音视频通话 SDK

更多特性

腾讯云

文档目录

更多特性
设置昵称、头像（全平台）

最近更新时间：2024-04-23 16:27:52

本文介绍如何设置用户的头像和昵称。

设置头像、昵称

如果您需要自定义昵称或头像，可以使用如下接口进行更新：

Android（Kotlin）

```
TUICallKit.createInstance(context).setSelfInfo("jack",
"https:/****/user_avatar.png", callback)
```

Android（Java）

```
TUICallKit.createInstance(context).setSelfInfo("jack",
"https:/****/user_avatar.png", callback);
```

iOS（Objective-C）

```
[[TUICallKit createInstance] setSelfInfo:@"" avatar:@"" succ:^{

} fail:^(int code, NSString *errMsg) {

}];
```

iOS（Swift）

```
TUICallKit.createInstance().setSelfInfo(nickname: "", avatar: "") {

} fail: { code, message in

}
```

Flutter（Dart）

```
TUIResult result = TUICallKit.instance.setSelfInfo('userName',
'url:********');
```

uni-app(Andorid&iOS)

```
const options = {
    nickName: 'jack',
    avatar: 'https:/****/user_avatar.png'
}
TUICallKit.setSelfInfo(options, (res) => {
    if (res.code === 0) {
        console.log('setSelfInfo success');
    } else {
        console.log(`setSelfInfo failed, error message = ${res.msg}`);
    }
});
```

Web&H5

```
try {
  await TUICallKitServer.setSelfInfo({ nickName: "jack", avatar:
"http://xxx" });
} catch (error: any) {
  alert(`[TUICallKit] Failed to call the setSelfInfo API. Reason:
${error}`);
}
```

注意：

因为用户隐私限制，非好友之间的通话，被叫的昵称和头像更新可能会有延迟，一次通话成功后就会顺利更新。

设置分辨率、填充模式(Web)

最近更新时间：2024-03-29 15:01:21

本位介绍如何设置分辨率、填充模式。

设置分辨率、填充模式

TUICallKit 组件提供了若干个功能开关，可以根据需要选择开启或关闭，具体参见 TUICallKit 属性介绍 。

- `videoDisplayMode`：画面显示模式。
- `videoResolution`：通话分辨率。

React

```
import { VideoDisplayMode, VideoResolution } from "@tencentcloud/call-uikit-react";

<TUICallKit
    videoDisplayMode={VideoDisplayMode.CONTAIN}
    videoResolution={VideoResolution.RESOLUTION_1080P} />
```

Vue

```
import { VideoDisplayMode, VideoResolution } from "@tencentcloud/call-uikit-vue";

<TUICallKit
    :videoDisplayMode="VideoDisplayMode.CONTAIN"
    :videoResolution="VideoResolution.RESOLUTION_1080P" />
```

videoDisplayMode

画面显示模式 `videoDisplayMode` 属性有三个值：

- `VideoDisplayMode.CONTAIN`
- `VideoDisplayMode.COVER`
- `VideoDisplayMode.FILL`，默认值是 `VideoDisplayMode.COVER` 。

属性	值	描述

videoDisplay Mode	VideoDisplayMode.CONTAIN	• 优先保证视频内容全部显示。 • 视频尺寸等比缩放，直至视频窗口的一边与视窗边框对齐。 • 如果视频尺寸与显示视窗尺寸不一致，在保持长宽比的前提下，将视频进行缩放后填满视窗，缩放后的视频四周会有一圈黑边。
	VideoDisplayMode.COVER	• 优先保证视窗被填满。 • 视频尺寸等比缩放，直至整个视窗被视频填满。 • 如果视频长宽与显示窗口不同，则视频流会按照显示视窗的比例进行周边裁剪或图像拉伸后填满视窗。
	VideoDisplayMode.FILL	• 保证视窗被填满的同时保证视频内容全部显示，但是不保证视频尺寸比例不变。 • 视频的宽高会被拉伸至和视窗尺寸一致。

videoResolution

分辨率 `videoResolution` **有三个值：**

- `VideoResolution.RESOLUTION_480P`
- `VideoResolution.RESOLUTION_720P`
- `VideoResolution.RESOLUTION_1080P`，**默认值是** `VideoResolution.RESOLUTION_480P` 。

分辨率说明：

视频 Profile	分辨率（宽 × 高）	帧率（fps）	码率（kbps）
480p	640 × 480	15	900
720p	1280 × 720	15	1500
1080p	1920 × 1080	15	2000

常见问题：

- iOS 13&14不支持编码高于 720P 的视频，建议在这两个系统版本限制最高采集 720P。参考 iOS Safari 已知问题 case 12 。
- Firefox 不支持自定义视频帧率（默认为 30fps）。
- 受系统性能占用，摄像头采集能力和浏览器限制等因素的影响，视频分辨率、帧率、码率的实际值不一定能够完全匹配设定值，在这种情况下，浏览器会自动调整 Profile 尽可能匹配设定值。

USB 摄像头

最近更新时间：2024-09-09 17:02:41

本文介绍了在 Android 系统设备上连接 USB 摄像头后，如何与 `TUICallKit` 组件相结合，以实现音视频通话的功能。 目前该功能支持原生 Android TUICallkit 组件和 Flutter（Android）TUICallKit 组件。

集成效果

实物连接图	视频通话效果
	

环境准备

- Android 5.0（SDK API Level 21）及以上版本。
- Gradle 4.2.1 及以上的版本。

- Android 系统设备：手机、平板或其他定制设备。

步骤一：准备条件

1. 在使用腾讯云提供的 USB 摄像头功能前，您需要前往控制台，为应用开通音视频服务，购买群组通话版套餐。具体步骤请参见 开通服务 。
2. 本插件需要与 TUICallKit 组件结合使用，请先接入 TUICallKit 组件：
   - Android
   - Flutter

步骤二：集成组件

在工程的 app 目录下的 `build.gradle` 文件中，添加以下依赖代码：

```
implementation "io.trtc.uikit:usb-camera:latest.release"
```

完成上述步骤后，您可以实现在 `TUICallKit` 组件中使用外接摄像头进行视频通话。

常见问题

Android 9 和 Android 10 版本的手机上无法开启摄像头?

问题原因：Android 9 和 Android 10 上申请 USB 权限的时候会检查 Camera 的权限，但是 Android 框架层检查出现异常，即使给了摄像头权限，还是会检查失败，导致无法申请 USB 权限。
解决方式：将应用的 targetSdkVersion 设置为 27 或以下版本。

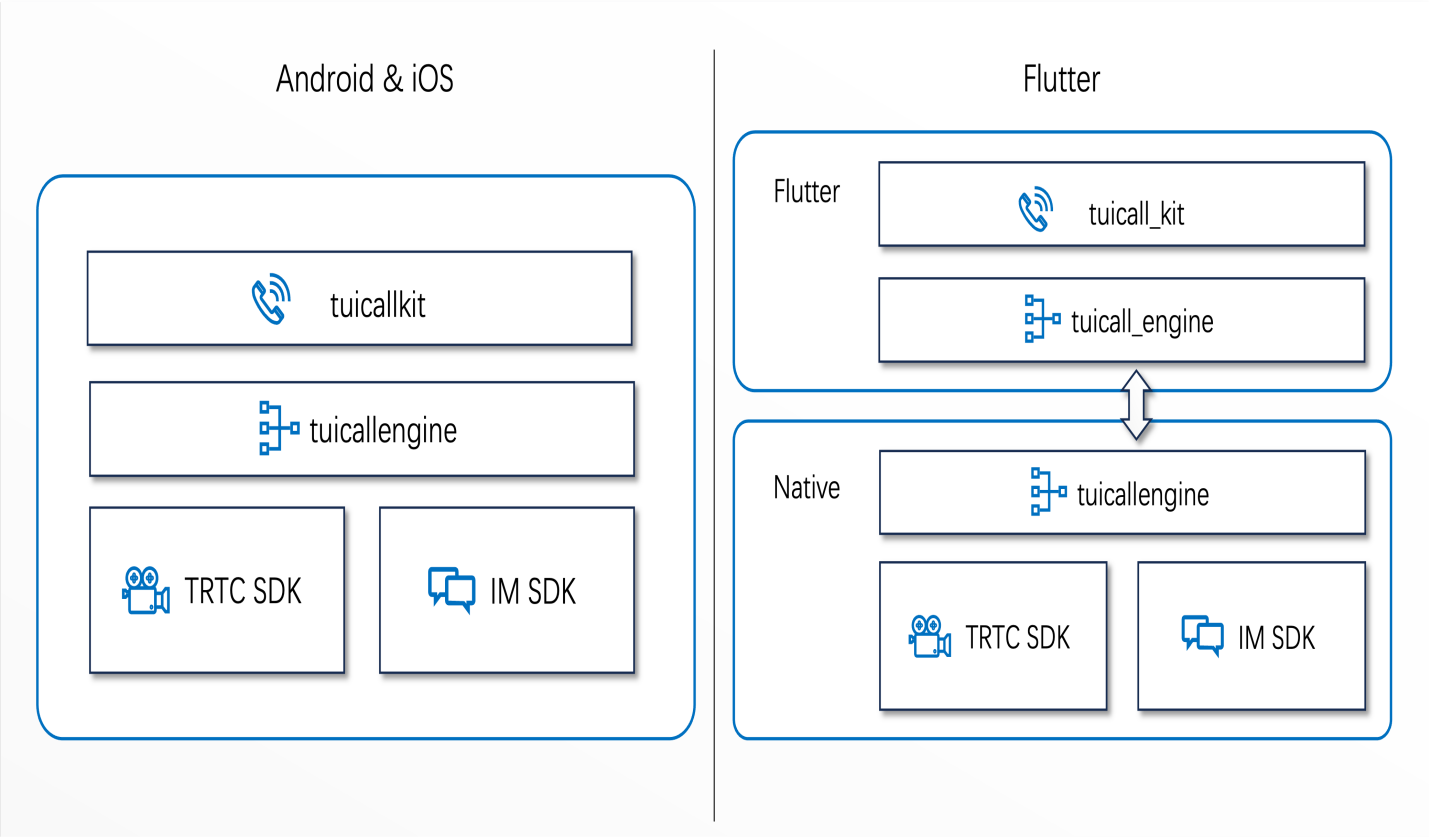
```
defaultConfig {
    targetSdkVersion 27
}
```

体积增量

最近更新时间：2024-12-26 17:28:22

本文将详细分析在基于不同开发框架的应用中集成 TUICallKit 可能带来的体积增量。

TUICallKit 组件结构



TUICallKit 体积增量分析

以 TUICallKit 2.6.0 为例，以下是不同开发框架集成之后各平台输出的应用的体积增量：

Android & iOS

系统	CPU 架构	增量
Android	arm64-v8a	16.2 MB
	armeabi-v7a	14.8 MB
iOS	arm64	15.5 MB

Flutter

系统	CPU 架构	增量
Android	arm64-v8a	14.9 MB
	armeabi-v7a	13.7 MB
iOS	arm64	15.5 MB

注意：

- 在 Android 平台的体积测试中，默认启用了 jniLibs 和 dex 的压缩功能。
- 本文档中的体积增量数据仅供参考。实际增量可能因插件版本、依赖项、构建配置、平台和资源文件等因素而有所波动。建议在集成插件后进行实际构建以获取准确数据。

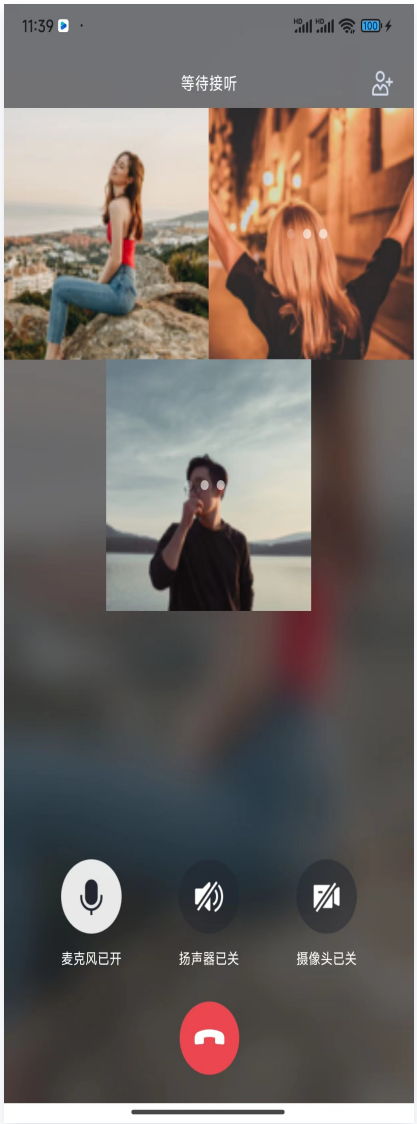
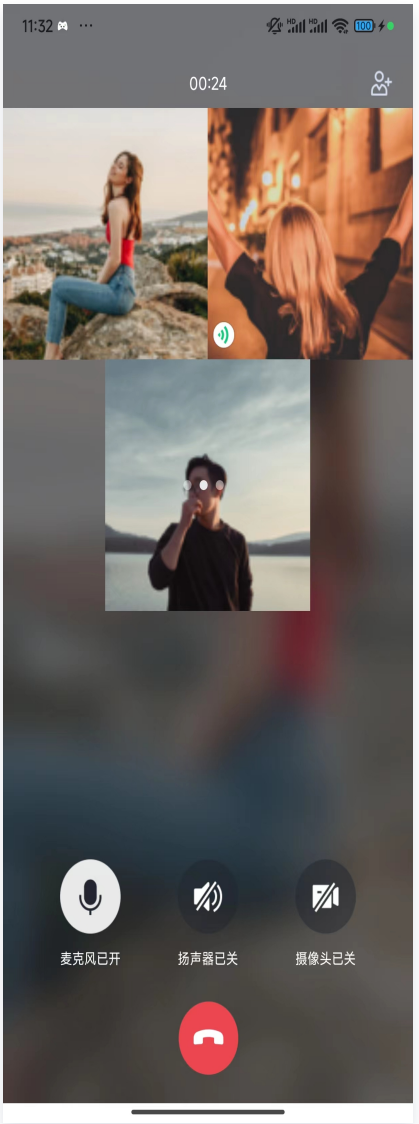
群组通话
Android&iOS&Flutter

最近更新时间：2024-03-29 15:01:22

本文介绍群组通话功能的使用，如发起群组通话、加入群组通话。

预期效果

TUICallKit 支持群组通话，预期效果见下图。

发起群组通话	收到群组通话请求	接受群组通话请求
		

创建 groupID

使用群组通话功能前，需要先创建群组，在已存在的群组中发起群组通话。

- 方式一：通过调用 IM API 创建群，详见 IM 群组管理 。
- 方式二：通过控制台手动创建群组，详见 控制台群组管理 。

群组通话

发起群组通话

调用 groupCall API 发起群通话。

Android（Kotlin）

```
TUICallKit.createInstance(context).groupCall("12345678",
Arrays.asList("jane", "mike", "tommy"), TUICallDefine.MediaType.Video)
```

Android（Java）

```
TUICallKit.createInstance(context).groupCall("12345678",
Arrays.asList("jane", "mike", "tommy"),TUICallDefine.MediaType.Video);
```

iOS（Swift）

```
import TUICallKit

TUICallKit.createInstance().groupCall(groupId: "12345678",
                                       userIdList: ["denny", "mike",
"tommy"],
                                       callMediaType: .video)
```

iOS（Objective-C）

```
#import <TUICallKit/TUICallKit.h>

[[TUICallKit createInstance] groupCall:@"12345678"
                            userIdList:@[@"denny", @"mike", @"tommy"]
                         callMediaType:TUICallMediaTypeVideo];
```

Flutter（Dart）

```
TUICallKit.instance.groupCall('0001', ['denny', 'mike', 'tommy'],
TUICallMediaType.video);
```

加入群组通话

调用 joinInGroupCall API 主动加入群组中已有的音视频通话。

Android（Kotlin）

```
RoomId roomId = RoomId();
roomId.intRoomId = 123321;
TUICallKit.createInstance(context).joinInGroupCall(roomId, "12345678",
TUICallDefine.MediaType.Video)
```

Android（Java）

```
RoomId roomId = new RoomId();
roomId.intRoomId = 123321;
TUICallKit.createInstance(context).joinInGroupCall(roomId , "12345678",
TUICallDefine.MediaType.Video);
```

iOS（Swift）

```
import TUICallKit
let roomId = TUIRoomId()
roomId.intRoomId = 123321
TUICallKit.createInstance().joinInGroupCall(roomId: roomId,
                                            groupId: "1234567",
                                            callMediaType: .video)
```

iOS（Objective-C）

```
#import <TUICallKit/TUICallKit.h>

TUIRoomId *roomId = [[TUIRoomId alloc] init];
```

```
roomId.intRoomId = 123321;

[[TUICallKit createInstance] roomId: roomId
                            groupId:@"223344"
                      callMediaType:TUICallMediaTypeVideo];
```

