直播推流 SDK 开发文档

目录

1. 资源获取

1.1 下载 SDK

Android SDK 下载：PLDroidMediaStreaming

IOS SDK 下载：PLMediaStreamingKit

1.2 下载 demo

我们提供了本地下载及扫码下载两种方式

下载地址请访问：https://developer.qiniu.com/pili/sdk/5028/push-the-sdk-download-experience

1.3 API 参考

iOS API 参考地址

Android API 参考地址

2. Android 推流端 SDK

2.1 概述

PLDroidMediaStreaming 是一个适用于 Android 的 RTMP 直播推流 SDK，可高度定制化和二次开发。特色是同时支持 H.264 软编 / 硬编和 AAC 软编 / 硬编。支持 Android Camera 画面捕获，并进行 H.264 编码，以及支持 Android 麦克风音频采样并进行 AAC 编码；还实现了一套可供开发者选择的编码参数集合，以便灵活调节相应的分辨率和码率；同时，SDK 提供数据源回调接口，用户可进行 Filter 处理。借助 PLDroidMediaStreaming ，开发者可以快速构建一款类 Android 直播应用。

2.2 功能特性

- 支持 H.264 和 AAC 软编（推荐）
- 支持 H.264 和 AAC 硬编
- 支持录屏
- 软编支持 Android Min API 15（Android 4.0.3）及其以上版本
- 硬编支持 Android Min API 18（Android 4.3）及其以上版本
- 支持构造带安全授权凭证的 RTMP 推流地址
- 支持 RTMP 封包及推流
- 支持 RTMP 推流自适应网络质量动态切换码率或自定义策略
- 支持数据源回调接口，可自定义 Filter (滤镜) 特效处理
- 支持前后置摄像头，以及动态切换
- 支持自动对焦
- 支持手动对焦
- 支持动态水印
- 支持美颜，以及调节磨皮、美白、红润效果
- 支持 Encoding Mirror 设置
- 支持 Zoom 操作
- 支持 Mute/Unmute
- 支持闪光灯操作
- 支持纯音频推流，以及后台运行
- 支持截帧功能
- 支持动态更改 Encoding Orientation
- 支持动态切换横竖屏
- 支持蓝牙麦克风
- 支持后台推流
- 支持双声道立体声
- 支持 QUIC 推流
- 支持 ARM, ARMv7a, ARM64v8a, X86 主流芯片体系架构

2.3 开发准备

2.3.1 设备以及系统

- 设备要求：搭载 Android 系统的设备
- 系统要求：Android 4.0.3(API 15) 及其以上

2.3.2 混淆

为了保证正常使用 SDK ，请在 proguard-rules.pro 文件中添加以下代码：

```
–keep class com.qiniu.pili.droid.streaming.** { *; }
```

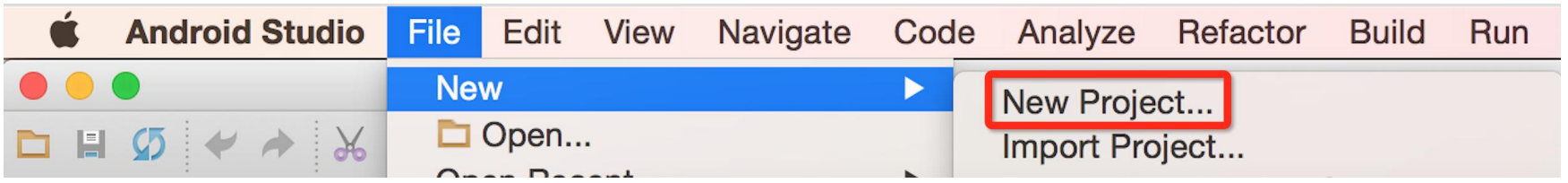
2.4 快速开始

2.4.1 开发环境配置

- 已全部完成 BOOK - I 中的所有操作。搭建好带有 Pili server sdk 的业务服务端，SDK 推流信息的输入来自服务端返回的推流地址或者 StreamJson（**推荐使用推流地址**）
- Android Studio 开发工具。官方下载地址
- 下载 Android 官方开发 SDK 。官方下载地址。PLDroidMediaStreaming 软编要求 Min API 15 和硬编要求 Android Min API 18
- 下载 PLDroidMediaStreaming 最新的 JAR 和 SO 文件。下载地址
- 请用真机调试代码，模拟器无法调试。

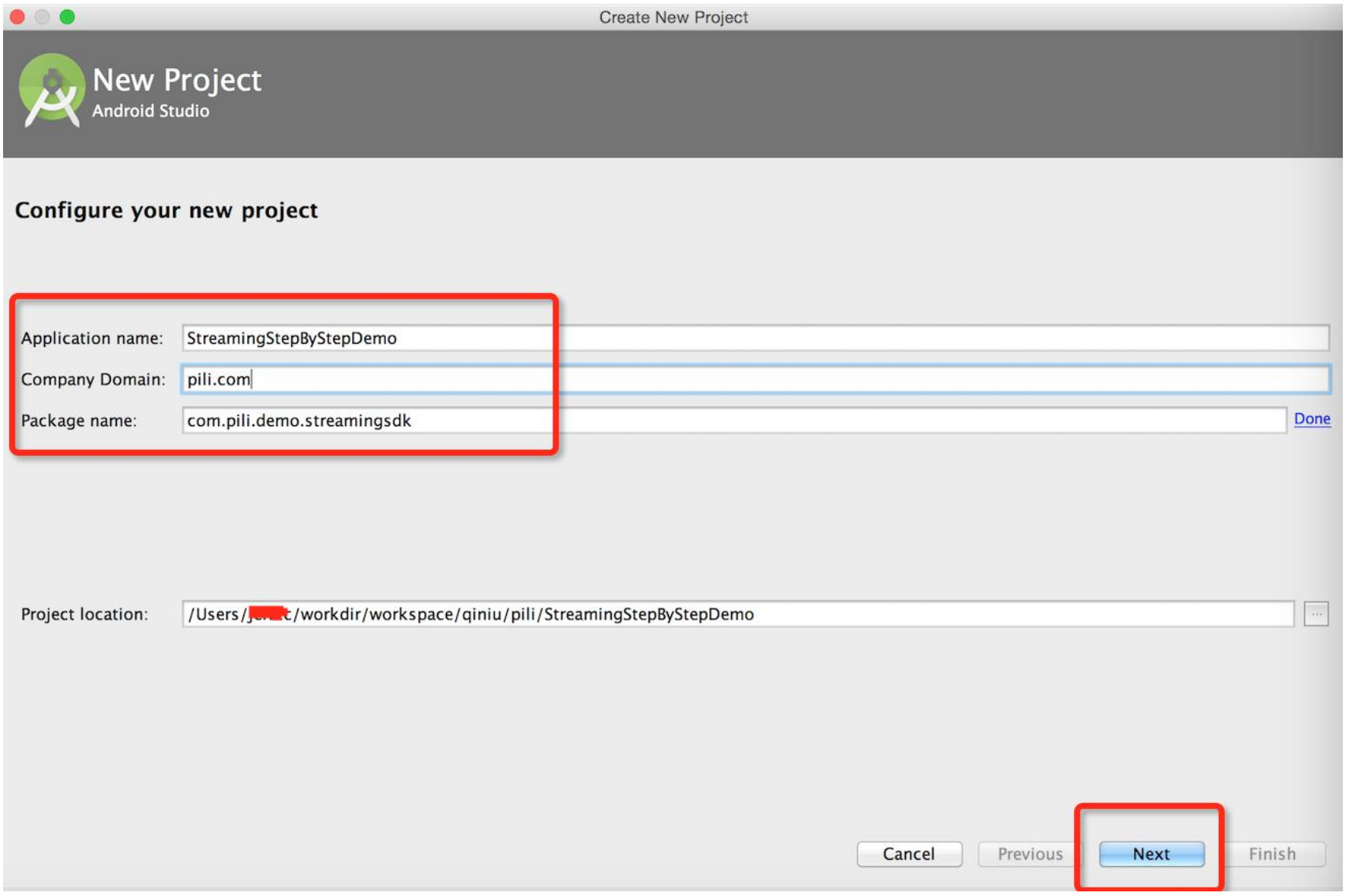
2.4.2 创建新工程

1、通过 Androname Studio 创建 Project

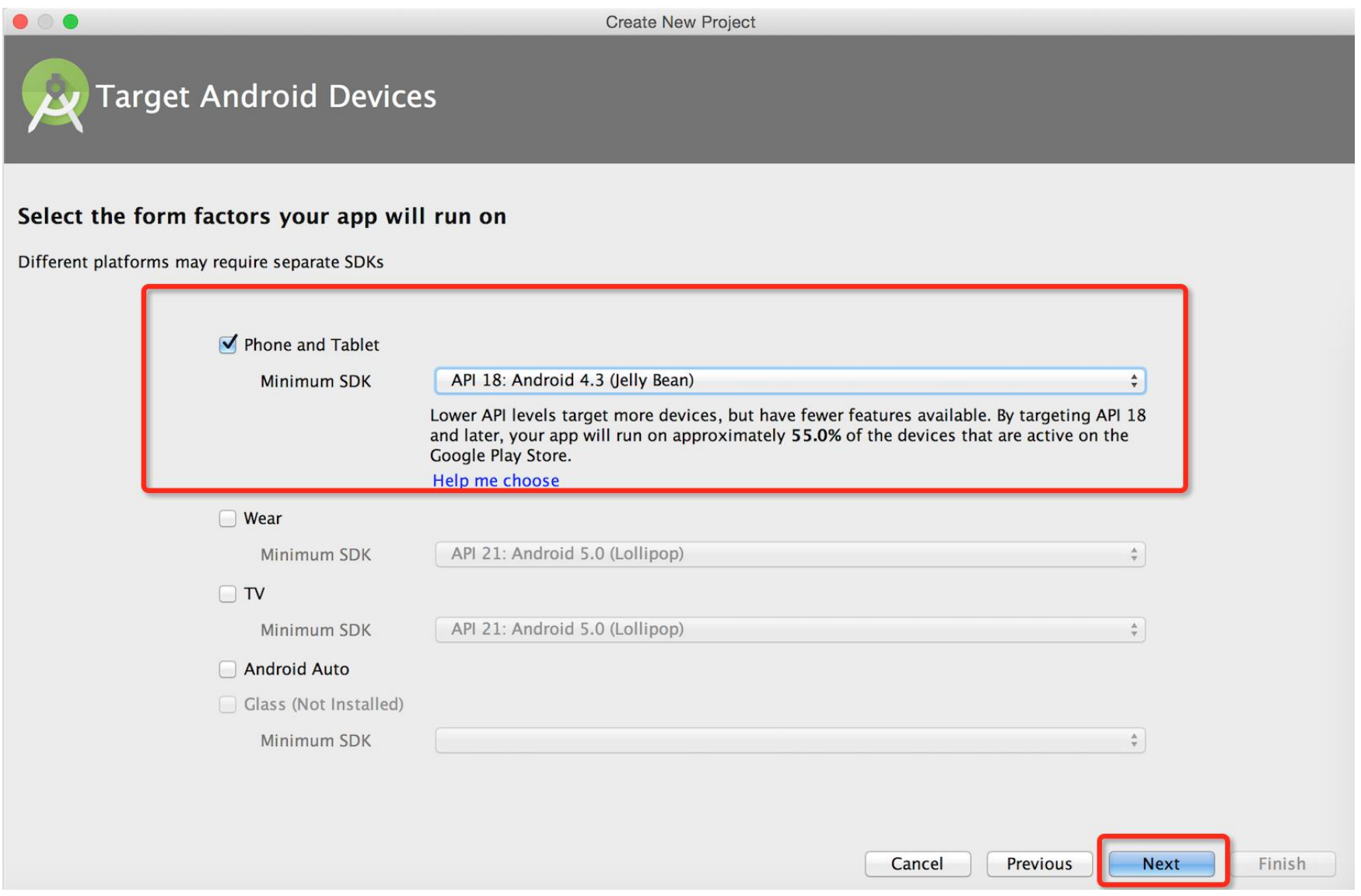


2、设置新项目

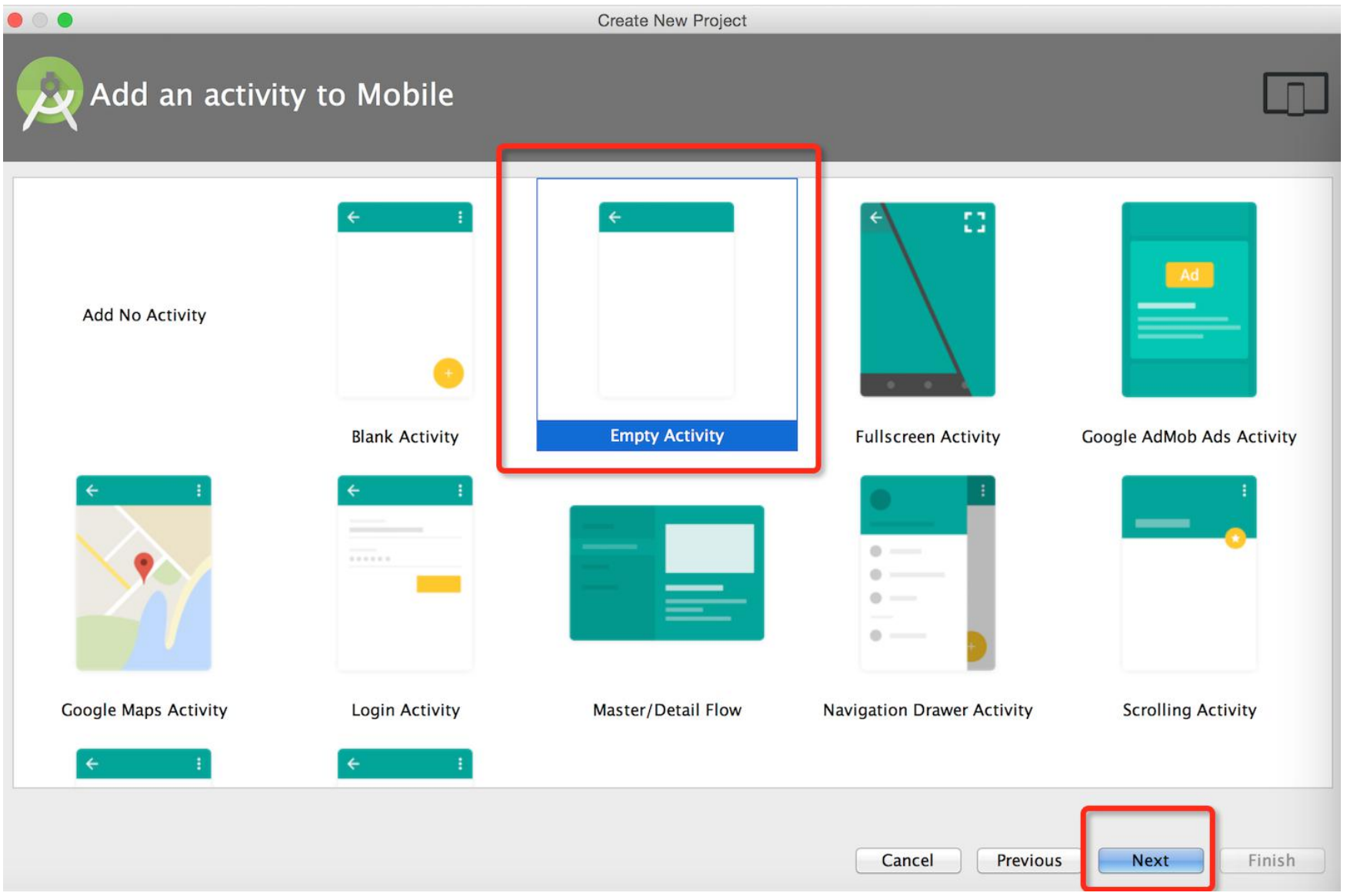
- 填写 Application id
- 填写 Company Domain
- 填写 Package id
- 选择 Project location
- 可以使用默认的填写项



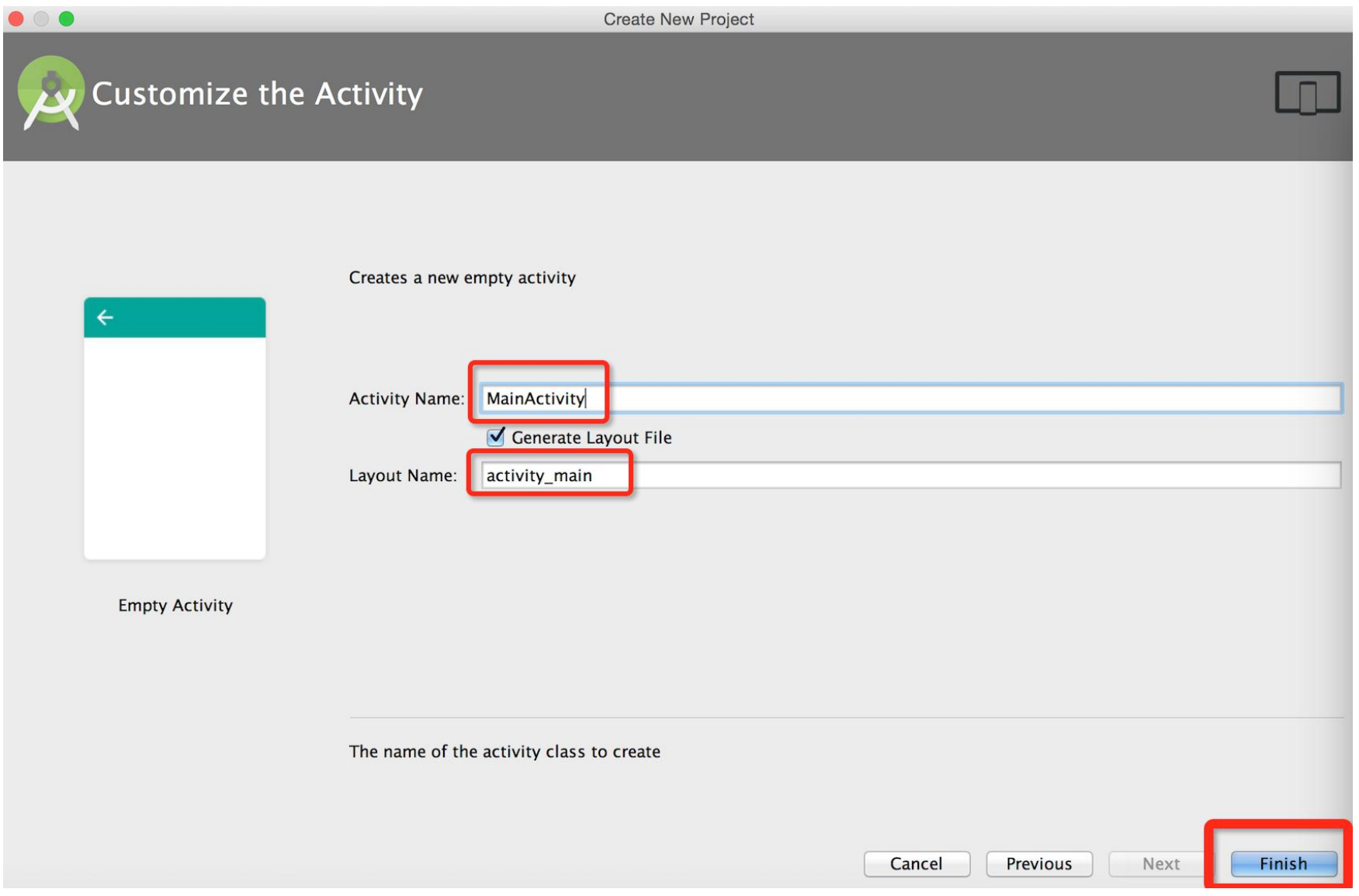
3、选择 Target Android Devices。本例中选择使用 MinimumSDK API 18（软编要求 MinimumSDK API 15 ； 硬编要求 MinimumSDK API 18）



