Tina Linux OTA
开发指南

版本号: 1.2

发布日期: 2025.08.19

版本历史

版本号	日期	制/修订人	内容描述
1.0	2024.12.26	AWA1615	初始版本
1.1	2025.04.11	AWA1615	完善配置说明
1.2	2025.08.19	AWA1615	添加专业术语

目　录

1 概述

OTA 是 Over The Air 的简称，顾名思义就是通过无线网络从服务器上下载更新文件对本地系统或文件进行升级，便于客户为其用户及时更新系统和应用以提供更好的产品服务，这对于客户和消费者都极其重要。

编写目的

本文主要服务于使用 Tina 软件平台的广大客户，以及帮助客户使用 Tina 平台的 OTA 升级系统并做二次开发。

1.2 适用范围

Allwinner 软件平台 Tina5.0。

1.3 相关人员

适用 Tina 平台的广大客户和关心 OTA 的相关人员。

1.4 文档约定

本文升级针对分区，而不是文件系统里某个具体的文件。

1.5 相关术语介绍

- 主系统：由./build.sh 或者 make 整体编译得到的内核镜像和 rootfs 镜像组成的系统
- recovery 系统：专指 swupdate_make_recovery_img 命令编译出来的 recovery 镜像

OTA 方案

1.6.1 recovery 系统方案

recovery 系统方案，是在主系统之外，增加一个 recovery 系统。升级时，主系统负责升级 recovery 系统，recovery 系统负责升级主系统。

这样如果升级中途发生掉电，也不会影响当前正在使用的这个系统。重启后仍可正常进入系统，继续完成升级。

一般 recovery 系统会使用 initramfs 功能，并大量裁剪不必要的应用，只保留 OTA 必需的功能，尽量减小。

recovery 系统方案优点：

1. recovery 系统可以做得比较小，对 AB 系统节省 flash 空间约 40% 或更高。

recovery 系统方案缺点：

1. recovery 系统一般不包含主应用，所以 OTA 期间，处于 recovery 系统中时，无法为用户正常提供服务。
2. 需要重启两次。

要维护两份系统配置，和系统和

1.6.2 AB 系统方案

AB 系统方案，是将原有的系统，增加一份。即 flash 上总共有 AB 两套系统。两套系统互相升级。OTA 时，若当前运行的是 A 系统，则升级 B 系统，升级完成后，设置标志，重启切换到 B 系统。OTA 时，若当前运行的是 B 系统，则升级 A 系统，升级完成后，设置标志，重启切换到 A 系统。

AB 系统方案优点：

新过程是在完整系统中进行的，更新期间可正常提供服务，用户无感知。最终做一次重启即可。

2. 逻辑简单，只重启一次。
3. 只维护一套系统配置。

AB 系统方案缺点：

1. flash 占用较大，增加一倍的内核与 rootfs 分区大小。

2 快速入门

在默认的 SDK，都使用 swupdate 去 OTA；会提供一个配置好 swupdate 功能的方案。

广大用户可以先基于此方案去使用。

OTA 的快捷命令封装于 build/buildbase.sh，执行以下命令获取：

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

说明

1、当前 OTA 基于分区升级，需要方案定好分区数量和大小，量产后无法调整设备分区。
2、swupdate 未支持 4G 以上的 OTA 包，所以一些客户自身的资源应单独存放到一个新分区；然后通过其他方法升级，比如 dd 命令。

⚠ 注意

全文多次提及 OTA 配置文件 sw-description 和 sw-subimgs.cfg 的目录，此处做汇总。

```
openwrt:
target/{IC}/openwrt/{IC}-{BOARD}/swupdate  #第一优先级
wrt/target/{IC}/{BOARD}/swupdate  #第二优先级。后文以此路径举例

buildroot:
target/{IC}/buildroot/{BOARD}/swupdate/  #第一优先级。后文以此路径举例
device/config/chips/{IC}/configs/{BOARD}/buildroot/swupdate  #第二优先级。
```

⚠ 注意

对于开启了 uboot 安全 env 功能即 uboot defconfig 开启了 CONFIG_SUNXI_BOOT_ENV_INLINE=y 的方案；env.cfg 的中的 bootcmd, boot_normal 已被屏蔽，修改不生效。需要在 uboot defconfig 文修改 CONFIG_SUNXI_BOOT_ENV_STRING 里面对应的变量，默认提供的方案都已修改好。下文依旧以 env.cfg 举例说明。

2.1 recovery 系统整包升级

2.1.1 配置

分区表指 sys_partition.fex，nor 介质对应 sys_partition_nor.fex

在分区表添加 recovery 分区，size 根据实际的 recovery.img 调节。

```
[partition]
    name        = recovery
    size        = 65536
    downloadfile = "recovery.fex"
    user_type    = 0x8000
```

env 默认是配置好的，查看是否有以下内容，有则跳过 env 配置；没有则添加，然后配置 boot_normal 命令，从 $boot_partition 变量指定的分区加载系统。

```
boot_partition=boot
root_partition=rootfs
```

2.1.2 编译

系统编译(openwrt)：./build.sh

2. recovery 系统编译：swupdate_make_recovery_img -j64；产物是 recovery.img

openwrt 构建编译的产物主要在：out/{IC}/{BOARD}/openwrt/

buildroot 构建编译的产物主要在：out/{IC}/{BOARD}/buildroot/

2.1.3 OTA 包

生成 OTA 包前需要执行一遍打包命令；

非安全固件打包：

```
./build.sh pack或p;
```

安全固件打包：

```
./build.sh pack_secure或p -s;
```

2. 查看 swupdate 配置文件，看有多少 sw-subimgs-xxx.cfg 格式的文件。

{IC}查看目录：out/{BOARD}/swupdate/

openwrt 查看目录：openwrt/target/{IC}/{IC}-{BOARD}/swupdate

有些方案能看到 sw-subimgs.cfg，此配置一般是 recovery 系统的 OTA 包配置，则生成 OTA 包的命令为：

```
swupdate_pack_swu
```

有些方案能看到 sw-subimgs-recovery.cfg，则生成 OTA 包的命令为：

```
swupdate_pack_swu -recovery
```

成功后生成最后的升级文件。

2.1.4 执行 OTA

1. 将 OTA 包推入设备端，如：

```
adb push xxx.swu /mnt/UDISK
```

2. 设备端使用 swupdate_cmd.sh 升级：

```
swupdate_cmd.sh -i /mnt/UDISK/xxx.swu -e stable,upgrade_recovery

参数说明：
-i: 指定设备端OTA包绝对路径
-e: 指定升级执行的步骤；与具体使用的sw-description有关，比如有些方案的参数是: -e stable,upgrade_recovery_emmc
```

2.2 AB 系统整包升级

2.2.1 配置

在分区表中，将原有的 boot 分区和 rootfs 分区，分区名改为 bootA 和 rootfsA。

将这两个分区配置拷贝一份，即新增两个分区，并把名字改为 bootB 和 rootfsB，downloadfile 不用改。

这样 flash 中就存在 A 系统 (bootA+rootfsA) 和 B 系统 (bootB+rootfsB)。

在 env 中，修改如下，表示烧录完默认从 A 系统启动

```
boot_partition=bootA
root_partition=rootfsA
```

并配置 boot_normal 命令，从 $boot_partition 变量指定的分区加载系统。

⚠ 注意

对安全启方案 AB 系统还要在 dragon_toc.cfg 中增加对 bootA、bootB 的签名配置；

```
diff --git a/configs/default/dragon_toc.cfg b/configs/default/dragon_toc.cfg
index 6941564..36be335 100644
--- a/configs/default/dragon_toc.cfg
+++ b/configs/default/dragon_toc.cfg
@@ -22,6 +22,8 @@ item=optee,           optee.fex,      SoCFirmwareContentCert_KEY
 item=u-boot,          u-boot.fex,     NonTrustedFirmwareContentCertPK
 item=scp,              scp.fex,        SoCFirmwareContentCert_KEY
 onlykey=boot,         boot.fex,       SCPFirmwareContentCertPK
+onlykey=bootA,        boot.fex,       SCPFirmwareContentCertPK
+onlykey=bootB,        boot.fex,       SCPFirmwareContentCertPK
 onlykey=riscv0,               amp_rv0.fex,    RISCV0_KEY
 onlydata=dtb,         sunxi.fex       NULL
 onlydata=board-cfg,   board.fex       NULL
```

图 2-1: 安全 AB 系统配置

2 编译

主系统编译：./build.sh 或 m -j64(openwrt)；

不用 recovery 系统，所以不需要执行 recovery 编译命令。

2.2.3 OTA 包

1. 在生成 OTA 包前需要执行一遍打包命令；

固件打包：

```
./build.sh pack或p;
```

安全固件打包：

```
./build.sh pack_secure或p -s;
```

2. 查看 swupdate 配置文件，看有多少 sw-subimgs-ab-xxx.cfg 格式的文件。

buildroot 查看目录：target/{IC}/buildroot/{BOARD}/swupdate/

wrt 查看目录：openwrt/target/{IC}/{IC}-{BOARD}/swupdate

有些方案能看到 sw-subimgs-ab.cfg，则生成 OTA 包的命令为：

```
swupdate_pack_swu -ab
```

有些方案能看到 sw-subimgs-ab-ubi.cfg，此配置用于 ubi nand；则生成 OTA 包的命令为：

```
swupdate_pack_swu -ab-ubi
```

命令执行成功最后的 log 提示 OTA 包的文件。

2.2.4 执行 OTA

1. 将 OTA 包推入设备端，如：

```
adb push xxx.swu /mnt/UDISK
```

2. 判断设备端当前所处系统；

使用 fw_printenv 读取判断当前的 boot_partition 和 root_partition 的值。

```
A系统：boot_partition=bootA, root_partition=rootfsA

: boot_partition=bootB,root_partition=rootfsB
```

3. 设备端使用 swupdate_cmd.sh 升级：

```
当前处于A系统：
swupdate_cmd.sh -i /mnt/UDISK/xxx.swu -e stable,now_A_next_B
当前处于B系统：
swupdate_cmd.sh -i /mnt/UDISK/xxx.swu -e stable,now_B_next_A

参数说明：
-i: 指定设备端OTA包绝对路径
-e: 指定升级执行的步骤；与具体使用的sw-description有关，比如有些方案的参数是: -e stable,now_A_next_B_emmc
```

2.3 AB 系统差分升级

1. 配置、编译、OTA 整包生成的操作与AB 系统整包升级一致
2. 将第一次得到的 AB 系统 OTA 整包复制到 SDK 根目录 (SDK 其他目录也可)，重命名为 base.swu
3. 修改内核、应用等，如添加打印；然后重新编译打包，生成新的 AB 系统 OTA 整包
4. 将上一步得到的 OTA 整包复制到第 2 步操作同等级目录，重命名为 new.swu
5. 执行以下命令制作差分文件

```
late_make_delta base.swu new.swu
```

6. 执行以下命令生成 AB 系统差分包

```
swupdate_pack_swu -ab-rdiff
```

7. 将 OTA 包推入设备端，如：

```
adb push xxx.swu /mnt/UDISK
```

8. 设备端使用 swupdate_cmd.sh 升级：

```
当前处于A系统：
swupdate_cmd.sh -i /mnt/UDISK/xxx.swu -e stable,now_A_next_B
当前处于B系统：
swupdate_cmd.sh -i /mnt/UDISK/xxx.swu -e stable,now_B_next_A
```

3 Tina SWUpdate OTA 介绍

3.1 swupdate 介绍

3.1.1 简介

date 框架是一提供开源的种灵活可靠的方式来更新嵌入式系统上的软件。

官方源码：

https://github.com/sbabic/swupdate

官方文档：

http://sbabic.github.io/swupdate/

非官方翻译的中文文档：

https://zqb-all.github.io/swupdate/

源码自带文档：

解压 swupdate-xxx.tar.xz，解压后的 doc 目录下即为此版本源码附带的文档。

社区论坛：

https://groups.google.com/forum/#!forum/swupdate

3.1.2 移植到 openwrt 的改动

移植到 openwrt 主要做了以下修改：

在 openwrt/package/allwinner/system/swupdate

- 仿照 busybox，添加了配置项，可通过 make menuconfig 直接配置。
- 添加 patch，支持了更新 boot0，uboot。
- 添加了自启动脚本。
- 默认启动 progress 在后台，输出到串口。这样升级时会打印进度条。实际方案不需要的话，可去除。客户应用可参考 progress 源码，自行获取进度信息。
- 默认启动一个脚本 swupdate_cmd.sh，负责完善参数，最终调用 swupdate。脚本介绍详见后续章节。

3.1.3 移植到 buildroot 的改动

相对于 openwrt，swupdate 移植到 buildroot 系统，主要是软件包存放路径不同：

- 对于 201902 版本：软件包位置在 buildroot/buildroot-201902/package/swupdate。
- 对于 202205 版本：软件包位置在 buildroot/buildroot-202205/package/swupdate。

3.2 配置

介绍overy

若选用主系统 +recovery 系统的方式，则需要一个 recovery 系统。

recovery 系统是一个带 initramfs 的 kernel。

recovery 系统的内核配置由 device/config/chips/{IC}/configs/{BOARD}/BoardConfig.mk 的 LICHEE_KERN_DEFCONF_RECOVERY 指定。

在 openwrt 环境下，该配置文件的路径在 openwrt/target/xxx/xxx/defconfig_ota。

如果没有此文件，可以拷贝 defconfig 为 defconfig_ota，再做配置裁剪。

在 buildroot 环境下，recovery 的配置文件可查看 LICHEE_BR_RAMFS_CONF 变量；实际配置的文件是：BoardConfig.mk；如存在以下内容：

```
LICHEE_BR_VER:=201902
LICHEE_BR_RAMFS_CONF:=sunxi_recovery_ramfs_defconfig
```

则指定 buildroot/buildroot-201902/configs/sunxi_recovery_ramfs_defconfig，配置具体方案时可将该文件拷贝一份为方案名字，再进行修改。

如果没有此文件，可以拷贝 xxx_defconfig 为 sunxiwxxpx_recovery_ramfs_defconfig，再做配置裁剪。

2 系统配置命令

3.2.2.1 openwrt 环境

对于主系统，使用：

```
make menuconfig
```

配置结果保存在：

```
openwrt/target/xxx/xxx/defconfig
```

ecovery 系统，使用：

```
make ota_menuconfig
```

配置结果保存在：

```
openwrt/target/xxx/xxx/defconfig_ota
```

3.2.2.2 buildroot 环境

对于主系统，使用：

令需要在根目录下执行

```
./build.sh buildroot_menuconfig    # 打开配置
./build.sh buildroot_saveconfig    # 保存配置
```

配置结果保存在：

```
buildroot/buildroot-xxx版本/configs/xxx_defconfig
```

对于 recovery 系统，使用 cbr 命令跳转到 buildroot/buildroot-xx 版本目录后，使用：

```
make sunxxiwxx_recovery_ramfs_defconfig    # 根据具体方案配置选择
make menuconfig
make savedefconfig
```

果保存在：

```
buildroot/buildroot-xx版本/configs/sunxxiwxx_recovery_ramfs_defconfig # 根据选择的方案保存配置
```

