MIUI ROM 定制教程

序言

为了帮助广大的 MIUI 发烧友将 MIUI ROM 移植到自己所喜爱的机型上，MIUI 开发组创建并开源了 patchrom 项目，同时发布此配套教程。该教程主要探讨的是如何在原厂 ROM 的基础上定制出自己的 MIUI ROM（原厂 ROM 指的是由手机生厂商发布的官方 ROM，具有最好的稳定性），这也是我们项目名称的由来（对已有的 ROM 作修改，类似于对软件打 patch 的过程，因此称作 patchrom）。虽然本教程着重于如何定制 MIUI ROM，但是其中涉及到的技术和概念是通用的，适用于任何 ROM 的定制。

要完全掌握此教程，你需要有 Linux 操作系统的使用经验，了解 Java 语言，有一定的 Android 编程经验最好。如果只是要修改资源（比如汉化等），基本上不需要有任何编程知识。

本教程分成六个章节：

第一章：搭建开发环境，简要的介绍如何准备必要的开发环境。
第二章：认识 Android 手机，从 ROM 开发者的角度来看 Android 手机系统的结构。
第三章：寻找合适的原厂 ROM，patchrom 项目是基于原厂 ROM 进行修改的，这一章会介绍一些准则来判断什么是合适的 ROM。
第四章：反编译，介绍 apktool 工具和反编译的基础知识。
第五章：移植 MIUI Framework，讲述如何移植 MIUI 框架层的代码。
第六章：移植 MIUI App，讲述如何移植 MIUI 的应用程序。
第七章：制作 ZIP 刷机包，讲述如何制作一个 ZIP 刷机包。

patchrom 项目网址：https://github.com/MiCode/PatchRom
你可以访问该网站下载到 patchrom 的所有代码，包括本教程。

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，我们建议你下载一份 google 发布的 android 源代码，这不是必需的，但是对于理解排查 ROM 适配中的一些错误有很大帮助。

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 以上的版本。

2. 安装 Android SDK

本节简要介绍如何在 Ubuntu 系统上安装 Android SDK。Windows 和 Mac 用户请参照 http://developer.android.com/sdk/installing.html

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。

```
$ sudo add-apt-repository "deb http://archive.canonical.com/ lucid partner"
$ sudo apt-get update
$ sudo apt-get install sun-java6-jdk
```

2.2 下载 Android SDK 包

从以下地址下载 Android SDK 包 http://dl.google.com/android/android-sdk_r16-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. .bashrc 来使对 PATH 环境变量的修改生效。

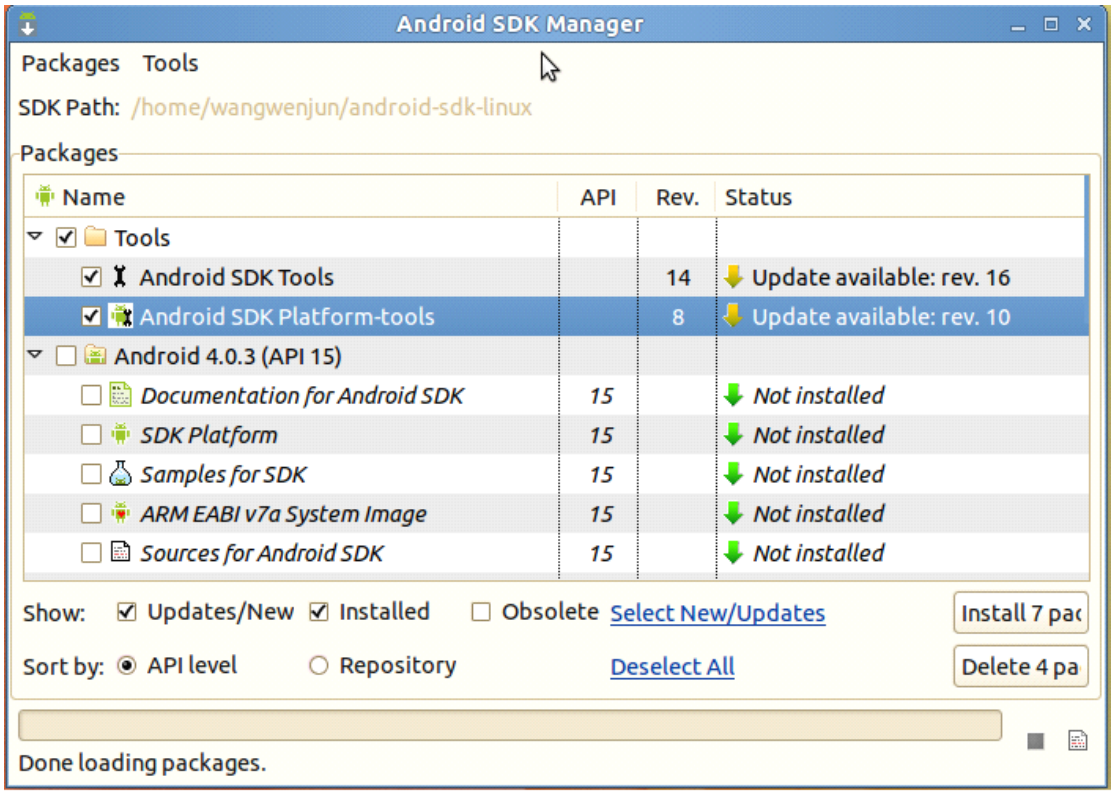
```
$ . ~/.bashrc
```

2.3 安装

运行命令 android 来启动 Android SDK Manager

$ android

启动结果如下图所示：



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，但是强烈建议你有 Android 程序开发基础。

2.4 adb

Android SDK 中对我们最重要的工具是 adb(android debug bridge)。在移植 MIUI ROM 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，同时也是验证上述的步骤是否成功，打开手机中的系统设置，选择应用程序—开发，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，如果显示和下面的信息类似，恭喜你，adb 可以识别你的手机了。

List of devices attached

304D1955996BE28E device

注意：

(1) 在 Windows 下，必须安装手机相应的驱动才能成功识别手机。

(2) 在 Ubuntu 下，有可能会提示“no such permissions”，这个时候有两种办法，第一种是以 root 的身份运行 adb。第二种办法：

a) 运行 lsusb 命令，对于我的三星手机，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. patchrom 项目

下面介绍 patchrom 的目录结构以及各目录的作用。

- android: 该目录下有 2 个子目录：system 和 src。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，我们只放 miui 修改过的文件，要下载完整的 android 源码，请参照 http://source.android.com/source/downloading.html。
  system 目录下存放的是由 google 发布的 android 源码编译而成的三个 jar 包:
  framework.jar, android.policy.jar 和 services.jar。这些 jar 包的作用在之后的章节会详细阐述。
- build: 该目录是一些与编译相关的脚本。
- tools: 该目录存放一些工具程序和脚本，在定制 ROM 和编译过程中需要使用这些程序。
- miui: 该目录下有 2 个子目录：system 和 src。system 目录下存放的是由 miui 源代码编译后的部分文件，这些文件是我们定制 MIUI ROM 所需要用到的所有文件。之后的章节会详细阐述。
- i9100: 针对每一个要定制的机型，创建一个单独的目录。该目录存放的是和三星 i9100 相关的一些修改和文件。之后的章节会以 i9100 为例详细阐述。