Flutter（Dart）

```
TUIRoomId roomId = TUIRoomId()
roomId.intRoomId = 123321;
TUICallKit.instance.joinInGroupCall(roomId, '1234567',
TUICallMediaType.video);
```

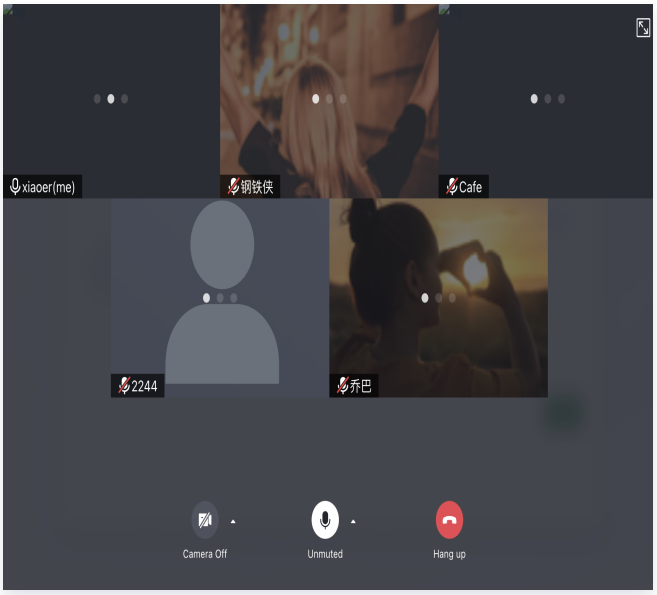
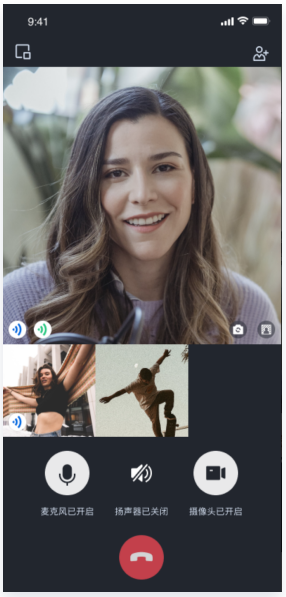
Web&H5&小程序

最近更新时间：2025-05-16 17:17:42

本文介绍群组通话功能的使用，如发起群组通话、加入群组通话。

预期效果

TUICallKit 支持群组通话，预期效果见下图。

Web 端	移动端
	

发起多人通话

调用 `calls` API 发起群通话。

```
try {
    const params = {
      userIDList: ['user1', 'user2'],
      type: TUICallType.VIDEO_CALL,
    }
  await TUICallKitServer.calls(params);
} catch (error: any) {
  alert(`[TUICallKit] groupCall failed. Reason:${error}`);
}
```

加入通话

调用 join API 主动加入已有的音视频通话。

```
try {
  await TUICallKitServer.join({callId: 'xxx'});
} catch (error: any) {
  alert(`[TUICallKit] join failed. Reason: ${error}`);
}
```

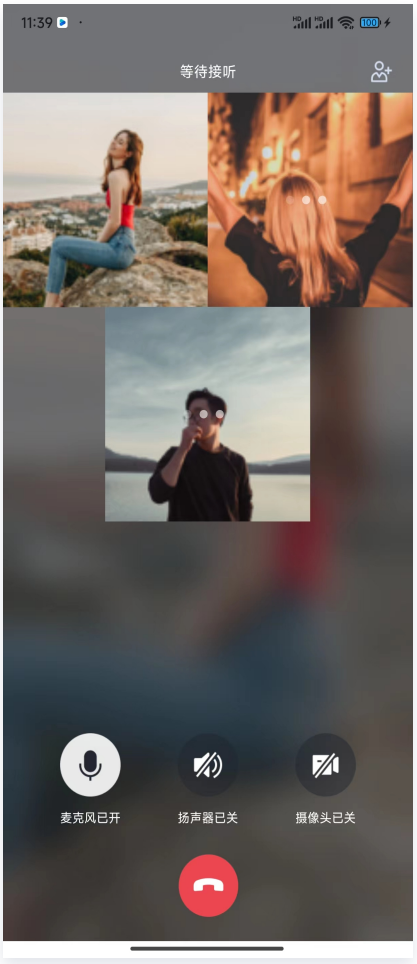
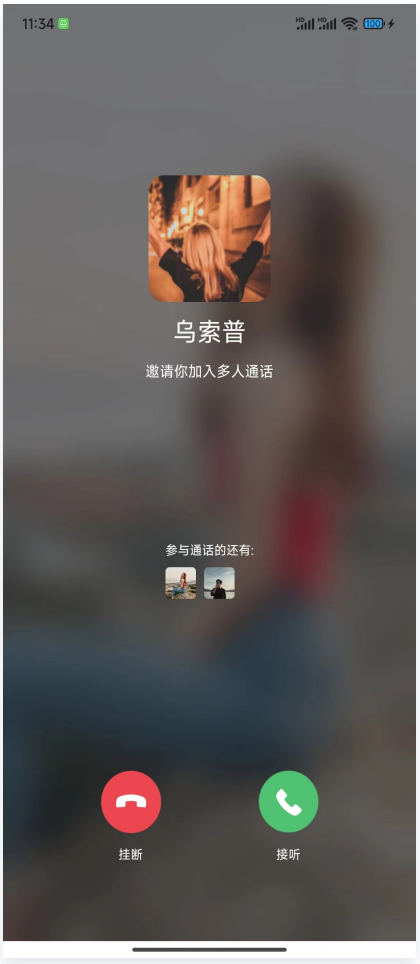
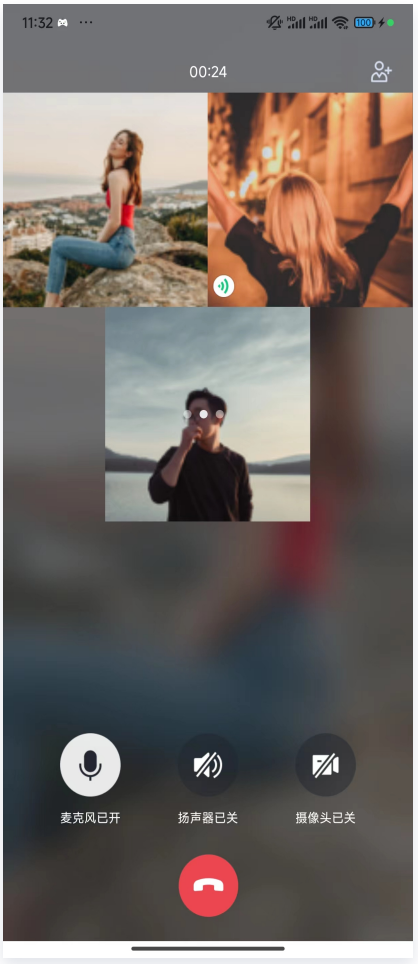
uni-app（客户端）

最近更新时间：2025-05-16 17:17:42

本文介绍群组通话功能的使用，如发起群组通话、加入群组通话。

预期效果

TUICallKit 支持群组通话，预期效果见下图。

发起群组通话	收到群组通话邀请	接受群组通话邀请
		

发起多人通话

调用 calls API 发起群通话。

```
const TUICallKit = uni.requireNativePlugin('TencentCloud-TUICallKit');
const options = {
  userIDList: ['mike', 'tom'],
```

```
  callMediaType: 1, // voice call(callMediaType = 1)、video
call(callMediaType = 2)
};
TUICallKit.calls(options, (res) => {
  if (res.code === 0) {
    console.log('call success');
  } else {
    console.log(`call failed, error message = ${res.msg}`);
  }
});
```

加入通话

调用 join API 主动加入群组中已有的音视频通话。

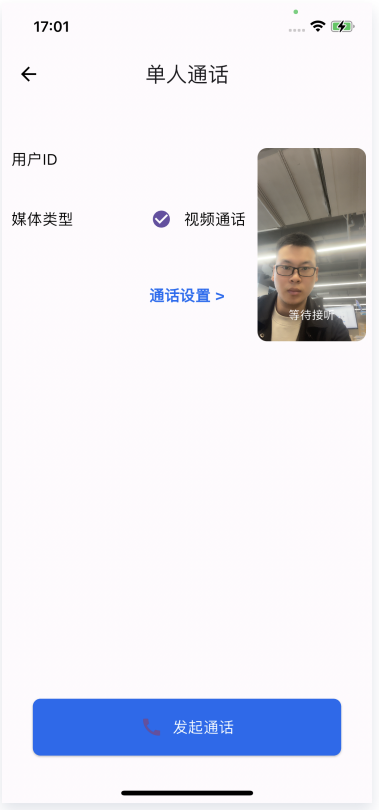
```
const TUICallKit = uni.requireNativePlugin('TencentCloud-TUICallKit');
const options = {
    callId: 'xxx',
};
TUICallKit.join(options, (res) => {
  if (res.code === 0) {
    console.log('join success');
  } else {
    console.log(`join failed, error message = ${res.msg}`);
  }
});
```

悬浮窗
Android&iOS&Flutter

最近更新时间：2024-03-29 15:01:22

本文介绍如何使用悬浮窗功能。

预期效果

开启悬浮窗按钮	语音通话悬浮窗	视频通话悬浮窗
		

悬浮窗功能

TUICallKit 允许用户在通话时使用通话界面左上角的悬浮窗按钮将通话界面缩小为悬浮窗。
您的业务如果需要启用该功能，可以使用 enableFloatWindow 方法，在 TUICallKit 组件初始化时开启该功能：

Android（Kotlin）

```
TUICallKit.createInstance(context).enableFloatWindow(true)
```

Android（Java）

```
TUICallKit.createInstance(context).enableFloatWindow(true);
```

iOS（Swift）

```
TUICallKit.createInstance().enableFloatWindow(true)
```

iOS（Objective-C）

```
[[TUICallKit createInstance] enableFloatWindow:YES];
```

Flutter（Dart）

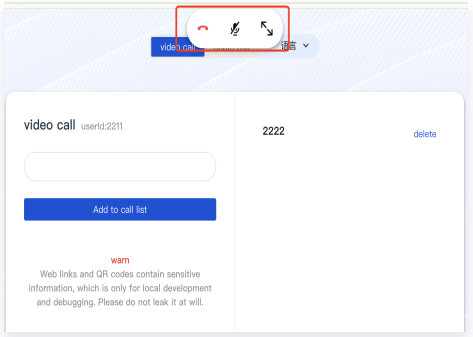
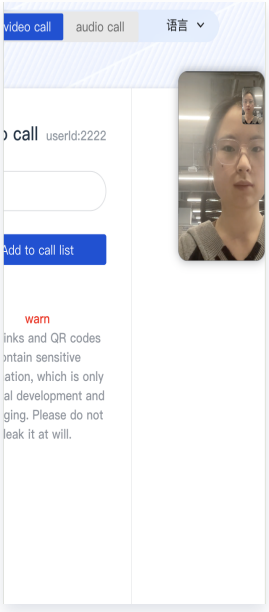
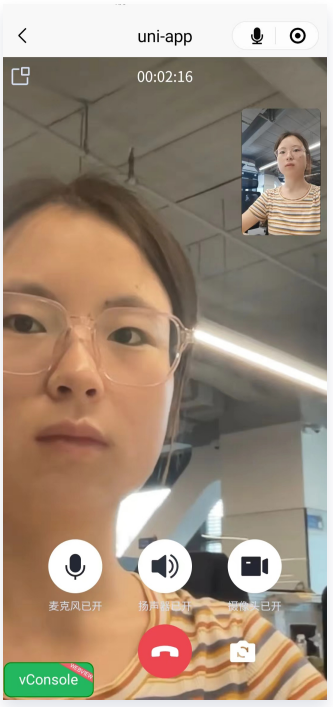
```
TUICallKit.instance.enableFloatWindow(true);
```

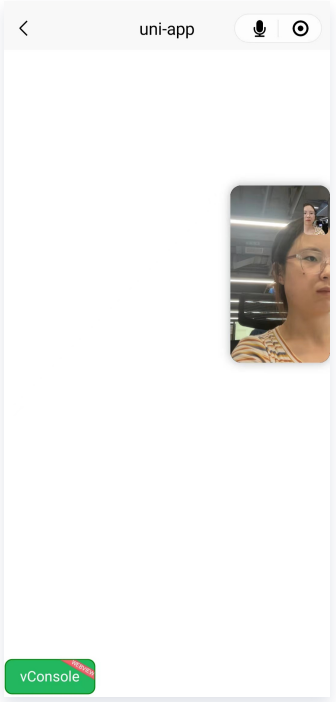
Web&H5&uni-app (小程序)

最近更新时间：2024-05-13 10:16:02

本文将介绍悬浮窗功能的使用。

预期效果

Web	H5	uni-app 小程序
		



悬浮窗功能

方式一：调用 `enableFloatWindow(enable: boolean)` **API 开启/关闭悬浮窗。**

> **说明：Vue&WeChat ⩾ v3.1.0 支持。**

```
TUICallKitServer.enableFloatWindow(true)
```

方式二：通过属性控制悬浮窗、全屏的开启/关闭。

- `allowedMinimized` **属性控制控制悬浮窗开启/关闭。**
- `allowedFullScreen` **属性控制全屏的开启/关闭。**

> **说明：WeChat 支持。**

React

```
<TUICallKit
      allowedMinimized={true}
      allowedFullScreen={true}
    />
```

Vue

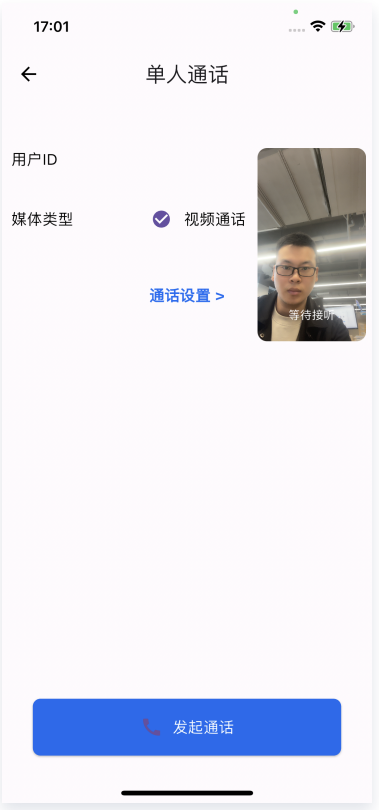
```
<TUICallKit
    :allowedMinimized="true"
    :allowedFullScreen="true"
    />
```

uni-app（客户端）

最近更新时间：2024-03-29 15:01:22

本文介绍如何使用悬浮窗功能。

预期效果

开启悬浮窗按钮	语音通话悬浮窗	视频通话悬浮窗
		

悬浮窗功能

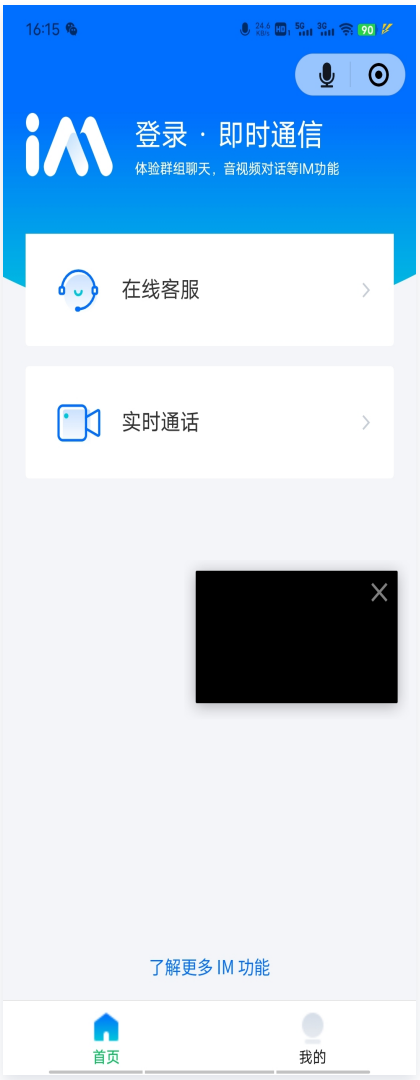
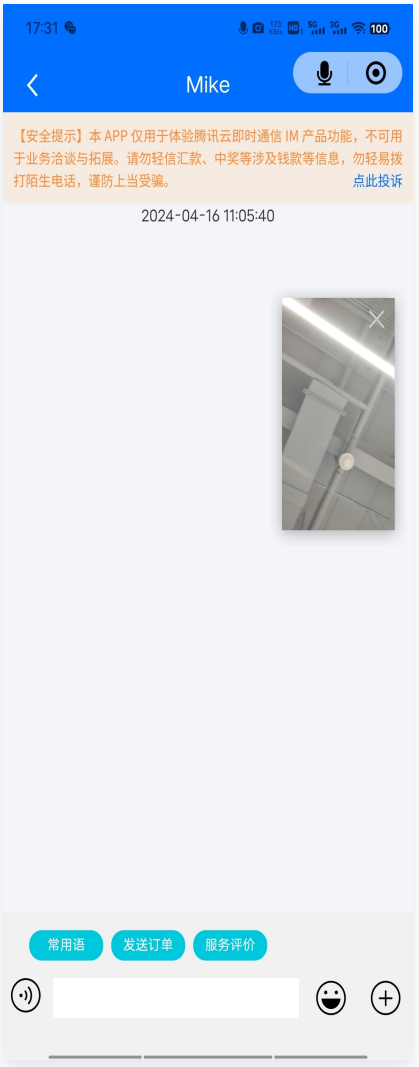
调用 `enableFloatWindow(enable: boolean)` API 开启/关闭悬浮窗。

```
const TUICallKit = uni.requireNativePlugin('TencentCloud-TUICallKit');
const enable = true;
TUICallKit.enableFloatWindow(enable);
```