3.2.3 主系统和 recovery 都需要的 swupdate 包

3.2.3.1 openwrt 环境

选上 swupdate 包。

```
Allwinner      --> System  ---> [*] swupdate
```

swupdate 中还有很多细分选项，一般用默认配置即可。需要的话可以做一些调整，比如裁剪掉网络部分。

swupdate 会依赖选中 uboot-envtools 包，以提供用户空间读写 env 分区的功能。

3.2.3.2 buildroot 环境

选上 swupdate 包：

```
Target packages  ---> System tools  ---> [*] swupdate
```

swupdate 中还有很多细分选项，一般用默认配置即可。需要的话可以做一些调整，比如裁剪掉网络部分。

另外在 buildroot 环境中，swupdate 包依赖的其它软件包不会依赖选中，需要手动选中：

```
Target packages ---> Hardware handling ---> [*] u-boot tools

Target packages ---> allwinner platform private package select ---> system ---> [*] ota-burnboot

packages ---> Libraries ---> Filesystem ---> -*- libconfig

nand 方案升级需要依赖的包：

Target packages ---> Filesystem and flash utilities ---> [*] mtd, jffs2 and ubi/ubifs tools

网络升级除了需要选中网络配置外，还需要依赖下面的包：

Target packages ---> Libraries ---> Networking ---> [*] libcurl

差分升级除了选中swupdate的rdiff功能，需要依赖的包：

Target packages ---> Libraries ---> Networking ---> [*] librsync

签名校验升级除了选中swupdate的签名校验升级功能，需要依赖的包:

packages ---> Libraries ---> Crypto ---> [*] openssl support
```

3.2.4 主系统和 recovery 都需要的 wifimanager daemon

如果想从网络升级，则需要启动系统自动联网。

一种实现方式是，使用 wifimanager daemon。当然，如果用户自己在脚本或应用中去做联网，则不需要此选项。

```
openwrt环境中：


Wireless  --->
  -*- wifimanager-v2.0..... Tina wifimanager-v2.0  --->
    -*-  wifimanager-v2.0-lib..... Tina wifimanager-v2.0 lib
    <*>  wifimanager-v2.0-demo..... Tina wifimanager-v2.0 app demo
  -*- wirelesscommon..... Allwinner Wi-Fi/BT Public lib

buildroot环境中：

Target packages --->
  allwinner platform private package select --->
    wireless --->
```

```
    [*] wifimanager-v2.0
    [*]   wifimanager-v2.0-lib
    [*]     Tina wifimanager-v2.0-demo
    -*- wireless_common

具体的模组需要参考实际情况配置。
```

3.2.5 配置主系统

就以上提到的几个包，暂时没有只针对主系统的需要选的包。

6 编译主系统

执行./build.sh 或 m(openwrt 快捷命令) 即生成主系统。

```
./build.sh
```

3.2.7 配置 recovery 系统

3.2.7.1 内核配置

a5.0独立内核编译，需要在 config.mk 中添加以下配置。

```
对于5.10以前的内核，一般使用的是：
LICHEE_KERN_DEFCONF_RECOVERY:=config-x.x-recovery

如果是5.10以上内核，一般使用的是：
LICHEE_KERN_DEFCONF_RECOVERY:=bsp_recovery_defconfig
```

在 Tina5.0 中，recovery 的内核需要支持 init ramdisk，以及 gzip 格式的压缩 ramdisk。内核配置 bsp_recovery_defconfig 中需要选中下面的选项：

```
CONFIG_BLK_DEV_INITRD=y
CONFIG_RD_GZIP=y
```

注意

在 Tina5.0 中，recovery 的内核可以通过./build.sh recovery_menuconfig 配置；然后执行./build.sh recovery_saveconfig 保存

修改过后需要重新 source、lunch。

按照上述修改后 recovery 系统默认使用 device/config/chips/xxx/configs/xxx/linux-xx/config-x.x-recovery 或 bsp_recovery_defconfig。

3.2.7.2 应用配置

在 openwrt 环境中，在主系统的基础上，复制一份 deconfig 为份 deconfig_ota，并作裁剪，更改配置以下选项即可：

```
make ota_menuconfig
  ---> Target Images
    [*] don't create rootfs.img for recovery
```

在 buildroot 环境中，由 BoardConfig.mk 的 LICHEE_BR_RAMFS_CONF 配置指定 recovery 的 defconfig 之后选手相关软件包即可。

8 编绕 recovery

要编译生成 recovery 系统，可使用：

```
swupdate_make_recovery_img
```

编译得到：

```
在openwrt环境中，编译后生成的文件是 out/${LICHEE_IC}/${LICHEE_BOARD}/openwrt/recovery.img

在buildroot环境中，编译后生成的文件是 out/${LICHEE_IC}/${LICHEE_BOARD}/buildroot/recovery.img
```

9 配置 env

本方案推荐使用 env 来保存信息，不使用 misc 分区。

uboot 会从 env 分区读取启动命令，并根据启动命令来启动系统。只要我们能在用户空间改动到 env，即可控制下次启动的系统。

openwrt 配置：

```
Utilities --->
  Boot Loaders --->
    -*- uboot-envtools..... read/modify U-Boot bootloader environment
```

buildroot 配置：

```
Target packages --->
  Hardware handling --->
    [*] u-boot tools
    [*]   fw_printenv
      choose fw_env.config for emmc/nand --->    # 根据介质选择是否有备份env
```

3.2.9.1 boot_partition 变量

增加一个 boot_partition 变量，用于指定要启动的内核所在分区。

配置 env 主要是修改 boot_normal 命令，将要启动的分区独立成 boot_partition 变量。

即从：

```
boot_normal=sunxi_flash read 43000000 boot;bootm 43000000
```

改成：

```
boot_partition=boot
boot_normal=sunxi_flash read 43000000 ${boot_partition};bootm 43000000
```

这样可以通过按键选择不同的系统，无需
按照 boot_normal 启动即可。

对于 recovery 方案，可设置 boot_partition 为 boot 或 recovery。OTA 切换系统时，只需要改变此变量即可达到切换主系统和 recovery 系统的目的。

对于 AB 系统方案，可设置为 boot_partition 为 bootA 或 bootB。OTA 切换系统时，只需要改变此变量即可达到切换 kernel 的目的。

3.2.9.2 root_partition 变量

动的 rootfs

uboot 会解析分区表，找出此变量指定的分区并在 cmdline 中指定 root 参数。

例如，在 env 中设置：

```
root_partition=rootfs
```

则启动时 uboot 会遍历分区表，找到名字为 rootfs 的分区，假设找到的分区为/dev/mmcblk0p5，则在 cmdline 中增加 root=/dev/mmcblk0p5。

kernel 需要挂载 rootfs 时，取出 root 参数，则得知需要挂载/dev/nand0p4 分区。

ecovery 方案，就一直设置 root_partition 为 rootfs 即可。主系统需要从 rootfs 分区读取数据，而 recovery 系统使用 initramfs，无需从 rootfs 分区读取数据即可正常运行 OTA 应用等。当然，recovery 系统中要更新 rootfs 的话，还是会访问 (写入)rootfs 分区的，但这个动作就跟 env 的 root_partition 无关了。

对于 AB 系统方案，可设置 root_partition 为 rootfsA 或 rootfsB，以匹配不同的系统。OTA 切换系统时，只需要改变此变量即可达到切换 rootfs 的目的。

3.2.10 配置备份 env

由于写入 env 时断电，可能导致 env 的数据被破坏，因此需要支持备份 env。由于使用习惯，也可以继续使用单个 env 分区，只是有掉电风险。

3.2.10.1 方式：增加 env-redund 分区

此方式是 uboot 原生功能。虽然也需要修改 sunxi 的 env 读取代码进行适配，但总体读写逻辑是社区原生的。

启用方法：

1. 增加 env-redund 分区。

将 env 分区复制一份，分区名改为 env-redund。

注意只是分区名修改为 env-redund，其 downloadfile 仍然指定为 env.fex

2. 支持在打包时制作冗余 env。

```
tina5.0配置方式：
在device/config/chips/{IC}/configs/{BOARD}/BoardConfig.mk文件添加
LICHEE_REDUNDANT_ENV_SIZE:=0x20000
然重新source,lunch
```

注意事项：

启用上述选项之后，打包时会调用 mkenvimage 工具来制作 env，对 env 的格式有一定要求。如注释和有效配置不能合并在一行。

若 env-x.x.cfg 中存在类似如下配置

```
bootcmd=run setargs_nand boot_normal#default nand boot
```

则需要改成：

```
lt nand boot
md=run setargs_nand boot_normal
```

3. 配置 uboot 并重新编译 uboot bin。

在对应方案的 uboot defconfig 文件，如：

```
brandy/brandy-2.0/u-boot-xx版本/configs/sunxxiwxx_defconfig
```

中增加配置：

```
CONFIG_SUNXI_REDUNDAND_ENVIRONMENT=y
CONFIG_SYS_REDUNDAND_ENVIRONMENT=y
CONFIG_SUNXI_ENV_REDUNDAND_PARTITION="env-redund"
CONFIG_ENV_SIZE=0x20000
```

重新编译 uboot。

4. 修改 fw_env.config

拷贝

```
brandy/brandy-2.0/u-boot-xx版本/tools/env/fw_env.config
```

到目标目录

```
openwrt:
openwrt/target/{IC}/{IC}-{BOARD}/base-files/etc/ （若使用procd-init）
openwrt/target/{IC}/{IC}-{BOARD}/busybox-init-base-files/etc/ （若使用busybox-init）

buildroot-2022:
target/{IC}/buildroot/{BOARD}/overlay/etc（依赖buildroot对应defconfig开启BR2_ROOTFS_OVERLAY）
```

修改拷贝后的 fw_env.config，增加备份 env 的配置。

如果使用 emmc 方案，例如原本最后一行为：

```
/dev/by-name/env      0x0000      0x20000
```

则增加一行：

```
y-name/env-redund 0x0000 0x20000
```

如果使用 ubinand 方案，例如原本最后一行为：

```
/dev/ubi0:env    0x0    0x20000    0x20000
```

则增加一行：

```
/dev/ubi0:env-redund   0x0    0x20000    0x20000
```

在 buildroot-2019 环境中，fw_env.config 存放在 buildroot/buildroot-201902/package/uboot-tools，makefile 根据选择的介质以及是否有备份 env 的情况将 fw_env.config 拷贝到小机端。

这样用户空间的 fw_printenv 和 fw_setenv 即可正确处理两份 env。

3.2.11 配置启动脚本

1. openwrt 环境 procd-init 是默认配置好的。

busybox-init 需要手工配置下。

拷贝

```
openwrt/package/allwinner/system/busybox-init-base-files/files/etc/init.d/load_script.conf
```

```
openwrt/target/{IC}/{IC}-{BOARD}/busybox-init-base-files/etc/init.d/
```

并在其中添加一行：

```
swupdate_autorun
```

2. buildroot 环境默认配置好

3.3 OTA 包

OTA 包中，需要包含 sw-description 文件，以及本次升级会用到的各个文件，例如 kernel，rootfs。

整个 OTA 包是 cpio 格式，且要求 sw-description 文件在第一个。

3.3.1 OTA 策略描述文件：sw-description

sw-description 文件是 swupdate 官方规定的，OTA 策略的描述文件，具体语法可参考 swupdate 官方文档。

几个示例：

```
在build/swupdate中
```

也可以自行为具体的方案编写描述文件：

```
在openwrt中，为openwrt/target/{IC}/{IC}-{BOARD}/swupdate/sw-description

在buildroot中，为target/{IC}/buildroot/{BOARD}/swupdate/sw-description
```

本文件在 SDK 中的存放路径和名字没有限定，只要最终打包进 OTA 包中，重命名为 sw-description 并放在第一个文件即可。

3.3.2 OTA包配置文件:cfg

sw-subimgs.cfg 是 tina 提供的，用于指示如何生成 OTA 包。

基本格式为

```
swota_file_list=(
#表示把文件xxx拷贝到swupdate目录下，重命名为yyy，并把yyy打包到最终的OTA包中
xxx:yyy
)
```

```
swota_copy_file_list=(
xxx拷贝到swupdate根目录下，重命名为yyy，但不把yyy打包到最终的OTA包中
xxx:yyy
)
```

swota_copy_file_list 存在的原因是，有一些文件我们只需要其 sha256 值，而不需要文件本身。例如使用差分包配合 readback handler 时，readback handler 需要原始镜像的 sha256 值用于校验。

例子：

```
swota_file_list=(
openwrt/target/${LICHEE_IC}/${LICHEE_IC}-${LICHEE_BOARD}/swupdate/sw-description-ab:sw-description
${LICHEE_PACK_OUT_DIR}/boot_package.fex:uboot
${LICHEE_PACK_OUT_DIR}/boot0_sdcard.fex:boot0
EE_PACK_OUT_DIR}/boot.fex:kernel
${LICHEE_PACK_OUT_DIR}/rootfs.fex:rootfs
)
```

tina 提供了几个示例：

```
build/swupdate/sw-subimgs.cfg          # 普通系统，recovery系统，整包升级。这是其余demo的基础版本。
build/swupdate/sw-subimgs-ab.cfg       # 改为AB系统。
build/swupdate/sw-subimgs-secure.cfg   # 改为安全系统。
build/swupdate/sw-subimgs-ab-rdiff.cfg # 改为AB方案，差分方案。
build/swupdate/sw-subimgs-readback.cfg # 增加回读校验。
build/swupdate/sw-subimgs-sign.cfg     # 增加签名校验。
build/swupdate/sw-subimgs-ubi.cfg      # 改为ubi方案。
```

也可以自行为具体的方案编写配置文件：

```
nwrt中，为openwrt/target/{IC}/{IC}-{BOARD}/swupdate/sw-subimgs.cfg

在buildroot中，为target/{IC}/buildroot/{BOARD}/swupdate/sw-subimgs.cfg
```

名字需要命名为 sw-subimg.cfg 或 sw-subimgsxxx.cfg，其中 xxx 可自定义。

这个限定主要是为了方便打包函数处理。在打包时，命令行传入参数 xxx，则会使用 sw-subimgsxxx.cfg 进行打包。

3.3.3 OTA 包生成：swupdate_pack_swu

wupdate_pack_swu 函数。

可以参考该函数，自行实现一套打包 swupdate 升级包的脚本。也可以直接使用，使用方式如下。

⚠ 注意

生成 OTA 包前需要先执行打包命令：pack

1. 准备好 sw-descrition 文件，具体作用和语法请参考官方 swupdate 说明文档，下文也有举例。

2. 准备好 sw-subimgs.cfg 文件，里面需要每一行列出一个打包需要的子镜像文件，即内核，rootfs 等。可以使用冒号指定打包进固件的文件名，也可以没有冒号则使用原文件名字。使用相对于 tina 根目录的相对路径进行描述。其中第一个必须为 sw-description。

3. 编译好所需的子镜像，例如主系统的内核和 rootfs，recovery 系统等。

4. 执行 swupdate_pack_swu 生成 swupdate 升级包。

不带参数执行，则会在特定路径下寻找 sw-subimgs.cfg，解析配置生成 OTA 包。不带参数默认生成 recovery 的 OTA 包。

带参数-xx 执行，则会在特定路径下寻找 sw-subimgs-xx.cfg，解析配置生成 OTA 包。例如执行

则会寻找 sw-subimgs-recovery.cfg

gs-xx.cfg。

注：不同介质使用的 boot0/uboot 镜像不同，swupate_pack_swu 需要 sys_config.fex 中的 storage_type 配置明确指出介质类型，才能取得正确的 boot0.img 和 uboot.img。

