百度云 OS 开发手册

目录

文档修改记录

修改时间	修改人	备注
2013/06/07	唐柳湘	文档初始内容
2013/06/13	张威平	ROOT 权限获取，新建机型工程
2013/07/02	唐柳湘	文档内容精简
2014/01/13	段启智	文档改版、内容梳理
2014/03/10	段启智	开发工具升级，文档更新
2014/04/18	段启智	去掉 make bringup，补充解冲突内容
2014/08/18	段启智	一键 ROM 适配
2014/11/17	段启智	去掉 autofix 的自动解冲突功能；文档局部修改

1 概览

1.1 定制原理

百度云 OS 制作的原理很简单，可以用四个字来形容“**拼包+插桩**”。

拼包，是指将百度的底包和厂商的底包有选择性地拼在一起，制作一个新的 ROM。我们定义好了应该修改或增加的文件（详情可见 build/configs/baidu_default.mk）， 然后用这些文件去覆盖厂商的文件，做成最后的 ROM。

完全以拼包的方式制作百度云的 ROM 存在局限。因为无法修改 so 库等文件，一旦存在兼容性的问题，将无法解决。因而我们需要 smali 插桩的方式在 framework 层注入百度的功能，解决兼容性的问题。

插桩，是指在 framework 层的 jar 包中，通过将百度自身的 feature 通过 smali 的方式插入原厂的 smali 代码中，从而让百度云 OS 能够在手机上跑起来。

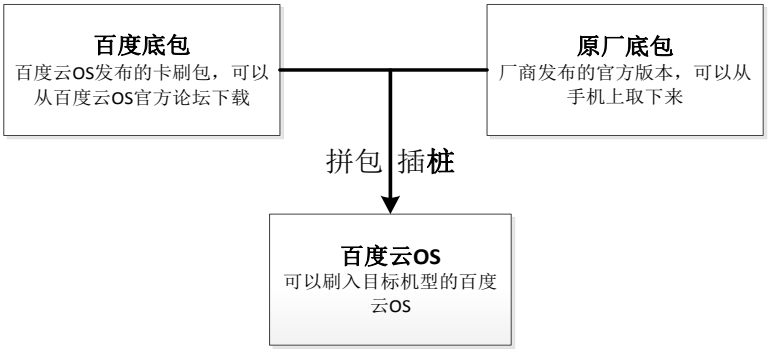

图 1. 百度云 OS 的定制原理

百度云 OS 的制作原理是“**拼包+插桩**”，所以选择合适的百度和原厂的底包很重要。如果两个底包不稳定或者差异太大，匹配起来的难度会很大，而且最终产出的 ROM 也会不稳定。

1.2 选择底包

原厂底包：是指可以刷入厂商手机的打包。可以通过工具，从手机上将该包取出来。

选择**厂商底包**应遵循以下原则：

* 稳定：一个 bug 少的原厂底包将会减少最终 ROM 的底层 bug。因为百度 ROM 的开发方式，决定了 framework 层以下的问题解决起来会比较麻烦。

* Android 版本：如原厂有 2.3 的版本、4.0 的版本，4.1 的版本，而能找到的适合移植的的百度底包只有 4.0 的，那建议选择原厂 4.0 的底包。

2 准备工作

2.1 操作系统

Linux 用户，Ubuntu 10.04+。

Windows 用户，可以通过虚拟机安装 Ubuntu 镜像。

2.2 系统环境

开发环境依赖于 jdk、git、curl、repo，安装方法如下。安装完成后，需要将这些工具配置到系统环境变量中。

```
$ sudo apt-get install sun-java6-jdk
$ sudo apt-get install git-core curl
$ mkdir ~/bin
$ curl https://raw.githubusercontent.com/baidurom/manifest/coron-4.2/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
```

如果不习惯使用命令行安装，也可以使用 Ubuntu 的软件中心图形界面安装 jdk 和 git，然后从 https://github.com/baidurom/manifest 下载 repo 程序。

