目录

1. 对热修复和插件化的理解
2. 热修复,插件化
3. 插件化原理分析

https://www.jianshu.com/p/cb1f0702d59f

http://www.10tiao.com/html/227/201703/2650239063/1.html

一、热修复：

Android 中的类加载器

以下是Android 5.0中的部分源码：

- PathClassLoader.java
- DexClassLoader.java
- BaseDexClassLoader.java
- DexPathList.java

1. PathClassLoader 与 DexClassLoader 的区别

使用场景：

1）PathClassLoader：只能加载已经安装到 Android 系统中的 apk 文件（/data/app 目录），是 Android 默认使用的类加载器。

2）DexClassLoader：可以加载任意目录下的 dex/jar/apk/zip 文件，比 PathClassLoader 更灵活，是实现热修复的重点。

代码差异：

因为 PathClassLoader 与 DexClassLoader 的源码都很简单，我就直接将它们的全部源码复制过来了：

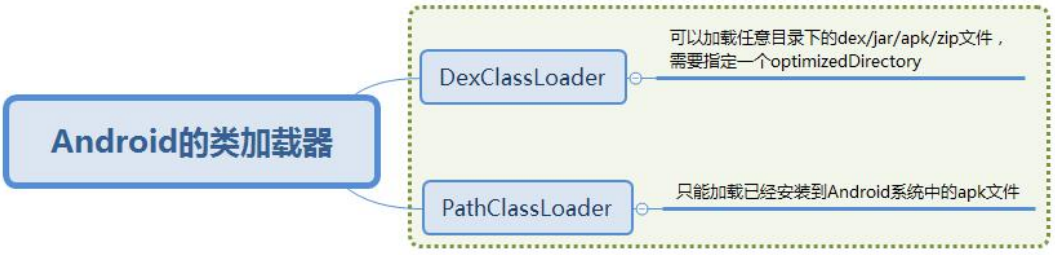
```
// PathClassLoader
public class PathClassLoader extends BaseDexClassLoader {
    public PathClassLoader(String dexPath, ClassLoader parent) {
```

```
        super(dexPath, null, null, parent);
    }

    public PathClassLoader(String dexPath, String librarySearchPath, ClassLoader
parent) {
        super(dexPath, null, librarySearchPath, parent);
    }
}


// DexClassLoaderpublic
class DexClassLoader extends BaseDexClassLoader {
    public DexClassLoader(String dexPath, String optimizedDirectory,
            String librarySearchPath, ClassLoader parent) {
        super(dexPath, new File(optimizedDirectory), librarySearchPath,
parent);
    }
}
```

通过比对，可以得出 2 个结论：

1）PathClassLoader 与 DexClassLoader 都继承于 BaseDexClassLoader。

2）PathClassLoader 与 DexClassLoader 在构造函数中都调用了父类的构造函数，但 DexClassLoader 多传了一个 optimizedDirectory。



2. BaseDexClassLoader

通过观察 PathClassLoader 与 DexClassLoader 的源码我们就可以确定，真正有意义的处理逻辑肯定在 BaseDexClassLoader 中，所以下面着重分析 BaseDexClassLoader 源码。

1）构造函数

先来看看 BaseDexClassLoader 的构造函数都做了什么：

```
public class BaseDexClassLoader extends ClassLoader {
    ...
    public BaseDexClassLoader(String dexPath, File optimizedDirectory, String
libraryPath, ClassLoader parent){
        super(parent);
        this.pathList = new DexPathList(this, dexPath, libraryPath,
optimizedDirectory);
    }
    ...
}
```

1）**dexPath**：要加载的程序文件（一般是 dex 文件，也可以是 jar/apk/zip 文件）所在目录。

2）**optimizedDirectory**：dex 文件的输出目录（因为在加载 jar/apk/zip 等压缩格式的程序文件时会解压出其中的 dex 文件，该目录就是专门用于存放这些被解压出来的 dex 文件的）。

