Android 15 OTA
开发指南

版本号: 1.0
发布日期: 2024.10.31

版本历史

版本号	日期	制/修订人	内容描述
1.0	2024.10.31	AWA1633	初始版本文档

目　录

插　图

1 前言

1.1 编写目的

本文档目的是让系统开发人员了解 OTA 升级的概念与整体架构，掌握 OTA 升级的流程以及使用方法，并了解 Android OTA 的定制化设计。

1.2 适用范围

本模块适用于 Android 15 系统。

1.3 相关人员

系统开发人员。

1.4 专业术语

- OTA: (over-the-air) 空中下载技术。指 Android 系统提供的标准软件升级方式，即通过无线网络下载更新包并无损失地升级系统，而无需通过有线方式进行连接。
- boot: boot 分区，包含 Linux 内核以及最小的 root 文件系统。它负责挂载 system 分区以及其他分区。
- system: system 分区，包含 AOSP(Android Open Source Project) 的系统应用程序以及库文件。正常操作下，该分区是以只读形式挂载。它的内容只能够在 OTA 升级过程中改变。
- recovery: 包含第二个 Linux 系统，包括 Linux 内核以及名为 recovery 的二进制可执行文件，recovery 的作用是用于读取更新包并将其内容更新至其他分区。
- vendor: vendor 分区包含在 AOSP(Android Open Source Project) 中不包含源码的系统程序以及库文件，该分区以只读形式挂载，它的内容只有在 OTA 升级才会被改变。
- data：用于存储 ota 包，在特殊情况下可更改 data 分区的内容。
- Virtual AB（简称：VAB）与 AB 系统类似的，不同在于 Virtual AB 的 super 分区是虚拟的，动态的。
- vendor_dlkm: vendor_dlkm 分区，位于 super 分区中。包含原来 vendor 分区下的 lib/modules 目录下的 ko 文件，为了方便 ko 文件单独地进行 OTA 升级。

2 基础介绍

2.1 基础简介

2.1.1 Device-mapper

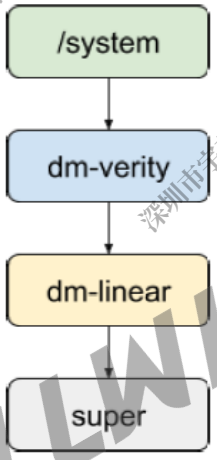

图 2-1: Device Mapping

System 分区的 dm 设备挂载栈如图上所示。

1. 最底层是物理分区：super 分区。
2. 第二层是用 dm-linear 技术实现的动态逻辑分区：为/dev/block/mapper/system_a 分区。
3. 第三层是用 dm-verity 技术实现的校验分区：为/dev/block/mapper/system-verity 分区。
4. 最后 system 分区挂载在/dev/block/mapper/system-verity 分区上面。

2.1.2 Dm-snapshot

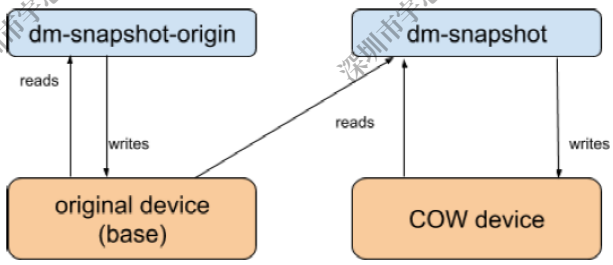

图 2-2: Dm Snapshot

Dm-snapshot 分区由四个设备组成。

1. 通常 system 分区被作为 base 设备。
2. COW 设备用来保存 base 设备所改动的文件。
3. Dm-snapshot 设备由 snapshot 目标创建。往 dm-snapshot 设备写文件，会写到 COW 设备中。从 dm-snapshot 设备读文件，则会从 base 设备和 COW 设备中读文件，如果文件通过 dm-snapshot 改变过，则从 COW 设备中读；否则从 base 设备中读。
4. dm-snapshot-origin 设备由 snapshot-origin 目标创建。读写文件都是从 base 设备中进行读写。

2.1.3 misc 分区介绍

misc 分区用于 BootControl、recovery 以及 bootloader 之间的通信。

```
0~4K:
struct bootloader_message_ab {
  //和non ab的bootloader_message是一样的，存储bootloader与recovery通信相关信息
  struct bootloader_message message;
  //ab 系统slot相关信息
  char slot_suffix[32];
  //用于存储Omaha更新信道，如果update_engine带Omaha编译
  char update_channel[128];
  //保留字
  char reserved[1888];
};

bootloader_message_ab.slot_suffix = bootloader_control

0 ~ 2k: bootloader_message

2k ~ 2k + 32:
struct bootloader_control {
  //NUL 终止的活动slot后缀
  char slot_suffix[4];
  //Bootloader Control AB 的魔数（也就是一个整数常量）
  uint32_t magic;
  //正在使用的结构版本
```

```
    uint8_t version;
    //正在管理的slot个数
    uint8_t nb_slot : 3;
    //尝试启动恢复的剩余次数
    uint8_t recovery_tries_remaining : 3;
    //动态分区的挂起snapshot合并的状态
    uint8_t merge_status : 3;
    //保留字
    uint8_t reserved[1];
    //所有的slot的信息，最多四个slot
    struct slot_metadata slot_info[4];
    //保留字
    uint8_t reserved1[8];
    //此字段之前的所有28个字节的CRC32（小尾数格式）
    uint32_t crc32_le;
} __attribute__((packed));

struct slot_metadata {
    //slot的优先级，15为最高，1为最低，0则为不可启动
    uint8_t priority : 4;
    //slot的尝试启动剩余次数
    uint8_t tries_remaining : 3;
    //slot是否成功启动，成功为1，否则为0
    uint8_t successful_boot : 1;
    //slot是否因为dm-verity损坏而损坏，是为1，否则为0
    uint8_t verity_corrupted : 1;
    //slot的保留字
    uint8_t reserved : 7;
} __attribute__((packed));
```

2.1.4 分区介绍

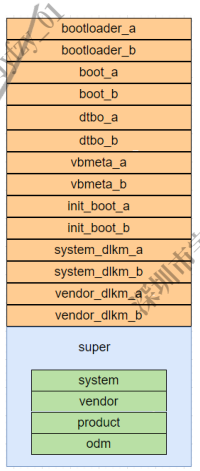

图 2-3: VAB Partitions

除了 super 分区，其他分区如果需要升级，都需要分为 AB 分区。因为 super 分区不用分 AB，而其他分区又比较小，所以 virtual AB 比 AB 系统要节省很多空间。

