直播 SDK

UI 组件库

【版权声明】

©2013-2026 腾讯云版权所有

本文档（含所有文字、数据、图片等内容）完整的著作权归腾讯云计算（北京）有限责任公司单独所有，未经腾讯云事先明确书面许可，任何主体不得以任何形式复制、修改、使用、抄袭、传播本文档全部或部分内容。前述行为构成对腾讯云著作权的侵犯，腾讯云将依法采取措施追究法律责任。

【商标声明】

及其它腾讯云服务相关的商标均为腾讯云计算（北京）有限责任公司及其关联公司所有。本文档涉及的第三方主体的商标，依法由权利人所有。未经腾讯云及有关权利人书面许可，任何主体不得以任何方式对前述商标进行使用、复制、修改、传播、抄录等行为，否则将构成对腾讯云及有关权利人商标权的侵犯，腾讯云将依法采取措施追究法律责任。

【服务声明】

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

【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

UI 组件库
主播核心页面
主播核心页面（iOS）
主播核心页面（Android）
观众核心页面
观众核心页面（iOS）
观众核心页面（Android）
直播视频组件
调整视频直播挂件 (Android)
调整视频直播挂件 (iOS UIKit)
直播视频组件 (Web Vue3)
直播视频组件 (Web React)
语聊麦位组件
调整界面风格
调整界面风格（Web 桌面浏览器）
直播列表
直播列表（Android）
直播列表（iOS）
直播列表（Web Vue3）
直播列表（Web React）
直播列表（Flutter）
直播列表（uni-app 客户端）
观众列表组件
观众列表组件 (Android Java)
观众列表组件 (iOS UIKit)
观众列表组件 (Web Vue3)
观众列表组件 (Web React)
礼物组件
礼物组件 (Android)
礼物组件 (iOS)
礼物组件 (Web Vue3)
礼物组件 (Web React)
弹幕组件
弹幕组件 (Android Kotlin)
弹幕组件 (iOS UIKit)

版权所有：腾讯云计算（北京）有限责任公司

弹幕组件 (Web Vue3)
弹幕组件 (Web React)
美颜调节面板
美颜调节面板 (Android Java)
美颜调节面板 (iOS UIKit)
美颜调节面板（Flutter）

UI 组件库

主播核心页面

主播核心页面（iOS）

最近更新时间：2026-05-20 14:47:33

`AnchorView` 主播端核心 UI 组件。开发者可以通过该组件快速搭建基础的直播界面。该组件提供了丰富的 `API` 接口与高度的可定制性。本文档将按照从基础按钮调整到复杂视图替换的顺序，指导开发者完成界面元素的按需定制。

页面结构示意图

准备工作

在开始调整开播页界面前，请先参考 主播开播 完成主流程的搭建。

功能概览

`AnchorView` 提供的核心自定义接口与属性

方法/属性	描述

`topRightItems`	用于灵活配置直播间右上角的按钮集合，支持自由添加自定义按钮或调整内置按钮的布局。
`bottomItems`	用于灵活配置直播间底部的按钮集合，支持自由添加自定义按钮或调整内置按钮的布局。
`replace(node:with:)`	用于将指定位置的默认组件（如顶部信息区、底部操作栏），替换为开发者自定义的全新视图。
`overlayView`	专属挂件图层，方便开发者自由添加需要在视频画面上方悬浮展示的全局业务UI。
`perform(action:)`	用于在自定义视图中直接触发内置默认逻辑，例如显示默认观众列表，显示默认连麦管理面板等。

快速开始

下面示例快速搭建一个秀场主播工作台：

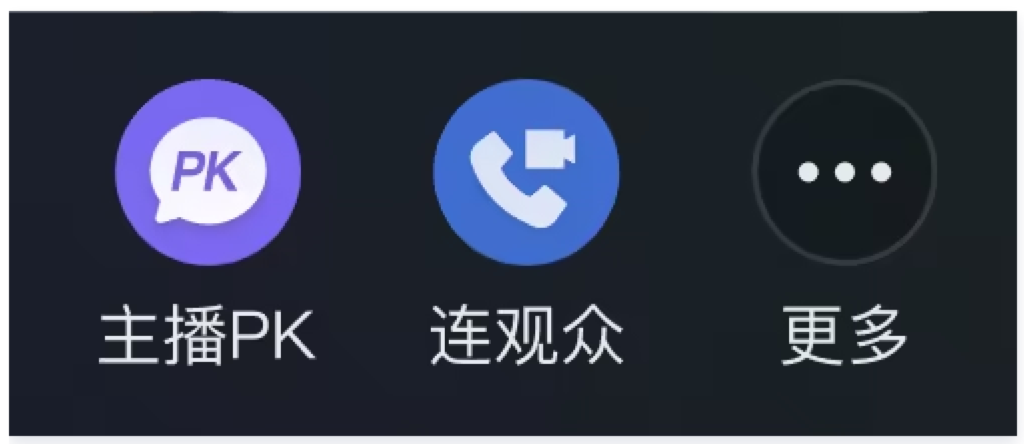
- 底部栏添加美颜按钮。
- 替换直播信息节点为自己业务风格的样式。
- 添加“人气榜”悬浮活动挂件。

```
import UIKit
import AtomicX

let anchorView: AnchorView

func setupUI() {
    // 步骤1：创建美颜按钮，配置按钮顺序
    let beautyButton = UIButton(type: .system)
    beautyButton.setTitle("美颜", for: .normal)
    // 依次排列：主播连麦、PK、自定义美颜按钮、更多
    anchorView.bottomItems = [.coHost, .battle, .custom(beautyButton),
.more]


    // 步骤2：调整直播间信息视图 LiveInfo 样式
    let customLiveInfoView = CustomLiveInfoView()
    anchorView.replace(node: .liveInfo, with: customLiveInfoView)


    // 步骤3：添加“人气榜”浮层挂件
```

```
    let rankWidget = RankWidgetView()
    // 将挂件视图添加到专属的挂件图层
    anchorView.overlayView.addSubview(rankWidget)
    // 设置布局约束，贴靠在左上角下方
    rankWidget.snp.makeConstraints { make in
        make.top.equalToSuperview().offset(120)
        make.leading.equalToSuperview().offset(12)
        make.width.equalTo(70)
        make.height.equalTo(30)
    }
}
```

调整底部操作按钮

底部工具栏是主播进行互动操作的核心区域。目前从左到右默认显示“主播 PK”、“连观众”、“更多”三个按钮，开发者可以通过 `.custom(UIView)` 插入自定义按钮，或通过 `bottomItems` 属性灵活增删内置功能、调整按钮顺序。



实现步骤

步骤1：准备自定义按钮视图。按需创建自定义按钮视图对象。
步骤2：更新底部按钮数组。组装包含 `AnchorBottomItem` 枚举的数组，并重新赋值给组件属性。

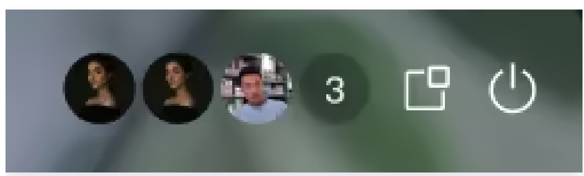
```
import UIKit
import AtomicX

let anchorView: AnchorView

// 示例场景：在底部工具栏仅保留连麦功能，并新增一个商品列表触发按钮
func adjustBottomBar() {
    // 步骤1：准备自定义按钮视图
```

版权所有：腾讯云计算（北京）有限责任公司

```
    let goodsButton = UIButton(type: .custom)
    goodsButton.setImage(UIImage(named: "shop_cart"), for: .normal)

    // 步骤2：更新底部按钮数组
    anchorView.bottomItems = [
        .coHost,
        .custom(goodsButton)
    ]
}
```

调整顶部操作按钮

顶部区域通常用于展示关键的房间信息和操作。目前从左到右默认显示“观众人数”、“悬浮窗”、“关闭”三个按钮，开发者可以通过 `topRightItems` 属性精简或增加顶部右上角的控制按钮。



实现步骤

- **步骤 1**：准备自定义按钮视图。按需创建自定义按钮视图对象。
- **步骤 2**：更新顶部按钮数组。将选定的枚举项或包装好的自定义视图赋值给 `topRightItems` 属性。

```
import UIKit
import AtomicX

let anchorView: AnchorView

// 示例场景：在右上角保留观众人数、关闭房间按钮，新增一个“举报”按钮
func adjustTopRightBar() {
   // 步骤1：准备自定义按钮视图
    let reportButton = UIButton(type: .custom)
    reportButton.setImage(UIImage(named: "report_btn"), for: .normal)

    // 步骤2：根据显示顺序更新顶部按钮数组
    anchorView.topRightItems = [.audienceCount, .custom(reportButton),
.close]
}
```

版权所有：腾讯云计算（北京）有限责任公司

替换界面指定区域视图

当调整按钮无法满足结构性修改需求（例如把页面左下角的弹幕输入框改成其他按钮）时，可使用 `replace` 接口整体替换指定区域的视图。
组件内部通过 `AnchorNode` 枚举定义了 5 个支持自定义替换的 UI 区域，可以结合开头的“页面结构示意图”理解：

AnchorNode	说明
`liveInfo`	左上角的主播与房间信息展示区域。
`topRightButtons`	右上角的系统控制按钮区域。
`networkInfo`	网络状态指示区域。
`bottomRightBar`	右下角的业务操作栏区域。
`barrageInput`	左下角的弹幕输入框触发区域。

> **说明：**
> `replace` 方法会完全替换指定节点，所以如果您替换了 `topRightButtons` 和 `bottomRightBar` 两个节点后，对应的 Items 属性会失效。建议您仅需调整按钮时使用前文介绍的 `Items` 属性，需要完全重构该区域布局时才使用 `replace` 。

布局规则

`replace` 接口会将自定义视图放入指定的 `slot` 区域。视图的位置由框架控制，开发者无需设置。视图的尺寸由其自身决定，推荐使用以下两种方式之一声明尺寸：

方式 1：使用内部约束链（推荐）

确保子视图的约束形成完整链条，从而自动撑开父视图。

```
class MyInfoView: UIView {
    init() {
        super.init(frame: .zero)
        let label = UILabel()
        label.text = "直播中"
        addSubview(label)
        label.snp.makeConstraints { make in
            make.edges.equalToSuperview().inset(12)   //由子视图撑开视图
        }
```

```
    }

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

方式 2：重写 intrinsicContentSize

通过重写属性直接指定视图的固定尺寸。

```
class MyInfoView: UIView {
    override var intrinsicContentSize: CGSize {
        CGSize(width: 200, height: 44)
    }
}
```

实现步骤

- **步骤 1**：创建自定义视图对象。
- **步骤 2**：调用 `AnchorView` 组件的 `replace` 接口，传入需要替换的节点枚举和新建的自定义视图。

```
let anchorView: AnchorView

// 示例场景：使用自定义的直播信息视图替换默认区域
func replaceLiveInfoNode() {
    // 步骤1：初始化符合布局规则的自定义视图
    let customInfoView = MyInfoView()
    customInfoView.backgroundColor = .darkGray

    // 步骤2：调用替换接口更新特定节点
    anchorView.replace(node: .liveInfo, with: customInfoView)
}
```

绑定事件与触发逻辑

当开发者使用自定义视图替换了默认按钮或局部节点后，需要自行接管该视图的交互事件。在事件响应方法中，开发者可以执行专属业务逻辑，也可以使用 `perform` 方法快速触发 `TUILiveKit` 内部的默认逻辑。

实现步骤

- **步骤 1：为自定义视图绑定事件，使用 `addTarget` 或手势识别器为视图添加点击事件。**
- **步骤 2：触发内置逻辑或执行业务代码，在事件回调中调用 `perform` 方法传入 `AnchorAction` 枚举，或执行其他业务代码。**

```
import UIKit
import AtomicX

class YourAnchorViewController {
    var anchorView: AnchorView?

    // 示例场景：为自定义的直播信息视图绑定点击事件，并复用 SDK 内置的主播信息面板
    func setupCustomLiveInfoNode() {
        // 步骤1：初始化自定义视图并绑定手势事件
        let customLiveInfoView = MyInfoView() // 自定义的直播信息view
        customLiveInfoView.backgroundColor = .darkGray

        // 开启用户交互并添加点击手势
        customLiveInfoView.isUserInteractionEnabled = true
        let tapGesture = UITapGestureRecognizer(target: self, action:
#selector(handleLiveInfoClick))
        customLiveInfoView.addGestureRecognizer(tapGesture)

        // 将自定义视图替换到直播信息节点
        anchorView?.replace(node: .liveInfo, with: customLiveInfoView)
    }

    // 步骤2：触发内置逻辑或业务代码
    @objc func handleLiveInfoClick() {
        // UI 节点视图自定义，但点击逻辑依然复用内置的主播信息弹窗
        anchorView?.perform(.showLiveInfo)
        // 或是弹出您自己实现的主播信息弹窗
    }
}
```

深度定制业务弹窗

当使用 `perform` 触发的内置默认面板无法满足具体的业务需求时，开发者可以彻底接管这部分逻辑，基于底层数据 `Core SDK` 构建全新的业务面板。

核心思路

开发者可使用底层数据接口 `AtomicXCore` 获取房间、用户及状态数据，完全自主地完成自定义视图的搭建与交互绑定。在完成自定义控制器的构建后，建议使用内部封装的 `AtomicPopover` 组件将其弹出，以确保您的自定义面板也能获得与 `SDK` 内置面板一致的丝滑手势拦截与平滑动画。

实现步骤

- **步骤 1**：构建自定义业务视图。创建一个独立的 `UIView` ，并在其内部引入 `AtomicXCore` 接口，用于拉取或监听底层业务数据以驱动 UI 更新。
- **步骤 2**：配置弹窗容器参数。实例化弹窗配置对象，并指定弹出位置、占用高度与动画类型。
- **步骤 3**：弹出自定义视图。实例化您的自定义视图，将其作为 `contentView` 传入 `AtomicPopover` 容器，并通过系统方法进行展示。

```
import UIKit
import AtomicX
import AtomicXCore // 引入底层数据接口进行业务开发

// 步骤1：构建自定义观众列表视图
class CustomAudienceListView: UIView {
    override init(frame: CGRect) {
        super.init(frame: frame)
        self.backgroundColor = .white
        setupUI()
        bindLiveData()
    }

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

    private func setupUI() {
        // 在此添加您的自定义 UI 控件，例如展示观众头像、用户等级等
    }

    private func bindLiveData() {
        // 使用 AtomicXCore 提供的核心接口获取当前房间状态或用户数据
        // 拿到核心数据后刷新上方构建的自定义 UI
    }
}

// 示例场景：从屏幕底部向上滑出一个高度占屏幕一半的完全自定义面板
```

```
func presentBusinessPanel(from parentViewController: UIViewController) {
    // 步骤2：配置弹窗容器参数
    let config = AtomicPopover.AtomicPopoverConfig(
        position: .bottom,
        height: .ratio(0.5),
        animation: .slideFromBottom
    )

    // 步骤3：弹出自定义视图
    let audienceListView = CustomAudienceListView()
    let popover = AtomicPopover(contentView: audienceListView,
configuration: config)

    parentViewController.present(popover, animated: true)
}
```

请参考以下文档，使用 `AtomicXCore` 接口实现自定义功能面板页

功能描述	参考文档
实现观众连线管理面板：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。	观众连线
实现主播跨房连线面板：连线主播互动管理，发起 / 接受 / 拒绝连线。	主播连线和 PK
实现观众列表：统计观众数量，监听观众进出事件。	观众列表
实现音频特效面板：变声（童声 / 男声）、混响（KTV 等）、耳返调节，实时切换特效。	音效

添加自定义悬浮挂件

复杂的直播场景常需要在视频画面上方悬浮展示活动图标或互动贴纸。这类需要定位于视频层之上且独立于基础布局的视图，要添加到 `overlayView` 图层中。

在实际业务中，悬浮挂件通常作为某个活动面板的入口。您可以将其与上一节介绍的 `AtomicPopover` 结合使用：给挂件绑定点击事件，并在点击后弹出自定义业务弹窗。

实现步骤

- **步骤 1**：创建挂件视图并开启交互。实例化悬浮控件，设置其尺寸与位置。
- **步骤 2**：绑定点击事件。为挂件添加手势识别器，用于响应主播的点击操作。
- **步骤 3**：添加至覆盖图层并联动弹窗。将挂件添加至 `overlayView` 中，并在点击回调中调用前文定义的自定义弹窗逻辑。

```
import UIKit
import TUILiveKit

class LiveRoomController: UIViewController {
    var anchorView: AnchorView?


    // 示例场景：在画面左上角悬浮显示红包挂件，点击后联动弹出前文定义的业务面板
    func addRedPacketWidget() {
        // 步骤 1：创建挂件视图并开启交互
        let redPacketWidget = UIImageView(image: UIImage(named:
"red_packet_icon"))
        redPacketWidget.frame = CGRect(x: 15, y: 120, width: 60, height:
60)
        redPacketWidget.isUserInteractionEnabled = true

        // 步骤 2：绑定点击事件
        let tapGesture = UITapGestureRecognizer(target: self, action:
#selector(handleRedPacketClick))
        redPacketWidget.addGestureRecognizer(tapGesture)

        // 步骤 3：添加至覆盖图层
        anchorView?.overlayView.addSubview(redPacketWidget)
    }

    @objc func handleRedPacketClick() {
        // 在此处调用前文写好的 presentBusinessPanel 方法，弹出深度定制的业务视
图
        //  presentBusinessPanel(from: self)
    }
}
```

常见问题

添加的自定义按钮为什么点击无响应?

使用 `.custom()` 或者 `replace` 传入自定义视图后，点击视图却无法触发绑定的事件。

排查建议：

- 检查视图尺寸与内部约束（高频原因）：这是最容易被忽略的问题。如果您传入了一个自定义的容器视图（例如替换了整个 `bottomRightBar` ），请务必确保内部子控件形成了完整的约束链以撑开父视图，或者为主视图重写了 `intrinsicContentSize` 。如果父视图实际尺寸为 0，即使内部按钮在屏幕上可见，点击事件也会因超出父视图边界而被系统直接丢弃。
- 检查交互属性：检查 `isUserInteractionEnabled` 属性。如果是 `UIImageView` 或普通的 `UIView` 容器，系统默认该属性为 `false` ，必须手动将其设置为 `true` 。
- 检查事件绑定与目标（ `Target` ）：确认是否正确添加了 `UITapGestureRecognizer` 或调用了 `addTarget` 方法。

动态隐藏或显示操作按钮?

在实际业务中，可能需要根据主播的等级或房间状态（如 PK 阶段隐藏连麦按钮），动态调整底部或顶部的工具栏。

实现方案： `AnchorView` 的 `bottomItems` 和 `topRightItems` 属性是支持响应式更新的。只需组装一个新的按钮数组并重新赋值给该属性， `SDK` 内部便会自动触发视图刷新，无需手动调用 `reloadData` 等重绘方法。

替换视图后出现尺寸异常?

主视图内部的子控件需形成完整的约束链，或重写 `intrinsicContentSize` （详见上文的“自定义视图布局规则”章节）。

主播核心页面（Android）

最近更新时间：2026-05-14 16:18:31

`AnchorView` 主播端核心 UI 组件。开发者可以通过该组件快速搭建基础的直播界面。该组件提供了丰富的 `API` 接口与高度的可定制性。本文档将按照从基础按钮调整到复杂视图替换的顺序，指导开发者完成界面元素的按需定制。

页面结构示意图

准备工作

在开始调整开播页界面前，请先参考 主播开播 完成主流程的搭建。

功能概览

`AnchorView` 提供的核心自定义接口与属性：

方法/属性	描述
`topRightItems`	用于灵活配置直播间右上角的按钮集合，支持自由添加自定义按钮或调整内置按钮的布局。
`bottomItems`	用于灵活配置直播间底部的按钮集合，支持自由添加自定义按钮或调整内置按钮的布局。

版权所有：腾讯云计算（北京）有限责任公司

`replace(node: AnchorNode, view: View?)`	用于将指定位置的默认组件（例如顶部信息区、底部操作栏），替换为开发者自定义的全新视图。
`overlayView`	专属挂件图层，方便开发者自由添加需要在视频画面上方悬浮展示的全局业务 UI。
`perform(action: AnchorAction)`	用于在自定义视图中直接触发内置默认逻辑，例如显示默认观众列表，显示默认连麦管理面板等。

快速开始

下面示例快速搭建一个秀场主播工作台：

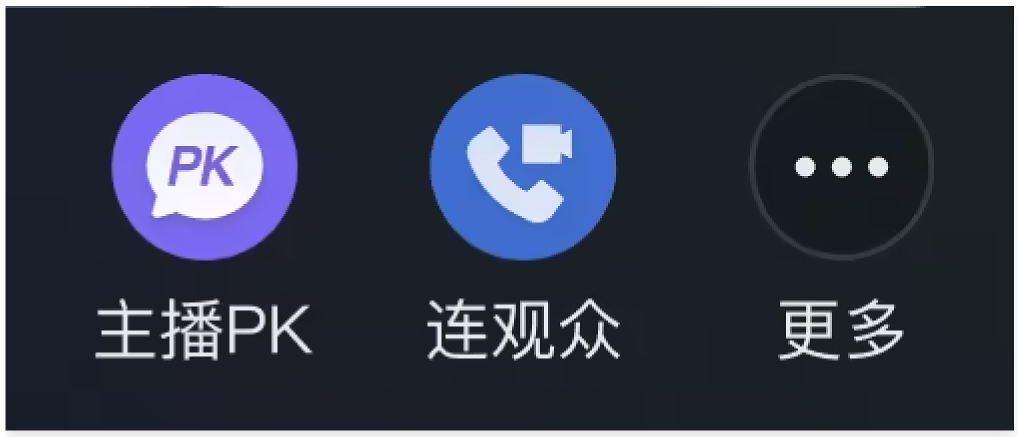
- 底部栏添加美颜按钮。
- 替换直播信息节点为自己业务风格的样式。
- 添加“人气榜”悬浮活动挂件。

```
import android.view.View
import android.widget.Button
import android.widget.FrameLayout
import com.trtc.uikit.livekit.features.anchorview.*

fun setupUI(anchorView: AnchorView) {
    val context = anchorView.context

    // 步骤1：创建美颜按钮，配置按钮顺序
    val beautyButton = Button(context).apply {
        text = "美颜"
    }
    // 依次排列：主播连麦(CoHost)、PK(Battle)、自定义美颜按钮、更多(More)
    anchorView.bottomItems = listOf(
        AnchorBottomItem.CoHost,
        AnchorBottomItem.Battle,
        AnchorBottomItem.Custom(beautyButton),
        AnchorBottomItem.More
    )

    // 步骤2：调整直播间信息视图 LiveInfo 样式
    val customLiveInfoView = CustomLiveInfoView(context)
    anchorView.replace(AnchorNode.LIVE_INFO, customLiveInfoView)
```

```
    // 步骤3：添加“人气榜”浮层挂件
    val rankWidget = RankWidgetView(context)
    // 设置布局参数，贴靠在左上角下方
    val params = FrameLayout.LayoutParams(dp2px(70f), dp2px(30f)).apply
{
        topMargin = dp2px(120f)
        leftMargin = dp2px(12f)
    }
    // 将挂件视图添加到专属的挂件图层
    anchorView.overlayView.addView(rankWidget, params)
}

// 辅助工具方法（示例）
fun dp2px(dpValue: Float): Int {
    val scale = Resources.getSystem().displayMetrics.density
    return (dpValue * scale + 0.5f).toInt()
}
```

调整底部操作按钮

底部工具栏是主播进行互动操作的核心区域。目前从左到右默认显示“主播 PK”、“连观众”、“更多”三个按钮，开发者可以通过 `AnchorBottomItem.Custom(View)` 插入自定义按钮，或通过 `bottomItems` 属性灵活增删内置功能、调整按钮顺序。



实现步骤

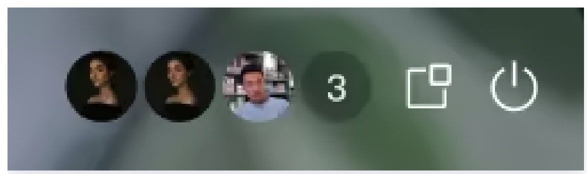
步骤1：准备自定义按钮视图。按需创建自定义按钮视图对象。
步骤2：更新底部按钮数组。组装包含 `AnchorBottomItem` 密封类的集合，并重新赋值给组件属性。

```
// 示例场景：在底部工具栏仅保留连麦功能(CoHost)，并新增一个商品列表触发按钮
fun adjustBottomBar(anchorView: AnchorView) {
```

```
    val context = anchorView.context

    // 步骤1：准备自定义按钮视图
    val goodsButton = ImageView(context).apply {
        setImageResource(R.drawable.shop_cart)
    }

    // 步骤2：更新底部按钮数组
    anchorView.bottomItems = listOf(
        AnchorBottomItem.CoHost,
        AnchorBottomItem.Custom(goodsButton)
    )
}
```

调整顶部操作按钮

顶部区域通常用于展示关键的房间信息和操作。目前从左到右默认显示“观众人数、“悬浮窗”、“关闭”三个按钮，开发者可以通过 `topRightItems` 属性精简或增加顶部右上角的控制按钮。



实现步骤

- **步骤 1：**准备自定义按钮视图。按需创建自定义按钮视图对象。
- **步骤 2：**更新顶部按钮数组。组装包含 `AnchorTopRightItem` 密封类的集合，并重新赋值给组件属性。

```
// 示例场景：在右上角保留观众人数、关闭房间按钮，新增一个“举报”按钮
fun adjustTopRightBar(anchorView: AnchorView) {
    val context = anchorView.context

    // 步骤1：准备自定义按钮视图
    val reportButton = ImageView(context).apply {
        setImageResource(R.drawable.report_btn)
    }

    // 步骤2：根据显示顺序更新顶部按钮数组
    anchorView.topRightItems = listOf(
        AnchorTopRightItem.AudienceCount,
```

```
        AnchorTopRightItem.Custom(reportButton),
        AnchorTopRightItem.Close
    )
}
```

替换界面指定区域视图

当调整按钮无法满足结构性修改需求（例如把页面左下角的弹幕输入框改成其他按钮）时，可使用 `replace` 接口整体替换指定区域的视图。

组件内部通过 `AnchorNode` 枚举定义了 5 个支持自定义替换的 `UI` 区域，可以结合开头的“页面结构示意图”理解：

AnchorNode	说明
`LIVE_INFO`	左上角的主播与房间信息展示区域。
`TOP_RIGHT_BUTTONS`	右上角的系统控制按钮区域。
`NETWORK_INFO`	网络状态指示区域。
`BOTTOM_RIGHT_BAR`	右下角的业务操作栏区域。
`BARRAGE_INPUT`	左下角的弹幕输入框触发区域。

实现步骤

- **步骤 1**：创建自定义视图对象。
- **步骤 2**：调用 `AnchorView` 组件的 `replace` 接口，传入需要替换的节点枚举和新建的自定义视图。

```
// 示例场景：使用自定义的直播信息视图替换默认区域
fun replaceLiveInfoNode(anchorView: AnchorView) {
    val context = anchorView.context

    // 步骤1：初始化符合业务需求的自定义视图
    val customInfoView = MyInfoView(context).apply {
        setBackgroundColor(Color.DKGRAY)
    }

    // 步骤2：调用替换接口更新特定节点
    anchorView.replace(AnchorNode.LIVE_INFO, customInfoView)
}
```

自定义视图布局规则

`replace` 接口会将自定义视图放入指定的区域。在 `Android` 中，您替换的 `View` 的尺寸通常由您为其设置的 `LayoutParams` 或其自身的 `onMeasure` 决定。推荐在自定义 `View` 内部使用 `WRAP_CONTENT` 或明确的宽高尺寸，以便它能自适应地撑开容器：

```
class MyInfoView(context: Context) : FrameLayout(context) {
    init {
        val label = TextView(context).apply {
            text = "直播中"
            setPadding(24, 24, 24, 24)
        }
        // 使用 WRAP_CONTENT 适应文字大小
        val params = LayoutParams(LayoutParams.WRAP_CONTENT,
LayoutParams.WRAP_CONTENT)
        addView(label, params)
    }
}
```

绑定事件与触发逻辑

当开发者使用自定义视图替换了默认按钮或局部节点后，需要自行接管该视图的交互事件。在事件响应方法中，开发者可以执行专属业务逻辑，也可以使用 `perform` 方法快速触发 `LiveKit` 内部的默认逻辑。

实现步骤

- **步骤 1**：为自定义视图绑定事件，使用 `setOnClickListener` 或手势识别器为视图添加点击事件。
- **步骤 2**：触发内置逻辑或执行业务代码，在事件回调中调用 `perform` 方法传入 `AnchorAction` 枚举，或执行其他业务代码。

```
// 示例场景：为自定义的直播信息视图绑定点击事件，并复用 SDK 内置的主播信息面板
fun setupCustomLiveInfoNode(anchorView: AnchorView) {
    val customLiveInfoView = MyInfoView(anchorView.context)

    // 步骤1：开启用户交互并添加点击事件
    customLiveInfoView.isClickable = true
    customLiveInfoView.setOnClickListener {
        // 步骤2：点击逻辑依然复用内置的主播信息弹窗
        anchorView.perform(AnchorAction.SHOW_LIVE_INFO)
        // 或者在此处弹出您自己实现的主播信息弹窗
```

```
    }

    anchorView.replace(AnchorNode.LIVE_INFO, customLiveInfoView)
}
```

深度定制业务弹窗

当使用 `perform` 触发的内置默认面板无法满足具体的业务需求时，开发者可以彻底接管这部分逻辑，基于底层数据接口 `AtomicXCore` 构建全新的业务面板。

核心思路

开发者可使用底层数据接口 `AtomicXCore` 获取房间、用户及状态数据。完全自主地完成自定义视图的搭建与交互绑定。在完成自定义视图的构建后，建议使用内部封装的 `AtomicPopover` 组件将其弹出，以确保您的自定义面板也能获得与 `SDK` 内置面板一致的背景遮罩、圆角风格及平滑的进出场动画。

实现步骤

- **步骤 1：**构建自定义业务视图。创建一个独立的 `View`，并在其内部引入底层核心数据接口（如 `LiveAudienceStore`），用于拉取或监听业务数据以驱动 UI 更新。
- **步骤 2：**实例化弹窗容器。实例化 `AtomicPopover` 对象，并指定其弹出位置（`BOTTOM` 或 `CENTER`）。
- **步骤 3：**配置并展示自定义视图。设置弹窗的高度模式（如按屏幕比例），将您的自定义视图通过 `setContent` 传入容器，最后调用 `show()` 进行展示。

```
import android.content.Context
import android.graphics.Color
import android.widget.FrameLayout
import io.trtc.tuikit.atomicx.widget.basicwidget.popover.AtomicPopover
// 引入底层数据接口进行业务开发
import io.trtc.tuikit.atomicxcore.api.live.LiveAudienceStore

// 步骤1：构建自定义业务视图（例如：观众列表）
class CustomAudienceListView(context: Context) : FrameLayout(context) {
    init {
        setBackgroundColor(Color.WHITE)
        setupUI()
        bindLiveData()
    }

    private fun setupUI() {
        // 在此添加您的自定义 UI 控件，例如展示观众头像、用户等级等列表
```

```
    }

    private fun bindLiveData() {
        // 使用 AtomicXCore 提供的核心接口获取当前房间状态或用户数据
        // 例如：LiveAudienceStore.create(liveID)...
        // 拿到核心数据后刷新上方构建的自定义 UI
    }
}

// 示例场景：从屏幕底部向上滑出一个高度占屏幕 50% 的完全自定义面板
fun presentBusinessPanel(context: Context) {
    // 步骤2：实例化弹窗容器，指定从底部弹出
    val popover = AtomicPopover(context,
AtomicPopover.PanelGravity.BOTTOM)

    // 步骤3：配置并展示自定义视图
    // 设置面板高度为屏幕高度的 50%（也可以使用 WrapContent 自适应高度）
    popover.setPanelHeight(AtomicPopover.PanelHeight.Ratio(0.5f))

    // 实例化自定义业务视图
    val audienceListView = CustomAudienceListView(context)

    // 将视图填入 Popover 并展示
    popover.setContent(audienceListView)
    popover.show()
}
```

请参考以下文档，使用 `AtomicXCore` 接口实现自定义功能面板页

功能描述	参考文档
实现观众连线管理面板：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。	观众连线
实现主播跨房连线面板：连线主播互动管理，发起 / 接受 / 拒绝连线。	主播连线和 PK
实现观众列表：统计观众数量，监听观众进出事件。	观众列表
实现音频特效面板：变声（童声 / 男声）、混响（KTV 等）、耳返调节，实时切换特效。	音效

添加自定义悬浮挂件

复杂的直播场景常需要在视频画面上方悬浮展示活动图标或互动贴纸。这类需要定位于视频层之上且独立于基础布局的视图，要添加到 `overlayView` 图层中。
在实际业务中，悬浮挂件通常作为某个活动面板的入口。您可以将其与上一节介绍的 `AtomicPopover` 结合使用：给挂件绑定点击事件，并在点击后弹出您的自定义业务弹窗。

实现步骤

- **步骤 1**：创建挂件视图并开启交互。实例化悬浮控件，设置其尺寸与位置。
- **步骤 2**：绑定点击事件。为挂件添加点击事件，用于响应主播的点击操作。
- **步骤 3**：添加至覆盖图层并联动弹窗。将挂件添加至 `overlayView` 中，并在点击回调中调用前文定义的自定义弹窗逻辑。

```
// 示例场景：在画面左上角悬浮显示红包挂件，点击后联动弹出前文定义的业务面板
fun addRedPacketWidget(anchorView: AnchorView) {
    val context = anchorView.context

    // 步骤 1：创建挂件视图并开启交互
    val redPacketWidget = ImageView(context).apply {
        setImageResource(R.drawable.red_packet_icon)
        isClickable = true
    }

    // 步骤 2：绑定点击事件
    redPacketWidget.setOnClickListener {
        presentBusinessPanel(context)
    }

    // 步骤 3：设置布局属性并添加至覆盖图层
    val params = FrameLayout.LayoutParams(dp2px(60f), dp2px(60f)).apply
{
        leftMargin = dp2px(15f)
        topMargin = dp2px(120f)
    }
    anchorView.overlayView.addView(redPacketWidget, params)
}
```

常见问题

添加的自定义按钮为什么点击无响应

使用 `AnchorBottomItem.Custom()` 或自定义节点传入视图后，点击无法触发事件，通常是由以下原因引起的：

排查建议：

1. 检查交互属性：如果是原生的 `ImageView` 或普通的 `View`，确保已设置 `isClickable = true`（部分控件默认是不可点击的）。
2. 检查手势作用域/布局大小：确认 `LayoutParams` 是否正确设置。如果自定义视图的宽/高为 0，或展示范围超出了其父容器的边界，点击事件将无法被正确拦截。
3. 确认事件绑定：确认是否正确调用了 `setOnClickListener` 。

动态隐藏或显示操作按钮

在实际业务中，可能需要根据主播的等级或房间状态（例如 `PK` 阶段隐藏连麦按钮），动态调整底部或顶部的工具栏。

实现方案：`AnchorView` 的 `bottomItems` 和 `topRightItems` 属性支持响应式更新。只需组装一个新的列表集合（`List`）并重新赋值给该属性，`SDK` 内部会自动触发视图刷新，无需手动调用 `invalidate()` 等重绘方法。

替换视图后出现尺寸异常或不显示

请确保您传入的主视图通过 `LayoutParams` 明确了宽高（推荐使用 `WRAP_CONTENT` 由内部子控件撑开，或者指定明确的 `dp` 尺寸），避免因父容器无法计算尺寸导致不渲染。

版权所有：腾讯云计算（北京）有限责任公司

观众核心页面

观众核心页面（iOS）

最近更新时间：2026-05-20 14:47:38

`AudienceView` 观众端核心 UI 组件。通过该组件，开发者可以快速搭建基础的观众直播界面。本文档将按照从基础按钮调整到复杂视图替换的顺序，指导开发者完成观众端界面的按需定制。

页面结构示意图

准备工作

在开始调整观众端界面前，请先参考 观众观看 完成观众进房的主流程搭建。

功能概览

与主播端直接创建并使用视图的结构不同，观众端由于需要支持上下滑动无缝切换直播间， `AudienceView` 是一个滑动容器。

在观众端，界面定制不是直接在 `AudienceView` 上进行的，而是作用于它内部的单房间视图 `AudienceLiveView` 。滑动容器在工作时会动态创建和展示这些单房间视图，开发者需要通过 `AudienceViewDelegate` 协议，在以下核心生命周期回调中获取到对应房间的视图实例（即回调参数中的 `liveView` ），并在正确的时机对其进行定制与资源管理：

方法	描述
`audienceView(_:onCreateLiveView liveView:for:)`	**视图创建回调**。此回调会向外传递刚刚实例化的 `AudienceLiveView` 。 **适用于提前确定的静态样式定制**。由于容器会提前预加载下个房间，该回调触发时视图可能尚未显示在屏幕上。通常在这里通过 `liveView` 执行替换内置组件、配置固定按钮等一次性布局操作。
`audienceView(_:liveViewDidAppear liveView:for:)`	**视图展示回调**。此回调会向外传递当前真正显示在屏幕上的 `AudienceLiveView` 。 **适用于依赖实时状态或信令的动态调整**。通常在这里记录当前活跃的 `liveView` 实例，以便在收到业务信令（如商品上架通知）时，精准定向更新当前屏幕上的 UI；或在此处开启房间专属的定时器与动效。
`audienceView(_:liveViewDidDisappear liveView:for:)`	**视图隐藏回调**。当观众滑出该房间或关闭界面时触发。 **适用于状态重置与资源清理**。通常在这里清空活跃的 `liveView` 记录，销毁在该房间内弹出的自定义业务面板、停止动效或清理定时器。

单房间视图 `AudienceLiveView` 提供的核心自定义接口与属性如下：

方法/属性	描述
`topRightItems`	用于灵活配置直播间右上角的按钮集合，支持自由添加自定义按钮或调整内置按钮的布局。
`bottomItems`	用于灵活配置直播间底部的按钮集合，支持自由添加自定义按钮或调整内置按钮的布局。
`replace(node:with:)`	用于将指定位置的默认组件（如顶部信息区、底部操作栏），替换为开发者自定义的全新视图。
`overlayView`	专属挂件图层，方便开发者自由添加需要在视频画面上方悬浮展示的全局业务 UI。
`perform(action:)`	用于在自定义视图中直接触发内置默认逻辑，例如显示默认观众列表，显示默认连麦管理面板等。

快速开始

下面示例快速搭建一个带货直播间观看页。主要流程包括：

- 隐藏不需要的连麦按钮并添加购物车按钮。
- 模拟接收商品上架通知并弹出商品卡片。

- 最后将点击事件路由至商品列表页。

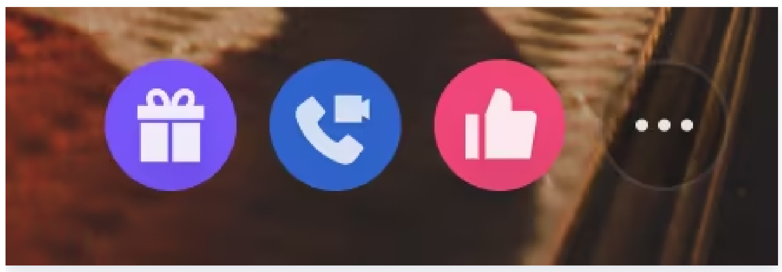
```
import UIKit
import TUILiveKit
import SnapKit

class AudienceViewController: UIViewController {
    weak var currentLiveView: AudienceLiveView?

    override func viewDidLoad() {
        super.viewDidLoad()
        // 步骤 1：初始化 AudienceView 并添加到当前控制器
        let audienceView = AudienceView(roomId: "your_room_id")
        audienceView.delegate = self

        view.addSubview(audienceView)
        audienceView.frame = view.bounds
    }

    // 业务方法：展示商品列表页
    func showProductListPanel() {
        print("展示商品列表页...")
    }

    // 业务方法：模拟收到 IM 商品推送通知
    func onReceiveProductPushMessage() {
        // 步骤 5：通过信令动态调整 UI 时，必须作用于当前正在展示的 liveView
        guard let liveView = currentLiveView else { return }

        let productCard = AudienceProductCardView() // 自定义商品卡片视图
        let productTap = UITapGestureRecognizer(target: self, action:
#selector(onProductCardTapped))
        productCard.addGestureRecognizer(productTap)

        liveView.overlayView.addSubview(productCard)
        productCard.snp.makeConstraints { make in
            make.trailing.equalToSuperview().offset(-12)
            make.bottom.equalToSuperview().offset(-100)
            make.width.equalTo(150)
        }
```

```
    }
}

extension AudienceViewController: AudienceViewDelegate {

    // 步骤 2：在创建回调中，进行静态的 UI 样式配置
    func audienceView(_ audienceView: AudienceView, onCreateLiveView
liveView: AudienceLiveView, for liveInfo: LiveInfo) {

        // 配置底部操作栏：隐藏连麦功能，并添加自定义“购物车”按钮
        let shopCartBtn = UIButton(type: .custom)
        shopCartBtn.setImage(UIImage(named: "shop_cart_icon"), for:
.normal)
        shopCartBtn.addTarget(self, action: #selector(onShopCartTapped),
for: .touchUpInside)

        // 重新排列底部栏：仅保留礼物、点赞，插入购物车按钮（不声明 .coGuest 即可隐
藏连麦）
        liveView.bottomItems = [.gift, .like, .custom(shopCartBtn)]
    }

    // 步骤 3：在展示回调中，记录当前活跃视图，为后续动态信令更新做准备
    func audienceView(_ audienceView: AudienceView, liveViewDidAppear
liveView: AudienceLiveView, for liveInfo: LiveInfo) {
        self.currentLiveView = liveView
    }

    // 步骤 4：在隐藏回调中，清理业务状态或销毁弹窗，防止视图复用导致状态错乱
    func audienceView(_ audienceView: AudienceView, liveViewDidDisappear
liveView: AudienceLiveView, for liveInfo: LiveInfo) {
        if self.currentLiveView === liveView {
            self.currentLiveView = nil
        }
    }

    // 步骤 6：处理业务点击事件
    @objc func onShopCartTapped() { showProductListPanel() }
    @objc func onProductCardTapped() { showProductListPanel() }
}
```

```
class AudienceProductCardView: UIView {
    // 自定义的商品卡片
}
```

调整底部操作按钮

底部工具栏是观众进行互动的核心区域。目前默认提供礼物、连麦、点赞、更多等按钮。开发者可通过 `bottomItems` 属性灵活增删内置功能，或通过 `.custom(UIView)` 插入自定义按钮。



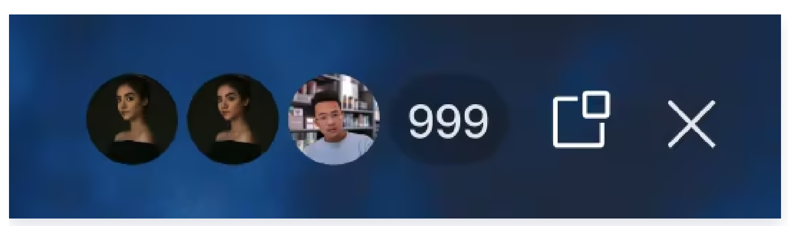
实现步骤

步骤 1： 准备自定义按钮视图。按需创建自定义按钮视图对象。
步骤 2： 更新底部按钮数组。实现 `AudienceViewDelegate` 回调，在相应的回调方法中把组装好的包含 `AudienceBottomItem` 枚举的数组重新赋值给视图属性。

```
func audienceView(_ audienceView: AudienceView,
                  onCreateLiveView liveView: AudienceLiveView,
                  for liveInfo: LiveInfo) {
    // 步骤 1：准备自定义按钮视图
    let shopButton = UIButton(type: .custom)
    shopButton.setImage(UIImage(named: "shop_cart"), for: .normal)

    // 步骤 2：更新底部按钮数组，保留送礼，隐藏连麦，新增商品按钮
    liveView.bottomItems = [
        .gift,
        .custom(shopButton)
    ]
}
```

调整顶部操作按钮

顶部区域展示房间信息与关键操作。默认提供观众人数、悬浮窗、退出三个按钮。开发者可通过 `topRightItems` 属性精简或增加控制按钮。



实现步骤

步骤 1： 准备自定义按钮视图。按需创建自定义按钮视图对象。
步骤 2： 更新顶部按钮数组。实现 `AudienceViewDelegate` 回调，在相应的回调方法中将枚举项或自定义视图赋值给 `topRightItems` 属性。

```
func audienceView(_ audienceView: AudienceView,
                  onCreateLiveView liveView: AudienceLiveView,
                  for liveInfo: LiveInfo) {
    // 步骤 1: 准备自定义按钮视图
    let reportButton = UIButton(type: .custom)
    reportButton.setImage(UIImage(named: "report_btn"), for: .normal)

    // 步骤 2: 更新顶部按钮数组（保留人数和关闭，新增举报）
    liveView.topRightItems = [.audienceCount, .custom(reportButton),
.close]
}
```

替换界面指定区域视图

当调整按钮无法满足结构性修改需求时，可使用 `replace` 接口整体替换指定区域的视图。内部通过 `AudienceNode` 枚举定义了 5 个支持替换的区域，可以结合开头的“页面结构示意图”理解：

`AudienceNode`	说明
`liveInfo`	左上角的主播与房间信息展示区域。
`topRightButtons`	右上角的系统控制按钮区域。
`networkInfo`	网络状态指示区域。
`bottomRightBar`	右下角的业务操作栏区域。
`barrageInput`	左下角的弹幕输入框触发区域。

布局规则

版权所有：腾讯云计算（北京）有限责任公司

`replace` 接口会将自定义视图放入指定的 `slot` 区域。视图的位置由框架控制，开发者无需设置。视图的尺寸由其自身决定，推荐使用以下两种方式之一声明尺寸：

方式 1：使用内部约束链（推荐）

确保子视图的约束形成完整链条，从而自动撑开父视图。

```
class MyInfoView: UIView {
    init() {
        super.init(frame: .zero)
        let label = UILabel()
        label.text = "直播中"
        addSubview(label)
        label.snp.makeConstraints { make in
            make.edges.equalToSuperview().inset(12)  //由子视图撑开视图
        }
    }
}
```

方式 2：重写 intrinsicContentSize

通过重写属性直接指定视图的固定尺寸。

```
class MyInfoView: UIView {
    override var intrinsicContentSize: CGSize {
        CGSize(width: 200, height: 44)
    }
}
```

实现步骤

- **步骤 1**：创建自定义视图对象，确保符合上文所述的布局规则。
- **步骤 2**：实现 `AudienceViewDelegate` 回调，在相应的回调方法中调用 `AudienceLiveView` 组件的 `replace` 接口，传入需要替换的节点枚举和新建的自定义视图。

```
func audienceView(_ audienceView: AudienceView,
                  onCreateLiveView liveView: AudienceLiveView,
                  for liveInfo: LiveInfo) {
    // 步骤 1：初始化符合布局规则的自定义视图
    let customInfoView = MyInfoView()
```

```
    customInfoView.backgroundColor = .darkGray

    // 步骤 2: 调用替换接口更新特定节点
    liveView.replace(node: .liveInfo, with: customInfoView)
}
```

绑定事件与触发逻辑

替换节点后，开发者需自行接管该视图的交互事件。可在事件回调中执行专属业务逻辑，也可以使用 `perform(action:)` 方法快速触发内置逻辑。支持的 `AudienceAction` 包括：展示礼物面板（`.showGiftPanel`）、展示观众列表（`.showAudienceList`）等。

实现步骤

- **步骤 1**：为自定义视图绑定事件，使用 `addTarget` 或手势识别器为视图添加点击事件。
- **步骤 2**：触发内置逻辑或执行业务代码，在事件回调中调用 `perform` 方法传入 `AudienceAction` 枚举，或执行其他业务代码。

```
func audienceView(_ audienceView: AudienceView,
                  onCreateLiveView liveView: AudienceLiveView,
                  for liveInfo: LiveInfo) {
    let btn = MyGiftButton()
    btn.liveView = liveView
    liveView.bottomItems = [.custom(btn)]
}

class MyGiftButton: UIButton {
    weak var liveView: AudienceLiveView?  // 弱引用

    override init(frame: CGRect) {
        super.init(frame: frame)
        setImage(UIImage(named: "custom_gift"), for: .normal)
        addTarget(self, action: #selector(onTap), for: .touchUpInside)
    }
    required init?(coder: NSCoder) { fatalError() }

    @objc func onTap() {
        // 弹出默认礼物面板
        liveView?.perform(.showGiftPanel)
        // 或者触发自定义逻辑
```

```
        // presentCustomPanel()
    }
}
```

深度定制业务弹窗

当使用 `perform` 触发的内置默认面板无法满足具体的业务需求时，开发者可以彻底接管这部分逻辑，基于底层数据 `Core SDK` 构建全新的业务面板。

核心思路

开发者可使用底层数据接口 `AtomicXCore` 获取房间、用户及状态数据，完全自主地完成自定义视图的搭建与交互绑定。在完成自定义控制器的构建后，建议使用内部封装的 `AtomicPopover` 组件将其弹出，以确保您的自定义面板也能获得与 `SDK` 内置面板一致的丝滑手势拦截与平滑动画。

实现步骤

- **步骤 1：**构建自定义业务视图。创建一个独立的 `UIView`，并在其内部引入 `AtomicXCore` 接口，用于拉取或监听底层业务数据以驱动 UI 更新。
- **步骤 2：**配置弹窗容器参数。实例化弹窗配置对象，并指定弹出位置、占用高度与动画类型。
- **步骤 3：**弹出自定义视图。实例化您的自定义视图，将其作为 `contentView` 传入 `AtomicPopover` 容器，并通过系统方法进行展示。

```
import AtomicXCore // 引入底层数据接口进行业务开发

 // 步骤1：构建自定义观众列表视图
 class CustomAudienceListView: UIView {
     override init(frame: CGRect) {
         super.init(frame: frame)
         self.backgroundColor = .white
         setupUI()
         bindLiveData()
     }

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

     private func setupUI() {
         // 在此添加您的自定义 UI 控件，例如展示观众头像、用户等级等
     }
```

```
    private func bindLiveData() {
        // 使用 AtomicXCore 提供的核心接口获取当前房间状态或用户数据
        // 拿到核心数据后刷新上方构建的自定义 UI
    }
}

// 示例场景：从屏幕底部向上滑出一个高度占屏幕一半的完全自定义面板
func presentBusinessPanel(from parentViewController: UIViewController)
{
    // 步骤2：配置弹窗容器参数
    let config = AtomicPopover.AtomicPopoverConfig(
        position: .bottom,
        height: .ratio(0.5),
        animation: .slideFromBottom
    )

    // 步骤3：弹出自定义视图
    let audienceListView = CustomAudienceListView()
    let popover = AtomicPopover(contentView: audienceListView,
configuration: config)

    parentViewController.present(popover, animated: true)
}
```

请参考以下文档，使用 `AtomicXCore` 接口实现自定义功能面板页

功能描述	参考文档
实现观众连线管理面板：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。	观众连线
实现主播跨房连线面板：连线主播互动管理，发起 / 接受 / 拒绝连线。	主播连线和 PK
实现观众列表：统计观众数量，监听观众进出事件。	观众列表
实现音频特效面板：变声（童声 / 男声）、混响（KTV 等）、耳返调节，实时切换特效。	音效

添加自定义悬浮挂件

复杂的直播场景常需要在视频画面上方悬浮展示活动图标或互动贴纸。这类需要定位于视频层之上、且独立于基础布局的视图，应统一添加到 overlayView 图层中。

在实际业务中，悬浮挂件通常作为某个活动面板的入口。建议将其与前文介绍的 AtomicPopover 组件结合使用：为挂件绑定点击事件，并在触发后弹出深度定制的业务弹窗。

挂件生命周期管理

结合视图的生命周期回调，悬浮挂件的添加与管理时机取决于具体的业务需求：

- 常驻型挂件（例如直播间固定的活动入口）：可以直接在 `onCreateLiveView` 回调中将其添加至 `overlayView` 。
- 动态型挂件（例如天降红包、临时弹出商品卡片）：应在接收到业务信令后，获取 `liveViewDidAppear` 记录的当前活跃视图实例，再向其 `overlayView` 中动态添加控件。
- 挂件与弹窗销毁：对于动态弹出的业务面板或挂件，可以在 `liveViewDidDisappear` 回调中执行视图的 `removeFromSuperview` 操作。这能有效避免用户滑动切房时，因视图复用导致的 UI 状态异常。

实现步骤

- **步骤 1**：创建挂件视图并开启交互。实例化悬浮控件，设置其尺寸与位置。
- **步骤 2**：绑定点击事件。为挂件添加手势识别器，用于响应观众的点击操作。
- **步骤 3**：添加至覆盖图层并联动弹窗。将挂件添加至 `overlayView` 中，并在点击回调中调用前文定义的自定义弹窗逻辑。

```
import UIKit
import TUILiveKit

class LiveRoomController: UIViewController {
    private weak var currentLiveView: AudienceLiveView? // 在AudienceViewDelegate中获取当前的liveView