3）**libraryPath**：加载程序文件时需要用到的库路径。

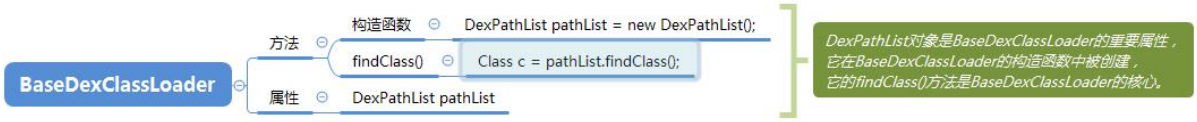
4）**parent**：父加载器

获取 class

类加载器肯定会提供有一个方法来供外界找到它所加载到的 class，该方法就是 findClass()，不过在 PathClassLoader 和 DexClassLoader 源码中都没有重写父类的 findClass()方法，但它们的父类 BaseDexClassLoader 就有重写 findClass()

BaseDexClassLoader 的 findClass() 方法实际上是通过

DexPathList 对象（pathList）的 findClass()方法来获取 class 的，而这个 DexPathList 对象恰好在之前的 BaseDexClassLoader 构造函数中就已经被创建好了。



3. DexPathList

DexPathList 的构造函数是将一个个的程序文件（可能是 dex、apk、jar、zip）封装成一个个 Element 对象，最后添加到 Element 集合中。

4. findClass()

结合DexPathList的构造函数,其实DexPathList的findClass()方法很简单，就只是对 Element 数组进行遍历，一旦找到类名与 name 相同的类时，就直接返回这个 class，找不到则返回 null。

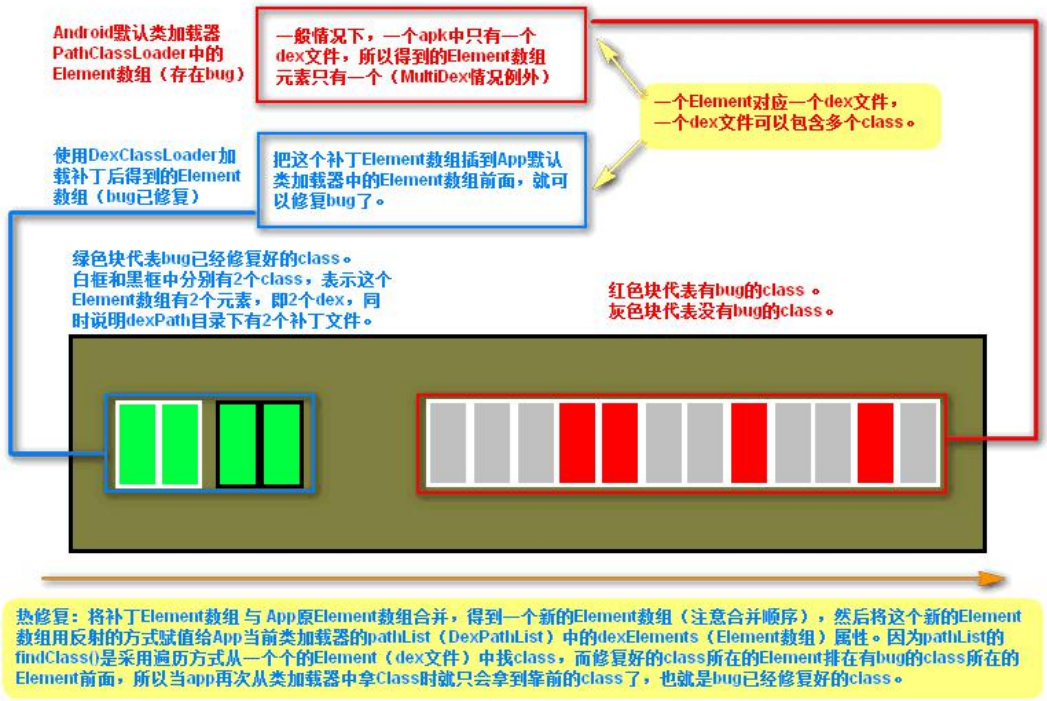
热修复的实现原理：

经过对 PathClassLoader、DexClassLoader、BaseDexClassLoader、DexPathList 的分析，我们知道，安卓的类加载器在加载一个类时会先从自身 DexPathList 对象中的 Element 数组中获取（Element[] dexElements）到对应的类，之后再加载。采用