2.3 开发环境

下载远程代码。包含开发工具以及所有参考教程。

```
$ repo init -u https://github.com/baidurom/manifest.git -b coron-4.2
$ repo sync
```

如果下载时，出现以下错误，多试几次即可(一般不超过 10 次，防火墙导致，难以避免)

```
fatal: Cannot get https://gerrit.googlesource.com/git-repo/clone.bundle
fatal: error [Errno 101] Network is unreachable
```

如果一直无法下载代码或者同步代码的过程太慢，可以使用如下命令：

```
$ repo init --repo-url https://github.com/baidurom/repo.git \
  -u https://github.com/baidurom/manifest.git -b coron-4.2 --no-repo-verify
$ repo sync --no-clone-bundle -c -j4
```

3 机型适配

3.1 前提条件

1) 获取 ROOT 权限。这是后续开发的基础。

ROOT 可以分为内核 ROOT 和漏洞 ROOT。

* 内核 ROOT 是指修改 boot.img 中的 default.prop 文件，直接使 adbd 服务以 ROOT 权限执行；

* 漏洞 ROOT，是指利用 Android 的漏洞，通过 su 程序获取 ROOT 权限。

一般市面上的一键 ROOT 程序，就是通过 Android 漏洞来获取 ROOT 权限。无论哪种 ROOT 方式，只要获取 ROOT 权限，都可以进行百度云 OS 的移植。

2) 环境初始化。初始化开发所需要的工具以及编译环境。

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

3) 新建机型目录。后续与该机型相关的开发工作，都将在该目录中完成。

```
$ mkdir devices/xxx -p          # xxx 为待开发的机型名称，譬如 lt26i， u930 等
$ cd devices/xxx                # 后续的开发工作都在机型目录中完成
```

3.2 新机适配

机型适配都在机型的目录中完成，所有适配的产出物也会生成在机型根目录下，后续执行命令时，都需要切换到机型的根目录。推荐开发者使用 Git 版本工具来管理机型目录，能为后续适配带来很多便利。

3.2.1 一键适配

通过 USB 线连接 PC 与待开发的手机，或将待移植的原厂底包命名为 ota.zip 放置于机型根目录。执行以下命令，便可开始一键适配。

```
$ coron fire
```

一键适配有一些关键的步骤，该命令会记录当前的执行到的步骤，如果其中某个步骤执行出错，只需要根据错误提示解决问题后，继续执行该命令即可。

1) config: 从手机或已有的原厂底包中拉取 boot.img 和 recovery.img，并自动解包 boot.img 和 recovery.img 生成机型配置文件 Makefile；

2) newproject: 从手机或已有的原厂底包中拉取原厂的所有文件，构建一个新机型工程；

3) patchall: 自动 Patch 需要植入的代码。既插桩；

4) autofix: 自动补充 Phone, SystemUI 等模块可能缺失的接口；

5) fullota: 编译机型，生成最终的卡刷包或可以刷入手机的 image。

以上适配步骤，也是都是可以单独执行的。下面详细说明每个步骤具体进行的操作。

3.2.2 具体流程

1) config 准备初始文件

开发一款新机型，需要 recovery.img 或 recovery.fstab，与该机型相应的配置文件 Makefile。使用以下命令，可以自动生成这些初始文件。

```
$ coron config
```

如果没有自行准备 boot.img 或 recovery.img，那该命令会尝试自动从手机中把这两个 image

文件拉下来。recovery.fstab 是 Android 的分区表，不同厂商的机型可能不一致，可以通过解开 recovery.img 得到。通常，只需要准备一个可用的 recovery.img，工具会自动完成解包过程。

该命令执行完毕后，会在机型目录下生成 Makefile 文件，包含了待开发机型的一些基本配置信息。当开发者需要对机型进行一些定制时，需要修改 Makefile 文件。

