一. 为什么要用热修复

在软件开发时，比如版本1.0的app上线了，然后用户安装，这个时候出现了bug。怎么办呢？肯定是修改出现bug的代码。于是修复完后，重新发布版本1.1上线，用户再进行安装进行使用，就解决了。但是这会出现一个问题，每次出现bug用户都要重新进行安装，用户的体验很不好。就像打游戏一样，每次游戏更新都要重新下载一次游戏然后再进行安装，肯定不合理。所以我们就可以用热修复来解决这个问题。w

二. 什么是热修复

在我们的应用上线出现bug需要修复时，不再进行发布新的安装包让用户下载安装，而是**下发补丁，即用户不用重新安装app，就可以修复缺陷的一种技术。**

> 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）。
> 2. 腾讯系：tinker：从java加载机制入手。

这里要注意一下：从java机制入手的话需要重启app才可以生效，而底层二进制入手（c语言）可以马上生效

三. 怎么实现热修复

1.基本原理

通俗地说，有一个数组，这个数组里存放的是dex文件(里面存放的是类的二进制数据，用来给安卓虚拟机加载)。在加载类文件时，从这个数组下标为0的地方开始找，如果找到了对应的文件，即使后面也有该类的.dex文件，也会停止查找。也就是说，**前面的会覆盖后面的。**

安卓的虚拟机是Dalvik，Java的虚拟机是jvm，他们加载的文件也不一样。

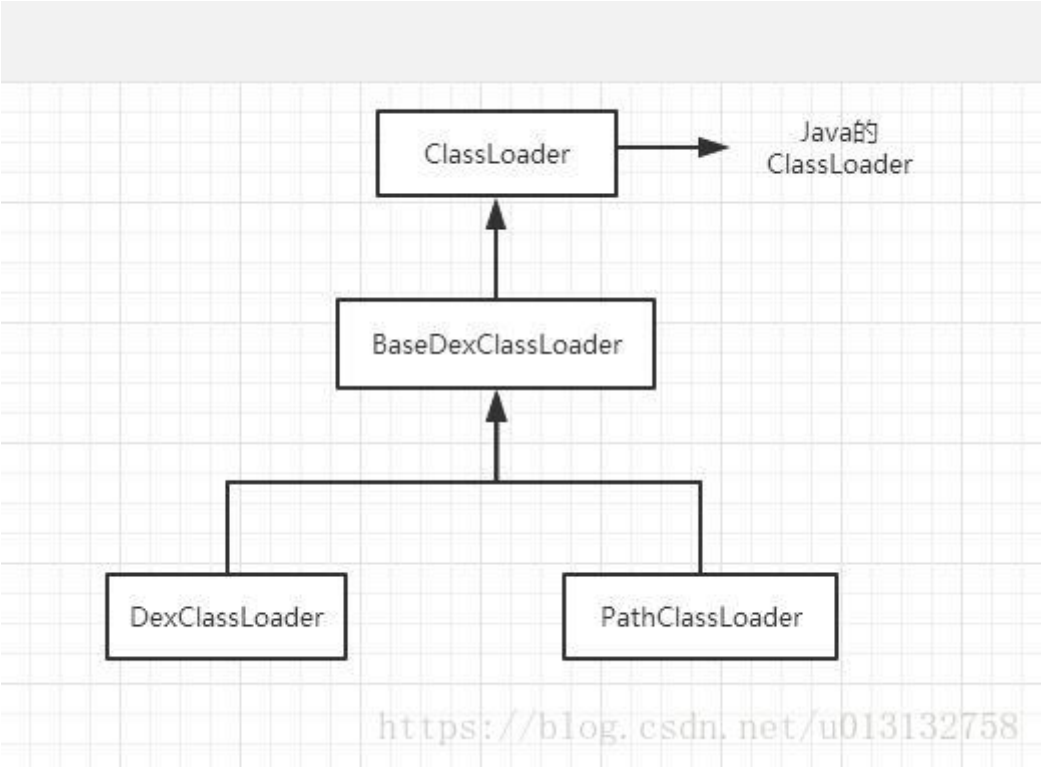
> 1、jvm是吧.java文本编译成.class字节码文件，在执行java程序的时候，类加载器把需要的类全部加载到内存当中去。davik虚拟机把.java文件编译成.class文件，又把.class文件转换成.dex文件，dalvik来执行.dex文件。实际上.dex文件就是把多个class文件中的常量、方法等放到一起。
> 2、在架构上jvm是基于栈的架构，所以每次访问数据cpu都要到内存中取到数据。而dalvik是基于寄存器的架构。寄存器是在cpu上的一块存储空间，cpu如果直接从寄存器上读取数据的话就会快很多。

