# Unity Export 导出到 Android Studio 再打包大致过程原理

deepwaterooo

November 25, 2022

## Contents



# 1 Unity 编译 Android 的原理解析和 apk 打包分析

- 最近由于想在 Scene 的脚本组件中，调用 Android 的 Activity 的相关接口，就需要弄明白 Scene 和 Activity 的实际对应关系，并对 Unity 调用 Android 的部分原理进行了研究。
- 本文主要探讨 Scene 和 Activity 之间的关系，以及 Unity 打包 apk 和 Android studio 打包 apk 的差别在什么地方？找到这种差别之后，可以怎么运用起来?
- 本文需要用到的工具：
  - Android 反编译工具——apktool
  - Android studio 自带的反编译功能

## 1.1 一、将 Unity 的 Scene 编译成 apk，apk 的程序入口会是什么？

- 新建一个 Unity 项目，创建一个 Scene，将 Unity 工程编译打包成 apk。
- 对编译出来的 apk，利用 apktool 进行反编译：apktool d unityTest.apk
- 得到的 AndroidManifest 文件如下：

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:installLocation="preferExternal"
    package="com.xfiction.p1"
    platformBuildVersionCode="25"
    platformBuildVersionName="7.1.1">

  <supports-screens
      android:anyDensity="true"
      android:largeScreens="true"
      android:normalScreens="true"
      android:smallScreens="true"
      android:xlargeScreens="true"/>

  <application
      android:banner="@drawable/app_banner"
      android:debuggable="false"
      android:icon="@drawable/app_icon"
      android:isGame="true"
      android:label="@string/app_name"
      android:theme="@style/UnityThemeSelector">

    <activity android:name="com.unity3d.player.UnityPlayerActivity"

        android:configChanges="locale|fontScale|keyboard|keyboardHidden|mcc|mnc|navigation|orientation|screenLayout|screenS
        android:label="@string/app_name"
        android:launchMode="singleTask"
        android:screenOrientation="fullSensor">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
        <category android:name="android.intent.category.LEANBACK_LAUNCHER"/>
      </intent-filter>
      <meta-data android:name="unityplayer.UnityActivity" android:value="true"/>
    </activity>
  </application>
  <uses-feature android:glEsVersion="0x00020000"/>
  <uses-feature android:name="android.hardware.touchscreen" android:required="false"/>
  <uses-feature android:name="android.hardware.touchscreen.multitouch" android:required="false"/>
  <uses-feature android:name="android.hardware.touchscreen.multitouch.distinct" android:required="false"/>
</manifest>
```

- 由该 AndroidManifest 文件可知，系统仍然存在主 Activity，名字为 com.unity3d.player.UnityPlayerA
- 言下之意，编译只包含 Scene 的 Unity 工程，打包成 Android apk，会以 com.unity3d.player.UnityPlay 作为主程序入口，那么问题来了，Scene 如何加载显示到这个 UnityPlayerActivity 呢？

## 1.2 二、UnityPlayerActivity 如何加载 Unity 中的 Scene？

### 1.2.1 UnityPlayerActivity

- 这个就要从 UnityPlayerActivity 源码入手了，Android 工程中使用 UnityPlayerActivity 需要依赖到 Unity 的 Android 插件 classes.jar（位于 Unity 安装目录，可以用 everything 软件查找查找得到），对其进行反编译得到 UnityPlayerActivity 的部分源码：

```
public class UnityPlayerActivity extends Activity {
    protected UnityPlayer mUnityPlayer;
    protected void onCreate(Bundle var1) {
        this.requestWindowFeature(1);
        super.onCreate(var1);
        this.getWindow().setFormat(2);
        this.mUnityPlayer = new UnityPlayer(this);
```

```
        this.setContentView(this.mUnityPlayer); // <<<<<<<<<<<<<<<<<<<< 最终的界面显示需要依赖到 UnityPlayer 的实例

        this.mUnityPlayer.requestFocus();
    }
}
```

- 虽然经过混淆，看起来比较费劲，但从代码 this.setContentView(this.mUnityPlayer) 可以看出，最终的界面显示需要依赖到 UnityPlayer 的实例。另外由于 Google 也做了一套 Unity VR 的 SDK，与 UnityPlayerActivity 相对应的类，就是 GoogleUnityActivity，下面也对它进行分析。

### 1.2.2 从 GoogleUnityActivity.java 再人手分析

- GoogleUnityActivity 是 google 推出的 VR SDK 中，用于实现 Unity Activity 的类，通过 google 查询其源码发现：1. GoogleUnityActivity.java 实际上的布局文件 activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <FrameLayout
        android:id="@+id/android_view_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@android:color/transparent" />
</FrameLayout>
```

