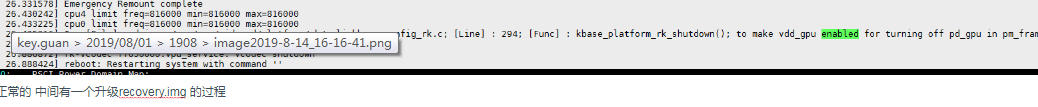
android的recovery模式

# NODO



正常的 中间有一个升级recovery.img 的过程

# 基本概念

eFuse：电子熔断，eFuse一次只有一根熔丝能够被[编程](https://baike.baidu.com/item/%E7%BC%96%E7%A8%8B)，就好像在硅片上建立了无数个交通岗哨，[控制信号](https://baike.baidu.com/item/%E6%8E%A7%E5%88%B6%E4%BF%A1%E5%8F%B7)的传输或停止，据悉这将把芯片中的电路运行效率提高上千倍。

本文将通过分析update.zip包在具体Android系统升级的过程，来理解Android系统中Recovery模式服务的工作原理。

* 先从update.zip包的制作开始，
* 然后是Android系统的启动模式分析，
* Recovery工作原理，如何从我们上层开始选择system update到重启到Recovery服务，以及在Recovery服务中具体怎样处理update.zip包升级的，我们的安装脚本updater-script怎样被解析并执行的等一系列问题。

分析过程中所用的Android源码是gingerbread0919（tcc88xx开发板标配的），测试开发板是tcc88xx。

# Recovery服务调起原理

从app层到Recovery服务

Android在升级系统时获得update.zip包的方式有两种。一种是离线升级，即手动拷贝升级包到SD卡中，通过settings-->About phone-->System Update-->选择从SD卡升级。

另一种是在线升级，即OTA Install(over the air)。在setting中触发升级按钮，检测服务器上的版本是否比本地版本高，如果低于或者等于本地版本，则提示已是最新版本，如果高于本地版本，则提示有新版本是否进行升级，点击立即升级，update.zip包一般被下载到系统的/CACHE分区或者Sd卡目录。

无论将升级包放在什么位置，在使用update.zip更新时都会重启并进入Recovery模式，然后启动recovery服务（/sbin/recovery）来安装我们的update.zip包。

为此，我们必须了解Recovery模式的工作原理以及Android系统重启时怎样进入Recovery工作模式而不是其他模式（如正常模式）

## ****从app层到Reboot****

在这里我们假设update.zip包已经制作好并拷贝到了SD卡中，并以Settings-->About Phone-->System Update-->Installed From SDCARD方式升级

原生DeviceInfoSettings是没有定义如何处理的，默认关闭了

String KEY\_UPDATE\_SETTING = "additional\_system\_update\_settings";

③rebootAndUpdate:在这个函数中Main System做了重启前的准备。继续跟踪下去会发现，在SystemUpdateService.java中的rebootAndUpdate函数中新建了一个线程，在这个线程中最后调用的就是RecoverySystem.installPackage(mContext,mFile)，我们的update.zip包也被传递进来了。

④RecoverySystem类：RecoverySystem类的源码所在文件路径为： **/frameworks/base/core/java/android/os/RecoverySystem.jav**a。我们关心的是installPackage(Context context,FilepackageFile)函数。这个函数首先根据我们传过来的包文件，获取这个包文件的绝对路径filename。然后将其拼成arg=“--update\_package=”+filename。它最终会被写入到BCB中。这个就是重启进入Recovery模式后，Recovery服务要进行的操作。它被传递到函数**bootCommand**(context,arg)。

⑤bootCommand()：在这个函数中才是Main System在重启前真正做的准备。主要做了以下事情，首先创建/cache/recovery/目录，删除这个目录下的command和log（可能不存在）文件在sqlite数据库中的备份。然后将上面④步中的arg命令写入到/cache/recovery/command文件中。下一步就是真正重启了。接下来看一下在重启函数reboot中所做的事情。

pm.reboot()：重启之前先获得了PowerManager（电源管理）并进一步获得其系统服务。然后调用了pm.reboot("recovery")函数。他就是/bionic/libc/bionic/reboot.cpp中的reboot函数。这个函数实际上是一个系统调用，即\_\_reboot(LINUX\_REBOOT\_MAGIC1,LINUX\_REBOOT\_MAGIC2,mode,NULL);从这个函数我们可以看出前两个参数就代表了我们的组合键，mode就是我们传过来的“recovery”。再进一步跟踪就到了汇编代码了，我们无法直接查看它的具体实现细节。但可以肯定的是 这个函数只将“recovery”参数传递过去了，之后将“boot-recovery”写入到了MISC分区的BCB数据块的command域中。这样在重启之后Bootloader才知道要进入Recovery模式。

在这里我们无法肯定Main System在重启之前对BCB的recovery域是否进行了操作。其实在重启前是否更新BCB的recovery域是不重要的，因为进入Recovery服务后，Recovery会自动去/cache/recovery/command中读取要进行的操作然后写入到BCB的recovery域中。

至此，Main System就开始重启并进入Recovery模式。在这之前Main System做的最实质的就是两件事，一是将“boot-recovery”写入BCB的command域，二是将--update\_package=/cache/update.zip”或则“--update\_package=/sdcard/update.zip”写入/cache/recovery/command文件中。下面的部分就开始重启并进入Recovery服务了。

android.Manifest.permission.***RECOVERY***

android系统提供给app层的升级主要在RecoverySystem中

1.先向bcb块写入升级命令（在/data/目录下有有特殊处理filepath会被改为@/cache/recovery/block.map，建议放在sdcard目录下）

2.先powerMS发送重启 Recovery命令

### 时序图

## 从reboot到Recovery服务

这个过程我们在上文（对照第一个图）已经讲过了。从Bootloader开始如果没有组合键按下，就从MISC分区读取BCB块的command域（在主系统时已经将“boot-recovery”写入）。然后就以Recovery模式开始启动。与正常启动不同的是Recovery模式下加载的镜像是recovery.img。这个镜像同boot.img类似，也包含了标准的内核和根文件系统。其后就与正常的启动系统类似，也是启动内核，然后启动文件系统。**在进入文件系统后会执行/init，init的配置文件就是/init.rc。这个配置文件来自bootable/recovery/etc/init.rc。**查看这个文件我们可以看到它做的事情很简单：

①设置环境变量。

②建立etc连接。

③新建目录，备用。

1. 挂载/tmp为内存文件系统tmpfs
2. 启动recovery（/sbin/recovery）服务。
3. 启动adbd服务(用于调试)。

这里最重要的就是当然就recovery服务了。在Recovery服务中将要完成我们的升级工作

# Recovery流程

Recovery服务毫无疑问是Recovery启动模式中最核心的部分。它完成Recovery模式所有的工作。Recovery程序对应的源码文件位于：/bootable/recovery/recovery.c。

## **Recovery的三类服务**

先看一下在这个源码文件中开始部分的一大段注释，这将对我们理解Recovery服务的主要功能有很大帮助。代码如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/mu0206mu/article/details/7465439)

1. /\*
2. \* The recovery tool communicates with the main system through /cache files.
3. \*   /cache/recovery/command - INPUT - command line for tool, one arg per line
4. \*   /cache/recovery/log - OUTPUT - combined log file from recovery run(s)
5. \*   /cache/recovery/intent - OUTPUT - intent that was passed in
6. \*
7. \* The arguments which may be supplied in the recovery.command file:
8. \*   --send\_intent=anystring - write the text out to recovery.intent
9. \*   --update\_package=path - verify install an OTA package file
10. \*   --wipe\_data - erase user data (and cache), then reboot
11. \*   --wipe\_cache - wipe cache (but not user data), then reboot
12. \*   --set\_encrypted\_filesystem=on|off - enables / diasables encrypted fs
13. \*
14. \* After completing, we remove /cache/recovery/command and reboot.
15. \* Arguments may also be supplied in the bootloader control block (BCB).
16. \* These important scenarios must be safely restartable at any point:
17. \*
18. \* FACTORY RESET
19. \* 1. user selects "factory reset"
20. \* 2. main system writes "--wipe\_data" to /cache/recovery/command
21. \* 3. main system reboots into recovery
22. \* 4. get\_args() writes BCB with "boot-recovery" and "--wipe\_data"
23. \*    -- after this, rebooting will restart the erase --
24. \* 5. erase\_volume() reformats /data
25. \* 6. erase\_volume() reformats /cache
26. \* 7. finish\_recovery() erases BCB
27. \*    -- after this, rebooting will restart the main system --
28. \* 8. main() calls reboot() to boot main system
29. \*
30. \* OTA INSTALL
31. \* 1. main system downloads OTA package to /cache/some-filename.zip
32. \* 2. main system writes "--update\_package=/cache/some-filename.zip"
33. \* 3. main system reboots into recovery
34. \* 4. get\_args() writes BCB with "boot-recovery" and "--update\_package=..."
35. \*    -- after this, rebooting will attempt to reinstall the update --
36. \* 5. install\_package() attempts to install the update
37. \*    NOTE: the package install must itself be restartable from any point
38. \* 6. finish\_recovery() erases BCB
39. \*    -- after this, rebooting will (try to) restart the main system --
40. \* 7. \*\* if install failed \*\*
41. \*    7a. prompt\_and\_wait() shows an error icon and waits for the user
42. \*    7b; the user reboots (pulling the battery, etc) into the main system
43. \* 8. main() calls maybe\_install\_firmware\_update()
44. \*    \*\* if the update contained radio/hboot firmware \*\*:
45. \*    8a. m\_i\_f\_u() writes BCB with "boot-recovery" and "--wipe\_cache"
46. \*        -- after this, rebooting will reformat cache & restart main system --
47. \*    8b. m\_i\_f\_u() writes firmware image into raw cache partition
48. \*    8c. m\_i\_f\_u() writes BCB with "update-radio/hboot" and "--wipe\_cache"
49. \*        -- after this, rebooting will attempt to reinstall firmware --
50. \*    8d. bootloader tries to flash firmware
51. \*    8e. bootloader writes BCB with "boot-recovery" (keeping "--wipe\_cache")
52. \*        -- after this, rebooting will reformat cache & restart main system --
53. \*    8f. erase\_volume() reformats /cache
54. \*    8g. finish\_recovery() erases BCB
55. \*        -- after this, rebooting will (try to) restart the main system --
56. \* 9. main() calls reboot() to boot main system
57. \*
58. \* SECURE FILE SYSTEMS ENABLE/DISABLE
59. \* 1. user selects "enable encrypted file systems"
60. \* 2. main system writes "--set\_encrypted\_filesystems=on|off" to
61. \*    /cache/recovery/command
62. \* 3. main system reboots into recovery
63. \* 4. get\_args() writes BCB with "boot-recovery" and
64. \*    "--set\_encrypted\_filesystems=on|off"
65. \*    -- after this, rebooting will restart the transition --
66. \* 5. read\_encrypted\_fs\_info() retrieves encrypted file systems settings from /data
67. \*    Settings include: property to specify the Encrypted FS istatus and
68. \*    FS encryption key if enabled (not yet implemented)
69. \* 6. erase\_volume() reformats /data
70. \* 7. erase\_volume() reformats /cache
71. \* 8. restore\_encrypted\_fs\_info() writes required encrypted file systems settings to /data
72. \*    Settings include: property to specify the Encrypted FS status and
73. \*    FS encryption key if enabled (not yet implemented)
74. \* 9. finish\_recovery() erases BCB
75. \*    -- after this, rebooting will restart the main system --
76. \* 10. main() calls reboot() to boot main system
77. \*/

          从注释中我们可以看到Recovery的服务内容主要有三类：

         ①FACTORY RESET，恢复出厂设置。

         ②OTA INSTALL，即我们的update.zip包升级。

         ③ENCRYPTED FILE SYSTEM ENABLE/DISABLE，使能/关闭加密文件系统。具体的每一类服务的大概工作流程，注释中都有，我们在下文中会详细讲解OTA INSTALL的工作流程。这三类服务的大概的流程都是通用的，只是不同操作体现与不同的操作细节。下面我们看Recovery服务的通用流程。

## **Recovery服务的通用流程**

在这里我们以OTA INSTALL的流程为例具体分析。并从相关函数的调用过程图开始，如下图:



### ui\_init()：

Recovery服务使用了一个基于framebuffer的简单ui（miniui）系统。这个函数对其进行了简单的初始化。在Recovery服务的过程中主要用于显示一个背景图片（正在安装或安装失败）和一个进度条（用于显示进度）。另外还启动了两个线程，一个用于处理进度条的显示（progress\_thread），另一个用于响应用户的按键（input\_thread）。

### get\_arg()：

这个函数主要做了上图中get\_arg()往右往下直到parse arg/v的工作。我们对照着流程一个一个看。

①get\_bootloader\_message()：主要工作是根据分区的文件格式类型（mtd或emmc）从MISC分区中读取BCB数据块到一个临时的变量中。

②然后开始判断Recovery服务是否有带命令行的参数（/sbin/recovery，根据现有的逻辑是没有的），若没有就从BCB中读取recovery域。如果读取失败则从/cache/recovery/command中读取然后：

这样这个BCB的临时变量中的recovery域就被更新了。在将这个BCB的临时变量写回真实的BCB之前，又更新的这个BCB临时变量的command域为“boot-recovery”。这样做的目的是如果在升级失败（比如升级还未结束就断电了）时，系统在重启之后还会进入Recovery模式，直到升级完成。

②在这个BCB临时变量的各个域都更新完成后使用set\_bootloader\_message()写回到真正的BCB块中

这个过程可以用一个简单的图来概括，这样更清晰：



### parserargc/argv：

解析我们获得参数。注册所解析的命令（register\_update\_command），在下面的操作中会根据这一步解析的值进行一步步的判断，然后进行相应的操作。