即jvm加载的是class文件，而安卓的虚拟机Dalvik是加载dex文件。

class文件： class文件是一种能够被JVM识别，加载并且执行的文件格式。

dex文件：它是Android系统的可执行文件，而class文件是jvm加载的。当java程序编译成class后，还需要使用dex工具将所有的class文件整合到一个dex文件。可以理解成1个dex文件里存放了很多class文件

2.Android的类加载机制



> Java程序在运行的时候,JVM通过类加载机制(ClassLoader)把class文件加载到内存中,只有class文件被载入内存,才能被其他class引用,使程序正确运行起来.

而安卓有更独特的类加载器，Android的类加载器有两种：`PathClassLoader`和`DexClassLoader`,他们都继承`BaseDexClassLoader`，`BaseDexClassLoader`继承自`ClassLoader`

`PathClassLoader`：**安卓默认的类加载器**，加载系统类和应用类，只能加载dex文件。

`DexClassLoader`：可以加载**任意位置**的jar，apk，dex文件。它相当于是解压文件，所以需要一个文件夹进行输出。可以用它加载我们的补丁。其构造函数需要2个文件，一个文件是我们要加载的dex,jar,apk文件，另外一个是解压文件后输出到哪里。

3.先了解一下PathClassLoader和DexClassLoader

`PathClassLoader`

```
public class PathClassLoader extends BaseDexClassLoader {
    public PathClassLoader(String dexPath, ClassLoader parent) {
    super(dexPath, null, null, parent);
}

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

`DexClassLoader`

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

可以看到，它们都只是调用了父类的构造函数。所以我们来看看BaseDexClassLoader：

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

@Override
protected Class<?> findClass(String name) throws ClassNotFoundException {
    List<Throwable> suppressedExceptions = new ArrayList<Throwable>();
    Class c = pathList.findClass(name, suppressedExceptions);
    if (c == null) {
        ClassNotFoundException cnfe = new ClassNotFoundException("Didn't find
class \"" + name + "\" on path: " + pathList);
        for (Throwable t : suppressedExceptions) {
            cnfe.addSuppressed(t);
        }
        throw cnfe;
    }
    return c;
}
```

在findclass方法中，是通过调用`pathList`的findclass方法来查找的。

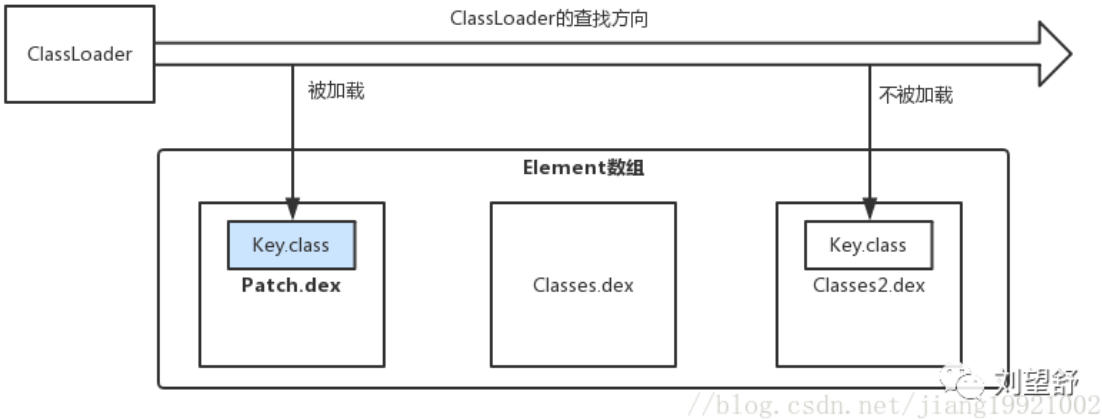
这里我们着重看一下`pathList`属性，在`BaseDexClassLoader`构造函数中创建了一个`DexPathList`类的实例,`DexPathList`的构造函数会创建一个`dexElements`数组。而这个数组就是关键，所以我们去看看。

