Smali 开发详解

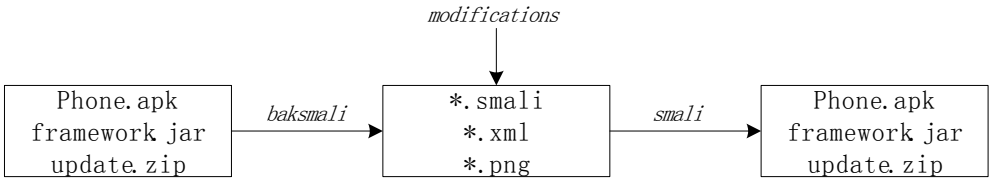
文档历史修改记录

修改记录	提交人	版本
文档创建，初始内容	段启智	V1.0

概述

在 Android 虚拟机(Dalvik)中，执行文件是 dex 格式，这种文件格式的编译器和反编译器分别是 Smali 和 Baksmali。因为 Dalvik 是一个冰岛渔村名字，所以编译器和反编译器的命名沿用了冰岛语中的 Smali 和 Baksmali。

本文涉及到的 Smali 开发方法，能够用于修改已有的可执行文件(APK, JAR)，甚至用于定制整个 Android 系统。简而言之，我们使用 Smali 相关的工具对 Android 可执行文件进行反编译，得到 Dalvik 虚拟机的汇编码，然后对这些汇编码进行修改，再重新编译，就得到了修改后的可执行文件。



为了描述方便，我们把编译和反编译的过程称为 Smali 和 Baksmali。通过 Smali 和 Baksmali 的方法，对 Android 进行修改定制，是“逆向工程”的思想：从最终产品出发，逆序整个开发流程，通过反编译的手段，在汇编码级分析代码逻辑，从而了解源代码的结构和产品设计的初衷。

本文详解 Smali 开发涉及到的工具和方法，辅以实例和常见问题，讲述通过 Smali 方法定制单独一个 APK 或整个 Android 系统的流程。一共分为以下章节：

1). Smali 环境，讲述 Smali 开发环境的构建，工具的使用；

2). 开发流程，分为 APK 定制和 Android 系统定制两个部分，讲述通过 Smali 方法修改 Android 的过程；

3). 实战演练，通过一些具体的实例，讲述 Smali 方法的应用；

4). 调试技巧，讲述 Smali 代码的断点调试方法；

5). 常见问题，汇总了 Smali 开发中常见的问题，以及对应的解决方法。

1. Smali 环境

1.1 开发环境

本章节介绍 Smali 开发的环境，包括操作系统平台、Smali 工具需要的运行环境、Smali 开发调试需要的集成开发环境，已经定制整个 Android 系统时需要的源码开发环境。

1). 操作系统

建议使用 Ubuntu 10 或 12 作为 Smali 的开发环境，因为 Ubuntu 平台下已有的工具和脚本都比较完善。

对于定制修改单个 APK 而言，Windows 环境下已有的工具和脚本已经足够；但对于定制修改整个 Android 系统而言，就需要对一些基础的工具进行封装和改善，使之能够适用于整个 Android 系统的反编译，同时，也需要一些系统的修改方法来保证 Smali 开发方法快速有效。

Ubuntu 下载地址：http://www.ubuntu.com/download/desktop

2). 运行环境

Java 运行环境，建议使用 JRE 1.6+，用于运行 Smali 的相关工具(譬如 smali.jar, baksmali.jar 等)，以及 Java 的签名工具 Jarsigner 等。如果需要运行 Android SDK，那 Java 运行环境也是必不可少的。

Java 下载地址：http://www.oracle.com/technetwork/java/javase/overview/index.html

3). 源码环境

一般而言，定制一个单独 APK 不需要 Android 源码的开发环境，但对于整个 Android 系统的定制，依赖于对 Android 源码的修改和调试，也需要借助源码来理解代码逻辑。具体请参见：

Android 源码下载和编译：http://source.android.com/source/building.html

4). 集成环境

对于 Smali 而言，一个简单的文本编辑器(譬如 vi, gedit)就能完成代码修改，但一个好的集成开发环境，能够为调试带来便利。推荐使用 Netbeans 作为集成开发环境，它能够断点调试 Smali 程序。

另外，Android SDK 中包含很多工具，能够给调试带来便利；在缺少真机时，也可以使

用模拟器来运行 APK。SDK 中的常用工具包含 adb，ddms，hierarchyview，这些都是必不可少的调试工具。

Netbeans 下载地址：https://netbeans.org/downloads/6.8/index.html

Android SDK 下载地址：http://developer.android.com/sdk/index.html

1.2 工具脚本

本章节介绍 Smali 相关的工具，这些工具能直接应用到开发和调试过程，也是分析和解析 Smali 问题的利器。对于整个 Android 系统而言，零散的工具虽然能够完成任务，但使用效率很低，需要在一些基础工具的基础上，构建一整套反编译工程，使工具的使用易于理解而且高效。

1.2.1 反编译和编译工具

Apktool.jar & Smali.jar & Baksmali.jar

Apktool 是 Google 提供的开源工具，用于反编译 Android 可执行程序，得到 Smali 代码和资源文件，修改后能够重新编译成可执行程序；同时也能用于断点调试 Smali 代码。

smali.jar 和 baksmali.jar 这两款工具就是实际的编译和反编译工具，apktool.jar 包含了这两者的功能，apktool.jar 的源代码就是对 smali.jar 和 baksmali.jar 的封装。实际使用过程中，我们通常直接使用 apktool.jar。

Apktool 工具详解：https://code.google.com/p/android-apktool/

Smali/Backsmali 详解：https://code.google.com/p/smali/

使用示例

```
java -jar apktool.jar d file.apk out/
```

反编译 file.apk 到 out/目录，d 表示 decode。

```
java -jar apktool.jar if famework-res.apk
```

装载资源文件，if 表示 install-framework。通常在反编译的时候会有一些警告，提示无法找到资源，这是因为应用程序通常会引用 Android 的一些资源，在反编译时，需要指定这些资源存放的位置。Android 原生只有一个资源包，但不同厂商通常会扩展自己的资源，这些资源包都在 system/framework/目录下，后缀名为 apk。在反编译之前，我们需要依次装载所有

的资源包。

```
java -jar apktool.jar b out/
```

编译 out/目录中的文件，b 表示 build。编译成功后，会在 out/目录下生成两个子目录：build/目录包含编译的中间产物，dist/目录包含最后的编译生成文件。需要注意的是，这个重新编译生成的可执行文件不能直接安装的手机，因为它还没有签名。

aapt

Aapt, Android Asset Packaging Tool 是 Android 提供的资源打包工具，其源码在 Android 的 frameworks/base/tools/aapt/目录。在编译 APK 的时候，aapt 会对资源进行编译，并生成 R.java 文件供代码通过资源 ID 访问资源。

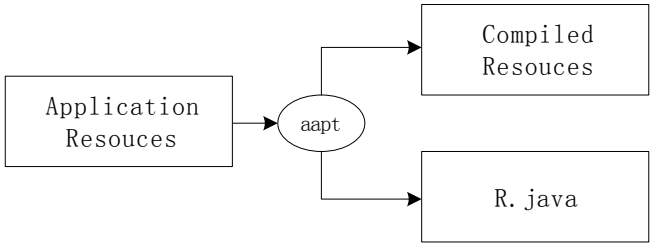


图 1.1 aapt 编译资源文件的过程

使用示例

```
aapt package -u -x -z -M  framework-res/AndroidManifest.xml \
        -A framework-res/assets \
        -S overlay/res1 \
        -S overlay/res2 \
        -S framework-res/res \
        -F out.apk
```

编译资源文件，-u 参数表示更新已有的资源包；-x 参数表示新增资源 ID；-z 表示需要提供资源的国际化信息；-M 参数表示指定 AndroidManifest.xml 文件；-A 参数表示指定 assets 目录；-S 参数表示待编译的资源文件目录，可以附带多个目录，出现重名资源时，会按照参数的排列顺序优先使用顺序靠前的资源；-F 参数表示输出的资源包名称。

1.2.1 密钥生成和签名工具

Keytool & Jarsigner & SignApk.jar

任何 Android 可执行文件都需要签名后才能安装。对于反编译修改后的 APK 或 JAR，都需要重新签名，通过 Keytool 生成密钥仓库，通过 Jarsigner 进行签名。

Keytool 是 Java 官方提供的密钥和证书管理工具，它将密钥(key)和证书(certificates)存在一个称为密钥仓库 keystore 的文件中，密钥用于对用户数据进行加密，密钥可分为对称密钥和非对称密钥；证书用于身份认证，在数字签名中指的是非对称密钥中的公钥。

Jarsigner 是 Java 官方提供的对 Jar 包进行签名和验证的工具。对称密钥或者非对称密钥中的私钥，都是通过用户口令来保护的，也就是只有通过口令才能解开对称密钥或非对称密钥的私钥。而公钥是公开的，通过私钥对数据摘要进行签名，就能通过与之配对的公钥解密，从而达到身份认证和数据完整性验证。

SignApk.jar 是 Android 提供的签名工具。签名后的应用或 Jar 包中会多出 META-INF 目录，下面包含三个文件：MANIFEST.MF(程序清单文件，它包含包中所有文件的摘要明文)，CERT.SF(加密文件，它是使用私钥对摘要明文加密后得到的密文信息，只有使用私钥配对的公钥才能解密该文件)，CERT.RSA(公钥和加密算法描述)。

Keytool 详解：http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html

Jarsinger 详解：http://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html

SignApk 详解：https://code.google.com/p/signapk/

使用示例

---

*keytool -genkey -alias myKeystoreAlias -keysize 1024 -keypass abcdef -storepass fedcba *

-keystore myKeystore

生成密钥仓库，命令中红色字体代表用户自定义输入的值。-alias 表示密钥仓库的别名，-keysize 表示密钥仓库的大小，-keypass 表示私钥的口令，-storepass 表示密钥仓库的口令，-keystore 表示生成密钥仓库的目录。注意-keypass 和-storepass 两个口令是有区别的，前者用于保护私钥，后者用于保护密钥仓库。

---

jarsigner -keystore myKeystore -keypass abcdef -storepass fedcba file.jar

---

使用密钥仓库对 Jar 包进行签名，需要指定密钥仓库的路径，私钥口令和密钥仓库的口令。

```
java -jar signapk.jar  platform.x509.pem  platform.pk8  file.apk  file_signed.apk
```

运行 signapk.jar 对 apk 进行签名。本例中，使用 Android 的 platform 形式的签名，其中*.x509.pem 和*.pk8 就是公钥和私钥对，Android 将其分开存放，也有工具将两者合并存入密钥仓库 keystore 文件中。

1.2.3 优化工具

Zipalign

Zipalign 是 Android 提供的可执行程序优化工具。在 Android 中，每个应用程序中储存的数据文件都会被多个进程访问，安装程序会读取应用程序的 AndroidManifest.xml 文件来处理与之相关的权限问题；Home 应用程序会读取资源文件来获取应用程序的名和图标；系统服务会因为很多种原因读取资源；此外，应用程序也会有自身用到资源文件。当资源文件通过内存映射对齐到 4 字节边界时，访问资源文件的代码才是有效率的。

Zipalign 详解：http://developer.android.com/tools/help/zipalign.html

使用示例

```
zipalign -c -v 4 file_signed.apk
```

对 file_signed.apk 进行优化，-v 表示在控制台输入优化信息，-c 表示确认对其方式。在 32 为系统上，对齐方式都是为 4 字节对齐。

1.2.4 打包和解包工具

mkbootimg & unpackbootimg & mkbootfs

对于 boot.img，我们可以将其解开，修改并重新打包。通常的修改包括：修改内核 root 权限，增加系统服务，增加 recovery 功能。

unpackbootimg 和 mkbootimg 是 Android 提供的解包和打包 boot.img 的工具，可以从 Android 源码中找到这个工具的源码(system/core/mkbootimg)。

mkbootfs 是 Android 提供的制作文件跟系统(Ramdisk)的工具，fs 的意思就是 file system。boot.img 包含文件头信息(Boot Header)、内核(Kernel)、文件系统(Ramdisk)以及二级装载(Secondary Stage Loader)，解开 boot.img 后就能得到这些组成部分，二级装载是可选的，不一定存在。通常，我们只能修改文件系统，文件系统是一个压缩包，需要将其解包然后再对里面的文件做一些必要的修改，修改完毕后再通过 mkbootfs 重新打包文件系统。

使用示例

unpackbootimg -i boot.img

解包 boot.img，-i 参数表示指定输入的镜像，即 input 的意思。同时可以附带-o 参数表示解包 boot.img 后的输出目录，如果没有指定，则默认输出在当前目录。正确解包 boot.img 后，会输出部分文件头信息(BASE, COMMAND_LINE, PAGESIZE)，内核(命名为 kernel 或 zImage)，文件系统压缩包(命名为 ramdisk.gz)。

注意：厂商不一定按照标准格式实现 boot.img，对于某些厂商的 boot.img，这个工具就不能正确解包。这时，我们需要寻求厂商发布的解包 boot.img 的工具。

mkbootimg --cmdline 'COMMAND_LINE' --kernel zImage --ramdisk ramdisk.gz --base 0x00200000 --pagesize 4096 -o boot-new.img