    // 示例场景：在画面左上角悬浮显示红包挂件，点击后联动弹出前文定义的业务面板
    func addRedPacketWidget() {
        // 步骤 1：创建挂件视图并开启交互
        let redPacketWidget = UIImageView(image: UIImage(named: "red_packet_icon"))
        redPacketWidget.frame = CGRect(x: 15, y: 120, width: 60, height: 60)
        redPacketWidget.isUserInteractionEnabled = true

        // 步骤 2：绑定点击事件
        let tapGesture = UITapGestureRecognizer(target: self, action: #selector(handleRedPacketClick))
```

版权所有：腾讯云计算（北京）有限责任公司

```
        redPacketWidget.addGestureRecognizer(tapGesture)

        // 步骤 3: 添加至覆盖图层
        currentLiveView?.overlayView.addSubview(redPacketWidget)
    }

    @objc func handleRedPacketClick() {
        // 在此处调用前文写好的 presentBusinessPanel 方法，弹出深度定制的业务视
图
        // presentBusinessPanel(from: self)
    }
}
```

常见问题

添加的自定义按钮为何点击无响应?

使用 `.custom()` 传入自定义视图后，点击视图却无法触发绑定的事件。

排查建议：

- 检查视图尺寸与内部约束（高频原因）：这是最容易被忽略的问题。如果您传入了一个自定义的容器视图（例如替换了整个 `bottomRightBar` ），请务必确保内部子控件形成了完整的约束链以撑开父视图，或者为主视图重写了 `intrinsicContentSize` 。如果父视图实际尺寸为 0，即使内部按钮在屏幕上可见，点击事件也会因超出父视图边界而被系统直接丢弃。
- 检查交互属性：检查 `isUserInteractionEnabled` 属性。如果是 `UIImageView` 或普通的 `UIView` 容器，系统默认该属性为 `false` ，必须手动将其设置为 `true` 。
- 检查手势作用域：如果自定义视图（或其子控件）的布局超出了其直接父容器的边界，超出部分的点击事件将无法响应。
- 检查事件绑定与目标（ `Target` ）：确认是否正确添加了 `UITapGestureRecognizer` 或调用了 `addTarget` 方法。另外，由于视图是在代理回调中动态生成的，请确保事件绑定的目标对象（如外部的 `controller` ）在视图存活期间未被释放。

动态隐藏或显示操作按钮?

在实际业务中，可能需要根据房间状态（例如带货期间隐藏连麦按钮），动态调整底部或顶部的工具栏。

实现方案 `AudienceLiveView` 的 `bottomItems` 和 `topRightItems` 属性支持响应式更新。只需组装一个新的按钮数组并重新赋值， `SDK` 内部会自动触发视图刷新，无需手动重绘。

排查建议：

在接收到信令并动态更新视图时，务必确保更新的是当前正在展示的视图实例。请使用 `liveViewDidAppear` 回调中记录的活跃实例，切勿误修改处于“预加载”状态的相邻房间视图。

替换视图后出现尺寸异常?

替换指定区域的视图后，如果发现 UI 压缩或超出屏幕，通常是由于约束不完整导致的。

排查建议：

请确保主视图内部的子控件形成了完整的约束链，或者为主视图重写了 `intrinsicContentSize` 属性。在滑动容器中，视图尺寸的自适应尤为重要，约束不完整极易导致视图在滑动切换时发生重叠或变形。

使用 replace 接口替换的视图在滑动时不显示？

在使用 `replace(node:with:)` 接口替换指定节点后，发现当开始滑动后当前屏幕上替换的自定义视图消失了，并且之后在滑动过程中一直不显示。这通常是因为开发者在不同的直播间视图中，错误地复用了同一个自定义视图实例。

原因剖析：

由于 `AudienceView` 是一个滑动容器，为了保证滑动的流畅性，系统会提前预加载接下来的相邻房间。这意味着 `onCreateLiveView` 代理方法会被提前调用。在 `iOS` 中，一个 `UIView` 实例同一时刻只能有一个父视图。如果您传入了一个共享的全局视图实例，当预加载下一个房间时，该视图就会被系统从当前可视房间中强制移除，并添加到尚未展示的预加载房间中，从而导致当前屏幕上看不到该视图。

错误示例（请避免这样使用）：

```
// ✕ 错误做法：在外部持有一个共享的视图实例
let sharedBrandView = MyBrandView()

public func audienceView(_ audienceView: AudienceView, onCreateLiveView
liveView: AudienceLiveView, for liveInfo: LiveInfo) {
    // ⚠ 警告：当预加载下一个 liveView 时，sharedBrandView 会被从当前屏幕拔出，
塞进下一个不可见的 liveView 中
    liveView.replace(node: .liveInfo, with: sharedBrandView)
}
```

正确做法：

请确保在 `onCreateLiveView` 回调中，每次都为新的 `AudienceLiveView` 创建一个全新的自定义视图实例。

```
public func audienceView(_ audienceView: AudienceView, onCreateLiveView
liveView: AudienceLiveView, for liveInfo: LiveInfo) {
    // ✅ 正确做法：每次触发回调时，都实例化一个全新的自定义视图
    let newBrandView = MyBrandView()
    liveView.replace(node: .liveInfo, with: newBrandView)
}
```

观众核心页面（Android）

最近更新时间：2026-05-14 16:18:31

`AudienceView` 观众端核心 UI 组件。通过该组件，开发者可以快速搭建基础的观众直播界面。本文档将按照从基础按钮调整到复杂视图替换的顺序，指导开发者完成观众端界面的按需定制。

页面结构示意图

准备工作

在开始调整观众端界面前，请先参考 观众观看 完成观众进房的主流程搭建。

功能概览

与主播端直接创建并使用视图的结构不同，观众端由于需要支持上下滑动无缝切换直播间， `AudienceView` 是一个滑动容器。

在观众端，界面定制不是直接在 `AudienceView` 上进行的，而是作用于它内部的单房间视图 `AudienceLiveView` 。滑动容器在工作时会动态创建和展示这些单房间视图，开发者需要通过 `AudienceViewListener` 协议，在以下核心生命周期回调中获取到对应房间的视图实例（即回调参数中的 `liveView` ），并在正确的时机对其进行定制与资源管理：

方法	描述

`onCreateLiveView(audienceView: AudienceLiveView, liveInfo: LiveInfo)`	**视图创建回调**。此回调会向外传递刚刚实例化的 `AudienceLiveView` 。 **适用于提前确定的静态样式定制**。由于容器会提前预加载下个房间，该回调触发时视图可能尚未显示在屏幕上。通常在这里通过 `liveView` 执行替换内置组件、配置固定按钮等一次性布局操作。
`onLiveViewDidAppear(audienceView: AudienceLiveView, liveInfo: LiveInfo)`	**视图展示回调**。此回调会向外传递当前真正显示在屏幕上的 `AudienceLiveView` 。 **适用于依赖实时状态或信令的动态调整**。通常在这里记录当前活跃的 `liveView` 实例，以便在收到业务信令（如商品上架通知）时，精准定向更新当前屏幕上的 UI；或在此处开启房间专属的定时器与动效。
`onLiveViewDidDisappear(audienceView: AudienceLiveView, liveInfo: LiveInfo)`	**视图隐藏回调**。当观众滑出该房间或关闭界面时触发。 **适用于状态重置与资源清理**。通常在这里清空活跃的 `liveView` 记录，销毁在该房间内弹出的自定义业务面板、停止动效或清理定时器。

单房间视图 `AudienceLiveView` 提供的核心自定义接口与属性如下：

方法/属性	描述
`topRightItems`	用于灵活配置直播间右上角的按钮集合，支持自由添加自定义按钮或调整内置按钮的布局。
`bottomItems`	用于灵活配置直播间底部的按钮集合，支持自由添加自定义按钮或调整内置按钮的布局。
`replace(node: AudienceNode, view: View?)`	用于将指定位置的默认组件（如顶部信息区、底部操作栏），替换为开发者自定义的全新视图。
`overlayView`	专属挂件图层，方便开发者自由添加需要在视频画面上方悬浮展示的全局业务 UI。
`perform(action:AudienceAction)`	用于在自定义视图中直接触发内置默认逻辑，例如显示默认观众列表，显示默认连麦管理面板等。

快速开始

下面示例快速搭建一个带货直播间观看页。主要流程包括：

- 隐藏不需要的连麦按钮并添加购物车按钮。
- 模拟接收商品上架通知并弹出商品卡片。
- 最后将点击事件路由至商品列表页。

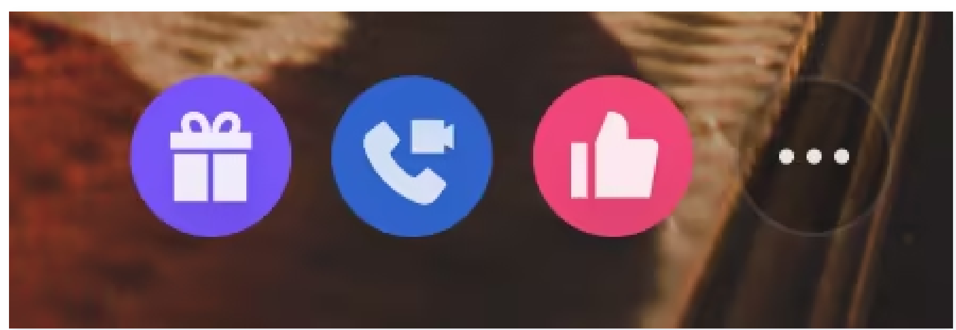
```
import android.os.Bundle
import android.view.Gravity
import android.widget.FrameLayout
import android.widget.ImageView
import androidx.appcompat.app.AppCompatActivity
import com.trtc.uikit.livekit.features.audienceview.AudienceLiveView
import com.trtc.uikit.livekit.features.audienceview.AudienceView
import com.trtc.uikit.livekit.features.audienceview.AudienceViewDefine
import
com.trtc.uikit.livekit.features.audienceview.AudienceViewDefine.Audience
BottomItem
import io.trtc.tuikit.atomicx.common.util.ScreenUtil.dip2px
import io.trtc.tuikit.atomicxcore.api.live.LiveInfo

class AudienceActivity : AppCompatActivity(),
AudienceViewDefine.AudienceViewListener {
    private var currentLiveView: AudienceLiveView? = null

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

        // 初始化容器 View
        val audienceView = AudienceView(this)
        setContentView(audienceView)

        // 注册监听并初始化
        audienceView.addListener(this)
        audienceView.init(this, "your_room_id")
    }

    // 业务方法：展示商品列表页
    private fun showProductListPanel() {
        println("展示商品列表页...")
    }

    // 业务方法：模拟收到 IM 商品推送通知
    fun onReceiveProductPushMessage() {
        // 必须作用于当前正在展示的 active 视图
        val liveView = currentLiveView ?: return
```

```
        val productCard = AudienceProductCardView(this)
        productCard.setOnClickListener { showProductListPanel() }

        val params = FrameLayout.LayoutParams(dip2px(150f),
FrameLayout.LayoutParams.WRAP_CONTENT).apply {
            gravity = Gravity.BOTTOM or Gravity.END
            rightMargin = dip2px(12f)
            bottomMargin = dip2px(100f)
        }
        liveView.overlayView.addView(productCard, params)
    }

    // --- AudienceViewListener 回调实现 ---

    override fun onCreateLiveView(liveView: AudienceLiveView, liveInfo:
LiveInfo) {
        // 配置底部操作栏：隐藏连麦功能，并添加自定义“购物车”按钮
        val shopCartBtn = ImageView(this).apply {
            setOnClickListener { showProductListPanel() }
        }

        liveView.bottomItems = listOf(
            AudienceBottomItem.Gift,
            AudienceBottomItem.Like,
            AudienceBottomItem.Custom(shopCartBtn)
        )
    }

    override fun onLiveViewDidAppear(liveView: AudienceLiveView,
liveInfo: LiveInfo) {
        currentLiveView = liveView
    }

    override fun onLiveViewDidDisappear(liveView: AudienceLiveView,
liveInfo: LiveInfo) {
        if (currentLiveView === liveView) {
            currentLiveView = null
        }
    }
```

```
    override fun onLiveEnded(roomId: String, ownerName: String,
ownerAvatarUrl: String) {}
    override fun onClickFloatWindow() {}
}

class AudienceProductCardView(context: android.content.Context) :
android.view.View(context) {
    // 自定义的商品卡片
}
```

调整底部操作按钮

底部工具栏是观众进行互动的核心区域。目前默认提供礼物、连麦、点赞、更多等按钮。开发者可通过 `bottomItems` 属性灵活增删内置功能，或通过 `AudienceBottomItem.Custom(View)` 插入自定义按钮。



实现步骤

步骤 1：准备自定义按钮视图。按需创建自定义按钮视图对象。
步骤 2：更新底部按钮数组。实现 `AudienceViewListener` **回调，在相应的回调方法中把组装好的包含** `AudienceBottomItem` **枚举的数组重新赋值给视图属性。**

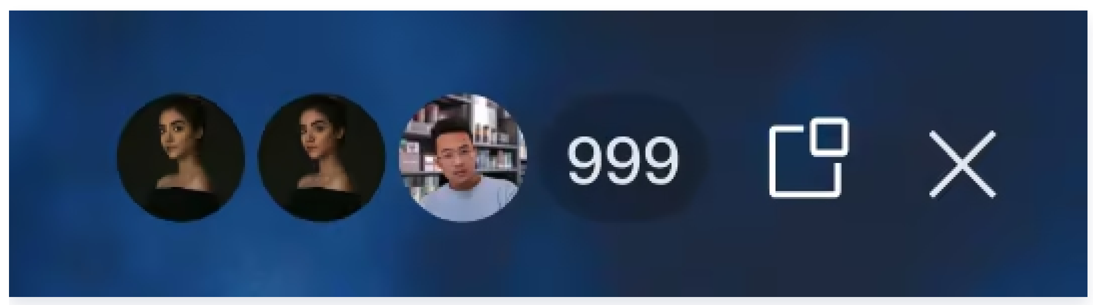
```
override fun onCreateLiveView(audienceView: AudienceLiveView, liveInfo:
LiveInfo) {
    // 步骤 1：准备自定义按钮视图
    val shopButton = ImageView(audienceView.context).apply {
        setImageResource(R.drawable.shop_cart)
    }

    // 步骤 2：更新底部按钮数组，保留送礼，隐藏连麦，新增商品按钮
    audienceView.bottomItems = listOf(
        AudienceBottomItem.Gift,
        AudienceBottomItem.Custom(shopButton)
```

版权所有：腾讯云计算（北京）有限责任公司

```
    )
}
```

调整顶部操作按钮

顶部区域展示房间信息与关键操作。默认提供观众人数、悬浮窗、退出三个按钮。开发者可通过 `topRightItems` 属性精简或增加控制按钮。



实现步骤

步骤 1：准备自定义按钮视图。按需创建自定义按钮视图对象。
步骤 2：更新顶部按钮数组。实现 `AudienceViewListener` 回调，在相应的回调方法中将枚举项或自定义视图赋值给 `topRightItems` 属性。

```
override fun onCreateLiveView(audienceView: AudienceLiveView, liveInfo:
LiveInfo) {
    // 步骤 1：准备自定义按钮视图
    val reportButton = ImageView(audienceView.context).apply {
        setImageResource(R.drawable.report_btn)
    }

    // 步骤 2：更新顶部按钮数组（保留人数和关闭，新增举报）
    audienceView.topRightItems = listOf(
        AudienceViewDefine.AudienceTopRightItem.AudienceCount,
        AudienceViewDefine.AudienceTopRightItem.Custom(reportButton),
        AudienceViewDefine.AudienceTopRightItem.Close
    )
}
```

替换界面指定区域视图

当调整按钮无法满足结构性修改需求时，可使用 `replace` 接口整体替换指定区域的视图。内部通过 `AudienceNode` 枚举定义了 5 个支持替换的区域，可以结合开头的“页面结构示意图”理解：

AudienceNode	说明
LIVE_INFO	左上角的主播与房间信息展示区域。
TOP_RIGHT_BUTTONS	右上角的系统控制按钮区域。
NETWORK_INFO	网络状态指示区域。
BOTTOM_RIGHT_BAR	右下角的业务操作栏区域。
BARRAGE_INPUT	左下角的弹幕输入框触发区域。

布局规则

`replace` 接口会将自定义视图放入指定的区域。视图的位置由框架控制，开发者无需设置。视图的尺寸由其自身决定，推荐使用以下两种方式之一声明尺寸：

方式 1：使用内部约束链（推荐）

确保子视图的约束形成完整链条，从而自动撑开父视图。

```
class MyInfoView(context: Context) : FrameLayout(context) {
    init {
        val label = TextView(context).apply {
            text = "直播中"
            setPadding(dip2px(12f), dip2px(12f), dip2px(12f),
dip2px(12f))
        }
        val params = LayoutParams(LayoutParams.WRAP_CONTENT,
LayoutParams.WRAP_CONTENT)
        addView(label, params)
    }
}
```

方式 2：重写 onMeasure 或固定 LayoutParams

通过重写自定义 `View` 的 `onMeasure` 方法，或者在初始化时赋予明确宽高的 `LayoutParams` 来决定视图尺寸。

```
class MyInfoView(context: Context) : FrameLayout(context) {
    init {
        // 赋予固定的宽高属性
```

```
            layoutParams = LayoutParams(dip2px(200f), dip2px(44f))
            setBackgroundColor(Color.DKGRAY)
        }
    }
```

实现步骤

- **步骤 1**：创建自定义视图对象，确保符合上文所述的布局规则。
- **步骤 2**：实现 `AudienceViewListener` 回调，在相应的回调方法中调用 `AudienceLiveView` 组件的 `replace` 接口，传入需要替换的节点枚举和新建的自定义视图。

```
override fun onCreateLiveView(liveView: AudienceLiveView, liveInfo:
LiveInfo) {
    // 步骤 1：初始化符合布局规则的自定义视图
    val customInfoView = MyInfoView(liveView.context)

    // 步骤 2：调用替换接口更新特定节点
    liveView.replace(AudienceViewDefine.AudienceNode.LIVE_INFO,
customInfoView)
}
```

绑定事件与触发逻辑

替换节点后，开发者需自行接管该视图的交互事件。可在事件回调中执行专属业务逻辑，也可以使用 `perform` 方法快速触发内置逻辑。支持的 `AudienceAction` 包括：展示礼物面板（ `SHOW_GIFT_PANEL` ）、展示观众列表（ `SHOW_AUDIENCE_LIST` ）等。

实现步骤

- **步骤 1**：为自定义视图绑定事件，使用 `setOnClickListener` 或手势识别器为视图添加点击事件。
- **步骤 2**：触发内置逻辑或执行业务代码，在事件回调中调用 `perform` 方法传入 `AudienceAction` 枚举，或执行其他业务代码。

```
override fun onCreateLiveView(audienceView: AudienceLiveView, liveInfo:
LiveInfo) {
    val btn = MyGiftButton(audienceView.context, audienceView)
    audienceView.bottomItems = listOf(AudienceBottomItem.Custom(btn))
}
```

```
class MyGiftButton(context: Context, private val liveView:
AudienceLiveView) :
androidx.appcompat.widget.AppCompatImageView(context) {
    init {
        setImageResource(R.drawable.custom_gift)
        setOnClickListener {
            // 弹出默认礼物面板

liveView.perform(AudienceViewDefine.AudienceAction.SHOW_GIFT_PANEL)
            // 或者触发自定义逻辑
            // presentCustomPanel()
        }
    }
}
```

深度定制业务弹窗

当使用 `perform` 触发的内置默认面板无法满足具体的业务需求时，开发者可以彻底接管这部分逻辑，基于底层数据 `AtomicXCore` 构建全新的业务面板。

核心思路

开发者可使用底层数据接口 `AtomicXCore` 获取房间、用户及状态数据，完全自主地完成自定义视图的搭建与交互绑定。在完成自定义控制器的构建后，建议使用内部封装的 `AtomicPopover` 组件将其弹出，以确保您的自定义面板也能获得与 `SDK` 内置面板一致的丝滑手势拦截与平滑动画。

实现步骤

- **步骤 1：**构建自定义业务视图。创建一个独立的 `View`，并在其内部引入底层核心数据接口，用于拉取或监听业务数据以驱动 UI 更新。
- **步骤 2：**实例化弹窗容器。实例化 `AtomicPopover` 对象，并指定其弹出位置（`BOTTOM` 或 `CENTER`）。
- **步骤 3：**配置并展示自定义视图。设置弹窗的高度模式（如按屏幕比例），将您的自定义视图通过 `setContent` 传入容器，最后调用 `show()` 进行展示。

```
import android.content.Context
import android.graphics.Color
import android.widget.FrameLayout
import io.trtc.tuikit.atomicx.widget.basicwidget.popover.AtomicPopover

// 步骤1：构建自定义业务视图（如：观众列表）
class CustomAudienceListView(context: Context) : FrameLayout(context) {
```

```
    init {
        setBackgroundColor(Color.WHITE)
        setupUI()
        bindLiveData()
    }

    private fun setupUI() {
        // 在此添加您的自定义 UI 控件，例如展示观众头像、用户等级等列表
    }

    private fun bindLiveData() {
        // 使用 AtomicXCore 提供的核心接口获取当前房间状态或用户数据
        // 例如：LiveAudienceStore.create(liveID)...
        // 拿到核心数据后刷新上方构建的自定义 UI
    }
}

// 示例场景：从屏幕底部向上滑出一个高度占屏幕 50% 的完全自定义面板
fun presentBusinessPanel(context: Context) {
    // 步骤2：实例化弹窗容器，指定从底部弹出
    val popover = AtomicPopover(context,
AtomicPopover.PanelGravity.BOTTOM)

    // 步骤3：配置并展示自定义视图
    // 设置面板高度为屏幕高度的 50%（也可以使用 WrapContent 自适应高度）
    popover.setPanelHeight(AtomicPopover.PanelHeight.Ratio(0.5f))

    // 实例化自定义业务视图
    val audienceListView = CustomAudienceListView(context)

    // 将视图填入 Popover 并展示
    popover.setContent(audienceListView)
    popover.show()
}
```

请参考以下文档，使用 `AtomicXCore` 接口实现自定义功能面板页

功能描述	参考文档

实现观众连线管理面板：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。	观众连线
实现主播跨房连线面板：连线主播互动管理，发起 / 接受 / 拒绝连线。	主播连线和 PK
实现观众列表：统计观众数量，监听观众进出事件。	观众列表
实现音频特效面板：变声（童声 / 男声）、混响（KTV 等）、耳返调节，实时切换特效。	音效

添加自定义悬浮挂件

复杂的直播场景常需要在视频画面上方悬浮展示活动图标或互动贴纸。这类需要定位于视频层之上、且独立于基础布局的视图，应统一添加到 overlayView 图层中。
在实际业务中，悬浮挂件通常作为某个活动面板的入口。建议将其与前文介绍的 AtomicPopover 组件结合使用：为挂件绑定点击事件，并在触发后弹出深度定制的业务弹窗。

挂件生命周期管理

结合视图的生命周期回调，悬浮挂件的添加与管理时机取决于具体的业务需求：

- 常驻型挂件（例如直播间固定的活动入口）：可以直接在 `onCreateLiveView` 回调中将其添加至 `overlayView` 。
- 动态型挂件（例如天降红包、临时弹出商品卡片）：应在接收到业务信令后，获取 `onLiveViewDidAppear` 记录的当前活跃视图实例，再向其 `overlayView` 中动态添加控件。
- 挂件与弹窗销毁：对于动态弹出的业务面板或挂件，可以在 `liveViewDidDisappear` 回调中执行视图的 `removeFromSuperview` 操作。这能有效避免用户滑动切房时，因视图复用导致的 UI 状态异常。

实现步骤

- **步骤 1**：创建挂件视图并开启交互。实例化悬浮控件，设置其尺寸与位置。
- **步骤 2**：绑定点击事件。为挂件设置 `setOnClickListener` ，用于响应观众的点击操作。
- **步骤 3**：添加至覆盖图层并联动弹窗。将挂件添加至 `overlayView` 中，并在点击回调中调用前文定义的自定义弹窗逻辑。

```
class LiveRoomActivity : AppCompatActivity() {
    // 在 AudienceViewListener 中获取当前的 liveView
    private var currentLiveView: AudienceLiveView? = null

    // 示例场景：在画面左上角悬浮显示红包挂件，点击后联动弹出业务面板
    fun addRedPacketWidget() {
        val activeView = currentLiveView ?: return
```

```
        // 步骤 1 & 2：创建挂件视图并开启交互
        val redPacketWidget = ImageView(this).apply {
            setImageResource(R.drawable.red_packet_icon)
            setOnClickListener { handleRedPacketClick() }
        }

        val params = FrameLayout.LayoutParams(dip2px(60f),
dip2px(60f)).apply {
            gravity = Gravity.TOP or Gravity.START
            topMargin = dip2px(120f)
            leftMargin = dip2px(15f)
        }

        // 步骤 3：添加至覆盖图层
        activeView.overlayView.addView(redPacketWidget, params)
    }

    private fun handleRedPacketClick() {
        // 在此处调用前文写好的 presentBusinessPanel 方法，弹出深度定制的业务视
图
        // presentBusinessPanel(this)
    }
}
```

常见问题

添加的自定义按钮为何点击无响应?

使用 `AudienceBottomItem.Custom()` 传入自定义视图后，点击视图却无法触发绑定的事件。

排查建议：

- **检查点击事件绑定（高频原因）：** `Android` 中默认的 `View` 往往不具备点击特性，请确保对传入的控件调用了 `setOnClickListener` ，或者在 `XML` 中/代码中将其 isClickable 设为了 true。
- **检查视图尺寸与边界：** 如果您的容器并没有声明具体的宽高（或没有通过子 `View` 撑起 `WRAP_CONTENT` ），导致父布局尺寸为 0，即便内部子按钮画出了边界外，点击事件也会因为超出父视图 `Rect` 范围而被系统的 `Touch` 机制丢弃。
- **检查父级拦截：** 确认外部是否有一层包裹图层在 `onInterceptTouchEvent` 中拦截了事件。

动态隐藏或显示操作按钮?

在实际业务中，可能需要根据房间状态（例如带货期间隐藏连麦按钮），动态调整底部或顶部的工具栏。

实现方案 `AudienceLiveView` 的 `bottomItems` 和 `topRightItems` 属性支持响应式更新。只需组装一个新的按钮数组并重新赋值，`SDK` 内部会自动触发视图刷新，无需手动重绘。需要注意的是在动态更新视图时，务必确保更新的是当前正在展示的视图实例。请使用 `onLiveViewDidAppear` 回调中记录的活跃实例，切勿误修改处于“预加载”状态的相邻房间视图。

使用 replace 接口替换的视图在滑动时不显示？

在使用 `replace(node, customView)` 接口替换指定节点后，发现当开始滑动后程序抛出 `illegalStateException: The specified child already has a parent...` 崩溃，或者屏幕上替换的自定义视图消失了。这通常是因为开发者在不同的直播间视图中，错误地复用了同一个自定义 `View` 实例。

原因剖析：

由于 `AudienceView` 是一个滑动容器，为了保证滑动的流畅性，系统会提前预加载接下来的相邻房间。这意味着 `onCreateLiveView` 代理方法会被提前调用。在 `Android` 中，一个 `View` 实例同一时刻只能拥有一个 `Parent` 。如果您传入了一个共享的全局 `View` 实例，当预加载下一个房间时，将其 `addView` 到新房间前，如果不手动 `removeView` 会直接导致崩溃；即使框架底层处理了强制拔出，也会导致该 `View` 从当前可视房间被剥离，从而引发显示异常。

错误示例（请避免这样使用）：

```
// ❌ 错误做法：在外部持有一个共享的视图实例
val sharedBrandView = MyBrandView(context)

override fun onCreateLiveView(audienceView: AudienceLiveView, liveInfo: LiveInfo) {
    // ⚠️ 警告：当预加载下一个 liveView 时，复用同一个 View 实例会导致 Parent 冲突！
    audienceView.replace(AudienceNode.LIVE_INFO, sharedBrandView)
}
```

正确做法：

请确保在 `onCreateLiveView` 回调中，每次都为新的 `AudienceLiveView` 创建一个全新的自定义视图实例。

```
override fun onCreateLiveView(audienceView: AudienceLiveView, liveInfo: LiveInfo) {
    // ✅ 正确做法：每次触发回调时，都实例化一个全新的自定义视图
    val newBrandView = MyBrandView(audienceView.context)
    audienceView.replace(AudienceNode.LIVE_INFO, newBrandView)
}
```

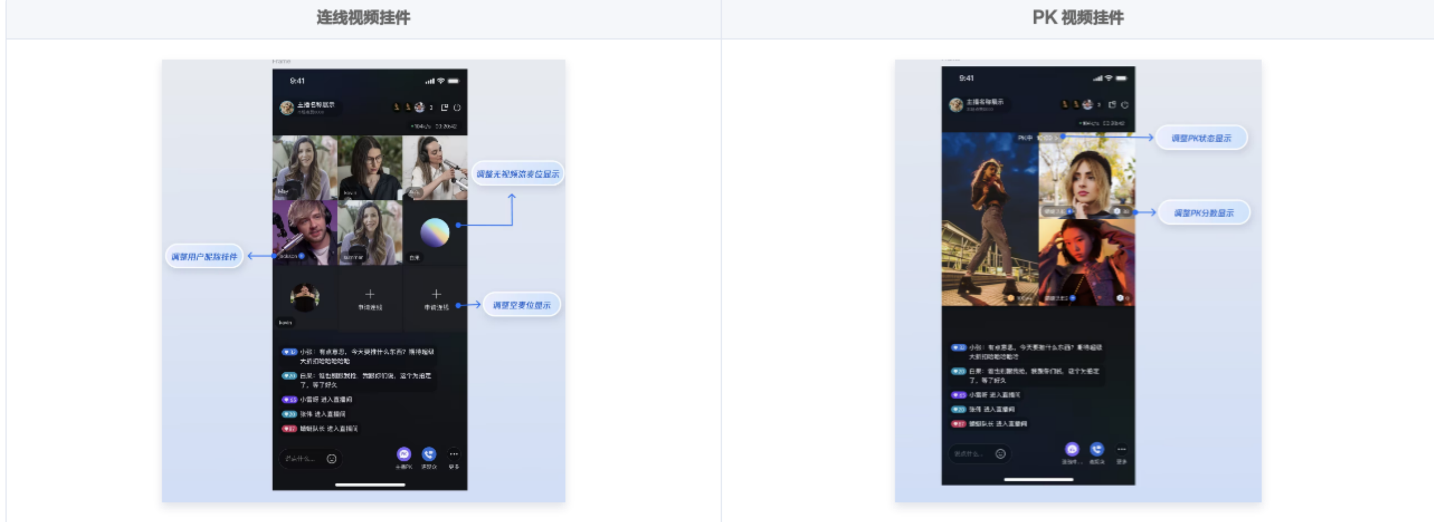
版权所有：腾讯云计算（北京）有限责任公司

直播视频组件
调整视频直播挂件 (Android)

最近更新时间：2026-05-09 18:01:12

`LiveCoreView` 是一个跨平台的视频直播核心组件，提供开播、观看、连麦和主播 PK 等关键能力。该组件通过挂件机制，支持在视频区域实时显示用户名、等级、`PK` 进度条等自定义信息。本文档指导开发者通过实现接口，在 `Android` 平台上快速定制专属的视频挂件 `UI` 。

效果展示



准备工作

在开始调整视频挂件前，请先参考 主播开播 和 观众观看 完成主流程的搭建。

核心原理

`LiveCoreView` 通过 `VideoViewAdapter` 代理支持自定义视图渲染。当业务场景发生变化（例如有人上麦、开始 `PK` ）时， `LiveCoreView` 会调用代理方法询问应该显示的视图。开发者实现对应的接口方法并返回自定义的 `View` 实例即可。

方法	描述	对应业务场景
`createCoGuestView`	创建观众连麦的挂件视图。	观众连麦、邀请上麦
`createCoHostView`	创建跨房连麦（主播连线）的挂件视图。	主播连线
`createBattleView`	创建 PK 场景中单个用户的挂件视图（例如头像、分数）。	主播 `PK`

`createBattleContainerView`	创建 PK 场景的整体容器视图（例如背景、分数 `PK` 条）。	主播 `PK`

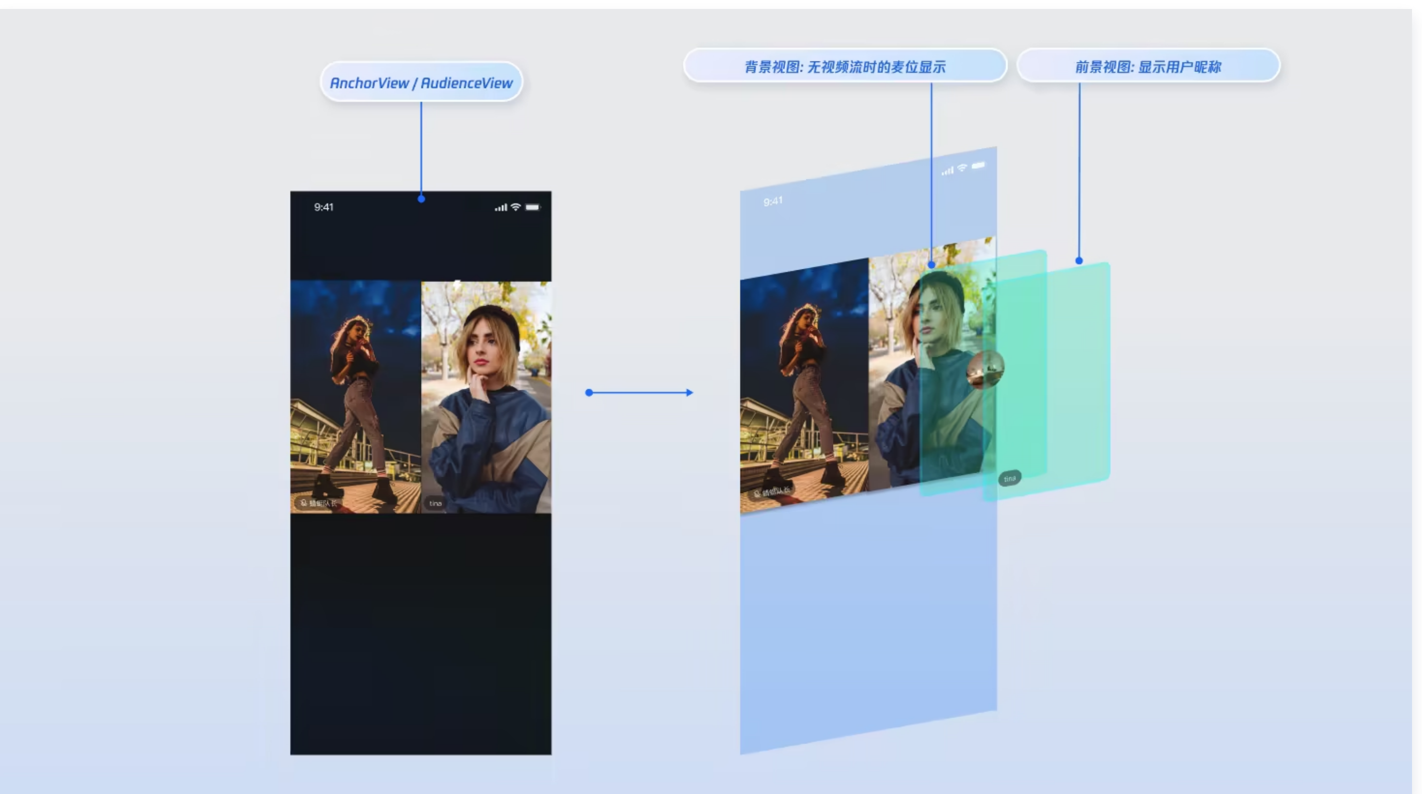
调整连线场景挂件

当观众与主播进行连线或者主播间跨房连线时，直播间界面会从单人模式切换为多人布局。此时，需要在连线的视频窗口上展示特定的用户信息，例如昵称、等级或“静音”状态提示，以便区分不同麦位。

适用场景

- 主播和观众视频连麦时，修改麦位上的用户信息（例如昵称、等级、静音图标）。
- 主播与主播跨房连线时，修改对方主播的显示样式（例如昵称、等级、静音图标）。
- 修改无视频画面时的默认背景（例如占位头像）。
- 修改空麦位时的视图实现。

视图层级示意



实现步骤

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

步骤 1. 准备自定义挂件 View

定义三个基础视图类，分别用于展示用户信息、空麦位提示和无视频流时的背景。
创建自定义的用户信息视图：

```
import android.content.Context
import android.view.View
import android.widget.FrameLayout
import android.widget.ImageView
import android.widget.TextView

class CustomInfoView(context: Context, name: String, isMuted: Boolean) :
FrameLayout(context) {
    private val nameTextView = TextView(context)
    private val muteIcon = ImageView(context)

    init {
        nameTextView.text = name
        muteIcon.visibility = if (isMuted) View.VISIBLE else View.GONE

        addView(nameTextView)
        addView(muteIcon)
        // 此处省略具体布局参数设置代码
    }
}
```

创建自定义的空麦位视图：

```
import android.content.Context
import android.widget.FrameLayout
import android.widget.ImageView
import android.widget.TextView

class EmptySeatView(context: Context) : FrameLayout(context) {
    private val addIcon = ImageView(context)
    private val addLabel = TextView(context)

    init {
        addLabel.text = "邀请连线"
        // 此处省略图片资源加载和布局参数设置代码
```

```
    }
}
```

创建自定义无视频流时的头像占位视图：

```
import android.content.Context
import android.widget.FrameLayout
import android.widget.ImageView

class CustomAvatarView(context: Context, avatarURL: String) :
FrameLayout(context) {
    private val avatarImageView = ImageView(context)

    init {
        // 实际项目中可使用 Glide 等图片加载库加载 avatarURL
        addView(avatarImageView)
        // 此处省略具体布局参数设置代码
    }
}
```

步骤 2. 实现 Adapter 逻辑

创建一个适配器类，实现 `VideoViewAdapter` 接口的方法 `createCoGuestView` （观众连线）和 `createCoHostView` （主播连线），返回刚才创建的自定义视图。

实现 `VideoViewAdapter` 中的 `createCoGuestView` 方法，返回观众连线视频挂件。

```
import android.content.Context
import android.view.View
import io.trtc.tuikit.atomicxcore.api.device.DeviceStatus
import io.trtc.tuikit.atomicxcore.api.live.SeatInfo
import io.trtc.tuikit.atomicxcore.api.view.VideoViewAdapter
import io.trtc.tuikit.atomicxcore.api.view.ViewLayer


class VideoWidgetAdapter(private val context: Context) :
VideoViewAdapter {
    override fun createCoGuestView(seatInfo: SeatInfo, viewLayer:
ViewLayer): View? {
        val isUserOnSeat = seatInfo.userInfo.userID.isNotEmpty()
```

版权所有：腾讯云计算（北京）有限责任公司

```
        return when (viewLayer) {
            ViewLayer.FOREGROUND -> {
                if (isUserOnSeat) {
                    // 非空麦位，返回自定义的前景视图
                    CustomInfoView(context, seatInfo.userInfo.userName,
seatInfo.userInfo.microphoneStatus == DeviceStatus.OFF)
                } else {
                    // 空麦位，返回自定义空麦位视图
                    EmptySeatView(context)
                }
            }
            ViewLayer.BACKGROUND -> {
                if (isUserOnSeat) {
                    // 返回自定义的背景视图（未开摄像头时显示）
                    CustomAvatarView(context,
seatInfo.userInfo.avatarURL)
                } else {
                    null
                }
            }
        }
    }
}
```

实现 `VideoViewAdapter` 中的 `createCoHostView` 方法，返回主播连线的视频挂件。

```
import android.content.Context
import android.view.View
import io.trtc.tuikit.atomicxcore.api.device.DeviceStatus
import io.trtc.tuikit.atomicxcore.api.live.SeatInfo
import io.trtc.tuikit.atomicxcore.api.view.VideoViewAdapter
import io.trtc.tuikit.atomicxcore.api.view.ViewLayer

class VideoWidgetAdapter(private val context: Context) :
VideoViewAdapter {

    override fun createCoHostView(seatInfo: SeatInfo, viewLayer:
ViewLayer): View? {
```

```
        val isUserOnSeat = seatInfo.userInfo.userID.isNotEmpty()

        return when (viewLayer) {
            ViewLayer.FOREGROUND -> {
                if (isUserOnSeat) {
                    // 返回自定义的前景视图，可以返回与观众连线不同的样式
                    CustomInfoView(context, seatInfo.userInfo.userName,
seatInfo.userInfo.microphoneStatus == DeviceStatus.OFF)
                } else {
                    // 返回自定义空麦位视图，可以返回与观众连线不同的样式
                    EmptySeatView(context)
                }
            }
            ViewLayer.BACKGROUND -> {
                if (isUserOnSeat) {
                    // 返回自定义的背景视图（未开摄像头时显示），可以返回与观众连
线不同的样式
                    CustomAvatarView(context,
seatInfo.userInfo.avatarURL)
                } else {
                    null
                }
            }
        }
    }
}
```

参数说明：

参数	类型	说明
`seatInfo`	`SeatInfo`	麦位信息对象，包含麦上用户的详细信息。
`seatInfo.userInfo.userName`	`String`	麦上用户的昵称。
`seatInfo.userInfo.avatarURL`	`String`	麦上用户的头像 URL。

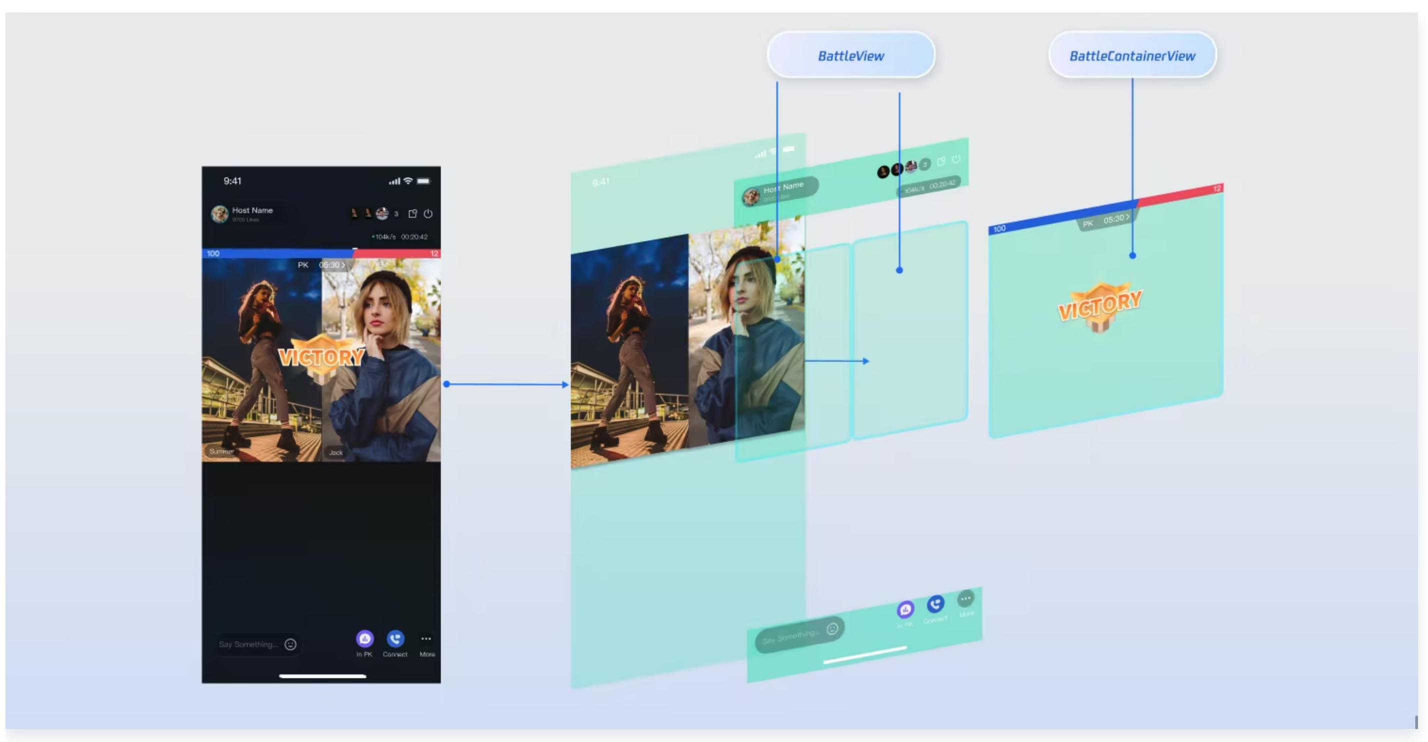
版权所有：腾讯云计算（北京）有限责任公司

`seatInfo.userInfo.microphoneStatus`	`DeviceStatus`	麦上用户的麦克风状态。
`seatInfo.userInfo.cameraStatus`	`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。

调整主播 PK 场景挂件

`PK` 是直播中互动性最强的环节。在 `PK` 过程中，画面通常根据参与人数被分割为几部分。开发者需要在画面上方或中间区域添加 `PK` 专用的 `UI` ，分数 `PK` 条、每个麦位上的分数显示、倒计时动画或“VS”特效图标，以营造紧张的竞技氛围。

视图层级示意



> **说明：**
> PK 功能依赖于连线功能，所以需要先在主播连线的情况下才能发起 PK 请求。

版权所有：腾讯云计算（北京）有限责任公司

实现步骤

步骤 1. 准备自定义 UI 组件

`PK` 场景通常需要两类视图：

- **单人挂件**：显示在每个主播窗口上（例如分数胶囊）。
- **全局容器**：覆盖在整体画面上（例如 VS 动画、倒计时）。

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

```
import android.content.Context
import android.widget.FrameLayout
import android.widget.ImageView

// 示例：单人分数条
class MyBattleScoreView(context: Context) : FrameLayout(context) {
    // 内部实现分数显示逻辑
}

// 示例：全局 VS 面板
class MyBattleContainer(context: Context) : FrameLayout(context) {
    private val battleTimeView = ImageView(context)
    // 内部实现倒计时和 VS 动画逻辑
}
```

步骤 2. 实现 Adapter 逻辑

在之前创建的 `VideoWidgetAdapter` 中，实现剩余的 `PK` 视图方法。

```
class VideoWidgetAdapter(private val context: Context) :
VideoViewAdapter {

    // 省略 createCoGuestView / createCoHostView 实现...