`DexPathList`：

```
final class DexPathList {
private Element[] dexElements;
    ...
public Class findClass(String name, List<Throwable> suppressed) {
        //遍历该数组
    for (Element element : dexElements) {
            //初始化
            DexFile dex = element.dexFile;
            if (dex != null) {
                //调用DexFile类的loadClassBinaryName方法返回Class实例
                Class clazz = dex.loadClassBinaryName(name, definingContext,
suppressed);
                if (clazz != null) {
```

```
                    return clazz;
                }
            }
        }
        return null;
    ...
    }
```

上面说的数组就是`dexElements`这个数组。在findclass方法中通过遍历这个数组初始化`DexFile`，如果不为空则调用`DexFile`类的`loadClassBinaryName`返回Class实例。



所以，归纳上面的话就是:`ClassLoader`会遍历这个数组,然后加载这个数组中的dex文件.
而`ClassLoader`在加载到正确的类之后,就不会再去加载有Bug的那个类了,我们把这个正确的类放在Dex文件中,让这个Dex文件排在`dexElements`这个数组的前面即可.

也就是：**1.通过`DexClassLoader`加载补丁，然后通过反射得到对应`dexElements`。**

2.拿到安卓默认的类加载器`PathClassLoader`，然后通过反射得到对应的`dexElemens`。

3.将第一步得到的数组和第二步得到的数组合并，并且第一步的数组中元素排在第二部数组元素前面(补丁元素放在前面)，并且重新赋值给`PathClassLoader`。

4.代码实现

首先定义一个热修复类

```
public class HotFix {
private final String TAG = "HotFix";
private final String FIELD_DEX_ELEMENTS = "dexElements";
private final String FIELD_PATH_LIST = "pathList";
private final String CLASS_NAME = "dalvik.system.BaseClassLoader";
private final String DEX_SUFFIX = ".dex";
private final String JAR_SUFFIX = ".jar";
private final String APK_SUFFIX = ".apk";
private final String SOURCE_DIR = "patch";
private final String OPTIMIZED_DIR = "odex";
}
```

然后是具体的方法

```
public void startFix(Context context) throws IllegalAccessException,
NoSuchFieldException, ClassNotFoundException {
        // 默认补丁目录
/storage/emulated/0/Android/data/rocketly.hotfixdemo/files/patch
        File sourceFile = context.getExternalFilesDir(SOURCE_DIR);
        if (!sourceFile.exists()) {
            Log.i(TAG, "补丁目录不存在");
            return;
        }
        // 默认 dex优化存放目录  /data/data/rocketly.hotfixdemo/odex
        File optFile = context.getDir(OPTIMIZE_DIR, Context.MODE_PRIVATE);
        if (!optFile.exists()) {
            optFile.mkdir();
        }
        StringBuilder sb = new StringBuilder();
        File[] listFiles = sourceFile.listFiles();
        for (int i = 0; i < listFiles.length; i++) {//遍历查找文件中patch开头, .dex
.jar .apk结尾的文件
            File file = listFiles[i];
            if (file.getName().startsWith("patch") &&
file.getName().endsWith(DEX_SUFFIX)//这里我默认的补丁文件名是patch
                    || file.getName().endsWith(JAR_SUFFIX)
                    || file.getName().endsWith(APK_SUFFIX)) {
                if (i != 0) {
                    sb.append(File.pathSeparator);//多个dex路径 添加默认分隔符 :
                }
                sb.append(file.getAbsolutePath());
            }
        }
        String dexPath = sb.toString();
        String optPath = optFile.getAbsolutePath();
  PathClassLoader pathClassLoader = contexxt.getClassLoader();//拿到系统默认的类加
载器PathClassLoader
    DexClassLoader dexClassLoader = new DexClassLoader(dexPath, optPath, null,
context.getClassLoader());//加载我们自己补丁的类加载器DexPathClassLoader
    Object pathElements = getElements(pathClassLoader);//获取PathClassLoader
Element[]
    Object dexElements = getElements(dexClassLoader);//获取DexClassLoader
Element[]
    Object combineArray = combineArray(pathElements, dexElements);//合并数组
    setDexElements(pathClassLoader, combineArray);//将合并后Element[]数组设置回
PathClassLoader pathList变量
}
```