- 布局文件中没有具体的内容，只包含一个 FrameLayout 布局。

### 1.2.3 重点看 GoogleUnityActivity 的 onCreate 函数：

```
public class GoogleUnityActivity   extends Activity
    implements ActivityCompat.OnRequestPermissionsResultCallback {
    protected void onCreate(Bundle savedInstanceState) {
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main); // <<<<<<<<<<<<<<<<<<<< 这里是说可能会有两三种不同的实现方式吗?
        setContentView(R.id.activity_main.xml)

        mUnityPlayer = new UnityPlayer(this);
        if (mUnityPlayer.getSettings().getBoolean("hide_status_bar", true)) {
            getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                    WindowManager.LayoutParams.FLAG_FULLSCREEN);
        }
        ((ViewGroup) findViewById(android.R.id.content)).addView(mUnityPlayer.getView(), 0);
        mUnityPlayer.requestFocus();
    }
}
```

- mUnityPlayer 作为 FrameLayoutView 加入到 view 集合中进行显示，注意这里查找的 id 是 android.R.id.content。根据官方对这个 id 的解释：android.R.id.content gives you the root element of a view, without having to know its actual name/type/ID. Check out Get root view from current activity

- 由此可见，GoogleUnityActivity 的实现原理，是创建一个只包含 FrameLayout 的空的帧布局，随后通过 addView 将 UnityPlayer 中的 View 加载到 GoogleUnityActivity 中进行显示。

- 看起来跟 UnityPlayerActivity 有异曲同工之妙，两者牵涉的类都是 UnityPlayer。

### 1.2.4 UnityPlayer 究竟是一个什么类呢？

- 对 classes.jar 包进行反编译得到 UnityPlayer 的部分代码：

```
public class UnityPlayer extends FrameLayout implements com.unity3d.player.a.a {
    public static Activity currentActivity = null;
    public UnityPlayer(ContextWrapper var1) {
        super(var1);
        if(var1 instanceof Activity) {
            currentActivity = (Activity)var1;
        }
    }
    public View getView() {
           return this;
    }
    public static native void UnitySendMessage(String var0, String var1, String var2);
    private final native boolean nativeRender();
    public void onCameraFrame(final com.unity3d.player.a var1, final byte[] var2) {
        final int var3 = var1.a();
        final Size var4 = var1.b();
        this.a(new UnityPlayer.c((byte)0) {
            public final void a() {
            UnityPlayer.this.nativeVideoFrameCallback(var3, var2, var4.width, var4.height);
                var1.a(var2);
            }
        });
    }
}
```

- 从代码中可以发现：
- UnityPlayer 实际上是继承于 FrameLayout；
- 并且自带一个 currentActivity 的成员变量，在构造函数中，直接传入 Activity 的相关参数；
- 在 getView 函数中直接返回该 FrameLayout；
- GoogleUnityActivity 通过 UnityPlayer 的构造函数，将其 context 传递给 UnityPlayer，并赋值给其成员变量 currentActivity。
- 由于 UnityPlayer 类做了混淆，关于渲染的核心功能也封装在 native 代码中，关于 Scene 转换到到 UnityPlayer 作为 FrameLayout，只能做一个简单的推测：通过调用 Android 的 GL 渲染引擎，在 native 层进行渲染，并同步到 FrameLayout 在 UnityPlayerActivity 上进行显示。

## 1.3 三、如何将 Scene 显示在自定义的 Activity 当中 (这里最好能找个真正的例子参考一下)

- 从以上研究的内容可知，假如要从要实现将 Scene 显示在固定的 Activity 当中，则需要对 Activity 的 oncreate 部分的 countview 和 unityplayer 进行处理。最简单的方法是写一个直接继承于 UnityPlayerActivity 或 GoogleUnityActivity 的类，并在类中写所需要的 Unity 调用 Android 的方法。这样 Scene 就会加载在特定的 Activity 当中，Unity c# 通过获取 currentActivity 变量就可以获取到该 Activity，并调用其中的函数。

## 1.4 四、Unity Android 插件需要注意的问题

- Android studio 工程包含多个 module 的依赖，则需要将对应的 module 编译的插件一起拷贝 Plugins/Android/lib 目录当中。
- 在第一步骤下，可以直接删除打包后的 aar library 目录，尤其里面假如带有 unity 的 Android 插件 classesjar，否则会编译报错。
- 多个 module 编译的时候，注意 manifest lablel 相关设置，另外就是 build.gradle 的 minSDKVersion 信息。否则会出现 manifest merger 失败的错误。

- 关于 Unity 的 Android Manifest 文件合并：Unity 编写一个 Scene，Android studio 写一个包含主 Activity 的 aar 包，放在 Plugins/Android 目录当中。用 Unity 编译 apk 出来之后，反编译他的 AndroidManifest 文件, 两个主 Activity，默认显示包含 Scene 的 Activity。解决方法：Unity 的 Manifest 文件合并，把一个 manifest 放到 Plugins/Android 目录下，就不会合并 manifest 了。

