ROM 逆向适配手册

目录

文档修改记录

修改时间	修改人	备注
2015/6/30	段启智	初始文档

1 概览

1.1 基本原理

Android ROM 逆向适配，是指利用**逆向工具**对已有的 ROM 包进行**定制处理**，重新生成一个与已有 ROM 类似且包含定制功能的 ROM 包。

逆向工具有一个共同的特点：将机器解析的文件格式，还原为人可读的文件格式。ROM 逆向适配主要涉及到以下一些逆向工具：

1). Apktool: 用于对 APK 和 JAR 包进行反汇编，生成可读的 smali 文件和 XML 文件。

2). bootimgpack: 用于对 boot.img 进行解包，生成可供修改的文件系统 RAMDISK，另外 linux 内核也会被解包出来，通常命名为 kernel 或 zImage。

3). systemimgpack：用于对 system.img 进行解包，生成可供修改的文件目录 SYSTEM，这个目录下包含了我们需要定制的 APK 和 JAR。

定制处理是将两个 ROM 包进行融合的过程，通常我们是将自己的系统移植到厂商的某一款手机上。我们的 ROM 包命名为 ***board***，厂商的 ROM 包命名为 ***vendor***，完整定制一个 ROM 包会涉及到应用层 Apk，框架层 Jar，Native 层 Lib 以及配置文件 Conf。借助于自动化工具，只需要简单的运行命令，就能完成繁琐的定制过程，最终融合后 ***merged*** 的 ROM 包，主体框架还是厂商的，因为我们不对厂商的驱动层和 HAL 层做任何修改，框架层也只做部分修改。这样，硬件相关的功能，譬如蓝牙、功能、相机等，还是由厂商保证的。

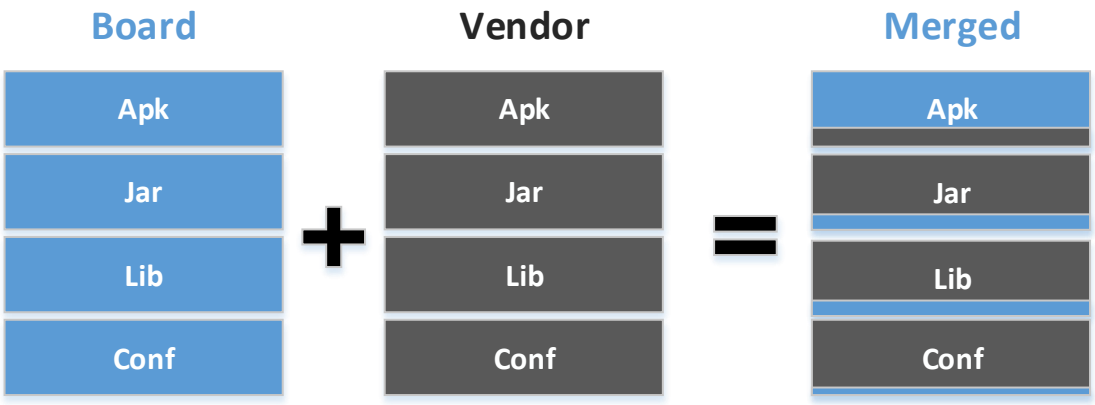

图 1. ROM 定制处理的过程

1). 应用层主要被 ***board*** 的 Apk 占据，同时包含少量 ***vendor*** 的 Apk。这是符合定制需要的，因为应用层体现了用户交互，移植的目的，就是为了让用户体验我们的交互，所以主要还是以我们的应用为主。

2). 框架层主要还是 ***vendor*** 的 Jar，同时我们会对 ***vendor*** 的部分 Jar 进行修改。对厂商的 Jar 进行修改，目的还是为了让我们的应用能够正常的运行，并植入我们一些影响系统行为的功能。

3). Native 层主要还是 ***vendor*** 的 Lib，同时会新增一些 ***board*** 的 Lib。对 Lib 的逆向分析是比较困难的，所以 ROM 逆向适配，一般不会对厂商的 Lib 进行修改，仅仅是新增我们需要的 Lib。