接下来我们开始编译生成 i9100 的定制 MIUI ROM，假定当前目录为 /home/patcher/patchrom 目录，

```
$ . build/envsetup.sh
$ cd i9100
$ make zipfile
```

以上命令运行完毕后，在 i9100 目录下会生成一个.build 子目录，该子目录下的 MIUI_9100.zip 文件即是我们发布的 i9100 刷机包。

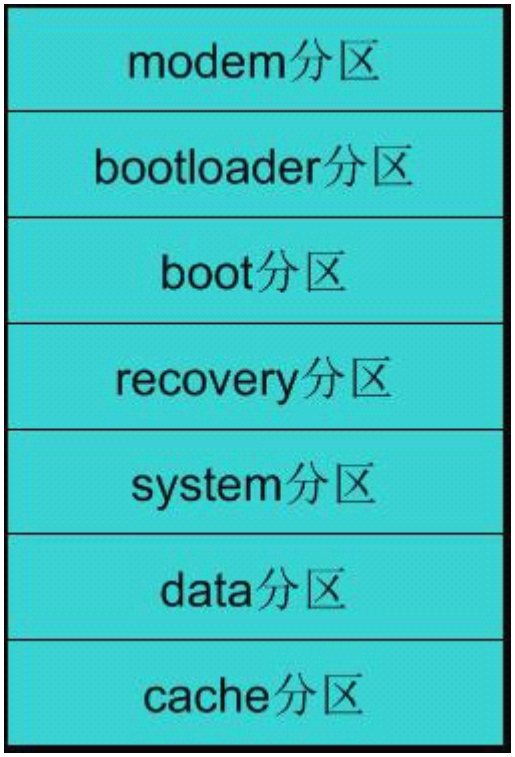
第二章 认识 Android 手机

写这篇文章时想起我的第一部 Android 手机 HTC Hero。买回来后，同事告诉我可以去刷机玩玩。刷机，怎么刷？同事说，你个土人，刷机都不知道，很多刷机论坛的，你去逛逛，挺简单的。我去逛了逛机锋论坛（那时还不知道 MIUI），打开一看，什么 recovery, radio, root 各种词汇扑面而来，oh my lady gaga，这么复杂。但是为了不被鄙视，而且闲着也是闲着，还是刷着玩玩吧。后来就结识 MIUI 来到了小米。

在这纷纷扰扰的 Android 世界里，如何找到那条刷机大道呢，或许它只是个传说，我们只是一直在探索。让我们从零开始来看一看，看能发现点什么。

1. bootloader

当我们拿到一款手机，第一件事应该就是按下电源键开机，那么从开机到进入到桌面程序这中间发生了些什么呢，我们从下面这张简化了的手机结构图开始：



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 分区等等。后面我们会逐渐介绍各个分区的用途。所谓的刷机我们可以简单的理解成把软件安装在手机的某些分区中，类似于我们在电脑上安

装 Windows 系统。

当按下电源键手机上电启动后，首先从 bootloader 分区中一个固定的地址开始执行指令，bootloader 分区分成两个部分，分别叫做 primary bootloader 和 secondary stage bootloader。Primary bootloader 主要执行硬件检测，确保硬件能正常工作后将 secondary stage bootloader 拷贝到内存(RAM)开始执行。Secondary stage bootloader 会进行一些硬件初始化工作，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。比如说大家所熟知的通过电源键和其它一些按键的组合，可以进入到 recovery，fastboot 或者选择启动模式的启动界面等。我们在论坛上看到的 bootloader 通常指的就是 secondary stage bootloader。不过我们不需要关心太多的细节，可以简单的理解为 bootloader 就是一段启动代码，根据用户按键有选择的进入某种启动模式。

fastboot 模式：fastboot 是 android 定义的一种简单的刷机协议，用户可以通过 fastboot 命令行工具来进行刷机。比如说 fastboot flash boot boot.img 这个命令就是把 boot.img 的内容刷写到 boot 分区中。一般的手机厂商不直接提供 fastboot 模式刷机，而是为了显示他们的牛 B 之处，总是会提供自己专有的刷机工具和刷机方法。比如说三星的 Odin，摩托的 RSD，华为的粉屏等等。但是其本质实际上是相同的，都是将软件直接 flash 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。一般的手机厂商都提供一个简单的 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 前提是 recovery 分区可以被刷写。大家在论坛上看到的解锁 bootloader，通常指的就是解锁 recovery 或 fastboot，允许刷写 recovery 分区，这样大家就可以用上喜爱的 CWM Recovery 了。

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需的通信功能，大家通常所的刷 RADIO 就是刷写 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，这里就不写了，而且我们在原厂 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。

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

#

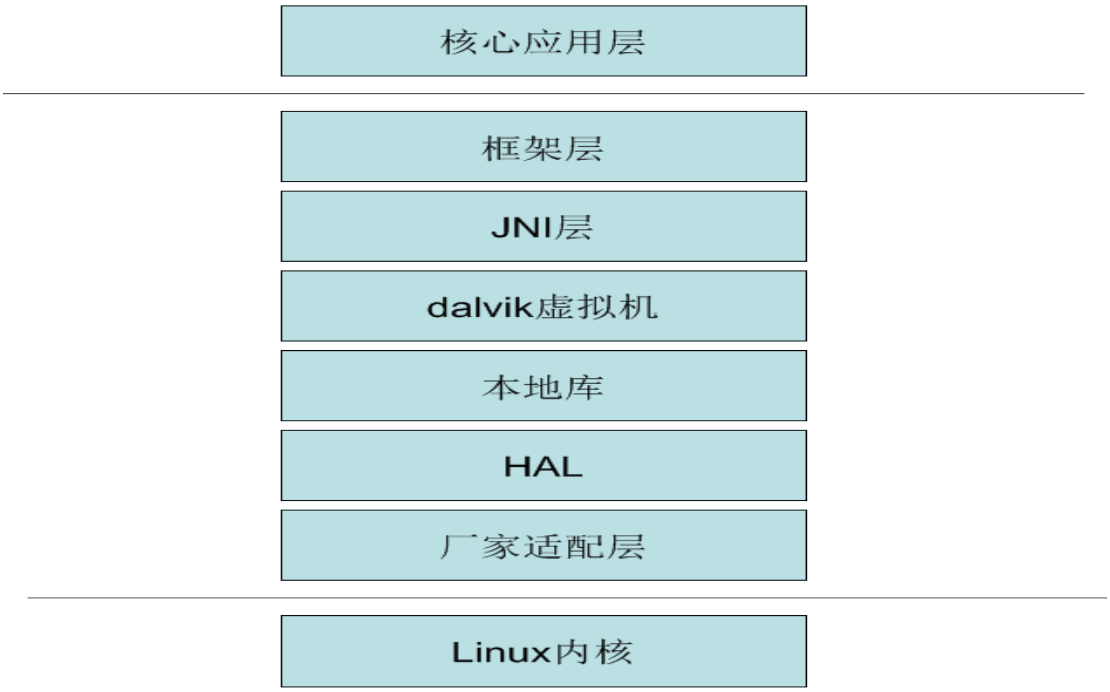
ADDITIONAL_DEFAULT_PROPERTIES
#
ro.secure=1
ro.allow.mock.location=1
ro.debuggable=0
persist.service.adb.enable=1。