### if(update\_package)：

判断update\_package是否有值，若有就表示需要升级更新包，此时就会调用install\_package()（即图中红色的第二个阶段）。在这一步中将要完成安装实际的升级包。这是最为复杂，也是升级update.zip包最为核心的部分。我们在下一节详细分析这一过程。为从宏观上理解Recovery服务的框架，我们将这一步先略过，假设已经安装完成了。我们接着往下走，看安装完成后Recovery怎样一步步结束服务，并重启到新的主系统的。

### if(wipe\_data/wipe\_cache)：

这一步判断实际是两步，在源码中是先判断是否擦除data分区（用户数据部分）的，然后再判断是否擦除cache分区。值得注意的是在擦除data分区的时候必须连带擦除cache分区。在只擦除cache分区的情形下可以不擦除data分区。

简单来说就是data包含了cache

### maybe\_install\_firmware\_update()：

如果升级包中包含/radio/hboot firmware的更新，则会调用这个函数。查看源码发现，在注释中（OTA INSTALL）有这一个流程。但是main函数中并没有显示调用这个函数。目前尚未发现到底是在什么地方处理。但是其流程还是向上面的图示一样。即，① 先向BCB中写入“boot-recovery”和“—wipe\_cache”之后将cache分区格式化，然后将firmware image 写入原始的cache分区中。②将命令“update-radio/hboot”和“—wipe\_cache”写入BCB中，然后开始重新安装firmware并刷新firmware。③之后又会进入图示中的末尾，即finish\_recovery()。

### prompt\_and\_wait()：

这个函数是在一个判断中被调用的。其意义是如果安装失败（update.zip包错误或验证签名失败），则等待用户的输入处理（如通过组合键reboot等）。

### finish\_recovery()：

这是Recovery关闭并进入Main System的必经之路。其大体流程如下：



① 将intent（字符串）的内容作为参数传进finish\_recovery中。如果有intent需要告知Main System，则将其写入/cache/recovery/intent中。这个intent的作用尚不知有何用。

② 将内存文件系统中的Recovery服务的日志（/tmp/recovery.log）拷贝到cache（/cache/recovery/log）分区中，以便告知重启后的Main System发生过什么。

③ 擦除MISC分区中的BCB数据块的内容，以便系统重启后不在进入Recovery模式而是进入更新后的主系统。

④ 删除/cache/recovery/command文件。这一步也是很重要的，因为重启后Bootloader会自动检索这个文件，如果未删除的话又会进入Recovery模式。原理在上面已经讲的很清楚了。

### reboot()：

这是一个系统调用。在这一步Recovery完成其服务重启并进入Main System。这次重启和在主系统中重启进入Recovery模式调用的函数是一样的，但是其方向是不一样的。所以参数也就不一样。查看源码发现，其重启模式是RB\_AUTOBOOT。这是一个系统的宏。

至此，我们对Recovery服务的整个流程框架已有了大概的认识。下面就是升级update.zip包时特有的也是Recovery服务中关于安装升级包最核心的第二个阶段。即我们图例中的红色2的那个分支。

**Recovery服务的核心install\_package函数**

## **核心函数install\_package（update.zip特有）**

和Recovery服务中的wipe\_data、wipe\_cache不同，install\_package()是升级update.zip特有的一部分，也是最核心的部分。在这一步才真正开始对我们的update.zip包进行处理。下面就开始分析这一部分。还是先看图例：



这一部分的源码文件位于： /bootable/recovery/install.c。这是一个没有main函数的源码文件

另，对于新版本的系统android5.1发现，这一部分已经拆为install和verfication这两部分了

下面顺着上面的流程图和源码来分析这一流程：

### roots.ensure\_path\_mount()：

先判断所传的update.zip包路径所在的分区是否已经挂载。如果没有则先挂载。

### verifier.load\_keys()：

加载公钥源文件，路径位于/res/keys。这个文件在Recovery镜像的根文件系统中。

### verify\_file()：

对升级包update.zip包进行签名验证。此处可以加入自己的加密逻辑

### mzOpenZipArchive()：

打开升级包，并将相关的信息拷贝到一个临时的ZipArchinve变量中。这一步并未对我们的update.zip包解压。

### try\_update\_binary()：

在这个函数中才是对我们的update.zip升级的地方。这个函数一开始先根据我们上一步获得的zip包信息，以及升级包的绝对路径将update\_binary文件拷贝到内存文件系统的/tmp/update\_binary中。以便后面使用。

### ⑥pipe()：

try\_update\_binary中调用：创建管道，用于下面的子进程和父进程之间的通信。

### fork()：

创建子进程。其中的子进程主要负责执行binary（execv(binary,args)，即执行我们的安装命令脚本），父进程负责接受子进程发送的命令去更新ui显示（显示当前的进度）。子父进程间通信依靠管道。

### 父进程

⑧其中，在创建子进程后，父进程有两个作用。一是通过管道接受子进程发送的命令来更新UI显示。二是等待子进程退出并返回INSTALL SUCCESS。其中子进程在解析执行安装脚本的同时所发送的命令有以下几种：

       progress  <frac> <secs>：根据第二个参数secs（秒）来设置进度条。

           set\_progress  <frac>：直接设置进度条，frac取值在0.0到0.1之间。

            firmware <”hboot”|”radio”><filename>：升级firmware时使用，在API  V3中不再使用。

           ui\_print <string>：在屏幕上显示字符串，即打印更新过程。

           execv(binary,args)的作用就是去执行binary程序，这个程序的实质就是去解析update.zip包中的updater-script脚本中的命令并执行。由此，Recovery服务就进入了实际安装update.zip包的过程。

## 开始升级

updater-script

### ****update\_binary的执行过程分析****

上一篇幅中的子进程所执行的程序binary实际上就是update.zip包中的update-binary。我们在上文中也说过，Recovery服务在做这一部分工作的时候是先将包中update-binary拷贝到内存文件系统中的/tmp/update\_binary，然后再执行的。update\_binary程序的源码位于gingerbread0919/bootable/recovery/updater/updater.c，

通过上面的源码来分析下这个程序的执行过程：

1. 函数参数以及版本的检查：当前updater binary API所支持的版本号有1，2，3这三个。
2. 获取管道并打开：在执行此程序的过程中向该管道写入命令，用于通知其父进程根据命令去更新UI显示。
3. 读取updater-script脚本：从update.zip包中将updater-script脚本读到一块动态内存中，供后面执行。
4. Configure edify’s functions：注册脚本中的语句处理函数，即识别脚本中命令的函数。主要有以下几类

RegisterBuiltins()：注册程序中控制流程的语句，如ifelse、assert、abort、stdout等。

RegisterInstallFunctions()：实际安装过程中安装所需的功能函数，比如mount、format、set\_progress、set\_perm等等。

RegisterDeviceExtensions()：与设备相关的额外添加項，在源码中并没有任何实现。FinishRegistration()：结束注册。

1. Parsethe script：调用yy\*库函数解析脚本，并将解析后的内容存放到一个Expr类型的python类中。主要函数是yy\_scan\_string()和yyparse()。
2. 执行脚本：核心函数是Evaluate（），它会调用其他的callback函数，而这些callback函数又会去调用Evaluate去解析不同的脚本片段，从而实现一个简单的脚本解释器。
3. 错误信息提示：最后就是根据Evaluate（）执行后的返回值，给出一些打印信息。

这一执行过程非常简单，最主要的函数就是Evaluate。它负责最终执行解析的脚本命令。而安装过程中的命令就是updater-script。

### ****updater-script脚本语法简介****

目前update-script脚本格式是edify，其与amend有何区别，暂不讨论，我们只分析其中主要的语法，以及脚本的流程控制。

我们顺着所生成的脚本来看其中主要涉及的语法。

1.assert(condition)：如果condition参数的计算结果为False，则停止脚本执行，否则继续执行脚本。

2.show\_progress(frac,sec)：frac表示进度完成的数值，sec表示整个过程的总秒数。主要用与显示UI上的进度条。

3.format(fs\_type,partition\_type,location)：fs\_type，文件系统类型，取值一般为“yaffs2”或“ext4”。Partition\_type，分区类型，一般取值为“MTD”或则“EMMC”。主要用于格式化为指定的文件系统。事例如下：format（”yaffs2”,”MTD”,”system”）。

4.mount(fs\_type,partition\_type,location,mount\_point)：前两个参数同上，location要挂载的设备，mount\_point挂载点。作用：挂载一个文件系统到指定的挂载点。

5.package\_extract\_dir(src\_path,destination\_path)：src\_path，要提取的目录，destination\_path目标目录。作用：从升级包内，提取目录到指定的位置。示例：package\_extract\_dir(“system”,”/system”)。

6.symlink(target,src1,src2,……,srcN)：target，字符串类型，是符号连接的目标。SrcX代表要创建的符号连接的目标点。示例：symlink(“toolbox”,”/system/bin/ps”),建立指向toolbox符号连接/system/bin/ps，值得注意的是，在建立新的符号连接之前，要断开已经存在的符号连接。

7.set\_perm(uid,gid,mode,file1,file2,……,fileN)：作用是设置单个文件或则一系列文件的权限，最少要指定一个文件。

8.set\_perm\_recursive(uid,gid,mode,dir1,dir2,……,dirN)：作用同上，但是这里同时改变的是一个或多个目录及其文件的权限。

9.package\_extract\_file(srcfile\_path,desfile\_paht)：srcfile\_path，要提取的文件，desfile\_path，提取文件的目标位置。示例：package\_extract\_file(“boot.img”,”/tmp/boot.img”)将升级包中的boot.img文件拷贝到内存文件系统的/tmp下。

10.write\_raw\_image(src-image,partition)：src-image源镜像文件，partition，目标分区。作用：将镜像写入目标分区。示例：write\_raw\_image(“/tmp/boot.img”,”boot”)将boot.img镜像写入到系统的boot分区。

11.getprop(key)：通过指定key的值来获取对应的属性信息。示例：getprop(“ro.product.device”)获取ro.product.device的属性值。

### updater-script脚本执行流程分析

先看一下在测试过程中用命令make otapackage生成的升级脚本如下：

1. assert(!less\_than\_int(1331176658, getprop("ro.build.date.utc")));
2. assert(getprop("ro.product.device") == "tcc8800" ||
3. getprop("ro.build.product") == "tcc8800");
4. show\_progress(0.500000, 0);
5. format("yaffs2", "MTD", "system");
6. mount("yaffs2", "MTD", "system", "/system");
7. package\_extract\_dir("recovery", "/system");
8. package\_extract\_dir("system", "/system");
9. symlink("busybox", "/system/bin/cp", "/system/bin/grep",
10. "/system/bin/tar", "/system/bin/unzip",
11. "/system/bin/vi");
12. symlink("toolbox", "/system/bin/cat", "/system/bin/chmod",
13. "/system/bin/chown", "/system/bin/cmp", "/system/bin/date",
14. "/system/bin/dd", "/system/bin/df", "/system/bin/dmesg",
15. "/system/bin/getevent", "/system/bin/getprop", "/system/bin/hd",
16. "/system/bin/id", "/system/bin/ifconfig", "/system/bin/iftop",
17. "/system/bin/insmod", "/system/bin/ioctl", "/system/bin/ionice",
18. "/system/bin/kill", "/system/bin/ln", "/system/bin/log",
19. "/system/bin/ls", "/system/bin/lsmod", "/system/bin/lsof",
20. "/system/bin/mkdir", "/system/bin/mount", "/system/bin/mv",
21. "/system/bin/nandread", "/system/bin/netstat",
22. "/system/bin/newfs\_msdos", "/system/bin/notify", "/system/bin/printenv",
23. "/system/bin/ps", "/system/bin/reboot", "/system/bin/renice",
24. "/system/bin/rm", "/system/bin/rmdir", "/system/bin/rmmod",
25. "/system/bin/route", "/system/bin/schedtop", "/system/bin/sendevent",
26. "/system/bin/setconsole", "/system/bin/setprop", "/system/bin/sleep",
27. "/system/bin/smd", "/system/bin/start", "/system/bin/stop",
28. "/system/bin/sync", "/system/bin/top", "/system/bin/umount",
29. "/system/bin/uptime", "/system/bin/vmstat", "/system/bin/watchprops",
30. "/system/bin/wipe");
31. set\_perm\_recursive(0, 0, 0755, 0644, "/system");
32. set\_perm\_recursive(0, 2000, 0755, 0755, "/system/bin");
33. set\_perm(0, 3003, 02750, "/system/bin/netcfg");
34. set\_perm(0, 3004, 02755, "/system/bin/ping");
35. set\_perm(0, 2000, 06750, "/system/bin/run-as");
36. set\_perm\_recursive(1002, 1002, 0755, 0440, "/system/etc/bluetooth");
37. set\_perm(0, 0, 0755, "/system/etc/bluetooth");
38. set\_perm(1000, 1000, 0640, "/system/etc/bluetooth/auto\_pairing.conf");
39. set\_perm(3002, 3002, 0444, "/system/etc/bluetooth/blacklist.conf");
40. set\_perm(1002, 1002, 0440, "/system/etc/dbus.conf");
41. set\_perm(1014, 2000, 0550, "/system/etc/dhcpcd/dhcpcd-run-hooks");
42. set\_perm(0, 2000, 0550, "/system/etc/init.goldfish.sh");
43. set\_perm(0, 0, 0544, "/system/etc/install-recovery.sh");
44. set\_perm\_recursive(0, 0, 0755, 0555, "/system/etc/ppp");
45. set\_perm\_recursive(0, 2000, 0755, 0755, "/system/xbin");
46. set\_perm(0, 0, 06755, "/system/xbin/librank");
47. set\_perm(0, 0, 06755, "/system/xbin/procmem");
48. set\_perm(0, 0, 06755, "/system/xbin/procrank");
49. set\_perm(0, 0, 06755, "/system/xbin/su");
50. set\_perm(0, 0, 06755, "/system/xbin/tcpdump");
51. show\_progress(0.200000, 0);
52. show\_progress(0.200000, 10);
53. assert(package\_extract\_file("boot.img", "/tmp/boot.img"),
54. write\_raw\_image("/tmp/boot.img", "boot"),
55. delete("/tmp/boot.img"));
56. show\_progress(0.100000, 0);
57. unmount("/system");