2) newproject 构建一个新的开发工程

在初始文件准备完毕后，便可以开始构建新的机型工程。

以下命令会自动从手机或者已有的原厂底包中抽取出所有原厂相关的文件。

```
$ coron newproject
```

该命令执行成功后，会根据 Makefile 配置在机型根目录下生成以下目录结构：

```
xxx/
├──── Makefile                    # 该机型的 Makefile 配置
├──── recovery.fstab              # 该机型的分区表信息，从 recovery.img 中提取
├──── framework-res/              # 反编译后的原厂的资源文件
├──── framework.jar.out/          # 反编译后的原厂 framework.jar 文件
├──── services.jar.out/           # 反编译后的原厂 services.jar 文件
├──── telephony-common.jar.out/   # 反编译后的原厂 telephony-common.jar 文件
├──── vendor/                     # 从原厂拉取的所有文件，包括 JAR 和 APK 等
├──── out/                        # 编译产出目录，中间文件等
```

3) patchall 自动 Patch 需要插桩的代码

在新机型工程生成完毕之后，执行以下命令自动将百度云 OS 的所有改动 Patch 到厂商的代码中。

```
$ coron patchall
```

该命令可以多次执行，不影响之前 patch 的内容。当需要重新 Patch 单个文件时，只需要将这个文件切换回厂商原来的代码，重新执行该命令即可。成功执行完该命令后，会在机型目录下，生成一个 autopatch 子目录，其结果如下所示：

```
autopatch/
├─── aosp/                  # AOSP(Android Open Source Project)的反编译代码
├─── bosp/                  # BOSP(Baidu Open Source Project)的反编译代码
├─── vendor_original/       # Patch 之前的厂商代码
├─── vendor_patched/        # Patch 之后的厂商代码
├─── patchall.xml           # autopatch 所涉及到的改动文件列表
```

冲突处理

如果厂商对 Android 原生改动较小，那就可以顺利 Patch。否则，将会产生 Patch 的冲突。所有产生冲突的文件会保存在机型目录下的 autopatch/reject/子目录中。

譬如 services.jar.out/smali/com/android/server/am/ActivityManagerService.smali，这个文件在自动 patch 的过程中产生了冲突，那么这个文件就会标记成冲突文件，并在 out/reject 目录中生成同样目录结构的文件，所有的冲突都以如下形式标注：

```
<<<<<<< VENDOR
   原厂(VENDOR)的代码块
=======
   百度(BOSP)的代码块
>>>>>>> BOSP
```

在解决冲突之前，需要理解三个目录：

* **厂商的代码**，即 VENDOR。在机型根目录下，开发者最终修改的都是厂商代码；

* **Android 原生代码**，即 AOSP。在 autopatch/aosp/目录下；

* **百度云 OS 代码**，即 BOSP。在 autopatch/bosp/目录下。

解决冲突一般分为三步：

第一步：确定冲突位置。对比【vendor_patched】和【reject】的代码可以找到冲突的具体位置，每一个冲突都有编号， 从 Conflict 0 开始。

第二步：确定百度云 OS 的改动。找到冲突位置后，对比【aosp】和【bosp】的代码来找出在冲突的位置处百度云 OS 是如何修改的 AOSP 的。

第三步：确定冲突的解法。之所以会产生冲突，是因为厂商(VENDOR)也在 AOSP 上有改动，一旦百度云 OS 和厂商在 AOSP 改动的位置相同，冲突就产生了。

4) autofix 自动补接口

一般而言，厂商会在 AOSP 上扩展一些接口，这些接口的定义在框架层的代码中，但这些接口的实现却在应用层的代码中，譬如 ITelephony.aidl, IStatusBar.aidl 就定义在 framework.jar 中，接口实现分别在 Phone.apk 和 SystemUI.apk 中。

由于应用层的 APK 都是使用 Baidu 的，并不会实现厂商定义的接口，所以会导致与这些缺失接口相关的运行时错误。使用以下命令，能自动补充部分缺失的接口。

```
$ coron autofix
```

