter { liveInfo ->
            liveInfo.categoryList.contains(categoryId)
        }
        _filteredLiveList.value = filteredList
    }
}

// 2. 在您的 LiveListActivity 中使用 Manager
class LiveListActivity : AppCompatActivity() {
    private val manager = LiveListManager()
    private lateinit var recyclerView: RecyclerView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_live_list)
        // setupUI()

        // 绑定数据
        CoroutineScope(Dispatchers.Main).launch {
            manager.filteredLiveList.collect { filteredList ->
                // 刷新UI
               // adapter.updateList(filteredList)
            }
        }

        // 首次拉取
        manager.fetchFirstPage()
    }

    // 当用户点击顶部分类标签时
```

```
    fun onCategorySelected(categoryId: Int) {
        manager.filterLiveList(categoryId)
    }

    // ... (RecyclerView 相关代码)
}
```

场景二：实现直播列表的滑动播放

用户可以通过上下滑动来切换直播间，当一个新的直播间滑动到屏幕中央时，视频会自动开始播放预览；当它滑出屏幕时，视频则会自动停止，以节省带宽和设备性能。

交互流程图

实现方式

`LiveCoreView` 支持多实例使用，我们为每一个 `RecyclerView.ViewHolder` 都创建一个独立的 `LiveCoreView` 实例。通过监听 `RecyclerView` 的滚动状态，我们可以精确地控制即将出现和已经离开屏幕的 ViewHolder 中的 `LiveCoreView` 何时开始和停止拉流，从而实现“即滑即播、即走即停”的效果。

代码示例

我们创建一个自定义的 `LiveFeedViewHolder`，它内部持有一个 `LiveCoreView`。然后在 Activity 中管理这些 ViewHolder 的播放状态。

```
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.RecyclerView
import io.trtc.tuikit.atomicxcore.api.live.LiveInfo
import io.trtc.tuikit.atomicxcore.api.view.CoreViewType
import io.trtc.tuikit.atomicxcore.api.view.LiveCoreView

// 1. 自定义 RecyclerView.ViewHolder，内部包含一个 LiveCoreView
class LiveFeedViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
    private var liveCoreView: LiveCoreView? = null

    fun setLiveInfo(liveInfo: LiveInfo) {
        // 为新的直播信息创建一个新的 LiveCoreView
        liveCoreView = LiveCoreView(itemView.context, viewType =
CoreViewType.PLAY_VIEW)
        liveCoreView?.let { view ->
            (itemView as ViewGroup).addView(view)
            view.layoutParams = ViewGroup.LayoutParams(
                ViewGroup.LayoutParams.MATCH_PARENT,
                ViewGroup.LayoutParams.MATCH_PARENT
            )
        }
    }

    fun startPlay(roomId: String) {
        liveCoreView?.startPreviewLiveStream(roomId, false, callback =
null)
    }