基础配置及依赖在 Android 方案配置下。

```
# device/softwinner/saturn/common/storage/config.mk

PRODUCT_VIRTUAL_AB ?= true

ifeq ($(PRODUCT_VIRTUAL_AB), true)

# boot hal
PRODUCT_PACKAGES += \
    android.hardware.boot@1.2-service \
    android.hardware.boot@1.2-impl \
    android.hardware.boot@1.2-impl.recovery

# Virtual AB

PRODUCT_PACKAGES += \
  bootctrl.updateboot \
  update_engine_sideload \
  update_boot  \
  update_recovery_boot  \
  update_engine  \
  update_verifier

PRODUCT_PACKAGES_DEBUG += \
    bootctl \
    update_engine_client \
    r.vendor

PRODUCT_HOST_PACKAGES += \
    brillo_update_payload

# Enable Virtual A/B
$(call inherit-product, $(SRC_TARGET_DIR)/product/virtual_ab_ota/compression_with_xor.mk)
endif

PRODUCT_PACKAGES += Update

# device/softwinner/common/grf/config.mk

CONFIG_AW_BUILD_VENDOR_TARGET_FILES_ONLY ?= false
CONFIG_AW_BUILD_FRAMEWORK_TARGET_FILES_ONLY ?= false

ifeq ($(CONFIG_AW_BUILD_VENDOR_TARGET_FILE_ONLY),true)
PRODUCT_BUILD_SYSTEM_IMAGE := false
PRODUCT_BUILD_PRODUCT_IMAGE := false
PRODUCT_BUILD_SYSTEM_EXT_IMAGE := false
PRODUCT_BUILD_SYSTEM_OTHER_IMAGE := false
PRODUCT_BUILD_SYSTEM_DLKM_IMAGE := false
PRODUCT_BUILD_PVMFW_IMAGE := false

PRODUCT_BUILD_SUPER_PARTITION := false
PRODUCT_BUILD_SUPER_EMPTY_IMAGE := false
endif

ifeq ($(CONFIG_AW_BUILD_FRAMEWORK_TARGET_FILE_ONLY),true)
PRODUCT_BUILD_BOOT_IMAGE := false
PRODUCT_BUILD_DEBUG_BOOT_IMAGE := false
PRODUCT_BUILD_DEBUG_VENDOR_BOOT_IMAGE := false
PRODUCT_BUILD_INIT_BOOT_IMAGE := false
PRODUCT_BUILD_RAMDISK_IMAGE := false
PRODUCT_BUILD_VENDOR_BOOT_IMAGE := false
```

```
PRODUCT_BUILD_VENDOR_DLKM_IMAGE := false
PRODUCT_BUILD_VENDOR_IMAGE := false
PRODUCT_BUILD_VENDOR_KERNEL_BOOT_IMAGE := false
PRODUCT_BUILD_DTBO_IMAGE := false

PRODUCT_BUILD_RECOVERY_IMAGE := false
PRODUCT_BUILD_ODM_DLKM_IMAGE := false
PRODUCT_BUILD_ODM_IMAGE := false
PRODUCT_BUILD_VBMETA_IMAGE := false
PRODUCT_BUILD_USERDATA_IMAGE := false
PRODUCT_BUILD_CACHE_IMAGE := false

PRODUCT_BUILD_SUPER_PARTITION := false
PRODUCT_BUILD_SUPER_EMPTY_IMAGE := false
else
PRODUCT_BUILD_VENDOR_BOOT_IMAGE := true
endif

AB_OTA_UPDATER := true
AB_OTA_PARTITIONS += \
  vbmeta_vendor \
  vbmeta_system

ifneq ($(PRODUCT_BUILD_VENDOR_IMAGE),false)
AB_OTA_PARTITIONS += vendor
endif
ifneq ($(PRODUCT_BUILD_VENDOR_DLKM_IMAGE),false)
AB_OTA_PARTITIONS += vendor_dlkm
endif
ifneq ($(PRODUCT_BUILD_BOOT_IMAGE),false)
AB_OTA_PARTITIONS += boot
endif
ifneq ($(PRODUCT_BUILD_VENDOR_BOOT_IMAGE),false)
AB_OTA_PARTITIONS += vendor_boot
endif
ifneq ($(PRODUCT_BUILD_INIT_BOOT_IMAGE),false)
AB_OTA_PARTITIONS += init_boot
endif
ifneq ($(PRODUCT_BUILD_VBMETA_IMAGE),false)
AB_OTA_PARTITIONS += vbmeta
endif
ifneq ($(PRODUCT_BUILD_SYSTEM_DLKM_IMAGE),false)
AB_OTA_PARTITIONS += system_dlkm
endif
ifneq ($(PRODUCT_BUILD_SYSTEM_IMAGE),false)
AB_OTA_PARTITIONS += system
endif
ifneq ($(PRODUCT_BUILD_PRODUCT_IMAGE),false)
AB_OTA_PARTITIONS += product
endif
ifneq ($(PRODUCT_BUILD_DTBO_IMAGE),false)
AB_OTA_PARTITIONS += dtbo
endif
```

3 OTA 升级流程

3.1 OTA 运行原理

Android 平台提供 Google diff arithmetic 差分机制，升级包支持完整升级以及差分升级，OTA 运行原理图如下所示。

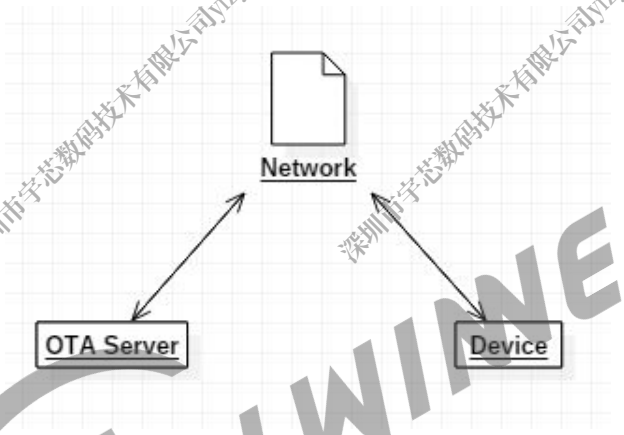

图 3-1: OTA 运行原理图

1. OTA Server 负责对更新包进行上传，下载以及版本的管理。
2. 开发者在修改 Android 系统后，通过差分制作工具制作出差分包，并使用客户端进行更新包上传和版本管理。
3. 设备通过 wifi 网络进行连接和下载，最后完成更新工作。

3.2 OTA 升级流程介绍

3.2.1 非安全 OTA 升级流程

更新包的目录结构。

```
.
├── META-INF
│   └── com
│       └── anroid
│           ├── metadata（更新包的内容信息，大小，位置等）
│           └── otacert (签名证书)
└── payload.bin(更新主要内容)
```

└──payload_properties.txt（更新包的大小，哈希值等）

典型的 OTA 升级流程具体可以分为如下步骤：

1. 设备会周期性检查 OTA 服务器，并确认更新包的可升级性。用户也可以通过将更新包放入内部存储，或者以 U 盘，SD 卡，移动硬盘的形式进行升级。
2. 更新包下载到 data 分区（可放置 data/ota_package，并给予文件 777 权限，selinux 相关已配置）。
3. 需要先提取更新包的 payload_properties.txt，并根据其信息对更新包做校验，校验更新内容和对应文件的完整性。
4. 调用 RecoverySystem.verifyPackage 进行升级校验（加密签名将会利用system/etc/security/otacerts.zip文件进行验证），调用 UpdateEngine.applyPayload 进行升级。
5. UpdateEngine 升级流程内容过多，将在新的一节描述。
6. 在校验阶段结束后，增加了 uboot、bootloader 等定制化更新内容。
7. 定制化更新后，需要重启 Merge 更新，Merge 流程内容过多，将在新的一节描述。
8. 失败回滚机制：重启时，如果更新失败（机器重启后无法正常开机），此时会回滚到上一版本。