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的两个属性：ro.secure 和 ro.debuggable。如果 ro.secure=0 则允许我们运行 adb root 命令。通常大家说得内核 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)可以授予程序 root 用户的权限。

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 分区的程序，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，我们先来看下面这张 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 虚拟机解释执行。

- 本地库：本地库一般是由 C/C++语言所开发，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工作，Android 定义了一套硬件接口，比如说为了使用相机，厂家的相机驱动必须提供的接口方法。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适配层的代码，这样在 HAL 层接口的实现只是一个简单的对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，内核中包含有各种硬件驱动，这些驱动不同的手机厂商不同的手机是不一样的。Linux 内核是支持驱动模块化机制的，简单的说就是允许用户动态的加载或者卸载某个硬件驱动，但是目前来看，手机厂商除了提供 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是用 RE 文件管理器）。

- system/lib: 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 MIUI 移植来说有 3 个最重要的 JAR 包(framework.jar, android.policy.jar, services.jar)。后续的文章会重点介绍这 3 个包。

- 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，比如说开机音乐，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。如何修改这些文件请参考网上对应机型形形色色的教程，这里不再赘叙。

- 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。其中有一个重要的命令叫 app_process 下一节单独介绍。

- system/xbin: 该目录下存放的是一些扩展的可执行文件，既该目录可以为空。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都是放在该目录。

- system/build.prop: build.prop 和上节说得根文件系统中的 default.prop 文件格式一样，都称为属性配置文件。它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 ROM 制作者都会修改一下 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（XXX 所修改）。我见过一个只是删了 system/app 的一些程序，然后修改 build.prop 中的 ro.build.display.id 和 ro.build.version.incremental 中的两个属性值打上自己的大名的 ROM。

- 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的有规律。一般来说，一些脚本程序，还有大家所熟悉 GPS 配置文件(gps.conf)和 APN 配置文件(apns-conf.xml)放在这个目录。像 HTC 将相机特效所使用的一些文件也放在这个目录下。

4. Zygote(app_process)

上一节提到 init 会执行一个重要的命令程序叫 app_process，一般大家称之为 Zygote。（Zygote 是卵的意思，所有的 Android 进程都是由它生出来的)。Zygote 首先会加载 dalvik 虚拟机，然后产生一个叫做 system_server 的进程。system_server 顾名思义被称作 Android 的系统服务程序，它主要管理整个 android 系统。system_server 启动完成后开始寻找一个叫做启动器的程序，找到之后由 zygote 开始启动执行启动器，这就是我们常见到的桌面程序。上面描述的是一个相当简化的启动过程，了解这些对于适配 MIUI 基本上就够了，如果大家对这些想进一步了解的话，请关注市面上各种 Android 内幕书籍。

5. 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，一般来说我们都会下载安装一些 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 和程序数据都丢失了。

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基本上放在这个分区。

6. 小结

本章主要是介绍了一下 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从整体上理解 ROM 移植。

第三章 寻找合适的原厂 ROM

1. 熟悉移植的机型

“千里之行，始于足下”。做移植之前，首先得熟悉我们要移植的机型。

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。在这里，不得不提到一个必逛的论坛：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所在机型的刷机方法，需要用到什么工具，多刷几个 ROM 玩玩，尽量熟悉刷机过程。

1.2 合适的原厂 ROM

在第一步熟悉了要移植的机型，刷了这大那大出的 ROM 后，接下来我们要开始集中精力寻找一个合适的原厂 ROM，因为一般来说，原厂 ROM 的稳定性最好。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（下面会详细介绍何为 ROOT 和 deodex）。

那么如何判断一个原厂 ROM 是否合适呢：

首先要版本合适，我们这个系列谈论的是基于原厂 ROM 移植 MIUI，目前 2.3 的 MIUI 是基于 android2.3.7 源码开发的，从 android2.3.3 到 android2.3.7 这几个版本变化都不太大，因此 2.3.3 到 2.3.7 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，表明是内核 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，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，定制 MIUI ROM 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。（之后不针对只有手机 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，不一定要求必需是 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 更好。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，我们在第七章中会介绍如何在手机现有系统的基础上制作 ZIP 包。

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 表示普通信息等）。

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)，表示程序所在的进程 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 之后，我们要确保在开机之初，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不打印任何信息，移植工作很难进行下去。如果只是简单的修改一些图片资源的话可以，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，如果我们能够修改 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 内核的办法，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，我们可以用下面的办法来修改它，但是如果不遵从，需要逛论坛详细的了解如何修改 boot 分区。有的机型遵从，比如说华为的 u8800，有的机型不遵从，比如说三星的 i9100, i9000 等。判断是否遵从有一个简单但不是一定准确的办法，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 内核。

$ mkdir ramdisk

$ cd ramdisk

$ gzip -dc ../boo.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。什么是 deodex？啊，这真的是一个 long long story。

话说 Android 发明之日起，准备让开发人员使用 Java 语言来在 Android 手机平台上进行应用程序开发（为什么用 Java, Java 程序员大喊道，谁用谁知道呀）。Java 程序一般使用 java 编译器(javac 命令）从源文件(.java 结尾的文件)编译成类文件(.class 结尾的文件，又被称作字节码)，然后很多类文件被打包成一个 JAR 包（JAR 包实际是一个 zip 压缩包），然后用 java 虚拟机解释执行这些类文件。采用类文件格式以及使用标准的 Java 虚拟机需要向 Java 的所有者（当时是 SUN 公司，后来被 Oracle 公司收购，默哀一下）缴纳授权费用并遵守相应的版权协议。Google 不想缴纳这笔费用并受协议的约束，于是这丫想出来一个“偷天换日，偷梁换柱”的方法，用的是 Java 的壳，但是那颗心已不是 Java 的心（正是因为 google 绕过了 Java 授权，所以现在 Oracle 紧咬着 google 打官司）。简单来说，就是当编译 Android 上的 Java 程序时，第一步还是编译成类文件打包成 JAR 包，然后会将这个 JAR 包转换为一个叫 classes.dex 的文件，这个 dex 文件是什么玩意呢，这是 google 发明出来的一个用于它自己的虚拟机上的一个字节码文件格式。Android 上得这个虚拟机就叫做 dalvik 虚拟机（dalvik 是冰岛的一个小镇的名字，当时 google 的工程师在给这个虚拟机苦思冥想一个名字，后来一个主要的工程师 Dan Bronstein 我的祖先当初生活在冰岛的这个小镇，就以它命名吧。据说 Dan 本人从没去过冰岛，Android 发布后，冰岛很是骄傲，当地的报纸专门登载了这件事并热切欢迎 Dan 回乡探亲）。那么 odex 是啥呢，它叫 optimized dex，即优化过的 dex 文件。