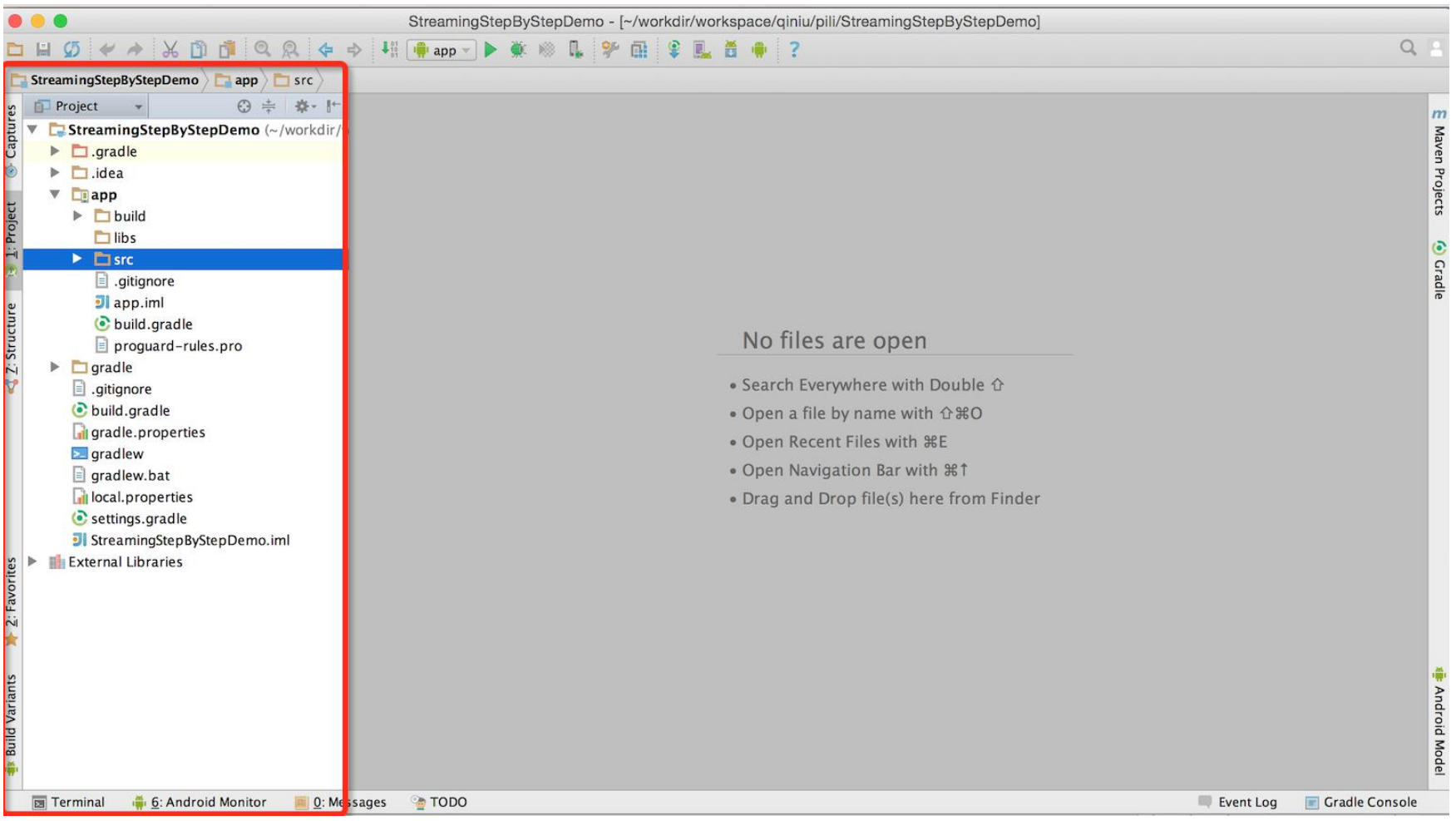
4、选择 Empty Activity



5、填写 Main Activity 信息，作为 android.intent.action.MAIN

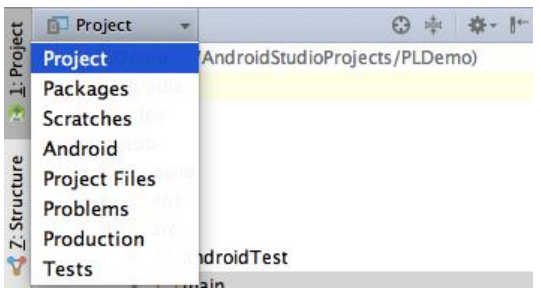


6、完成创建

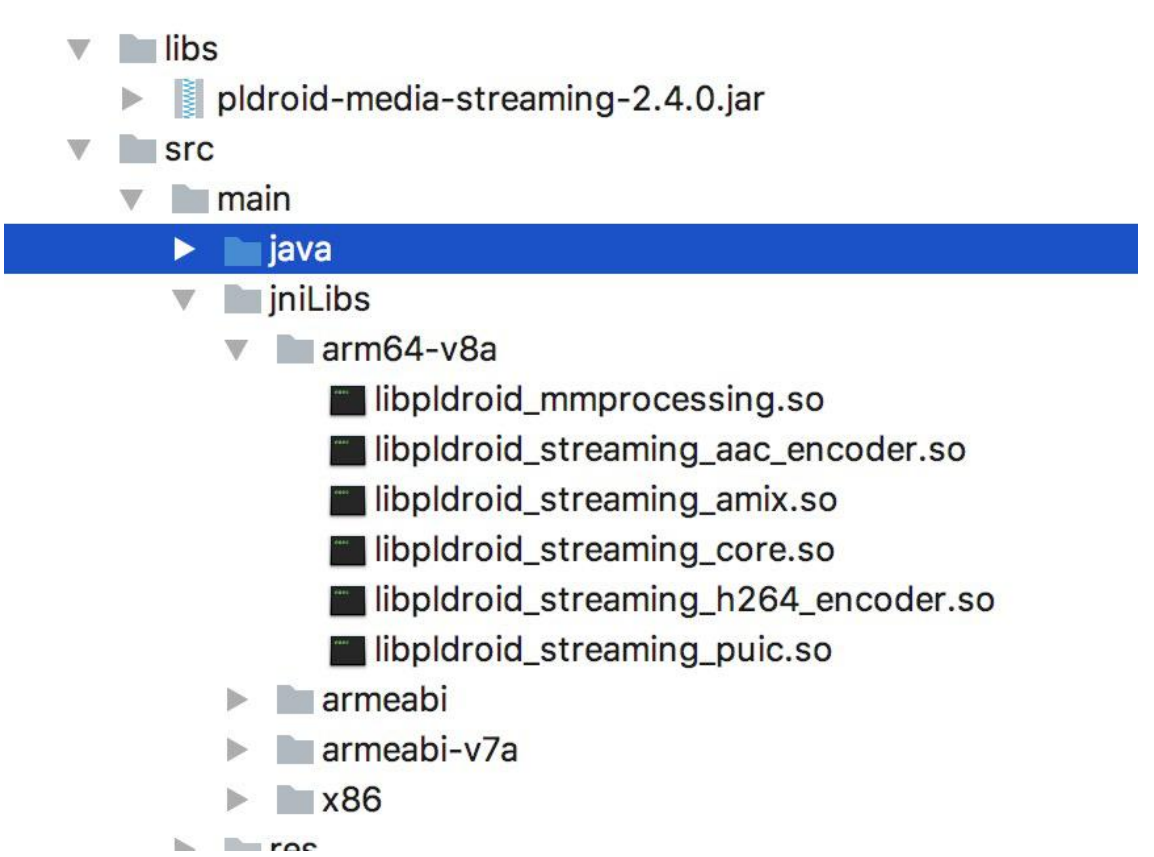


2.4.3 导入 SDK

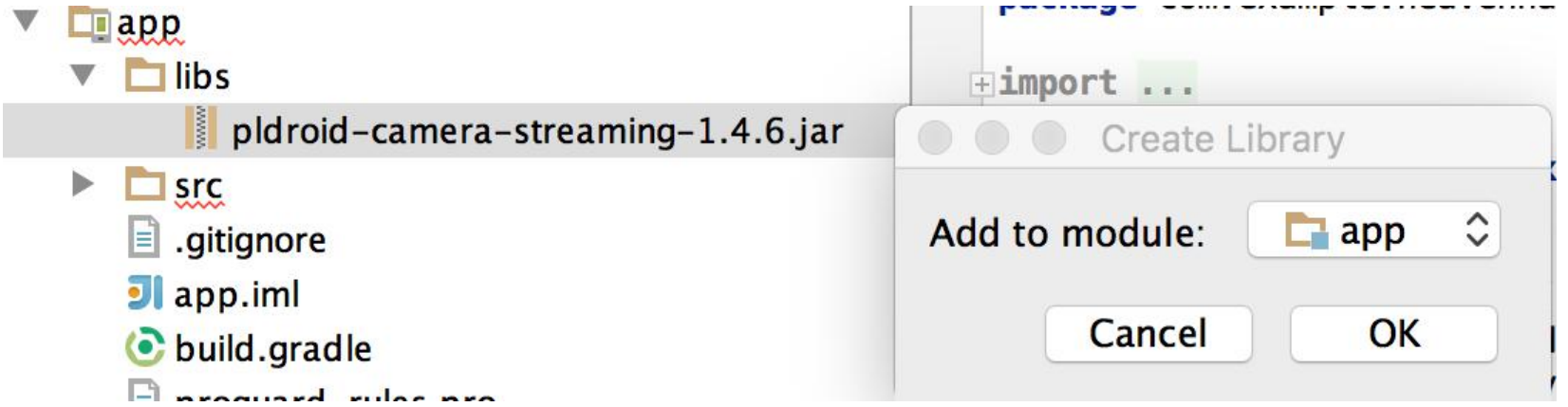
1、将右侧文件夹目录切换为 Project 视图



2、在 app/src/main 目录下创建 jniLibs 目录。按图所示，将文件导入对应的目录。



3、选中 lib 目录下 pldroid-media-streaming-2.4.0.jar，右键添加新建库，如图所示



4、导入完成，双击 build.gradle 文件查看内容，lib 目录中的文件已经自动导入，涉及的文件名如下：

```
// JAR

pldroid-media-streaming-2.4.0.jar


// SO

libpldroid_streaming_aac_encoder.so

libpldroid_streaming_core.so

libpldroid_streaming_h264_encoder.so

libpldroid_mmprocessing.so

libpldroid_streaming_amix.so

libpldroid_streaming_puic.so
```

2.5 创建基础推流实例

2.5.1 添加相关权限并注册 Activity

- 在 app/src/main 目录中的 AndroidManifest.xml 中增加 uses-permission 和 uses-feature 声明，并注册推流 Activity : SWCameraStreamingActivity

```
<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

    package="com.qiniu.pili.droid.streaming.demo" >



    <uses-permission android:name="android.permission.INTERNET" />

    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

    <uses-permission android:name="android.permission.RECORD_AUDIO" />

    <uses-permission android:name="android.permission.CAMERA" />

    <uses-permission android:name="android.permission.WAKE_LOCK" />

    <uses-feature android:name="android.hardware.camera.autofocus" />

    <uses-feature android:glEsVersion="0x00020000" android:required="true" />

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>



    <application

        android:allowBackup="true"

        android:name=".StreamingApplication"

        android:icon="@mipmap/ic_launcher"

        android:label="@string/app_id"

        android:supportsRtl="true"

        android:theme="@style/AppTheme" >

        <activity android:name=".MainActivity" >
```

```
        <intent-filter>

            <action android:name="android.intent.action.MAIN" />



            <category android:name="android.intent.category.LAUNCHER" />

        </intent-filter>

    </activity>



    <activity android:name=".StreamingByCameraActivity" >

        </activity>

    </application>



</manifest>
```

2.5.2 添加 happy-dns 依赖

检查在 app 目录下的 build.gradle ，并且按照如下修改：确认在 app 目录下，打开如图所示目录文件

```
apply plugin: 'com.android.application'



android {

compileSdkVersion 22

buildToolsVersion "22.0.1"



defaultConfig {

    applicationId "com.qiniu.pili.droid.streaming.demo"

    minSdkVersion 18

    targetSdkVersion 22

    versionCode 1
```

```
        versionid "1.0"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }


dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.qiniu:happy-dns:0.2.7'
    compile 'com.android.support:appcompat-v7:22.0.0'
    compile files('libs/pldroid-camera-streaming-2.1.0.jar')
}
```

2.5.3 实现自己的 Application

```
public class StreamingApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
        StreamingEnv.init(getApplicationContext());
    }
}
```

2.5.4 创建主界面

- 查看 app/src/main/java 目录中的 MainActivity.java 文件。为了演示方便，将文件中的 MainActivity 父类 AppCompatActivity 更改为 Activity，即 public class MainActivity extends AppCompatActivity，修改为 public class MainActivity extends Activity，MainActivity 其主要工作包括：
- 通过 start Button 去 app server 异步请求推流地址或者 stream Json（**注：当前推荐使用推流地址进行推流，stream Json 已不推荐使用**）
- 推流地址获取成功之后，启动 SWCameraStreamingActivity

```
public class MainActivity extends Activity {

    private static final String TAG = "MainActivity";


private String requestPublishURL() {

    try {

        // Replace "Your app server" by your app sever url which can get the publish URL as the SDK's input.

        HttpURLConnection httpConn = (HttpURLConnection) new URL("Your app server").openConnection();

        httpConn.setRequestMethod("POST");

        httpConn.setConnectTimeout(5000);

        httpConn.setReadTimeout(10000);

        int responseCode = httpConn.getResponseCode();

        if (responseCode != HttpURLConnection.HTTP_OK) {

            return null;

        }



        int length = httpConn.getContentLength();

        if (length <= 0) {

            return null;

        }

        InputStream is = httpConn.getInputStream();
```

```
            byte[] data = new byte[length];
            int read = is.read(data);
            is.close();
            if (read <= 0) {
                return null;
            }
            return new String(data, 0, read);
        } catch (Exception e) {
            showToast("Network error!");
        }
        return null;
    }


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button btn = (Button) findViewById(R.id.start);
        btn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                new Thread(new Runnable() {
                    @Override
                    public void run() {
                        try {
```

```
                    // Get the publish URL from http
                    String publishURL = requestPublishURL();


                    Intent intent = new Intent(MainActivity.this, StreamingByCameraActivity.class);
                    intent.putExtra("stream_publish_url", publishURL);
                    startActivity(intent);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
});
}
}
```

2.5.5 创建主界面布局文件

- 查看 app/src/main/res/layout 目录下的 activity_main.xml
- 底部 TAB 切换至 Text 面板，粘贴如下代码

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" android:layout_width="match_parent"
    android:layout_height="match_parent" android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin" tools:context=".MainActivity">
```

```
    <Button
        android:id="@+id/start"
        android:text="start"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</RelativeLayout>
```

2.5.6 创建推流界面

创建名为 SWCameraStreamingActivity 的 Empty Activity，SWCameraStreamingActivity 的主要工作包括：

- SWCameraStreamingActivity 获取 MainActivity 从 app server 获取到的推流地址
- 在 onCreate 通过 stream json 初始化推流 SDK 的核心类 MediaStreamingManager
- 在 onResume 中调用 mMediaStreamingManager.resume();
- 在接收到 READY 之后，开始推流 mMediaStreamingManager.startStreaming();，startStreaming 需要在非 UI 线程中进行操作。

在 app/src/main/java 目录下创建 StreamingByCameraActivity 文件，代码如下：

```
public class StreamingByCameraActivity extends Activity
        implements StreamingStateChangedListener, StreamStatusCallback, AudioSourceCallback,
StreamingSessionListener {

    CameraPreviewFrameView mCameraPreviewSurfaceView;
    private MediaStreamingManager mMediaStreamingManager;
    private StreamingProfile mProfile;
    private String TAG = "StreamingByCameraActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_camera_streaming);

        init();

}



private void init() {

        //get form you server

        String publishURLFromServer = "rtmpp://xxxx/xx/x";



mCameraPreviewSurfaceView = findViewById(R.id.cameraPreview_surfaceView);

try {

        //encoding setting

        mProfile = new StreamingProfile();

        mProfile.setVideoQuality(StreamingProfile.VIDEO_QUALITY_HIGH1)

                        .setAudioQuality(StreamingProfile.AUDIO_QUALITY_MEDIUM2)

                        .setEncodingSizeLevel(StreamingProfile.VIDEO_ENCODING_HEIGHT_480)

                        .setEncoderRCMode(StreamingProfile.EncoderRCModes.QUALITY_PRIORITY)

                        .setPublishUrl(publishURLFromServer);



        //preview setting

        CameraStreamingSetting camerasetting = new CameraStreamingSetting();

        camerasetting.setCameraId(Camera.CameraInfo.CAMERA_FACING_BACK)

                        .setContinuousFocusModeEnabled(true)

                        .setCameraPrvSizeLevel(CameraStreamingSetting.PREVIEW_SIZE_LEVEL.MEDIUM)

                        .setCameraPrvSizeRatio(CameraStreamingSetting.PREVIEW_SIZE_RATIO.RATIO_16_9);
```

```
            //streaming engine init and setListener
            mMediaStreamingManager = new MediaStreamingManager(this, mCameraPreviewSurfaceView,
AVCodecType.SW_VIDEO_WITH_SW_AUDIO_CODEC);  // soft codec
            mMediaStreamingManager.prepare(camerasetting, mProfile);
            mMediaStreamingManager.setStreamingStateListener(this);
            mMediaStreamingManager.setStreamingSessionListener(this);
            mMediaStreamingManager.setStreamStatusCallback(this);
            mMediaStreamingManager.setAudioSourceCallback(this);
        } catch (URISyntaxException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void onStateChanged(StreamingState streamingState, Object extra) {

        Log.e(TAG, "streamingState = " + streamingState + "extra = " + extra);
        switch (streamingState) {
            case PREPARING:
                Log.e(TAG, "PREPARING");
                break;
            case READY:
                Log.e(TAG, "READY");
                // start streaming when READY
```

```
new Thread(new Runnable() {
    @Override
    public void run() {
        if (mMediaStreamingManager != null) {
            mMediaStreamingManager.startStreaming();
        }
    }
}).start();
        break;
    case CONNECTING:
        Log.e(TAG, "连接中");
        break;
    case STREAMING:
        Log.e(TAG, "推流中");
        // The av packet had been sent.
        break;
    case SHUTDOWN:
        Log.e(TAG, "直播中断");
        // The streaming had been finished.
        break;
    case IOERROR:
        // Network connect error.
        Log.e(TAG, "网络连接失败");
        break;
    case OPEN_CAMERA_FAIL:
```