下面分析下这个脚本的执行过程：

①比较时间戳：如果升级包较旧则终止脚本的执行。

②匹配设备信息：如果和当前的设备信息不一致，则停止脚本的执行。

③显示进度条：如果以上两步匹配则开始显示升级进度条。

④格式化system分区并挂载。

⑤提取包中的recovery以及system目录下的内容到系统的/system下。

⑥为/system/bin/下的命令文件建立符号连接。

⑦设置/system/下目录以及文件的属性。

⑧将包中的boot.img提取到/tmp/boot.img。

⑨将/tmp/boot.img镜像文件写入到boot分区。

⑩完成后卸载/system。

以上就是updater-script脚本中的语法，及其执行的具体过程。通过分析其执行流程，我们可以发现在执行过程中，并未将升级包另外解压到一个地方，而是需要什么提取什么。值得主要的是在提取recovery和system目录中的内容时，一并放在了/system/下。在操作的过程中，并未删除或改变update.zip包中的任何内容。在实际的更新完成后，我们的update.zip包确实还存在于原来的位置。

## factory reset恢复出厂

按照前面所列的8条步骤，其中1-6及7-8都与 main 通用流程一样，不再复述

### 执行命令

echo --wipe\_all > /cache/recovery/command

reboot recovery

**else if** (should\_wipe\_data) {  
 **if** (!wipe\_data(device)) {  
 status = INSTALL\_ERROR;  
 }  
}

### wipe\_data

**static bool** wipe\_data(Device\* device) {  
 modified\_flash = **true**;  
  
 ui->Print(**"\n-- Wiping data...\n"**);  
 **bool** success =  
 device->PreWipeData() &&  
 erase\_volume(**"/data"**) &&  
 (has\_cache ? erase\_volume(**"/cache"**) : **true**) &&  
 device->PostWipeData();  
 ui->Print(**"Data wipe %s.\n"**, success ? **"complete"** : **"failed"**);  
 **return** success;  
}

此处擦除的是两个分区

 \* 5. erase\_volume() reformats /data  
 \* 6. erase\_volume() reformats /cache

最后就是

clear\_sdcard\_update\_bootloader\_message();

看看 erase\_volume() 函数先：

**[cpp]** [view plain](https://blog.csdn.net/andyhuabing/article/details/9248713) [copy](https://blog.csdn.net/andyhuabing/article/details/9248713)

1. **static** **int**
2. erase\_volume(**const** **char** \*volume) {
3. ui\_set\_background(BACKGROUND\_ICON\_INSTALLING);
4. ui\_show\_indeterminate\_progress();
5. ui\_print("Formatting %s...\n", volume);
7. ensure\_path\_unmounted(volume);
9. **if** (strcmp(volume, "/cache") == 0) {
10. // Any part of the log we'd copied to cache is now gone.
11. // Reset the pointer so we copy from the beginning of the temp
12. // log.
13. tmplog\_offset = 0;
14. }
16. **return** format\_volume(volume);
17. }

上面红字标明的是重要函数调用

int ensure\_path\_unmounted(const char\* path) {

Volume\* v = volume\_for\_path(path);

 result = scan\_mounted\_volumes();

return unmount\_mounted\_volume(mv);

}

就是将指定的path中径mount point进行卸载掉，而 format\_volume的主要功能就是：

MtdWriteContext \*write = mtd\_write\_partition(partition);

mtd\_erase\_blocks(write, -1);

mtd\_write\_close(write);

不要细说了吧，就是将整个分区数据全清掉。

最后一个函数：

void  
clear\_sdcard\_update\_bootloader\_message() {  
    struct bootloader\_message boot;  
    memset(&boot, 0, sizeof(boot));  
    set\_bootloader\_message(&boot);  
}

就是将misc分区数据重置清0

这样子就完成的恢复出厂设置的情况了。将 data/cache分区erase擦掉就好了。

通常我们会自己划分一个分区meta分区

# update.zip包的制作-编译原理

## 分区

Format\_volume

Android手机和平板的内部存储分区列表如下：

/boot

/system

/recovery

/data

/cache

/misc

另外还有 SD card分区：

/sdcard

/sd-ext

请注意只有/sdcard是所有Android设备都有的，/sd-ext只在某些设备上可以看到。现在让我们分别分析这些分区的内容和功能。

### boot

这个分区上有Android的引导程序，包括内核和内存操作程序。没有这个分区设备就不能被引导。恢复系统的时候会擦除这个分区，并且必须重新安装引导程序和ROM才能重启系统

### system

这个分区上是除了内核和内存操作之外的整个操作系统。里面包含了Android用户接口和预先安装的系统应用。擦除了这个分区就会删除掉Andorid系统，所以你需要进入recovery模式或者bootloader模式去安装一个新的ROM。

### recovery

recovery分区被认为是另一个启动分区，你可以启动设备进入recovery控制台去执行高级的系统恢复和管理操作。

### data

这个分区保存着用户数据。通讯录、短信、设置和你安装的apps都在这个分区上。擦除这个分区相当于恢复出厂设置，当你第一次启动设备的时候或者在安装了官方或者客户的ROM之后系统会自动重建这个分区。当你执行恢复出厂设置时，就是在擦除这个分区。

### cache

这个分区是Android系统存储频繁访问的数据和app的地方。擦除这个分区不影响你的个人数据，当你继续使用设备时，被擦除的数据就会自动被创建。

### misc

这个分区包含各种复杂的类似于on/off的系统设置。这些设置可能是USB配置和某些硬件配置信息。这是一个重要的分区，如果该分区损坏或者丢失，设备的功能可能就工作不正常。

### sdcard

这个分区不是内部的存储区，而是SD card。这是你个人存储大文件的地方，存储多媒体文件、文档、ROM文件等等。如果你备份了这个分区上文件，那么擦除这个分区将会很安全。不过请注意一些用户安装的apps会存储数据和设置参数到SD card上，擦除这个分区会掉的这些数据。

### sd-ext

这个分区不是Android设备的标准分区，但是在有客户ROM的情况下变得很流行。

以上我们总结了Android各个分区的功能。当你安装ROM时或者系统询问你要删除某个分区时，你就会准确地知道哪些会丢失哪些不会丢失，因此你就会去备份那些将会丢失文件。

## update.zip包目录结构

|----boot.img

|----system/

|----recovery/

`|----recovery-from-boot.p

`|----etc/

`|----install-recovery.sh

|---META-INF/

`|CERT.RSA

`|CERT.SF

`|MANIFEST.MF

`|----com/

`|----google/

`|----android/

`|----update-binary

`|----updater-script

|----android/

`|----metadata

## 目录详解

以上是我们用命令make otapackage 制作的update.zip包的标准目录结构。

### boot.img

是更新boot分区所需要的文件。这个boot.img主要包括kernel+ramdisk。

### system

system/目录的内容在升级后会放在系统的system分区。主要用来更新系统的一些应用或则应用会用到的一些库等等。可以将Android源码编译out/target/product/tcc8800/system/中的所有文件拷贝到这个目录来代替。此处我们可以得到一个获取待安装包的信息的方法，比如获取属性

|  |
| --- |
| Properties prop = **new** Properties(); ZipFile zipFile = **new** ZipFile(itFile); ZipEntry zipEntry = zipFile.getEntry(**"system/build.prop"**); InputStream input = zipFile.getInputStream(zipEntry) ; *// 得到一个压缩实体的输入流* prop.load(input); bean.**mVersionCode** = prop.getProperty(**"persist.system.version"**); *formatMd*(prop.getProperty(**"ro.build.date.utc"**)); input.close() ; *// 关闭输入流* |

### recovery

   recovery/目录中的recovery-from-boot.p是boot.img和recovery.img的补丁(patch),主要用来更新recovery分区，其中etc/目录下的install-recovery.sh是更新脚本。

### update-binary

    update-binary是一个二进制文件，相当于一个脚本解释器，能够识别updater-script中描述的操作。该文件在Android源码编译后out/target/product/tcc8800/system/bin/updater生成，可将updater重命名为update-binary得到。  
    该文件在具体的更新包中的名字由源码中bootable/recovery/install.c中的宏ASSUMED\_UPDATE\_BINARY\_NAME的值而定。

### updater-script

此文件是一个脚本文件，具体描述了更新过程。我们可以根据具体情况编写该脚本来适应我们的具体需求。该文件的命名由源码中bootable/recovery/updater/updater.c文件中的宏SCRIPT\_NAME的值而定。

这个文件是从哪里差产生的呢

### metadata

文件是描述设备信息及环境变量的元数据。主要包括一些编译选项，签名公钥，时间戳以及设备型号等。

我们还可以在包中添加userdata目录，来更新系统中的用户数据部分。这部分内容在更新后会存放在系统的/data目录下。

### update.zip包的签名

update.zip更新包在制作完成后需要对其签名，否则在升级时会出现认证失败的错误提示。而且签名要使用和目标板一致的加密公钥。加密公钥及加密需要的三个文件在Android源码编译后生成的具体路径为：

               out/host/linux-x86/framework/signapk.jar

               build/target/product/security/testkey.x509.pem

               build/target/product/security/testkey.pk8 。

     我们用命令make otapackage制作生成的update.zip包是已签过名的，如果自己做update.zip包时必须手动对其签名。

     具体的加密方法：$ java –jar gingerbread/out/host/linux/framework/signapk.jar –w gingerbread/build/target/product/security/testkey.x509.pem                                      gingerbread/build/target/product/security/testkey.pk8 update.zip update\_signed.zip  
     以上命令在update.zip包所在的路径下执行，其中signapk.jar testkey.x509.pem以及testkey.pk8文件的引用使用绝对路径。update.zip 是我们已经打好的包，update\_signed.zip包是命令执行完生成的已经签过名的包。

### MANIFEST.MF

这个manifest文件定义了与包的组成结构相关的数据。类似Android应用的mainfest.xml文件。

### CERT.RSA：

与签名文件相关联的签名程序块文件，它存储了用于签名JAR文件的公共签名。

### CERT.SF：

这是JAR文件的签名文件，其中前缀CERT代表签名者。  
        另外，在具体升级时，对update.zip包检查时大致会分三步：①检验SF文件与RSA文件是否匹配。②检验MANIFEST.MF与签名文件中的digest是否一致。③检验包中的文件与MANIFEST中所描述的是否一致。

## 生成过程分析

  对于update.zip包的制作有两种方式，即手动制作和命令生成。

### 手动制作

      第一种手动制作：即按照update.zip的目录结构手动创建我们需要的目录。然后将对应的文件拷贝到相应的目录下，比如我们向系统中新加一个应用程序。可以将新增的应用拷贝到我们新建的update/system/app/下（system目录是事先拷贝编译源码后生成的system目录），打包并签名后，拷贝到SD卡就可以使用了。这种方式在实际的tcc8800开发板中未测试成功。签名部分未通过，可能与具体的开发板相关。

### 自动制作

      第二种制作方式：命令制作。Android源码系统中为我们提供了制作update.zip刷机包的命令，即make otapackage。该命令在编译源码完成后并在源码根目录下执行。 具体操作方式：在源码根目录下执行

                ①$ . build/envsetup.sh。

                ②$ lunch 然后选择你需要的配置(如17)。

                ③$ make otapackage。

          在编译完源码后最好再执行一遍上面的①、②步防止执行③时出现未找到对应规则的错误提示。命令执行完成后生成的升级包所在位置在out/target/product/full\_tcc8800\_evm\_target\_files-eng.mumu.20120309.111059.zip将这个包重新命名为update.zip，并拷贝到SD卡中即可使用。 这种方式(即完全升级)在tcc8800开发板中已测试成功。

#### 过程分析

       使用make otapackage命令生成update.zip的过程分析。  
       在源码根目录下执行make otapackage命令生成update.zip包主要分为两步，第一步是根据Makefile执行编译生成一个update原包（zip格式）。第二步是运行一个python脚本，并以上一步准备的zip包作为输入，最终生成我们需要的升级包。下面进一步分析这两个过程。

##### 生成原始包

        第一步：编译Makefile。对应的Makefile文件所在位置：build/core/Makefile。从该文件的884行（tcc8800,gingerbread0919）开始会生成一个zip包，这个包最后会用来制作OTA package 或者filesystem image。先将这部分的对应的Makefile贴出来如下：

[plain] [view plaincopy](http://blog.csdn.net/mu0206mu/article/details/7399822)