讲了这么多，你只需要理解 odex 是一种优化过的 dex 文件就行了，至于怎么优化的不在我们讲述的范围内。odex 文件是互相依赖的，简单的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 文件变为 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 脚本来自动的做 deodex 操作。

Windows 用户可以下载 http://www.xeudoxus.com/android/xUltimate-v2.3.3.zip 这个工具来做 deodex，具体的用法请参照该工具的帮助文档。

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 的时候基本上不改这个文件。这里帮助你有个大致的了解。

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，和 Windows 下的窗口类似。AndroidManifest.xml 是一个 xml 格式的清单文件，就像你去超市买东西会打印出一个购物清单，AndroidManifest.xml 也起着一个清单的作用，它告诉系统，我有这些 Activity。（实际情况远比这复杂，想学 Android 编程的同学请看这个 http://developer.android.com/guide/index.html，好好学习其中的内容）。

具体到 HelloActivity 下的 AndroidManifest.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>
    </application>
```

其中有一行<category android:name="android.intent.category.LAUNCHER" />，包含这一行的 Activity 会显示在桌面中，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，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-hdpi, drawable-mdpi, drawable-ldpi 等，这些 hdpi,mdpi,lpid 分别表示高/中/低分辨率，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 给出颜色值的定义，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 HellloActivity 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/strings.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，将每一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在编回 apk 文件，运行如下命令：

```
$ 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，可以换一个名字或者放在另外一个目录下。

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失败，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，对的，现在显示在你面前的是“你好，世界！”

汉化成功了，是的，汉化就这么简单。如果你只想停留在汉化或者替换图片这个阶段，从这里开始以后的文章不用看了。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，为什么这样替换一下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，我们就的大致的了解一下资源的编译过程。首先我们看看 values 目录下一个有意思的文件 publics.xml，它的内容如下：

```
<?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，这个数字是干嘛的呢？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识符，这个标识符分成 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 下。/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，依次增加。中间的一个字节表示资源的类型，每一个类型的数字标识符是不一样的，最后的 2 个字节是资源的序号，统一类型的资源序号从 0 依次往上递增。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 publics.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 publics.xml 里，这样可以保证替换资源后资源的 id 不会变化。

为啥资源的 id 这么重要，如果变了，会怎么样呢，这得结合代码理解。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。资源编译器会生成一个查找表，对于每一个 id，查找表中保存了这个 id 对应的名字和值（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文件。

最后说一句，资源的 ID 非常重要，运行 adb pull /system/framework/framework-res.apk 反编译这个文件，好好的消化一下这一节的内容吧。

4. smali

终于迎来我们最重要的部分了 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 代码可读性差太多了，但是和传统的 x86 或者其他体系结构下的汇编文件那又是好读多了。虽然有工具可以直接把这些反汇编成 java 代码，但是好不了太多，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 组织源代码的方式一样，smali 目录下的文件也是按文件包的包名结构组织目录结构的，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, txtView:Landroid/widget/TextView;
    const/high16 v1, 0x7f04

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

    .line 40
    return-void
.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 表示对应的源代码的行号，这个对调试很重要。.metho 和.end method 表示一个方法定义的开始和结束。Smali 文件中的这些指令的功能请参照 http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html，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
txtView.setText(R.string.hello_activity_text_text)

我们现在想将这行代码改成 txtView.setText("Happy, Cracker!")，将.line 39 到.line 40 行的

代码改为：

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

再按照上一节所说得重新编译，签名，安装运行，好了，现在出现在你面前的是 Happy, Cracker! 了，真 happy！

接下来的两章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。

第五章 移植 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，在没有源代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。但是对于下面几层的代码几乎都是以机器码的形式存在，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，大家有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 代码达到移植 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得了。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，简单的整个替换 MIUI 框架层和核心应用层的代码无法工作。

2. 移植规范

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

android 有两个子目录 src 和 system，src 目录是 google 发布的 android2.3.7 的部分源码。

system 目录下放的是由源码编译而成的三个 jar 包。在移植的时候不要修改这个目录。src/frameworks/base/services 中的文件属于 services.jar 包，src/frameworks/base/policy 中的文件属于 android.policy.jar 包，src/frameworks/base 中的其它文件属于 framework.jar 包。

framework.jar 是 android 运行的基础框架，比如每个 APP 需要用到的 Actitity 的实现等。services.jar 是 android 中的一些服务的实现，这些服务基本上都运行在 system_server 进程中。android.policy.jar 是手机策略以及锁屏的实现。

2.2 miui

miui 下也是有两个子目录 src 和 system。其中 src/frameworks/base 子目录和 android 中的 src/frameworks/base 子目录是一对一的关系，放的是 miui 修改过的代码。src/frameworks/miui 目录下放的是 miui 对原生资源的修改，有直接替换的图片，还有一些 xml 文件的改动。

system 分为四个子目录，目录结构映射手机中的 system 目录，也就是说这些目录中的文件最终会放到手机相应的目录中：

- system/framework：存放由 miui 源码编译而成的 android.policy.jar, framework.jar, services.jar 以及 miui 专属资源包 framework-miui-res.apk。
- system/app：存放 miui 核心应用
- system/media：一些锁屏，主题相关文件。
- system/xbin：busybox 程序和 miui 专属的 invoke-as 程序。

在移植的时候不要修改 miui 目录。

2.3 i9100

i9100 是我们会移植三星 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 ROM 刷机包，比如 I9100ZCKJ1.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，framework-res，LogsProvider，MediaProvider，Phone，Settings。这些目录相应的是从刷机包中 system/framework/android.policy.jar，system/framework/framework.jar，system/framework/services.jar，system/app/LogsProvider.apk，system/app/MediaProvider.apk，system/app/Phone.apk，system/app/Settings.apk 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。比如 SystemUI 目录修改了 MIUISystemUI.apk 反编译代码中的两个 smali 文件。之所以这样组织是因为基于原厂 ROM 文件修改的，初始代码不再变动，只是后续的修改。而基于 MIUI 文件修改的，MIUI 自身的源代码不断变化，相应的整个反编译的代码也不断变化。因此只存放我们修改过的 smali 文件。
- 有一个 makefile，i9100 中的 makefile 给出了详细的注释，请参考该 makfile 写其它机型的 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。

- misc 目录：存放一些在打包过程中需要的文件，在 makefile 里面将该目录的文件放到刷机包中。

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，挑选几个有代表性的讲解，剩下的大家可以自己去做。

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, miui 和 i9100 这三个目录中，分别反编译其中的 framework.jar 文件，在这三个目录中会产生 framework.jar.out 目录。

为了方便比较差异，我们建议在 android, miui, i9100 中分别建一个目录 noline，先将 framework.jar.out 目录复制到这个目录中，然后使用 patchorom/tools 中的脚本 rmline.sh 运行如下命令，假定在 patchrom 目录下：

$ tools/rmline.sh miui/noline/framework.jar.out

rmline.sh 是用以把 smali 所有以.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 logcat 的异常错误信息会给出在出错代码的行号，这样方便我们定位错误。

接下来用大家说熟悉的文件比较工具来比较差异，Linux 下推荐 meld, Windows 下推荐 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 表示这个方法使用 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，在编译器的术语这叫做基本块。对于这种新增的代码，我们找出它的上下文，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 Resources.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/graphics/drawable/Drawable;
move-result-object v6
if-nez v6, :cond_1

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

红色代码是 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的，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:cond_6 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-eqz 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，我们去 i9100 中找到对应的代码逻辑。对于这个例子，我们完全可以以".xml"作为一个关键字去 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，我们来看看:cond_1 的代码是

在干嘛？: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
```