getElements：获得classLoader所对应的 `dexElements` 这个数组

```
 private Object getElements(ClassLoader classLoader) throws
ClassNotFoundException, NoSuchFieldException, IllegalAccessException {
        Class<?> BaseDexClassLoaderClazz = Class.forName(CLASS_NAME);//拿到
BaseDexClassLoader Class
       Field pathListField =
BaseDexClassLoaderClazz.getDeclaredField(FIELD_PATH_LIST);//拿到pathList字段
        pathListField.setAccessible(true);
        Object DexPathList = pathListField.get(classLoader);//拿到DexPathList对象
       Field dexElementsField =
DexPathList.getClass().getDeclaredField(FIELD_DEX_ELEMENTS);//拿到dexElements字段
        dexElementsField.setAccessible(true);
        return dexElementsField.get(DexPathList);//返回Element[]数组
    }
```

combineArray：把我们补丁对应的数组元素（`DexClassLoader`的`dexElements`)放到系统默认的数组(`PathListClassLoader`的`dexElements`）前面。

```
private Object combineArray(Object pathElements, Object dexElements) {
        Class<?> componentType = pathElements.getClass().getComponentType();
        int i = Array.getLength(pathElements);
        int j = Array.getLength(dexElements);
        int k = i + j;
        Object result = Array.newInstance(componentType, k);// 创建一个类型为
componentType，长度为k的新数组
        System.arraycopy(dexElements, 0, result, 0, j);
        System.arraycopy(pathElements, 0, result, j, i);
        return result;
    }
```

setElements：把合并后的数组设置给系统默认类加载器`PathClassLoader`。

```
  private void setDexElements(ClassLoader classLoader, Object value) throws
ClassNotFoundException, NoSuchFieldException, IllegalAccessException {
        Class<?> BaseDexClassLoaderClazz = Class.forName(CLASS_NAME);
        Field pathListField =
BaseDexClassLoaderClazz.getDeclaredField(FIELD_PATH_LIST);
        pathListField.setAccessible(true);
        Object dexPathList = pathListField.get(classLoader);

        Field dexElementsField =
dexPathList.getClass().getDeclaredField(FIELD_DEX_ELEMENTS);
        dexElementsField.setAccessible(true);
        dexElementsField.set(dexPathList, value);
    }
```