4). 配置文件主要还是 ***vendor*** 的，譬如 build.prop, system/etc 目录下的一些配置，这些都是与系统功能相关，我们修改的配置一般就是为了配合框架层的修改。

1.2 适配工具

适配工具是一整套子工具的合集，它对逆向工具、打包工具、自动化工具进行封装整合，提供机型适配环境和编译环境，梳理出了完整易用的**适配流程**。对适配工具而言，输入就是两个 ROM 包，即 ***board*** 和 ***vendor***；输出就是一个融合的 ROM 包，即 ***merged***。

适配工具已经内置了 ***board*** 的 ROM 包，需要做的就是将其适配到厂商的某款手机上。
通常，可以从手机厂商的官网或者第三方渠道(譬如 Cyanogenmod)找到待适配手机的 ROM 包，即 ***vendor***。找到一个稳定的厂商 ROM 包，刷入手机后功能正常，就可以开始这款机型的适配了：

1). 配置适配信息(**config**)，目的是为了生成待适配机型的配置文件 Makefile，这个文件的内容将指导后续的行为。Makefile 中一些配置项的值，是工具自动从当前机型中获取的，譬如机型的屏幕密度 DENSITY、分辨率 RESOLUTION，这要求连接上手机才能正确获取这些配置项。

2). 新建机型适配工程(**newproject**)，目的是为了生成一个与机型适配相关的文件目录。后续的适配工作都会聚焦在这个目录。工具会从厂商的 ROM 包中提取出适配需要的内容，包括：SYSTEM 分区的所有文件、文件的属性和访问权限、APK 签名信息等。提取完成后，会对文件做一次处理，譬如将 odex 格式还原为 dex 格式、将部分 Jar 包反汇编。

3). 自动化插桩(**patchall**)，目的是为了将我们在框架层的改动，自动合入到厂商的框架层代码中去。这个过程的原理是“diff-patch”，对比 ***board*** 和 ***aosp***，将差异的代码以补丁的形式打到 ***vendor*** 的代码中去。自动化插桩可能会导致文件冲突(**conflict**)，因此这个过程后，还需要开发人员手工解决冲突。工具采用了“三路对比”、“Smali 文件拆分”、“Smali 内容转换”等技术，来减少自动化插桩产生的冲突。同时，“多 BASE 机型”设计也可以极大的减少冲突。这些技术字眼很生涩，需要开发者在使用的过程中慢慢体会。

4). 生成新的 ROM 包(**fullota**)，目的是为了生成可以刷入手机的文件。这个过程会根据 Makefile 的指导，来完成不同的动作，譬如：配置厂商的哪些文件是需要保留的或者修改的、配置哪些 APK 需要转换资源 ID、配置最终生成产物的形式等。工具提供的编译环境，来完成对文件的重新打包，包括重新打包 boot.img、apk、jar、system.img 等。大部分 apk 会被重新签名(在 Selinux 环境下尤为重要)。

2. 准备工作

2.1 系统环境

推荐使用 Ubuntu12.04+，推荐为适配分配 20G 以上的工作分区。

Windows 用户，可以通过虚拟机安装 Ubuntu 镜像。

2.2 开发环境

开发环境依赖于 jdk、git、curl、repo，安装方法如下：

```
$ sudo apt-get install openjdk-7-jdk                 # Android 5.0 以上的移植，需要安装 JDK7
$ sudo apt-get install git                           #  适配工具是通过 git 进行版本管理的
$ sudo apt-get install curl                          #  需要 curl 来下载 repo
$ mkdir -p ~/bin && export PATH=~/bin:$PATH          #  新建一个 bin 目录，并添加至环境变量
$ curl https://raw.githubusercontent.com/Flyme-OS/manifest/lollipop-5.0/repo > ~/bin/repo
$ chmod a+x ~/bin/repo                               #  为 repo 添加执行权限
```