具体可直接查看 build 目录中 swupdate_pack_swu 的实现。参考 build/buildbase.sh 文件。

3.4 recovery 系统方案举例

配置分区和

在分区表中，增加一个 recovery 分区，用于保存 recovery 系统。

size 根据实际 recovery 系统的大小，再加点裕量。

download_file 可以留空，因为 OTA 第一步就是写入一个 recovery 系统。

当然也可以配置上 download_file，并在打包固件之前先编译好 recovery 系统，一并打包到固件中，这样出厂就带 recovery 系统，后续的 OTA 执行过程，可以考虑不写入 recovert 系统，用现成的，直接重启并升级主系统。

download_file 为 recovery.fex，分区表添加 recovery 参考：

```
[partition]
    name        = recovery
    size        = 65536
    downloadfile = "recovery.fex"
    user_type    = 0x8000
```

在 env 中指定：

```
boot_partition=boot
root_partition=rootfs
```

并配置 boot_normal 命令，从 $boot_partition 变量指定的分区加载系统。

3.4.2 配置主系统

lunch 选择方案后，make menuconfig，选上 swupdate。

3.4.3 配置 recovery 系统

3.4.3.1 内核配置

操作如下：

```
cd device/config/chips/{IC}/configs/{BOARD}/linux-x.x
cp config-5.4 config-5.4-recovery     # linux-5.4及以前环境
cp bsp_defconfig bsp_recovery_defconfig # linux-5.4以后环境
```

在 device/config/chips/{IC}/configs/{BOARD}/buildroot 目录下，修改 BoardConfig.mk 文件，linux-5.4 添加下面这行代码。

```
LICHEE_KERN_DEFCONF_RECOVERY:=config-5.4-recovery
```

linux-5.4 以后的内核添加面这行代码

```
LICHEE_KERN_DEFCONF_RECOVERY:=bsp_recovery_defconfig
```

添加后要回到根目录下，重新 source，./build.sh config，重新选择方案。

recovery 的内核需要支持 init ramdisk，以及 gzip 格式的压缩 ramdisk。内核配置 bsp_recovery_defconfig 中需要选中下面的选项：

```
CONFIG_BLK_DEV_INITRD=y
CONFIG_RD_GZIP=y
```

recovery 系统整个运行在 ram 中，如果系统过大会无法启动，所以需要进行裁剪。将不必要的包尽量从 recovery 系统的内核中去掉。

在根目录执行：

```
./build.sh recovery_menuconfig # 打开 recovery 内核配置
./build.sh recovery_saveconfig  # 保存 recovery 内核配置
```

可以对 recovery 系统的应用进行裁剪。

3.4.3.2 openwrt 环境

假设没有现成的 recovery 系统配置，则我们从主系统配置修改得到。lunch 选择方案后，拷贝配置文件。

```
cplat(快捷命令)
cp defconfig defconfig_ota
```

根据上文介绍，make ota_menuconfig 选上 swupdate 等必要的配置。

recovery 系统整个运行在 ram 中，如果系统过大会无法启动，所以需要进行裁剪。make ota_menuconfig，将不必要的包尽量从 recovery 系统中去掉。

3.4.3 buildroot

假设没有现成的 recovery 系统配置，则我们从主系统配置修改得到。需要在根目录执行./build.sh config 选择方案后，拷贝配置文件。

recovery 的软件包配置在 buildroot/buildroot-xx 版本/configs 目录下，如果没有方案对应配置，可以将 sunxxiwxx_recovery_ramfs_defconfig 拷贝为当前方案配置。例：

```
cp sunxxiwxx_defconfig sunxxiwxx_recovery_ramfs_defconfig
在device/config/chips/{IC}/configs/{BOARD}/buildroot目录下，修改BoardConfig.mk文件，添加下面这行代码。

LICHEE_BR_RAMFS_CONF=sunxxiwxx_recovery_ramfs_defconfig
```

要回到根目录执行./build.sh config

3.4.4 准备 sw-description

这里我们直接使用：

```
在openwrt中：

openwrt/target/{IC}/{IC}-{BOARD}/swupdate/sw-description

在buildroot中：

/{IC}/buildroot/{BOARD}/swupdate/sw-subimgs.cfg
```

内容如下，中文部分是注释，原文件中没有。

```
/* 固定格式，最外层为software = { } */
software =
{
  /* 版本号和描述 */
  version = "0.1.0";
  description = "Firmware update for Tina Project";

  /*
```

```
 * 外层tag，stable，
 * 没有特殊含义，就理解为一个字符串标志即可。
 * 可以修改，调用的时候传入匹配的字符串即可
 */
stable = {

  /*
   * 内层tag，upgrade_recovery,
   * 当调用swupdate xxx -e stable,upgrade_recovery时，就会匹配到这部分，执行 {} 内的动作，
    * 可以修改，调用的时候传入匹配的字符串即可
   */
  /* upgrade recovery,uboot,boot0 ==> change swu_mode,boot_partition ==> reboot */
  upgrade_recovery = {
    /* 这部分是为了在主系统中，升级recovery系统，升级uboot和boot0 */
    /* upgrade recovery */
    images: ( /* 处理各个image */
      {
源文件是OTA包中的recovery文件 */
        device = "/dev/by-name/recovery"; /* 要写到/dev/by-name/recovery节点中, 这个节点在tina上就对应recovery分区 */
        /*volume = "recovery";*/  /* 当使用ubinand方案时，需要将上面的device = "/dev/by-name/recovery";注释掉，修改为当前语句 */
        installed-directly = true; /* 流式升级，即从网络升级时边下载边写入, 而不是先完整下载到本地再写入，避免占用额外的RAM或ROM */
      },
      {
        filename = "uboot"; /* 源文件是OTA包中的uboot文件 */
        type = "awuboot"; /* type为awuboot，则swupdate会调用对应的handler做处理 */
      },
      {
        filename = "boot0"; /* 源文件是OTA包中的boot0文件 */
        type = "awboot0"; /* type为awuboot，则swupdate会调用对应的handler做处理 */
      }

    /* image处理完之后，需要设置一些标志，切换状态 */
    /* change swu_mode to upgrade_kernel,boot_partition to recovery & reboot*/
    bootenv: ( /* 处理bootenv，会修改uboot的env分区 */
      {
        /* 设置env:swu_mode=upgrade_kernel, 这是为了记录OTA进度 */
        name = "swu_mode";
        value = "upgrade_kernel";
      },
      {
        /* 设置env:boot_partition=recovery, 这是为了切换系统，下次uboot就会启动recovery系统(kernel位于recovery分区) */
        name = "boot_partition";
        value = "recovery";
      },
      {
        /* 设置env:swu_next=reboot, 这是为了跟外部脚本配合，指示外部脚本做reboot动作 */
        name = "swu_next";
        value = "reboot";
      }
      /* 实际有什么其他需求，都可以灵活增删标志来解决, 外部脚本和应用可通过fw_setenv/fw_printenv操作env */
      /* 注意，以上几个env，是一起在ram中修改好再写入的, 不会出现部分写入部分未写入的情况 */
    );
  };

  /*
   * 内层tag，upgrade_kernel,
```

```
        * 当调用swupdate xxx -e stable,upgrade_kernel时，就会匹配到这部分，执行 {} 内的动作，
         * 可以修改，调用的时候传入匹配的字符串即可。
        */
        /* upgrade kernel,rootfs ==> change sw_mode */
        upgrade_kernel = {
          /* upgrade kernel,rootfs */
          /* image部分，不赘述 */
          images: (
            {
              filename = "kernel";
              device = "/dev/by-name/boot";
              /*volume = "boot";*/     /* 当使用ubinand方案时，需要将上面的device = "/dev/by-name/boot";注释掉，修改为
          当前语句 */
              installed-directly = true;
            },
            {
              filename = "rootfs";
device = "/dev/by-name/rootfs";
              /*volume = "rootfs";*/     /* 当使用ubinand方案时，需要将上面的device = "/dev/by-name/rootfs";注释掉，修改
          为当前语句 */
              installed-directly = true;
            }
          );
          /* change sw_mode to upgrade_usr,change boot_partition to boot */
          bootenv: (
            {
              /* 设置env:swu_mode=upgrade_usr, 这是为了记录OTA进度 */
              name = "swu_mode";
              value = "upgrade_usr";
            },
            {
              /* 设置env:boot_partition=boot, 这是为了切换系统，下次uboot就会启动主系统(kernel位于boot分区) */
              name = "boot_partition";
value = "boot";
            }
          );
        };

        /* 内层tag，upgrade_usr,
          当调用swupdate xxx -e stable,upgrade_usr时，就会匹配到这部分，执行 {} 内的动作，
          可以修改，调用的时候传入匹配的字符串即可 */
        /* upgrade usr ==> clean ==> reboot */
        upgrade_usr = {
          /*
           * misc-upgrade的小容量方案，将usr拆成独立分区了。
                   * 这里我们不需要，如果保留的话，不做任何image操作即可。
           * 也可以彻底删除这一部分，并将上面的upgrade_usr改掉。
                   */
          /* upgrade usr */

          /* OTA结束，清空各种标志 */
          /* clean swu_param,swu_software,swu_mode & reboot */
          bootenv: (
            {
              name = "swu_param";
              value = "";
            },
            {
              name = "swu_software";
              value = "";
```

```
            },
            {
                name = "swu_mode";
                value = "";
            },
            {
                name = "swu_next";
                value = "reboot";
            }
        );
    };

};

/* 当没有匹配上面的tag，进入对应的处理流程时，则运行到此处。我们默认清除掉一些状态 */
/* when not call with -e xxx,xxx   just clean */
bootenv: (

    name = "swu_param";
    value = "";
  },
  {
    name = "swu_software";
    value = "";
  },
  {
    name = "swu_mode";
    value = "";
  },
  {
    name = "swu_version";
    value = "";
  }


}
```

说明：

升级过程会进行两次重启。具体的：

（1）升级 recovery 分区 (recovery)，uboot(uboot)，boot0(boot0) 。设置 boot_partition 为 recovery。

（2）重启，进入 recovery 系统。

（3）升级内核 (kernel) 和 rootfs(rootfs) 。设置 boot_partition 为 boot。

（4）重启，进入主系统，升级完成。

3.4.5 准备 sw-subimgs.cfg

我们直接看下 tina 默认的：

在openwrt环境中：

target/{IC}/{BOARD}/swupdate/sw-subimgs.cfg

在buildroot环境中：

target/{IC}/buildroot/{BOARD}/swupdate/sw-subimgs-recovery.cfg

以 openwrt 为例。内容如下，中文部分是注释，原文件中没有。

```
swota_file_list=(
#取得sw-description，放到OTA包中。
#注意第一行必须为sw-description。如果源文件不叫sw-description，可在此处加:sw-description做一次重命名
openwrt/target/${LICHEE_IC}/${LICHEE_IC}-${LICHEE_BOARD}/swupdate/sw-description-recovery:sw-description
#取得recovery.img，重命名为recovery，放到OTA包中。以下雷同
${LICHEE_PACK_OUT_DIR}/recovery.fex:recovery
#uboot.img和boot0.img是执行swupdate_pack_swu时自动拷贝得到的，需配置sys_config.fex中的storage_type，不配置则

${LICHEE_PACK_OUT_DIR}/boot_package.fex:uboot
#注：boot0没有修改的话，以下这行可去除，其他雷同，可按需升级
${LICHEE_PLAT_OUT}/boot0.img:boot0

${LICHEE_PACK_OUT_DIR}/boot.fex:kernel
${LICHEE_PACK_OUT_DIR}/rootfs.fex:rootfs
#下面这行是给小容量方案预留的，目前注释掉
#${LICHEE_PLAT_OUT}//usr.img:usr
)
```

说明：

指明打包 swupdate 升级包所需的各个文件的位置。这些文件会被拷贝到 out 目录下，再生成 swupdate OTA 包。

3.4.6 编译 OTA 包所需的子镜像

编译 kernel 和 rootfs。

```
make -j<N>
```

编译 recovery 系统。

```
swupdate_make_recovery_img -j<N>
```

编译 uboot。

```
muboot
```

打包，若需要升级 boot0/uboot，则是必要步骤，打包会将 boot0 和 uboot 拷贝到 out 目录下，并对头部参数等进行修改。生成的固件也可用于测试。注：如果希望生成的固件的 recovery 分区是有系统的，则需要先编译 recovery 系统，再打包。

```
pack / pack -s
```

生成 OTA 包。因为我们使用的就是 sw-subimgs.cfg，所以不同带参数。

注意，如果方案目录下存在 sw-subimgs.cfg，则优先用方案目录下的。没有方案特定配置才用 build/swu 下的。如果需要升级
age_type 参数，swupdate_pack_swu 才能正确拷贝对应的 boot0/uboot。

```
swupdate_pack_swu
```

3.4.7 执行 OTA

3.4.7.1 准备 OTA 包

对于测试来说，直接推入。

```
sh xxx.swu /mnt/UDISK
```

实际应用时，可从先从网络下载到本地，再调用 swupdate，也可以直接传入 url 给 swupdate。

3.4.7.2 调用 swupdate

若使用原生的 swupdate，则调用：

```
swupdate -i /mnt/UDISK/<board>.swu -e stable,upgrade_recovery
```

但这样不会在自启动的时候帮我们准备好 swupdate 所需的-e 参数。

以使用辅助脚本：

```
swupdate_cmd.sh -i /mnt/UDISK/xxx.swu -e stable,upgrade_recovery
```

3.5 AB 系统方案举例

3.5.1 配置分区和 env

在分区表中，将原有的 boot 分区和 rootfs 分区，分区名改为 bootA 和 rootfsA。

将这两个分区配置拷贝一份，即新增两个分区，并把名字改为 bootB 和 rootfsB。

这样 flash 中就存在 A 系统 (bootA+rootfsA) 和 B 系统 (bootB+rootfsB)。

一般是一个系统烧录两份。即分区表中的 bootA 和 bootB 都指定的 boot.fex，rootfsA 和 rootfsB 都指定的 rootfs.fex。

在 env 中，指定：

```
boot_partition=bootA
root_partition=rootfsA
```

并配置 boot_normal 命令，从 $boot_partition 变量指定的分区加载系统。

3.5.2 配置主系统

3.5.2.1 openwrt 环境

lunch 选择方案后，make menuconfig，选上 swupdate。

3.5.2.2 buildroot

./build.sh config 选择方案后，./build.sh buildroot_menuconfig 以及./build.sh buildroot_saveconfig 选择 swupdate 以及相关依赖配置，并保存。

3.5.3 配置 recovery 系统

AB 系统方案没有使用 recovery 系统，无需配置和生成。

3.5.4 准备 sw-description

这里我们直接使用：

```
在openwrt环境中：

openwrt/target/{IC}/{IC}-{BOARD}/swupdate/sw-description-ab 或 sw-description-ab-ubi

在buildroot环境中：

target/{IC}/buildroot/{BOARD}/swupdate/sw-description-ab
```

