ColorOS_PatchRom 教程

更改记录			
版本	更改日期	更改内容	修订人
V0.1	2014-06-20	创建	季正康

目　录

第一章搭建开发环境

1. 操作系统

patchrom 的操作主要在 linux 下进行，推荐 ubuntu12 以上的版本。

2. 安装 jdk

首先需要安装Java开发工具包，本文中统一约定$表示Terminal中的命令提示符(没有root权限)，其后的文字表示输入的命令。

从以下地址http://www.oracle.com/technetwork/java/javase/downloads/index.html下载Java开发工具包.我们推荐下载Java SE 6 Update38版本。

我们对下载下来的文件进行安装：

```
$ sudo chmod 755 jdk-6u38-linux-x64.bin
$ sudo -s ./jdk-6u38-linux-x64.bin /opt
```

接下来编辑home目录下的.bashrc文件，配置我们所需要的PATH环境变量：

```
$ vim ~/.bashrc
```

在文件最后添加：

```
# set java environment
JAVA_HOME=/opt/jdk1.6.0_38
export JRE_HOME=${JAVA_HOME}/jre
export CLASSPATH=.:${JAVA_HOME}/lib:${JRE_HOME}/lib
export PATH=${JAVA_HOME}/bin:$PATH
```

运行命令来使我们修改的PATH环境变量生效。

```
$ . ~/.bashrc
```

最后我们检查我们的JDK是否安装成功，输入：

```
$ java -version
```

出现如下提示，说明安装成功，如不成功，请参照以上步骤再次尝试。

```
java version "1.6.0_38"
Java(TM) SE Runtime Environment (build 1.6.0_38-b05)
Java HotSpot(TM) 64-Bit Server VM (build 20.13-b02, mixed mode)
```

3. adb

Android SDK 中对我们最重要的工具是 adb(android debug bridge)以及 aapt。在适配的过程中，最常用的命令是 adb logcat，该命令会打印出详细的调试信息，帮助我们定位错误。

为了验证 adb 是否工作，同时也是验证上述步骤是否成功，打开手机中的系统设置－－开发人

员选项，确保选中“USB 调试”，然后用 USB 线连接你的手机，在 Ubuntu Shell 下运行命令 adb devices，如果显示的信息和下面类似，那么恭喜你，adb 能识别你的手机了。

```
jizhengkang@ubuntu-15:~/ColorOS/3c$ adb devices
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
List of devices attached
MBO7UO5PCYVSLF75        device
```

注意：在 Ubuntu 下，有可能会提示“no such permissions”，这个时候有两个办法，第一种是以 root 的身份运行 adb。第二种方法：

a） 运行 lsusb 命令，对于我的手机，输出如下：

```
jizhengkang@ubuntu-15:~/ColorOS/3c$ lsusb
Bus 001 Device 002: ID 80ee:0021 VirtualBox USB Tablet
Bus 001 Device 003: ID 12d1:1052 Huawei Technologies Co., Ltd.
Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
```

找到手机对应的那一行，记录下 12d1:1052，这个分别表示该设备的 vendorld 和 productID。如果不确定手机对应的是哪一行，可以在连上手机前后运行 lsusb，找到区别的那一行。

b） 在/etc/udev/rules.d 目录下新建一个文件 99-android.rules。编辑如下：

SUBSYSTEM=="usb",ATTRS{idVendor}=="12d1",ATTRS{idProduct}=="1052",MODE="0666"

c） 重启 usb 服务，sudo restart udev, 重连手机。

```
jizhengkang@ubuntu-15:~/ColorOS/3c$ sudo restart udev
[sudo] password for jizhengkang:
udev start/running, process 16939
```

4. 同步源码

说说怎么同步 ColorOS 的 patchrom，首先你得下载个 repo，并且设置一些权限使其可以执行：

```
curl http://git-repo.googlecode.com/files/repo-1.13 > ~/bin/repo
chmod 777 ~/bin/repo
```

为了方便使用，得在环境变量里面加上 repo 的路径，以后就可以直接使用 repo，而不需要输入完整的路径了。

```
gedit ~/.bashrc      (注意，这个是图形界面的，如果使用命令行的可以用 vi)
```

在文件的最后面加上下面这句：

```
export PATH=${PATH}:~/bin
```

OK 准备工作完成了，下面开始同步啦。

```
mkdir ColorOS
cd ColorOS
repo init -u git://github.com/ColorOS/manifest.git -b ColorOS_JB2_2.0     非 mtk 的用这个
repo init -u git://github.com/ColorOS/manifest.git -b ColorOS_MTK_JB2_2.0     mtk 的用这个
repo sync
```

以上都是 4.2 的分支，4.4 的环境我们正在紧张的调试中，敬请期待!!!