原生小程序

最近更新时间：2024-04-23 16:27:51

预期效果

开启悬浮窗按钮	语音通话悬浮窗	视频通话悬浮窗
		

悬浮窗功能

调用 `enableFloatWindow(enable: boolean)` API 开启/关闭悬浮窗。

> **说明：3.0.0 > WeChat ≥ v2.1.3 支持**

```
TUICallKitServer.enableFloatWindow(true)
```

警告：

1. 群组通话/插件接入暂不支持悬浮窗功能。
2. 由于 live-pusher 底层限制，视频通话下的悬浮窗显示本地用户。
3. 悬浮窗 UI 无法自定义，音频通话下的悬浮窗显示黑框。
4. 需要配合全局监听使用。

美颜特效

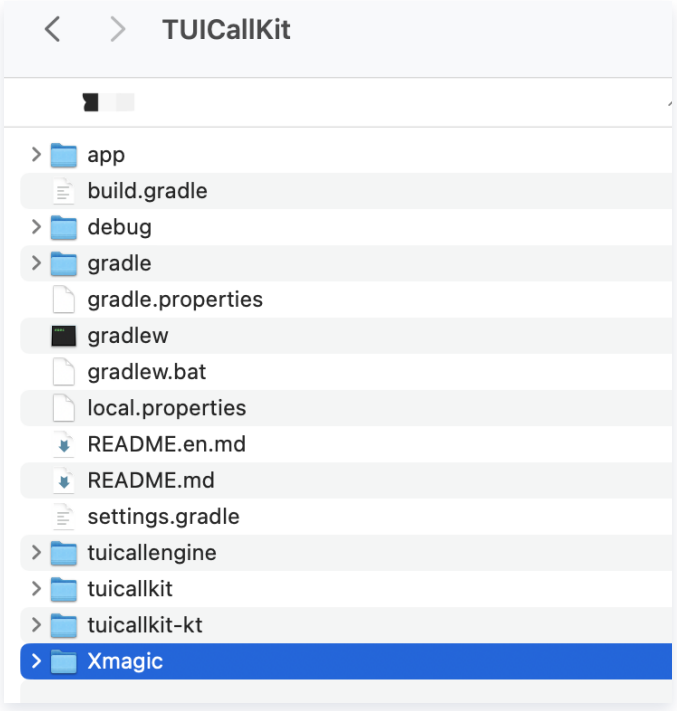
Android

最近更新时间：2023-07-14 09:23:31

TUICallEngine 中提供了 getTRTCCloudInstance() 接口，可以通过该接口调用 TRTC 的高级特性 setLocalVideoProcessListener()，实现接入第三方美颜。
本文将介绍如何在 TUICallKit 中接入 腾讯特效 SDK，**其他第三方美颜接入方法类似，请结合第三方 SDK 文档进行接入。**

接入准备

按照 腾讯特效 TRTC 接入指引，下载**场景化 Demo 的 zip 包**，按照文档替换资源，将 `Xmagic` 目录复制到自己的工程中，完成 步骤一；导入后的目录结构如下：



导入 Xmagic

1. 在工程根目录下找到 `settings.gradle` 文件，在其中增加如下代码，将 Xmagic 组件导入到您当前的项目中。

```
include ':Xmagic'
```

2. 在 tuicallkit 目录下找到 `build.gradle` 文件，并在其中增加如下代码，声明 tuicallkit 对新加入的 Xmagic 组件的依赖。

```
api project(':Xmagic')
```

3. 修改 Xmagic 目录下 `build.gradle` 中的 compileSdkVersion、minSdkVersion、TRTC SDK 版本、Glide 版本等与您的工程保持一致；依赖的 Xmagic SDK 版本为您的套餐包版本。

```
api 'com.tencent.liteav:LiteAVSDK_TRTC:11.1.0.13111'
api 'com.github.bumptech.glide:glide:4.12.0'

api 'com.tencent.mediacloud:TencentEffect_S1-04:3.0.0.13'
```

4. 修改 `XMagicImpl.java` 文件，替换自己的 `authKey` 和 `authLicenceUrl`。

```
public static final String authKey = YOUR_LICENSE_KEY;
public static final String authLicenceUrl = YOUR_LICENSE_URL;
```

初始化 Xmagic

Xmagic 提供了美颜面板供使用者调用，TUICallKit 集成的时候，可以在 BaseCallActivity 文件中增加布局，每次拉起通话界面时展示，也可以在显示出通话界面后以 Dialog 的形式展示，注意生命周期即可。可参考 腾讯特效 TRTC 接入指引 中 Demo 程的 `ThirdBeautyActivity` 类。

1. 授权：

```
//鉴权注意事项及错误码详情，请参考
https://cloud.tencent.com/document/product/616/65891#.E6.AD.A5.E9.AA.A
4.E4.B8.80.EF.BC.9A.E9.89.B4.E6.9D.83
XMagicImpl.init(getApplicationContext());

if
(TUICallDefine.MediaType.Video.equals(TUICallingStatusManager.sharedIn
stance(getApplicationContext()).getMediaType())) {
    if (XmagicLoadAssetsView.isCopyedRes) {
        initXMagic();
        return;
    }

    XMagicImpl.checkAuth((errorCode, msg) -> {
        if (errorCode == TELicenseCheck.ERROR_OK) {
            showLoadResourceView();
        } else {
            Log.e("Xmagic", "please check url and key" + errorCode +
" , " + msg);
```

```
        }
    });
  }
```

2. 初始化素材：

```
private void showLoadResourceView() {
     if (XmagicLoadAssetsView.isCopyedRes) {
         XmagicResParser.parseRes(getApplicationContext());
         initXMagic();
     } else {
         loadAssetsView = new XmagicLoadAssetsView(this);
         loadAssetsView.setOnAssetsLoadFinishListener(() -> {
             XmagicResParser.parseRes(getApplicationContext());
             initXMagic();
         });
     }
 }
```

设置纹理回调和添加美颜效果

```
//1. set TRTCVideoFrameListener, refer to:
{https://liteav.sdk.qcloud.com/doc/api/zh-
cn/group__TRTCCloud__android.html#a0b565dc8c77df7fb826f0c45d8ad2d85}
TRTCCloud trtcCloud =
TUICallEngine.createInstance(this).getTRTCCloudInstance();
trtcCloud.setLocalVideoProcessListener(TRTCCloudDef.TRTC_VIDEO_PIXEL_F
ORMAT_Texture_2D,
        TRTCCloudDef.TRTC_VIDEO_BUFFER_TYPE_TEXTURE, new
TRTCCloudListener.TRTCVideoFrameListener() {
            @Override
            public void onGLContextCreated() { //2. GLContext created
              if (isGLContextCreated) {
                  isGLContextCreated = false;
              } else {
                  runOnUiThread(() -> initXMagic());
              }
            }
            @Override
            public int onProcessVideoFrame(TRTCCloudDef.TRTCVideoFrame
srcFrame, TRTCCloudDef.TRTCVideoFrame dstFrame) {
```

```
            //3. invoke xmagic module, more details refer to
{https://liteav.sdk.qcloud.com/doc/api/zh-
cn/group__TRTCCloudListener__android.html#a22afb08b2a1a18563c7be28c904
b166a}
                if (mXMagic != null) {
                    dstFrame.texture.textureId =
mXMagic.process(srcFrame.texture.textureId, srcFrame.width,
srcFrame.height);
                }
                return dstFrame.texture.textureId;
            }
            @Override
            public void onGLContextDestory() {
            //4. GLContext destroyed
                if (mXMagic != null) {
                  mXMagic.onDestroy();
                }
            }
});
```

1. 布局中添加 SDK 美颜面板：

```
 <RelativeLayout
     android:layout_above="@+id/ll_edit_info"
     android:id="@+id/livepusher_bp_beauty_pannel"
     android:layout_width="match_parent"
     android:layout_height="wrap_content" />
```

2. 初始化面板：

```
   mBeautyPanelView = findViewById(R.id.livepusher_bp_beauty_pannel);

   private void initXMagic() {
       if (mXMagic == null) {
           mXMagic = new XMagicImpl(this, mBeautyPanelView);
       }else {
           mXMagic.onResume();
       }
   }
```

完成上述步骤后，即可在音视频通话中使用第三方美颜。

集成效果

下图是 TUICallKit 含 UI 组件集成腾讯特效 SDK 面板的效果：

常见问题

集成 Xmagic 后，动效、美妆、分割下内容怎么为空?

请先检查套餐是否包含该特效；如果套餐包满足，请检查 /asset/下的资源文件是否包含 MotionRes/2dMotionRes 等目录及文件。

iOS

最近更新时间：2023-07-21 16:04:02

TUICallEngine 中提供了 getTRTCCloudInstance() 接口，可以通过该接口调用 TRTC 的高级特性 setLocalVideoProcessDelegete:pixelFormat:bufferType: 实现接入第三方美颜。
本文将介绍如何在 TUICallKit 中接入 腾讯特效 SDK，**其他第三方美颜接入方法类似，请结合第三方 SDK 文档进行接入。**

导入组件

您可以使用 CocoaPods 导入组件，具体如下：
1. 在您的 `Podfile` 文件中添加以下依赖。

Objective-C

```
pod 'XMagic'
```

2. 执行以下命令，安装组件。

```
pod install
```

如果无法安装 XMagic 最新版本，执行以下命令更新本地的 CocoaPods 仓库列表。

```
pod repo update
```

3. 添加美颜资源到实际项目工程中：
   3.1 下载并解压对应套餐的 SDK 和美颜资源，将 **resources 文件夹下的除 LightCore.bundle、Light3DPlugin.bundle、LightBodyPlugin.bundle、LightHandPlugin.bundle、LightSegmentPlugin.bundle、audio2exp.bundle 以外的其它 bundle 资源**添加到实际工程中。
   3.2 在 Build Settings 中的 Other Linker Flags 添加 `-ObjC` 。
4. 将 `Bundle Identifier` 修改成与申请的测试授权一致。

授权

1. 申请授权，得到 LicenseURL 和 LicenseKEY，请参见 License 指引 。

> ⚠ **注意：**
> 正常情况下，只要 App 成功联网一次，就能完成鉴权流程，因此您**不需要**把 License 文件放到工程的工程目录里。但是如果您的 App 在从未联网的情况下也需要使用 SDK 相关功能，那么您可以把

> License 文件下载下来放到工程目录，作为保底方案，此时 License 文件名必须是 `v_cube.license` 。

2. 在相关业务模块的初始化代码中设置 URL 和 KEY，触发 license 下载，避免在使用前才临时去下载。也可以在 AppDelegate 的 didFinishLaunchingWithOptions 方法里触发下载。其中，LicenseURL 和 LicenseKey 是控制台绑定 License 时生成的授权信息。

```
[TELicenseCheck setTELicense:LicenseURL key:LicenseKey
completion:^(NSInteger authresult, NSString * _Nonnull errorMsg) {
    if (authresult == TELicenseCheckOk) {
        NSLog(@"鉴权成功");
    } else {
        NSLog(@"鉴权失败");
    }
}];
```

设置 SDK 素材资源路径

```
#import <XMagic/XMagic.h>

- (void)buildBeautySDK:(int)width and:(int)height texture:
(unsigned)textureID {
    NSString *beautyConfigPath =
[NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES) lastObject];
    beautyConfigPath = [beautyConfigPath
stringByAppendingPathComponent:@"beauty_config.json"];
    NSFileManager *localFileManager=[[NSFileManager alloc] init];
    BOOL isDir = YES;
    NSDictionary * beautyConfigJson = @{};

    if ([localFileManager fileExistsAtPath:beautyConfigPath
isDirectory:&isDir] && !isDir) {
        NSString *beautyConfigJsonStr = [NSString
stringWithContentsOfFile:beautyConfigPath encoding:NSUTF8StringEncoding
error:nil];
        NSError *jsonError;
        NSData *objectData = [beautyConfigJsonStr
dataUsingEncoding:NSUTF8StringEncoding];
        beautyConfigJson = [NSJSONSerialization
JSONObjectWithData:objectData
```

```
options:NSJSONReadingMutableContainers
                                                    error:&jsonError];
    }

    NSDictionary *assetsDict = @{@"core_name":@"LightCore.bundle",
                                 @"root_path":[[NSBundle mainBundle]
bundlePath],
                                 @"tnn_"
                                 @"beauty_config":beautyConfigJson
    };

    // 初始化SDK：width 和 height 分别是 texture 的宽高
    self.xMagicKit = [[XMagic alloc]
initWithRenderSize:CGSizeMake(width, height) assetsDict:assetsDict];
}
```

配置美颜各种效果（详细美颜效果配置请参考 美颜参数说明 ）

```
- (int)configPropertyWithType:(NSString *_Nonnull)propertyType withName:
(NSString *_Nonnull)propertyName withData:
(NSString*_Nonnull)propertyValue withExtraInfo:(id _Nullable)extraInfo;
```

进行渲染处理

TRTC SDK 设置第三方美颜的视频数据回调：设置该回调之后，TRTC SDK 会把采集到的视频帧通过您设置的 delegate 回调出来，用于第三方美颜组件进行二次处理。

```
#import <TUICallEngine/TUICallEngine.h>

[[[TUICallEngine createInstance] getTRTCCloudInstance]
setLocalVideoProcessDelegete:self
pixelFormat:TRTCVideoPixelFormat_Texture_2D
bufferType:TRTCVideoBufferType_Texture];
```

在视频帧回调接口 onProcessVideoFrame:(TRTCVideoFrame *_Nonnull)srcFrame dstFrame:(TRTCVideoFrame *_Nonnull)dstFrame
中，构造 YTProcessInput 传入到 SDK 内做渲染处理。

```
#pragma mark - TRTCVideoFrameDelegate
```

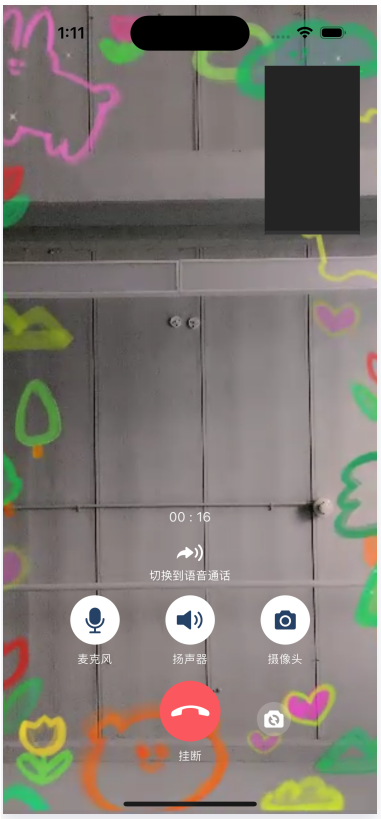
```
- (uint32_t)onProcessVideoFrame:(TRTCVideoFrame *_Nonnull)srcFrame
dstFrame:(TRTCVideoFrame *_Nonnull)dstFrame {
    if (!self.xMagicKit) {
        [self buildBeautySDK:srcFrame.width and:srcFrame.height
texture:srcFrame.textureId];
        self.heightF = srcFrame.height;
        self.widthF = srcFrame.width;
    }

    if(self.xMagicKit!=nil && (self.heightF!=srcFrame.height ||
self.widthF!=srcFrame.width)){
        self.heightF = srcFrame.height;
        self.widthF = srcFrame.width;
        [self.xMagicKit setRenderSize:CGSizeMake(srcFrame.width,
srcFrame.height)];
    }

    YTProcessInput *input = [[YTProcessInput alloc] init];
    input.textureData = [[YTTextureData alloc] init];
    input.textureData.texture = srcFrame.textureId;
    input.textureData.textureWidth = srcFrame.width;
    input.textureData.textureHeight = srcFrame.height;
    input.dataType = kYTTextureData;
    YTProcessOutput *output = [self.xMagicKit process:input
withOrigin:YtLightImageOriginTopLeft
withOrientation:YtLightCameraRotation0];
    dstFrame.textureId = output.textureData.texture;
    return 0;
}
```