```
            Log.e(TAG, "摄像头打开失败");
            // Failed to open camera.
            break;
        case DISCONNECTED:
            Log.e(TAG, "已经断开连接");
            // The socket is broken while streaming
            break;
        case TORCH_INFO:
            Log.e(TAG, "开启闪光灯");
            break;

    }
}

@Override
protected void onResume() {
    super.onResume();
    mMediaStreamingManager.resume();
}

@Override
protected void onPause() {
    super.onPause();
    // You must invoke pause here.
    mMediaStreamingManager.pause();
```

```
}



@Override

public void notifyStreamStatusChanged(StreamingProfile.StreamStatus status) {

    Log.e(TAG, "StreamStatus = " + status);

}



@Override

public void onAudioSourceAvailable(ByteBuffer srcBuffer, int size, long tsInNanoTime, boolean isEof) {



}



@Override

public boolean onRecordAudioFailedHandled(int code) {

    Log.i(TAG, "onRecordAudioFailedHandled");

    return false;

}



@Override

public boolean onRestartStreamingHandled(int code) {

    Log.i(TAG, "onRestartStreamingHandled");

    new Thread(new Runnable() {

        @Override

        public void run() {

            if (mMediaStreamingManager != null) {
```

```
                    mMediaStreamingManager.startStreaming();
                }
            }
        }).start();
        return false;
    }

    @Override
    public void onBackPressed() {
        super.onBackPressed();
        Intent intent = new Intent(this, MainActivity.class);
        startActivity(intent);
    }

    @Override
    public Camera.Size onPreviewSizeSelected(List<Camera.Size> list) {
        return null;
    }

    @Override
    public int onPreviewFpsSelected(List<int[]> list) {
        return -1;
    }
```

2.5.7 创建推流界面布局文件

- 查看 app/src/main/res/layout 中的 activity_swcamera_streaming.xml
- 切换至 Text 面板，粘贴如下内容

```
<?xml version="1.0" encoding="utf-8"?>



<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"

    xmlns:tools="http://schemas.android.com/tools"

    android:id="@+id/content"

    android:layout_width="match_parent"

    android:layout_height="match_parent"

    android:background="@color/background_floating_material_dark"

    tools:context=".SWCameraStreamingActivity" >



    <com.qiniu.pili.droid.streaming.demo.ui.CameraPreviewFrameView

        android:id="@+id/cameraPreview_surfaceView"

        android:layout_width="match_parent"

        android:layout_height="match_parent"

        android:layout_gravity="center" />

</RelativeLayout>
```

启动 APP 之后，当点击 start button，就可以开始推流了。

2.5.8 测试播放效果

- 测试方法: 从 app server 获取到推流对应的播放地址，输入到播放器中进行播放。

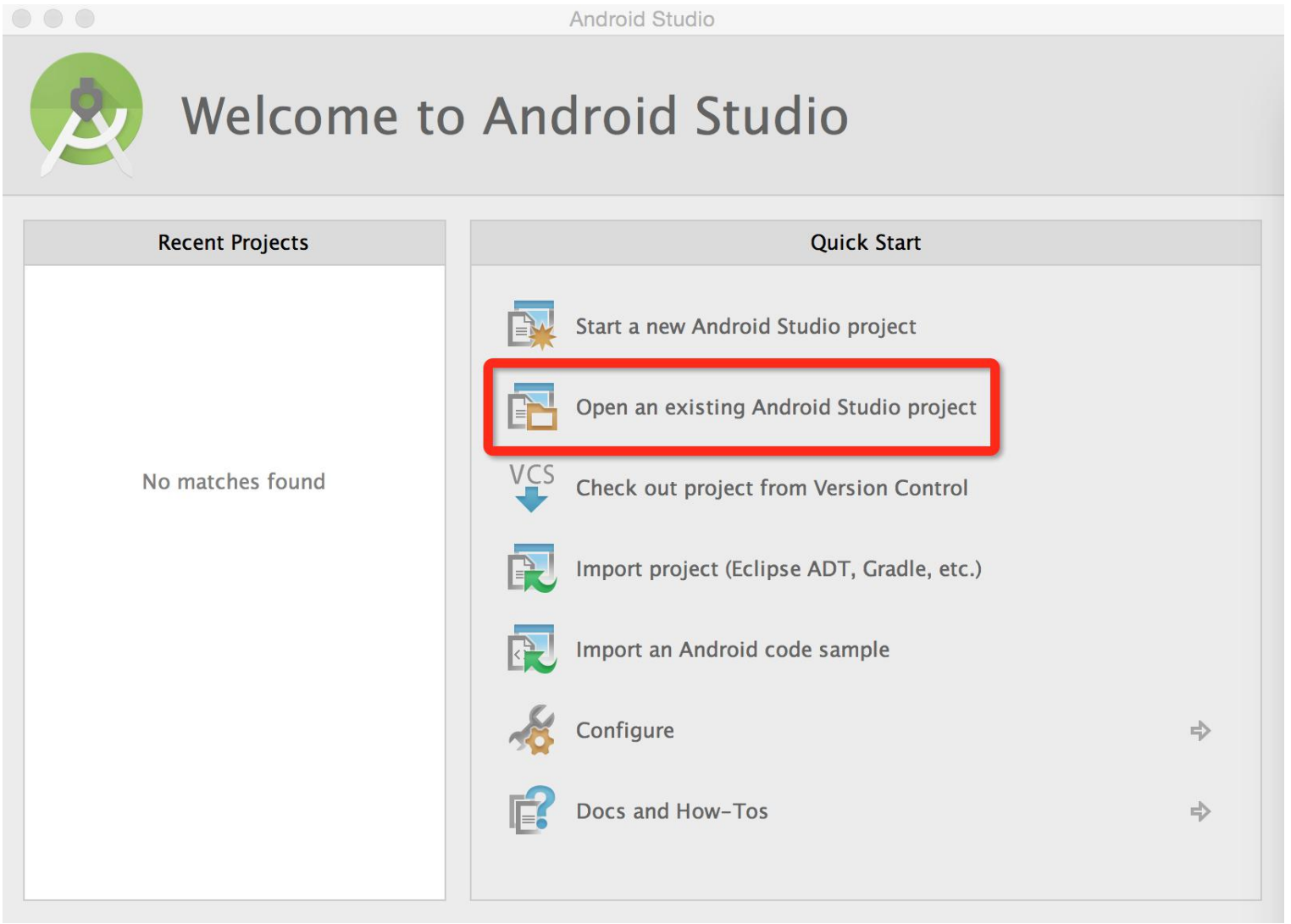
2.6 从 SDK DEMO 开始使用

2.6.1 下载 SDK DEMO

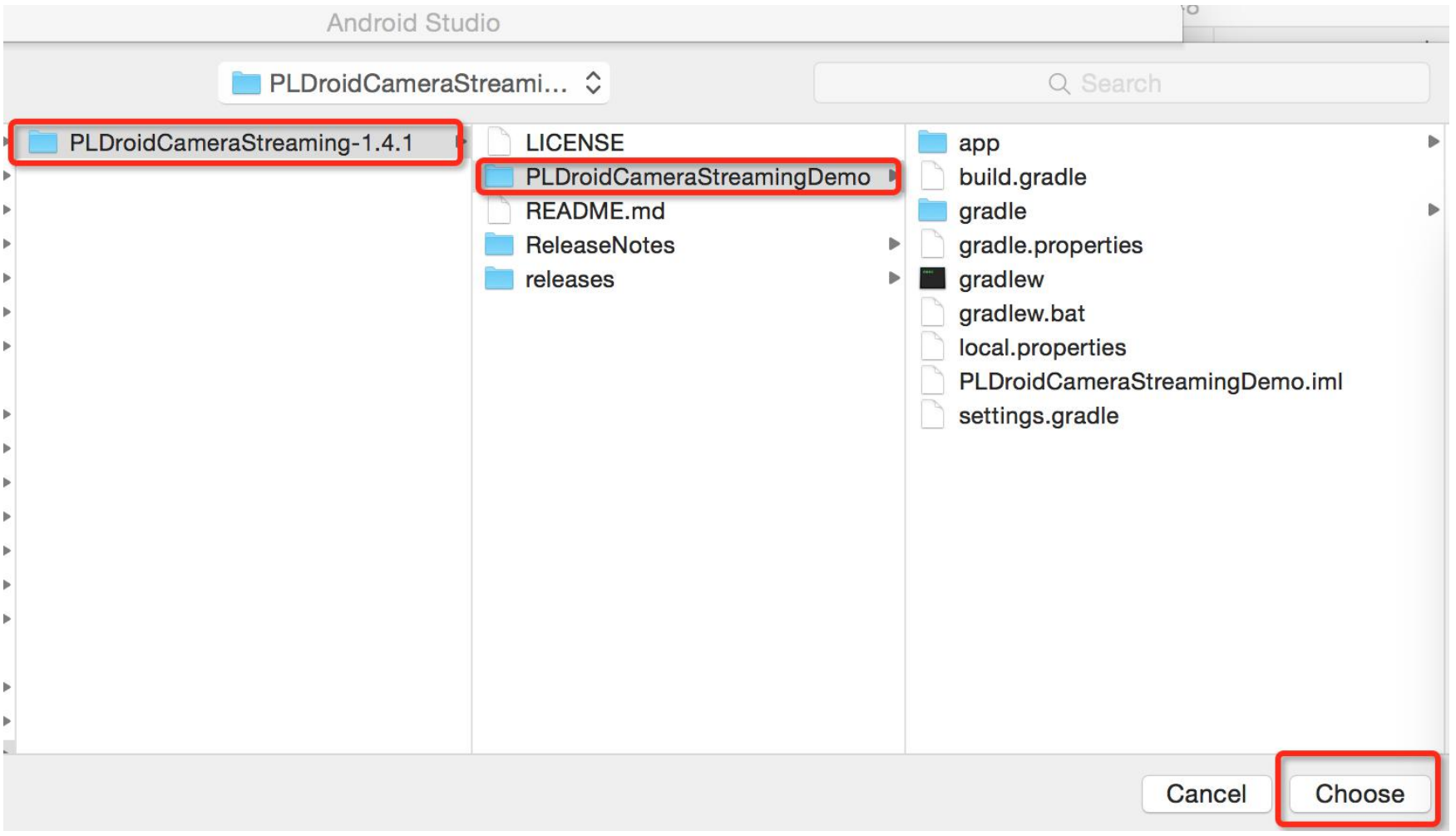
- 点击这里下载 SDK Demo。

2.6.2 导入 Project 到 Android Studio

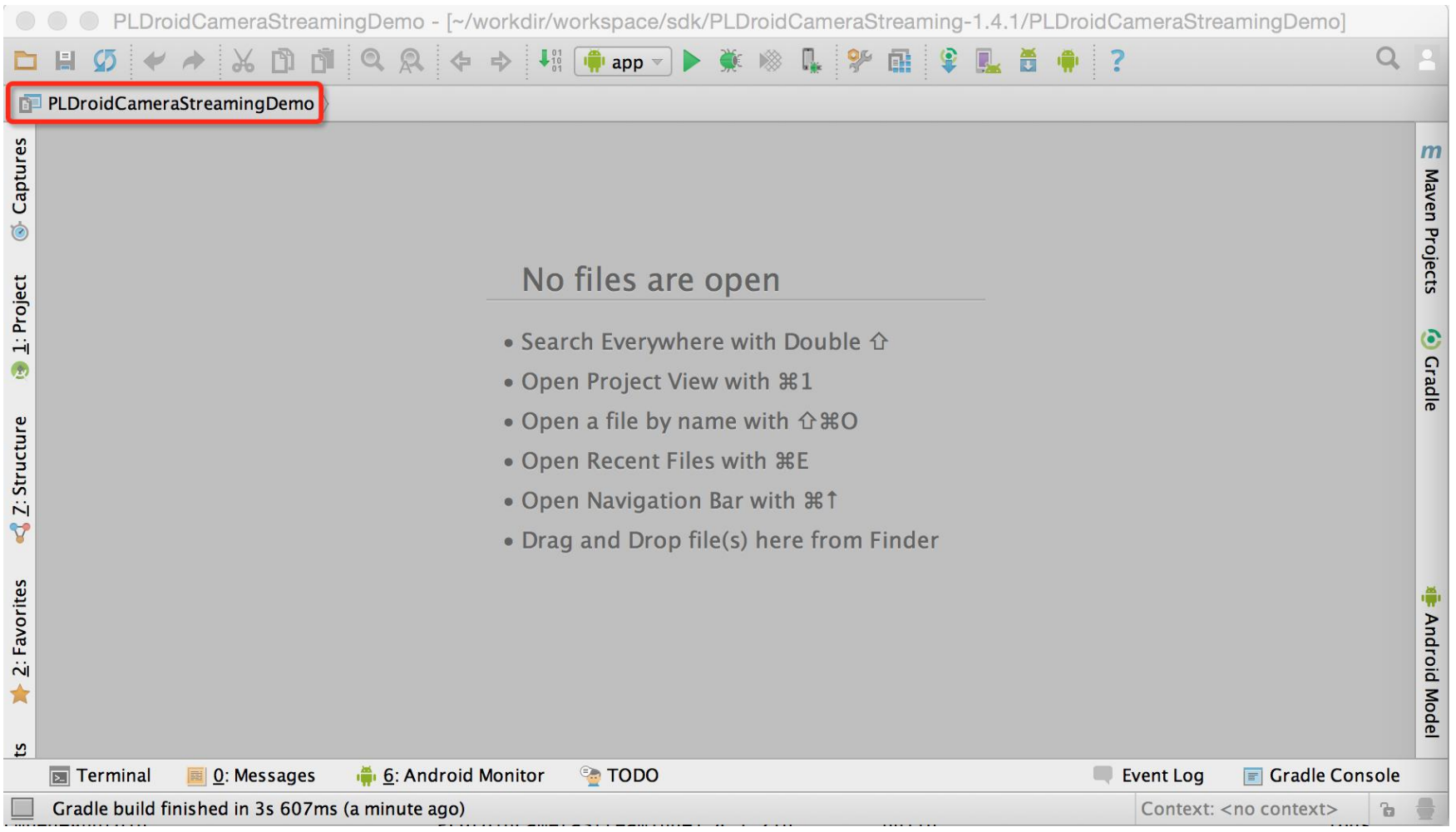
1、启动 Android Studio 并选择 Open an existing Android Studio project。



2、选择 PLDroidCameraStreamingDemo 工程。从目录 ~/workdir/workspace/sdk/PLDroidCameraStreaming-1.4.1/ 中选择 DroidCameraStreamingDemo 工程，选择完成后，点击 Choose 按钮。



3、恭喜你，导入完成。



2.6.3 已有工程导入 SDK

SDK 下载地址。

Demo Project 目录结构说明。如下图所示：

红色框：是 Demo 依赖 SDK 的 JAR 文件。您也可以在 build.gradle 中自定义其路径：

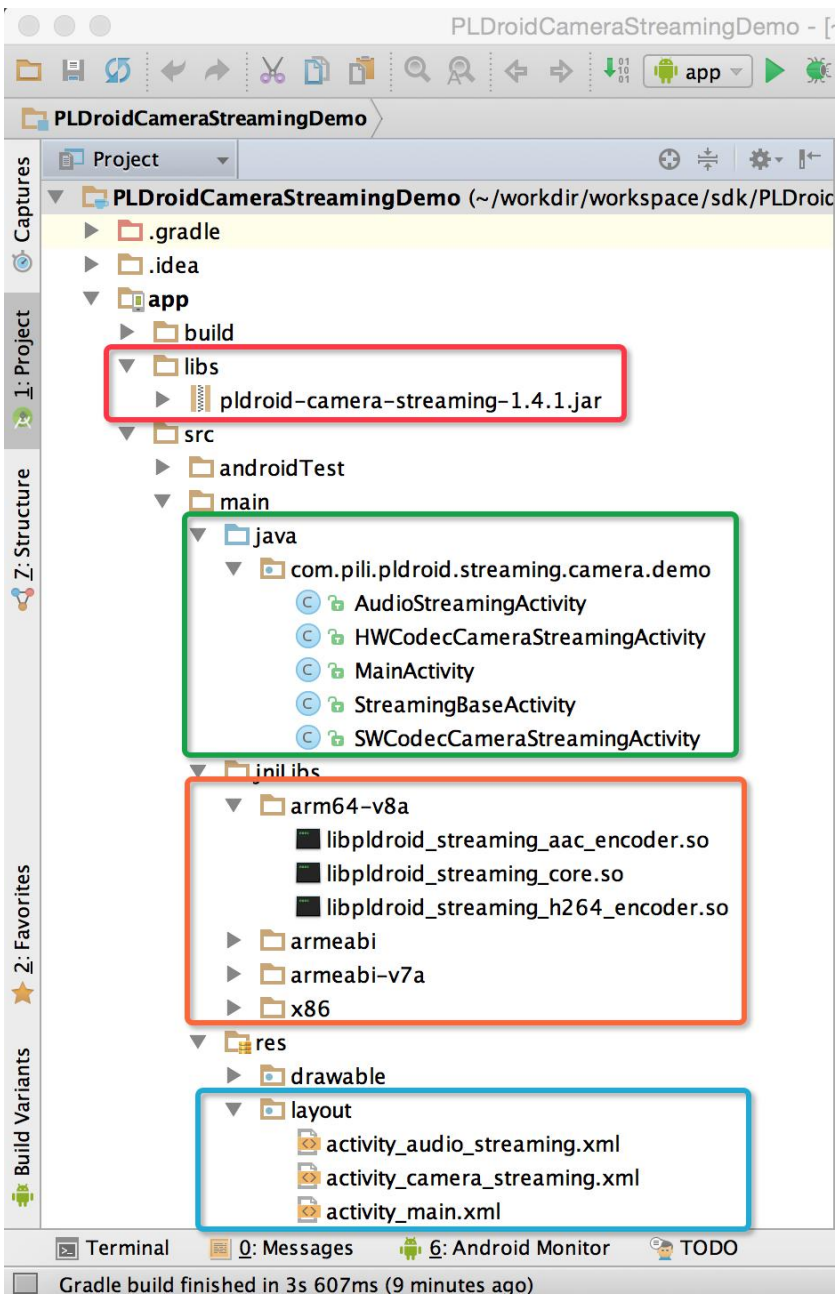
```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.qiniu:happy-dns:0.2.7'
    compile files('libs/pldroid-camera-streaming-1.4.1.jar')
}
```

绿色框：是 Demo Java 代码部分。

- HWCodecCameraStreamingActivity 是硬编的样例代码；
- SWCodecCameraStreamingActivity 是软编的样例代码；
- AudioStreamingActivity 是纯音频的样例代码

橙色框：是 Demo 依赖 SDK 的动态链接库文件。目前 SDK 支持主流的 ARM, ARMv7a, ARM64v8a, X86 芯片体系架构。

蓝色框：是 Demo 的布局文件。



2.7 功能使用

2.7.1 摄像头参数配置

所有摄像头相关的配置，都在 CameraStreamingSetting 类中进行。

2.7.1.1 前置／后置摄像头配置

前后置摄像头配置：

```
mCameraStreamingSetting.setCameraId(Camera.CameraInfo.CAMERA_FACING_FRONT); // 前置摄像头

mCameraStreamingSetting.setCameraId(Camera.CameraInfo.CAMERA_FACING_BACK); // 后置摄像头
```

注：默认 CAMERA_FACING_BACK

若设定的摄像头打开失败，将会回调 StreamingState.OPEN_CAMERA_FAIL .

2.7.1.2 Camera 对焦相关

- setContinuousFocusModeEnabled

  若希望关闭自动对焦功能，可以：

