

# MIUI\_V4 定制教程

|                            |    |
|----------------------------|----|
| 第一章 搭建开发环境.....            | 3  |
| 1.操作系统.....                | 3  |
| 2.安装 Android SDK.....      | 3  |
| 2.1 安装 JDK.....            | 3  |
| 2.2 下载 Android SDK 包.....  | 4  |
| 2.3 安装.....                | 4  |
| 2.4 adb.....               | 5  |
| 3.同步 MIUI 代码.....          | 6  |
| 4.patchrom 项目.....         | 6  |
| 第二章 认识 Android 手机.....     | 8  |
| 1.bootloader.....          | 8  |
| 2.正常启动.....                | 9  |
| 3.System 分区.....           | 10 |
| 4.data 和 cache 分区.....     | 12 |
| 5.小结.....                  | 12 |
| 第三章 寻找合适的原厂 ROM.....       | 13 |
| 1.熟悉适配的机型.....             | 13 |
| 1.1 逛论坛刷机.....             | 13 |
| 1.2 合适的原厂 ROM.....         | 13 |
| 1.3 adb logcat.....        | 14 |
| 2.修改 boot.img.....         | 15 |
| 3.deodex.....              | 16 |
| 4.Makefile.....            | 16 |
| 5.workspace.....           | 17 |
| 6.firstpatch.....          | 17 |
| 7.fullota.....             | 18 |
| 第四章 反编译.....               | 19 |
| 1.反编译.....                 | 19 |
| 2.AndroidManifest.xml..... | 19 |
| 3.资源.....                  | 20 |
| 4.smali.....               | 22 |
| 第五章 适配 MIUI Framework..... | 25 |
| 1.为什么使用代码插桩.....           | 25 |
| 2.适配规范.....                | 25 |
| 2.1 android, miui.....     | 25 |
| 2.2 i9100.....             | 25 |
| 3.移植资源.....                | 26 |
| 4.修改 smali.....            | 26 |
| 4.1 比较差异.....              | 27 |
| 4.2 直接替换.....              | 28 |

|                            |    |
|----------------------------|----|
| 4.3 线性代码.....              | 28 |
| 4.4 条件判断.....              | 29 |
| 4.5 逻辑推理.....              | 30 |
| 5.smali 代码注入.....          | 31 |
| 5.1 确定需要注入的 smali 代码.....  | 31 |
| 5.3 注入代码.....              | 33 |
| 5.4 编译 smali 代码.....       | 34 |
| 5.5 调试 smali 代码.....       | 35 |
| 5.6 调试 smali 问题以及追踪方法..... | 35 |
| 6.建议.....                  | 36 |
| 中兴 U950 的适配过程.....         | 37 |

# 第一章 搭建开发环境

“工欲善其事，必先利其器”。在开始定制 MIUI ROM 之前，我们需要搭建好必要的开发环境。

本教程的主旨是如何基于原厂 ROM 修改。我们所涉及的修改理论上说是不需要源码的，对源码开发感兴趣的可以参照 <http://source.android.com>。对于 ROM 开发者来说，我们建议下载一份 google 发布的 android 源码，这不是必须的，但是对于理解排查 ROM 适配中的一些错误有很大的帮助。

## 1.操作系统

定制 MIUI ROM 所涉及的技术本身对操作系统没有特殊要求，Windows，Linux 和 Mac 系统都可以。但是 patchrom 项目是基于 Linux 开发的，确切的说，是基于 Ubuntu 开发的，我们推荐使用 Ubuntu10 以上的系统。目前，我们还没有计划开发运行在 Windows 和 Mac 系统上面的 patchrom 项目。

## 2.安装 Android SDK

本节只简要介绍如何在 Ubuntu 系统上安装 Android SDK。

### 2.1 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，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
```

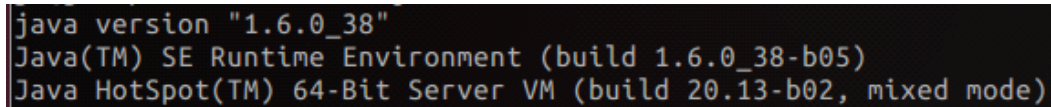
运行命令来使我们修改的 PATH 环境变量生效。

```
$ ~/.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。

A terminal window with a black background and white text. The output of the 'java -version' command is displayed. The first line is 'java version "1.6.0\_38"'. The second line is 'Java(TM) SE Runtime Environment (build 1.6.0\_38-b05)'. The third line is 'Java HotSpot(TM) 64-Bit Server VM (build 20.13-b02, mixed mode)'.

```
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)
```

## 2.2 下载 Android SDK 包

从以下地址下载 Android SDK 包 [http://dl.google.com/android/android-sdk\\_r21.0.1-linux.tgz](http://dl.google.com/android/android-sdk_r21.0.1-linux.tgz)  
解压到你的 home 目录下，假定解压后的目录为/home/patcher/android-sdk-linux。

接下来编辑 home 目录下的.bashrc 文件，修改 PATH 环境变量：

```
export PATH=~/.android-sdk-linux/platform-tools:~/.android-sdk-linux/tools:$PATH
```

运行命令来使我们修改的 PATH 环境变量的修改生效。

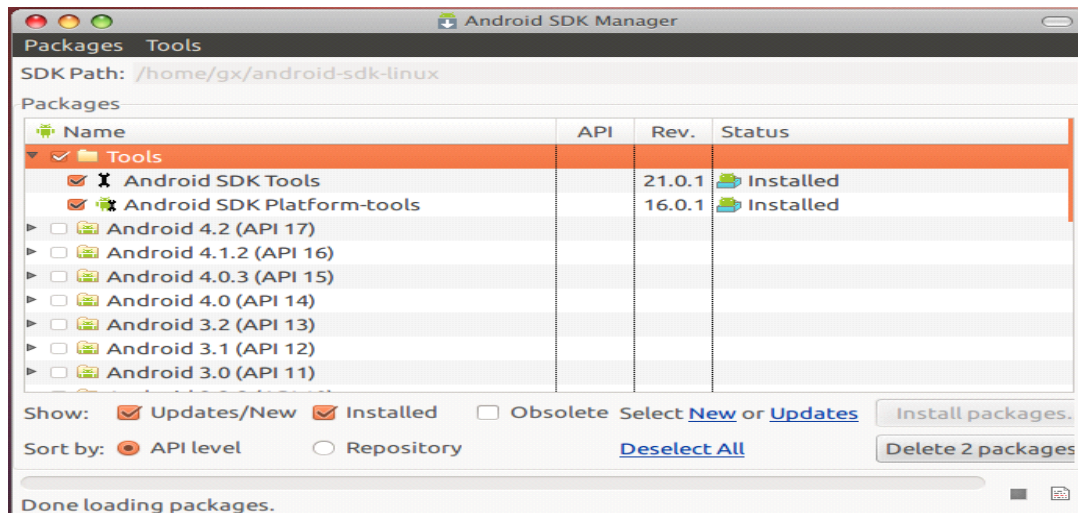
```
$ ~/.bashrc
```

## 2.3 安装

运行命令 android 来启动 Android SDK Manager

```
$ android
```

安装完成的结果如下图所示：



选中 Android SDK Tools 和 Android SDK Platform-tools，然后点击安装，接下来跟随应用程序的说明进行安装。这一步完成后，我们所需要的 Android SDK 也安装完毕了。

注：在 <http://developer.android.com/sdk/installing.html> 网页中，大家会看到需要安装 Eclipse，定制 MIUI ROM 不需要安装 Eclipse，这个是开发 Android 程序所需要的。

## 2.4 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。

```
List of devices attached
ACE87B700C0A    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：

```
Bus 002 Device 001:ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 098:ID 04e8:685e Samsung Electronics Co.,Ltd
.....
```

找到手机对应的那一行，记录下 04e8:685e，这个分别表示该设备的 vendorId 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：

```
SUBSYSTEMS="usb", ATTRS{idVendor}="04e8",ATTRS{idProduct}="685e",
MODE="0666", OWNER="登录用户名"
```

c) 重启 usb 服务，sudo restart udev，重连手机。

### 3.同步 MIUI 代码

我们创建一个 patchrom 目录，使用 patchrom 的树来初始化你的本地仓库

```
$ mkdir patchrom
```

```
$ cd patchrom
```

```
$ repo init -u git://github.com/Micode/patchrom.git -b ics
```

然后我们就可以开始同步啦。

```
$ repo sync
```

### 4.patchrom 项目

下面介绍 patchrom 的目录结构以及各目录的作用。

- ◆ android: 该目录下面有 5 个子目录。其中 src 目录和将要介绍的 miui/src 目录是一对一的关系。android/src 是 google 发布的 android 源码，miui/src 是 miui 在 google 源码基础上所做的修改。为了节省空间，在这两个目录下面，我们只放了 miui 修改过的文件。framework.jar.out, android.policy.jar.out, services.jar.out 是 miui 修改过的文件，与 google-framework 下面的文件是一对一的关系，google-framework 是基于 android 源码制作的，没有进行过修改。
- ◆ build: 该目录是一些与编译相关的脚本，包含了所有的 makefile 的构建。
- ◆ miui: 该目录下主要有 2 个子目录：system 和 src。system 目录下存放了的是由 miui 源码编译后的文件，这些文件是我们定制 MIUI ROM 所需要用到的所有文件。
- ◆ tools: 该目录下面包含了所有的脚本和工具程序，在编译过程中需要使用这些程序。

接下来我们开始编译生成 i9300 的定制 MIUI ROM，假定当前目录为 /home/patcher/patchrom 目录，

```
$ . build/envsetup.sh
```

```
$ cd i9300
```

```
$ make fullota
```

以上命令运行完毕后，在 i9300 目录下会生成一个 out 子目录，在子目录下的 fullota.zip 文件就是我们发布的 i9300 的刷机包了，可以通过 recovery 刷入你的 i9300。

MIUI 一直坚持每周五的更新，称为橙色星期五，下面就让我们来体验一下 OTA 的制作过程。

我们假定我们制作了 3.1.18 的 fullota，在 out 目录下面有 target\_files.zip，我们把它重命名为 last\_target\_files.zip。在 3.1.25 的时候，我们重新进行 fullota，这个时候，在 out 目录下面，我们就有了，last\_target\_files.zip 和 target\_files.zip，我们就是靠这两个文件来实现 OTA。这里我们需要注意的是，上一个版本里面的 target\_files

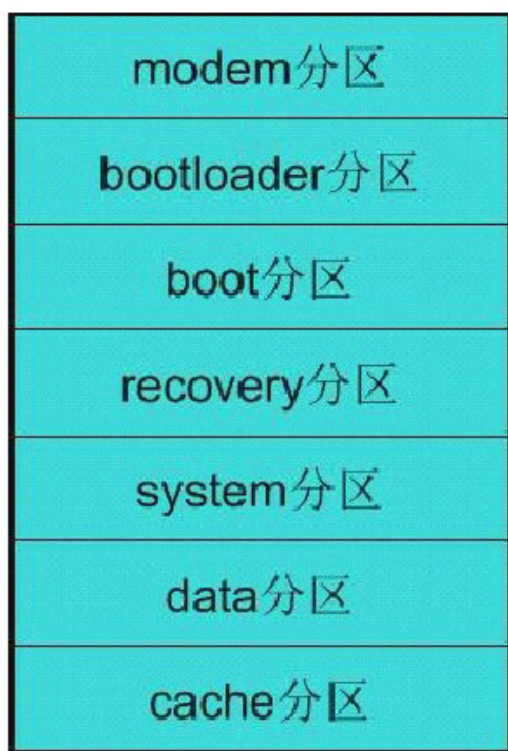
必须重命名，不然会在你重新 fullota 以后被替换成新的文件，就不能实现 OTA 的了。