3.2.2 UpdateEngine 升级流程

升级代码包含在了：android/system/update_engine/目录下

通过 main.cc 进入升级流程, 升级主要分为：

1. 升级包下载。
2. 初始化。
3. 更新。

3.2.2.1 初始化

初始化阶段包含了。

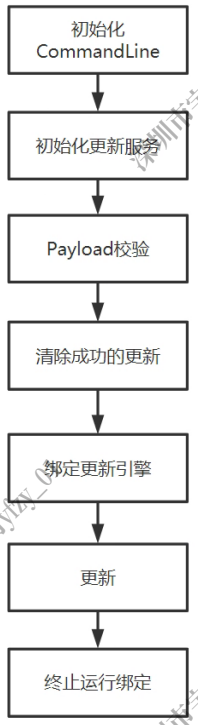

图 3-2: Update Init

1. 初始化阶段内容在：update_engine_client_android.cc。
2. 初始化 CommandLine。
3. 获取 UpdateEngineService 服务，并检测服务状态。
4. 校验 Payload，检查头部内容，然后分配对应空闲空间。
5. 清除成功的更新，主要清除上一次成功升级的残留文件或状态。
6. 绑定更新引擎 daemon。
7. 应用更新。
8. binder 终止运行通知绑定更新引擎 daemon。

下面将主要描述应用更新阶段的内容，通过更新服务最终调用到 update_attempter_android.cc 的 ApplyPayload。

3.2.2.2 更新

如下图。

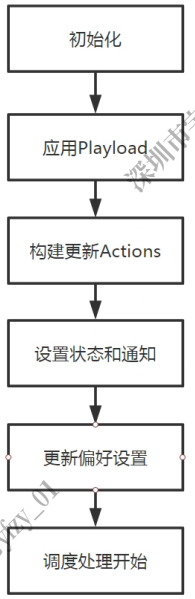

图 3-3: Update Main

更新阶段包含了：

1. 更新前准备。
2. 构建更新 Actions。
3. 设置更新状态和通知。
4. 更新偏好。
5. 执行更新 Actions。
6. 更新 Merge。

下面将对前五个部分进行展开讲述，下一节再详细描述 Merge。

一、更新前准备

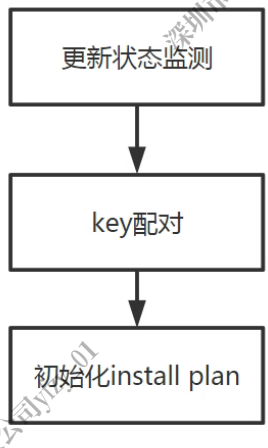

图 3-4: Apply Payload

1. 更新状态监测。
2. key 头部校验。
3. 初始化 Install Plan。

```
参数解释
install_plan_:
download_url 下载路径
version 版本
payload：{size, metadata_size, type}
public_key_rsa 公钥
hash_checks_mandatory 强制hash检查
is_resume payload 简介信息
source_slot 原slot
target_slot 目标slot
powerwash_required 强力清洗
switch_slot_on_reboot 重启交换slot
run_post_install 安装后运行
write_verity 写入验证
```

install_plan 包含了源和目标 slot、payload 信息、校验公钥、交换 slot、执行安装后流程等等内容。

初始化 InstallPlan 后，会初始化 fetcher，一个 HttpFetcher 对象，包含了更新包的信息、下载方式等等，主要用于 Download 更新包阶段。然后开始构建更新的 Actions。

二、构建更新 Actions

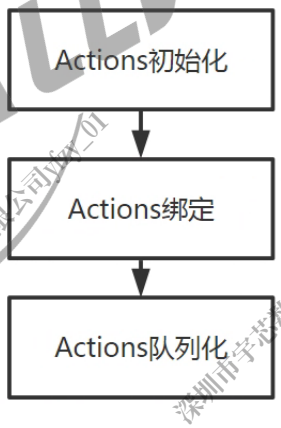

图 3-5: Build Actions

构建 Actions 主要是将 update_boot_flags_action，cleanup_previous_update_action，install_plan_action，download_action，filesystem_verifier_action，postinstall_runner_action 六个 Actions 推入 processor 的 Actions 队列中。

存储使用的是一个双端队列，定义信息在 android/system/core/fs_mgr/libsnapshot/include/libsnapshot/snapshot.h。

三、设置更新状态和通知

主要流程为：

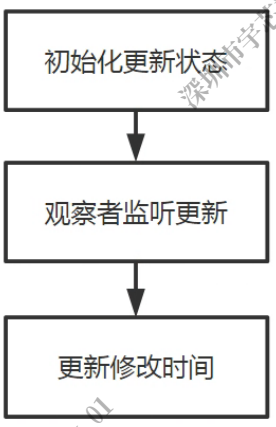

图 3-6: Set Status And Notify

更新状态分为：
None 更新结束
Initiated 初始化
Unverified 未验证
Merging 合并更新
MergeNeedsReboot 请求reboot合并更新
MergeCompleted 合并完成
MergeFailed 合并失败
Cancelled 取消