    // 1. 创建【PK 单人信息】挂件 （显示在每个主播视频上方）
    override fun createBattleView(seatInfo: SeatInfo): View? {
        return MyBattleScoreView(context)
    }
```

```
    // 2. 创建【PK 全局容器】挂件 （显示在整个视频区域上方）
    override fun createBattleContainerView(): View? {
        return MyBattleContainer(context)
    }
}
```

集成与生效（重要）

这是最关键的一步。您需要将实现了代理逻辑的 `VideoWidgetAdapter` 注入到直播核心流程中。
主播端集成：需在初始化外层容器前，先初始化 `LiveCoreView` 并通过 `setVideoViewAdapter` 设置适配器。

```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import android.view.ViewGroup
import com.trtc.uikit.livekit.features.anchorview.AnchorView
import io.trtc.tuikit.atomicxcore.api.live.LiveInfo
import io.trtc.tuikit.atomicxcore.api.view.LiveCoreView


class AnchorActivity : AppCompatActivity() {
    private lateinit var anchorView: AnchorView
    private lateinit var widgetAdapter: VideoWidgetAdapter

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

        // 初始化适配器对象
        widgetAdapter = VideoWidgetAdapter(this)

        // 假设此处已获取 liveInfo
        val liveInfo = LiveInfo()

        // 1. 仅传入 Context 初始化 LiveCoreView
        val videoView = LiveCoreView(this)
        // 2. 调用 setVideoViewAdapter 传入自定义适配器
        videoView.setVideoViewAdapter(widgetAdapter)

        // 3. 将核心视频组件传入外层容器
```

```
        anchorView = AnchorView(this, liveInfo, videoView)

        setContentView(anchorView, ViewGroup.LayoutParams(
            ViewGroup.LayoutParams.MATCH_PARENT,
            ViewGroup.LayoutParams.MATCH_PARENT
        ))
    }
}
```

观众端集成：观众端通过 `AudienceContainerViewDelegate` 来回调核心视图的创建时机。

```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import io.trtc.tuikit.atomicxcore.api.live.LiveInfo
import io.trtc.tuikit.atomicxcore.api.view.LiveCoreView

class AudienceActivity : AppCompatActivity(),
AudienceContainerViewDelegate {
    private lateinit var audienceView: AudienceContainerView
    private lateinit var widgetAdapter: VideoWidgetAdapter

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

        widgetAdapter = VideoWidgetAdapter(this)

        val roomId = "your_room_id"
        audienceView = AudienceContainerView(this, roomId)
        audienceView.setDelegate(this)

        setContentView(audienceView)
    }

    override fun onCreateCoreView(liveInfo: LiveInfo): LiveCoreView? {
        val view = LiveCoreView(this)
        // 注入适配器
        view.setVideoViewAdapter(widgetAdapter)
        return view
    }
```

```
}
```

进阶：获取实时业务数据

在开发 `PK` 等复杂场景时，您会发现 `SeatInfo` 只提供了基础麦位信息。如果您需要获取实时倒计时、 `PK` 分数等业务数据，就需要在您的自定义视图中对接 `AtomicXCore` 中的核心数据。

Store/Component	功能描述	API 文档
`CoGuestStore`	观众连线数据：已连线用户列表、邀请列表，申请列表等。	API 文档
`CoHostStore`	主播连线数据：已连线用户列表、邀请列表，申请列表等。	API 文档
`BattleStore`	`PK` 数据：当前 `PK` 信息、 `PK` 用户列表、 `PK` 分数列表。	API 文档

常见问题

设置了适配器，但自定义视图没有显示?

请检查设置时机是否太晚。在 `Android` 的组件装载链路中， `LiveCoreView` 的适配器赋值必须在它被传入 `AnchorView` 之前完成。如果 `AnchorView` 已经完成了初始化，它内部会直接加载 `SDK` 的默认挂件视图。
解决：请务必遵循严格的初始化顺序：先 `new LiveCoreView` > 再调用 `setVideoViewAdapter` 注入自定义 `UI` > 最后实例化 `AnchorView` 并传入配置好的 `CoreView` 。

我只想修改“连麦”视图，“PK”视图想保留默认的，怎么办?

`VideoViewAdapter` 是一种完全接管模式。一旦您设置了自定义的代理， `SDK` 默认的 `AnchorView` / `AudienceView` 内部实现的代理逻辑就会完全失效。
解决：您需要在您的代理类中实现所有相关方法。对于您不想修改的部分（例如 `PK` 视图），您可以：

1. **复制源码**：找到 `LiveKit` 源码中默认的实现类，在您的代理中返回这些默认类的实例。
2. **手动实现**：参考默认样式，快速写一个类似的简单视图返回。

自定义挂件显示了，但无法点击?

这是 `Android` 经典的事件分发冲突问题。由于前景视图（ `.FOREGROUND` ）叠加在视频渲染层之上：

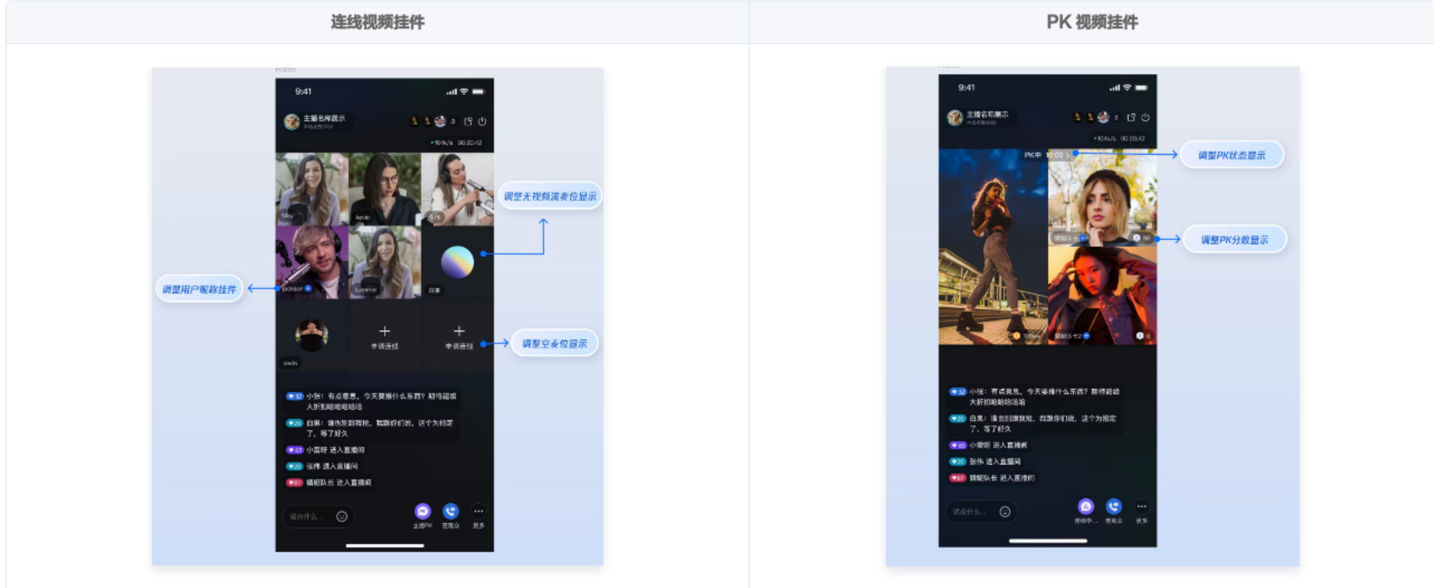
1. 请确保您返回的 `CustomView` 设置了 `isClickable = true` 或注册了 `setOnClickListener` 。
2. 如果您的挂件是一个嵌套的 `ViewGroup` （例如 `FrameLayout` ），请检查其父容器是否错误地拦截了触摸事件（例如被外层控件触发了 `onInterceptTouchEvent` 且返回了 `true` ）。

调整视频直播挂件 (iOS UIKit)

最近更新时间：2026-05-09 18:01:19

`LiveCoreView` 是一个跨平台的视频直播核心组件，提供开播、观看、连麦和主播 PK 等关键能力。它通过“挂件”机制，支持在视频区域实时显示用户名、等级、PK 进度条等自定义信息。本文档将指导您如何通过实现代理协议，在 `iOS` 平台上快速定制专属的视频挂件 UI。

效果展示



准备工作

在开始调整视频挂件前，请先参考 主播开播 和 观众观看 完成主流程的搭建。

核心原理

`LiveCoreView` 通过 `VideoViewDelegate` 代理支持自定义视图渲染。当业务场景发生变化（例如有人上麦、开始 PK）时，`LiveCoreView` 会调用代理方法询问应显示的视图。实现对应的代理方法并返回自定义的 UIView 实例即可。

方法	描述	对应业务场景
`createCoGuestView`	创建观众连麦的挂件视图	观众连麦、邀请上麦
`createCoHostView`	创建跨房连麦（主播连线）的挂件视图	主播连线
`createBattleView`	创建 PK 场景中单个用户的挂件视图（例如头像、分数）	主播 PK

版权所有：腾讯云计算（北京）有限责任公司

`createBattleContainerView`	创建 PK 场景的整体容器视图（例如背景、分数 PK 条）	主播 PK

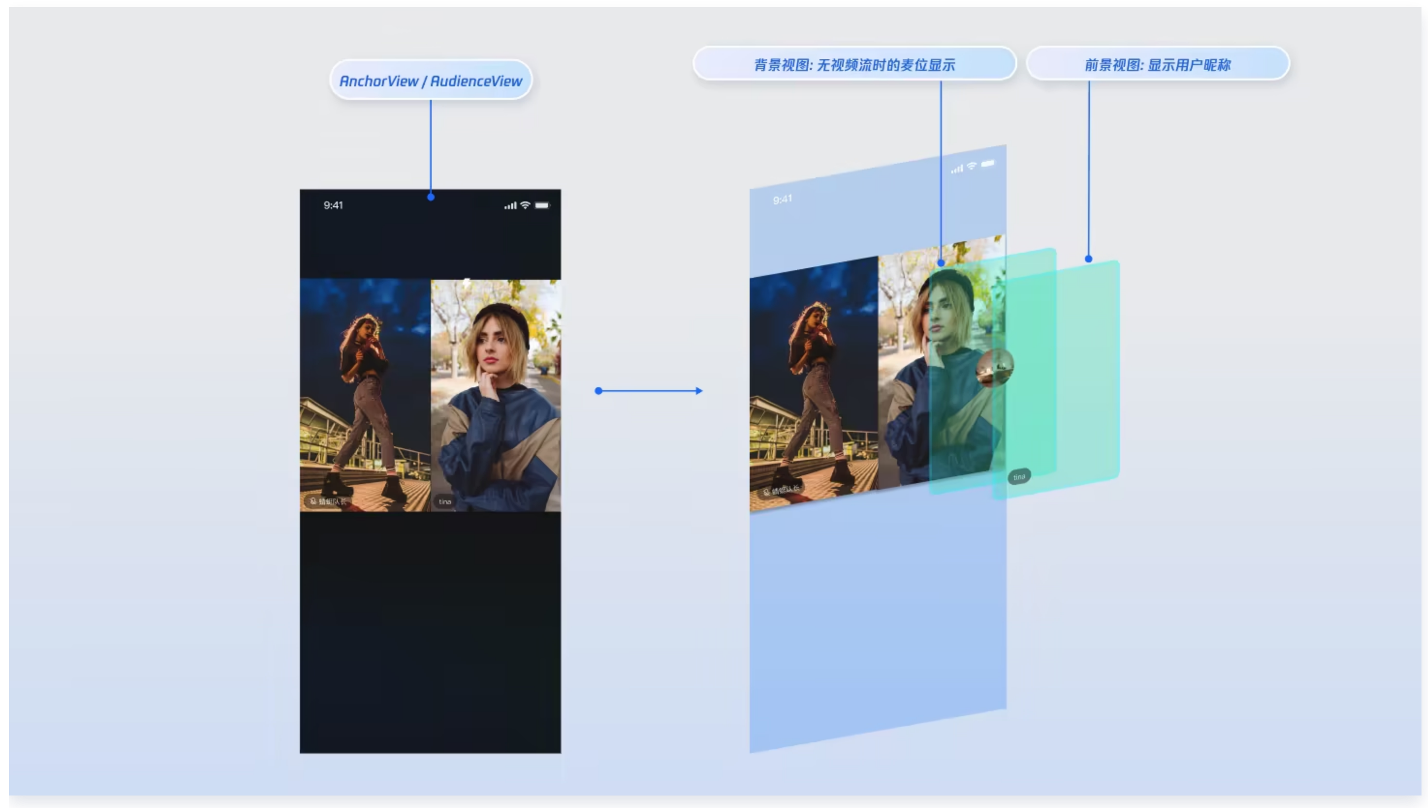
调整连线场景挂件

当观众与主播进行连线或者主播间跨房连线时，直播间界面会从单人模式切换为多人布局。此时，需要在连线的视频窗口上展示特定的用户信息，例如昵称、等级或“静音”状态提示，以便区分不同麦位。

适用场景

- 主播和观众视频连麦时，修改麦位上的用户信息（例如昵称、等级、静音图标）。
- 主播与主播跨房连线时，修改对方主播的显示样式（例如昵称、等级、静音图标）。
- 修改无视频画面时的默认背景（例如占位头像）。
- 修改空麦位时的视图实现。

视图层级示意



实现步骤

步骤1：准备自定义挂件 View

定义三个基础视图类，分别用于展示用户信息、空麦位提示和无视频流时的背景。

创建自定义的用户信息视图：

> **说明：**
> 您可以参考 TUILiveKit 开源项目中的 AnchorCoGuestView.swift 文件来了解默认效果的实现逻辑。

```
import UIKit

class CustomInfoView: UIView {
    let nameLabel = UILabel()
    let muteIcon = UIImageView(image: UIImage(named: "mute_icon"))

    init(name: String, isMuted: Bool) {
        super.init(frame: .zero)
        nameLabel.text = name
        muteIcon.isHidden = !isMuted
    }

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

    // ... layout code ...
}
```

创建自定义的空麦位视图：

> **说明：**
> 您可以参考 TUILiveKit 开源项目中的 AnchorEmptySeatView.swift 文件来了解默认效果的实现逻辑。

```
import UIKit

class EmptySeatView: UIView {
    let addIcon = UIImageView(image: UIImage(named: "add_icon"))
    let addLabel = UILabel()
    // ... layout code ...
}
```

版权所有：腾讯云计算（北京）有限责任公司

创建自定义无视频流时的头像占位视图：

> **说明：**
> 您可以参考 TUILiveKit 开源项目中的 AnchorBackgroundWidgetView.swift 文件来了解默认效果的实现逻辑。

```
import UIKit

class CustomAvatarView: UIView {
    let avatarView = UIImageView(frame: .zero)

    init(avatarURL: String) {
        super.init(frame: .zero)
        // load avatar URL
    }

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

    // ... layout code ...
}
```

步骤2：实现代理逻辑

创建一个代理类（或在现有 `ViewController` 中扩展），实现 `VideoViewDelegate` 的方法 `createCoGuestView` （观众连线）和 `createCoHostView` （主播连线），返回刚才创建的自定义视图。

实现 `VideoViewDelegate` 中的 `createCoGuestView` 方法，返回观众连线视频挂件。

```
class VideoWidgetProvider: VideoViewDelegate {
    /// seatInfo: 麦位数据 (用户信息、音视频状态)
    /// viewLayer: 视图层级 (.foreground 前景视图 / .background 背景视图)
    func createCoGuestView(seatInfo: SeatInfo, viewLayer: ViewLayer) ->
UIView? {
        let isUserOnSeat = !seatInfo.userInfo.userID.isEmpty

        switch viewLayer {
        case .foreground:
            if isUserOnSeat {
```

```
                // 非空麦位，返回自定义的前景视图
                let widget = CustomInfoView(name:
seatInfo.userInfo.userName,
                                            isMuted:
seatInfo.userInfo.microphoneStatus == .off)
                return widget
            }
            // 空麦位，返回自定义空麦位视图
            return EmptySeatView()
        case .background:
            if isUserOnSeat {
                // 返回自定义的背景视图（未开摄像头时显示）
                let bgView = CustomAvatarView(avatarURL:
seatInfo.userInfo.avatarURL)
                return bgView
            }
            return nil
        }
    }
}
```

实现 `VideoViewDelegate` 中的 `createCoHostView` 方法返回主播连线的视频挂件。

```
class VideoWidgetProvider: VideoViewDelegate {
    /// seatInfo: 麦位数据 （用户信息、音视频状态）
    /// viewLayer: 视图层级 (.foreground 前景视图 / .background 背景视图)
    func createCoHostView(seatInfo: SeatInfo, viewLayer: ViewLayer) ->
UIView? {
        let isUserOnSeat = !seatInfo.userInfo.userID.isEmpty

        switch viewLayer {
        case .foreground:
            if isUserOnSeat {
                // 返回自定义的前景视图，可以返回与观众连线不同的样式
                let widget = CustomInfoView(name:
seatInfo.userInfo.userName,
                                            isMuted:
seatInfo.userInfo.microphoneStatus == .off)
                return widget
```

```
                }
                // 返回自定义空麦位视图，可以返回与观众连线不同的样式
                return EmptySeatView()
            case .background:
                if isUserOnSeat {
                    // 返回自定义的背景视图（未开摄像头时显示），可以返回与观众连线不同的样式
                    let bgView = CustomAvatarView(avatarURL: seatInfo.userInfo.avatarURL)
                    return bgView
                }
                return nil
            }
        }
}
```

参数说明：

参数	类型	说明
`seatInfo`	`SeatInfo`	麦位信息对象，包含麦上用户的详细信息。
`seatInfo.userInfo.userName`	`String`	麦上用户的昵称。
`seatInfo.userInfo.avatarURL`	`String`	麦上用户的头像 URL。
`seatInfo.userInfo.microphoneStatus`	`DeviceStatus`	麦上用户的麦克风状态。
`seatInfo.userInfo.cameraStatus`	`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。

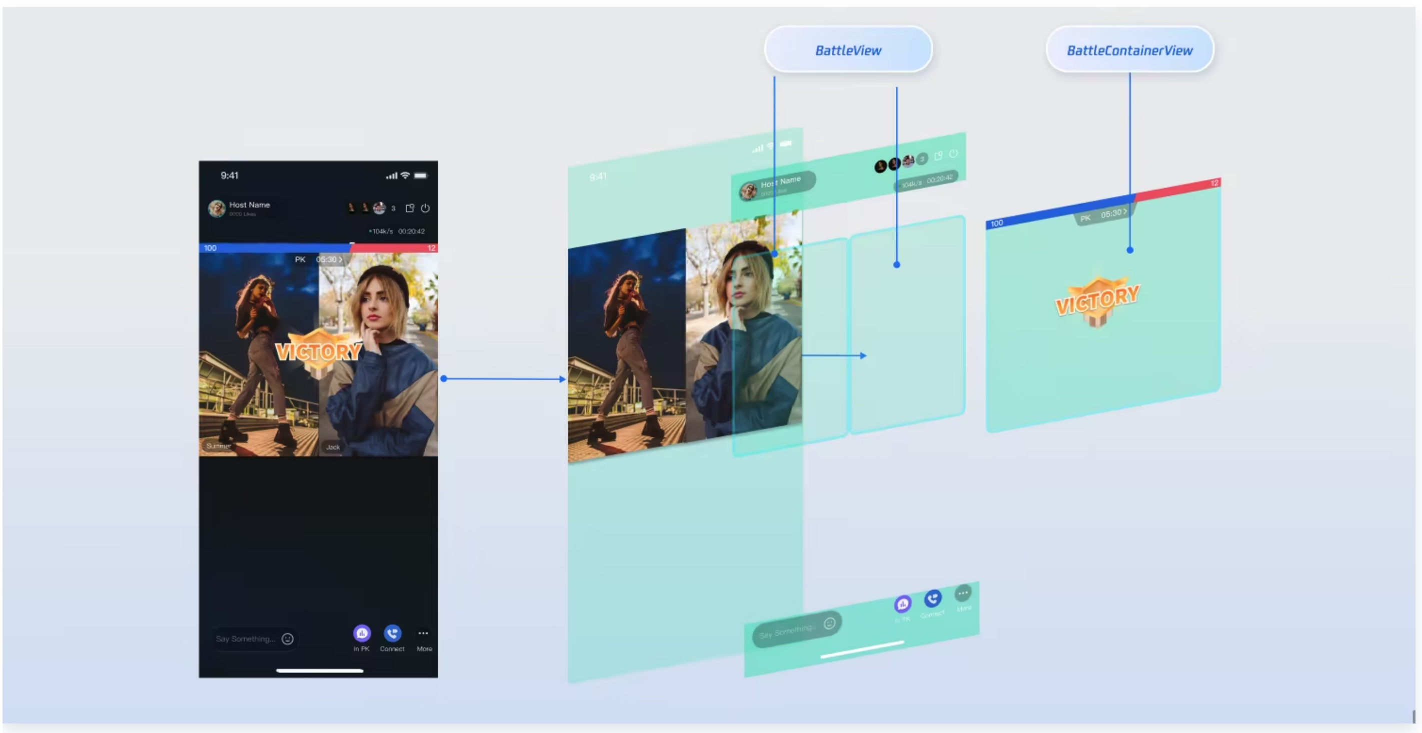
版权所有：腾讯云计算（北京）有限责任公司

		• `.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。

调整主播 PK 场景挂件

PK 是直播中互动性最强的环节。在 PK 过程中，画面通常根据参与人数被分割为几部分。开发者需要在画面上方或中间区域添加 PK 专用的 UI，分数 PK 条、每个麦位上的分数显示、倒计时动画或“VS”特效图标，以营造紧张的竞技氛围。

视图层级示意



> ⓘ **说明：**
> PK 功能依赖于连线功能，所以需要先在主播连线的情况下才能发起 PK 请求。

实现步骤

步骤1：准备自定义 UI 组件

PK 场景通常需要两类视图：

1. **单人挂件**：显示在每个主播窗口上（例如：分数胶囊）。
2. **全局容器**：覆盖在整体画面上（例如：VS 动画、倒计时）。

> ⓘ **说明：**

> 您可以参考 TUILiveKit 开源项目中的 AnchorBattleMemberInfoView.swift 和 AnchorBattleInfoView.swift 文件来了解默认效果的实现逻辑。

```
// 示例：单人分数条
class MyBattleScoreView: UIView {
    private let scoreView = UIView()

    // ... layout code ...
}

// 示例：全局 VS 面板
class MyBattleContainer: UIView {
    private let battleTimeView = UIImageView(frame: .zero)
    // 内部实现倒计时和 VS 动画
}
```

步骤2：实现代理逻辑

实现 `VideoViewDelegate` 的剩余方法 `createBattleView` 和 `createBattleContainerView` 。

```
class VideoWidgetProvider: VideoViewDelegate {
    /// 1. 创建【PK 单人信息】挂件（显示在每个主播视频上方）
    func createBattleView(seatInfo: SeatInfo) -> UIView? {
        let scoreView = MyBattleScoreView()
        return scoreView
    }

    /// 2. 创建【PK 全局容器】挂件（显示在整个视频区域上方）
    func createBattleContainerView() -> UIView? {
        let container = MyBattleContainer()
        return container
    }
}
```

集成与生效（重要）

这是最关键的一步。您需要将实现了代理逻辑的 `VideoWidgetProvider` 注入到直播核心流程中。

主播端集成：在初始化 `AnchorView` 之前，先初始化 `LiveCoreView` 并接管其代理。

```
import AtomicXCore
import SnapKit

class YourAnchorViewController: UIViewController {
    private let anchorView: AnchorView
    private let widgetProvider = VideoWidgetProvider()

    init(liveInfo: LiveInfo) {
        let videoView = LiveCoreView(viewType: .pushView)
        videoView.videoViewDelegate = widgetProvider // 接管代理
        anchorView = AnchorView(liveInfo: liveInfo, coreView: videoView)
        super.init(nibName: nil, bundle: nil)
    }

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

    override func viewDidLoad() {
       super.viewDidLoad()
       view.addSubview(anchorView)

       anchorView.snp.makeConstraints { make in
           make.edges.equalToSuperview()
       }
   }
}
```

观众端集成：观众端通过 `AudienceContainerViewDelegate` 来回调核心视图的创建时机。

```
class YourAudienceViewController: UIViewController {
    private let audienceView: AudienceContainerView
    private let widgetProvider = VideoWidgetProvider()

    public init(roomId: String) {
        self.audienceView = AudienceContainerView(roomId: roomId)
        super.init(nibName: nil, bundle: nil)
        self.audienceView.delegate = self
    }
```

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

extension YourAudienceViewController: AudienceContainerViewDelegate {
    func onCreateCoreView(for liveInfo: LiveInfo) -> LiveCoreView? {
        let view = LiveCoreView(viewType: .playView)
        view.videoViewDelegate = widgetProvider // 接管代理
        return view
    }
}
```

进阶：获取实时业务数据

在开发 PK 等复杂场景时，您会发现 `SeatInfo` 只提供了基础麦位信息。如果您需要获取实时倒计时、PK 分数等业务数据，就需要在您的自定义视图中对接 `AtomicXCore` 中的核心数据。

Store/Component	功能描述	API 文档
CoGuestStore	观众连线数据：已连线用户列表、邀请列表，申请列表等。	API 文档
CoHostStore	主播连线数据：已连线用户列表、邀请列表，申请列表等。	API 文档
BattleStore	PK 数据：当前 PK 信息、PK 用户列表、PK 分数列表。	API 文档

常见问题

为什么我设置了代理，但自定义视图没有显示?

请检查以下两点：

1. **代理对象被释放了**：`videoViewDelegate` 是弱引用。如果您在函数内部定义代理（例如 `let delegate = VideoWidgetProvider()`），函数执行结束后代理对象会被立即释放。
   解决：请务必将代理对象声明为 `ViewController` 的属性（成员变量），保持强引用。
2. **设置时机太晚**：创建主播视图的挂件时必须在创建 `AnchorView` 之前完成 `LiveCoreView` 的 `videoViewDelegate` 的赋值，否则 `AnchorView` 已经完成了默认视图的加载

我只想修改“连麦”视图，“PK”视图想保留默认的，怎么办?

`VideoViewDelegate` 是一种完全接管模式。一旦您设置了自定义的代理，SDK 默认的 `AnchorView` / `AudienceView` 内部实现的代理逻辑就会完全失效。

解决：您需要在您的代理类中实现所有相关方法。对于您不想修改的部分（例如 PK 视图），您可以：

1. **复制源码**：找到 `LiveKit` 源码中默认的实现类（例如 `AnchorBattleInfoView` ），在您的代理中返回这些默认类的实例。
2. **手动实现**：参考默认样式，快速写一个类似的简单视图返回。

自定义挂件显示了，但无法点击?

因为前景视图（ `.foreground` ）始终位于视频层之上，所以您要检查您的 View 是否位于 `.foreground` 层，且 `isUserInteractionEnabled` 为 `true` ，同时确保它的父视图没有禁用交互。

直播视频组件 (Web Vue3)

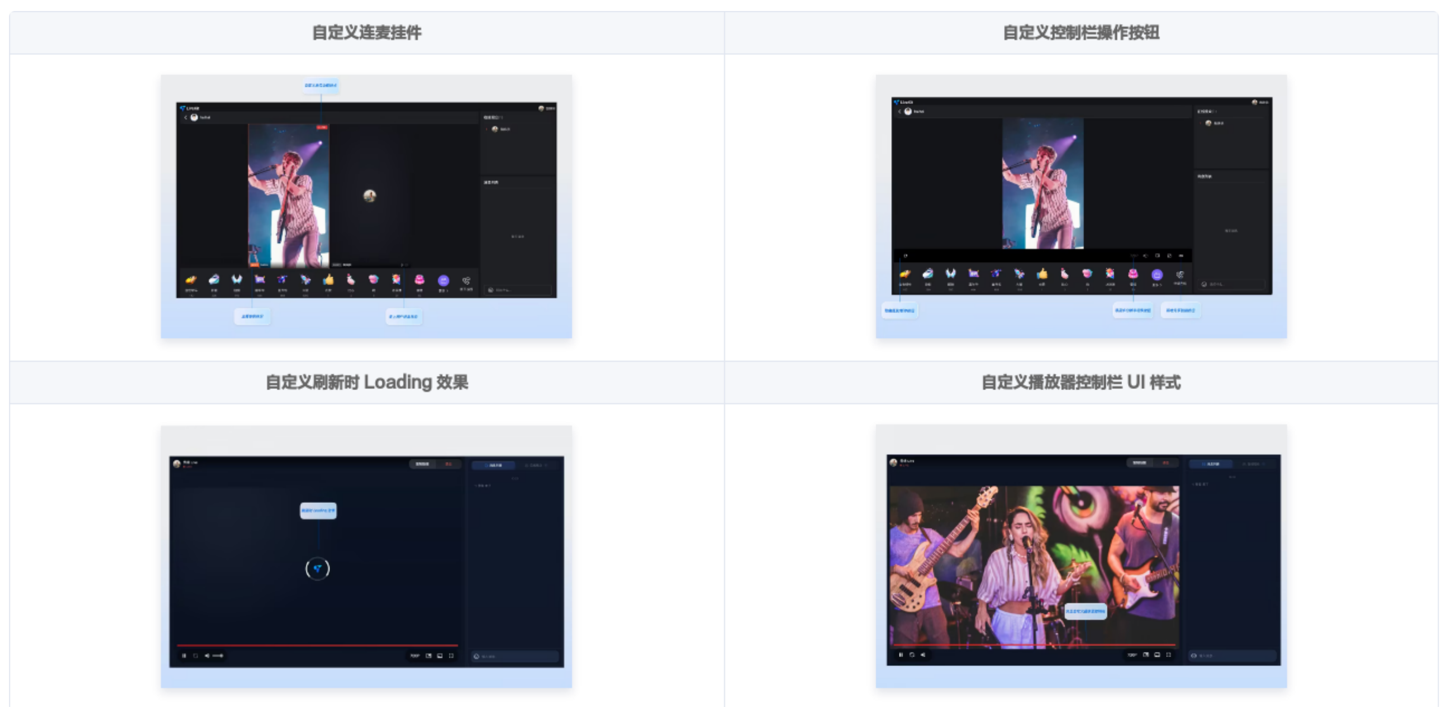
最近更新时间：2026-04-28 10:39:03

本文详细介绍了**直播视频组件（LiveView）**的接入方式与自定义能力。您可以直接参考本文示例快速集成组件，也可以根据业务需求对控制栏、插槽和播放器 UI 进行深度定制。

核心功能

功能分类	具体能力
智能流切换	LiveView 能够根据当前用户的身份（观众或连麦者）自动切换流类型。 • **观众模式：** 组件将播放超低延迟视频流，确保数百万观众都能流畅观看，同时大幅节省流量成本。 • **连线模式：** 组件会自动切换到实时音视频流，提供毫秒级的超低延迟，保证连线用户之间实时、清晰的互动体验。
多人连麦布局	自动适配单人与多人连麦场景的视频流布局，包含 `Loading` 动画和主播离开提示等状态展示。
内置播放器控制栏	开箱即用的播放器控制栏，包含**播放/暂停、音量调节、清晰度切换（360P–1080P）、画中画、全屏**等功能。
可定制化 UI	LiveView 提供从轻度到深度的渐进式自定义能力： • **插槽注入**：通过 Slot 在视频流上叠加水印、自定义 Loading、自定义连麦麦位 UI 等。 • **控制栏配置**：隐藏/禁用/替换图标等方式定制内置按钮，或添加业务自定义按钮。 • **自定义控制栏 UI**：隐藏内置控制栏，通过 `useLivePlayerState()` 提供的完整响应式状态和控制方法，构建 100% 自定义的播放控制界面。

效果展示



准备工作

在开始调整视频挂件前，请先参考 主播开播 和 观众观看 完成主流程的搭建。

组件定制化

LiveView 提供从轻度到深度的渐进式自定义能力，您可以根据业务需求选择合适的定制方式。

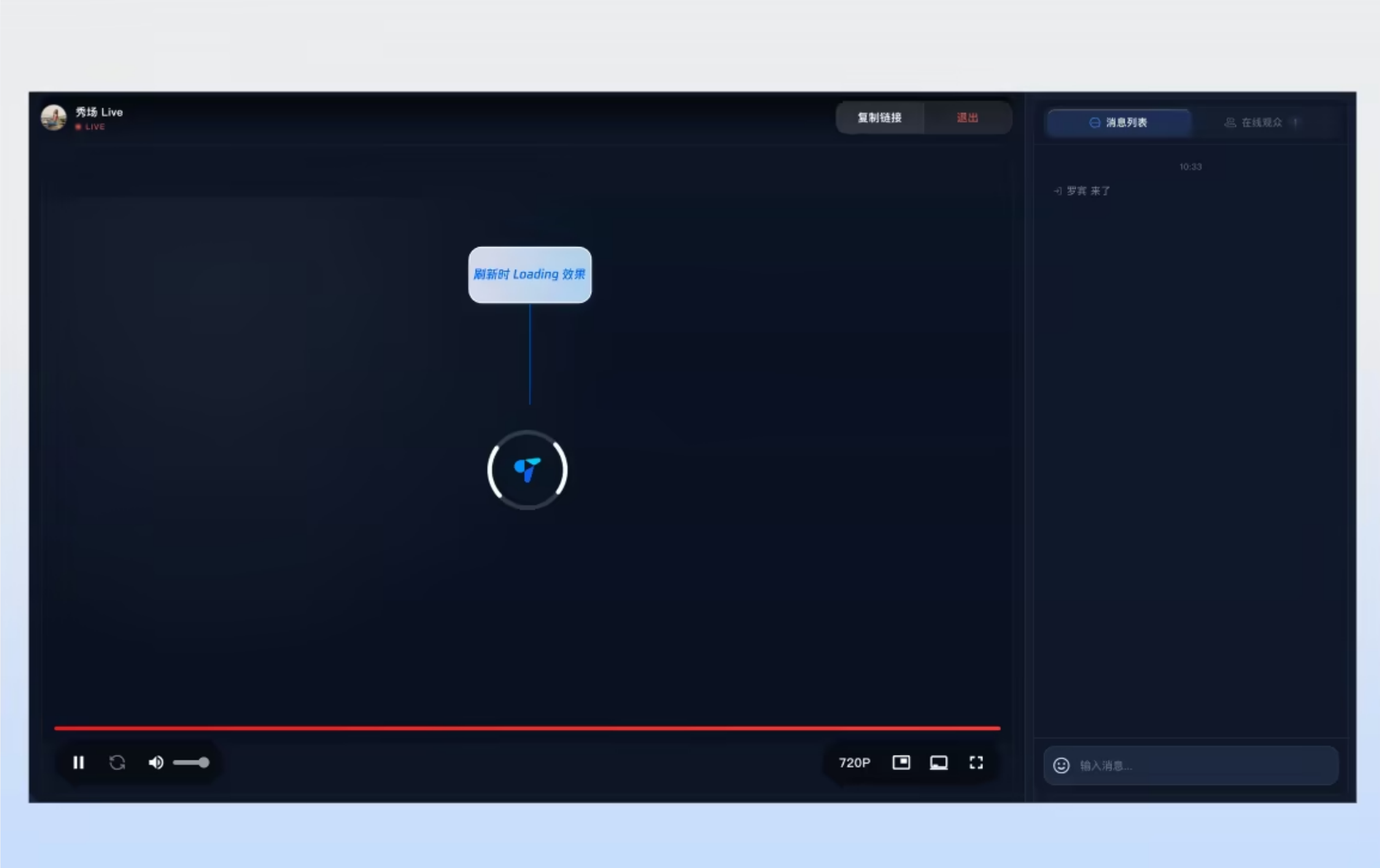
组件插槽

LiveView 提供以下插槽，用于自定义视频流上层的 UI 内容。

名称	参数	参数类型	说明
center-overlay	–	–	插槽内容会居中叠加在视频流上层，适合放置暂停按钮、水印等自定义元素。
streamViewUI	userInfo	SeatUserInfo	当前麦位上的用户信息，包含 `userId` 、 `userName` 、 `avatarUrl` 、 `microphoneStatus` 、 `cameraStatus` 等字段。如果麦位为空，则 `userInfo.userId` 为空字符串。

center-overlay 插槽

使用 center-overlay 插槽可以在视频流上层叠加自定义内容，不影响底层视频播放。适合放置水印、暂停按钮、视频刷新时 Loading 等覆盖层元素。



基本用法

```
<LiveView>
  <template #center-overlay>
    <div class="watermark-container">
      <div class="watermark-grid">
        <span v-for="n in 20" :key="n" class="watermark-
item">Live</span>
      </div>
    </div>
  </template>
</LiveView>

<style scoped>.watermark-
container{position:absolute;inset:0;overflow:hidden;pointer-
events:none;user-select:none}.watermark-
```

版权所有：腾讯云计算（北京）有限责任公司

```
grid{position:absolute;top:-50%;left:-50%;width:200%;height:200%;display
:flex;flex-wrap:wrap;align-content:center;justify-
content:center;gap:60px 80px;transform:rotate(-30deg)}.watermark-
item{color:rgba(255,255,255,.15);font-size:18px;font-weight:700;letter-
spacing:2px;white-space:nowrap}</style>
```

自定义刷新 Loading 示例

结合 `useLivePlayerState()` 的 `refresh` 方法，实现带品牌 Logo 的刷新加载动画：

```
<template>
  <LiveView>
    <template #center-overlay>
      <div v-if="isRefreshing" class="refresh-overlay">
        <img class="refresh-logo" src="./logo.svg" alt="logo" />
        <div class="refresh-spinner" />
        <span class="refresh-text">加载中...</span>
      </div>
      <div v-else class="watermark">LIVE</div>
    </template>
  </LiveView>
  <button @click="handleRefresh">刷新</button>
</template>

<script setup lang="ts">
import { ref } from 'vue';
import { LiveView, useLivePlayerState } from 'tuikit-atomicx-vue3';

const { refresh } = useLivePlayerState();
const isRefreshing = ref(false);

async function handleRefresh() {
  isRefreshing.value = true;
  await refresh();
  isRefreshing.value = false;
}
</script>

<style scoped>.refresh-overlay{display:flex;flex-direction:column;align-
items:center;gap:12px;pointer-events:none}.refresh-
```

版权所有：腾讯云计算（北京）有限责任公司

```
logo{width:40px;height:40px;object-fit:contain}.refresh-
spinner{width:32px;height:32px;border:3px solid
rgba(255,255,255,.2);border-top-color:#fff;border-
radius:50%;animation:spin 1s linear infinite}@keyframes
spin{to{transform:rotate(360deg)}}.refresh-
text{color:rgba(255,255,255,.8);font-
size:14px}.watermark{color:rgba(255,255,255,.3);font-size:24px;font-
weight:700;letter-spacing:2px;pointer-events:none;user-select:none;text-
shadow:0 2px 8px rgba(0,0,0,.3)}</style>
```

streamViewUI 插槽

使用 streamViewUI 插槽可以完全接管每个麦位的 UI 渲染。您将获得当前麦位的用户信息，可自由实现主播/观众标识、设备状态展示、自定义布局等。

基本用法

```
<template>
  <LiveView @empty-seat-click="handleApplyForSeat">
    <template #streamViewUI="{ userInfo }">
```

```
      <div class="custom-stream-ui">
        <span class="user-name">{{ userInfo.userName || '未知用户' }}
</span>
      </div>
    </template>
  </LiveView>
</template>

<script setup lang="ts">
import { LiveView } from 'tuikit-atomicx-vue3';

const handleApplyForSeat = (seatIndex: number) => {
  console.log('点击了空麦位，索引：', seatIndex);
  // 在此处发起连麦申请
};
</script>

<style scoped>.custom-stream-
ui{position:absolute;bottom:0;left:0;right:0;padding:8px
12px;background:linear-gradient(transparent,rgba(0,0,0,.6));pointer-
events:none}.custom-stream-ui .user-name{color:#fff;font-size:14px;font-
weight:500}</style>
```

自定义连麦 UI 示例

```
<template>
  <LiveView>
    <template #streamViewUI="{ userInfo }">
      <div class="stream-overlay">
        <div class="top-bar">
          <div class="role-badge" :class="{ 'anchor-badge':
isAnchor(userInfo) }">
            <span class="badge-dot" />
            {{ isAnchor(userInfo) ? '主播' : '观众' }}
          </div>
        </div>
        <div class="bottom-bar">
          <div class="user-info">
```

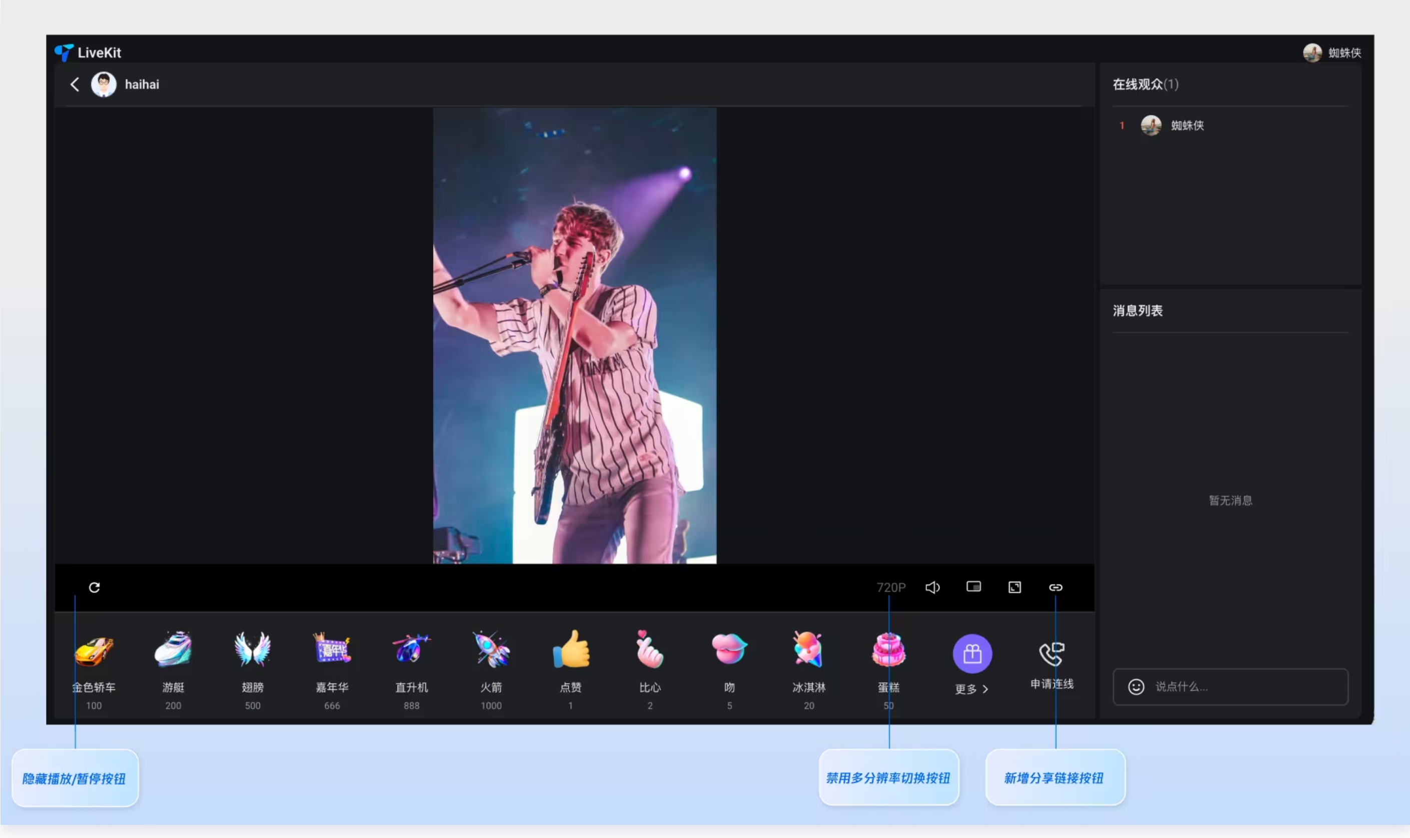
```
            <span class="user-name-text">{{ userInfo.userName || '未知用户' }}</span>
          </div>
          <div class="device-icons">
            <div class="icon-wrapper" :class="{ off: !isMicOn(userInfo) }">
              <svg viewBox="0 0 24 24" fill="none">
                <path d="M12 2a3 3 0 0 0-3 3v6a3 3 0 0 0 6 0V5a3 3 0 0 0-3-3Z" fill="currentColor" />
                <path d="M19 11a7 7 0 0 1-14 0" stroke="currentColor" stroke-width="1.5" stroke-linecap="round" />
                <path d="M12 18v3M9 21h6" stroke="currentColor" stroke-width="1.5" stroke-linecap="round" />
              </svg>
              <svg v-if="!isMicOn(userInfo)" class="slash-line" viewBox="0 0 24 24">
                <line x1="5" y1="2" x2="19" y2="22" stroke="#ff4d4f" stroke-width="2" stroke-linecap="round" />
              </svg>
            </div>
            <div class="icon-wrapper" :class="{ off: !isCameraOn(userInfo) }">
              <svg viewBox="0 0 24 24" fill="none">
                <rect x="2" y="5.5" width="14.5" height="13" rx="2.5" fill="currentColor" />
                <path d="M17.5 10l3.5-2.2v8.4L17.5 14" fill="currentColor" />
              </svg>
              <svg v-if="!isCameraOn(userInfo)" class="slash-line" viewBox="0 0 24 24">
                <line x1="5" y1="2" x2="19" y2="22" stroke="#ff4d4f" stroke-width="2" stroke-linecap="round" />
              </svg>
            </div>
          </div>
        </div>
      </div>
    </template>
  </LiveView>
</template>
```

```
<script setup lang="ts">
import { LiveView, useLiveListState, DeviceStatus } from 'tuikit-
atomicx-vue3';
import type { SeatUserInfo } from 'tuikit-atomicx-vue3';

const { currentLive } = useLiveListState();

const isAnchor = (userInfo: SeatUserInfo) => {
  return userInfo.userId === currentLive.value?.liveOwner?.userId;
};

const isMicOn = (userInfo: SeatUserInfo) => {
  return userInfo.microphoneStatus === DeviceStatus.On;
};

const isCameraOn = (userInfo: SeatUserInfo) => {
  return userInfo.cameraStatus === DeviceStatus.On;
};
</script>

<style scoped>.stream-
overlay{position:absolute;top:0;left:0;width:100%;height:100%;pointer-
events:none;box-sizing:border-box;border-
radius:6px;overflow:hidden}.top-
bar{position:absolute;top:6px;left:6px;z-index:2}.role-
badge{display:inline-flex;align-items:center;gap:4px;padding:2px
8px;border-radius:10px;font-size:10px;font-weight:500;line-
height:16px;background:rgba(0,0,0,.45);backdrop-filter:blur(6px);-
webkit-backdrop-filter:blur(6px);color:rgba(255,255,255,.85)}.anchor-
badge{background:linear-
gradient(135deg,#ff6b35,#e8461e);color:#fff}.badge-
dot{width:5px;height:5px;border-radius:50%;background:currentColor;flex-
shrink:0}.anchor-badge .badge-dot{background:#fff;animation:pulse-dot 2s
ease-in-out infinite}@keyframes pulse-dot{0%,100%{opacity:1}50%
{opacity:.4}}.bottom-bar{position:absolute;bottom:0;left:0;right:0;z-
index:2;display:flex;align-items:flex-end;justify-content:space-
between;padding:28px 8px 6px;background:linear-gradient(to
top,rgba(0,0,0,.6) 0%,rgba(0,0,0,.2) 50%,transparent 100%)}.user-
info{flex:1;min-width:0}.user-name-text{display:block;color:#fff;font-
```

```
size:11px;font-weight:500;line-height:16px;text-shadow:0 1px 3px
rgba(0,0,0,.7);overflow:hidden;text-overflow:ellipsis;white-
space:nowrap}.device-icons{display:flex;align-items:center;gap:3px;flex-
shrink:0;margin-left:6px}.icon-
wrapper{position:relative;width:16px;height:16px;color:rgba(255,255,255,
.9);transition:color .2s ease}.icon-
wrapper.off{color:rgba(255,255,255,.3)}.icon-wrapper
svg{position:absolute;top:0;left:0;width:100%;height:100%}.icon-wrapper
.slash-line{z-index:1}</style>
```

自定义控制栏操作按钮

LiveView 内置了播放器控制栏（播放/暂停、音量、清晰度、画中画、全屏等按钮）。通过 useLivePlayerState() ，您可以 **控制播放行为、定制控制栏按钮、添加自定义按钮、监听播放器事件**。



隐藏/禁用内置按钮

buttons 对象包含所有内置按钮（ `play` 、 `volume` 、 `resolution` 、 `pictureInPicture` 、 `fullscreen` ），直接修改即可实时生效，每个按钮支持以下属性（ ButtonState ）：

属性	类型	说明
visible	boolean	是否显示按钮。

disabled	boolean	是否禁用按钮。
icon	Component | (() => VNode)	自定义按钮**默认态**图标。 • **Play** 按钮：播放时显示（即“暂停”图标，点击可暂停）。 • **Volume** 按钮：未静音时显示（即“喇叭”图标）。 • **PictureInPicture** 按钮：非画中画时显示。 • **Fullscreen** 按钮：非全屏时显示。
activeIcon	Component | (() => VNode)	自定义按钮**激活态**图标。 • **Play** 按钮：暂停时显示（即“播放”图标，点击可恢复播放）。 • **Volume** 按钮：静音时显示（即“静音”图标）。 • **PictureInPicture** 按钮：画中画模式下显示。 • **Fullscreen** 按钮：全屏模式下显示。
tooltip	string	鼠标悬浮时的提示文案。

```
<script setup lang="ts">
import { useLivePlayerState } from 'tuikit-atomicx-vue3';
import MyPauseIcon from './icons/MyPauseIcon.vue';
import MyPlayIcon from './icons/MyPlayIcon.vue';
import MyVolumeOnIcon from './icons/MyVolumeOnIcon.vue';
import MyVolumeOffIcon from './icons/MyVolumeOffIcon.vue';

const { buttons } = useLivePlayerState();

// 隐藏/禁用按钮
buttons.pictureInPicture.visible = false;         // 隐藏画中画按钮
buttons.fullscreen.disabled = true;               // 禁用全屏按钮

// 自定义图标
buttons.play.icon = MyPauseIcon;                  // 默认：播放时显示（暂停按钮图标）
buttons.play.activeIcon = MyPlayIcon;             // 激活：暂停时显示（恢复播放图标）
buttons.volume.icon = MyVolumeOnIcon;             // 默认值：未静音时显示
buttons.volume.activeIcon = MyVolumeOffIcon;      // 激活：静音时显示

// 自定义工具提示
```

```
buttons.resolution.tooltip = 'Switch Quality';
</script>
```

添加自定义按钮

通过 addCustomButtons 在控制栏添加业务按钮（例如分享、刷新）：

```
<script setup lang="ts">
import { useLivePlayerState } from 'tuikit-atomicx-vue3';
import ShareIcon from './icons/ShareIcon.vue';

const { addCustomButtons, refresh } = useLivePlayerState();

addCustomButtons([
  {
    id: 'share',
    icon: ShareIcon,
    onClick: () => navigator.clipboard.writeText(window.location.href),
    tooltip: 'Share',
    position: 'end',  // 'start' | 'end' | { slot:
'left'|'center'|'right' } | { anchor: ButtonId, position:
'before'|'after' }
  },
]);
</script>
```

监听播放器事件

支持的事件有：

事件名	事件参数	说明
PlayStateChange	isPlaying: boolean	当播放状态发生变化（播放/暂停）时触发。
VolumeChange	volume: number	当音量发生变化时触发。
FullscreenChange	isFullscreen: boolean	当进入或退出全屏模式时触发。

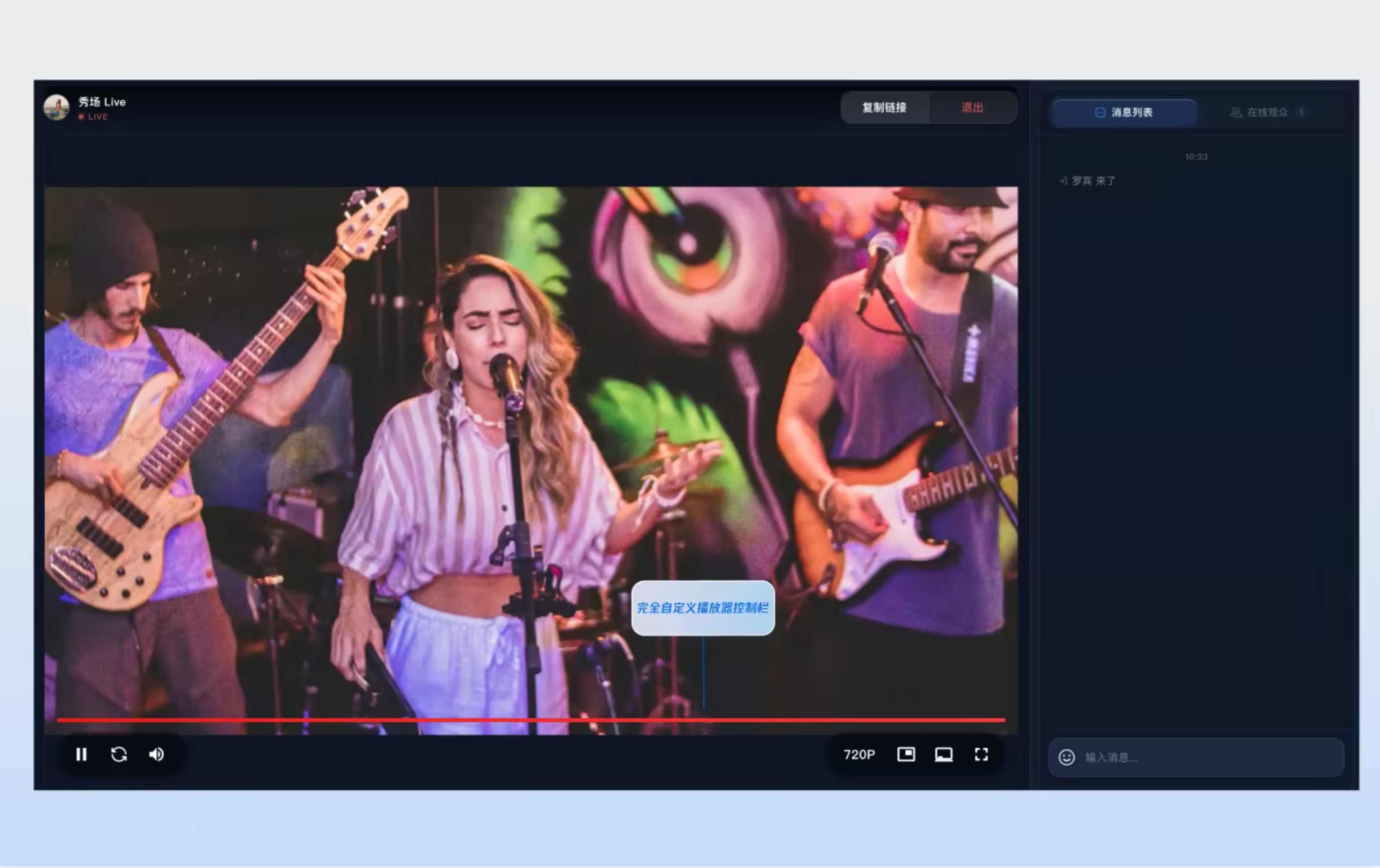
PictureInPictureChange	isPictureInPicture: boolean	当进入或退出画中画模式时触发。
ResolutionChange	resolution: Resolution	当清晰度切换时触发。
ControlBarVisibilityChange	visible: boolean	当控制栏可见性发生变化（显示/隐藏）时触发。

```
<script setup lang="ts">
import { ref } from 'vue';
import { useLivePlayerState, PlayerControlEvent } from 'tuikit-atomicx-vue3';

const { subscribeEvent } = useLivePlayerState();

const titleVisible = ref(true);
const roomNameVisible = ref(true);

// 根据全屏状态显示/隐藏直播间标题
subscribeEvent(PlayerControlEvent.FullscreenChange, (isFullscreen: boolean) => {
  titleVisible.value = !isFullscreen;
});

// 根据控制栏显隐状态同步显示/隐藏房间名称
subscribeEvent(PlayerControlEvent.ControlBarVisibilityChange, (visible: boolean) => {
  roomNameVisible.value = visible;
});
</script>
```

自定义控制栏 UI

当内置控制栏的样式和布局无法满足您的业务需求时，您可以隐藏内置控制栏，使用 useLivePlayerState 提供的全部**响应式状态**和**控制方法**构建 100% 自定义的播放器界面。

> **说明：** `useLivePlayerState()` 是全局单例，在任意组件中调用都共享同一份状态，无需通过 props 层层传递。



步骤一：隐藏内置控制栏

```
<script setup lang="ts">
import { useLivePlayerState } from 'tuikit-atomicx-vue3';

const { buttons } = useLivePlayerState();

// 隐藏所有内置按钮，内置控制栏将不再渲染
Object.values(buttons).forEach(btn => (btn.visible = false));
</script>
```

步骤二：用响应式状态和控制方法驱动自定义 UI

```
<template>
  <div class="my-player" @mouseenter="showControls"
@mouseleave="startAutoHide" @mousemove="showControls">
    <LiveView />
    <div class="my-controls" :class="{ hidden: !controlsVisible }">
      <div class="controls-left">
```

```
        <button class="ctrl-btn" @click="isPlaying ? pause() :
resume()">
          <svg v-if="!isPlaying" viewBox="0 0 24 24"
fill="currentColor"><path d="M8 5v14l11-7z" /></svg>
          <svg v-else viewBox="0 0 24 24" fill="currentColor"><rect
x="6" y="4" width="4" height="16" rx="1" /><rect x="14" y="4" width="4"
height="16" rx="1" /></svg>
        </button>
        <button class="ctrl-btn" @click="refresh()">
          <svg viewBox="0 0 24 24" fill="none" stroke="currentColor"
stroke-width="2" stroke-linecap="round"><path d="M1 4v6h6" /><path
d="M23 20v-6h-6" /><path d="M20.49 9A9 9 0 0 0 5.64 5.64L1 10M23 14l-
4.64 4.36A9 9 0 0 1 3.51 15" /></svg>
        </button>
        <div class="volume-group">
          <button class="ctrl-btn" @click="isMuted ? unmute() : mute()">
            <svg v-if="!isMuted" viewBox="0 0 24 24"
fill="currentColor"><path d="M11 5L6 9H2v6h4l5 4V5z" /><path d="M15.54
8.46a5 5 0 0 1 0 7.07" fill="none" stroke="currentColor" stroke-
width="2" stroke-linecap="round" /><path d="M19.07 4.93a10 10 0 0 1 0
14.14" fill="none" stroke="currentColor" stroke-width="2" stroke-
linecap="round" /></svg>
            <svg v-else viewBox="0 0 24 24" fill="currentColor"><path
d="M11 5L6 9H2v6h4l5 4V5z" /><line x1="23" y1="9" x2="17" y2="15"
stroke="currentColor" stroke-width="2" stroke-linecap="round" /><line
x1="17" y1="9" x2="23" y2="15" stroke="currentColor" stroke-width="2"
stroke-linecap="round" /></svg>
          </button>
          <input class="volume-slider" type="range"
:value="currentVolume" min="0" max="100"
            @input="setVolume(Number(($event.target as
HTMLInputElement).value))" />
        </div>
      </div>
      <div class="controls-right">
        <select v-if="resolutionList.value.length > 0"
class="resolution-select"
          :value="currentResolution.value?.value"
          @change="switchResolution(resolutionList.value.find(r =>
r.value === Number(($event.target as HTMLSelectElement).value)))">
```

```
          <option v-for="r in resolutionList.value" :key="r.value"
:value="r.value">{{ r.label }}</option>
        </select>
        <button class="ctrl-btn" @click="isPictureInPicture ?
exitPictureInPicture() : requestPictureInPicture()">
          <svg viewBox="0 0 24 24" fill="none" stroke="currentColor"
stroke-width="2" stroke-linecap="round" stroke-linejoin="round"><rect
x="2" y="3" width="20" height="14" rx="2" /><rect v-
if="!isPictureInPicture" x="11" y="9" width="9" height="6" rx="1"
fill="currentColor" stroke="none" /><path v-else d="M15 19l-3-3m0 0l3-
3m-3 3h8" /></svg>
        </button>
        <button class="ctrl-btn" @click="isFullscreen ? exitFullscreen()
: requestFullscreen()">
          <svg v-if="!isFullscreen" viewBox="0 0 24 24" fill="none"
stroke="currentColor" stroke-width="2" stroke-linecap="round"><path
d="M8 3H5a2 2 0 0 0-2 2v3m18 0V5a2 2 0 0 0-2-2h-3m0 18h3a2 2 0 0 0 2-2v-
3M3 16v3a2 2 0 0 0 2 2h3" /></svg>
          <svg v-else viewBox="0 0 24 24" fill="none"
stroke="currentColor" stroke-width="2" stroke-linecap="round"><path
d="M4 14h4v4M20 10h-4V6M14 10h4V6M4 14h4v4" /></svg>
        </button>
      </div>
    </div>
  </div>
</template>

<script setup lang="ts">
import { ref } from 'vue';
import { LiveView, useLivePlayerState } from 'tuikit-atomicx-vue3';

const {
  isPlaying, currentVolume, isMuted, isFullscreen,
  isPictureInPicture, currentResolution, resolutionList,
  buttons,
  pause, resume, refresh,
  setVolume, mute, unmute,
  requestFullscreen, exitFullscreen,
  requestPictureInPicture, exitPictureInPicture,
  switchResolution,
```

```
} = useLivePlayerState();

// 隐藏所有内置按钮
Object.values(buttons).forEach(btn => (btn.visible = false));

// 控制栏自动显隐
const controlsVisible = ref(true);
let autoHideTimer: ReturnType<typeof setTimeout> | null = null;

function showControls() {
  controlsVisible.value = true;
  startAutoHide();
}

function startAutoHide() {
  if (autoHideTimer) clearTimeout(autoHideTimer);
  autoHideTimer = setTimeout(() => { controlsVisible.value = false; },
3000);
}
</script>

<style scoped>.my-player{position:relative;width:100%;height:100%}.my-
controls{position:absolute;bottom:0;left:0;right:0;z-index:20;pointer-
events:auto;display:flex;align-items:center;justify-content:space-
between;padding:8px 12px;background:linear-gradient(to
top,rgba(0,0,0,.75) 0%,rgba(0,0,0,.3) 70%,transparent
100%);transition:opacity .3s ease,transform .3s ease}.my-
controls.hidden{opacity:0;transform:translateY(8px);pointer-
events:none}.controls-left,.controls-right{display:flex;align-
items:center;gap:4px}.ctrl-btn{display:flex;align-items:center;justify-
content:center;width:32px;height:32px;padding:0;border:none;border-
radius:6px;background:transparent;color:rgba(255,255,255,.85);cursor:poi
nter;transition:background .15s ease,color .15s ease}.ctrl-
btn:hover{background:rgba(255,255,255,.12);color:#fff}.ctrl-
btn:active{background:rgba(255,255,255,.2)}.ctrl-btn
svg{width:18px;height:18px}.volume-group{display:flex;align-
items:center;gap:2px}.volume-slider{width:70px;height:3px;-webkit-
appearance:none;appearance:none;background:rgba(255,255,255,.25);border-
radius:2px;outline:none;cursor:pointer}.volume-slider::-webkit-slider-
thumb{-webkit-appearance:none;width:10px;height:10px;border-
```

版权所有：腾讯云计算（北京）有限责任公司

```
radius:50%;background:#fff;cursor:pointer;box-shadow:0 0 4px
rgba(0,0,0,.3)}.resolution-select{height:28px;padding:0 8px;border:1px
solid rgba(255,255,255,.15);border-
radius:6px;background:rgba(0,0,0,.4);color:rgba(255,255,255,.85);font-
size:11px;cursor:pointer;outline:none}.resolution-select:hover{border-
color:rgba(255,255,255,.35)}.resolution-select
option{background:#1a1a1a;color:#fff}</style>
```

播放控制 API 汇总

以下 API 均通过 useLivePlayerState() 获取：

分类	API	说明
响应式状态	isPlaying、isMuted、currentVolume、isFullscreen、isPictureInPicture、controlBarVisible、currentResolution、resolutionList、buttons	可直接绑定到模板的响应式状态。buttons 可通过赋值控制内置按钮的显隐、禁用和图标。
播放控制	pause()、resume()、refresh()	暂停、恢复播放、刷新（重新拉流）。
音量控制	setVolume(volume)、mute()、unmute()	设置音量（0–100）、静音、取消静音。
全屏	requestFullscreen()、exitFullscreen()	进入/退出全屏模式。
画中画	requestPictureInPicture()、exitPictureInPicture()	进入/退出画中画模式。
清晰度	switchResolution(resolution)	切换清晰度，参数为 Resolution 对象（含 label 和 value）。
控制栏	showControlBar()、hideControlBar()、setAutoHideDelay(ms)	显示/隐藏控制栏、设置自动隐藏延迟。
自定义按钮	addCustomButtons(buttons)	添加自定义按钮到控制栏，相同 id 自动更新。
事件订阅	subscribeEvent(event, callback), unsubscribeEvent(event, callback)	订阅/取消订阅播放器事件。建议在进入直播间前完成注册。

常见问题

如何解决浏览器自动播放受限导致的黑屏问题?