```
假定当前的目录为/home/patcher/patchrom,  
$ . build/envsetup.sh  
$ ./tools/releasetools/ota_from_target_files -k ../build/security/testkey -i last_target_files.zip  
out/target_files.zip ota_update.zip  
-k: 指定签名的 key  
-i: 指定指定上次的 target_files  
out/target_files.zip: 指的是这次新的 target_files  
ota_update.zip: 指定生成的 OTA 包的文件名
```

## 第二章 认识 Android 手机

### 1.bootloader

当我们拿到一款手机，第一件事情应该就是安装上电池，然后按下电源键开机。那么，从开机到进入到桌面程序，这中间到底发生了些什么呢，我们就从下面这张简化了的手机结构图开始我们的探秘之旅：



注意：该结构图并不反映手机的实际分区顺序和位置，只是一个逻辑结构图。

大家可以简单的把手机的 ROM 存储类比为我们的电脑上的硬盘，这个硬盘被分成了几个分区：bootloader 分区，boot 分区，system 分区等。后面我们会逐个介绍各个分区的用途。所谓的刷机我们可以简单的理解为是把软件安装在手机的某些分区中，类似我们在电脑上面安装操作系统。

当按下电源键手机通电启动后，首先从 bootloader 分区中一个固定的地址开始执行指令。bootloader 分区可以分成两个部分，分别叫做 primary bootloader 和 secondary bootloader。Primary bootloader 主要执行硬件检测的功能，确保硬件能正常工作后将 secondary stage bootloader 拷贝到内存（RAM）中开始执行。Secondary stage bootloader 会进行一些硬件初始化的工作，比如获取内存大小信息等，然后根据用户的按键进入到某种启动模式。比如，大家所熟悉的通过电源键和其他一些按键的组合，可以进入到 recovery，



fastboot 或者选择启动模式的启动界面等等。我们在论坛上面看到的 bootloader 通常指的是 secondary stage bootloader。不过我们不需要关心太多的细节，可以简单地理解为 bootloader 就是一段启动代码，根据用户按键有选择的进入某种模式。

**fastboot 模式：**fastboot 是 android 定义的一种简单地刷机协议，用户可以通过 fastboot 命令行工具来进行刷机。比如：fastboot flash boot boot.img，这个命令就能把 boot.img 刷写进 boot 分区中。一般手机厂家不会直接提供 fastboot 模式刷机，而是为了显示他们的牛 B 之处，总是会提供自己专用的刷机工具和刷机方法。比如说：三星的 Odin，摩托罗拉的 RSD，华为的粉屏等等。但是，他们的本质实际上是相同的，都是将程序直接刷写到各个分区中。

**recovery 模式：**当进入 recovery 模式时，secondary stage bootloader 从 recovery 分区开始启动，recovery 分区是一个独立的 Linux 系统，当 recovery 分区上的 Linux 内核启动完毕后，开始执行第一个程序 init（init 程序是 Linux 系统所有程序的老祖宗）。init 会启动一个叫做 recovery 的程序（recovery 模式的名称也是由此而来）。通过 recovery 程序，用户可以执行清除数据，安装刷机包等操作。一般的手机厂家都提供一个简单的 recovery 程序，而大名鼎鼎的 CWM Recovery 就是一个加入了很多增强功能的 recovery 程序，要想用上 CWM Recovery 的前提是 recovery 分区可以被刷写。大家在论坛上面看到的解锁 bootloader，一般指的是解锁 recovery 或者 fastboot，允许刷写 recovery 分区，这样大家就能用上喜爱的 CWM Recovery 了。我们 MIUI 也为大家准备了 recovery，是按照我们的 ROM 的风格进行设计的，利用触屏进行控制。Recovery 的源码我们也已经在 github 上面开源，大家可以进行下载，recovery 的制作教程，我们也会尽快的整理教程进行发布。

手机除了普通的 CPU 芯片以外，还有 MODEM 处理器芯片。该芯片的功能就是实现手机必须的通信功能，大家通常所刷的 RADIO 就是刷写的 modem 分区。

## 2.正常启动

当我们只是按下电源键开机时，会进入正常启动模式。Secondary stage bootloader 会从 boot 分区开始启动。Boot 分区的格式是固定的，首先是一个头部，然后是 Linux 内核，最后是用作根文件系统的 ramdisk。

当 Linux 内核启动完毕后，就开始执行根文件系统中的 init 程序，init 程序会读取启动脚本文件（init.rc 和 init.xxxx.rc）。启动脚本文件的格式大家可以在网上找到很多参考资料。我们在原厂 ROM 上移植 MIUI 的原则是不修改 boot 分区，因为有一些机型无法修改 boot 分区。

根文件中有一个重要的配置文件，叫 default.prop，该文件的内容一般为：

```
#
#ADDITIONAL_DEFAULT_PROPERTIES
#
ro.secure=1
ro.allow.mock.location=1
```

```
ro.debuggable=0
persist.service.adb.enable=1。
```

文件中的每一行对某个属性赋值，在后续的文章中我们还会谈到属性。这里面大家需要注意两个属性：ro.secure 和 ro.debuggable。如果 ro.secure=0，则允许我们运行 adb root 命令，通常大家说的内核 ROOT 指的就是 ro.secure=0。而一般所说的 ROOT 权限指的是手机上有一个名为授权管理的程序（Superuser.apk）可以授权程序 root 用户的授权。

init 程序读取启动脚本，执行脚本中指定的动作和命令，脚本中的一部分是运行 system 分区程序，下一节我们就来看看 system 分区的结构。

### 3.System 分区

在讲 system 分区之前，我们先来看一下这张 Android 的软件系统架构图。



从上到下依次为：

- ◆ 核心应用层：这一层就是大家平常所接触的各种各样的系统的自带应用，比如联系人，电话，音乐等。应用层往下就是开发人员所接触的。
- ◆ 框架层：这一层是 Android 系统的核心，它提供了整个 Android 系统运作的机制，像窗口管理，程序安装包管理，开发人员所接触的 activity，service，broadcast 等。
- ◆ JNI 层：JNI 层是 Java 程序和底层操作系统通信的一个机制，它使得 Java 代码可以调用 C/C++代码来访问底层操作系统的 API。
- ◆ Dalvik 虚拟机：Android 开发使用 Java 语言，应用程序的 Java 代码会被编译成 dalvik 虚拟机字节码，这些字节码由 dalvik 虚拟机解析执行。
- ◆ 本地库：本地库一般是有 C/C++语言所开发的，直接编译成相应 CPU 的机器码，这其中包含标准 C 库，用以绘制图形的 skia 库，浏览器核心引擎 webkit 等。
- ◆ HAL：硬件抽象层，为了和各个厂家的不同硬件相互配合工作，Android 定义了一台硬件接口，比如说为了使用相机，厂家的相机驱动必须提供接口方法，这样使得上层的代码可以独立于不同的硬件运行。
- ◆ 厂家适配层：本来 Android 定义的 HAL 层是直接和厂家提供的设备驱动打交道的，但是

目前厂家不想开源 HAL 部分的代码,因此很多厂家都提供了一个我称之为厂家是赔偿呢过的代码,这样在 HAL 层接口的实现只是一个简单地对的厂家适配层接口函数的调用。

- ◆ 内核:这一层就是大家熟悉的 Linux 内核,内核包含了各种硬件驱动,这些驱动不同的手机厂家的手机是不一样的。Linux 内核是支持驱动模块化机制的,简单地说就是允许用户动态的加载或者卸载某个硬件驱动,但是目前看来,手机厂家除了提供 WIFI 驱动单独加载外,其它驱动都是和内核绑定在一起的。

从这张软件结构图来看,除了内核是放在 boot 分区外,其它层的代码都是在 system 分区中,下面结合这张图来介绍 system 分区的主要目录内容:

- ◆ system/app:app 目录下存放的是核心应用,也就是大家熟知的系统 APP,这些系统自带的程序是不能简单的卸载的,要通过一些特殊的方式才能删除(大家熟悉的一种方法就是 ROOT 完了以后,使用 RE 文件管理器)。
- ◆ system/lib: 目录下存放的是组成 JNI 层, Dalvik 虚拟机,本地库, HAL 层和厂家适配层的所有动态链接库(.so 文件)。
- ◆ system/framework: 该目录下存放的是框架层的 JAR 包,其中对适配 MIUI 来说,有 3 个最重要的 JAR 包(framework.jar, android.policy.jar, services.jar)。后面我们会重点介绍这 3 个 JAR 包。
- ◆ system/fonts: 该目录下存放的是系统缺省的字体文件。
- ◆ system/media: 该目录下存放的是系统所使用的各种媒体文件,比如开机动画,壁纸文件等。不同的手机该目录的组织方式可能不一样。
- ◆ system/bin: 该目录下存放的是一些可执行文件,基本上是由 C/C++编写的。
- ◆ system/sbin: 该目录下存放的是一些扩展的可执行文件,所以该目录可以为空。大家常用的 busybox 就放在该目录下。Busybox 所建立的各种符号链接命令也都在这个目录下。
- ◆ system/build.prop: build.prop 和上节说的根文件系统中的 default.prop 文件格式一样,都称为属性配置文件。他们都定义了一些属性值,代码可以读取或者修改这些属性值。属性值有一些命名规范:
  - ro 开头的表示只读属性,即这些属性的值得代码是无法修改的。
  - persist 开头的表示这些属性值会保存在文件中,这样重新启动之后这些值还会保留。
  - 其它的属性一般以所属的类别开头,这些属性是可读可写的,但是对它们的修改重启之后不会保留。

很多 romer 都会修改一下 build.prop 信息,里面的一些以 ro.build 开头的属性就是你在手机设置中的关于手机里看到的。可以通过修改 build.prop 文件来将这个 ROM 打上自己的印记。

- ◆ `system/etc`: 该目录存放一些配置文件, 和属性配置文件不一样, 这下面的配置文件可能稍微没那么有规律。一般来说, 一些脚本程序, 还有大家所熟悉的 GPS 配置文件 (`gps.conf`) 和 APN 配置文件 (`apns-conf.xml`) 放在这个目录。像 HTC 的特效相机所使用的一些文件也放在这个目录下。

## 4.data 和 cache 分区

这一节简单的介绍一下 data 和 cache 分区。当我们开机进入桌面程序后, 一般来说我们会下载安装一些 APP, 这些 APP 都安装在 `data/app` 目录下。所有的 Android 程序生成的数据基本上都保存在 `data/data` 目录下。Wipe data 实质上就是格式化 data 分区, 这样我们安装的所有 APP 和程序数据就丢失了。

Cache 分区从名字上来看是用来缓存一些文件的, 比如说一些音乐下载的临时文件, 或者下载管理下载的内容存放在这个分区。

## 5.小结

本章主要是介绍 Android 手机的硬软件结构以及主要分区的内容, 并简要的介绍了一些开机启动过程。了解这些内容有助于我们在整体上理解 ROM 适配。

# 第三章 寻找合适的原厂 ROM

## 1.熟悉适配的机型

### 1.1 逛论坛刷机

想要打人先学会被打，想做刷机包先学会刷机。先去各大论坛逛逛，了解你的机型是如何刷机的。我们不得不提到一个避光的论坛：<http://forum.xda-developers.com/>。这是国外的一个手机论坛，该论坛技术性强，用户富有分享精神，机型全面。

这个期间一定要掌握你的机型的刷机方法，需要用到的工具，多刷几个 ROM，熟悉刷机过程。

### 1.2 合适的原厂 ROM

在第一步熟悉了要移植的机型，刷了各个版本的 ROM 后，接下来我们要开始集中精力寻找一个合适的原厂 ROM。一般来说，原厂 ROM 的稳定性好。这个时候能找到那种在原厂 ROM 的基础上仅做过 ROOT 和 deodex 的版本是最好的。

那么如何判断一个原厂 ROM 是否合适呢？？

首先要版本合适，我们这个系统讨论的是基于原厂 ROM 适配 MIUI，目前 4.0 的 MIUI 是基于 android4.0.4 源码开发的，从 android4.0.1 到 android4.0.4 这几个版本变化都不太大，因此 4.0.1 到 4.0.4 的原厂 ROM 版本都是合适的。

其次检查所安装的 ROM 是否有 root 权限。Root 权限分两种：

第一种是手机 root：这种 root 权限的外在表现是你的手机上安装了一个授权管理软件；

第二种是内核 root：这种 root 权限的外在表现如下：

在 Ubuntu Shell 下运行如下命令：

`$adb root` (该命令的含义是以 root 权限运行 adb)

`$adb remount` (该命令的含义是将 system 分区的权限设成可读可写)

如果这两条命令都成功，表明内核 root。运行 adb shell，可以看到手机 shell 提示符为#。

如果上述两条命令失败了，运行 adb shell 可以看到手机 shell 提示符为\$。如果此时运行 su 命令，手机弹出是否授予 root 权限，这说明手机上安装了授权管理程序。这种情况下运行 su 命令后，手机 shell 提示符也会变为#。

在之后的章节我们会看到，适配 MIUI 的关键是能修改 system 分区的内容，这两种 root

权限都可以将 system 分区设成可读写的，只是内核 root 权限提供了最大的方便性，强烈推荐找到一个内核 root 过的 ROM。

Patchrom 项目提供的工具和脚本是基于你的手机获取了内核 root 权限，如果是手机 root，也是可以的，但是需要修改一下子脚本。因为只是手机 root，adb remount 命令会失败。这个时候需要在手机 shell 里重新 remount system 分区，并且修改 system 的目录权限，这样才能修改 system 分区的内容，而且需要修改我们提供的某些脚本。（之后不针对手机 root 权限做出特殊说明，我们相信你知道如何在这种模式下修改 system 分区）。

最后检查所选择的 ROM 可以进入 Recovery 模式刷机，不一定要求必须是 CWM Recovery，有 wipe data/cache 和安装 zip 包等功能的简单 Recovery 就可以，当然有 CWM Recovery 是更好了。针对每个机型，要想自动生成 MIUI ROM 刷机包，我们要求对每个机型提供一个 zip 格式的刷机包，该刷机包可以通过 Recovery 安装。对于有的机型，如果没有找到可以通过 Recovery 安装的 zip 包怎么办，不用担心，我们下面就教你怎么从手机里面提取出符合要求的 zip 包。

首先，假设我们现在有一个机器叫做 mi，电脑当前的目录是 patchrom：

```
$ . build/envsetup.sh
$ mkdir mi
$ cd mi
```

现在，我们将手机连接上电脑，并且我们要确保 USB 调试在打开状态下。

\$ adb reboot recovery（如果此命令不起作用，那么就手动重新启动，通过组合按键进入 Recovery）

\$ ../tools/ota\_target\_from\_phone -r（-r 表示我们在 Recovery 模式下运行此工具，我们也能在正常模式下运行，请参照此工具的脚本，从而知道此工具的更完善的用法）

此工具将会生成一个 stockrom.zip 文件和元数据目录，我们可以用它来适配 MIUI。

## 1.3 adb logcat

在第一章介绍过 adb 是一个非常重要的命令，其中在机型适配过程中我们最常用的就是 adb logcat。通过这个命令我们可以看到详细的 log 信息。

每一行的大致格式为：

```
I/ActivityManager( 585):Starting activity:Intent{action=android.intent.action..}
```

其中第一个字母表示信息优先级别（E 表示错误，W 表示警告，I 表示普通信息）。

斜杠后的 ActivityManager 表示信息标记 tag，通常标记表示了打印出相应信息的模块或程序。

可以通过 adb logcat tag:\* \*:S 只显示相应 tag 打印的所有信息。

括号中的数字表示进程 ID (pid)，表示程序所在进程 ID。

冒号后的句子就是具体的信息说明了，当我们遇到错误的时候 adb logcat 会给出详细的错误信息，我们通过这些错误信息去定位错误。

在机型适配中常用 adb logcat\*:E 来查看所有的错误信息。

详细的 adb 说明可以参考 <http://developer.android.com/guide/developing/tools/adb.html>。在选定好 ROM 以后，我们要确保开机之初，差不多是显示开机动画时 adb logcat 命令就能显示详细的 log 信息。如果 adb logcat 只是在桌面程序出现之后才打印信息或者根本就不打印任何信息，适配工作就很难进行下去了。如果只是简单的修改一些图片资源的话，那是完全可以的，但是对于适配 MIUI 来说，我们要求在适配机型一开始就确保 adb logcat 功能的正常运行。

## 2.修改 boot.img

在第二章认识 Android 手机中我们提到过，内核 root 的关键是根文件系统中 default.prop 文件的两个属性 ro.secure 和 ro.debuggable 的值。根文件系统和内核一起放在 boot 分区中，如果我们能修改 boot 分区中的这个文件，那么我们就可以自己 root 内核了。

下面介绍一个 root 内核的方法，一般来说，某个机型的完整刷机包下有一个 boot.img 文件，该文件就是 boot 分区的镜像文件，安装刷机包时，会使用该文件刷写 boot 分区。Google 给 boot.img 文件定义了一个标准的格式，如果遵从这个标准格式，我们可以用下面的方法来修改它，但是如果不遵从，需要逛论坛详细的了解如何修改 boot 分区。判断是否遵从有一个简单但是不一定准确的办法，如果刷机包中的文件名为 boot.img，一般来说遵从。如果不是，那么不遵从。比如说三星的 i9100 的刷机包中这个文件命名为 zImage（可以在 xda 论坛上找到详细的修改 zImage 的教程）。

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