内容如下，中文部分是注释，原文件中没有。

```
/* 固定格式，最外层为software = { } */
software =
{
  /* 版本号和描述 */
  version = "0.1.0";
  description = "Firmware update for Tina Project";

  /*
   * 外层tag，stable，
   * 没有特殊含义，就理解为一个字符串标志即可。
   * 可以修改，调用的时候传入匹配的字符串即可。
```

```
 */
stable = {

 /*
  * 内层tag，now_A_next_B,
  * 当调用swupdate xxx -e stable,now_A_next_B时，就会匹配到这部分，执行 {} 内的动作，
   * 可以修改，调用的时候传入匹配的字符串即可。
  */
 /* now in systemA, we need to upgrade systemB(bootB, rootfsB) */
 now_A_next_B = {
   /* 这部分是描述，当前处于A系统，需要更新B系统，该执行的动作。执行完后下次启动为B系统 */
   images: ( /* 处理各个image */
     {
       filename = "kernel"; /* 源文件是OTA包中的kernel文件 */
       device = "/dev/by-name/bootB"; /* 要写到/dev/by-name/bootB节点中, 这个节点在tina上就对应bootB分区 */
       /*volume = "bootB";*/    /* 当使用ubinand方案时，需要将上面的device = "/dev/by-name/bootB";注释掉，修改
为当前语句 */
测试升级，即在网络传输过程中写入，避免占用额
外的RAM或ROM */
     },
     {
       filename = "rootfs"; /* 同上，但处理rootfs，不赘述 */
       device = "/dev/by-name/rootfsB";
       /*volume = "rootfsB";*/     /* 当使用ubinand方案时，需要将上面的device = "/dev/by-name/rootfsB";注释掉，修
改为当前语句 */
       installed-directly = true;
     },
     {
       filename = "uboot"; /* 源文件是OTA包中的uboot文件 */
       type = "awuboot"; /* type为awuboot，则swupdate会调用对应的handler做处理 */
     },
     {
       filename = "boot0"; /* 源文件是OTA包中的boot0文件 */
为atype=
       }
   );
   /* image处理完之后，需要设置一些标志，切换状态 */
   bootenv: ( /* 处理bootenv，会修改uboot的env分区 */
     {
       /* 设置env:swu_mode=upgrade_kernel, 这是为了记录OTA进度, 对于AB系统来说，此时已经升级完成，置空 */
       name = "swu_mode";
       value = "";
     },
     {
       /* 设置env:boot_partition=bootB, 这是为了切换系统，下次uboot就会启动B系统(kernel位于bootB分区) */
       name = "boot_partition";
       value = "bootB";
     },
     {
       /* 设置env:root_partition=rootfsB, 这是为了切换系统，下次uboot就会通过cmdline指示挂载B系统的rootfs */
       name = "root_partition";
       value = "rootfsB";
     },
     {
       /* 兼容另外的切换方式，可以先不管 */
       name = "systemAB_next";
       value = "B";
     },
     {
       /* 设置env:swu_next=reboot, 这是为了跟外部脚本配合，指示外部脚本做reboot动作 */
```

```
            name = "swu_next";
            value = "reboot";
        }
    );
};

/*
 * 内层tag，now_B_next_A,
 * 当调用swupdate xxx -e stable,now_B_next_A时，就会匹配到这部分，执行 {} 内的动作，
  * 可以修改，调用的时候传入匹配的字符串即可
 */
/* now in systemB, we need to upgrade systemA(bootA, rootfsA) */
now_B_next_A = {
    /* 这里面就不赘述了, 跟上面基本一致，只是AB互换了 */
    images: (
        {
            filename = "kernel";
device = "/dev/by-name/bootA";
            /*volume = "bootA";*/    /* 当使用ubinand方案时，需要将上面的device = "/dev/by-name/bootA";注释掉，修改为当前语句 */
            installed-directly = true;
        },
        {
            filename = "rootfs";
            device = "/dev/by-name/rootfsA";
            /*volume = "rootfsA";*/    /* 当使用ubinand方案时，需要将上面的device = "/dev/by-name/rootfsA";注释掉，修改为当前语句 */
            installed-directly = true;
        },
        {
            filename = "uboot";
            type = "awuboot";
        },
{
            filename = "boot0";
            type = "awboot0";
        }
    );
    bootenv: (
        {
            name = "swu_mode";
            value = "";
        },
        {
            name = "boot_partition";
            value = "bootA";
        },
        {
            name = "root_partition";
            value = "rootfsA";
        },
        {
            name = "systemAB_next";
            value = "A";
        },
        {
            name = "swu_next";
            value = "reboot";
        }
    );
```

```
    };
  };

  /* 当没有匹配上面的tag，进入对应的处理流程时，则运行到此处。我们默认清除掉一些状态 */
  /* when not call with -e xxx,xxx   just clean */
  bootenv: (
    {
      name = "swu_param";
      value = "";
    },
    {
      name = "swu_software";
      value = "";
    },
    {
      name = "swu_mode";
      value = "";

    {
      name = "swu_version";
      value = "";
    }
  );

}
```

说明：

升级过程会进行一次重启。具体的：

（1）升级 kernel 和 rootfs 到另一个系统所在分区，升级 uboot(uboot)，boot0(boot0) 。设置 partition 为切换系统。

（2）重启，进入新系统。

3.5.5 准备 sw-subimgs.cfg

我们直接看下 tina 默认的：

```
在openwrt环境中：

openwrt/target/{IC}/{IC}-{BOARD}/swupdate/sw-subimgs-ab.cfg 或 sw-subimgs-ab-ubi.cfg

在buildroot环境中：

target/{IC}/buildroot/{BOARD}/swupdate/sw-subimgs-ab.cfg
```

以 openwrt 为例。内容如下，中文部分是注释，原文件中没有。

```
swota_file_list=(
#取得sw-description-ab，重命名成sw-description，放到OTA包中。
#注意第一行必须为sw-description
openwrt/target/${LICHEE_IC}/${LICHEE_IC}-${LICHEE_BOARD}/swupdate/sw-description-ab:sw-description
#取得uboot.img，重命名为uboot，放到OTA包中。以下雷同
#uboot.img和boot0.img是执行swupdate_pack_swu时自动拷贝得到的，需配置sys_config.fex中的storage_type
${LICHEE_PLAT_OUT}/uboot.img:uboot
```

```
#注：boot0没有修改的话，以下这行可去除，其他雷同，可按需升级
${LICHEE_PLAT_OUT}/boot0.img:boot0

${LICHEE_PACK_OUT_DIR}/boot.fex:kernel
${LICHEE_PACK_OUT_DIR}/rootfs.fex:rootfs
)
```

说明：

指明打包 swupdate 升级包所需的各个文件的位置。这些文件会被拷贝到 out 目录下，再生成 swupdate OTA 包。

3.5.6 编译 OTA 包所需的子镜像

ernel 和 rootfs

```
make -j<N>
```

编译 uboot。

```
muboot
```

打包，若需要升级 boot0/uboot，则是必要步骤，打包会将 boot0 和 uboot 拷贝到 out 目录下，并对头部参数等进行修改。生成的固件也可用于测试。

```
pack / pack -s
```

生成 OTA 包。因为我们使用的是 sw-subimgs-ab.cfg 或 sw-subimgs-ab-ubi.cfg，所以调用时带参

或 -ab-ubi。

注意，如果方案目录下存在 sw-subimgs-ab.cfg，则优先用方案目录下的。没有方案特定配置才用 build/swupdate 下的。

```
swupdate_pack_swu -ab 或 swupdate_pack_swu -ab-ubi
```

3.5.7 执行 OTA

3.5.7.1 准备 OTA 包

对于测试来说，直接推入。

```
adb push <board>.swu /mnt/UDISK
```

实际应用时，可从先从网络下载到本地，再调用 swupdate，也可以直接传入 url 给 swupdate。

3.5.7.2 判断 AB 系统

对于 AB 系统方案来说，必须判断当前所处系统，才能知道需要升级哪个分区的数据。

判断当前是处于 A 系统还是 B 系统。

方式一：直接使用 fw_printenv 读取判断当前的 boot_partition 和 root_partition 的值。

3.5.7.3 调用 swupdate

若使用原生的 swupdate，则调用：

```
当前处于A系统：
swupdate -i /mnt/UDISK/<board>.swu -e stable,now_A_next_B
当前处于B系统：
swupdate -i /mnt/UDISK/<board>.swu -e stable,now_B_next_A
```

但这样不会在自启动的时候帮我们准备好 swupdate 所需的-e 参数。

我们可以使用辅助脚本：

```
当前处于A系统：
swupdate_cmd.sh -i /mnt/UDISK/<board>.swu -e stable,now_A_next_B
当前处于B系统：
swupdate_cmd.sh -i /mnt/UDISK/<board>.swu -e stable,now_B_next_A
```

3.6 辅助脚本 swupdate_cmd.sh

为什么需要辅助脚本?

因为我们需要启动时能自动调用 swupdate，自动传递合适的-e 参数给 swupdate，需要在合适的时候调用重启。

具体可直接看下脚本内容。

其基本思路是，当带参数调用时，脚本从传入的参数中，取出”-e xxx,yyy” 部分，将其余参数原样保存为 env 的 swu_param 变量。

取出的”-e xxx,yyy” 中的 xxx 保存到 env 的 swu_software 变量，yyy 保存为 env 的 swu_mode 变量。

然后就取出变量，循环调用。

```
swupdate $swu_param -e "$swu_software，$swu_mode"
```

sw-description 中可以通过改变 env 的 swu_software 和 swu_mode 变量，来影响下次的调用参数。

实际应用时，可不使用此脚本，直接在主应用中，调用 swupdate 即可。但要自行做好-e 参数的

3.6.1 自适应 nand/emmc

⚠ 注意

此功能只是开发阶段测试使用；一般项目量产不会同时使用 2 种存储介质都作为系统启动介质；
普遍是一种介质作为系统启动，另外一种介质作为客户资源的存储介质。
如：系统从 nand 启动，emmc 作为客户多媒体资源存放的存储介质。则系统 OTA 升级主要负责 nand 升级，emmc 使用其他方法单独维护。

目前有很多方案有两种介质，所以在脚本中添加了可以区分 nand/emmc 方案的升级命令。

工作原理：在升级包中包含两种介质 (nand/emmc) 的升级包，主要就是 boot0 使用的升级文件不同，将两种介质使用的所有文件全部打包进 OTA 包中，通过该辅助脚本在输入的命令后添加后缀，用来区分 nand/emmc 的介质，在策略描述文件中也包含两种介质的升级流程，不同的介质通过添加不同的后缀走不同的流程。例如：

```
get_flash_type()
{
  case "$(sed 's/ /\n/g' /proc/cmdline | awk -F= '/^root=/{print $2}')" in
  /dev/mmcblk*)
    echo emmc
    ;;

/ubi*)
    echo ubinand
    ;;
  /dev/nand*)
    echo rawnand
    ;;
#   /dev/mtdblock*)
#     echo nor
#     ;;
  esac
}

diff --git a/swupdate_cmd.sh b/swupdate_cmd.sh
index b070e8b..141d149 100755
--- a/swupdate_cmd.sh
+++ b/swupdate_cmd.sh
@@ -89,8 +89,8 @@ mkdir -p /var/lock
   echo "swu_software $swu_software" >> /tmp/swupdate_param_file
   swu_mode=$(echo "$swu_param_e" | awk -F ',' '{print $2}')
 #   echo "swu_mode: ##$swu_mode##"
-#   swu_mode=${swu_mode}_$(get_flash_type)
-#   echo "swu_mode after fix to emmc/nand: ##$swu_mode##"
+   swu_mode=${swu_mode}_$(get_flash_type)
+   echo "swu_mode after fix to emmc/nand: ##$swu_mode##"
   echo "swu_mode $swu_mode" >> /tmp/swupdate_param_file
   fw_setenv -s /tmp/swupdate_param_file
   sync
```

利用 get_flash_type()，通过 cmdline 获取当前的介质，并在输入的命令后面添加 emmc/nand 的
走不同的升级流程。例如：

```
software =
{
    version = "0.1.0";
    description = "Firmware update for xxx Project";

    stable = {

        /* ------------------------ For emmc ------------------------*/
        /* now in systemA, we need to upgrade systemB(bootB, rootfsB) */
        now_A_next_B_emmc = {
            images: (
                {
                    filename = "kernelB";
                    device = "/dev/by-name/bootB";
                    installed-directly = true;
                },
                {
                    filename = "rootfs";
                    device = "/dev/by-name/rootfsB";
                    installed-directly = true;
                },
                {
                    filename = "uboot";
                    type = "awuboot";
                },
                {
                    filename = "boot0_emmc";
                    type = "awboot0";
                }


            bootenv: (
                {
                    name = "swu_mode";
                    value = "writenv_now_A_next_B";
                }
            );
        }
.....

        now_A_next_B_ubinand = {
            images: (
                {
                    filename = "kernelB";
                    volume = "bootB"
installed-directly = true;
                },
                {
                    filename = "rootfs";
                    volume = "rootfsB"
                    installed-directly = true;
                },
                {
                    filename = "uboot";
                    type = "awuboot";
                },
                {
```

```
            filename = "boot0_nand";
            type = "awboot0";
        }
    );

    bootenv: (
        {
            name = "swu_mode";
            value = "writenv_now_A_next_B";
        }
    );
}
```

可以看出，上面为两个 AB 升级的例子，从 A 系统升级到 B 系统，因为两个升级流程使用的 boot0 文件不同，分别在 now_A_next_B 命令后添加 _emmc 和 _ubinand 后缀。

3.7 版本号

3.7.1 使用方式

在 sw-description 文件中，会配置一个版本号字符串，如：

```
software =
{
    version = "1.0.0";
    ...
}
```

如果需要在升级时检查版本号，则可使用 -N 参数，传入的参数代表小机端当前的版本号。如果不需要，则不传递 -N 参数，忽略版本号即可。

swupdate 会进行比较，如果 OTA 包中 sw-description 文件配置的版本号小于当前版本号，则不允许升级。

如何在小机端保存，获取，更新版本号，需要自定义，swupdate 没有规定具体的方式。

3.7.2 实现例子

以下需要根据实际版本要求作修改即可。

此处提供一种依赖系统 env 的实现方式供参考。

1. 初始化设备端版本号。

首先需要定义设备端的版本号存放在哪，如何获取。

本方法定义设备端的版本号保存于 env 之中，用 swu_version 记录。

则在 SDK 中，需在 env-x.x.cfg 中添加一行：

```
ersion=1.0.0
```

表示此时版本为 1.0.0，烧录固件后可执行 fw_printenv 查看。

此步骤如果不做，则第一次烧录固件后 env 中不存在 swu_version，调用 swupdate 时也无法传入获得并版本号，则第一次升级时不会检查版本。

注：这是 tina 自定义的，可修改。只要读写这个版本号的地方均配套修改即可。实际应用时版本号可以存在任意分区中，或者存放在文件系统的文件中，或者硬编码在系统和应用的二进制中，swupdate 未做限制。

2. 在 sw-description 中，设置 OTA 包版本号。

升级时如果检查到 OTA 包的 sw-description 中的 version，小于通过 -N 参数传入的版本号，则不允许升级。

```
software =
{
  version = "2.0.0";
  ...
```

例如当设备端的 env 中设置了swu_version=2.0.0，则调用 swupdate_cmd.sh 时，会自动获取此参数并在调用 swupdate 时传入-N 2.0.0。