只不过是 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，都会产生一个单独的 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，编译器自动合成了一个 access$000 方法，假如当我们在一个较复杂的内部类中加入了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 文件差异较大，

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。这个正是由于编译器为这个私有方法合成了一个方法导致所有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它合成方法不冲突的名字，具体的做法可以参照 i9100 中 android.policy.jar.out 中对这个文件 smali 代码的修改。

5. 建议

最后想对修改 smali 代码给出一些建议：

(1) 细心，仔细的定位插入代码在相应机型代码中的插入位置。

(2) 要注意局部变量序号的改变。

(3) 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。

(4) 出现错误不要紧，检查 adb logcat 的错误信息，找出错误发生的原因。修改 smali 代码没那么难，多实践一定会掌握相应的技巧。

第六章 移植 MIUI APP

所有需要移植的 MIUI APP 放在 patchrom/miui/system/app 目录下。移植 APP 相对简单，直接把对应的 APK 放到 system 分区 app 目录下即可。但是可惜的是，有一些限制导致我们需要做一些额外的工作。本节就具体的描述移植 APP 中可能遇到的一些问题。

1. MIUI APP 一览

APK	说明
Contacts.apk	联系人程序
ContactsProvider.apk	联系人数据库
Mms.apk	短信程序
TelephonyProvider.apk	短信相关数据库
TelocationProvider.apk	来电归属地相关数据库
Music.apk	音乐程序
Notes.apk	便签程序
MIUISystemUI.apk	系统通知栏程序
ThemeManager.apk	主题管理器程序
Updater.apk	系统更新程序
Torch.apk	手电筒程序
Launcher2.apk	启动器程序
DownloadProvider.apk	下载管理相关数据库
DownloadProviderUi.apk	下载管理程序

2. 一个遗憾：打电话程序

打电话程序一般是位于 system/app 下的 Phone.apk。在上面的列表中，我们并没有看到 MIUI 的 Phone.apk，为什么呢？

与一般的 APK 不同，Phone.apk 必须访问 Android 的私有 API。Android 的公有 API 对所有的第三方开发人员是一样的，即 Android SDK 开放的 API。这也是为什么目前市场上有很多第三方的联系人，短信等程序，但是确没有打电话程序。这些私有 API 的实现和接口各个不同的机型有可能不一样，而且更复杂的是，虽然有的时候接口一样，但是语义不一样。(Phone.apk 需要与 RIL 层通信，RIL 层全称为 radio interface layer，这一层又是对底层 RADIO 通信的一个封装，Phone.apk 通过发送称之为 RIL Command 的一些命令和 RIL 层通信，不同架构的机型这些命令有差别)。

理论上说，MIUI 的 Phone.apk 完全可以移植到新的机型，只是移植难度大一些。在使用由 framework 层提供的私有 API 的地方，要保证使用目标机型 framework 层所提供的私有 API。要了解目标机型支持的 RIL Command 以及相应的语义，使用目标机型所定义的 RIL 通信命令。这些需要对 Phone 的源代码很熟悉，要仔细研究目标机型 Phone.apk 反编译后的 smali 代码。

由于上述的一些限制，在项目初期我们做了一个艰难的决定，不移植 Phone.apk。

3. 一个遗憾引发的问题

容易理解的是，打电话程序访问联系人信息，当拨打或接听电话时，显示号码所对应的联系人，在相应的联系人下面存取通话记录等。Android 系统为这些操作定义了一套接口。如果目标机型的 Phone.apk 只是使用这些接口访问联系人信息，那么我们无需对 Phone.apk 做任何修改。但是可惜的是，现实往往没有那么简单。

举个例子，在移植 i9100 时我们发现在联系人程序里没有通话记录信息，Android 定义了一个类 CallLog，调用其中的 allCall 方法来保存通话记录信息。通过反编译 Phone 的代码，我们发现 i9100 使用了一个专有的 LogsProvider 来存取通话记录，完全没有调用 allCall 方法。通过修改 Phone 中的 CallLogAsync$AddCallTask.smali 文件，调用 allCall 来保存通话记录信息。

之后又陆续碰到一些电话相关的问题，比如拨打接听电话无法显示联系人，点击通知栏的未接电话通知无法进入拨号界面等。有一些问题比较好解决，可以在 adb logcat 中看到异常信息或错误信息。有一些问题没有错误提示，比如说接听电话无法显示联系人，只显示来电号码等。这个时候就需要从问题回溯找出原因，当接听电话的时候，系统中一定有一个方法能够通过来电号码查找联系人，那我们就需要找到这个方法，然后看看这个方法是否有问题。

总而言之，电话和联系人程序的结合是比较紧密的，而我们在移植过程中发现的种种问题也基本上是由于电话和联系人的一些通信接口导致的，这个时候我们就需要细心的逐个排查这些问题。

4. 系统通知栏

在开发 MIUI APP 时，我们严格遵循一条规则：不使用 framework 私有的资源。这个原因在之前我们介绍 android 资源内部机制的时候有涉及过，framework 资源是放在 framework-res.apk 中。对于私有资源，不同的 framework-res.apk 这些私有资源的 ID 一般也是不同的。但是系统通知栏是其中的一个特列，系统通知栏必须访问某些私有资源 ID，这个情况怎么办呢。我们在开发系统通知栏程序的时候也考虑到了这一点，把所有对私有资源的访问封装在一个类中，位于 patchrom/miui/system/app/MIUISystemUI/ResourceMapper.java。

```
public final class ResourceMapper {
    private static SparseIntArray sMapping;
    static {
        sMapping = new SparseIntArray();
```

```
            sMapping.put(com.android.internal.R.drawable.stat_sys_battery,
com.miui.internal.R.drawable.stat_sys_battery);
            sMapping.put(com.android.internal.R.drawable.stat_sys_battery_charge,
com.miui.internal.R.drawable.stat_sys_battery_charge);
            sMapping.put(com.android.internal.R.drawable.stat_sys_battery_unknown,
com.miui.internal.R.drawable.stat_sys_battery_unknown);
            sMapping.put(com.android.internal.R.layout.status_bar_latest_event_content,
com.miui.internal.R.layout.status_bar_latest_event_content);
        }

        public static int get(int resid) {
            int result = sMapping.get(resid);
            return result != 0 ? result : resid;
        }
    }
```