```
mCameraStreamingSetting.setContinuousFocusModeEnabled(false);
```

注：默认开启

- 通过 setFocusMode 设置对焦模式

您可以设置不同的对焦模式，目前 SDK 支持的对焦模式有：

```
– CameraStreamingSetting.FOCUS_MODE_CONTINUOUS_PICTURE // 自动对焦（Picture）

– CameraStreamingSetting.FOCUS_MODE_CONTINUOUS_VIDEO    // 自动对焦（Video）

– CameraStreamingSetting.FOCUS_MODE_AUTO                // 手动对焦
```

FOCUS_MODE_CONTINUOUS_PICTURE 与 FOCUS_MODE_CONTINUOUS_VIDEO 最大的区别在于，FOCUS_MODE_CONTINUOUS_PICTURE 对焦会比 FOCUS_MODE_CONTINUOUS_VIDEO 更加频繁，功耗会更高，建议使用 FOCUS_MODE_CONTINUOUS_VIDEO。

可以通过 CameraStreamingSetting#setFocusMode 来设置不同的对焦模式：

```
mCameraStreamingSetting.setFocusMode(CameraStreamingSetting.FOCUS_MODE_CONTINUOUS_PICTURE);

mCameraStreamingSetting.setFocusMode(CameraStreamingSetting.FOCUS_MODE_CONTINUOUS_VIDEO);

mCameraStreamingSetting.setFocusMode(CameraStreamingSetting.FOCUS_MODE_AUTO);
```

注：默认值为 CameraStreamingSetting.FOCUS_MODE_CONTINUOUS_VIDEO

- setResetTouchFocusDelayInMs

  当对焦模式处理 CameraStreamingSetting.FOCUS_MODE_AUTO，并触发了手动对焦之后，若希望隔一段时间恢复到自动对焦模式，就可以调用：

```
mCameraStreamingSetting.setResetTouchFocusDelayInMs(3000); // 单位毫秒
```

注：默认值为 3000 ms

2.7.1.3 Camera 预览 size

为了兼容更多的 Camera ，SDK 采用了相关的措施：

- 使用 PREVIEW_SIZE_LEVEL 和 PREVIEW_SIZE_RATIO 共同确定一个 Preview Size.
- 用户也可以通过 StreamingSessionListener#onPreviewSizeSelected 自定义选择一个合适的 Preview Size，onPreviewSizeSelected 的参数 list 是 Camera 系统支持的 preview size 列表（Camera.Parameters#getSupportedPreviewSizes()），SDK 内部会对 list 做从小到大的排序，可以安全地使用 list 中的 preview size。如果 onPreviewSizeSelected 返回为 null，代表放弃自定义选择，那么 SDK 会使用前面的策略选择一个合适的 Preview Size，否则使用 onPreviewSizeSelected 的返回值。

SDK 目前分别支持的 PREVIEW_SIZE_LEVEL 和 PREVIEW_SIZE_RATIO 分别是：

```
// PREVIEW_SIZE_LEVEL

– SMALL

– MEDIUM

– LARGE


// PREVIEW_SIZE_RATIO

– RATIO_4_3

– RATIO_16_9
```

需要注意的是：在某些机型上，list 可能为 null。

2.7.1.4 Camera 预览 FPS

- SDK 默认会根据推流帧率自动选择一个合适的采集帧率.
- 用户也可以通过 StreamingSessionListener#onPreviewFpsSelected 自定义选择一个合适的预览 FPS，onPreviewFpsSelected 的参数 list 是 Camera 系统支持的预览 FPS 列表

（Camera.Parameters#getSupportedPreviewFpsRange()）。如果 onPreviewFpsSelected 返回为 -1，代表放弃自定义选择，那么 SDK 会使用前面的策略选择一个合适的预览 FPS，否则使用 onPreviewFpsSelected 的返回值。

2.7.1.5 RecordingHint

可以通过 CameraStreamingSetting 对象设置 Recording hint，以此来提升数据源的帧率。

```
CameraStreamingSetting setting = new CameraStreamingSetting();

setting.setRecordingHint(false);
```

需要注意的是，在部分机型开启 Recording Hint 之后，会出现画面卡帧等风险，所以请慎用该 API。如果需要实现高 fps 推流，可以考虑开启并加入白名单机制。

2.7.1.6 Encoding Mirror

对于有对前置摄像头进行 Mirror 操作的用户来说，只需通过 CameraStreamingSetting#setFrontCameraMirror 设置即可。该操作目前仅针对播放端有效。可以避免前置摄像头拍摄字体镜像反转问题。

```
boolean isMirror = xxx; // false as default

mCameraStreamingSetting.setFrontCameraMirror(isMirror);
```

2.7.1.7 内置美颜

内置美颜流程的开启通过 CameraStreamingSetting#setBuiltInFaceBeautyEnabled(boolean eanble) 进行，注意，若希望自定义美颜，需要 disable 该接口，否则行为未知。

在初始化 CameraStreamingSetting 的时候，可以初始化对应的美颜参数：

```
// FaceBeautySetting 中的参数依次为：beautyLevel，whiten，redden，即磨皮程度、美白程度以及红润程度，取值范围为[0.0f, 1.0f]

mCameraStreamingSetting.setFaceBeautySetting(new CameraStreamingSetting.FaceBeautySetting(1.0f, 1.0f, 0.8f))
        .setVideoFilter(CameraStreamingSetting.VIDEO_FILTER_TYPE.VIDEO_FILTER_BEAUTY)
```

2.7.2 麦克风参数配置

所有麦克风相关的配置，都在 MicrophoneStreamingSetting 类中进行。

若希望增加蓝牙麦克风的支持，需要：

```
mMicrophoneStreamingSetting.setBluetoothSCOEnabled(true);
```

注：默认值为 false

2.7.3 推流参数设置

所有推流相关的参数配置，都在 StreamingProfile 类中进行。

2.7.3.1 推流地址配置

streamJsonStrFromServer 是由服务端返回的一段 JSON String，该 JSON String 描述了 Stream 的结构。通常，您可以使用 Pili 服务端 SDK 的 getStream(streamId) 方法来获取一个 Stream 对象，在服务端并将该对象以 JSON String 格式输出，该输出即是 streamJsonStrFromServer 变量的内容。从业务服务器获取对应的 Stream 可参考 MainActivity.java 中的 requestByHttpPost 方法。

```
String streamJsonStrFromServer = "stream json string from your server";



JSONObject streamJson = null;

try {

    streamJson = new JSONObject(streamJsonStrFromServer);

} catch (JSONException e) {

    e.printStackTrace();

}

mStreamingProfile.setStream(new Stream(streamJson));
```

设置 Stream 只需要保证在 MediaStreamingManager#startStreaming() 之前调用即可，SDK 接受以下调用流程：

```
mStreamingProfile.setStream(new Stream(mJSONObject1));

mMediaStreamingManager.setStreamingProfile(mProfile);

mMediaStreamingManager.startStreaming();
```

推流 Url

从 v2.0.0 开始，在加入推流域名白名单之后，可以使用如下接口直接推流：

```
mProfile.setPublishUrl("rtmp://xxx.xxx/xxx/xxx");
```

2.7.3.2 VideoQuality

VideoQuality 是对视频质量相关参数的一个抽象，它的值对应的参数表如下：

Level	Fps	Video Bitrate(Kbps)
VIDEO_QUALITY_LOW1	12	150
VIDEO_QUALITY_LOW2	15	264
VIDEO_QUALITY_LOW3	15	350
VIDEO_QUALITY_MEDIUM1	30	512
VIDEO_QUALITY_MEDIUM2	30	800
VIDEO_QUALITY_MEDIUM3	30	1000
VIDEO_QUALITY_HIGH1	30	1200
VIDEO_QUALITY_HIGH2	30	1500
VIDEO_QUALITY_HIGH3	30	2000

你只需要通过 StreamingProfile#setVideoQuality 进行设置即可，如：

```
mStreamingProfile.setVideoQuality(StreamingProfile.VIDEO_QUALITY_MEDIUM1);
```

其含义为，设置视频的 fps 为 30，码率为 512 kbps

2.7.3.3 AudioQuality

AudioQuality 是对音频质量相关参数的一个抽象，它的值对应的参数表如下：

Level	Audio Bitrate(Kbps)	Audio Sample Rate(Hz)
AUDIO_QUALITY_LOW1	18	44100
AUDIO_QUALITY_LOW2	24	44100
AUDIO_QUALITY_MEDIUM1	32	44100
AUDIO_QUALITY_MEDIUM2	48	44100
AUDIO_QUALITY_HIGH1	96	44100
AUDIO_QUALITY_HIGH2	128	44100

你只需要通过 StreamingProfile#setAudioQuality 进行设置即可，如：

```
mStreamingProfile.setAudioQuality(StreamingProfile.AUDIO_QUALITY_HIGH1);
```

其含义为，设置音频的采样率为 44100 HZ，码率为 96 kbps。

2.7.3.4 AVProfile

当需要自定义 video 的 fps、bitrate、profile 或者 audio 的 sample rate、bitrate，可以通过 AVProfile 设置。

其中 video profile 有以下选项

Level	Comment
StreamingProfile.H264Profile.HIGH	更好的质量，但有一些性能的损耗
StreamingProfile.H264Profile.MAIN	质量与性能较好的平衡
StreamingProfile.H264Profile.BASELINE	更好的性能，但一般的质量

AVProfile 的使用例子如下

```
// audio sample rate is 44100, audio bitrate is 48 * 1024 bps
StreamingProfile.AudioProfile aProfile = new StreamingProfile.AudioProfile(44100, 48 * 1024);

// fps is 20, video bitrate is 1000 * 1024 bps, maxKeyFrameInterval is 60, profile is HIGH
StreamingProfile.VideoProfile vProfile = new StreamingProfile.VideoProfile(20, 1000 * 1024, 60,
StreamingProfile.H264Profile.HIGH);

StreamingProfile.AVProfile avProfile = new StreamingProfile.AVProfile(vProfile, aProfile);

mStreamingProfile.setAVProfile(avProfile)
```

注：44100 是 Android 平台唯一保证所以设备支持的采样率，为了避免音频兼容性问题，建议设置为 44100。

StreamingProfile#setAVProfile 的优先级高于 Quality，也就是说，当同时调用了 Quality 和 AVProfile 的设置，AVProfile 会覆盖 Quality 的设置值，比如：

```
StreamingProfile.AudioProfile aProfile = new StreamingProfile.AudioProfile(44100, 48 * 1024);
StreamingProfile.VideoProfile vProfile = new StreamingProfile.VideoProfile(20, 1000 * 1024, 60);
StreamingProfile.AVProfile avProfile = new StreamingProfile.AVProfile(vProfile, aProfile);

mStreamingProfile.setAVProfile(avProfile)
                        .setVideoQuality(StreamingProfile.VIDEO_QUALITY_MEDIUM1) // |30|512|90|
```

```
                .setAudioQuality(StreamingProfile.AUDIO_QUALITY_HIGH1);  // |96|44100|
```

最终设定的值应该为：

- 音频：48 * 1024, 44100
- 视频：20, 1000 * 1024, 60

2.7.3.5 HappyDns 支持

为了防止 Dns 被劫持，SDK 加入了 HappyDns 支持，可以从这里查阅源码。 从 v1.4.6 版本开始，需要在宿主项目中的 build.gradle 中加入如下语句：

```
dependencies {

    ...

    compile 'com.qiniu:happy-dns:0.2.7'

    ...

}
```

通过 StreamingProfile 设定自定义 DnsManager，如下：

```
public static DnsManager getMyDnsManager() {

        IResolver r0 = new DnspodFree();

        IResolver r1 = AndroidDnsServer.defaultResolver();

        IResolver r2 = null;

        try {

            r2 = new Resolver(InetAddress.getByName("119.29.29.29"));

        } catch (IOException ex) {

            ex.printStackTrace();

        }

        return new DnsManager(NetworkInfo.normal, new IResolver[]{r0, r1, r2});

}
```

```
StreamingProfile mProfile = new StreamingProfile();

// Setting null explicitly, means give up {@link DnsManager} and access by the original host.

mStreamingProfile.setDnsManager(getMyDnsManager()); // set your DnsManager
```

若显示地设置为 null，即：

```
mStreamingProfile.setDnsManager(null);
```

SDK 会使用系统 DNS 解析，而不会使用 DnsManager 来进行 Dns 解析。

若不调用 StreamingProfile#setDnsManager 方法，SDK 会默认的创建一个 DnsManager 来对 Dns 进行解析。

2.7.3.6 软编的 EncoderRCModes

目前 RC mode 支持的类型：

- EncoderRCModes.QUALITY_PRIORITY: 质量优先，实际的码率可能高于设置的码率
- EncoderRCModes.BITRATE_PRIORITY: 码率优先，更精确地码率控制

可通过 StreamingProfile 的 setEncoderRCMode 方法进行设置，如下：

```
mStreamingProfile.setEncoderRCMode(StreamingProfile.EncoderRCModes.QUALITY_PRIORITY);
```

注：默认值为 EncoderRCModes.QUALITY_PRIORITY

2.7.3.7 Encoding size 的设定

SDK 将 Preview size 和 Encoding size 已经分离，他们之间互不影响。Preview size 代表的是 Camera 本地预览的 size，encoding size 代表的是编码时候的 size，即播放端不做处理时候看到视频的 size。

有两种方式可以设置 Encoding size：

- 使用内置的 encoding size level：StreamingProfile#setEncodingSizeLevel
- 设定一个 encoding size 偏好值：StreamingProfile#setPreferredVideoEncodingSize(int, int)

SDK 会秉持一个原则，用户偏好值的优先级会高于内置的设置。若同时调用了上述 api，偏好设置会覆盖内置的 encoding size level。

内置的 Encoding size level 对应的分辨率：

Level	Resolution(16:9)	Resolution(4:3)
VIDEO_ENCODING_HEIGHT_240	424 x 240	320 x 240
VIDEO_ENCODING_HEIGHT_480	848 x 480	640 x 480
VIDEO_ENCODING_HEIGHT_544	960 x 544	720 x 544
VIDEO_ENCODING_HEIGHT_720	1280 x 720	960 x 720

Level	Resolution(16:9)	Resolution(4:3)
VIDEO_ENCODING_HEIGHT_1088	1920 x 1088	1440 x 1088

2.7.3.8 StreamStatus 反馈配置

对于推流信息的反馈，可以通过 StreamingProfile#setStreamStatusConfig 来设定其反馈的频率，

```
mStreamingProfile#setStreamStatusConfig(new StreamingProfile.StreamStatusConfig(3)); // 单位为秒
```

其含义为，若注册了 mMediaStreamingManager.setStreamStatusCallback ，每隔 3 秒回调 StreamStatus 信息。

2.7.3.9 动态更改 Orientation

动态更改 Orientation，包括动态更改 Encoding Orientation 以及动态切换横竖屏。

- 动态更改 Encoding Orientation

更改的是编码后图像的方向，但需要重新推流才会生效；目前支持的 ENCODING_ORIENTATION 的类型有：PORT 和 LAND

用户不设置的情况下， Encoding Orientation 会依赖 Activity 的 Orientation，即有如下对应关系：

// 不调用 setEncodingOrientation 情况下，SDK 会默认根据如下关系进行设置 Encoding Orientation

Activity Orientation	->	Encoding Orientation
ActivityInfo.SCREEN_ORIENTATION_PORTRAIT	->	PORT
ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE	->	LAND

设置 ENCODING_ORIENTATION.PORT 之后，播放端会观看竖屏的画面；

设置 ENCODING_ORIENTATION.LAND 之后，播放端会观看横屏的画面。

其设置方式如下：

```
mProfile.setEncodingOrientation(StreamingProfile.ENCODING_ORIENTATION.PORT);
mMediaStreamingManager.setStreamingProfile(mProfile); // notify MediaStreamingManager that StreamingProfile had
been changed.
```

- 动态切换横竖屏

  用户切换 Activity 方向后，相应地调整 Camera 的预览显示效果。

  在更改了 Activity Orientation 之后，需要调用 notifyActivityOrientationChanged 通知 MediaStreamingManager。

比如：

```
setRequestedOrientation(isEncOrientationPort ? ActivityInfo.SCREEN_ORIENTATION_PORTRAIT :
ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);

mMediaStreamingManager.notifyActivityOrientationChanged();
```

需要注意的是，为了防止 setRequestedOrientation 调用后 Activity 的重建，需要在 AndroidManifest.xml 里面配置对应的 Activity 的如下属性：

```
android:configChanges="orientation|screenSize"
```

可查看 Demo 中的 .SWCodecCameraStreamingActivity 的配置。

2.7.3.10 自适应码率

由于无线网络相对于有线网络，可靠性较低，会经常遇到信号覆盖不佳导致的高丢包、高延时等问题，特别是在用网高峰期，由于带宽有限，网络拥塞的情况时有发生。自 v2.1.0 起，PLDroidMediaStreaming 提供了对应的优化方案。可以通过如下 API 开启自适应码率(注意：SDK 默认是关闭自适应码率的)：

```
/**
 * adaptive bitrate adjust mode
 */
public enum BitrateAdjustMode {
    Auto,    // SDK 自适应码率调节
    Manual, // 用户自己实现码率调节，范围：10kbps~10Mbps
    Disable // 关闭码率调节
}

mProfile.setBitrateAdjustMode(StreamingProfile.BitrateAdjustMode.Auto);
```

开启自适应码率后，当 SDK 检测到网络状况差时，会尝试降低码率，直到 StreamingProfile.VIDEO_QUALITY_LOW1 (150 Kbps)；反之，会提升码率，直到用户设定的 Target Bitrate（通过 StreamingProfile#setVideoQuality 或 StreamingProfile#VideoProfile 设定的码率值）

另外，可以通过下面接口控制自适应码率调节的范围:

```
/**
 * Sets the range value of video adaptive bitrate.
 *
 * @param minBitrate min bitrate, unit: bps
 * @param maxBitrate max bitrate, unit: bps
 * */
```

```
public StreamingProfile setVideoAdaptiveBitrateRange(int minBitrate, int maxBitrate)
```

此时，码率调节策略为：

- 用户网络没有触发自动码率调节，码率一直保持在 Target Bitrate（通过 StreamingProfile#setVideoQuality 或 StreamingProfile#VideoProfile 设定的码率值）附近
- 触发自动码率调节后：
    - 向下：
        - 逐级调整，直到自适应码率调节范围的下限(minBitrate)
    - 向上：
        - 如果调节范围的上限（maxBitrate）大于 Target Bitrate，最高会调整到 Target Bitrate
        - 如果调节范围的上限（maxBitrate）小于 Target Bitrate，最高会调整到调节范围的上限(maxBitrate)

2.7.3.11 FilterMode 参数设置

- 当图像采集尺寸与推流尺寸不一致时，SDK 会对采集图像进行 resize 操作，通过 FilterMode 参数，可以对 resize 算法进行设置。
- 相关接口如下：

```
/**

 * filter mode for libyuv

 */

public enum YuvFilterMode {

    None,           // Point sample; Fastest.

    Linear,         // Filter horizontally only.

    Bilinear,       // Faster than box, but lower quality scaling down.

    Box             // Highest quality.