```
$ ./tools/unpackbootimg -i boot.img
```

输出类似如下文字：

```
BOARD_KERNEL_CMDLINE console=ttyDCC0 androidboot.hardware=xxx
```

```
BOARD_KERNEL_BASE 00200000
```

```
BOARD_PAGE_SIZE 4096
```

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 Linux 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 default.prop 文件。

```
$ cd ..
```

```
$ ./tools/mkbootfs ./ramdisk | gzip > ramdisk-new.gz
```

将 ramdisk 目录重新打包。

```
$ ./tools/mkbootimg --cmdline 'console=ttyDCC0 androidboot.hardware=xxx'
--kernel boot.img-kernel --ramdisk ramdisk-new.gz --base 0x00200000 --pagesize
4096 -o boot-new.img
```

运行该命令生成新的 boot.img,

--cmdline: 该选项为之前打印的 BOARD\_KERNEL\_CMDLINE

--kernel: 该选项为 Linux 内核文件

--ramdisk: 该选项为根文件系统压缩包

--base: 该选项为之前打印的 BOARD\_KERNEL\_BASE

--pagesize: 该选项为之前打印的 BOARD\_PAGE\_SIZE

-o: 该选项为输出文件名

### 3.deodex

当我们把要移植的机型按照上述步骤刷好了合适的原厂ROM后，第一件事就是需要做 deodex。

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：

```
$ 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。

### 4.Makefile

上面我们已经说了怎么选取合适的底包，我们需要一个 makefile 文件，才能使我们的 patchrom 工作起来。下面是最基本的 patchrom：

```
local-zip-file           :=stockrom.zip
local-out-zip-file       :=
local-miui-modified-apps :=
local-modified-apps      :=
local-miui-removed-apps  :=
local-phone-apps         :=
local-pre-zip            :=local-zip-misc
```



```
include $(PORT_BUILD)/porting.mk
local-zip-misc:
```

下面我们来解释一下这个文件的各个定义，理解了这些定义，你能清楚的看透整个创建系统了。

- ◆ local-zip-file: 这个 ROM 就是上面我们寻找的合适的底包，必须对应各个机型
- ◆ local-out-zip-file: 当你运行” make zipfile” 以后，输出的 ROM 的名字
- ◆ local-miui-modified-apps: 在 patchrom/build/miuiapps.mk 中列出了所有的 miui apps，这个变量定义了哪些是我们修改过的 miui apps。
- ◆ local-modified-apps: 这个变量定义了哪些 apps 是我们在 local-zip-file 中修改过的
- ◆ local-miui-removed-apps: 通常不是所有的 miui apps 都是适合我们的设备的。这个定义了我们哪些 miui apps 是我们不需要的。
- ◆ local-phone-apps: 这个定义了我们需从 local-zip-file 里面去除的 apps。正如我们所看到的，我们给 miui apps 设置了黑名单，给 stockrom 里面的 apps 设置了白名单。这个定义我们需要的就是我们需要在 stockrom 里面留下来的 apps。
- ◆ local-zip-misc: 这个目标允许我们根据自己的设备的情况，做一些自定义。

## 5.workspace

首先我们运行的” make workspace” 命令，这个命令将会为我们准备工作区。进行的操作很简单，就是从 stockrom.zip 提取出 framework.jar, android.policy.jar, services.jar, framework-res.apk, 并且用 apktool 这个工具来拆解他们。

## 6.firstpatch

第二个 make 命令，我们运行” make firstpatch”。待命令运行完成以后，在我们的文件夹下面会产生了一个 tmp 文件，里面有 5 个子文件夹：

- (1) old\_smali: 去掉.line 的来自 Google 原版的 framework.jar 的 smali 代码
- (2) new\_smali: 去掉.line 的来自 MIUI 修改过的 framework.jar 的 smali 代码
- (3) dst\_smali\_orig: 去掉.line 的来自 stockrom 的 framework.jar 的 smali 代码
- (4) dst\_smali\_patched: 去掉.line 的已经 patch 过 MIUI 代码的 smali 代码，smali 代码来自 stockrom 的 framework.jar 中
- (5) reject: 有冲突的 patch 文件，需要我们手动来解决的部分

## 7.fullota

在我们解决完这些冲突以后，你就可以运行” make fullota”来建立你的最终的 ROM 了。然后通过 Recovery，将 ROM 刷入你的手机，开机进入桌面，如果进不了桌面也不用担心，前面我们已经讲解了 adb 的运用，我们利用这个工具来找到 bug 并且修改。

## 第四章 反编译

这一章我们来详细的看看反编译，想要修改一个系统自带的应用程序和它的代码，在没有源码的情况下，我们就不得不用反编译来修改。

和很多书籍一样，为了向经典的”Hello, World”致敬，我们也从一个简单的程序开始 HelloActivity.apk。当你把这个 APK 安装到手机上运行后，在屏幕上面就显示一行文字”Hello, World!”

### 1.反编译

假定在 patchrom 目录下：

```
$ ./tools/apktool d HelloActivity.apk
```

这条命令运行完后，在当前目录下会生成一个名为 HelloActivity 的目录。

该目录的结构为（名称后跟/表示这是一个目录）：

```
HelloActivity/
|-----AndroidManifest.xml
|-----apktool.yml
|-----res/
|-----smali/
```

apktool.yml 是 apktool 生成的一个配置文件，基本上你不需要修改这个文件。下面的章节我们逐个介绍剩下的 AndroidManifest.xml 文件和 res, smali 目录。

### 2.AndroidManifest.xml

要想完全理解这个文件，你得对 Android 的内部运作机制非常清楚。幸好我们修改一个 APK 的时候基本上不修改这个文件。这里帮助你有一个大致的了解。

Android 安装程序一般叫 apk 文件（apk 是 Android Package 的缩写，表示 Android 安装包）。一般来说，程序都会有一个或者多个 Activity，Activity 是什么呢？从概念说它是一个和用户交互的窗口，你每天使用 Android 手机的时候基本上你打交道的每个界面都是一个 Activity。AndroidManifest.xml 是一个 xml 格式的清单文件，就像你去超市买东西打印出来的一个购物清单，AndroidMainfest.xml 也起着一个清单的作用，它告诉系统，我有这些 Activity。（实际情况远比这个复杂得多，想学 Android 编程的通信请看这个 <http://developer.android.com/guide/index.html>）。

具体到 HelloActivity 下的 AndroidMainfest.xml 文件，大家可以找到如下内容：

```
<application
Android:label="@string/app_name" android:icon="@drawable/ic_launcher_hello" >
    <activity android:name=" HelloActivity" >
        <intent-filter>
```

```

        <action android:name=" android.intent.action.MAIN" />
        <category android:name=" android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
</applicatin>

```

其中有一行<category android:name=" android.intent.category.LAUNCHER" />, 包含这一行的 Activity 会显示在桌面上, 就是说你可以通过桌面显示的图标启动这个 Activity。里面还有 android:label=" @string/app\_name" android:icon=" @drawable/ic\_launcher\_hello"。这两个属性是做什么的呢, android:label 表示程序显示在桌面上的名字, android:icon 表示程序显示在桌面上的图标。如果想要改显示的名字和图标, 修改其后的两个资源, 如何修改资源, 下一节详细介绍。

### 3.资源

res 目录下放置了程序所需要的所有资源。资源是什么呢?? 一般来说, 一个图形用户界面 (GUI) 程序总是会使用一些图片, 或者显示的文字的大小和颜色等, 或者界面的布局, 比如显示的界面上面是文字, 下面是两个按钮等等。这些程序的一个重要特点就是用户界面和代码逻辑的分离。当我们需要替换图片或者简单修改界面布局的时候, 不需要改变代码。而 Android 程序会将这些代码中需要用到的文件都放在 res 目录下面, 称之为资源。

可以看到 res 目录的内容为:

```

res/
|-----drawable-hdpi/
|           |-----ic_launcher_hello.png
|-----layout/
|           |-----hello_activity.xml
|-----values/
|           |-----ids.xml
|           |-----public.xml
|           |-----strings.xml

```

对于 HelloActivity 来说, res 目录下有三个子目录 drawable-hdpi, layout, values。由于 HelloActivity 比较简单, 因此 res 下内容不多, 但是一个复杂的程序 res 目录下内容相应的也会比较多, 但是基本原理都是一样的。

res 下面的子目录基本上是按照资源类型来分类组织的, 以 drawable 开头的表示图片资源, 大家可能会看到 drawable-xhdpi, drawable-hdpi, drawable-mdpi, drawable-ldpi 等, 这些 xhdpi, hdpi, mdpi, ldpi 分别表示分辨率的不同级别, 会根据不同的屏幕分辨率选择不同的图片。要想替换图片, 替换这些目录下的图片就可以了。(替换图片比这稍复杂一点, 一般替换图片, 最好保持和原图片兼容, 比如说色系, 尺寸以及点 9 图片的一些参数等)。

以 layout 开头的表示布局文件，用来描述程序的界面。anim 子目录存放程序使用到的动画，xml 开头的目录存放程序用到的一些 xml 文件等。

values 开头的目录下面存放一些我们称之为基本元素的定义，比如说 colors.xml 给出颜色的定义，dimens.xml 给出一些大小的定义。Strings.xml 是一些字符串的定义。我们看看 HelloActivity 的 strings.xml 文件。

```
<?Xml version=" 1.0" encoding=" utf-8" ?>
<resources>
    <string name=" hello_activity_text_text" >Hello,World!</string>
    <string name=" app_name" >HelloWorld</string>
</resources>
```

其中的以<string 开头表示这是一个字符串，name 是给这个资源起的一个名字，后面的字符串表示这个字符串的值。Android 的资源大致按这种形式来组织的，先将资源分成几种类型，然后每一种类型的所有资源取一个名字，这个名字对应了这个资源的值/内容。

我们一般修改资源，通常情况下是修改图片或者汉化。汉化比较简单，values/string.xml 文件存放程序用到的所有英文字符串值。要汉化，首先在 values 下建立一个目录 values-zh-rCN。然后将 values/strings.xml 拷贝到该目录中，将每个字符串翻译成中文。我们现在将 strings.xml 拷贝到 values-zh-rCN 目录下，并将文件内容改为：

```
<?xml version=" 1.0" encoding=" utf-8" ?>
<resources>
    <string name=" hello_activity_text_text" >你好，世界！</string>
    <string name=" app_name" >你好世界</string>
</resources>
```

现在我们需要把修改后的文件再编译回 apk 文件，运行命令如下：

```
$ ./tools/apktool b HelloActivity HelloActivity.apk
```

这条命令表示编译 HelloActivity 目录的内容，输出文件为 HelloActivity.apk，如果你不想覆盖原有的文件，乐意换一个名字或者放在另一个目录下面。

接下来我们需要对生成的 APK 进行签名，

```
$ ./tools/sign.sh HelloActivity.apk
```

```
$ adb install -r HelloActivity.apk.signed.aligned
```

注意最后一条命令，如果你失败了，如果你不是用我们提供的 HelloActivity 做实验的话，会发生签名不一致的错误，这个时候先卸载原来的，再安装。运行看看，现在显示在你面前的就是“你好，世界！”

汉化成功了，是的！汉化就是那么的简单。如果你只想停留在汉化或者替换图片的这个阶段，从这里开始以后的文章就不用看了。如果你没有 Android 编程基础，从这里开始以后的文章也不用看了。

到底发生了什么魔法，为什么这样替换一些图片或者改字符串就能改变程序最终运行的

结果呢，想要理解这个，我们就得大致的了解一下资源的编译过程。首先我们看看 values 目录下的一个有意思的文件 public.xml，他的内容如下：

```
<?xml version=" 1.0" encoding=" utf-8" ?>
<resources>
    <public type=" drawable" name=" ic_launcher_hello" id=" 0x7f020000" >
    <public type=" layout" name=" hello_activity" id=" 0x7f030000" >
    <public type=" string" name=" hello_activity_text_text" id=" 0x7f040000" >
    <public type=" string" name=" app_name" id=" 0x7f040001" >
    <public type=" id" name=" text" id=" 0x7f050000" />
</resources>
```

每一行的 id 后面都有一个看起来很奇怪的数字，这些数字是干什么的呢？Android 下有一个资源编译器会编译 res 目录下的所有文件，它为每一个资源名字分配一个数字标示符，这个标示符分成 3 个部分，最前面的 1 个字节表示包名，所有的 apk 这个字节都是 7f。表示这些资源是非共享的，其它 APK 访问不到。所有那些可以共享的资源放在 system/framework/framework-res.apk 里面。System/framework 往往还有其它共享的资源包。这些共享的资源包前面的 1 个字节从 0x1 开始，一次增加。中间的一个字节表示资源的类型，每一个类型的数字标示符是不一样的，最后的 2 个字节是资源的序号，统一类型的资源序号从 0 一次往上递增。一般来说，资源 id 是由资源编译器（aapt）自动产生的，但是定义在 public.xml 中的值告诉编译器，你必须为这个 id 使用这个值。Apktool 会为所有的资源名称定义这个值在 public.xml 里，这样可以保证替换资源后资源的 id 不会变化。