集成效果

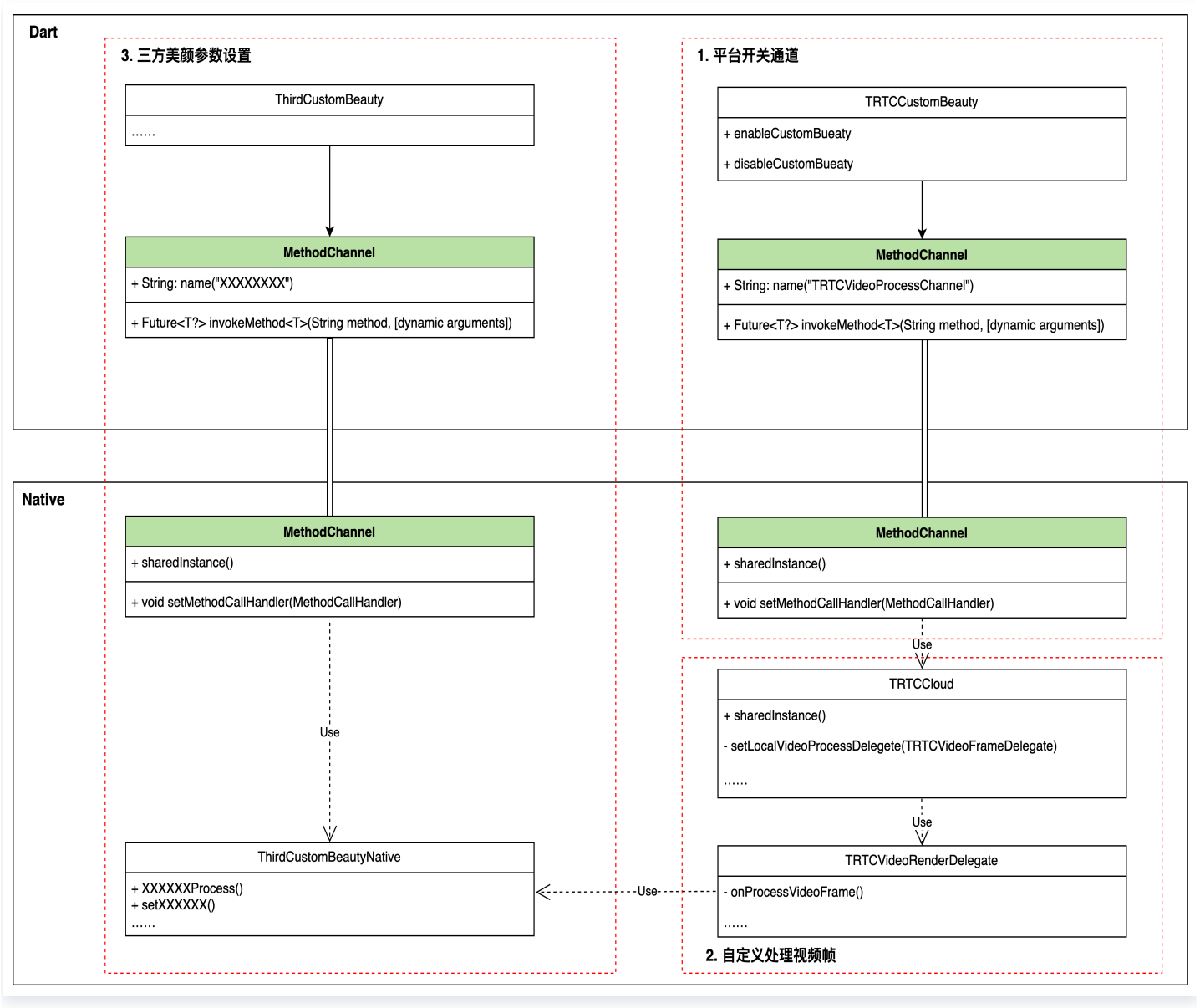
下图是 TUICallKit 含 UI 组件集成腾讯特效 SDK 的效果：


1:11
00 : 16
切换到语音通话
麦克风
扬声器
摄像头
挂断

Flutter

最近更新时间：2024-03-08 17:43:11

本篇文档主要介绍在 TUICallKit 中接入美颜特效的方法。
在 Flutter 中完成自定义美颜处理，需要通过 TRTC 自定义视频渲染完成。由于 Flutter 不善于进行大量实时数据传输的特性，涉及 TRTC 的自定义视频渲染部分需要在 Native 部分完成。 具体的方案如下：



接入方案分为 3 步：
第一步：通过 MethodChannel 通道 开启/关闭 TRTC 自定义渲染逻辑；
第二步 ： 在 TRTC 自定义渲染处理逻辑 onProcessVideoFrame() 中使用 美颜处理模块 对原始视频帧进行处理；
第三步：客户的美颜处理模块也需要在 Dart 通过接口设置当前美颜的参数，客户可以通过 MethodChannel 的方式设置美颜参数。这部分客户可根据自己的需求和使用的美颜自行定制。

接入第三方美颜特效

步骤1：实现 Dart 层到 Native 的“开始/结束”美颜的控制接口

实现 Dart 层接口:

```
final channel = MethodChannel('TUICallKitCustomBeauty');

void enableTUICallKitCustomBeauty() async {
    await channel.invokeMethod('enableTUICallKitCustomBeauty');
}

void disableTUICallKitCustomBeauty() async {
    await channel.invokeMethod('disableTUICallKitCustomBeauty');
}
```

实现对应 Native 层接口:

java

```
public class MainActivity extends FlutterActivity {
    private static final String channelName = "TUICallKitCustomBeauty";

    private MethodChannel channel;

    @Override
    public void configureFlutterEngine(@NonNull FlutterEngine
flutterEngine) {
        super.configureFlutterEngine(flutterEngine);

        channel = new
MethodChannel(flutterEngine.getDartExecutor().getBinaryMessenger(),
channelName);
        channel.setMethodCallHandler(((call, result) -> {
            switch (call.method) {
                case "enableTUICallKitCustomBeauty":
                    enableTUICallKitCustomBeauty();
                    break;
                case "disableTUICallKitCustomBeauty":
                    disableTUICallKitCustomBeauty();
                    break;
                default:
```

```
                    break;
            }
            result.success("");
        }));
    }

    public void enableTUICallKitCustomBeauty() {

    }

    public void disableTUICallKitCustomBeauty() {

    }
}
```

swift

```
@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {
    var channel: FlutterMethodChannel?

    override func application(_ application: UIApplication,
                             didFinishLaunchingWithOptions
launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        GeneratedPluginRegistrant.register(with: self)

        guard let controller = window?.rootViewController as?
FlutterViewController else {
            fatalError("Invalid root view controller")
        }
        channel = FlutterMethodChannel(name: "TUICallKitCustomBeauty",
binaryMessenger: controller.binaryMessenger)
        channel?.setMethodCallHandler({ [weak self] call, result in
            guard let self = self else { return }
            switch (call.method) {
            case "enableTUICallKitCustomBeauty":
                self.enableTUICallKitCustomBeauty()
                break
            case "disableTUICallKitCustomBeauty":
                self.disableTUICallKitCustomBeauty()
                break
            default:
```

```
                break
            }
        })
        result(nil)
        return super.application(application,
didFinishLaunchingWithOptions: launchOptions)
    }

    func enableTUICallKitCustomBeauty() {

    }

    func disableTUICallKitCustomBeauty() {

    }
}
```

步骤2：在 Native 的 TRTC 自定义渲染逻辑中完成美颜处理

> ⚠ **注意：**
> **Android 在接入美颜过程中需要先依赖 LiteAVSDK_Professional，在Android工程的app/build.gradle中添加一下依赖：**
>
> ```
> dependencies{
>     api "com.tencent.liteav:LiteAVSDK_Professional:latest.release"
> }
> ```

java

```
void enableTUICallKitCustomBeauty() {
    TRTCCloud.sharedInstance(getApplicationContext()).

setLocalVideoProcessListener(TRTC_VIDEO_PIXEL_FORMAT_Texture_2D,
                        TRTC_VIDEO_BUFFER_TYPE_TEXTURE, new
VideoFrameListerer());

}

void disableTUICallKitCustomBeauty() {
    TRTCCloud.sharedInstance(getApplicationContext()).

setLocalVideoProcessListener(TRTC_VIDEO_PIXEL_FORMAT_Texture_2D,
```

```
                        TRTC_VIDEO_BUFFER_TYPE_TEXTURE, null);
}

class VideoFrameListerer implements
TRTCCloudListener.TRTCVideoFrameListener {
    private XXXBeautyModel  mBeautyModel =
XXXBeautyModel.sharedInstance();

    @Override
    public int onProcessVideoFrame(TRTCCloudDef.TRTCVideoFrame
trtcVideoFrame,
                                   TRTCCloudDef.TRTCVideoFrame
trtcVideoFrame1) {
        // 美颜处理逻辑
        mBeautyModel.process(trtcVideoFrame, trtcVideoFrame1);
        ……

        return 0;
    }

    @Override
    public void onGLContextCreated() {
    }

    @Override
    public void onGLContextDestory() {
    }
}
```

swift

```
let videoFrameListener: TRTCVideoFrameListener =
TRTCVideoFrameListener()

func enableTUICallKitCustomBeauty() {

TRTCCloud.sharedInstance().setLocalVideoProcessDelegete(videoFrameListen
er, pixelFormat: ._Texture_2D, bufferType: .texture)
}

func disableTUICallKitCustomBeauty() {
```

```
    TRTCCloud.sharedInstance().setLocalVideoProcessDelegete(nil,
pixelFormat: ._Texture_2D, bufferType: .texture)
}

class TRTCVideoFrameListener: NSObject, TRTCVideoFrameDelegate {
    let bueutyModel = XXXXBeautyModel.shareIntance()
    func onProcessVideoFrame(_ srcFrame: TRTCVideoFrame, dstFrame:
TRTCVideoFrame) -> UInt32 {
        // 美颜处理逻辑
        bueutyModel.onProcessVideoFrame(srcFrame, dstFrame)
        ……

        return 0
    }
}
```

步骤3：客户自定义第三方美颜参数控制逻辑

此部分您可以根据自己的需求和具体使用的美颜模块，参考 步骤1 种实现美颜参数的设置功能，具体实现根据具体使用情况决定。

接入腾讯美颜特效

腾讯美颜特效的接入方法也同样遵循上述方法，现在以腾讯美颜特效为例，详细介绍接入方法：

步骤1：美颜资源下载与集成

1. 根据您购买的套餐 下载 SDK 。
2. 添加文件到自己的工程中：

Android

1. 在 app 模块下找到 build.gradle 文件，添加您对应套餐的 maven 引用地址，例如您选择的是S1-04套餐，则添加如下：

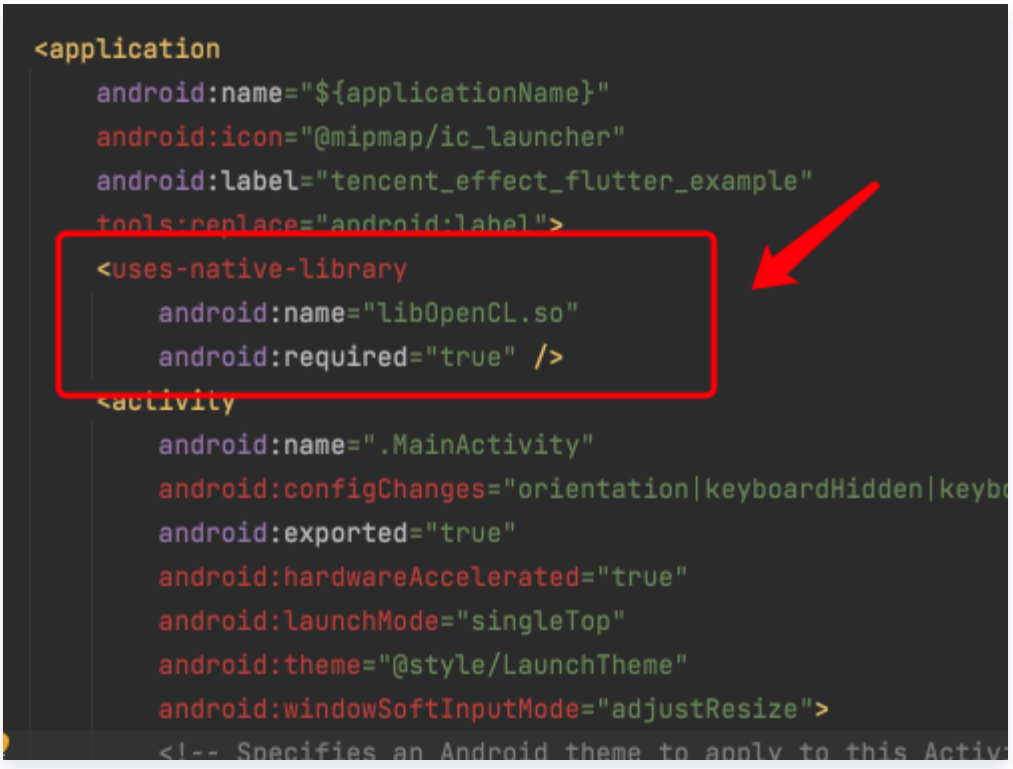
```
dependencies {
    implementation 'com.tencent.mediacloud:TencentEffect_S1-
04:latest.release'
}
```

各套餐对应的 maven 地址，请参见文档 。

2. 在 app 模块下找到 src/main/assets 文件夹，如果没有则创建，检查下载的 SDK 包中是否有 MotionRes 文件夹，如果有则将此文件夹拷贝到 `../src/main/assets` 目录下。
3. 在 app 模块下找到 AndroidManifest.xml 文件，在 application 表填内添加如下标签。

```
 <uses-native-library
        android:name="libOpenCL.so"
        android:required="true" />
        //此处的true 表示如果没有此库，则应用将无法正常运行。系统不允许在没有此库的设备上安装应用。
        //false表示应用可以使用此库（如果存在），但专门在没有此库的情况下运行（如果有必要）。系统允许安装应用，即使不存在此库也是如此。如果您使用 "false"，则需要自行负责妥善处理库不存在的情况。
        //Android 官网介绍：https://developer.android.com/guide/topics/manifest/uses-native-library-element
```

添加后如下图：



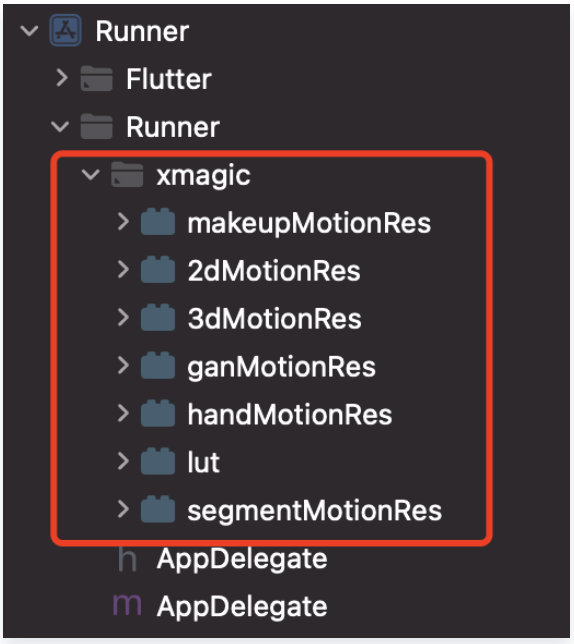
4. 混淆配置
   - 如果您在打 release 包时，启用了编译优化（把 minifyEnabled 设置为 true），会裁掉一些未在 java 层调用的代码，而这些代码有可能会被 native 层调用，从而引起 `no xxx method` 的异常。
   - 如果您启用了这样的编译优化，那就要添加这些 keep 规则，防止 xmagic 的代码被裁掉：

```
-keep class com.tencent.xmagic.** { *;}
```

```
-keep class org.light.** { *;}
-keep class org.libpag.** { *;}
-keep class org.extra.** { *;}
-keep class com.gyailib.**{ *;}
-keep class com.tencent.cloud.iai.lib.** { *;}
-keep class com.tencent.beacon.** { *;}
-keep class com.tencent.qimei.** { *;}
-keep class androidx.exifinterface.** { *;}
```

iOS

1. 添加美颜资源到您的工程。添加后如下图（您的资源种类跟下图不完全一致）：



2. 在 demo 中把 demo/lib/producer 里面的4个类：BeautyDataManager、BeautyPropertyProducer、BeautyPropertyProducerAndroid 和 BeautyPropertyProducerIOS 复制添加到自己的 Flutter 工程中，这4个类是用来配置美颜资源，把美颜类型展示在美颜面板中。