}



/**

 * Sets the YUV filter mode

 * @param mode the {@link YuvFilterMode}, default: {@link YuvFilterMode#None}

 * @return this

 */
```

```
public StreamingProfile setYuvFilterMode(YuvFilterMode mode)
```

- 该参数只对软编和硬编 YUV 有效

2.7.4 水印设置

所有水印相关的配置，都在 WatermarkSetting 类中进行。

2.7.4.1 水印位置信息

水印的位置信息，目前内置四个方位，如：

```
public enum WATERMARK_LOCATION {

    NORTH_WEST,

    NORTH_EAST,

    SOUTH_WEST,

    SOUTH_EAST,

}
```

分别在屏幕的位置如下图所示：

```
/**

 * define the relative location of watermark on the screen when start streaming

 *

 *     |  NorthWest       |                  |       NorthEast

 *     |                  |                  |

 *     |                  |                  |

 *     |  ----------------|------------------|----------------

 *     |                  |                  |

 *     |                  |                  |

 *     |                  |                  |

 *     |                  |                  |

 *     |  ----------------|------------------|----------------

 *     |                  |                  |
```

```
*     |                   |                   |

*     |   SouthWest       |                   |       SouthEast

*

*/
```

自定义水印的位置信息

除了通过 WATERMARK_LOCATION 的四个固定的内置位置，还可以自定义水印的位置信息：

```
    /**

     * Set the custom position of watermark top-left point (percentage of surfaceview).

     * top-left of the surfaceview is the origin of the coordinate system.

     * positive x-axis pointing right and the positive y-axis pointing down.

     * normal values of both x and y MUST be [0.0f-1.0f]

     *

     * meanwhile unset the location

     * @param x

     * @param y

     */

    public WatermarkSetting setCustomPosition(float x, float y);
```

使用方式如下：

```
// 以 Preview 的中心点为起点进行水印的绘制

watermarksetting.setCustomPosition(0.5f, 0.5f);
```

2.7.4.2 水印显示大小

```
/**

 * define de relative size of watermark

 */

 public enum WATERMARK_SIZE {

       LARGE,
```

```
    MEDIUM,

    SMALL,

}
```

2.4.0 版本后提供了设置自定义水印像素大小的功能，使用方式如下：

```
watermarksetting.setCustomSize(int width, int height);
```

2.7.4.3 构造 WatermarkSetting

传入 drawable 对象作为水印资源：

```
WatermarkSetting watermarksetting = new WatermarkSetting(mContext, R.drawable.qiniu_logo,
WatermarkSetting.WATERMARK_LOCATION.SOUTH_WEST, WatermarkSetting.WATERMARK_SIZE.MEDIUM, 100); //
100 为 alpha 值
```

传入图片的绝对路径作为水印资源：

```
WatermarkSetting watermarksetting = WatermarkSetting(mContext, "watermark resource absolute path",
WatermarkSetting.WATERMARK_LOCATION.SOUTH_WEST, WatermarkSetting.WATERMARK_SIZE.MEDIUM, 100) {
```

2.7.5 核心类 MediaStreamingManager

所有音视频推流相关的具体操作，都在 MediaStreamingManager 中进行。

2.7.5.1 构造 MediaStreamingManager

在构造 MediaStreamingManager 阶段会确定其编码的类型，目前 SDK 支持的编码类型有：

```
– AVCodecType.HW_VIDEO_WITH_HW_AUDIO_CODEC, // 视频硬编，音频硬编

– AVCodecType.SW_VIDEO_WITH_HW_AUDIO_CODEC, // 视频软编，音频硬编

– AVCodecType.SW_VIDEO_WITH_SW_AUDIO_CODEC, // 视频软编，音频软编

– AVCodecType.SW_AUDIO_CODEC,                // 纯音频软编

– AVCodecType.HW_AUDIO_CODEC,                // 纯音频硬编

– AVCodecType.SW_VIDEO_CODEC,                // 纯视频软编

– AVCodecType.HW_VIDEO_CODEC;                // 纯视频硬编
```

构造带有视频 MediaStreamingManager，需要传入 GLSurfaceView

```
mMediaStreamingManager = new MediaStreamingManager(mContext, mGLSurfaceView,

EncodingType.SW_VIDEO_WITH_SW_AUDIO_CODEC);
```

构造完毕后，需调用 MediaStreamingManager#prepare 向 SDK 提供对应的配置信息，以 v1.6.2 版本为例：

```
mMediaStreamingManager.prepare(mCameraStreamingSetting, mMicrophoneStreamingSetting, mWatermarkSetting,

mProfile);
```

2.7.5.2 设置 Listener

为了更好的和 SDK 交互，接受各种状态和其他信息，需要注册对应的 Listener:

```
mMediaStreamingManager.setStreamingStateListener(this);

mMediaStreamingManager.setStreamingSessionListener(this);

mMediaStreamingManager.setStreamStatusCallback(this);
```

- StreamingStateChangedListener

  接口原型如下：

```
/**

 * Callback interface for Streaming State.

 * <p>

 *

 * Called on an "arbitrary thread".

 *

 * */

public interface StreamingStateChangedListener {

    /**

     * Invoked if the {@link StreamingState} changed

     *

     * @param status the specified {@link StreamingState}

     * @param extra the extra information

     * */
```

```
    void onStateChanged(StreamingState status, Object extra);

}
```

onStateChanged 中 status 对应的含义分别为：

```
public enum StreamingState {

    /**

     * The initial state.

     *

     * */

    UNKNOWN,



    /**

     * Preparing the environment for network connection.

     * <p>

     * The first state after calling {@link StreamingManager#startStreaming()}

     *

     * */

    PREPARING,



    /**

     * <ol>

     *      <li>{@link StreamingManager#resume()} done in pure audio streaming</li>

     *      <li>{@link StreamingManager#resume()} done and camera be activated in AV streaming.</li>

     * </ol>

     * */
```

```
READY,



/**

 * Being connecting.

 *

 * */

CONNECTING,



/**

 * The av datas start sending successfully.

 *

 * */

STREAMING,



/**

 * Streaming has been finished, and you can {@link StreamingManager#startStreaming()} again.

 *

 * */

SHUTDOWN,



/**

 * Connect error.

 *

 * The following is the possible case:

 *
```

```
     * <ol>
     *     <li>Stream is invalid</li>
     *     <li>Network is unreachable</li>
     * </ol>
     *
     * */
    IOERROR,

    /**
     * Notify the camera switched.
     * <p>
     * extra will including the info the new camera id.
     *
     * <ol>
     *     <li>Camera.CameraInfo.CAMERA_FACING_FRONT</li>
     *     <li>Camera.CameraInfo.CAMERA_FACING_BACK</li>
     * </ol>
     *
     * <pre>
     *     <code>
     *         Log.i(TAG, "current camera id:" + (Integer)extra);
     *     </code>
     * </pre>
     * */
    CAMERA_SWITCHED,
```

```
/**
 * Notify the torch info after camera be active.
 * <p>
 * extra will including the info if the device support the Torch.
 *
 * <ol>
 *      <li>true, supported</li>
 *      <li>false, unsupported</li>
 * </ol>
 *
 * <pre>
 *      <code>
 *          final boolean isSupportedTorch = (Boolean) extra;
 *      </code>
 * </pre>
 *
 * */
TORCH_INFO,


/**
 * Sending buffer is empty.
 *
 * */
SENDING_BUFFER_EMPTY,
```

```
/**
 * Sending buffer have been full.
 *
 * */
SENDING_BUFFER_FULL,

/**
 * Sending buffer have few items witch waiting to be sent.
 *
 * */
SENDING_BUFFER_HAS_FEW_ITEMS,

/**
 * Sending buffer have many items witch waiting to be sent.
 *
 * */
SENDING_BUFFER_HAS_MANY_ITEMS,

/**
 * The network connection has been broken.
 *
 * */
DISCONNECTED,
```

```
/**
 * if the device hasn't the supported preview size, then it will select the default preview
 * size which mismatch the specified {@link CameraStreamingSetting.PREVIEW_SIZE_RATIO}.
 *
 * */
NO_SUPPORTED_PREVIEW_SIZE,


/**
 * {@link AudioRecord#startRecording()} failed.
 *
 * */
AUDIO_RECORDING_FAIL,


/**
 * camera open failed.
 *
 * */
OPEN_CAMERA_FAIL,


/**
 * Do not support NV21 preview format.
 *
 * */
NO_NV21_PREVIEW_FORMAT,
```

```
/**
 * Invalid streaming url.
 *
 * Gets the message after call {@link StreamingManager#setStreamingProfile(StreamingProfile)} if streaming
 * url is invalid. Also gets the url as the extra info.
 * */
INVALID_STREAMING_URL,

/**
 * The network had been built successfully.
 *
 * */
CONNECTED,


/**
 * Invalid streaming url.
 *
 * Gets the message after call {@link MediaStreamingManager#setStreamingProfile(StreamingProfile)} if streaming
     * url is invalid. Also gets the url as the extra info.
     * */
    UNAUTHORIZED_STREAMING_URL,
}
```

- StreamingSessionListener

  接口原型如下:

```
/**
 * Callback interface for some particular Streaming incidents.
```

```
 * <p>
 *
 * Called on an "arbitrary thread".
 *
 * */
public interface StreamingSessionListener {
    /**
     * Invoked when audio recording failed.
     * <p>
     *
     * @param code error code. <b>Unspecified now</b>.
     *
     * @return true means you handled the event; otherwise, given up.
     *
     * */
    boolean onRecordAudioFailedHandled(int code);


    /**
     * Restart streaming notification.
     * <p>
     *
     * When the network-connection is broken, {@link StreamingState#DISCONNECTED} will notified first,
     * and then invoked this method if the environment of restart streaming is ready.
     *
     * <p>
```

```
* SDK won't limit the number of invocation.
*
* @param code error code. <b>Unspecified now</b>.
*
* @return true means you handled the event; otherwise, given up and then {@link StreamingState#SHUTDOWN}
*                   will be notified.
*
* */
boolean onRestartStreamingHandled(int code);



/**
 * Invoked after camera object constructed.
 * <p>
 * The supported list exists in following cases:
 * <ol>
 *      <li>If didn't set the
 *      {@link CameraStreamingSetting#setCameraPrvSizeRatio} when
 *      initialize {@link CameraStreamingSetting},
 *      the whole supported list would be passed</li>
 *      <li>If {@link CameraStreamingSetting#setCameraPrvSizeRatio}
 *      was set, the supported preview size which filtered by the specified ratio would be passed</li>
 * </ol>
 *
 * @param list supported camera preview list which sorted from smallest to largest. The list maybe null.
 *
```

```
     * @return null means you give up selection and SDK will help you select a proper preview
     *                 size; otherwise, the returned size will be effective.
     *
     * */
    Camera.Size onPreviewSizeSelected(List<Camera.Size> list);


    /**
     * Custom preview fps, invoked after camera object constructed.
     *
     * @param   supportedPreviewFpsRange
     *                 a list of supported preview fps ranges by Camera. This method returns a
     *                 list with at least one element. Every element is an int array
     *                 of two values – minimum fps and maximum fps.
     * @return –1 means you give up selection and SDK will help you select a proper preview
     *                  fps; otherwise, the returned index will be effective.
     */
    int onPreviewFpsSelected(final List<int[]> supportedPreviewFpsRange);
}
```

可以在 StreamingSessionListener 处理一些重连、音频读取失败、preview size 的自定义操作。

```
@Override
public boolean onRecordAudioFailedHandled(int err) {
    mMediaStreamingManager.updateEncodingType(AVCodecType.SW_VIDEO_CODEC);
    mMediaStreamingManager.startStreaming();
    return true;
}
```

```
@Override

public boolean onRestartStreamingHandled(int err) {

      return mMediaStreamingManager.startStreaming();

}



@Override

public Camera.Size onPreviewSizeSelected(List<Camera.Size> list) {

    if (list != null) {

          for (Camera.Size s : list) {

                Log.i(TAG, "w:" + s.width + ", h:" + s.height);

          }

//                return "your choice";

    }

    return null;

}
```

在消费了 onRecordAudioFailedHandled 或 onRestartStreamingHandled 之后，您应该返回 true 通知 SDK；若不做任何处理，返回 false。

- StreamStatusCallback

  接口原型如下：

```
/**

 * Callback interface used to notify {@link StreamingProfile.StreamStatus}.

 */

public interface StreamStatusCallback {



   /**
```

```
    * Called per the {@link StreamingProfile.StreamStatusConfig#getIntervalMs}

    *

    * @param status the new {@link StreamingProfile.StreamStatus}

    *

    * */

    void notifyStreamStatusChanged(final StreamingProfile.StreamStatus status);

}
```

注意：notifyStreamStatusChanged 运行在非 UI 线程中。

StreamStatus 的定义如下：

```
/**

 * The nested class is for feedbacking the av status in real time.

 *

 * <p>

 * You can set the {@link StreamStatusConfig} to get the preferred callback frequency.

 *

 * */

public static class StreamStatus {

    /**

     * Audio frame per second.

     * */

    public int audioFps;



    /**

     * Video frame per second.

     * */
```

```
    public int videoFps;



    /**

     * Audio and video total bits per second.

     * */

    public int totalAVBitrate;    // bps



    /**

     * Audio and video total bits per second.

     * */


    /**

     * Audio bits per second.

     * */

    public int audioBitrate;    // bps



    /**

     * Video bits per second.

     * */

    public int videoBitrate;    // bps

}
```

2.7.5.3 resume

MediaStreamingManager#resume 会进行 Camera 的打开操作，当成功打开后，会返回 STATE.READY 消息，用户可以在接受到 STATE.READY 之后，安全地进行推流操作。

```
mMediaStreamingManager.resume();
```

若在一个 Activity 中进行推流操作，建议 mMediaStreamingManager.resume() 在 Activity#onResume 中被调用。

2.7.5.4 开始推流

由于 startStreaming 会进行网络链接相关操作，因此需要将 startStreaming 运行在非 UI 线程，否则可能会发生崩溃现象。

```
mMediaStreamingManager.startStreaming();
```

2.7.5.5 手动对焦

对焦之前传入 Focus Indicator，如果不进行设置，对焦过程中将会没有对应的 UI 显示。

```
// You should call this after getting {@link STATE#READY}.

mMediaStreamingManager.setFocusAreaIndicator(mRotateLayout,

                        mRotateLayout.findViewById(R.id.focus_indicator));
```

点击屏幕触发手动对焦，并设置对应的坐标值。

```
// You should call this after getting {@link STATE#READY}.

mMediaStreamingManager.doSingleTapUp((int) e.getX(), (int) e.getY());
```

2.7.5.6 Zoom

Camera Zoom 操作。

```
// mCurrentZoom must be in the range of [0, mMediaStreamingManager.getMaxZoom()]

// You should call this after getting {@link STATE#READY}.

if (mMediaStreamingManager.isZoomSupported()) {

   mMediaStreamingManager.setZoomValue(mCurrentZoom);

}
```

可以获取到当前的 Zoom 值：

```
mMediaStreamingManager.getZoom();
```

2.7.5.7 闪光灯操作

开启闪光灯。

```
mMediaStreamingManager.turnLightOn();
```

关闭闪光灯。

```
mMediaStreamingManager.turnLightOff();
```

2.7.5.8 切换摄像头

切换摄像头。

```
mMediaStreamingManager.switchCamera();
```

2.7.5.9 禁音推流

在推流过程中，将声音禁用掉：

```
mMediaStreamingManager.mute(true);
```

恢复声音：

```
mMediaStreamingManager.mute(false);
```

注：默认为 false

2.7.5.10 截帧

在 Camera 正常预览之后，可以正常进行截帧功能。

在调用 captureFrame 的时候，您需要传入 width 和 height，以及 FrameCapturedCallback，如果传入的 width 或者 height 小于等于 0，SDK 返回的 Bitmap 将会是预览的尺寸 。SDK 完成截帧之后，会回调 onFrameCaptured，并将结果以参数的形式返回给调用者。

```
mMediaStreamingManager.captureFrame(w, h, new FrameCapturedCallback() {

    @Override

    public void onFrameCaptured(Bitmap bmp) {



    }

}
```

注意：调用者有义务对 Bitmap 进行回收释放。截帧失败，bmp 会为 null。

2.7.5.11 停止推流

停止当前推流。

```
mMediaStreamingManager.stopStreaming();
```

2.7.5.12 Log 管理

当 enabled 设置为 true ，SDK Native 层的 log 将会被打开；当设置为 false，SDK Native 层的 log 将会被关闭。默认处于打开状态。

```
mMediaStreamingManager.setNativeLoggingEnabled(false);
```

注：默认值为 true。建议 Release 版本置为 false。

2.7.5.13 pause

退出 MediaStreamingManager，该操作会主动断开当前的流链接，并关闭 Camera 和释放相应的资源。

```
mMediaStreamingManager.pause();
```

2.7.5.14 destroy

释放不紧要资源。

```
mMediaStreamingManager.destroy();
```

2.7.5.15 自定义音频数据处理

用户可以通过下面回调接口，获取当前音频数据，实现自定义音频数据处理。

```
// 注册音频采集数据回调

mMediaStreamingManager.setAudioSourceCallback(AudioSourceCallback callback);



public interface AudioSourceCallback {

    /**

     * 回调音频采集 PCM 数据

     *

     * @param srcBuffer       音频 PCM 数据，该 buffer 是 direct ByteBuffer。

     * @param size            buffer 的大小

     * @param tsInNanoTime    时间戳，单位：纳秒

     * @param isEof           采集结束标志

     */

    void onAudioSourceAvailable(ByteBuffer srcBuffer, int size, long tsInNanoTime, boolean isEof);

}
```

2.7.5.16 动态水印

通过调用 MediaStreamingManager.updateWatermarkSetting 方法可以动态改变水印的内容、位置、大小。

```
WatermarkSetting watermarkSetting = new WatermarkSetting(context);
```

```
watermarkSetting.setResourceId(R.drawable.qiniu_logo);

watermarkSetting.setAlpha(50);

watermarkSetting.setSize(WatermarkSetting.WATERMARK_SIZE.LARGE);

watermarkSetting.setLocation(WatermarkSetting.WATERMARK_LOCATION.SOUTH_EAST);

mMediaStreamingManager.updateWatermarkSetting(newWatermarkSetting);
```

2.7.6 自定义滤镜

2.7.6.1 软编模式滤镜实现

需要分别处理预览显示的 filter 效果和 encoding 的 filter 效果：预览显示通过实现 SurfaceTextureCallback interface；encoding 通过实现 StreamingPreviewCallback interface。两者分别实现，互不影响。

- encoding 部分

```
public interface StreamingPreviewCallback {

    public boolean onPreviewFrame(byte[] bytes, int width, int height, int rotation, int fmt, long tsInNanoTime);

}
```

onPreviewFrame 会回调 NV21 格式的 YUV 数据，进行 filter 算法处理后的结果仍然保存在 bytes 数组中，SDK 会将 bytes 中的数据当作数据源进行后续的编码和封包等操作；onPreviewFrame 运行在名称为 CameraManagerHt 的子线程中；onPreviewFrame 仅在 STATE.STREAMING 状态下被回调。

- 预览显示部分

```
public interface SurfaceTextureCallback {

    void onSurfaceCreated();

    void onSurfaceChanged(int width, int height);

    void onSurfaceDestroyed();

    int onDrawFrame(int texId, int width, int height);