为什么资源的 id 这么重要，如果变了，会怎么样呢，这得结合代码来理解。我们在 Java 代码里通常这样引用资源，比如 R.string.app\_name。这个 Java 代码经过编译后，这条引用直接变成了资源 id，即 0x7f040001，所以你在下面反编译后的 smali 代码里面是看不到 R.string.app\_name 这个东西的，只能看到 0x7f040001。资源编译器会产生一个查找表，对于每个 id，查找表中保存了这个 id 对于的名字和值（如果是文件，则为文件所在路径）。程序在运行的时候，会根据 id 去查询这个查找表找到对应的资源的值或者文件。

最后说一句，资源 ID 真的非常重要，运行 adb pull /system/framework/framework-res.apk 反编译这个文件，好好的咀嚼一下这一节的内容吧。

## 4.smali

终于我们迎来了最重要的部分了——smali 目录，smali 目录存放的是反编译后的 Java 代码，文件名以 smali 结尾，故称作 smali 文件。这些代码比一般的 Java 代码可读性差很多，但是和传统的 x86 或者其他体系结构下的汇编文件相比较的话，那还是好读了很多。虽然有工具可以直接把这些反编译成 java 代码，但是好不了很多，我们还是直接读取修改 smali 文件。

我们来看一下反编译后的 smali 目录下的 HelloActivity.smali 文件，和 java 组织源码的方式一样，smali 目录下的文件也是按照文件包的包名结构来组织目录结构的，文件的内容如下：

```

.class public Lcom/example/android/helloactivity/HelloActivity;
.super Landroid/app/Activity
.source "HelloActivity.java"

# direct methods
.method public constructor<init>()V
    .locals 0

    .prologue
    .line 27
    invoke-direct {p0}, Landroid/app/Activity; -> <init>()V

    return-void
.end method

# virtual methods
.method public onCreate(Landroid/os/Bundle;)V
    .locals 2
    .parameter " savedInstanceState"

    .prologue
    .line 33
    invoke-super {p0, p1}, Landroid/app/Activity; -> onCreate(Landroid/os/Bundle;)V

    .line 37
    const/high16 v1, 0x7f03

    invoke-virtual {p0, v1},
        Lcom/example/android/helloactivity/HelloActivity; -> setContentView(I)V

    .line 38
    const/high16 v1, 0x7f05

    invoke-virtual {p0, v1},
        Lcom/example/android/helloactivity/HelloActivity; -> findViewById(I)Landroid/view/View;

    move-result-object v0

    check-cast v0, Landroid/widget/TextView;

    .line 39
    .local v0, textView:Landroid/widget/TextView;
    const/high16 v1, 0x7f04

```

```

        invoke-virtual {v0, v1}, Landroid/widget/TextView; -> setText(I)V

        .line 40
        return-vvoid
    .end method

```

文件中的以#开头的文字表示注释，以.开头的叫做 annotations，其中的.line 表示对于的源代码的行号，这个对调试很重要。.method 和.end method 表示一个方法定义的开始和结束。Smali 文件中的重要指定的功能，请参照本文档的副页，里面有详细的介绍。有过 java 编程基础的很容易理解这些指令。

其中.line 39 的代码对应的源代码是

```
textView.setText(R.string.hello_activity_text_text)
```

我们现在想将这行代码改成 textView.setText(“Happy, Cracker!”)，将.line39 到.line40 行的代码改为：

```

        .line 39
        .local v0, textView:Landroid/widget/TextView;
        const-string v1, "Happy, Cracker!"
        invoke-virtual {v0, v1}, Landroid/widget/TextView; -> setText(Ljava/lang/CharSequence;)V

```

再按照上一节所说的重新编译，签名，安装运行。OK，现在出现在你面前的就是 Happy, Cracker! 了。

接下来的两张我们都会介绍如何直接修改 smali 代码从而改变程序的功能，这种方法我们叫做代码插桩，接下来的两章我们将会用代码插桩的方法将 MIUI 的功能加到原生 ROM 中去了。



# 第五章 适配 MIUI Framework

## 1.为什么使用代码插桩

首先我们来回顾第一章中的 Android 软件架构图,这个图中框架层的代码完全是由 Java 语言编写的,对于这两层的代码,在没有源码的情况下,我们可以采取代码插桩的方式来注入我们的代码。但是对于下面的几层的代码几乎都是以机器码的形式存在的,机器码也是可以修改的,但是修改难度和修改 smali 代码的难度不可同日而语。我们这个系列的文章不介绍如何修改这些机器码,大家感兴趣的可以参考网上的相关资料。MIUI 是基于源码开发的,为了提升整个效率,我们会修改下面几层的代码,比如说我们修改了 dalvik 虚拟机,skia 绘图库等。幸好这些修改不多而且有些是为了提升性能的,不影响 MIUI 的整体功能。MIUI 的绝大部分修改都是对框架层和核心应用层,这样保证了我们在原厂 ROM 的基础上修改这两个层的 smali 代码达到适配 MIUI 的目的。

大家看到这里可能有一个疑问,我们直接替换原厂 ROM 框架层和核心应用层这两层的代码不就可以了?不行,因为各个层次之间是有关联的,框架层和下层代码的一些调用接口是各个厂家自己扩展的,简单的整体替换 MIUI 框架层和核心应用层的代码,是无法工作的。

## 2.适配规范

本节我们通过详细的介绍 android,miui 和 i9100 这三个目录来探讨一下代码的组织规范。

### 2.1 android, miui

在我们开源的代码中,android 和 miui 的里面的文件,我们在第一章已经解释过里面的各个文件夹的作用了,在这里我们就不解释了。

### 2.2 i9100

i9100 是我们为适配三星 i9100 机型新建的一个目录,每个机型都需要新建一个目录。该目录的组织规范为:

- ◆ 放置一个原厂的 ROM 刷机包,比如 stockrom.zip。
- ◆ 基于原厂 ROM 中的文件反编译修改的,需要存放反编译后整个目录的内容。比如:  
android.policy.jar.out , framework.jar.out , services.jar.out ,

framework2.jar.out。这些目录相对应的是刷机包中的 system/framework/android.policy.jar，system/framework/framework.jar，system/framework/framework2.jar，system/framework/services.jar 这些文件反编译而来的。

- ◆ 基于 MIUI 的文件反编译修改的，只存放反编译后整个目录中修改的部分。
- ◆ 有一个 makefile，i9100 中的 makefile 给出了详细的注释，请参考该 makefile 写其他机型的 makefile。有了 makefile 之后，很多工作就可以自动化了。具体的可以参考 build/porting.mk 和 build/util.mk 中的注释。基本 makefile 在 build 文件夹中，可以选用。

### 3.移植资源

移植资源就是修改 framework-res.apk，先反编译原厂 ROM 中的 framework-res.apk，然后根据 miui/src/frameworks/miui 目录下的文件，修改反编译 framework-res 目录中的相应文件。

### 4.修改 smali

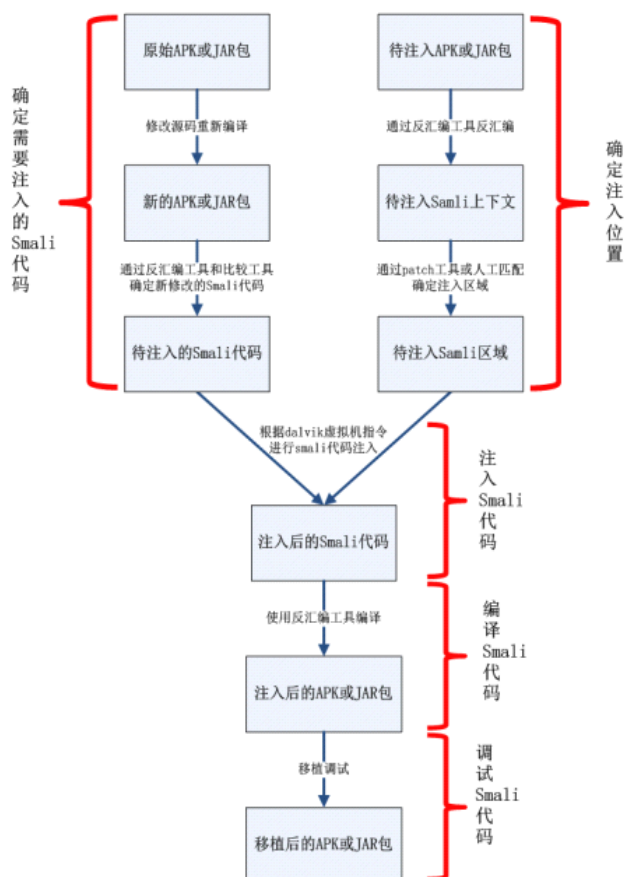
这一节我们以 i9100 为例，重点介绍如何修改原厂 ROM 的 smali，将 MIUI 的修改应用到上面去。我们不会将所有的修改都会在文中列出来，挑选几个有代表性的讲解，剩下的大家可以自己去做。

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 miui 和原生 android 的差异，比较 i9100 和原生 android 的差异。

以 framework.jar 为例，首先在 android，i9100 这两个目录中，我们在 android 目录下需要 /android/framework.jar.our 和 /android/google-framework/framework.jar.out，目录，然后我们反编译 i9100 里面的 framework.jar 文件，会产生 framework.jar.out 文件。

为了方便比较差异性，我们建议在 android 和 i9100 中分别建立一个目录 noline，android 目录下的 noline 因为有两个 framework.jar.out，我们可以进行重命名，来区分开。使用 patchrom/tools 中的脚本 rmline.sh 运行如下命令，假定在 patchrom 目录下：

```
$ ./tools/rmline.sh i9100/noline/framework.jar.out
```

rmline.sh 是用来把 smali 中所有的以 .line 开头的行去掉，这样我们容易比较 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 miui 和 android, i9100 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：当比较 miui 和 android 的 smali 代码时，你会发现我们并没有修改这些 smali 文件对应的 Java 文件，不需要修改这些文件，我们只需要修改与我们修改的源文件对应的 smali 文件）。

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 miui 改了其中一个方法 getTopLevelResources，而 i9100 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

还是以 ActivityThread.smali 为例，对于这个文件，miui 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，miui 修改了这个方法，i9100 也修改了这个方法。怎么办呢，我们先分析一下 miui 修改的代码：

```
.method final applyConfigurationToResourcesLocked(Landroid/content/res/Configuration;)Z
...
invoke-virtual {v5,p1},
    Landroid/content/res/Configuration;->updateFrom(Landroid/content/res/Configuration;)I
move-result v0
.local v0,changes:I
invoke-static {v0,Landroid/app/MiuiThemeHelper;->handleExtraConfigurationChanges(I)V
invoke-virtual {p0,v7},
    Landroid/app/ActivityThread;->getDisplayMetricsLocked(Z)Landroid/util/DisplayMetrics;
move-result-object v1
.local v1,dm:Landroid/util/DisplayMetrics;
```

在上面将miui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 8 表示这个方法是用8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miui新增了一个静态方法调用，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，只新增了一个静态方法调用。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i9100的该方法的smali代码中找到相应的位置，把这个修改应用到i9100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miui插入的代码并不是一个线性代码，而是有条件判断的。我们以Resoueces.smali为例，miui修改了其中的loadDrawable方法，修改后的结果如下：

```
.method loadDrawable(Landroid/util/TypedValue;I)Landroid/graphics/drawable/Drawable;
.end local v8      #e:Ljava/lang/Exception;
.end local v13     #rnf:Landroid/content/res/Resources$NotFoundException;
:cond_6

invoke-virtual/range {p0..p2},
Landroid/content/res/Resources;->loadOverlayDrawable(Landroid/util/TypedValue;I)Landroid/gra
phics/drawable/Drawable;
move-result-object v6
if-nez v6,:cond_1

:try_start_1
Move-object/from16 v0,p0
```

红色的代码是miui插入的代码，我们再看一下i9100相对于原生android对这个方法的改动，发现改动非常大。这种情况怎么办呢？这种情况下的关键是找到所插入代码的入口点和出口点（即这段代码是从哪执行而来，执行完毕后又往哪去开始执行代码）。

首先，我们发现插入代码的前面是一个标号:cond\_6，这说明程序中应该有一个跳转语句跳转到这个标号:cond\_6。而且这种程序应该也可以从:cond6上面的语句顺序执行而来（即它可能有两个入口点），我们分别去找这两个入口点的代码。首先我们去找哪个语句使用了:cond\_6，找到如下代码：

```
const-string v15,".xml"
invoke-virtual {v9,v15},Ljava/lang/String;->endsWith(Ljava/lang/String;)Z
move-result v15

if-eqt v15,:cond_6
```

可以发现这段代码是在判断v9这个字符串是否以".xml"结尾，如果不是的话，跳转到:cond\_6。好了，我们去i9100中找到相应的代码逻辑。对于这个例子，我们完全可以以".xml"作为关键字去i9100的loadDrawable方法中搜索一下，定位到如下代码：

```
const-string v17,".xml"
move-object v0,v10
move-object/from16 v1,v17
invoke-virtual {v0,v1},Ljava/lang/String;->endsWith(Ljava/lang/String;)Z
move-result v17
if-eqz v17,:cond_b
```

这段代码的逻辑和我们在miui中找到的代码一样，看来，我们应该把miui插入的代码插入到:cond\_b之后。找到i9100代码中的:cond\_b之后，我们看看这条代码后面的代码，发现和我们插入的代码和后面的代码基本类似，这下可以确定miui新插入的代码应该放在:cond\_b之后了。

再来看看出口点，miui插入的代码有两个出口点（是一个条件判断），

```
if-nez v6,:cond_1
```

如果v6为空往下执行，如果不为空，则跳转到:cond\_1，好，我们一起来看看:cond\_1

的代码是起什么作用？:cond\_1的代码如下：

```
:cond_1
:goto_1
if-eqz v6, :cond_2
move-object/from16 v0, p1
iget v0, v0, Landroid/util/TypedValue;->changingConfigurations:I
```