四、更新偏好

主要流程为：

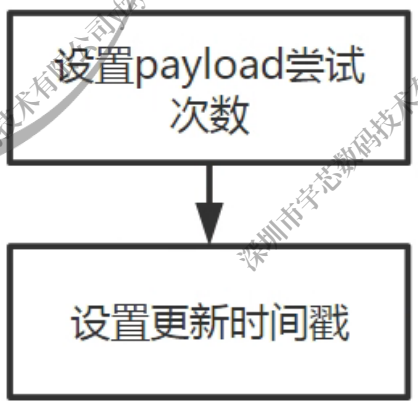

图 3-7: Update Prefs

更新偏好的主要工作就是通过 metrics_utils 工具类记录信息。

五、更新 Actions

主要流程为。

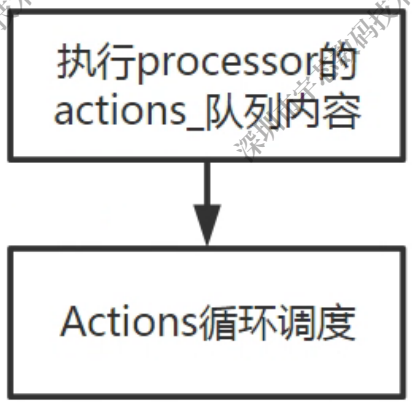

图 3-8: Actions Done

执行更新主要工作就是，将之前构建的 Actions 一个一个执行并 push, 会执行每一个 Actions 的 PerformAction()。

更新流程主要包含以下六个 Actions。

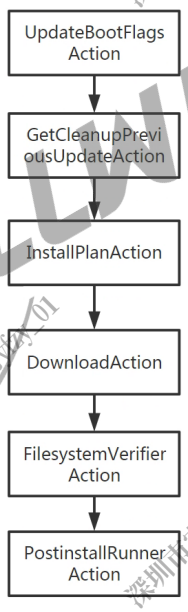

图 3-9: Actions

下面将详细讲述, 下面所提及到的文件都以 update_engine 目录为基准。

1. update_boot_flags_action

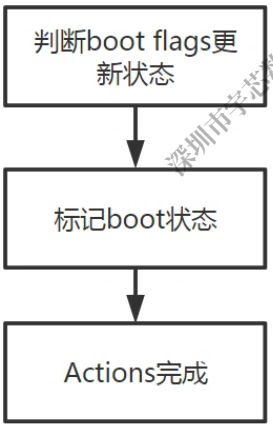

图 3-10: Update Boot Flags Action

主要代码包含在了 update_boot_flags_action.cc, 作用是：标记 boot 更新开始。

2. cleanup_previous_update_action

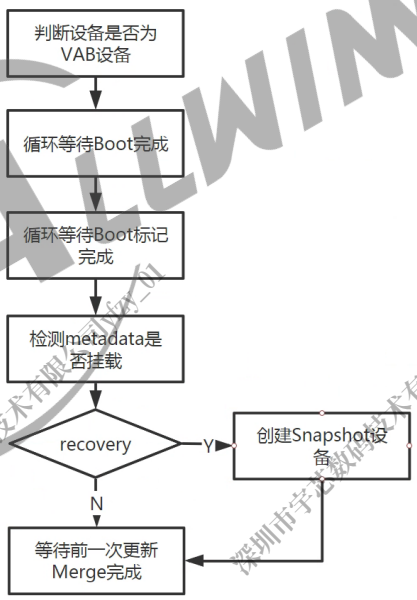

图 3-11: Cleanup Previous Update Action

主要代码包含在了 cleanup_previous_update_action.cc，作用是：清除前一次的更新。

进入该流程首先会等待当前系统启动完成（sys.boot_completed 属性标记为 true），然后检测 Boot 是否被标记为启动成功。直到标记成功后，就会检测 metadata 分区的挂载。判断当前是否处于 recovery，如果处于，则创建 SnapShot 设备，最后等待前一次更新合并成功；不处于则直接等待前一次更新合并成功。合并结果循环获取 UpdateState，并解析当前处于什么阶段，合并成功后，则会返回 kSuccess。

3. install_plan_action

主要代码包含在了 payload_consumer/install_plan.h，主要作用是传递 install_plan_ 参数。

4. download_action

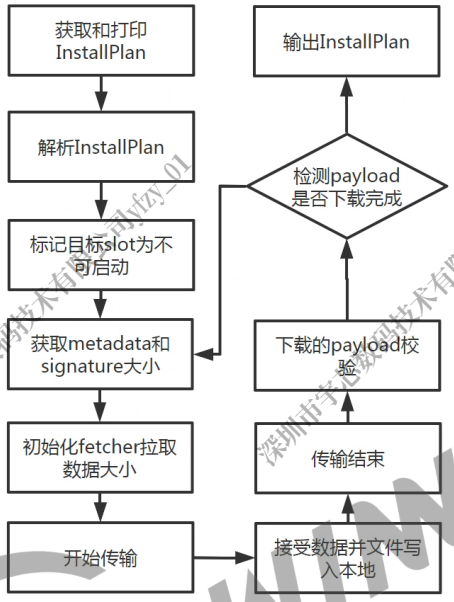

图 3-12: Download Action

主要代码包含在了 download_action.cc 和 payload_consumer/delta_performer.cc，作用是：下载更新包并校验。

进入该流程首先会获取 InstallPlanActions 传递的 install_plan_，然后对其进行解析，获取更新包的一些信息，下载路径之类的。在下载时会将目标 slot 标记为不可启动，然后获取 metedata 和 signature 的大小，再去拿 payload 的大小，再初始化 fetcher，以及计算要拉取升级包的大小。开始传输时，会初始化 curl，然后判断使用 HTTP/HTTPS/file 等方式拉取更新包。拉取时，通过 Write 去写文件（最后会调用到 delta_performer.cc），在传输结束后，会对 payload 进行校验，并检查是否下载完成（有时升级会连续两次及以上，payload 的数量就会大于 1）。下载完成就会输出 install_plan_，反之就会继续下载。

5. filesystem_verifier_action

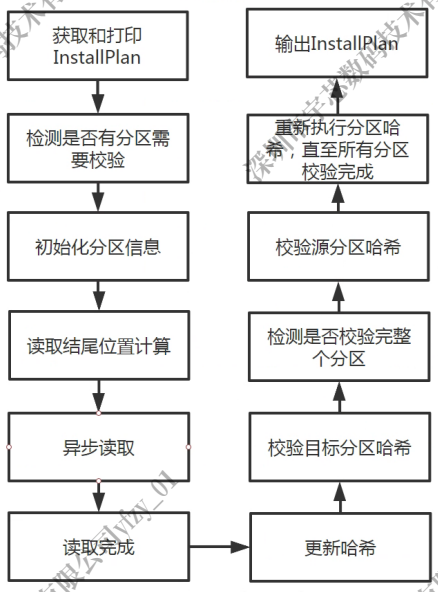

图 3-13: File system Verifier Action

主要代码包含在了 payload_consumer/filesystem_verifier_action.cc，作用是：校验所有分区。

进入该流程首先会获取 InstallPlanActions 传递的 install_plan_，然后解析和检测是否有分区需要校验：没有，则直接返回流程成功结束；有，进入分区校验流程。初始化分区信息，检测当前分区索引，开始计算当前分区结尾的位置。根据分区位置读取分区，读取完成之后会更新对应哈希，再进行目标分区哈希校验，校验完整个分区后，校验源分区哈希。直至校验完所有分区后，返回成功, 输出 install_plan_。

6. postinstall_runner_action

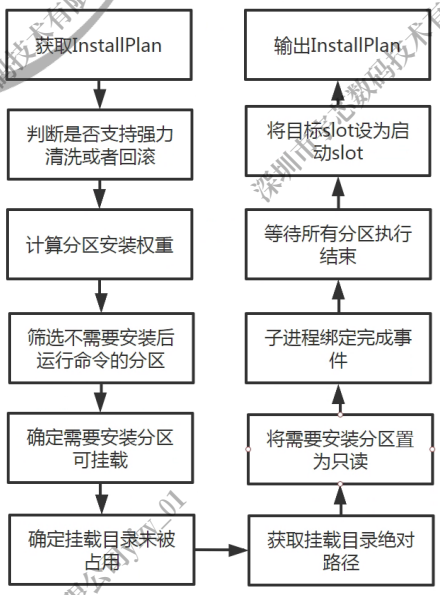

图 3-14: Post install Runner Action

主要代码包含在了 payload_consumer/filesystem_verifier_action.cc，作用是：校验分区。

进入该流程首先会获取 InstallPlanActions 传递的 install_plan_，然后解析是否支持强力清洗和回滚，因为回滚的时候总是会强力清洗的。根据分区的 run_postinstall 值开始计算分区的权重，依此可以获取运行时间等等值。然后筛选需安装后运行命令的分区，并确定它们是可挂载的，且未被占用。逐个分区遍历，获取挂载的绝对路径，将其置为只读。然后调用子进程异步执行，并绑定完成事件。待所有分区执行结束后，将升级的目标 slot 置为下次启动 slot，最后输出对应的 InstallPlan。