打包 boot.img，--cmdline 参数表示内核命令，--kernel 表示内核，--base 表示文件系统的基地址，--pagesize 参数表示 boot.img 的页大小，-o 参数表示打包输出的 boot.img 路径。所需要的输入参数，都是通过解包 boot.img 得到的。

mkdir ramdisk && cd ramdisk && gzip -dc ../ramdisk.gz | cpio -i

解包文件系统 Ramdisk。在解包后 boot.img 的目录下执行上述命令，就能将文件系统解包到一个新建的文件目录 ramdisk/中。文件系统是有目录结构的，通常而言，我们需要对*.rc 等一些配置文件进行修改，植入我们所需要的功能。

mkbootfs ./ramdisk | gzip > ramdisk-new.gz

打包文件系统 Ramdisk。在解包 boot.img 的目录下，执行上述命令，就能重新将 ramdisk/目录打包成 ramdisk-new.gz。

unyaffs & mkyaffs2image & make_ext4fs

对于 system.img，我们可以将其解包，修改并重新打包。system.img 是 system/目录的一个映像，采用 yaffs2 格式(Yet Another Flash File System)。解包 system.img 后，得到的目录就是 system/，它包含了 Android 所有的系统应用程序、库文件、二进制可执行文件、配置文件等。我们对整个 Android 系统进行定制，最主要的修改就集中在这个目录。

unyaffs 和 mkyaffs2image 是 Google 的一个开源项目，用于解包和打包 yaffs2 格式镜像的工具。

make_ext4fs 是 Android 提供的用于制作 ext4 文件系统镜像的工具。

unyaffs 详解：https://code.google.com/p/unyaffs/

使用示例

unyaffs system.img

解包 system.img，会在当前目录下生成 system/目录

mkyaffs2image system system.img

将 system/目录打包成 system.img

make_ext4fs -l 419430400 -a system system.img.ext4 system/

打包 system/目录成 ext4 文件系统镜像。-l 参数表示生成的 system.img 的分区大小，-a 参数表示挂载点，本命令中挂载点是 system/分区，输入参数 system/表示待打包的 system 目录，system.img.ext4 表示输出的镜像文件。

1.2.5 其他辅助工具

dextojar & JD-GUI

dextojar 是一个开源项目，用于将 Android dex 格式的文件反编译成 Java class 文件，相当于 Android 的 dx 程序的逆过程。

JD-GUI 是一个有图形界面的反编译器，用于将 Java class 文件反编译成源代码。

有了这两个工具的支持，我们就能看到一个 APK 或 Android JAR 包的源代码。但这两个工具并不一定是完全有效的，在源代码混淆的情况下，我们反编译得到的 Java 源代码还是很难看懂。

dextojar 详解：https://code.google.com/p/dex2jar/

JD-GUI 详解：http://java.decompiler.free.fr/?q=jdgui

使用示例

dextojar clasess.dex

反编译 dex 格式的文件，生成 Java class 格式的文件。

1.3 编译环境

编译环境是针对于定制整个 Android 系统，频繁的使用多种工具效率很低，因此，需要对一些基本的工具进行封装，构造出一套完整的编译环境。

在给定一个厂商手机的情况下，编译环境需要完成以下主要的工作：

* 从手机中取出厂商的 boot.img 和 system.img，并自动将取出的镜像文件解包；
* 创建 Smali 工程，自动生成工程目录，自动反编译必要的 APK 和 JAR；
* 提供一些自动化的工具，能够便利地修改厂商的代码。譬如自动化合并；
* 编译所有修改后的 APK 和 JAR。根据配置选项，拷贝其他未改动的文件；
* 对编译产出的包进行签名，生成可用的卡刷包。

编译环境的涉及到的工作流程如图 1.1 所示。

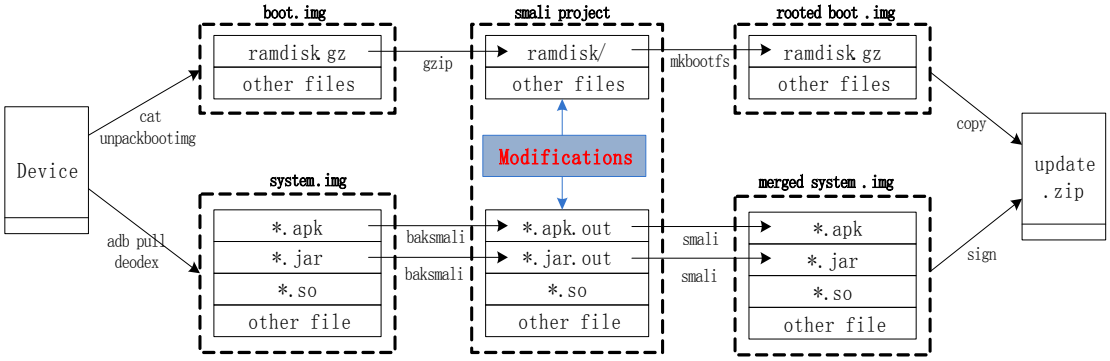


图 1.1 Smali 编译环境的工作流程

可以看到整个编译过程，主要是对 system.img 中多个 APK 和 JAR 的定制，这些定制会涉及到修改 APK 和 JAR 相关的配置文件和底层库。编译环境应该做到灵活的配置哪些 APK 和 JAR 编译到最终的 system.img。对反编译出的代码进行修改，主要集中在与系统功能关联紧密的程序：framework.jar.out/, services.jar.out/，android.policy.jar.out/， framework-res/，Phone/，这些目录重编译后分别对应到 framework.jar，services.jar，andorid.policy.jar，framework-res.apk，Phone.apk。

对于 boot.img 而言，编译环境最初会将其解包，生成目录，编译环境能够重新打包这个目录，生成修改后的 boot.img，并将其置入最终的卡刷包中。

2. 开发流程

本章节分别介绍单个 APK 和整个 Android 系统的定制开发流程。APK 的定制是整个 Android 系统定制的基础。

2.1 APK 定制流程

对 APK 进行定制的目的主要有 APK 美化，加入或去除 APK 的广告，增加或减少 APK 的功能；此外，学习一些优秀的 APK 的结构，了解设计意图，也是反编译 APK 的一个用途。通过解压程序，譬如 winrar，可以打开任意一个 APK，在反编译之前，APK 的目录结构为：

```
target.apk
+- assets/                  不经过 aapt 编译的资源文件
+- META-INF/                文件摘要，摘要加密和签名证书文件目录
+- res/                     资源文件目录，二进制格式
+- resources.arsc           经过 aapt 编译过的资源文件
+- classes.dex              Android 可执行文件，dex 格式
+- AndroidManifest.xml      反编译后的程序清单文件，二进制格式
```

2.1.1 装载 framework

一般而言，Android 市场上复杂一点的 APK 都会使用系统资源(Java 代码中以 android.R. 引用的资源)，这些资源文件都存放在 system/framework/framework-res.apk 这个包中。有些厂商会扩展自己的资源包，譬如，HTC 就扩展了 com.htc.resources.apk 这个资源包，HTC 自身的一些应用程序很可能引用 com.htc.resources.apk 这个包里的资源。为了能够反编译 APK，我们需要先装载这些资源包。

首先，通过 adb pull 命令把手机上 system/framework/目录下，所有资源相关的包都拉下来(一般后缀名是 apk)。

然后通过以下命令，装载资源包：

```
java -jar apktool.jar if framework-res.apk
```

正确装载后，可以在~/apktool/目录下找到一个 1.apk，它表示所有资源 ID 为 0x1 开始，Android 不允许有重复的资源 ID，如果厂商扩展的资源包，那么扩展包中的资源 ID 就是以

0x2 开始，装载扩展资源包时，就会在~/apktool/目录下出现一个 2.apk。依次类推，如果还有其他扩展资源包，那么对应的资源 ID 就是 0x3，0x4 开始。

2.1.2 反编译 APK

使用如下命令，反编译需要定制的目标 target.apk。

```
java -jar apktool.jar d target.apk
```

成功反编译后，会在当前目录下生成 target/目录，其目录结构如下：

```
target/
+- assets/                  不需要反编译的资源文件目录
+- res/                     反编译后的资源文件目录
+- smali/                   反编译后的源代码目录，文件后缀名为 smali
+- AndroidManifest.xml      反编译后的程序清单文件
+- apktool.yaml             反编译生成的文件，供 apktool 使用
```

可以发现，已经没有了 META-INF 目录，说明反编译会丢失签名信息。对比反编译之前的文件，XML 文件是可读的，已经不是二进制格式，说明该过程会将二进制的资源文件还原成其原来的格式。不过还是存在信息的丢失：注释和原来的排版都已经丢失了。

如果我们只需要修改资源，则可以根据自身的需要进行反编译，apktool 提供相应的参数，譬如-s 参数表示不反编译代码，-r 参数表示不反编译资源。

2.1.3 修改 Smali 代码

修改之前，需要对 Smali 的语法有一些了解。粗略的理解 Smali 代码片段并不难，它类似于汇编码，通过一些指令在操作寄存器。通常，对于需要的功能，直接写 Smali 并不容易，可以采用的一个方法是，先把功能用源码的方式实现，然后反编译得到 Smali 代码，再把 Smali 代码合并到目标代码中。

以加入日志代码为例，假定需要在反编译后的代码中加入日志，打印变量信息，那么可以先用一个简单示例 APK，完成以下源代码：

```
void mockCaller() {
    myLog("Caller Here");
```

```
}
private static void myLog(Object o) {
    Log.d("HelloSmali", o.toString());
}
```

反编译该示例 APK，得到该源代码对应的 Smali 代码如下：

```
# virtual methods
.method mockCaller()V
    .locals 1
    .prologue
    .line 23
    const-string v0, "Caller Here"
    invoke-static {v0}, Lorg/demo/hellosmali/MainActivity;->myLog(Ljava/lang/Object;)V
    .line 24
     return-void
.end method

.method private static myLog(Ljava/lang/Object;)V
    .locals 2
    .parameter "o"
    .prologue
    .line 24
    const-string v0, "HelloSmali"
    invoke-virtual {p0}, Ljava/lang/Object;->toString()Ljava/lang/String;
    move-result-object v1
    invoke-static {v0, v1}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I
    .line 25
    return-void
.end method
```

打印日志的代码就是 myLog()函数中.line 24 和.line 25 之间的 Smali 代码片段，调用代码是 mockCaller()函数中.line 23 到.line 24 之间的 Smali 代码片段。要在目标 Smali 代码中实现打印日志的功能，首先需要先将 myLog()函数合并，然后在需要打印日志的地方，调用 myLog()函数，传入的变量值 v0，就是待打印的变量值。调用的方式与下面的方法类似：

invoke-static {v0}, Lorg/demo/hellosmali/MainActivity;->myLog(Ljava/lang/Object;)V

2.1.4 重编译并签名 APK

通过以下命令可以重新编译修改的 APK

```
java -jar apktool.jar b target/
```

如果所有的修改都正确有效，那么在 target 目录下会生成两个目录 build/和 dist/，分别存放编译的中间结果和最终产出。但遗憾的是，对于同样功能的代码，相比 Java 源代码而言，Smali 代码行数很多，而且寄存器变量容易误用，所以容易导致编译出错，或者在后续运行时出现异常。建议尽量以代码块的方式合并 Smali 代码，减少对原来逻辑的破坏，而且在合并代码的时候，要留意寄存器变量的使用。

重编译 APK 后，通过压缩工具解开，可以发现 APK 里面并没有 META-INF/目录，这样的 APK 是不能安装的手机的，对 APK 进行签名后，就会生成 META-INF/目录，这个过程是必要的。通过以下命令对 APK 进行签名，签名完成后，生成了 target_signed.apk，这个 APK 已经可以安装到手机。但发布该 APK 的时候，建议使用 zipalign 工具对 apk 进行优化。

```
java -jar signapk.jar  mykey.x509.pem  mykey.pk8  target.apk  target_signed.apk
zipalign -c -v 4 target_signed.apk
```

这里涉及到到一个很重要的概念，就是签名，命令中 mykey.x509.pem 和 mykey.pk8 是通过工具 keytool 生成的公钥和私钥对，它包含了签名者的信息，这个签名信息可以认为是独一无二的，重新签名后，这个 APK 已经不是反编译前的原来那个 APK 签名，因此，再发布这个 APK 的时候，它的签名者已经变了，这会带来一些影响：

1). 如果原来的 APK 是一个知名机构(譬如 Google Inc.)发布的，那重新签名后，发布机构已经变成了修改者，那么安装者容易质疑这个 APK 的安全性和有效性；

2). 重新签名后的 APK 不再享有原来的升级。

2.2 Android 系统定制流程

本小节重点讲述整个 Android 系统的定制流程，简单的，可以将这个过程理解为多个 APK 和 JAR 包的同时定制。为了将整个过程描述清楚，需要引入一些基本的概念：

Android Source：Android 源代码，这部分代码可以编译出整个 Android 原生的系统；

our_update.zip：刷机包，在 Android 源代码基础上修改，编译得到的刷机包。这部分修改的源代码是我们已知的，它是我们在 Android 原生基础上的一些扩展；

oem_update.zip：刷机包，厂商在 Android 源代码基础上的修改，编译得到的刷机包。这个刷机包是可以刷入厂商对应的手机的，也可以通过工具从已有的手机中取出文件，生成这个刷机包；