我们在i9100中发现了一段类似的代码：

```
:cond_1
:goto_1
if-eqz v7, :cond_2
move-object/from16 v0, p1
iget v0, v0, Landroid/util/TypedValue;->changingConfigurations:I
```

只不过v6变成了v7，说明这段代码检测v7的值，因此我们需要将我们插入的代码改为：

```
invoke-virtual/range {p0..p2}
Landroid/content/res/Resources;->loadOverlayDrawable(Landroid/util/TypedValue;I)Landroid
/graphics/drawable/Drawable;
move-result-object v7
if-nez v7, :cond_1
```

## 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 ActivityThread\$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{
        void 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

        .line 6
        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

```

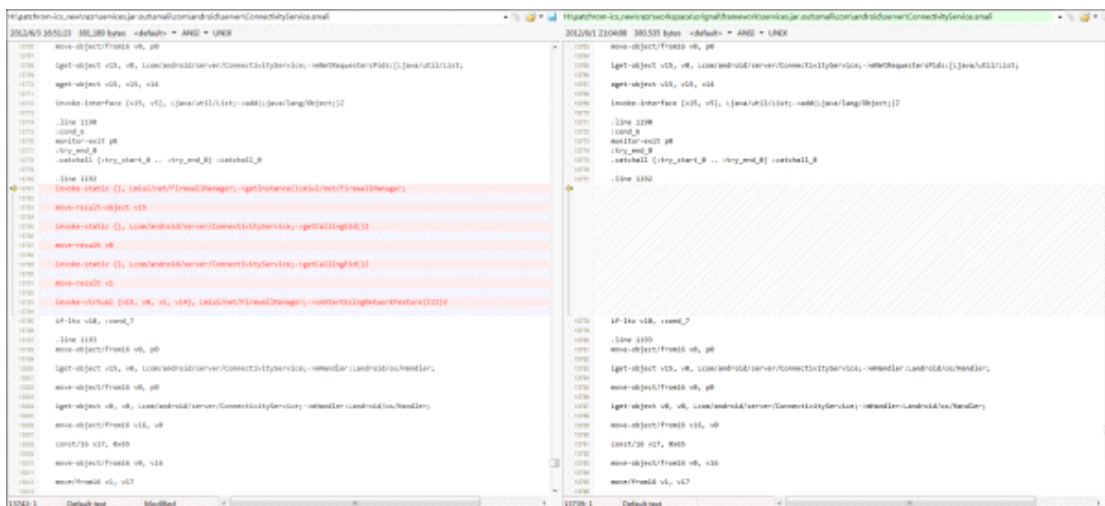
可以看到,编译器自动合成一个access\$000方法,假如当我们在一个较复杂的内部类中加入一个对外部类私有方法的调用,虽然只是导致新合成了一个方法,但是这些合成的方法名可能都会有变化,这样的结果就是smali文件的差异性比较大。

比如说对于KeyguardViewMediator.java文件,我们在其中的mBroadcastReceiver的定义最后加了一行代码adjustStatusBarLocked(),但是最后生成的smali文件却差别比较大。这个正式由于编译器为这个私有方法的调用,虽然只是导致新和成了一个方法,但是这些合成的方法命名被打乱,这种情况下我们修改代码无需作这么大的改动,自己为这个私有方法合成一个和其他合成方法不冲突的名字,具体的做法可以参照i9100中的android.policy.jar.out中对这个文件smali代码的修改。

## 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。

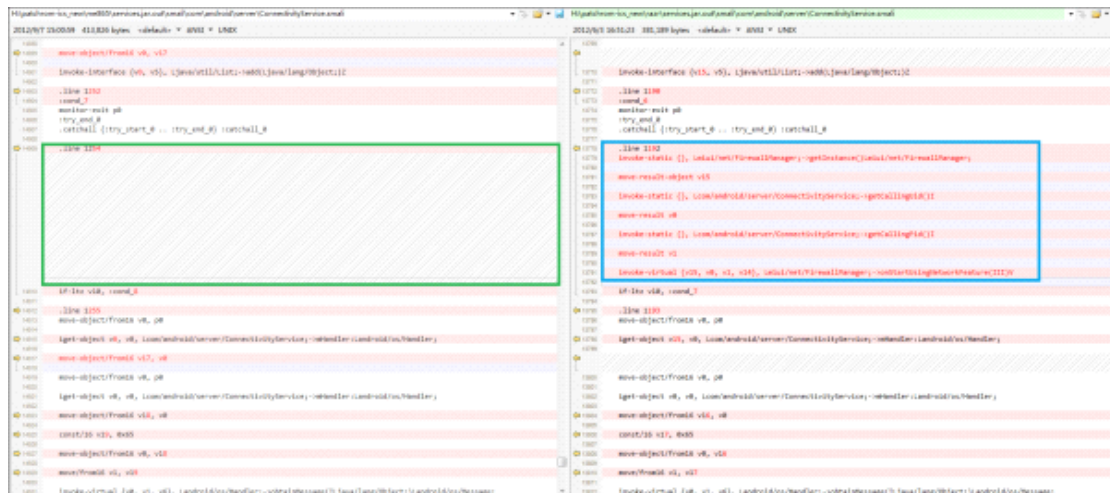
如果带注入的 smali 代码是从 PhoneWindowManager\$5.smali 提取的，一般不能够直接将其注入到目标机型的 PhoneWindowManager\$5.smali 文件中，因为不同机型的匿名内部类顺序不同，实现不同，smali 文件也不同。一般需要通过逐个比较 PhoneWindowManager\$5.smali 附近的几个文件的 smali 代码，看看其函数调用，函数名字，类继承关系是否相同来确定注入哪个文件。

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：



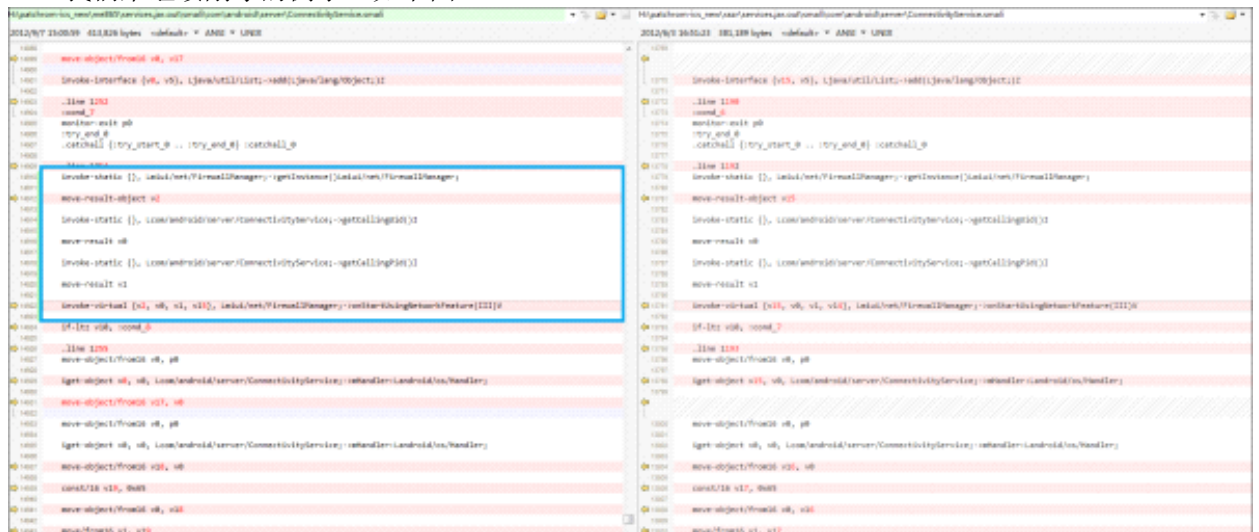


图中有很多红色的不同，其中蓝框是我们刚才确定的需要注入的samli代码。通过上下文匹配，可以发现绿框的位置是samli代码需要注入的区域。尽管上下有很多指令和变量不同，但是这并不影响我们的工作。

### 5.3 注入代码

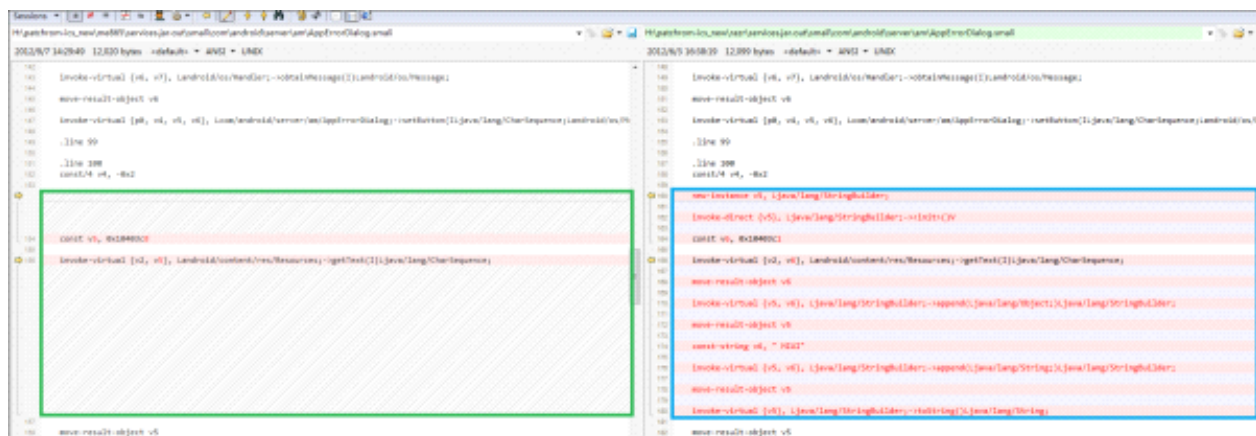
首先，将待注入的smali代码注入对应的区域。其次，对注入的Smali代码进行“本地化”——修改变量、跳转标号、逻辑判断标号等，使之符合当前的Smali代码实现，完成“嫁接”工作。当然，如果情况很复杂，需要重写对应的Smali代码或者重构java源代码，来完成最终的代码注入。

我们来继续刚才的例子，如下图

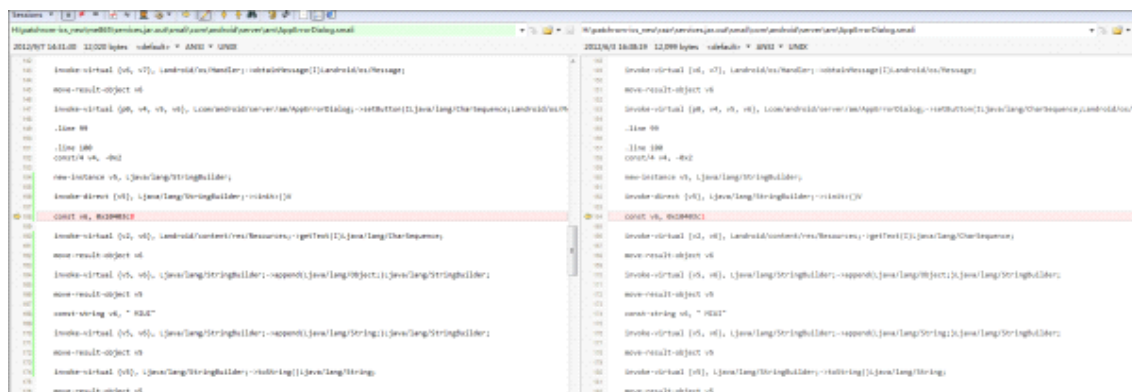


其中蓝框内是最终移植的代码，可以看到其中修改了move-result-object v2和invoke-virtual {v2, v0, v1, v15}, Lmiui/net/FirewallManager;→onStartUsingNetworkFeature(III)V 的变量，这主要是因为 invoke-virtual {v2, v0, v1, v15}, Lmiui/net/FirewallManager;→onStartUsingNetworkFeature(III)V在新的Smali代码中v15变量有其他的用途，我们需要找一个上下文无关的变量完成函数调用时的变量传递，所以这里将move-result-object v15改为move-result-object v2。并且修改了onStartUsingNetworkFeature()函数参数列表。

另外，移植中还有一类关于资源 I D 的代码注入比较特殊，这里举例说明：



现在我们需要将蓝框内的代码注入到绿框的位置，但是其中const v6, 0x10403c1阻挡了前进的步伐。我们不能鲁莽的将蓝框代码合入绿框，这样会导致资源无法找到运行时错误。我们需要使用反汇编原始ROM的framework-res.apk和目标ROM的framework-res.apk，根据0x10403c1ID值找到原始ROMframework-res.apk中对应的资源名字，再根据资源名字到目标ROM的framework-res.apk中查找对应的资源ID值。而后将其替换为目标ROM中的资源ID值。所以最终移植后的代码如下图：



需要说明的是 0x1 开头的资源都是 framework-res.apk 中的资源，0x2 开头的一般是厂商自己的资源，例如摩托的是 moto-res.apk, HTC 的是 com.htc.resources.apk。miui 自己的资源是 0x6 开头，位于 framework-miui-res.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`。

- ◆ 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。

## 5.5 调试 smali 代码

调试 smali 代码主要任务是解决注入代码后导致的运行时错误。具体的说，就是使注入后的 smali 代码通过 `dalvik` 虚拟机的字节码校验。获取错误的方法相对简单，使用下面两条命令即可：