的是数组遍历的方式，不过注意，遍历出来的是一个个的 dex 文件。

在 for 循环中，首先遍历出来的是 dex 文件，然后再是从 dex 文件中获取 class，所以，我们只要让修复好的 class 打包成一个 dex 文件，放于 Element 数组的第一个元素，这样就能保证获取到的 class 是最新修复好的 class 了（当然，有 bug 的 class 也是存在的，不过是放在了 Element 数组的最后一个元素中，所以没有机会被拿到而已）。

利用 PathClassLoader 和 DexClassLoader 去加载与 bug 类同名的类，替换掉 bug 类，进而达到修复 bug 的目的，原理是在 app 打包的时候阻止类打上 CLASS_ISPREVERIFIED 标志，然后在热修复的时候动态改变 BaseDexClassLoader 对象间接引用的 dexElements，替换掉旧的类。

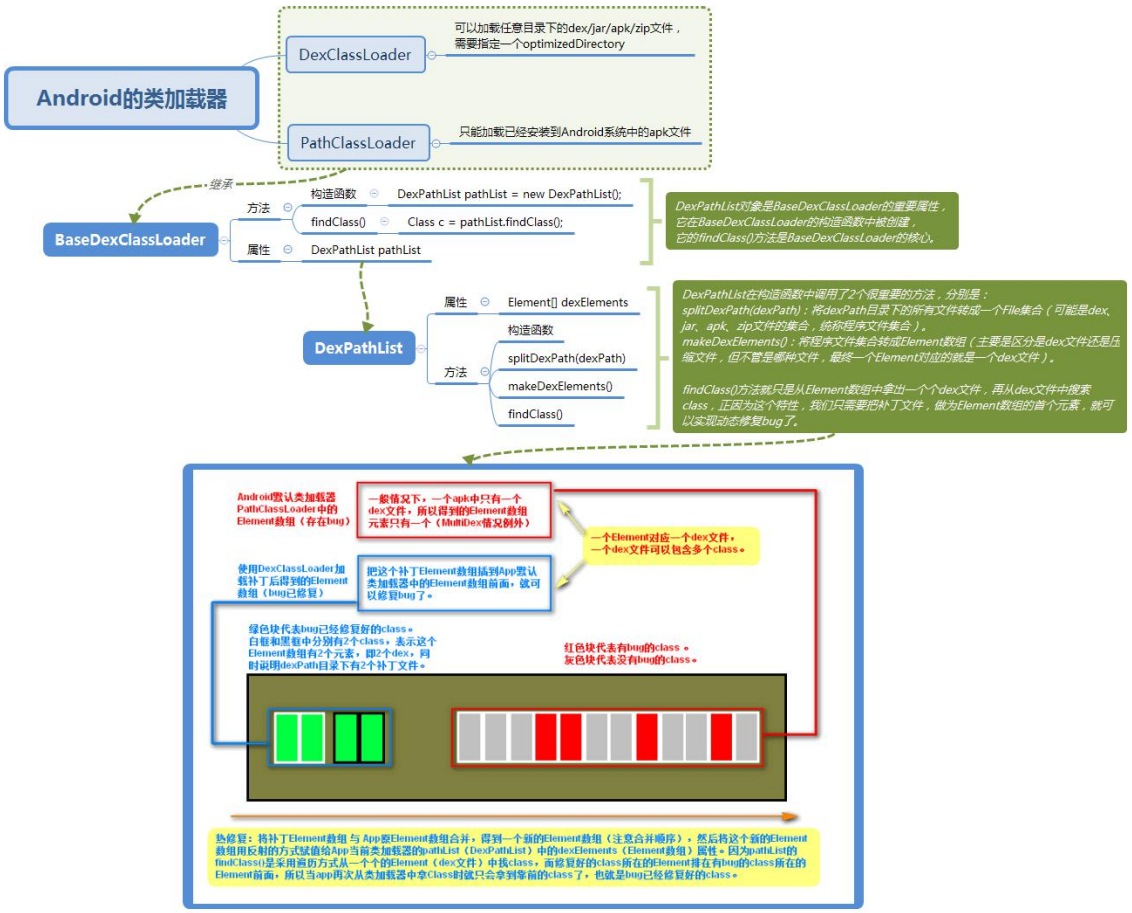


目前较火的热修复方案大致分为两派，分别是：

1) **阿里系：DeXposed、andfix：**从底层二进制入手（c 语言）。阿里 andFix hook 方法在 native 的具体字段。art 虚拟机上是一个叫 ArtMethod 的结构体。通过修改该结构体上有 bug 的字段来达到修复 bug 方法的目的,但这个 artMethod 是根据安卓原生的结构写死的，国内很多第三方厂家会改写 ArtMethod 结构，导致替换失效。

2) **腾讯系：tinker：**从 java 加载机制入手。qq 的 dex 插装就类似上面分析的那种。通过将修复的 dex 文件插入到 app 的 dexFileList 的前面，达到更新 bug 的效果，但是不能及时生效，需要重启。但虚拟机在安装期间会为类打上 CLASS_ISPREVERIFIED 标志，是为了提高性能的，我们强制防止类被打上标志是否会有些影响性能

3) **美团 robust** 是在编译器为每个方法插入了一段逻辑代码，并为每个类创建了一个 ChangeQuickRedirect 静态成员变量，当它不为空会转入新的代码逻辑达到修复 bug 的目的。有点是兼容性高,但是会增加应用体积



二、插件化