主要就是通过反射获取字段然后合并数组，最后设置回去。

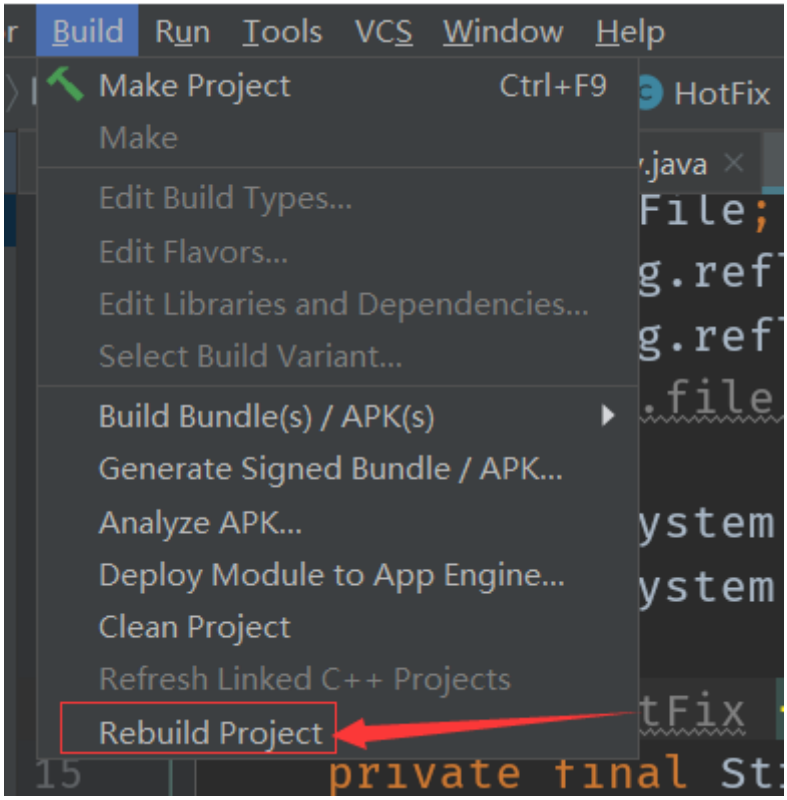
这里好像有个问题,可参考QQ空间团队的 安卓App热补丁动态修复技术介绍

另外，因为有文件的读写，所以别忘了权限声明。

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

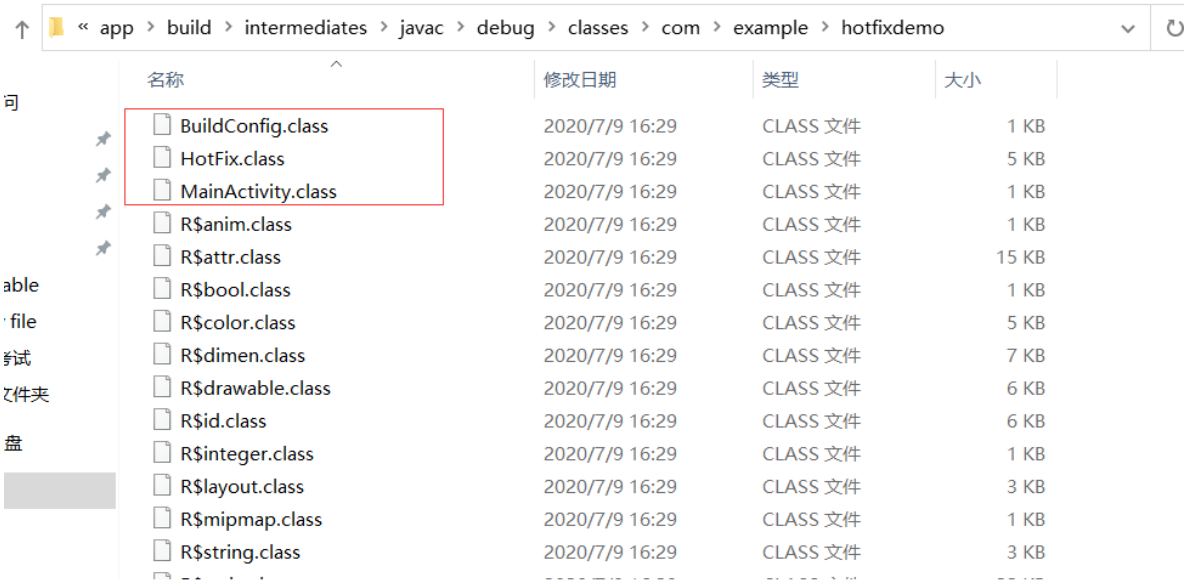
五.怎样打热修复补丁包

1.点击AS的Rebuild Project



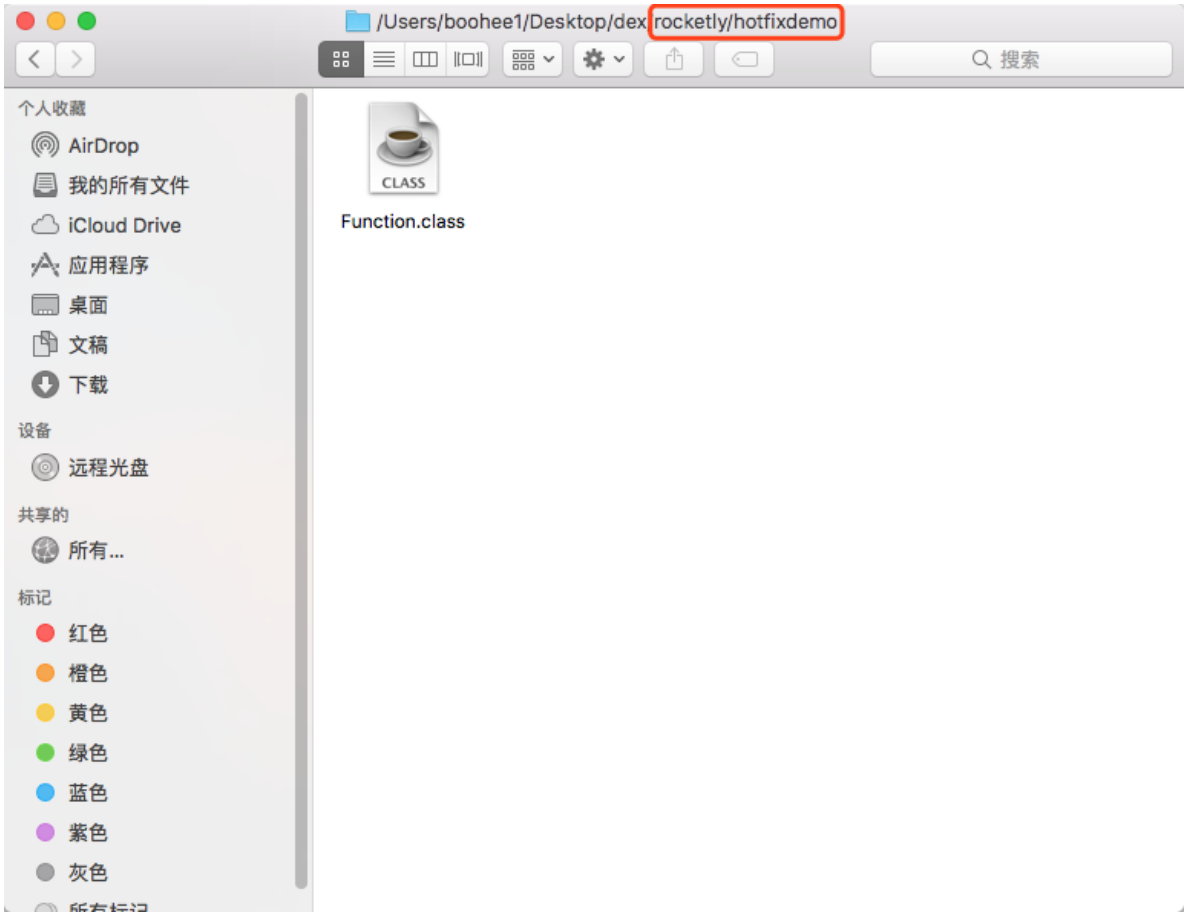