这个类的作用很简单，就是就对私有资源的访问完全封装起来，这样使得系统通知栏程序的其它代码都不直接访问私有资源。为了移植系统通知栏，我们需要先反编译 MIUISystemUI.apk，修改反编译后的 ResourceMapper.smali 文件，即将 com.android.internal.R.drawable.stat_sys_battery 的资源 ID 修改成在目标机型中对应的资源 ID。(如何获取目标机型中对应的资源 ID，可以查看 framework-res/res/values/publics.xml 中该资源名称对应的 ID)。

5. 其它程序

其它程序基本上直接放到 system 分区的 app 目录下即可使用。启动器程序可能需要做一些小的修改，这些修改基本上都比较容易，都是从问题回溯，找到出现问题的根源。

第七章 制作刷机包

刷机包即是我们通常在 recovery 模式下安装的 zip 包，刷机包的原理非常简单易懂。本章就讲述刷机包的原理，掌握了这些之后，制作刷机包就不难了。

1. 刷机包结构

以 patchrom/i9100/I9100ZCKJ1.zip 为例，我们先来看看它的目录结构：

```
I9100ZCKJ1/
    |----------data/
    |----------system/
    |----------flash_image
    |----------modem.bin
```

```
|----------zImage
|----------META-INF/
                |-------------CERT.RSA
                |-------------CERT.SF
                |-------------MANIFEST.MF
                |-------------com/google/android
                                        |---------------update-binary
                                        |---------------updater-script
```

一个刷机包必不可少的部分就是 META-INF 目录，其中的 CERT.RSA, CERT.SF 和 MANIFEST.MF 这三个文件是与签名相关的。com/google/android 目录下存放了两个文件：update-binary 和 updater-script。updater-script 是一个刷机包的核心，它是一个脚本文件，它由一系列的命令所在组成，每一个命令以分号结束。update-binary 则是一个脚本解释器。大家可以认为 updater-srcript 就是一个指挥中枢，发布了一系列的命令。update-binary 则是一个忠实的执行者，它按顺序的逐个执行指挥中枢发布的命令。

updater-script 的所涉及到的命令语法和语义的详细描述请参照 http://www.freeyourandroid.com/guide/introdution_to_edify。

2. updater-srcript 例解

本节以 I9100ZCKJ1.zip 中完整的 updater-script 文件为例，辅以注释说明 updater-srcript 文件的各个命令，使得用户可以快速的掌握刷机包的制作。

```
ui_print(" ");
ui_print("================================================");
ui_print("=                  Welcome to                  =");
ui_print("=              Samsung Galaxy S II             =");
ui_print("=              MIUI GT-I9100 ROM               =");
ui_print("================================================");
ui_print("=                    MIUI                      =");
ui_print("================================================");
```

连续的打印命令，在屏幕上逐行显示指定的字符串。

```
show_progress(0.500000, 0);
```

显示当前的安装进度完成了 50%的增长。

```
ui_print("Formatting Partitions");
unmount("/system");
unmount("/cache");
unmount("/data");
format("ext4", "EMMC", "/dev/block/mmcblk0p9");
```

format("ext4", "EMMC", "/dev/block/mmcblk0p7");
format("ext4", "EMMC", "/dev/block/mmcblk0p10");
mount("ext4", "EMMC", "/dev/block/mmcblk0p9", "/system");
mount("ext4", "EMMC", "/dev/block/mmcblk0p7", "/cache");
mount("ext4", "EMMC", "/dev/block/mmcblk0p10", "/data");

在屏幕上显示 Formating Partitions，对于 system, cache, data 分区，分别执行卸载，格式化，挂载操作。这几条命令的作用就是清空了 system, cache, data 分区中的所有内容，然后把这些分区挂载在/system, /cache 和/data 目录下。是否格式化 system 分区是我们判断一个刷机包是否完整包或者升级包的依据，完整包意味着刷写整个 system 分区，升级包意味着只是更新 system 分区的某些文件。但是从技术角度来说，没有必要区分完整包和升级包，它们的内在结构是完全一致的，都是由 updater-script 决定的。

ui_print("Extracting /system");
package_extract_dir("system", "/system");
ui_print("Extracting /data");
package_extract_dir("data", "/data");

打印提示信息，将刷机包中 system 和 data 目录的内容完整的拷贝到/system 和/data 目录，这两个 package_extract_dir 命令真正完成刷写 system 和 data 分区。

ui_print("Symlinking");
symlink("busybox", "/system/xbin/[", "/system/xbin/[[", "/system/xbin/acpid", "/system/xbin/addgroup", "/system/xbin/adduser", "/system/xbin/adjtimex", "/system/xbin/ar", "/system/xbin/arp", "/system/xbin/arping", "/system/xbin/ash", "/system/xbin/awk", "/system/xbin/basename", "/system/xbin/bbconfig", "/system/xbin/beep", "/system/xbin/blkid", "/system/xbin/bootchartd", "/system/xbin/brctl", "/system/xbin/bunzip2", "/system/xbin/bzcat", "/system/xbin/bzip2", "/system/xbin/cal", "/system/xbin/catv", "/system/xbin/chat", "/system/xbin/chattr", "/system/xbin/chgrp", "/system/xbin/chpasswd", "/system/xbin/chpst", "/system/xbin/chroot", "/system/xbin/chrt", "/system/xbin/chvt", "/system/xbin/cksum", "/system/xbin/clear", "/system/xbin/comm", "/system/xbin/conspy", "/system/xbin/cp", "/system/xbin/cpio", "/system/xbin/crond", "/system/xbin/crontab", "/system/xbin/cryptpw", "/system/xbin/cttyhack", "/system/xbin/cut", "/system/xbin/dc", "/system/xbin/deallocvt", "/system/xbin/delgroup", "/system/xbin/deluser", "/system/xbin/depmod", "/system/xbin/devmem", "/system/xbin/dhcprelay", "/system/xbin/diff", "/system/xbin/dirname", "/system/xbin/dnsd", "/system/xbin/dnsdomainname", "/system/xbin/dos2unix", "/system/xbin/dpkg", "/system/xbin/dpkg-deb", "/system/xbin/du", "/system/xbin/dumpkmap", "/system/xbin/dumpleases", "/system/xbin/echo", "/system/xbin/ed", "/system/xbin/egrep", "/system/xbin/eject", "/system/xbin/env", "/system/xbin/envdir", "/system/xbin/envuidgid", "/system/xbin/ether-wake", "/system/xbin/expand", "/system/xbin/expr", "/system/xbin/fakeidentd", "/system/xbin/false", "/system/xbin/fbset", "/system/xbin/fbsplash", "/system/xbin/fdflush", "/system/xbin/fdformat", "/system/xbin/fdisk", "/system/xbin/fgconsole", "/system/xbin/fgrep", "/system/xbin/find", "/system/xbin/findfs", "/system/xbin/flash_eraseall", "/system/xbin/flash_lock", "/system/xbin/flash_unlock", "/system/xbin/flashcp", "/system/xbin/flock", "/system/xbin/fold", "/system/xbin/free", "/system/xbin/freeramdisk", "/system/xbin/fsck", "/system/xbin/fsck.minix", "/system/xbin/fsync", "/system/xbin/ftpd",