```
$ adb logcat | grep dalvikvm
```

```
$ adb logcat | grep VFY
```

其中 VFY 的信息会给出 smali 代码出错的文件、函数以及出错的原因，`dalvikvm` 的信息可以给出调用栈，以及上下文执行过程，都比较实用贴心。

## 5.6 调试 smali 问题以及追踪方法

- ◆ 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。
- ◆ 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-director` 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。
- ◆ 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。

- ◆ 签名不正确。可以通过 `adb logcat | grep mismatch` 命令确认哪个 `package` 签名不正确。只需对签名不正确的包重新签名即可。当然如果有很多签名不一致的错误，建议大家对所有的 APK 重新签名。
- ◆ 资源找不到。这个问题的原因有很多种，这里列举一些常见的原因：
  - A: 系统资源文件签名不正确，导致没有加载系统资源，进而无法找到相应的资源。其中，系统资源文件是指 `system/framework` 目录下的 `apk` 文件。
  - B: `smali` 代码中的资源 ID 移植错误，无法在系统资源中找到对应的资源。
  - C: 资源相关的类移植存在问题，导致无法加载相关资源。

另外，在调试时，我们可能需要追踪代码执行路径，但是苦于没有 Debug。下面我们就来说一些简单的追踪方法。

- ◆ 增加简单的 `smali` 日志信息：
  - A. 修改函数的 `.local` 变量，在原来基础上增加两个变量，例如 `v11`, `v12`。
  - B. 在需要打印日志的地方增加如下 `smali` 代码
 

```
const-string v11, " @@@@@"
const-string v12, " interceptPowerKeyDown enter"
invoke-static {v11,v12}, Landroid/util/Log;:->e(Ljava/lang/String;Ljava/lang/String;)I
```

 如果增加的变量为 `v28` 和 `v29`，则需要使用下面的语句。
 

```
invoke-static/range {v28..v29}, Landroid/util/Log;:->e(Ljava/lang/String;Ljava/lang/String;)I
```

- ◆ 打印程序调用栈的方法：
  - A. 修改函数的 `.local` 变量，在原来基础上增加一个变量，例如 `v11`。
  - B. 在需要打印调用栈的地方增加如下 `smali` 代码：
 

```
new-instance v1 Ljava/lang/Exception;
invoke-direct {v1, Ljava/lang/Exception;:-<init>V
invoke-virtual {v1, Ljava/lang/Exception;:->printStackTrace()V
```

## 6.建议

最后，对修改 `smali` 代码给出一些意见：

- ◆ 细心，仔细的定位插入代码在相应机型代码中的插入位置。
- ◆ 要注意局部变量序号的改变。
- ◆ 不要一次修改完所有的文件再用 `apktool` 重新编译，如果插入代码有错误，会无法编译。但是 `apktool` 的编译出错信息是天书，你无从知道是哪个文件改错了。
- ◆ 出现错误不要紧，检查错误原因，检查的方法还有常见错误的解决方法，我们在上面都已经说过了。修改 `smali` 代码没那么难，多实践一定会掌握相应的技巧。

# 中兴 U950 的适配过程

我们以中兴 U950 为例，简单介绍从开始的基础环境搭建之后到初步适配后的整个过程。

现在我们手上已经下载了官方底包，我们首先需要将官方 ROM 做一些必要的处理。

把官方 ROM 里的 `/system/app` 和 `/system/framework` 下的文件需要经过 deodex 化，将官方 ROM 里的以一个 .apk 文件配一个 .odex 文件（或者 .jar 配一个 .odex 文件）形式存在的文件合并成单独的一个 .apk 文件（或者 .jar）文件，只有经过 deodex 化的 ROM 可以进行反编译，否则是不行的。进行 deodex 化的方法有很多：

1. 可以在 linux 环境下直接使用 patchtom 的 tools 里自带的 deodex.sh 这个脚本（不过有时候我遇到过一点问题）对整个官方包 deodex 化并生成一个完整的 deodex 过的刷机包
2. 采用其它 windows/linux 下的小工具，单独对 `/system/app` 和 `/system/framework` 这两个文件夹进 deodex 化后再替换回官方包得到一个刷机包。这里有一些需要稍微注意的问题：如果你采取方法

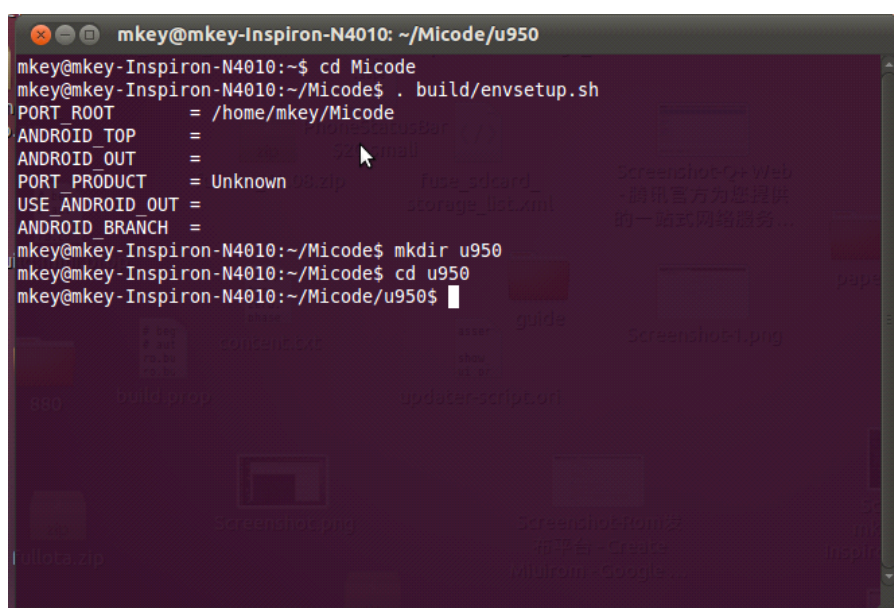
2. 部分存在于 `/system/app` 下的 apk 文件本身可能本身就只是以单独的一个 apk 文件存在的，换句话说就是不存在同名的 .odex 文件，这样的文件经过统一的 deodex 化后可能会提示 deodex 失败（因为小工具没有找到对应的 .odex 文件，就认为这个 .apk 文件是不可靠的），生成的 deodexed-app 文件夹里就不会有这些本身就不需要 deodex 化的 apk 文件，这时候我们可能需要手动把它们补回去。`/system/framework` 下的 `framework-res.apk` 也会有这种情况。

OK，上面的过程进行完毕后，我们还需要做一些其他的小工作：开启 boot 的 adb 调试并给我们的刷机包加上 root 权限。其中开启 boot 的 adb 调试总的来说是一个必须要做的工作，因为我们在适配过程中植入 smali 后重新编译回去刷写进机器一般是不能直接开机的（或者说其实基本不可能），我们需要在开机过程中实时查看 logcat 抛出的日志信息，对代码进行相应的修正，如果没有 log 输出，我们基本是没有办法进行调试的。这里开启的方法一般而言不过是解开 boot，修改 `default.prop` 里的相应属性，有时候需要添加 `persist.sys.usb.config=adb` 这一句话就可以了，不过也不能一概而论，有时候甚至需要修改 `init.rc` 里的相应服务或者和 `build.prop` 里的相应属性配合添加才能成功把 adb 调试

打开。那么怎么确认我们成功开启了 boot 的远程 adb 调试功能呢？其实很简单，把修改后的包含修改了 boot 的刷机包（或者直接刷写 boot 也可以），将手机关机，再用 adb 调试桥，将手机插在电脑上，如果在开机过程中 adb logcat 能成功输出 Android 系统的加载日志（包括各种软件和硬件信息），就说明 adb 调试已经成功打开了，否则我们还要多次尝试。只有将 adb 调试打开后，我们的适配工作才能真正开始进行。至于给刷机包本身加 root 权限，这个资料很多这里不再赘述。

现在我们假设上面的工作我们都已经搞定，现在我们就需要将我们修改过后的刷机包刷进我们的机器，这里建议在适配之前最好这个机型已经有现成的第三方 recovery，当然也可以自己动手做一个，不过不是我们这里介绍的主要内容，就不多说了。现在我们就可以正式开始我们的工作。

打开控制终端，进入目的文件夹，执行环境搭建脚本，建立 u950 文件夹

A terminal window titled 'mkey@mkey-Inspiron-N4010: ~/Micode/u950'. The terminal shows the following commands and output:

```
mkey@mkey-Inspiron-N4010:~$ cd Micode
mkey@mkey-Inspiron-N4010:~/Micode$ . build/envsetup.sh
PORT_ROOT      = /home/mkey/Micode
ANDROID_TOP     = 
ANDROID_OUT     = 
PORT_PRODUCT    = Unknown
USE_ANDROID_OUT = 
ANDROID_BRANCH  = 
mkey@mkey-Inspiron-N4010:~/Micode$ mkdir u950
mkey@mkey-Inspiron-N4010:~/Micode$ cd u950
mkey@mkey-Inspiron-N4010:~/Micode/u950$
```

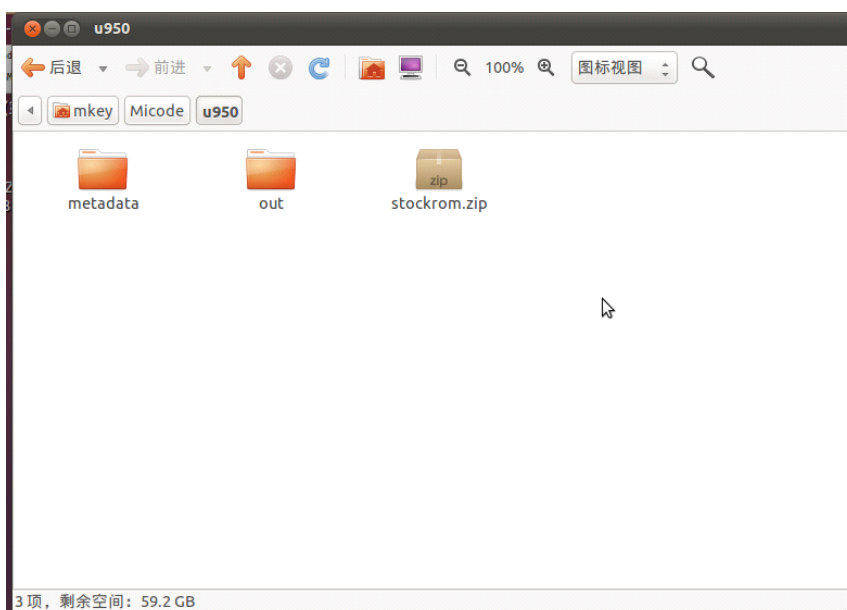
将机器进入 recovery 模式（第三方 recovery 且 recovery 本身开启了 adb 调试）接上 USB，执行命令

```
../tools/releasetools/ota_target_from_phone -r
```

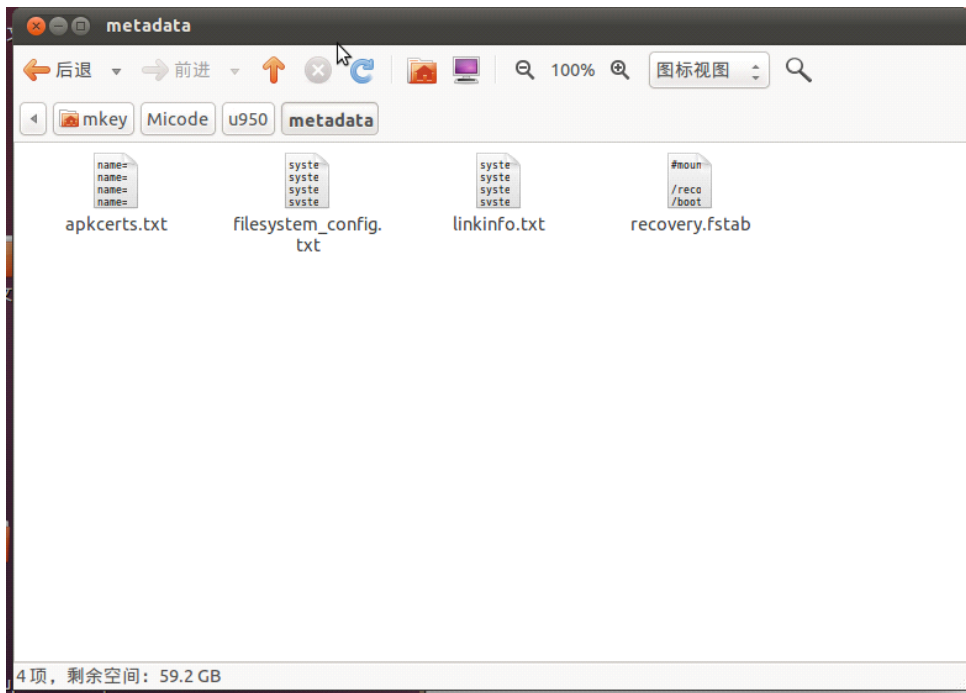