5. patchrom 项目

下面介绍 patchrom 的目录结构以及各目录的作用。

build 目录是一些编译脚本以及 color 的一些需要替换的文件；
device 目录是 patchrom 的主要工作目录，放着一些编译的脚本和 makefile；
manifest 目录方便 repo 下载的目录，里面还有帮助文档；
smali 目录存放 google 原生和 color 的 smali 文件；
tools 目录存放 patchrom 中需要使用的一些工具。

第二章寻找合适的底包

1. 合适的底包

一般基于 patchrom 适配有三种底包，aosp、cm 以及原厂底包，各有优劣，一般来说原厂底包是适配最麻烦的，但也是最稳定的。如果开发者是第一次接触 patchrom，可以尝试使用可以正常使用的 aosp 版本或者 cm 版本，熟练一点的还是推荐使用原厂底包，稳定性可以得到保证。

2. 刷机方法

为了持续的升级，一般的厂商都会提供卡刷的方法。但是正规的厂商都会做卡刷包的签名验证，第三方的卡刷包基本上都是刷不了的。这时候需要去一些论坛社区学习别人的刷机方式和找到适合自己机器的 recovery。首推 http://forum.xda-developers.com/，国外一个非常不错的论坛，国内的 ColorOS 论坛也很不错。

3. boot 的修改

修改 boot 主要是为了两个目的 1.增加 adb 的 root 权限 2.增加 oppo-framework.jar 的加载。为了这两个目的需要修改 bootimg 里面的 default.prop 和 init.rc。

一般解开 bootimg 可以采用如下的方法。一般来说，一个机型的完整刷机包里面都有一个 boot.img 的文件，绝大部分的 bootimg 都遵循 google 给定的默认格式。

假定我们在 patchrom 目录下，给定了一个 boot.img，运行如下命令：

```
$ ./tools/bootimgtools/split_bootimg.pl boot.img
```

输出类似如下文字：

```
Page size: 2048 (0x00000800)
Kernel size: 4114304 (0x003ec780)
Ramdisk size: 747137 (0x000b6681)
Second size: 0 (0x00000000)
Board name:
Command line: console=ttyDCC0 androidboot.hardware=xxx
Writing boot.img-kernel ... complete.
Writing boot.img-ramdisk.gz ... complete.
```

如果这些输出有乱码，那么可以判定该 boot.img 不遵从标准格式。记下这些参数，接下来还要用到。同时在 patchrom 目录下会看到一个 boot.img-ramdisk.gz 文件，该文件即是根文件系统的压缩包。还有一个 boot.img-kernel 文件，该文件即是 kernel 文件。

```
$ mkdir ramdisk
$ cd ramdisk
$ gzip -dc ../boot.img-ramdisk.gz | cpio -i
```

运行这三个命令后，ramdisk 目录即为手机启动后的根文件系统目录，用任何编辑器修改 init.rc 和 default.prop。

init.rc 中修改如下字符串：

export BOOTCLASSPATH /system/framework/core.jar:/system/framework/core-junit.jar:/system/framework/bouncycastle.jar:/system/framework/ext.jar:/system/framework/framework.jar:/system/framework/telephony-common.jar:/system/framework/mms-common.jar:/system/framework/android.policy.jar:/system/framework/services.jar:/system/framework/apache-xml.jar:/system/framework/mediatek-common.jar:/system/framework/mediatek-framework.jar:/system/framework/secondary-framework.jar:/system/framework/CustomProperties.jar:/system/framework/mediatek-telephony-common.jar:/system/framework/mediatek-op.jar:/system/framework/hwframework.jar

在该行后面添加:/system/framework/oppo-framework.jar(注意，有英文字符的冒号)

default.prop 中修改如下几个参数，结果如下：

```
ro.secure=0
ro.adb.secure=0
persist.sys.usb.config=mass_storage,adb
```

```
jizhengkang@ubuntu-15:~/ColorOS/3c$ cd ..
jizhengkang@ubuntu-15:~/ColorOS/3c$ ./tools/mkbootfs ./ramdisk | gzip > ramdisk-new.gz
jizhengkang@ubuntu-15:~/ColorOS/3c$./tools/mkbootimg --cmdline 'console=ttyDCC0 androidboot.hardware=xxx'--kernel boot.img-kernel --ramdisk ramdisk-new.gz --base 0x00200000 --pagesize 4096 -o boot-new.img
```