插件化一般就是提供一个 apk（插件）文件，然后在程序中 load 该 apk，那么如何加载 apk 中的类呢？其实就是通过这个 DexClassLoader。

PathClassLoader 和 DexClassLoader 都继承自 BaseDexClassLoader

1、Android 使用 PathClassLoader 作为其类加载器，只能去加载已经安装到 Android 系统中的 apk 文件；

2、DexClassLoader 可以从.jar 和.apk 类型的文件内部加载

classes.dex 文件就好了。热修复也用到这个类。

(1) 动态改变 BaseDexClassLoader 对象间接引用的 dexElements;

(2) 在 app 打包的时候，阻止相关类去打上 CLASS_ISPREVERIFIED 标志。

(3)

我们使用 hook 思想代理 startActivity 这个方法，使用占坑的方式，也就是说我们可以提前在 AndroidManifest 中固定写死一个 Activity，这个 Activity 只不过是一个傀儡，我们在启动我们插件 apk 的时候使用它去系统层校检合法性，然后等真正创建 Activity 的时候再通过 hook 思想拦截 Activity 的创建方法，提前将信息更换回来创建真正的插件 apk。

1. startActivity 的时候最终会走到 AMS 的 startActivity 方法

2. 系统会检查一堆的信息验证这个 Activity 是否合法。

3. 然后会回调 ActivityThread 的 Handler 里的 handleLaunchActivity

4. 在这里走到了 performLaunchActivity 方法去创建 Activity 并回调一系列生命周期的方法

5. 创建 Activity 的时候会创建一个 LoaderApk 对象，然后使用这个对象的 getClassLoader 来创建 Activity

6. 我们查看 getClassLoader() 方法发现返回的是

PathClassLoader，然后他继承自 BaseDexClassLoader

7. 然后我们查看 BaseDexClassLoader 发现他创建时创建了一个 DexPathList 类型的 pathList 对象，然后在 findClass 时调用了 pathList.findClass 的方法