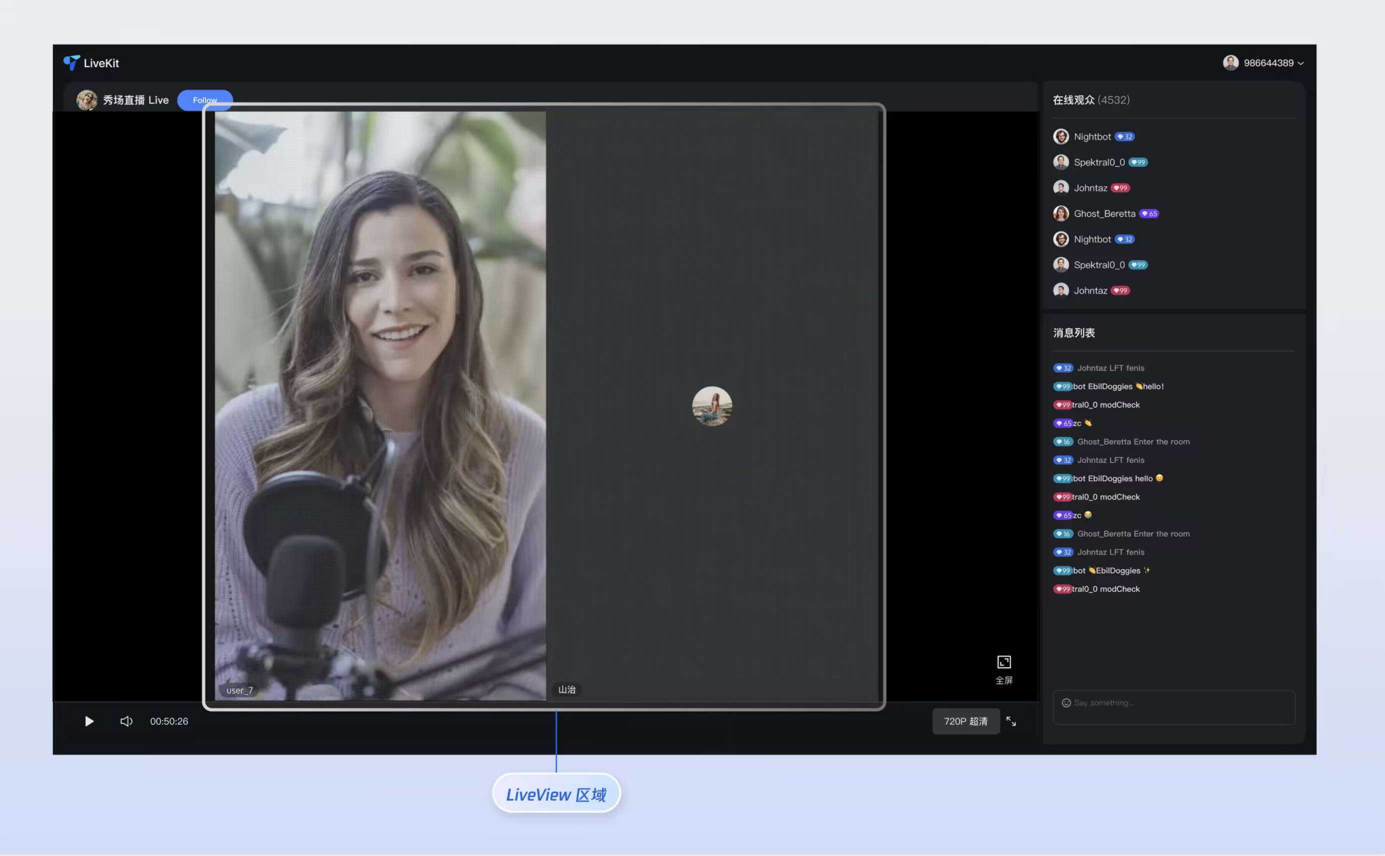
出于用户体验考虑，现代浏览器对网页自动播放功能实施了限制性策略。您可以参考如下代码自定义播放受限弹窗，引导用户与页面交互后恢复播放。

```
import TUIRoomEngine, { TUIAutoPlayCallbackInfo, TUIRoomEvents } from
'@tencentcloud/tuiroom-engine-js';
import { TUIMessageBox } from '@tencentcloud/uikit-base-component-vue3';
import { useRoomEngine } from 'tuikit-atomicx-vue3';

// LiveView 组件的播放能力由底层的 TUIRoomEngine 提供支持。您可以监听该引擎的事件
来处理更底层的播放问题
const roomEngine = useRoomEngine();

TUIRoomEngine.once('ready', () => {
  roomEngine.instance?.on(TUIRoomEvents.onAutoPlayFailed,
handleAutoPlayFailed);
});

function handleAutoPlayFailed(event: TUIAutoPlayCallbackInfo) {
  // 可以自行弹窗引导用户与页面发生交互，交互后调用 event.resume() 方法恢复播放
  TUIMessageBox.alert({
    content: 'Content is ready. Click the button to start playback',
    confirmText: 'Play',
    callback: () => {
      event.resume();
    }
  });
}
```

直播视频组件 (Web React)

最近更新时间：2026-03-03 18:01:03

本文对**直播视频组件（LiveView）**进行了详细的介绍，您可以在已有项目中直接参考本文示例集成我们开发好的组件，也可以根据您的需求按照文档中的组件定制化部分对样式，布局进行深度的定制。



核心功能

功能分类	具体能力
智能流切换	LiveView 能够根据当前用户的身份（观众或连麦者）自动切换流类型。 • 观众模式： 组件将播放超低延迟视频流，确保数百万观众都能流畅观看，同时大幅节省流量成本。 • 连线模式： 组件会自动切换到实时音视频流，提供毫秒级的超低延迟，保证连线用户之间实时、清晰的互动体验。
可定制化 UI	为了满足多样化的业务场景，LiveView 提供组件 UI 自定义插槽，让您能够完全掌控连麦用户的视频流区域，重写其 UI 展示，灵活定义连麦用户的头像、昵称、状态等信息，轻松打造符合您品牌风格的独特视觉体验。

组件接入

版权所有：腾讯云计算（北京）有限责任公司

步骤1：环境配置及开通服务

在进行快速接入之前，您需要参考 准备工作（Web React），满足相关环境配置及开通对应服务。

步骤2：安装依赖

npm

```
npm install tuikit-atomicx-react @tencentcloud/uikit-base-component-react --save
npm install sass --save-dev
```

pnpm

```
pnpm add tuikit-atomicx-react @tencentcloud/uikit-base-component-react
pnpm add sass --dev
```

yarn

```
yarn add tuikit-atomicx-react @tencentcloud/uikit-base-component-react
yarn add sass --dev
```

步骤3：加入直播间

在您的项目中引入并使用 **LiveView** 组件，可直接复制如下代码示例至您的项目中，观看对应直播间的直播视频。

LivePlayer.tsx

```
import React, { useCallback, useEffect } from "react";
import { MessageBox, Dialog, UIKitProvider } from "@tencentcloud/uikit-base-component-react";
import TUIRoomEngine, { TUIRoomEvents } from "@tencentcloud/tuiroom-engine-js";
import { LiveView, LiveListEvent, useLiveListState, useRoomEngine } from "tuikit-atomicx-react";
import styles from "./LivePlayer.module.scss";

interface LivePlayerProps {
```

```
  className?: string;
  liveId?: string;
}

const LivePlayer: React.FC<LivePlayerProps> = ({ className }) => {
  const roomEngine = useRoomEngine();
  const { currentLive, joinLive, subscribeEvent, unsubscribeEvent } =
useLiveListState();

  useEffect(() => {
    if (!currentLive?.liveId) {
      joinLive({ liveId: ''});  // 输入您要加入的直播间 ID，也可以是来自外部传
入的组件参数或者 URL 参数等。
    }
  }, [currentLive?.liveId, joinLive]);

  const handleAutoPlayFailed = useCallback(() => {
    MessageBox.alert({
      content: '内容已准备就绪，点击【播放】按钮开始播放',
      confirmText: '播放',
      showClose: false,
      modal: false,
    });
  }, []);

  const handleKickedOutOfLive = useCallback(() => {
    Dialog.open({
      content: '你已被踢出直播间',
      confirmText: '确认',
      className: styles.livePlayer__liveDialog,
      showCancel: false,
      showClose: false,
      modal: true,
      center: true,
      onConfirm: () => {
        Dialog.close();
        // 这里可以添加您自己的业务逻辑，如跳转到首页或直播列表页
      },
      onClose: () => {
        // 这里可以添加您自己的业务逻辑，如跳转到首页或直播列表页
```

```
      },
    });
  }, []);

  const handleLiveEnded = useCallback(() => {
    Dialog.open({
      content: '直播已结束',
      confirmText: '确认',
      className: styles.livePlayer__liveDialog,
      showCancel: false,
      showClose: false,
      modal: true,
      center: true,
      onConfirm: () => {
        Dialog.close();
        // 这里可以添加您自己的业务逻辑，如跳转到首页或直播列表页
      },
      onClose: () => {
        // 这里可以添加您自己的业务逻辑，如跳转到首页或直播列表页
      },
    });
  }, []);

  // Setup event listeners
  useEffect(() => {
    if (roomEngine.instance) {
      roomEngine.instance.on(TUIRoomEvents.onAutoPlayFailed,
handleAutoPlayFailed);
    } else {
      TUIRoomEngine.once("ready", () => {
        roomEngine.instance?.on(TUIRoomEvents.onAutoPlayFailed,
handleAutoPlayFailed);
      });
    }

    subscribeEvent(LiveListEvent.ON_LIVE_ENDED, handleLiveEnded);
    subscribeEvent(LiveListEvent.ON_KICKED_OUT_OF_LIVE,
handleKickedOutOfLive);

    return () => {
```

版权所有：腾讯云计算（北京）有限责任公司

```
      roomEngine.instance?.off(TUIRoomEvents.onAutoPlayFailed,
handleAutoPlayFailed);
      unsubscribeEvent(LiveListEvent.ON_LIVE_ENDED, handleLiveEnded);
      unsubscribeEvent(LiveListEvent.ON_KICKED_OUT_OF_LIVE,
handleKickedOutOfLive);
    };
  }, [handleAutoPlayFailed, handleLiveEnded, handleKickedOutOfLive,
roomEngine.instance, subscribeEvent, unsubscribeEvent]);

  return (
    <UIKitProvider theme="dark">
      <div className={`${styles.livePlayer} ${className || ''}`}>
        <LiveView />
      </div>
    </UIKitProvider>
  );
};

export default LivePlayer;
```

LivePlayer.module.scss

```
@mixin scrollbar {
  &::-webkit-scrollbar {
    width: 6px;
    background: transparent;
  }

  &::-webkit-scrollbar-track {
    background: transparent;
  }

  &::-webkit-scrollbar-thumb {
    background: var(--uikit-color-gray-3);
    border-radius: 3px;
    border: 2px solid transparent;
    background-clip: padding-box;

    &:hover {
```

```
      background: var(--uikit-color-gray-3);
    }
  }
}

.livePlayer {
  display: flex;
  width: 100%;
  height: 100%;
  border-radius: 8px;
  overflow: hidden;
  @include scrollbar;
}

.livePlayer__liveDialog {
  text-align: center;
}
```

下一步

接入直播视频组件后，您可能还想继续接入**弹幕消息、礼物、观众列表**等功能，可以参阅下表指引文档，继续接入这些功能。

功能	描述	集成指引
聊天弹幕组件	支持发送、接收并显示文字、表情消息。	弹幕系统 (Web React)
直播送礼组件	展示配置的礼物列表，支持发送礼物、礼物播放。	礼物系统 (Web React)
观众列表组件	展示当前直播间观众信息。	观众列表组件 (Web React)

版权所有：腾讯云计算（北京）有限责任公司

语聊麦位组件

最近更新时间：2026-06-01 15:18:49

组件概述

语聊麦位组件（SeatGridView）是专门为语音聊天室场景设计的核心 UI 控件。它自动关联麦位状态，支持展示用户信息、音量波纹及房主标识。通过本文档，您可以根据业务需求（例如多人交友、游戏开黑）快速调整麦位布局或完全自定义麦位视图。

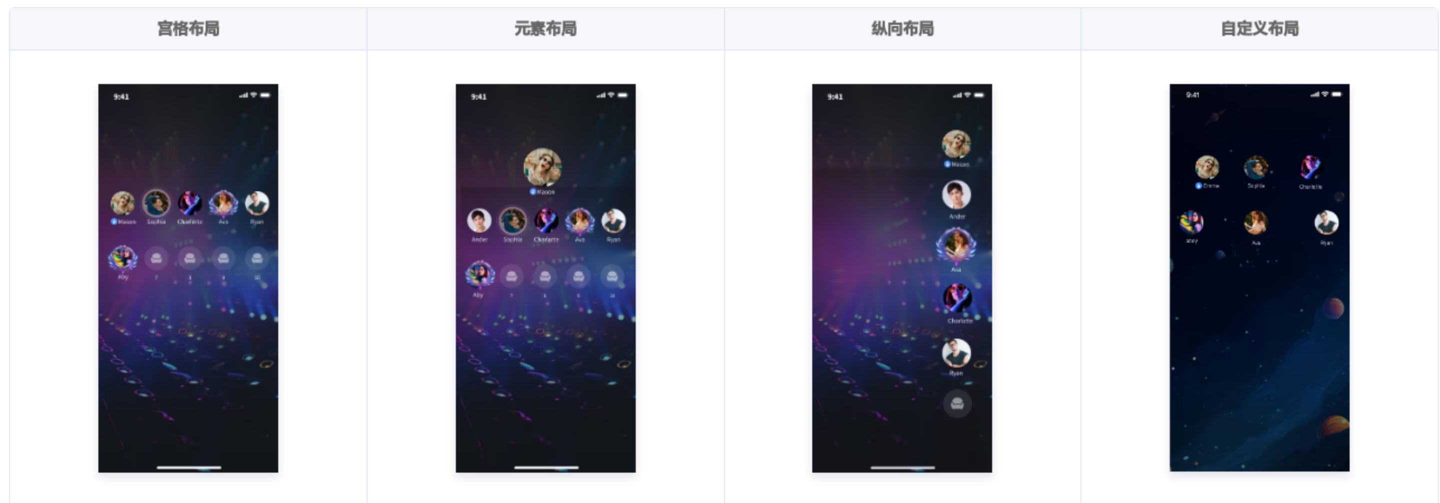
准备工作

在开始调整语聊麦位组件前，请先参考 主播开播 和 观众观看 完成主流程的搭建。

设置麦位列表布局

组件内置了多种布局模式，适用于不同的语聊场景。您可以通过 `setLayoutMode` 接口快速切换，无需手动计算位置。

效果预览



接口说明

Android

Kotlin

```
fun setLayoutMode(layoutMode: VoiceRoomDefine.LayoutMode,
layoutConfig: VoiceRoomDefine.SeatViewLayoutConfig?)
```

iOS

Swift

```swift
public func setLayoutMode(layoutMode: SGLayoutMode, layoutConfig:
SGSeatViewLayoutConfig? = nil)
```

参数	描述
`layoutMode`	布局模式： • GRID（宫格布局）：适用于多人语聊场景，所有麦位均匀分布。 • FOCUS（元素布局）：适用于有主麦位的场景，突出显示主麦位。 • VERTICAL（纵向布局）：适用于竖屏场景，麦位垂直排列。 • FREE（自定义布局）：适用于需要完全自定义布局的场景 。
`layoutConfig`	自定义的行列布局信息，仅当 `layoutMode` 为 `FREE` 时需传入此参数。

代码示例

您可以通过以下方式快速设置麦位列表布局。

Android

Kotlin

```kotlin
// 设置宫格布局
seatGridView.setLayoutMode(VoiceRoomDefine.LayoutMode.GRID, null)

// 设置元素布局
```

```
seatGridView.setLayoutMode(VoiceRoomDefine.LayoutMode.FOCUS, null)

// 设置纵向布局
seatGridView.setLayoutMode(VoiceRoomDefine.LayoutMode.VERTICAL,
null)

// 设置自定义布局
val layoutConfig = VoiceRoomDefine.SeatViewLayoutConfig().apply {
     rowConfigs = ArrayList()
     rowSpacing =  dp2px(10); // 每行的间距
 }
 // 第一行配置
 val rowConfig1 = VoiceRoomDefine.SeatViewLayoutRowConfig().apply {
     count = 3 // 第一行显示的数量
     seatSize = VoiceRoomDefine.Size(dp2px(50), dp2px(50)) // 第一行显
示的每个麦位视图大小
     seatSpacing = dp2px(10) // 第一行每个麦位的水平间距
     alignment = VoiceRoomDefine.SeatViewLayoutRowAlignment.CENTER
// 第一行麦位的对齐方式
 }
 layoutConfig.rowConfigs.add(rowConfig1)
// 第二行配置
 val rowConfig2 = VoiceRoomDefine.SeatViewLayoutRowConfig().apply {
     count = 3 // 第二行显示的数量
     seatSize = VoiceRoomDefine.Size(dp2px(50), dp2px(50)) // 第二行显
示的每个麦位大小
     seatSpacing = dp2px(10) // 第二行每个麦位的水平间距
     alignment =
VoiceRoomDefine.SeatViewLayoutRowAlignment.SPACE_AROUND // 第二行麦位
的对齐方式
 }
 layoutConfig.rowConfigs.add(rowConfig2)
 seatGridView.setLayoutMode(VoiceRoomDefine.LayoutMode.FREE,
layoutConfig)
```

Java

```
// 设置宫格布局
seatGridView.setLayoutMode(VoiceRoomDefine.LayoutMode.GRID, null)
```

```
// 设置元素布局
seatGridView.setLayoutMode(VoiceRoomDefine.LayoutMode.FOCUS, null)

// 设置纵向布局
seatGridView.setLayoutMode(VoiceRoomDefine.LayoutMode.VERTICAL,
null)

// 设置自由布局
VoiceRoomDefine.SeatViewLayoutConfig layoutConfig = new
VoiceRoomDefine.SeatViewLayoutConfig();
layoutConfig.rowConfigs = new ArrayList<>();
layoutConfig.rowSpacing =  dp2px(10); // 每行的间距
// 第一行配置
VoiceRoomDefine.SeatViewLayoutRowConfig rowConfig1 = new
VoiceRoomDefine.SeatViewLayoutRowConfig();
rowConfig1.count = 3; // 第一行显示的数量
rowConfig1.seatSize = new VoiceRoomDefine.Size(dp2px(50),dp2px(50));
// 第一行显示的每个麦位视图大小
rowConfig1.seatSpacing = dp2px(10); // 第一行每个麦位的水平间距
rowConfig1.alignment =
VoiceRoomDefine.SeatViewLayoutRowAlignment.CENTER; // 第一行麦位的对齐
方式
layoutConfig.rowConfigs.add(rowConfig1);
// 第二行配置
VoiceRoomDefine.SeatViewLayoutRowConfig rowConfig2 = new
VoiceRoomDefine.SeatViewLayoutRowConfig();
rowConfig2.count = 3; // 第二行显示的数量
rowConfig2.seatSize = new VoiceRoomDefine.Size(dp2px(50),dp2px(50));
// 第二行显示的每个麦位视图大小
rowConfig2.seatSpacing = dp2px(10); // 第二行每个麦位的水平间距
rowConfig2.alignment =
VoiceRoomDefine.SeatViewLayoutRowAlignment.SPACE_AROUND; // 第二行麦位
的对齐方式
layoutConfig.rowConfigs.add(rowConfig2);
seatGridView.setLayoutMode(VoiceRoomDefine.LayoutMode.FREE,
layoutConfig);
```

iOS

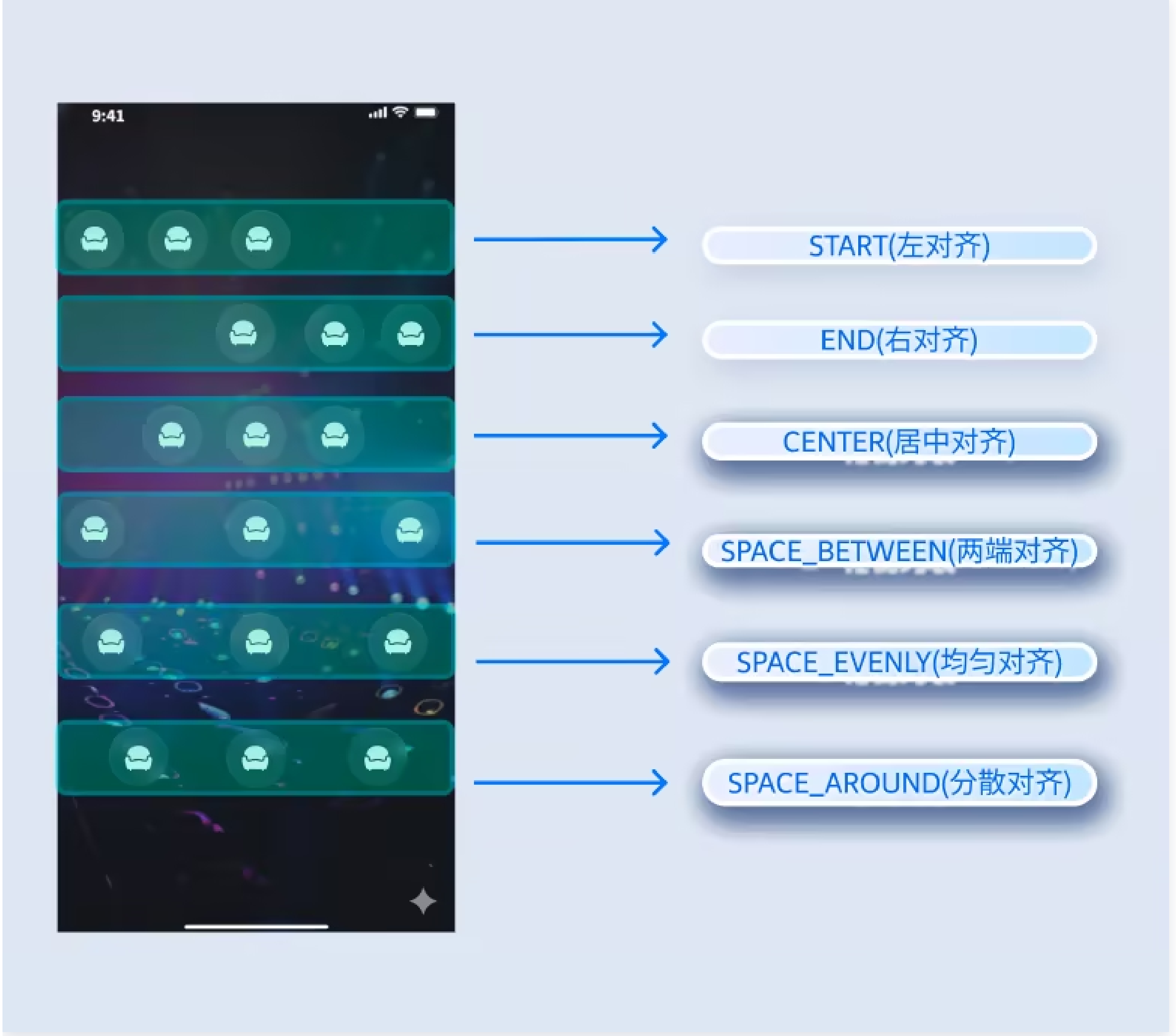
Swift

```
import LiveStreamCore

// 设置宫格布局
seatGridView.setLayoutMode(layoutMode: .grid)

// 设置元素布局
seatGridView.setLayoutMode(layoutMode: .focus)

// 设置纵向布局
seatGridView.setLayoutMode(layoutMode: .vertical)

// 设置自定义布局
// 第一行配置
let rowConfig1 = SGSeatViewLayoutRowConfig(count: 3, // 第一行显示的数
量
                                            seatSpacing: 10, // 第一行
每个麦位的水平间距
                                            seatSize: CGSize(width:
50, height: 50), // 第一行显示的每个麦位视图大小
                                            alignment: .center) // 第
一行麦位的对齐方式

 // 第二行配置
let rowConfig2 = SGSeatViewLayoutRowConfig(count: 3, // 第二行显示的数
量
                                            seatSpacing: 10, // 第二行
每个麦位的水平间距
                                            seatSize: CGSize(width:
50, height: 50), // 第二行显示的每个麦位视图大小
                                            alignment: .spaceAround)
// 第二行麦位的对齐方式

let layoutConfig = SGSeatViewLayoutConfig(rowConfigs: [rowConfig1,
rowConfig2],
```

版权所有：腾讯云计算（北京）有限责任公司

```
                                   rowSpacing: 10)

seatGridView.setLayoutMode(layoutMode: .free, layoutConfig:
layoutConfig)
```

自定义布局对齐方式示意图:

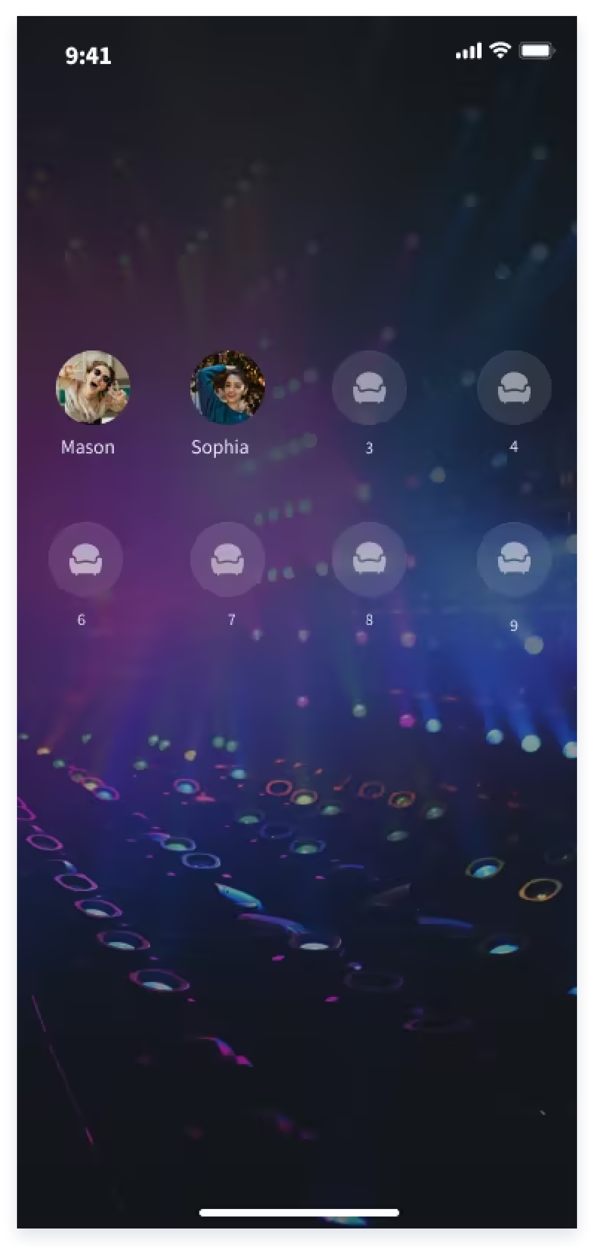


自定义麦位视图 UI

如果默认的麦位样式无法满足业务需求，您可以通过设置适配器（ `Adapter/Delegate` ）来完全托管麦位的渲染逻辑。

效果预览

默认麦位视图	自定义麦位视图示例



接口说明

Android

Kotlin

```kotlin
interface SeatViewAdapter {
    fun createSeatView(seatGridView: SeatGridView, seatInfo:
TUIRoomDefine.SeatInfo): View
```

```
    fun updateSeatView(seatGridView: SeatGridView, seatInfo:
TUIRoomDefine.SeatInfo, seatView: View, )
    fun updateUserVolume(seatGridView: SeatGridView, volume: Int,
seatView: View)
}

fun setSeatViewAdapter(adapter: VoiceRoomDefine.SeatViewAdapter?)
```

参数	描述
`seatGridView`	当前语聊麦位组件。
`seatInfo`	当前麦位信息。
`volume`	当前麦上用户音量。
`seatView`	您自定义的麦位 `view` 。
`adapter`	自定义麦位适配器。

iOS

Swift

```swift
@objc public protocol SGSeatViewDelegate {
    func seatGridView(_ view: SeatGridView, createSeatView seatInfo:
TUISeatInfo) -> UIView?
    func seatGridView(_ view: SeatGridView, updateSeatView seatInfo:
TUISeatInfo, seatView: UIView)
    func seatGridView(_ view: SeatGridView, updateUserVolume volume:
Int, seatView: UIView)
}

public func setSeatViewDelegate(_ delegate: SGSeatViewDelegate)
```

参数	描述
`view`	当前语聊麦位组件。
`seatInfo`	当前麦位信息。
`volume`	当前麦上用户音量。
`seatView`	您自定义的麦位 `view` 。
`delegate`	自定义麦位 `delegate` 。

代码示例

Android

如果默认的 UI 不满足需求，您可以自定义麦位 UI ，可参考如下方式快速设置麦位布局。

Kotlin

```kotlin
val adapter = object : VoiceRoomDefine.SeatViewAdapter {
    override fun createSeatView(seatGridView: SeatGridView,
seatInfo: TUIRoomDefine.SeatInfo): View {
        return TestSeatInfoView(context, seatGridView, seatInfo)
    }

    override fun updateSeatView(seatGridView: SeatGridView,seatInfo:
TUIRoomDefine.SeatInfo, seatView: View) {
        (seatView as TestSeatInfoView).updateSeatView(seatGridView,
seatInfo)
    }

    override fun updateUserVolume(seatGridView: SeatGridView,
volume: Int, customSeatView: View) {
        (customSeatView as
TestSeatInfoView).updateUserVolume(seatGridView, volume)
```

```
    }
}

seatGridView.setSeatViewAdapter(adapter)

class TestSeatInfoView constructor(context: Context, seatGridView:
SeatGridView, seatInfo: TUIRoomDefine.SeatInfo) :
FrameLayout(context) {
    init {
        initView() //初始化view
    }

    fun updateSeatView(seatGridView: SeatGridView, seatInfo:
TUIRoomDefine.SeatInfo) {
        updateView(seatInfo) //更新自定义麦位视图UI
    }

    fun updateUserVolume(seatGridView: SeatGridView, volume: Int) {
        updateUserVolume(volume) //更新音量变化UI
    }
}
```

Java

```java
VoiceRoomDefine.SeatViewAdapter adapter = new
VoiceRoomDefine.SeatViewAdapter() {
    @Override
    public View createSeatView(SeatGridView seatGridView,
TUIRoomDefine.SeatInfo seatInfo) {
        return new TestSeatInfoView(getApplicationContext(),
seatGridView, seatInfo);
    }

    @Override
    public void updateSeatView(SeatGridView seatGridView,
TUIRoomDefine.SeatInfo seatInfo,
                                View customSeatView) {
        ((TestSeatInfoView)
customSeatView).updateSeatView(seatGridView, seatInfo);
```

```
    }

    @Override
    public void updateUserVolume(SeatGridView seatGridView, int
volume, View customSeatView) {
        ((TestSeatInfoView)
customSeatView).updateUserVolume(seatGridView, volume);
    }
};
seatGridView.setSeatViewAdapter(adapter);

public class TestSeatInfoView extends FrameLayout {

    public TestSeatInfoView(@NonNull Context context, SeatGridView
seatGridView, TUIRoomDefine.SeatInfo seatInfo) {
        super(context);
        initView();
    }

    public void updateSeatView(SeatGridView seatGridView,
TUIRoomDefine.SeatInfo seatInfo) {
        updateView(seatInfo);
    }

    public void updateUserVolume(SeatGridView seatGridView, int
volume) {
        updateUserVolume(volume);
    }
}
```

iOS

如果默认的 UI 不满足需求，您可以自定义麦位 UI ，可参考如下方式快速设置麦位布局。

Swift

```
import LiveStreamCore
```

```
class TestSeatViewDelegate: SGSeatViewDelegate {
    func seatGridView(_ view: SeatGridView, createSeatView seatInfo:
TUISeatInfo) -> UIView? {
      return TestSeatInfoView(seatGridView: view, seatInfo:
seatInfo)
    }

    func seatGridView(_ view: SeatGridView, updateSeatView seatInfo:
TUISeatInfo, seatView: UIView) {
      if let seatView = seatView as? TestSeatInfoView {
        seatView.updateSeatView(seatGridView: view, seatInfo:
seatInfo)
      }
    }

    func seatGridView(_ view: SeatGridView, updateUserVolume volume:
Int, seatView: UIView) {
      if let seatView = seatView as? TestSeatInfoView {
        seatView.updateUserVolume(seatGridView: view, volume:
volume)
      }
    }
}

seatGridView.setSeatViewDelegate(TestSeatViewDelegate())

class TestSeatInfoView: UIView {
    init(seatGridView: SeatGridView, seatInfo: TUISeatInfo) {
        super.init(frame: .zero)
        initView() // 初始化view
    }

    func updateSeatView(seatGridView: SeatGridView, seatInfo:
TUISeatInfo) {
        updateView(seatInfo) // 更新自定义麦位视图UI
    }

    func updateUserVolume(seatGridView: SeatGridView, volume: Int) {
        updateUserVolume(volume) // 更新音量变化UI
```

版权所有：腾讯云计算（北京）有限责任公司

```
        }
    }
```

常见问题

自定义适配器后，如何获取麦位点击事件?

您需要在自定义 `View` 内部设置点击监听，在回调中处理业务逻辑。

切换布局后麦位数据会丢失吗?

不会。`SeatGridView` 内部维护了数据状态，`setLayoutMode` 仅改变视觉排布，不会影响用户的上麦状态。

如何监听麦位列表的实时变化（例如用户上麦/下麦）?

您无需在适配器中主动监听。`SeatGridView` 内部已实现了麦位注册监听，当任何麦位状态变化时，会自动回调 `updateSeatView` 方法。您只需在该方法中更新 UI 即可。

版权所有：腾讯云计算（北京）有限责任公司

调整界面风格
调整界面风格（Web 桌面浏览器）

最近更新时间：2026-07-02 11:12:46

默认的直播界面能够帮您极速跑通流程，但真实的商业场景往往需要专属的品牌特点。为了让直播间完美契合您的业务诉求，TUILiveKit 观看端提供了三类灵活的 UI 控制能力：

- 一键切换场景主题 `(Style Preset)`：只需一行配置，即可加载商务等行业专属主题，快速匹配页面气质。
- 按需裁剪界面布局：灵活隐藏或开启特定的功能模块，减少非关键干扰，聚焦核心流程。
- 强大的扩展能力：支持无缝嵌入您的专属业务组件（如自定义功能按钮、悬浮业务卡片等），兼顾标准化与灵活性。

前置条件

从 GitHub 下载 TUILiveKit Demo 源码，安装依赖，并且完成相关账号信息配置，可参考 跑通 Demo 。

如何加载不同界面

以商务场景为例：在应用根组件 `App.vue` 中，仅需要为 `UIKitProvider` 增加 `style-preset="business"` 即可。

```
<UIKitProvider theme="dark" style-preset="business">
  <router-view />
```

版权所有：腾讯云计算（北京）有限责任公司

```
</UIKitProvider>
```

适用场景说明

商务场景

商务场景（ `business` ）是我们为企业官网和客户演示场景提供的一套直播界面风格。相比于默认风格，它的核心目标是：让页面更专业、更稳重、更容易建立信任感。

商务场景预设对界面进行了以下优化：

优化维度	具体调整	设计意图
视觉色调	采用深色系背景 + 低饱和度的强调色	营造沉稳、专业的视觉氛围，契合企业品牌调性
布局结构	精简顶部信息区，突出视频主体	减少视觉干扰，让观众聚焦于演示内容
控件风格	底部栏采用简洁的图标式设计，悬停时显示文字提示	保持界面清爽的同时，确保功能可发现性
交互反馈	采用微妙的过渡动画和状态变化	传递精致、可靠的产品质感
信息密度	隐藏娱乐向元素（如礼物特效、弹幕动画等）	聚焦核心业务流程，适配正式的商务沟通场景

版权所有：腾讯云计算（北京）有限责任公司

自定义底部栏按钮

> **说明：**
> 我们的商务场景 UI 属于专业定制，能满足大多数标准企业需求。同时，也为了适配您独特的品牌诉求，我们开放了强大 UI 自定义接口与插槽能力。

底部控制栏是高频操作区域，通过提供底部栏的 UI 定制接口，您可以轻松打通直播间与自身业务的闭环：

- 替换默认控件：将默认的 **“画中画”**、**“切换分辨率”**或**“全屏”**等按钮，替换为您团队自行设计的、符合品牌规范的 UI 图标。
- 新增业务入口：在底部栏中插入**“联系专属顾问”**、**“一键分享”**或**“下载白皮书”**等自定义功能按钮。

```
// 示例：底部栏增加联系顾问按钮
<script setup lang="ts">
import { useLivePlayerState } from 'tuikit-atomicx-vue3';
import ConsultIcon from './icons/ConsultIcon.vue';  // 建议替换为您项目中实际的客服或顾问图标组件

const { addCustomButtons } = useLivePlayerState();
const handleConsultClick = () => {
  // TODO: 替换为您的真实业务闭环，例如：唤起一个企业微信客服弹窗、打开侧边留资抽屉，或者跳转到第三方客服链接
  console.log('触发联系顾问操作，准备唤起业务弹窗...');
  alert('您好，您的专属业务顾问即将为您服务！');
};

addCustomButtons([
  {
    id: 'contact-consultant',
    icon: ConsultIcon,
    onClick: handleConsultClick,
    tooltip: '联系顾问',
    position: 'end',
  },
]);
</script>
```

自定义视频区域

视频区域是整个页面的核心视觉焦点。利用视频区域的图层叠加插槽，您可以在视频流上方悬浮**任何自定义的 UI 元素**：

- 品牌水印与贴片：在视频画面的角落固定展示**企业的 Logo、专属活动水印**或**版权声明**，增强品牌曝光；
- 动态业务面板：在视频上方悬浮展示**主讲人的详细介绍卡片、实时滚动的数据看板**。

```
// 示例：视频区域增加企业 Logo 水印
<LiveView>
  <template #center-overlay>
   <div class="brand-watermark">
     <img src="/your-company-logo.png" alt="Logo" />
   </div>
 </template>
</LiveView>
```

> **注意：**
> 如需获取更多关于上述 UI 自定义接口或插槽的详细使用说明，详细请参见 直播视频组件 。

直播列表

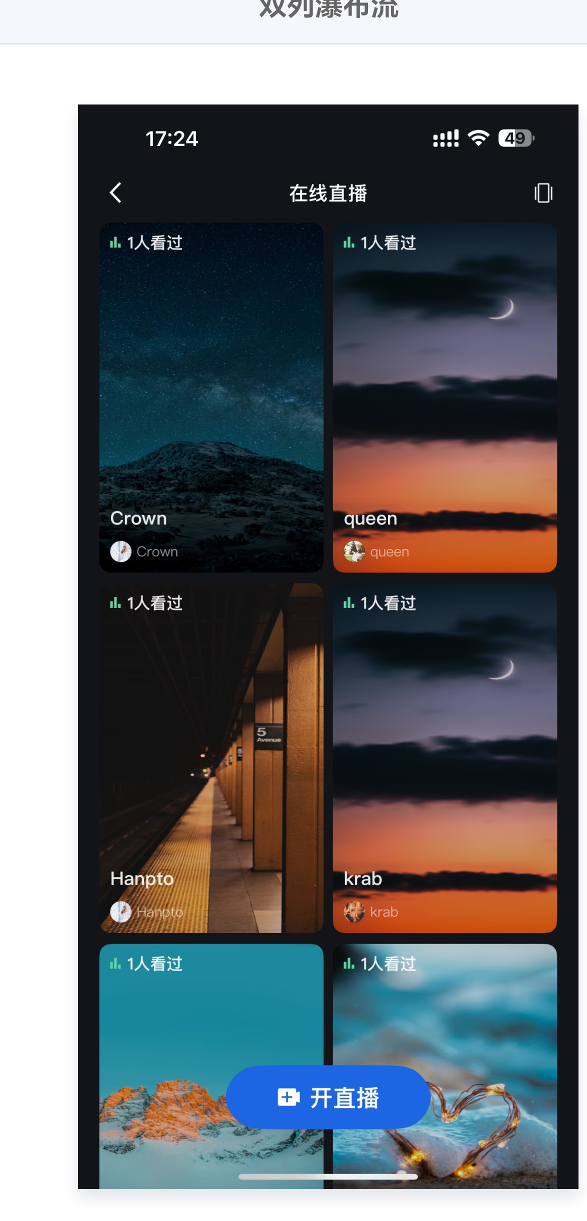
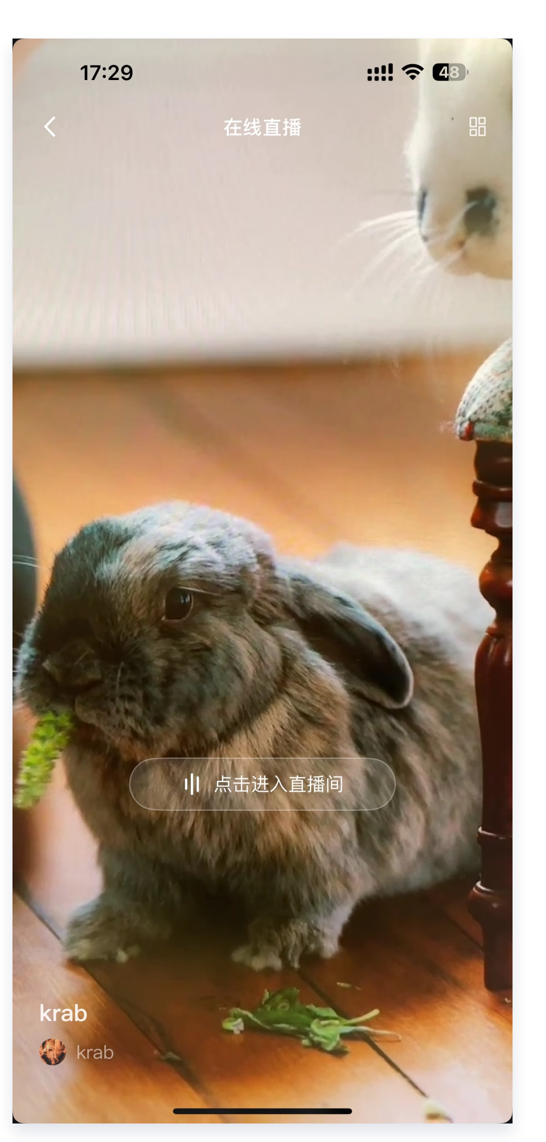
直播列表（Android）

最近更新时间：2026-06-22 10:22:04

功能预览

本文对 TUILiveKit 中的**直播列表页面**进行了详细的介绍，您可以在已有项目中直接参考本文档集成我们开发好的直播列表页面，也可以根据您的需求按照文档中的内容对页面的样式，布局以及功能项进行深度的定制。

- **双列瀑布流**：默认同时预览2个直播间的画面。
- **单列瀑布流**：默认同时预览1个直播间的画面（**仅视频直播支持单列预览**）。

双列瀑布流	单列瀑布流
	

> **注意：**
> 预览直播间画面时，预览时长都会被计入观众的音视频时长，组件详细计费相关内容请参考 计费说明 。

快速接入

步骤 1. 开通服务

参考 开通服务 文档开通「体验版」或「大规模直播版」套餐。

步骤 2. 代码集成

参考 准备工作 接入 `TUILiveKit` 。

步骤 3. 添加直播列表瀑布流视图

您可以根据需要选择使用单列或双列瀑布流。

Kotlin

```kotlin
import android.content.Context
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import com.trtc.uikit.livekit.features.livelist.LiveListView
import com.trtc.uikit.livekit.features.livelist.Style

class YourActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // 1.创建并初始化瀑布流视图
        val view = createLiveListView(this)
        // 2.将创建的 LiveListView 添加到您的 Activity 或 Fragment 中
        setContentView(view)
    }

    fun createLiveListView(context: Context): LiveListView {
        val liveListView = LiveListView(context)
        // 此处代码为双列瀑布流初始化示例，若您需要使用单列瀑布流，请将初始化代码改为
liveListView.init(this, Style.SINGLE_COLUMN)
        liveListView.init(this, Style.DOUBLE_COLUMN)
        return liveListView
    }
```

```
}
```

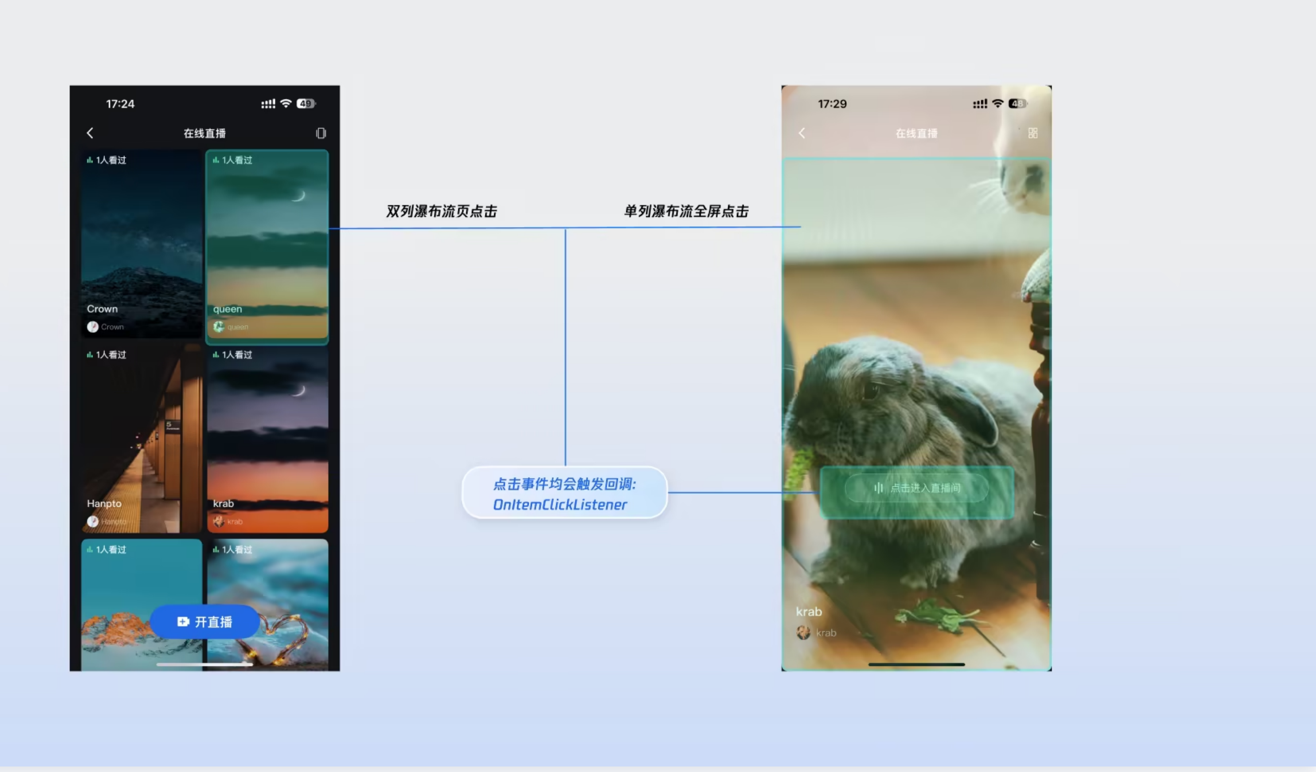
Java

```java
import android.content.Context;
import android.os.Bundle;

import androidx.appcompat.app.AppCompatActivity;

import com.trtc.uikit.livekit.features.livelist.LiveListView;
import com.trtc.uikit.livekit.features.livelist.Style;

public class YourActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // 1.创建并初始化瀑布流视图
        LiveListView view = createLiveListView(this);
        // 2.将创建的 LiveListView 添加到您的 Activity 或 Fragment 中
        setContentView(view);
    }

    private LiveListView createLiveListView(Context context) {
        LiveListView liveListView = new LiveListView(context);
        // 此处代码为双列瀑布流初始化示例，若您需要使用单列瀑布流，请将初始化代码改为
liveListView.init(this, Style.SINGLE_COLUMN, null, null);
        liveListView.init(this, Style.DOUBLE_COLUMN, null, null);
        return liveListView;
    }
}
```

步骤 4. 实现直播列表页到观众观看页的跳转

双列瀑布流直播列表和单列瀑布流均会通过 `OnItemClickListener` 回调点击事件，您只需在直播列表瀑布流视图中实现 `OnItemClickListener` 来响应用户的点击事件，并在 `onItemClick` 中实现跳转进入观众观看页的功能，观众观看页实现可参考 观众观看 。

交互示例：



代码示例：

Kotlin

```kotlin
fun createLiveListView(context: Context): LiveListView {
    val liveListView = LiveListView(context)
    // 此处代码为双列瀑布流初始化示例，若您需要使用单列瀑布流，请将初始化代码改为
liveListView.init(this, Style.SINGLE_COLUMN)
    liveListView.init(this, Style.DOUBLE_COLUMN)

    liveListView.setOnItemClickListener { view, liveInfo ->
        // 点击直播列表项时，跳转进入到观众观看页面
        val intent = Intent(context, YourAudienceActivity::class.java)
        intent.putExtra("liveId", liveInfo.roomId)
        context.startActivity(intent)
    }
    return liveListView
}
```

Java

```java
private LiveListView createLiveListView(Context context) {
```

```
    LiveListView liveListView = new LiveListView(context);
    // 此处代码为双列瀑布流初始化示例，若您需要使用单列瀑布流，请将初始化代码改为
liveListView.init(this, Style.SINGLE_COLUMN, null, null);
    liveListView.init(this, Style.DOUBLE_COLUMN, null, null);

    liveListView.setOnItemClickListener((view, liveInfo) -> {
        // 点击直播列表项时，跳转进入到观众观看页面
        Intent intent = new Intent(context, YourAudienceActivity.class);
        intent.putExtra("liveId", liveInfo.roomId);
        context.startActivity(intent);
    });
    return liveListView;
}
```

自定义您的界面布局

TUILiveKit 提供了灵活的接口定制直播列表瀑布流组件，您可以根据业务需求自定义数据源和列表项样式。

自定义数据源

如果您的后台有单独的直播列表数据，可以通过实现 `LiveListDataSource` 接口来自定义数据源，而不使用组件默认的列表数据。

Kotlin

```kotlin
// 1.导入依赖
import com.trtc.uikit.livekit.features.livelist.LiveListDataSource;

// 2.通过实现 LiveListDataSource 自定义数据源
val dataSource = object : LiveListDataSource {
    override fun fetchLiveList(param: FetchLiveListParam,callback:
LiveListCallback) {
        // 对接自己的业务后台，按照下面的格式返回数据给UI组件
        val liveInfoList = mutableListOf<TUILiveListManager.LiveInfo>()
        val liveInfo = TUILiveListManager.LiveInfo().apply {
            roomInfo = TUIRoomDefine.RoomInfo().apply {
                roomId = "live_123456"
                name = "live_123456"
            }
        }
```

```
        liveInfoList.add(liveInfo)
        val cursor = "aabbccdd"
        callback.onSuccess(cursor, liveInfoList)
    }
}

// 3.初始化时传入自定义的 dataSource
liveListView.init(this, Style.DOUBLE_COLUMN, dataSource = dataSource)
```

Java

```
// 1.导入依赖
import com.trtc.uikit.livekit.features.livelist.LiveListDataSource;

// 2.通过实现 LiveListDataSource 自定义数据源
LiveListDataSource dataSource = (param, callback) -> {
    //对接自己的业务后台，按照下面的格式返回数据给UI组件
    List<TUILiveListManager.LiveInfo> liveInfoList = new ArrayList<>();
    TUILiveListManager.LiveInfo liveInfo = new
TUILiveListManager.LiveInfo();
    liveInfo.roomInfo = new TUIRoomDefine.RoomInfo();
    liveInfo.roomInfo.roomId = "live_123456";
    liveInfo.roomInfo.name = "live_123456";
    liveInfoList.add(liveInfo);
    String cursor = "aabbccdd";
    callback.onSuccess(cursor, liveInfoList);
};

// 3.初始化时传入自定义的 dataSource
liveListView.init(this, Style.DOUBLE_COLUMN, null, dataSource);
```

自定义挂件

瀑布流列表项底部默认显示视频流画面或直播封面。如果您需要自定义列表项顶部的 UI 元素（例如主播头像、直播标题等），可以通过实现 `LiveListViewAdapter` 接口来完成。

Kotlin

```
// 1.导入依赖
```

版权所有：腾讯云计算（北京）有限责任公司

```
import com.trtc.uikit.livekit.features.livelist.LiveListViewAdapter;

// 2.通过实现 LiveListViewAdapter 自定义挂件
val liveListViewAdapter = object : LiveListViewAdapter {
    override fun createLiveInfoView(liveInfo:
TUILiveListManager.LiveInfo): View {
        // 自定义挂件 view
        val widgetView = YourView(context)
        widgetView.init(liveInfo)
        return widgetView
    }


    override fun updateLiveInfoView(view: View, liveInfo:
TUILiveListManager.LiveInfo) {
        // 更新挂件 view 中绑定的数据
        val widgetView = view as YourView
        widgetView.updateLiveInfoView(liveInfo)
    }
}

// 3.初始化时传入自定义的 liveListViewAdapter
liveListView.init(this, Style.DOUBLE_COLUMN, adapter =
liveListViewAdapter)
```

Java

```
// 1.导入依赖
import com.trtc.uikit.livekit.features.livelist.LiveListViewAdapter;

// 2.通过实现 LiveListViewAdapter 自定义挂件
LiveListViewAdapter liveListViewAdapter = new LiveListViewAdapter() {
    @Override
    public View createLiveInfoView(TUILiveListManager.LiveInfo liveInfo)
{
        // 自定义挂件view
        CustomView widgetView = new CustomView(context);
        widgetView.init(liveInfo);
        return widgetView;
```

```
    }


    @Override
    public void updateLiveInfoView(View view,
TUILiveListManager.LiveInfo liveInfo) {
        // 更新挂件view中绑定的数据
        CustomView widgetView = (CustomView) view;
        widgetView.updateLiveInfoView(liveInfo);
    }
};

// 3.初始化时传入自定义的 liveListViewAdapter
liveListView.init(this, Style.DOUBLE_COLUMN, liveListViewAdapter, null);
```

下一步

恭喜您，现在您已经成功集成了 **直播列表** 功能。接下来，您可以根据您的业务场景实现**主播开播**、**观众观看**等功能

视频直播场景：

功能	描述	集成指引
主播开播	实现主播开播全流程功能，包括开播前的准备和开播后的各种互动。	主播开播
观众观看	实现观众进入主播的直播间后观看直播，实现观众连麦 、直播间信息、在线观众、弹幕显示等功能。	观众观看

语聊房场景：

功能	描述	集成指引
主播开播	实现主播开播语聊房全流程功能，包括开播前的准备和开播后的各种互动。	主播开播
观众观看	实现观众进入主播的语聊房后进行互动，如上麦、收发弹幕等功能。	观众观看

常见问题

集成直播列表功能后页面没有任何直播怎么办？

如果您看到空白页面，需要检查您是否已完成 登录步骤 。为了测试该功能，您可以使用两台设备：一台设备用于开播，另一台设备在直播列表页面，就能拉取到已开播的直播间。

单列瀑布流和双列瀑布流有什么区别？

版权所有：腾讯云计算（北京）有限责任公司

单列瀑布流一次预览一个直播间画面，而双列瀑布流则同时预览两个直播间画面。您可以根据自己的设计需求选择合适的布局。

预览直播画面是否收费?