最后更新 Actions 执行完毕，准备重启。

3.2.3 Merge 流程

合并流程主要描述重启后，Super 分区如何合并更新后的内容。

1. 升级前的 System

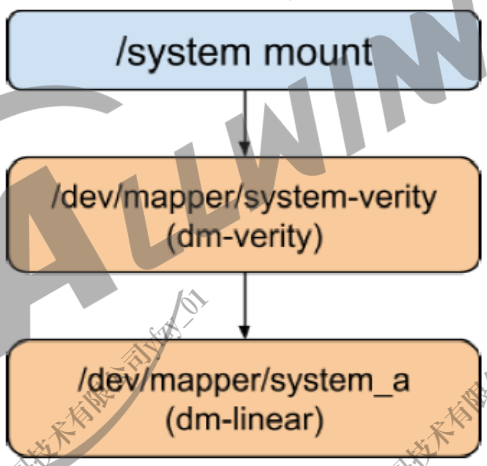

图 3-15: System Partition PreUpdate

2. 升级时

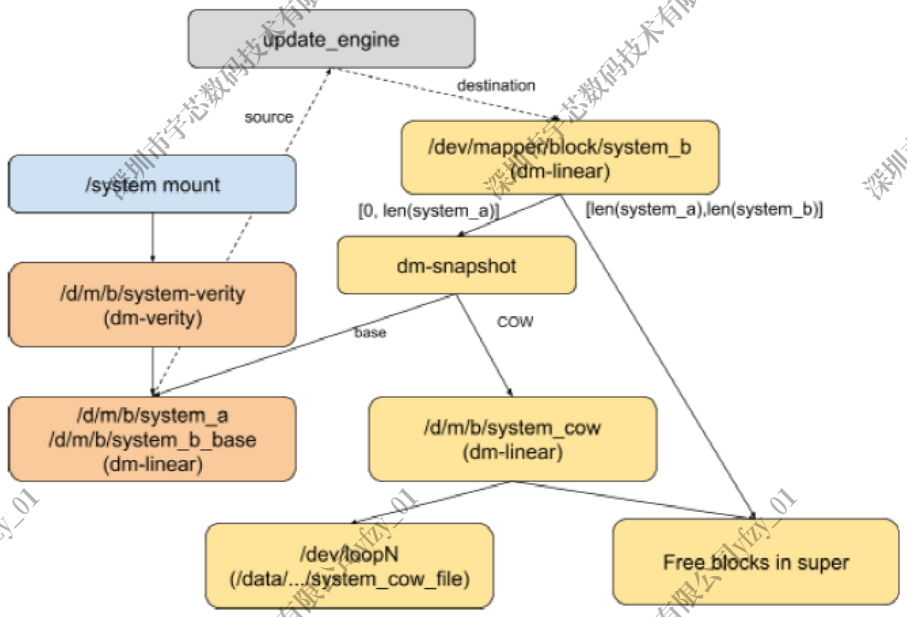

图 3-16: System Partition Updating

当 update_engine 开始写一个新的 system 分区的时候，它会创建以下设备：

（1）基于 system_a 设备创建 system_b_base 设备（dm-5 设备）。

（2）在/data/gsi/ota/目录下创建 system_b-cow-img.img.0000 文件，然后通过/dev/block/loop0 设备进行挂载。然后创建 dm-6 设备将其映射到 system_cow 上。

（3）创建 dm-7（dm-snapshot）设备，将其映射为 system_b。

（4）打开 dm-snapshot 设备，将 OTA 升级包中的 system.img 写进去，此时写的内容，就写到了/data/gsi/ota/system_b-cow-img.img.0000 文件中。

（5）升级成功之后，调用 boot_ctrl 中的 SetActiveBootSlot 函数：将 _a（目前 slot）的 priority 值设小（如果最大就减 1）。然后将 _b（要升级的 slot）的 priority 值设为 15（最大），以及 tries_remaining 的值设为 6（升级失败，最多可以重启 6 次）。

```
（1）在升级过程中创建设备：
[INFO:dynamic_partition_control_android.cc(319)] Loaded metadata from slot B in /dev/block/by-name/super
 update_engine: Successfully unmapped snapshot system_b
vold   : Disk at 253:5 changed
 update_engine: [libfs_mgr]Created logical partition system_b-base on device /dev/block/dm-5
vold   : Disk at 7:1 changed
gsid   : Created loop device /dev/block/loop1 for file /data/gsi/ota/system_b-cow-img.img.0000
 update_engine: Mapped system_b-cow-img to /dev/block/loop1
update_engine: Calling GetMappedImageDevice with local image manager; device /dev/block/loop1may not be
       available in first stage init!
vold   : Disk at 253:6 changed
update_engine: Mapped COW device for system_b at /dev/block/dm-6
vold   : Disk at 253:7 changed
update_engine: Mapped system_b as snapshot device at /dev/block/dm-7
[INFO:dynamic_partition_control_android.cc(173)] Succesfully mapped system_b to device mapper (force_writable = 1);
       device path at /dev/block/dm-7
（2）升级成功之后，调用boot_ctrl中的SetActiveBootSlot函数：
[INFO:postinstall_runner_action.cc(376)] All post-install commands succeeded
```

3. OTA 升级完成后，重启，uboot 端 slot 切换流程。（Google uboot 实现范例）

（1）遍历所有 slot，如果该 slot 的 tries_remaining 为 0 则跳过（不从该 slot 启动，当 _b 系统无法启动，则在这里切换回 _a 系统），然后选择 priority 最高的 slot 进行启动（升级后, ab 中 _b 最高）。（2）如果当前 slot 的 successful_boot 为 0（OTA 升级重启成功后，会将此值设为 1），则 tries_remaining 的值减 1。（3）将 slot_suffix 的值改为 _b。