8. 然后我们查看 DexPathList 类 中的 findClass 发现他内部维护了一个 Element[] dexElements 的 dex 数组，findClass 时是从数组中遍历查找的

共同原理：

都使用 ClassLoader 来实现的加载的新的功能类，都可以使用 PathClassLoader 与 DexClassLoader

不同的是：

热修复：热修复是体现在 bug 修复方面的，它实现的是不需要重新发版和重新安装，就可以去修复已知的 bug。热修复因为是为了修复 Bug 的，所以要将新的同名类替代同名的 Bug 类，要抢先加载新的类而不是 Bug 类，所以多做两件事：在原先的 app 打包的时候，阻止相关类去打上 CLASS_ISPREVERIFIED 标志，还有在热修复时动态改变 BaseDexClassLoader 对象间接引用的 dexElements，这样才能抢先代替 Bug 类，完成系统不加载旧的 Bug 类

插件化：插件化是体现在功能拆分方面的，它将某个功能独立提取出来，独立开发，独立测试，再插入到主应用中。以此来减少主应用的规模。插件化只是增加新的功能类或者是资源文件，所以不涉及

抢先加载旧的类这样的使命，就避过了阻止相关类去打上CLASS_ISPREVERIFIED 标志和还有在热修复时动态改变BaseDexClassLoader 对象间接引用的 dexElements

所以插件化比热修复简单，热修复是在插件化的基础上在进行替旧的 Bug 类

4. 模块化实现（好处，原因）
5. 项目组件化的理解

模块化是一种处理复杂系统分解为更好的可管理模块的方式。

为什么模块间解耦，复用？

原因：对业务进行模块化拆分后，为了使各业务模块间解耦，因此各个都是独立的模块，它们之间是没有依赖关系。每个模块负责的功能不同，业务逻辑不同，模块间业务解耦。模块功能比较单一，可在多个项目中使用。