配置好依赖的系统环境后，就可以下载适配工具了：

```
$ repo init -u https://github.com/FlymeOS/manifest.git -b flyme-4.5
$ repo sync
```

如果连接一直失败或者下载代码过慢，可以使用如下命令下载适配工具：

```
$ repo init --repo-url https://github.com/FlymeOS/repo.git \
            -u https://github.com/FlymeOS/manifest.git -b flyme-4.5 \
            --no-repo-verify -b flyme-4.5
$ repo sync --no-clone-bundle -c -j4
```

2.3 选择底包

ROM 逆向适配是将 ***board*** 和 ***vendor*** 进行融合，所以选择合适的 ROM 包很重要。

适配工具提供的 ROM 包(***board***)会定期更新，包括演进新功能、修复 Bug、升级 Android 版本等。除了使用适配工具默认提供的 ROM 包，开发者也可以自行准备一个 ***board***。

选择厂商的 ROM 包(***vendor***)应遵循以下原则：

1). **稳定**：一个 Bug 少的厂商 ROM 包将会减少最终融合 ROM 包的底层 Bug。因为逆向适配保留的是厂商 ROM 包的底层框架，厂商 ROM 包的稳定性直接决定了最终的稳定性。

2). **Android 版本**：如厂商的 ROM 包有 5.0 版本 5.1 的版本，但 ***board*** 的版本只有 5.0，那么，建议使用厂商 5.1 版本的 ROM 包。

3. 机型适配

机型适配就是逆向工程思想的实践，利用工具将两个 ROM 包进行融合。然而，要适配出一个功能稳定的新 ROM 包，是有很多兼容性问题的需要解决的，这需要对 Android 有一定程度的了解，甚至需要深入分析两个 ROM 包的实现逻辑，尤其在没有源码的情况下，开发者只能分析汇编级别的 smali 代码，这极大考验着开发者的耐心。

任何一项技术的掌握，都遵循由浅入深的规律。对于 ROM 逆向适配而言的入门者而言，一开始对 Android 不了解没有关系，对 Smali 不了解没有关系，对 Linux 不了解也没有关系，适配一个具体的机型，就是学习和交流的过程，适配经验是攒出来的。

3.1 前提条件

3.1.1 获取 ROOT 权限

获取 ROOT 权限是后续适配的基础。ROOT 可以分为内核 ROOT 和漏洞 ROOT。

1). 内核 ROOT 是指修改 boot.img 中的 default.prop 文件，使 adbd 以 ROOT 权限执行；

2). 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，就可以进行 ROM 逆向适配。

3.1.2 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
```

上述命令实际上是运行了 build 目录下的一个 shell 脚本，目的是初始化适配环境，同时，也得到了一些新的方便使用的命令，譬如：croot, unpack_bootimg, idtoname, apktool, rmline 等。

3.1.3 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 -p devices/xxx          # 自定义待开发的机型名称
$ cd devices/xxx                # 后续的开发工作都在机型目录中完成
```

每新增一个机型，就可以在 devices 目录下新建一个目录，其名称是完全自定义的(推荐以机

型的 model 值来命名，譬如 HTC G2 的国际版 model 值为 d802)。

3.2 适配流程

推荐开发者在有实际手机的情况下进行适配，一方面，在 Android 5.0 上，有一些必要的文件信息需要从手机上获取；另一方面，很多适配出现的问题，是需要在真机上调试才能解决的。

即便没有手机，也可以将厂商的 ROM 包命名为 ***ota.zip***，置于机型的根目录进行适配。除非对机型适配已经非常熟练，否则，并不推荐脱离手机进行适配。

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 配置机型(config)

开发一款新机型，需要 recovery.fstab 和配置文件 Makefile。使用以下命令，可以自动生成这些初始文件：

```
$ flyme config                    # 该命令的功能等价于 makeconfig
```

该命令会尝试自动从手机中提取 boot.img 和 recovery.img。如果提取失败，则需要开发者自行准备 boot.img 和 recovery.img，置入机型根目录下，再重新执行该命令。

recovery.fstab 是 Android 的分区表，通常在 recovery.img 中会包含这个文件。工具会自动从机型根目录下 recovery.img 抽取出 recovery.fstab。在生成刷机包时，需要用到 recovery.fstab。