## 1.5 五、Unity 打包 Android apk 的结构探究

- 由于 Unity 开发 Android 时，常常设计到 Unity + Visual 和 Android studio 的环境切换，Unity 的开发往往会更快一些，更多的是 Android java 侧的代码编写和调试。
- 这种情况时，有没有一种方法，能够将 Unity 编译好的 Unity Scene 和 c# 相关文件，放到 Android studio 中进行打包，从而实现直接在 Android studio 中进行调试?
- 方法原理倒是很简单，通过对比 Unity 打包的 apk，与普通的 Android apk 的文件差别，找出 Unity 文件存放的目录，随后对应存放到 Android studio 工程目录中，最后通过 Android studio 完成对 Unity 相关文件的打包。
- 首先将 apk 添加 zip 的后缀，方便用 beyond compare 进行对比：
- 发现只是多了 assert/bin 目录，在这个目录之下，可以看到 unity 相关 dll 库
- 将该文件，拷贝到 Android studio 工程的 src/main/assert 目录之下；
- 在 Android studio 调试时，可以将 aar library 工程设置为 app 工程，这样就可以编译 apk 运行到手机了。
- 用 Android studio 对该工程进行编译，发现 assert/bin 目录成功被打包进去。
- 直接 apk install 运行，可以看到跟 Unity 编译打包的 apk，是相同的效果。
- 相反，假如 Android 工程调试好之后，则直接编译成 app 模式修改成 library 模式，进行 build 之后，就会生成 aar 库，此时将 aar 库拷贝到 Plugins/Android/lib 目录当中，注意要删除 aar 库中的 assert/bin，因为这个目录是我们先前从 Unity 拷贝过去的，假如不删除，在 unity 里面会出现重复打包导致的文件冲突的情况。
- 由于当将 Unity 打包之后的 bin 目录拷贝到 Android studio 工程之后，Android studio 此时是一个 library 工程，需要转换为 app 工程。关于这其中涉及到的 Android studio library 和 app 的转换，通过设置 build.gradle 文件来实现：
- app 模式：apply plugin: 'com.android.application'
- library 模式：apply plugin: 'com.android.library'
- 不过在设置这两种模式时，需要注意 applicationId "com.example.yin.myapplication" 的设置，假如是 library 模式，则需要直接注释掉。
- 假如 Android 的 java 部分重新调试好之后，重新将 app 模式改成 library 模式，进行 build，将生成的 aar 包，拷贝到 Unity Android Plugin 目录中，就可以直接在 Unity 看运行效果了。**不过一定要记得删除 Android studio 打包的 aar 文件里面的 assert/bin 目录，以防止在 Unity 中重复打包**。

### 1.6 四、结论：

- Unity 中的 Scene 在 Android 中，其实对应于 Activity 的 FrameLayout，每个 Scene 的运行都有其 Activity 环境，通过 currentActivity 变量可以获取得到。
- 要实现自定义的 Activity 能够具备直接加载 Scene 的功能，则需要其继承于 UnityPlayerActivity 或者 GoogleUnityActivity，再或者，直接自定义实现 UnityActivity 类。
- 提升 Unity+Android Plugin 项目开发效率的方法：直接将 Unity 打包的 apk 中的 assert/bin 目录拷贝到 Android studio 工程的 src/main/assert 目录当中，并且将 Android 工程配置成 app 模式，就可以直接在 Android studio 上面，对整个 Unity+android plugin 的工程进行调试。 Android studio 部分调试好之后，需要修改 build.gradle 文件，重新将 app 模式修改为 library 模式，编译出 aar 包文件，删除原来拷贝过来的 unity 部分，放入到 unity 的 Plugins/Android/lib 目录下进行使用即可。

## 2 Unity 构建安卓原理详解

- 构建系统
- Unity 支持两个 Android 构建系统：__Gradle__ 和 Internal。
- Android 构建过程所涉及的步骤如下：
  - 准备和构建 Unity 资源。
  - 编译脚本。
  - 处理插件。
  - 将资源拆分为放入 APK 和 OBB 的各个部分（如果已选择 __Split Application Binary__）。
  - 使用 AAPT 实用程序构建 Android 资源（仅限内部构建）。
  - 生成 Android 清单。
  - 将库清单合并到 Android 清单中（仅限内部构建）。
  - 将 Java 代码编译为 Dalvik 可执行文件格式 (DEX)（仅限内部构建）。
  - 构建 IL2CPP 库（如果已选择 __IL2CPP Scripting Backend__）。
  - 构建并优化 APK 和 OBB 包。
- Gradle 构建系统
  - Gradle 构建系统使用 Gradle 来构建 APK 或以 Gradle 格式导出项目，然后可以将其导入 Android Studio。选择此构建系统时，Unity 将执行与 Internal 构建系统相同的步骤，但不包括使用 AAPT 进行资源编译、合并清单以及运行 DEX。然后，Unity 生成 build.gradle 文件（以及其他所需的配置文件），并调用 Gradle 可执行文件，在此过程中向其传递任务名称和工作目录。最后，由 Gradle 构建 APK。
  - 有关更多详细信息，请参阅 Gradle for Android。
- Internal 构建系统
  - Internal 构建系统使用 Android SDK 实用程序创建 APK，从而构建并优化 APK 和 OBB 包。