1. # -----------------------------------------------------------------
2. # A zip of the directories that map to the target filesystem.
3. # This zip can be used to create an OTA package or filesystem image
4. # as a post-build step.
5. #
6. name := $(TARGET\_PRODUCT)
7. ifeq ($(TARGET\_BUILD\_TYPE),debug)
8. name := $(name)\_debug
9. endif
10. name := $(name)-target\_files-$(FILE\_NAME\_TAG)
12. intermediates := $(call intermediates-dir-for,PACKAGING,target\_files)
13. BUILT\_TARGET\_FILES\_PACKAGE := $(intermediates)/$(name).zip
14. $(BUILT\_TARGET\_FILES\_PACKAGE): intermediates := $(intermediates)
15. $(BUILT\_TARGET\_FILES\_PACKAGE): \
16. zip\_root := $(intermediates)/$(name)
18. # $(1): Directory to copy
19. # $(2): Location to copy it to
20. # The "ls -A" is to prevent "acp s/\* d" from failing if s is empty.
21. define package\_files-copy-root
22. if [ -d "$(strip $(1))" -a "$$(ls -A $(1))" ]; then \
23. mkdir -p $(2) && \
24. $(ACP) -rd $(strip $(1))/\* $(2); \
25. fi
26. endef
28. built\_ota\_tools := \
29. $(call intermediates-dir-for,EXECUTABLES,applypatch)/applypatch \
30. $(call intermediates-dir-for,EXECUTABLES,applypatch\_static)/applypatch\_static \
31. $(call intermediates-dir-for,EXECUTABLES,check\_prereq)/check\_prereq \
32. $(call intermediates-dir-for,EXECUTABLES,updater)/updater
33. $(BUILT\_TARGET\_FILES\_PACKAGE): PRIVATE\_OTA\_TOOLS := $(built\_ota\_tools)
35. $(BUILT\_TARGET\_FILES\_PACKAGE): PRIVATE\_RECOVERY\_API\_VERSION := $(RECOVERY\_API\_VERSION)
37. ifeq ($(TARGET\_RELEASETOOLS\_EXTENSIONS),)
38. # default to common dir for device vendor
39. $(BUILT\_TARGET\_FILES\_PACKAGE): tool\_extensions := $(TARGET\_DEVICE\_DIR)/../common
40. else
41. $(BUILT\_TARGET\_FILES\_PACKAGE): tool\_extensions := $(TARGET\_RELEASETOOLS\_EXTENSIONS)
42. endif
44. # Depending on the various images guarantees that the underlying
45. # directories are up-to-date.
46. $(BUILT\_TARGET\_FILES\_PACKAGE): \
47. $(INSTALLED\_BOOTIMAGE\_TARGET) \
48. $(INSTALLED\_RADIOIMAGE\_TARGET) \
49. $(INSTALLED\_RECOVERYIMAGE\_TARGET) \
50. $(INSTALLED\_SYSTEMIMAGE) \
51. $(INSTALLED\_USERDATAIMAGE\_TARGET) \
52. $(INSTALLED\_ANDROID\_INFO\_TXT\_TARGET) \
53. $(built\_ota\_tools) \
54. $(APKCERTS\_FILE) \
55. $(HOST\_OUT\_EXECUTABLES)/fs\_config \
56. | $(ACP)
57. @echo "Package target files: $@"
58. $(hide) rm -rf $@ $(zip\_root)
59. $(hide) mkdir -p $(dir $@) $(zip\_root)
60. @# Components of the recovery image
61. $(hide) mkdir -p $(zip\_root)/RECOVERY
62. $(hide) $(call package\_files-copy-root, \
63. $(TARGET\_RECOVERY\_ROOT\_OUT),$(zip\_root)/RECOVERY/RAMDISK)
64. ifdef INSTALLED\_KERNEL\_TARGET
65. $(hide) $(ACP) $(INSTALLED\_KERNEL\_TARGET) $(zip\_root)/RECOVERY/kernel
66. endif
67. ifdef INSTALLED\_2NDBOOTLOADER\_TARGET
68. $(hide) $(ACP) \
69. $(INSTALLED\_2NDBOOTLOADER\_TARGET) $(zip\_root)/RECOVERY/second
70. endif
71. ifdef BOARD\_KERNEL\_CMDLINE
72. $(hide) echo "$(BOARD\_KERNEL\_CMDLINE)" > $(zip\_root)/RECOVERY/cmdline
73. endif
74. ifdef BOARD\_KERNEL\_BASE
75. $(hide) echo "$(BOARD\_KERNEL\_BASE)" > $(zip\_root)/RECOVERY/base
76. endif
77. ifdef BOARD\_KERNEL\_PAGESIZE
78. $(hide) echo "$(BOARD\_KERNEL\_PAGESIZE)" > $(zip\_root)/RECOVERY/pagesize
79. endif
80. @# Components of the boot image
81. $(hide) mkdir -p $(zip\_root)/BOOT
82. $(hide) $(call package\_files-copy-root, \
83. $(TARGET\_ROOT\_OUT),$(zip\_root)/BOOT/RAMDISK)
84. ifdef INSTALLED\_KERNEL\_TARGET
85. $(hide) $(ACP) $(INSTALLED\_KERNEL\_TARGET) $(zip\_root)/BOOT/kernel
86. endif
87. ifdef INSTALLED\_2NDBOOTLOADER\_TARGET
88. $(hide) $(ACP) \
89. $(INSTALLED\_2NDBOOTLOADER\_TARGET) $(zip\_root)/BOOT/second
90. endif
91. ifdef BOARD\_KERNEL\_CMDLINE
92. $(hide) echo "$(BOARD\_KERNEL\_CMDLINE)" > $(zip\_root)/BOOT/cmdline
93. endif
94. ifdef BOARD\_KERNEL\_BASE
95. $(hide) echo "$(BOARD\_KERNEL\_BASE)" > $(zip\_root)/BOOT/base
96. endif
97. ifdef BOARD\_KERNEL\_PAGESIZE
98. $(hide) echo "$(BOARD\_KERNEL\_PAGESIZE)" > $(zip\_root)/BOOT/pagesize
99. endif
100. $(hide) $(foreach t,$(INSTALLED\_RADIOIMAGE\_TARGET),\
101. mkdir -p $(zip\_root)/RADIO; \
102. $(ACP) $(t) $(zip\_root)/RADIO/$(notdir $(t));)
103. @# Contents of the system image
104. $(hide) $(call package\_files-copy-root, \
105. $(SYSTEMIMAGE\_SOURCE\_DIR),$(zip\_root)/SYSTEM)
106. @# Contents of the data image
107. $(hide) $(call package\_files-copy-root, \
108. $(TARGET\_OUT\_DATA),$(zip\_root)/DATA)
109. @# Extra contents of the OTA package
110. $(hide) mkdir -p $(zip\_root)/OTA/bin
111. $(hide) $(ACP) $(INSTALLED\_ANDROID\_INFO\_TXT\_TARGET) $(zip\_root)/OTA/
112. $(hide) $(ACP) $(PRIVATE\_OTA\_TOOLS) $(zip\_root)/OTA/bin/
113. @# Files that do not end up in any images, but are necessary to
114. @# build them.
115. $(hide) mkdir -p $(zip\_root)/META
116. $(hide) $(ACP) $(APKCERTS\_FILE) $(zip\_root)/META/apkcerts.txt
117. $(hide) echo "$(PRODUCT\_OTA\_PUBLIC\_KEYS)" > $(zip\_root)/META/otakeys.txt
118. $(hide) echo "recovery\_api\_version=$(PRIVATE\_RECOVERY\_API\_VERSION)" > $(zip\_root)/META/misc\_info.txt
119. ifdef BOARD\_FLASH\_BLOCK\_SIZE
120. $(hide) echo "blocksize=$(BOARD\_FLASH\_BLOCK\_SIZE)" >> $(zip\_root)/META/misc\_info.txt
121. endif
122. ifdef BOARD\_BOOTIMAGE\_PARTITION\_SIZE
123. $(hide) echo "boot\_size=$(BOARD\_BOOTIMAGE\_PARTITION\_SIZE)" >> $(zip\_root)/META/misc\_info.txt
124. endif
125. ifdef BOARD\_RECOVERYIMAGE\_PARTITION\_SIZE
126. $(hide) echo "recovery\_size=$(BOARD\_RECOVERYIMAGE\_PARTITION\_SIZE)" >> $(zip\_root)/META/misc\_info.txt
127. endif
128. ifdef BOARD\_SYSTEMIMAGE\_PARTITION\_SIZE
129. $(hide) echo "system\_size=$(BOARD\_SYSTEMIMAGE\_PARTITION\_SIZE)" >> $(zip\_root)/META/misc\_info.txt
130. endif
131. ifdef BOARD\_USERDATAIMAGE\_PARTITION\_SIZE
132. $(hide) echo "userdata\_size=$(BOARD\_USERDATAIMAGE\_PARTITION\_SIZE)" >> $(zip\_root)/META/misc\_info.txt
133. endif
134. $(hide) echo "tool\_extensions=$(tool\_extensions)" >> $(zip\_root)/META/misc\_info.txt
135. ifdef mkyaffs2\_extra\_flags
136. $(hide) echo "mkyaffs2\_extra\_flags=$(mkyaffs2\_extra\_flags)" >> $(zip\_root)/META/misc\_info.txt
137. endif
138. @# Zip everything up, preserving symlinks
139. $(hide) (cd $(zip\_root) && zip -qry ../$(notdir $@) .)
140. @# Run fs\_config on all the system files in the zip, and save the output
141. $(hide) zipinfo -1 $@ | awk -F/ 'BEGIN { OFS="/" } /^SYSTEM\// {$$1 = "system"; print}' | $(HOST\_OUT\_EXECUTABLES)/fs\_config > $(zip\_root)/META/filesystem\_config.txt
142. $(hide) (cd $(zip\_root) && zip -q ../$(notdir $@) META/filesystem\_config.txt)

145. target-files-package: $(BUILT\_TARGET\_FILES\_PACKAGE)

148. ifneq ($(TARGET\_SIMULATOR),true)
149. ifneq ($(TARGET\_PRODUCT),sdk)
150. ifneq ($(TARGET\_DEVICE),generic)
151. ifneq ($(TARGET\_NO\_KERNEL),true)
152. ifneq ($(recovery\_fstab),)

            根据上面的Makefile可以分析这个包的生成过程：

            首先创建一个root\_zip根目录，并依次在此目录下创建所需要的如下其他目录

            ①创建RECOVERY目录，并填充该目录的内容，包括kernel的镜像和recovery根文件系统的镜像。此目录最终用于生成recovery.img。

            ②创建并填充BOOT目录。包含kernel和cmdline以及pagesize大小等，该目录最终用来生成boot.img。  
            ③向SYSTEM目录填充system image。  
            ④向DATA填充data image。  
            ⑤用于生成OTA package包所需要的额外的内容。主要包括一些bin命令。  
            ⑥创建META目录并向该目录下添加一些文本文件，如apkcerts.txt(描述apk文件用到的认证证书)，misc\_info.txt(描述Flash内存的块大小以及boot、recovery、system、userdata等分区的大小信息)。  
            ⑦使用保留连接选项压缩我们在上面获得的root\_zip目录。  
            ⑧使用fs\_config(build/tools/fs\_config)配置上面的zip包内所有的系统文件(system/下各目录、文件)的权限属主等信息。fs\_config包含了一个头文件#include“private/android\_filesystem\_config.h”。在这个头文件中以硬编码的方式设定了system目录下各文件的权限、属主。执行完配置后会将配置后的信息以文本方式输出 到META/filesystem\_config.txt中。并再一次zip压缩成我们最终需要的原始包。

##### 生成升级包

     第二步：上面的zip包只是一个编译过程中生成的原始包。这个原始zip包在实际的编译过程中有两个作用，一是用来生成OTA update升级包，二是用来生成系统镜像。在编译过程中若生成OTA update升级包时会调用（具体位置在Makefile的1037行到1058行）一个名为ota\_from\_target\_files的python脚本，位置在/build/tools/releasetools/ota\_from\_target\_files。这个脚本的作用是以第一步生成的zip原始包作为输入，最终生成可用的OTA升级zip包。

     下面我们分析使用这个脚本生成最终OTA升级包的过程。

首先看一下这个脚本开始部分的帮助文档。代码如下：

  下面简单翻译一下他们的使用方法以及选项的作用。

                        Usage: ota\_from\_target\_files [flags] input\_target\_files output\_ota\_package  
                        -b 过时的。  
                        -k签名所使用的密钥  
                        -i生成增量OTA包时使用此选项。后面我们会用到这个选项来生成OTA增量包。  
                        -w是否清除userdata分区  
                        -n在升级时是否不检查时间戳，缺省要检查，即缺省情况下只能基于旧版本升级。  
                        -e是否有额外运行的脚本  
                        -m执行过程中生成脚本(updater-script)所需要的格式,目前有两种即amend和edify。对应上两种版本升级时会采用不同的解释器。缺省会同时生成两种格式的脚 本。  
                        -p定义脚本用到的一些可执行文件的路径。  
                        -s定义额外运行脚本的路径。  
                        -x定义额外运行的脚本可能用的键值对。  
                        -v执行过程中打印出执行的命令。  
                        -h命令帮助

下面我们分析ota\_from\_target\_files这个python脚本是怎样生成最终zip包的。先讲这个脚本的代码贴出来如下：

[python] [view plaincopy](http://blog.csdn.net/mu0206mu/article/details/7399822)

1. **import** sys
3. **if** sys.hexversion < 0x02040000:
4. **print** >> sys.stderr, "Python 2.4 or newer is required."
5. sys.exit(1)
7. **import** copy
8. **import** errno
9. **import** os
10. **import** re
11. **import** sha
12. **import** subprocess
13. **import** tempfile
14. **import** time
15. **import** zipfile
17. **import** common
18. **import** edify\_generator
20. OPTIONS = common.OPTIONS
21. OPTIONS.package\_key = "build/target/product/security/testkey"
22. OPTIONS.incremental\_source = None
23. OPTIONS.require\_verbatim = set()
24. OPTIONS.prohibit\_verbatim = set(("system/build.prop",))
25. OPTIONS.patch\_threshold = 0.95
26. OPTIONS.wipe\_user\_data = False
27. OPTIONS.omit\_prereq = False
28. OPTIONS.extra\_script = None
29. OPTIONS.worker\_threads = 3
31. **def** MostPopularKey(d, default):
32. """Given a dict, return the key corresponding to the largest
33. value.  Returns 'default' if the dict is empty."""
34. x = [(v, k) **for** (k, v) **in** d.iteritems()]
35. **if** **not** x: **return** default
36. x.sort()
37. **return** x[-1][1]