将 ramdisk 目录重新打包。

运行该命令生成新的 boot.img，

--cmdline：该选项为之前打印的 Command line

--kernel：该选项为 Linux 内核文件

--ramdisk：该选项为根文件系统压缩包

--base：一般为 0x00200000

--pagesize：该选项为之前打印的 Page size

-o：该选项为输出文件名

以上方法可以适用于绝大部分的底包，但有些机器的做法比较另类，这时就需要去一些专业的论坛寻求帮助，看看有什么办法可以解开 bootimg，在此不作详述。

4. deodex

当我们获取到了一个合适的rom之后，我们首先要判断下该rom是否是odex优化过的。对于deodex，你只需要理解odex是一种优化过的dex文件就行了，至于怎么优化的，这个不在我们讲述的范围内。odex文件是相互依赖的，简单地理解就 是我们改了其中一个文件，其它的odex文件就不起

作用了。为此，我们必须做一个deodex操作，就是将odex文件变成dex文件，让这些文件可以独立修改。

一般来说原厂ROM发布时都是以odex文件格式发布的，如何判断呢？运行如下命令：

```
jizhengkang@ubuntu-15:~/ColorOS/3c$ adb shell ls /system/framework
```

如果看到很多以odex结尾的文件，那么该ROM就是做过odex的，大家可以用patchrom/tools目录下的deodex.sh脚本来自动化的做deodex操作。

```
jizhengkang@ubuntu-15:~/ColorOS/3c$ deodex.sh Honor_3C_T10_B606.zip
temp dir: /work/work3/id_jizhengkang/ColorOS/3c/tempdir.5mX
unzip    to /work/work3/id_jizhengkang/ColorOS/3c/tempdir.5mX
processing                                                framework/core.jar
/work/work3/id_jizhengkang/ColorOS/device/smali_out/framework/core.odex_out
processing                                                framework/am.jar
/work/work3/id_jizhengkang/ColorOS/device/smali_out/framework/am.odex_out
processing                                                framework/android.policy.jar
/work/work3/id_jizhengkang/ColorOS/device/smali_out/framework/android.policy.odex_out
...
...
...
processing                                                app/UserDictionaryProvider.apk
/work/work3/id_jizhengkang/ColorOS/device/smali_out/app/UserDictionaryProvider.odex_out
processing                                                app/VisualizationWallpapers.apk
/work/work3/id_jizhengkang/ColorOS/device/smali_out/app/VisualizationWallpapers.odex_out
processing                                                app/VoiceUnlock.apk
/work/work3/id_jizhengkang/ColorOS/device/smali_out/app/VoiceUnlock.odex_out
processing                                                app/VpnDialogs.apk
/work/work3/id_jizhengkang/ColorOS/device/smali_out/app/VpnDialogs.odex_out
processing                                                app/YGPS.apk
/work/work3/id_jizhengkang/ColorOS/device/smali_out/app/YGPS.odex_out
zip deodexed update package!!!
/work/work3/id_jizhengkang/ColorOS/3c/tempdir.5mX
deodex Honor_3C_T10_B606.zip ok
output package is update.deodexed.zip
```

转化之后的 dex 版本的包在 device 目录下，最好能试刷一下这个版本，看能否正常使用，防止白白浪费时间插桩。

5. 环境变量

为了方便进行多个机型的适配与维护，以及后续的升级，不建议直接在 device 目录下面进行操作。你可以使用如下的方法，完整的复制 device 目录一份重命名，以荣耀 3c 为例(以下均以 3c 为例)，重命名为 3c，进入 3c 目录，执行. setenv.sh(注意有空格)

```
jizhengkang@ubuntu-15:~$ cd ColorOS
jizhengkang@ubuntu-15:~/ColorOS$ ls
build  device  manifest  smali  tools
jizhengkang@ubuntu-15:~/ColorOS$ cp -rf device/ 3c
jizhengkang@ubuntu-15:~/ColorOS$ ls
3c  build  device  manifest  smali  tools
jizhengkang@ubuntu-15:~/ColorOS$ cd 3c
jizhengkang@ubuntu-15:~/ColorOS/3c$ . setenv.sh
PORT_ROOT          = /work/work3/id_jizhengkang/ColorOS
PORT_BUILD         = /work/work3/id_jizhengkang/ColorOS/build
PORT_TOOLS         = /work/work3/id_jizhengkang/ColorOS/tools
PORT_DEVICE        = /work/work3/id_jizhengkang/ColorOS/3c
ANDROID_TOP        =
ANDROID_OUT        =
/work/work3/id_jizhengkang/ColorOS/3c
jizhengkang@ubuntu-15:~/ColorOS/3c$
```