宏观来看，我们是对厂商的 oem_update.zip 这个刷机包进行定制，需要定制的内容就是我们在 Android 源代码上的修改。通过 Smali 的方法，就是将我们的刷机包 our_update.zip 和 oem_update.zip 整个系统进行反编译，然后将我们的改动合并到厂商的刷机包中，使得厂商的刷机包 oem_update.zip 中，具备我们需要的功能。原理示意如图 2.1 所示。

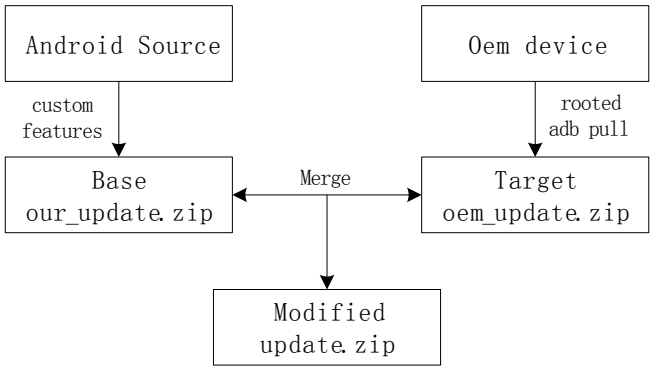


图 2.1 整个 Android 系统定制的原理

我们的刷机包 our_update.zip 称为 Base，它是我们的基线版本，之所以称之为 Base，是因为我们这个包可以适应不同的厂商，它只是我们在 Android 原生基础上改动的一个载体；厂商的刷机包 oem_updated.zip 称为 Target，它是我们待定制的目标；合并后的刷机包 update.zip 包含了我们的修改，最终将由我们进行签名后发布。

上述内容只是宏观理解整个 Android 系统的定制原理，实际操作过程中，还有诸多细节，下面就详细介绍定制步骤。

2.2.1 解锁 BootLoader

Boot Loader 是在操作系统内核运行之前运行的一段小程序。这段小程序完成硬件设备的初始化、建立内存空间的映射图，以便为最终调用操作系统内核 kernel 准备好正确的环境。这个过程完毕后，系统启动就进入了 boot 分区，即我们的 boot.img 刷入的分区，然后读取手机的操作系统内核 kernel，启动 Android 系统。

Bootloader 跟硬件关联最为紧密，很多厂商(譬如 HTC，SONY 等)在出厂时，都会给 Bootloader 上“锁”，防止用户烧写非官方验证的 boot.img。如果我们需要对 boot.img 做修改，就需要先解锁 Bootloader。

为了满足高端用户的需要，厂商一般都提供解锁 Bootloader 的方法，不过也会附带解锁后的免责声明，所以从解锁 Bootloader 开始，定制 Android 系统就是一个有风险的举动。不过，对于发烧友而言，风险与乐趣成正相关，定制整个系统带来的乐趣远胜于风险。

虽然解锁 Bootloader 的命令依具体厂商而定，但解锁的过程还是类似：

1). 在厂商官网上注册申请，一般要求输入手机设备的 ID 和邮箱地址；

2). 成功申请后，得到一个解锁 Bootloader 的 Key，它和手机设备 ID 是关联的，所以通常不能用这个 Key 解锁其他手机设备；

3). 在手机设备进入 Bootloader 模式下，通过厂商提供的命令和 Key，解锁 Bootloader。

另外，现在市面上也有一些一键解锁 Bootloader 的工具，懒人必备。

2.2.2 定制 Boot.img

1). 解包 Boot.img

Boot.img 是 boot 分区的一个镜像文件，Android 定义了它的标准格式，但厂商有可能在标准格式上做修改，所以解包 Boot.img 的工具并不是通用的，在修改 Boot.img 之前，需要先找到解包它的方法。有的厂商会官方提供解包和打包 Boot.img 的工具，譬如 SONY。

以 Andorid 定义的标准格式为例，通过以下命令解包 Boot.img：

```
unpackbootimg -i boot.img
```

解包后，得到的目录结构如下所示：

```
boot.img/
+- base                    文本格式，表示文件系统的基地址
```

```
+- cmdline          文本格式，内核的一个命令参数
+- pagesize         文本格式，页大小
+- kernel           二进制格式，内核
+- ramdisk.gz       压缩包，文件系统
```

虽然 Boot.img 的格式不一，但其基本结构是一致的，如图 2.2 所示：

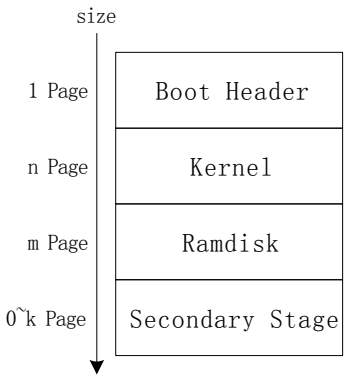


图 2.2 Boot.img 的结构

Boot Header 定义了 boot.img 的一些基本信息，包括页大小 pagesize、Kernel 的大小与起始地址、Ramdisk 的大小和起始地址等，Boot Header 本身会占用 1 页的大小，如果不足会以全 0 补全；

Kernel 是操作系统内核文件，是一个二进制文件，在缺少内核源码的情况下，我们只能作简单替换，不能修改。Kernel 占用的大小不是固定的；

Ramdisk 是一个文件系统，以压缩包的形式存在，这个压缩包的大小也不是固定的。一般而言，定制 boot.img 主要的修改就集中在文件系统；

Secondary Stage Loader 二级装载器是可选的，依厂商的 Boot.img 结构设计而定。我们无需关心这部分数据。

2) 修改 Boot.img

修改 Boot.img 的需求主要来自两方面，获取 ROOT 权限和添加开机时启动的系统服务，这些修改修改都在 Ramdisk 中，所以首先需要将 Ramdisk 则个压缩包解开，在解包 Boot.img 后的目录下执行以下命令：

```
mkdir ramdisk && cd ramdisk && gzip -dc ../ramdisk.gz | cpio -i
```

解开它文件系统压缩包，就得到了一个目录结构。其中，最主要的就是 default.prop 和一些后缀名为 rc 的文件，它们都是文本格式，也是我们定制 boot.img 做修改的主要文件。

Ramdisk/	
+- sbin/	可执行文件目录
+- init	系统启动程序，通常为二进制格式
+- init.rc	启动程序的配置文本文件，通常还有其他一些后缀为 rc 的文件
+- default.prop	属性配置文件，文本
+- ...	其他目录或文件，不同厂商增加的一些功能程序

获取 ROOT 权限

在 ramdisk/default.prop 文件中，定义了一些系统属性，需要修改以下属性来获取 ROOT 权限，并正常启动调试功能。

ro.secure=0	控制 adbd 的启动身份，0 表示 root，1 表示 shell
persist.service.adb.enable=1	与 rc 文件配合使用，设置为 1，通常表示开机启动 adbd
ro.debuggable=1	开启串口调试，设置为 1，通常会开启控制台输出

添加开机启动的服务

在 ramdisk/init.rc 文件中，定义了一些开启动服务，譬如常见的有：ril-daemon 的 service 块，netd 的 service 块等。如果需要自行定义，只需遵循 init.rc 的文件格式，在其中添加即可。Init 二进制程序启动的时候，会读取*.rc 文件中配置。

3). 重新打包 Boot.img

对 Ramdisk 进行修改后，需要生成新的 Boot.img。首先需要将修改后的 Ramdisk/目录重新打包，然后再重新打包 boot.img。

```
mkbootfs ./ramdisk | gzip > ramdisk-new.gz
mkbootimg --cmdline 'cmdline --kernel kernel --ramdisk ramdisk-new.gz --base base --pagesize pagesize -o boot-new.img
```

重新生成 Boot.img 所需要的参数，都是解包 Boot.img 的时候生成的。这样，就能够生成一个新的 boot-new.img，它需要具备 ROOT 权限，才能满足后续定制的需要。

总结一下，修改 Boot.img 的过程如图 2.3 所示：

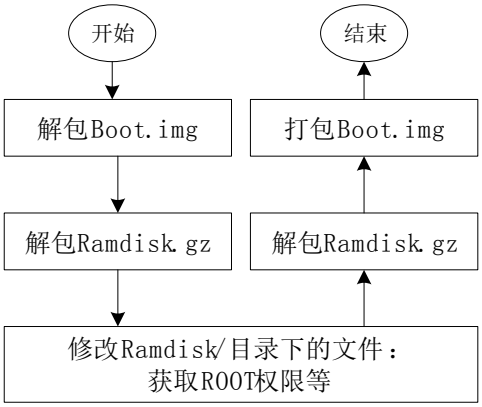


图 2.3 定制 Boot.img 的过程

2.2.3 定制 Recovery.img

1). 独立的 Recovery.img 结构

Recovery.img 是具备恢复功能的 Boot.img，从结构上讲，Recovery.img 和 Boot.img 是一致的；从功能上讲，Recovery.img 比 Boot.img 多了一个恢复功能，类似于 Windows 上的 Ghost 系统。当手机进入到 Recovery 模式时，就可以看到 Recovery 的主界面和其所具备的功能，一般使用率最高的就是选择“apply update.zip from sdcard”，意思就是从 SDCard 上选择一个卡刷包，将其刷入系统。

由于与 Boot.img 的结构一致，所以解包、修改和打包的方法都与 Boot.img 是一致的。与 Boot.img 的不同的细节体现在文件系统 Ramdisk 上，如图 2.4 所示。解包 Recovery.img 的 Ramdisk 后，会发现里面通常会具备 sbin/recovery 程序，而且配置文件也不同(init.rc,default.prop)，Recovery 的功能就是由这个二进制程序实现的；另外，Recovery 具备自己的界面，所以会比 Boot.img 的 Ramdisk 多出一个 res/目录，存放 Recovery 相关的界面资源。

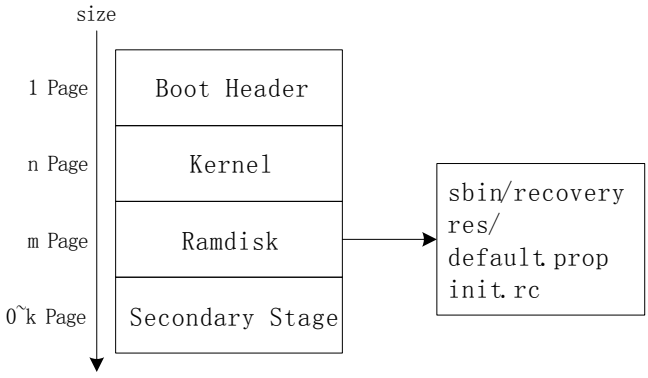


图 2.4 Recovery.img 的结构

2). 具备 Recovery 功能的 Boot.img

之所以有独立 Recovery.img 这个的概念，是因为这个镜像文件是刷入手机的 Recovery 分区。与之对应的，Boot.img 是刷入 Boot 分区，System.img 是刷入 System 分区。但并不是每个手机都有 Recovery 分区，所以独立的 Recovery.img 在这类手机上就没有意义。

一般而言，对于有 Recovery 分区的手机，都能从网上找到可以刷入手机的 Recovery.img，所以，定制 Recovery.img 的需求并不大；但对于没有 Recovery 分区的手机而言，如果需要 Recovery 功能，就需要从其他分区(Boot 分区，System 分区)衍生出这个功能。基于定制 Boot.img 的基础，可以将 Recovery 功能定制到 Boot.img，下面介绍一个参考做法。

考虑到系统启动时，都需要经过 init 二进制程序，正常情况下，这个程序会读取 Boot.img 相关的配置文件(*rc 等)，然后启动 Android 系统服务。如果将它修改，读取 Recovery.img 相关的配置文件，那么就能启动 Recovery 程序。也就是说，只要改变 init 程序的启动配置，就能控制进入 Android 系统还是进入 Recovery。其原理如图 2.5 所示。

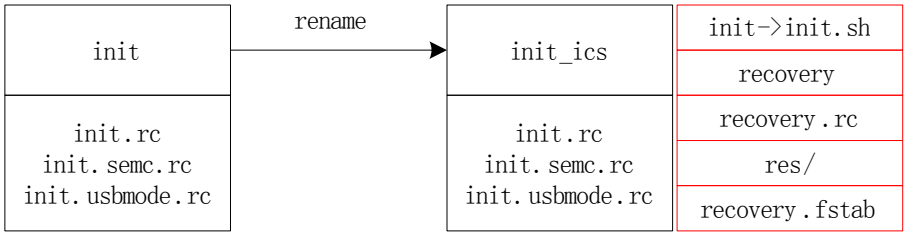


图 2.5 具备 Recovery 功能的 Boot.img 的 Ramdisk

图 2.5 左侧是原来 Boot.img 的 Ramdisk/目录下的主要文件，右侧是植入 Recovery 功能的 Ramdisk/目录。主要有以下几点变化：

* 增加了 Recovery 功能相关的文件：recovery 二进制、res/、recovery.fstab；

* 将 Recovery.img 本来的 init.rc 重命名为 reocvery.rc；

* 将 init 二进制程序重命名为 init_ics，增加了指向 init.sh 脚本的 init 链接；

这个 init.sh 脚本就完成了开机时的控制操作，在特定情况下，譬如音量控制按键响应，启动到 Recovery 模式。init.sh 脚本的主体内容如下：

```
if [ -s /dev/input/event0 ]
then
    rm /init.rc
    mv /recovery.rc /init.rc
```

```
    /init_ics
else
    /init_ics
fi
```