是的，预览直播间画面时长会计入观众的音视频时长，这部分是需要付费的。详细的 计费说明 可以参考文档中的相关内容。

我使用了自定义数据源 (LiveListDataSource)，但列表不显示/不刷新怎么办?

请确保您正确实现了 `LiveListDataSource` 接口。重点检查以下几点:

1. 检查 `fetchLiveList` 方法是否被正确调用。
2. 确保在获取数据后（无论成功或失败）都调用了 `callback.onSuccess` 或 `callback.onFailure` 。
3. 检查您的业务后台接口是否返回了正确的数据格式。

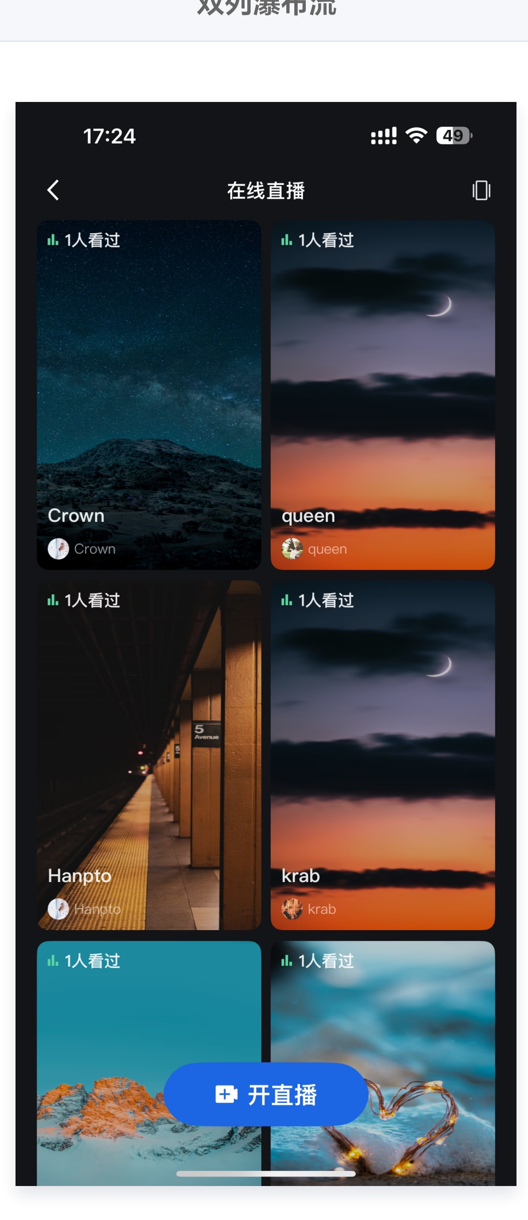
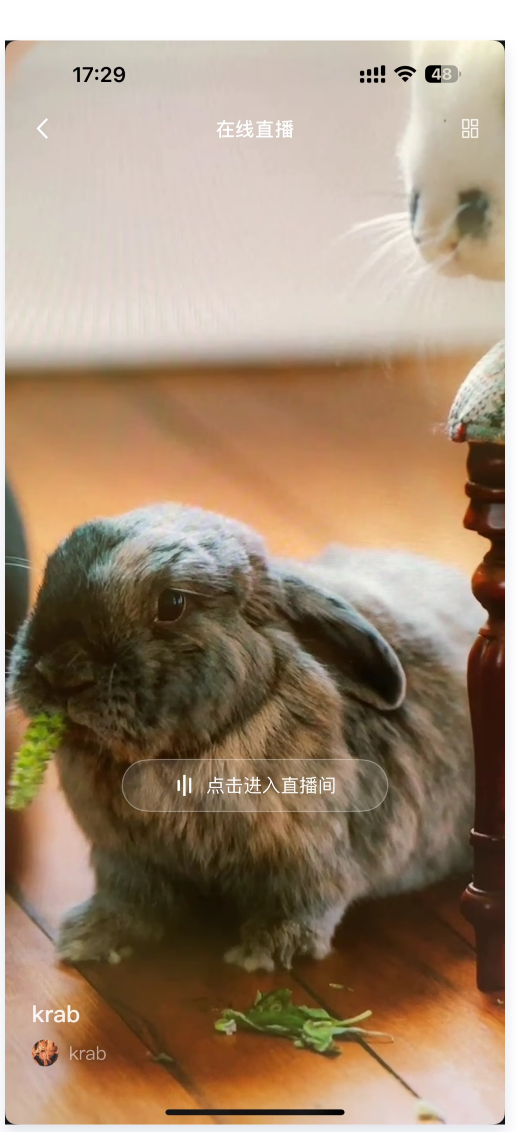
直播列表（iOS）

最近更新时间：2026-06-22 10:22:10

功能预览

本文对 TUILiveKit 中的**直播列表页面**进行了详细的介绍，您可以在已有项目中直接参考本文档集成我们开发好的直播列表页面，也可以根据您的需求按照文档中的内容对页面的样式，布局以及功能项进行深度的定制。

- **双列瀑布流**：默认同时预览2个直播间的画面。
- **单列瀑布流**：默认同时预览1个直播间的画面（**仅视频直播支持单列预览**）。

双列瀑布流	单列瀑布流
	

> **注意：**

> 预览直播间画面时，预览时长都会被计入观众的音视频时长，组件详细计费相关内容请参考 计费说明 。

快速接入

步骤1：开通服务

参考 开通服务 文档开通「体验版」或「大规模直播版」套餐。

步骤2：代码集成

参考 准备工作 接入 `TUILiveKit` 。

步骤3：添加直播列表瀑布流视图

您可以根据需要选择使用单列或双列瀑布流。

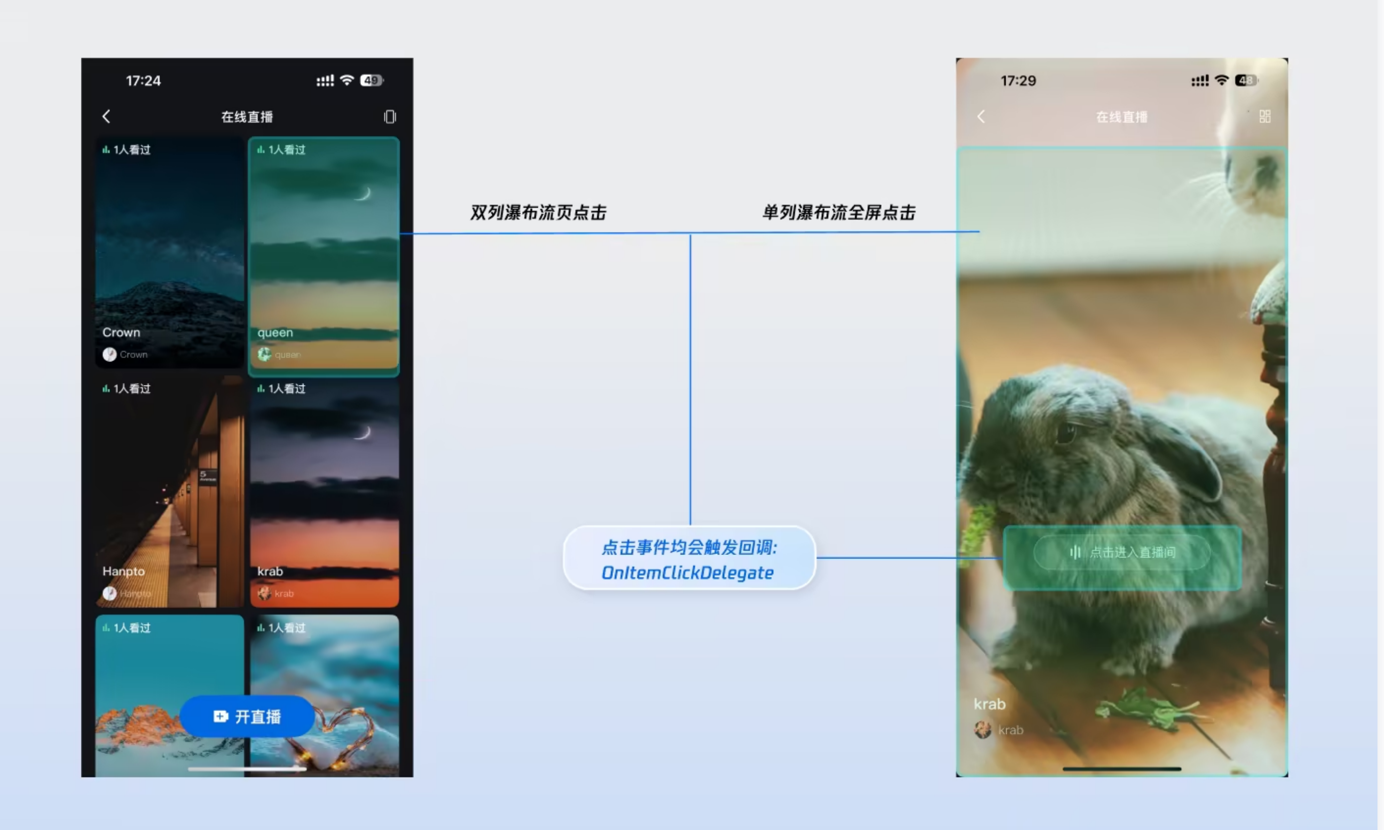
Swift

```swift
// 示例： YourLiveListViewController 代表您直播列表瀑布流的视图控制器
class YourLiveListViewController: UIViewController {

    // 1. 声明 liveListView 作为成员变量
    //         - style: .doubleColumn // 双列瀑布流
    //         - style: .singleColumn // 单列瀑布流
    // 示例为双列瀑布流，如需单列传入 .singleColumn即可
    private let liveListView = LiveListView(style: .doubleColumn)

    public override func viewDidLoad() {
        super.viewDidLoad()
        // 2. 将 liveListView 添加到视图上
        view.addSubview(liveListView)
        liveListView.snp.makeConstraints { make in
            make.edges.equalToSuperview()
        }
        // 3. 设置列表的点击事件代理
        liveListView.itemClickDelegate = self
    }
}
```

步骤4：实现直播列表页到观众观看页的路由跳转

双列瀑布流直播列表 和 **单列瀑布流** 均会通过 `OnItemClickDelegate` 回调点击事件，您只需在直播列表视图控制器中实现 `OnItemClickDelegate` 来响应用户的点击事件，并在 `onItemClick` 中实现跳转进入观众观看页的功能，观众观看页的实现可参考 观众观看：

交互示例：



代码示例：

Swift

```swift
// 在您的直播瀑布流列表视图控制器中实现 OnItemClickDelegate，示例：
YourLiveListViewController
extension YourLiveListViewController: OnItemClickDelegate {

    func onItemClick(liveInfo: LiveInfo, frame: CGRect) {
        // 1. 实例化您的观众观看视图控制器
        let audienceVC = YourAudienceViewController(roomId:
liveInfo.roomId)
        audienceVC.modalPresentationStyle = .fullScreen
        // 2. 跳转到您的观众观看视图控制器
        present(audienceVC, animated: false)
    }
```

```
}
```

步骤5：直播列表的页面跳转交互优化

为了让您在页面跳转过程中获得更加完整与流畅的体验，结合 iOS 页面路由的系统特性， `LiveListView` 组件提供了 `refreshLiveList` 和 `onRouteToNextPage` 方法，方便您仅通过简单的一行代码调用，就能优化直播列表展示与视频流播放体验。具体优化方式如下：

```Swift
// 在您的直播瀑布流列表视图控制器中实现，示例：YourLiveListViewController
class YourLiveListViewController: UIViewController {

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)
        // refreshLiveList: 在viewDidAppear时调用此方法，以确保每次回到当前页面都能拿到最新的列表
        liveListView.refreshLiveList()
    }

    override func viewWillDisappear(_ animated: Bool) {
        super.viewWillDisappear(animated)
        // onRouteToNextPage: 在viewWillDisappear时调用此方法，当您进入别的页面时停止当前直播列表页播放的视频流
        liveListView.onRouteToNextPage()
    }
}
```

自定义您的界面布局

TUILiveKit 提供了灵活的接口定制直播列表瀑布流组件，您可以根据业务需求自定义数据源和列表项样式。

自定义数据源

如果您的后台有单独的直播列表数据，可以通过实现 `LiveListDataSource` 接口来自定义数据源，而不使用组件默认的列表数据。

```Swift
```

```
// 示例：YourLiveListViewController 代表您直播列表瀑布流的视图控制器
class YourLiveListViewController: UIViewController {

    private let liveListView: LiveListView  =  LiveListView(style:
.doubleColumn)

    public override func viewDidLoad() {
        super.viewDidLoad()
        view.addSubview(liveListView)
        liveListView.snp.makeConstraints { make in
            make.edges.equalToSuperview()
        }
        liveListView.itemClickDelegate = self
        // 1. 设置自定义数据源代理
        liveListView.dataSource = self
    }
}

// 2. 实现自定义数据源代理：LiveListDataSource
extension YourLiveListViewController: LiveListDataSource {

    public func fetchLiveList(cursor: String, onSuccess: @escaping
LiveListBlock, onError: @escaping TUIErrorBlock) {
        // 3. 对接自己的业务后台，按照下面的格式返回数据给UI组件
        var liveInfoList: [LiveInfo] = []
        var liveInfo = LiveInfo()
        liveInfo.roomId = "live_123456"
        liveInfo.name = "live_123456"
        liveInfoList.append(liveInfo)
        let cursor = "aabbccdd"
        onSuccess(cursor, liveInfoList)
    }
}
```

自定义挂件

瀑布流列表项底部默认显示视频流画面或直播封面。如果您需要自定义列表项顶部的 UI 元素（例如主播头像、直播标题等），可以通过实现 `LiveListViewAdapter` 接口来完成。

Swift

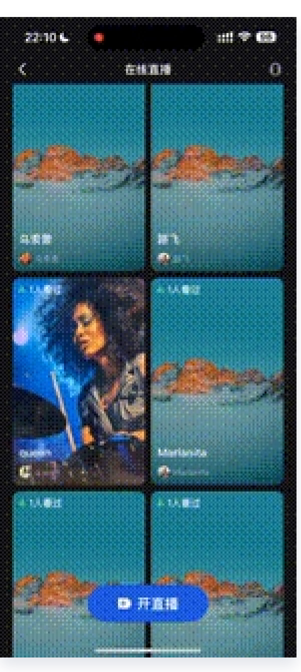
```
// 示例: YourLiveListViewController 代表您直播列表瀑布流的视图控制器
class YourLiveListViewController: UIViewController {

    private let liveListView: LiveListView  =  LiveListView(style:
.doubleColumn)

    public override func viewDidLoad() {
        super.viewDidLoad()
        view.addSubview(liveListView)
        liveListView.snp.makeConstraints { make in
            make.edges.equalToSuperview()
        }
        liveListView.itemClickDelegate = self
        liveListView.dataSource = self
        // 1. 设置自定义挂件代理
        liveListView.adapter = self
    }
}

// 2. 实现自定义挂件代理
extension YourLiveListViewController: LiveListViewAdapter {
    public func createLiveInfoView(info: LiveInfo) -> UIView {
        // 自定义挂件view
        return YourCustomView(liveInfo: info)
    }

    public func updateLiveInfoView(view: UIView, info: LiveInfo) {
        if let infoView = view as? YourCustomView {
            // 更新挂件view中绑定的数据
            infoView.updateView(liveInfo: info)
        }
    }
}
```

自定义转场动画

在**双列瀑布流直播列表**和**单列瀑布流**中跳转到观众观看页，为了转场效果更好，往往需要自定义转场动画。`TUILiveKit` 在 `GitHub` 仓库开源了直播转场动画 LiveTransitioningDelegate ，您只需设置 `audienceVC.transitioningDelegate` 即可实现跟 TUILiveKit 一样的转场动画：

Swift

```swift
// 在您的直播瀑布流列表视图控制器中的 OnItemClickDelegate 回调中，示例：YourLiveListViewController
extension YourLiveListViewController: OnItemClickDelegate {

    func onItemClick(liveInfo: LiveInfo, frame: CGRect) {
        let audienceVC = YourAudienceViewController(roomId: liveInfo.roomId)
        audienceVC.modalPresentationStyle = .fullScreen
        // 设置过渡动画代理
        audienceVC.transitioningDelegate = LiveTransitioningDelegate(originFrame: frame)
        present(audienceVC, animated: false)
    }

}
```

双列瀑布流直播列表转场动画	单列瀑布流转场动画
	

下一步

恭喜您，现在您已经成功集成了 **直播列表** 功能。接下来，您可以根据您的业务场景实现**主播开播、观众观看**等功能
视频直播场景：

功能	描述	集成指引
主播开播	实现主播开播全流程功能，包括开播前的准备和开播后的各种互动	主播开播
观众观看	实现观众进入主播的直播间后观看直播，实现观众连麦 、直播间信息、在线观众、弹幕显示等功能	观众观看

语聊房场景：

功能	描述	集成指引
主播开播	实现主播开播语聊房全流程功能，包括开播前的准备和开播后的各种互动。	主播开播
观众观看	实现观众进入主播的语聊房后进行互动，如上麦、收发弹幕等功能。	观众观看

常见问题

集成直播列表功能后页面没有任何直播怎么办?

如果您看到空白页面，需要检查您是否已完成 登录步骤 。为了测试该功能，您可以使用两台设备：一台设备用于开播，另一台设备在直播列表页面，就能拉取到已开播的直播间。

单列瀑布流和双列瀑布流有什么区别?

单列瀑布流一次预览一个直播间画面，而双列瀑布流则同时预览两个直播间画面。您可以根据自己的设计需求选择合适的布局。

预览直播画面是否收费?

是的，预览直播间画面时长会计入观众的音视频时长，这部分是需要付费的。详细的 计费说明 可以参考文档中的相关内容。

我使用了自定义数据源 (LiveListDataSource)，但列表不显示/不刷新怎么办?

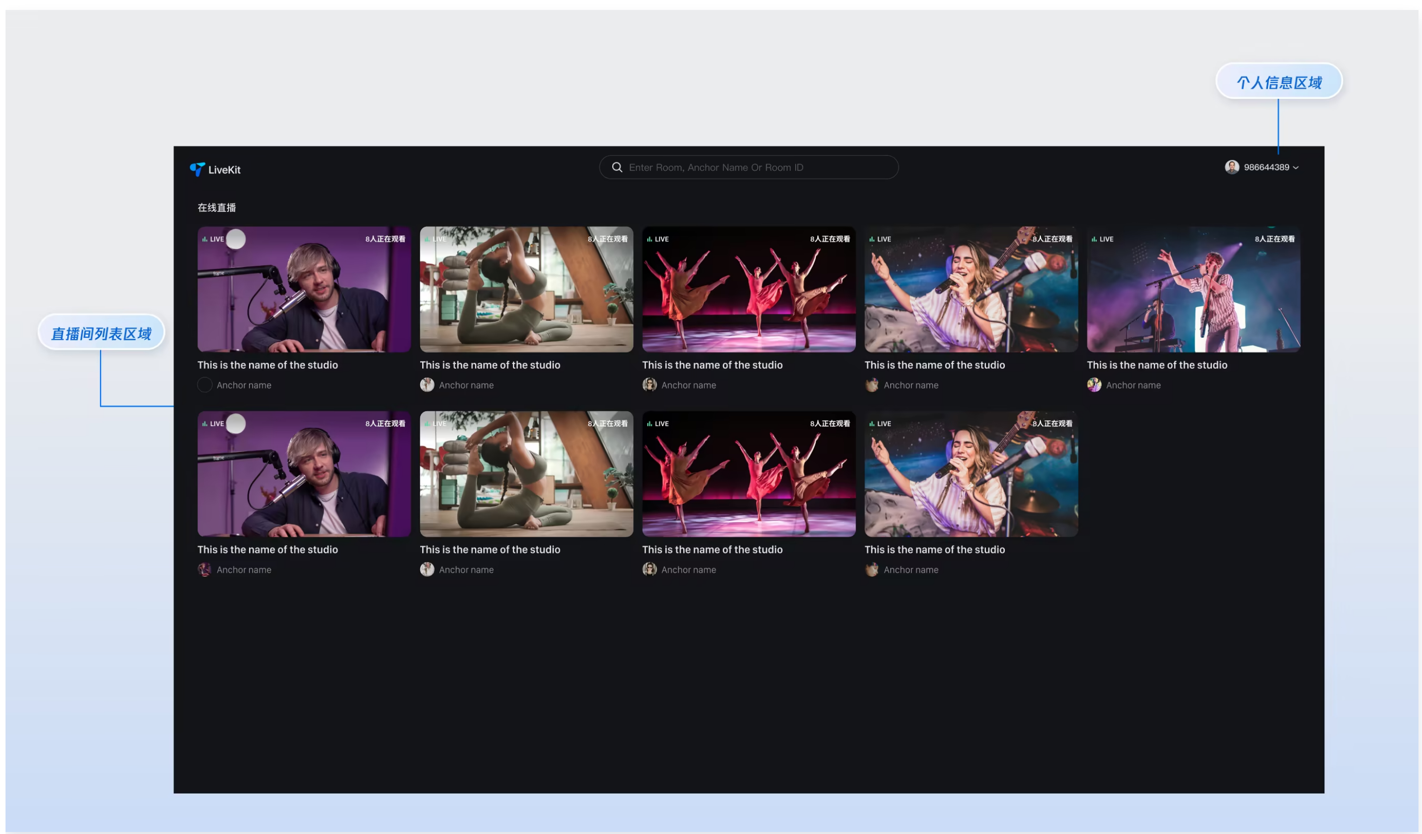
请确保您正确实现了 `LiveListDataSource` 接口。重点检查以下几点：
- 检查 `fetchLiveList` 方法是否被正确调用。
- 确保在获取数据后（无论成功或失败）都调用了 `callback.onSuccess` 或 `callback.onFailure` 。
- 检查您的业务后台接口是否返回了正确的数据格式。

直播列表（Web Vue3）

最近更新时间：2026-04-20 17:34:02

概述

本文对 TUILiveKit Demo 中的**直播列表页面**进行了详细的介绍，您可以在已有项目中直接参考本文档集成我们开发好的直播列表页面，也可以根据您的需求按照文档中的内容对页面的样式，布局以及功能项进行深度的定制。



快速体验

您可以在 在线体验 中使用 在线开播网站 快速开启一场直播，同时您也可以使用 在线观看网站 来查看账户下直播列表。

快速接入

步骤1：环境配置及开通服务

在进行快速接入之前，请参考 准备工作 集成组件并实现登录。

步骤2：安装依赖

您可以选择以下任一方式安装依赖：

npm

```
npm install tuikit-atomicx-vue3 @tencentcloud/uikit-base-component-vue3
--save
```

pnpm

```
pnpm add tuikit-atomicx-vue3 @tencentcloud/uikit-base-component-vue3
```

yarn

```
yarn add tuikit-atomicx-vue3 @tencentcloud/uikit-base-component-vue3
```

步骤3: 直播列表页面接入

在您的项目下创建 **live-list.vue** 文件，可直接复制如下代码至您的项目中集成**直播列表页面**。

> **注意：**
> 您可以直接复制如下代码至您的工程中集成示例工程，也可以访问 直播列表 地址，查看更加详细的源码内容。

```
<template>
  <UIKitProvider theme="dark">
    <div class="container">
      <header class="header">
        <h1 class="title">在线直播</h1>
      </header>
      <main class="main">
        <LiveList />
      </main>
    </div>
  </UIKitProvider>
</template>

<script setup lang="ts">
import { onMounted } from 'vue';
import { LiveList, useLoginState } from 'tuikit-atomicx-vue3';
```

```
import { UIKitProvider } from '@tencentcloud/uikit-base-component-vue3';

const { login } = useLoginState();

async function initLogin() {
  try {
    await login({
      sdkAppId: 0,          // SDKAppID, 可以参考步骤 1 获取
      userId: '',           // UserID, 可以参考步骤 1 获取
      userSig: '',          // userSig, 可以参考步骤 1 获取
    });
  } catch (error) {
    console.error('登录失败:', error);
  }
}

onMounted(async () => {
  await initLogin();
});
</script>

<style scoped>:global(*),:global(::after),:global(::before){box-
sizing:border-box;margin:0}.container{display:flex;flex-
direction:column;height:100vh;width:100vw;background:var(--bg-color-
default)}.header{display:flex;align-items:center;flex-
shrink:0}.title{margin:0;font-size:16px;font-weight:600;color:var(--
text-color-primary);letter-
spacing:-.5px}.main{flex:1;padding:24px;overflow-y:auto;min-height:0}
</style>
```

⚠ **注意：**
若您需要实现直播列表到进入直播间的跳转逻辑，可参考本文中 指定直播间 部分内容。

启动项目

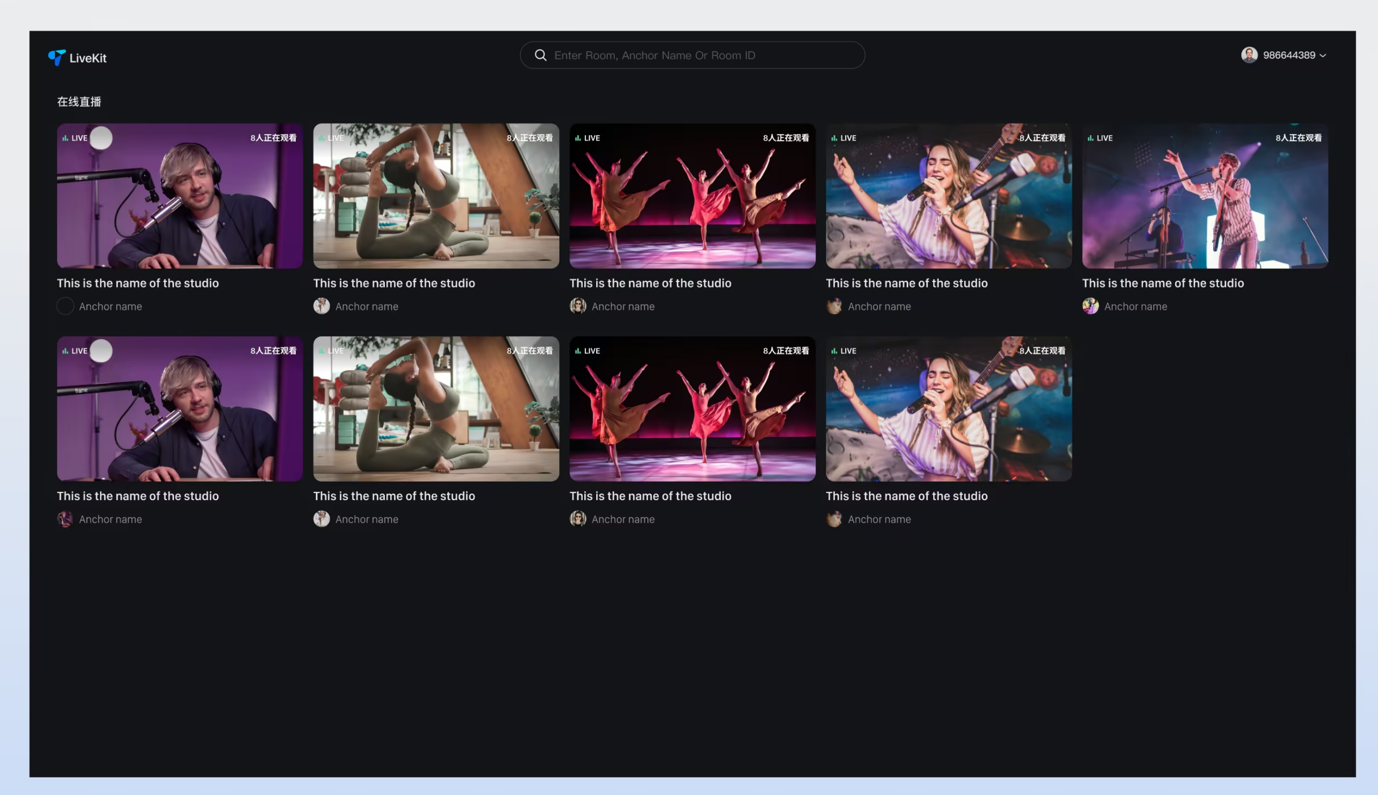
开启您的直播列表页面

通过下列命令启动成功后，通过访问本地地址打开直播列表页面。

版权所有：腾讯云计算（北京）有限责任公司

> **说明：**
> 上述命令执行成功后，请在浏览器地址栏输入本地访问地址（您可以根据项目参考，例如 `http://localhost:5173/live-list`，具体端口号可能因您的项目配置而有所不同），即可看到直播列表页面。

```
npm run dev
```



指定直播间

由于涉及到直播列表(或首页)到观看页面的跳转逻辑，您需要配置 Vue Router 来实现页面跳转 。您可以在项目 `src` 目录下新建 `router` 文件夹，并创建 `index.ts` 文件。然后，在您的主文件（例如 **main.ts** 或 **index.ts**）中引入并使用路由。可参见 GitHub 代码示例 。

步骤1：配置页面路由

请配置您的 Vue Router，将 `live-list.vue` 映射到直播列表页面。

> **注意：**
> - 如果您的业务项目中已有 `src/router/index.ts` 文件，请将以下内容合并到已有代码中。
> - 如果尚未配置路由，请新建 `src/router/index.ts` 文件并添加以下配置。

```
import { createRouter, createWebHistory } from 'vue-router';

const routes = [
  {
    // 直播列表页面，注意以下代码路径为您项目实际路径
    path: '/live-list',
    component: () => import('../live-list.vue'),
  },
  // 若您需要实现通过点击直播列表直播间封面到对应直播间进行观看，则需要参考如下示例配置观看页面的路由
  {
    // 路由跳转至观看页面，注意以下代码路径为您项目实际路径
    path: '/live-player',
    component: () => import('../live-player.vue'),
  },
];

const router = createRouter({
  history: createWebHistory(),
  routes,
});
export default router;
```

步骤2：配置 src/main.ts 文件

将路由和多语言配置合并到您项目中的入口文件 src/main.ts 中：

```
// src/main.ts
import { createApp } from 'vue';
import App from './App.vue';
import router from './router';

const app = createApp(App);
app.use(router);
app.mount('#app');
```

步骤3：配置 live-list 文件

版权所有：腾讯云计算（北京）有限责任公司

您需要将以下内容合并到上文中的 步骤3 内容中，进行直播列表到观看页面的跳转。

```
// live-list.vue 可增量添加至您的代码中
<template>
  <UIKitProvider>
     // .....省略部分参考快速接入步骤 3 部分
     <LiveList @live-room-click="handleLiveRoomClick" />
  </UIKitProvider>
</template>

<script setup lang="ts">
import { useRouter, useRoute } from 'vue-router';

const router = useRouter();
const route = useRoute();

function handleLiveRoomClick(liveInfo: LiveInfo) {
  if (liveInfo?.liveId) {
    router.push({ path: '/live-player', query: { ...route.query, liveId:
liveInfo.liveId } });
  }
}

</script>
```

自由定制

根据上述功能展示图，我们也支持您根据项目需求对直播列表页面进行 UI 定制。主要可供定制的能力如下表所示。

类别	功能	描述
直播间列表	自定义直播间列表区域展示	支持： • 直播间信息文字 展示/隐藏、UI 进行定制。 • 直播间一行/一列展示固定数量定制。
个人信息	自定义个人信息展示	支持： • 展示/隐藏个人信息。 • 个人信息字体、颜色 UI 自定义设置。

颜色主题及语言

通过配置 `App.vue` 中 `UIKitProvider` 的入参，修改主题及语言的默认值。

UIKitProvider 参数	可选值	默认值
theme	"light" | "dark"	"light"
language	"zh-CN" | "en-US"	–

```
<UIKitProvider theme="light">
  <router-view />
</UIKitProvider>

<script setup lang="ts">
import { UIKitProvider } from '@tencentcloud/uikit-base-component-vue3';
```

按钮 Button 和 图标 Icon

您可以对按钮、图标等控件进行自定义。以 `live-list.vue` 为例，对 `Button` 、 `Icon` 控件执行增删改等操作都可以满足您的自定义需求。除布局调整外，我们还支持对颜色、字体、圆角以及输入框、弹框等内容进行全面修改，充分满足您的 UI 定制需求。

类别	功能	描述
素材管理	自定义素材管理区域展示	支持调整 Icon 的大小、颜色或对 Icon 进行替换。
直播工具	自定义直播工具信息展示	支持调整 Icon 的大小、颜色或对 Icon 进行替换。
在线观众	自定义观众信息展示	支持： • 展示/隐藏观众等级。 • 观众信息字体、颜色 UI 自定义设置。 • 替换为您需要的 Icon 风格。
消息列表	自定义消息弹幕区域展示	支持： • 展示/隐藏聊天输入区域。 • 支持 UI 定制聊天气泡风格、定制观众等级等内容。

设置昵称及头像

如果您需要自定义昵称或头像，可以在 步骤 3 中的示例代码中通过 setSelfInfo 进行设置，具体示例如下：

```
import { useLoginState } from 'tuikit-atomicx-vue3';
```

```
const { setSelfInfo } = useLoginState();

// 设置用户个人信息
await setSelfInfo({
  userName: '张三',
  avatarUrl: 'https://example.com/avatar.png',
});
```

下一步

恭喜您，现在您已经成功集成了**直播列表页面**。接下来，您可以实现**主播开播页**、**观众观看页**和 **UI 自定义**等内容，可参考下表：

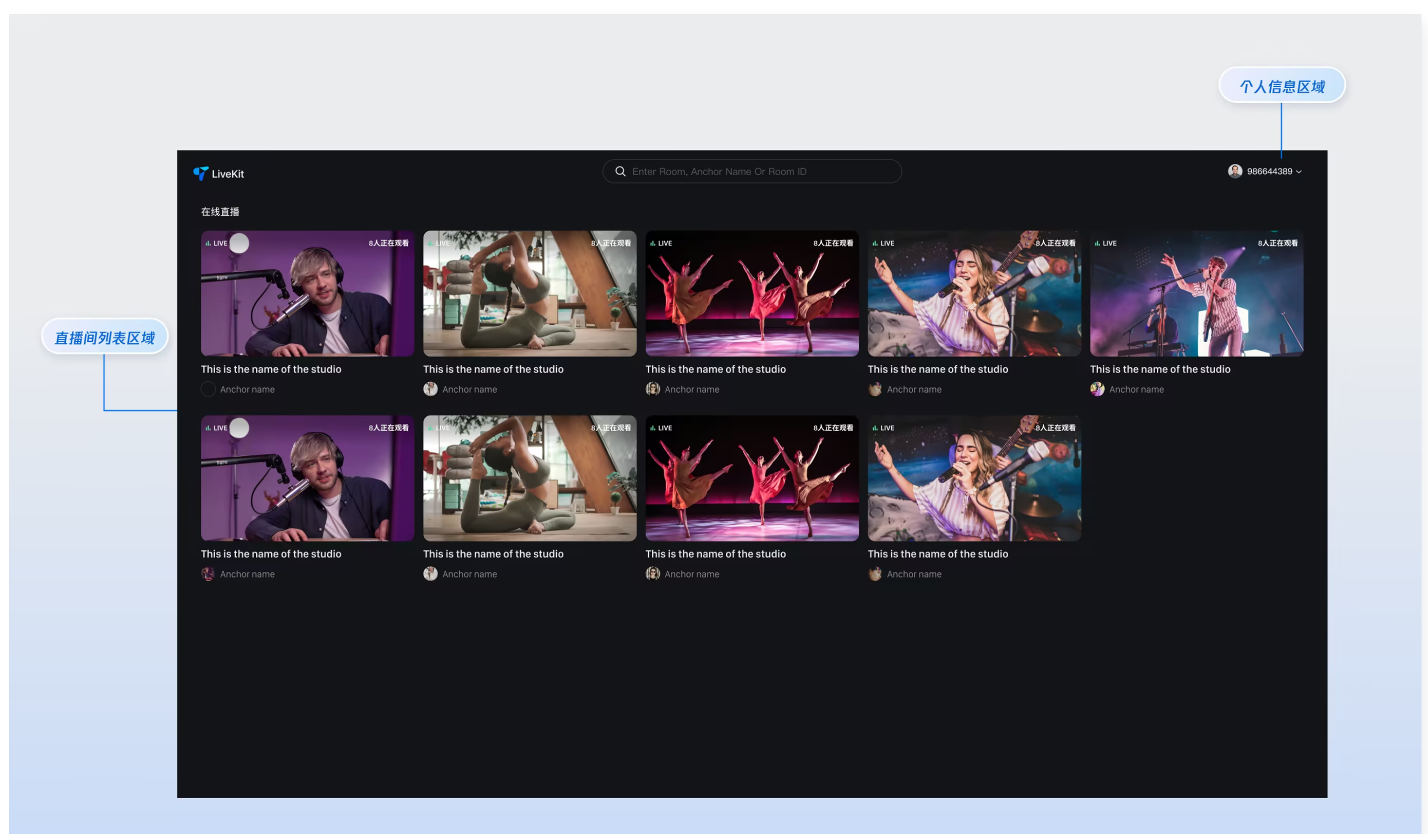
功能	描述	集成指引
观众观看	实现观众进入主播的直播间后观看直播，实现观众连麦 、直播间信息、在线观众、弹幕显示等功能。	观众观看（Web Vue3）
主播开播	主播在当前页面开始直播及相关设置。	主播开播（Web Vue3）
UI 自定义	更详细的 UI 组件自定义指引。	UI 自定义
直播管理系统	运营平台，支持直播间管控。	直播管理系统（Vue3）

直播列表（Web React）

最近更新时间：2026-04-20 17:34:02

功能概览

本文将详细介绍 TUILiveKit React Demo 中的直播列表页面，引导您在自己的项目中集成 React 直播列表页面，并对页面的样式、功能进行定制。



快速接入

步骤1：环境配置及开通服务

在进行快速接入之前，请参考 准备工作（Web React） 集成组件并实现登录。

步骤2：安装依赖

您可以选择以下任一方式安装依赖：

npm

```
npm install tuikit-atomicx-react @tencentcloud/uikit-base-component-react --save
```

版权所有：腾讯云计算（北京）有限责任公司

```
npm install sass --save-dev
```

pnpm

```
pnpm add tuikit-atomicx-react @tencentcloud/uikit-base-component-react
pnpm add sass --dev
```

yarn

```
yarn add tuikit-atomicx-react @tencentcloud/uikit-base-component-react
yarn add sass --dev
```

步骤3：接入直播列表页面

在您的项目下创建 `LiveListView.tsx` 和 `LiveListView.module.scss` 文件，可直接复制如下代码至您的项目中集成**直播列表页面**。

> **注意：**
> 您可以直接复制如下代码至您的代码工程中集成，也可以访问 观众观看 开源代码，查看完整的源码内容。

LiveListView.tsx

```
import React, { useEffect, useCallback } from 'react';
import { useNavigate, useSearchParams } from 'react-router-dom';
import { LiveList, useLoginState } from 'tuikit-atomicx-react';
import type { LiveInfo } from 'tuikit-atomicx-react';
import { UIKitProvider, MessageBox } from '@tencentcloud/uikit-base-
component-react';
import styles from './LiveListView.module.scss';

const LiveListView: React.FC = () => {
  const navigate = useNavigate();
  const [searchParams] = useSearchParams();
  const { loginUserInfo, login, setSelfInfo } = useLoginState();

  const handleLiveRoomClick = (liveInfo: LiveInfo) => {
```

```
    if (loginUserInfo?.userId === liveInfo.liveOwner?.userId) {
      MessageBox.alert({
        title: '警告',
        content: '无法查看自己的直播间',
      });
      return;
    }

    if (liveInfo?.liveId) {
      // 可以在此处添加业务处理，跳转到观看页面
      // // 从【直播列表页面】跳转【观众观看页面】示例
      // const newParams = new URLSearchParams(searchParams);
      // newParams.set('liveId', liveInfo.liveId);
      // // 直播间 ID 放入 URL 参数，观众观看页面可根据 URL 参数进入对应直播间
      // navigate(`/live-player?${newParams.toString()}`);
    }
  };

  const initLogin = useCallback(async () => {
    try {
      await login({
        SDKAppID: 0,     // 请替换为您的 SDKAppID（服务开通时获取）
        userID: '',      // 请替换为您的用户 ID
        userSig: '',     // 请替换为您的用户签名（详细获取方式请参阅【步骤1：环境
配置及开通服务】文档）
      });
      await setSelfInfo({
        userName: '',    // 用户昵称，会显示在成员列表、聊天消息中。不设置昵称时，
将显示用户 ID
        avatarUrl: '',   // 用户头像，必须为完整的 URL 图片地址，比如：
https://your.domain.com/avatar-default.png
      });
    } catch (error) {
      console.error('登录失败:', error);
    }
  }, [login, setSelfInfo]);

  useEffect(() => {
    async function init() {
      await initLogin();
```

版权所有：腾讯云计算（北京）有限责任公司

```
    }

    if (!loginUserInfo?.userId) {
      init();
    } else {
      console.log('[LiveList]用户已登录:', loginUserInfo.userId);
    }
  }, [initLogin, loginUserInfo?.userId]);

  return (
    <UIKitProvider theme="dark">
      <div className={styles.liveList}>
        {/* 直播列表组件：设置为显示 5 列、注册直播间点击事件处理函数 */}
        <LiveList columnCount={5} onLiveRoomClick={handleLiveRoomClick}
/>
      </div>
    </UIKitProvider>
  );
};

export default LiveListView;
```

LiveListView.module.scss

```
.liveList {
  display: flex;
  flex-direction: column;
  width: 100%;
  height: 100%;
  background-color: var(--uikit-bg-color-topbar);
  color: var(--uikit-text-color-primary);
  overflow: auto;
  box-sizing: border-box;
}
```

步骤4：页面路由配置

如果您是从直播列表页面跳转到直播观看页面，您需要配置 React Router 页面路由。在项目中新建或修改 `src/router/index.tsx` 文件。然后，在您的主文件（例如 `main.tsx` 或 `App.tsx` ）中引入并使用路由。可参见 GitHub 代码示例 。

1. **页面路由组件**

   在页面路由组件中，配置直播列表页面。

```
// src/router/index.tsx
import { createHashRouter } from 'react-router-dom';
import { LiveListView } from '../views/LiveList';
import { LivePlayerView } from '../views/LivePlayer';

// 路由保护组件
const ProtectedRoute = ({ children }: { children: React.ReactNode; })
=> {
  return (
    <>{children}</>
  );
};

const routes = [
  // 直播列表
  {
    path: '/live-list',
    element: <LiveListView />,
  },
  // // 若您需要实现点击直播列表直播间封面，跳转到直播观看页面功能，则需要配置观看
页面路由
  // // 观看页面详细接入，请参阅【观众观看 -> 观众观看（Web React）】
  // {
  //   path: '/live-player',
  //   element: <LivePlayerView />,
  // }
];

export const router = createHashRouter(
  routes.map(route => ({
    ...route,
    element: <ProtectedRoute>{route.element}</ProtectedRoute>,
  }))
```

```
);
```

2. **配置 App.tsx 文件**

在 `App.tsx` 文件中，使用上面的**路由组件** `src/router/index.tsx` 。

```
// src/App.tsx
import { RouterProvider } from 'react-router-dom'
import { router } from './router'
import './App.css'

function App() {
  return (
    <RouterProvider router={router} />
  )
}

export default App
```

步骤5：启动项目

进入项目根目录，执行以下命令启动项目，启动后即可访问直播列表页面。

```
npm run dev
```

首次启动，可能因没有直播间而显示空列表，可以访问我们提供的 在线开播网站，使用您的 SDKAppID 和用户账号登录后，开启一场直播后，再来刷新一次直播列表页面，就能在直播列表页面看到直播间数据。

> **注意：**
> 为了正常体验直播功能，**直播开播和直播观看，必须使用不同的用户 ID 登录**，否则会出现登录账号互踢，无法体验。

自由定制

颜色主题及语言

你可以使用 UIKitProvider 组件，修改默认的主题和语言。

UIKitProvider 参数	可选值	默认值

版权所有：腾讯云计算（北京）有限责任公司

theme	"light"	"dark"	"light"
language	"zh-CN"	"en-US"	-

1. **在 App.tsx 中全局设置**

   在 App.tsx 中，以 UIKitProvider 作为根元素。

```
import { RouterProvider } from 'react-router-dom'
import { UIKitProvider } from '@tencentcloud/uikit-base-component-
react'
import { router } from './router'
import './App.css'

function App() {
  return (
    <UIKitProvider theme="dark" language='zh-CN'>
      <RouterProvider router={router} />
    </UIKitProvider>
  );
}

export default App
```

2. **在单个页面或组件中设置**

   在 React 组件中，以 UIKitProvider 作为根节点元素。以下示例代码截取自 “快速接入 步骤 3” 的代码片段。

```
// ... 省略其它依赖引入
import { UIKitProvider } from '@tencentcloud/uikit-base-component-
react'

const LiveListView: React.FC = () => {
  // ... 省略其它代码
  return (
    <UIKitProvider theme="dark">
      <div className={styles.liveList}>
        <LiveList columnCount={5} onLiveRoomClick=
{handleLiveRoomClick} />
      </div>
```

```
      </UIKitProvider>
    );
  };

  export default LiveListView;
```

设置昵称和头像

上文中，“快速接入：步骤 3” 已包含昵称和头像设置代码，如下面代码所示，设置的头像和昵称，将显示在自己和其它直播间成员的成员列表、聊天消息中。如果不主动设置昵称和头像，昵称将显示为登录时的用户 ID、头像将显示为默认头像。

```
  const initLogin = useCallback(async () => {
    try {
      await login({
        SDKAppID: 0,      // SDKAppID 服务开通时获取的 SDKAppID
        userID: '',       // UserID 用户 ID
        userSig: '',      // userSig 用户签名，详细获取方式请参阅【准备工作】文档
      });
      await setSelfInfo({
        userName: '',     // 用户昵称，会显示在成员列表、聊天消息中。不设置昵称时，
将显示用户 ID
        avatarUrl: '',    // 用户头像，必须为完整的 URL 图片地址，比如：
https://your.domain.com/avatar-default.png
      });
    } catch (error) {
      console.error('登录失败:', error);
    }
  }, [login, setSelfInfo]);
```