    fun stopPlay(roomId: String) {
        liveCoreView?.stopPreviewLiveStream(roomId)
    }
}
```

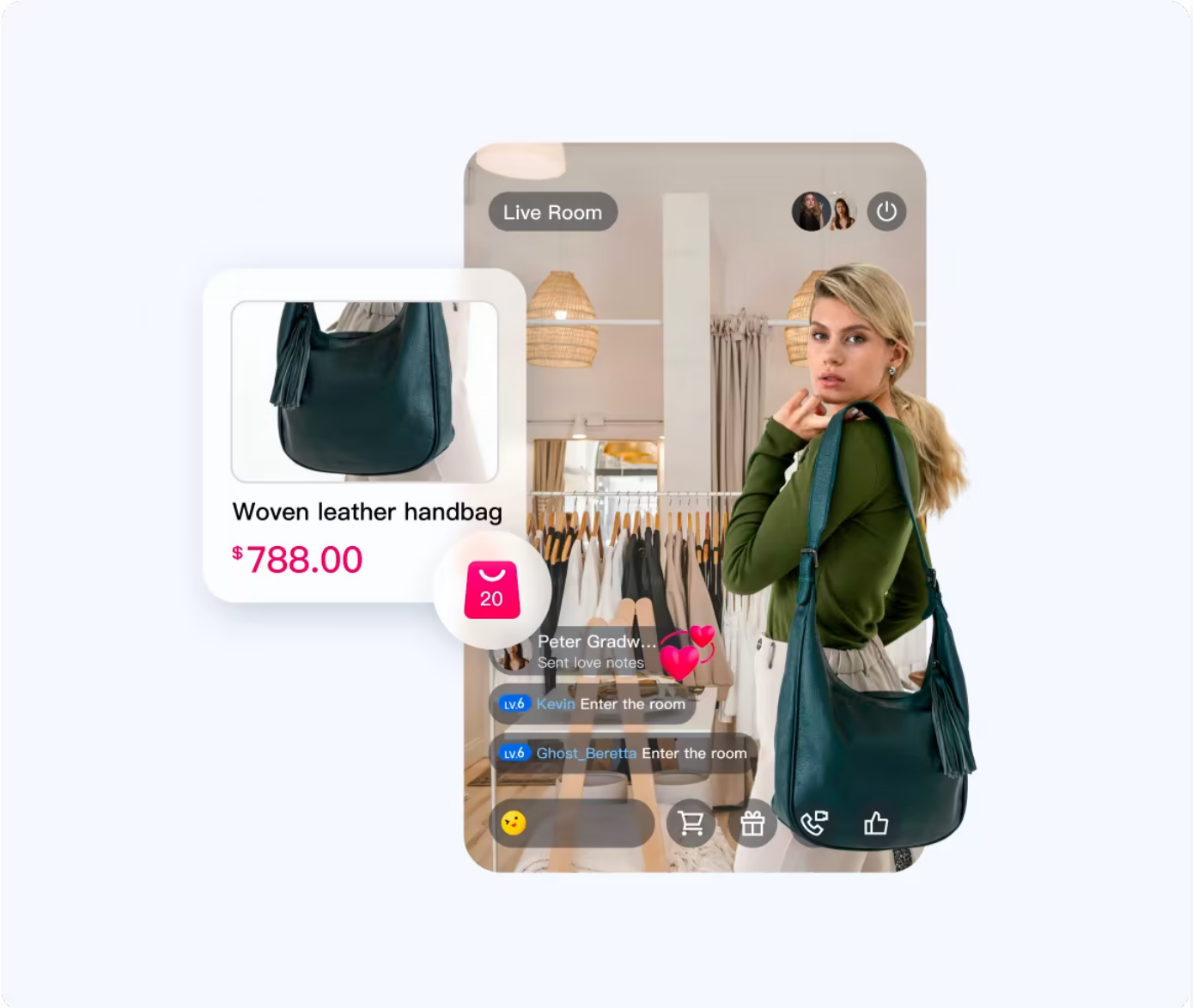
```
// 2. 在 Activity 中管理播放逻辑
class LiveFeedActivity : AppCompatActivity() {

    private lateinit var recyclerView: RecyclerView
    private var liveList: List<LiveInfo> = emptyList()
    private var currentPlayingPosition: Int = -1

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_live_feed)
        setupRecyclerView()
    }

    private fun setupRecyclerView() {
        recyclerView = findViewById(R.id.recyclerView)
        recyclerView.layoutManager = LinearLayoutManager(this,
LinearLayoutManager.VERTICAL, false)
        recyclerView.adapter = LiveFeedAdapter(liveList) { position ->
            playVideoAtPosition(position)
        }

        // 监听滚动状态
        recyclerView.addOnScrollListener(object :
RecyclerView.OnScrollListener() {
            override fun onScrollStateChanged(recyclerView:
RecyclerView, newState: Int) {
                super.onScrollStateChanged(recyclerView, newState)
                if (newState == RecyclerView.SCROLL_STATE_IDLE) {
                    val layoutManager = recyclerView.layoutManager as
LinearLayoutManager
                    val firstVisiblePosition =
layoutManager.findFirstCompletelyVisibleItemPosition()
                    if (firstVisiblePosition !=
RecyclerView.NO_POSITION) {
                        playVideoAtPosition(firstVisiblePosition)
                    }
                }
            }
        })
    }
```

```
    private fun playVideoAtPosition(position: Int) {
        // 只有当居中的位置变化时才切换播放
        if (currentPlayingPosition != position) {
            // 停止当前播放
            if (currentPlayingPosition != -1) {
                val currentViewHolder =
recyclerView.findViewHolderForAdapterPosition(currentPlayingPosition)
                if (currentViewHolder is LiveFeedViewHolder) {
                    val liveInfo = liveList[currentPlayingPosition]
                    currentViewHolder.stopPlay(liveInfo.liveID)
                }
            }