5) fullota 编译机型生成卡刷包

在解决完冲突以后，可以通过以下命令开始编译整个机型，如果编译成功，将会生成一个卡刷包和其他相关的 image(譬如 system.img，boot.img 等)。

```
$ coron fullota
```

将编译成功后的产出刷入手机，当出现不能起机、卡在开机动画或者起机后出现某些应用程序 Crash，则需要分析开机日志，解决办法可以参考常见问题。

3.3 版本升级

1) upgrade 定期版本升级

在适配完一款新机型后，当百度云 OS 有版本更新时，会发布最新的代码改动，通过以下命令，便能自动升级到最新版本。

```
$ make upgrade
```

该命令默认会追踪 devices/base 目录，并在当前机型的 autopatch 目录下，生成 last_bosp 和 bosp 目录，根据升级改动的文件列表 upgrade.xml，自动将最新的改动插桩到目标机型中。

通过 BASE 参数，可以修改默认追踪的机型；通过 LAST_COMMIT 参数可以自定义从追踪机型的上一个提交开始升级。使用方式如下：

```
$ make upgrade    BASE=xxx    LAST_COMMIT=xxx
```

2) porting 从已有机型移植

对于一些**同厂商同系列**的机型，可以使用 porting 命令从已有机型上移植改动。这套方法源于我们的一个适配经验：**“通常，在一个机型上的改动，可以复用到其他机型。”**

譬如，百度已经开源了 Sony LT26i，HTC T328T 等机型的代码，那么再 Sony LT28i，HTCT329 这些机型时，就可以直接从已有机型移植所需要的提交。使用 make porting 命令，便可从已有机型移植所需要的改动。

```
$ make porting    BASE=xxx                # 从参考机型 xxx 移植所需要的改动
```

通过 BASE 参数，指定参考的机型，开发者需要根据经验，确保参考机型的改动能够复用到待开发的机型。该命令执行完后，会显示机型 xxx 的所有改动记录，开发者可以根据需要选择所需要的改动，选中后，该命令自动合并到待开发机型。**需要说明的是，BASE 指定的机型必须是通过 Git 版本管理起来的**。

为了提高开发效率，开发者可以将 BASE 参数配置在 Makefile 文件中，配置完毕后直接使用 make porting 命令即可。

```
$ make porting
```

4. 常用命令

命令	说明	备注

构建一个新机型相关的命令		
coron config	需连上 USB 数据线使用。 自动生成机型的 Makefile 配置。	之前的 makeconfig 命令依旧支持
coron newproject	需连上 USB 数据线使用。 构建一个新的机型工程。	根据 Makefile 配置，在机型目录下生成必要的文件。
coron patchall	自动 patch 所有改动。Patch 后需要手工解决自动 Patch 产生的冲突。	可以多次执行，不影响之前执行的结果。
编译相关的命令		
coron fullota	进行全编译，生成可用的卡刷包和分区镜像(boot.img, system.img 等)，	ota、otapackage、fullota 和 make 四种方法等价。
coron clean	清除机型编译的产物，包括 out/目录和其他临时文件。	
make XXX	编译单个模块。XXX 为需要修改的模块名称，譬如 framework、services、Phone 等，	在 Makfile 中配置需要插桩改动的模块，为 app 或者 jar
make XXX.phone	连接 USB 数据线使用。 编译单个模块，并将该模块 Push 到手机上对应的目录。	譬如 make services.phone 会将编译产出 services.jar 直接 Push 到 system/framework/
make clean-XXX	清除单个模块的编译产出。XXX 为修改的模块名。	
升级 ROM 版本相关的命令		
make porting	Makefile 配置 BASE 属性值之后使用。 从已有的机型移植所需要的提交。	同厂商同系列机型的插桩通常是可以复用的。通过该命令从已有机型移植改动记录。
make upgrade	当百度云 OS 发布最新改动后，通过该命令可以自动完成升级。	