如上，设置了所有的环境变量，把你准备好的底包重命名为 update.zip 放到 3c 目录下，到底，环境都准备好了。

第三章反编译以及移植 framework

1. 自动插桩

在这一步，我们需要将需要插桩的几个文件进行反编译，生成 smali 文件以方便插桩。全部手动插桩是个复杂的工作，我们提供了自动化的工具，不过既然是工具，肯定有不完善的地方，需要人工进行修复，也就是一般所说的修改 reject。反编译与自动插桩简化由一步完成

```
jizhengkang@ubuntu-15:~/ColorOS/3c$ ls
apps apps.conf ForWrite.txt important.txt makefile setenv.sh update update.zip
util.mk  将可用的升级包命名为 update.zip 放在该目录下
jizhengkang@ubuntu-15:~/ColorOS/3c$ make firstpatch
cleansmali
rm -rf `pwd`/tmp_system
rm -rf smali
mkdir -p `pwd`/tmp_system
mkdir -p smali

baksmali single jar file ---->android.policy.jar.out
…
…
…
…
…
…
/work/work3/id_jizhengkang/ColorOS/3c/.orig



>>> patch color into target framework is done. Please look at
/work/work3/id_jizhengkang/ColorOS/3c/temp/reject to resolve any conflicts!
mkdir -p /work/work3/id_jizhengkang/ColorOS/3c/custom-update
Please look at /work/work3/id_jizhengkang/ColorOS/3c/temp/reject to resolve any conflicts!
```

3c/temp/reject 目录下有所有需要解决的 reject，对比 3c/smali 跟 3c/temp/dst_smali_orig，3c/smali 是自动插桩之后的文件，3c/temp/dst_smali_orig 是反编译之后去掉了一些调试信息的文件。根据 3c/temp/reject 里面的 reject 来修改 3c/smali。

简单介绍下 temp 下面各个文件夹

```
jizhengkang@ubuntu-15:~/ColorOS/3c$ ls -l temp/
total 32
drwxrwxr-x 7 jizhengkang jizhengkang 4096 Jun 19 20:50 android_smali
drwxrwxr-x 8 jizhengkang jizhengkang 4096 Jun 19 20:50 android_smali.nochange
drwxrwxr-x 7 jizhengkang jizhengkang 4096 Jun 19 20:50 color_smali
drwxrwxr-x 8 jizhengkang jizhengkang 4096 Jun 19 20:50 color_smali.nochange
drwxrwxr-x 7 jizhengkang jizhengkang 4096 Jun 19 20:51 dst_smali_orig
drwxrwxr-x 8 jizhengkang jizhengkang 4096 Jun 19 20:50 dst_smali_orig.nochange
drwxrwxr-x 7 jizhengkang jizhengkang 4096 Jun 19 20:52 dst_smali_patched
drwxrwxr-x 7 jizhengkang jizhengkang 4096 Jun 19 20:52 reject
```

在自动插桩过程中，为了加快速度，做了一些小小的优化，把一份完整的 smali 分成了两部分，比如 android_smali 和 android_smali.nochange，后缀带有 nochange 的表示是 color 没有修改的部分，这部分在自动 patch 的时候不予理会，可以省略自动 patch 过程中比较这部分文件的时间，速度加快很多。

android_smali：通过 google 源码或者 MTK 源码编译出来的 framework 文件反编译出来的 smali 文件，去除了没有修改的部分
android_smali.nochange：通过 google 源码或者 MTK 源码编译出来的 framework 文件反编译出来的 smali 文件中没有修改的部分
color_smali：通过 color 源码编译出来的 framework 文件反编译出来的 smali 文件，去除了没有修改的部分
color_smali.nochange：通过 color 源码编译出来的 framework 文件反编译出来的 smali 文件中没有修改的部分
dst_smali_orig：通过底包中的的 framework 文件反编译出来的 smali 文件，去除了没有修改的部分
dst_smali_orig.nochange：通过底包中的 framework 文件反编译出来的 smali 文件中没有修改的部分
dst_smali_patched：自动 patch 生成的中间文件，一般可以无视
reject：自动 patch 过程中出现的冲突，在生成卡刷包之前，必须要解决掉所有的冲突