# 3 Unity 是怎么打包 APK 文件的

- 在 Unity 里面有几个特殊的文件夹是跟打包 APK 有关的。首先我们就来了解一下，这些文件夹里面的内容是经历了哪些操作才被放到 APK 里面的呢?
- 在 Unity 的 Assets 目录下，Plugins/Android 无疑是其中的重中之重，首先我们先来看一个常见的 Plugins/Android 目录是什么样子的。

```
-Android
-- ApolloBase
-- ApolloPlugins
-- assets
-- libs
-- res
-- AndroidManifest.xml
```

- 后面的四个是 Android 工程的文件。前面两个文件夹是我们引用的第三方库，他们也会被打包到 APK 中。我们这个时候如果点进去前两个文件夹，我们会发现他们的目录结构跟 Android 这个目录也很像，大概是一下这个样子的。

```
-ApolloPlugins
-- libs
-- res
-- AndroidManifest.xml
-- project.properties
```

- 比较上下两层的目录接口我们可以发现有很多相似的部分，如：libs、res、assets 文件夹以及 AndroidManifest.xml 文件。这些其实都是一个标准的 Android 项目的所需要的文件。Unity 自带的 Android 打包工具的作用就是把上述这几个文件夹里面的内容以固定的方式组织起来压缩到 APK 文件里面。

## 3.1 Android 打包工具都会做什么样的操作。

- **libs 文件夹里面有很多.jar 文件**，以及被放在固定名字的文件夹里面的.so 文件。*.jar 文件是 Java 编译器把.java 代码编译后的文件，Android 在打包的时候会把项目里面的所有 jar 文件进行一次合并、压缩、重新编译变成 classes.dex 文件被放在 APK 根目录下。当应用被执行的时候 Android 系统内的 Java 虚拟机（Dalvik 或者 Art），会去解读 classes.dex 里面的字节码并且执行。把众多 jar 包编译成 classes.dex 文件是打包 Android 应用不可或缺的一步。
  - 看到这里有人可能会想不对啊，这一步只将 jar 包打成 dex 文件，那之前的 java 文件生成 jar 文件难道不是在这一步做吗? 没错，这里用的 jar 包一般是由其他 Android 的 IDE 生成完成后再拷贝过来的。本文后面的部分会涉及到怎么使用 Android 的 IDE 并且生成必要的文件。
- **libs 文件夹的**.so 文件 * 则是可以动态的被 Android 系统加载的库文件，一般是由 C/C++ 撰写而成然后编译成的二进制文件。要注意的是，由于实际执行这些二进制库的 CPU 的架构不一样，所以同样的 CC++ 代码一般会针对不同的 CPU 架构生成几分不同的文件。这就是为什么 libs 文件夹里面通常都有 armeabi-v7a、armeabi、x86 等几个固定的文件夹，而且里面的.so 文件也都是有相同的命名方式。Java 虚拟机在加载这些动态库的时候会根据当前 CPU 的架构来选择对应的 so 文件。有时候这些 so 文件是可以在不同的 CPU 架构上执行的，只是在不对应的架构上执行速度会慢一些，所以当追求速度的时候可以给针对每个架构输出对应的 so 文件，当追求包体大小的时候输出一个 armeabi 的 so 文件就可以了。
- **assets 文件夹**，这个里面的东西最简单了，在打包 APK 的时候，这些文件里面的内容会被原封不动的被拷贝到 APK 根目录下的 assets 文件夹。这个文件夹有几个特性。
  - 里面的文件基本不会被 Android 的打包工具修改，应用里面要用的时候可以读出来。
  - 打出包以后，这个文件夹是只读的，不能修改。
  - 读取这个文件夹里面的内容的时候要通过特定的 Android API 来读取，参考 getAssets()。

  - 基于上述两点，在 Unity 中，要读取这部分内容要通过 WWW 来进行加载。
  - 除了 Plugins/Android 内的所有 assets 文件夹里面的文件会连同 StreamingAssets 目录下的文件一起被放到 APK 根目录下的 assets 文件夹。