```
mkey@mkey-Inspiron-N4010: ~/Micode/u950
mkey@mkey-Inspiron-N4010:~$ cd Micode
mkey@mkey-Inspiron-N4010:~/Micode$ . build/envsetup.sh
PORT_ROOT      = /home/mkey/Micode
ANDROID_TOP    =
ANDROID_OUT    =
PORT_PRODUCT   = Unknown
USE_ANDROID_OUT =
ANDROID_BRANCH =
mkey@mkey-Inspiron-N4010:~/Micode$ mkdir u950
mkey@mkey-Inspiron-N4010:~/Micode$ cd u950
mkey@mkey-Inspiron-N4010:~/Micode/u950$ ../tools/releasetools/ota_target_from_ph
one -r
```

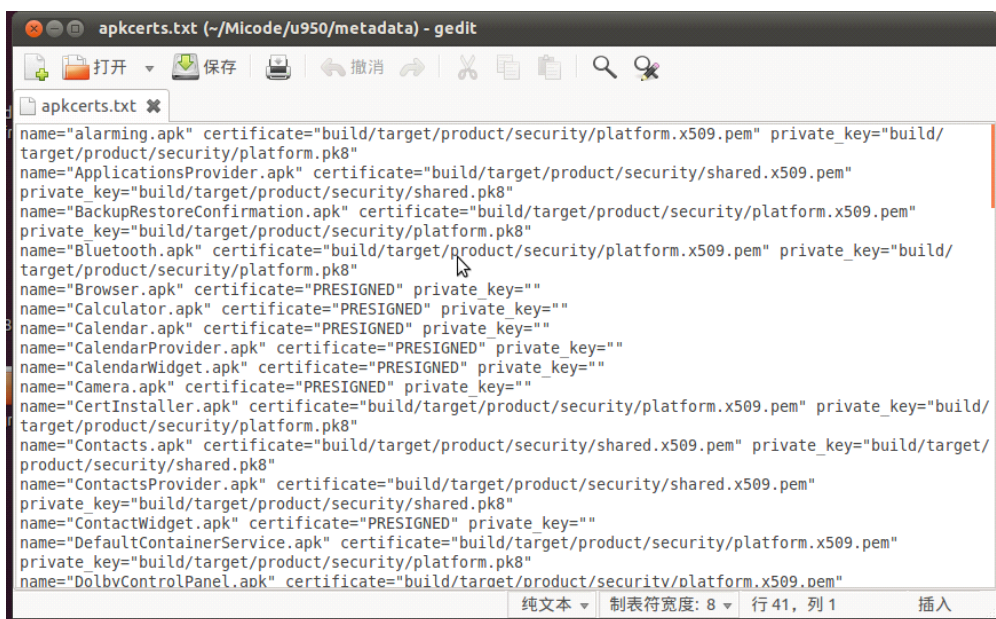
此条命令的意图在于从 recovery 模式下利用挂载手机各个分区的办法，人为提取出来一个官方 ROM 的 zip 包。提取过程需要执行一段时间，如果中途报错，尝试重新执行 `. Build/envsetup.sh` 或者将 adb 调试赋予管理员权限，再重新执行命令一般就可以了。提取完成后的目录组织结构入下图：



stockrom.zip 就是我们提取出来的官方 ROM。有人肯定要问，为什么要通过这种方法把官方包刷进去再用 recovery 重新提取出来，这不是多此一举么。其实不是的，我们可以打开 metadata 文件夹：



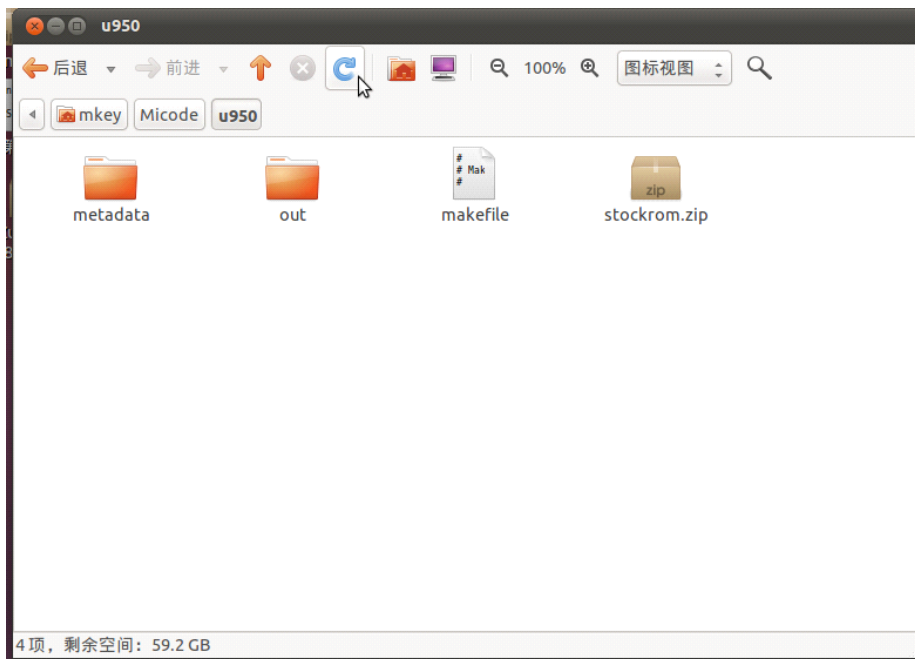
打开 apkcerts.txt 文件:



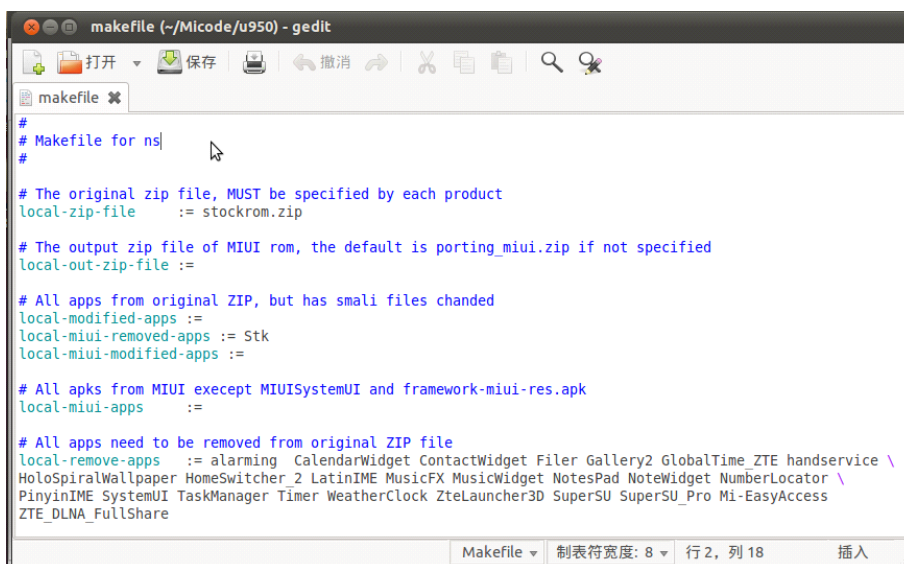
我们可以通过观察发现里面的内容实际上是记录了从刷机包里提取的 apk 文件的签名信息，只有有了这些签名信息，我们在最后打包的时候才能正确利用 miui 提供的签名密钥对所有 apk 重新进行签名（签名在 build/security 下），而 recovery.fstab 里就提取了系统分区表信息，其他两个文件里是一些链接信息和权限信息。这些都是必要的，而不通过这种方法是没办法获取这些信息的。



接下来我们需要写一个 makefile 配置文件，模板在 /android/ 下，里面有一个初步的 makefile 文件模板，我们可以复制到我们的工作目录下：



我们打开 makefile 文件做初步的配置（很多配置可以根据需要后期自行添加）

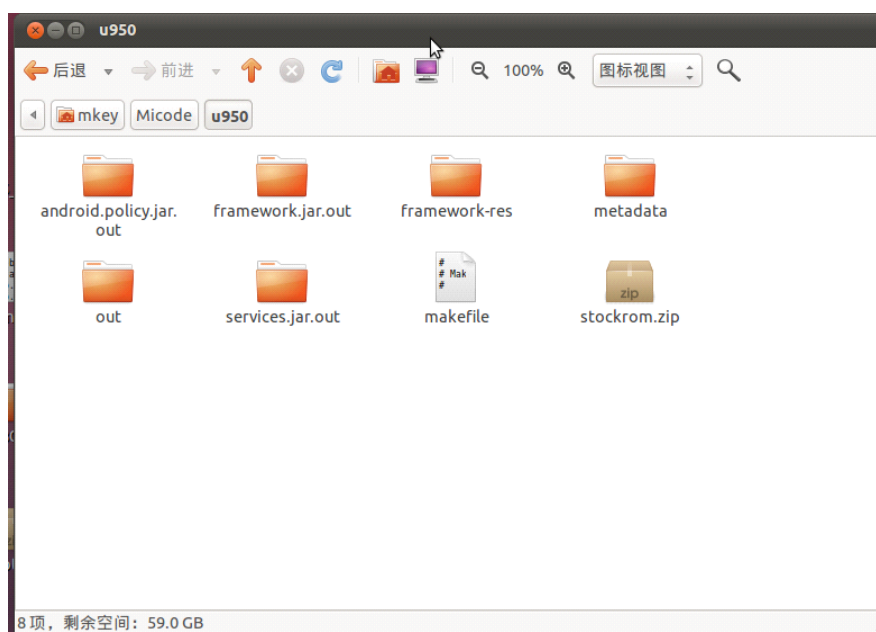


我们可以从它们的名字里看出来对应的意义：如 local-miui-removed-apps 就是指定不添加 miui 提供的 Stk.apk，local-remove-apps 指定的是从我们提取的 stockrom.zip 里面去除的 apk，这些 apk 在打包后不会加入我们编译生成的 miui 包里。

接下来开始反编译/编译的工作：我们在工作目录执行 make workspace 命令

```
mkey@mkey-Inspiron-N4010: ~/Micode/u950
mkey@mkey-Inspiron-N4010:~/Micode/u950$ make workspace
/home/mkey/Micode/build/util.mk:193: 警告：覆盖关于目标“stockrom.zip”的命令
/home/mkey/Micode/build/porting.mk:328: 警告：忽略关于目标“stockrom.zip”的旧命令
>>> Install framework resources for apktool...
install framework-miui-res.apk
/home/mkey/Micode/tools/apktool --quiet if /home/mkey/Micode/miui/system/framework/framework-miui-res.apk
unzip >/dev/null stockrom.zip "system/framework/*.apk" -d out
install out/system/framework/framework-res.apk
<<< install framework resources completed!
unzip >/dev/null stockrom.zip system/framework/services.jar -d out
/home/mkey/Micode/tools/apktool --quiet d -f out/system/framework/services.jar services.jar.out
unzip >/dev/null stockrom.zip system/framework/android.policy.jar -d out
/home/mkey/Micode/tools/apktool --quiet d -f out/system/framework/android.policy.jar android.policy.jar.out
unzip >/dev/null stockrom.zip system/framework/framework.jar -d out
/home/mkey/Micode/tools/apktool --quiet d -f out/system/framework/framework.jar framework.jar.out
if unzip >/dev/null stockrom.zip system/framework/framework-res.apk -d out 2>/dev/null; then /home/mkey/Micode/tools/apktool --quiet d -f out/system/framework/framework-res.apk framework-res ; else echo system/framework/framework-res.apk does not exist, ignored!; fi
fix the apktool multiple position substitution bug
```

我们再观察一下工作目录：



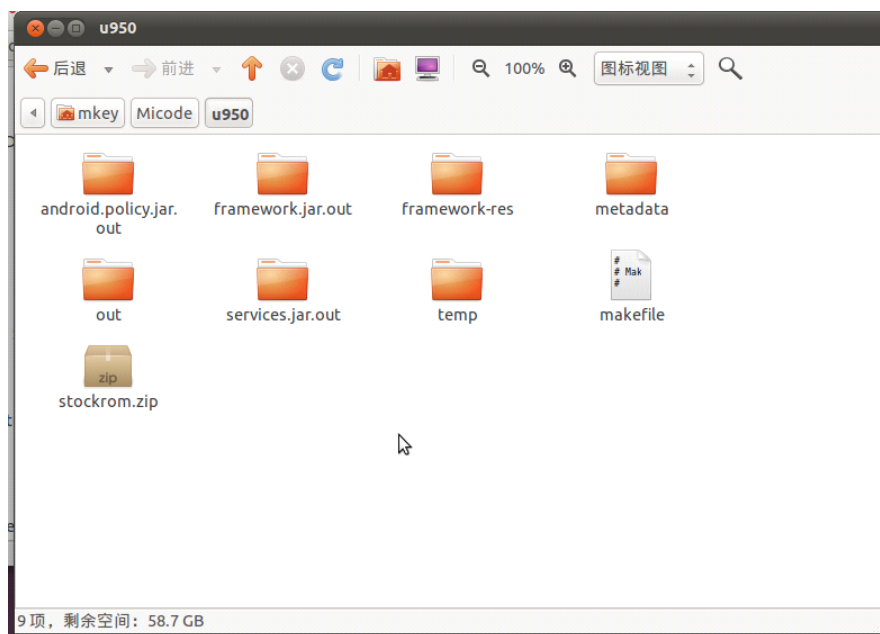
我们可以发现，这个过程实际上就是把原厂 ROM 里的 android.policy.jar, framework.jar, services.jar 这三个 jar 反编译成为 smali 码，便于我们的 smali 代码植入，framework-res.apk 也同样被解开，实际上是为了在其中插入 miui 的资源文件。因此我们其实可以认为，miui 的过程中就是修改

android.policy.jar, framework.jar, services.jar 三个 jar 包，并加入对应的资源。

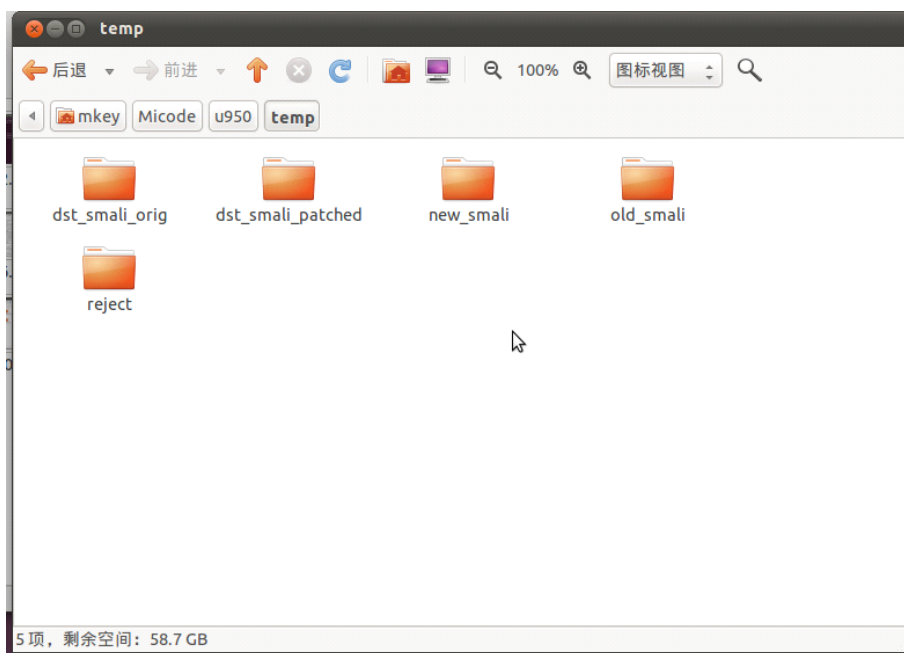
接下来执行 `make firstpatch` 命令（如果提示 `aapt` 错误, 把 miui 提供的 `aapt` 加入环境变量即可），这个过程是 miui 利用自动化脚本自动插入和修改 `smali` 的过程，对象就是 `services`, `android.policy`, `framework` 里的 `smali`，

这个过程需要持续十分钟到二十分钟。

这一步完成后，我们再次进入工作目录观察一下，我们可以发现多出了一个 `temp` 文件夹：



打开 `temp` 文件夹：



里面有五个文件夹：

old\_smali:里面的是标准的 google 代码的 smali 形式

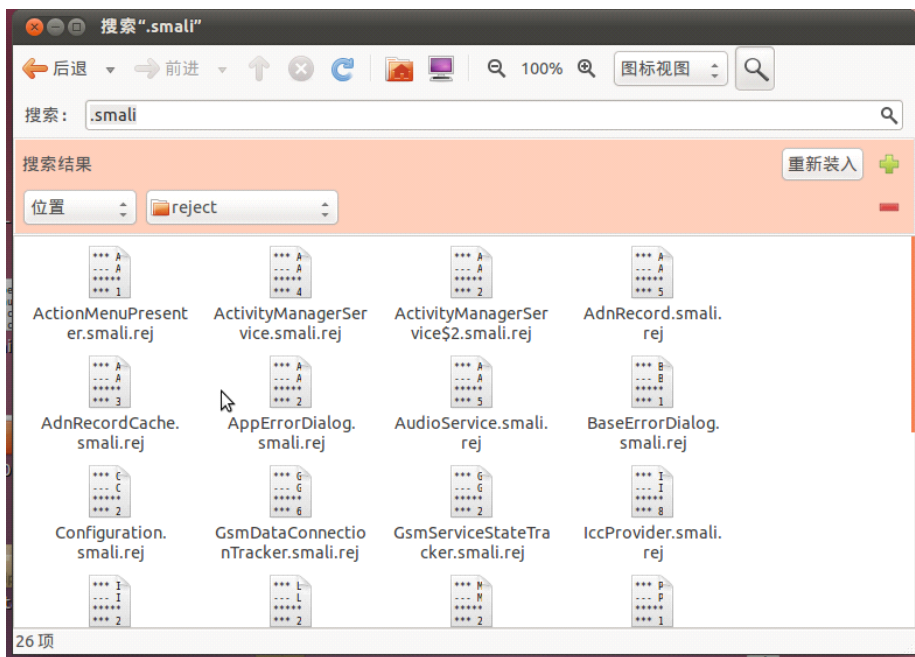
new\_smali:里面的是标准的 google 代码经过修改成为的标准 miui 代码的 smali 形式

dst\_smali\_orig: 里面的代码是我们的目标机型的 smali 代码

dst\_smali\_patched:里面的代码是我们的目标机型的 smali 代码经过 miui 自动插桩过后的 smali 代码（所谓插桩就是 make firstpatch 这个过程中工具参照 old\_smali 和 new\_smali 之间的差异，将这些差异应用到 dst\_smali\_orig 文件夹上的过程）

reject: 工具将 old\_smali 和 new\_smali 之间的差异应用到 dst\_smali\_orig 文件夹上的过程中因为某些原因无法应用成功的部分。

我们重点关注的部分就是 reject 部分，我们需要参照 old\_smali 和 new\_smali 之间的差异，手动地将这些自动化工具没能将其应用的改动手动进行改动：



我们打开 reject 文件夹中的 ServerThread.smali.rej 观察一下其中的内容



\*\*\* 470,480 \*\*\*