2.找到class文件

然后就可以在app的build/intermediates/javac/debug/classes/包名 下找到我们的class文件



3.接下来生成dex文件。

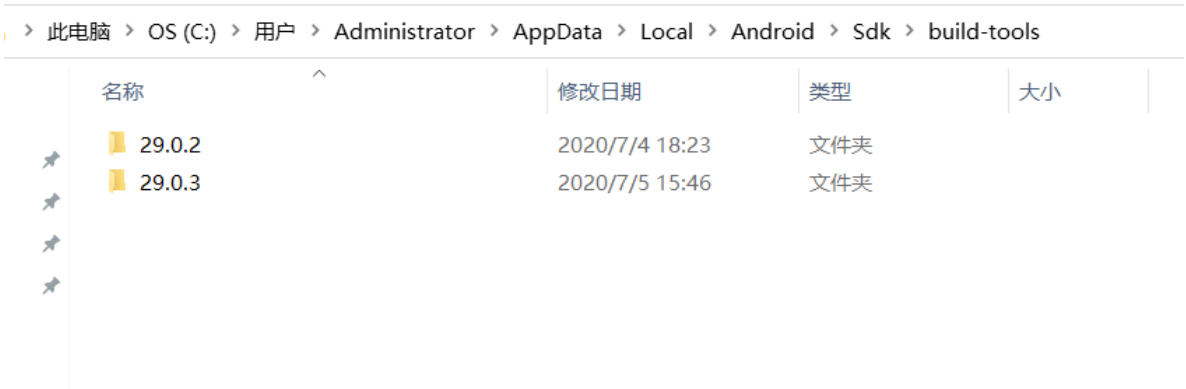
首先把修复好的bug类class文件连同包目录一起复制到指定目录，比如桌面下。



然后就可以在命令行通过dx指令生成dex文件。有2种用法：

1. 配置环境变量（添加到classpath），然后命令行窗口（终端）可以在任意位置使用。
2. 不配环境变量，直接在build-tools/安卓版本 目录下使用命令行窗口（终端）使用。