此时若 OTA 包中定义了 version = “1.0.0”，则此时升级会降低版本号，拒绝升级。

此时若 OTA 包中定义了 version = “2.0.0”，则此次升级不会降低版本号，可以升级。

此时若 OTA 包中定义了 version = “3.0.0”，则此次升级不会降低版本号，可以升级。

注：这是 swupdate 原生的 OTA 包版本号规则，不是 tina 自定义的。

3. 更新设备端版本号。

本方式版本号定义在 env 中，则升级 kernel 和 rootfs 分区不会自动更新版本号，需要主动修改 env。

若版本号定义在env中，则不必添加这种操作，因为更新
rootfs 时版本号就自然更新了。但缺点是版本号跟 rootfs 绑定了，每次 OTA 必须升级 rootfs 才能更新版本号。

添加一个设置 version 代表 swu_version 的 env 操作，在 OTA 时自动更新版本号。

```
software =
{
  #表示这个OTA包的版本号，给swupdate读取检查的。原生规定的。
  version = "2.0.0";
  ...
```

```
        bootenv: (
          ...
          {
            #表示这个OTA包的版本号，OTA时会写入env分区，用于在下次OTA时读出作为-N参数的值。Tina自定义的。
            name = "swu_version";
            value = "2.0.0";
          }
          ...
        );
    ...
}
```

注意，这么做的话，更新版本时需要修改 env 中的版本号，以使得新的固件包拥有新的版本号，以及更新 sw-description 的两个位置，一处是最上面的 version = xxx 的版本号，一处是 bootenv 操作中的版本号，以使得 OTA 包拥有新的版本号，以及能在 OTA 时写入新版本号。

4. 读取设备端版本号传给 swupdate。

假如小机端是用脚本调用，则可用如下方式读取并传给 swupdate：

```
swu_version=$(fw_printenv -n swu_version)

swupdate ... -N $swu_version
```

更好的方式是判断非空才传入，如此可支持不在 env 中提前配置好 swu_version。

```
check_version_para=""
[ x"$swu_version" != x"" ] && {
    echo "now version is $swu_version"
k_version_para="-N $swu_version"
}
swupdate ... $check_version_para
```

注：

如果不使用版本号，则不在 env 中设置 swu_version，也不在 bootenv 中写 swu_version 即可。

如果 sw_description 中的版本号一直保持 v1.0.0，也总是能升级。

3.8 签名校验

3.8.1 检验原理

OTA 包中包含了 sw-decsription 文件和各个具体的镜像，如 kernel，rootfs。

如果对整个 OTA 包进行完整校验，则会对流式升级造成影响，要求必须把整个 OTA 包下载下来，才能判断出校验是否通过。

为了避免上述问题，swupdate 的校验是分镜像的，首先从 OTA 包最前面取出两个文件，即 sw-description 和使用传入的公钥校验 sw-description，校验通过则认为 sw-description 可信，则说明其中描述的 image 和 sha256 也是可信的。

后续无需再使用公钥，直接校验每个镜像的 sha256 即可。因此可以逐个镜像处理，无需全部下载完毕再处理。

3.8.2 配置

swupdate 支持使用签名校验功能，需要在编译时选中对应功能。

出于安全考虑，一旦使能了校验，则 swupdate 不再支持不使用签名的更新调用。

```
在openwrt中配置主系统：

make menuconfig --->
  Allwinner  --->
    System  --->
      <*> swupdate  --->
        SSL implementation to use (None)  ---> (OpenSSL)
        [*]   Enable verification of signed images
        Signature verification algorithm (RSA PKCS#1.5)  --->(选择校验算法，此处以RSA为例)

注意，recovery系统也需要对应进行配置，即：

make ota_menuconfig ---> ...(重复以上配置)

buildroot环境中配置主系统：
./build.sh buildroot_menuconfig
Target packages --->
  System tools --->
    [*] swupdate  --->
      SSL implementation to use (None)  ---> (OpenSSL)
      [*]   Enable verification of signed images
      Signature verification algorithm (None) ---> (RSA PKCS#1.5) (选择校验算法，此处以RSA为例)

recovery系统：
在buildroot/buildroot-xx版本/目录下执行make xxx_defconfig(recovery对应的defconfig配置)。

然后执行make menuconfig 更改配置，最后执行 make savedefconfig 保存配置。
```

3 使用方法

在 PC 端使用私钥签名 OTA 包。

在小机端调用 swupdate 时，使用-k 参数传入公钥。

3.8.4 初始化 key

Tina 封装了一条命令，生成默认的密钥对。执行：

```
swupdate_init_key
```

3.8.4.1 openwrt 环境

执行后会使用默认密码生成密钥对并拷贝到指定目录：

```
密码/私钥/公钥：
password:openwrt/target/{IC}/{IC}-{BOARD}/swupdate/swupdate_priv.password
e key:openwrt/target/{IC}/{IC}-{BOARD}/swupdate/swupdate_priv.pem
public key:openwrt/target/{IC}/{IC}-{BOARD}/swupdate/swupdate_public.pem
公钥拷贝到base-files中，供使用procd-init的方案使用
public key:openwrt/target/{IC}/{IC}-{BOARD}/base-files/swupdate_public.pem
公钥拷贝到busybox-init-base-files中，供使用busybox-init的方案使用
public key:openwrt/target/{IC}/{IC}-{BOARD}/busybox-init-base-files/swupdate_public.pem
```

3.8.4.2 buildroot 环境

执行后会使用默认密码生成密钥对并拷贝到指定目录：

```
密码/私钥/公钥：
password:target/{IC}/buildroot/{BOARD}/swupdate/swupdate_priv.password
private key:target/{IC}/buildroot/{BOARD}/swupdate/swupdate_priv.pem
public key:target/{IC}/buildroot/{BOARD}/swupdate/swupdate_public.pem
公钥拷贝到busybox-init-base-files中，
public key:buildroot/config/buildroot/allwinner/system/busybox-init-base-files/etc/swupdate_public.pem
```

此步骤仅为方便调试使用，只需要做一次。

用户也可使用自己的密码自行生成密钥，生成密钥的具体命令可参考 build/buildbase.sh 中 swupdate_init_key 的实现：

```
local password="swupdate";
echo "$password" > swupdate_priv.password;
echo "-------------------- init priv key --------------------";
openssl genrsa -aes256 -passout file:swupdate_priv.password -out swupdate_priv.pem;
echo "-------------------- init public key --------------------";
openssl rsa -in swupdate_priv.pem -passin file:swupdate_priv.password -out swupdate_public.pem -outform PEM -
    pubout;
```

如上，生成的密钥一般放到方案目录下，即可在打包 OTA 包时自动使用。

主要就是调用 openssl 生成，私钥拷贝到 SDK 指定目录，供生成 OTA 包时使用。公钥放到设备端，供设备端执行 OTA 时使用。

密钥的作用是校验 OTA 包，意味着拿到密钥的人即可生成可通过校验的 OTA 包，因此正式产品中一般密钥只掌握在少数人手中，并采取适当措施避免泄漏或丢失。

一种可参考的实践方式是，正式密钥做好备份，并仅部署在有权限管控的服务器上，只能代码入
包，自动编译生成无法拿到密钥自行本地生成用于正式产品的

目前脚本支持自动在生成 OTA 包时，更新 sha256 的值。但需要在 sw-description 中，手工添加：

```
sha256 = @文件名
```

如：

```
$ git diff sw-description
diff --git a/sw-description b/sw-description
index ed04b64..467ac3b 100644
--- a/sw-description
+++ b/sw-description
@@ -9,14 +9,17 @@ software =
          {
filename = "boot_initramfs_recovery.img"
            device = "/dev/by-name/recovery";
+            sha256 = "@boot_initramfs_recovery.img"
          },
          {
            filename = "boot_package.fex"
            type = "awuboot";
+            sha256 = "@boot_package.fex"
          },
          {
            filename = "boot0_nand.fex"
            type = "awboot0";
+            sha256 = "@boot0_nand.fex"
          }

        );
4,10 +37,12 @@ software =
          {
            filename = "boot.img";
            device = "/dev/by-name/boot";
+            sha256 = "@boot.img"
          },
          {
            filename = "rootfs.img";
            device = "/dev/by-name/rootfs";
+            sha256 = "@rootfs.img"
          }
        );
        bootenv: (
```

在打包 OTA 包时，脚本自动算出 sha256 的值，并替换到上述位置，再完成 OTA 包的生成。

：

```
build/swupdate/sw-description-sign
build/swupdate/sw-subimgs-sign.cfg

注：
调用swupdate_pack_swu 则会使用sw-subimgs.cfg，其中默认指定了使用sw-description做为最终的sw-description。
调用swupdate_pack_swu -sign则会使用sw-subimgs-sign.cfg，其中默认指定了使用sw-description-sign做为最终的sw-
      description。
即关键还是看使用哪份sw-subimgs.cfg，以及sw-subimgs.cfg中如何指定。
```

3.8.5 添加 sw-description.sig

签名的 OTA 包，需要生成签名文件 sw-description.sig，并使其在 OTA 包中，紧随在 sw-description 后面。

目前脚本中自动处理。

3.8.6 生成 OTA 包

方法不变，脚本中会检测 defconfig 的配置，并自动完成签名等动作。

3.8.7 将公钥放置到小机端

目前脚本中生成 key 的时候，自动拷贝了。如需手工处理，可参考如下方式。

3.8.7.1 openwrt 环境

对于 procd-init：

```
cplat
mkdir -p ./base-files
update_public.pem ./base-files/etc/
```

对于 busybox-init

```
cplat
mkdir -p ./busybox-init-base-files/
cp swupdate_public.pem ./busybox-init-base-files/etc/
```

3.8.7.2 buildroot 环境

对于 busybox-init

```
drootconfig/buildroot/allwinner/system/busybox-init-base-files/etc/
cp <SDK>/target/{IC}/buildroot/{BOARD}/swupdate/swupdate_public.pem ./
```

3.8.8 在小机端调用

在原本的命令基础上，加上 -k /etc/swupdate_public.pem 即可，如：

```
swupdate_cmd.sh -v -i /mnt/UDISK/xxx.swu -k /etc/swupdate_public.pem -e stable,upgrade_recovery
```

压缩

swupdate 支持对镜像先解压，再写入目标位置，当前支持 gzip 和 zstd 两种压缩算法。

3.9.1 配置

使用 gzip 压缩无需配置，使用 zstd 则需选上

```
make menuconfig --> Allwinner  ---> <*> swupdate --> [*]   Zstd compression support
```

2 生成压缩镜像

如果希望每次打包固件自动生成，则可修改 build/pack，在 function do_pack_linux() 函数的最后加上压缩的动作。

```
生成gz镜像:
gzip -k -f boot.fex
gzip -k -f rootfs.fex
gzip -k -f recovery.fex #如果使用AB方案，则无需recovery

生成lzma镜像:
zstd -k -f boot.fex -T0
zstd -k -f rootfs.fex -T0
zstd -k -f recovery.fex -T0
```

对于 lzma，若需要调整压缩率，可指定 0-19 的数字 (数字越大，压缩率越高，耗时越长)，如

```
zstd -19 -k -f boot.fex -T0
zstd -19 -k -f rootfs.fex -T0
zstd -19 -k -f recovery.fex -T0
```

如果不希望每次打包固件多耗时间，则需自行在生成 OTA 包之前，使用上述命令制作好压缩镜像。原始的 boot.fex，rootfs.fex，recovery.fex 在 out/{IC}/{BOARD}/openwrt(或 buildroot)/目录下。

3.9.3 sw-subimgs.cfg 配置压缩镜像

以 rootfs 为例，将原本未压缩的版本

```
${LICHEE_PACK_OUT_DIR}/rootfs.fex:rootfs
```

改成压缩的

```
${LICHEE_PACK_OUT_DIR}/rootfs.fex.gz:rootfs.gz
```

或

```
${LICHEE_PACK_OUT_DIR}/rootfs.fex.zst:rootfs.zst
```

注：为了方便差分包处理或者压缩镜像后缀名，并自动解压。

3.9.4 sw-description 配置压缩镜像

以 rootfs 为例，将原本未压缩的版本

```
        {
          filename = "rootfs";
          device = "/dev/by-name/rootfs";
          installed-directly = true;
  sha256 = "@rootfs";
},
```

换成

```
        {
          filename = "rootfs.gz";
          device = "/dev/by-name/rootfs";
          installed-directly = true;
          sha256 = "@rootfs.gz";
          compressed = "zlib";
        },
```

或

```
        {
          filename = "rootfs.zst";
          device = "/dev/by-name/rootfsB";
          installed-directly = true;
          sha256 = "@rootfs.zst";
          compressed = "zstd";
        },
```

3.10 调用 OTA

swupdate_cmd.sh , 用于给 swupdate 传入相关参数，切换更新状态，以及不断重试。

3.10.1 进度条

swupdate 提供了 progress 程序，该程序会在后台运行，从 socket 获取进度信息，打印进度条到串口。

具体方案可参考其实现 (在 swupdate 源码中搜索 progress)，自行在应用中获取进度，通过屏幕等其他方式进行指示。

3.10.2 重启

1. 调用 swupdate 的时候加上 -p reboot，则 swupdate 更新完毕后，会执行 reboot。

2.swupdate_cmd.sh 支持检测 env 中的 swu_next 变量，如果为 reboot，则脚本中执行 reboot。可在 sw-description 中设置此变量。

3. 如果调用 progress 的时候加上 -r 参数，则 progress 会在检测到更新完成后，执行 reboot。

3.10.3 本地升级示例

的目录下包推送到U盘下，如放在

PC 端执行：

```
adb push xxx.swu /mnt/UDISK
```

小机端执行 (不带签名校验版本)：

```
swupdate_cmd.sh -i /mnt/UDISK/xxx.swu -e stable,upgrade_recovery
```

小机端执行 (带签名校验版本)：

```
swupdate_cmd.sh -i /mnt/UDISK/xxx.swu -k /etc/swupdate_public.pem -e stable,upgrade_recovery
```

.4 网络升级示例

启动服务器：

```
linux系统：
进入OTA包所在文件夹，如：
cd swupdate/

sudo python -m  SimpleHTTPServer 80 #启动一个服务器
或
sudo python -m http.server 80

windows系统：
打开OTA包所在文件夹；然后按住Shift键，并点击鼠标右键，
择在此处打开Powershell
sudo python -m http.server 80
```

小机端命令，使用-d -uxxx，xxx 为 url。

例如 (不带签名校验版本)：

```
swupdate_cmd.sh -d -uhttp://192.168.35.112/xxx.swu -e stable,upgrade_recovery
```

例如 (带签名校验版本)：

```
swupdate_cmd.sh -d -uhttp://192.168.35.112/xxx.swu  -k /etc/swupdate_public.pem -e stable,upgrade_recovery
```

依赖外部程序，OTA 包存放在网络服务器。

3.10.5 错误处理

如何判断 swupdate 升级出错?