- **res 文件夹**里面一般放的是 xml 文件以及一些图片素材文件。xml 文件一般来说有以下几种：
  - 布局文件，被放在 res 中以 layout 开头的文件夹中，文件里描述的一般都是原生界面的布局信息。由于 Unity 游戏的显示是直接通过 GL 指令来完成的，所以我们一般不会涉及到这些文件。
  - 字符串定义文件，一般被放到 values 文件夹下，这个里面可以定义一些字符串在里面，方便程序做国际
  - 化还有本地化用。当然有时候被放到里面的还有其他 xml 会引用到的字符串，一般常见的是 app 的名称。
  - 动画文件，一般定义的是 Android 原生界面元素的动画，对于 Unity 游戏，我们一般也不会涉及他。
  - 图片资源，一般放在以 drawable 为开头的文件夹内。这些文件夹的后缀一般会根据手机的像素密度来来进行区分，这样我们可以往这些文件夹内放入对应像素密度的图片资源。
  - 例如后缀为 ldpi 的 drawable 文件夹里面的图片的尺寸一般来说会是整个系列里面最小的，因为这个文件夹的内容会被放到像素密度最低的那些手机上运行。而一般 1080p 或者 2k 甚至 4k 的手机在读取图片的时候会从后缀为 xxxxhdpi 的文件夹里面去读，这样才可以保证应用内的图像清晰。图片资源在打包过程中会被放到 APK 的 res 文件夹内的对应目录。
  - Android 还有其他一些常见的 xml 文件，这里就不一一列举了。
  - res 文件夹下的 xml 文件在被打包的时候会被转换成一种读取效率更高的一种特殊格式（也是二进制的格式），命名的时候还是以 xml 为结尾被放到 APK 包里面的 res 文件夹下，其目录结构会跟打包之前的目录结构相对应。
  - 除了转换 xml 之外，Android 的打包工具还会把 res 文件夹下的资源文件跟代码静态引用到的资源文件的映射给建立起来，放到 APK 根目录的 resources.arsc 文件。这一步可以确保安卓应用启动的时候可以加载出正确的界面，是打包 Android 应用不可或缺的一步。

- **AndroidManifest.xml**，这份文件太重要了，这是一份给 Android 系统读取的指引，在 Android 系统安装、启动应用的时候，他会首先来读取这个文件的内容，分析出这个应用分别使用了那些基本的元素，以及应该从 classes.dex 文件内读取哪一段代码来使用又或者是应该往桌面上放哪个图标，这个应用能不能被拿来 debug 等等。在后面的部分会有详细解释。打包工具在处理 Unity 项目里面的 AndroidManifest 文件时会将所有 AndroidManifest 文件的内容合并到一起，也就是说主项目引用到的库项目里面如果也有 AndroidManifest 文件，都会被合并到一起。这样就不需要手动复制粘贴。需要说明的是，这份文件在打包 Android 程序的时候是必不可少的，但是在 Unity 打包的时候，他会先检查 Plugins 目录下有没有这份文件，如果没有就会用一个自带的 AndroidManifest 来代替。此外，Unity 还会自动检查项目中 AndroidManifest 里面的某些信息是不是默认值，如果是的话，会拿 Unity 项目中的值来进行替换。例如，游戏的 App 名称以及图标等。