如果检测到音量按键，则将 recovery.rc 重命名为 init.rc，然后启动 init_ics，这样就能进入 Recovery 模式。

2.2.4 插桩 Smali

插桩是指将我们的 Smali 代码合并到厂商的代码中去，使厂商具备我们所需的功能。

为了描述方便，后文出现的 oem 对应到“厂商”的范畴，our 对应到“我们”的范畴，output 对应到将“我们”与“厂商”的功能合并后的“输出”范畴。

1). 手工合并 VS. 自动合并

理论上，可以采用全手工的方式，将我们的 Smali 代码合并到厂商的代码中。如果只是简单的修改厂商的代码，那全手工方式是有效的，但如果涉及到的修改很多，即我们需要深度定制厂商的代码，那么全手工方式的工作量巨大，而且容易出错。所以，需要通过工具做一些必要的自动化操作，自动化合并的对象可以分为 3 种类型：

类型 1：对于资源文件的改动，涉及到资源 ID 的变化，将我们的改动抽离出来，通过 overlay 的方式，利用 aapt 自动打包多个路径下的资源包。合并的原理如图 2.6 所示：

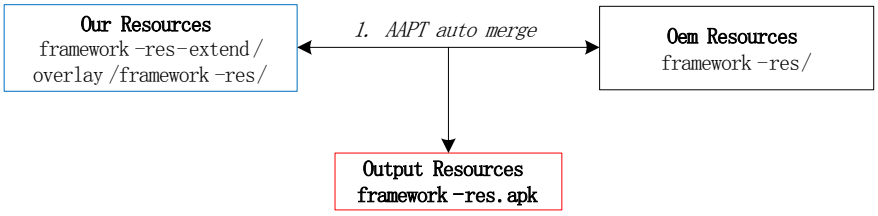


图 2.6 资源文件自动合并的原理

我们的不直接在 Android 原生的 framework-res/目录下进行改动，而是将所有的改动放在 overlay/目录，譬如，修改原生一个字符串的名字或图片，增加或删除一个图片，这些改动都放在 overlay/目录中。Aapt 在编译资源的时候，可以指定多个目录同时编译，这样就做到了资源的自动合并。

资源文件涉及到一个很重要的概念，就是资源 ID。Android 为所有公开的资源创建了一个资源名到资源 ID 的映射，保存在 framework-res/values/public.xml 文件中，其他地方都是

通过资源 ID 引用需要的资源。但当我们修改厂商的资源文件包是，难免会在厂商的基础上增加一些资源，这样 public.xml 的资源 ID 就会发生变化，所以在引用这些资源 ID 的地方都需要做对应的修改，才能保证资源引用正确。

此外，每一个资源包的资源 ID 的编号是有规律的，framework-res/包中的所有资源 ID 都是以 0x01 为起始的；如果厂商有扩展，例如 HTC 扩展了自己的资源包 com.htc.resources.apk，这个包中的资源 ID 就必须以 0x02 为起始。

类型 2：对于不涉及到修改厂商逻辑的代码，通过脚本自动拷贝到厂商代码中。譬如，我们只需要在厂商代码中添加一个完整的文件，或者只需要在厂商文件的末尾附加一段代码。

如果我们的代码中不存在对资源 ID 的引用，那么直接拷贝就厂商的代码中就可以；但如果我们的代码中引用了资源 ID，那这个 ID 必然是映射到我们的 framework-res.apk 的那些资源，把这些代码直接拷贝到厂商的代码中，导致的结果就是资源 ID 映射到的资源异常，甚至在厂商的 framework-res.apk 中都不存在这个资源 ID。所以，需要对这种类型的代码做一个资源 ID 的转换。其原理如图 2.7 所示。

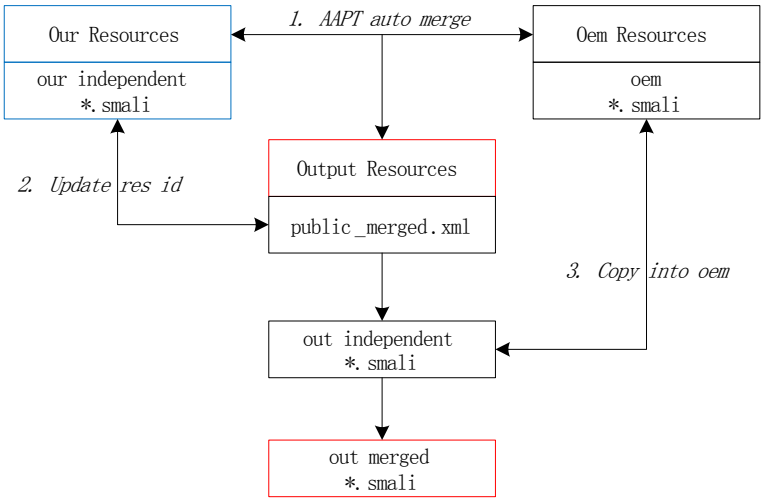


图 2.7　独立文件的自动合并原理

* 在资源自动合并完成后， 输出合并的资源包中有一个 public.xml，我们把它重命名为 public_merged.xml；

* 利用 public_merged.xml 更新我们独立文件中的资源 ID；

* 将修改资源 ID 后的独立文件直接拷贝到厂商的代码中，便生成了合并后的文件。

类型 3：对于涉及到修改厂商逻辑的代码，改动格式比较规范(即便采用手工方式，也很容易合并)，我们采用打补丁的方式自动合并。所谓补丁，是指我们在 Android 原生逻辑上的改动，这些改动可以通过工作自动补到厂商你代码中。其原理如图 2.8 所示：

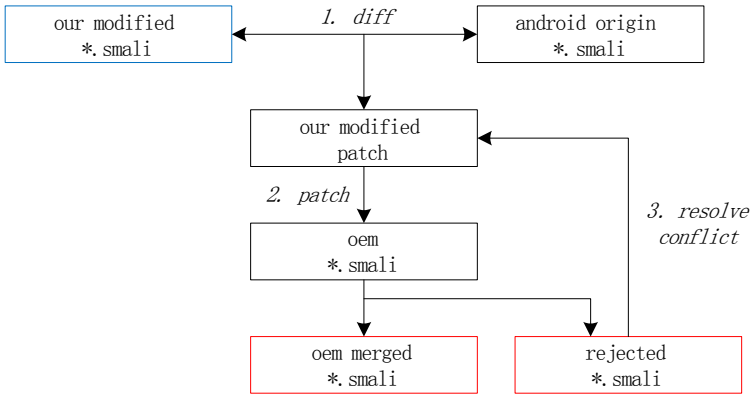


图 2.8 通过打补丁的方式自动合并的原理

* 将我们改动的文件与 Android 原生的文件对比，生成一个补丁；

* 将这个补丁打到厂商对应的 Smali 代码中；

* 打补丁会有两个结果，成功时，就生成了合并后的文件；失败时，就会提示产生冲突，这时只需要解决冲突，重新打上补丁即可。

如何使得打补丁的冲突尽可能少？一方面，我们的代码应该对 Android 原生产生尽量少的污染，即少改动 Android 原生的逻辑，这样从源头上抑制 Patch 的大小，冲突自然就少；另一方面，也依赖于经验的积累，当冲突不可避免的时候，加深对逻辑改动的理解，能够更快的解决冲突。

需要说明的是，自动合并只是我们为了提高效率的一个方法，这个方法并不是万能的。即便自动合并完全成功，可能由于厂商代码逻辑的改动，我们合并完后甚至会影响厂商代码的功能。这时候，还是依赖于手工分析合并的错误来解决问题。

2). 必要的 Bring Up 代码

我们的目的是使得厂商具备我们所需要的功能，更精确地说，是希望我们的应用程序能够正确运行在修改后的厂商代码上。我们的主要改动都是 app 和 framework，这些改动都体现在 system.img 中，最终形成的 system.img 的组成如图 2.9 所示。

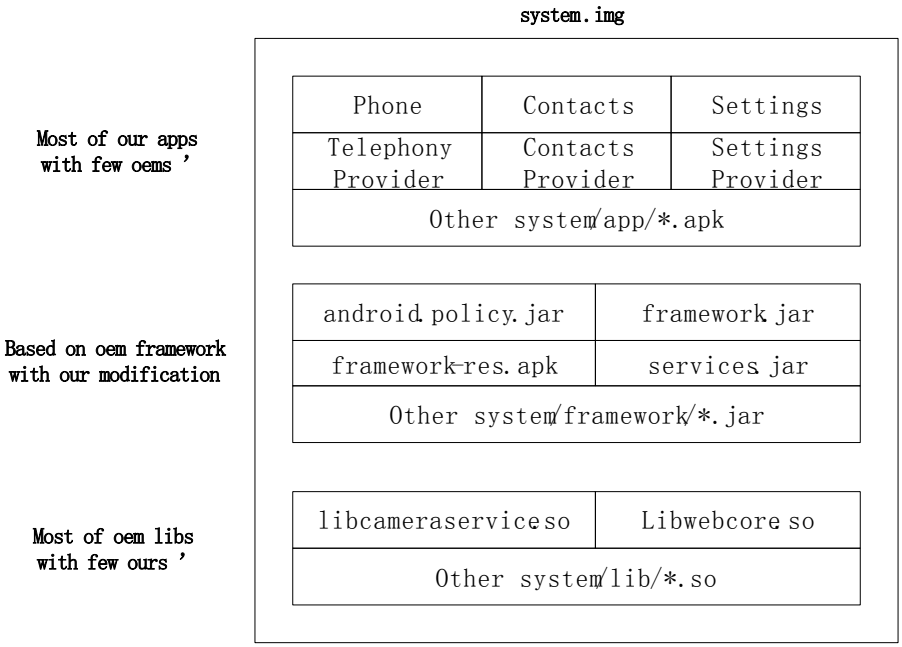


图 2.9 最终合并完后的 system.img 的组成

从 app 到底层库的纵向维度来看最终的 system.img 的组成，有以下特点：

* app 层绝大部分应用都是使用我们的。只有少部分与厂商 framework 代码结合紧密的应用，可能会使用厂商的。譬如 Camera.apk 或 Phone.apk，厂商的 framework 可能很难兼容我们的这几个应用；

* framework 层是我们在厂商基础上改的，增加了我们需要的接口和功能；

* lib 层与硬件关联紧密，所以一般都保持厂商的库不动。但我们会在原生基础上扩展很多库，这些只需要简单的拷贝到 system/lib 目录即可。

可以想象，越是深度定制，对 framework 的改动就会越多，但我们有的改动很难与厂商的改动保持兼容。这时候，就需要对整个定制过程分阶段进行，第一阶段，就需要挑选一些必要的 Bring Up 代码，这些代码通常包括：

* **影响系统起机的代码**。譬如，一个我们的系统程序，调用了我们定制的接口，而在厂商的代码中没有这个接口;

* **影响全局功能的代码**。譬如，在我们的资源合并后，发现系统的界面风格并没有改变。或者，我们添加的服务并没有成功运行。

正常情况下，这两部分代码合并完毕后，系统可以起机，能够运行大部分我们的应用程序，界面风格已经定制成功。至于部分应用程序运行出错，我们后续可以再逐步调试，一边合并其他功能性代码，一边验证。

3). Telephony 相关代码的特殊性

Telephony 相关的代码是指 framework/base/telephony/目录下的代码和上层通信相关的 app，包括电话 Phone、联系人 Contacts、彩信 Mms、流量监控 TrafficMonitor 以及相关的数据库。这部分代码之所以特殊，是因为厂商的改动较大，甚至不遵循 Android 原生的逻辑，这就导致我们在合并这部分代码时，较难保证兼容性。厂商改动这部分代码的动机主要有：

* **扩展通信功能**。作为手机最基础的功能，必然是厂商的必争之地；

* **支持双卡**。厂商的产品线也很多，一套代码兼容多个产品，就迫使必须改动 Android 原生的逻辑。

可以想象，将一个不具备双卡功能的电话 app，移植到一个支持双卡的手机上，保证兼容性是极其困难的。因此，在定制一款机型之前，了解其 Telephony 代码的结构，评估合并的难易程度，就显得很关键。合并 Telephony 代码评估标准是我们 app 调用的 framework 接口，是否能无误的合并到厂商代码中。

导致 app 与 framework 代码不兼容的场景主要有两种，如图 2.10 所示。

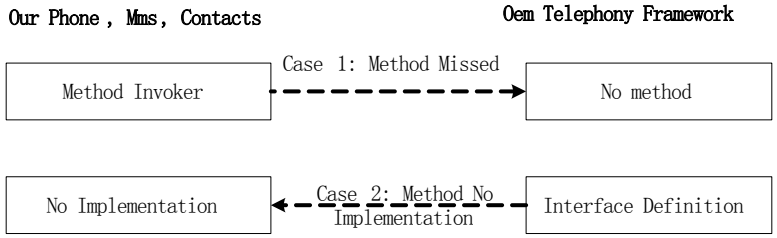


图 2.10 app 与 framework 不兼容的场景

* **场景 1**：我们 app 调用的接口，在厂商的 framework 中没有实现。这时候，我们要么在厂商的 framework 代码中，补充这个接口的实现；要么在我们的 app 中，去掉这个接口的调用。之所以出现这种场景，是因为我们也在 Android 原生基础上扩展了功能。