1. 调用 swupdate 时获得并判断返回值是否为 0。
2. 读取 env 变量 recovery_status。根据 swupdate 官方文档，swupdate 开始执行时，会设置 recovery_status=“progress”，升级完成会清除这个变量，升级失败则设置 recovery_status=“failed”

3.11 裁剪

swupdate 本身是可配置的, 不需要某些功能时，可将其裁剪掉。

```
    <*> swupdate  --->
```

例如，不需要使用 swupdate 来从网络下载 OTA 包的话，则可将

```
[*]   Enable image downloading
```

不需要更新 boot0/uboot 的话，则将

```
Image Handlers  --->

  [*] allwinner boot0/uboot
```

取消掉

3.12 调试

3.12.1 直接调用 swupdate

目前 swupdate_cmd.sh 主要有两个作用：

1. 自启动，无限重试。

2. 在主系统和 recovery 系统中，传入不同的 -e 参数给 swupdate。

出问题时，可以不使用 swupdate_cmd.sh，手工直接调用 swupdate，在后面加上合适的-e 参
察输出 log。

如：

```
swupdate -v -i  xxx.swu  -e stable,boot

swupdate -v -i  xxx.swu  -e stable,recovery
```

3.12.2 手工切换系统

按上述 env 的配置，启动的系统，是由 boot_partition 变量控制的。

/lock

切换到主系统：

```
fw_setenv boot_partition boot

reboot
```

切换到 recovery 系统：

```
fw_setenv boot_partition recovery

reboot
```

观察当前变量：

```
fw_printenv
```

3.12.3 更新 boot0/uboot

目前更新 boot0，uboot 实际功能是由另一个软件包 ota-burnboot 完成的，swupdate 只是准备数据，并调用 ota-burnboot 提供的动态库。

如果更新失败，先尝试手工使用 ota-burnboot0 xxx 和 ota-burnuboot xxx 能否正常更新。以确定是 ota-burnboot 的问题，还是 swupdate 的问题。

3.12.4 解压 OTA 包

swupdate 的 OTA 包，本质上是一个 cpio 格式的包，直接使用通用的 cpio 解包命令即可。

```
cpio -idv < xxx.swu
```

3.12.5 校验 OTA 包

当使能了签名校验，会对 sw-description 签名生成 sw-description.sig，如果校验失败，可以在 PC 端手工验证下：

使用 RSA 时，build/envsetup.sh 中调用的命令是：

```
openssl dgst -sha256 -sign "$priv_key_file" $password_para "$SWU_DIR/sw-description" > "$SWU_DIR/sw-description.
    sig"
```

则对应的密钥验证签名命令为：

```
openssl dgst -prverify swupdate_priv.pem -sha256 -signature sw-description.sig sw-description
```

证签名的命令为：

```
openssl dgst -verify swupdate_public.pem -sha256 -signature sw-description.sig sw-description
```

3.13 测试固件示例

3.13.1 生成方式

一般而言，测试需要两个有差异的 OTA 包，如，uboot 和 kernel 的 log 有差异，rootfs 的文件有差异。

测试升级。

3.13.1.1 准备工作

如果需要网络更新，OTA 不负责联网，所以需要选上 wifimanager。

```
在openwrt中：

Allwinner
---> Wireless
   ---> -*- wifimanager-v2.0..... Tina wifimanager-v2.0  --->
     -*-  wifimanager-v2.0-lib..... Tina wifimanager-v2.0 lib
     <*>    wifimanager-v2.0-demo..... Tina wifimanager-v2.0 app demo
   -*- wirelesscommon..... Allwinner Wi-Fi/BT Public lib

在buildroot环境中：

Target packages --->
  allwinner platform private package select --->
    wireless --->
      [*] wifimanager-v2.0
      [*]  wifimanager-v2.0-lib
      [*]    Tina wifimanager-v2.0-demo
      -*- wireless_common
```

具体的模组需要参考实际情况配置。

3.13.1.2 生成固件 1 和 OTA 包 1

重新编译 boot，使得编译时间更新：

```
mboot
```

创建文件以标记 rootfs;

重新编译 recovery 系统：

```
swupdate_make_recovery_img
```

重新编译打包，使得编译时间更新：

```
mp -j32
```

生成 OTA 包 1：

```
swupdate_pack_swu
```

将得到产物重命名为 OTA1.swu

3.13.1.3 生成固件 2 和 OTA 包 2

重新编译 uboot，使得编译时间更新：

```
muboot
```

创建文件以标记 rootfs;

重新编译 recovery 系统：

```
swupdate_make_recovery_img
```

重新编译打包，使得编译时间更新：

```
mp -j32
```

TA 包 2：

```
swupdate_pack_swu
```

将得到产物重命名为 OTA2.swu

3.13.2 使用方式

任意选择一个 OTA 固件烧录后，可在此基础上进行本地升级或网络升级。

3.13.2.1 本地升级方式

PC 端执行：

```
adb push OTA1.swu /mnt/UDISK
```

小机端执行：

```
swupdate_cmd.sh  -i /mnt/UDISK/OTA1.swu
```

3.13.2.2 网络升级方式

搭建服务器：

```
sudo python -m http.server 80
```

小机端联网：

```
wifi -c 账号 密码
```

小机端执行：

```
swupdate_cmd.sh -d -uhttp://192.168.xxx.xxx/OTA1.swu
```

注明：启动后会自动联网，连网后等待 OTA 后台脚本尝试更新。中途掉电重启后，正常会在启动后几十秒内，成功联网并开始继续更新。

3.13.2.3 升级过程

升级过程会进行两次重启。具体的：

（1）升级 recovery 分区 (boot_initramfs_recovery.img)，uboot(boot_package.fex)，boot0(boot0_nand.fex)。

（2）重启，进入 recovery 系统。

（3）升级内核 (boot，img) 和 rootfs(rootfs.img)。

重启，进入主系统，升级完成。

3.13.2.4 判断升级

从 log 中的时间和 rootfs 文件可以判断当前运行的版本。

4 升级定制分区

如果定制了一个分区，并需要对此分区进行 OTA，则需要：

1. 确认需不需要备份。
2. 将分区文件加入 OTA 包。
3. 确定升级策略，在 sw-description 中增加对此分区的处理。

3.14.1 备份

在 OTA 过程随时可能掉电，如果掉电时正在升级分区 mypart，则重启后 mypart 的数据是不完整的。

是否需要备份主要取决于 mypart 所保存的文件是否影响继续进行 OTA。

例如 mypart 中保存的是开机音乐，被损坏的结果只是开机无声音，但开机后能正常继续 OTA，则无需备份。

例如 mypart 中保存的是应用，且 OTA 中途掉电后重启，需要应用负责联网从网络重新下载 OTA 包，则需要备份，否则无法继续 OTA，设备就无法恢复了。

例如 mypart 中保存的是 dsp，启动过程没有有效的 dsp 镜像会无法启动，则需要备份，否则无法

备就无法恢复OTA。

3.14.2 无需备份

假设分区表中定义了：

```
[partition]
    name        = mypart
    size        = 512
    downloadfile = "mypart.fex"
    user_type    = 0x8000
```

在：

```
my_dir/mypart.fex
```

则在 sw-subimg.cfg 中增加一行（注意要放到 sw-description 之后，因为 sw-description 必须是第一个文件），将 mypart.fex 打包到 OTA 包中，重命名为 mypart：

```
my_dir/mypart.fex:mypart
```

在 sw-description 中增加升级动作的定义。例如可以在升级 rootfs 之后，升级该分区：

```
software =
{
  ...
  stable = {
    ...
    upgrade_kernel = {
      images: (
        ...
        {
          filename = "rootfs";
          device = "/dev/by-name/rootfs";
          installed-directly = true;
        },
        {
          filename = "mypart";
          device = "/dev/by-name/mypart";
          installed-directly = true;

  ...
      );
    ...
}
```

3.14.3 需要备份

在分区表中定义好两个分区，这样升级过程掉电，总有一份是完整的。

```
[partition]
  name       = mypart
  size       = 512
  downloadfile = "mypart.fex"


[partition]
  name       = mypart-r
  size       = 512
  downloadfile = "mypart.fex"
  user_type    = 0x8000
_type = 0x8000
```

文件放在：

```
my_dir/mypart.fex
```

则在 sw-subimg.cfg 中增加一行（注意要放到 sw-description 之后，因为 sw-description 必须是第一个文件），将 mypart.fex 打包到 OTA 包中，重命名为 mypart：

```
r/mypart.fex:mypart
```

有两个分区，则需要条件决定使用哪一个，可考虑在 env 中定义一个变量：

```
mypart_partition=mypart
```

使用 mypart 时，要先读取 env 的 mypart_partition 的值来决定要使用哪个分区。

在 sw-description 中，定义好要升级的分区和 bootenv，保证每次升级那个未在使用的分区。这样即使掉电也无妨。

```
software =
{
  ...
  stable = {
    upgrade_recovery = {
      images:
        {
          filename = "recovery";
          device = "/dev/by-name/recovery";
          installed-directly = true;
        },
    /* 更新mypart-r， 此时掉电，mypart分区是完整的 */
        {
          filename = "mypart-r";
          device = "/dev/by-name/mypart-r";
          installed-directly = true;
        }

      );
      bootenv: (
        {
          name = "swu_mode";
          value = "upgrade_kernel";
        },
        {
          name = "boot_partition";
          value = "recovery";
        },
    /* 设置这个env，指示下次启动，即启动到recovery分区时，配套使用mypart-r */
        {
          name = "mypart_partition";
          value = "mypart-r";
        },

          name = "swu_next";
          value = "reboot";
        }
      );
    };

    upgrade_kernel = {
      images: (
        {
          filename = "kernel";
          device = "/dev/by-name/boot";
          installed-directly = true;
        },
        {
          filename = "rootfs";
          device = "/dev/by-name/rootfs";
          installed-directly = true;
        },
    /* 更新mypart， 此时掉电，mypart分区还是完整的 */
        {
          filename = "mypart-r";
          device = "/dev/by-name/mypart-r";
          installed-directly = true;
        }
      );
      bootenv: (
```

{

```
        {
          name = "swu_mode";
          value = "upgrade_usr";
        },
    /* 设置这个env，指示下次启动，即启动到正常系统时，使用mypart */
        {
          name = "mypart_partition";
          value = "mypart";
        },
        {
          name = "boot_partition";
          value = "boot";
        }
      );
    };
...
```

3.15 handler 说明

此处只对一些 handler 做简介。

具体的 handler 的用法，请参考 swupdate 官方文档说明。

3.15.1 awboot

1.1 nand/emmc

全志拓展的 handler，用于支持升级全志 boot0 和 uboot，实质上会调用外部的 ota-burnboot 包完成升级。

目前只支持 nand/emmc，详见 ota-burnboot 章节。

使用方式：

选上 handler 支持：

```
    <*> swupdate --->
      Image Handlers -->
*] allwinner boot0/uboot
```

注意，recovery 系统也需要对应进行配置;

在 sw-description 中指定 type 即可：

```
{
  filename = "uboot"; /* 源文件是OTA包中的uboot文件 */
  type = "awuboot"; /* type为awuboot，则swupdate会调用对应的handler做处理 */
},
{
  filename = "boot0"; /* 源文件是OTA包中的boot0文件 */
```

```
    type = "awboot0"; /* type为awuboot，则swupdate会调用对应的handler做处理 */
}
```

3.15.1.2 nor

对于 nor 方案，升级 boot0/uboot 有掉电风险。如确需升级，可直接配置合适偏移即可，不使用 awboot handler。

例如已知 boot0 存放在偏移为 0 处，uboot 存放在偏移为 24k 处，则配置：

```
{
    filename = "boot0";
    device = "/dev/mtdblock0";
    installed-directly = true;
},
{
    filename = "uboot";
    device = "/dev/mtdblock0";
    offset = "24k"
    installed-directly = true;
}
```

3.15.2 readback

用于支持在 sw-description 中配置 sha256，在升级后读出数据进行校验。

AB 系统升级时在应用差分包后读出校验，以确认差分得到的结果是对的，再切换系统。

需选上对应 handler。

```
openwrt环境：

make menuconfig/make ota_menuconfig
--> Allwinner
    System  --->
        -->  swupdate
            --> <*>   Allow to add sha256 hash to each image
            --> Image Handlers
                --> [*] readback



./build.sh buildroot_menuconfig
--> Target packages
    --> System tools
        --> swupdate
            --> [*]   Allow to add sha256 hash to each image
            Image Handlers  --->
                --> [*] readback

上面为主系统的配置方式，recovery系统配置方案请参考recovery配置。
```

3.15.2.1 示例

```
build/swupdate/sw-description-readback
build/swupdate/sw-subimgs-readback.cfg
```

3.15.3 ubi

用于 ubi 方案。

需先选上 MTD 支持：

```
openwrt/buildroot环境：

make menuconfig
--> swupdate
 --> Swupdate Settings
  ---> General Configuration
   ---> [*] MTD support

注意：buildroot系统需要选中依赖的MTD包：

Target packages --->
  Filesystem and flash utilities --->
    [*] mtd, jffs2 and ubi/ubifs tools
```

再选上对应 handler：

```
wrt/buildroot环境：


make menuconfig
--> swupdate
 --> Image Handlers
  --> [*] ubivol
```

3.15.3.1 示例

请参考：

```
build/swupdate/sw-subimgs-ubi.cfg
build/swupdate/sw-description-ubi
```

最后使用 swupdate_pack_swu -ubi 命令打包 OTA 包

3.15.4 rdiff

rdiff handle 用于差分包升级。

需选上对应 handler：

```
openwrt/buildroot系统：

make menuconfig
--> swupdate
 --> Image Handlers
  --> [*] rdiff

#若使用AB系统方案，则无recovery系统，则make ota_menuconfig的配置可不做。

在buildroot系统中需要选中对应的lib：

Target packages --->
  Libraries --->
    Networking --->
      [*] librsync
```

4.1 特性

在 https://librsync.github.io/page_rdiff.html 中指出

```
rdiff cannot update files in place: the output file must not be the same as the input file.
```

即不支持原地更新，即应用差分包将 A0 更新成 A1，需要有足够空间存储 A0 和 A1，不能直接对 A0 进行改动。

```
rdiff does not currently check that the delta is being applied to the correct file. If a delta is applied to the wrong basis file,
      the results will be garbage.
```

不校验原文件，如果将差分包应用于错误的文件，则会得到无效的输出文件，但不会报错。

```
The basis file must allow random access. This means it must be a regular file rather than a pipe or socket.
```

原文件必须支持随机访问，因此不能从管道或 socket 中获取原文件。

更多介绍请参考：https://librsync.github.io/

3.15.4.2 示例

首先需要将方案修改为 AB 系统的方案，请参考上文的 “AB 系统方案举例”。

date 的配置文件请参考：

```
build/swupdate/sw-description-ab-rdiff
build/swupdate/sw-subimgs-ab-rdiff.cfg
```

差分文件的生成使用 rdiff：

```
$ rdiff -h
Usage: rdiff [OPTIONS] signature [BASIS [SIGNATURE]]
       [OPTIONS] delta SIGNATURE [NEWFILE [DELTA]]
       [OPTIONS] patch BASIS [DELTA [NEWFILE]]
```

tina 封装了一条命令，用于解出两个 swu 包并生成各个子镜像的差分文件。

一种参考的差分包生成方式是，先按普通的升级方式生成整包。

固件对应V1.swu，则可使用：

```
swupdate_make_delta V1.swu V2.swu
```

生成差分文件。

再将生成的差分文件，用于生成差分的 OTA 包。

例如：

```
make && pack  #生成V1固件
swupdate_pack_swu -ab  #得到xxx-ab.swu
cp xxx-ab.swu V1.swu #保存V1的OTA包


make && pack  #生成V2固件
swupdate_pack_swu -ab  #得到xxx-ab.swu
cp xxx-ab.swu V2.swu #保存V2的OTA包

swupdate_make_delta V1.swu V2.swu #生成差分镜像
swupdate_pack_swu -ab-rdiff  #生成差分OTA包，这一步用到了刚刚生成的差分镜像
```