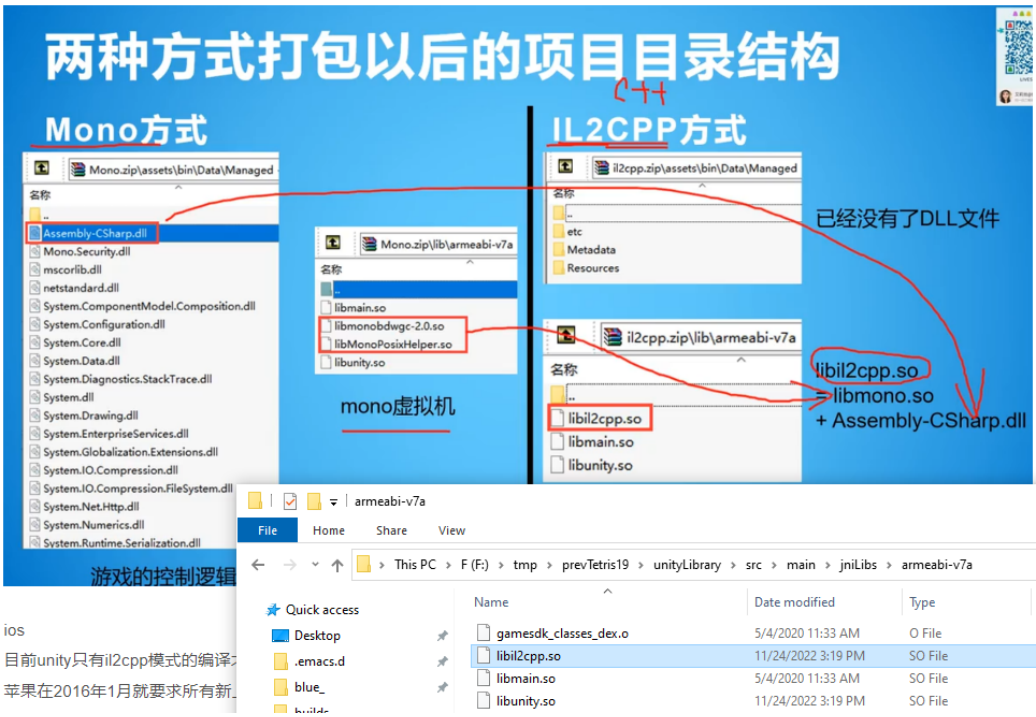
- **project.properties**，这份文件一般只有在库项目里面能看得到，里面的内容极少，就只有一句话 android.library=true。但是少了这份文件 Android 的打包工具就不会认为这个文件夹里面是个 Android 的库项目，从而在打包的时候整个文件夹会被忽略。这有时候不会影响到打包的流程，打包过程中也不会报错，但是打出的 APK 包缺少资源或者代码，一跑就崩溃。关于这份文件，其实在 Unity 的官方文档上并没有详细的描述（因为他实际上是 Android 项目的基础知识），导致很多刚刚接触 Unity-Android 开发的开发者在这里栽坑。曾经有个很早就开始用 Unity 做 Android 游戏的老前辈告诉我要搞定 Unity 中的 Android 库依赖的做法是

用 Eclipse 打开 Plugins/Android 文件夹，把里面的所有的项目依赖处理好就行了。殊不知这样将 Unity 项目跟 Eclipse 项目耦合在一起的做法是不太合理的，会造成 Unity 项目开启的时候缓慢。

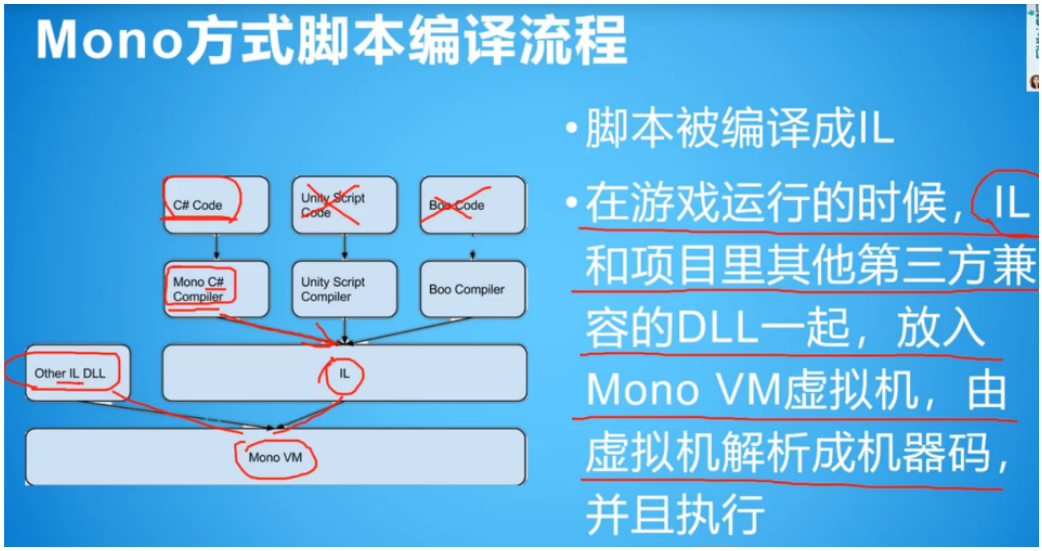
- **其他文件夹例如 aidl 以及 jni** 在 Unity 生成 APK 这一步一般不会涉及到，这里不展开。
- 看到了上述介绍的 Unity 打包 APK 的基础知识我们知道了往 Plugins/Android 目录下放什么样的文件会对 APK 包产生什么样的影响。但是实际上上述的内容只是着重的讲了 Unity 是怎么打包 APK，所以接下来会简述一下打包这个步骤到底是怎么完成的。
- **Android 提供了一个叫做 aapt 的工具，这个工具的全称是 Android Asset Packaging Tool**，这个工具完成了上述大部分的对资源文件处理的工作，而 Unity 则是通过对 Android 提供的工具链（Android Build Tools）的一系列调用从而完成打包 APK 的操作。这里感觉有点像我们写了个 bat/bash 脚本，这个脚本按照顺序调用 Android 提供的工具一样。在一些常见的 Android IDE 里面，这样的“bat/bash 脚本”往往是一个完整的构建系统。最早的 Android IDE 是 Eclipse，他的构建系统是 Ant，是基于 XML 配置的构建系统。后来 Android 团队推出了 Android 专用的 IDE——Android Studio（这个在文章后面会有详述），他的构建系统则是换成了 gradle，从基于 xml 的配置一下子升级到了语言（DSL, Domain Specific Language）的层级，给使用 Android Studio 的人带来更多的弹性。
- 写到这里我想很多人都清楚了要怎么把 Android 的 SDK/插件放到 Unity 里面并且打包到 Unity 里面。这时候应该有人会说，光会放这些文件不够啊，我还需要知道自己怎么写 Android 的代码并且输出相应的 SDK/插件给 Unity 使用啊。1