3.2.2 构建工程(newproject)

在 Makefile，recovery.fstab 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中抽取出所有原厂相关的文件：

```
$ flyme newproject                    # 该命令的功能等价于 make 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
```

其中 vendor 目录包含了所以从厂商提取出的文件信息，这个目录下的内容，一般我们不做修改，而是将要修改的内容再解析出来，放到根目录下，譬如 framework-res 就是 vendor/system/framework-res.apk 反汇编的产物，将要修改的文件解析到根目录这个过程是由工具自动完成的，同样的，还有 framework.jar.out 是 vendor/system/framework.jar 反汇编的产物，services.jar.out 是 vendor/system/services.jar 的反汇编产物。

3.2.3 自动插桩(patchall)

在新机型工程生成完毕之后，执行以下命令会完成自动插桩：

```
$ flyme patchall                      # 该命令的功能等价于 make patchall
```

在执行该命令时，会在当前机型的根目录下生成名为 autopatch 的子目录，包含如下内容：

```
autopatch/
 ├──── aosp/              # AOSP(Android Open Source Project)的反编译代码
 ├──── bosp/              # BOSP(Board Open Source Project)的反编译代码
 ├──── patchall.xml       # autopatch 所涉及到的改动文件列表
 ├──── vendor_patched/    # 自动插桩之后的厂商文件
 ├──── vendor_orig/       # 自动插桩之前的厂商文件
 ├──── reject/            # 产生冲突的文件
```

注：autopatch 目录下的所有内容，都是为了自动插桩和解决冲突所用，不需要进行修改。

自动插桩的过程，就是对比 aosp 和 bosp 的改动，生成一个文件改动列表 patchall.xml，然后将差异的地方，以补丁的形式 patch 到机型根目录下*.jar.out 和 framework-res 目录对应的文件中去。插桩过程中产生冲突的文件，会保留在 reject 子目录中；插桩之前的代码 vendor_orig 和插桩之后的代码 vendor_patched，都保留下来了方便开发者解决冲突，

3.2.4 冲突处理(conflict)

之所以在这些文件中会产生冲突，是因为 ***board*** 和 ***vendor*** 在这些文件的相同地方对 aosp 有改动。如果厂商对 aosp 的改动小，那冲突自然就少。自动插桩会将能够插入的代码，全都插入到*.jar.out 和 framework-res 中，无法插入的代码就以冲突的形式保存在 autopatch/reject 目录中。

针对一个具体文件 services.jar.out/smali/com/android/server/am/ActivityManagerService.smali，自动插桩后，部分代码已经插入了该文件，但产生冲突代码并没有插入该文件，而是在 autopatch/reject/services.jar.out/smali/com/android/server/am/ActivityManagerService.smali 这个文件中，以如下形式的标注：

```
<<<<<<< VENDOR
    vendor 的代码块
=======
    board 的代码块
>>>>>>> BOSP
```

一个冲突块起始于“<<<<<<<”，终止于“>>>>>>>”，中间用“=======”进行了分割，上半块的代码是 ***vendor*** 的代码块，下半块的代码是 ***board*** 的代码块，这表示这两块代码与 aosp 的代码都不同，即 ***board*** 和 ***vendor*** 在这个地方对 aosp 有改动。

解决冲突，需要在 services.jar.out/smali/com/android/server/am/ActivityManagerService.smali 这个文件中，对冲突的代码进行手工调整。一般可以遵循以下三个步骤：

1). **找到冲突位置**。在机型根目录下，对比*.jar.out 和 autopatch/reject/*.jar.out 两个目录。可以找到所有产生冲突的文件，以及冲突的具体位置，每一个冲突都有编号，从 Conflict 0 开始。