为什么可单独编译某个模块，提升开发效率？

原因：每个模块实际上也是一个完整的项目，可以进行单独编译，调试

为什么可以多团队并行开发，测试？

原因：每个团队负责不同的模块，提升开发，测试效率

组件化与模块化

组件化是指以重用化为目的，将一个系统拆分为一个个单独的组件

1）避免重复造轮子，节省开发维护成本；

2）降低项目复杂性，提升开发效率；

3）多个团队公用同一个组件，在一定层度上确保了技术方案的统一性。

模块化业务分层：由下到上

1）基础组件层：底层使用的库和封装的一些工具库（libs），比如 okhttp,rxjava,rxandroid,glide 等

2）业务组件层：与业务相关，封装第三方 sdk，比如封装后的支付，即时通行等

3）业务模块层：按照业务划分模块，比如说 IM 模块，资讯模块等

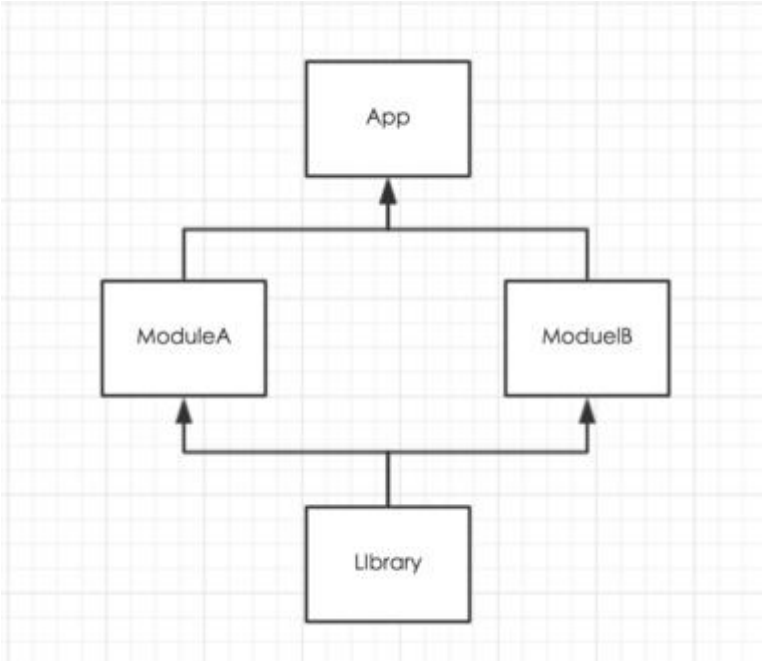
Android 模块化开发介绍；

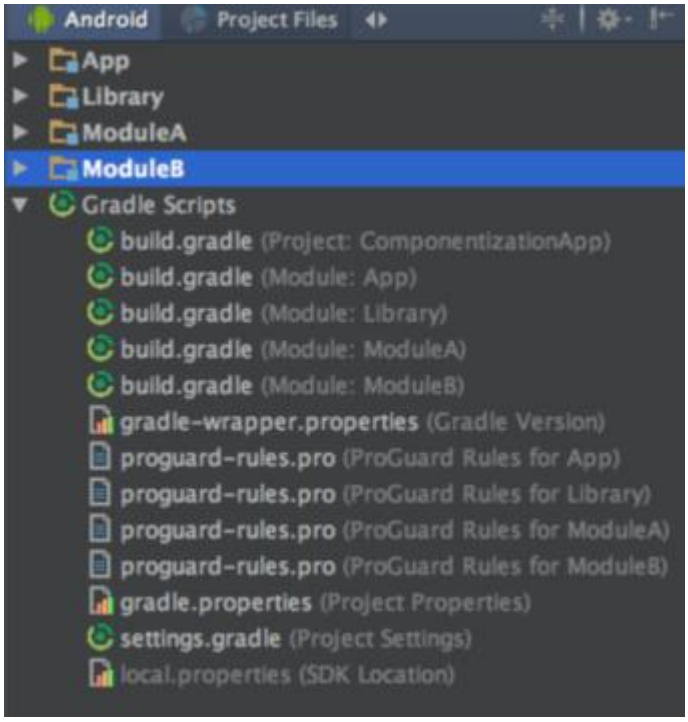
模块化开发思路就是：单独开发每个模块，用集成的方式把他们组合起来，就能拼出一个 app。app 可以理解成很多功能模块的组合，而且有些功能模块是通用的，必备的，像自动更新，反馈，推送，都可以提炼成模块，和搭积木很像，由一个壳包含很多个模块。

Android 模块化开发的好处；

1) 模块升级会单独升级，升级的时候往往不只是增加资源，有时候会去删资源,这样做和其它的模块的资源不掺乎。比如:我用umeng的自动更新 sdk 时就需要我连着资源一块进来，加进来容易以后不想用 umeng 的了再挑出去就很费事了。
2) 再假如 push 开始用的百度的后来换成极光的，对 app 的组合者其实是不关心的，对他影响很小。解耦很好。
3) 如果是主力带实习生这种开发的话，核心代码就不用和他们分享，让他们去做独立的功能，做好直接调用就行。以后整理代码，重构什么的都只重构这个模块的代码，他们不会不小心改了核心代码。

模块化开发的架构分层





命名建议：

底层：Library

中间层：Module + 业务或功能名字

上层：App + 项目名字

建议分层进行

1. **底层：**包含基础库和底层库

1）基础库：包含所有模块需要的依赖库，以及一些工具类，比如封装了的常用网络请求，封装图片处理 fresco，数据库相关等，还包含所有模块需要的依赖库；

2）底层库：主要是使用 C/C++开发的跨平台的引擎或者库，以 so 的形式存在。例如：游戏引擎 cocos2d

2. **中间层**

先分模块肯定要按照功能分，独立的一个功能，不能杂。比如、更新、登录、分享、播放，都可以。

其次、我采取 aar 的形式作为模块的最小单位，为什么是 aar 不是 jar，更不是 library，因为 jar 不能带资源只能带 java 代码，library 的话太容易被修改了，aar 的好处是能带资源并且是编译好的，不能被修改。保证了模块的版本不会在被别人调用的时候随意修改，如果想修改就要联系做 aar 的人，让他去升级 aar 的版本。