* **场景 2**：厂商 framework 定义的接口，在我们的 app 中没有实现。譬如，ITelephony.aidl 这个文件在 framework 的代码中，厂商一般都会在这个文件中扩展很多接口定义；但这个文件的接口实现在 Phone 的 PhoneInterfaceManager 类中，而我们的 phone 事先并不知道厂商会扩展什么接口。对于这种场景，通常只能在我们的 Phone app 代码中补充接口的实现。如果实现困难，则只能用厂商的 Phone app。

2.2.5 版本迭代与发布

当整个系统定制完毕，并通过功能测试，那就可以发布出去给大家使用了。然而，到此只是一锤子买卖，随着时间的推移，厂商会更新代码，我们也会有新的功能添加，为了能够让我们发布出去的定制包具备更稳定的性能和更丰富的功能，我们需要进行版本升级。所谓版本升级，是指我们不断地将厂商的新改动和我们的新改动重新合并到已经发布的定制包，然后再发布的过程。这个过程如图 2.11 所示。

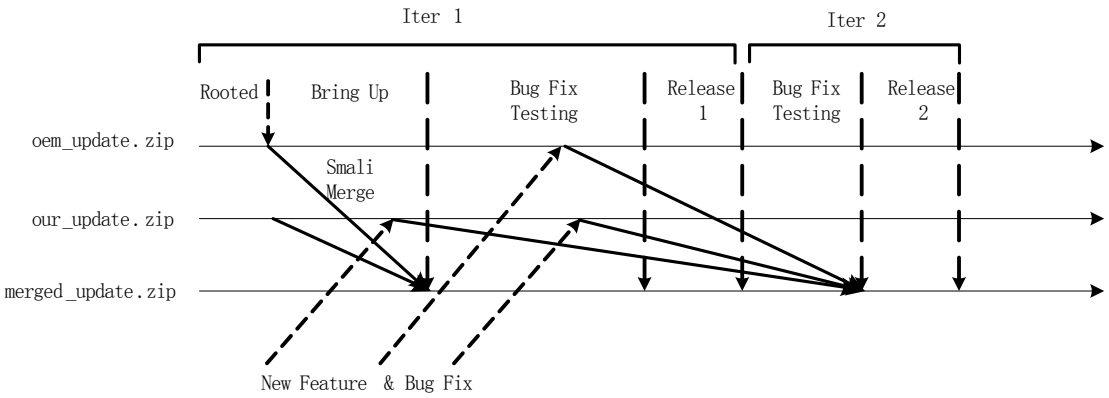


图 2.11 版本迭代和发布的过程

第一个迭代，通常会经过以下几个阶段：

* 准备好厂商的刷机包和我们的刷机包，获取厂商手机的 root 权限；

* 合并必要的 Smali 的代码，让手机能够正常起机，基本功能正常；

* 持续的 Bug 修复和测试，直到版本稳定；

* 第一次发布。譬如撰写刷机方法、版本特点等文案、将刷机包放到专业的论坛，供其他用户下载使用

在第一个迭代进行的过程中，我们或者厂商的包都可能已经发生更新，譬如修复了上一个版本的 Bug，或者新增了功能。而这部分改动，是不会影响到我们第一个版本的发布的，即我们第一个版本不包含这些改动。要包含这些改动，就要求继续演进我们的版本，进入第二个迭代，这个迭代通常会经过以下几个阶段：

* 合并我们或厂商新增加的代码；

* 持续的 Bug 修复和测试，直到新的版本稳定；

* 第二次发布。撰写系统更新文档，将更新包放到专业的论坛。

后续，还有可能发生第三个、第四个甚至更多的版本迭代，经过的阶段都与第二个迭代类似。我们发布的版本也跟随着厂商的进度在不断的更新。

3. 断点调试

3.1 调试环境

Smali 代码是可以断点调试的，可以查看函数调用栈，以及运行时的变量信息。断点调试能够帮助理解代码逻辑，也能够极大的提升解决 Bug 的效率。

工具	建议版本	下载地址
Ubuntu	10 或 12	http://www.ubuntu.com/download/desktop
Netbeans	6.8	https://netbeans.org/downloads/6.8/index.html
Apktool	1.4.2	https://code.google.com/p/android-apktool/
DDMS	SDK 16	http://developer.android.com/sdk/index.html

注意：

Netbeans 6.8 以上的版本，不支持 Smali 断点，原因未知。

Apktool 1.4.2 以上的版本，反编译参数-d 会导致异常，原因未知。

3.2 调试步骤

对于可运行的 apk 或 jar，进行 Smali 代码的断点调试，通常包括以下步骤，以 services.jar 为例。

1). 反编译可执行程序

```
adb pull system/framework/services.jar .
apktool d -d services.jar out/
```

-d 参数表示反编译的代码附带调试信息。带上-d 参数，反编译后，out/目录下会包含后缀名为 java(不是 smali)的文件。

如果是反编译 apk，则保 AndroidManifest.xml 的<application>标签中包含属性 android:debuggable="true"

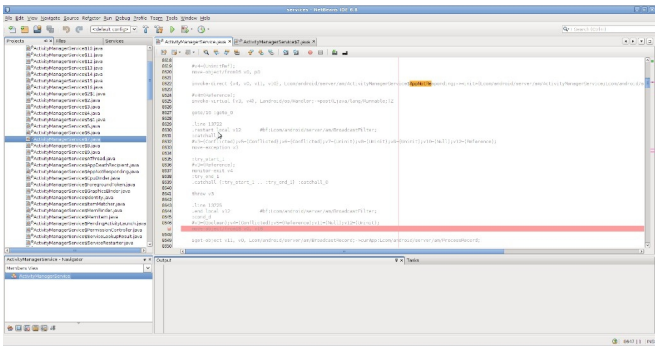
2). 创建 Netbeans 工程

打开 Netbeans，新建一个 Java 工程，选择反编译出的源代码目录 out/smali：

File->New Project->Java->Java Project with Existing Sources->Next

->Select out Directory->Next->Existing Source->Select out/smali Directory->Finish

这样，便成功新建了一个工程，打开工程文件，可以看到，虽然文件类型为 java 但文件内容还是 smali 代码。这时，已经可以在代码行的左侧单击打上断点。如图，红色行表示断点所在处。



3). 重编译安装

为了能够调试，我们需要重新把带调试信息的可执行程序安装到目标手机上，使用如下命令重新编译，签名，安装。重新编译时，需要带上-d 参数。

---

apktool b -d out/ services.jar

sign.sh services.jar

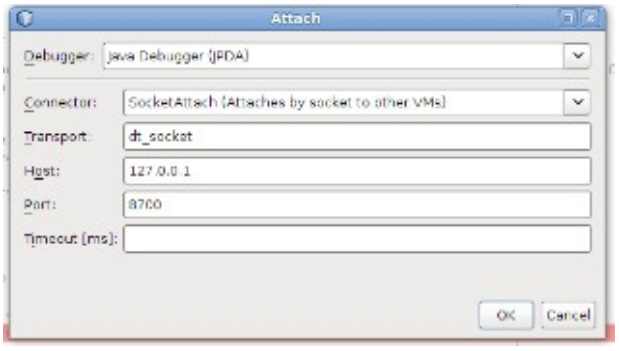
adb push services.jar.signed system/framework/services.jar

---

4). 启动调试

ddms 在 SDK_ROOT/tools 目录下，运行它，可以看到调试端口的信息，通常为 86XX/8700，单击选择待调试的进程。调试端口号是 Java 远程调试的概念，简而言之，就是待调试的程序运行在远程服务器上(目标手机)，提供一个调试端口(例如: 8632)；本地客户端(DDMS 环境)通过调试端口(8700)连接远程端口，进而实现调试信息的通讯。

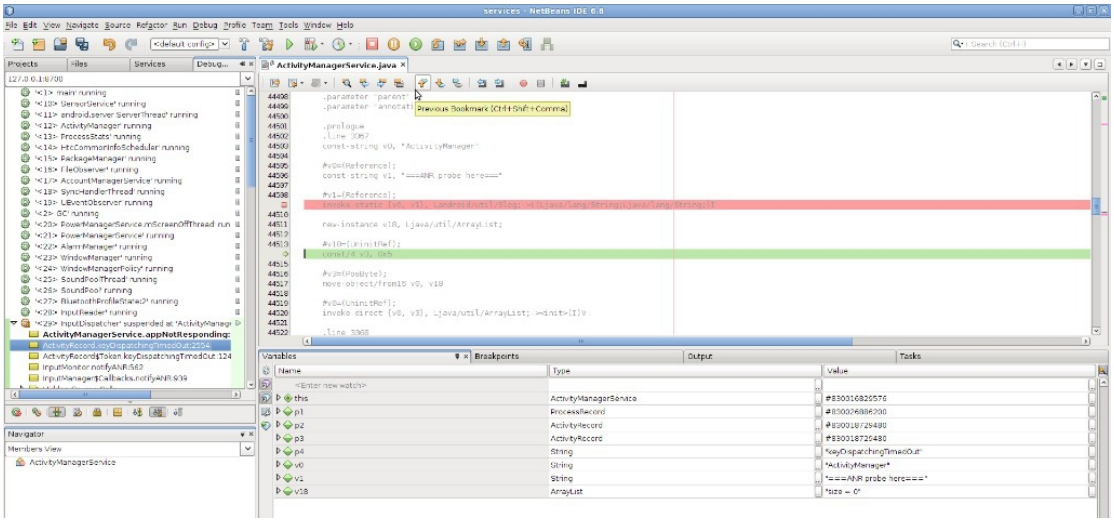
运行 Netbeans->Debug->Attach Debugger...，会弹出调试设置窗口：



设置 Host 为 127.0.0.1，端口 8700，单击 OK 后，如果成功连接，则会在 Netbeans 中显示调试面板，可见当前调试进程中运行的所有线程。

5). 查看运行时信息

操作手机，当运行到断点处时，程序会停下来；通过 F4-F8 以及其他组合键，控制代码运行。可以从 Debugging View 面板看到函数调用栈信息，Variables 面板看到变量的类型和值：



4. 实战演练

本章节通过一些具体的实例，详解通过 Smali 方法的应用。

4.1 Hello World

本例将演示如何反编译美丽说 meilishuo.apk，并在其显示界面显示一个 Toast。

首先，从美丽说的官网下载该 APK，并反编译，会在当前目录下生成 meilishuo 目录；

```
java -jar apktool.jar d meilishuo.apk
```

然后，找到程序的主界面 MainActivity.smali；定位到 onResume()函数；将以下代码的片段添加在 returen-void 这条语句之前。

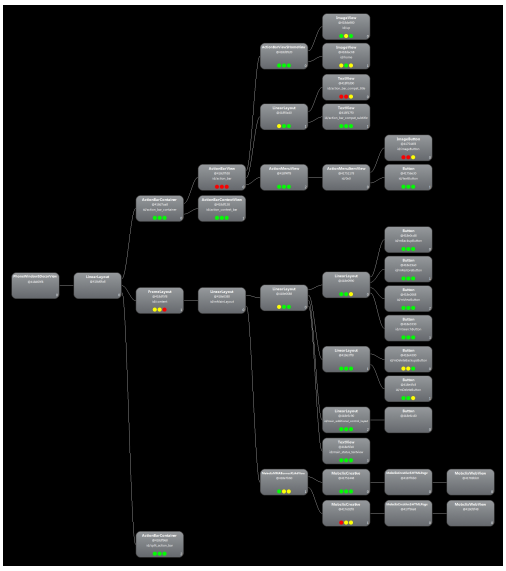
```
const-string v0, "Hello World. Smali Here"

const/4 v1, 0x1

invoke-static {p0, v0, v1}, Landroid/widget/Toast;->makeText(Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/widget/Toast;

move-result-object v0

invoke-virtual {v0}, Landroid/widget/Toast;->show()V
```

以上代码片段表示，弹出一个 toast，内容时”Hello World. Smali Here”

最后，重新编译 APK 并签名。安装 APK，可以启动应用程序后可以看到前后的对比。

4.2 去除 APK 中的广告

我们以网上下载一个 APK 为例：SMSBackup.apk，其主界面显示如下图所示，在主界面的最下方有一个广告横幅。通过 Hierachyview 这个工具，我们可以看到界面的布局，我们的目的是去除这个横幅。



分析界面布局，发现广告横幅其实就是嵌入在主界面的一个 View，它的类名是 MobclixMMABannerXLAdView。要在界面上加入这个 View，要么显示的在布局的 XML 文件中定义，要么通过代码动态的加载。采用 APK 定制章节中所描述的方法，反编译该 APK。

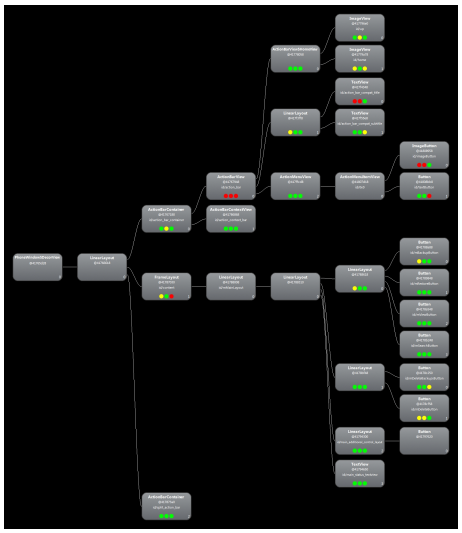
首先，找到主界面的布局文件为 res/layout/main.xml，用编辑器打开该文件，发现并没有显示的定义 MobclixMMABannerXLAdView，所以可以推断这个 View 是动态的在代码中加载的。