4. OTA 升级完成后，重启过程中。

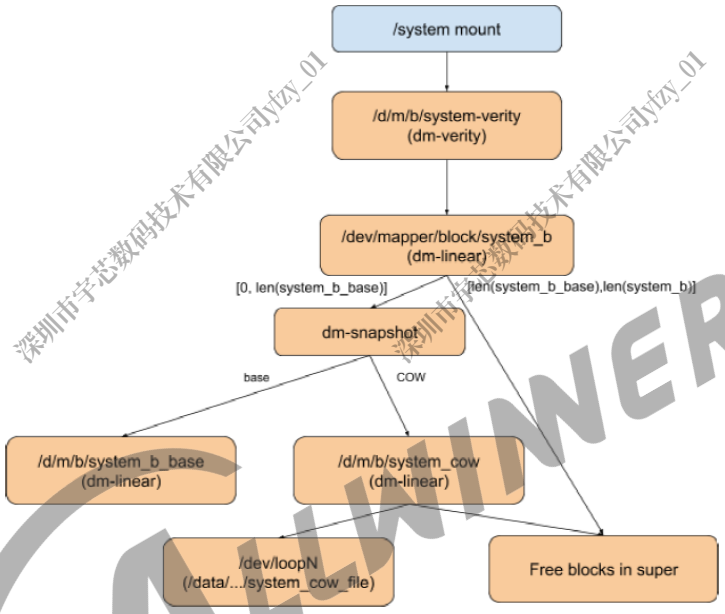

图 3-17: System Partition Updated

（1）创建 dm-0 设备，作为 system_b_base 设备（映射 super 分区中的 system 分区）。
（2）创建 dm-1 设备，作为 system_b-cow-img 设备（映射 data 分区中的 system_b-cow-img.img.0000 文件）。
（3）创建 dm-2 设备，映射为 system_cow 设备。
（4）创建 dm-3 设备，映射为 dm-snapshot 设备（system_b 分区）。
（5）将 system 分区挂载在 dm-3 上（dm-snapshot 设备）。
（6）然后系统启动读 system 分区的时候，如果读到的文件，在 system_cow 中记录到有改变，则直接读 system_cow 中的文件。

```
在first stage init过程中创建dm设备：
init: Creating logical partitions with snapshots as needed
random: init: uninitialized urandom read (16 bytes read)
init: [libfs_mgr]Created logical partition system_b-base on device /dev/block/dm-0
random: init: uninitialized urandom read (16 bytes read)
init: [libfs_mgr]Created logical partition system_b-cow-img on device /dev/block/dm-1
init: Mapped system_b-cow-img to 253:1
init: Mapped COW device for system_b at /dev/block/dm-2
init: Mapped system_b as snapshot device at /dev/block/dm-3
```

5. 重启完成后，开始 snapshot-merge。

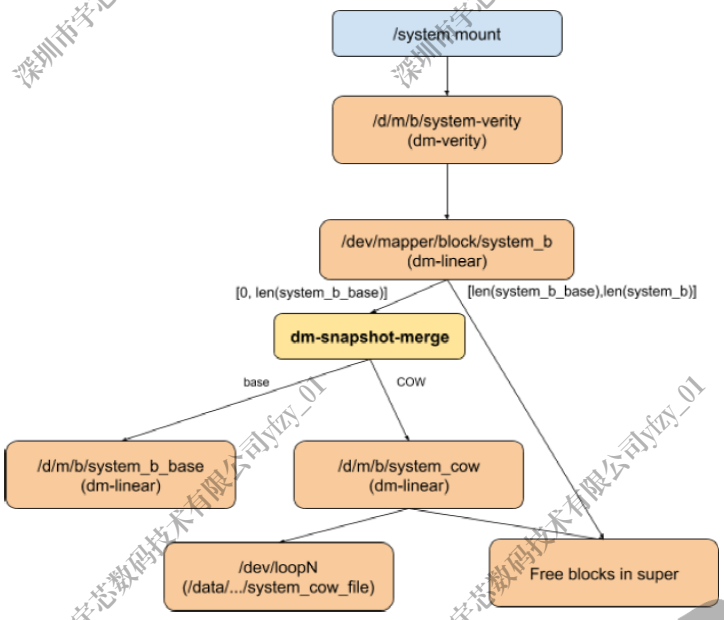

图 3-18: Snapshot Merge

（1）重启成功后，调用 BootControl 的 MarkBootSuccessful 函数，将 successful_boot 和 tries_remaining 设为 1。

（2）将 dm-snapshot 设备设为 merge 状态，将 cow 设备和 base 设备的内容进行合并。

（3）umaped 相关的设备。

（4）合并成功之后，删除/data/gsi/ota 目录下的 img 文件以及/metadata/ota 目录下的相关文件。

```
（1）启动完成之后，调用BootControl的MarkBootSuccessful函数。
[INFO:cleanup_previous_update_action.cc(137)] Boot completed, waiting on markBootSuccessful()
（2）进行合并，大概30秒左右。
17:34:51.739365 [INFO:cleanup_previous_update_action.cc(338)] Attempting to initiate merge.
设置dm-snapshot为merge：
update_engine: Successfully switched snapshot device to a merge target: system_b
[INFO:cleanup_previous_update_action.cc(302)] Waiting for merge to complete: 8%.
[INFO:cleanup_previous_update_action.cc(302)] Waiting for merge to complete: 20%.
[INFO:cleanup_previous_update_action.cc(302)] Waiting for merge to complete: 27%.
[INFO:cleanup_previous_update_action.cc(302)] Waiting for merge to complete: 33%.
[INFO:cleanup_previous_update_action.cc(302)] Waiting for merge to complete: 40%.
[INFO:cleanup_previous_update_action.cc(302)] Waiting for merge to complete: 48%.
[INFO:cleanup_previous_update_action.cc(302)] Waiting for merge to complete: 55%.
[INFO:cleanup_previous_update_action.cc(302)] Waiting for merge to complete: 63%.
[INFO:cleanup_previous_update_action.cc(302)] Waiting for merge to complete: 70%.
[INFO:cleanup_previous_update_action.cc(302)] Waiting for merge to complete: 78%.
[INFO:cleanup_previous_update_action.cc(302)] Waiting for merge to complete: 85%.
[INFO:cleanup_previous_update_action.cc(302)] Waiting for merge to complete: 93%.
（3）unmap设备，然后清除data目录下的img文件。
update_engine: Removing all update state.
（4）merge完成。
17:35:21.018875 [INFO:cleanup_previous_update_action.cc(262)] Merge finished with state MergeCompleted.
```

6. Snapshot-merge 完成后

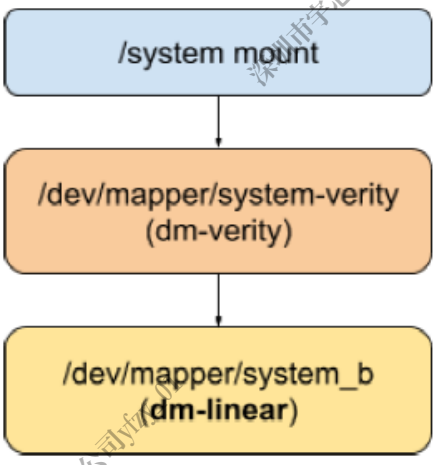

图 3-19: Snapshot Merged

4 OTA 模块使用说明

4.1 OTA 的升级范围

原生 Android 提供的 update_engine 升级程序支持更新 product 分区、system 分区、vendor 分区、boot 分区、vendor_boot 分区、dtbo 分区、vbmeta 分区、vbmeta_vendor 分区、vbmeta_system 分区、vendor_dlkm 分区、system_dlkm 分区、init_boot 分区。除此之外，我们根据产品特点，给 update_engine 扩展了一些专有功能，以满足 BSP 的更新需要。

分区类型	是否支持	是否原生升级内容
boot 分区更新	√	√
init_boot 分区更新	√	√
system 分区更新	√	√
vendor 分区更新	√	√
system_dlkm 分区更新	√	√
vendor_dlkm 分区更新	√	√
product 分区更新	√	√
vendor_boot 分区更新	√	√
dtbo 分区更新	√	√
vbmeta 分区更新	√	√
vbmeta_vendor 分区更新	√	√
vbmeta_system 分区更新	√	√
env 分区更新	√	×
bootloader 分区更新	√	×
boot0/uboot 升级	√	×
sys_config.fex 更新	√	×
sys_partition.fex 更新	×	×