            // 开始新的播放
            val newViewHolder =
recyclerView.findViewHolderForAdapterPosition(position)
            if (newViewHolder is LiveFeedViewHolder) {
                val liveInfo = liveList[position]
                newViewHolder.startPlay(liveInfo.liveID)
                currentPlayingPosition = position
            }
        }
    }

    // RecyclerView Adapter
    inner class LiveFeedAdapter(
        private var liveList: List<LiveInfo>,
        private val onItemClick: (Int) -> Unit
    ) : RecyclerView.Adapter<LiveFeedViewHolder>() {

        override fun onCreateViewHolder(parent: ViewGroup, viewType:
Int): LiveFeedViewHolder {
            val view = LayoutInflater.from(parent.context)
                .inflate(R.layout.item_live_feed, parent, false)
            return LiveFeedViewHolder(view)
        }

        override fun onBindViewHolder(holder: LiveFeedViewHolder,
position: Int) {
```

```
            val liveInfo = liveList[position]
            holder.setLiveInfo(liveInfo)
            holder.itemView.setOnClickListener {
                onItemClick(position)
            }
        }

        override fun getItemCount(): Int = liveList.size
    }
}
```

场景三：实现电商直播的商品信息展示

在电商直播中，主播正在讲解一款商品。此时，所有观众的直播画面下方都会弹出一张商品卡片，实时展示该商品的图片、名称和价格。当主播切换讲解下一件商品时，所有观众端的卡片都会自动、同步更新。



实现方式

主播端将当前商品的结构化信息（建议使用 JSON 格式）通过 `updateLiveMetaData` 接口设置到一个自定义的 key（例如 "`product_info`"）中。AtomicXCore 会将这个变更实时同步给所有观众。观众端只需订阅 `LiveListStore.liveState.currentLive`，监听 `metaData` 的变化，一旦发现 `product_info` 的值更新，就解析其中的 JSON 数据并刷新商品卡片 UI。

代码示例

```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import io.trtc.tuikit.atomicxcore.api.CompletionHandler
import io.trtc.tuikit.atomicxcore.api.live.LiveListStore
import org.json.JSONObject
import org.json.JSONException
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

// 1. 定义一个商品信息数据类
data class Product(
    val id: String,
    val name: String,
    val price: String,
    val imageUrl: String
)

// 2. 主播端：在 YourAnchorActivity 中添加推送商品的方法
class YourAnchorActivity : AppCompatActivity() {
    fun pushProductInfo(product: Product) {
        try {
            val jsonObject = JSONObject().apply {
                put("id", product.id)
                put("name", product.name)
                put("price", product.price)
                put("imageUrl", product.imageUrl)
            }
            val jsonString = jsonObject.toString()


            val metaData = hashMapOf("product_info" to jsonString)
```

```
            // 使用全局单例的 liveListStore 来更新 metaData
            LiveListStore.shared().updateLiveMetaData(
                metaData,
                object : CompletionHandler {
                    override fun onSuccess() {
                        println("商品 ${product.name} 信息推送成功")
                    }

                    override fun onFailure(code: Int, desc: String) {
                        println("商品信息推送失败: $desc")
                    }
                }
            )
        } catch (e: JSONException) {
            println("JSON序列化失败: ${e.message}")
        }
    }
}

// 3. 观众端：在 YourAudienceActivity 中订阅并响应
class YourAudienceActivity : AppCompatActivity() {

    private val liveListStore = LiveListStore.shared()
    private val coroutineScope = CoroutineScope(Dispatchers.Main)
    // 假设您有一个自定义的商品卡片视图
    // private lateinit var productCardView: ProductCardView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_audience)
        // ... (setupUI, 布局 productCardView)
        joinLive()
        subscribeToProductUpdates()
    }

    private fun subscribeToProductUpdates() {
        coroutineScope.launch {
            liveListStore.liveState.currentLive.collect { currentLive ->
```

```
                val productInfoJson =
currentLive.metaData["product_info"]
                if (productInfoJson != null) {
                    try {
                        val jsonObject = JSONObject(productInfoJson)
                        val product = Product(
                            id = jsonObject.getString("id"),
                            name = jsonObject.getString("name"),
                            price = jsonObject.getString("price"),
                            imageUrl = jsonObject.getString("imageUrl")
                        )
                        // productCardView.updateProduct(product)
                        // productCardView.show()
                    } catch (e: JSONException) {
                        println("商品信息解析失败: ${e.message}")
                        // productCardView.hide()
                    }
                } else {
                    // productCardView.hide() // 如果没有商品信息，则隐藏卡片
                }
            }
        }
    }

    override fun onDestroy() {
        super.onDestroy()
        coroutineScope.cancel()
    }