步骤2：引用 Flutter 版本 SDK

- **GitHub 引用**：在工程的 pubspec.yaml 文件中添加如下引用：

```
tencent_effect_flutter:
  git:
```

```
      url: https://github.com/TencentCloud/tencenteffect-sdk-flutter
```

- **本地引用：从 tencent_effect_flutter 下载最新版本的 tencent_effect_flutter，然后把文件夹 `android` 、 `ios` 、 `lib` 和文件 `pubspec.yaml` 、 `tencent_effect_flutter.iml` 添加到工程目录下，然后在工程的 pubspec.yaml 文件中添加如下引用：（可参考demo）**

```
tencent_effect_flutter:
    path: ../
```

tencent_effect_flutter 只是提供一个桥接，里面依赖的 XMagic 默认是最新版的，真正实现美颜是 XMagic。
如果要使用最新版本的美颜 SDK，您可以通过以下步骤进行 SDK 升级：

Android

在工程目录下执行命令： `flutter pub upgrade` 或者在 `subspec.yaml` 页面的右上角单击 Pub upgrade。

iOS

在工程目录下执行命令：flutter pub upgrade，然后在 iOS 目录下执行命令： `pod update` 。

步骤3：实现 Dart 层到 Native 的 开始/结束 美颜的控制接口

此部分可以参考 接入第三方美颜特效/步骤1 ，此处不在赘述。

步骤4：在 Native 的 TRTC 自定义渲染逻辑中完成美颜处理

Android

Android 在接入美颜过程中需要先依赖 LiteAVSDK_Professional， 在Android工程的 app/build.gradle中添加一下依赖：

```
dependencies{
    api "com.tencent.liteav:LiteAVSDK_Professional:latest.release"
}
```

```
void enableTUICallKitCustomBeauty() {
    TRTCCloud.sharedInstance(getApplicationContext()).
```

```
setLocalVideoProcessListener(TRTC_VIDEO_PIXEL_FORMAT_Texture_2D,
                        TRTC_VIDEO_BUFFER_TYPE_TEXTURE, new
VideoFrameListerer());

}

void disableTUICallKitCustomBeauty() {
    TRTCCloud.sharedInstance(getApplicationContext()).

setLocalVideoProcessListener(TRTC_VIDEO_PIXEL_FORMAT_Texture_2D,
                        TRTC_VIDEO_BUFFER_TYPE_TEXTURE, null);
}

class VideoFrameListerer implements
TRTCCloudListener.TRTCVideoFrameListener {
    private XXXBeautyModel  mBeautyModel =
XXXBeautyModel.sharedInstance();

    @Override
    public int onProcessVideoFrame(TRTCCloudDef.TRTCVideoFrame
trtcVideoFrame,
                                   TRTCCloudDef.TRTCVideoFrame
trtcVideoFrame1) {
       trtcVideoFrame1.texture.textureId =
XmagicApiManager.getInstance()
        .process(trtcVideoFrame.texture.textureId,
trtcVideoFrame.width, trtcVideoFrame.height);
        return 0;
    }

    @Override
    public void onGLContextCreated() {
    }

    @Override
    public void onGLContextDestory() {
    }
}
```

iOS

```
import TXLiteAVSDK_Professional
import tencent_effect_flutter

let videoFrameListener: TRTCVideoFrameListener =
TRTCVideoFrameListener()

func enableTUICallKitCustomBeauty() {

TRTCCloud.sharedInstance().setLocalVideoProcessDelegete(videoFrameLi
stener, pixelFormat: ._Texture_2D, bufferType: .texture)
}

func disableTUICallKitCustomBeauty() {
    TRTCCloud.sharedInstance().setLocalVideoProcessDelegete(nil,
pixelFormat: ._Texture_2D, bufferType: .texture)
}

class TRTCVideoFrameListener: NSObject, TRTCVideoFrameDelegate {
    func onProcessVideoFrame(_ srcFrame: TRTCVideoFrame, dstFrame:
TRTCVideoFrame) -> UInt32 {
        dstFrame.textureId =
GLuint(XmagicApiManager.shareSingleton().getTextureId(ConvertBeautyF
rame.convertTRTCVideoFrame(trtcVideoFrame: srcFrame)))
        return 0
    }
}


public class ConvertBeautyFrame: NSObject {
    public static func convertToTRTCPixelFormat(beautyPixelFormat:
ITXCustomBeautyPixelFormat) -> TRTCVideoPixelFormat {
        switch beautyPixelFormat {
        case .Unknown:
            return ._Unknown
        case .I420:
            return ._I420
        case .Texture2D:
            return ._Texture_2D
        case .BGRA:
            return ._32BGRA
```

```
        case .NV12:
            return ._NV12
        }
    }

    public static func convertTRTCVideoFrame(trtcVideoFrame:
TRTCVideoFrame) -> ITXCustomBeautyVideoFrame {
        let beautyVideoFrame = ITXCustomBeautyVideoFrame()
        beautyVideoFrame.data = trtcVideoFrame.data
        beautyVideoFrame.pixelBuffer = trtcVideoFrame.pixelBuffer
        beautyVideoFrame.width = UInt(trtcVideoFrame.width)
        beautyVideoFrame.height = UInt(trtcVideoFrame.height)
        beautyVideoFrame.textureId = trtcVideoFrame.textureId
        switch trtcVideoFrame.rotation {
        case ._0:
            beautyVideoFrame.rotation = .rotation_0
        case ._90:
            beautyVideoFrame.rotation = .rotation_90
        case ._180:
            beautyVideoFrame.rotation = .rotation_180
        case ._270:
            beautyVideoFrame.rotation = .rotation_270
        default:
            beautyVideoFrame.rotation = .rotation_0
        }
        switch trtcVideoFrame.pixelFormat {
        case ._Unknown:
            beautyVideoFrame.pixelFormat = .Unknown
        case ._I420:
            beautyVideoFrame.pixelFormat = .I420
        case ._Texture_2D:
            beautyVideoFrame.pixelFormat = .Texture2D
        case ._32BGRA:
            beautyVideoFrame.pixelFormat = .BGRA
        case ._NV12:
            beautyVideoFrame.pixelFormat = .NV12
        default:
            beautyVideoFrame.pixelFormat = .Unknown
        }
        beautyVideoFrame.bufferType =
ITXCustomBeautyBufferType(rawValue:
trtcVideoFrame.bufferType.rawValue) ?? .Unknown
        beautyVideoFrame.timestamp = trtcVideoFrame.timestamp
```

```
            return beautyVideoFrame
        }
    }
```

步骤5：开启美颜并设置美颜参数

完成上述配置后，可以通过 `enableTUICallKitCustomBeauty()/disableTUICallKitCustomBeauty()` 开启/关闭美颜。可以通过 腾讯特效美颜 Flutter 接口 设置美颜参数。

自定义铃声
Android

最近更新时间：2024-09-05 15:18:01

本文介绍如何替换 TUICallKit 的来电铃声，来电铃声分为**应用铃声**和**离线推送铃声**。

设置应用铃声

设置应用铃声有两种方式：

1、替换默认的铃声资源

如果您通过源码依赖 TUICallKit 组件，您可以替换 `res\raw` 文件夹下的音频文件来达到替换铃声的目的：

文件名	用途
phone_dialing.mp3	发起呼叫时的铃音
phone_ringing.mp3	接到呼叫时的铃音

2、通过 API 设置来电铃声

您也可以通过 setCallingBell 接口设置被叫端收到的来电铃声。

Kotlin

```
TUICallKit.createInstance(context).setCallingBell(filePath)
```

Java

```
TUICallKit.createInstance(context).setCallingBell(filePath);
```

设置静音模式

如果您的被叫端收到邀请，不需要响铃，您可以通过 enableMuteMode 设置静音模式。

Kotlin

```
TUICallKit.createInstance(context).enableMuteMode(true)
```

Java

```
TUICallKit.createInstance(context).enableMuteMode(true);
```

设置离线推送铃音

离线推送默认铃声是由系统决定的，可以在 手机设置 中，进入应用详情，查看或更改应用的通知铃声。
离线通知铃声仅支持以下厂商自定义：**华为、小米、FCM 和 APNs**，其他厂商暂不支持。离线推送铃声设置详见：

- 华为、小米、FCM（通知消息）自定义推送铃声 。
- FCM（数据消息）自定义来电铃声 。

iOS

最近更新时间：2024-09-05 15:18:01

本文介绍如何替换 TUICallKit 的来电铃声，来电铃声分为**应用铃声**和**离线推送铃声**。

设置应用铃声

设置应用铃声有两种方式：

1、替换默认的铃声资源

如果您通过源码依赖 TUICallKit 组件，您可以替换 `Resources\AudioFile` 文件夹下的音频文件来达到替换铃声的目的：

文件名	用途
phone_dialing.mp3	发起呼叫时的铃音
phone_ringing.mp3	接到呼叫时的铃音

2、通过 API 设置来电铃声

您也可以通过 setCallingBell 接口设置被叫端收到的来电铃声。

Swift

```
TUICallKit.createInstance().setCallingBell(filePath: " ")
```

Objective-C

```
[[TUICallKit createInstance] setCallingBell:@" "];
```

设置静音模式

如果您的被叫端收到邀请，不需要响铃，您可以通过 enableMuteMode 设置静音模式。

Swift

```
TUICallKit.createInstance().setCallingBell(enable: true)
```

Objective-C

```
[[TUICallKit createInstance] enableMuteMode: YES];
```

设置离线推送铃音

VoIP 推送不支持自定义推送铃声。APNs 推送可以在调用 call 接口拨打电话时设置 params 的 offlinePushInfo 中的 iOSSound 字段， iOSSound 传语音文件名。

⚠ 注意：

- 离线推送声音设置（仅对 iOS 生效），如果要自定义 iOSSound，需要先把语音文件链接进 Xcode 工程，然后把语音文件名（带后缀名）设置给 iOSSound。
- 铃声时长应小于30s。

Objective-C

```
[[TUICallKit createInstance] call:@"mike 的 id" params:[self
getCallParams] callMediaType:TUICallMediaTypeVideo];

- (TUICallParams *)getCallParams {
    TUIOfflinePushInfo *offlinePushInfo = [self createOfflinePushInfo];
    TUICallParams *callParams = [TUICallParams new];
    callParams.offlinePushInfo = offlinePushInfo;
    callParams.timeout = 30;
    return callParams;
}

+ (TUIOfflinePushInfo *)createOfflinePushInfo {
    TUIOfflinePushInfo *pushInfo = [TUIOfflinePushInfo new];
    pushInfo.title = @"";
    pushInfo.desc =
TUICallingLocalize(@"TUICallKit.have.new.invitation");
    pushInfo.iOSPushType = TUICallIOSOfflinePushTypeAPNs;
    pushInfo.ignoreIOSBadge = NO;
    pushInfo.iOSSound = @"phone_ringing.mp3";
    pushInfo.AndroidSound = @"phone_ringing";
    // OPPO必须设置ChannelID才可以收到推送消息，这个channelID需要和控制台一致
    // OPPO must set a ChannelID to receive push messages. This
channelID needs to be the same as the console.
```

```
    pushInfo.AndroidOPPOChannelID = @"tuikit";
    // FCM channel ID, you need change PrivateConstants.java and set
"fcmPushChannelId"
    pushInfo.AndroidFCMChannelID = @"fcm_push_channel";
    // VIVO message type: 0-push message, 1-System message(have a higher
delivery rate)
    pushInfo.AndroidVIVOClassification = 1;
    // HuaWei message type:
https://developer.huawei.com/consumer/cn/doc/development/HMSCore-
Guides/message-classification-0000001149358835
    pushInfo.AndroidHuaWeiCategory = @"IM";
    return pushInfo;
}
```

Swift

```
let params = TUICallParams()
let pushInfo: TUIOfflinePushInfo = TUIOfflinePushInfo()
pushInfo.title = "TUICallKit"
pushInfo.desc = "TUICallKit.have.new.invitation"
pushInfo.iOSPushType = .apns
pushInfo.ignoreIOSBadge = false
pushInfo.iOSSound = "phone_ringing.mp3"
pushInfo.androidSound = "phone_ringing"
// OPPO必须设置ChannelID才可以收到推送消息，这个channelID需要和控制台一致
// OPPO must set a ChannelID to receive push messages. This channelID
needs to be the same as the console.
pushInfo.androidOPPOChannelID = "tuikit"
// FCM channel ID, you need change PrivateConstants.java and set
"fcmPushChannelId"
pushInfo.androidFCMChannelID = "fcm_push_channel"
// VIVO message type: 0-push message, 1-System message(have a higher
delivery rate)
pushInfo.androidVIVOClassification = 1
// HuaWei message type:
https://developer.huawei.com/consumer/cn/doc/development/HMSCore-
Guides/message-classification-0000001149358835
pushInfo.androidHuaWeiCategory = "IM"
params.userData = "User Data"
params.timeout = 30
params.offlinePushInfo = pushInfo
```

```
TUICallKit.createInstance().call(userId: "123456", callMediaType:
.audio, params: params) {

} fail: {
    code, message in
}
```

Web&H5&小程序

最近更新时间：2024-04-03 15:47:11

本文介绍自定义铃声、静音来电铃声功能的使用。

自定义来电铃声

设置铃声接口

Web&H5 端	小程序端
• 仅限传入本地 MP3 格式的文件地址，确保该文件可以访问的。 • 如需恢复默认铃声 filePath 传空即可。 • 使用 ES6 import 方式引入铃声文件。	• 传入本地铃声文件应为相对当前小程序项目的绝对路径。 • uni-app 打包 Vue2 项目没有铃声，具体参见：Vue2 uni-app 打包微信小程序项目中，铃声没有声音?

说明： Vue ⩾ v3.0.0 支持

Web 端

```
import filePath from '../public/ring.mp3';

try {
  await TUICallKitServer.setCallingBell(filePath?: string);
} catch (error: any) {
  alert(`[TUICallKit] setCallingBell API failed. Reason: ${error}`);
}
```

小程序端

```
try {
  await TUICallKitServer.setCallingBell('/static/ring.mp3'); // 相对当前小程序项目的绝对路径
} catch (error: any) {
  alert(`[TUICallKit] setCallingBell API failed. Reason: ${error}`);
}
```

替换音频文件

如果您通过源码集成 TUICallKit 组件，通过替换铃声文件来达到替换铃声的目的。

- uni-app 小程序：替换 `'TUICallKit/src/TUICallService/assets/'` 文件。
- 微信小程序：替换 `'TUICallKit/static/'` 文件。

文件名	用途
phone_dialing.mp3	发起呼叫时的铃音
phone_ringing.mp3	接到呼叫时的铃音

静音来电铃声

- 开启/关闭来电铃声。
- 开启后，收到通话请求时，不会播放来电铃声。

> **说明：** Vue&WeChat ⩾ v3.1.2 支持

```
try {
  await TUICallKitServer.enableMuteMode(enable: boolean);
} catch (error: any) {
  alert(`[TUICallKit] enableMuteMode API failed. Reason: ${error}`);
}
```

uni-app（客户端）

最近更新时间：2024-04-03 15:47:11

本文介绍自定义铃声、静音来电铃声功能的使用。

自定义来电铃声

设置自定义来电铃音，这里仅限传入本地文件地址，需要确保该文件目录是应用可以访问的。

```
const TUICallKit = uni.requireNativePlugin('TencentCloud-TUICallKit');
const tempFilePath = './static/rain.mp3'; // Locally stored audio files
let musicFilePath = '';

uni.saveFile({
  tempFilePath: tempFilePath,
  success: (res) => {
    console.warn('保存文件成功 = ', JSON.stringify(res));
    musicFilePath =  res.savedFilePath;
    musicFilePath = plus.io.convertLocalFileSystemURL(musicFilePath);
    // Set ringtone
    TUICallKit.setCallingBell(musicFilePath, (res) => {
      if (res.code === 0) {
        console.log('setCallingBell success');
      } else {
        console.log(`setCallingBell failed, error message =
${res.msg}`);
      }
    });
```

静音来电铃声

- 开启/关闭来电铃声。
- 开启后，收到通话请求时，不会播放来电铃声。

```
const TUICallKit = uni.requireNativePlugin('TencentCloud-TUICallKit');
const enable = true;
TUICallKit.enableMuteMode(enable);
```