}
```

四个回调均执行在 GL rendering thread；如果 onDrawFrame 直接返回 texId，代表放弃 filter 处理，否则 onDrawFrame 应该返回一个 filter 算法处理过的纹理 id； 自定义的 Texture id，即 onDrawFrame 返回的纹理 id， 必须是 GLES20.GL_TEXTURE_2D 类型；SDK 回调的纹理 id，即 onDrawFrame 的参数 texId 类型为 GLES11Ext.GL_TEXTURE_EXTERNAL_OES。

2.7.6.2 硬编模式滤镜实现

硬编模式下仅需要实现 SurfaceTextureCallback interface 就可实现预览显示和 streaming。

2.7.7 录屏

PLDroidMediaStreaming 封装好了录屏相关的底层操作，用户可以非常方便的进行录屏推流。其步骤如下：

1、AndroidManifest.xml 注册 com.qiniu.pili.droid.streaming.screen.ScreenCaptureRequestActivity

```
<activity

 android:name="com.qiniu.pili.droid.streaming.screen.ScreenCaptureRequestActivity"

 android:theme="@android:style/Theme.Translucent.NoTitleBar" >

</activity>
```

2、构造核心类 ScreenStreamingManager

ScreenStreamingManager 封装了屏幕的录制、音频的采集，编码，封包和推流操作。用户只需要简单的调用相关 API 即可实现录屏推流：

```
// 构造 ScreenStreamingManager

ScreenStreamingManager screenStreamingManager = new ScreenStreamingManager();



// 配置相关参数

screenStreamingManager.prepare(context, screenSetting, null, streamingProfile);



// 开始推流

screenStreamingManager.startStreaming();



// 停止推流

screenStreamingManager.stopStreaming();



// 销毁

screenStreamingManager.destroy();
```

3、自定义音频数据处理

用户可以通过下面回调接口，获取当前音频数据，实现自定义音频数据处理。

```
// 注册音频采集数据回调

screenStreamingManager.setAudioSourceCallback(AudioSourceCallback callback);



public interface AudioSourceCallback {

    /**

     * 回调音频采集 PCM 数据

     *

     * @param srcBuffer        音频 PCM 数据，该 buffer 是 direct ByteBuffer。

     * @param size             buffer 的大小

     * @param tsInNanoTime     时间戳，单位：纳秒

     * @param isEof            采集结束标志

     */

    void onAudioSourceAvailable(ByteBuffer srcBuffer, int size, long tsInNanoTime, boolean isEof);

}
```

2.7.8 StreamingManager

StreamingManager 是类似 MediaStreamingManager 的一个类，两者的区别是：StreamingManager 不带采集，仅包含编码、封包推流模块，从功能层面可以理解为：

MediaStreamingManager = 采集模块 + (处理模块) + StreamingManager

其调用过程类似于 MediaStreamingManager：

```
构造 StreamingManager -> prepare -> resume -> startStreaming -> inputAudioFrame/inputVideoFrame ->
stopStreaming -> pause -> destroy
```

具体可以参考 Demo 中的 ImportStreamingActivity.java

PS：若希望使用自己已有项目中的采集 / 处理模块，可以选用 StreamingManager.

2.7.8.1 外部输入音频

```
void inputAudioFrame(ByteBuffer buffer, int size, long tsInNanoTime, boolean isEof);

void inputAudioFrame(byte[] buffer, long tsInNanoTime, boolean isEof);
```

可以选择传入 ByteBuffer 或者 byte[] 类型的 PCM 源数据，传入的时间戳为 nano time。

2.7.8.2 外部输入视频

```
void inputVideoFrame(ByteBuffer buffer, int size, int width, int height, int rotation, boolean mirror, int fmt, long
tsInNanoTime);

void inputVideoFrame(byte[] buffer, int width, int height, int rotation, boolean mirror, int fmt, long tsInNanoTime);
```

可以选择传入 ByteBuffer 或者 byte[] 类型的 YUV 数据，其中：

- width 和 height，分别为该 frame 的宽和高，单位像素
- rotation ，指该 frame 需要选择的角度（0，90，180，360），若自己已经处理好角度的旋转，rotation 参数为 0
- mirror，指是否对该 frame 做镜像处理
- fmt，指该 frame 的格式，目前支持 NV21 和 I420，即：PLFourCC.FOURCC_NV21 和 PLFourCC.FOURCC_I420
- tsInNanoTime，指该 frame 对应的时间戳，单位为纳秒

2.7.8.3 getInputSurface

getInputSurface 须在 startStreaming 被调用成功之后，从 MediaCodec 获取其 Surface 类型的 InputSurface，用户可以在这个 Surface 上面进行自定义绘制，绘制好后使用 frameAvailable 通知 SDK 进行编码。

若希望使用该高级功能，需在 AVCodecType.HW_VIDEO_SURFACE_AS_INPUT_WITH_HW_AUDIO_CODEC 模式下，否则会抛出异常。

2.7.8.4 动态改变预览镜像

若希望在推流过程中动态改变摄像头预览的镜像效果，可以使用如下 API：

```
mMediaStreamingManager.setPreviewMirror(boolean mirror)
```

2.7.8.5 动态改变推流镜像

若希望在推流过程中动态改变推流的镜像效果，可以使用如下 API：

```
mMediaStreamingManager.setEncodingMirror(boolean mirror)
```

2.7.8.6 推流时增加背景音乐

若希望在推流过程中增加背景音乐，可以使用如下 API：

```
        mAudioMixer = mMediaStreamingManager.getAudioMixer();

        mAudioMixer.setOnAudioMixListener(new OnAudioMixListener() {
    @Override
    public void onStatusChanged(MixStatus mixStatus) {
        mMixToggleBtn.post(new Runnable() {
            @Override
            public void run() {
                ...
            }
        });
    }

        @Override
        public void onProgress(long progress, long duration) {
            // time in Us
        }
});
mAudioFile = Cache.getAudioFile(this); //  背景音乐文件路径
if (mAudioFile != null) {
```

```
        try {
            mAudioMixer.setFile(mAudioFile, true); // true/false 是否循环
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    boolean s = mAudioMixer.play();
    text = s ? "mixing play success" : "mixing play failed !!!";
```

2.7.8.7 返听／耳返功能

若希望在推流过程中开启返听（耳返），可以使用如下 API：

```
mMediaStreamingManager.startPlayback();

mMediaStreamingManager.stopPlayback();
```

2.7.8.8 QUIC 推流

QUIC 是基于 UDP 开发的可靠传输协议，在弱网下拥有更好的推流效果，相比于 TCP 拥有更低的延迟，可抵抗更高的丢包率。

- 通过下面接口开启/关闭 QUIC 推流，在使用 QUIC 之前请确认直播服务端支持 QUIC 协议

```
mProfile.setQuicEnable(quicEnable);
```

3. iOS 推流端 SDK

3.1 概述

PLMediaStreamingKit 是一个适用于 iOS 的 RTMP 直播推流 SDK，可高度定制化和二次开发。SDK 提供 RTMP 推流的全套解决方案，包括采集，处理（美颜，水印等），编码，封包，发送。特色是支持 H.264 硬编码，以及支持 AAC-LC 硬编码；同时，还根据移动网络环境的多变性，实现了一套可供开发者灵活选择的编码参数集合。

3.2 功能特性

功能	描述	版本
支持硬件编码	更低的 CPU 占用及发热量	v1.0.0(+)
支持 ARM7, ARM64 指令集	为最新设备优化	v1.0.0(+)
提供音视频配置分离	配置解耦	v1.0.0(+)
支持推流时码率变更	更方便定制流畅度/清晰度策略	v1.0.0(+)
支持弱网丢帧策略	不必担心累计延时，保障实时性	v1.0.0(+)
支持模拟器运行	不影响模拟器快速调试	v1.0.0(+)
支持 RTMP 协议直播推流	保证秒级实时性	v1.0.0(+)
支持后台音频推流	轻松实现边推流边聊天等操作	v1.0.0(+)
提供多码率可选	更自由的配置	v1.1.2(+)
提供 H.264 视频编码	多种 profile level 可设定	v1.1.2(+)
支持多分辨率编码	更可控的清晰度	v1.1.2(+)
提供 AAC 音频编码	当前采用 AAC-LC	v1.1.2(+)
提供 HeaderDoc 文档	开发中使用 Quick Help 及时阅读文档	v1.1.3(+)
支持美颜滤镜	轻松实现更美真人秀	v1.7.0(+)
支持水印功能	彰显自身特色	v1.7.0(+)
提供内置音效及音频文件播放功能	轻松实现各种音效	v2.1.0(+)
支持返听功能	唱歌更易把握节奏	v2.1.0(+)
支持截屏功能	轻松分享美好瞬间	v2.1.2(+)
支持 iOS 10 ReplayKit 录屏	方便分享游戏过程	v2.1.4(+)
支持苹果 ATS 安全标准	安全性更高	v2.1.6(+)
支持后台推图片功能	观看体验更佳	v2.2.1(+)
支持 QUIC 推流功能	弱网推流更流畅	v2.3.0(+)

3.3 快速开始

3.3.1 开发环境配置

- Xcode 开发工具。App Store 下载地址
- 安装 CocoaPods。了解 CocoaPods 使用方法。官方网站

3.3.2 导入 SDK

3.3.2.1 使用 CocoaPods 导入

推荐使用 CocoaPods 的方式导入，步骤如下：

- 在工作目录中创建名称为 Podfile 的文件
- 在 Podfile 中添加如下一行

```
pod 'PLMediaStreamingKit'
```

- 在终端中运行

```
$ pod install
```

到此，你已完成了 PLMediaStreamingKit 的添加。

此外，如果你希望将 PLMediaStreamingKit 从旧版本升级到新版本，可以在终端中运行

```
$ pod update
```

3.3.2.2 手动导入

我们建议使用 CocoaPods 导入，如果由于特殊原因需要手动导入，可以按照如下步骤进行：

- 将 Pod/Library 目录下的 PLMediaStreamingKit.framework 和 HappyDNS.framework 加入到工程中；
- 在对应的 Target 的 Embedded Binaries 设置中加入 PLMediaStreamingKit.framework 和 HappyDNS.framework；
- 在工程对应 TARGET 中，右侧 Tab 选择 "Build Phases"，在 "Link Binary With Libraries" 中加入 UIKit、AVFoundation、CoreGraphics、CFNetwork、CoreMedia、AudioToolbox 这些 framework，并加入 libc++.tdb、libz.tdb 及 libresolv.tbd；
- 在工程对应 TARGET 中，右侧 Tab 选择 "Build Settings"，在 "Other Linker Flags" 中加入 "-ObjC" 选项；

3.3.3 初始化推流逻辑

3.3.3.1 添加引用并初始化 SDK 使用环境

在 AppDelegate.m 中添加引用

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

并在 – (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions 中添加如下代码:

```
[PLStreamingEnv initEnv];
```

然后在 ViewController.m 中添加引用

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

3.3.3.2 添加 session 属性

添加 session 属性在 ViewController.m

```
@property (nonatomic, strong) PLMediaStreamingSession *session;
```

3.3.4 创建流对象

3.3.4.1 创建视频和音频的采集和编码配置对象

当前使用默认配置，之后可以深入研究按照自己的需求作更改

```
PLVideoCaptureConfiguration *videoCaptureConfiguration = [PLVideoCaptureConfiguration defaultConfiguration];

PLAudioCaptureConfiguration *audioCaptureConfiguration = [PLAudioCaptureConfiguration defaultConfiguration];

PLVideoStreamingConfiguration *videoStreamingConfiguration = [PLVideoStreamingConfiguration
defaultConfiguration];

PLAudioStreamingConfiguration *audioStreamingConfiguration = [PLAudioStreamingConfiguration
defaultConfiguration];
```

3.3.4.2 创建推流 session 对象

```
self.session = [[PLMediaStreamingSession alloc] initWithVideoCaptureConfiguration:videoCaptureConfiguration
audioCaptureConfiguration:audioCaptureConfiguration videoStreamingConfiguration:videoStreamingConfiguration
audioStreamingConfiguration:audioStreamingConfiguration stream:nil];
```

3.3.5 预览摄像头拍摄效果

将预览视图添加为当前视图的子视图

```
[self.view addSubview:self.session.previewView];
```

3.3.6 添加推流操作

取一个最简单的场景，就是点击一个按钮，然后触发发起直播的操作。

3.3.6.1 添加触发按钮

我们在 view 上添加一个按钮吧。 我们在 – (void)viewDidLoad 方法最后添加如下代码

```
UIButton *button = [UIButton buttonWithType:UIButtonTypeSystem];

[button setTitle:@"start" forState:UIControlStateNormal];

button.frame = CGRectMake(0, 0, 100, 44);

button.center = CGPointMake(CGRectGetMidX([UIScreen mainScreen].bounds), CGRectGetHeight([UIScreen mainScreen].bounds) – 80);

[button addTarget:self action:@selector(actionButtonPressed:) forControlEvents:UIControlEventTouchUpInside];

[self.view addSubview:button];
```

3.3.6.2 创建推流地址

在实际开发过程中，为了您的 App 有更好的用户体验，建议提前从服务器端获取推流地址

```
NSURL *pushURL = [NSURL URLWithString:@"your push url"];
```

3.3.6.3 实现按钮动作

```
– (void)actionButtonPressed:(id)sender {

    [self.session startStreamingWithPushURL:pushURL feedback:^(PLStreamStartStateFeedback feedback) {

            if (feedback == PLStreamStartStateSuccess) {

                NSLog(@"Streaming started.");

            }

            else {

                NSLog(@"Oops.");

            }

    }];