    // ... (joinLive 和 leaveLive 方法)
}
```

API 文档

关于 LiveListStore 及其相关类的所有公开接口、属性和方法的详细信息，请参阅随 AtomicXCore 框架的官方 API 文档。本文档使用到的相关 Store 如下：

Store/Component	功能描述	API 文档

LiveCoreView	直播视频流展示与交互的核心视图组件：负责视频流渲染和视图挂件处理，支持主播直播、观众连麦、主播连线等场景。	API 文档
LiveListStore	直播间全生命周期管理：创建 / 加入 / 离开 / 销毁房间，查询房间列表，修改直播信息（名称、公告等），监听直播状态（例如被踢出、结束）。	API 文档

常见问题

使用 `updateLiveMetaData` 时有哪些需要注意的限制和规则?

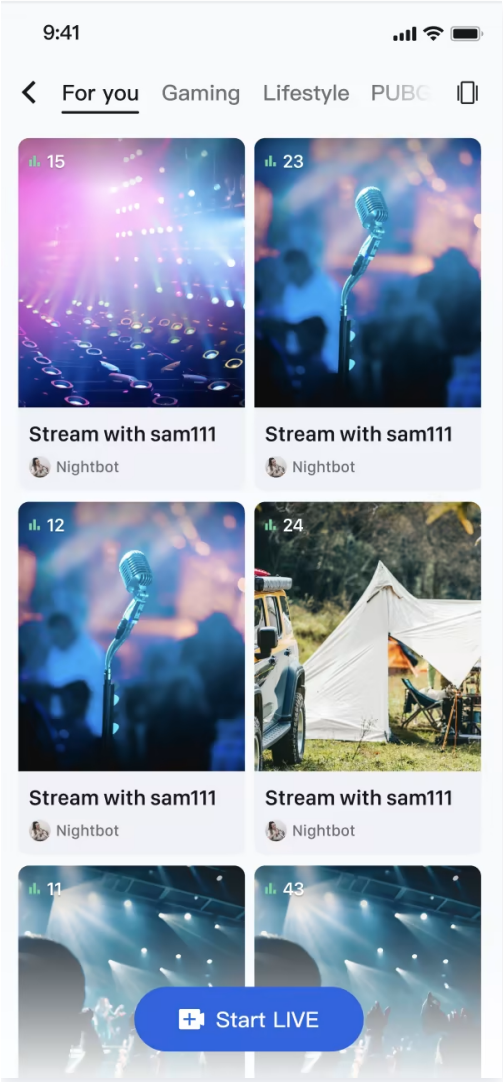
为了保证系统的稳定和高效， `metaData` 的使用遵循以下规则：

- **权限:** 只有房主和管理员可以调用 `updateLiveMetaData` 。普通观众没有权限。
- **数量与大小限制:**
  - 单个房间最多支持 **10** 个 key。
  - 每个 key 的长度不超过 **50 字节**，每个 value 的长度不超过 **2KB**。
  - 单个房间所有 value 的总大小不超过 **16KB**。
- **冲突解决:** `metaData` 的更新机制是"后来者覆盖"。如果多个管理员在短时间内修改同一个 key，最后一次的修改会生效。建议在业务设计上避免多人同时修改同一个关键信息。

直播间列表（iOS）

最近更新时间：2025-11-21 14:15:35

`LiveListStore` 是 `AtomicXCore` 中负责管理直播房间列表、创建、加入以及维护房间状态的核心模块。通过 `LiveListStore`，您可以为您的应用构建完整的直播生命周期管理。



核心功能

- **直播列表拉取**：获取当前所有公开的直播间列表，支持分页加载。
- **直播生命周期管理**：提供从创建、开播、加入、离开到结束直播的全套流程接口。
- **直播信息更新**：主播可以随时更新直播间的公开信息，例如封面、公告等。
- **实时事件监听**：监听直播结束、用户被踢出等关键事件。
- **自定义业务数据**：强大的自定义元数据（`MetaData`）能力，允许您在房间内存储和同步任何业务相关的信息，例如直播状态、音乐信息、自定义角色等 。

核心概念

在开始集成之前，我们先通过下表了解一下 `LiveListStore` 相关的几个核心概念：

核心概念	类型	核心职责与描述
`LiveInfo`	`struct`	代表一个直播间的所有信息模型。包含了直播ID（`liveID`）、名称（`liveName`）、房主信息（`liveOwner`）以及自定义元数据（`metaData`）等。
`LiveListState`	`struct`	代表直播列表模块的当前状态。其核心属性 liveList 是一个 `[LiveInfo]` 数组，存储了拉取到的直播列表；`currentLive` 则代表用户当前所在的直播间信息。
`LiveListEvent`	`enum`	代表直播房间的全局事件。分为 `.onLiveEnded`（直播结束）和 `.onKickedOutOfLive`（被踢出直播）两种，用于处理关键的房间状态变更。
`LiveListStore`	`class`	这是与直播列表和房间生命周期交互的核心管理类。它是一个全局单例（shared），负责所有直播房间的创建、加入、信息更新等操作。

实现步骤

步骤1：组件集成

请参考 开始直播 集成 AtomicXCore，并完成 主播开播 和 观众观看 的功能实现。

步骤2：实现观众从直播列表进入直播间

创建一个展示直播列表的页面，该页面使用 `UICollectionView` 来布局直播间卡片。当用户点击某个卡片时，获取该直播间的 `liveId`，并跳转到观众观看页面。