值得注意的是，BSP 中的关于模块的大部分配置都集中在 board.dts 中，如果需要更新 board.dts 的配置，需要更新 vendor_boot 分区。如果希望自己实现制作 ota 包的脚本，可以参考以下目录的脚本文件：android/device/softwinner/common/vendorsetup.sh。

4.2 制作 OTA 包步骤

使用 OTA 前首先需要区分三个包：

- TargetFile：包含制作时当前编译版本的 system 分区，boot 分区，recovery 分区等内容，可用于制作 OTA 完整包和差分包。
- OTA 完整包：包含本次升级版本的所有内容，可以从之前各个版本直接升级到当前的版本。制作完整包需要当前版本的 TargetFile。
- OTA 差分包：包含本次升级版本和之前特定一个版本的升级内容，只适用于之前特定一个版本升级到当前版本。制作差分包需要之前特定版本的 TargetFile 和当前版本的 TargetFile。

4.2.1 制作 OTA 完整包

4.2.1.1 制作 OTA 完整包命令

1. Android 15 及以后 GRF 版本打包过程。

这里以 15 + 13 的 grf sdk 为例，如果是 15 + 14 的 grf sdk，则将 android13_sdk_path 改为 android14_sdk_path 即可。

```
$ source build/envsetup.sh
$ lunch
$ make -j8
$ make --dist [-d] [-v] -m ../android13_sdk_path/
```

如果需要对固件进行签名，把相关签名文件放入 android/vendor/security 目录，流程不变。对于安全固件，需要加上-v 参数启用安全系统校验。

```
$make --dist -m ../android13_sdk_path/ 非安全
$make --dist -d -m ../android13_sdk_path/ 卡打印升级包
$make --dist -v -m ../android13_sdk_path/  安全
$make --dist -d -v -m ../android13_sdk_path/ 安全卡打印升级包
```

使用上述打包过程 make –dist -m ../android13_sdk_path/后会自动生成目标文件包（target-files-package）路径为：$OUT/obj/PACKAGING/target_files_intermediates/$TARGET_PRODUCT-target_files.zip。 若包含签名目标文件包，则路径为：$OUT/signed_target_files.zip 注：生成的 target_files.zip 文件需要与固件一同保存，用于后续生成 OTA 包。

2. 正常非 GRF 版本打包过程。

```
$ source build/envsetup.sh
$ lunch
$ make -j8
$ pack4dist[-d][-v]
```

如果需要对固件进行签名，把相关签名文件放入 android/vendor/security 目录，流程不变。对于安全固件，需要加上-v 参数启用安全系统校验。

```
$pack4dist 非安全
$pack4dist -d 卡打印升级包
$pack4dist  -v 安全
$pack4dist  -d -v 安全卡打印升级包
```

使用上述打包过程 pack4dist 后会自动生成目标文件包（target-files-package）路径为：$OUT/obj/PACKAGING/target_files_intermediates/$TARGET_PRODUCT-target_files.zip。 若包含签名目标文件包，则路径为：$OUT/signed_target_files.zip 注：生成的 target_files.zip 文件需要与固件一同保存，用于后续生成 OTA 包。

4.2.1.2 OTA 命令执行过程

1. Android 15 及以后版本 GRF 打包过程。

这里以 15 + 13 的 grf sdk 为例，如果是 15 + 14 的 grf sdk，则将 android13_sdk_path 改为 android14_sdk_path 即可。

```
make --dist [-d] [-v] -m ../android13_sdk_path/ 命令比pack4dist[-d][-v]命令在生成OTA包前多了三个步骤。

1. 生成Android 13的targetfile。

2. 生成Android 15的targetfile。

3. 使用merge_target_files命令，合并Android 13和Android 15的targetfile。

生成targetfile之后，剩下的步骤都和下面的pack4dist命令的步骤一致。
```

2. Android 13 及以前版本 OTA 命令执行过程。

pack4dist 后会自动生成目标文件包是因为封装了如下命令：

1.TargetFile 签名

制作带签名的 OTA 升级包的流程如下。

```
$ sign_target_files_apks -d [key_path] [unsigned_target_file.zip] [signed_target_file.zip]
```

[key_path] 为存放 key 文件夹的路径，（如果没有签名文件，则默认生成不签名的 ota 包）需要包括 4 个 key 分别是 media，platform，releasekey，shared，具体包含以下文件：media.pem，media.x509.pem，platform.pk8，releasekey.pem，releasekey.x509.pem，shared.pk8，media.pk8，platform.pem，platform.x509.pem，releasekey.pk8，shared.pem，shared.x509.pem [unsigned_target_file.zip] 表示上一步生成的没有签名的 TargetFile, [signed_target_file.zip] 表示命令输出得到的经过签名的 TargetFile,

2. 从签名过的 TargetFile 得到镜像（boot.img,system.img 和 recovery.img)

```
$ img_from_target_files [signed_target_file.zip]  [img.zip]
```

[signed_target_file.zip] 表示经过签名的 TargetFile, [img.zip] 表示命令输出得到的镜像压缩包。

3. 解压 img.zip，得到的 boot.img,system.img 和 recovery.img 复制到 out/target/product/[device]/下面，重新 pack 得到可烧录的固件，是签名过的固件。

4. 生成 ota 包完整包

```
$ ota_from_target_files [target_file.zip] [ota_full.zip]
```

[target_file.zip] 表示最终的 TargetFile

[ota_full.zip] 表示命令输出得到的 OTA 完整包

4.2.2 制作 OTA 差分包

1. Android 15 及以后 GRF 版本打包过程。

这里以 15 + 13 的 grf sdk 为例，如果是 15 + 14 的 grf sdk，则将 android13_sdk_path 改为 android14_sdk_path 即可。

```
关键步骤是把上一个版本的target_file重命名成old_target_files.zip，放到android根目录。

1. 先source和lunch，然后make installclean && make --dist -d -v -m ../android13_sdk_path/ 。
2. 保存固件和15的target_file和13的target_file。
3. 修改内容。
4. 把target_file重命名成old_target_files.zip，放到android根目录。
5. make installclean && && make --dist -d -v -m ../android13_sdk_path/。
6. 生成full-ota包和inc-ota，完整包和差分包。
```

2. 正常非 GRF 版本打包过程

```
关键步骤是把上一个版本的target_file重命名成old_target_files.zip，放到android根目录。

1. 先source和lunch，然后make installclean && pack4dist [-d] [-v]。
2. 保存固件和target_file。
3. 修改内容。
4. 把target_file重命名成old_target_files.zip，放到android根目录。
5. make installclean && pack4dist [-d] [-v]。
6. 生成full-ota包和inc-ota，完整包和差分包。
```

注意事项:

1. 该差分包仅对指定的前一版本固件有效。
2. 制作一个完整包，也会生成当前版本的一个 target-file 文件包。

5 使用 OTA 包升级

5.1 从应用升级

5.1.1 选择应用

将 ota 包放到外部存储或者内部存储中在桌面通过以下方式找到升级入口，选择 ota 包进行升级。

1. 系统语言为英文时:Settings–>System–>Local Update。

2. 系统语言为中文时: 设置–> 系统–> 本地升级。

5.2 update_engine_client 命令升级，适用于 debug 模式下

bin 文件升级先解压 OTA 升级包, 得到目录描述内容：

```
参数解释：
payload为升级文件的位置，也可以使用http的方式
size为FILE_SIZE
headers：内容为payload_properties.txt，每个升级包内都是不一样的
offset：值等于metadata内payload.bin的偏移，如payload.bin:1351:7964794的offset为1351

通过adb输入以下命令：
update_engine_client \
--update \
--follow \
--payload=file:///sdcard/payload.bin \
--size=624814625 \
--headers="
FILE_HASH=65bxQVtEZNdJpfcz7DCyQZDT4UWZgYaR1m4adSswX6g=
FILE_SIZE=624814625
METADATA_HASH=p9G5IbC4ZbCUxBe+nyRPfxHnkwEV/6irObVLFclPQ24=
METADATA_SIZE=43017
"

OTA升级包升级
update_engine_client \
--update \
--follow \
--payload=file:///sdcard/update.zip \
--offset=6788 \
--size=315116676 \
--headers="FILE_HASH=q8A90+LeJsBEpvoE2yGimQ/CkFfCyq48y60H1Kc/
```

```
FILE_SIZE=315116676
METADATA_HASH=6h4l0R5bhAKdMtlAUeKGVI+ygsPlJn78kji6Rzg/g4Q=
METADATA_SIZE=50062
"
```

5.3 Recovery 模式下的升级 (不支持差分包)

5.3.1 Apply update from ADB

1.将固件放在PC端，如：E:/update.zip。
2.进入Recovery。
3.选择Apply update from ADB。
4.打开cmd，并输入adb sideload E:/update.zip。
5.等待打印Install from ADB complete.升级完成。
6.选择reboot system now重启并进入android。

5.3.2 Apply update from TFcard or USB

1.将固件放入TF卡或U盘中。
2.进入Recovery。
3.插入TF卡或U盘。
4.在Recovery菜单中选择Apply update from TFcard or USB。
5.找到升级包的路径并选择开始升级。
6.等待打印Install from SD card complete.升级完成。
7.选择reboot system now重启并进入android。

6 OTA 压缩功能配置

在 device/softwinner/ceres / common/storage/config.mk，更改 virtual_ab 配置。

```
$(call inherit-product, $(SRC_TARGET_DIR)/product/virtual_ab_ota/compression_with_xor.mk)
```

该配置支持 OTA 升级使用 io_uring、userspace、xor compression。

6.1 io_uring

io_uring 是 Linux 5.1 中引入的一套新的 syscall 接口，用于支持异步 IO，是实现与内核间高效率交互的一种模式。Android13 支持在 OTA 升级时使用 io_uring。

```
#build/make/target/product/virtual_ab_ota
ro.virtual_ab.io_uring.enabled = true
```

可能通过设置属性打开或关闭该功能。

6.2 userspace snapshot

Android 13 及更高版本的设备，虚拟 A/B 中快照的合并过程支持由 snapuserd 用户空间组件执行。通过属性控制该功能开关。

```
ro.virtual_ab.userspace.snapshots.enabled = true
```

6.3 xor compression

通过属性控制 xor compression 功能开关。

```
ro.virtual_ab.compression.xor.enabled = true
```

在旧块和新块之间存储 XOR 压缩字节。如果虚拟 A/B 更新只更改了某个块中的一些字节，XOR 压缩存储方案所用的空间会少于默认存储方案，因为快照不会存储所有 4K 字节。这样可以缩减快照大小，因为 XOR 数据包含许多零，并且比原始块数据更易于压缩。

该功能还需在升级包支持，需要在打包中加入命令–enable_vabc_xor true。

```
ota_from_target_files --enable_vabc_xor true final_target_files full_ota
```

7 静默升级

7.1 简介

静默升级是指，通过 GOTA 应用和 GOTA dashboard（谷歌 OTA 信息中心）https://partner.android.com/ota，实现熄屏静默升级。在谷歌 OTA 信息中心部署好 OTA 差分包后, 设备会通过 GOTA 应用自动下载安装包安装。OTA 包安装完成后，会以静默的方式 reboot，reboot 一般会在凌晨 2：00 进行，为了不影响用户正常休息，此过程不会播放开机音乐，开机动画也会变成预置的 dark 主题的动画。

7.2 静默升级要求

- 开机动画要求为 dark 主题，避免刺眼的颜色。
- 开机音乐要关闭，避免打扰用户。

7.3 自定义开机动画预置

动画预置到下面路径，

```
device/softwinner/ceres/common/media/bootanimation/
```

文件格式为 zip，且不能压缩。

其中，bootanimation.zip 为正常开机动画，bootanimation_dark.zip 为静默升级开机动画。

7.4 原理

- GOTA 安装完升级包后，会在合适时间 reboot；当此次升级为静默升级，reboot reason 将会包含 “unattended”。
- 通过判断 reboot reason，如果是静默升级，设置属性 persist.sys.unattended_update 为 true。
- BootAnimation 检测到 persist.sys.unattended_update 为 true 则会更换开机动画并且关闭开机音乐。
- OTA merge 完成后，会重新设置 persist.sys.unattended_update 为 false。

8 FAQ

8.1 升级注意事项

8.1.1 OTA 不能改变分区数目及其大小

Recovery 只是一个运行在 Linux 上的一个普通应用程序，它并没有能力对现有分区表进行调整，所以第一次量产时就要将分区的数目和大小确定清楚，杜绝后续升级调整分区数目及其大小的想法，OTA 不能改变分区数目和分区的大小。

8.1.2 misc 分区需要有足够的权限被读写

misc 分区是 Recovery 与 Android 之间的桥梁，如果 misc 分区的读写权限过高，会导致上层应用无法对其写入数据，则会令 Recovery 功能异常。检验此功能是否存在问题时，请确保 misc 分区的设备节点/dev/block/xxx 和其软链接/dev/block/by-name/misc 有足够的权限被读写。

```
root@android:/dev/block/by-name# ls -l
lrwxrwxrwx root     root   2020-11-10 07:16 misc -> /dev/block/mmcblk0p10(misc分区软链接)
...
root@android:/dev/block # ls -l
brw-rw---- 1 system root 179,   10 2020-11-10 10:15 /dev/block/mmcblk0p10
.....
```

8.1.3 Recovery 升级

注意事项:

```
recovery模式下：

1. 因无权限读写data分区，所以没办法创建snapshot，既没办法进行差分包升级。
2. 如果super分区size小于镜像两倍，完整包升级会擦除原有的super分区。
3. 如果super分区size大于镜像两倍，完整包升级不会擦除原有的super分区。
```

8.2 制作 OTA 包常见问题和注意事项

1. TargetFile 和固件是否匹配的区分方法在 Android 设备执行 adb pull /system/build.prop 会得到这个固件的 build.prop 文件对于 TargetFile，解压出来查看 SYSTEM/build.prop，对比这两

个 build.prop 如果一致，表示这个固件和 TargetFile 是匹配的。

2. metadata 匹配失败必须获取对应的 payload_properties.txt，每个升级包内都是不一样的。
3. 差分包版本匹配失败查看更新包内：META->INF->com->anroid->metadata ，描述了从什么版本升级到什么版本。

著作权声明

本文档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本文档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本文档内容的部分或全部，且不得以任何形式传播。

商标声明





（不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。