注：推荐使用 BeyondCompare 进行目录对比

2). 分析冲突原因。找到冲突位置后，对比 autopatch/aosp 和 autopatch/bosp 两个目录，找到在冲突位置处，***board*** 是如何对 aosp 进行修改的，同时可以根据冲突的内容，看到 ***vendor*** 是如何对 aosp 进行修改的。

3). 确定冲突解法。有些冲突容易解决，甚至可以瞬间解决。一些难以解决的冲突依赖于冲突位置处的上下文，很多时候都是由于 ***board*** 和 ***vendor*** 在 smali 寄存器变量的使用差异导致的，我们需要从上下文中判断出寄存器变量的语义。这考察开发者着耐心。

3.2.5 编译机型(fullota)

在正确解决完冲突以后，可以通过以下命令开始编译整个机型：

```
$ flyme fullota                    # 该命令的功能等价于 make fullota
```

该命令会对所有修改的文件重新打包，完成 ***board*** 和 ***vendor*** 的最后融合。，如果编译成功，将会生成一些用于刷机的文件，包括新的 ROM 包、system.img 等。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，一些常见问题的解决办法可以参考常见问题。

3.2.6 版本升级(upgrade)

在适配完一款新机型后，就可以发布到公开市场供刷机爱好者下载了。当 ***board*** 有版本更新时，开发者只需要使用如下命令就可以将将最新的改动自动插入到已有的代码：

```
$ flyme upgrade
```

该命令会自动跟踪的 BASE 机型，将最新的改动自动插桩，原理与 patchall 相同，不过要自动插桩的文件仅仅是两个 ***board*** 版本之间的差异。在 autopatch 目录下，会生成如下目录：

```
autopatch/
├── last_bosp/          # 上一次的 BOSP 反汇编代码
├── bosp/               # 最新的 BOSP 反汇编代码
├── upgrade.xml         # 更新操作所涉及到的改动文件列表
```

当需要从指定的上一个版本升级时，可以使用如下命令：

```
$ flyme upgrade LAST_COMMIT=xxx
```

通过 LAST_COMMIT 参数，指定了上一个提交的 SHA1 值。LAST_COMMIT 是从 BASE 机型的提交记录中获取的，可以在 BASE 机型目录下运行 git log 命令看到所有的提交记录。使用该命令会自动插入从 BASE 机型的 LAST_COMMIT 开始到最新的提交之间的差异。

3.3 适配技巧

3.3.1 flyme 命令集

相比 make 命令而言，更推荐使用 flyme 命令集。flyme 命令集是在 make 命令上的封装，完成 make 命令相同功能的同时，提供了更多的错误帮助信息。可以使用如下命令来查看 flyme 命令集的帮助文档：

```
$ flyme help
```

flyme 命令集对整个适配流程进行了封装，原则上提供“**一键适配**”的功能，当我们熟悉某款机型，就可以使用如下命令来完成所有的适配操作，该命令会依次运行 config、newproject、patchall、fullota 这些流程，并自动构建 git 库。当某个流程节点出错时，只需要修复错误，再次敲击以下命令，就可以接续的跑完流程：

```
$ flyme fire
```

3.3.2 upgrade 两个 ROM 包

upgrade 命令除了基于 BASE 机型自动插桩升级，还可以基于两个 ROM 包自动插桩。只需要取两个不同的 ROM 包，分别命名为 ***last_board.zip*** 和 ***board.zip***，置入机型根目录的 board 子目录下，然后运行以下命令：

```
$ flyme upgrade
```

该命令不需要借助于 git，会在机型的 autopatch 目录下生成 last_bosp 和 bosp 目录，分别对应两个 ROM 包逆向处理的结果，同时也会对比 last_bosp 和 bosp 的差异，生成 upgrade.xml，自动插桩原理与 patchall 是一致的。

3.3.3 porting 已有机型

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，官方适配 LG G2(D802)这款机型，那么再适配 LG 的其他机型时，就可以使用如下命令进行插桩(而不是使用 patchall)，这样做的效果是：能够最大限度的减少冲突。

```
$ make porting BASE=xxx
```