40. **def** IsSymlink(info):
41. """Return true if the zipfile.ZipInfo object passed in represents a
42. symlink."""
43. **return** (info.external\_attr >> 16) == 0120777

46. **class** Item:
47. """Items represent the metadata (user, group, mode) of files and
48. directories in the system image."""
49. ITEMS = {}
50. **def** \_\_init\_\_(self, name, dir=False):
51. self.name = name
52. self.uid = None
53. self.gid = None
54. self.mode = None
55. self.dir = dir
57. **if** name:
58. self.parent = Item.Get(os.path.dirname(name), dir=True)
59. self.parent.children.append(self)
60. **else**:
61. self.parent = None
62. **if** dir:
63. self.children = []
65. **def** Dump(self, indent=0):
66. **if** self.uid **is** **not** None:
67. **print** "%s%s %d %d %o" % ("  "\*indent, self.name, self.uid, self.gid, self.mode)
68. **else**:
69. **print** "%s%s %s %s %s" % ("  "\*indent, self.name, self.uid, self.gid, self.mode)
70. **if** self.dir:
71. **print** "%s%s" % ("  "\*indent, self.descendants)
72. **print** "%s%s" % ("  "\*indent, self.best\_subtree)
73. **for** i **in** self.children:
74. i.Dump(indent=indent+1)
76. @classmethod
77. **def** Get(cls, name, dir=False):
78. **if** name **not** **in** cls.ITEMS:
79. cls.ITEMS[name] = Item(name, dir=dir)
80. **return** cls.ITEMS[name]
82. @classmethod
83. **def** GetMetadata(cls, input\_zip):
85. **try**:
86. # See if the target\_files contains a record of what the uid,
87. # gid, and mode is supposed to be.
88. output = input\_zip.read("META/filesystem\_config.txt")
89. **except** KeyError:
90. # Run the external 'fs\_config' program to determine the desired
91. # uid, gid, and mode for every Item object.  Note this uses the
92. # one in the client now, which might not be the same as the one
93. # used when this target\_files was built.
94. p = common.Run(["fs\_config"], stdin=subprocess.PIPE,
95. stdout=subprocess.PIPE, stderr=subprocess.PIPE)
96. suffix = { False: "", True: "/" }
97. input = "".join(["%s%s\n" % (i.name, suffix[i.dir])
98. **for** i **in** cls.ITEMS.itervalues() **if** i.name])
99. output, error = p.communicate(input)
100. **assert** **not** error
102. **for** line **in** output.split("\n"):
103. **if** **not** line: **continue**
104. name, uid, gid, mode = line.split()
105. i = cls.ITEMS.get(name, None)
106. **if** i **is** **not** None:
107. i.uid = int(uid)
108. i.gid = int(gid)
109. i.mode = int(mode, 8)
110. **if** i.dir:
111. i.children.sort(key=**lambda** i: i.name)
113. # set metadata for the files generated by this script.
114. i = cls.ITEMS.get("system/recovery-from-boot.p", None)
115. **if** i: i.uid, i.gid, i.mode = 0, 0, 0644
116. i = cls.ITEMS.get("system/etc/install-recovery.sh", None)
117. **if** i: i.uid, i.gid, i.mode = 0, 0, 0544
119. **def** CountChildMetadata(self):
120. """Count up the (uid, gid, mode) tuples for all children and
121. determine the best strategy for using set\_perm\_recursive and
122. set\_perm to correctly chown/chmod all the files to their desired
123. values.  Recursively calls itself for all descendants.
125. Returns a dict of {(uid, gid, dmode, fmode): count} counting up
126. all descendants of this node.  (dmode or fmode may be None.)  Also
127. sets the best\_subtree of each directory Item to the (uid, gid,
128. dmode, fmode) tuple that will match the most descendants of that
129. Item.
130. """
132. **assert** self.dir
133. d = self.descendants = {(self.uid, self.gid, self.mode, None): 1}
134. **for** i **in** self.children:
135. **if** i.dir:
136. **for** k, v **in** i.CountChildMetadata().iteritems():
137. d[k] = d.get(k, 0) + v
138. **else**:
139. k = (i.uid, i.gid, None, i.mode)
140. d[k] = d.get(k, 0) + 1
142. # Find the (uid, gid, dmode, fmode) tuple that matches the most
143. # descendants.
145. # First, find the (uid, gid) pair that matches the most
146. # descendants.
147. ug = {}
148. **for** (uid, gid, \_, \_), count **in** d.iteritems():
149. ug[(uid, gid)] = ug.get((uid, gid), 0) + count
150. ug = MostPopularKey(ug, (0, 0))
152. # Now find the dmode and fmode that match the most descendants
153. # with that (uid, gid), and choose those.
154. best\_dmode = (0, 0755)
155. best\_fmode = (0, 0644)
156. **for** k, count **in** d.iteritems():
157. **if** k[:2] != ug: **continue**
158. **if** k[2] **is** **not** None **and** count >= best\_dmode[0]: best\_dmode = (count, k[2])
159. **if** k[3] **is** **not** None **and** count >= best\_fmode[0]: best\_fmode = (count, k[3])
160. self.best\_subtree = ug + (best\_dmode[1], best\_fmode[1])
162. **return** d
164. **def** SetPermissions(self, script):
165. """Append set\_perm/set\_perm\_recursive commands to 'script' to
166. set all permissions, users, and groups for the tree of files
167. rooted at 'self'."""
169. self.CountChildMetadata()
171. **def** recurse(item, current):
172. # current is the (uid, gid, dmode, fmode) tuple that the current
173. # item (and all its children) have already been set to.  We only
174. # need to issue set\_perm/set\_perm\_recursive commands if we're
175. # supposed to be something different.
176. **if** item.dir:
177. **if** current != item.best\_subtree:
178. script.SetPermissionsRecursive("/"+item.name, \*item.best\_subtree)
179. current = item.best\_subtree
181. **if** item.uid != current[0] **or** item.gid != current[1] **or** \
182. item.mode != current[2]:
183. script.SetPermissions("/"+item.name, item.uid, item.gid, item.mode)
185. **for** i **in** item.children:
186. recurse(i, current)
187. **else**:
188. **if** item.uid != current[0] **or** item.gid != current[1] **or** \
189. item.mode != current[3]:
190. script.SetPermissions("/"+item.name, item.uid, item.gid, item.mode)
192. recurse(self, (-1, -1, -1, -1))

195. **def** CopySystemFiles(input\_zip, output\_zip=None,
196. substitute=None):
197. """Copies files underneath system/ in the input zip to the output
198. zip.  Populates the Item class with their metadata, and returns a
199. list of symlinks.  output\_zip may be None, in which case the copy is
200. skipped (but the other side effects still happen).  substitute is an
201. optional dict of {output filename: contents} to be output instead of
202. certain input files.
203. """
205. symlinks = []
207. **for** info **in** input\_zip.infolist():
208. **if** info.filename.startswith("SYSTEM/"):
209. basefilename = info.filename[7:]
210. **if** IsSymlink(info):
211. symlinks.append((input\_zip.read(info.filename),
212. "/system/" + basefilename))
213. **else**:
214. info2 = copy.copy(info)
215. fn = info2.filename = "system/" + basefilename
216. **if** substitute **and** fn **in** substitute **and** substitute[fn] **is** None:
217. **continue**
218. **if** output\_zip **is** **not** None:
219. **if** substitute **and** fn **in** substitute:
220. data = substitute[fn]
221. **else**:
222. data = input\_zip.read(info.filename)
223. output\_zip.writestr(info2, data)
224. **if** fn.endswith("/"):
225. Item.Get(fn[:-1], dir=True)
226. **else**:
227. Item.Get(fn, dir=False)
229. symlinks.sort()
230. **return** symlinks

233. **def** SignOutput(temp\_zip\_name, output\_zip\_name):
234. key\_passwords = common.GetKeyPasswords([OPTIONS.package\_key])
235. pw = key\_passwords[OPTIONS.package\_key]
237. common.SignFile(temp\_zip\_name, output\_zip\_name, OPTIONS.package\_key, pw,
238. whole\_file=True)

241. **def** AppendAssertions(script, input\_zip):
242. device = GetBuildProp("ro.product.device", input\_zip)
243. script.AssertDevice(device)

246. **def** MakeRecoveryPatch(output\_zip, recovery\_img, boot\_img):
247. """Generate a binary patch that creates the recovery image starting
248. with the boot image.  (Most of the space in these images is just the
249. kernel, which is identical for the two, so the resulting patch
250. should be efficient.)  Add it to the output zip, along with a shell
251. script that is run from init.rc on first boot to actually do the
252. patching and install the new recovery image.
254. recovery\_img and boot\_img should be File objects for the
255. corresponding images.
257. Returns an Item for the shell script, which must be made
258. executable.
259. """
261. d = common.Difference(recovery\_img, boot\_img)
262. \_, \_, patch = d.ComputePatch()
263. common.ZipWriteStr(output\_zip, "recovery/recovery-from-boot.p", patch)
264. Item.Get("system/recovery-from-boot.p", dir=False)
266. boot\_type, boot\_device = common.GetTypeAndDevice("/boot", OPTIONS.info\_dict)
267. recovery\_type, recovery\_device = common.GetTypeAndDevice("/recovery", OPTIONS.info\_dict)
269. # Images with different content will have a different first page, so
270. # we check to see if this recovery has already been installed by
271. # testing just the first 2k.
272. HEADER\_SIZE = 2048
273. header\_sha1 = sha.sha(recovery\_img.data[:HEADER\_SIZE]).hexdigest()
274. sh = """#!/system/bin/sh
275. if ! applypatch -c %(recovery\_type)s:%(recovery\_device)s:%(header\_size)d:%(header\_sha1)s; then
276. log -t recovery "Installing new recovery image"
277. applypatch %(boot\_type)s:%(boot\_device)s:%(boot\_size)d:%(boot\_sha1)s %(recovery\_type)s:%(recovery\_device)s %(recovery\_sha1)s %(recovery\_size)d %(boot\_sha1)s:/system/recovery-from-boot.p
278. else
279. log -t recovery "Recovery image already installed"
280. fi
281. """ % { 'boot\_size': boot\_img.size,
282. 'boot\_sha1': boot\_img.sha1,
283. 'header\_size': HEADER\_SIZE,
284. 'header\_sha1': header\_sha1,
285. 'recovery\_size': recovery\_img.size,
286. 'recovery\_sha1': recovery\_img.sha1,
287. 'boot\_type': boot\_type,
288. 'boot\_device': boot\_device,
289. 'recovery\_type': recovery\_type,
290. 'recovery\_device': recovery\_device,
291. }
292. common.ZipWriteStr(output\_zip, "recovery/etc/install-recovery.sh", sh)
293. **return** Item.Get("system/etc/install-recovery.sh", dir=False)

296. **def** WriteFullOTAPackage(input\_zip, output\_zip):
297. # TODO: how to determine this?  We don't know what version it will
298. # be installed on top of.  For now, we expect the API just won't
299. # change very often.
300. script = edify\_generator.EdifyGenerator(3, OPTIONS.info\_dict)
302. metadata = {"post-build": GetBuildProp("ro.build.fingerprint", input\_zip),
303. "pre-device": GetBuildProp("ro.product.device", input\_zip),
304. "post-timestamp": GetBuildProp("ro.build.date.utc", input\_zip),
305. }
307. device\_specific = common.DeviceSpecificParams(
308. input\_zip=input\_zip,
309. input\_version=OPTIONS.info\_dict["recovery\_api\_version"],
310. output\_zip=output\_zip,
311. script=script,
312. input\_tmp=OPTIONS.input\_tmp,
313. metadata=metadata,
314. info\_dict=OPTIONS.info\_dict)
316. **if** **not** OPTIONS.omit\_prereq:
317. ts = GetBuildProp("ro.build.date.utc", input\_zip)
318. script.AssertOlderBuild(ts)
320. AppendAssertions(script, input\_zip)
321. device\_specific.FullOTA\_Assertions()
323. script.ShowProgress(0.5, 0)
325. **if** OPTIONS.wipe\_user\_data:
326. script.FormatPartition("/data")
328. script.FormatPartition("/system")
329. script.Mount("/system")
330. script.UnpackPackageDir("recovery", "/system")
331. script.UnpackPackageDir("system", "/system")
333. symlinks = CopySystemFiles(input\_zip, output\_zip)
334. script.MakeSymlinks(symlinks)
336. boot\_img = common.File("boot.img", common.BuildBootableImage(
337. os.path.join(OPTIONS.input\_tmp, "BOOT")))
338. recovery\_img = common.File("recovery.img", common.BuildBootableImage(
339. os.path.join(OPTIONS.input\_tmp, "RECOVERY")))
340. MakeRecoveryPatch(output\_zip, recovery\_img, boot\_img)
342. Item.GetMetadata(input\_zip)
343. Item.Get("system").SetPermissions(script)
345. common.CheckSize(boot\_img.data, "boot.img", OPTIONS.info\_dict)
346. common.ZipWriteStr(output\_zip, "boot.img", boot\_img.data)
347. script.ShowProgress(0.2, 0)
349. script.ShowProgress(0.2, 10)
350. script.WriteRawImage("/boot", "boot.img")
352. script.ShowProgress(0.1, 0)
353. device\_specific.FullOTA\_InstallEnd()
355. **if** OPTIONS.extra\_script **is** **not** None:
356. script.AppendExtra(OPTIONS.extra\_script)
358. script.UnmountAll()
359. script.AddToZip(input\_zip, output\_zip)
360. WriteMetadata(metadata, output\_zip)

363. **def** WriteMetadata(metadata, output\_zip):
364. common.ZipWriteStr(output\_zip, "META-INF/com/android/metadata",
365. "".join(["%s=%s\n" % kv
366. **for** kv **in** sorted(metadata.iteritems())]))