然后，将 MobclixMMABannerXLAdView 作为关键词，全局的检索，在检索结果中发现在 smali\com\riteshsahu\SMSBackupRestore\AdsHelper.smali 这个文件中有对这个关键词的引用。用编辑器打开 AdsHelper.smali 这个文件，定位到第 59 行为如下代码：

```
.line 26
.local v0, adView:Lcom/mobclix/android/sdk/MobclixAdView;
invoke-virtual {p1, v0, v2},
Landroid/view/ViewGroup;->addView(Landroid/view/View;Landroid/view/ViewGroup$LayoutParams;)V
```

这段 Smali 代码的意思就是调用 ViewGroupd.addView()函数，将 MobclixAdView 加载到当前的 ViewGroup 中，从而实现了将广告横幅加入显示界面。既然已经找到目标代码，我们只需要去掉该部分代码即可，.line 26 表示这部分 Smali 代码对应到源 Java 代码的第 26 行，我们直接将这部分代码块删除。

最后，重新编译 APK，签名后，将其安装到手机。可以发现界面上广告横幅已经去掉。通过 Hierachyview 查看界面布局，MobclixMMABannerXLAdView 也已经被去掉。



需要补充一句，既然我们对这个 APK 进行了修改，然后重新签名，再发布这个 APK 时，就相当于以我们的名义去发布。

4.3 对 Android 系统进行定制

本小节以 Sony LT26ii 为参考机型，介绍 Smali 移植 Baidu ROM 的方法。

1). 解锁 Bootloader

目前，已有一键解锁 BootLoader 的工具，譬如刷机精灵，卓大师等，可自行下载使用。如果一键解锁失败，可以尝试以下手工解锁的步骤：

1). 登录 Sony 官方解锁 BootLoader 网站：

http://unlockbootloader.sonymobile.com/unlock/step1

2). 按照官网向导，依次下一步。其中有一些注意项：

同意 Sony 的解锁责任声明，同时也意味着丧失手机保修资格

Sony 只要求输入 IMEI 号的前 14 位。通过拨号盘，输入*#06#，便可得到 15 位的 IMEI

号；

输入认证邮箱，Sony 会将解锁 Key 发送至认证邮箱

3). 上述步骤成功后，将 Sony 发送的邮件，内容包含解锁的 Key；

4). 通过命令行输入以下命令

```
adb reboot bootloader
fastboot -i 0x0fce oem unlock 0xKey
```

其中，-i 参数 0x0fce 表示 Sony 的 Vendor ID；Key 就是收到的 Sony 解锁邮件中的解锁密钥。

5). 如果上述过程没有报错，则说明解锁成功。在手机拨号界面输入*#*#7378423#*#*进入工程模式，选择 Service info ->Configuration，如果可以看到 Bootloader unlocked: Yes 说明已经解锁！ (已刷过机的用户无法查询)

2). 定制 Boot.img

Sony 的 boot.img 称为 kernel，格式为*.elf。Sony 自行定义了 kernel 的格式，提供制作 kernel 的脚本。访问以下网址便可下载 Sony 打包相关的工具：

http://developer.sonymobile.com/2012/03/20/xperia-s-open-source-archive-released-with-building-instructions/

打包脚本使用 Python 语言编写，所以需要需要下载安装 Python 的运行环境。

Sony 没有解包的脚本，可自行写一个脚本 unpackelf.py，脚本内容见附录。通过以下命令解包：

```
python unpackelf.py boot.img boot_dir
```

其中，boot.img 是待解的包，boot_dir 是输出目录。

成功解包后，boot_dir 目录下会有 0、1、2 三个文件。使用 file 命令便知，initramfs（或称 ramdisk）是 1，通过以下命令便可解开 ramdisk

```
mkdir ramdisk && cd ramdisk && gunzip < ../1| cpio -i
```

对 ramdisk 目录下的文件(init.rc, default.prop 等)做修改，譬如添加启动服务，内核 ROOT，默认启动 adbd 等。打开文件 default.prop，修改为以下属性，便可实现内核 ROOT：

```
ro.secure=0
```

```
ro.debuggable=1
```

修改后，重新压缩生成新的 ramdisk-new.gz，在 ramdisk 目录执行以下命令：

```
find . | cpio -o -H newc | gzip > ../ramdisk-new.gz
```

注意这里一定要指定 cpio 的格式为 newc.

通过 Sony 官方的打包脚本 mkelf.py 就可以重新打包 kernel.elf，即 boot.img。

```
mkelf.py -o kernel.elf zImage@0x40208000 ramdisk-new.img@0x41300000,ramdisk RPM.bin@0x20000,rpm
```

该脚本接收的参数 file@addr[, name]，file 就是输入文件名，譬如 zImage 就是内核文件名(boot.img-kernel)，ramdisk-new.img 就是文件系统；addr 是文件的地址，可以通过二进制查看：vim 直接打开 Image，然后输入命令%!xxd，可以格式化查看。

打包后，在手机进入 bootloader 后，通过以下命令，可以将新的 kernel 刷入手机：

```
fastboot -i 0x0fce flash boot kernel.elf
```

重启后，adb root 命令可以正确执行，说明已经获得手机的 root 权限。

3). 插桩 Smali

必要的 Bring Up 代码

为了能够让 Baidu 的 ROM 能够正常起机，有一些必要的 Feature 需要合并，否则会出现系统不能起机或 APP Crash 的情况。其他一些非必要的合并，只会导致到局部功能的异常，不会影响的系统起机。在实际的操作过程中，我们通常可以把必要的 Feature 先合并，让系统跑起来，再逐个合并并验证其他代码。必要的 Bring Up 代码包括：

*** Baidu 新增的服务的启动代码**

譬如系统 init 时启动的 Baidu 服务，framework-res-yi(百度资源)的加载，YiServiceLoader(百度服务)的加载等；

举例说明，修改 ramdisk/init.rc 文件，增加启机时的 Baidu 服务：

```
service backuprestore /system/bin/backuprestore
    class main
    socket backuprestore stream 600 system system
```

```
service WordSegService /system/bin/WordSegService
    class main
    oneshot
service propertystart /system/bin/property_start.sh
    disabled
    oneshot
service propertystop /system/bin/property_stop.sh
    disabled
    oneshot
service serviceext /sbin/serviceext
    class main
    oneshot
```

*** Baidu 新增但原厂没有的接口**

譬如 getIccCardType()这个接口是 Baidu 新增加的，非 MTK 平台的厂商代码中一般不会实现，我们需要把这个接口实现插桩到厂商代码中。这个接口比较特殊，从接口定义(IPhoneSubInfo.aidl)，到接口实现(PhoneSubInfo)，再到接口封装(IccCard, TelephonyManager)都需要增加这个接口。一共需要修改以下文件：

```
framework.jar.out/smali/android/telephony/TelephonyManager.smali
framework.jar.out/smali/com/android/internal/telephony/IPhoneSubInfo$Stub$Proxy.smali
framework.jar.out/smali/com/android/internal/telephony/IPhoneSubInfo.smali
framework.jar.out/smali/com/android/internal/telephony/IccCard.smali
framework.jar.out/smali/com/android/internal/telephony/PhoneSubInfo.smali
```

*** 原厂新增但 Baidu 没有实现的接口**

譬如 ITelephony.aidl 接口定义在 framework 中，但它的实现却在 Phone 的 PhoneInterfaceManager.java，如果有接口没有实现，但又被 framework 的其他代码调用，就会导致 Phone 频繁 Crash，这时，我们只需要在 PhoneInterfaceManager.smali 中补充缺失的接口实现即可。同样的还有 IStatusbar.aidl，它的接口实现在 SystemUI 的 CommandQueue.java

中。

对于这类接口，不同机型会有差别，Sony Lt26ii 就不需要补充 PhoneInterfaceManager.java 的接口。我们采用的策略是“**存在 BUG 的时候，再补充缺失的接口**”，即 BUG 驱动。如果存在要在 App 补充实现的接口，因为改动方式固定，所以我们也采用自动合并的方式。

以 HTC 的手机为例，在 ITelephony.smali 代码中扩展了 getHtcNetworkType()接口，但是在我们的 Phone app 中并没有实现，导致系统起机时 Phone app 不停的 crash，所以需要补充该接口。我们将需要实现的接口单独放在 PhoneInterfaceManager.part 文件中，在编译时，自动将该文件中的内容附加到原文件 PhoneInterfaceManager.smali 末尾，这样就能实现了自动合并缺失的接口。

```
.method public getHtcNetworkType([Ljava/lang/String;)I
    .locals 2
    .parameter "stack"
    .prologue
    .line 3041
    invoke-virtual {p0}, Lcom/android/phone/PhoneInterfaceManager;->getNetworkType()I
    move-result v1
    .line 3042
    .local v1, networkType:I
    const/4 v0, 0x0
    .line 3110
    .local v0, debugNetworkTypeShown:Z
    return v1
.end method
```

其他功能性代码

这部分代码不影响系统的正常使用。譬如联系人头像的优化、多主题，即便不合并这些代码，也只会导致局部功能的缺失。

对于这部分代码，我们提供自动合并的脚本。但针对一款新机型，我们还是以手工合并为主，因为不同厂商的代码存在很大的差异，譬如可能在厂商的代码中，变量 v3 对应的值

与我们合并的代码使用的变量 v3 意义不一样，如果自动合并，很容易就导致变量使用错误。最有效的方式是在理解厂商逻辑的基础上，自动合并部分代码，然后手工解决合并导致的 Bug。

以短信拦截为例，Baidu 的能够拦截黑名单的短信，如果不合并这部分代码，并不影响短信接收的基本功能；如果需要这部分功能，只需在 SMSDispatcher.smali 文件中修改短信接收的逻辑即可。代码片段如下所示：

```
.method private prehandleMsg(Lcom/android/internal/telephony/SmsMessageBase;[[B)Z
    .locals 3
    .parameter "sms"
    .parameter "pdus"
    .prologue
    const/4 v0, 0x1
    ...
.end method
```

在 dispatchMessage()中，调用 dispathPdu 方法之前，加入以下代码逻辑。可以看到，在原生逻辑上的修改，需要引用原来的变量 v11，所以需要根据上下文逻辑来判断 v11 具体是什么值。

```
    invoke-direct {p0, p1, v11}, Lcom/android/internal/telephony/SMSDispatcher;->prehandleMsg(Lcom/android/internal/telephony/SmsMessageBase;[[B)Z
    move-result v1
    if-eqz v1, :cond_9527
    const/4 v0, 0x01
    move v8, v0
    .line 506
    goto :goto_0
    .line 509
    :cond_9527
```

invoke-virtual {p0, v11},
Lcom/android/internal/telephony/SMSDispatcher;->dispatchPdus([[B)V

以上代码是否有效，还依赖于厂商的逻辑实现，有的厂商(譬如 HTC)，对接收短信的逻辑改动较大，所以合并我们的代码会比较困难。总而言之，功能代码的合并，依赖于对厂商逻辑的理解。

4). 制作刷机包失败时

并一定每次定制都能成功，在制作的刷机包导致系统无法起机，但又找不到原因时，我们还应该寻求能够“救砖”的方案，对于 LT26ii 这款机型只要强刷 SONY 官方的包，就可以救砖。按照如下步骤：

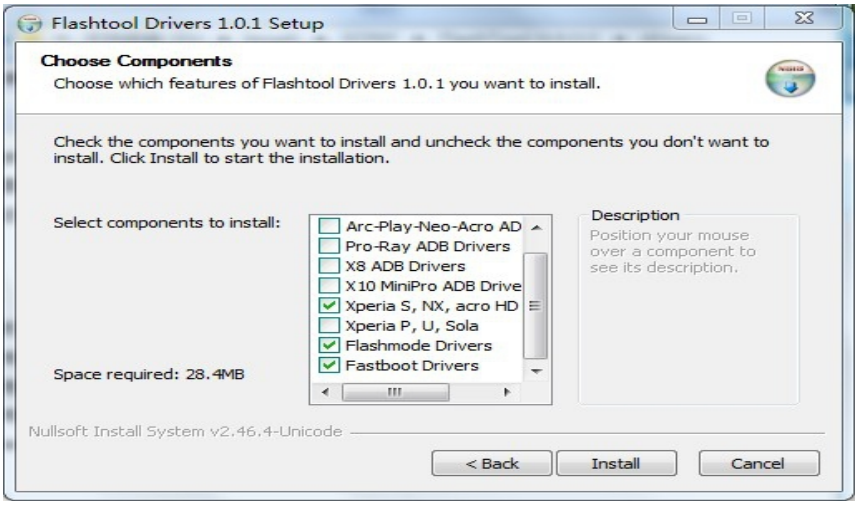
Step 1. 下载 Flashtool FlashTool V0.9.0.0 集成 32，64 位全部驱动