Flutter

最近更新时间：2024-09-05 15:18:01

本文介绍如何替换 TUICallKit 的来电铃声，来电铃声分为**应用铃声**和**离线推送铃声**。

设置应用铃声

设置应用铃声有两种方式：

1、替换默认的铃声资源

如果您通过源码依赖 TUICallKit 组件，您可以替换 **assets\audios** 文件夹下的音频文件来达到替换铃声的目的：

文件名	用途
phone_dialing.mp3	发起呼叫时的铃音
phone_ringing.mp3	接到呼叫时的铃音

2、通过 API 设置来电铃声

您也可以通过 setCallingBell 接口设置被叫端收到的来电铃声。

```
TUICallKit.instance.setCallingBell('flie path');
```

设置静音模式

如果您的被叫端收到邀请，不需要响铃，您可以通过 enableMuteMode 设置静音模式。

```
TUICallKit.instance.enableMuteMode(true);
```

设置离线推送铃音

1、iOS

VoIP 推送不支持自定义推送铃声。 APNs 推送可以修改 `call` 和 `groupcall` 接口中的参数 `params` 中的 `TUIOfflinePushInfo.iOSSound` 设置 `iOS` 平台上的离线消息铃声。

2、Android

> **注意：**
> 接口支持华为、小米、FCM。

FCM 的推送铃声为设置的应用铃声。

华为、小米 和 APNS 推送铃声设置请在调用 Call 和 GroupCall 的时候设置 `TUIOfflinePushInfo.iOSSound` 的 `iOSSound` 及 `androidSound` 字段。具体详情请参见 自定义铃声 。

监听通话状态
Android&iOS&Flutter

最近更新时间：2024-03-29 15:01:22

本文介绍 TUICallKit 组件通话状态回调的使用。

通话状态监听

如果您的业务需要**监听通话的状态**，例如通话开始、结束等通话过程中的事件，可以参考如下代码：

Android（Kotlin）

```
private val observer: TUICallObserver = object : TUICallObserver() {
    override fun onCallBegin(roomId: TUICommonDefine.RoomId?,
callMediaType: TUICallDefine.MediaType?, callRole: TUICallDefine.Role?)
{
    }
    override fun onCallEnd(roomId: TUICommonDefine.RoomId?,
callMediaType: TUICallDefine.MediaType?, callRole: TUICallDefine.Role?,
totalTime: Long) {
    }
    override fun onUserNetworkQualityChanged(networkQualityList:
MutableList<TUICommonDefine.NetworkQualityInfo>?) {
    }
}
private fun initData() {
    TUICallEngine.createInstance(context).addObserver(observer)
}
```

Android（Java）

```
private TUICallObserver observer = new TUICallObserver() {
    @Override
    public void onCallBegin(TUICommonDefine.RoomId roomId,
TUICallDefine.MediaType callMediaType, TUICallDefine.Role callRole) {
    }
    public void onCallEnd(TUICommonDefine.RoomId roomId,
TUICallDefine.MediaType callMediaType,TUICallDefine.Role callRole, long
totalTime) {
    }
```

```
    public void
onUserNetworkQualityChanged(List<TUICommonDefine.NetworkQualityInfo>
networkQualityList) {
    }
};

private void initData(){
    TUICallEngine.createInstance(context).addObserver(observer);
}
```

iOS（Swift）

```
import TUICallEngine

TUICallEngine.createInstance().addObserver(self)

public func onCallBegin(roomId: TUIRoomId, callMediaType:
TUICallMediaType, callRole: TUICallRole) {

}
public func onCallEnd(roomId: TUIRoomId, callMediaType:
TUICallMediaType, callRole: TUICallRole, totalTime: Float) {

}
public func onUserNetworkQualityChanged(networkQualityList:
[TUINetworkQualityInfo]) {

}
```

iOS（Objective-C）

```
#import <TUICallEngine/TUICallEngine.h>

[[TUICallEngine createInstance] addObserver:self];

- (void)onCallBegin:(TUIRoomId *)roomId callMediaType:
(TUICallMediaType)callMediaType callRole:(TUICallRole)callRole {

}
- (void)onCallEnd:(TUIRoomId *)roomId callMediaType:
(TUICallMediaType)callMediaType callRole:(TUICallRole)callRole
```

```
totalTime:(float)totalTime {

}
- (void)onUserNetworkQualityChanged:(NSArray<TUINetworkQualityInfo *>
*)networkQualityList {

}
```

Flutter（Dart）

```
TUICallObserver observer = TUICallObserver(
    onError: (int code, String message) {
      //您的回调处理逻辑
    }, onCallBegin: (TUIRoomId roomId, TUICallMediaType callMediaType,
TUICallRole callRole) {
      //您的回调处理逻辑
    }, onCallEnd: (TUIRoomId roomId, TUICallMediaType callMediaType,
TUICallRole callRole, double totalTime) {
      //您的回调处理逻辑
    },, onUserNetworkQualityChanged: (List<TUINetworkQualityInfo>
networkQualityList) {
      //您的回调处理逻辑
    }, onCallReceived: (String callerId, List<String> calleeIdList,
String groupId, TUICallMediaType callMediaType) {
      //您的回调处理逻辑
    }
    ……
))

TUICallEngine.instance.addObserver(observer);
```

注意：

在 Android 平台设置 TUICallObserver 监听回调，需保证回调所在类不会被清除，举例：不建议在 LoginActivity 中添加监听，LoginActivity 销毁的时候，回调也会被清除；建议在应用的 Application 中或者应用主界面进行监听。

Web&H5

最近更新时间：2025-05-16 17:17:42

本文介绍 TUICallKit 组件通话状态回调的使用。

通话状态监听

如果您的业务需要**监听通话的状态**，例如通话开始、结束等通话过程中的事件（详见 TUICallEvent ），可以参见如下代码：

```
import { TUICallEvent } from '@tencentcloud/call-engine-js';

let handleUserEnter = function(event) {
    console.log('TUICallEvent.USER_ENTER: ', event);
};

TUICallKitServer.getTUICallEngineInstance().on(TUICallEvent.USER_ENTER,
handleUserEnter);
TUICallKitServer.getTUICallEngineInstance().off(TUICallEvent.USER_ENTER,
handleUserEnter);
```

组件回调事件

TUICallKit 组件提供了通话状态回调，可以用于业务侧实现更多交互逻辑，具体参见 TUICallKit 属性介绍 。

- `beforeCalling`：通话开始前会执行。
- `afterCalling`：通话完成后执行。

> **说明：** WeChat 暂不支持。

React

```
function App() {
  const handleBeforeCalling = () => {
    console.log("[TUICallKit Demo] beforeCalling");
  };
  const handleAfterCalling = () => {
    console.log("[TUICallKit Demo] afterCalling");
  };
  return (
    <TUICallKit
```

```
      beforeCalling={handleBeforeCalling}
      afterCalling={handleAfterCalling} />
  )
}
```

Vue

```
<template>
  <TUICallKit
    :beforeCalling="handleBeforeCalling"
    :afterCalling="handleAfterCalling" />
</template>
<script setup>
function handleBeforeCalling() {
  console.log("[TUICallKit Demo] beforeCalling");
}
function handleAfterCalling() {
  console.log("[TUICallKit Demo] afterCalling");
}
</script>
```

uni-app（客户端）

最近更新时间：2024-04-30 09:47:21

本文介绍 TUICallKit 组件通话状态回调的使用。

通话状态监听

如果您的业务需要**监听通话的状态**，例如通话开始、结束等通话过程中的事件（详见 TUICallEvent），请参见如下代码：

```
const TUICallEngine = uni.requireNativePlugin('TencentCloud-TUICallKit-TUICallEngine');

function handleError(res) {
    console.log('onError', JSON.stringify(res);
}
TUICallEngine.addEventListener('onError', handleError);
TUICallEngine.removeEventListener('onError', handleError);
```

语言设置
Android

最近更新时间：2024-08-06 10:00:22

支持的语言

目前支持**简体中文、英文、日文和阿拉伯语。**

切换语言

TUICallKit 默认语言与手机系统保持一致 。如果需要切换语言，可以使用 `TUIThemeManager.getInstance().changeLanguage` 切换语言，以切换英文为例：

```
……
import com.tencent.qcloud.tuicore.TUIThemeManager;

public class MainActivity extends BaseActivity {
    @Override
  public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);


TUIThemeManager.getInstance().changeLanguage(getApplicationContext(),
"en");
      ……
    }
    ……
}
```

添加新的语言

第一步：源码集成

1. 在 Github 中克隆/下载代码，然后拷贝 Android 目录下的 tuicallkit-kt 子目录到您当前工程中的 app 同一级目录中，如下图所示。

2. 在工程根目录下找到 `settings.gradle.kts`（或`settings.gradle`）文件，在其中增加如下代码，导入 `tuicallkit-kt` 组件到项目中。

setting.gradle.kts

```
include(":tuicallkit-kt")
```

settings.gradle

```
include ':tuicallkit-kt'
```

3. 在 app 目录下找到 `build.gradle.kts(或build.gradle)` 文件，在 `dependencies` 中增加如下代码，声明当前 app 对新加入的组件的依赖。

build.gradle.kts

```
dependencies {
    api(project(":tuicallkit-kt"))
}
```

build.gradle

```
dependencies {
    api project(':tuicallkit-kt')
}
```

第二步：新增语言包

以西班牙语为例：

1. 新增西班牙语文件。

   进入到 `TUICallKit` 源码文件目录下的 `src/main/res` 目录下，新增 `value-es/strings.xml` 文件 。
2. 将 `src/main/res/values-en/strings.xml` 中的内容复制到新增的 `src/main/res/values-es/strings.xml` 文件中。
3. 将 `src/main/res/values-es/strings.xml` 中的英文翻译为西班牙语。
4. 新增语言。

```
  ……
import com.tencent.qcloud.tuicore.TUIThemeManager;

public class MainActivity extends BaseActivity {
    @Override
  public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      Locale locale = new Locale("es");
      TUIThemeManager.addLanguage("es", locale);
      ……
    }
    ……
}
```

iOS

最近更新时间：2024-08-06 10:00:22

支持的语言

目前支持**简体中文、英文、日文和阿拉伯语。**

切换语言

TUICallKit 默认语言与手机系统保持一致 。如果需要切换语言，可以使用 `TUIGlobalization.setPreferredLanguage` 切换语言，以切换英文为例：

```
……
import TUICore

func steLanguage() {
    TUIGlobalization.setPreferredLanguage("en")
}
```

添加新的语言

第一步：源码集成

1. 在 Github 中克隆/下载代码。
2. 在 Application 工程的 Podfile 文件中依赖下载到本地 TUICallKit 源码。

```
target 'TUICallKitApp' do
  use_frameworks!
  ……
  pod 'TUICallKit-Swift/Professional', :path => "Your Download
Path/TUICallKit/iOS"

end
```

3. 执行 `pod update` 命令，更新依赖。

第二步：新增语言包

以西班牙语为例：

1. 新增西班牙语文件。

   进入到 `TUICallKit` 源码文件目录下的 iOS/TUICallKit-Swift/Resources 目录下，新增 `es.lproj/Localized.strings` 文件 。
2. 将 `iOS/TUICallKit-Swift/Resources/en.lproj/Localized.strings` 中的内容复制到新增的 `iOS/TUICallKit-Swift/Resources/es.lproj/Localized.strings` 文件中。
3. 将 `iOS/TUICallKit-Swift/Resources/es.lproj/Localized.strings` 中的英文翻译为西班牙语。
4. 进入到 Application 工程的 Podfile 所在的目录文件夹下执行 `pod install` 命令，更新依赖。

```
  ……
import com.tencent.qcloud.tuicore.TUIThemeManager;

public class MainActivity extends BaseActivity {
    @Override
  public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      Locale locale = new Locale("es");
      TUIThemeManager.addLanguage("es", locale);
      ……
    }
    ……
}
```

Web&H5

最近更新时间：2024-07-26 14:54:11

支持的语言

目前支持**简体中文、英文和日文**。

切换语言

TUICallKit 会优先获取浏览器语言，如果是中文，英文或者日文中的一种，则以浏览器语言为主，否则使用英文。如果想切换语言，可以使用 setLanguage 接口。

```
import { TUICallKitServer, TUICallType } from '@tencentcloud/call-uikit-vue';

TUICallKitServer.setLanguage("zh-cn"); // "en" | "zh-cn" | "ja_JP"
```

添加新的语言

如果您有其他语言需要支持，可以通过源码集成方式修改语言源文件实现。

第一步：源码集成

> ⚠ **注意：**
> 源码集成适用于 `Vue + Typescript` 项目且 `TUICallKit` 版本号 ≥ 3.2.2。

1. **下载源码**

Vue3

```
npm install @tencentcloud/call-uikit-vue
```

2. **将源码拷贝到自己的项目中，以拷贝到 `src/components/` 目录为例：**

macOS + Vue3

```
mkdir -p ./src/components/TUICallKit && cp -r
./node_modules/@tencentcloud/call-uikit-vue/*
./src/components/TUICallKit
```

Windows + Vue3

```
xcopy .\node_modules\@tencentcloud\call-uikit-vue
.\src\components\TUICallKit /i /e
```

3. **修改引入路径**

   需要将 TUICallKit 改为从本地文件中引入，如下方代码。其他用法细节可参考见 TUICallKit 快速接入 。

```
import { TUICallKit, TUICallKitServer, TUICallType } from
"./components/TUICallKit/src/index";
```

4. **解决源码拷贝可能导致的报错**

   如果您在使 TUICallKit 组件时遇到了报错，请不要担心，大多数情况下这是由于 ESLint 和 TSConfig 配置不一致造成的。您可以查阅文档，按照要求正确配置即可。如果您需要帮助，请随时联系我们，我们将确保您能够成功地使用此组件。以下是几个常见的问题：

ESLint 报错

若 TUICallKit 与您项目的代码风格不一致导致报错，可将本组件目录屏蔽，如在项目根目录增加 `.eslintignore` 文件，如：

```
# .eslintignore
src/components/TUICallKit
```

TypeScript 报错

1. 如遇 Cannot find module '../package.json' 报错，是因为 TUICallKit 内引用了 JSON 文件，可在 tsconfig.json 中添加相关配置，示例：

```
{
  "compilerOptions": {
    "resolveJsonModule": true
  }
}
```

   其他 TSConfig 问题请参见 TSConfig Reference。

2. 如遇 Uncaught SyntaxError: Invalid or unexpected token 报错，是因为 TUICallKit 使用了装饰器，可在 tsconfig.json 中添加相关配置，示例：

```
{
  "compilerOptions": {
    "experimentalDecorators": true
  }
}
```

第二步：新增语言包

以新增越南语为例：

1. 创建目标语言源文件。

   在 src/components/TUICallKit/src/TUICallService/locales 目录下新增 vi.ts 文件，复制 src/components/TUICallKit/src/TUICallService/locales/zh-cn.ts 内容到 vi.ts，然后将 json 的 value 翻译成越南语。

```
export const vi = {     // 注意这里的导出变量
  'hangup': '挂断',
  'reject': '拒绝',
  'accept': '接受',
  'camera': '摄像头',
  'microphone': '麦克风',
  'speaker': '扬声器',
  'open camera': '打开摄像头',
  'close camera': '关闭摄像头',
  'open microphone': '打开麦克风',
  'close microphone': '关闭麦克风',
  'video-to-audio': '转语音通话',
  'virtual-background': '模糊背景',
  'other side reject call': '对方已拒绝',
  'reject call': '拒绝通话',
  'cancel': '取消通话',
  ...
};
```

2. 从 index.ts 导出

   修改 src/components/TUICallKit/src/TUICallService/locales/index.ts 文件。

```
import { TUIStore } from '../CallService/index';
```

```
import { NAME, StoreName } from '../const/index';
import { en } from './en';
import { zh } from './zh-cn';
import { ja_JP } from './ja_JP';
import { vi } from './vi';  // 新增语言文件导入

.....

export const languageData: languageDataType = {
  en,
  'zh-cn': zh,
  ja_JP,
  vi,        // 新增语言文件导出
};
```

3. 新增 LanguageType 枚举。

修改 src/components/TUICallKit/src/TUICallService/const/call.ts

```
export enum LanguageType {
  EN = 'en',
  'ZH-CN' = 'zh-cn',
  JA_JP = 'ja_JP',
  VI = 'vi',    // 新增枚举类型
}
```

4. 切换语言

在项目中通过调用 setLanguage 接口切换语言。

```
import { TUICallKitServer, TUICallType } from '@tencentcloud/call-
uikit-vue';

TUICallKitServer.setLanguage("vi");
```