2. 修复 reject

这一节我们以 3c 为例，重点介绍如何修改原厂 ROM 的 smali，将 color 的修改应用到上面去。我们不会将所有的修改都会在文中列出来，挑选几个有代表性的讲解，剩下的大家可以自己去做。
smali 代码注入只能应对函数级别的移植，对于类级别的移植是无能为力的。具体地说，如果你想修改一个类的继承、包含关系、接口结构等，都是非常困难的。修改类成员的变量访问控制权限，类方法实现，对于这些操作，smali 代码注入的方法是可以实现的。这主要是因为 smali 级代码的灵活性已经远低于 java 源代码，而且经过编译优化后，更注重程序的执行效率。
本质上讲，smali 代码注入就是在已有 apk 或者 jar 包中插入一些 dalvik 虚拟机的指令，从而改变原来程序执行的路径或者行为。
这个过程大致分为五步——确定需要注入的 smali 代码，确定注入位置，注入 smali 代码，编译 smali 代码，调试 smali 代码。
总体流程如下图：

4.1 比较差异

这里的比较差异包含两个部分：比较 color 和原生 android 的差异，比较 3c 和原生 android 的差异。以 framework.jar 为例，首先在 smali，3c 这两个目录中，我们在 smali 下需要 smali/color/framework.jar.our 和 smali/android/framework.jar.our 目录，然后我们反编译 3c 底包里面的 framework.jar 文件，生成待插桩的 framework.jar.out 目录在 3c/smali/下。

rmline.sh 是用来把 smali 中所有的以.line 开头的行去掉，便于我们比较 smali 代码上的差别。但是对 smali 代码进行修改，我们还是在没有去掉.line 的版本上修改，.line 对于我们调试代码很有帮助，在程序运行出错抛出异常的时候，adb logat 的异常错误信息会给出其出错的代码的行号，这样方便我们定位错误。

接下来用文件比较工具来比较差异，Linux 和 Windows 下，我们都推荐使用 Beyond Compare。在比较 color 和 android，3c 和 android 的区别时，我们不比较那些相同的和新加的文件，只比较修改过的文件。（注：生成的 reject 都是修改过 java 产生的差异，没有修改过的 java 反编译之后有时也会产生差异，不过自动插桩的时候都自动排除了，及 temp/**.nochange 文件夹下面的文件，无需关注）。下面我们就开始修改 smali 文件了，我将这些修改分成 3 种情况，选择有代表性的 4 个文件加以介绍，这 4 种情况难度依次增加。

4.2 直接替换

以 ActivityThread.smali 为例，比较发现 color 改了其中一个方法 getTopLevelResources，而 3c 和原生 android 的实现完全一样，这种情形是最简单也是最 happy 的，我们改的地方要适配的机型原厂 ROM 完全没有修改，直接替换就可以了。

4.3 线性代码

以如下代码段为例，color 一共改了两个方法，一个是上面介绍的，另一个是 applyConfigurationToResourcesLocked。通过比较得知，color 修改了这个方法，3c 也修改了这个方法。怎么办呢，我们先分析一下 color 修改的代码：

```
.method final applyConfigurationToResourcesLocked(Landroid/content/res/Configuration;)Z
。。。

invoke-virtual {p0, v0}, Landroid/app/ActivityThread;->freeTextLayoutCachesIfNeeded(I)V
if-eqz v0,
:cond_7
:goto_2
move v6, v5
goto :goto_0
:cond_7
move v5, v6
goto :goto_2.
.end    method
```

在上面将 3c 增加的代码用红色标出，在讲述之前，先解释一下 smali 代码的一些规律：所有的局部变量用 v 开头，方法的顶部.locals x 表示这个方法是用 x 个局部变量。所有的参数用 p 开头，局部

变量和参数都是从 0 开始编号。对于非静态方法来说，p0 就是对象本身的引用，即 this 指针。
这里 color 新增了一个方法调用，对于这种顺序执行的一段代码，我们称之为线性代码。这个例子比较的简单，只新增了一个方法调用。线性代码的特点是只有一个入口和一个出口，在编译器的术语中这叫做基本块。对于这种新增的代码，我们找出他的上下文，即修改的代码前后的操作。然后在 3c 的该方法的 smali 代码中找到相应的位置，把这个修改应用到 3c 中去。这种修改也相对简单，插入代码的相应位置比较好定位。