# 4 unity3d 打包发布篇–MONO 和 IL2CPP 原理

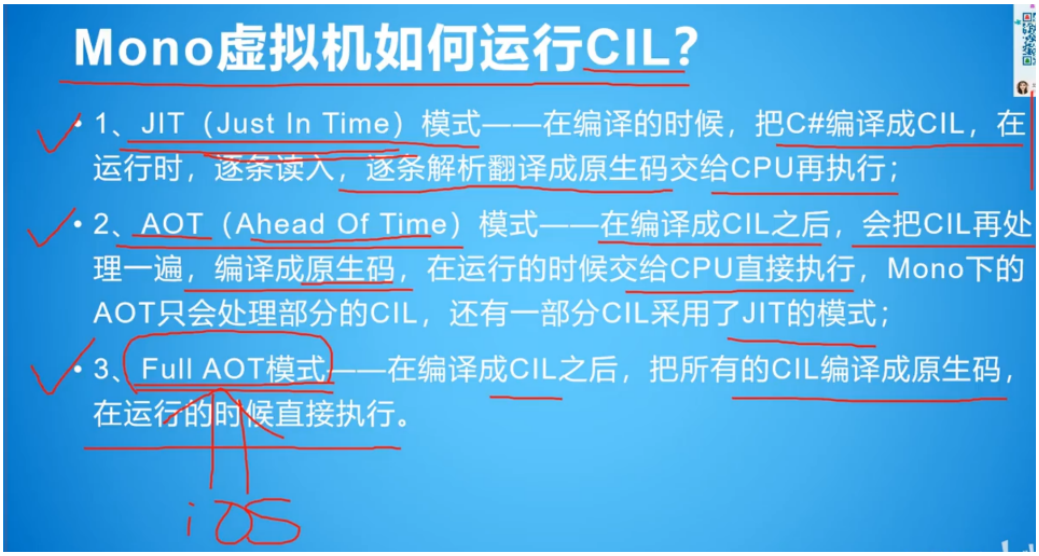
- 两种方式打包以后的项目目录结构



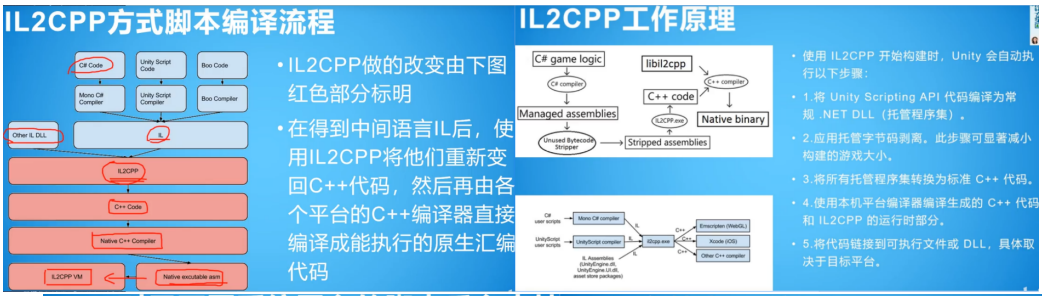
## 4.1 Mono 方面

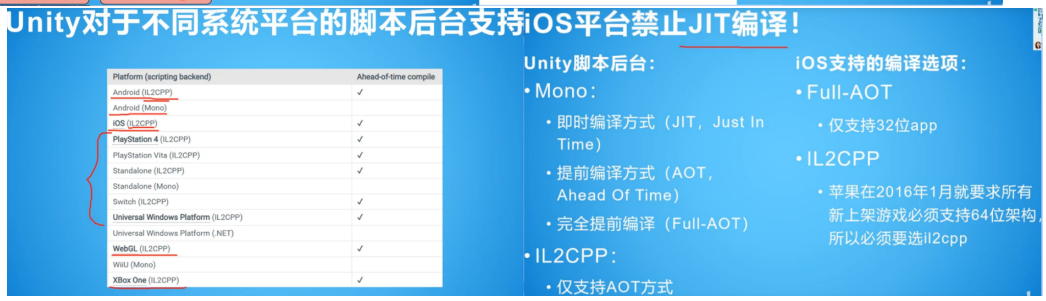



- mono 是一款开源、免费、可定制的跨平台.NET 运行环境。由于.net 只能在 windows，mono 相当于是一个.net CLR 的跨平台变种，就是为了解决跨平台的移植问题。
- 在运行 IL 方面上，热更也需要基于 JIT，安卓支持即时编译 JIT，虽然热更方便，但如果代码中有病毒木马，也一样编译。

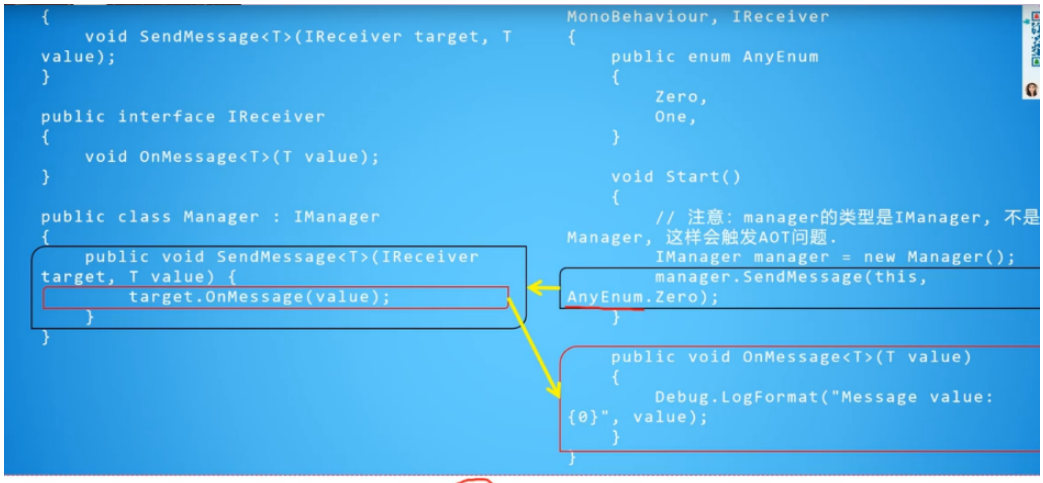


## 4.2 IL2CPP：




- 最早 IOS 是支持 MONO，但 MONO 只能支持 32 位，而且 2016 年后苹果要求必须 64 位。
- IOS 出于安全考虑，不允许 JIT，而且因为禁止脚本为动态分配内存赋予执行权限，所以使用反射会有限制，只能静态编译，只能 FULL AOT 或者 il2cpp，热更相对于安卓就比较麻烦。



```
{
    void SendMessage<T>(IReceiver target, T
value);
}

public interface IReceiver
{
    void OnMessage<T>(T value);
}

public class Manager : IManager
{
    public void SendMessage<T>(IReceiver
target, T value) {
        target.OnMessage(value);
    }
}
```

```
MonoBehaviour, IReceiver
{
    public enum AnyEnum
    {
        Zero,