```
import AtomicXCore
import SnapKit
import RTCRoomEngine
import Combine

class LiveListViewController: UIViewController,
UICollectionViewDataSource, UICollectionViewDelegate,
UICollectionViewDelegateFlowLayout {

    private let liveListStore = LiveListStore.shared
    private var cancellables = Set<AnyCancellable>()
    private var liveList: [LiveInfo] = []

    private var collectionView: UICollectionView!
```

```
    override func viewDidLoad() {
        super.viewDidLoad()
        setupUI()
        bindStore()
        fetchLiveList()
    }

    private func bindStore() {
        // 订阅 state，自动接收列表更新
        liveListStore.state
            .subscribe(StatePublisherSelector(keyPath:
\LiveListState.liveList))
            .receive(on: DispatchQueue.main)
            .sink { [weak self] fetchedList in
                self?.liveList = fetchedList
                self?.collectionView.reloadData()
            }
            .store(in: &cancellables)
    }

    private func fetchLiveList() {
        liveListStore.fetchLiveList(cursor: "", count: 20) { result in
            if case .failure(let error) = result {
                print("直播列表拉取失败：\(error.localizedDescription)")
            }
        }
    }

    // 当用户点击列表中的某个Cell时
    func collectionView(_ collectionView: UICollectionView,
didSelectItemAt indexPath: IndexPath) {
        let selectedLiveInfo = liveList[indexPath.item]

        // 创建观众观看页面，并将 liveId 传入
        let audienceVC = YourAudienceViewController(liveId:
selectedLiveInfo.liveID)
        audienceVC.modalPresentationStyle = .fullScreen
        present(audienceVC, animated: true)
    }
```

```
    // --- UICollectionViewDataSource, Delegate, and UI setup methods --
-
    private func setupUI() {
        let layout = UICollectionViewFlowLayout()
        // ... (可以自定义您的布局)
        collectionView = UICollectionView(frame: view.bounds,
collectionViewLayout: layout)
        collectionView.dataSource = self
        collectionView.delegate = self
        collectionView.register(UICollectionViewCell.self,
forCellWithReuseIdentifier: "LiveCell")
        view.addSubview(collectionView)
    }

    func collectionView(_ collectionView: UICollectionView,
numberOfItemsInSection section: Int) -> Int {
        return liveList.count
    }

    func collectionView(_ collectionView: UICollectionView,
cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        let cell =
collectionView.dequeueReusableCell(withReuseIdentifier: "LiveCell", for:
indexPath)
        cell.backgroundColor = .lightGray // 自定义您的Cell样式
        // let liveInfo = liveList[indexPath.item]
        // ... (例如: cell.titleLabel.text = liveInfo.liveName)
        return cell
    }