"/system/xbin/ftpget", "/system/xbin/ftpput", "/system/xbin/fuser", "/system/xbin/getopt", "/system/xbin/getty", "/system/xbin/grep", "/system/xbin/gunzip", "/system/xbin/halt", "/system/xbin/hdparm", "/system/xbin/head", "/system/xbin/hexdump", "/system/xbin/hostid", "/system/xbin/hostname", "/system/xbin/httpd", "/system/xbin/hush", "/system/xbin/hwclock", "/system/xbin/ifdown", "/system/xbin/ifenslave", "/system/xbin/ifplugd", "/system/xbin/ifup", "/system/xbin/inetd", "/system/xbin/init", "/system/xbin/inotifyd", "/system/xbin/install", "/system/xbin/ipaddr", "/system/xbin/ipcalc", "/system/xbin/ipcrm", "/system/xbin/ipcs", "/system/xbin/iplink", "/system/xbin/iproute", "/system/xbin/iprule", "/system/xbin/iptunnel", "/system/xbin/kbd_mode", "/system/xbin/killall", "/system/xbin/killall5", "/system/xbin/klogd", "/system/xbin/last", "/system/xbin/length", "/system/xbin/less", "/system/xbin/linux32", "/system/xbin/linux64", "/system/xbin/linuxrc", "/system/xbin/loadfont", "/system/xbin/loadkmap", "/system/xbin/logger", "/system/xbin/login", "/system/xbin/logname", "/system/xbin/logread", "/system/xbin/losetup", "/system/xbin/lpd", "/system/xbin/lpq", "/system/xbin/lpr", "/system/xbin/lsattr", "/system/xbin/lspci", "/system/xbin/lsusb", "/system/xbin/lzcat", "/system/xbin/lzma", "/system/xbin/lzop", "/system/xbin/lzopcat", "/system/xbin/makedevs", "/system/xbin/makemime", "/system/xbin/man", "/system/xbin/md5sum", "/system/xbin/mdev", "/system/xbin/mesg", "/system/xbin/microcom", "/system/xbin/mkdosfs", "/system/xbin/mke2fs", "/system/xbin/mkfifo", "/system/xbin/mkfs.ext2", "/system/xbin/mkfs.minix", "/system/xbin/mkfs.reiser", "/system/xbin/mkfs.vfat", "/system/xbin/mknod", "/system/xbin/mkpasswd", "/system/xbin/mkswap", "/system/xbin/mktemp", "/system/xbin/modinfo", "/system/xbin/modprobe", "/system/xbin/more", "/system/xbin/mountpoint", "/system/xbin/mt", "/system/xbin/nameif", "/system/xbin/nc", "/system/xbin/nice", "/system/xbin/nmeter", "/system/xbin/nohup", "/system/xbin/nslookup", "/system/xbin/ntpd", "/system/xbin/od", "/system/xbin/openvt", "/system/xbin/passwd", "/system/xbin/patch", "/system/xbin/pgrep", "/system/xbin/pidof", "/system/xbin/ping6", "/system/xbin/pipe_progress", "/system/xbin/pivot_root", "/system/xbin/pkill", "/system/xbin/popmaildir", "/system/xbin/poweroff", "/system/xbin/printf", "/system/xbin/pscan", "/system/xbin/pwd", "/system/xbin/raidautorun", "/system/xbin/rdate", "/system/xbin/rdev", "/system/xbin/readlink", "/system/xbin/readprofile", "/system/xbin/realpath", "/system/xbin/reformime", "/system/xbin/reset", "/system/xbin/resize", "/system/xbin/rev", "/system/xbin/rpm", "/system/xbin/rpm2cpio", "/system/xbin/rtcwake", "/system/xbin/run-parts", "/system/xbin/runlevel", "/system/xbin/runsv", "/system/xbin/runsvdir", "/system/xbin/rx", "/system/xbin/script", "/system/xbin/scriptreplay", "/system/xbin/sed", "/system/xbin/sendmail", "/system/xbin/seq", "/system/xbin/setarch", "/system/xbin/setfont", "/system/xbin/setkeycodes", "/system/xbin/setlogcons", "/system/xbin/setsid", "/system/xbin/setuidgid", "/system/xbin/sha1sum", "/system/xbin/sha256sum", "/system/xbin/sha512sum", "/system/xbin/showkey", "/system/xbin/slattach", "/system/xbin/smemcap", "/system/xbin/softlimit", "/system/xbin/sort", "/system/xbin/split", "/system/xbin/start-stop-daemon", "/system/xbin/stat", "/system/xbin/strings", "/system/xbin/stty", "/system/xbin/sulogin", "/system/xbin/sum", "/system/xbin/sv", "/system/xbin/svlogd", "/system/xbin/swapoff", "/system/xbin/swapon", "/system/xbin/switch_root", "/system/xbin/sysctl", "/system/xbin/syslogd", "/system/xbin/tac", "/system/xbin/tail", "/system/xbin/tar", "/system/xbin/taskset", "/system/xbin/tcpsvd", "/system/xbin/tee", "/system/xbin/telnet", "/system/xbin/telnetd", "/system/xbin/test", "/system/xbin/tftp",

```
"/system/xbin/tftpd", "/system/xbin/time", "/system/xbin/timeout", "/system/xbin/touch",
"/system/xbin/tr", "/system/xbin/traceroute", "/system/xbin/traceroute6", "/system/xbin/true",
"/system/xbin/tty", "/system/xbin/ttysize", "/system/xbin/tunctl", "/system/xbin/tune2fs",
"/system/xbin/ubiattach", "/system/xbin/ubidetach", "/system/xbin/udhcpc",
"/system/xbin/udhcpd", "/system/xbin/udpsvd", "/system/xbin/uname",
"/system/xbin/uncompress", "/system/xbin/unexpand", "/system/xbin/uniq",
"/system/xbin/unix2dos", "/system/xbin/unlzma", "/system/xbin/unlzop", "/system/xbin/unxz",
"/system/xbin/unzip", "/system/xbin/uptime", "/system/xbin/usleep", "/system/xbin/uudecode",
"/system/xbin/uuencode", "/system/xbin/vconfig", "/system/xbin/vi", "/system/xbin/vlock",
"/system/xbin/volname", "/system/xbin/wall", "/system/xbin/watch", "/system/xbin/watchdog",
"/system/xbin/wc", "/system/xbin/wget", "/system/xbin/which", "/system/xbin/who",
"/system/xbin/whoami", "/system/xbin/xargs", "/system/xbin/xz", "/system/xbin/xzcat",
"/system/xbin/yes", "/system/xbin/zcat", "/system/xbin/zcip");
    symlink("toolbox", "/system/bin/cat","/system/bin/chmod",
        "/system/bin/chown",
        "/system/bin/cmp","/system/bin/date",
        "/system/bin/dd","/system/bin/df",
        "/system/bin/dmesg","/system/bin/getevent",
        "/system/bin/getprop","/system/bin/hd",
        "/system/bin/id","/system/bin/ifconfig",
        "/system/bin/iftop","/system/bin/insmod",
        "/system/bin/ioctl","/system/bin/ionice",
        "/system/bin/kill","/system/bin/ln",
        "/system/bin/log","/system/bin/ls",
        "/system/bin/lsmod","/system/bin/lsof","/system/bin/mkdir",
        "/system/bin/mount","/system/bin/mv",
        "/system/bin/nandread","/system/bin/netstat",
        "/system/bin/newfs_msdos","/system/bin/notify",
        "/system/bin/printenv","/system/bin/ps","/system/bin/reboot",
        "/system/bin/renice","/system/bin/rm",
        "/system/bin/rmdir","/system/bin/rmmod",
        "/system/bin/route","/system/bin/schedtop",
        "/system/bin/sendevent","/system/bin/setconsole",
        "/system/bin/setprop","/system/bin/sleep",
        "/system/bin/smd","/system/bin/start",
        "/system/bin/stop","/system/bin/sync",
        "/system/bin/top","/system/bin/umount","/system/bin/uptime",
        "/system/bin/vmstat","/system/bin/watchprops",
        "/system/bin/wipe");
    symlink("/system/etc/ppp/ip-up-vpn", "/system/etc/ppp/ip-down-vpn");
    symlink("/system/bin/dumpstate", "/system/bin/dumpmesg");
    symlink("/system/bin/debuggerd", "/system/bin/csview");
```