下一步

恭喜您，现在您已经成功集成了**直播列表页面**。接下来，您可以继续进一步接入 **观众观看页**、**UI 自定义**等内容：

功能	描述	集成指引
观众观看	实现观众进入主播的直播间后观看直播，实现观众连麦 、直播间信息、在线观众、弹幕显示等功能。	观众观看（Web React）
UI 自定义	更详细的 UI 组件自定义指引。	UI 自定义

版权所有：腾讯云计算（北京）有限责任公司

直播管理系统	运营平台，支持直播间管控。	直播管理系统（React）

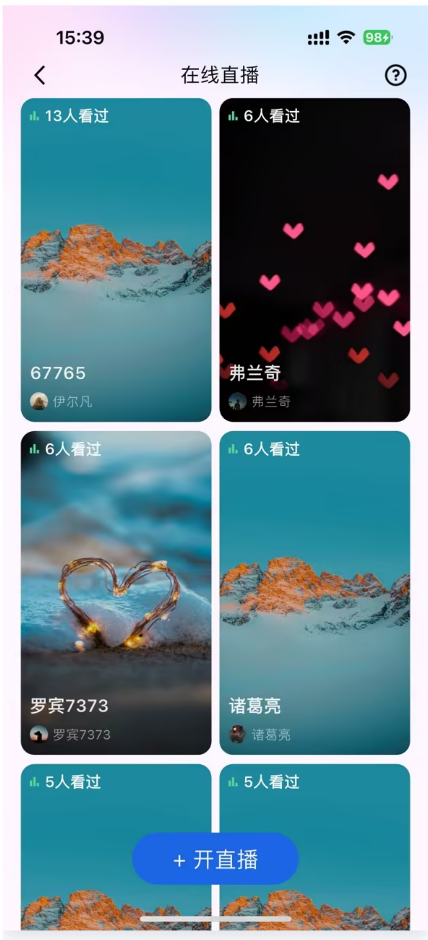
版权所有：腾讯云计算（北京）有限责任公司

直播列表（Flutter）

最近更新时间：2026-06-22 10:22:18

功能预览

本文详细介绍了 TUILiveKit 中的**直播列表页面**功能。您可以直接在现有项目中参考本文档，集成我们已开发完成的直播列表页面。该页面支持下拉刷新、上拉加载以及点击加入直播间等功能。



快速接入

步骤 1. 开通服务

参考 开通服务 文档开通「体验版」或「大规模直播版」套餐。

步骤 2. 代码集成

参考 准备工作 接入 `TUILiveKit` 。

步骤 3. 添加直播列表页面

在您需要进入直播列表页面的调用入口（具体由您的业务决定），执行如下操作，拉起直播列表页面或将直播列表页面集成到自己的 Widget 树中：

直接跳转

```
// 跳转到直播列表页面
Navigator.push(context, MaterialPageRoute(
  builder: (context) {
     return LiveListWidget();
}));
```

集成到 Widget 树

```
// --- 根据您的Widget树结构，选择以下一种方式集成 ---

// [选项一] 作为唯一子Widget（单子树）
// 适用于Container、Padding等通常只包含一个子Widget的容器
Container(
     child: LiveListWidget() // 在此处集成直播列表页
)

// [选项二] 作为多个子Widget之一（多子树）
// 适用于Column、Row、Stack等可以包含多个子Widget的布局
Stack(
     children: [
       YourOtherWidget(), // 您的其他子Widget
       LiveListWidget(), // 在此处集成直播列表页
       YourOtherWidget(), // 您的其他子Widget
])
```

下一步

恭喜您，现在您已经成功集成了 **直播列表** 功能。接下来，您可以根据您的业务场景实现**主播开播、观众观看**等功能**视频直播场景：**

功能	描述	集成指引

主播开播	实现主播开播全流程功能，包括开播前的准备和开播后的各种互动。	主播开播
观众观看	实现观众进入主播的直播间后观看直播，实现观众连麦 、直播间信息、在线观众、弹幕显示等功能。	观众观看

语聊房场景：

功能	描述	集成指引
主播开播	实现主播开播全流程功能，包括开播前的准备和开播后的各种互动。	主播开播
观众观看	实现观众进入主播的直播间后观看直播，实现观众连麦 、直播间信息、在线观众、弹幕显示等功能。	观众观看

常见问题

集成直播列表功能后页面没有任何直播怎么办？

如果您看到空白页面，需要检查您是否已完成 登录步骤 。为了测试该功能，您可以使用两台设备：一台设备用于开播，另一台设备在直播列表页面，就能拉取到已开播的直播间。

直播列表（uni-app 客户端）

最近更新时间：2026-02-26 15:52:27

本文将指导您如何快速集成直播列表页，体验查看直播列表以及预览直播间信息功能。

功能预览

直播列表页接入

步骤1：集成核心文件

已集成核心文件请忽略，已经按照 代码集成指引 将 TUILiveKit 的 `pages` 、 `uni_modules` 等核心文件拷贝到了您的项目中。

步骤2：完成登录

已登录请忽略该步骤，您的应用已经调用了 登录接口 并成功登录。这是使用所有功能的基础

版权所有：腾讯云计算（北京）有限责任公司

步骤3：制作并使用自定义基座

已制作自定义基座请忽略，由于直播功能依赖原生插件，您必须在 HBuilderX 中为您的项目制作并使用自定义调试基座来运行。

步骤4：注册直播列表页面

现在，您需要在 `pages.json` 文件中告诉您的应用，这个新页面是存在的。
打开您项目根目录下的 `pages.json` 文件，在 `"pages"` 数组中，添加以下对象来注册直播列表页。

```
{
    "pages": [
        // ... 您项目已有的其他页面配置
        {
           "path": "pages/scenes/live/livelist/index",
            "style": {
                "navigationBarTitleText": "在线直播"
            }
        },
    ]
    // ... 其他配置
}
```

步骤5：跳转直播列表页面

在您需要展示直播列表的地方（具体由您的业务决定，在其点击事件里执行），调用 `uni.navigateTo` 或 `uni.redirectTo` 方法，即可进入直播列表页面

```
// 示例：在一个按钮的点击事件中
function startBroadcast() {
  uni.redirectTo({
    url: '/pages/scenes/live/livelist/index', // URL 对应 pages.json 中配置的 path
    });
}
```

定制您的 UI

根据上述功能展示图，我们也支持您根据项目需求对直播列表页面进行 UI 定制能力。主要可供定制的能力如下表所示。

类别	功能	描述
直播间列表	自定义直播间列表区域展示	支持： • 直播间信息文字调整大小、字体、颜色 • 修改每行展示直播间数量

直播间信息文字调整大小、字体、颜色

uniapp 官方对 nvue 页面限制字体的大小和颜色只能在 `text` 标签上设置，对于您想修改的文字，直接修改其 `css` 样式即可，下方示例以 “直播间名称” 为例

```
<text class="live-title" :lines="1">{{ live.liveName }}</text>
.live-title {
  font-size: 28rpx;
  font-weight: 500;
  color: #fff;
  margin-bottom: 8rpx;
  lines: 1;
  max-width: 400rpx;
  overflow: hidden;
  text-overflow: ellipsis;
}
```

修改其 `css` 属性：

```
<text class="live-title" :lines="1">{{ live.liveName }}</text>
.live-title {
  font-size: 35rpx;
  font-weight: 500;
  color: pink;
  margin-bottom: 8rpx;
  lines: 1;
  max-width: 400rpx;
  overflow: hidden;
  text-overflow: ellipsis;
}
```

最终效果如下：

修改每行展示直播间数量

通过设置每行展示的元素数量来修改每行展示的直播间数量：

```
<cell v-for="(row, rowIndex) in groupedLiveList"
:key="`row-${rowIndex}-${row[0]?.liveId || 'empty'}`">
    ......
</cell>
const groupedLiveList = computed(() => {
  const groups = [];
  for (let i = 0; i < liveList.value.length; i += 2) {
    const row = liveList.value.slice(i, i + 2); // 将直播列表分组，每行两个元
素
    groups.push(row);
  }
  return groups;
});
```

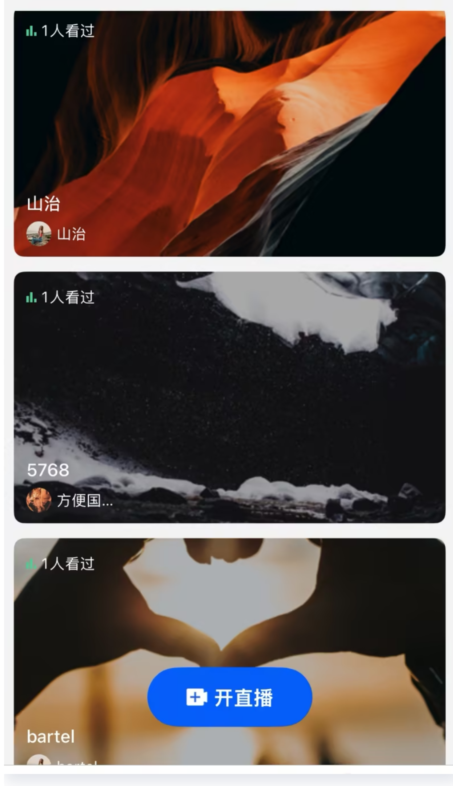
修改每行展示的直播间数量，示例为每行展示一个：

```
<cell v-for="(row, rowIndex) in groupedLiveList"
:key="`row-${rowIndex}-${row[0]?.liveId || 'empty'}`">
    ......
</cell>

const groupedLiveList = computed(() => {
  const groups = [];
  for (let i = 0; i < liveList.value.length; i += 1) {
```

版权所有：腾讯云计算（北京）有限责任公司

```
    const row = liveList.value.slice(i, i + 1); // 将直播列表分组，每行一个元素
    groups.push(row);
  }
  return groups;
});
```

最终效果：



其他参考文档

- TUILiveKit uni-app 主播开播页
- TUILiveKit uni-app 观众观看页

版权所有：腾讯云计算（北京）有限责任公司

观众列表组件

观众列表组件 (Android Java)

最近更新时间：2025-12-10 16:28:31

观众列表组件（AudienceListView）封装了直播间在线人数展示和观众列表弹窗功能。您可以轻松地将其集成到您的 UI 中，为您的直播间添加观众互动的基础能力。

效果展示

快速开始

步骤1：开通服务

参考 开通服务 文档开通**体验版**或**大规模直播版**套餐。

步骤2：代码集成

参考 准备工作 接入 `TUILiveKit` 。

步骤3：集成观众列表组件

您可以通过代码或 XML 两种方式创建 `AudienceListView` 。

- 在代码中创建：

```
  AudienceListView audienceListView = new
AudienceListView(getContext());
```

- 在 XML 布局文件中定义：

```
<com.trtc.uikit.livekit.component.audiencelist.AudienceListView
    android:id="@+id/audience_list_view"
    android:layout_width="135dp"
    android:layout_height="24dp"
    android:layout_gravity="end" />
```

步骤4：初始化组件

在成功进入直播间后，调用 `AudienceListView` 的 `init` 方法，以绑定数据和事件。

> **说明：**
> 此操作必须在进房成功的回调中执行。

```
audienceListView.init(roomInfo);
```

自定义组件

观众列表组件提供了直播间内的观众点击回调接口，该回调支持您根据自己的业务需求实现直播间内的观众管理。

组件接口

接口	参数	说明
onUserItemClick	TUIRoomDefine.UserInfo	观众列表点击回调

实现观众管理

默认情况下，点击观众列表中的用户会弹出管理弹框。您可以通过实现 `onUserItemClick` 回调，并通过 **LiveAudienceStore** 来执行具体操作，例如实现将用户踢出直播间的功能：

```
import android.os.Bundle
import androidx.appcompat.app.AlertDialog
```

```
import androidx.appcompat.app.AppCompatActivity
import com.tencent.cloud.tuikit.engine.extension.TUILiveListManager
import com.tencent.cloud.tuikit.engine.room.TUIRoomDefine
import com.trtc.uikit.livekit.R
import com.trtc.uikit.livekit.component.audiencelist.AudienceListView
import io.trtc.tuikit.atomicxcore.api.live.LiveAudienceStore

class YourAnchorActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_your_anchor)
        // 1.定义观众列表组件
        var audienceListView = AudienceListView(this@YourAnchorActivity)
        var liveInfo = TUILiveListManager.LiveInfo().apply {
            roomId = "yourLiveId"
        }
        // 2.初始化组件，传入直播间信息
        audienceListView.init(liveInfo)
        // 3.设置监听器，监听用户点击事件
        audienceListView.setOnUserItemClickListener(object :
            AudienceListView.OnUserItemClickListener {
            override fun onUserItemClick(userInfo:
TUIRoomDefine.UserInfo) {
                // 4.处理用户点击事件，例如踢出直播间
                kickUserOutOfRoom(liveInfo.roomId, userInfo)
            }
        })
    }

    private fun kickUserOutOfRoom(liveId: String, userInfo:
TUIRoomDefine.UserInfo) {
        AlertDialog.Builder(this@YourAnchorActivity)
            .setTitle("观众管理")
            .setMessage("您是否要将${userInfo.userName}踢出直播间")
            .setPositiveButton("确定") { dialog, which ->

LiveAudienceStore.create(liveId).kickUserOutOfRoom(userInfo.userId,
null)
                dialog.dismiss()
```

版权所有：腾讯云计算（北京）有限责任公司

```
            }
            .setNegativeButton("取消") { dialog, which ->
                dialog.dismiss()
            }
            .create()
            .show()
    }
}
```

观众列表组件 (iOS UIKit)

最近更新时间：2025-12-10 16:28:31

观众列表组件（AudienceListView）封装了直播间在线人数展示和观众列表弹窗功能。您可以轻松地将其集成到您的 UI 中，为您的直播间添加观众互动的基础能力。

效果展示

快速开始

步骤1：开通服务

参考 开通服务 文档开通**体验版**或**大规模直播版**套餐。

步骤2：代码集成

参考 准备工作 完成 TUILiveKit 组件接入。

步骤3：集成观众列表组件

以主播开播的场景为例，将 AudienceListView 添加到您的主播视图控制器中，示例代码如下：

版权所有：腾讯云计算（北京）有限责任公司

```
import TUILiveKit

class YourAnchorViewController: UIViewController {

    // 1. 定义观众列表组件
    private let audienceListView = AudienceListView()
    private let liveId: String

    // ... 其他代码 ...

    public override func viewDidLoad() {
        super.viewDidLoad()

        // 2. 将组件添加到您的视图中并设置布局
        view.addSubview(audienceListView)
        audienceListView.snp.remakeConstraints { make in
            make.centerY.equalTo(closeButton)
            make.trailing.equalTo(closeButton.snp.leading).offset(-8)

make.leading.greaterThanOrEqualTo(liveInfoView.snp.trailing).offset(20)
        }

        // 3. 关联直播房间 id
        audienceListView.initialize(liveId: liveId)
    }
}
```

自定义组件

观众列表组件提供了直播间内的观众点击回调接口，该回调支持您根据自己的业务需求实现直播间内的观众管理。

组件接口

接口	参数	说明
onUserManageButtonClicked	LiveUserInfo	观众列表点击回调

```
public struct LiveUserInfo {

    public var userId: String          // 观众用户 id
```

```
    public var userName: String          // 观众用户昵称
    public var avatarURL: String         // 观众用户头像

    public init()
}
```

实现观众管理

默认情况下，点击观众列表中的用户会弹出管理弹框。您可以通过实现 `onUserManageButtonClicked` 闭包，并通过 **LiveAudienceStore** 来执行具体操作，例如实现将用户踢出直播间的功能：

```
import UIKit
import TUILiveKit
import AtomicXCore

class YourAnchorViewController: UIViewController {
    // 1. 定义观众列表组件和 Store
    private let audienceListView = AudienceListView()
    private lazy var liveAudienceStore =
LiveAudienceStore.create(liveId: self.liveId)
    private let liveId: String

    public override func viewDidLoad() {
        super.viewDidLoad()
        // ... 其他代码 ...

        // 2. 为组件绑定自定义管理事件
        audienceListView.onUserManageButtonClicked = { [weak self]
userInfo in
            self?.showUserManagementAlert(for: userInfo)
        }
    }

    // 3. 实现自定义的用户管理弹框
    private func showUserManagementAlert(for user: LiveUserInfo) {
        let alertController = UIAlertController(title: "观众管理",
message: "您希望对 \(user.userName) 做什么? ", preferredStyle:
.actionSheet)
```

```
        let kickOutAction = UIAlertAction(title: "踢出直播间", style:
.destructive) { [weak self] _ in
            self?.kickOutUser(user)
        }

        let cancelAction = UIAlertAction(title: "取消", style: .cancel)

        alertController.addAction(kickOutAction)
        alertController.addAction(cancelAction)

        present(alertController, animated: true)
    }

    // 4. 调用 Store 方法将用户踢出
    private func kickOutUser(_ user: LiveUserInfo) {
        liveAudienceStore.kickOutUser(user.userId) {
            print("Successfully kicked out \(user.userName).")
        } onError: { error, message in
            print("Failed to kick out user, error: \(message)")
        }
    }
}
```

观众列表组件 (Web Vue3)

最近更新时间：2026-04-28 10:38:56

`TUILiveKit` 观众列表组件为直播场景提供实时在线观众展示方案，帮助开发者直观掌握直播间观众构成。通过本文档，可快速集成观众列表功能，并支持深度定制以满足业务需求。

组件构成

组件名称	具体内容
观众列表组件（LiveAudienceList）	负责实时展示直播间在线观众的组件，提供**头像、昵称展示、自定义观众标识和完整列表项定制**等能力，支持通过插槽和属性进行灵活配置。

组件接入

步骤1：环境配置及开通服务

在进行快速接入之前，请参考 准备工作，完成相关环境配置及开通对应服务。

步骤2：安装依赖

```
npm
```

```
npm install tuikit-atomicx-vue3 @tencentcloud/uikit-base-component-vue3
--save
```

pnpm

```
pnpm add tuikit-atomicx-vue3 @tencentcloud/uikit-base-component-vue3
```

yarn

```
yarn add tuikit-atomicx-vue3 @tencentcloud/uikit-base-component-vue3
```

步骤3：集成观众列表组件

在项目中引入并使用观众列表组件，可直接复制如下代码示例用于展示直播间观众列表。

```
<template>
  <UIKitProvider theme="dark">
    <div class="app">
        <div class="live-audience-container">
          <LiveAudienceList class="live-audience-list"/>
      </div>
    </div>
  </UIKitProvider>
</template>

<script setup lang="ts">
import { onMounted } from 'vue';
import { UIKitProvider } from '@tencentcloud/uikit-base-component-vue3';
import { LiveAudienceList, useLoginState, useLiveListState } from
'tuikit-atomicx-vue3';

const { login } = useLoginState();
const { joinLive } = useLiveListState();

async function initLogin() {
  try {
    await login({
```

```
      sdkAppId: 0,          // SDKAppID, 可以参考步骤 1 获取
      userId: '',           // UserID, 可以参考步骤 1 获取
      userSig: '',          // userSig, 可以参考步骤 1 获取
    });
  } catch (error) {
    console.error('登录失败:', error);
  }
}

onMounted(async () => {
  await initLogin();
  await joinLive({
    liveId: '输入对应直播间 LiveId',      // 进入直播间
  });
});
</script>

<style>.live-audience-
container{display:flex;height:100%;width:300px;padding:20px}.live-
audience-list{width:100%;height:100%}</style>
```

自定义组件

观众列表组件提供了灵活的自定义能力，您可以根据业务需求选择不同的定制方式：

定制场景	推荐方式	说明
仅调整组件容器的高度、样式等	Props: `height` / `style`	最简单，几行属性配置即可实现。
在观众头像和昵称之间添加自定义标识（如 VIP 徽章、禁言标识）	Slot: `audience-mark`	通过插槽注入自定义标识元素，保留默认的列表项布局。
完全替换整个观众列表项的渲染，实现排名、自定义布局等	Slot: `audience-item`	最灵活，完全接管每个列表项的渲染逻辑。**优先级高于** `audience-mark` 。

修改前	修改后	
	自定义观众标识（audience-mark）	完全接管列表项渲染（audience-item）

自定义观众标识（audience-mark 插槽）

通过 audience-mark 插槽，您可以在观众头像和昵称之间插入自定义标识元素（例如 VIP 徽章、禁言状态等），同时保留组件默认的列表项布局。

插槽参数

插槽名	参数名	参数类型	说明
audience-mark	audience	AudienceInfo	当前观众信息对象，包含 `userId`、`userName`、`avatarUrl`、`isMessageDisabled` 等字段。

以下示例演示了如何根据观众数据动态渲染 VIP 身份徽章和禁言状态标识：

```
<template>
  <LiveAudienceList>
    <template #audience-mark="{ audience }">
      <div class="custom-mark">
        <span v-if="isVip(audience.userId)" class="vip-badge">VIP</span>
        <span v-if="audience.isMessageDisabled" class="muted-badge">□
</span>
      </div>
    </template>
  </LiveAudienceList>
</template>

<script setup lang="ts">
import { LiveAudienceList } from 'tuikit-atomicx-vue3';

const vipUserIds = ['user_vip_001', 'user_vip_002'];
const isVip = (userId: string) => vipUserIds.includes(userId);
</script>
```

版权所有：腾讯云计算（北京）有限责任公司

```
<style scoped>.custom-mark{display:flex;align-items:center;gap:4px}.vip-
badge{background:linear-
gradient(135deg,#ffd700,#ffb347);color:#fff;font-size:10px;font-
weight:600;padding:2px 6px;border-radius:4px}.muted-badge{font-
size:14px;opacity:.7}</style>
```

完全接管列表项渲染（audience-item 插槽）

使用 audience-item 插槽可获取观众的完整信息和列表索引，自由实现排名、自定义布局等效果。

插槽参数

插槽名	参数名	参数类型	说明
audience-item	index	number	当前观众在列表中的索引位置。
	audience	AudienceInfo	当前观众信息对象，包含 `userId`、`userName`、`avatarUrl` 等字段。

以下示例演示了如何利用 `index` 和 `audience` 数据实现带排名奖牌的观众列表：

```
<template>
  <LiveAudienceList>
    <template #audience-item="{ index, audience }">
      <div class="custom-audience-item">
        <span v-if="index === 0" class="medal">�</span>
        <span v-else-if="index === 1" class="medal">�</span>
        <span v-else-if="index === 2" class="medal">�</span>
        <span v-else class="rank-number">{{ index + 1 }}</span>
        <img :src="audience.avatarUrl" class="avatar" />
        <span class="name">{{ audience.userName || audience.userId }}
</span>
      </div>
    </template>
  </LiveAudienceList>
</template>

<script setup lang="ts">
import { LiveAudienceList } from 'tuikit-atomicx-vue3';
</script>
```

```
<style scoped>.custom-audience-item{display:flex;align-
items:center;gap:10px;padding:8px 12px;border-
radius:10px;background:rgba(255,255,255,0.05)}.medal{font-
size:20px}.rank-number{font-size:14px;font-
weight:600;color:rgba(255,255,255,0.5);min-width:24px;text-
align:center}.avatar{width:36px;height:36px;border-radius:50%;object-
fit:cover}.name{font-
size:14px;color:rgba(255,255,255,0.85);overflow:hidden;text-
overflow:ellipsis;white-space:nowrap}</style>
```

props 速查表

参数名	参数类型	默认值	说明
height	String	'500px'	组件高度，支持 CSS 单位（px、%、vh 等）。
style	CSSProperties	{}	自定义样式对象，用于覆盖组件默认样式。

常见问题

如何区分房主和观众?

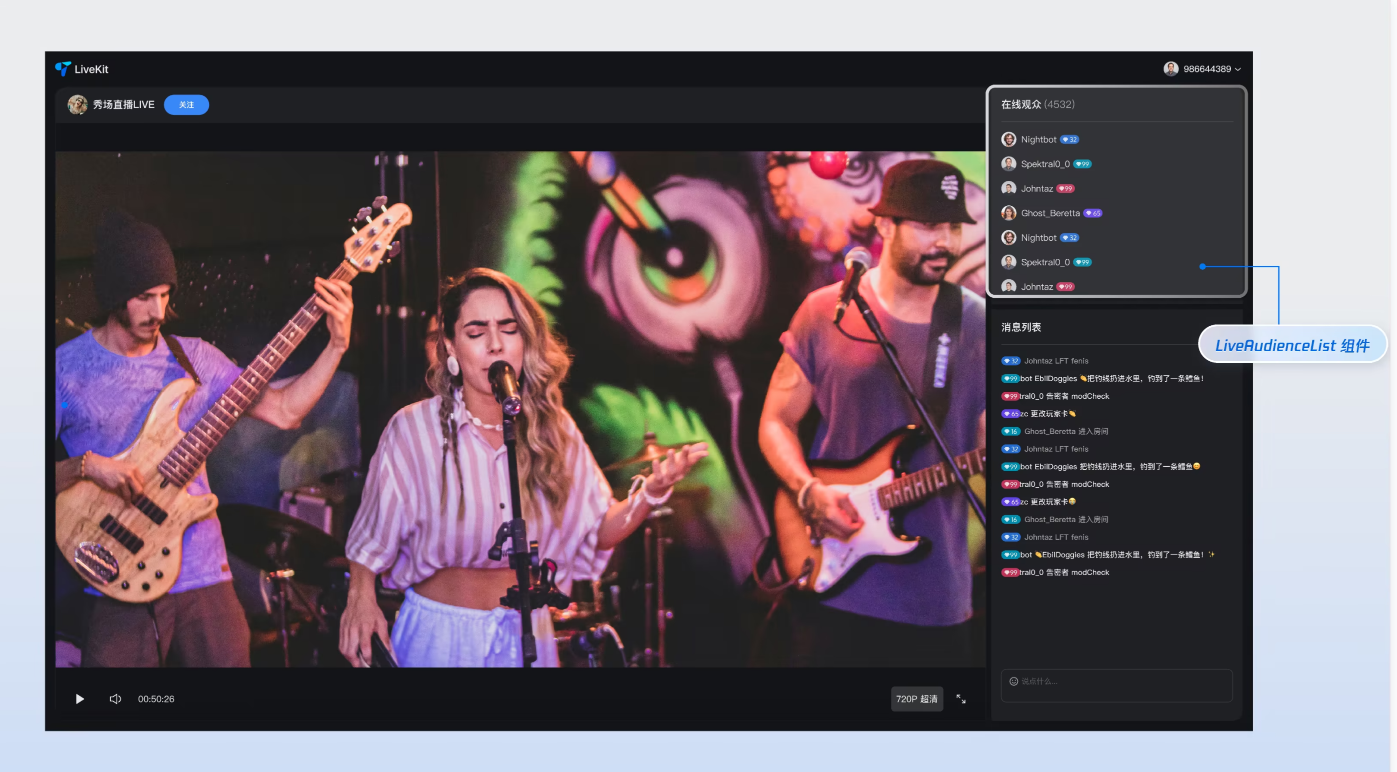
您可以通过 `useLiveListState` 获取当前直播间信息，将观众的 `userId` 与直播间 ownerId 对比来判断：

```
import { useLiveListState } from 'tuikit-atomicx-vue3';

const { currentLive } = useLiveListState();

function isLiveOwner(userId: string): boolean {
  return userId === currentLive.value?.liveOwner.userId;
}
```

观众列表组件 (Web React)

最近更新时间：2026-03-03 18:01:03

本文对**观众列表组件（LiveAudienceList）**进行了详细的介绍，您可以在已有项目中直接参考本文示例集成我们开发好的组件，也可以根据您的需求按照文档中的组件定制化部分对样式，布局进行深度的定制。



核心功能

功能分类	具体能力
实时观众展示	实时显示直播间在线观众列表，支持头像、昵称展示，提供清晰的观众信息视图，让主播能够直观了解观众构成。
响应式设计	组件支持桌面端和移动端两套 UI 方案，自动适配不同设备屏幕尺寸，提供一致的用户体验，满足多平台直播需求。
可定制化 UI	提供灵活的插槽机制，支持自定义观众标记、头像样式等元素，让您能够根据业务需求定制观众列表的展示效果，打造独特的视觉体验。

组件接入

步骤1：环境配置及开通服务

在进行快速接入之前，您需要参见 准备工作（Web React），满足相关环境配置及开通对应服务。

版权所有：腾讯云计算（北京）有限责任公司

步骤2：安装依赖

npm

```
npm install tuikit-atomicx-react @tencentcloud/uikit-base-component-
react --save
npm install sass --save-dev
```

pnpm

```
pnpm add tuikit-atomicx-react @tencentcloud/uikit-base-component-react
pnpm add sass --dev
```

yarn

```
yarn add tuikit-atomicx-react @tencentcloud/uikit-base-component-react
yarn add sass --dev
```

步骤3：集成观众列表组件

在您的项目中引入并使用**观众列表组件**，可直接复制如下代码示例至您的项目中展示直播间观众列表。

AudienceList.tsx

```
import React from "react";
import { useUIKit } from "@tencentcloud/uikit-base-component-react";
import { useLiveAudienceState, LiveAudienceList } from "tuikit-atomicx-
react";
import styles from "./AudienceList.module.scss";

const AudienceList: React.FC = () => {
  const { t } = useUIKit();
  const { audienceCount } = useLiveAudienceState();

  return (
    <div className={styles.livePlayer__audienceList}>
      <div className={styles.livePlayer__audienceListTitle}>
        <span>{t('live_player_view.audience_list_title')} </span>
```

```
          <span className={styles.livePlayer__audienceCount}>
({audienceCount})</span>
        </div>
        <div className={styles.livePlayer__audienceListContent}>
          <LiveAudienceList height="100%" />
        </div>
      </div>
  );
};

export default AudienceList;
```

AudienceList.module.scss

```
.livePlayer__audienceList {
  display: flex;
  flex-direction: column;
  flex-shrink: 0;
  height: 30%;
  padding: 8px;
  background: var(--uikit-bg-color-operate);

  .livePlayer__audienceListTitle {
    padding: 12px 0;
    border-bottom: 1px solid var(--uikit-stroke-color-primary);
    @include text-size-16;
  }

  .livePlayer__audienceCount {
    font-weight: 400;
    color: var(--uikit-text-color-secondary);
  }

  .livePlayer__audienceListContent {
    flex: 1;
    overflow: hidden;
  }
}
```

组件定制化

组件属性

LiveAudienceList 组件属性

Props 属性	类型	默认值	是否必填	说明
children	(params: { audience: AudienceInfo; }) => React.ReactNode	-	否	自定义观众标记（例如徽章、身份标识等），显示在头像和昵称之间。
className	String	-	否	自定义 CSS class 类型名
style	CSSProperties	-	否	自定义 CSS 样式
height	String	-	否	观众列表高度。

AudienceInfo 数据类型

属性	类型	默认值	说明
userId	String	-	观众 ID
userName	String	-	观众名称（昵称）
avatarUrl	String	-	观众头像地址
userRole	Number	2	观众角色。 • 0：直播间房主 • 1：直播间管理员 • 2：普通观众
isMessageDisabled	Boolean	false	是否被禁止发送文字、表情消息。
joinedTimestamp	Number	0	观众进入直播间的时间。
customInfo	Record<String, any>	-	用户自定义属性

自定义示例

版权所有：腾讯云计算（北京）有限责任公司

```
import { LiveAudienceList } from 'tuikit-atomicx-react';
import type { AudienceInfo } from 'tuikit-atomicx-react';

// 自定义组件属性，必须与 LiveAudienceList 组件 children 属性相同
interface CustomAudienceProps {
  params: {audience: AudienceInfo}
}

// 用户自定义组件，显示用户角色
const CustomAudience: React.FC<CustomAudienceProps> = ({ params }) => {
  return (
    <div className="custom-audience-item">
      {
        params.audience.userRole === 2 ? "[观众]" :
(params.audience.userRole === 1 ? "[管理员]" : "[主播]")
      }
    </div>
  );
};

const LivePlayer: React.FC<LivePlayerProps> = ({ className }) => {
  return (
    <div className={`${styles.livePlayer} ${className || ''}`}>
          <div className={styles.livePlayer__audienceListContent}>
            {/* 设置 height、className、style 属性 */}
            <LiveAudienceList height="100%" className="my-class-name"
style={{backgroundColor: "transparent"}}>
              {(params) => <CustomAudience params={params} />} {/* 在播放
页观众列表中，使用自定义组件， 显示用户角色 */}
            </LiveAudienceList>
          </div>
    </div>
  );
};
```

下一步

接入直播视频组件后，您可能还想继续接入**弹幕消息、礼物**等功能，可以参阅下表指引文档，继续接入这些功能。

功能	描述	集成指引
聊天弹幕组件	支持发送、接收并显示文字、表情消息。	弹幕系统（Web React）
直播送礼组件	展示配置的礼物列表，支持发送礼物、礼物播放。	礼物系统（Web React）

版权所有：腾讯云计算（北京）有限责任公司

礼物组件

礼物组件 (Android)

最近更新时间：2026-06-22 10:21:36

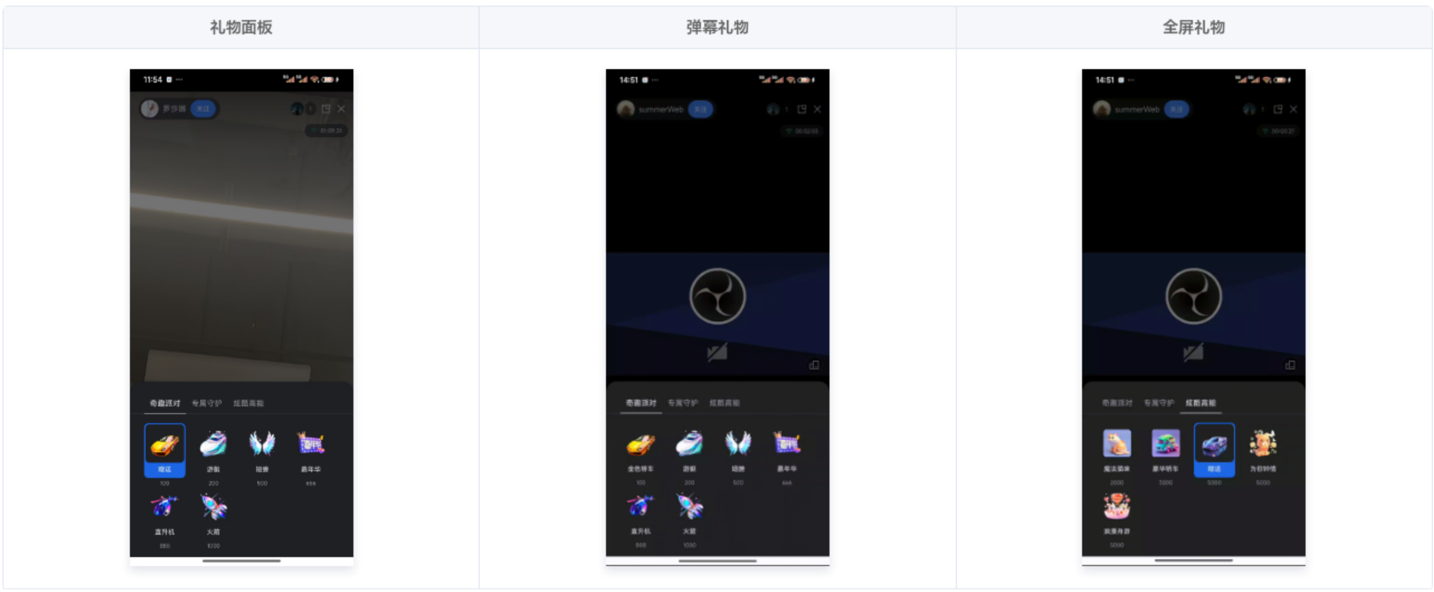
本文档将指导您如何在 `Android` 项目中快速对 `TUILiveKit` 的直播送礼组件进行自定义。

组件概述

`TUILiveKit` 的礼物功能主要由以下三个核心视图组件构成：

组件名称	类名	功能描述
礼物列表	`GiftListView`	展示礼物列表，处理用户的点击选择和发送事件。
礼物播放组件	`GiftPlayView`	负责接收礼物消息，并在屏幕上渲染对应的动画特效（例如 SVGA 动画）。

效果演示



快速开始

步骤1：开通服务

请参考 开通服务，领取 `TUILiveKit` 体验版或者开通付费版。

> **说明：**

> 使用礼物系统要求开通 `TUILiveKit` 体验版、多人连麦版或大规模直播版，不同套餐中可配置的礼物数量有所不同，详细参见 功能与计费说明 中的**礼物系统**说明，您可按需选购。

步骤2：代码集成

- **工程配置**：请参考 准备工作 完成 `TUILiveKit` 接入。
- **版本限制**：`TUILiveKit` >= 3.2.0。

步骤3：接入礼物列表

`GiftListView` 通常以弹窗或底部面板的形式展示。您也可以将其添加到 `Activity` 或 `Fragment` 的布局中。

```
import android.os.Bundle
import android.widget.FrameLayout
import androidx.appcompat.app.AppCompatActivity
import com.trtc.uikit.livekit.R
import com.trtc.uikit.livekit.component.gift.GiftListView

class YourLiveActivity : AppCompatActivity() {

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

        // 获取页面的根布局容器
        val rootView = findViewById<FrameLayout>(R.id.root_view)

        // 1. 创建礼物列表组件实例
        val giftListView = GiftListView(this)

        // 2. 初始化组件，传入当前的房间 ID (RoomId)
        // 注意：请确保此时已获取到有效的 roomId
        giftListView.init("your_room_id")

        // 3. 将组件添加到视图层级中
        rootView.addView(giftListView)
    }
}
```

版权所有：腾讯云计算（北京）有限责任公司

步骤4：接入礼物特效播放组件

`GiftPlayView` 是一个透明的覆盖层，用于播放礼物动画。它通常覆盖在视频画面的上方，但位于交互按钮的下方。

```
import android.os.Bundle
import android.widget.FrameLayout
import androidx.appcompat.app.AppCompatActivity
import com.trtc.uikit.livekit.R
import com.trtc.uikit.livekit.component.gift.GiftPlayView

class YourLiveActivity : AppCompatActivity() {

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

        val rootView = findViewById<FrameLayout>(R.id.root_view)

        // 1. 创建礼物播放组件实例
        val giftPlayView = GiftPlayView(this)

        // 2. 初始化组件，传入当前的房间 ID
        // 组件内部会自动监听该房间的礼物消息并触发播放
        giftPlayView.init("your_room_id")

        // 3. 将组件添加到视图层级中
        // 建议将此 View 添加到视频层之上，UI 控件层之下
        rootView.addView(giftPlayView)
    }
}
```

步骤 5：导入默认礼物

您新申请的应用在礼物后台默认是没有配置任何礼物的，所以集成后您的礼物发送面板默认是空的，为了您能快速体验我们的礼物收发效果，您可以使用 服务端 API 一键导入我们为您预置的礼物素材。

下一步

完成上述 `UI` 集成后，您已经实现了客户端的礼物展示能力。为了构建一个完整的商业化礼物功能，您还需要参考 礼物系统后端对接与进阶功能 文档，完成以下核心业务逻辑的对接：

版权所有：腾讯云计算（北京）有限责任公司

- **自定义礼物配置**：通过服务端 `API` 上传您自己的礼物图标、动画文件，并设置价格。
- **礼物扣费回调**：配置回调地址，将送礼请求转发至您的计费后台，实现“余额校验”与“扣费”闭环。
- **PK 分数联动**：在主播 `PK` 场景下，将礼物价值实时转换为 `PK` 分数。
- **数据统计**：获取礼物发送记录、总价值等运营数据。
- **升级礼物动画特效 SDK**：如果 `SVGA` 无法满足需求，可集成高级播放器支持 `MP4/PAG` 等高清透明动画。

礼物组件 (iOS)

最近更新时间：2026-06-22 10:21:41

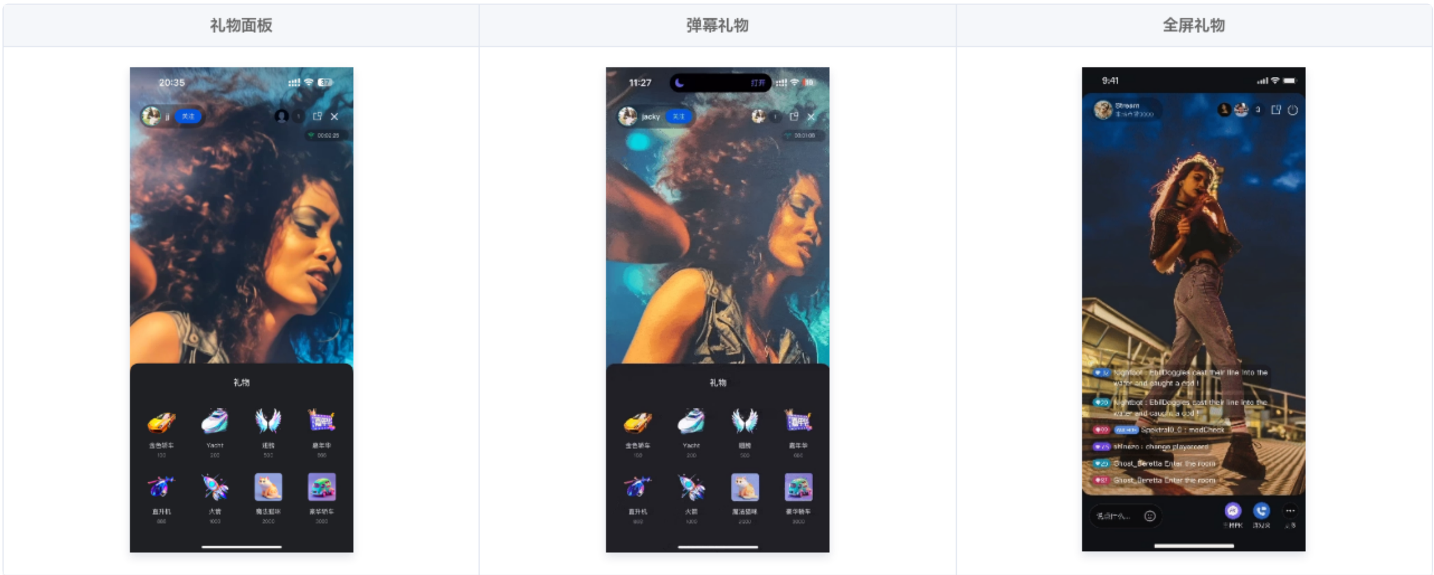
本文档将指导您如何在 iOS 项目中快速集成 TUILiveKit 的直播送礼组件。

组件概述

TUILiveKit 的礼物功能主要由以下两个核心视图组件构成:

组件名称	类名	功能描述
礼物选择面板	`GiftListView`	展示礼物列表，处理用户的点击选择和发送事件。
礼物播放组件	`GiftPlayView`	负责接收礼物消息，并在屏幕上渲染对应的动画特效（例如 SVGA 动画）。

效果演示



快速开始

步骤 1. 开通服务

参考 开通服务 文档开通 TUILiveKit，使用礼物系统要求开通体验版、多人连麦版或大规模直播版。不同套餐中可配置的礼物数量有所不同，详见 功能与计费说明 中的**礼物系统**说明，您可按需选购。

步骤 2. 代码集成

- **工程配置**：请参考 准备工作 完成 `TUILiveKit` 接入。
- **版本限制**： `TUILiveKit` >= 3.2.0。

版权所有：腾讯云计算（北京）有限责任公司

步骤 3：接入礼物列表展示页面

在您的应用中接入礼物列表展示页面，使观众能够直观地看到可赠送的礼物列表。请参考示例代码创建 `GiftListView` 组件并添加到视图：

```
import TUILiveKit

class YourGiftViewController: UIViewController {
    // 1. 创建 GiftListView 对象
    //        - roomId: 与观众当前进入的直播间 roomId 一致
    lazy var giftListView = {
      let view = GiftListView(roomId: liveId)
      return view
    }()

    private let liveId: String
    // ... 其他代码 ...

    public override func viewDidLoad() {
        super.viewDidLoad()
        // 2. 将组件添加到您的视图中并设置布局
        view.addSubView(giftListView)
        giftPlayView.snp.remakeConstraints { make in
            make.leading.trailing.equalToSuperview()
            make.height.equalTo(256)
            make.bottom.equalToSuperview()
        }
    }
}
```

步骤 4：接入礼物特效播放页面

在您的应用中接入礼物特效播放页面， `GiftPlayView` 组件已内置礼物消息接收与播放能力，请参考示例代码创建 `GiftPlayView` 组件并添加到视图：

```
import TUILiveKit
// YourAnchorViewController 代表您的主播视图控制器，观众端可参考以下示例：
class YourAnchorViewController: UIViewController {
    // 1. 创建并初始化 GiftPlayView 对象
    //        - roomId: 与观众当前进入的直播间 roomId 一致
```

```
    lazy var giftPlayView = {
      let view = GiftPlayView(roomId: liveId)
      return view
    }()

    private let liveId: String
    // ... 其他代码 ...

    public override func viewDidLoad() {
        super.viewDidLoad()
        // 2. 将组件添加到您的视图中并设置布局
        view.addSubView(giftPlayView)
        giftPlayView.snp.remakeConstraints { make in
          make.edges.equalToSuperview()
        }
    }
}
```

步骤 5：导入默认礼物

您新申请的应用在礼物后台默认是没有配置任何礼物的，所以集成后您的礼物发送面板默认是空的，为了您能快速体验我们的礼物收发效果，您可以使用 服务端 API 一键导入我们为您预置的礼物素材。

下一步

完成上述 UI 集成后，您已经实现了客户端的礼物展示能力。为了构建一个完整的商业化礼物功能，您还需要参考 礼物系统后端对接与进阶功能 文档，完成以下核心业务逻辑的对接：

- **自定义礼物配置**：通过服务端 API 上传您自己的礼物图标、动画文件，并设置价格。
- **礼物扣费回调**：配置回调地址，将送礼请求转发至您的计费后台，实现“余额校验”与“扣费”闭环。
- **PK 分数联动**：在主播 PK 场景下，将礼物价值实时转换为 PK 分数。
- **数据统计**：获取礼物发送记录、总价值等运营数据。
- **升级礼物动画特效 SDK**：如果 SVGA 无法满足需求，可集成高级播放器支持 MP4/PAG 等高清透明动画。

版权所有：腾讯云计算（北京）有限责任公司

礼物组件 (Web Vue3)

最近更新时间：2026-06-22 10:21:46

本文将引导您在自己的项目中，快速接入**TUILiveKit** 直播送礼组件：**LiveGift** 。

核心功能

功能	说明
礼物列表	显示配置的礼物列表，可以点击“更多”按钮查看完整礼物列表。
礼物发送	礼物列表中选中礼物后，可以点击发送。发送的礼物信息会显示在聊天消息中。
礼物播放	支持 SVG 2D、SVG 3D 和 MP4 视频礼物，SVG 3D 礼物支持全屏播放。

组件接入

步骤1：环境配置与开通服务

在进行快速接入之前，您需要参考 准备工作（Web Vue3），满足相关环境配置及开通对应服务。

> **说明：**
> 使用礼物系统要求开通 TUILiveKit 体验版、多人连麦版或大规模直播版，不同套餐中可配置的礼物数量有所不同，详细参见 TUILiveKit 价格总览 中的**礼物系统**说明，您可按需选购。

步骤2：安装依赖

npm

```
npm install tuikit-atomicx-vue3 @tencentcloud/uikit-base-component-vue3
--save
```

pnpm

```
pnpm add tuikit-atomicx-vue3 @tencentcloud/uikit-base-component-vue3
```

yarn

```
yarn add tuikit-atomicx-vue3 @tencentcloud/uikit-base-component-vue3
```

步骤3：集成直播送礼组件

在您的项目中引入并使用**直播送礼组件**，可直接复制如下示例代码到您的项目中，实现**礼物列表**、**发送礼物**、**3D 礼物播放**。

```
<template>
    <div class="gift-panel">
        <LiveGift />
    </div>
</template>

<script setup lang="ts">
import { LiveGift } from 'tuikit-atomicx-vue3';
</script>

<style lang="scss" scoped>
.gift-panel{
  padding: 6px 0;
```

版权所有：腾讯云计算（北京）有限责任公司

```
  border-top: 1px solid var(--stroke-color-primary);
  background-color: var(--bg-color-operate);
  display: flex;

  &.disabled {
    pointer-events: none;
    opacity: 0.5;
    cursor: not-allowed;
  }
}
</style>
```

步骤4：导入默认礼物

您新申请的应用在礼物后台默认是没有配置任何礼物的，所以集成后您的礼物发送面板默认是空的，为了您能快速体验我们的礼物收发效果，您可以使用 服务端 API 一键导入我们为您预置的礼物素材。

下一步

完成上述 UI 集成后，您已经实现了客户端的礼物展示能力。为了构建一个完整的商业化礼物功能，您还需要参考 礼物系统后端对接与进阶功能 文档，完成以下核心业务逻辑的对接：

- **自定义礼物配置**：通过服务端 API 上传您自己的礼物图标、动画文件，并设置价格。
- **礼物扣费回调**：配置回调地址，将送礼请求转发至您的计费后台，实现“余额校验”与“扣费”闭环。
- **PK 分数联动**：在主播 PK 场景下，将礼物价值实时转换为 PK 分数。
- **数据统计**：获取礼物发送记录、总价值等运营数据。
- **升级礼物动画特效 SDK**：如果 SVGA 无法满足需求，可集成高级播放器支持 MP4/PAG 等高清透明动画。

版权所有：腾讯云计算（北京）有限责任公司

礼物组件 (Web React)

最近更新时间：2026-06-22 10:21:51

本文将引导您在自己的项目中，快速接入**TUILiveKit** 直播送礼组件：**LiveGift** 。

核心功能

功能	说明
礼物列表	显示配置的礼物列表，可以点击“更多”按钮查看完整礼物列表。
礼物发送	礼物列表中选中礼物后，可以点击发送。发送的礼物信息会显示在聊天消息中。
礼物播放	支持 SVG 2D、SVG 3D 和 MP4 视频礼物，SVG 3D 礼物支持全屏播放。

组件接入

步骤1：环境配置与开通服务

在进行快速接入之前，您需要参考 准备工作（Web React），满足相关环境配置及开通对应服务。

版权所有：腾讯云计算（北京）有限责任公司

> **说明：**
> 使用礼物系统要求开通 TUILiveKit 体验版、多人连麦版或大规模直播版，不同套餐中可配置的礼物数量有所不同，详细参见 TUILiveKit 价格总览 中的**礼物系统**说明，您可按需选购。

步骤2：安装依赖

npm

```
npm install tuikit-atomicx-react @tencentcloud/uikit-base-component-
react --save
```

pnpm

```
pnpm install tuikit-atomicx-react @tencentcloud/uikit-base-component-
react
```

yarn

```
yarn add tuikit-atomicx-react @tencentcloud/uikit-base-component-react
```

步骤3：集成直播送礼组件

在您的项目中引入并使用**直播送礼组件**，可直接复制如下示例代码到您的项目中，实现**礼物列表、发送礼物、3D 礼物播放**。

GiftPanel.tsx

```
import React from "react";
import { LiveGift } from "tuikit-atomicx-react";
import styles from "./GiftPanel.module.scss";
const GiftPanel: React.FC = () => {
  return (
    <div className={styles.livePlayer__giftPanel}>
      <LiveGift />
    </div>
  )
};
```

```
export default GiftPanel;
```

GiftPanel.module.scss

```
.livePlayer__giftPanel{
  width: 100%;
  height: 130px;
  flex-shrink: 0;
  border-top: 1px solid var(--uikit-stroke-color-primary);
  background: var(--uikit-bg-color-operate);
}
```

步骤 4：导入默认礼物

您新申请的应用在礼物后台默认是没有配置任何礼物的，所以集成后您的礼物发送面板默认是空的，为了您能快速体验我们的礼物收发效果，您可以使用 服务端 API 一键导入我们为您预置的礼物素材。

下一步

完成上述 UI 集成后，您已经实现了客户端的礼物展示能力。为了构建一个完整的商业化礼物功能，您还需要参考 礼物系统后端对接与进阶功能 文档，完成以下核心业务逻辑的对接：

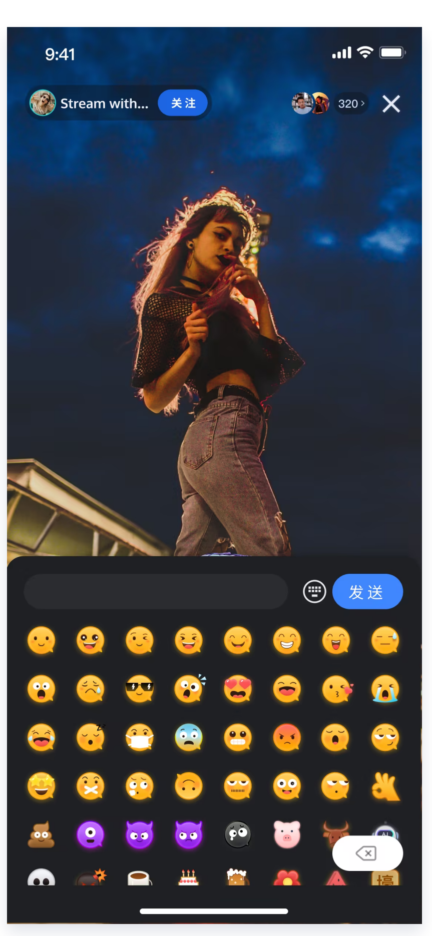
- **自定义礼物配置**：通过服务端 API 上传您自己的礼物图标、动画文件，并设置价格。
- **礼物扣费回调**：配置回调地址，将送礼请求转发至您的计费后台，实现“余额校验”与“扣费”闭环。
- **PK 分数联动**：在主播 PK 场景下，将礼物价值实时转换为 PK 分数。
- **数据统计**：获取礼物发送记录、总价值等运营数据。
- **升级礼物动画特效 SDK**：如果 SVGA 无法满足需求，可集成高级播放器支持 MP4/PAG 等高清透明动画。