Step 2. 下载制作好的官方 FTF 固件 LT26ii XL_6.1.A.2.50_WORLD_liqucn.com.zip

解压之后得到 LT26ii XL_6.1.A.2.50_WORLD.ftf 将其放到 firmwares 文件夹下

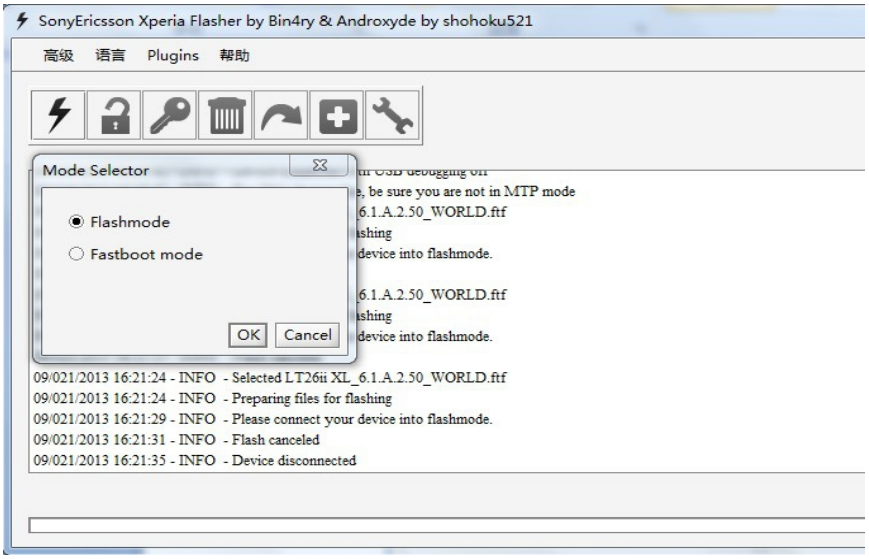
Step 3. 安装驱动

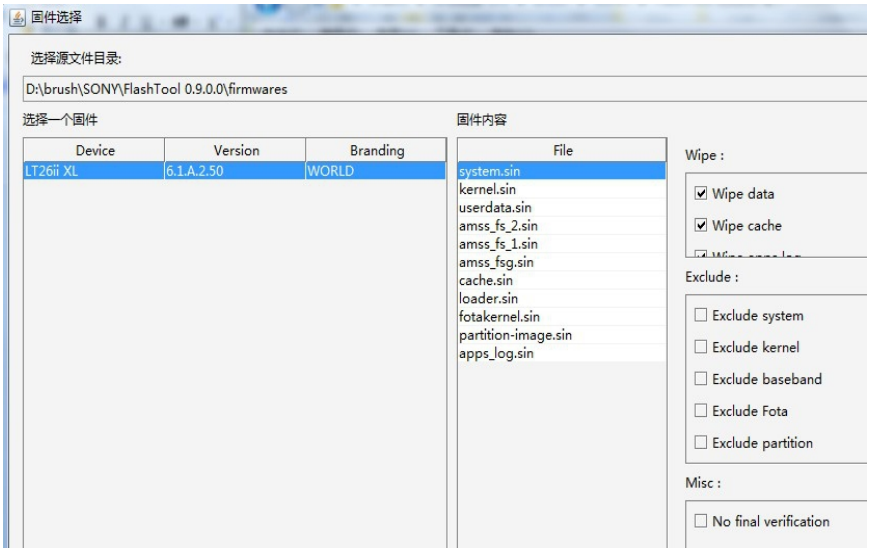
运行 drivers 目录下的 flashtool-drives.exe，多选下图所示的驱动安装。

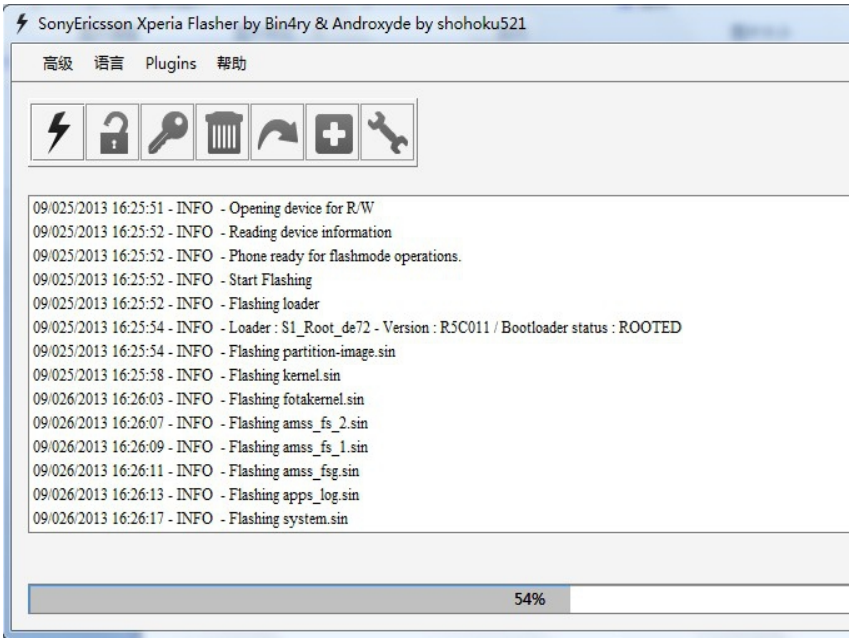


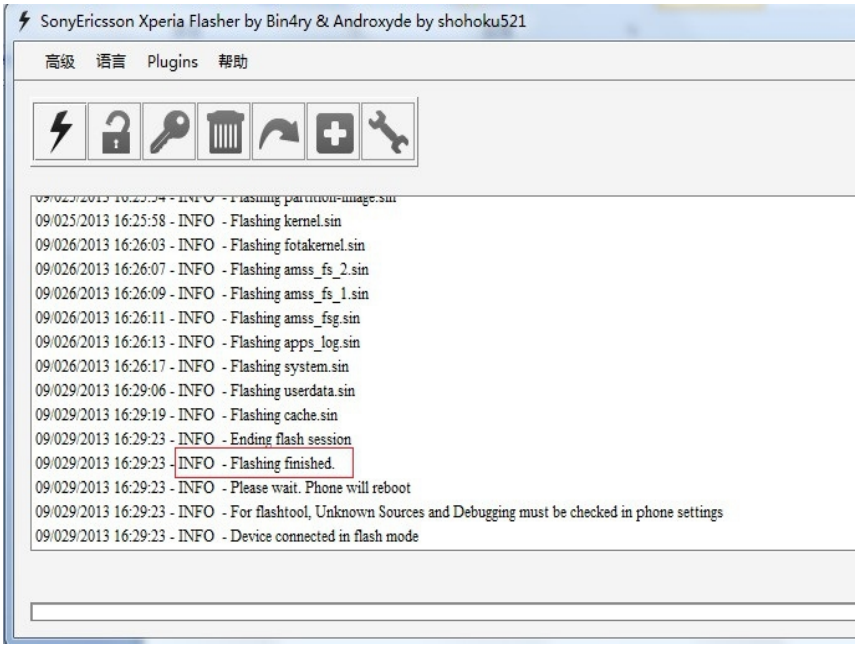
Step 4. 用 Flashtool 强刷官方固件

在关机状态下，按音量向下键，接上电源线，使手机进入 flash mode，此时呼吸灯为绿色。


SonyEricsson Xperia Flasher by Bin4ry & Androxyde by shohoku521
高级 语言 Plugins 帮助
Mode Selector
Flashmode
Fastboot mode
OK
Cancel
, be sure you are not in MTP mode
6.1.A.2.50_WORLD.ftf
device into flashmode.
09/021/2013 16:21:24 - INFO - Selected LT26ii XL_6.1.A.2.50_WORLD.ftf
09/021/2013 16:21:24 - INFO - Preparing files for flashing
09/021/2013 16:21:29 - INFO - Please connect your device into flashmode.
09/021/2013 16:21:31 - INFO - Flash canceled
09/021/2013 16:21:35 - INFO - Device disconnected


固件选择
选择源文件目录:
D:\brush\SONY\FlashTool 0.9.0.0\firmwares
选择一个固件
固件内容
Device
Version
Branding
LT26ii XL
6.1.A.2.50
WORLD
File
system.sin
kernel.sin
userdata.sin
amss_fs_2.sin
amss_fs_1.sin
amss_fsg.sin
cache.sin
loader.sin
fotakernel.sin
partition-image.sin
apps_log.sin
Wipe :
Wipe data
Wipe cache
Exclude :
Exclude system
Exclude kernel
Exclude baseband
Exclude Fota
Exclude partition
Misc :
No final verification


SonyEricsson Xperia Flasher by Bin4ry & Androxyde by shohoku521
高级 语言 Plugins 帮助
09/025/2013 16:25:51 - INFO - Opening device for R/W
09/025/2013 16:25:52 - INFO - Reading device information
09/025/2013 16:25:52 - INFO - Phone ready for flashmode operations.
09/025/2013 16:25:52 - INFO - Start Flashing
09/025/2013 16:25:52 - INFO - Flashing loader
09/025/2013 16:25:54 - INFO - Loader : S1_Root_de72 - Version : R5C011 / Bootloader status : ROOTED
09/025/2013 16:25:54 - INFO - Flashing partition-image.sin
09/025/2013 16:25:58 - INFO - Flashing kernel.sin
09/026/2013 16:26:03 - INFO - Flashing fotakernel.sin
09/026/2013 16:26:07 - INFO - Flashing amss_fs_2.sin
09/026/2013 16:26:09 - INFO - Flashing amss_fs_1.sin
09/026/2013 16:26:11 - INFO - Flashing amss_fsg.sin
09/026/2013 16:26:13 - INFO - Flashing apps_log.sin
09/026/2013 16:26:17 - INFO - Flashing system.sin
54%



进度条读取成功后，出现上面的红色英文，代表成功！此时，断开手机与电脑的连接，直接按手机的开机键。就刷回了 Sony 原生的系统。

5. 常见问题

本章节汇总了 Smali 开发中常见的一些问题，包括 Smali 定制 Android 系统后没有日志，无法进入系统，编译失败等。需要说明的是，这些问题只是针对具体机型遇到问题的一个参考，并不具备通用性。

5.1 开机启动异常

1). 无法显示开机日志

问题：没有开机日志，就没法定位系统不能启动的原因。通常是由于 adbd 在开机时没有启动导致，这时，adb devices 的结果显示为空。

解决：

Step 1. 解包 boot.img，进入 ramdisk 目录，查看 default.prop 文件。通常有两个属性与 adb 相关，检查这两个属性的值：

persist.service.adb.enable = 1，表示使能 adb。

persist.sys.usb.config = adb，表示将该属性设置为 adb。在与 usb 相关的 rc 文件中(通常为 init.usb.rc，但不同厂商会有区别)，会有针对该属性的配置，当该属性被设置成为指定值时，将会启动 adbd；

Step 2. 重新打包生成 boot.img，刷入系统，双清重启。

2). 应用签名问题导致无法进入系统

问题：在开机时，会启动一系列的系统服务，如果一些核心的服务无法启动，就会导致系统无法起来。最常见的是 SettingsProvider 签名问题，导致 Provider 无法启动，日志如下：

```
/system/app/SettingsProvider.apk changed; collecting certs
Package com.android.providers.settings has no signatures that match those in shared user android.uid.system; ignoring!
...
Failed to find provider info for settings
**********
********** Failure starting core service
java.lang.NullPointerException
```

at android.provider.Settings$NameValueCache.getString(Settings.java:718)
at android.provider.Settings$Secure.getString(Settings.java:2256)
at android.provider.Settings$Secure.getInt(Settings.java:2324)
at com.android.server.am.CoreSettingsObserver.populateCoreSettings(CoreSettingsObserver.java:93)
at com.android.server.am.CoreSettingsObserver.sendCoreSettings(CoreSettingsObserver.java:70)
at com.android.server.am.CoreSettingsObserver.<init>(CoreSettingsObserver.java:55)
at com.android.server.am.ActivityManagerService.installSystemProviders(ActivityManagerService.java:6419)
at com.android.server.ServerThread.run(SystemServer.java:208)

解决：

Step 1. 对整个刷机包(zip)和 SettingsProvider.apk 采用相同的签名，安装重新签名后的 apk；

Step 2. 双清后重启。主要是为了清除已有的 Settings 数据库，如果数据库已经存在，则不会重新构建，如果与已有的数据库版本不匹配，则会导致数据库升降级的异常。Settings 的数据库在/data/data/com.android.providers.settings/databases/目录。

3). 链接库问题导致系统服务无法启动

问题：由于使用厂商的 framework 和 so 库是紧密相关的，如果轻易替换厂商的一些 so 库，或者在厂商的 framework 中添加一些扩展的接口，则会导致链接库错误，即 java 层调用的接口在 so 库中没有。如以下 Log 所示，就表示 libwebcore.so 这个库中缺少接口。

*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'SEMC/LT26i_1257-6744/LT26i:4.0.4/6.1.A.2.45/0vd_zw:user/test-keys'
pid: 135, tid: 135 >>> zygote <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr deadd00d
r0 00000000 r1 00924390 r2 00000000 r3 00000000
r4 deadd00d r5 408ddc58 r6 0000020c r7 5b51292c
r8 00000000 r9 408d8f90 10 5b511816 fp 00000000
ip 00000000 sp beb413f0 lr 4087698f pc 4087698e cpsr 60000030
...
I/rmt_storage(195): rmt_storage fop(1): bytes transferred = 3145216
#00 pc 0005098e /system/lib/libdvm.so (dvmAbort)
#01 pc 00075ef4 /system/lib/libdvm.so
#02 pc 00076692 /system/lib/libdvm.so (_Z24dvmFindSystemClassNoInitPKc)
#03 pc 000767fe /system/lib/libdvm.so (_Z18dvmFindClassNoInitPKcP6Object)
/rmt_storage(195): rmt_storage fop(1): bytes transferred = 512
#04 pc 0005545a /system/lib/libdvm.so
#05 pc 0027ed26 /system/lib/libwebcore.so
#06 pc 00122c74 /system/lib/libwebcore.so (JNI_OnLoad)
#07 pc 0005aa14 /system/lib/libdvm.so (_Z17dvmLoadNativeCodePKcP6ObjectPPc)
#08 pc 00072c0c /system/lib/libdvm.so