通过 BASE 参数指定要从哪个机型来移植提交，该命令触发时，会罗列出 BASE 机型所有的提交记录，每一个提交记录都由一个 7 位的 SHA 值唯一标识。通过命令行交互，选择要移植的提交范围即可。以待适配的机型 test 为例，在执行完 config 和 newproject 后，我们使用了 porting 命令，出现了如下的交互界面：

```
duanqizhi@xo:/Flyme-OS/devices/test$ make porting BASE=base
>>> Porting ...
I/precondition: Start preparing essential files in ./autopatch/
  73e8478  update patch for base to 20150530
  e68b685  update some files
  630969b  update the Makefile
  9fba8b7  refine boot.img.out and vendor/META/apkcerts.txt
  e549328  add some missing files
  fc5f3e4  remove unnecessary files
  916d67b  newproject
  59b5836  config
  8ca466f  Initial empty repository

  Each 7 bits SHA1 code identify a commit on base,   You could input:
  - Only one single commit, like: 73e8478
    will porting changes between the selected and the latest from base to your device
  - Two commits as a range, like: 8ca466f 73e8478
    will porting changes between the two selected from base to your device

>>> Input the 7 bits SHA1 commit ID (q to exit): 916d67b 73e8478
```

首先，呈现了一个 BASE 的提交列表，每一行都包含 7 位的提交 ID，提交内容的简要描述。然后，提示开发者选择 7 位的提交 ID。如果只选择一个，就会移植从选择的提交 ID 开始到

最新的提交之间的差异；如果选择两个，就会移植所选择的两个提交 ID 之间的差异。

最后，我们选择了从 916d67b 到 73e8478 这两个提交 ID。那么接下来的过程，就是自动插桩这两个提交之间的差异了，原理与 patchall 相同，所以冲突的解法也是一样的。

也可以通过 porting 命令的参数来指定提交 ID：

```
$ make porting BASE=xxx COMMIT1=xxx COMMIT2=xxx
```

该命令直接指定了两个提交 ID，COMMIT1 和 COMMIT2 所指定的 ID，同交互的方式选择的一致。当熟悉 BASE 机型的提交记录时，使用这种方式，能够更便捷的移植。

可以将一些常用的输入参数配置在 Makefile 中，避免每次在命令行敲入相同的参数。譬如在 Makefile 中配置 BASE := xxx，这样，命令行输入 porting 命令时，就不再需要指定 BASE 参数了。

4. 常用命令

命令	说明	备注
新机型适配相关		
flyme config	需连上 USB 数据线使用。 自动生成机型配置文件 Makefile	功能同 makeconfig
flyme newproject	推荐通过 USB 数据线连接手机。 构建一个新的机型工程。	该命令会根据 Makefile 的配置，在根目录下生成必要的文件
flyme patchall	自动插桩所有的改动，需要手工解决自动插桩的冲突。	该命令原则上可以多次运行。
编译相关的命令		
flyme fullota	编译整个机型，生成可用的 ROM 和通过 fastboot 刷入手机的分区镜像。	该命令等价于 make，make fullota， make otapackage。
flyme clean	清除机型编译的产物，包括 out 目录和 board 目录下的临时文件。	该命令等价于 make clean 当需要重新编译机型时，可以使用 clean，把上一次残留的结果删除。
flyme cleanall	清除机型插桩与编译相关的产物，包括 board，autopatch，out 这三个目录	该命令等价于 make clean-all 当需要编译新版本或重新插桩时，可以使用先 cleanall 命令。
flyme XXX	编译单个模块。XXX 为需要修改的模块名称，譬如 framework、services	该命令等价于 make XXX。
flyme XXX.phone	需连上 USB 数据线使用。编译单个模块，并将编译产出 push 到手机。	该命令等价于 make XXX.phone
升级 ROM 版本相关的命令		
flyme upgrade	跟踪 BASE 机型，自动插桩从上一次开始到最新提交之间的差异。 该命令还支持一些参数。	该命令等价于 make upgrade。 要编译出新版本，还需要接着使用 fullota 命令