```

invoke-static {v3, v9}, Landroid/util/Slog; -> i(Ljava/lang/String;Ljava/lang/String;)I
!new-instance v63, Lcom/android/server/LightsService;
    move-object/from16 v0, v63
!invoke-direct {v0, v4}, Lcom/android/server/LightsService; -> <init>(Landroid/content/Context;)V
    .catch Ljava/lang/RuntimeException; {:try_start_5 .. :try_end_5} :catch_1
---
```

```

        :try_end_5

        .catch Ljava/lang/RuntimeException; {:try_start_5 .. :try_end_5} :catch_1

--- 470,480 ----

        invoke-static {v3, v9}, Landroid/util/Slog;->i(Ljava/lang/String;Ljava/lang/String;)I

        !new-instance v63, Lcom/android/server/MiuiLightsService;

        move-object/from16 v0, v63

        !invoke-direct {v0, v4}, Lcom/android/server/MiuiLightsService;-><init>(Landroid/content/Context;)V

        :try_end_5

        .catch Ljava/lang/RuntimeException; {:try_start_5 .. :try_end_5} :catch_1

```

这一部分的意思是在 old\_smali 的 ServerThread.smali 的某一行，有

```

        invoke-static {v3, v9}, Landroid/util/Slog;->i(Ljava/lang/String;Ljava/lang/String;)I

        ! new-instance v63, Lcom/android/server/LightsService;

        move-object/from16 v0, v63

        !invoke-direct {v0, v4}, Lcom/android/server/LightsService;-><init>(Landroid/content/Context;)V

        :try_end_5

        .catch Ljava/lang/RuntimeException; {:try_start_5 .. :try_end_5} :catch_1

```

这么一段代码，经过修改后成为 new\_smali 中的 ServerThread.smali 中的这一段：

```

        invoke-static {v3, v9}, Landroid/util/Slog;->i(Ljava/lang/String;Ljava/lang/String;)I

        !new-instance v63, Lcom/android/server/MiuiLightsService;

        move-object/from16 v0, v63

        !invoke-direct {v0, v4}, Lcom/android/server/MiuiLightsService;-><init>(Landroid/content/Context;)V

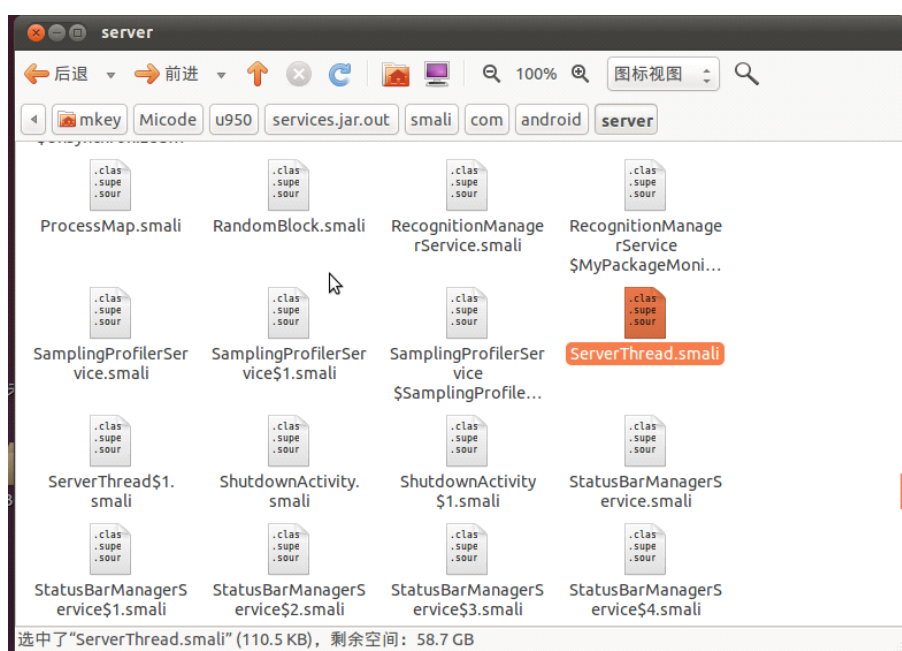
        :try_end_5

        .catch Ljava/lang/RuntimeException; {:try_start_5 .. :try_end_5} :catch_1

```

可是因为某种原因自动插桩工具没有将这一行插桩成功，我们就需要到工作目录下的 `/Micode/u950/services.jar.out/smali/com/android/server/ServerThread.smali` 里手动应用以上不能自动应用的改动。这里简单介绍一下，

\*.rej 文件中的 “!” 号代表这一行有改动，“+” 代表在此处添加了代码，同样的 “-” 代表这行代码被删除。



我们打开 `Micode/u950/services.jar.out/smali/com/android/server/ServerThread.smali`，经过查找，我们发现在这段代码：

```
new-instance v63, Lcom/android/server/LightsService;
move-object/from16 v0, v63
invoke-direct {v0, v4}, Lcom/android/server/LightsService;-><init>(Landroid/content/Context;)V
:try_end_7

.catch Ljava/lang/RuntimeException; {:try_start_7 .. :try_end_7} :catch_1
```

和上面的代码段进行对比，发现工具自动插桩失败的原因在于“:try\_end\_5”和“:try\_end\_7”之间的差别，因此工具认为该文件里没有找到需要变动的那段代码，因此这个所谓的“差异”就应用失败，而应用失败的部分就变成了.rej 文件。那我们对比后就可以把上述代码段改成：

```
new-instance v63, Lcom/android/server/MiuiLightsService;

move-object/from16 v0, v63

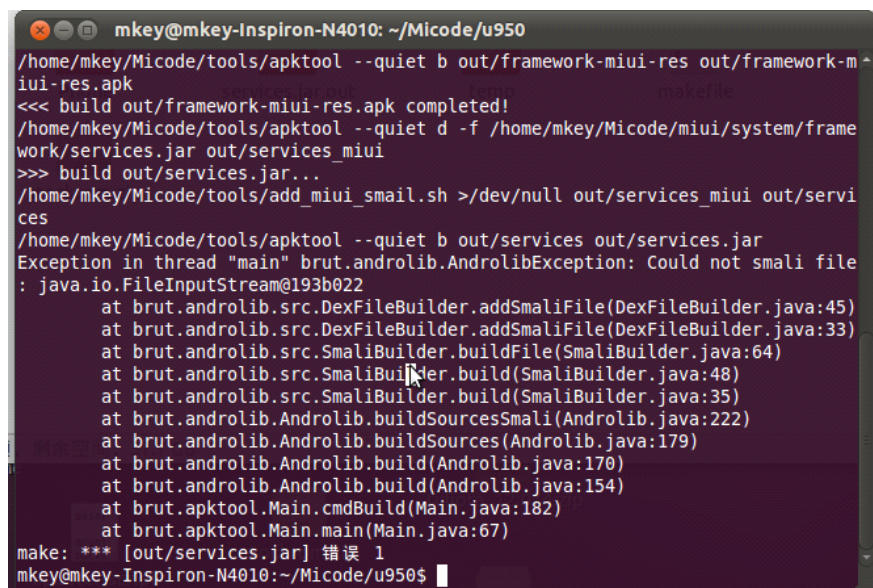
invoke-direct {v0, v4}, Lcom/android/server/MiuiLightsService;-><init>(Landroid/content/Context;
)V

:try_end_7

.catch Ljava/lang/RuntimeException; {:try_start_7 .. :try_end_7} :catch_1
```

这个 reject 就算被我们人工搞定了。我们接下来我们用同样的方法把剩下的 reject 一一手动插桩。

假设我们所有的 reject 都改好了，接下来我们要执行 make fullota 来执行最后的打包工作：



```
mkey@mkey-Inspiron-N4010: ~/Micode/u950
/home/mkey/Micode/tools/apktool --quiet b out/framework-miui-res out/framework-miui-res.apk
<<< build out/framework-miui-res.apk completed!
/home/mkey/Micode/tools/apktool --quiet d -f /home/mkey/Micode/miui/system/framework/services.jar out/services_miui
>>> build out/services.jar...
/home/mkey/Micode/tools/add_miui_smail.sh >/dev/null out/services_miui out/services
/home/mkey/Micode/tools/apktool --quiet b out/services out/services.jar
Exception in thread "main" brut.androlib.AndroLibException: Could not smali file
: java.io.FileInputStream@193b022
    at brut.androlib.src.DexFileBuilder.addSmaliFile(DexFileBuilder.java:45)
    at brut.androlib.src.DexFileBuilder.addSmaliFile(DexFileBuilder.java:33)
    at brut.androlib.src.SmaliBuilder.buildFile(SmaliBuilder.java:64)
    at brut.androlib.src.SmaliBuilder.build(SmaliBuilder.java:48)
    at brut.androlib.src.SmaliBuilder.build(SmaliBuilder.java:35)
    at brut.androlib.AndroLib.buildSourcesSmali(AndroLib.java:222)
    at brut.androlib.AndroLib.buildSources(AndroLib.java:179)
    at brut.androlib.AndroLib.build(AndroLib.java:170)
    at brut.androlib.AndroLib.build(AndroLib.java:154)
    at brut.apktool.Main.cmdBuild(Main.java:182)
    at brut.apktool.Main.main(Main.java:67)
make: *** [out/services.jar] 错误 1
mkey@mkey-Inspiron-N4010:~/Micode/u950$
```

很不幸，我们在重新打包的时候出错了，说明我们插桩后的 smali 代码有明显的错误，apktool 发现了不允许通过，因此报错。像以上的情况就是 services 部分的代码出错了，



可是遇到编译错误时，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。这里和大家分享一下我自己在适配过程中应对这类问题的一些小技巧。

如果修改完所有 reject 后不能顺利编译通过，比如 android.policy 编译不过的话，一般在 PhonewindowManager.smali 以及相关类出错的概率比较高，我们可以采取一些技巧：把原来的 smali，就是没 patch 的那几个替换进去，一次将可疑的出错 smali 文件用原来的 smali 文件替换一半，然后逐步缩小范围，最终可以锁定是哪一个 smali 文件出错导致编译无法通过的。framework 部分编译不通过一般是通信层的错误，如 Cdma/GsmDataConnectionTracker.smali 以及相关类，我们也可以通过同样的方法锁定。services 部分则稍微好一些。锁定是哪些 smali 文件导致编译不通过后我们可以对比 old\_smali 及 new\_smali 对应的改动手动进行插桩。