371. **def** LoadSystemFiles(z):
372. """Load all the files from SYSTEM/... in a given target-files
373. ZipFile, and return a dict of {filename: File object}."""
374. out = {}
375. **for** info **in** z.infolist():
376. **if** info.filename.startswith("SYSTEM/") **and** **not** IsSymlink(info):
377. fn = "system/" + info.filename[7:]
378. data = z.read(info.filename)
379. out[fn] = common.File(fn, data)
380. **return** out

383. **def** GetBuildProp(property, z):
384. """Return the fingerprint of the build of a given target-files
385. ZipFile object."""
386. bp = z.read("SYSTEM/build.prop")
387. **if** **not** property:
388. **return** bp
389. m = re.search(re.escape(property) + r"=(.\*)\n", bp)
390. **if** **not** m:
391. **raise** common.ExternalError("couldn't find %s in build.prop" % (property,))
392. **return** m.group(1).strip()

395. **def** WriteIncrementalOTAPackage(target\_zip, source\_zip, output\_zip):
396. source\_version = OPTIONS.source\_info\_dict["recovery\_api\_version"]
397. target\_version = OPTIONS.target\_info\_dict["recovery\_api\_version"]
399. **if** source\_version == 0:
400. **print** ("WARNING: generating edify script for a source that "
401. "can't install it.")
402. script = edify\_generator.EdifyGenerator(source\_version, OPTIONS.info\_dict)
404. metadata = {"pre-device": GetBuildProp("ro.product.device", source\_zip),
405. "post-timestamp": GetBuildProp("ro.build.date.utc", target\_zip),
406. }
408. device\_specific = common.DeviceSpecificParams(
409. source\_zip=source\_zip,
410. source\_version=source\_version,
411. target\_zip=target\_zip,
412. target\_version=target\_version,
413. output\_zip=output\_zip,
414. script=script,
415. metadata=metadata,
416. info\_dict=OPTIONS.info\_dict)
418. **print** "Loading target..."
419. target\_data = LoadSystemFiles(target\_zip)
420. **print** "Loading source..."
421. source\_data = LoadSystemFiles(source\_zip)
423. verbatim\_targets = []
424. patch\_list = []
425. diffs = []
426. largest\_source\_size = 0
427. **for** fn **in** sorted(target\_data.keys()):
428. tf = target\_data[fn]
429. **assert** fn == tf.name
430. sf = source\_data.get(fn, None)
432. **if** sf **is** None **or** fn **in** OPTIONS.require\_verbatim:
433. # This file should be included verbatim
434. **if** fn **in** OPTIONS.prohibit\_verbatim:
435. **raise** common.ExternalError("\"%s\" must be sent verbatim" % (fn,))
436. **print** "send", fn, "verbatim"
437. tf.AddToZip(output\_zip)
438. verbatim\_targets.append((fn, tf.size))
439. **elif** tf.sha1 != sf.sha1:
440. # File is different; consider sending as a patch
441. diffs.append(common.Difference(tf, sf))
442. **else**:
443. # Target file identical to source.
444. **pass**
446. common.ComputeDifferences(diffs)
448. **for** diff **in** diffs:
449. tf, sf, d = diff.GetPatch()
450. **if** d **is** None **or** len(d) > tf.size \* OPTIONS.patch\_threshold:
451. # patch is almost as big as the file; don't bother patching
452. tf.AddToZip(output\_zip)
453. verbatim\_targets.append((tf.name, tf.size))
454. **else**:
455. common.ZipWriteStr(output\_zip, "patch/" + tf.name + ".p", d)
456. patch\_list.append((tf.name, tf, sf, tf.size, sha.sha(d).hexdigest()))
457. largest\_source\_size = max(largest\_source\_size, sf.size)
459. source\_fp = GetBuildProp("ro.build.fingerprint", source\_zip)
460. target\_fp = GetBuildProp("ro.build.fingerprint", target\_zip)
461. metadata["pre-build"] = source\_fp
462. metadata["post-build"] = target\_fp
464. script.Mount("/system")
465. script.AssertSomeFingerprint(source\_fp, target\_fp)
467. source\_boot = common.File("/tmp/boot.img",
468. common.BuildBootableImage(
469. os.path.join(OPTIONS.source\_tmp, "BOOT")))
470. target\_boot = common.File("/tmp/boot.img",
471. common.BuildBootableImage(
472. os.path.join(OPTIONS.target\_tmp, "BOOT")))
473. updating\_boot = (source\_boot.data != target\_boot.data)
475. source\_recovery = common.File("system/recovery.img",
476. common.BuildBootableImage(
477. os.path.join(OPTIONS.source\_tmp, "RECOVERY")))
478. target\_recovery = common.File("system/recovery.img",
479. common.BuildBootableImage(
480. os.path.join(OPTIONS.target\_tmp, "RECOVERY")))
481. updating\_recovery = (source\_recovery.data != target\_recovery.data)
483. # Here's how we divide up the progress bar:
484. #  0.1 for verifying the start state (PatchCheck calls)
485. #  0.8 for applying patches (ApplyPatch calls)
486. #  0.1 for unpacking verbatim files, symlinking, and doing the
487. #      device-specific commands.
489. AppendAssertions(script, target\_zip)
490. device\_specific.IncrementalOTA\_Assertions()
492. script.Print("Verifying current system...")
494. script.ShowProgress(0.1, 0)
495. total\_verify\_size = float(sum([i[2].size **for** i **in** patch\_list]) + 1)
496. **if** updating\_boot:
497. total\_verify\_size += source\_boot.size
498. so\_far = 0
500. **for** fn, tf, sf, size, patch\_sha **in** patch\_list:
501. script.PatchCheck("/"+fn, tf.sha1, sf.sha1)
502. so\_far += sf.size
503. script.SetProgress(so\_far / total\_verify\_size)
505. **if** updating\_boot:
506. d = common.Difference(target\_boot, source\_boot)
507. \_, \_, d = d.ComputePatch()
508. **print** "boot      target: %d  source: %d  diff: %d" % (
509. target\_boot.size, source\_boot.size, len(d))
511. common.ZipWriteStr(output\_zip, "patch/boot.img.p", d)
513. boot\_type, boot\_device = common.GetTypeAndDevice("/boot", OPTIONS.info\_dict)
515. script.PatchCheck("%s:%s:%d:%s:%d:%s" %
516. (boot\_type, boot\_device,
517. source\_boot.size, source\_boot.sha1,
518. target\_boot.size, target\_boot.sha1))
519. so\_far += source\_boot.size
520. script.SetProgress(so\_far / total\_verify\_size)
522. **if** patch\_list **or** updating\_recovery **or** updating\_boot:
523. script.CacheFreeSpaceCheck(largest\_source\_size)
525. device\_specific.IncrementalOTA\_VerifyEnd()
527. script.Comment("---- start making changes here ----")
529. **if** OPTIONS.wipe\_user\_data:
530. script.Print("Erasing user data...")
531. script.FormatPartition("/data")
533. script.Print("Removing unneeded files...")
534. script.DeleteFiles(["/"+i[0] **for** i **in** verbatim\_targets] +
535. ["/"+i **for** i **in** sorted(source\_data)
536. **if** i **not** **in** target\_data] +
537. ["/system/recovery.img"])
539. script.ShowProgress(0.8, 0)
540. total\_patch\_size = float(sum([i[1].size **for** i **in** patch\_list]) + 1)
541. **if** updating\_boot:
542. total\_patch\_size += target\_boot.size
543. so\_far = 0
545. script.Print("Patching system files...")
546. **for** fn, tf, sf, size, \_ **in** patch\_list:
547. script.ApplyPatch("/"+fn, "-", tf.size, tf.sha1, sf.sha1, "patch/"+fn+".p")
548. so\_far += tf.size
549. script.SetProgress(so\_far / total\_patch\_size)
551. **if** updating\_boot:
552. # Produce the boot image by applying a patch to the current
553. # contents of the boot partition, and write it back to the
554. # partition.
555. script.Print("Patching boot image...")
556. script.ApplyPatch("%s:%s:%d:%s:%d:%s"
557. % (boot\_type, boot\_device,
558. source\_boot.size, source\_boot.sha1,
559. target\_boot.size, target\_boot.sha1),
560. "-",
561. target\_boot.size, target\_boot.sha1,
562. source\_boot.sha1, "patch/boot.img.p")
563. so\_far += target\_boot.size
564. script.SetProgress(so\_far / total\_patch\_size)
565. **print** "boot image changed; including."
566. **else**:
567. **print** "boot image unchanged; skipping."
569. **if** updating\_recovery:
570. # Is it better to generate recovery as a patch from the current
571. # boot image, or from the previous recovery image?  For large
572. # updates with significant kernel changes, probably the former.
573. # For small updates where the kernel hasn't changed, almost
574. # certainly the latter.  We pick the first option.  Future
575. # complicated schemes may let us effectively use both.
576. #
577. # A wacky possibility: as long as there is room in the boot
578. # partition, include the binaries and image files from recovery in
579. # the boot image (though not in the ramdisk) so they can be used
580. # as fodder for constructing the recovery image.
581. MakeRecoveryPatch(output\_zip, target\_recovery, target\_boot)
582. script.DeleteFiles(["/system/recovery-from-boot.p",
583. "/system/etc/install-recovery.sh"])
584. **print** "recovery image changed; including as patch from boot."
585. **else**:
586. **print** "recovery image unchanged; skipping."
588. script.ShowProgress(0.1, 10)
590. target\_symlinks = CopySystemFiles(target\_zip, None)
592. target\_symlinks\_d = dict([(i[1], i[0]) **for** i **in** target\_symlinks])
593. temp\_script = script.MakeTemporary()
594. Item.GetMetadata(target\_zip)
595. Item.Get("system").SetPermissions(temp\_script)
597. # Note that this call will mess up the tree of Items, so make sure
598. # we're done with it.
599. source\_symlinks = CopySystemFiles(source\_zip, None)
600. source\_symlinks\_d = dict([(i[1], i[0]) **for** i **in** source\_symlinks])
602. # Delete all the symlinks in source that aren't in target.  This
603. # needs to happen before verbatim files are unpacked, in case a
604. # symlink in the source is replaced by a real file in the target.
605. to\_delete = []
606. **for** dest, link **in** source\_symlinks:
607. **if** link **not** **in** target\_symlinks\_d:
608. to\_delete.append(link)
609. script.DeleteFiles(to\_delete)
611. **if** verbatim\_targets:
612. script.Print("Unpacking new files...")
613. script.UnpackPackageDir("system", "/system")
615. **if** updating\_recovery:
616. script.Print("Unpacking new recovery...")
617. script.UnpackPackageDir("recovery", "/system")
619. script.Print("Symlinks and permissions...")
621. # Create all the symlinks that don't already exist, or point to
622. # somewhere different than what we want.  Delete each symlink before
623. # creating it, since the 'symlink' command won't overwrite.
624. to\_create = []
625. **for** dest, link **in** target\_symlinks:
626. **if** link **in** source\_symlinks\_d:
627. **if** dest != source\_symlinks\_d[link]:
628. to\_create.append((dest, link))
629. **else**:
630. to\_create.append((dest, link))
631. script.DeleteFiles([i[1] **for** i **in** to\_create])
632. script.MakeSymlinks(to\_create)
634. # Now that the symlinks are created, we can set all the
635. # permissions.
636. script.AppendScript(temp\_script)
638. # Do device-specific installation (eg, write radio image).
639. device\_specific.IncrementalOTA\_InstallEnd()
641. **if** OPTIONS.extra\_script **is** **not** None:
642. scirpt.AppendExtra(OPTIONS.extra\_script)
644. script.AddToZip(target\_zip, output\_zip)
645. WriteMetadata(metadata, output\_zip)

648. **def** main(argv):
650. **def** option\_handler(o, a):
651. **if** o **in** ("-b", "--board\_config"):
652. **pass**   # deprecated
653. **elif** o **in** ("-k", "--package\_key"):
654. OPTIONS.package\_key = a
655. **elif** o **in** ("-i", "--incremental\_from"):
656. OPTIONS.incremental\_source = a
657. **elif** o **in** ("-w", "--wipe\_user\_data"):
658. OPTIONS.wipe\_user\_data = True
659. **elif** o **in** ("-n", "--no\_prereq"):
660. OPTIONS.omit\_prereq = True
661. **elif** o **in** ("-e", "--extra\_script"):
662. OPTIONS.extra\_script = a
663. **elif** o **in** ("--worker\_threads"):
664. OPTIONS.worker\_threads = int(a)
665. **else**:
666. **return** False
667. **return** True
669. args = common.ParseOptions(argv, \_\_doc\_\_,
670. extra\_opts="b:k:i:d:wne:",
671. extra\_long\_opts=["board\_config=",
672. "package\_key=",
673. "incremental\_from=",
674. "wipe\_user\_data",
675. "no\_prereq",
676. "extra\_script=",
677. "worker\_threads="],
678. extra\_option\_handler=option\_handler)
680. **if** len(args) != 2:
681. common.Usage(\_\_doc\_\_)
682. sys.exit(1)
684. **if** OPTIONS.extra\_script **is** **not** None:
685. OPTIONS.extra\_script = open(OPTIONS.extra\_script).read()
687. **print** "unzipping target target-files..."
688. OPTIONS.input\_tmp = common.UnzipTemp(args[0])
690. OPTIONS.target\_tmp = OPTIONS.input\_tmp
691. input\_zip = zipfile.ZipFile(args[0], "r")
692. OPTIONS.info\_dict = common.LoadInfoDict(input\_zip)
693. **if** OPTIONS.verbose:
694. **print** "--- target info ---"
695. common.DumpInfoDict(OPTIONS.info\_dict)
697. **if** OPTIONS.device\_specific **is** None:
698. OPTIONS.device\_specific = OPTIONS.info\_dict.get("tool\_extensions", None)
699. **if** OPTIONS.device\_specific **is** **not** None:
700. OPTIONS.device\_specific = os.path.normpath(OPTIONS.device\_specific)
701. **print** "using device-specific extensions in", OPTIONS.device\_specific
703. **if** OPTIONS.package\_key:
704. temp\_zip\_file = tempfile.NamedTemporaryFile()
705. output\_zip = zipfile.ZipFile(temp\_zip\_file, "w",
706. compression=zipfile.ZIP\_DEFLATED)
707. **else**:
708. output\_zip = zipfile.ZipFile(args[1], "w",
709. compression=zipfile.ZIP\_DEFLATED)
711. **if** OPTIONS.incremental\_source **is** None:
712. WriteFullOTAPackage(input\_zip, output\_zip)
713. **else**:
714. **print** "unzipping source target-files..."
715. OPTIONS.source\_tmp = common.UnzipTemp(OPTIONS.incremental\_source)
716. source\_zip = zipfile.ZipFile(OPTIONS.incremental\_source, "r")
717. OPTIONS.target\_info\_dict = OPTIONS.info\_dict
718. OPTIONS.source\_info\_dict = common.LoadInfoDict(source\_zip)
719. **if** OPTIONS.verbose:
720. **print** "--- source info ---"
721. common.DumpInfoDict(OPTIONS.source\_info\_dict)
722. WriteIncrementalOTAPackage(input\_zip, source\_zip, output\_zip)
724. output\_zip.close()
725. **if** OPTIONS.package\_key:
726. SignOutput(temp\_zip\_file.name, args[1])
727. temp\_zip\_file.close()
729. common.Cleanup()
731. **print** "done."