4.4 条件判断

这种情况指的是 color 插入的代码并不是一个线性代码，而是有条件判断的。color 加的判断语句一般会用 cond_oppo_x 来表示，很明显就能看出是 color 加的。我们举一个例子：

```
.localp1,cache:Landroid/util/LongSparseArray;,"Landroid/util/LongSparseArray<Ljava/lang/ref/WeakReference<Landroid/graphics/drawable/Drawable$ConstantState;>;>;"
invoke-static{p1,p2},Landroid/content/res/Resources$Injector;->needNewResources(Landroid/util/LongSparseArray;I)
    move-result v4
    if-eqz v4, :cond_oppo_1
    :cond_oppo_0
    return-void
    :cond_oppo_1
invoke-virtual {p1}, Landroid/util/LongSparseArray;->size()I
。。。
if-ge v2, v0, :cond_oppo_0
```

从上面的例子中可以看到，这个是 color 插入一段条件语句。如果只有 color 修改插入了 if 语句直接插入即可，但是如果出现原厂也修改了，会出现很多 cond 数字不同的情况，这种情况就需要我们分析手动插入。下面就来讲解一下如何应对这种情况。

1、首先要确定这段 if 语句插入的位置，如何确定，通过上下文来分析，例如上面这个例子，我们看到 color 插入这段代码的上文是声明了一个局部变量 p1，那我们就要去找到 3c 原厂代码中这一句。然后插入代码段。

2、接着我们要看一下满足什么情况才应该跳转到 cond_oppo_0，跳转 cond_oppo_0 这句在未修改的源码中表现为 if-ge v2, v0, :cond_1，但是在 3c 的代码中未必是表示为这样的语句，我们需要联系上下文去找到在 3c 中这句话是在哪里然后将其改为跳转到 cond_oppo_0。

在这里我向大家讲述一个非常有用的知识：同一个方法内 cond_x 是不可能单独出现的，例如有个 cond_1，同一个方法内必定会有另一个 cond_1 否则编译会报错，或者这段代码是无效的代码。:cond_1 一处是声明跳转后所做的事，另一处则是表示满足何种条件跳转。在代码中看到的 goto_x 也是同样的道理。

4.5 逻辑推理

这种情况一般和内部类相关，你会发现在源码中的改动很小，但是在反编译后的 smali 代码改动确很大。

对于 java 文件中的每一个内部类，都产生一个单独的 smali 文件，比如 AcitvityThread$1.smali，这些文件的命名规范是如果是匿名类，外部类+$+数字。否则的话是外部类+$+内部类的名字。

当在内部类中调用外部类的私有方法时，编译器会自动合成一个静态函数。比如下面这个类：

```
public class Hello{
```

```
public class A{
viod func(){
setup();
}
}
private void setup(){
}
}
```

我们在内部类 A 的 func 方法中调用了外部类的 setup 方法，最终编译的 smali 代码为：
Hello$A.smali 文件代码片段：

```
# virtual methods
.method func()V
.locals 1
.prologue
.line 5
iget-object v0,p0,LHello$A;->this$0:LHello;
#calls:LHello;->setup()V
invoke-static{v0},LHello;->access$000(LHello;)V
.line6
return-void
.end method
```

Hello.smali 代码片段：

```
.method static synthetic access$000(LHello;)V
.locals 0
.parameter
.prologue
.line 1
invoke-direct{p0},LHello;->setup()V
return-void
.end method
```

可以看到，编译器自动合成一个 access$000 方法，假如当我们在一个较复杂的内部类中加入一个对外部类私有方法的调用，虽然只是导致新合成了一个方法，但是这些合成的方法名可能都会有变化，这样的结果就是 smali 文件的差异性比较大。

5.smali 代码注入

5.1 确定需要注入的 smali 代码

首先我们要确定需要注入的 smali 代码，比对的方法和工具，我们在上面已经说过了，这里就不多加叙述了。例如下面的红色区域就是需要注入的 smali 代码，

5.2 确定注入的位置

这一步的看似简单，实际工作中有很多难点，主要是有些注入位置比较难确定，需要不断的尝试。使用 APKTOOL 反汇编待注入的 APK 或 JAR 包后，首先需要确认需要注入的 smali 文件是哪个。这个主要是针对含有匿名内部类的 Java 文件而言。例如，移植 PhoneWindowManager.java 文件的修改时，反汇编之后会有很多 PhoneWindowManager$1.smali,PhoneWindowManager$2.smali...类似的文件。这些文件就是匿名内部类的 smali 代码，由于没有名字，所以编译后只能用$XXX 来区分。