    func collectionView(_ collectionView: UICollectionView, layout
collectionViewLayout: UICollectionViewLayout, sizeForItemAt indexPath:
IndexPath) -> CGSize {
        // 单列、双列布局可通过自定义您的Cell大小来实现
        let width = (view.bounds.width - 30) / 2
        return CGSize(width: width, height: width * 1.2)
    }
}
```

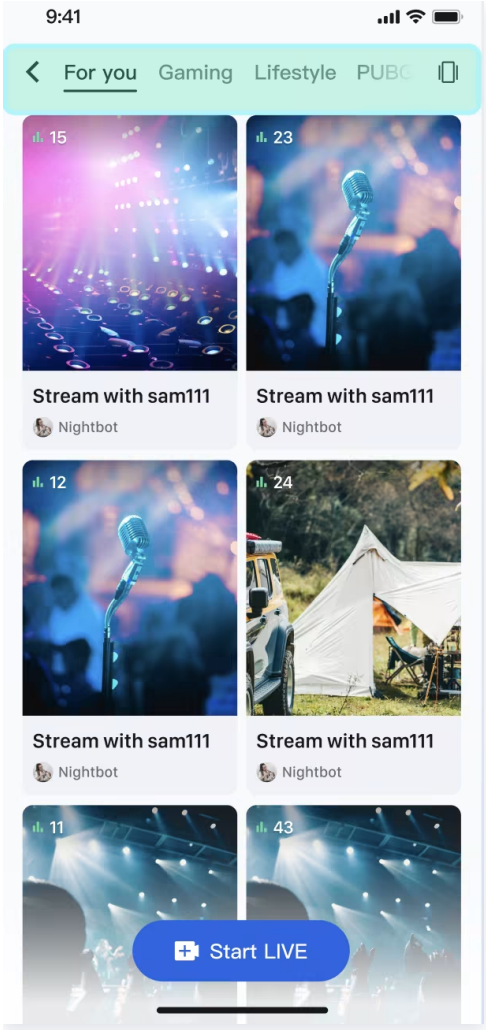
LiveInfo 参数说明

参数名	类型	描述
`liveID`	`String`	直播间的唯一标识符
`liveName`	`String`	直播间的标题
`coverURL`	`String`	直播间的封面图片地址
`liveOwner`	LiveUserInfo	房主的个人信息
`totalViewerCount`	`Int`	直播间的总观看人数
`categoryList`	`[NSNumber]`	直播间的分类标签列表
`notice`	`String`	直播间的公告信息
`metaData`	`[String: String]`	开发者自定义的元数据，用于实现复杂的业务场景

功能进阶

场景一：实现直播列表的分类展示

在 App 的直播广场页，顶部设有“热门”、“音乐”、“游戏”等分类标签。用户点击不同的标签后，下方的直播列表会动态筛选，只展示对应分类的直播间，从而帮助用户快速发现感兴趣的内容。



实现方式

核心是利用 `LiveInfo` 模型中的 `categoryList` 属性。当主播开播设置分类后，`fetchLiveList` 返回的 `LiveInfo` 对象中就会包含这些分类信息。您的 App 在获取到完整的直播列表后，只需在客户端根据用户选择的分类，对这个列表进行一次简单的筛选，然后刷新 UI 即可。

代码示例

以下示例展示了如何在 `LiveListViewController` 中扩展一个 `LiveListManager` 来处理数据和筛选逻辑。

```
import AtomicXCore
import Combine

// 1. 创建一个数据管理器来封装数据获取和筛选逻辑
class LiveListManager {
    private let liveListStore = LiveListStore.shared
    private var cancellables = Set<AnyCancellable>()
    private var fullLiveList: [LiveInfo] = []
```

```
    // 对外暴露最终的直播列表
    let filteredLiveListPublisher = CurrentValueSubject<[LiveInfo],
Never>([])

    init() {
        liveListStore.state
            .subscribe(StatePublisherSelector(keyPath:
\LiveListState.liveList))
            .receive(on: DispatchQueue.main)
            .sink { [weak self] fetchedList in
                self?.fullLiveList = fetchedList
                // 默认将完整列表发布出去
                self?.filteredLiveListPublisher.send(fetchedList)
            }
            .store(in: &cancellables)
    }

    func fetchFirstPage() {
        liveListStore.fetchLiveList(cursor: "", count: 20) { _ in }
    }

    /// 根据分类筛选直播列表
    func filterLiveList(by categoryId: NSNumber?) {
        guard let categoryId = categoryId else {
            // 如果 categoryId 为 nil，则显示完整列表
            filteredLiveListPublisher.send(fullLiveList)
            return
        }

        let filteredList = fullLiveList.filter { liveInfo in
            liveInfo.categoryList.contains(categoryId)
        }
        filteredLiveListPublisher.send(filteredList)
    }
}

// 2. 在您的 LiveListViewController 中使用 Manager
class LiveListViewController: UIViewController,
UICollectionViewDataSource, UICollectionViewDelegate {
```

```
    private let manager = LiveListManager()
    private var cancellables = Set<AnyCancellable>()
    private var liveList: [LiveInfo] = []
    private var collectionView: UICollectionView!

    override func viewDidLoad() {
        super.viewDidLoad()
        // ... setupUI ...