734. **if** \_\_name\_\_ == '\_\_main\_\_':
735. **try**:
736. common.CloseInheritedPipes()
737. main(sys.argv[1:])
738. **except** common.ExternalError, e:
739. **print**
740. **print** "   ERROR: %s" % (e,)
741. **print**
742. sys.exit(1)

主函数main是python的入口函数，我们从main函数开始看，大概看一下main函数（脚本最后）里的流程就能知道脚本的执行过程了。

* 1. 在main函数的开头，首先将用户设定的option选项存入OPTIONS变量中，它是一个python中的类。紧接着判断有没有额外的脚本，如果有就读入到OPTIONS变量中。
  2. 解压缩输入的zip包，即我们在上文生成的原始zip包。然后判断是否用到device-specific extensions（设备扩展）如果用到，随即读入到OPTIONS变量中。
  3. 判断是否签名，然后判断是否有新内容的增量源，有的话就解压该增量源包放入一个临时变量中（source\_zip）。自此，所有的准备工作已完毕，随即会调用该 脚本中最主要的函数WriteFullOTAPackage(input\_zip,output\_zip)
  4. WriteFullOTAPackage函数的处理过程是先获得脚本的生成器。默认格式是edify。然后获得metadata元数据，此数据来至于Android的一些环境变量。然后获得设备配置参数比如api函数的版本。然后判断是否忽略时间戳。
  5. WriteFullOTAPackage函数做完准备工作后就开始生成升级用的脚本文件(updater-script)了。生成脚本文件后将上一步获得的metadata元数据写入到输出包out\_zip。

比如：AppendAssertions(script, OPTIONS.info\_dict, oem\_dict)

device\_specific.FullOTA\_Assertions()来实现产品判断

* 1. 至此一个完整的update.zip升级包就生成了。生成位置在：out/target/product/tcc8800/full\_tcc8800\_evm-ota-eng.mumu.20120315.155326.zip。将升级包拷贝到SD卡中就可以用来升级了。

## 增量包update.zip的生成

### 生成增量包步骤

在上面的过程中生成的update.zip升级包是全部系统的升级包。大小有80M多。这对手机用户来说，用来升级的流量是很大的。而且在实际升级中，我们只希望能够升级我们改变的那部分内容。这就需要使用增量包来升级。生成增量包的过程也需要上文中提到的ota\_from\_target\_files.py的参与。

         下面是制作update.zip增量包的过程。

1. 在源码根目录下依次执行下列命令  
      $ . build/envsetup.sh  
      $ lunch 选择17  
      $ make  
       $ make otapackage  
      执行上面的命令后会在out/target/product/tcc8800/下生成我们第一个系统升级包。我们先将其命名为A.zip
2. 在源码中修改我们需要改变的部分，比如修改内核配置，增加新的驱动等等。修改后再一次执行上面的命令。就会生成第二个我们修改后生成的update.zip升级包。将  其命名为B.zip。

③ 在上文中我们看了ota\_from\_target\_files.py脚本的使用帮助，其中选项-i就是用来生成差分增量包的。使用方法是以上面的A.zip 和B.zip包作为输入，以update.zip包作  为输出。生成的update.zip就是我们最后需要的增量包。

     具体使用方式是：将上述两个包拷贝到源码根目录下，然后执行下面的命令。

     $ ./build/tools/releasetools/ota\_from\_target\_files -i A.zip B.zip update.zip。

在执行上述命令时会出现未找到recovery\_api\_version的错误。原因是在执行上面的脚本时如果使用选项i则会调用WriteIncrementalOTAPackage会从A包和B包中的META目录下搜索misc\_info.txt来读取recovery\_api\_version的值。但是在执行make  otapackage命令时生成的update.zip包中没有这个目录更没有这个文档。

       此时我们就需要使用执行make otapackage生成的原始的zip包。这个包的位置在out/target/product/tcc8800/obj/PACKAGING/target\_files\_intermediates/目录下，它是在用命令make otapackage之后的中间生产物，是最原始的升级包。我们将两次编译的生成的包分别重命名为A.zip和B.zip，并拷贝到SD卡根目录下重复执行上面的命令：

     $ ./build/tools/releasetools/ota\_form\_target\_files -i A.zip B.zip update.zip。

     在上述命令即将执行完毕时就可以得到差分包

[如果调试生成过程中问题，这里有个不错的实例](http://blog.csdn.net/mlj1668956679/article/details/10069405)

### 差分包update.zip的更新测试

在上面制作差分包脚本命令中，生成差分包的原理是，参照第一个参数（target\_zip），将第二个参数（source\_zip）中不同的部分输出到第三个参数（output\_zip）中。其中target\_zip与source\_zip的先后顺序不同，产生的差分包也将不同。

在实际的测试过程中，我们的增量包要删除之前添加的一个应用（在使用update.zip全包升级时增加的），其他的部分如内核都没有改动，所以生成的差分包很简单，只有META-INF这个文件夹。主要的不同都体现在updater-script脚本中，其中的#----start make changes here----之后的部分就是做出改变的部分，最主要的脚本命令是： delete(“/system/app/CheckUpdateAll.apk” , “/system/recovery.img”);在具体更新时它将删除CheckUpdateAll.apk这个应用。

为了大家参考，还是把这个差分包的升级脚本贴出来

[plain] [view plaincopy](http://blog.csdn.net/mu0206mu/article/details/7464551)

1. mount("yaffs2", "MTD", "system", "/system");
2. assert(file\_getprop("/system/build.prop", "ro.build.fingerprint") == "telechips/full\_tcc8800\_evm/tcc8800:2.3.5/GRJ90/eng.mumu.20120309.100232:eng/test-keys" ||
3. file\_getprop("/system/build.prop", "ro.build.fingerprint") == "telechips/full\_tcc8800\_evm/tcc8800:2.3.5/GRJ90/eng.mumu.20120309.100232:eng/test-keys");
4. assert(getprop("ro.product.device") == "tcc8800" ||
5. getprop("ro.build.product") == "tcc8800");
6. ui\_print("Verifying current system...");
7. show\_progress(0.100000, 0);
9. # ---- start making changes here ----
11. ui\_print("Removing unneeded files...");
12. delete("/system/app/CheckUpdateAll.apk",
13. "/system/recovery.img");
14. show\_progress(0.800000, 0);
15. ui\_print("Patching system files...");
16. show\_progress(0.100000, 10);
17. ui\_print("Symlinks and permissions...");
18. set\_perm\_recursive(0, 0, 0755, 0644, "/system");
19. set\_perm\_recursive(0, 2000, 0755, 0755, "/system/bin");
20. set\_perm(0, 3003, 02750, "/system/bin/netcfg");
21. set\_perm(0, 3004, 02755, "/system/bin/ping");
22. set\_perm(0, 2000, 06750, "/system/bin/run-as");
23. set\_perm\_recursive(1002, 1002, 0755, 0440, "/system/etc/bluetooth");
24. set\_perm(0, 0, 0755, "/system/etc/bluetooth");
25. set\_perm(1000, 1000, 0640, "/system/etc/bluetooth/auto\_pairing.conf");
26. set\_perm(3002, 3002, 0444, "/system/etc/bluetooth/blacklist.conf");
27. set\_perm(1002, 1002, 0440, "/system/etc/dbus.conf");
28. set\_perm(1014, 2000, 0550, "/system/etc/dhcpcd/dhcpcd-run-hooks");
29. set\_perm(0, 2000, 0550, "/system/etc/init.goldfish.sh");
30. set\_perm\_recursive(0, 0, 0755, 0555, "/system/etc/ppp");
31. set\_perm\_recursive(0, 2000, 0755, 0755, "/system/xbin");
32. set\_perm(0, 0, 06755, "/system/xbin/librank");
33. set\_perm(0, 0, 06755, "/system/xbin/procmem");
34. set\_perm(0, 0, 06755, "/system/xbin/procrank");
35. set\_perm(0, 0, 06755, "/system/xbin/su");
36. set\_perm(0, 0, 06755, "/system/xbin/tcpdump");
37. unmount("/system");

在做更新测试时，我们要以target\_zip系统为依据，也就是更新之前的开发板系统是用target\_zip包升级后的系统。否则会更新就会失败，因为在更新时会从系统对应的目录下读取设备以及时间戳等信息(updater-script脚本一开始的部分)，进行匹配正确后才进行下一步的安装。

所有准备都完成后，将我们制作的差分包放到SD卡中，在Settings-->About Phone-->System Update-->Installed From SDCARD执行更新。最后更新完成并重启后，我们会发现之前的CheckUpdateAll.apk被成功删掉了，大功告成！

V1 = V2 – update.zip

注意：如果有签名的情况，请做相应的处理吧，怎么处理呢，再签名一次就好了？

# [Recovery模式工作原理](http://blog.csdn.net/mu0206mu/article/details/7464987)

在使用update.zip包升级时怎样从主系统（main system）重启进入Recovery模式，进入Recovery模式后怎样判断做何种操作，以及怎样获得主系统发送给Recovery服务的命令，这一系列问题的解决是通过整个软件平台的不同部分之间的密切通信配合来完成的。为此，我们必须要了解Recovery模式的工作原理，这样才能知道我们的update.zip包是怎样一步步进入Recovery中升级并最后到达主系统的。

## Android系统启动模式

首先我们要了解Android系统启动后可能会进入的几种工作模式。先看下图：



由上图可知Android系统启动后可能进入的模式有以下几种：

### MAGIC KEY（组合键）

即用户在启动后通过按下组合键，进入不同的工作模式，具体有两种：

① camera + power：若用户在启动刚开始按了camera+power组合键则会进入bootloader模式，并可进一步进入fastboot（快速刷机模式）。

② home + power ：若用户在启动刚开始按了home+power组合键，系统会直接进入Recovery模式。以这种方式进入Recovery模式时系统会进入一个简单的UI（使用了minui）界面，用来提示用户进一步操作。在tcc8800开发板中提供了一下几种选项操作：

“reboot system now”

“apply update from sdcard”

“wipe data/factory reset”

“wipe cache partition”

### 正常启动

若启动过程中用户没有按下任何组合键，bootloader会读取位于MISC分区的启动控制信息块BCB（Bootloader Control Block）。它是一个结构体，存放着启动命令command。根据不同的命令，系统又 可以进入三种不同的启动模式。我们先看一下这个结构体的定义。

struct bootloader\_message{

char command[32]; //存放不同的启动命令

char status[32]; //update-radio或update-hboot完成存放执行结果

char recovery[1024]; //存放/cache/recovery/command中的命令

};

我们先看command可能的值，其他的在后文具体分析。command可能的值有两种，与值为空（即没有命令）一起区分三种启动模式。

①command=="boot-recovery"时，系统会进入Recovery模式。Recovery服务会具体根据/cache/recovery/command中的命令执行相应的操作（例如，升级update.zip或擦除cache，data等）。

②command=="update-radia"或"update-hboot"时，系统会进入更新firmware（更新bootloader），具体由bootloader完成。

③command为空时，即没有任何命令，系统会进入正常的启动，最后进入主系统（main system）。这种是最通常的启动流程。

Android系统不同的启动模式的进入是在不同的情形下触发的，我们从SD卡中升级我们的update.zip时会进入Recovery模式是其中一种，其他的比如：系统崩溃，或则在命令行输入启动命令式也会进入Recovery或其他的启动模式。

为了解我们的update.zip包具体是怎样在Recovery模式中更新完成，并重启到主系统的，我们还要分析Android中Recovery模式的工作原理。

## Recovery模式中的三个部分

Recovery的工作需要整个软件平台的配合，从通信架构上来看，主要有三个部分。

①MainSystem：即上面提到的正常启动模式（BCB中无命令），是用boot.img启动的系统，Android的正常工作模式。更新时，在这种模式中我们的上层操作就是使用OTA或则从SD卡中升级update.zip包。在重启进入Recovery模式之前，会向BCB中写入命令，以便在重启后告诉bootloader进入Recovery模式。

②Recovery：系统进入Recovery模式后会装载Recovery分区，该分区包含recovery.img（同boot.img相同，包含了标准的内核和根文件系统）。进入该模式后主要是运行Recovery服务（/sbin/recovery）来做相应的操作（重启、升级update.zip、擦除cache分区等）。