如果带注入的 smali 代码是从 PhoneWindowManager$5.smali 提取的，一般不能够直接将其注入到目标机型的 PhoneWindowManager$5.smali 文件中，因为不同机型的匿名内部类顺序不同，实现不同，smali 文件也不同。一般需要通过逐个比较 PhoneWindowManager$5.smali 附近的几个文件的 smali 代码，看看其函数调用，函数名字，类继承关系是否相同来确定注入哪个文件。

当然对于没有匿名内部类的 Java 文件可以直接使用对应的 smali 文件注入即可。

其次，确定了注入文件之后，就需要进一步确认待注入区域。由于 smali 代码中的每个变量的类型是不固定的，再加上编译器的优化，导致不同 ROM 的 APK 或 JAR 包反汇编后，会有很多不同。这个并不影响我们的工作，我们重点关注 smali 代码的“行为”——函数调用顺序，逻辑判断顺序，类成员变量访问顺序，即可大致确定注入区域。另外，对于新增的 smali 代码区域可以随意些，新增变量直接追加在变量声明尾部即可，新增函数直接增加在文件尾部。总的来说这个工作还是非常经验化的，需要长时间的反复尝试才能更准确的确定注入区域。

最后，继续上面例子，如图：

图中有很多红色的不同，其中蓝框是我们刚才确定的需要注入的 samli 代码。通过上下文匹配，可以发现绿框的位置是 samli 代码需要注入的区域。尽管上下有很多指令和变量不同，但是这并不影响我们的工作。

5.3 注入代码

首先，将待注入的 smali 代码注入对应的区域。其次，对注入的 Smali 代码进行“本地化”——修改变量、跳转标号、逻辑判断标号等，使之符合当前的 Smali 代码实现，完成“嫁接”工作。当然，如果情况很复杂，需要重写对应的 Smali 代码或者重构 java 源代码，来完成最终的代码注入。
需要说明的是 color 自己的资源是 0xc 开头，位于 oppo-framework.apk 中。

5.4 编译 smali 代码

Smali 编译过程相对简单，使用 apktool b XXX XXX.apk 即可将 Smali 代码编译成 apk 或 jar 包。但是当遇到编译错误时，apktool 工具给出的错误信息少之又少，以至于我们只能手动查找哪个文件 Samli 代码移植错误。

这里，我们总结了一些 smali 代码移植时可能遇到的编译错误。

函数调用(invoke-virtual 等指令)的参数只能使用 v0~v15，使用超过 v15 的变量会报错。修复这个

问题有两种方法：
A.使用 invoke-virtual/range {p1 .. p1}指令，但是这里要求变量名称需要连续。B.增加 move-object/from16 v0, v18 类似指令，调整变量名，使之小于等于 v15。
B.增加 move-object/from16 v0, v18 类似指令，调整变量名，使之小于等于 v15。

函数调用中 p0 相当于函数可用变量值+1，pN 相当于函数可用变量值+N。例如函数.local 值为 16，表明函数可用变量值为 v0~v15，则 p0 相当于 v16，p1 相当于 v17。
例如，下图左侧蓝框所在代码编译不过，后来检查了代码所在的函数.local 为 33，p0 相当于 v33，所以编译不过，修改为右侧绿框才正常。

跳转标号重叠。这里主要是指出现了两个相同的标号的情况，例如 cond_11 等，导致无法编译过。解决方法就是修改冲突的标号以及相关跳转语句。其实这个标号叫什么都无所谓，你甚至可以叫 ABCD_XXX,只要可以与对应的 goto 语句呼应即可。

使用没有定义的变量。每个函数可以使多少变量都在函数体内的第一句.local 中声明，例如.local30 表明这个函数可以使用 v0~v29，如果使用 v30 就会编译错误。

有时有些 smali 实在太难插桩，可以尝试整个函数甚至整个文件的替换，不过这种情况需要特别谨慎，因为非常容易出错，替换之后即便可以开机也需要仔细测试，这种方法非常不推荐，只限于实在没有办法的时候可以尝试。