弹幕组件
弹幕组件 (Android Kotlin)

最近更新时间：2026-02-26 16:00:48

功能预览

`TUILiveKit` 弹幕系统为直播场景提供完整互动解决方案，能够增强直播的互动性和趣味性。通过本文档，您可快速实现直播间弹幕互动功能，并支持深度定制以满足业务需求。

弹幕消息发送组件（BarrageInputView）	弹幕消息展示组件（BarrageStreamView）
	

组件构成

组件名称	具体内容
弹幕消息展示组件（BarrageStreamView）	负责实时展示和管理弹幕消息流的组件，提供**消息列表渲染、时间聚合、用户交互和响应式适配**等完整的消息展示解决方案。

弹幕消息发送组件（BarrageInputView）	提供富文本编辑和消息发送功能的输入组件，集成**表情选择器、字符限制和状态管理**，为用户提供流畅的消息输入体验。

快速开始

步骤1：开通服务

参考 开通服务 文档开通「体验版」或「大规模直播版」套餐。

步骤2：代码集成

参考 准备工作 接入 `TUILiveKit` 。

步骤3：接入弹幕消息发送组件

在您的应用中接入弹幕消息发送组件，使主播/观众能够方便地发送弹幕消息。请参考示例代码创建 `BarrageInputView` 组件并添加到您的视图：

Java

```
// 1. 创建 BarrageInputView 对象
val barrageInputView = BarrageInputView(context)

// 2. 初始化 BarrageInputView ，liveId 为您当前进入的直播间房间 Id
barrageInputView.init("your liveId")

// 3. 将 BarrageInputView 对象添加到您的视图上
yourBarrageInputContainerView.addView(barrageInputView)
```

说明：

1. 弹幕消息发送组件支持**系统键盘**和**表情键盘**切换。
2. 为尊重表情设计版权， `TUILiveKit` 工程中不包含大表情元素切图，正式上线商用前请您替换为自己设计或拥有版权的其他表情包。下图所示默认的**小黄脸表情包版权归腾讯云所有**，可有偿授权使用，如果需要获得授权可 提交工单 联系我们。

步骤4：接入弹幕消息展示组件

在您的应用中接入弹幕消息展示组件，使主播/观众能够直观地看到已发送的弹幕消息列表。请参考示例代码创建 `BarrageStreamView` 组件并添加到视图：

```java
// 1. 创建 BarrageStreamView 对象
val barrageStreamView = BarrageStreamView(context)

// 2. 初始化 BarrageStreamView 对象，liveId 为您当前进入的直播间房间id，ownerId 为当前房主的 userId，用来区分房主与观众的显示效果
barrageStreamView.init("your liveId", "your ownerId")

// 3. 将 BarrageStreamView 对象添加到您的视图上
yourBarrageContainerView.addView(barrageStreamView)
```

插入本地弹幕消息

`BarrageStreamView` 提供了插入本地消息的能力，该消息仅在本地显示，通常用于“系统提示”、“欢迎语”或“操作反馈”等。可以参考如下代码插入消息并显示到本地弹幕中（代码示例为在弹幕中插入一条文本消息）：

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

// 1. 创建 Barrage 对象
val barrage = Barrage()
```

```
// 2. 初始化 Barrage 消息内容，您可以根据业务诉求，自定义您的消息
barrage.textContent = "your barrage context"
barrage.sender.userID = "your sender userId"
barrage.sender.userName = "your sender userName"
barrage.sender.avatarURL = "your sender avatarURL"

// 3. 将创建的弹幕消息插入本地弹幕展示组件中
barrageStreamView.insertBarrages(barrage)
```

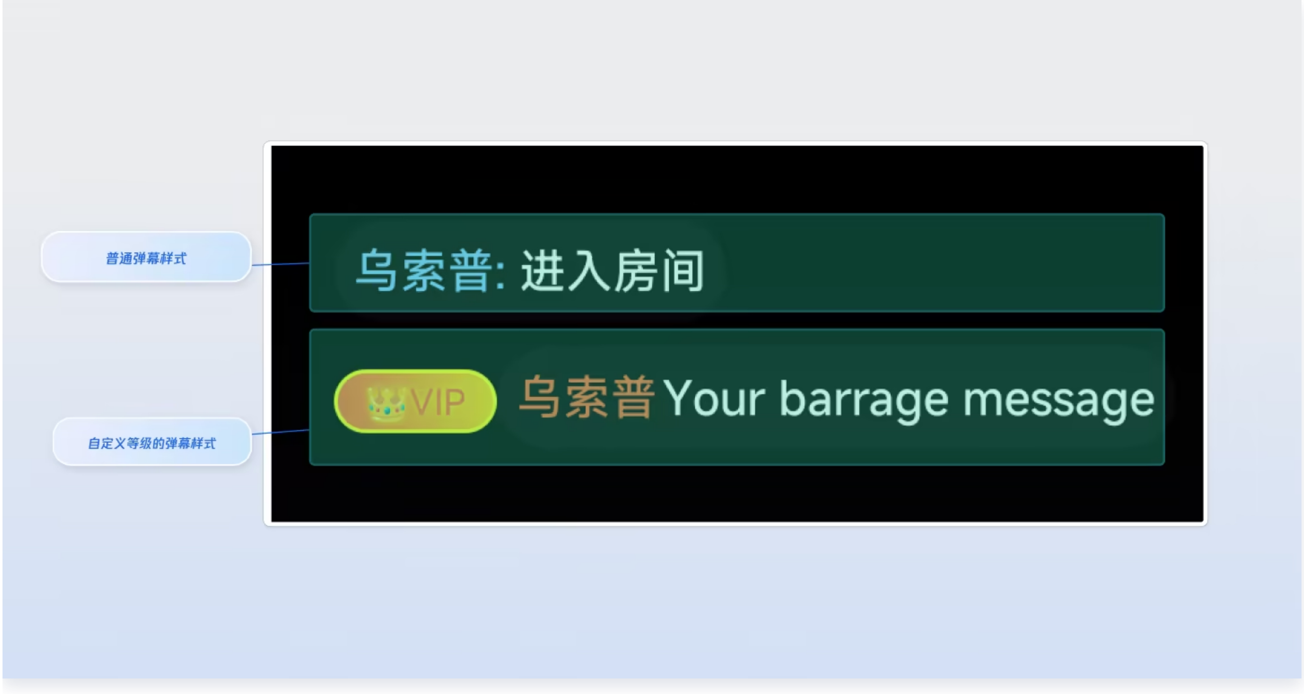
自定义弹幕消息样式

`TUILiveKit` 的弹幕系统支持高度的样式定制。您可以根据消息携带的“业务标签（Tag）”来决定其展示效果。本章节分为两部分：如何修改系统默认弹幕，以及如何自定义本地插入的消息样式。

场景一：修改默认弹幕样式

如果您希望修改弹幕组件默认的文本弹幕或礼物弹幕的 UI 效果，可以直接覆盖其预设的适配器。

效果示例



实现步骤

1. **实现 UI 适配器**

   创建一个类实现 `BarrageItemAdapter`，定义您理想的布局和数据绑定逻辑。

```
class YourBarrageAdapter(private val context: Context) :
BarrageItemAdapter {
```

```
    override fun onCreateViewHolder(parent: ViewGroup):
RecyclerView.ViewHolder {
        // 按照您的业务诉求自定义 UI 视图
        val view =
LayoutInflater.from(parent.context).inflate(R.layout.item_barrage_your
_custom, parent, false)
        return YourViewHolder(view)
    }

    override fun onBindViewHolder(holder: RecyclerView.ViewHolder,
position: Int, barrage: Barrage) {
        (holder as? YourViewHolder)?.bind(barrage)
    }

    private inner class YourViewHolder(view: View) :
        RecyclerView.ViewHolder(view) {

        fun bind(barrage: Barrage) {
            // 绑定数据
        }
    }
}
```

2. **注册并覆盖默认样式**

   在 `BarrageStreamView` 初始化后进行注册。`TUILiveKit` 当前已内置的弹幕类型定义：类型 0 为普通文本消息，类型 1 为礼物消息。

```
// 覆盖默认文本弹幕 (类型 0)
barrageStreamView.setItemAdapter(0, YourBarrageAdapter())

// 覆盖默认礼物弹幕 (类型 1)
barrageStreamView.setItemAdapter(1, YourGiftBarrageAdapter())
```

场景二：自定义插入的本地弹幕消息样式

此场景适用于由您的业务逻辑触发（例如：系统欢迎语、操作反馈），且仅在本地显示的弹幕。通过“定义标签”与“识别标签”实现定制。

实现步骤

1. **自定义弹幕标识**

   在调用 `UI` 接口插入消息前，为弹幕对象关联一个自定义的业务标识（例如 `type = "system_notice"`）。

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

private fun insertSystemWelcome() {
    val barrage = Barrage()
    barrage.textContent = "欢迎来到直播间，请文明发言。"

    // 配置业务标识，用于后续 UI 识别
    val extInfo = HashMap<String, String>()
    extInfo["type"] = "system_notice"
    barrage.extensionInfo = extInfo

    // 直接调用 UI 组件接口插入，不涉及远端发送
    barrageStreamView.insertBarrages(barrage)
}
```

2. **定义弹幕识别逻辑**

   实现 `BarrageItemTypeDelegate` 接口，根据标签返回您自定义的样式 `ID`（如定义类型 100 为您的自定义样式 ID）。

```
class YourBarrageTypeDelegate : BarrageItemTypeDelegate {
    override fun getItemType(position: Int, barrage: Barrage): Int {
        val type = barrage.extensionInfo?.get("type")
        return when (type) {
            "system_notice" -> 100 // 自定义样式 ID
            else -> 0              // 默认走普通文本弹幕的逻辑
        }
    }
}
```

3. **实现并注册自定义适配器**

   定义 `YourSystemNoticeAdapter` 来承接 `ItemType 100` 的渲染工作。

```
// 1. 定义适配器
class YourSystemNoticeAdapter : BarrageItemAdapter {
```

```
    override fun onCreateViewHolder(parent: ViewGroup):
RecyclerView.ViewHolder {
        val view = LayoutInflater.from(parent.context)
            .inflate(R.layout.item_barrage_system_notice, parent,
false)
        return SystemNoticeViewHolder(view)
    }

    override fun onBindViewHolder(holder: RecyclerView.ViewHolder,
position: Int, barrage: Barrage) {
        (holder as? SystemNoticeViewHolder)?.bind(barrage)
    }

    private inner class SystemNoticeViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
        fun bind(barrage: Barrage) {
            // 渲染自定义视图逻辑
        }
    }
}

// 2. 绑定到组件
barrageStreamView.setItemTypeDelegate(YourBarrageTypeDelegate())
barrageStreamView.setItemAdapter(100, YourSystemNoticeAdapter())
```

常见问题

如何在自定义弹幕消息中区分房主和观众?

您也可以在自定义弹幕样式时，根据 `Barrage` 消息中的 `sender.userID` 字段来判断是否为房主，并显示您想要的特定效果。

为什么我的自定义样式弹幕无法显示?

如果自定义样式未生效，请按以下步骤排查：

- **步骤1**：注册状态检查。确保您已在 `BarrageStreamView` 初始化后，正确调用了 `setItemTypeDelegate` 和 `setItemAdapter` 方法。
- **步骤2**：类型 `ID` 匹配。检查 `getItemType` 返回的整数 ID 是否与注册适配器时传入的完全一致。
- **步骤3**：数据源检查。确认发送端在 `extensionInfo` 中确实包含了用于识别类型的标识位。

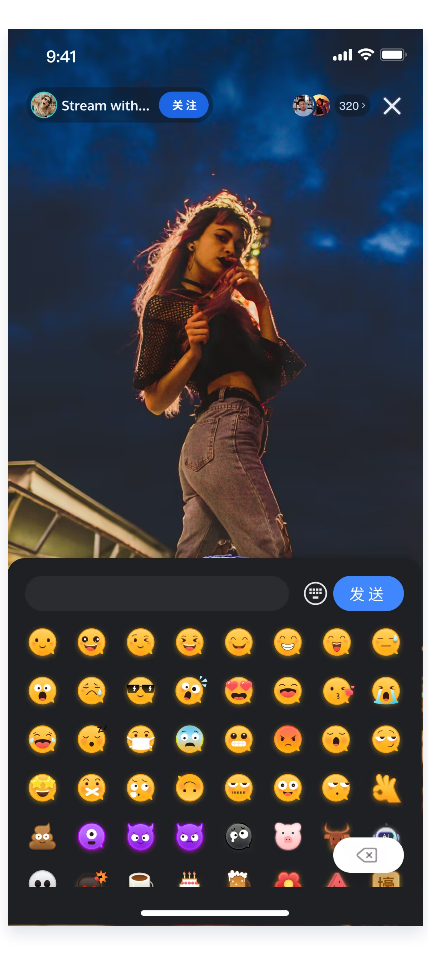
版权所有：腾讯云计算（北京）有限责任公司

弹幕组件 (iOS UIKit)

最近更新时间：2026-03-03 18:01:03

功能预览

`TUILiveKit` 直播弹幕为直播场景提供完整互动解决方案，能够增强直播的互动性和趣味性。通过本文档，您可快速实现直播间弹幕互动功能，并支持深度定制以满足业务需求。

弹幕消息发送组件（BarrageInputView）	弹幕消息展示组件（BarrageStreamView）
	

组件构成

组件名称	具体内容
弹幕消息组件（BarrageStreamView）	负责实时展示和管理弹幕消息流的组件，提供**消息列表渲染、时间聚合、用户交互和响应式适配**等完整的消息展示解决方案。
消息发送组件（BarrageInputView）	提供富文本编辑和消息发送功能的输入组件，集成**表情选择器、字符限制、状态管理和跨平台适配**，为用户提供流畅的消息输入体验。

版权所有：腾讯云计算（北京）有限责任公司

快速开始

步骤1：开通服务

参考 开通服务 文档开通**体验版**或**大规模直播版**套餐。

步骤2：代码集成

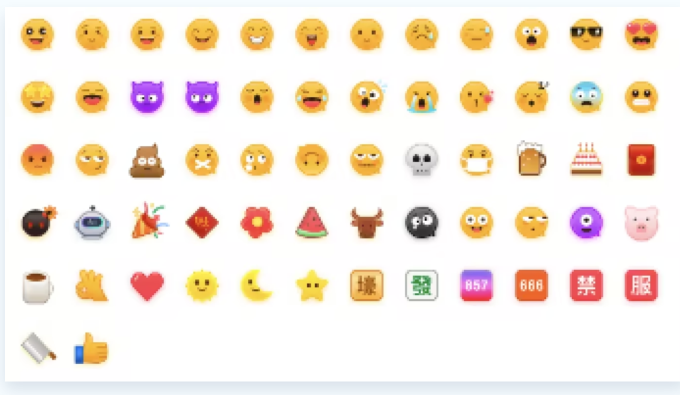
参考 准备工作 完成 TUILiveKit 组件接入。

步骤3：接入弹幕消息发送组件

在您的应用中接入弹幕消息发送组件，使主播/观众能够方便地发送弹幕消息。请参考示例代码创建 `BarrageInputView` 组件并添加到您的视图

Swift

```
// 1. 创建 BarrageInputView 对象
let barrageInputView = BarrageInputView(roomId: roomId)
// ...此处将barrageInputView添加到您的父视图上并调整布局
```

> **说明：**
> 1. 弹幕消息发送组件支持**系统键盘**和**表情键盘**切换。
> 2. 为尊重表情设计版权，`TUILiveKit` 工程中不包含大表情元素切图，正式上线商用前请您替换为自己设计或拥有版权的其他表情包。下图所示默认的**小黄脸表情包版权归腾讯云所有**，可有偿授权使用，如果需要获得授权可 提交工单 联系我们。
>
> 

步骤4：接入弹幕消息展示组件

在需要展示弹幕的位置，创建并初始化 `BarrageStreamView` 来展示弹幕消息。`ownerId` 用于区分房主和观众的显示效果。

Swift

```
// 1. 创建 BarrageStreamView 对象
let barrageDisplayView = BarrageStreamView(roomId: roomId)

// 2. 此处将barrageInputView添加到您的父视图上并调整布局
barrageDisplayView.setOwnerId(ownerId)
```

步骤5：插入本地弹幕消息

`BarrageStreamView` 提供了 `insertBarrages` 接口，用于插入自定义消息（例如礼物消息、直播间公告等）。您可以通过自定义样式实现不同的展示效果。

> **说明：**
> 此操作必须在进房成功后执行。

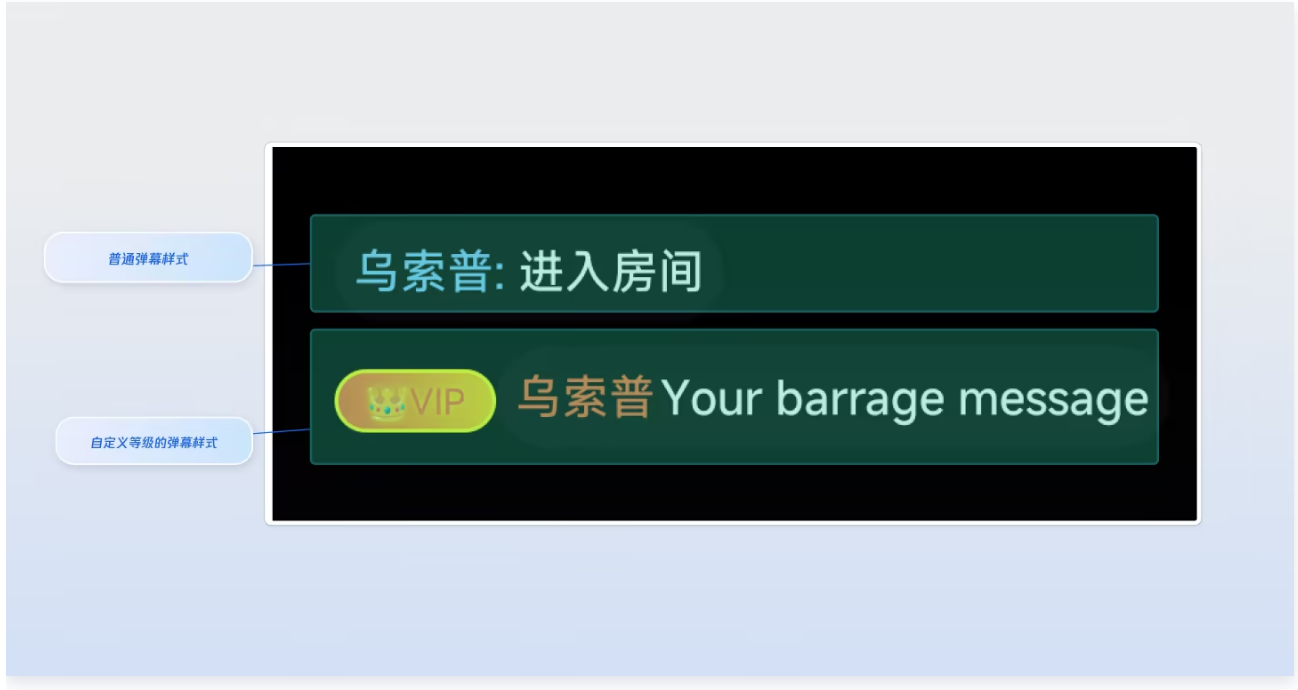
Swift

```
import TUILiveKit

// 示例：在弹幕区插入一条礼物消息
let barrage = Barrage()
barrage.textContent = "gift"
barrage.sender.userId = "sender.userId"
barrage.sender.userName = "sender.userName"
barrage.sender.avatarUrl = "sender.avatarUrl"
barrage.sender.level = "sender.level"

let giftCount = 1
barrage.extensionInfo = [
    "TYPE": "GIFTMESSAGE",
    "gift_name": gift.name,
    "gift_count": "\(giftCount)",
    "gift_icon_url": gift.iconURL,
    "gift_receiver_username": userName
]

barrageDisplayView.insertBarrages([barrage])
```

版权所有：腾讯云计算（北京）有限责任公司

自定义弹幕消息样式

效果示例



弹幕消息样式默认有两种：普通弹幕（样式为 0）和自定义消息样式。如果您需要更多样式，可实现 `BarrageStreamView` **的代理方法** `BarrageStreamViewDelegate` 。

Swift

```
class MyViewController: BarrageStreamViewDelegate {
    let barrageDisplayView = BarrageStreamView(roomId: roomId)

    override func viewDidLoad() {
        super.viewDidLoad()
        barrageDisplayView.delegate = self  // 设置BarrageStreamView的代理

        // ...
    }

    func onBarrageClicked(user: LiveUserInfo) {
        // 此处您可添加弹幕消息点击事件处理逻辑，user为消息发送者信息
    }

    func barrageDisplayView(_ barrageDisplayView: BarrageStreamView,
createCustomCell barrage: Barrage) -> UIView? {
```

```
        guard let type = barrage.extensionInfo?["TYPE"], type ==
"GIFTMESSAGE" else {
            // 是否需要使用自定义UI，如不需要返回 nil 即可
            return nil
            // 如果需要，也可以根据消息类型返回特定的样式
            // return GiftBarrageView(barrage: barrage)
        }

        // 返回自定义消息样式UI
        return CustomBarrageView(barrage: barrage)
    }
}


// 自定义UI
class CustomBarrageView: UIView {
    let barrage: TUIBarrage
    init(barrage: TUIBarrage) {
        self.barrage = barrage
        super.init(frame: .zero)
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
    // ...此处布局与绘制您自己的UI
}
```

常见问题

为什么我无法看到弹幕消息?

请检查以下几点：

- 确保您已经正确初始化了 `BarrageInputView` 和 `BarrageStreamView`，并且传递了正确的 `LiveID`。
- 检查网络连接是否正常。

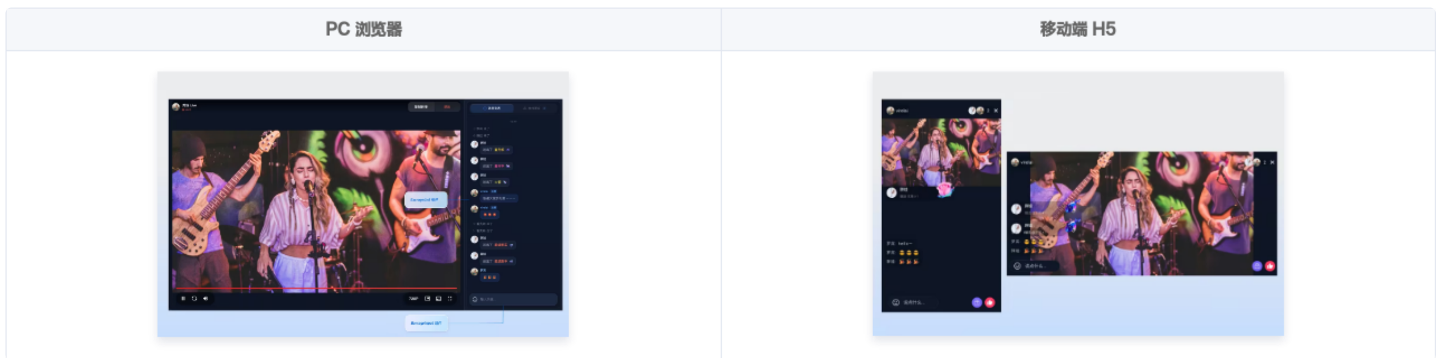
如何在弹幕消息中区分房主和观众?

在初始化 `BarrageStreamView` 时，您需要传入 `ownerId` 参数。组件会根据 `ownerId` 自动识别房主发送的弹幕，并应用不同的样式。您也可以通过自定义弹幕样式，根据 `Barrage` 消息中的 `user.userId` 字段来判断是否为房主，并显示您想要的特定效果。

弹幕组件 (Web Vue3)

最近更新时间：2026-05-11 16:56:11

`TUILiveKit` 弹幕系统为直播场景提供完整互动解决方案，能够增强直播的互动性和趣味性。通过本文档，您可快速实现直播间弹幕互动功能，并支持深度定制以满足业务需求。



组件构成

组件名称	具体内容
弹幕消息组件（BarrageList）	负责实时展示和管理弹幕消息流的组件，提供**消息列表渲染、时间聚合、用户交互和响应式适配**等完整的消息展示解决方案。
消息发送组件（BarrageInput）	提供富文本编辑和消息发送功能的输入组件，集成**表情选择器、字符限制、状态管理和跨平台适配**，为用户提供流畅的消息输入体验。

组件接入

步骤1：配置环境并开通服务

在进行快速接入之前，请参考 准备工作，完成相关环境配置及开通对应服务。

步骤2：安装依赖

npm

```
npm install tuikit-atomicx-vue3 @tencentcloud/uikit-base-component-vue3
--save
```

pnpm

```
pnpm add tuikit-atomicx-vue3 @tencentcloud/uikit-base-component-vue3
```

版权所有：腾讯云计算（北京）有限责任公司

yarn

```
yarn add tuikit-atomicx-vue3 @tencentcloud/uikit-base-component-vue3
```

步骤3：接入弹幕消息和发送组件

在您的项目中引入并使用弹幕组件，可直接复制如下示例至您的项目中展示**完整的直播间弹幕消息组件以及消息发送组件**。

```
<template>
  <UIKitProvider theme="dark">
    <div class="app">
      <div class="chat-container">
        <div class="chat-content">
          <BarrageList class="barrage-list" />
        </div>
        <div class="chat-input">
          <BarrageInput class="barrage-input" />
        </div>
      </div>
    </div>
  </UIKitProvider>
</template>

<script setup lang="ts">
import { onMounted, ref } from 'vue';
import { UIKitProvider } from '@tencentcloud/uikit-base-component-vue3';
import { BarrageList, BarrageInput, useLoginState, useLiveListState }
from 'tuikit-atomicx-vue3';

const { login } = useLoginState();
const { joinLive } = useLiveListState();

async function initLogin() {
  try {
    await login({
      sdkAppId: 0,          // SDKAppID, 可以参考步骤 1 获取
      userId: '',           // UserID, 可以参考步骤 1 获取
      userSig: '',          // userSig, 可以参考步骤 1 获取
```

```
    });
  } catch (error) {
    console.error('登录失败:', error);
  }
}

onMounted(async () => {
  await initLogin();
  await joinLive({
    liveId: '输入对应直播间 LiveId',      // 输入对应 liveId 进入直播间
  });
});
</script>

<style scoped>.app{width:100vw;height:100vh;display:flex;justify-
content:center;align-items:center;padding:20px;box-sizing:border-
box}.chat-container{width:100%;max-width:500px;height:600px;border-
radius:16px;display:flex;flex-direction:column;overflow:hidden}.chat-
content{flex:1;overflow:hidden}.barrage-
list{width:100%;height:100%}.chat-input{background-color:var(--bg-color-
dialog);padding:16px}.barrage-input{width:100%}</style>
```

自定义组件

弹幕系统的两个核心组件均支持灵活的自定义能力，您可以根据业务需求选择不同的定制方式。

弹幕消息组件（BarrageList）自定义

弹幕消息组件提供了三个层次的自定义能力，您可以根据定制深度选择合适的方式：

定制场景	推荐方式	说明
仅调整容器/消息项的颜色、间距等样式	Props: `containerStyle` / `itemStyle`	最简单，几行样式代码即可实现。
替换整个消息渲染组件，但保留组件内部的消息分类逻辑	Props: `Message`	传入自定义 Vue 组件，控制单条消息的渲染方式。（**礼物消息的渲染无法控制**）
完全接管消息渲染，需要按消息类型差异化展示（例如区分礼物	Slot: `message-item`	最灵活，所有消息（含礼物）都通过插槽传递。

消息）		

> ⚠ **注意：**
> 使用 BarrageList 的默认消息组件时，表情消息会自动渲染为图片。但当您通过 `Message Props` 或 `message-item` Slot 接管自定义消息渲染时，message.textContent 中的表情编码（例如 [TUIEmoji_Flower]）不会自动解析，需要您参考 如何处理自定义弹幕消息时的表情渲染 处理。

containerStyle + itemStyle 调整样式

通过 `containerStyle`、`itemStyle`、`style`、`height` 等 Props，您可以快速调整弹幕消息组件的外观，无需编写额外组件。

```
<BarrageList
  :containerStyle="{
    padding: '10px',
    overflow: 'hidden',
  }"
  :itemStyle="{
    background: 'rgba(99, 102, 241, 0.12)',
    borderRadius: '12px',
    padding: '8px 12px',
    border: '1px solid rgba(139, 92, 246, 0.2)',
    boxShadow: '0 1px 4px rgba(0, 0, 0, 0.15)',
    marginRight: '6px',
    boxSizing: 'border-box',
  }"
/>
```

替换消息组件（Message Props）

如果样式调整无法满足需求，您可以编写一个自定义 Vue 组件，然后通过 BarrageList 的 `Message` Props 传入，从而完全替换默认的消息渲染逻辑。BarrageList 在渲染每条消息时，会将当前弹幕消息对象作为 `message` Props 传递给您的自定义组件。
因此，您的自定义组件需要声明接收一个 `message` Props：

Props	类型	说明
message	Barrage	当前弹幕消息对象，包含 `textContent`（文本内容）、`sender`（发送者信息）、`timestampInSecond`（发送时间戳）、`data`（自定义数据 JSON 字符串）等字段。

步骤 1：创建自定义消息组件

```
// MyCustomMessage.vue
<template>
  <div class="custom-message">
    <div class="message-header">
      <span class="user-name">{{ getSenderName(message) }}</span>
      <span class="message-time">{{
formatTime(message.timestampInSecond) }}</span>
    </div>
    <div class="message-content">
      {{ getMessageText(message) }}
    </div>
  </div>
</template>

<script setup lang="ts">
import type { Barrage } from 'tuikit-atomicx-vue3';

const props = defineProps<{
  message: Barrage;
}>();

const formatTime = (timestampInSecond: number) => {
  return new Date(timestampInSecond * 1000).toLocaleTimeString('zh-CN',
{
    hour: '2-digit',
    minute: '2-digit'
  });
```

```
};

const getSenderName = (message: Barrage) => {
  const sender = message.sender;
  return sender.nameCard || sender.userName || sender.userId || '匿名用
户';
};

const getMessageText = (message: Barrage) => {
  if (message.textContent) {
    return message.textContent;
  }
  return '';
};
</script>

<style scoped>.custom-message{background:linear-gradient(135deg,#667eea
0%,#764ba2 100%);color:white;padding:12px;border-radius:12px;margin:4px
0;box-shadow:0 2px 8px rgba(0,0,0,0.1);transition:transform 0.2s
ease}.custom-message:hover{transform:translateY(-2px)}.message-
header{display:flex;justify-content:space-between;align-
items:center;margin-bottom:8px;font-size:12px;opacity:0.9}.user-
name{font-weight:500;max-width:120px;overflow:hidden;text-
overflow:ellipsis;white-space:nowrap}.message-time{font-
size:11px;opacity:0.7}.message-content{font-size:14px;line-
height:1.4;word-break:break-word}</style>
```

步骤 2：在 BarrageList 中使用自定义组件

```
<template>
  <BarrageList :Message="MyCustomMessage" />
</template>

<script setup lang="ts">
import MyCustomMessage from "./MyCustomMessage.vue";
</script>
```

完全接管消息渲染（Slot 插槽）

当您需要按消息类型进行差异化渲染（例如区分普通消息和礼物消息），可以使用 message-item 插槽完全接管消息渲染，需要注意的是：

1. message-item 插槽的优先级高于 Message Props，两者同时使用时以插槽为准。
2. PC 和 H5 两端的插槽行为一致，所有消息（包括礼物消息）都会传递给插槽。
3. 不使用插槽时，H5 端默认不渲染礼物消息（礼物消息在独立的礼物区域展示）。

插槽参数

插槽名	参数名	参数类型	说明
message-item	message	Barrage	当前弹幕消息对象，包含 `textContent`（文本内容）、`sender`（发送者）、`messageType`（消息类型）、`data`（自定义数据 JSON 字符串）等字段。
	sender	Barrage Sender	发送者信息，包含 `userId`、`userName`、`avatarUrl` 等字段。

```
<template>
  <BarrageList>
    <template #message-item="{ message, sender }">
      <!-- 礼物消息 -->
      <div v-if="isGiftMessage(message)" class="slot-gift-item">
        <img v-if="getGiftIcon(message)" class="slot-gift-img"
:src="getGiftIcon(message)" :alt="getGiftName(message)" />
        <div class="slot-gift-info">
          <span class="slot-gift-sender">{{ sender.nameCard ||
sender.userName || sender.userId }}</span>
          <span class="slot-gift-detail">
            送出 <span class="slot-gift-name">{{ getGiftName(message) }}
</span>
            <span class="slot-gift-count"> x{{ getGiftCount(message) }}
</span>
          </span>
        </div>
      </div>
      <!-- 普通文本消息 -->
      <div v-else class="slot-text-item">
        <span v-if="sender.userId === currentLive?.liveOwner?.userId"
class="slot-owner-badge">主播</span>
```

```
        <span class="slot-sender-name">{{ sender.nameCard ||
sender.userName || sender.userId }}:</span>
        <span class="slot-message-text">{{ message.textContent }}</span>
      </div>
    </template>
  </BarrageList>
</template>

<script setup lang="ts">
import { BarrageList, useLiveListState } from 'tuikit-atomicx-vue3';

const { currentLive } = useLiveListState();

function safelyParseJSON(str: string): any {
  try {
    return JSON.parse(str);
  } catch {
    return null;
  }
}

function isGiftMessage(message: { data?: string }): boolean {
  if (!message.data) return false;
  const data = safelyParseJSON(message.data);
  return data?.type === 'gift';
}

function getGiftData(message: { data?: string }) {
  if (!message.data) return null;
  const data = safelyParseJSON(message.data);
  return data?.type === 'gift' ? data : null;
}

function getGiftName(message: { data?: string }): string {
  return getGiftData(message)?.giftInfo?.name || '礼物';
}

function getGiftIcon(message: { data?: string }): string {
  return getGiftData(message)?.giftInfo?.iconUrl || '';
}
```

```
function getGiftCount(message: { data?: string }): number {
  return getGiftData(message)?.count || 1;
}
</script>

<style scoped>.slot-text-item{font-size:12px;line-height:1.6;padding:6px
12px;border-radius:10px;background:rgba(255,255,255,0.05);word-
break:break-word}.slot-owner-badge{display:inline-
block;background:linear-
gradient(135deg,#f59e0b,#ef4444);color:#fff;font-size:10px;font-
weight:600;padding:1px 8px;border-radius:10px;margin-right:6px;vertical-
align:middle;line-height:1.6}.slot-sender-
name{color:rgba(167,139,250,0.85);font-weight:500;margin-
right:4px}.slot-message-text{color:rgba(255,255,255,0.85)}.slot-gift-
item{display:flex;align-items:center;gap:10px;padding:10px 14px;border-
radius:14px;background:linear-
gradient(135deg,rgba(255,0,128,0.15),rgba(255,140,0,0.12)
50%,rgba(168,85,247,0.1));border:1px solid rgba(255,0,128,0.25);box-
shadow:0 0 12px rgba(255,0,128,0.1),inset 0 0 12px
rgba(255,140,0,0.05)}.slot-gift-img{width:40px;height:40px;flex-
shrink:0;object-fit:contain;filter:drop-shadow(0 0 6px
rgba(255,140,0,0.5))}.slot-gift-info{display:flex;flex-
direction:column;gap:2px;min-width:0}.slot-gift-sender{font-
size:12px;color:rgba(255,255,255,0.8);font-
weight:500;overflow:hidden;text-overflow:ellipsis;white-
space:nowrap}.slot-gift-detail{font-
size:12px;color:rgba(255,255,255,0.5)}.slot-gift-
name{color:#ffaa00;font-weight:700;text-shadow:0 0 8px
rgba(255,170,0,0.4)}.slot-gift-count{color:#ff4081;font-weight:800;font-
size:13px;margin-left:2px;text-shadow:0 0 8px rgba(255,64,129,0.4)}
</style>
```

Props 速查表

参数名	参数类型	默认值	说明
Message	Component	Message	自定义消息组件。
containerStyle	CSSProperties	-	自定义消息列表容器样

			式。
itemStyle	CSSProperties	-	自定义单条消息项样式。
height	String	-	组件高度，支持 CSS 单位。
style	CSSProperties	-	指定根元素自定义样式。

消息发送组件（BarrageInput）自定义配置

消息发送组件提供了样式、尺寸、输入行为和发送流程等多个维度的自定义能力，以下按使用场景逐一说明。

调整样式

消息发送组件提供了 `containerStyle` 、 `containerClass` 属性用于自定义外观。

通过内联样式自定义

给 `containerStyle` 属性传递一个样式对象，可调整输入框容器的样式。

```
<BarrageInput
  :containerStyle="{
    background: 'linear-gradient(135deg, rgba(255, 0, 128, 0.1) 0%,
rgba(99, 102, 241, 0.15) 50%, rgba(168, 85, 247, 0.1) 100%)',
    border: '1.5px solid rgba(168, 85, 247, 0.35)',
    borderRadius: '24px',
    padding: '8px 18px',
    boxShadow: '0 0 20px rgba(139, 92, 246, 0.15), 0 0 40px rgba(255, 0,
128, 0.05), inset 0 0 12px rgba(99, 102, 241, 0.08)',
  }"
/>
```

通过 CSS 类名自定义

给 `containerClass` 属性传递一个类名字符串，可使用自定义 CSS 类控制样式。

```
<template>
  <BarrageInput containerClass="cyberpunk-input" />
</template>

<style>.cyberpunk-input{background:linear-
gradient(135deg,rgba(15,10,40,0.9),rgba(30,15,60,0.85))!important;border
:1.5px solid transparent!important;border-
radius:16px!important;padding:8px 18px!important;box-shadow:0 0 15px
rgba(139,92,246,0.2),0 0 30px rgba(255,0,128,0.08),inset 0 1px 0
rgba(255,255,255,0.05)!important;transition:box-shadow .3s
ease!important;border-image:linear-
gradient(135deg,#ff0080,#7c3aed,#06b6d4) 1!important;border-image-
slice:1!important}.cyberpunk-input:focus-within{box-shadow:0 0 24px
rgba(139,92,246,0.35),0 0 48px rgba(255,0,128,0.12),inset 0 1px 0
rgba(255,255,255,0.08)!important}</style>
```

调整尺寸

通过 `width`、`height`、`minHeight`、`maxHeight` 参数，您可以灵活地控制 BarrageInput 的尺寸。

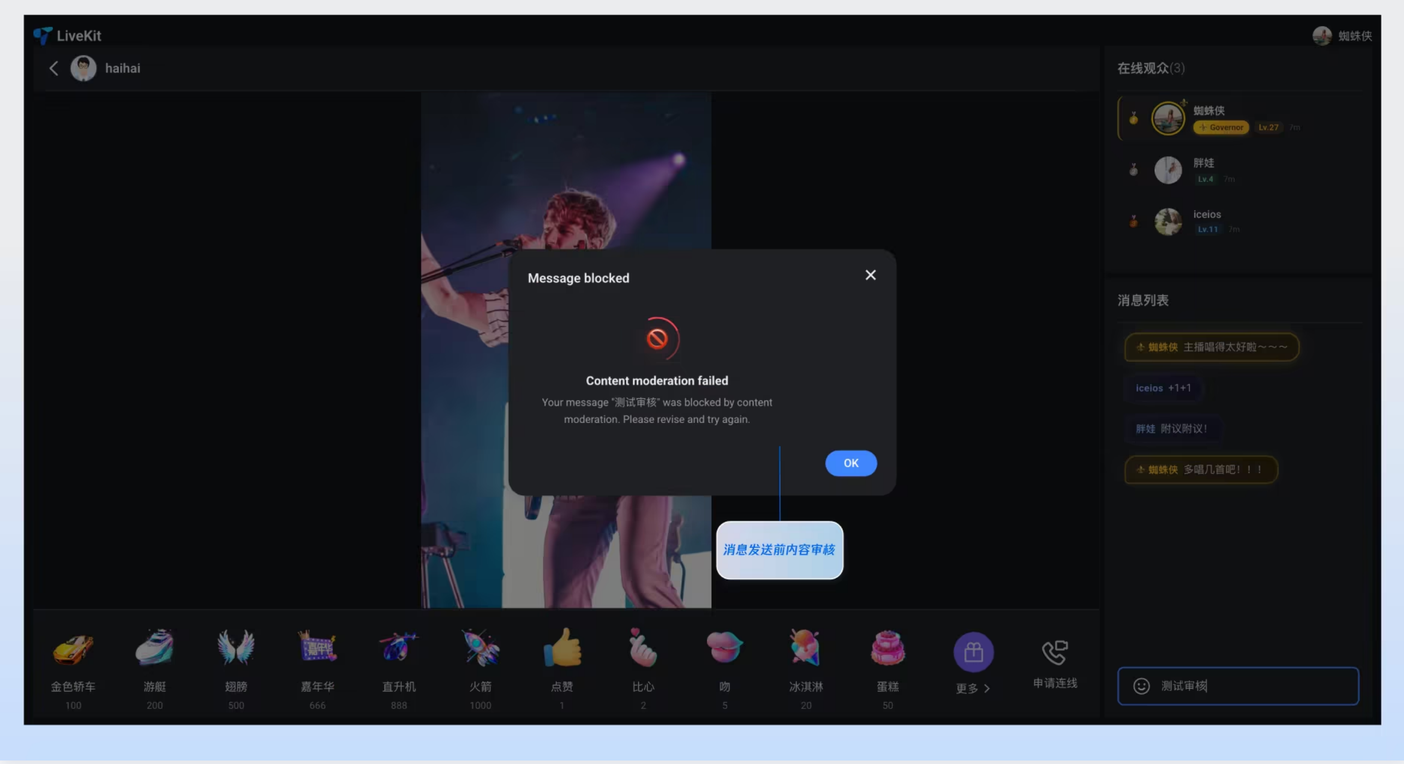
```
<BarrageInput
  width="70%"
  height="60px"
  minHeight="60px"
  maxHeight="60px"
  :containerStyle="{
    margin: '0 auto',
    background: 'linear-gradient(135deg, rgba(99, 102, 241, 0.12) 0%,
rgba(168, 85, 247, 0.08) 100%)',
    border: '1.5px solid rgba(139, 92, 246, 0.3)',
    borderRadius: '16px',
    padding: '10px 18px',
    boxShadow: '0 0 16px rgba(139, 92, 246, 0.12)',
  }"
/>
```

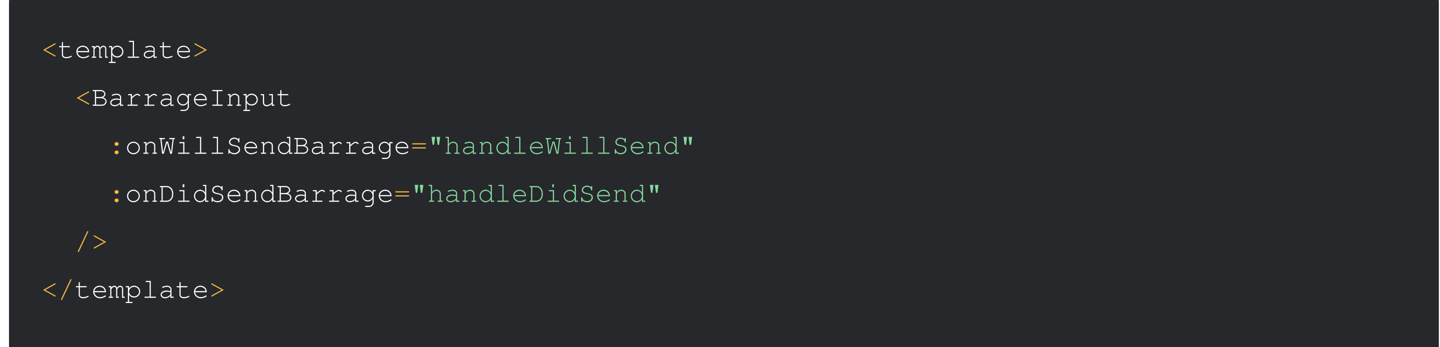
介入发送流程

消息发送组件提供了两个钩子，允许您在弹幕发送前后介入处理流程：

- onWillSendBarrage：发送前触发，返回 `false` 可拦截发送（支持异步），适用于内容审核、敏感词过滤。
- onDidSendBarrage：发送成功后触发，适用于埋点统计、发送成功提示。

参数名	类型	说明
onWillSendBarrage	(message: Barrage) => void | boolean | Promise<boolean>;	弹幕发送前的回调钩子。返回 `false` 可拦截本次发送，返回 `true` 或 `void` 则允许发送。
onDidSendBarrage	(message: Barrage) => void;	弹幕发送成功后的回调钩子，适用于埋点统计、发送成功提示等场景。





```
<template>
  <BarrageInput
    :onWillSendBarrage="handleWillSend"
    :onDidSendBarrage="handleDidSend"
  />
</template>
```

```
<script setup lang="ts">
import { BarrageInput } from 'tuikit-atomicx-vue3';
import type { Barrage } from 'tuikit-atomicx-vue3';

// 发送前内容过滤 - 返回 false 可拦截发送
function handleWillSend(message: Barrage): boolean {
  const sensitiveWords = ['spam', 'abuse'];
  const hasSensitive = sensitiveWords.some(word => (message.textContent
|| '').includes(word));
  if (hasSensitive) {
    alert('消息包含敏感内容，已拦截发送');
    return false;
  }
  return true;
}

// 发送成功后的埋点统计
function handleDidSend(message: Barrage) {
  console.log('已发送:', message.textContent);
}
</script>
```

监听事件

消息发送组件支持以下事件：

事件名	参数	说明
focus	-	输入框获得焦点时触发。
blur	-	输入框失去焦点时触发。

Props 速查表

参数名	类型	默认值	说明
containerClass	String	''	自定义容器的 CSS 类名。
containerStyle	Record<string, any>	{}	自定义容器的内联样式。

width	String	-	组件宽度，支持 CSS 单位。
height	String	-	组件高度，支持 CSS 单位。
minHeight	String	'40px'	组件最小高度，支持 CSS 单位。
maxHeight	String	'140px'	组件最大高度，支持 CSS 单位。
placeholder	String	-	输入框占位符文本。
disabled	Boolean	false	是否禁用输入框。
autoFocus	Boolean	true	是否自动聚焦到输入框。
maxLength	Number	80	输入内容的最大字符数限制。
onWillSendBarrage	(message: Barrage) => void | boolean | Promise<boolean>;	-	弹幕发送前的回调钩子。接收即将发送的 `Barrage` 消息对象作为参数，返回 `false` 可拦截本次发送，返回 `true` 或 `void` 则允许发送。支持异步回调（`Promise<boolean>`），适用于内容审核、敏感词过滤等场景。
onDidSendBarrage	(message: Barrage) => void;	-	弹幕发送成功后的回调钩子。接收已成功发送的 `Barrage` 消息对象作为参数，适用于埋点统计、发送成功提示等场景。

进阶场景：如何快速实现“弹幕抽奖”

弹幕抽奖是直播互动的核心玩法。通过 `tuikit-atomicx-vue3` 提供的自定义消息能力，您可以轻松实现从“参与抽奖”到“中奖公示”的全流程。

1. 主播端：发起抽奖与开奖

主播点击"发起抽奖"后，通过 sendCustomMessage 向直播间所有人广播一条抽奖开始的消息。之后通过 subscribeEvent 实时监听观众的参与请求，记录参与名单。主播点击"开奖"时随机抽取中奖者，并再次通过自定义消息将中奖结果广播给所有人。

```
<template>
  <!-- Lottery control panel for host -->
  <div v-if="isInLive" class="lottery-panel">
    <div v-if="!lotteryActive && !lotteryWinner">
      <button @click="startLottery">? 发起抽奖</button>
    </div>
    <div v-else-if="lotteryActive">
      <span>抽奖进行中 · {{ lotteryParticipants.length }} 人参与</span>
      <button :disabled="lotteryParticipants.length === 0"
@click="drawLottery">? 开奖</button>
    </div>
    <div v-else-if="lotteryWinner">
      <span>? 中奖: {{ lotteryWinner.userName }}</span>
      <button @click="startLottery">再来一轮</button>
    </div>
  </div>
</template>

<script setup lang="ts">
import { ref, computed } from 'vue';
import {
  useBarrageState,
  useLoginState,
  useLiveListState,
  BarrageType,
  BarrageEvent,
} from 'tuikit-atomicx-vue3';

const { sendCustomMessage, subscribeEvent, unsubscribeEvent } =
useBarrageState();
const { loginUserInfo } = useLoginState();
const { currentLive } = useLiveListState();
const isInLive = computed(() => !!currentLive.value?.liveId);

const lotteryActive = ref(false);
const lotteryParticipants = ref<{ userId: string; userName: string }[]>
([]);
const lotteryWinner = ref<{ userId: string; userName: string } | null>
(null);
```

```
// Step 1: 主播发送 LOTTERY_START 类型的消息给所有观众
const startLottery = async () => {
  lotteryActive.value = true;
  lotteryParticipants.value = [];
  lotteryWinner.value = null;
  await sendCustomMessage({
    businessId: 'LOTTERY_START',
    data: JSON.stringify({
      lotteryId: `LOTTERY_${Date.now()}`,
      hostName: loginUserInfo.value?.userName ||
loginUserInfo.value?.userId || '',
    }),
  });
};

// Step 2: 监听来自观众 LOTTERY_JOIN 类型的消息
const handleLotteryBarrage = (barrage: any) => {
  if (barrage.messageType === BarrageType.custom && barrage.businessId
=== 'LOTTERY_JOIN') {
    try {
      const data = JSON.parse(barrage.data);
      const already = lotteryParticipants.value.some(p => p.userId ===
barrage.sender?.userId);
      if (!already) {
        lotteryParticipants.value.push({
          userId: barrage.sender?.userId,
          userName: data.userName || barrage.sender?.userName || '',
        });
      }
    } catch {
      // ignore
    }
  }
};


// Step 3: 主播随机选择一个「中奖者」并发送 LOTTERY_RESULT 类型的消息
const drawLottery = async () => {
  if (lotteryParticipants.value.length === 0) return;
```

版权所有：腾讯云计算（北京）有限责任公司

```
  const winnerIndex = Math.floor(Math.random() *
lotteryParticipants.value.length);
  const winner = lotteryParticipants.value[winnerIndex];
  lotteryWinner.value = winner;
  await sendCustomMessage({
    businessId: 'LOTTERY_RESULT',
    data: JSON.stringify({
      winnerId: winner.userId,
      winnerName: winner.userName,
    }),
  });
  lotteryActive.value = false;
};


subscribeEvent(BarrageEvent.onBarrageReceived, handleLotteryBarrage);
</script>

<style scoped>.lottery-panel{padding:8px 0;border-top:1px solid
rgba(255,255,255,0.1);display:flex;justify-
content:center;gap:10px;align-items:center}.lottery-panel
button{padding:6px 20px;border:none;border-radius:20px;font-
size:13px;font-weight:600;cursor:pointer;color:#fff;background:linear-
gradient(135deg,#ff0080,#7c3aed);box-shadow:0 0 12px
rgba(255,0,128,0.2);transition:all .2s ease}.lottery-panel
button:hover:not(:disabled){box-shadow:0 0 20px
rgba(255,0,128,0.35);transform:translateY(-1px)}.lottery-panel
button:disabled{opacity:.5;cursor:not-allowed}.lottery-panel span{font-
size:12px;color:rgba(255,255,255,0.7)}</style>
```

2. 观众端：参与抽奖

观众通过 subscribeEvent 监听直播间消息。当收到主播发起的抽奖消息时，显示"参与抽奖"按钮；观众点击参与后，向直播间广播一条参与请求；当收到主播公布的中奖结果时，展示中奖横幅。

```
<template>
  <!-- Show join button when lottery is active -->
  <div v-if="lotteryActive" class="lottery-bar">
    <button :disabled="hasJoinedLottery" @click="joinLottery">
      {{ hasJoinedLottery ? '✅ 已参与' : '▯ 参与抽奖' }}
```

```
    </button>
    <span>{{ lotteryParticipants.length }} 人已参与</span>
  </div>
  <!-- Show winner banner -->
  <div v-if="lotteryWinnerName" class="lottery-winner-banner">
    ? 恭喜 {{ lotteryWinnerName }} 中奖!
  </div>
</template>

<script setup lang="ts">
import { ref } from 'vue';
import {
  useBarrageState,
  useLoginState,
  BarrageType,
  BarrageEvent,
} from 'tuikit-atomicx-vue3';

const { sendCustomMessage, subscribeEvent, unsubscribeEvent } =
useBarrageState();
const { loginUserInfo } = useLoginState();

const lotteryActive = ref(false);
const hasJoinedLottery = ref(false);
const lotteryParticipants = ref<string[]>([]);
const lotteryWinnerName = ref('');

// 当观众点击 "参与抽奖" 时发送 LOTTERY_JOIN 类型的消息
const joinLottery = async () => {
  await sendCustomMessage({
    businessId: 'LOTTERY_JOIN',
    data: JSON.stringify({
      lotteryId: 'ACT_2026_001',
      userName: loginUserInfo.value?.userName ||
loginUserInfo.value?.userId || '',
      timestamp: Date.now(),
    }),
  });
  hasJoinedLottery.value = true;
};
```

```
// 监听所有 LOTTERY_START 类型的消息
const handleLotteryBarrage = (barrage: any) => {
  if (barrage.messageType !== BarrageType.custom) return;

  if (barrage.businessId === 'LOTTERY_START') {
    // 主播开始了新的抽奖
    lotteryActive.value = true;
    hasJoinedLottery.value = false;
    lotteryParticipants.value = [];
    lotteryWinnerName.value = '';
  } else if (barrage.businessId === 'LOTTERY_JOIN') {
    // 有人参与了抽奖
    try {
      const data = JSON.parse(barrage.data);
      if (!lotteryParticipants.value.includes(data.userName)) {
        lotteryParticipants.value.push(data.userName);
      }
    } catch {
      // ignore
    }
  } else if (barrage.businessId === 'LOTTERY_RESULT') {
    // 主播宣布「中奖者」
    lotteryActive.value = false;
    try {
      const data = JSON.parse(barrage.data);
      lotteryWinnerName.value = data.winnerName || '';
      setTimeout(() => { lotteryWinnerName.value = ''; }, 5000);
    } catch {
      // ignore
    }
  }
};

subscribeEvent(BarrageEvent.onBarrageReceived, handleLotteryBarrage);
</script>

<style scoped>.lottery-bar{display:flex;align-
items:center;gap:10px;padding:8px 16px;border-top:1px solid
rgba(255,255,255,0.1)}.lottery-bar button{padding:6px
```

```
16px;border:none;border-radius:20px;font-size:13px;font-
weight:600;cursor:pointer;color:#fff;background:linear-
gradient(135deg,#ff0080,#7c3aed);box-shadow:0 0 12px
rgba(255,0,128,0.2);transition:all .2s ease}.lottery-bar
button:hover:not(:disabled){box-shadow:0 0 20px
rgba(255,0,128,0.35);transform:translateY(-1px)}.lottery-bar
button:disabled{opacity:.6;cursor:not-
allowed;background:rgba(139,92,246,0.3);box-shadow:none}.lottery-bar
span{font-size:12px;color:rgba(255,255,255,0.5)}.lottery-winner-
banner{padding:10px 14px;text-align:center;font-size:13px;font-
weight:500;color:#fff;background:linear-
gradient(135deg,rgba(245,158,11,0.15),rgba(239,68,68,0.1));border:1px
solid rgba(245,158,11,0.25);border-radius:12px;margin:4px
0;animation:banner-pop .4s ease}.lottery-winner-banner .winner-
name{color:#fbbf24;font-weight:700}@keyframes banner-pop{0%
{opacity:0;transform:scale(.9)}100%{opacity:1;transform:scale(1)}}
</style>
```

通过 BarrageList 的 `message-item` 插槽对不同类型的抽奖消息进行差异化渲染。

```
<template>
  <BarrageList>
    <template #message-item="{ message, sender }">
      <!-- 主播发起抽奖 -->
      <div v-if="message.businessId === 'LOTTERY_START'" class="lottery-
sys-msg">
        □ 主播发起了抽奖，快来参与！
      </div>
      <!-- 观众参与抽奖 -->
      <div v-else-if="message.businessId === 'LOTTERY_JOIN'"
class="lottery-join-msg">
        □ {{ sender.nameCard || sender.userName || sender.userId }} 参加
了抽奖
      </div>
      <!-- 中奖结果 -->
      <div v-else-if="message.businessId === 'LOTTERY_RESULT'"
class="lottery-result-msg">
        □ 恭喜 <span class="winner-name">{{ getWinnerName(message) }}
</span> 中奖！
```

```
      </div>
      <!-- 普通文本消息 -->
      <div v-else-if="message.textContent" class="normal-msg">
        <span class="msg-sender">{{ sender.nameCard || sender.userName
|| sender.userId }}:</span>
        <span>{{ message.textContent }}</span>
      </div>
    </template>
  </BarrageList>
</template>

<script setup lang="ts">
import { BarrageList } from 'tuikit-atomicx-vue3';

function getWinnerName(message: { data?: string }): string {
  if (!message.data) return '';
  try {
    const data = JSON.parse(message.data);
    return data.winnerName || '';
  } catch {
    return '';
  }
}
</script>

<style scoped>.lottery-sys-msg{font-size:12px;padding:8px 14px;border-
radius:12px;background:linear-
gradient(135deg,rgba(255,0,128,0.12),rgba(168,85,247,0.1));border:1px
solid rgba(255,0,128,0.2);color:rgba(255,200,50,0.95);font-
weight:500;text-align:center}.lottery-join-msg{font-
size:12px;padding:6px 12px;border-
radius:10px;background:rgba(139,92,246,0.08);color:rgba(167,139,250,0.85
)}.lottery-result-msg{font-size:13px;padding:10px 14px;border-
radius:12px;background:linear-
gradient(135deg,rgba(245,158,11,0.15),rgba(239,68,68,0.1));border:1px
solid rgba(245,158,11,0.25);color:#fff;font-weight:500;text-
align:center}.winner-name{color:#fbbf24;font-weight:700}.normal-
msg{font-size:12px;line-height:1.6;padding:4px 0;word-break:break-
```

```
word}.msg-sender{color:rgba(167,139,250,0.85);font-weight:500;margin-
right:4px}</style>
```

常见问题

如何在自定义弹幕消息中区分房主和观众?