        // 绑定数据
        manager.filteredLiveListPublisher
            .receive(on: DispatchQueue.main)
            .sink { [weak self] filteredList in
                self?.liveList = filteredList
                self?.collectionView.reloadData()
            }
            .store(in: &cancellables)

        // 首次拉取
        manager.fetchFirstPage()
    }

    // 当用户点击顶部分类标签 (UISegmentedControl) 时
    @objc func categorySegmentDidChange(_ sender: UISegmentedControl) {
        let selectedCategoryId: NSNumber? = 1 // 假设 "音乐" 分类的ID是 1
        manager.filterLiveList(by: selectedCategoryId)
    }

    // ... (UICollectionView 相关代码)
}
```

场景二：实现直播列表的滑动播放

用户可以通过上下滑动来切换直播间，当一个新的直播间滑动到屏幕中央时，视频会自动开始播放预览；当它滑出屏幕时，视频则会自动停止，以节省带宽和设备性能。

交互流程图

实现方式

`LiveCoreView` 支持多实例使用，我们为每一个 `UICollectionViewCell` 都创建一个独立的 `LiveCoreView` 实例。通过监听 `UICollectionView` 的滚动代理方法，我们可以精确地控制即将出现和已经离开屏幕的 Cell 中的 `LiveCoreView` 何时开始和停止拉流，从而实现“即滑即播、即走即停”的效果。

代码示例

我们创建一个自定义的 `LiveFeedCell`，它内部持有一个 `LiveCoreView`。然后在 UIViewController 中管理这些 Cell 的播放状态。

```
import UIKit
import AtomicXCore

// 1. 自定义 UICollectionViewCell，内部包含一个 LiveCoreView
class LiveFeedCell: UICollectionViewCell {
    private var liveCoreView: LiveCoreView?

    func setLiveInfo(_ liveInfo: LiveInfo) {
        // 为新的直播信息创建一个新的 LiveCoreView
        liveCoreView = LiveCoreView(viewType: .playView)
        guard let liveCoreView = liveCoreView else { return }
        contentView.addSubview(liveCoreView)
        liveCoreView.frame = contentView.bounds
```

```
    }

    func startPlay(roomId: String) {
        liveCoreView?.startPreviewLiveStream(roomId: roomId,
isMuteAudio: false)
    }

    func stopPlay(roomId: String) {
        liveCoreView?.stopPreviewLiveStream(roomId: roomId)
    }
}

// 2. 在 ViewController 中管理播放逻辑
class LiveFeedViewController: UIViewController,
UICollectionViewDataSource, UICollectionViewDelegate {

    private var collectionView: UICollectionView!
    private var liveList: [LiveInfo] = []
    private var currentPlayingIndexPath: IndexPath?

    // 当滑动完全停止后，此代理方法会被调用
    func scrollViewDidEndDecelerating(_ scrollView: UIScrollView) {
        let page = Int(scrollView.contentOffset.y / view.frame.height)
        let indexPath = IndexPath(item: page, section: 0)

        // 只有当居中的 Cell 变化时才切换播放
        if currentPlayingIndexPath != indexPath {
            playVideo(at: indexPath)
        }
    }

    // 当 Cell 即将离开屏幕时调用
    func collectionView(_ collectionView: UICollectionView,
didEndDisplaying cell: UICollectionViewCell, forItemAt indexPath:
IndexPath) {
        if let liveCell = cell as? LiveFeedCell {
            let liveInfo = liveList[indexPath.item]
            liveCell.stopPlay(roomId: liveInfo.liveID)
        }
    }
```

```
    private func playVideo(at indexPath: IndexPath) {
        if let cell = collectionView.cellForItem(at: indexPath) as?
LiveFeedCell {
            let liveInfo = liveList[indexPath.item]
            cell.startPlay(roomId: liveInfo.liveID)
            currentPlayingIndexPath = indexPath
        }
    }