5. 常见问题

5.1. ROOT 权限相关

1). 如何内核 ROOT？

解包 boot.img 后，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
```

2). 没有内核 ROOT，能够移植百度云 OS 吗？

移植百度云 OS 的前提是获取手机的 ROOT 权限，但并不一定需要内核 ROOT。可以通过已有的一些一键 ROOT 应用，譬如百度一键 ROOT、卓大师等工具，对手机进行一键 ROOT，使得 su 程序具备 ROOT 用户的权限。后续的移植过程与内核 ROOT 是完全一致的。

3). 如何从手机上获取 boot.img？

通过解包一个机型可用的 recovery.img，查看 recovery.fstab 文件，便可以知道分区信息，通过 cat 命令，便可将 boot 分区 cat 出来。但是，boot 分区信息有时候并不直观，需要视具体的机型而定。譬如 MTK 平台，可以通过以下方法：

第一步，将 su 放入 system 分区

a). 刷入第三方 recovery；

b). 如果有第三方“root 破解 ota 包”，可以刷入“root 破解 ota 包”； 一般第三方 root 破解包都是通过在 system 分区放入 su 和 super.apk 来实现 root；

c). 如果没有第三方“root 破解 ota 包”，那么就需要手动将 su 放进 system 分区； 解压 recovery.img，在 RAMDISK/etc/recovery.fstab 可以找到 system 分区挂载点；

```
# mount point fstype device [device2]
/system ext4 /dev/block/mmcblk0p5
```

手机在 recovery 状态下，将手机与 PC 连接，adb shell 进入手机，挂载 system 分区

```
$ adb shell
android:/# mount /dev/block/mmcblk0p5 /system
```

将准备好的 su push 到手机上

```
$ adb push su /system/bin/
$ adb shell chmod 4755 /system/bin/su
```

d)．重启手机，进入系统

第二步、通过 su 得到 boot 分区

a)． adb shell 进入手机

b)．执行 su 命令，获取超级用户权限。

```
$ adb shell
android:/$ su
android:/#              [如果此时命令提示符变成“#”，则表明 su 成功]
```

c)．找到 boot.img 分区

cat proc/dumchar_info 查看手机分区表

```
android:/# cat proc/dumchar_info
Part_Name     Size                  StartAddr              Type     MapTo
bootimg    0x0000000000600000 0x0000000000988000    2     /dev/block/mmcblk0
recovery   0x0000000000600000 0x0000000000f88000    2     /dev/block/mmcblk0
```

可以看到 boot.img 是从 0x988000 地址开始到 0xf88000 结束，大小为 0x0600000

d)．将 boot.img 分区 cat 到 sdcard 上 cat 整个分区时间会很长，执行了 5 秒后"Ctrl + C" 切断即可

```
android:/# cat /dev/block/mmcblk0 > /mnt/sdcard2/block_file
```

e)．将取得的 boot.img 分区，按分区地址裁剪

```
$ adb pull /mnt/sdcard2/block_file .
$ head -c 16285696 block_file > head_file
    [16285696 <- 0xf88000 不支持十六进制，转成十进制]
$ tail -c 6291456 head_file > boot.img
    [6291456 <- 600000 不支持十六进制，转成十进制]
```

f). 解压 boot.img 如果能成功解压 boot.img 得到 kernel 和 RAMDISK，则表明提取 boot.img 成功

5.2. 编译、反编译、刷机异常

1). method index is too large.

```
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)
```

问题原因：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。

解决方法：

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). /main.mk: No such file or directory

解决方法：没有初始化环境变量导致，请去根目录执行 source ./build/envsetup.sh

4). 编译资源出错

```
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:
...
```

问题原因：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。

解决办法：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。

5). 打包 system.img 出错