这些命令创建符号连接，第一个 symlink 创建所有的 busybox 命令。

```
ui_print("Setting Permissions");
set_perm_recursive(0, 0, 0755, 0644, "/system");
set_perm_recursive(0, 2000, 0755, 0755, "/system/bin");
set_perm(0, 3003, 02750, "/system/bin/netcfg");
set_perm(0, 3004, 02755, "/system/bin/ping");
set_perm(0, 2000, 06750, "/system/bin/run-as");
set_perm_recursive(1002, 1002, 0755, 0440, "/system/etc/bluetooth");
set_perm(0, 0, 0755, "/system/etc/bluetooth");
set_perm(1000, 1000, 0640, "/system/etc/bluetooth/auto_pairing.conf");
set_perm(3002, 3002, 0444, "/system/etc/bluetooth/blacklist.conf");
set_perm(1002, 1002, 0440, "/system/etc/dbus.conf");
set_perm(1014, 2000, 0550, "/system/etc/dhcpcd/dhcpcd-run-hooks");
set_perm_recursive(0, 2000, 06755, 06755, "/system/etc/init.d");
set_perm(0, 0, 06755, "/system/etc/init.d");
set_perm(0, 2000, 0550, "/system/etc/init.goldfish.sh");
set_perm_recursive(0, 0, 0755, 04755, "/system/etc/ppp");
set_perm(0, 0, 04755, "/system/etc/ppp/ip-up-vpn");
set_perm(0, 0, 04755, "/system/etc/ppp/ip-down-vpn");
set_perm_recursive(0, 2000, 0755, 0755, "/system/xbin");
set_perm(0, 0, 06755, "/system/xbin/su");
set_perm(0, 0, 06755, "/system/xbin/busybox");
set_perm(0, 0, 06755, "/system/xbin/invoke-as");
set_perm_recursive(1000, 1000, 0771, 0771, "/data");
```

这些命令设置文件和目录的 uid, gid 和访问权限。

```
show_progress(0.200000, 0);
show_progress(0.200000, 10);
show_progress(0.200000, 10);
```

进度显示相关，在 20 秒内显示安装进度增长了 60%。

```
package_extract_file("flash_image", "/tmp/flash_image");
set_perm(0, 0, 0777, "/tmp/flash_image");
ui_print("Installation du kernel...");
ui_print(" ");
assert(package_extract_file("zImage", "/tmp/zImage"),
run_program("/tmp/flash_image", "/dev/block/mmcblk0p5", "/tmp/zImage"),
delete("/tmp/zImage"));
```

这些命令的作用就是刷写 boot 分区，将刷机包中 zImage 的内容刷写到 boot 分区。如果不想改变 boot 分区很简单，删除这些命令。

```
ui_print("Installation modem...");
ui_print(" ");
show_progress(0.2, 15);
assert(package_extract_file("modem.bin", "/tmp/modem.bin"),
```

```
run_program("/tmp/flash_image", "/dev/block/mmcblk0p8", "/tmp/modem.bin"),
```
这些命令的作用是刷写 moden 分区，即我们通常所说的刷写 RADIO。

```
delete("/tmp/modem.bin"));
delete("/tmp/flash_image");
show_progress(0.100000, 0);
unmount("/system");
unmount("/cache");
unmount("/data");
ui_print("       ____    ____    _     _________");
ui_print("      / __ \/ __ \/ | / / ____/");
ui_print("    / / / / / /  |/ / __/      ");
ui_print(" / /_/ / /_/ / /|  / /___      ");
ui_print("/_____/\____/_/ |_/_____/      ");
ui_print("----------------------------");
ui_print("                Please Reboot Phone");
```
一些善后操作，删除临时文件，卸载相应的挂载点，这样保证我们对分区的修改不会丢失，然后打印一些好玩的信息。

3. 制作刷机包

看完上一节，大家可能会觉得一个完整的 updater-script 还是比较复杂的，但是基本上我们不需要写一个完整的 updater-script。就如第三章中所提到的，在开始定制 MIUI ROM 之前我们需要寻找一个合适的原厂 ROM 刷机包，这个刷机包中就包含了写好的 updater-script，我们想要加入些什么功能，只需要在适当的位置加入一些命令。

比如在 I9100ZCKJ1.zip 包的 updater-script 中，我们就加入了这个命令：

```
set_perm(0, 0, 06755, "/system/xbin/invoke-as");
```
invoke-as 是 MIUI 编写的一个程序，这条命令将 invoke-as 设为一个可执行的 set-uid 程序。什么是 set-uid 程序，请 google。

明白了 updater-script 的原理，在找不到 ZIP 包的情况下，我们也可以很容易的做出 ZIP 包满足 patchrom 项目的需要。把我们在移植过程中需要修改的所有 system 目录下的文件从手机中 pull 出来，然后写如下一个简单的 updater-script， 按照之前所说的组织方式压缩成 ZIP 包:

```
mount("ext4", "EMMC", "/dev/block/mmcblk0p9", "/system");
package_extract_dir("system", "/system")
unmount("/system")
```
即可。有了这样一个 ZIP 包，patchrom 的脚本就可以自动的打包了。

其实除了通过 recovery 安装 ZIP 包这种刷机方式，对于某个机型，可能有某些特殊的刷机方式，比如说对于 i9100 可以制作 odin 包，这些大家多多在网上查找相应的资料即可。