用 android studio，打 aar 用 maven；aar 其实就是依赖，只不过之前的依赖都是存到了 maven 远程库里，自己用的话可以自己建和私有的 maven 库，太蛮烦的话可以直接用本地的 aar 文件做依赖。

3. 上层

将所有的业务模块聚合在一起，加上配置，形成主应用，一个模块化做的好的应用，主应用应该很简单，并且非常的稳定。

6. 描述清点击 Android Studio 的 build 按钮后发生了什么

gradle 插件

我们创建一个 Android Application，至少要包含一个 application module，并且在 build.gradle 中设置 application 的 gradle 插件：

apply plugin: 'com.android.application'

这样，编译这种 module 时将才成.apk 文件。

一个 application module 不可以依赖另一个 application module，只可以依赖 library，就是配置 gradle library 插件的 module：

apply plugin: 'com.android.library'

发布这样的 module 时，得到的将是一个.aar 文件，与.jar 文件相比，aar 文件可以包含一些 android 相关的东西：比如资源文件和 manifest 文件。

编译：

编译一个 application module 或者 library 文件，大致可以分为 gradle task 代表的五个阶段：

1) **准备依赖包（Preparation of dependecies）：**在这个阶段 gradle 检测 module 依赖的所有 library 是否 ready。如果这个 module 依赖于另一个 module，则另一个 module 也要被编译。

2) **合并资源并处理清单（Merging resources and processing Manifest）：**在这个阶段之后，资源和 Manifest 文件被打包。

3) **编译（Compiling）：**在这个阶段处理编译器的注解，源码被编译成字节码。

4) **后期处理（Postprocessing）：** 所有带 “transform”前缀的 task 都是这个阶段进行处理的。

5) **包装和出版（Packaging and publishing）：** 这个阶段 library 生成.aar 文件，application 生成.apk 文件。