您可以根据 `Barrage` 消息中的 sender.userId 和直播间 ownerId 对比来判断是否为房主，并显示您想要的特定效果。

```
import { useLiveListState } from 'tuikit-atomicx-vue3';

const { currentLive } = useLiveListState();

function isLiveOwner(userId: string): boolean {
  return userId === currentLive.value?.liveOwner.userId;
}
```

如何拦截用户发送的违规弹幕?

您可以利用 `BarrageInput` 的 onWillSendBarrage 发送前钩子：

- 同步/异步拦截：该钩子支持返回 `Promise<boolean>` 。您可以调用后台的敏感词过滤接口，若返回 false，弹幕将不会发出。
- 示例：

```
async function handleWillSend(message) {
  const isLegal = await checkMessageWithAI(message.textContent);
  return isLegal; // 若为 false，组件内部会自动停止发送流程
}
```

如何处理自定义弹幕消息时的表情渲染?

使用 BarrageList 的默认消息组件时，表情消息会自动渲染为图片。但当您通过 Message Props 或 message-item Slot 自定义消息渲染时，message.textContent 中的表情编码（例如 [TUIEmoji_Flower]）不会自动解析，需要您自行处理。

处理自定义弹幕消息时的表情渲染

表情编码格式

弹幕消息中的表情以 [TUIEmoji_xxx] 格式嵌入在 textContent 中。例如：

```
大家好[TUIEmoji_Flower][TUIEmoji_Flower][TUIEmoji_Flower]
```

每个表情编码对应一张托管在 CDN 上的图片，基础 URL 为：

```
https://web.sdk.qcloud.com/im/assets/emoji-plugin/
```

解决方案

步骤1：创建表情解析工具

创建一个 emojiParser.ts 工具文件，将 [TUIEmoji_xxx] 编码解析为可渲染的片段数组：

```
// utils/emojiParser.ts

const EMOJI_BASE_URL =
'https://web.sdk.qcloud.com/im/assets/emoji-plugin/';

const EMOJI_URL_MAP: Record<string, string> = {
  '[TUIEmoji_Expect]': 'emoji_0@2x.png',
  '[TUIEmoji_Blink]': 'emoji_1@2x.png',
  '[TUIEmoji_Guffaw]': 'emoji_2@2x.png',
  '[TUIEmoji_KindSmile]': 'emoji_3@2x.png',
  '[TUIEmoji_Haha]': 'emoji_4@2x.png',
  '[TUIEmoji_Cheerful]': 'emoji_5@2x.png',
  '[TUIEmoji_Smile]': 'emoji_6@2x.png',
  '[TUIEmoji_Sorrow]': 'emoji_7@2x.png',
  '[TUIEmoji_Speechless]': 'emoji_8@2x.png',
  '[TUIEmoji_Amazed]': 'emoji_9@2x.png',
  '[TUIEmoji_Complacent]': 'emoji_10@2x.png',
  '[TUIEmoji_Lustful]': 'emoji_11@2x.png',
  '[TUIEmoji_Stareyes]': 'emoji_12@2x.png',
  '[TUIEmoji_Giggle]': 'emoji_13@2x.png',
  '[TUIEmoji_Daemon]': 'emoji_14@2x.png',
  '[TUIEmoji_Rage]': 'emoji_15@2x.png',
  '[TUIEmoji_Yawn]': 'emoji_16@2x.png',
  '[TUIEmoji_TearsLaugh]': 'emoji_17@2x.png',
  '[TUIEmoji_Silly]': 'emoji_18@2x.png',
  '[TUIEmoji_Wail]': 'emoji_19@2x.png',
  '[TUIEmoji_Kiss]': 'emoji_20@2x.png',
  '[TUIEmoji_Trapped]': 'emoji_21@2x.png',
  '[TUIEmoji_Fear]': 'emoji_22@2x.png',
```

```
    '[TUIEmoji_BareTeeth]': 'emoji_23@2x.png',
    '[TUIEmoji_FlareUp]': 'emoji_24@2x.png',
    '[TUIEmoji_Tact]': 'emoji_25@2x.png',
    '[TUIEmoji_Shit]': 'emoji_26@2x.png',
    '[TUIEmoji_ShutUp]': 'emoji_27@2x.png',
    '[TUIEmoji_Sigh]': 'emoji_28@2x.png',
    '[TUIEmoji_Hehe]': 'emoji_29@2x.png',
    '[TUIEmoji_Silent]': 'emoji_30@2x.png',
    '[TUIEmoji_Skull]': 'emoji_31@2x.png',
    '[TUIEmoji_Mask]': 'emoji_32@2x.png',
    '[TUIEmoji_Beer]': 'emoji_33@2x.png',
    '[TUIEmoji_Cake]': 'emoji_34@2x.png',
    '[TUIEmoji_RedPacket]': 'emoji_35@2x.png',
    '[TUIEmoji_Bombs]': 'emoji_36@2x.png',
    '[TUIEmoji_Ai]': 'emoji_37@2x.png',
    '[TUIEmoji_Celebrate]': 'emoji_38@2x.png',
    '[TUIEmoji_Bless]': 'emoji_39@2x.png',
    '[TUIEmoji_Flower]': 'emoji_40@2x.png',
    '[TUIEmoji_Watermelon]': 'emoji_41@2x.png',
    '[TUIEmoji_Cow]': 'emoji_42@2x.png',
    '[TUIEmoji_Fool]': 'emoji_43@2x.png',
    '[TUIEmoji_Surprised]': 'emoji_44@2x.png',
    '[TUIEmoji_Askance]': 'emoji_45@2x.png',
    '[TUIEmoji_Monster]': 'emoji_46@2x.png',
    '[TUIEmoji_Pig]': 'emoji_47@2x.png',
    '[TUIEmoji_Coffee]': 'emoji_48@2x.png',
    '[TUIEmoji_Ok]': 'emoji_49@2x.png',
    '[TUIEmoji_Heart]': 'emoji_50@2x.png',
    '[TUIEmoji_Sun]': 'emoji_51@2x.png',
    '[TUIEmoji_Moon]': 'emoji_52@2x.png',
    '[TUIEmoji_Star]': 'emoji_53@2x.png',
    '[TUIEmoji_Rich]': 'emoji_54@2x.png',
    '[TUIEmoji_Fortune]': 'emoji_55@2x.png',
    '[TUIEmoji_857]': 'emoji_56@2x.png',
    '[TUIEmoji_666]': 'emoji_57@2x.png',
    '[TUIEmoji_Prohibit]': 'emoji_58@2x.png',
    '[TUIEmoji_Convinced]': 'emoji_59@2x.png',
    '[TUIEmoji_Knife]': 'emoji_60@2x.png',
    '[TUIEmoji_Like]': 'emoji_61@2x.png',
  };
```

```
export type EmojiSegment =
  | { type: 'text'; text: string }
  | { type: 'emoji'; src: string; key: string }
  | { type: 'custom'; key: string };

/**
 * Parse textContent containing [TUIEmoji_xxx] codes into renderable segments.
 * Also supports custom emoji with [@custom_xxx] format.
 *
 * @example
 * parseEmoji('Hello[TUIEmoji_Flower]!')
 * // Returns:
 * // [
 * //   { type: 'text', text: 'Hello' },
 * //   { type: 'emoji', src: 'https://...emoji_40@2x.png', key: '[TUIEmoji_Flower]' },
 * //   { type: 'text', text: '!' },
 * // ]
 */
export function parseEmoji(text: string): EmojiSegment[] {
  const segments: EmojiSegment[] = [];
  let temp = text;
  while (temp !== '') {
    const left = temp.indexOf('[');
    const right = temp.indexOf(']');
    if (left === 0) {
      if (right === -1) {
        segments.push({ type: 'text', text: temp });
        temp = '';
      } else {
        const emojiKey = temp.slice(0, right + 1);
        if (emojiKey.indexOf('@custom') > -1) {
          // Custom emoji: [@custom_xxx] format, render with your own image source
          segments.push({ type: 'custom', key: emojiKey });
          temp = temp.substring(right + 1);
        } else if (EMOJI_URL_MAP[emojiKey]) {
          segments.push({
```

版权所有：腾讯云计算（北京）有限责任公司

```
            type: 'emoji',
            src: EMOJI_BASE_URL + EMOJI_URL_MAP[emojiKey],
            key: emojiKey,
          });
          temp = temp.substring(right + 1);
        } else {
          segments.push({ type: 'text', text: '[' });
          temp = temp.slice(1);
        }
      }
    } else if (left === -1) {
      segments.push({ type: 'text', text: temp });
      temp = '';
    } else {
      segments.push({ type: 'text', text: temp.slice(0, left) });
      temp = temp.substring(left);
    }
  }
  return segments;
}
```

步骤 2：在自定义组件中使用

在 Message Props 自定义组件中使用

```
// CustomBarrageMessage.vue
<template>
  <div class="custom-message">
    <span class="sender">{{ message.sender.userName }}:</span>
    <span class="content">
      <template v-for="(seg, i) in parseEmoji(message.textContent
|| '')" :key="i">
        <img v-if="seg.type === 'emoji'" class="emoji"
:src="seg.src" :alt="seg.key" />
        <span v-else-if="seg.type === 'custom'" class="custom-
emoji">{{ seg.key }}</span>
        <span v-else>{{ seg.text }}</span>
      </template>
    </span>
  </div>
</template>
```

```
<script setup lang="ts">
import type { Barrage } from 'tuikit-atomicx-vue3';
import { parseEmoji } from './utils/emojiParser';

defineProps<{ message: Barrage }>();
</script>

<style scoped>
.emoji {
  width: 20px;
  height: 20px;
  vertical-align: middle;
  margin: 0 1px;
}
</style>
```

在 Slot 插槽中使用

```
<template>
  <BarrageList>
    <template #message-item="{ message, sender }">
      <div class="barrage-item">
        <span class="sender">{{ sender.userName }}:</span>
        <span class="content">
          <template v-for="(seg, i) in
parseEmoji(message.textContent || '')" :key="i">
            <img v-if="seg.type === 'emoji'" class="emoji"
:src="seg.src" :alt="seg.key" />
            <span v-else-if="seg.type === 'custom'"
class="custom-emoji">{{ seg.key }}</span>
            <span v-else>{{ seg.text }}</span>
          </template>
        </span>
      </div>
    </template>
  </BarrageList>
</template>

<script setup lang="ts">
```

```
import { BarrageList } from 'tuikit-atomicx-vue3';
import { parseEmoji } from './utils/emojiParser';
</script>

<style scoped>
.emoji {
  width: 20px;
  height: 20px;
  vertical-align: middle;
  margin: 0 1px;
}
</style>
```

解析原理

parseEmoji 函数从左到右扫描文本，遇到 [时尝试匹配表情编码：

1. 自定义表情（[@custom_xxx]）：生成一个 custom 类型片段，您需要根据 key 自行加载对应的图片资源。
2. 内置表情（[TUIEmoji_xxx]）：匹配成功则生成 emoji 类型片段，包含对应的 CDN 图片地址。
3. 匹配失败（例如 [普通文本]）：将 [作为普通文本处理，继续扫描。
4. 无 [符号：整段作为纯文本输出。

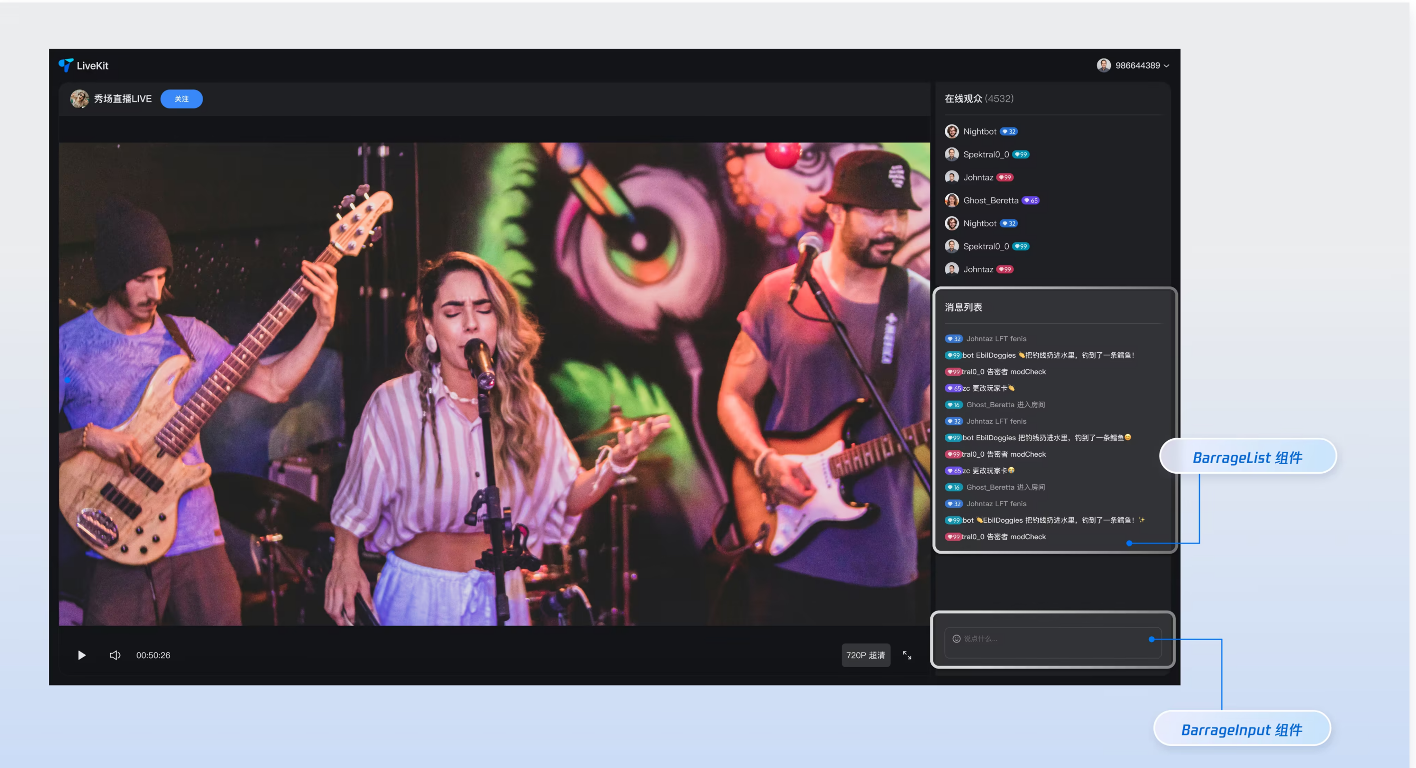
说明：

- 默认的 BarrageList 内置消息组件已包含表情解析能力，仅在自定义渲染时需要手动处理。
- 表情图片托管在腾讯云 CDN（https://web.sdk.qcloud.com/im/assets/emoji-plugin/），请确保您的网络环境可以访问该地址。
- 自定义表情使用 [@custom_xxx] 格式，parseEmoji 会将其识别为 custom 类型片段并保留 key，您需要根据业务自行映射图片地址进行渲染。
- 如需扩展内置表情，可在 EMOJI_URL_MAP 中添加对应的 key-value 映射。

弹幕组件 (Web React)

最近更新时间：2026-04-20 17:34:03

本文对**弹幕组件**进行了详细的介绍，其中包含**弹幕消息组件**（BarrageList）和**消息发送组件**（BarrageInput）。您可以在已有项目中直接参考本文示例集成我们开发好的整体组件，也可以根据您的需求按照文档中的组件自定义部分对样式和布局进行深度的定制。



组件构成

组件名称	具体内容
弹幕消息组件（BarrageList）	负责实时展示和管理弹幕消息流的组件，提供**消息列表渲染、时间聚合、用户交互和响应式适配**等完整的消息展示解决方案。
消息发送组件（BarrageInput）	提供富文本编辑和消息发送功能的输入组件，集成**表情选择器、字符限制、状态管理和跨平台适配**，为用户提供流畅的消息输入体验。

组件接入

步骤1：环境配置及开通服务

在进行快速接入之前，您需要参考 准备工作，满足相关环境配置及开通对应服务。

步骤2：安装依赖

版权所有：腾讯云计算（北京）有限责任公司

npm

```
npm install tuikit-atomicx-react @tencentcloud/uikit-base-component-react --save
npm install sass --save-dev
```

pnpm

```
pnpm add tuikit-atomicx-react @tencentcloud/uikit-base-component-react
pnpm add sass --dev
```

yarn

```
yarn add tuikit-atomicx-react @tencentcloud/uikit-base-component-react
yarn add sass --dev
```

步骤3：集成弹幕组件

在您的项目中引入并使用**直播弹幕组件**，可直接复制如下示例代码到您的项目中，展示完整的直播间**弹幕消息组件**以及**消息发送组件**。

MessageList.tsx

```
import React from "react";
import { useUIKit } from "@tencentcloud/uikit-base-component-react";
import { BarrageList, BarrageInput } from "tuikit-atomicx-react";
import styles from "./MessageList.module.scss";

const MessageList: React.FC = () => {
  const { t } = useUIKit();

  return (
    <div className={styles.livePlayer__messageList}>
      <div className={styles.livePlayer__messageListTitle}>
        <span>{t('live_player_view.message_list_title')}</span>
      </div>
      <div className={styles.livePlayer__messageListContent}>
```

```
        <BarrageList />
        <BarrageInput />
      </div>
    </div>
  );
};

export default MessageList;
```

MessageList.module.scss

```
.livePlayer__messageList {
  display: flex;
  flex-direction: column;
  flex: 1 0 auto;
  margin-top: 8px;
  padding: 8px;
  background: var(--uikit-bg-color-operate);

  .livePlayer__messageListTitle {
    padding: 12px 0;
    border-bottom: 1px solid var(--uikit-stroke-color-primary);
    @include text-size-16;
  }

  .livePlayer__messageListContent {
    display: flex;
    flex: 1;
    flex-direction: column;
  }
}
```

自定义组件

弹幕消息组件和消息发送组件提供了丰富的 Props 属性，用于功能和 UI 展示设置。

弹幕消息组件（BarrageList）

Props 属性名	参数类型	默认值	说明
Message	Component	IBarrageMessageComponentProps	自定义消息组件。
containerStyle	CSSProperties	–	自定义消息列表容器样式。
itemStyle	CSSProperties	–	自定义单条消息项样式。
height	String	–	组件高度，支持 CSS 单位。
style	CSSProperties	–	指定根元素样式的自定义样式。
className	String	–	设置在组件根 DOM 节点上的自定义样式类名。

消息发送组件（BarrageInput）

Props 属性名	类型	默认值	说明
containerClass	String	''	自定义容器的 CSS 类名。
containerStyle	CSSProperties	{}	自定义容器的内联样式。
width	String	–	组件宽度，支持 CSS 单位。
height	String	–	组件高度，支持 CSS 单位。
minHeight	String	'40px'	组件最小高度，支持 CSS 单位。
maxHeight	String	'140px'	组件最大高度，支持 CSS 单位。
placeholder	String	–	输入框占位符文本。
disabled	Boolean	false	是否禁用输入框。
autoFocus	Boolean	true	是否自动聚焦到输入框。
maxLength	Number	80	输入内容的最大字符数限制。
onFocus	() => void	–	输入框获得焦点事件处理函数。
onBlur	() => void	–	输入框失去焦点事件处理函数。

版权所有：腾讯云计算（北京）有限责任公司

自定义示例

自定义样式和尺寸

```
// 消息列表
<BarrageList
  className="custom-barrage-list-name"
  style={{backgroundColor: "#FFFFFF"}}
  containerStyle={{backgroundColor: "#999999"}}
  itemStyle={{backgroundColor: "#000000"}}
  height="200px" />

// 消息输入
<BarrageInput
  className="custom-barrage-input-name"
  autoFocus
  disabled={false}
  width="100%"
  height="100px"
  placeholder="请输入弹幕"
  />
```

自定义消息

```
import React from 'react';
import { BarrageList } from 'tuikit-atomicx-react';
import type { Barrage } from 'tuikit-atomicx-react';

interface ICustomMessageComponentProps {
  message: Barrage;
  isLastInChunk?: boolean;
  style?: React.CSSProperties;
}

const CustomMessage: React.FC<ICustomMessageComponentProps> = ({ message
}) => {
  return (
    <div className="my-message-item">
      {message.sender.userName}: {message.textContent}
```

```
    </div>
  );
};

// 消息列表中，使用自定义消息组件
<BarrageList
  Message={CustomMessage}
/>
```

下一步

接入直播视频组件后，您可能还想继续接入**礼物、观众列表**等功能，可以参阅下表指引文档，继续接入这些功能。

功能	描述	集成指引
直播送礼组件	展示配置的礼物列表，支持发送礼物、礼物播放。	直播送礼组件 (Web React)
观众列表组件	展示当前直播间观众信息。	观众列表组件 (Web React)

版权所有：腾讯云计算（北京）有限责任公司

美颜调节面板

美颜调节面板 (Android Java)

最近更新时间：2025-11-21 14:15:34

TUILiveKit 提供了两种美颜特效方案：**基础美颜**（内置）和**高级美颜**（需额外集成和付费）。您可以根据自己的需求选择合适的方案。

- **基础美颜**

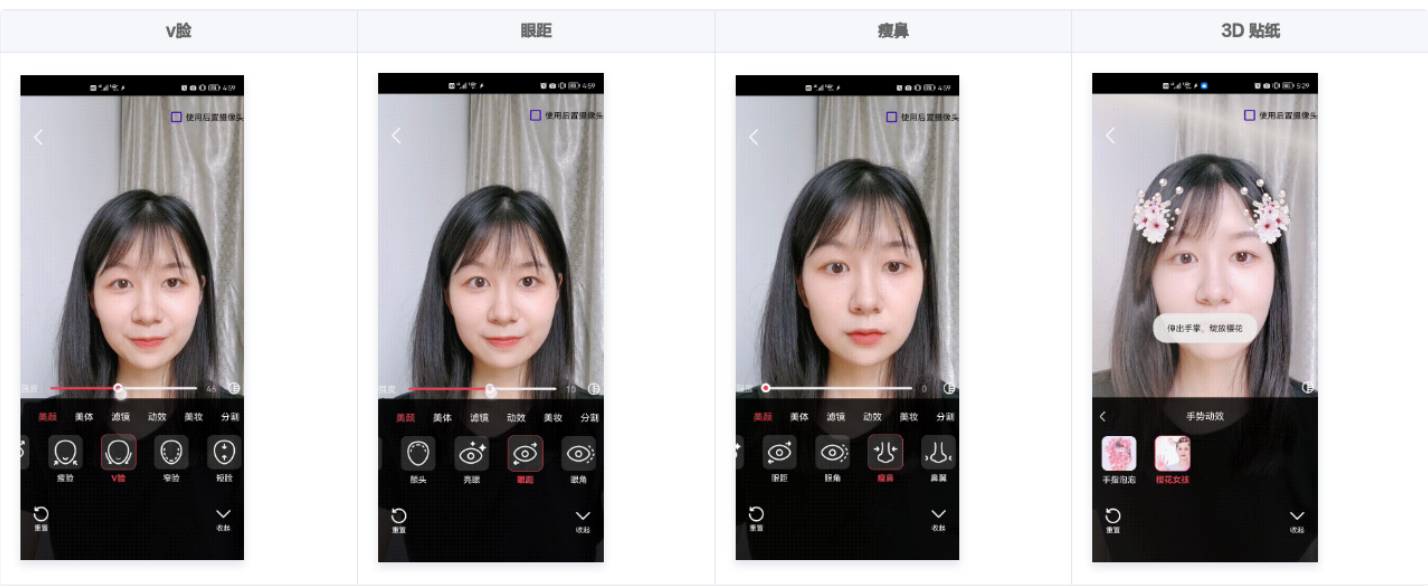
  基础美颜功能已默认集成在 TUILiveKit 中，无需任何额外配置。它提供了 美白、磨皮 和 红润 效果，并支持美颜强度调节。

- **高级美颜**

  高级美颜采用腾讯特效 SDK，提供了更加丰富和专业的美颜效果，例如 V 脸、眼距、瘦鼻、3D 贴纸等。

> **注意：**
>
> 高级美颜功能需要单独付费，详情请参见 腾讯特效 SDK 。

效果展示



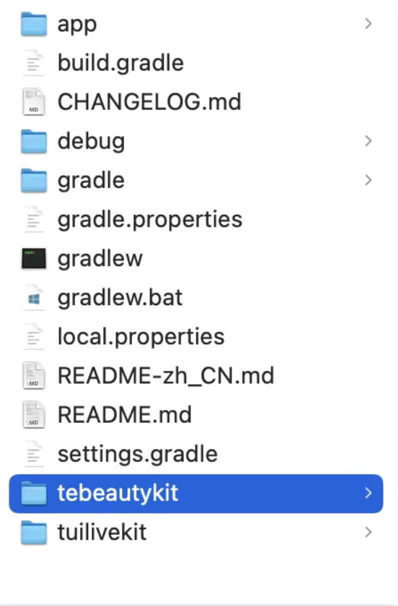
前提条件

参考 准备工作 完成 TUILiveKit 组件接入。

组件接入

步骤 1：集成 tebeautykit

下载并拷贝文件：下载并解压 `TUILiveKit` ，将 `Android/tebeautykit` 文件夹拷贝到您工程的根目录，使其与 app 文件夹同级。



步骤 2：工程配置

编辑您工程根目录的 `settings.gradle` 文件，添加以下代码来导入 `tebeautykit` 模块。

settings.gradle.kts

```
include ':tebeautykit'
```

settings.gradle

```
include ':tebeautykit'
```

步骤 3：鉴权和设置美颜资源

- 获取授权：请先申请授权，获取您的 `LicenseUrl` 和 `LicenseKey` ，具体请参阅 License 指引 。
- 添加鉴权代码：在您的应用初始化位置，添加以下鉴权代码，并替换为您的 美颜套餐编号 、 `LicenseUrl` 和 `LicenseKey` 。

Java

```
TEBeautySettings.getInstance().initBeautySettings(context,
                                 S1_07,               // 将S1_07替换为您
购买的美颜套餐编号
```

```
                                        "LicenseUrl",       // 请替换
LicenseUrl
                                        "LicenseKey");      // 请替换
LicenseKey
```

恭喜您

完成以上步骤后，即可成功集成 TUILiveKit 高级美颜功能。

美颜调节面板 (iOS UIKit)

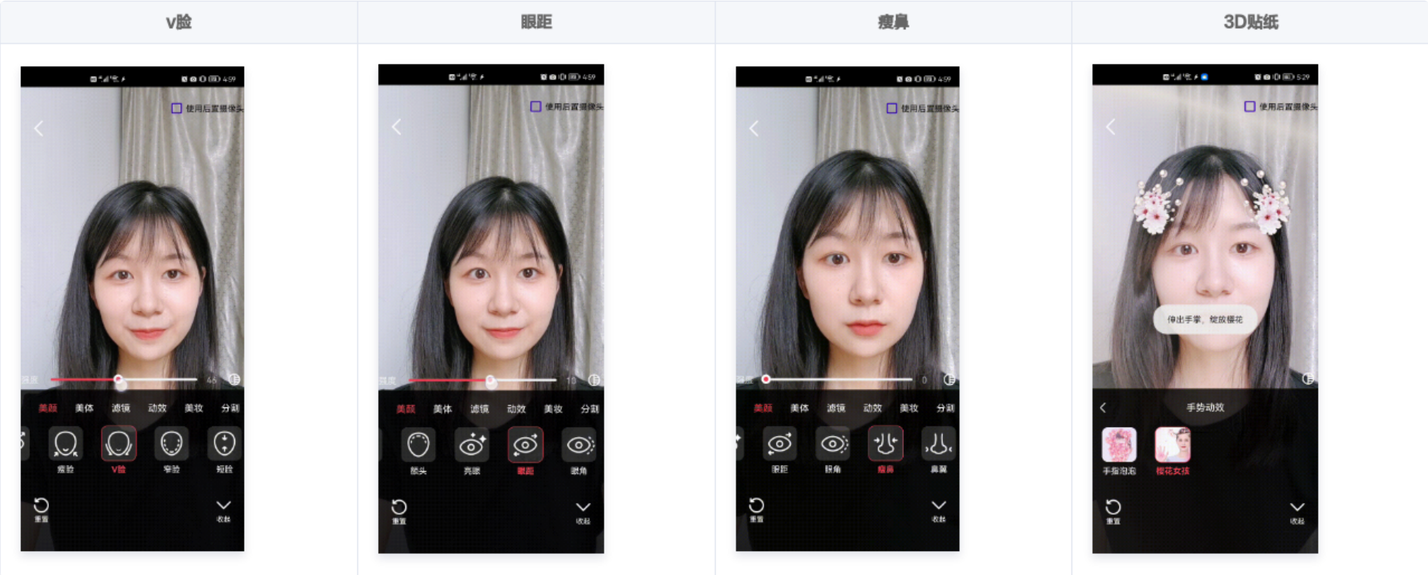
最近更新时间：2025-11-21 14:15:34

TUILiveKit 提供了两种美颜特效方案：**基础美颜**（内置）和 **高级美颜**（需额外集成和付费）。您可以根据自己的需求选择合适的方案。

- **基础美颜**
  基础美颜功能已默认集成在 TUILiveKit 中，无需任何额外配置。它提供了 美白、磨皮 和 红润 效果，并支持美颜强度调节。
- **高级美颜**
  高级美颜采用腾讯特效 SDK，提供了更加丰富和专业的美颜效果，例如 V 脸、眼距、瘦鼻、3D 贴纸等。

> ⚠ **注意：**
> 高级美颜功能需要单独付费，详情请参见 腾讯特效 SDK。

效果展示



前提条件

参考 准备工作 完成 TUILiveKit 组件接入。

组件接入

步骤 1：集成 tebeautykit

下载并拷贝文件：下载并解压 `TUILiveKit`，把 `iOS/TEBeautyKit` 文件夹拷贝到自己的工程中，和 `Podfile` 同级目录。

版权所有：腾讯云计算（北京）有限责任公司

Example
README-zh_CN.md
README.md
TCEffectPlayerKit
TEBeautyKit
TUILiveKit

步骤 2：工程配置

编辑 `Podfile` 文件，添加下面的代码，然后执行 `pod install`：

Swift

```
pod 'TEBeautyKit',:podspec => './TEBeautyKit/TEBeautyKit.podspec'
```

步骤 3：鉴权和设置美颜资源

- 获取授权：请先申请授权，获取您的 `LicenseUrl` 和 `LicenseKey`，具体请参阅 License 指引。
- 添加鉴权代码：在您的应用初始化位置，添加以下鉴权代码，并替换为您的 美颜套餐编号、`LicenseUrl` 和 `LicenseKey`。

Swift

```
//
//  AppDelegate.swift
//

import TEBeautyKit

func application(_ application: UIApplication,
                 didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        TEBeautyKit.setTELicense("LicenseURL", key: "LicenseKEY") {
code, message in
            TEUIConfig.shareInstance().setPanelLevel(.S1_07) // 将S1_07替
换为您购买的美颜套餐编号
        }
        return true
}
```

版权所有：腾讯云计算（北京）有限责任公司

恭喜您

完成以上步骤后，即可成功集成 TUILiveKit 高级美颜功能。

美颜调节面板（Flutter）

最近更新时间：2026-06-15 18:07:43

TUILiveKit 提供了两种美颜特效方案：**基础美颜**（内置）和**高级美颜**（需额外集成和付费）。您可以根据自己的需求选择合适的方案。

- **基础美颜**

  基础美颜功能已默认集成在 TUILiveKit 中，无需任何额外配置。它提供了 美白、磨皮 和 红润 效果，并支持美颜强度调节。

- **高级美颜**

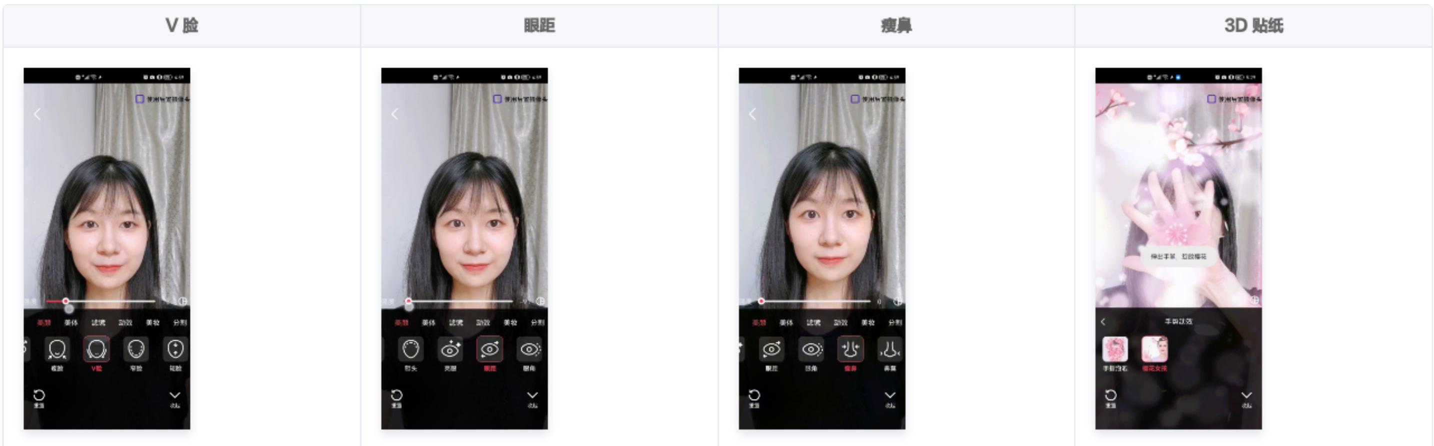
  高级美颜采用腾讯特效 SDK，提供了更加丰富和专业的美颜效果，例如 V 脸、眼距、瘦鼻、3D 贴纸等。

两种美颜面板的效果图：

基础美颜面板	高级美颜面板

高级美颜效果展示

V 脸	眼距	瘦鼻	3D 贴纸



前提条件

参考 准备工作 完成 TUILiveKit 组件接入。

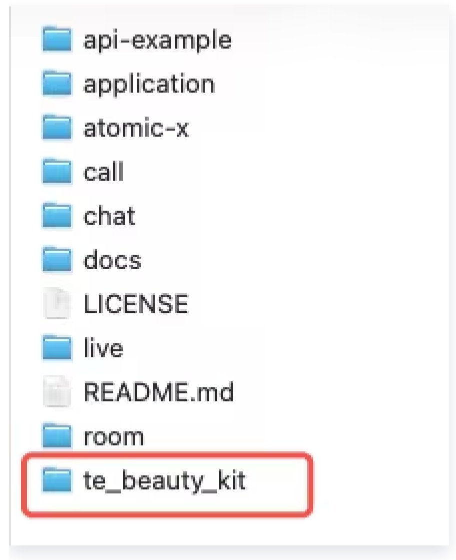
组件接入

`TUILiveKit` 默认支持基础美颜，无需单独接入。如果您需要使用高级美颜，请按照以下流程完成接入工作。

> ⚠ **注意：**
> 高级美颜功能需要单独付费购买套餐，详情请参见 腾讯特效 SDK 。

步骤 1：集成 te_beauty_kit 适配模块

下载并拷贝文件：从 GitHub 下载 TUIKit_Flutter 源码，将 `te_beauty_kit` 文件夹拷贝到您的工程，使其与 app 文件夹同级。



然后，在您的 Flutter 工程的 pubspec.yaml 文件里添加 `te_beauty_kit` 依赖：

```
dependencies:
    te_beauty_kit:
      path: ../te_beauty_kit # 请使用您的真实路径
```

> ⚠ **注意：**
> 后续接入流程，需要使用您购买的美颜套餐。为了简化流程，将以美颜套餐 S1-07 为例，帮助您快速接入完整流程。组件接入完成之后，请按照 更换套餐，完成您实际套餐的更换工作。

步骤 2：集成美颜面板资源（必做）

美颜面板资源

美颜面板资源主要包括 2 部分：

- 面板配置素材：面板配置 JSON，相关图片等，用于展示美颜特效功能项。

资源	用途

`assets/beauty_panel/*.json`	面板 UI 配置（分组、按钮、滑杆）。
`assets/beauty_panel/panel_icon/`	面板按钮图标 PNG。

- 美颜特效素材：滤镜 LUT、2D/3D 动效贴纸等具体特效的素材。

资源	用途	对应面板能力
`lut/`	滤镜调色查找表。	滤镜 Tab
`MotionRes/2dMotionRes/`	2D 动态贴纸 / 大头特效。	贴纸 Tab
`MotionRes/3dMotionRes/`	3D 模型贴纸。	贴纸 Tab
`MotionRes/handMotionRes/`	手势触发贴纸。	手势 Tab
`MotionRes/makeupRes/`	整妆套装（口红 + 眼影 + 腮红等）。	美妆 Tab
`MotionRes/light_makeup/`	轻美妆素材。	轻美妆 Tab
`MotionRes/segmentMotionRes/`	人像分割（绿幕 / 抠图 / 背景虚化）。	分割 Tab

集成美颜面板资源

1. 为便于快速集成，请从 GitHub 下载 TUIKit_Flutter ，从这里获取默认套餐 S1-07 的资源。
2. **集成面板配置素材**：复制 `TUIKit_Flutter/application/assets/beauty_panel` 到您的 flutter 工程的 assets 目录。

```
application/assets/
├── beauty_panel/                # 面板 JSON 配置 + 图标
│   ├── beauty.json
│   ├── beauty_template.json
│   ├── beauty_template_ios.json
│   ├── beauty_image.json
│   ├── beauty_shape.json
│   ├── beauty_makeup.json
│   ├── beauty_body.json
```

```
│   ├── light_makeup.json
│   ├── makeup.json
│   ├── motion_2d.json
│   ├── motion_3d.json
│   ├── motion_gesture.json
│   ├── segmentation.json
│   ├── lut.json
│   └── panel_icon/                 # 面板内的所有 PNG 图标
```

在您的 Flutter 工程的 `pubspec.yaml` 文件里，添加 assets 资源配置：

```
flutter:
  assets:
    - assets/beauty_panel/
    - assets/beauty_panel/panel_icon/beauty/
    - assets/beauty_panel/panel_icon/beauty/beauty_smooth/
    - assets/beauty_panel/panel_icon/beauty/eyes_makeup/
    - assets/beauty_panel/panel_icon/beauty/kouhong/
    - assets/beauty_panel/panel_icon/beauty/liti/
    - assets/beauty_panel/panel_icon/beauty/saihong/
    - assets/beauty_panel/panel_icon/beauty_body/
    - assets/beauty_panel/panel_icon/beauty_template/
    - assets/beauty_panel/panel_icon/light_makeup_icon/
    - assets/beauty_panel/panel_icon/lut_icon/
    - assets/beauty_panel/panel_icon/motions_icon/
```

3. **集成美颜特效素材**，Android 和 iOS 需要分别集成：

Android

复制 `TUIKit_Flutter/application/android/app/src/main/assets` 下的 `lut` 和 `MotionRes` 2 个目录，到 Android app 的 assets 里。目录如下：

```
android/app/src/main/assets/
├── lut/                        # 滤镜 LUT 调色图
│   ├── baixi_lf.png
│   ├── dongjing_lf.png
│   ├── moren_lf.png
│   ├── xindong_lf.png
```

```
│   └── ziran_lf.png
└── MotionRes/                  # 动效 / 美妆 / 分割素材
    ├── 2dMotionRes/            # 2D 动效贴纸
    ├── 3dMotionRes/            # 3D 动效贴纸
    ├── handMotionRes/          # 手势贴纸
    ├── ganMotionRes/           # 一镜到底动效
    ├── makeupRes/              # 整妆套装
    ├── light_makeup/           # 轻美妆
    └── segmentMotionRes/       # 人像分割 / 绿幕
```

iOS

复制 `TUIKit_Flutter/application/ios/xmagic` 目录，到 iOS app 的根目录里。目录如下：

```
ios/                                  # 你的 ios 工程根目录
└── xmagic/
    ├── lut.bundle/                   # 滤镜 LUT 调色图
    ├── 2dMotionRes.bundle/           # 2D 动效贴纸素材
    ├── 3dMotionRes.bundle/           # 3D 动效贴纸素材
    ├── makeupMotionRes.bundle/       # 美妆素材
    ├── ganMotionRes.bundle/          # 一镜到底动效
    ├── handMotionRes.bundle/         # 手势贴纸
    ├── lightMakeupRes.bundle/        # 轻美妆素材
    └── segmentMotionRes.bundle/      # 人像分割 / 绿幕素材
```

然后，把整个 `xmagic/` 目录拖入 Xcode 工程，**选择** `Create folder references` **（蓝色文件夹），不要使用 group**，并确保 Target Membership 勾选了主 App。

4. **套餐与 panel JSON 对照表**：必须把所选 `BeautyLevel` 对应的所有 JSON 都导入，否则相关 Tab 会缺失或空白。

套餐（`BeautyLevel`）	需要导入的 panel JSON（全集）
`A1_00`	`beauty_template`，`beauty`，`lut`

A1_01	beauty_template , beauty , beauty_image , beauty_base_shape , lut
A1_02	beauty_template , beauty , beauty_image , beauty_base_shape , lut , motion_2d
A1_03	beauty_template , beauty , beauty_image , beauty_general_shape , lut , motion_2d
A1_04	beauty_template , beauty , beauty_image , beauty_general_shape , lut
A1_05	beauty_template , beauty , beauty_image , beauty_general_shape , lut , motion_2d , segmentation
A1_06	beauty_template , beauty , beauty_image , beauty_general_shape , lut , motion_2d , makeup
S1_00	beauty_template , beauty , beauty_image , beauty_shape , beauty_makeup , lut
S1_01	beauty_template , beauty , beauty_image , beauty_shape , beauty_makeup , lut , motion_2d , motion_3d , makeup , light_makeup
S1_02	beauty_template , beauty , beauty_image , beauty_shape , beauty_makeup , lut , motion_2d , motion_3d , motion_gesture , makeup , light_makeup
S1_03	beauty_template , beauty , beauty_image , beauty_shape , beauty_makeup , lut , motion_2d , motion_3d , makeup , light_makeup , segmentation
S1_04	beauty_template , beauty , beauty_image , beauty_shape , beauty_makeup , lut , motion_2d , motion_3d , motion_gesture , makeup , light_makeup , segmentation
S1_05	beauty_template , beauty , beauty_image , beauty_shape , beauty_makeup , lut , beauty_body , motion_2d , motion_3d , makeup , light_makeup , segmentation
S1_06	beauty_template , beauty , beauty_image , beauty_shape , beauty_makeup , lut , beauty_body , motion_2d , motion_3d , motion_ges

	ture, makeup, light_makeup, segmentation
S1_07	beauty_template, beauty, beauty_image, beauty_shape, beauty_makeup, lut, beauty_body, motion_2d, motion_3d, motion_gesture, makeup, light_makeup, segmentation

步骤 3：工程配置

1. 在您的 Flutter 工程的 pubspec.yaml 文件里添加相关配置：

```
# pubspec.yaml

# 根级配置
EffectPlayer:
  sub_spec: 'NoXMagic'
TencentEffect:
  te_sub_spec: 'S1-07'
```

> **说明：**
> 1. **EffectPlayer.sub_spec**：可选值是 Default 、 NoXMagic 。腾讯特效播放器和腾讯特效美颜这 2 个库如果一起使用，会出现 xmagic 库编译冲突问题。为了解决该问题，你可以设置 NoXMagic 方式，只编译一份 xmagic 库即可。 TUILiveKit 默认接入了礼物组件 live_uikit_gift ，而 live_uikit_gift 接入腾讯特效播放器。
> 2. **TencentEffect.te_sub_spec**：设置您的套餐类型，更多类型请查看 步骤 2 第 4 步。

- **iOS 平台**：如果 **EffectPlayer.sub_spec** 设置为 NoXMagic ，iOS app 的 Podfile 文件中禁用 YTCommonXMagic 。因为腾讯特效美颜默认会导入该库。

```
# 注释 YTCommonXMagic 库
# pod 'YTCommonXMagic', :podspec => 'https://mediacloud-76607.gzc.vod.tencent-cloud.com/MediaX/iOS/podspec/release/YTCommonXMagic_1.3.1/YTCommonXMagic.podspec'
```

- **Android 平台**：确保根目录 settings.gradle 的 dependencyResolutionManagement 中包含腾讯 Maven 源（ tebeautykit 模块依赖的腾讯特效 SDK 从该源拉取）：

```
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
```

版权所有：腾讯云计算（北京）有限责任公司

```
    repositories {
        google()
        mavenCentral()
        // 新增腾讯 Maven 源
        maven { url
'https://mirrors.tencent.com/repository/maven/thirdparty' }
    }
}
```

2. Android 和 iOS 原生层注册美颜处理器：

Android

```
package com.tencent.application

import android.os.Bundle
import android.os.PersistableBundle
import io.flutter.embedding.android.FlutterActivity
// 导入相关文件
import com.tencent.trtcplugin.TRTCPlugin
import com.tencent.effect.tencent_effect_flutter.XmagicProcesserFactory

class MainActivity: FlutterActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // 注册美颜处理器
        TRTCPlugin.setBeautyProcesserFactory(XmagicProcesserFactory())
    }
}
```

iOS

```
import Flutter
import UIKit
// 导入相关文件
import tencent_effect_flutter
import tencent_rtc_sdk
```

版权所有：腾讯云计算（北京）有限责任公司

```
@main
@objc class AppDelegate: FlutterAppDelegate {
  override func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?
  ) -> Bool {
    GeneratedPluginRegistrant.register(with: self)

    // 注册美颜处理器
    let instance = XmagicProcesserFactory()
    TencentRTCCloud.setBeautyProcesserFactory(factory: instance)

    return super.application(application, didFinishLaunchingWithOptions:
launchOptions)
  }
}
```

步骤 4：鉴权和初始化

1. 在 `main.dart` 配置 `te_beauty_kit` 模块的国际化：

```
//
// main.dart
//

// 导入文件
import 'package:te_beauty_kit/te_beauty_kit.dart';

// 您自己的APP主类
class XXX extends StatelessWidget {
  const XXX({super.key});

 @override
  Widget build(BuildContext context) {
    return MaterialApp(
      ......
      localizationsDelegates: [
      ...LiveKitLocalizations.localizationsDelegates,
      // 新增 tebeautykit 模块的国际化
```

版权所有：腾讯云计算（北京）有限责任公司

```
        ...TEBeautyKitLocalizations.localizationsDelegates,
        ],
        ......
      );
    }
  }
```

2. 在 App 启动或登录时，设置 `License` 鉴权 和 美颜套餐类型：

```
// 导入文件
import 'package:te_beauty_kit/te_beauty_kit.dart';

// 初始化腾讯美颜
TUIBeautyKit.instance.init("YOUR_LICENSE_URL", "YOUR_LICENSE_KEY",
BeautyLevel.S1_07);
```

`BeautyLevel` 可选值（与购买的腾讯特效 SDK 套餐一一对应）：

```
A1_00  A1_01  A1_02  A1_03  A1_04  A1_05  A1_06
S1_00  S1_01  S1_02  S1_03  S1_04  S1_05  S1_06  S1_07
```

更新套餐

1. 请确认您购买的套餐类型，并联系腾讯特效 SDK 商务支持，获取对应的美颜面板资源（包括**面板配置素材**和**美颜特效素材**）。
2. 参考 集成美颜面板资源，更换**面板配置素材**和**美颜特效素材**。
3. 参考 工程配置，修改 `TencentEffect/te_sub_spec` 配置的套餐类型。
4. 参考 鉴权和初始化 第 2 步，修改鉴权时传入的套餐类型。

验证

完成以上步骤后，启动 App 进入开播预览页，点击美颜按钮，查看弹出的美颜面板。

常见问题

License 鉴权失败怎么办?

请检查 `LicenseUrl` / `LicenseKey` 是否与套餐绑定、所选 `BeautyLevel` 是否与 License 匹配，并确认应用的 `applicationId` 与 License 申请时绑定的包名一致。

版权所有：腾讯云计算（北京）有限责任公司

是否支持动态切换套餐?

不支持。需要编译阶段确定套餐版本，以便下载相关依赖库。