第四章制作刷机包

制作刷机包其实是个简单的工作

1.apps.conf

修改修复完所有的 reject 并验证可以编译通过，根据需要配置 apps.conf 文件，apps.conf 文件类似如下：

```
APPS_NEED_RESIGN := \
                    **.apk

APPS_NOT_RESIGN := \
                    **.apk

APPS_EXTRA      := \
                    **.apk

APPS_MTK_ONLY     := \
                    Phone.apk

APPS_QCOM_ONLY   := \
                    Phone.apk

APPS_KEEP_ORIGIN := \
                    **.apk
```

APPS_KEEP_ORIGIN 是底包中需要保留的 apk 应用，比如蓝牙、相机、FusedLocation 等。注意在 APPS_KEEP_ORIGIN 中添加某个应用之后，如果其他的各项中包括同名的，需要删除那一行。

2.makefile

目前需要修改的主要是两个地方

1) 作者和渠道

```
AUTHOR_NAME := jizhengkang
FROM_CHANNEL := oppo
```

这个定义需要修改，一般的，只需要修改 AUTHOR_NAME，这个后面修改为自己的名字，

推荐采用论坛注册名字。此定义用于统计开发者发布的 rom 的活跃量，千万记得修改!!!不然统计的数据全部变成我的了!!!

2) 需要比较的文件

```
COLOR_FRAMEWORK_JARS := android.policy framework telephony-common services pm secondary-framework
```

如果有其他的文件也加进来，用空格隔开，文件名不加.jar 后缀。

3.custom-update

一些需要修改或者保留的文件，按照 update 的目录层次，放在 custom-update 下面，比如修改之后的 boot.img。

4.编译

执行 make fullota，系统先会自动回编译上面 COLOR_FRAMEWORK_JARS 指定的所有 jar 文件，拷贝并且给 apk 签名。最后在当前目录下面生成的 color-update.zip 就是生成的卡刷包。

第五章调试 smali

虽然在上面解决 reject 的时候已经可以编译通过，并且生成了卡刷包，但第一次往往都是不能正常开机的，这时候就显示出最开始修改 bootimg 的作用了，修改 bootimg 的一个作用就是打开 adb，现在就该 adb 大显身手了。

连接手机，shell 窗口输入 adb logcat > adblog，一般打印几分钟的 log，查看 adblog 文件，先搜索 VFY，优先解决所有的 VFY 错误，然后在搜索解决 exception 错误。

一般来说，解决一个错误就重新编译一次，根据修改的 smali 文件所在的文件夹使用 make **.jar 的命令，生成的文件在 3c/out/framework/目录下，重挂载 system 为可读，把生成的文件 push 进去，重启手机，重新开始抓取 adblog，重复以上调试过程，直到手机可以正常开机。

```
jizhengkang@ubuntu-15:~/ColorOS/3c$ make framework.jar
smali single jar file ---->framework.jar
mkdir -p out/framework
/work/work3/id_jizhengkang/ColorOS/tools/apktool b smali/framework.jar.out out/framework/framework.jar
I: Checking whether sources has changed...
I: Smaling...
W: Could not find resources
I: Building apk file...
jizhengkang@ubuntu-15:~/ColorOS/3c$ adb devices
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
List of devices attached
MBO7UO5PCYVSLF75 device

jizhengkang@ubuntu-15:~/ColorOS/3c$ adb remount
remount succeeded
jizhengkang@ubuntu-15:~/ColorOS/3c$ adb push out/framework/framework.jar system/framework/
110 KB/s (8445 bytes in 0.074s)
jizhengkang@ubuntu-15:~/ColorOS/3c$ adb reboot
```

一般来说，smali 的问题有如下几种：

a) 函数变量列表与声明不同，这个主要体现在下面两个方面：A.函数调用的变量类型与函数声明不同。通过追踪变量在上下文的赋值动作来解决。B.函数变量列表中变量少于或者多于函数声明的变量。通过核对函数声明来解决。

b) 函数调用方式不正确。例如：public和包访问函数使用invoke-virtual调用，private函数使用invoke-direct调用，接口函数使用invoke-interface调用。如果使用错误，会导致运行时错误。需要调整相关的Smali代码。

c) 类接口没有实现。主要是由于增加了新的子类没有实现原有父类接口导致的，只需增加空实现即可修复。

第六章一些命令

make firstpatch
make fullota 生成可供使用的刷机包
make clean

make resource 编译生成 framework-res.apk 和 oppo-framework.apk
make **.jar 编译 smali 目录下的**.jar.out，生成文件是 out/framework/**.jar
make **.jar.out 反编译 update/system/**.jar,反编译出来的结果是 smali 目录下的**.jar.out

make update 升级
make count 计算显示 reject 数量