...
beb41680 5b51292c /system/lib/libwebcore.so
beb41694 4089c697 /system/lib/libdvm.so

解决：由于链接库错误导致无法起机时，通常只需要替换回厂商的 so 库，或排查 framework 改动的接口即可。

5.2 通信功能异常

1). Phone Crash

问题：在厂商的代码中，缺少 Baidu Phone 需要的接口，或者厂商的接口没有在 PhoneInterfaceManager.java 这个类中实现，都会导致 Phone Crash。通常都会提示 NoSuchMethodError。如以下所示日志。

```
FATAL EXCEPTION: main
java.lang.NoSuchMethodError: com.android.internal.telephony.CallManager.registerForLineControlInfo
        at com.android.phone.CallNotifier.registerForNotifications(CallNotifier.java:1091)
        at com.android.phone.CallNotifier.<init>(CallNotifier.java:231)
         at com.android.phone.CallNotifier.init(CallNotifier.java:214)
        at com.android.phone.PhoneApp.onCreate(PhoneApp.java:556)
        at android.app.Instrumentation.callApplicationOnCreate(Instrumentation.java:999)
        at android.app.ActivityThread.handleBindApplication(ActivityThread.java:4151)
        at android.app.ActivityThread.access$1300(ActivityThread.java:130)
        at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1255)
        at android.os.Handler.dispatchMessage(Handler.java:99)
        at android.os.Looper.loop(Looper.java:137)
        at android.app.ActivityThread.main(ActivityThread.java:4745)
        at java.lang.reflect.Method.invokeNative(Native Method)
        at java.lang.reflect.Method.invoke(Method.java:511)
        at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:786)
        at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:553)
        at dalvik.system.NativeStart.main(Native Method)
Shutting down VM
```

解决：在对应缺少接口的类中，把接口实现补充。针对上述出错的日志信息，需要在 CallManager 这个类中，补充 registerForLineControlInfo 这个方法的实现。

5.3 编译、反编译、刷机的错误

1). 编译厂商的 framework.jar.out 出错

问题：在定制整个 Android 系统时，我们会往厂商的 framework 中注入很多代码，由于 framework.jar 包本来就已经包含了很多方法，再注入新的方法会导致超出一个 JAR 包的方法的数量限制。这时，会出现如下日志：

```
I: Checking whether sources has changed...
I: Smaling...
Exception in thread "main" org.jf.dexlib.Util.ExceptionWithContext: method index is too large.
        at org.jf.dexlib.Util.ExceptionWithContext.withContext(ExceptionWithContext.java:54)
        at org.jf.dexlib.Item.addExceptionContext(Item.java:177)
        at org.jf.dexlib.Item.writeTo(Item.java:120)
        at org.jf.dexlib.Section.writeTo(Section.java:119)
        at org.jf.dexlib.DexFile.writeTo(DexFile.java:716)
        at brut.androlib.src.DexFileBuilder.getAsByteArray(DexFileBuilder.java:75)
        at brut.androlib.src.DexFileBuilder.writeTo(DexFileBuilder.java:58)
        at brut.androlib.src.SmaliBuilder.build(SmaliBuilder.java:50)
        at brut.androlib.src.SmaliBuilder.build(SmaliBuilder.java:35)
        at brut.androlib.Androlib.buildSourcesSmali(Androlib.java:243)
        at brut.androlib.Androlib.buildSources(Androlib.java:200)
        at brut.androlib.Androlib.build(Androlib.java:191)
        at brut.androlib.Androlib.build(Androlib.java:174)
        at brut.apktool.Main.cmdBuild(Main.java:185)
        at brut.apktool.Main.main(Main.java:70)
Caused by: java.lang.RuntimeException: method index is too large.
        at org.jf.dexlib.Code.Format.Instruction35c.writeInstruction(Instruction35c.java:102)
        at org.jf.dexlib.Code.Instruction.write(Instruction.java:57)
        at org.jf.dexlib.CodeItem.writeItem(CodeItem.java:258)
        at org.jf.dexlib.Item.writeTo(Item.java:117)
        ... 12 more
code_item @0x1a6ef4 (Landroid/opengl/GLErrorWrapper;->glFramebufferRenderbufferOES(IIII)V)
```

解决：

如果一个 JAR 包的方法超出数量限制，则需要将该 JAR 包进行拆分。只需要挑选一个文件目录，譬如 android/app、android/widget，重新打包成一个新的 JAR 包即可。这个新的 JAR 包的命名并没有限制，我们可以任意取名，譬如 secondary_framework.jar。

通常，厂商也会对 framework 做拆分，因此，我们可以在厂商的代码中看到类似于 framework_ext.jar 等拆分后的 jar 包。在拆分后，需要把 JAR 包加到 BOOTCLASSPATH 环境变量中，这个变量在 boot.img 中定制，因此，一旦新拆分一个 JAR 包，就需要修改 boot.img。如果不修改 boot.img，我们可以利用厂商已经拆分的结果，不重新新建一个 JAR 包，而是继续将拆分的代码放到厂商已经拆分的 JAR 包中。

2). 编译资源出错

问题：资源编译时通过 APPT 完成的，通常资源编译出错时，都会提示 AAPT 无法正常执行。譬如以下日志，是由于资源格式不能正确解析导致。在源码环境下，可能 APPT 能够正常解析这些格式，但 APKTOOL 这个工具却不一定能正常解析，主要是编码格式问题导致。

```
I: Checking whether sources has changed...
I: Checking whether resources has changed...
I: Building resources...
error: Multiple substitutions specified in non-positional format; did you mean to add the formatted="false" attribute?
error: Unexpected end tag string
error: Multiple substitutions specified in non-positional format; did you mean to add the formatted="false" attribute?
error: Unexpected end tag string
Exception in thread "main" brut.androlib.AndrolibException:
brut.common.BrutException: could not exec command: [aapt, p, -F, /tmp/APKTOOL302094639260091014.tmp, -I
...
        at brut.androlib.res.AndrolibResources.aaptPackage(AndrolibResources.java:251)
        at brut.androlib.Androlib.buildResourcesFull(Androlib.java:324)
        at brut.androlib.Androlib.buildResources(Androlib.java:269)
        at brut.androlib.Androlib.build(Androlib.java:192)
        at brut.androlib.Androlib.build(Androlib.java:174)
        at brut.apktool.Main.cmdBuild(Main.java:185)
        at brut.apktool.Main.main(Main.java:70)
Caused by: brut.common.BrutException:
...
/media/data/source/smali/devices/u930/out/obj/system/app/Phone.bYp/AndroidManifest.xml]
        at brut.util.OS.exec(OS.java:83)
        at brut.androlib.res.AndrolibResources.aaptPackage(AndrolibResources.java:249)
        ... 6 more
```

解决：根据出错的日志信息，找到格式出错的资源年文件。上例中 AndoridManifest.xml 这个文件格式错误，修改文件格式，譬如去除多余的空格，重新编译即可。

3). 打包 system.img 出错

问题：在使用 make_ext4fs 这个工具打包 system.img 时，如果打包后的 system.img 会超出预先分配的大小，就会提示无法成功打包 system.img.

```
make_ext4fs -s -l 537919488 -a system /tmp/tmpddYzrZ /tmp/targetfiles-KWw86n/system
```

Creating filesystem with parameters:

Size: 537919488

Block size: 4096

Blocks per group: 32768

Inodes per group: 6576

Inode size: 256

Journal blocks: 2052

Label:

Blocks: 131072

Block groups: 4

Reserved block group size: 39

error: do_inode_allocate_extents: Failed to allocate 109 blocks

解决：找到 misc_info.txt 这个文件，修改 system_size，修改预先分配给 system.img 的阈值，重新编译。

参考资料

[1]. Smali. An assembler/disassembler for Android's dex format.

https://code.google.com/p/smali/

[2]. Dedexer. A disassembler tool for DEX files.

http://sourceforge.net/projects/dedexer/

[3]. Android-apktool.A tool for reverse engineering Android apk files.

https://code.google.com/p/android-apktool/

[4]. Dalvik VM Instruction Formats.

http://source.android.com/tech/dalvik/instruction-formats.html

[5]. Smali TypesMethodsAndFields.

https://code.google.com/p/smali/wiki/TypesMethodsAndFields

[6]. 丰生强. Android 软件安全与逆向分析. 2013-02-01

附录

A. Smali 语法

无论是普通类、抽象类、接口类或者内部类，在反编译出的代码中，它们都以单独的 smali 文件来存放。每个 smali 文件都由若干条语句组成，所有的语句都遵循着一套语法规范。在 smali 文件的头 3 行描述了当前类的一些信息，格式如下。

```
.class < 访问权限> [ 修饰关键字] < 类名>
.super < 父类名>
.source <源文件名>
```

打开 HelloWorld.smali 文件，头 3 行代码如下。

```
.class public LHelloWorld;
.super Landroid/app/Activity;
.source "HelloWorld.java"
```

第 1 行“.class”指令指定了当前类的类名。在本例中，类的访问权限为 public ，类名为“LHelloWorld;”，类名开头的 L 是遵循 Dalvik 字节码的相关约定，表示后面跟随的字符串为一个类。

第 2 行的“.super ”指令指定了当前类的父类。本例中的“LHelloWorld;”的父类为“Landroid/app/Activity;”。

第 3 行的“.source ”指令指定了当前类的源文件名。

前 3 行代码过后就是类的主体部分了，一个类可以由多个字段或方法组成。smali 文件中字段的声明使用“.field”指令。字段有静态字段与实例字段两种。静态字段的声明格式如下。

```
# static fields
.field < 访问权限> static [ 修饰关键字] < 字段名>:< 字段类型>
```

baksmali 在生成 smali 文件时，会在静态字段声明的起始处添加“static fields ”注释，smali 文件中的注释与 Dalvik 语法一样，也是以井号“# ”开头。“.field”指令后面跟着的是访

问权限，可以是 public、private 、protected 之一。修饰关键字描述了字段的其它属性，如 synthetic 。指令的最后是字段名与字段类型，使用冒号“：”分隔，语法上与 Dalvik 也是一样的。

实例字段的声明与静态字段类似，只是少了 static 关键字，它的格式如下。

```
# instance fields
.field < 访问权限> [ 修饰关键字] < 字段名>:< 字段类型>
```

比如以下的实例字段声明。

```
# instance fields
.field private btn:Landroid/widget/Button;
```

第 1 行的“instance fields ”是 baksmali 生成的注释，第 2 行表示一个私有字段 btn，它的类型为“Landroid/widget/Button; ”。

如果一个类中含有方法，那么类中必然会有相关方法的反汇编代码，smali 文件中方法的声明使用“.method ”指令。方法有直接方法与虚方法两种。直接方法的声明格式如下。

```
# direct methods
.method <访问权限> [ 修饰关键字] < 方法原型>
    <.locals>
    [.parameter]
    [.prologue]
    [.line]
    <代码体>
.end method
```

“direct methods”是 baksmali 添加的注释，访问权限和修饰关键字与字段的描述相同，方法原型描述了方法的名称、参数与返回值。“.locals ”指定了使用的局部变量的个数。“.parameter”指定了方法的参数，与 Dalvik 语法中使用“.parameters ”指定参数个数不同，每个“.parameter”指令表明使用一个参数，比如方法中有使用到 3 个参数，那么就会出现 3 条

“.parameter”指令。“.prologue ”指定了代码的开始处，混淆过的代码可能去掉了该指令。“.line ”指定了该处指令在源代码中的行号，同样的，混淆过的代码可能去除了行号信息。

虚方法的声明与直接方法相同，只是起始处的注释为“virtual methods”。

如果一个类实现了接口，会在 smali 文件中使用“.implements ”指令指出。相应的格式声明如下。

```
# interfaces
.implements < 接口名>
```

“# interfaces ”是 baksmali 添加的接口注释，“.implements ”是接口关键字，后面的接口名是 DexClassDef 结构中 interfacesOff 字段指定的内容。

如果一个类使用了注解，会在 smali 文件中使用“.annotation ”指令指出。注解的格式声明如下。

```
# annotations
.annotation [ 注解属性] < 注解类名>
    [ 注解字段 =  值]
.end annotation
```

注解的作用范围可以是类、方法或字段。如果注解的作用范围是类，“.annotation ”指令会直接定义在 smali 文件中，如果是方法或字段，“.annotation ”指令则会包含在方法或字段定义中。例如下面的代码。

```
# instance fields
.field public sayWhat:Ljava/lang/String;
    .annotation runtime LMyAnnoField;
        info = "Hello my friend"
    .end annotation
.end field
```

实例字段 sayWhat 为 String 类型，它使用了 MyAnnoField 注解，注解字段 info 值为

“Hello my friend”。将其转换为 Java 代码为：

```
@ MyAnnoField(info = "Hello my friend")
public String sayWhat;
```

B. 本文涉及到的术语

Baksmali：反编译器，取自冰岛语

Boot Loader： 操作系统内核运行之前运行的一段小程序

Dalvik： Android 虚拟机，取自冰岛语

Kernel：操作系统内核

Smali：编译器，取自冰岛语