}
```

3.4 功能使用

3.4.1 音视频采集和编码配置

PLMediaStreamingKit 中通过不同的 configuration 设置不同的采集或编码配置信息，对应的有：

- PLVideoCaptureConfiguration 视频采集配置
- PLAudioCaptureConfiguration 音频采集配置
- PLVideoStreamingConfiguration 视频编码配置
- PLAudioStreamingConfiguration 音频编码配置

可以通过如下途径来设置 configuration：

- 在 PLMediaStreamingSession init 时传递对应的 configuration
- 在推流前、推流中、推流结束后调用 – (void)reloadVideoStreamingConfiguration:(PLVideoStreamingConfiguration *)videoStreamingConfiguration; 重置视频编码配置
- 对于视频采集配置，可以直接设置 PLMediaStreamingSession 相关的属性；

需要注意的是，通过 reload 方法重置 configuration 时，需要确保传递的 configuration 与当前 session 已经持有的不是一个对象。

3.4.1.1 视频采集参数

1.自定义视频采集参数

当前的 PLVideoCaptureConfiguration 中可自行设定的参数有

- videoFrameRate
    - 即 FPS，每一秒所包含的视频帧数
- sessionPreset
    - 即采集时的画幅分辨率大小
- previewMirrorFrontFacing
    - 是否在使用前置摄像头采集的时候镜像预览画面
- previewMirrorRearFacing
    - 是否在使用后置摄像头采集的时候镜像预览画面
- streamMirrorFrontFacing
    - 是否在使用前置摄像头采集的时候镜像编码画面
- streamMirrorRearFacing
    - 是否在使用后置摄像头采集的时候镜像编码画面

- position
    - 开启 PLMediaStreamingSession 的时候默认使用前置还是后置摄像头
- videoOrientation
    - 开启 PLMediaStreamingSession 的时候默认使用哪个旋转方向

需要注意的是指定分辨率的 sessionPreset 例如 AVCaptureSessionPreset1920x1080 并非所有机型的所有摄像头均支持，在设置相应的采集分辨率之前请务必保证做过充分的机型适配测试，避免在某些机型使用该机型摄像头不支持的 sessionPreset。另外，如果使用只指定采集质量的 sessionPreset，例如 AVCaptureSessionPresetMedium，那系统会根据当前摄像头的支持情况使用相应质量等级的分辨率进行采集。

3.4.1.2 音频采集参数

自定义音频采集参数

当前的 PLAudioCaptureConfiguration 中可自行设定的参数有

- channelsPerFrame
    - 采集时的声道数，默认为 1，并非所有采集设备都支持多声道数据的采集，可以通过检查 [AVAudioSession sharedInstance].maximumInputNumberOfChannels 得到当前采集设备支持的最大声道数。
- acousticEchoCancellationEnable
    - 回声消除开关，默认为 NO。普通直播用到回声消除的场景不多，当用户开启返听功能，并且使用外放时，可打开这个开关，防止产生尖锐的啸叫声。

3.4.1.3 视频编码参数

当不确定视频编码具体的参数该如何设定时，你可以选择 SDK 内置的几种视频编码质量。

Quality 的对比

Quality	VideoSize	FPS	ProfileLevel	Video BitRate(Kbps)
kPLVideoStreamingQualityLow1	272x480	24	Baseline AutoLevel	128
kPLVideoStreamingQualityLow2	272x480	24	Baseline AutoLevel	256
kPLVideoStreamingQualityLow3	272x480	24	Baseline AutoLevel	512
kPLVideoStreamingQualityMedium1	368x640	24	High AutoLevel	512
kPLVideoStreamingQualityMedium2	368x640	24	High AutoLevel	768

Quality	VideoSize	FPS	ProfileLevel	Video BitRate(Kbps)
kPLVideoStreamingQualityMedium3	368x640	24	High AutoLevel	1024
kPLVideoStreamingQualityHigh1	720x1280	24	High AutoLevel	1024
kPLVideoStreamingQualityHigh2	720x1280	24	High AutoLevel	1280
kPLVideoStreamingQualityHigh3	720x1280	24	High AutoLevel	1536

自定义编码参数

当前的 PLVideoStreamingConfiguration 中可自行设定的参数有

- videoProfileLevel
    - H.264 编码时对应的 profile level 影响编码压缩算法的复杂度和编码耗能。设置的越高压缩率越高，算法复杂度越高，相应的可能带来发热量更大的情况
- videoSize
    - 编码的分辨率，对于采集到的图像，编码前会按照这个分辨率来做拉伸或者裁剪
- expectedSourceVideoFrameRate
    - 预期视频的编码帧率，这个数值对编码器的来说并不是直接限定了 fps，而是给编码器一个预期的视频帧率，最终编码的视频帧率，是由实际输入的数据决定的
- videoMaxKeyframeInterval
    - 两个关键帧的帧间隔，一般设置为 FPS 的三倍
- averageVideoBitRate
    - 平均的编码码率，设定后编码时的码率并不会是恒定不变，静物较低，动态物体会相应升高
- videoEncoderType
    - H.264 编码器类型，默认采用 PLH264EncoderType_AVFoundation 编码方式，在 iOS 8 及以上的系统可采用 PLH264EncoderType_VideoToolbox，编码效率更高

PLMediaStreamingKit 为了防止编码参数设定失败而导致编码失败，出现推流无视频的情况，依据 videoProfileLevel 限定了其他参数的范围，该限定范围针对 Quality 生成的配置同样有效。参见以下表格：

ProfileLevel	Max VideoSize	Max FPS	Max Video BitRate(Mbps)
Baseline 30	(720, 480)	30	10
Baseline 31	(1280, 720)	30	14
Baseline 41	(1920, 1080)	30	50
Main 30	(720, 480)	30	10

ProfileLevel	Max VideoSize	Max FPS	Max Video BitRate(Mbps)
Main 31	(1280, 720)	30	14
Main 32	(1280, 1024)	30	20
Main 41	(1920, 1080)	30	50
High 40	(1920, 1080)	30	25
High 41	(1920, 1080)	30	62.5

码率、fps、分辨对清晰度及流畅度的影响

对于码率（BitRate）、FPS（frame per second）、分辨率（VideoSize）三者的关系，有必要在这里做一些说明，以便你根据自己产品的需要可以有的放矢的调节各个参数。

一个视频流个人的感受一般来说会有卡顿、模糊等消极的情况，虽然我们都不愿意接受消极情况的出现，但是在 UGC 甚至 PGC 的直播场景中，都不可避免的要面对。因为直播推流实时性很强烈，所以为了保证这一实时性，在网络带宽不足或者上行速度不佳的情况下，都需要做出选择。

要么选择更好的流程度但牺牲清晰度（模糊），要么选择更好的清晰度但牺牲流畅度（卡顿），这一层的选择大多由产品决定。

一般来说，当选定了一个分辨率后，推流过程中就不会对分辨率做变更，但可以对码率和 FPS 做出调节，从而达到上述两种情况的选择。

效果	码率	FPS
流畅度	负相关	正相关
清晰度	正相关	负相关

通过这个关联，我们就可以容易的知道该如何从技术层面做出调整。在追求更好的流畅度时，我们可以适当降低码率，如果 FPS 已经较高（如 30）时，可以维持 FPS 不变更，如果此时因为码率太低而画面无法接受，可以再适当调低 FPS；在追求更清晰的画质时，可以提高码率，FPS 调节至 24 左右人眼大多还会识别为流畅，如果可以接受有轻微卡顿，那么可以将 FPS 设置的更低，比如 20 甚至 15。

总之，这三者之间一起构建其了画面清晰和视频流畅的感觉，但最终参数是否能满意需要自己不断调整和调优，从而满足产品层面的需求。

3.4.1.4 音频编码参数

相比于视频繁杂的参数，当前 PLAudioStreamingConfiguration 可配置的参数较为简单，目前提供音频码率和编码器的配置，音频编码可选择 AAC-LC 或者 HE-AAC。

各 Quality 的对比：

Quality	Audio BitRate(Kbps)
kPLAudioStreamingQualityHigh1	64
kPLAudioStreamingQualityHigh2	96
kPLAudioStreamingQualityHigh3	128

3.4.1.5 切换视频配置

为了满足推流中因网络变更，网络拥塞等情况下对码率、FPS 等参数的调节，PLMediaStreamingSession 提供了重置编码参数的方法，因为在重置编码器时会重新发送编码参数信息，可能触发播放器重置解码器或者清除缓存的操作（依据播放器自身行为而定），所以推流中切换编码参数时，观看短可能出现短暂（但视觉可感知）的卡顿。因此建议不要频繁的切换编码参数，避免因此带来的播放端体验问题。

- 在推流前、推流中、推流结束后调用 – (void)reloadVideoStreamingConfiguration:(PLVideoStreamingConfiguration *)videoStreamingConfiguration; 来重置 configuration

需要注意的是，通过 reload 方法重置 configuration 时，需要确保传递的 configuration 与当前 session 已经持有的不是一个对象。

3.4.1.6 建议编码参数

提示：以下为建议值，可根据产品需求自行更改调节。

UGC 场景，因为主播方所在的网络环境参差不齐，所以不易将码率设置的过高，此处我们给出建议设定

- WiFi: video Medium2 或者自定义编码参数时设定码率为 600~800Kbps
- 3G/4G: video Medium1 或者自定义编码参数时设定码率为 400~600Kbps

PGC 场景，因为主播方所在网络一般都会有较高的要求，并且主播网络质量大多可以保障带宽充足，此处我们给出建议设定

- WiFi: video High1 或者自定义编码参数时设定码率为 1000~1200Kbps
- 3G/4G: video Medium2 或者自定义编码参数时设定码率为 600~800Kbps

对于 PGC 中的 3G/4G 场景，假定 PGC 时会配备较好的外置热点保证上行带宽充足。

3.4.1.7 如何只推音频

当你只需要推送音频时，并不需要额外的增加代码，只需要在创建 PLMediaStreamingSession 时，只传入 PLAudioStreamingConfiguration 和 PLAudioCaptureConfiguration 对象即可，这样 PLMediaStreamingSession 就不会在内部创建视频采集和编码的相关内容，推流时也只会发音频配置信息和音频数据。

3.4.1.8 返听

返听又被称之为耳返，指声音通过主播的麦克风被录入之后，立即从主播的耳机中播出来。返听功能如果搭配音效功能一起使用，可以让主播听到经音效处理后（如加入混响效果）的自己的声音，会有一种独特感觉。

返听功能可通过 PLMediaStreamingSession 的 playback 属性进行开启或关闭。注意，只有在推流进行时，返听功能才会起作用。因为只有开始推流之后，SDK 才会打开麦克风并开始录音。

此外，建议通过业务逻辑禁止主播在没有插上耳机的情况下使用返听功能。虽然 SDK 允许用户即便没有插入耳机却照样可以开启返听，但那并不意味着我们建议你这么做。因为在 iPhone 没有接入耳机的时候开启返听，iPhone 的麦克风录入的声音会从 iPhone 的扬声器中立即播放出来，从而再次被麦克风录入，如此反复几秒后将变成尖锐的电流音。想象一下你在 KTV 把话筒凑到音响附近后听到的令人不快的刺耳声音吧。因此，我们强烈建议开发者在业务逻辑层进行判定，当主播开启返听功能时，如果拔掉耳机，请将 playback 属性设为 NO 以关闭返听功能。

3.4.1.9 音效

SDK 内置的音效模块会对主播通过麦克风录入的声音进行处理，从而让人听起来有不一样的感觉。例如加入“回声”音效后，主播的声音听起来就好像置身于空旷的讲堂一般；加入“混响”音效后，主播的声音听起来则更浑厚有力。

音效会影响返听功能，经过音效处理后的声音将被主播自己的耳机播放，音响产生的效果也会被推流出去，从而被观众听到。

SDK 的音效功能是对 iOS 的 Audio Unit 进行的封装，使开发者可以抽身于 Audio Unit 复杂的 API 泥潭。音效的添加、修改、删除都可以通过操作下面这个属性进行：

```
@property (nonatomic, strong) NSArray<PLAudioEffectConfiguration *> *audioEffectConfigurations;
```

这是一个由 PLAudioEffectConfiguration 对象构成的数组，每一个 PLAudioEffectConfiguration 对象对应一种音效。如果你需要同时开启多个音效，只需像如下示例把它们全部放在一个数组中即可：

```
mediaStreamingSession.audioEffectConfigurations = @[effect0, effect1, effect2, ...];
```

如果你想删除某个音效，只需要重新构造一个数组，令它唯独不包含那个你想要删除的音效,然后再重新赋值该属性即可。如果你想关闭音效功能，只需要设置一个空数组，或设置 nil 即可。注意，对音效的操作是立即生效的，不需要重启推流。

构成数组的元素必须是 PLAudioEffectConfiguration 对象或它的子类的对象。SDK 提供了众多的配置对象供你选择,这些配置对象全部都是 PLAudioEffectConfiguration 的子类对象。每一种配置对象往往对应一种 kAudioUnitType_Effect 类型的 Audio Unit。如果你熟悉 Audio Unit，你会发现每一种 kAudioUnitType_Effect 的子类型，SDK 中都有一种配置类与之对应。例如，sub type 为 kAudioUnitSubType_Reverb2 的 Audio Unit 在 SDK 中对应的配置类为 PLAudioEffectReverb2Configuration。而 Reverb2 的所有可使用的属性都可以在 PLAudioEffectReverb2Configuration 的成员变量中找到。你可以通过

```
id effect = [PLAudioEffectReverb2Configuration defaultConfiguration];
```

来构造一个所有成员变量都取默认值的配置对象。然后，通过类似

```
...



effect.gain = 0.8;

effect.decayTimeAt0Hz = 1.2;



...
```

来设置构造好的配置对象的属性。

至此，你应该已经明白了如何构造任何你想要的音效配置了。

- 首先,你需要查找 Apple 的 Audio Unit 的 API 文档,在所有 type 为 kAudioUnitType_Effect 的 Audio Unit 中挑选一个 sub type，作为你想要的音效，然后根据 sub type 的名字找到 SDK 中对应的配置类。例如，在之前的例子中，sub type 为 kAudioUnitSubType_Reverb2，因此配置类的名字为 PLAudioEffectReverb2Configuration。
- 之后调用 [PLAudioEffectXXXConfiguration defaultConfiguration] 来 构造一个全部属性为默认值的配置对象。
- 调整属性的值，来得到你想要的音效效果。
- 重复之前的步骤，直到构造出所有你需要的音效配置对象，并全部装入一个 数组。
- 通过设置 mediaStreamingSession.audioEffectConfigurations = @[...]; 来让你之前准备的音效配置生效。

除了与 Audio Unit 一一对应的音效配置类，我们还提供了预设的音效类，PLAudioEffectModeConfiguration。你可以通过如下三个方法获取三种预设混响音效配置

```
[PLAudioEffectModeConfiguration reverbLowLevelModeConfiguration];

[PLAudioEffectModeConfiguration reverbMediumLevelModeConfiguration];

[PLAudioEffectModeConfiguration reverbHeightLevelModeConfiguration];
```

mediaStreamingSession.audioEffectConfigurations 这个数组里的音效配置对象是有顺序的，这个顺序最终将和 Audio Unit 在 AUGraph 中的顺序保持一致。如果你不了解 Audio Unit 在 AUGraph 中的顺序对最终产生的音效有什么影响，其实也无妨，实际上你随意地将音频对象排列最终产生的效果用肉耳听起来差别也不大（若你有更高的追求，那么你需要理解这个顺序的意义）。

3.4.1.10 混音

当前版本的 SDK 允许主播在推流的同时，播放本地音频文件。主播麦克风录入的声音，在经过音效处理(如果有)后，会与音频文件的内容混合，然后推流出去让观众听到。同时，如果主播开启了返听功能，亦可以从耳机听到音频文件播放出的声音。

场景举例：直播中，主播唱歌，通过播放音频文件来获得伴奏。结合返听功能，主播可以从耳机听到伴奏音乐以及自己的唱出的歌声。同时观众最终听到的也是混合了伴奏的主播的歌声。

要开启音频文件播放功能,首先需要构造播放器实例,通过如下方法构造

```
PLAudioPlayer *player = [mediaStreamingSession audioPlayerWithFilePath:@"audio file path"];
```

之后，所有与音频文件播放相关的功能就都基于 player 进行了。

当你播放完音频文件之后，且不打算再使用该功能时，需要释放掉 player，可通过调用

```
[mediaStreamingSession closeCurrentAudio];
```

来释放之前获取的播放器实例。

注意：播放器使用完必须关闭，否则它将一直占用着资源（例如音频文件的句柄）。

每当音频文件播放完毕，会回调如下方法询问你是否把该音频文件重新播放一遍

```
– (BOOL)didAudioFilePlayingFinishedAndShouldAudioPlayerPlayAgain:(PLAudioPlayer *)audioPlayer
```

该方法可以让你知道音频文件什么时候播放完毕，同时通过返回一个 BOOL 值，来控制播放器的行为。例如，如果你想做单曲循环效果，可以如此实现该方法

```
– (BOOL)didAudioFilePlayingFinishedAndShouldAudioPlayerPlayAgain:(PLAudioPlayer *)audioPlayer {

    return YES;

}
```

如果你想实现顺序播放效果，可以如此实现该方法

```
– (BOOL)didAudioFilePlayingFinishedAndShouldAudioPlayerPlayAgain:(PLAudioPlayer *)audioPlayer {

    audioPlayer.audioFilePath = @"/path/to/next/audio/file/name.mp3";

    return YES;

}
```

3.4.2 DNS 优化

在大陆一些地区或特别的运营商线路，存在较为普遍的 DNS 劫持问题，而这对于依赖 DNS 解析 rtmp 流地址的 PLMediaStreamingKit 来说是很糟糕的情况，为了解决这一问题，我们引入了 HappyDNS 这个库，以便可以实现 httpDNS，localDNS 等方式解决这类问题。

HappyDNS

你可以点击这里 跳转到 HappyDNS 的 GitHub 主页，在那里查看更详细的介绍和使用。

默认情况下，你所创建的 PLMediaStreamingSession 对象，内部持有一个 HappyDNS 对应的 manager 对象，来负责处理 DNS 解析。

如果你期望按照不同的规则来做 DNS 解析，那么你可以在创建 PLMediaStreamingSession 前，创建好自己的 QNDnsManager 对象，我们在 PLMediaStreamingSession 中提供了一个 init 方法满足这类需求，你可以传递自己的 QNDnsManager 对象给 PLMediaStreamingSession，从而定制化 DNS 解析。

3.4.3 流状态获取

在 PLMediaStreamingKit 中，通过反馈 PLMediaStreamingSession 的状态来反馈流的状态。我们定义了几种状态，确保 PLMediaStreamingSession 对象在有限的几个状态间切换，并可以较好的反应流的状态。

状态名	含义
PLStreamStateUnknow	初始化时指定的状态，不会有任何状态会跳转到这一状态
PLStreamStateConnecting	RTMP 流链接中的状态
PLStreamStateConnected	RTMP 已连接成功时的状态
PLStreamStateDisconnecting	RTMP 正常断开时，正在断开的状态
PLStreamStateDisconnected	RTMP 正常断开时，已断开的状态
PLStreamStateAutoReconnecting	正在等待自动重连状态
PLStreamStateError	因非正常原因导致 RTMP 流断开，如包发送失败、流校验失败等

3.4.3.1 state 状态回调

state 状态对应的 Delegate 回调方法是

```
– (void)mediaStreamingSession:(PLMediaStreamingSession *)session streamStateDidChange:(PLStreamState)state;
```

只有在正常连接，正常断开的情况下跳转的状态才会触发这一回调。所谓正常连接是指通过调用 –startStreamingWithFeedback: 方法使得流连接的各种状态，而所谓正常断开是指调用 –stopStreaming 方法使得流断开的各种状态。所以只有以下四种状态会触发这一回调方法。

- PLStreamStateConnecting
- PLStreamStateConnected

- PLStreamStateDisconnecting
- PLStreamStateDisconnected

3.4.3.2 error 状态回调

error 状态对应的 Delegate 回调方法是

```
– (void)mediaStreamingSession:(PLMediaStreamingSession *)session didDisconnectWithError:(NSError *)error;
```

除了调用 –stopStreaming 之外的所有导致流断开的情况，都被归属于非正常断开的情况，此时就会触发该回调。对于错误的处理，我们不建议触发了一次 error 后就停掉，最好可以在此时尝试有限次数的重连，详见重连小节。

3.4.3.3 status 状态回调

除了 state 作为流本身状态的切换，我们还提供了流实时情况的反馈接口

```
– (void)mediaStreamingSession:(PLMediaStreamingSession *)session streamStatusDidUpdate:(PLStreamStatus
*)status;
```

默认情况下，该回调每隔 3s 调用一次，每次包含了这 3s 内音视频的 fps 和总共的码率（注意单位是 kbps）。你可以通过 PLMediaStreamingSession 的 statusUpdateInterval 属性来读取或更改这个回调的间隔。

3.4.3.4 产品层面的反馈

status 的状态回调可以很好的反应发送情况，及网络是否流畅，是否拥塞。所以此处可以作为产品层面对弱网情况决策的一个入口。

一般的，当 status.videoFPS 比预设的 FPS 明显小时（小于等于 20%），并且维持几秒都是如此，那么就可以判定为当前主播所在的网络为弱网环境，可以给主播视觉上的提示，或者主动降低编码配置，甚至直接断掉主播的流，这些都由具体的产品需求而定，而此处只是给出一个入口的提示和建议。

3.4.4 网络异常处理

直播中，网络异常的情况比我们能意料到的可能会多不少，常见的情况一般有

- 网络环境切换，比如 3G/4G 与 Wi-Fi 环境切换
- 网络不可达，网络断开属于这一类
- 带宽不足，可能触发发送失败
- 上行链路不佳，直接影响流发送速度

作为开发者我们不能乐观的认为只要是 Wi-Fi 网就是好的，因为即便是 Wi-Fi 也有可能因为运营商上行限制，共享网络带宽等因素导致以上网络异常情况的出现。

为何在直播中要面对这么多的网络异常情况，而在其他上传/下载中很少遇到的，这是因为直播对实时性的要求使得它不得面对这一情况，即无论网络是否抖动，是否能一直良好，直播都要尽可能是可持续，可观看的状态。

3.4.4.1 重连

PLMediaStreamingSession 内置了自动重连功能，但默认处于关闭状态。之所以默认关闭，一方面是考虑到 App 的业务逻辑场景多样而负责，对于直播重连的次数、时机、间隔都会有不同的需求，此时让开发者自己来决定是否重连，以及尝试重连的次数会更加合理；另一方面是兼容旧版本业务层面可能已实现的自动重连逻辑。

如果你想直接使用内置的自动重连功能，可通过将 PLMediaStreamingSession 的 autoReconnectEnable 属性设置为 YES 来开启，并需要注意如下几点：

- 自动重连次数上限目前设定为 3 次，重连的等待时间会由首次的 0~2s 之间逐步递增到第三次的 4~6s 之间
- 等待重连期间，streamState 处于 PLStreamStateAutoReconnecting 状态，业务层可根据该状态来更新用户界面
- 网络异常的 error Delegate 回调只有在达到最大重连次数后还未连接成功时才会被触发

若你想自己实现自动重连逻辑，可以利用以下网络异常所触发的 error Delegate 回调接口来添加相应的逻辑：

```
– (void)mediaStreamingSession:(PLMediaStreamingSession *)session didDisconnectWithError:(NSError *)error;
```

你可以在这个方法内通过重新调用 –startStreamingWithFeedback: 方法来尝试重连。此处建议不要立即重连，而是采用重连间隔加倍的方式，比如共尝试 3 次重连，第一次等待 0.5s，第二次等待 1s，第三次等待 2s，这样的方式主要考虑到弱网时网络带宽的缓解需要时间，而加倍重连可以更容易在网络恢复的时候连接，而非在网络已经拥塞时还不断做无用功的重连。

PLMediaStreamingSession 内置了网络切换监测功能，但默认处于关闭状态。开启后，网络在 WWAN(3G/4G) 和 Wi-Fi 之间相互切换时，我们提供了一个回调 connectionChangeActionCallback 属性，它的函数签名如下

```
typedef BOOL(^ConnectionChangeActionCallback)(PLNetworkStateTransition transition);
```

该回调函数传入参数为当前网络的切换状态 PLNetworkStateTransition。 返回值为布尔值，YES 表示在某种切换状态下允许推流自动重启，NO 则代表该状态下不应自动重启。例如在 PLNetworkStateTransitionWWANToWiFi 状态，即网络从 3G/4G 切换到 Wi-Fi 后，基于节省流量等需求考虑，你可能需要进行一次快速的重连，使得数据可以通过 Wi-Fi 网络发送，此时返回 YES 即可。反之，如果推流过程中 Wi-Fi 断掉切换到 3G/4G，此时在未征得用户同意使用移动流量推流时，可返回 NO 不自动重启推流。以下为参考逻辑