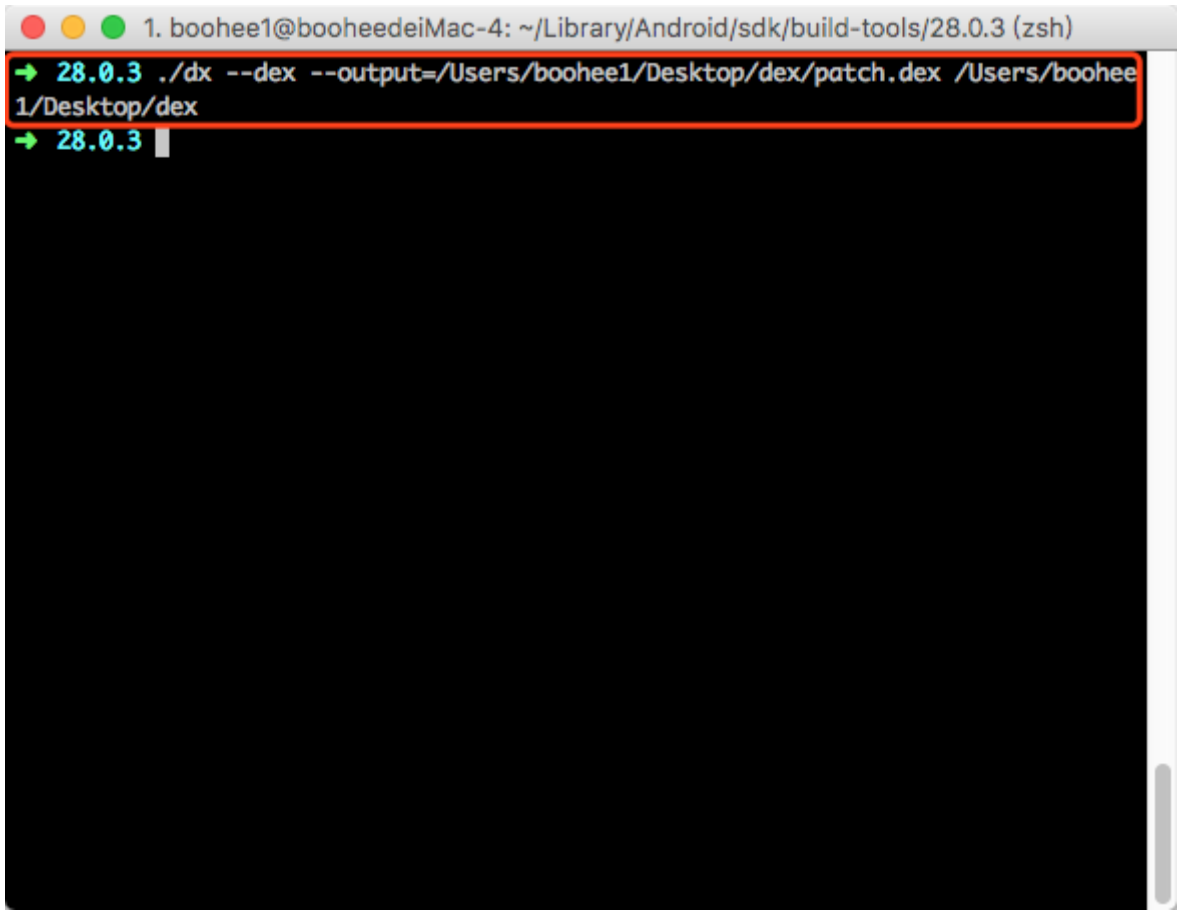
具体位置：



我电脑打开这个会闪退 所以演示不了..

打开后输入dx指令生成dex文件：

dx --dex --output=输出的dex文件完整路径 (空格) 要打包的完整class文件所在目录



六.目前国内热修复的框架

> Android热修复目前各大厂商都有自己的热修复工具，主要分为两大主流以阿里为代表的Native层替换方法表中的方法实现热修复［AndFix ,Sophix等］,和以腾讯美团为代表的在JAVA层实现热修复[Tinker，Robust等]。后者要实现热修复必须要重启APP，而前者则不需要重启APP，直接在虚拟机的方法区实现方法替换。

类别	成员
阿里系	AndFix，Dexposed，阿里百川，Sophix
腾讯系	微信的Tinker，QQ空间的超级补丁，手机QQ的Qfix
知名公司	美团的Robust，饿了么的Amigo，美丽说蘑菇街的Aceso
其他	RocooFix，Nuwa，AnoleFix

部分热修复框架特性的对比：

特性	AndFix	Tinker/Amigo	QQ空间	Robust/Aceso
即使生效	√	×	×	√
方法替换	√	√	√	√
类替换	×	√	×	×
类结构修改	×	√	√	×
资源替换	×	√	×	×
so替换	×	√	×	×
支持gradle	×	√	×	×
支持ART	√	√	√	√
支持Android7.0	√	√	√	√

写在最后

因为我也是刚刚接触这个概念，肯定有很多地方不足或者很多地方讲错了，还请各位大佬多多包涵

给大家推荐一些博客

热修复——深入浅出原理与实现

Android学习——手把手教你实现Android热修复