```
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
```

问题原因：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.

解决办法：找到 vendor/META/misc_info.txt 这个文件，修改预先分配给 system.img 的阈值，将其改大，重新编译即可。

6). All register args must fit in 4 bits

所有的寄存器参数必须是 4 bit 的，实际上也就是寄存器编号不能超过 16，和上面的意思是一样的。因为从 dalvik 字节码层面上来说，方法调用的参数都是用 4-bit 保存的。比如：invoke-direct {v20} …这样肯定是不行的，因为 v20 已经超过了 16。

需要注意的是用参数对象时有可能会忘记这个问题。比如：invoke-direct{p0}…，第一眼看去，是 p0，才第一个嘛，怎么会超过 16 呢？这里需要回忆一下前面讲过的内容了，方法参数实际也是占用寄存器的，而且是占用的最后的寄存器，如果一个方法使用了 n 个局部变量（寄存器），那么参数使用的寄存器是从 n+1 开始的。如果 n 超过 16 了，那 px 也就必然超过 16 了。怎么办呢？有两种方案：

方案一：将超过 16 的寄存器先 move 到 16 以下的寄存器，然后使用：

move-object/from16 v0, p0

invoke-direct {v0} …

方案二：改用 invoke-direct/range，range 系列 invoke 操作符参数都是用 1 字节（8-bit）来存的，因此能表示的范围可以达 256：

invoke-direct/range {p0 .. p0}

8). A list of registers can only have a maximum of 5 registers

invoke 调用时参数不能超过 5 个。超过 5 个参数的应该使用 invoke-range 系列操作符。

9). xx cannot fit into a nibble

数组大小不对。xx 是定义的数组大小，但实际初始化时的元素个数不是 xx 就会导致这个问题

5.2. 开机无法进入系统

1). 无法显示开机日志

问题原因：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。

解决办法：

第一步， 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；

第二步，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

```
/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)
```

问题原因：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。

解决办法：

第一步，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；

第二步，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

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
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
```

问题原因：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。

解决办法：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。

4). 一直停留在开机动画，无法进入系统(卡开机动画)

卡开机动画的原因有很多，通常情况是抛出 AndroidRuntime 的异常，导致 Android 的 SystemServer 无法启动。而抛出 AndroidRuntime 的场景又非常多，最好的解决办法就是根据开机日志，逐个解决 AndroidRuntime 的问题。

通常有以下情况会抛出 AndroidRuntime 异常：

* 插桩的代码不符合 Smali 语法规范，导致某个类出现 Java.lang.VerifyError。这种场景需要检查 Smali 代码改动，察看寄存器变量是否使用正确、是否存在 Smali 语法错误等；

* 缺少某个类、方法或者属性，导致出现 NoSuchClass, NoSuchMethod, NoSuchField 异常。这种场景需要在出现异常的位置，补充缺失的类、方法和属性；

* Smali 改动破坏原厂逻辑，导致系统重要的服务启动失败。由于厂商代码各异，插桩的代码很难具备通用性，所以对于所有厂商采用同样的 patch，可能会导致原厂逻辑遭到破坏。这种场景下，需要检索开机日志，找到启动失败的系统服务，根据代码逻辑排查问题。

5.3. 通信功能异常

1). Phone Crash

```
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
```

问题原因：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。

解决办法：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。可以新建一个 CallManager.smali.part，其内容为缺失的接口。具体请查看 Makefile 配置。

2). 无法使用数据网络

使用插桩的方法制作 ROM，在解决完所有 Phone Crash 的问题后，基本的通信功能应该是正常的，通常不会出现数据网络无法使用的情况。如果出现了，请确认百度底包和厂商底包的是否为相同的平台，譬如，均为高通双卡平台或者均为高通单卡平台。

5.4. 其他

1). 怎样修改百度 apk 的资源

例如我要修改 BaiduCamera.apk 的资源，则需要按以下步骤操作即可：

1、 在 Makefile 中增加以下配置：

    baidu_modify_apps += BaiduCamera

2、 然后在项目目录下创建目录 BaiduCamera/res

    mkdir –p BaiduCamera/res

3、 然后 BaiduCamera/res 增加对应需要替换 BaiduCamera.apk 的资源

2)．怎样修改厂商apk的资源

例如我要修改厂商的FMRadio.apk的图标，则需要按以下步骤操作即可：

1、 在Makefile中增加以下配置：

vendor_modify_apps += FMRadio

2、 按以下操作，反编译FMRadio.apk，在项目的根目录下执行：

```
$ ifdir vendor/system/framework/
$ apktool d vendor/system/app/FMRadio.apk
```