3.15.4.3 开销问题

差分升级的主要好处在于节省传输的文件大小。而不是节省 ram 和 rom 的占用。

在进行差分升级时，需要使用版本匹配的旧版本镜像，加上差分包，生成新版本镜像。

rsync 不支持原地更新，必须有额外的空间保存新生成的镜像。

从掉电安全的角度考虑，在新版本镜像完整保存到 flash 之前，旧版本镜像不能破坏，否则一旦中途掉电，将无法再次使用旧镜像 + 差分包生成新镜像，只能联网下载完整的 OTA 包。

以上限制，导致 flash 必须在旧镜像之外，有足够 flash 空间用于存放新的镜像。

对于 recovery 方案，原本的：

```
从OTA包获取新recovery写入recovery分区 -->
reboot -->
从OTA包获取新kernel写入boot分区 -->
```

就需要变成：

```
从OTA包获取recovery差分包，读recovery分区，生成新recovery暂存到文件系统中 -->
从文件系统获取新recovery写入recovery分区 -->
reboot -->
...
```

总体较为麻烦，且需要文件系统足够大。

对于 AB 方案，原本的：

```
从OTA包获取新kernel写入bootB分区
从OTA包获取新rootfs写入rootfsB分区 -->
reboot
```

就需要变成：

```
从OTA包获取kernel差分包，读出bootA分区，合并生成新kernel写入bootB分区 -->
从OTA包获取rootfs差分包，读出rootfsA分区，合并生成新rootfs写入bootB分区 -->
reboot
```

不需要依赖额外的文件系统空间。

因此，若希望节省 ram/rom 占用，差分包并非解决的办法。若希望使用差分包，建议配合 AB 系统使用。

3.15.4.4 管理问题

差分包的一个麻烦问题在于，差分包必须跟设备端的版本匹配。而出厂之后的设备，可能存在多种版本。

例如出厂为 V1，当前最新为 V4，则设备可能处于 V1，V2，V3。此时若使用整包升级，则无需区分。

若使用差分升级，一种策略是为每个旧版本生成一个差分包，则需要制作三个差分包 V1_4，V2_4，V3_4。在 OTA 时先判断设备端和云端版本，再使用对应的差分包。

另一种策略是，只为上一个版本生成差分包 V3_4，并额外准备一个整包。在 OTA 时先判断设备端版本和云端版本，若可相差一个版本则使用差分包，若跨版本则使用整包。

不管哪一种，都需要应用做出额外的判断。这一点需要主应用和云端服务器做好处理。

3.15.4.5 校验问题

目前社区支持的 rdiff 本身并不包含较好的校验机制，即应用一个版本不对的差分包，也能跑完升级流程。这样一旦出错，就会导致机器变砖。

一种可参考的做法是 readback 功能。即应用差分包，写入目标分区之后，将更新后的目标分区数据读出，校验其 sha256 是否符合预期，校验成功才切换系统，校验失败则报错。

可参考

```
build/swupdate/sw-description-ab-rdiff-sign
build/swupdate/sw-subimgs-ab-rdiff-sign.cfg
```

其中增加了 readback 的处理。

具体的，sw-subimgs-xxx.cfg 中可以配置 swota_copy_file_list，指定一些文件只拷贝到 swupdate 目录，不打包到最终的 OTA 包中。因为在这个场景下，我们需要原始的 boot0，kernel，rootfs 等文件来计算 sha256，但并不需要将其加入最终的 OTA 包中。

3.15.4.6 跟 ubi 的配合问题

swupdate 官方目前没有支持 rdiff 用于 ubi 卷。

目前采用增加预处理脚本的方式来兼容，即在执行 rdiff handler 之前，先调用脚本使用 ubiupdatevol 创建一个可用于升级目标 ubi 卷的 fifo。随后 rdiff handler 即可将此 fifo 当作目标裸设备，无需特殊处理 ubi 卷。在后处理脚本中，再通过填 0 的方式，结束所有的 ubiupdatevol。

参考如下文件夹中的配置和脚本：

```
openwrt查看目录：openwrt/target/{IC}/{IC}-{BOARD}/swupdate
buildroot查看目录：target/{IC}/buildroot/{BOARD}/swupdate/
```

前提仍然是配置好 AB 系统，使用方式：

```
# 编译系统，得到要烧录的固件，记为V1
make && pack

# 生成OTA包，此OTA包的各个分区镜像跟V1固件的镜像一致
swupdate_pack_swu -ab
cp xxx-ab.swu ab_V1.swu

# 进行修改，编译出V2的系统


# 生成OTA包，此OTA包的各个分区镜像跟V2固件的镜像一致
swupdate_pack_swu -ab
cp xxx-ab.swu ab_V2.swu

# 此时 ab_V2.swu 可用于正常的OTA升级
# 生成各个镜像的差分文件
swupdate_make_delta ./ab_V1.swu ./ab_V2.swu

# 基于上一步得到的差分文件，生成差分OTA包
swupdate_pack_swu -ab-rdiff
```

&& pack

和 postinstall 脚本

功能是在 OTA 升级过程中执行 OTA 包里脚本。

pretinstall：表示升级分区镜像前执行的脚本。但在 swupdate_cmd.sh 之后

postinstall：表示升级分区镜像后执行的脚本。

配置，默认已经选上：

```
make menuconfig
--> swupdate
 --> Swupdate Settings
  ---> General Configuration
   ---> [*] enable pre and postinstall scripts
```

再选上对应 handler，默认已经选上：

```
make menuconfig
-->  swupdate
 --> Image Handlers
  --> [*] shellscript
```

3.15.5.1 示例

以非 ubi 格式，recovery 系统配置文件为例。

1. 在 openwrt 环境，在 openwrt/target/xxx/xxx/swupdate 中创建。

2. 在 buildroot 环境，在 target/{IC}/buildroot/{BOARD}/swupdate 目录下创建。

3. 修改 swupdate/sw-subimgs.cfg 文件，以 openwrt 为例

```
swota_file_list=(
openwrt/target/${LICHEE_IC}/${LICHEE_IC}-${LICHEE_BOARD}/swupdate/sw-description
${LICHEE_PLAT_OUT}/boot0.img:boot0
${LICHEE_PLAT_OUT}/uboot.img:uboot
${LICHEE_PACK_OUT_DIR}/boot.fex:kernel
${LICHEE_PACK_OUT_DIR}/rootfs.fex:rootfs
openwrt/target/${LICHEE_IC}/${LICHEE_IC}-${LICHEE_BOARD}/swupdate/preinstall_A.sh
openwrt/target/${LICHEE_IC}/${LICHEE_IC}-${LICHEE_BOARD}/swupdate/postinstall_A.sh
)
```

4. 修改 swupdate/sw-description 文件，将这一段复制到对应的节点。

```
        scripts: (
          {
            filename = "preinstall_A.sh";
            type = "preinstall";
          },
          {
            filename = "postinstall_A.sh";
            type = "postinstall";
          }
```

比如想在升级 recovery 系统前后执行脚本

```
 stable = {

     /* upgrade recovery,uboot,boot0 ==> change swu_mode,boot_partition ==> reboot */
     upgrade_recovery = {
       /* upgrade recovery */
       images: (
         {
           filename = "recovery";
```

```
            device = "/dev/by-name/recovery";
            installed-directly = true;
          },
          {
            filename = "uboot";
            type = "awuboot";
          },
          {
            filename = "boot0";
            type = "awboot0";
          }
        );
        scripts: (
          {
            filename = "preinstall_A.sh";
            type = "preinstall";
          },
          {
            filename = "postinstall_A.sh";
            type = "postinstall";
          }
        );
        /* change swu_mode to upgrade_kernel,boot_partition to recovery & reboot*/
        bootenv: (
          {
            name = "swu_mode";
            value = "upgrade_kernel";
          },
          {
            name = "boot_partition";
            value = "recovery";
          },
          {
            name = "swu_next";
            value = "reboot";
          }
        );
      };
```

3.16 升级开机 logo

开机 logo 对应的分区是 boot-resource，所以只需要升级此分区即可。

3.16.1 配置文件 sw-subimgs

创建 openwrt/target/{IC}/{IC}-{BOARD}/swupdate/sw-subimgs-logo.cfg 文件，用来把 logo 镜像打包到 OTA 包里，内容如下：

```
swota_file_list=(
openwrt/target/${LICHEE_IC}/${LICHEE_IC}-${LICHEE_BOARD}/swupdate/sw-description-logo:sw-description
${LICHEE_PACK_OUT_DIR}/boot-resource.fex:new_logo
)
```

3.16.2 配置 sw-description 文件

创建 swupdate/sw-description-logo 文件，表示升级 logo 后重启，内容如下：

```
software =
{
    version = "0.1.0";
    description = "Firmware update for Tina Project";
    stable = {
        /* upgrade boot-resource for new logo ==> reboot */
        upgrade_logo = {
            /* upgrade boot-resource */
            images: (
                {
                    filename = "new_logo";
                    device = "/dev/by-name/boot-resource";
                    installed-directly = true;
                }
            );
            /* clean & reboot*/
            bootenv: (
                {
                    name = "swu_param";
                    value = "";
                },
                {
                    name = "swu_software";
                    value = "";
                },
                {
                    name = "swu_mode";
value = "";
                },
                {
                    name = "swu_next";
                    value = "reboot";
                }
            );
        };
    };
}
```

3.16.3 使用

端执行命令：swupdate_打包_pack_swu -logo

2.PC 通过 adb push 把 OTA 推到设备端/mnt/UDISK 目录。

3. 设备端执行升级命令：swupdate_cmd.sh -i /mnt/UDISK/tina-xxx.swu -e stable,upgrade_logo。

YT快起方案

3.17.1 功能介绍

目前对快起方案的 OTA，制定了一套相对完整的升级流程，为了考虑平台兼容性，快起方案的 OTA 依然使用 swupdate 框架，并且使用方法上没有进行修改，只需要注意几个配置，会在下面详细介绍。

3.17.2 系统介绍

[illegible] 中实现。

所以切换系统的方式没有更改，还是通过 env 中的 boot_partition 以及 root_partition 变量切换。

3.17.2.1 AB 系统

除了与正常的使用：修改分区表、env 外，还需要参考以下内容修改。

目前 AB 系统使用了两个内核镜像区分，A 系统使用正常编译的 boot.img，B 系统使用额外编译的 bootB.img。

内核的编译命令：

```
swupdate_make_kernelB_img
```

拷贝一份 board.dts 改名为 board-B.dts，与正常的编译流程相同，但是使用的设备树为 board-B.dts。生成的 boot.img 拷贝为 bootB.img，作为 B 系统的内核镜像。

目前将设备树添加到内核镜像中，将 board.dts 打包到 boot.img，将 board-B.dts 打包到 bootB.img。

AB 系统的设备树不同点只在于 chosen 节点中的 root 变量，例如，下面为快起方案的 AB 系统的设备树使用说明：

:

```
chosen {
    bootargs = "earlycon=uart8250,mmio32,0x02500000 clk_ignore_unused initcall_debug=0 console=ttyAS0,115200
      loglevel=8 root=/dev/mmcblk0p3 rootwait init=/init rdinit=/rdinit partitions=bootA@mmcblk0p1:
      bootB@mmcblk0p2:rootfsA@mmcblk0p3:rootfsB@mmcblk0p4:boot-resource@mmcblk0p5:env@mmcblk0p6:
      env-redund@mmcblk0p7:boottone@mmcblk0p8:rootfs_data@mmcblk0p9:private@mmcblk0p10:
      UDISK@mmcblk0p11 ......";
  };
```

B 系统：

```
chosen {
    bootargs = "earlycon=uart8250,mmio32,0x02500000 clk_ignore_unused initcall_debug=0 console=ttyAS0,115200
      loglevel=8 root=/dev/mmcblk0p4 rootwait init=/init rdinit=/rdinit partitions=bootA@mmcblk0p1:
      bootB@mmcblk0p2:rootfsA@mmcblk0p3:rootfsB@mmcblk0p4:boot-resource@mmcblk0p5:env@mmcblk0p6:
      env-redund@mmcblk0p7:boottone@mmcblk0p8:rootfs_data@mmcblk0p9:private@mmcblk0p10:
      UDISK@mmcblk0p11 ......";
  };
```

可以看到，两个设备树的 chosen 节点基本上相同，只有 root=/dev/mmcblk0p3 root=/dev/mmcblk0p4 的差异。

比如：

chosen 节点中 root=/dev/mmcblk0p3，partitions=…rootfsA@mmcblk0p3…，那么当前的 rootfs 就是 rootfsA，就是 A 系统。

chosen 节点中，root=/dev/mmcblk0p4，partitions=…rootfsB@mmcblk0p4…
rootfs 就是 rootfsB，就是 B 系统。

注意：chosen 节点中的 partitions 变量的值，一定要与分区表 (sys_partition.fex) 中的顺序一一对齐，否则会引起打包失败的问题。

使用方法与非快起方案一样，使用 swupdate_pack_swu 命令打包 OTA 包。

因为 B 系统使用独立的内核镜像，那么 A 系统和 B 系统的内核升级时需要使用不同的内核镜像，所以需要将 B 系统的内核镜像打包到 OTA 包中。

例如：AB 系统的配置文件为：

```
swota_file_list=(
openwrt/target/xxx/xxx/swupdate/sw-description-ab:sw-description
${LICHEE_PACK_OUT_DIR}/boot_package.fex:uboot       # uboot镜像
${LICHEE_PACK_OUT_DIR}/boot0_sdcard.fex:boot0_emmc   # emmc介质使用的boot0文件
${LICHEE_PACK_OUT_DIR}/boot0_nand.fex:boot0_nand     # nand介质使用的boot0文件
${LICHEE_PACK_OUT_DIR}/boot.fex:kernel           # A系统的内核镜像
${LICHEE_PACK_OUT_DIR}/bootB.fex:kernelB          # B系统的内核镜像
${LICHEE_PACK_OUT_DIR}/rootfs.fex:rootfs          # rootfs镜像
)
```

AB 系统的策略描述文件，在升级 AB 系统的内核时，更改了上面在配置文件中打包的内核镜像名称，变更为：

```
......
  now_A_next_B = {
      images: (
        {
          filename = "kernelB";
          device = "/dev/by-name/bootB";
          installed-directly = true;
        }
......

  now_B_next_A = {
      images: (
        {
```

```
            filename = "kernel";
            device = "/dev/by-name/bootA";
            installed-directly = true;
        }
......
```