```
session.connectionChangeActionCallback = ^(PLNetworkStateTransition transition) {
        switch (transition) {
            case PLNetworkStateTransitionWWANToWiFi:
                return YES;

            case PLNetworkStateTransitionWiFiToWWAN:
                return NO;

            default:
                break;
        }

        return NO;
    };
```

如果该属性未被初始化赋值，则 SDK 内部出于节省用户移动网络流量的目的，会默认在 Wi-Fi 切换到 3G/4G 时断开推流。此时，你可以自行监听网络状态，调用 –restartStreamingWithFeedback: 方法来快速重连。

3.4.4.2 弱网优化

移动直播过程中存在着各种各样的网络挑战。由于无线网络相对于有线网络，可靠性较低，会经常遇到信号覆盖不佳导致的高丢包、高延时等问题，特别是在用网高峰期，由于带宽有限，网络拥塞的情况时有发生。自 v2.1.3 起，PLMediaStreamingKit 内置了一套弱网优化方案，可以满足以下两个诉求：

- 能动态地适应网络质量，即在质量不佳的网络下，能够自动下调视频编码的输出码率和帧率，而当网络质量恢复稳定时，输出码率和帧率也应得到相应回升，并能在调节过程中使得码率与帧率变化相对平稳。
- 在直播端网络质量稳定时，确保编码器输出的码率和帧率恒定在一个期望的最高值，以提供良好的清晰度和流畅度。

这套弱网优化方案包含两个工作模块：

- 自适应码率模块，能够在期望码率与设定的最低码率间做出调节，适应网络抖动引发的数据带宽变化。
- 动态帧率模块，能够在期望帧率与一个最低帧率间做出调节，动态调整输出的视频数据量。

这两个模块可并行工作，可以单独开启或关闭，开发者可根据自己的业务场景来决定该方案的应用形态。一般情况下，如果开发者想使用该解决方案，建议将两个调节模块都开启，可达到我们测试的最佳效果。利用码率与帧率调整相互配合作用，一方面有效控制网络波动情况下的音视频数据发送量，缓解网络拥塞，同时又能给播放端带来流畅的观看体验；另一方面在网络质

量恢复时能够确保音视频的码率帧率是以设定的最优质量配置进行推流，并且能使不同码率帧率配置切换时以一种更为平滑的方式进行，不会给观看端带来画质波动的突兀感。

自适应码率调节可以通过 PLMediaStreamingSession 的如下接口开启:

```
– (void)enableAdaptiveBitrateControlWithMinVideoBitRate:(NSUInteger)minVideoBitRate;
```

其关闭接口为:

```
– (void)disableAdaptiveBitrateControl;
```

开启该机制时，需设置允许向下调节的最低码率（注意其单位为 bps，如设置最低为 200 Kbps，应传入参数值为 200*1024），以便使自动调整后的码率不会低于该范围。该机制根据网络吞吐量及 TCP 发送时间来调节推流的码率，在网络带宽变小导致发送缓冲区数据持续增长时，SDK 内部将适当降低推流码率，若情况得不到改善，则会重复该过程直至平均码率降至用户设置的最低值；反之，当一段时间内网络带宽充裕，SDK 将适当增加推流码率，直至达到预设的推流码率。

动态帧率的开关为 PLMediaStreamingSession 的 dynamicFrameEnable 属性，开启后，自动调整的最大帧率不会超过预设在 videoStreamingConfiguration 配置中的 expectedSourceVideoFrameRate，最低不会小于 10 FPS。

默认情况下，这两个模块处于关闭状态，是为了兼容旧版本中开发者可能已自行实现的弱网调节机制。若开发者想使用我们的内置方案，请确保您自定义的机制已被关闭。

3.4.5 水印和美颜

3.4.5.1 水印

PLMediaStreamingKit 支持内置水印功能，你可以根据自己的需要添加水印或移除水印，并且能够自由设置水印的大小和位置。需要注意的是水印功能对预览和直播流均生效。

添加水印

```
–(void)setWaterMarkWithImage:(UIImage *)wateMarkImage position:(CGPoint)position;
```

该方法将为直播流添加一个水印，水印的大小由 wateMarkImage 的大小决定，位置由 position 决定，需要注意的是这些值都是以采集数据的像素点为单位的。例如我们使用 AVCaptureSessionPreset1280x720 进行采集，同时 wateMarkImage.size 为 (100, 100) 对应的 origin 为 (200, 300)，那么水印的位置将在大小为 1280x720 的采集画幅中位于 (200, 300) 的位置，大小为 (100, 100)。

移除水印

```
–(void)clearWaterMark;
```

该方法用于移除已添加的水印

3.4.5.2 美颜

'PLMediaStreamingSession' 支持内置美颜功能，你可以根据自己的需要选择开关美颜功能，并且能够自由调节包括美颜，美白，红润等在内的参数。需要注意的是水印功能对预览和直播流均生效。

按照默认参数开启或关闭美颜

```
–(void)setBeautifyModeOn:(BOOL)beautifyModeOn;
```

设置美颜程度，范围为 0 ~ 1

```
–(void)setBeautify:(CGFloat)beautify;
```

设置美白程度，范围为 0 ~ 1

```
–(void)setWhiten:(CGFloat)whiten;
```

设置红润程度，范围为 0 ~ 1

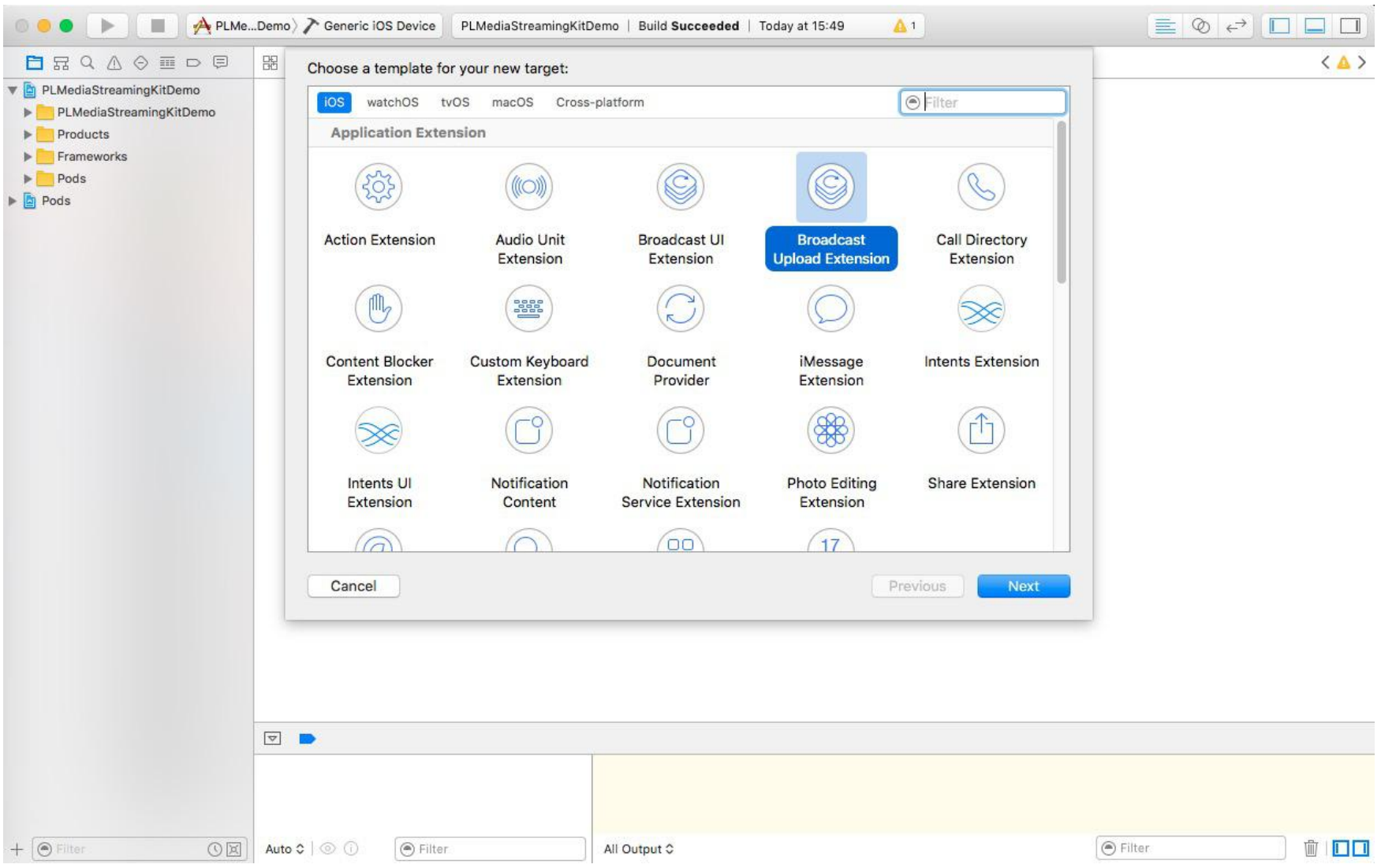
```
–(void)setRedden:(CGFloat)redden;
```

3.4.6 录屏推流

PLStreamingKit 支持 iOS 10 新增的录屏推流（ReplayKit Live）功能，开发者可通过构建 App Extension 来调用推流 API 实现实时游戏直播等功能。需要注意的是，实时直播需要游戏或 App 本身实现对 ReplayKit 的支持。

3.4.6.1 创建 Broadcast Upload Extension

在原有直播 App 中添加一个类型为 Broadcast Upload Extension 的新 Target，如图所示：



Xcode 会额外自动创建一个类型为 Broadcast UI Extension 的 Target，用于显示调用 Broadcast Upload Extension 的用户界面。

3.4.6.2 添加推流管理类

创建推流 API 调用管理类，添加头文件引用：

```
#import <PLMediaStreamingKit/PLStreamingKit.h>
```

头文件参考

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



#import <PLMediaStreamingKit/PLStreamingKit.h>



@interface BroadcastManager : NSObject
```

```
@property (nonatomic, strong) PLStreamingSession *session;



+ (instancetype)sharedBroadcastManager;

– (PLStreamState)streamState;

– (void)pushVideoSampleBuffer:(CMSampleBufferRef)sampleBuffer;

– (void)pushAudioSampleBuffer:(CMSampleBufferRef)sampleBuffer withChannelID:(const NSString *)channelID;



@end
```

类实现参考

```
@interface BroadcastManager ()<PLStreamingSessionDelegate>



@end



@implementation BroadcastManager



static BroadcastManager *_instance;



– (instancetype)init

{

    if (self = [super init]) {

        [PLStreamingEnv initEnv];



        PLVideoStreamingConfiguration *videoConfiguration = [[PLVideoStreamingConfiguration alloc]
initWithVideoSize:CGSizeMake(1280, 720) expectedSourceVideoFrameRate:24 videoMaxKeyframeInterval:24*3
averageVideoBitRate:1000*1024 videoProfileLevel:AVVideoProfileLevelH264High41];
```

```
        PLAudioStreamingConfiguration *audioConfiguration = [PLAudioStreamingConfiguration
defaultConfiguration];
        audioConfiguration.inputAudioChannelDescriptions = @[kPLAudioChannelApp, kPLAudioChannelMic];

        self.session = [[PLStreamingSession alloc] initWithVideoStreamingConfiguration:videoConfiguration
                                                                    audioStreamingConfiguration:audioConfiguration
                                                                                stream:nil];
        self.session.delegate = self;
        #warning 以下 pushURL 需替换为一个真实的流地址
        NSString *pushURL = nil;
        dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(0.5 * NSEC_PER_SEC)),
dispatch_get_main_queue(), ^{
            [self.session startWithPushURL:[NSURL URLWithString:pushURL]
feedback:^(PLStreamStartStateFeedback feedback) {
                if (PLStreamStartStateSuccess == feedback) {
                    NSLog(@"connect success");
                } else {
                    NSLog(@"connect failed");
                }
            }];
        });
    }
    return self;
}

+ (void)initialize
```

```
{
    _instance = [[BroadcastManager alloc] init];
}

- (PLStreamState)streamState
{
    return self.session.streamState;
}

- (void)pushVideoSampleBuffer:(CMSampleBufferRef)sampleBuffer
{
    [self.session pushVideoSampleBuffer:sampleBuffer];
}

- (void)pushAudioSampleBuffer:(CMSampleBufferRef)sampleBuffer withChannelID:(const NSString *)channelID
{
    [self.session pushAudioSampleBuffer:sampleBuffer withChannelID:channelID completion:nil];
}

+ (instancetype)sharedBroadcastManager
{
    return _instance;
}

// 实现其他必要的协议方法
```

```
- (void)streamingSession:(PLStreamingSession *)session didDisconnectWithError:(NSError *)error
{
    NSLog(@"error : %@", error);
}

- (void)streamingSession:(PLStreamingSession *)session streamStatusDidUpdate:(PLStreamStatus *)status
{
    NSLog(@"%@", status);
}

@end
```

注意 PLAudioStreamingConfiguration 实例生成时必需注册音频流来源

```
audioConfiguration.inputAudioChannelDescriptions = @[kPLAudioChannelApp, kPLAudioChannelMic];
```

其中 kPLAudioChannelApp 对应于 RPSampleBufferTypeAudioApp，是 ReplayKit Live 回调的 app 音频数据，kPLAudioChannelMic 对应于 RPSampleBufferTypeAudioMic，是 ReplayKit Live 回调的 mic 音频数据。之所以需要显示声明，是为了在 PLStreamingKit 在音频编码前将两路音频流进行混音。

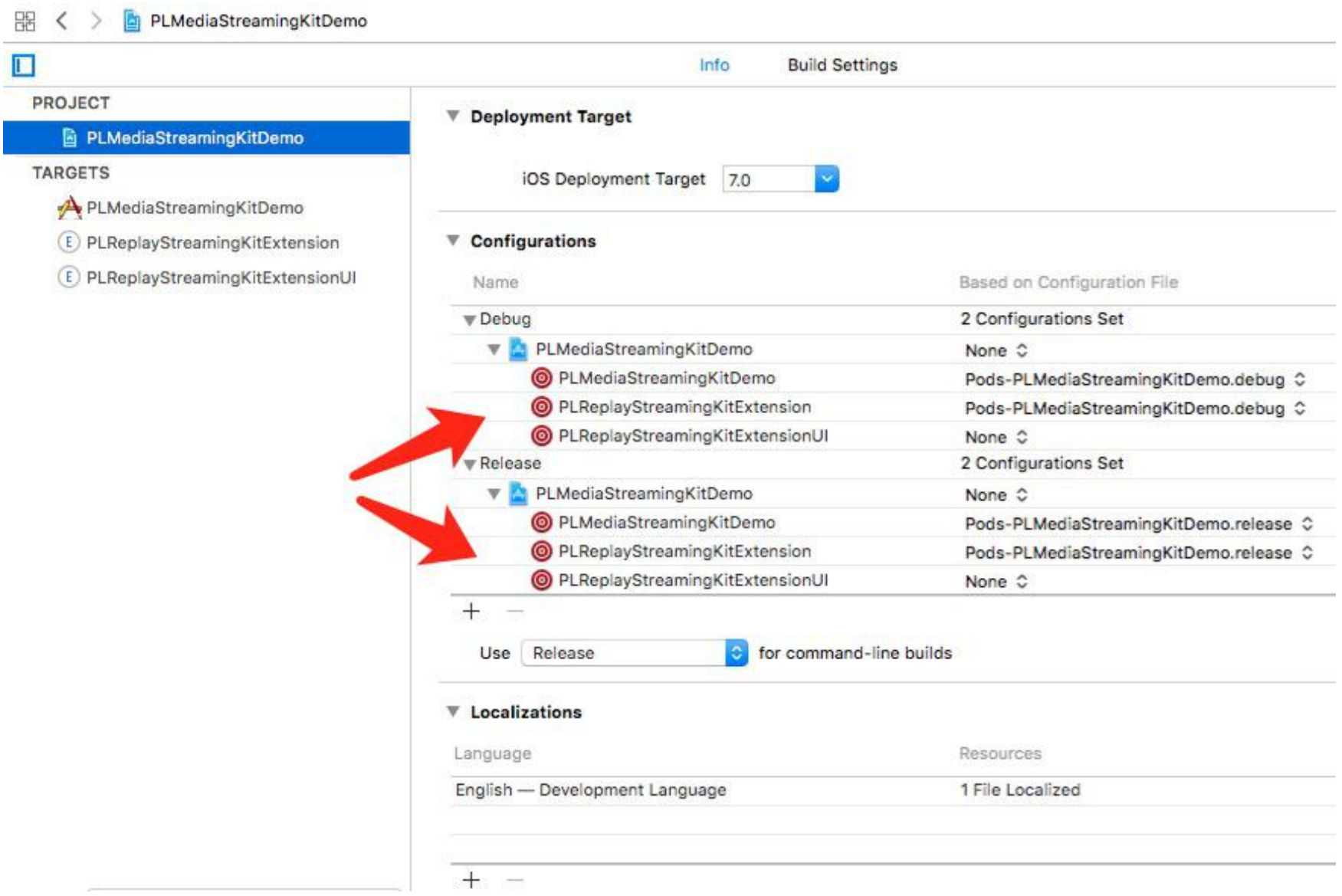
在自动生成的 SampleHandler.m 中实现 RPBroadcastSampleHandler 协议部分方法如下：

```
- (void)processSampleBuffer:(CMSampleBufferRef)sampleBuffer withType:(RPSampleBufferType)sampleBufferType {
    if ([BroadcastManager sharedBroadcastManager].streamState == PLStreamStateConnected) {
        switch (sampleBufferType) {
            case RPSampleBufferTypeVideo:
                // Handle video sample buffer
                [[BroadcastManager sharedBroadcastManager] pushVideoSampleBuffer:sampleBuffer];
                break;
            case RPSampleBufferTypeAudioApp:
```

```
                    // Handle audio sample buffer for app audio

                    [[BroadcastManager sharedBroadcastManager] pushAudioSampleBuffer:sampleBuffer
withChannelID:kPLAudioChannelApp];

                    break;

                case RPSampleBufferTypeAudioMic:

                    // Handle audio sample buffer for mic audio

                    [[BroadcastManager sharedBroadcastManager] pushAudioSampleBuffer:sampleBuffer
withChannelID:kPLAudioChannelMic];

                    break;



                default:

                    break;

            }

        }

}
```

3.4.6.3 一些注意点

如果你使用 CocoaPods 管理依赖库，可能会在编译 broadcast extension target 时遇到 link error，此时请检查 Podfile 里是否为 broadcast extension target 添加相应的依赖；或者可检查以下工程设置是否更改：



3.4.7 后台推图片

App 一旦切到后台，摄像头的采集功能会被系统暂停掉，观众会看到主播的画面会卡住，此时如果能推送一张图片告诉观众主播进入后台了，将带来更好的观看体验。具体实现方法如下：开发者需要自行监听 App 进入后台/前台的通知，当 App 进入后台时，使用 PLMediaStreamingSession 的如下接口

```
– (void)setPushImage:(nullable UIImage *)image;
```

设置一张推流图片，PLMediaStreamingSession 会持续为您推送该图片，当 App 返回前台不再需要推送图片时，重新调用以上接口并传入 nil 即可。

3.4.8 QUIC 推流

QUIC 是基于 UDP 开发的可靠传输协议，在弱网下拥有更好的推流效果，相比于 TCP 拥有更低的延迟，可抵抗更高的丢包率。可通过下面接口开启/关闭 QUIC 推流：

```
@property (nonatomic, assign, getter=isQuicEnable) BOOL quicEnable;
```

使用 QUIC 推流前，请确保直播服务端支持 QUIC 协议。