Flutter

最近更新时间：2024-08-06 10:00:22

支持的语言

目前支持**简体中文、英文和日文，默认语言为英文** 。

切换语言

`TUICallKit` 不单独提供语言切换的接口， `TUICallKit` 根据当前 `Application` 的 `MaterialApp` （或 CupertinoApp 等风格组件）使用的语言自适切换，切换 `MaterialApp` （或 CupertinoApp 等风格组件）使用的语言即可。

添加新的语言

第一步：源码集成

1. 下载源码

   进入 `https://pub.dev/packages/tencent_calls_uikit` 下载最新 `TUICallKit` 源码。
2. 依赖本地源码

   在 `Application` 工程的 `pubspec.yaml` 文件中修改 `TUICallKit` 为本地依赖：

```
dependencies:
  tencent_calls_uikit:
    path: /TUICallKit 本地路径/
```

第二步：新增语言包

以西班牙语为例：

1. 新增西班牙语文件。

   进入到 `TUICallKit` 源码文件目录下的 `lib/src/i18n` 目录下，新增 `strings_es.i18n.json` 。
2. 将 `lib/src/i18n/strings.i18n.json` 中的内容复制到新增的 `lib/src/i18n/strings_es.i18n.json` 文件中。
3. 将 `lib/src/i18n/strings_es.i18n.json` 中的英文翻译为西班牙语。
4. 更新翻译包

   在 `TUICallKit` 源码文件目录下进入命令行工具，执行以下命令更新翻译包：

```
flutter pub add fast_i18n
```

```
flutter pub run fast_i18n
```

5. 更新 `TUICallKit` 自适应设置语言方法。

进入到 `lib/src/i18n/i18n_utils.dart` 源码文件中修改 `setLanguage` 方法，修改如下：

```
  static setLanguage(Locale currentLocale) {
  switch (currentLocale.languageCode) {
    case 'zh':
      {
        CallKitI18nUtils(null, 'zh');
        break;
      }
    case 'en':
      {
        CallKitI18nUtils(null, 'en');
        break;
      }
    case 'ja':
      {
        CallKitI18nUtils(null, 'ja');
        break;
      }
    // 新增 case 'es'
    case 'es':
      {
        CallKitI18nUtils(null, 'es');
        break;
      }
  }
}
```

uni-app（客户端）

最近更新时间：2024-08-06 10:00:22

支持的语言

目前支持**简体中文、英文、日文**和**阿拉伯语**。

切换语言

TUICallKit 语言与手机系统保持一致 。

添加新的语言

uni-app（客户端）暂不支持新增语言。

如果您有新增语言的需求，请通过 zhiliao 平台和技术、产品进行沟通。

体积优化
uni-app（小程序）

最近更新时间：2024-08-26 16:51:01

警告：

影响包体积的通常原因是项目中的静态资源过大，其他分包中的第三方依赖被打入主包中，具体可以参见 通用解决方案 。

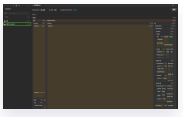
特别注意：单独集成 TUIKit 和 TUIKit 与 TUICallKit 的融合场景，大约**占用主包体积约 800 KB，需要预留对应的主包体积。**

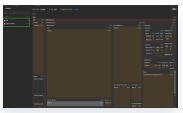
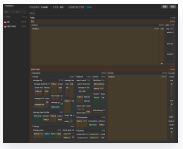
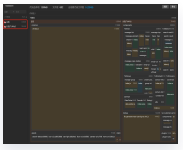
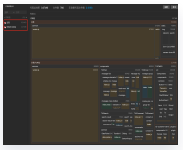
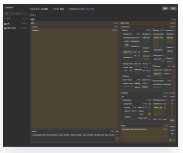
如何查看小程序包体积占比

在微信开发者工具中，顶部操作栏右侧 > 详情 > 代码依赖分析，可以查看编译后的包的情况与各文件大小。

包体积占比参照

> **注意：**
> 这里是统一使用**分包集成**方式并经过下面的优化手段后的体积占用，可以通过对比下表，快速分析项目中的包体积占用，并**预留相对应的主包体积**。

场景	环境	需预留主包体积	分包体积	截图	说明
单独集成 TUICallKit	Vue2	**不占用主包体积**	1.48 MB		主包仅包含 uniapp 相关打包产物

	Vue3	**不占用主包体积**	1.37 MB		主包仅包含 uniapp 相关打包产物
单独集成 TUIKit	Vue2	**约 800 KB**	1.48 MB		主包中包含 Chat SDK 和 TUICore 约**800**KB
	Vue3	**约 800 KB**	764 KB		主包中包含 Chat SDK 和 TUICore 约**800**KB
TUIKit 与 TUICallKit 的融合场景	Vue2	**约 800 KB**	2.07 MB		主包中包含 Chat SDK 和 TUICore 约**800**KB
	Vue3	**约 800 KB**	1.22 MB		主包中包含 Chat SDK 和 TUICore 约**800**KB

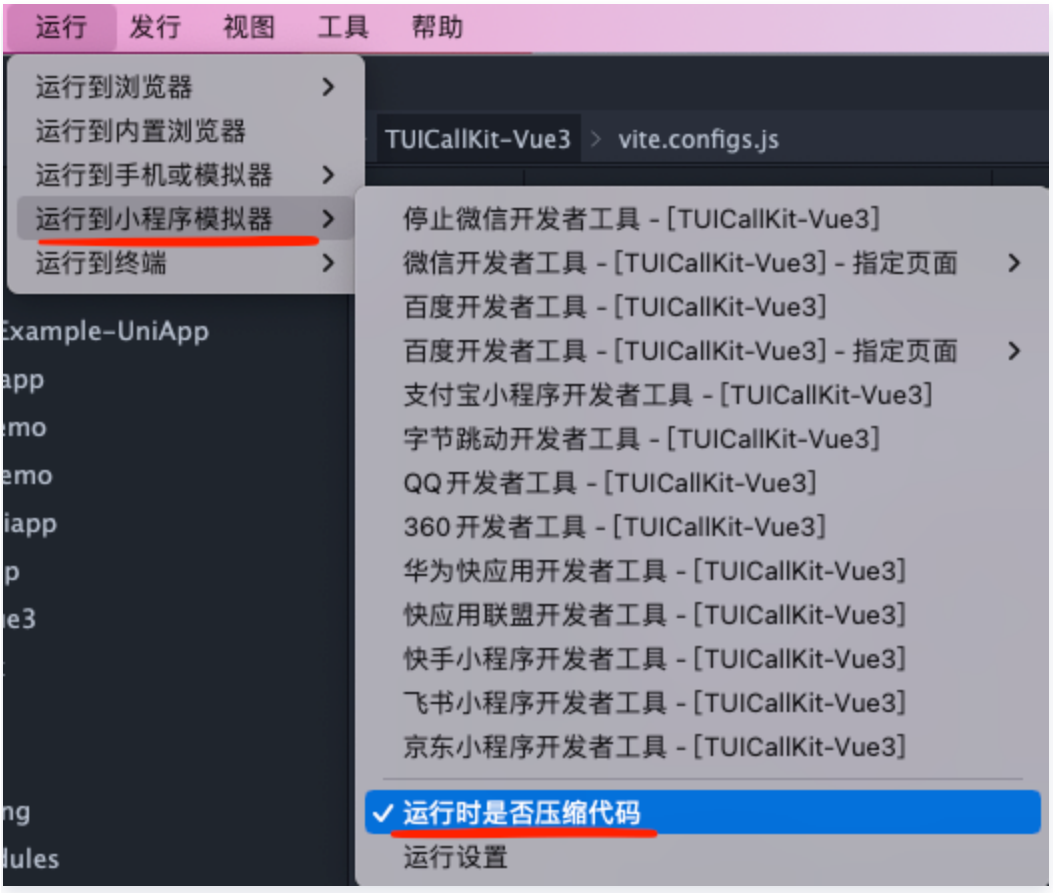
说明：
在单独集成 TUIKit和融合场景下，Chat SDK 和 TUICore 位于主包中，大约**占用主包体积约 800 KB，需要预留对应的主包体积。**

IDE 配置优化方案

Hbuilder 运行时相关配置

顶部操作栏 > 运行 > 运行到小程序模拟器 > 运行时是否压缩代码（勾选）

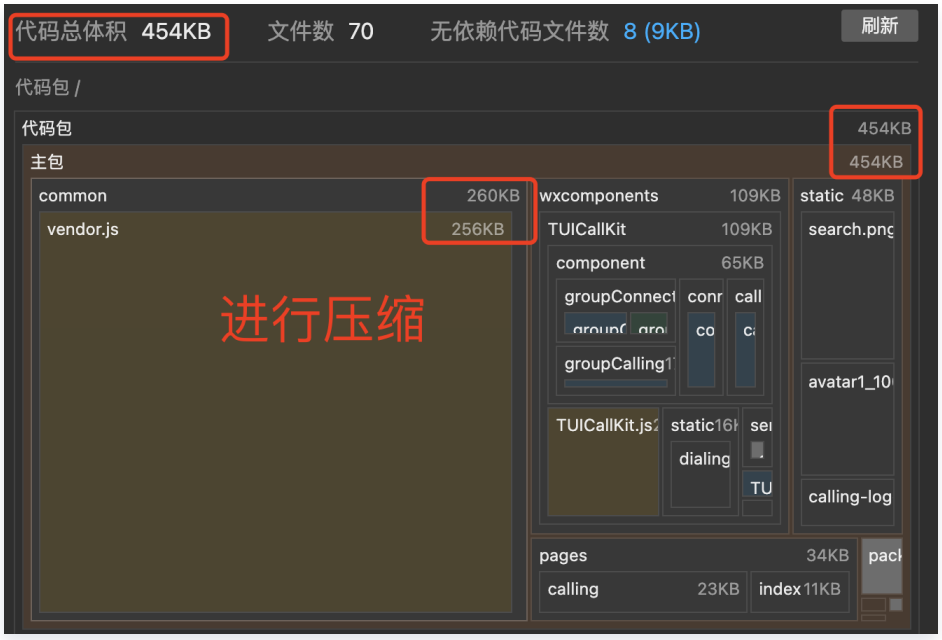
> **说明：**
> 该步骤能有效减少第三方插件或组件库的代码体积，但也仅限代码，静态资源不会被压缩。



下图给出了是否勾选运行时是否压缩代码的效果对比（**851KB -> 454KB，降低 50%**）

- **不进行压缩：**

- 进行压缩



微信开发者工具上传代码配置

> **说明：**
> 顶部操作栏 > 详情 > 上传代码时自动压缩样式文件 / 上传代码时自动压缩脚本 / 上传代码时自动压缩wxml文件 / 上传代码时过滤无依赖文件（勾选）

使用发行模式

在 HBuilderX 的工具栏中依次单击发行 > 小程序-微信。

> **说明：**
> 小程序发布前使用 HBuilderX 的发行功能进行打包。使用发行方式进行打包会删除代码中的 Source Map 等信息，打包体积最小。

然后单击**发行**，会在项目目录中的 `unpackage/dist/build/mp-weixin` 文件夹中生成微信小程序源码，再使用微信开发者工具打开该文件夹即可。

通用优化方案

删除 Debug 脚本

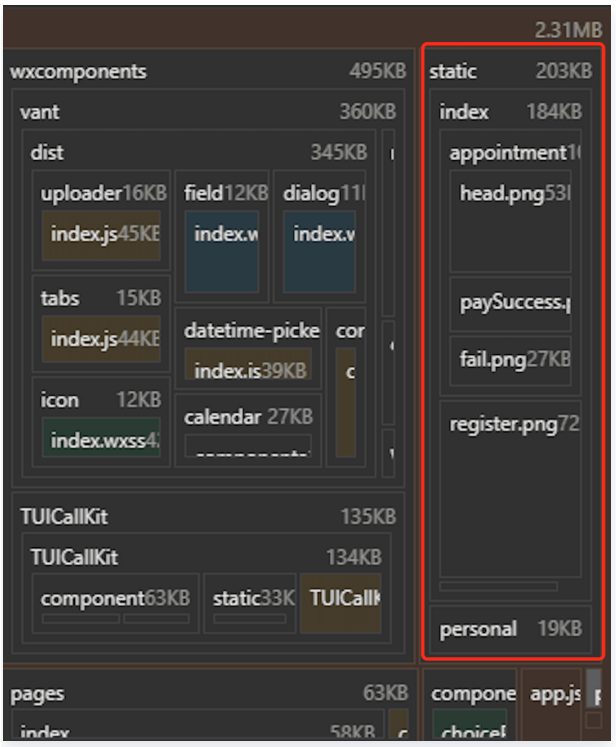
> **说明：**
> 出于体积和安全双重因素考虑，请在发布前删除项目目录下 `debug` 文件夹。可以节省约**150kb包体积**，生产环境推荐在服务端生成。

TUICallKit 的脚本文件存放于 `TUICallKit/debug` 目录。
TUIKit 的脚本文件存放于 `TUIKit/debug` 目录。

托管项目中的静态资源

> **警告：**
> 这一步往往是最影响包体积的一部分。

下面通过一个典型案例来说明：



从上图中的包体积分析中可以看出来， staic 中的静态资源达到了203KB，这些图片资源完全可以托管到服务器上，为主包节约203KB的大小。

公共样式统一加载

公共样式需要统一进行加载。不要每次进行引入，公共样式统一加载的好处不仅体现在减少代码重复，还有利于减少小程序包体积。

针对第三方 UI 库按需引入

使用 UI 库时只引入自己使用到的部分，按需引入只加载项目中实际使用的组件和样式，避免了整个 UI 库被打包，从而减小了最终产物的体积。

使用外部依赖优化包体积

> **警告：**
> Vue3 项目其他分包中的第三方依赖也会被打入主包中，导致主包体积膨胀，可以使用这种方式避免将分包中其他第三方依赖被打入主包中。

具体可以参见 如何使用外部依赖 。

单独集成 TUICallKit 体积优化

Vue2 项目

> **说明：**

> Vue2 使用 **webpack** 打包，独立分包中都会有单独的 vendor.js ，分包中的相关依赖不会打到主包里，不存在包体积问题。

TUICallKit 作为独立分包之后，**没有占有任何主包体积**，用户只需在 TUICallKit 分包中完成相关业务即可，包体积问题可以参考上述的**通用体积优化方案**。

Vue3 项目

> **说明：**
> Vue3 使用 vite 打包，分包中的依赖会被打入主包中，造成**主包体积过大**问题。

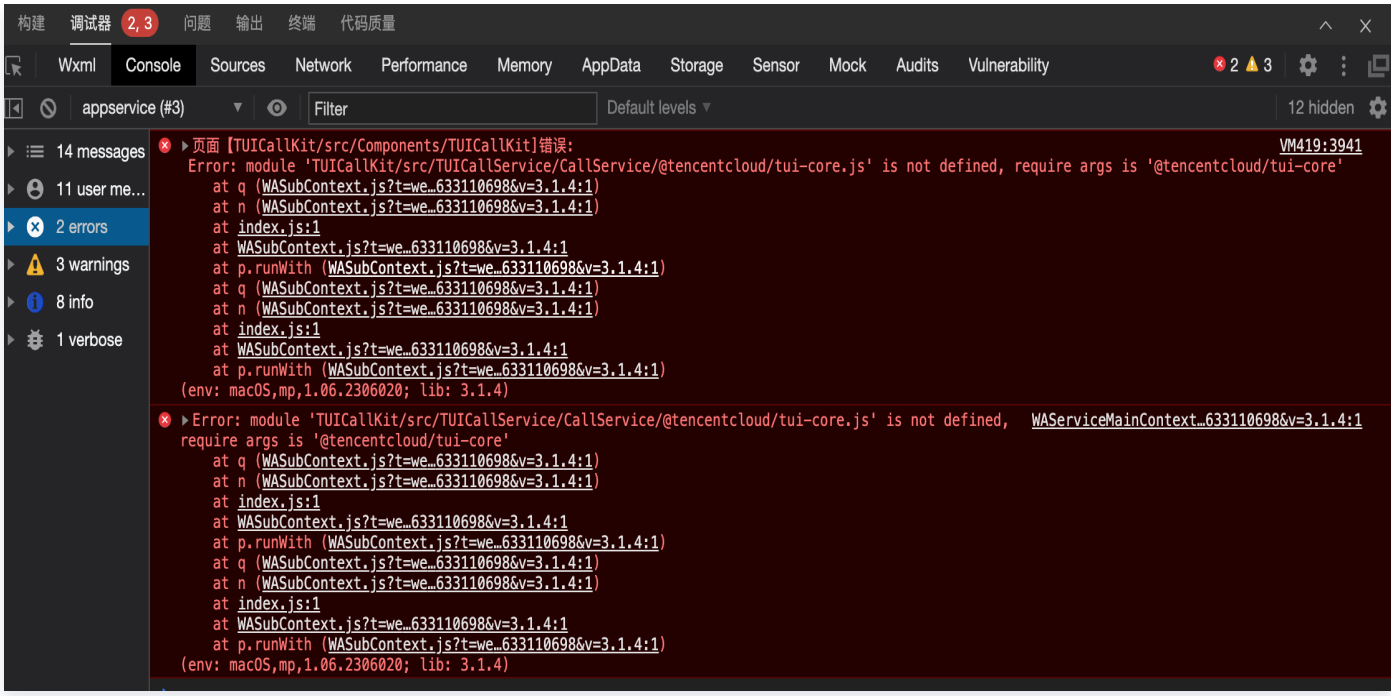
1. 在根目录下新建 vite.config.js 文件

```
import { defineConfig } from "vite";
import uni from "@dcloudio/vite-plugin-uni";

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [uni()],
  optimizeDeps: {
    // 这里的 include 以 TUICallKit 为例，可以使用这种方式避免将分包中其他第三方依赖被打入主包中
    include: ["tuicall-engine-wx",'@tencentcloud/tui-core', '@tencentcloud/chat'],
  },
  build: {
    rollupOptions: {
      external: ["tuicall-engine-wx",'@tencentcloud/tui-core', '@tencentcloud/chat'],
    },
  },
});
```