    // ... (UICollectionViewDataSource 的其他方法)
    func collectionView(_ collectionView: UICollectionView,
cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        let cell =
collectionView.dequeueReusableCell(withReuseIdentifier: "LiveFeedCell",
for: indexPath) as! LiveFeedCell
        cell.setLiveInfo(liveList[indexPath.item])
        return cell
    }
}
```

场景三：实现电商直播的商品信息展示

在电商直播中，主播正在讲解一款商品。此时，所有观众的直播画面下方都会弹出一张商品卡片，实时展示该商品的图片、名称和价格。当主播切换讲解下一件商品时，所有观众端的卡片都会自动、同步更新。

实现方式

主播端将当前商品的结构化信息（建议使用 JSON 格式）通过 `updateLiveMetaData` 接口设置到一个自定义的 key（例如 "`product_info`"）中。AtomicXCore 会将这个变更实时同步给所有观众。观众端只需订阅 `LiveListStore.state`，监听 `metaData` 的变化，一旦发现 `product_info` 的值更新，就解析其中的 JSON 数据并刷新商品卡片 UI。

代码示例

```
import AtomicXCore
import Combine

// 1. 定义一个商品信息模型，遵循 Codable 以便与JSON互转
struct Product: Codable {
    let id: String
    let name: String
    let price: String
    let imageUrl: String
}

// 2. 主播端：在 YourAnchorViewController 中添加推送商品的方法
extension YourAnchorViewController {
```

```
    func pushProductInfo(_ product: Product) {
        guard let jsonData = try? JSONEncoder().encode(product),
              let jsonString = String(data: jsonData, encoding: .utf8)
else { return }

        let metaData = ["product_info": jsonString]

        // 使用全局单例的 liveListStore 来更新 metaData
        LiveListStore.shared.updateLiveMetaData(metaData) { result in
            if case .success = result {
                print("商品 \(product.name) 信息推送成功")
            }
        }
    }
}

// 3. 观众端：在 YourAudienceViewController 中订阅并响应
class YourAudienceViewController: UIViewController {

    // ... （已有代码）
    private var cancellables = Set<AnyCancellable>()
    // 假设您有一个自定义的商品卡片视图
    // private let productCardView = ProductCardView()

    override func viewDidLoad() {
        super.viewDidLoad()
        view.backgroundColor = .black
        // ... (setupUI, 布局 productCardView)
        joinLive()
        subscribeToProductUpdates()
    }

    private func subscribeToProductUpdates() {
        LiveListStore.shared.state
            // 监听当前房间的 metaDatass
            .subscribe(StatePublisherSelector(keyPath:
\LiveListState.currentLive.metaData))
            .removeDuplicates()
            .receive(on: DispatchQueue.main)
            .sink { metaData in
```

```
                guard let jsonString = metaData["product_info"],
                      let data = jsonString.data(using: .utf8),
                      let product = try?
JSONDecoder().decode(Product.self, from: data) else {
                    // self.productCardView.hide() // 如果没有商品信息，则隐
藏卡片
                    return
                }
                // self.productCardView.update(with: product) // 使用新数
据更新商品卡片UI
                // self.productCardView.show()
            }
            .store(in: &cancellables)
    }

    // ... (joinLive 和 leaveLive 方法)
}
```

API 文档

关于 LiveListStore 及其相关类的所有公开接口、属性和方法的详细信息，请参阅随 AtomicXCore 框架的官方 API 文档。本文档使用到的相关 Store 如下：

Store/Component	功能描述	API 文档
LiveCoreView	直播视频流展示与交互的核心视图组件：负责视频流渲染和视图挂件处理，支持主播直播、观众连麦、主播连线等场景。	API 文档
LiveListStore	直播间全生命周期管理：创建 / 加入 / 离开 / 销毁房间，查询房间列表，修改直播信息（名称、公告等），监听直播状态（例如被踢出、结束）。	API 文档

常见问题

使用 `updateLiveMetaData` 时有哪些需要注意的限制和规则？

为了保证系统的稳定和高效， `metaData` 的使用遵循以下规则：

- **权限:** 只有房主和管理员可以调用 `updateLiveMetaData` 。普通观众没有权限。
- **数量与大小限制:**
  - 单个房间最多支持 10 个 key。
  - 每个 key 的长度不超过 50 字节，每个 value 的长度不超过 2KB。

- 单个房间所有 value 的总大小不超过 16KB。

- **冲突解决：** `metaData` 的更新机制是“后来者覆盖”。如果多个管理员在短时间内修改同一个 key，最后一次的修改会生效。建议在业务设计上避免多人同时修改同一个关键信息。