3、 根据android:icon的值，找到对应的icon图片，然后替换即可

3). 怎样修改 init.rc

在机型目录下的 vendor/BOOT/RAMDISK/init.rc 中修改。

附录

附录 A. 配置 Makefile

在新建项目的时候就需要配置好项目目录下的 Makefile，可基于一个相近的项目的 Makefile 进行修改，也可以通过 makeconfig 命令自动生成 Makefile，具体的可配置项如下：

* 编译选项配置

配置项为大写字母，其属性值为字符串。

配置项	属性值	示意
DALVIK_VM_BUILD	27、28、xx	目前只支持 27 和 28 两种 用于区分 libdvm.so 的版本，在做 odex 的时候会用到。
DENSITY	Xhdpi、hdpi、xx	分辨率
BAIDU_BASE_DEVICE	I9250、xx	百度底包

* 定制模块的配置

配置项为小写字母，其属性值为字符串数组，以空格区分不同字符串。

配置项	属性值	示意
vendor_modify_images	boot、recovery	配置原厂需要修改的 Image，会将机型根目录下的 boot.img 解包到 vendor/BOOT/目录，编译时会从 vendor/BOOT/目录重新打包
原厂 app/jar 相关的配置项		
vendor_remove_dirs	譬如 app、appbackup 等，属性值定义相对于目录 vendor/system/	配置原厂需要去除的目录，默认需要删除的目录在定义在 vendor_default.mk
vendor_remove_files	如 bin/zchgd 等，属性值定义相对于目录 vendor/system/	配置原厂需要去除的文件，默认需要删除的文件定义在 vendor_default.mk
vendor_saved_apps	如 Bluetooth、Nfc、Stk 等，属性值定义相对于目录 vendor/system/app/	配置原厂需要保留的 app，将采用原厂的签名类型进行签名
vendor_modify_apps	如 Settings.apk，属性值定义相对于目录 vendor/system/app	配置原厂需要修改的 app，譬如需要使用修改后的原厂 Settings，则需要将原厂的 Settings.apk 反编译到机型根目录，然后添加该配置项。
vendor_modify_jars	如 framework、services 等，属性值定义相对于目录 vendor/system/framework/	配置原厂需要修改的 jar，通常有 framework 和 services，即需要修改原厂的 framework.jar 和 services.jar，如果有原厂的其他 jar 包需要修改，则需要将 jar 包反编译到机型根目录，然后添加该配置项
百度 app/jar 相关的配置项		
baidu_saved_apps **baidu_saved_jars** **baidu_remove_apps**	百度的 app/jar 控制属性，属性值定义与 vendor_xxx_xxx 类似，	配置百度需要保留或删除的 app/jar

baidu_remove_jars		
baidu_modify_apps	如 Phone、SystemUI、Settings 等，属性值定义相对于目录 baidu/system/app/	配置百度需要修改的 app，这些 app 属于百度，但需要针对厂商进行适配，譬如向 Phone.apk 中添加缺失的接口等，修改都是以 patch 的方式体现，譬如 PhoneInterfaceManager.smali.part 文件中，定义缺失的接口 getIccCardType()
baidu_modify_jars	如 android.policy，属性值定义相对于目录 baidu/system/framework/	配置百度需要修改的 jar，这些 jar 属于百度，但需要针对厂商进行适配，譬如向 android.policy.jar 中添加缺失的接口等，修改方式同 baidu_modiy_apps
属性相关的配置项		
remove_property	如 dev.defaultwallpaper 等	将 remove_property 定义的属性从 system/buid.prop 文件中去除
override_property	如 qemu.hw.mainkeys=1 等	将 override_property 定义的属性写入 system/build.prop 文件