一个粗糙的构建流程

gradle 构建 APK 的流程大致如下图：

1. Android 编译器（5.0 之前是 Dalvik，之后是 ART）将项目的源代码（包括一些第三方库、jar 包和 aar 包）转换成 DEX 文件，将其他资源转换成已编译资源。

2. APK 打包器将 DEX 文件和已编译资源在使用秘钥签署后打包。

3. 在生成最终 APK 之前，打包器会使用 zipalign 等工具对应用进行优化，减少其在设备上运行时的内存占用。

构建流程结束后获得测试或发布用的 apk。

一个稍详细的构建流程

更详细的 gradle 构建过程如下图：

图中的矩形表示用到或者生成的文件，椭圆表示工具。

1. 通过 aapt 打包 res 资源文件，生成 R.java、resources.arsc 和 res 文件

2. 处理.aidl 文件，生成对应的 Java 接口文件

3. 通过 Java Compiler 编译 R.java、Java 接口文件、Java 源文件，生成.class 文件

4. 通过 dex 命令，将.class 文件和第三方库中的.class 文件处理生成 classes.dex

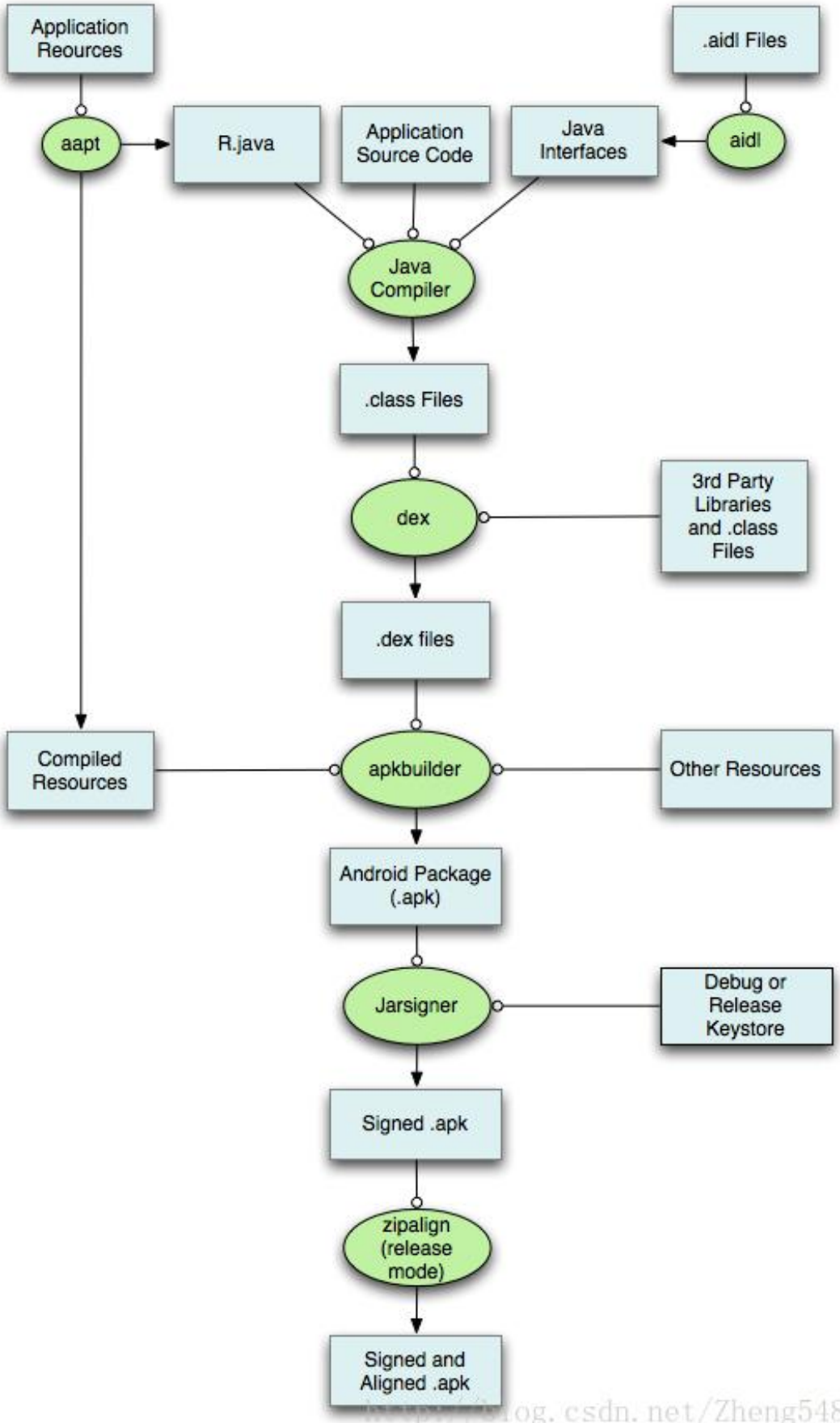
5. 通过 apkbuilder 工具，将 aapt 生成的 resources.arsc 和 res 文件、assets 文件和 classes.dex 一起打包生成 apk

6. 通过 Jarsigner 工具，对上面的 apk 进行 debug 或 release 签名

7. 通过 zipalign 工具，将签名后的 apk 进行对齐处理。

这样就得到了一个可以安装运行的 Android 程序。

这样的流程仍然不是详细的，下面的这幅图带你看下什么叫详细。



Android 使用 gradle 构建生成的 apk 关键就是 aapt 处理资源文件，aidl 处理.aidl，javac 生成.class 文件，proguard 混淆后再由 dex 生成.dex 文件，由 apkbuilder 签名后再经 zipalign 对齐字节码就可以上线发布了。