        One,
    }

    void Start()
    {
        // 注意：manager的类型是IManager，不是
Manager，这样会触发AOT问题.
        IManager manager = new Manager();
        manager.SendMessage(this,
AnyEnum.Zero);
    }

    public void OnMessage<T>(T value)
    {
        Debug.LogFormat("Message value:
{0}", value);
    }
}
```

```
ExecutionEngineException: Attempting to JIT compile method
'Manager:SendMessage<AOTProblemExample/AnyEnum>
(IReceiver,AOTProblemExample/AnyEnum)' while running with --aot-only. at
AOTProblemExample.Start () [0x00000] in <filename unknown>:0
```

- 反射：
  - System.Reflection可用（只要编译器可以推断通过反射使用的代码需要在运行时存在），但System.Reflection.Emit 命名空间中的任何方法不可用
  - 关于System.Reflection.Emit如何像病毒一样动态生成代码，请参见《Unity小白的游戏梦》课程演示
- 序列化：
  - 如果一个类型或一个方法仅通过反射被创建或被调用，则AOT 编译器无法检测到需要为该类型或方法生成代码
- 泛型虚方法：
  - 泛型虚方法由于在编译时类型不确定，编译器也不会在编译期生成针对特定类型的泛型方法调用

- 在有泛型的情况下，代码很可能会报错，因为泛型 T 只有在执行的时候才知道自己的类型，属于动态的，所以静态编译会直接跳过这句代码，在运行的时候就会报错：尝试 JIT 的 error.