2. 构建 npm

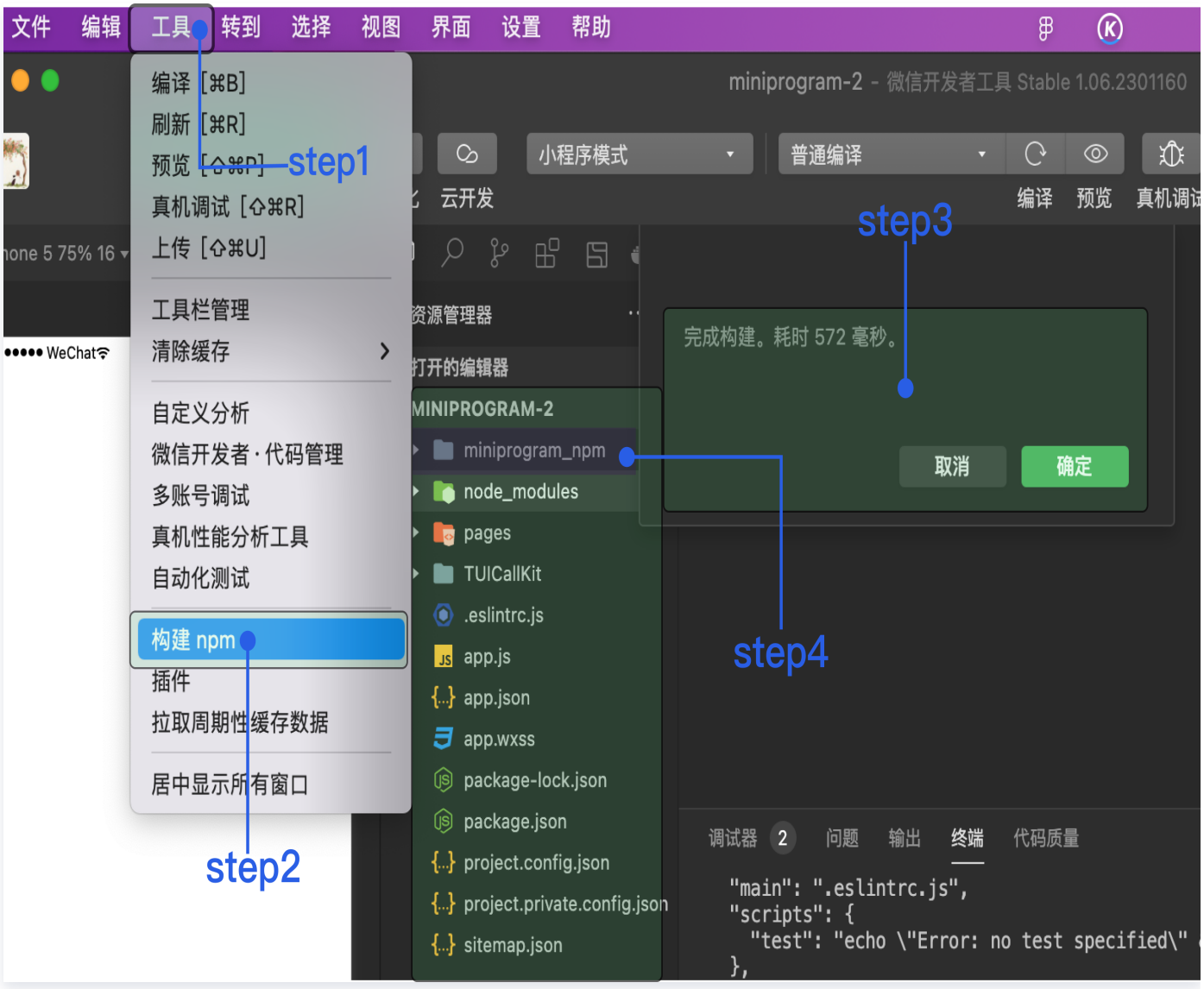
因为 HBuilder 运行到小程序后，项目中的 package.json、node_modules 都不存在了。需要 微信开发者工具 的终端里，重新安装依赖。



3. 进入分包页面，重新下载依赖。

```
cd TUICallKit && npm init -y && npm i @tencentcloud/call-uikit-wechat
```

4. 构建 npm。



单独集成 TUIKit 体积优化

> **说明：**
> 包体积问题可以参考上述的**通用体积优化方案**。

由于要保证和 uniapp 打包 App 保证一致性 ，登录逻辑需要位于主包，主包中包含 Chat SDK 和 TUICore，大约占用约 800 KB，需要预留相关的主包体积。

TUIKit 和 TUICallKit 融合场景体积优化

Vue2 项目

> **说明：**

> 包体积问题可以参考上述的**通用体积优化方案**。

由于 TUICallKit 和 TUIKit 处于同一个分包下，分包体积会超出2MB， 需要将 Chat SDK 和 TUICore 转移到主包中，大约**占用主包体积约800KB，需要预留相关的主包体积。**

Vue3 项目

使用外部依赖的方式优化包体积

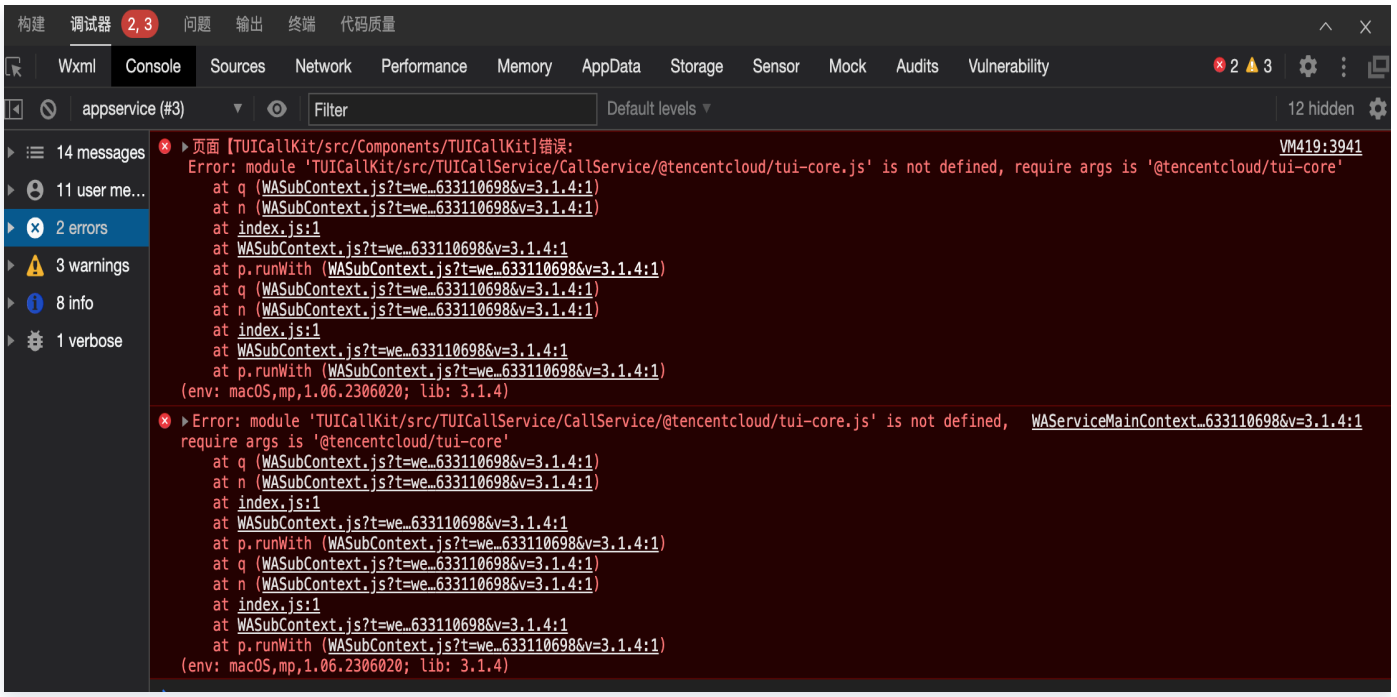
1. 在根目录下新建 vite.config.js 文件

```
import { defineConfig } from "vite";
import uni from "@dcloudio/vite-plugin-uni";

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [uni()],
  optimizeDeps: {
    // 这里的 include 以 TUICallKit 为例，可以使用这种方式避免将分包中其他第三方依赖被打入主包中
    include: ["tuicall-engine-wx",'@tencentcloud/tui-core', '@tencentcloud/chat'],
  },
  build: {
    rollupOptions: {
      external: ["tuicall-engine-wx",'@tencentcloud/tui-core', '@tencentcloud/chat'],
    },
  },
});
```

2. 构建 npm

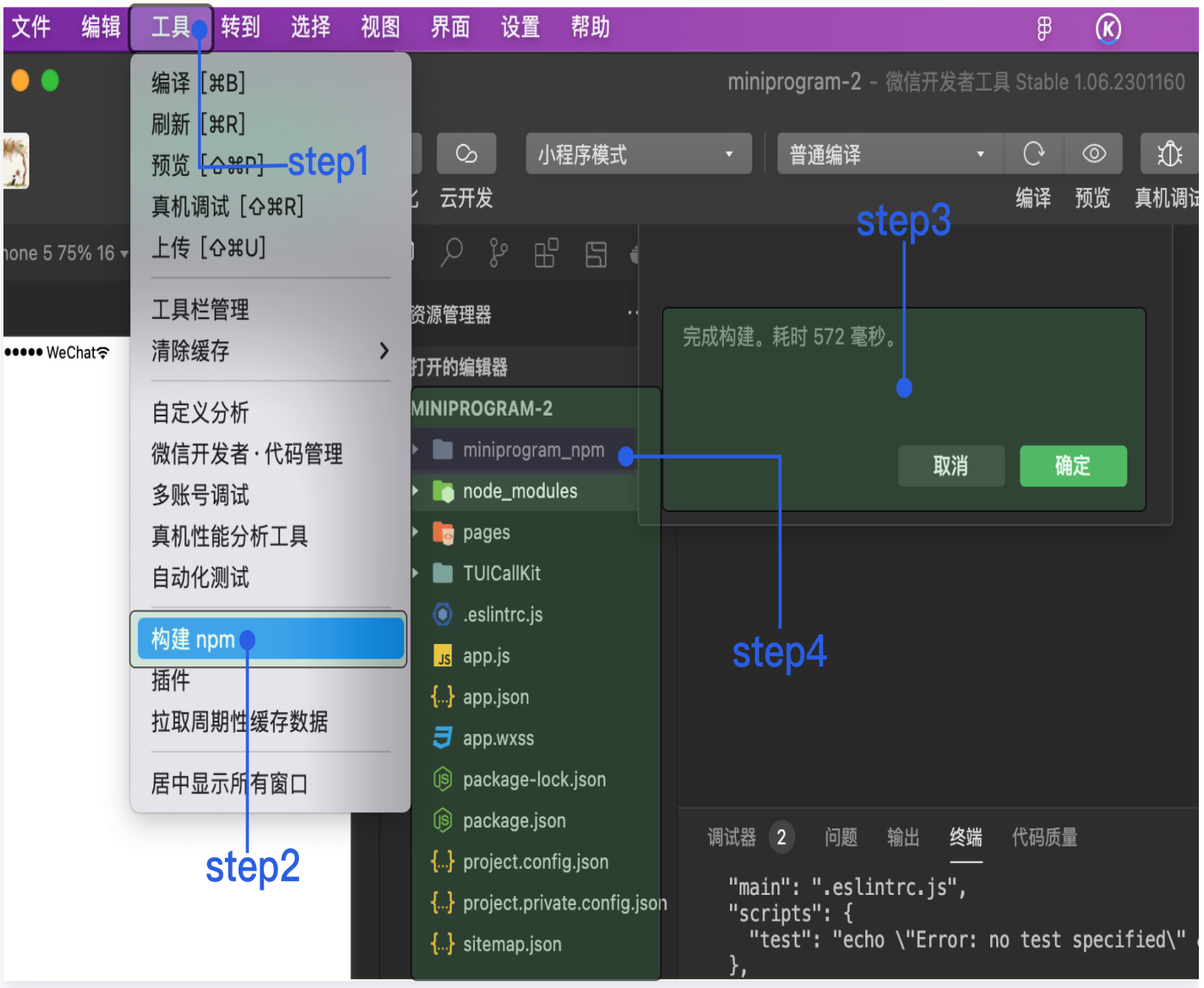
因为 HBuilder 运行到小程序后，项目中的 package.json、node_modules 都不存在了。需要 微信开发者工具 的终端里，重新安装依赖。



3. 进入分包页面，重新下载依赖。

```
cd TUIKit && npm init -y && npm i @tencentcloud/call-uikit-wechat
```

4. 构建 npm。



高级用法

警告：

下面这一部分，通过移除 UIKit 的某些您可能未使用到的组件来降低包体积，适用于上述的方法都未解决包体积问题时，请谨慎抉择！

移除客服插件

说明：

注释以下五段代码，大约可以减小约 50 KB 的包体积。

1. TUIKit/plugins/plugin-components/index.ts。

```
import TUIChatEngine, { IMessageModel } from '@tencentcloud/chat-uikit-engine';
// import { isCustomerServicePluginMessage } from './message-customer/index';
import { JSONToObject } from '../../utils/type-check';
```

```
export function isPluginMessage(message: IMessageModel): boolean {
  return (
    message.type === TUIChatEngine.TYPES.MSG_CUSTOM
    && (isCallMessage(message)
    // || isCustomerServicePluginMessage(message as any)
    || isRoomCardMessage(message)
    || isRoomSignalingMessage(message))
  );
}
```

2. TUIKit/plugins/plugin-components/message-plugin.vue。

```
<template #messageBubble>
  <MessageCallC2C
    v-if="pluginMessageType.pluginType === 'call'"
    :message="props.message"
    :signalingInfo="messageSignalingInfo"
    :customContent="messageCustomContent"
  />
  <!-- <MessageCustomerService
    v-if="pluginMessageType.pluginType === 'customer'"
    :message="props.message"
  /> -->
  <MessageRoom
    v-if="pluginMessageType.pluginType === 'room'"
    :message="props.message"
  />
</template>
</MessagePluginLayout>
</template>
```

移除群通话组件

说明：

- 单独集成 TUICallKit，位于 TUICallKit/src/Components/TUICallKit.vue。
- 融合场景下，位于 TUIKit/TUICallKit/src/Components/TUICallKit.vue。
- 注释以下代码，大约可以减小约 40 KB 的包体积。

```vue
<template>
  <div v-if="callInfoContextValue.callStatus !== CallStatus.IDLE" :style="[{ visibility: floatWindowContextValue.isFloatWindow ?
    <SingleCall v-if="!callInfoContextValue.isGroupCall" class="singCall"></SingleCall>
    <GroupCall v-else class="singCall"></GroupCall>
  </div>
</template>

<script lang="ts" setup>
import { ref, onMounted, reactive, onUnmounted, toRefs, watchEffect, provide } from '../adapter-vue';
import SingleCall from './components/SingleCall/SingleCall.vue';
import GroupCall from './components/GroupCall/GroupCall.vue';
import { TUIGlobal, TUIStore, StoreName, TUICallKitServer, NAME, CallStatus, CallMediaType } from '../TUICallService/index';
import {
  CallInfoContextKey,
  CallerUserInfoContextKey,
  UserInfoExcludeVolumeContextKey,
  FloatWindowContextKey,
  TCallerUserInfoValue,
  TFloatWindowContextValue,
  CustomUIConfigContextKey,
} from './context';
```