③Bootloader：除了正常的加载启动系统之外，还会通过读取MISC分区（BCB）获得来至Main system和Recovery的消息。

## Recovery模式中的两个通信接口

在Recovery服务中上述的三个实体之间的通信是必不可少的，他们相互之间又有以下两个通信接口。

### 通过CACHE分区中的三个文件

Recovery通过/cache/recovery/目录下的三个文件与mian system通信。具体如下

①/cache/recovery/command：这个文件保存着Main system传给Recovery的命令行，每一行就是一条命令，支持一下几种的组合。

--send\_intent=anystring //write the text out to recovery/intent 在Recovery结束时在finish\_recovery函数中将定义的intent字符串作为参数传进来，并写入到/cache/recovery/intent中

--update\_package=root:path //verify install an OTA package file Main system将这条命令写入时,代表系统需要升级，在进入Recovery模式后，将该文件中的命令读取并写入BCB中，然后进行相应的更新update.zip包的操作。

--wipe\_data //erase user data(and cache),then reboot。擦除用户数据。擦除data分区时必须要擦除cache分区。

--wipe\_cache //wipe cache(but not user data),then reboot。擦除cache分区。

②/cache/recovery/log：Recovery模式在工作中的log打印。在recovery服务运行过程中，stdout以及stderr会重定位到/tmp/recovery.log在recovery退出之前会将其转存到/cache/recovery/log中，供查看。

1. /cache/recovery/intent：Recovery传递给Main system的信息。作用不详。

### 通过BCB（Bootloader Control Block）

BCB是bootloader与Recovery的通信接口，也是Bootloader与Main system之间的通信接口。存储在flash中的MISC分区，占用三个page，其本身就是一个结构体，具体成员以及各成员含义如下：

struct bootloader\_message{

char command[32];

char status[32];

char recovery[1024];

};

①command成员：其可能的取值我们在上文已经分析过了，即当我们想要在重启进入Recovery模式时，会更新这个成员的值。另外在成功更新后结束Recovery时，会清除这个成员的值，防止重启时再次进入Recovery模式。

②status：在完成相应的更新后，Bootloader会将执行结果写入到这个字段。

③recovery：可被Main System写入，也可被Recovery服务程序写入。该文件的内容格式为：

“recovery\n

<recovery command>\n

<recovery command>”

该文件存储的就是一个字符串，必须以recovery\n开头，否则这个字段的所有内容域会被忽略。“recovery\n”之后的部分，是/cache/recovery/command支持的命令。可以将其理解为Recovery操作过程中对命令操作的备份。Recovery对其操作的过程为：先读取BCB然后读取/cache/recovery/command，然后将二者重新写回BCB，这样在进入Main system之前，确保操作被执行。在操作之后进入Main system之前，Recovery又会清空BCB的command域和recovery域，这样确保重启后不再进入Recovery模式。

## 如何从Main System重启并进入Recovery模式

我们先看一下以上三个部分是怎样进行通信的，先看下图：



我们只看从Main System如何进入Recovery模式，其他的通信在后文中详述。先从Main System开始看，当我们在Main System使用update.zip包进行升级时，系统会重启并进入Recovery模式。在系统重启之前，我们可以看到，Main System定会向BCB中的command域写入boot-recovery(粉红色线)，用来告知Bootloader重启后进入recovery模式。这一步是必须的。至于Main System是否向recovery域写入值我们在源码中不能肯定这一点。即便如此，重启进入Recovery模式后Bootloader会从/cache/recovery/command中读取值并放入到BCB的recovery域。而Main System在重启之前肯定会向/cache/recovery/command中写入Recovery将要进行的操作命令。

至此，我们就大概知道了，在上层使用update.zip升级时，主系统是怎样告知重启后的系统进入Recovery模式的，以及在Recovery模式中完成什么样的操作。

下一篇开始分析第一个阶段，即我们在上层使用update.zip包升级时，Main System怎样重启并进入Recovery服务的细节流程。

# Uncrypt服务

bootable/recovery/uncrypt/

bootable/recovery/uncrypt/uncrypt.rc

定义了属性启动方法

service uncrypt /system/bin/uncrypt  
service setup-bcb /system/bin/uncrypt --setup-bcb  
 class main  
 socket uncrypt stream 600 system system  
 disabled  
 oneshot  
service clear-bcb /system/bin/uncrypt --clear-bcb

## 参考

recovery uncrypt功能解析

<https://blog.csdn.net/csdn66_2016/article/details/81984399>

# 查分优缺点

三、相同

    1.整包的优势

* 不容易出错-忽略设备上的任意修改
* 不需要依赖特殊的固件

    2.整包的劣势

* 整包的大小比较接近整个固件的镜像，下载更新包将是一个很漫长的过程。
* 存在一定的安全漏洞-可以在固件比较接近的镜像上进行升级实用

    3.差分的优势

* 大小
* 安全-必须是特定的节点才能进行升级
* 相对于整包来说更容易控制

    4.差分的劣势

* 在升级过程中比较容易出错，流程控制上更严谨。

# LOG分析

## recovery.main()->SetSdcardRootPath

I/ [File] : bootable/recovery/recovery.cpp; [Line] : 1439; [Func] : SetSdcardRootPath; ExternalSD1\_ROOT: /mnt/external\_sd1

failed to create /cache dir,err=File exists!

## recovery.main()->ensure\_path\_mounted()

failed to create /cache dir,err=File exists!

W/ [File] : bootable/recovery/roots.cpp; [Line] : 123; [Func] : ensure\_path\_mounted; fs\_type=ext4

## recovery.main()->get\_args

D/ [File] : bootable/recovery/recovery.cpp; [Line] : 2386; [Func] : main; to dump args befor get\_args() :

1 argument(s) :

/sbin/recovery

D/ [File] : bootable/recovery/recovery.cpp; [Line] : 629; [Func] : get\_args; to dump 'boot' :

command :

status :

recovery :

stage :

I/ [File] : bootable/recovery/recovery.cpp; [Line] : 680; [Func] : get\_args; Got arguments from /cache/recovery/command

D/ [File] : bootable/recovery/recovery.cpp; [Line] : 2403; [Func] : main; to dump args after get\_args() :

3 argument(s) :

/sbin/recovery

--update\_package=/mnt/external\_sd/update.zip

--locale=en\_US

locale is [en\_US]

stage is []

reason is [(null)]

## GetUI

相关代码

Device\* device = make\_device();

ui = device->GetUI();

ui->SetLocale(locale);

ui->Init();

ui->SetBackground(RecoveryUI::NONE);

-----------------------------------

can't open /dev/tty0: No such file or directory

fb0 reports (possibly inaccurate):

vi.bits\_per\_pixel = 16

vi.red.offset = 11 .length = 5

vi.green.offset = 5 .length = 6

vi.blue.offset = 0 .length = 5

fi.line\_length = 2160

framebuffer: 0 (1080 x 1920) double\_buffer 1

load png icon\_installing, width 280 height 2975 channels 3

found frames = 7

load png icon\_error, width 254 height 257 channels 3

load png progress\_empty, width 252 height 7 channels 3

load png progress\_fill, width 252 height 7 channels 3

load png stage\_empty, width 50 height 50 channels 1

load png stage\_fill, width 50 height 50 channels 1

installing\_text: en (277 x 33 @ 374)

erasing\_text: en (95 x 33 @ 374)

no\_command\_text: en (150 x 33 @ 374)

error\_text: en (57 x 33 @ 374)

Recovery system v5.0

## 检查文件系统挂载

I/ [File] : bootable/recovery/recovery.cpp; [Line] : 2532; [Func] : main; Recovery system v5.0, built at '11:30:11', on 'Aug 4 2017'.

failed to create /mnt/external\_sd dir,err=File exists!

W/ [File] : bootable/recovery/roots.cpp; [Line] : 123; [Func] : ensure\_path\_mounted; fs\_type=vfat

read cmdline

Not enter sdboot!

auto sdcard update path: /mnt/external\_sd/FirmwareUpdate/auto\_sd\_update.tag

failed to create /metadata dir,err=File exists!

W/ [File] : bootable/recovery/roots.cpp; [Line] : 123; [Func] : ensure\_path\_mounted; fs\_type=ext4

failed to create /radical\_update dir,err=File exists!

W/ [File] : bootable/recovery/roots.cpp; [Line] : 123; [Func] : ensure\_path\_mounted; fs\_type=ext4

## bootmode

bootmode = emmc

Command: "/sbin/recovery" "--update\_package=/mnt/external\_sd/update.zip" "--locale=en\_US"

rm500:/dev/graphics $ getprop | grep emmc

[ro.boot.mode]: [emmc]

[ro.bootmode]: [emmc]

## 打印当前系统属性

## recovery.main()->install.install\_package()开始真正的升级

I/ [File] : bootable/recovery/recovery.cpp; [Line] : 2727; [Func] : main; .KP : to install ota\_package /mnt/external\_sd/update.zip

Finding update package...

I/ [File] : bootable/recovery/install.cpp; [Line] : 348; [Func] : really\_install\_package; Update location: /mnt/external\_sd/update.zip

Opening update package...

failed to create /mnt/external\_sd dir,err=File exists!

W/ [File] : bootable/recovery/roots.cpp; [Line] : 123; [Func] : ensure\_path\_mounted; fs\_type=vfat

## install->really\_install\_package

### checkout\_kpad\_packet(不需要加密可以跳过)

E:is normal ota packetE/ [File] : bootable/recovery/install.cpp; [Line] : 273; [Func] : checkout\_dji\_packet; is normal ota packetI/ [File] : bootable/recovery/verifier.cpp; [Line] : 515; [Func] : load\_keys; read key e=3 hash=32

I/ [File] : bootable/recovery/verifier.cpp; [Line] : 515; [Func] : load\_keys; read key e=3 hash=20

I/ [File] : bootable/recovery/install.cpp; [Line] : 383; [Func] : really\_install\_package; 2 key(s) loaded from /res/keys

Verifying update package...

E:rk verify mode.

E/ [File] : bootable/recovery/verifier.cpp; [Line] : 231; [Func] : verify\_file; rk verify mode.

E:int rk\_verify\_file(unsigned char\*, size\_t) publickey len 2048

E/ [File] : bootable/recovery/verifier.cpp; [Line] : 171; [Func] : rk\_verify\_file; int rk\_verify\_file(unsigned char\*, size\_t) publickey len 2048

E:footer is signed

E/ [File] : bootable/recovery/verifier.cpp; [Line] : 183; [Func] : rk\_verify\_file; footer is signed

E:image size 879951846

E/ [File] : bootable/recovery/verifier.cpp; [Line] : 192; [Func] : rk\_verify\_file; image size 879951846

E:md5 value 2564b515b7d5420f3ca341eb09dff379

E/ [File] : bootable/recovery/verifier.cpp; [Line] : 197; [Func] : rk\_verify\_file; md5 value 2564b515b7d5420f3ca341eb09dff379

E:md5\_decrypt 2564b515b7d5420f3ca341eb09dff379

E/ [File] : bootable/recovery/verifier.cpp; [Line] : 209; [Func] : rk\_verify\_file; md5\_decrypt 2564b515b7d5420f3ca341eb09dff379

E:sign verify pass

E/ [File] : bootable/recovery/verifier.cpp; [Line] : 212; [Func] : rk\_verify\_file; sign verify pass

### 挂载文件，开始升级

W/ [File] : bootable/recovery/roots.cpp; [Line] : 123; [Func] : ensure\_path\_mounted; fs\_type=ext4

failed to create /system dir,err=File exists!

W/ [File] : bootable/recovery/roots.cpp; [Line] : 123; [Func] : ensure\_path\_mounted; fs\_type=ext4

D/ [File] : bootable/recovery/install.cpp; [Line] : 439; [Func] : really\_install\_package; no ru\_pkg is applied.

Installing update...

### try\_update\_binary，调用脚本

// If the package contains an update binary, extract it and run it.

把新的升级包的update.bin提取出来"/tmp/update\_binary";

execv(binary, (char\* const\*)args);

# 升级失败

计算固件MD5值，黑屏自动重生级

V2固件md5值：

wrongFooter is wrong？？？

用imag刷机，cache/recovery也会被全清

last\_log.N.txt:N越大表示

是服务器本身的问题？

是key/dji账号切换问题

和3399同时编译问题导致

# 总结

以上着重分析了Android系统中Recovery模式中的一种，即我们做好的update.zip包在系统更新时所走过的流程。其核心部分就是Recovery服务的工作原理。其他两种FACTORY RESET、ENCRYPTED FILE SYSTEM ENABLE/DISABLE与OTA INSTALL是相

通的。重点是要理解Recovery服务的工作原理。另外详细分析其升级过程，对于我们在实际升级时，可以根据我们的需要做出相应的修改。

# TASK

uboot

bootloader分析<http://blog.omitol.com/2017/09/30/Bypass-QCOM-Secure-Boot/>

update.zip和bin包的有何风险，其实目前本质都一样？

那么为何还要区分image和zip呢

Zip要小得多

img重命名为zip真的可以刷机

直接提供ota的zip包究竟安全不？

# REF

[重点参考文章](http://blog.csdn.net/mu0206mu)

[Android系统Recovery工作原理之使用update.zip升级过程分析](http://blog.csdn.net/mlj1668956679/article/details/10069355)

[二update.zip差分包问题的解决](http://blog.csdn.net/mlj1668956679/article/details/10069405)

[Android系统Recovery工作原理之使用update.zip](http://dengzhangtao.iteye.com/blog/1941808)

[【Android OTA】系统的更新升级](https://moshuqi.github.io/2017/04/23/Android-OTA-%E7%B3%BB%E7%BB%9F%E7%9A%84%E6%9B%B4%E6%96%B0%E5%8D%87%E7%BA%A7/)

[自定义Android系统升级流程](https://www.jianshu.com/p/947be507a73e)