附录 B. 常用工具使用

1. 解包/打包 boot.img、recovery.img

bootimgpack 是一个智能解包和打包 Android boot.img 的工具，采用 Python 和 能够自适应不同手机厂商的 boot.img 格式,智能完成解包。bootimgpack 同时提供命令和图形界面两种操作方式。

a). 解包 boot.img 或 recovery.img

```
$ unpack_bootimg boot.img output/
```

b). 重新打包 boot.img 或 recovery

```
$ pack_bootimg BOOT/ boot.img
```

2. 安装 framework 资源（install framework，if)

安装 framework 资源是反编译 apk 前必须要做的工作！不然 apk 的编译会出现问题。

a). 安装单个 framework 资源 apk

```
$ apktool if framework-res.apk
```

b). 安装 framework 目录下所有资源

使用：ifdir FRAMEWORK_DIR

```
$ ifdir vendor/system/framework          # 安装 vendor/system/framework 下所有的资源 apk
```

3. 反编译目录下所有的 apk 或 jar

使用：decode_all.sh IN_DIR [OUT_DIR]

1. IN_DIR：apk 或 jar 所在的目录
2. OUT_DIR：反编译得到的 smali 的目标目录，如果为空，则为当前目录

```
$ decode_all source/system/framework/ baidu_framework
```

会将source/system/framework/所有的apk和jar包，通过apktool反编译到baidu_framework目录。

4. odex 转 dex(deodex，去除 odex)

用法：deodex.sh ODEX.zip

1. ODEX.zip：需要 odex 的 zip 包

```
$ deodex ota-full_ns.zip          # 去除 zip 包的 odex，输出为 ota-full_ns.zip.deodex.zip
```

5. 资源 id 与资源名称映射(idtoname.py, nametoid.py)

a). idtoname.py:资源 id 转换为#type@name#

用法：idtoname.py PUBLIC_XML SMALI_DIR

1. PUBLIC_XML：framework-res 里的 public.xml
2. SMALI_DIR：smali 文件所在目录

```
$ idtoname framework-res/res/values/public.xml framework.jar.ou
```

会将 framework.jar.out 目录下所有的 smali 文件里的资源 id 转换为 #type@name#的方式。注：这里的 framework-res 必须与 framework.jar.out 对应，不然会出现不一致的情况。

b). nametoid.py:将 #type@name#转换为资源 id

用法：nametoid.py PUBLIC_XML SMALI_DIR

1. PUBLIC_XML：framework-res 里的 public.xml
2. SMALI_DIR：smali 文件所在目录

作用：将 smali 代码里的资源#type@name#转换为 id 的方式，与 idtoname.py 相反

```
$ nametoid framework-res/res/values/public.xml framework.jar.out
```

会将 framework.jar.out 目录下所有的 smali 文件里的#type@name#[ta]转换为资源 id。

6. 快速 cd 命令

在 coron 根目录下，执行完 source build/envsetup.sh 之后，会生成以下命令：

croot：会 cd 到 coron 的根目录

cxxx：会 cd 到 devices 下 xxx 对应的机型目录

7. 定制脚本的作用

脚本名称	用途	参数
custom_app.sh	在编译 app、framework 里的 apk 的时候会调用，主要是为了方便对单个 apk 的编译进行定制。	1、 apk 的 basename：比如 Phone.apk 对应的则为 Phone 2、 smali 代码的路径：用 apktool 反编译得到的 smali 代码的路径
cusotm_jar.sh	在编译 framework 里的 jar 的时候会调用，主要是为了方便对单个 jar 的编译进行定制	1、 jar 的 basename：比如 framework.jar 对应的则为 framework 2、 smali 代码的路径：用 apktool 反编译得到的 smali 代码的路径
custom_targetfiles.sh	在打包成 target-files.zip 前会被	无

	调用，可以在 custom_targetfiles.sh 里面去除或增加一些文件等	