在使用方法上，与原本的非快起方案没有区别。

3.17.2.2 recovery 系统

快起方案与非快起方案的 recovery 系统的使用是没有差别的。

除了增加 recovery 分区，同时也要在设备树 board.dts 中的 chosen 节点添加 recovery。

chosen 变量节点的顺序要和分区表中的分区顺序一致。

```
chosen {
    bootargs = "earlycon=uart8250,mmio32,0x02500000 clk_ignore_unused initcall_debug=0 console=ttyAS0,115200
      loglevel=8 root=/dev/mmcblk0p2 rootwait init=/init rdinit=/rdinit partitions=boot@mmcblk0p1:
      rootfs@mmcblk0p2:boot-resource@mmcblk0p3:env@mmcblk0p4:env-redund@mmcblk0p5:
      boottone@mmcblk0p6:rootfs_data@mmcblk0p7:private@mmcblk0p8:recovery@mmcblk0p9:
      UDISK@mmcblk0p10 ......";
};
```

recovery 系统的使用方法与非快起方案一样，使用 swupdate_make_recovery_img 命令编译 recovery 系统。

使用 swupdate_pack_swu 命令打包 OTA 包。

4 ota-burnboot 介绍

4.1 章节说明

此章节主要介绍如何在 OTA 时升级 boot0/uboot。

具包含两个方面内容：

OTA 命令升级 boot0 和 uboot。

OTA 升级 boot0 和 uboot 的 C/C++ APIs。

4.2 概念说明

表 4-1: ota-burnboot 相关概念说明表

说明	
boot0	较为简单，主要作用是初始化 dram 并加载运行 uboot。一般不需修改。
uboot	功能较丰富，支持烧写，启动内核，烧 key 及其他一些定制化的功能。
sys_config	全志特有的配置文件，对于使用 linux3.4/uboot2011 的平台，在打包之后 sys_config 会跟 uboot 拼接到一起。对于使用 linux3.10/uboot2014 及更高版本的平台，sys_config 会在打包阶段，跟设备树的配置合并，生成最终的 dtb。linux5.4 开始不再合并到 dtb。
dtb	设备树，由 dts 配置和 sys_config 配置综合得到。
ot.fex	使用 linux3.4/uboot2011 的平台最终用到的 uboot+sys_config。
boot_package.fex	使用 linux3.10/uboot2014 及更高版本的平台最终用到的 uboot，其实包含的文件由配置文件 boot_package.cfg 决定，一般至少包含了 uboot 和 dtb，安全方案会包含一些安全所需文件文件，可能还有 bootlogo 等文件。
toc0.fex	安全方案使用的 boot0。

说明	
toc1.fex	安全方案使用的 uboot，类似 boot_package.fex 说明，其中实际也包含了 dts 等多个文件。

即，本文介绍的升级 uboot，其实是升级 uboot+dtb 这样的一个整体文件。后文不再区分更新 uboot，更新 sys_config，更新 dtb。这几个打包完毕是合成一个文件的，暂不支持单独更新其中一个，需整体更新。

b用于更新的

获取用于 OTA 的 boot0 与 uboot 的 bin 文件，用于加入 OTA 包中。

4.3.1 编译 boot0 uboot

如果原本的固件生成流程已经包含编译 uboot，则正常编译固件即可。

否则可按照如下步骤编译生成 uboot

```
$ source build/envsetup.sh

$ lunch
  => 选择方案。
$ muboot
  => 编译 uboot。
$ mboot0
  => 编译 boot0 (注意，此命令在大多数平台无效，因为boot0不开源，SDK中提供了编译好的bin文件)。
```

置环境变量。

编译后会自行拷贝 bin 文件到该平台的目录下，即：

```
device/config/chips/${CHIP}/bin
```

编译出的 boot0/uboot 还不能直接用于 OTA，请继续编译和打包固件，如执行：

```
$ make -j <N>
=> 编译命令，若只修改 boot0/uboot/sys_config 无需重新编译，可跳过。
$ pack [-d]。
=> 非安全方案的打包命令。
$ pack -s [-d]
=> 安全方案的打包命令。
```

修改了 dts 则需要执行，重新编译。

4.3.2 关于更新 boot0

大多数平台，代码环境中并不包含 boot0 相关代码，因此无法编译 boot0。

一般情况下并不需要修改 boot0，而是直接使用提供的 boot0 的 bin 文件即可。

boot0 mboot0

4.3.3 Bin 文件路径

4.3.3.1 使用 uboot2018 及更高版本的非安全方案

根据对应存储介质选择 bin。

boot0：

```
out/pack_out/boot0_nand.fex    ： nand 方案使用的 boot0。
out/pack_out/boot0_sdcard.fex  ： mmc 方案使用的 boot0。
out/pack_out/boot0_spinor.fex  ： nor 方案使用的 boot0。
```

uboot：

```
out/pack_out/boot_package.fex      ： nand/mmc 方案使用的 uboot。
out/pack_out/boot_package_nor.fex  ： nor 方案使用的 uboot。
```

4.3.3.2 安全方案

:

```
out/pack_out/toc0.fex          ： 安全方案使用的 boot0。
```

uboot：

```
out/pack_out/toc1.fex          ： 安全方案使用的 uboot。
```

4.4 OTA 升级命令

4.4.1 支持 OTA 升级命令

升级 boot0 与 uboot 分别使用 ota-burnboot0 与 ota-burnuboot 命令。

两个命令都是 OTA 升级 boot0 和 uboot 的 C/C++ APIs 的封装。

要支持本功能，需要选中 ota-burnboot 的包，即：

选择以下配置：

```
<*>ota-burnboot
```

在 Linux-5.15 内核中，nand 驱动没有 ioctl，在没有 ioctl 的情况下，目前 nand 通过/dev/ot0、dev 节点更新

4.4.2 ota-burnboot0

4.4.2.1 命令说明

```
$ Usage: ota-burnboot0 <boot0-image>
```

升级 boot0，其中 boot0-image 是镜像的路径。

，安全和非安全方案所使用的 boot0-image 是不同的，具体见 “用于更新的 bin 文件” 章节。

4.4.2.2 使用示例

```
root@TinaLinux:/# ota-burnboot0 /tmp/boot0_nand.fex
Burn Boot0 Success
```

4.4.3 ota-burnuboot

4.4.3.1 命令说明

```
$ Usage: ota-burnuboot <uboot-image>
```

升级 uboot，其中 uboot-image 是镜像的路径。请注意，安全和非安全方案，不同的 uboot 版本，所使用的 uboot-image 是不同的，具体见 “用于更新的 bin 文件” 章节。

4.4.3.2 使用示例

```
root@TinaLinux:/# ota-burnuboot /tmp/u-boot.fex
Burn Uboot Success
```

4.5 OTA 升级 C/C++ APIs

包含头文件 OTA_BurnBoot.h，使用库 libota-burnboot.so。

4.5.1 int OTA_burnboot0(const char *img_path)

- 作用：烧写 boot0
- 参数：
  - img_path：boot0 镜像路径
- 返回：
  - 0：成功；
  - 非零：失败。

4.5.2 int OTA_burnuboot(const char *img_path)

- 作用：烧写 uboot
- 参数：
  - img_path：uboot 镜像路径
- 返回：
  - 0：成功；
  - 非零：失败。

4.6 底层实现

4.6.1 如何保证安全更新 boot0/uboot

前提条件是，flash 中存有不止一份 boot0/uboot。在这个基础上，启动流程需支持校验并选择完整的 boot0/uboot 进行启动，更新流程需保证任意时刻掉电，flash 上总存在至少一份可用的 boot0/uboot。

4.6.2 Nand Flash NFTL 方案实现

在 nand nftl 方案中，boot0 和 uboot 是由 nand 驱动管理，保存在物理地址中，逻辑分区不可见。

Nand 驱动会保存多份 boot0 和 uboot，启动时，从第一份开始依次尝试，直到找到一份完整的 boot0/uboot 进行使用。

更新 boot0/uboot 时，上层调用 nand 驱动提供的接口，驱动中会从第一份开始依次更新，多份全部更新完毕后返回。因此可保证在 OTA 过程中任意时刻掉电，flash 中均有至少一份完整的 boot0/uboot 可用。再次启动后，只需重新调用更新接口进行更新，直到调用成功返回即可。

目前 nand 中的多份 boot0/uboot 是由 nand 驱动管理的，只能整体更新，暂不支持单独更新其中

。

4.6.3 Nand Flash UBI 方案实现

在 nand ubi 方案中，boot0 一般存放于 mtd0 中，uboot 存放于 mtd1 中。

与 nftl 方案一样，底层实际是保存多份 boot0 和 uboot。启动时，从第一份开始依次尝试，直到找到一份完整的 boot0/uboot 进行使用。对上提供多份统一的更新接口，软件包会通过对 mtd 的 iotcl 接口发起更新。

注：用户空间直接读写/dev/mtdx 节点，需要内核使能 CONFIG_MTD_CHAR=y。

4.6.4 MMC Flash 实现

在 mmc 方案中，boot0 和 uboot 各有两份，存在 mmc 上的指定偏移处，逻辑分区不可见。需要读写可直接操作/dev/mmcblk0 节点的指定偏移。

具体位置:

```
1 sector = 512 bytes = 0.5k。

boot0/toc0 保存了两份，offset1: 16 sector，offset2: 256 sector。

uboot保存了两份，offset1: 24576 sector
```

启动时会先读取 offset1，如果完整性校验失败，则读取 offset2。

⚠ 注意

更新时，默认只更新 offset1，而 offset2 是保持在出厂状态的。只要 offset1 正常更新了，则启动时会优先使用。如果在更新 offset1 的过程中掉电导致数据损坏，则自动使用 offset2 进行启动。
如需定制策略，例如改成每次 offset1 和 offset2 均更新，可自行修改 ota-burnboot 代码。

现NOR Flash

nor 方案中，只保存一份 boot0 和 uboot，更新过程中掉电可能导致无法启动，只能进行刷机。故目前未实现 ota 更新，需后续扩展。

4.7 dram 参数更新

dram 初始化是在 boot0 阶段，若需要 OTA 升级 dram 参数，就需要升 boot0。

⚠ 注意

参数需要修改，请与全志官方联系。

4.7.1 emmc 介质升级 dram 参数不生效

为了兼容部分芯片在 usb 烧录时更新了 dram 参数到设备的 boot0 头部，遗留下的代码。每次升级 boot0，以设备端原有的 dram 参数为准。

实际上在大部分芯片不需要，在 nand,nor 介质上也没有实现。所以，可以修改文件：

```
platform/allwinner/system/ota-burnboot/src/BurnSdBoot.c
```

memcpy 部分代码注释，具体代码：

```
static int updateSdBoot0(int fd, BufferExtractCookie *cookie) {
...
   /* copy the ddr param from old to new */
   //memcpy((void *)newToc0Config->dram_para, (void *)oldToc0Config->dram_para, (32 *4));
...
   /* copy the ddr param from old to new */
   //memcpy((void *)newBoot0->prvt_head.dram_para, (void *)oldBoot0->prvt_head.dram_para, (32 *4));
...
}
```

4.7.2 dram 参数训练

部分芯片 dram 参数训练时间较长，出现在烧录阶段或者 OTA 升级 boot0 之后第一次启动。

这个是必要的，为了让 dram 参数更适应对应的设备。

dram 参数训练的结果保存到 flash 的 boot_para 区域，nor,nand,emmc 有各自的实现。

由于 boot0 是极少需要更新的；而且影响的只有 OTA 升级后的第一次启动时间，对后续再次启动时间无影响。

所以，不建议专门去处这种现象。

若是连 OTA 后第一次启动时间较长也无法接受的，需要提问题单与全志官方联系处理。

全志内部的优化处理是 OTA 升级 boot0 后重启，不重新进行 dram training 动作；直接使用 boot_param 区域的 dram 参数。

带来的优点是

```
SDK dram参数无变更时，OTA升级boot0后，第一次启动时间无明显增长。
```

缺点是：

OTA升级boot0后第一次重启

1. 需要单独维护优化后的boot0
2. 若是出现SDK dram参数需要变更，必需回退优化补丁，再去升级boot0；
3. 设备端必然需要耗时，因为SDK dram参数变了，
设备端dram参数需重新训练。

5 其他功能

5.1 在用户空间操作 env

Tina 中支持了 uboot-envtools 软件包。

```
<*> uboot-envtools
```

即可将uboot env部分使用的变量。

5.2 AB 系统切换

5.2.1 使用背景

1、OTA 升级时防止升级失败而导致变砖;

2、运行过程中系统损坏可以切换到另外一个系统;

他需要切换系统的情况。

5.2.2 uboot 原生启动计数机制

uboot 原生支持了启动计数功能。该功能主要涉及三个变量。

```
upgrade_available=0/1 #总开关，设置1才会进行bootcount++及检查切换，设为0则表示关闭此功能
bootcount=N     #启动计数，若upgrade_available=1，则每次启动bootcount++，并用于跟bootlimit比较
bootlimit=5     #配置判定启动失败的阈值
```

即编译支持该功能后，当 upgrade_available=1，则 uboot 会维护一个 bootcount 计数值，每次动加一，并判断 bootcount 是否超过 bootlimit。如果未超过，则正常启动，执行 env 中配置的 bootcmd 命令。如果超过，则执行 env 中的 altbootcmd 命令。

支持此功能后，启动脚本或主应用需要在合适的时机清除 bootcount 计数值，表示已经正常启动。

配置：

```
使能：配置CONFIG_BOOTCOUNT_LIMIT
选择bootcount存放位置，
如存放在env分区(掉电不丢失)：配置CONFIG_BOOTCOUNT_ENV
```

```
如存放在RTC寄存器(掉电丢失)：配置CONFIG_BOOTCOUNT_RTC
如需存放在其他位置可自行拓展
```

在用户空间可配置 upgrade_available 的值对此功能进行动态开关。例如可在 OTA 之前打开，确认 OTA 成功后关闭。也可一直保持打开。在用户空间，需要清空 bootcount，否则多次重启就会导致 bootcount 超过 bootlimit。

5.2.3 uboot 部分 AB 系统切换逻辑及标志说明

在 uboot 中，AB 系统有两种切换方式分别对应上面 2 中场景：

1、应用自动切换（对应场景 1，ota 升级切换）；

[illegible]

5.2.4 全志定制系统切换

5.2.4.1 应用自动切换

全志自己添加了 CONFIG_SUNXI_SWITCH_SYSTEM 功能进行系统切换。目前尚未大量使用，供参考。

[illegible]

在 flash 的 env 分区中设置了 2 个标志变量 systemAB_now，systemAB_next。systemAB_now：由 uboot 阶段进行设置，该标志位系统应用只读不能写（通过工具 fw_printenv，或其他工具）。systemAB_next：无论是 uboot 还是系统应用都可读可写（一般 uboot 不会主动修改，除非系统已经损坏需要切换）。初始值一般设置为

```
systemAB_next=A      #表示下次启动A系统
systemAB_now=A       #表示当前为A系统，不设置也可以，uboot会自动设置
```

当前系统应用可以通过 systemAB_now 标志识别当前系统是 A 系统还是 B 系统。系统应用可以把标志变量 systemAB_next=A/B 写到 env 分区里，然后重启。重启后 uboot 会去检查 systemAB_next 这个标志，如果标志是 systemAB_next=A，uboot 会启动 A 系统，如果是 systemAB_next
[illegible]
temAB_now 供系统读取。

底层设置：

对上接口定义的是 systemA/B，以系统为单位。目前的系统组成定义为包含 kernel 和 rootfs 分区。考虑不同方案，分区名可能不同，因此支持用环境变量指定具体分区名，而不是 hardcode 为某个分区名。

```
systemA=boot          A系统kernel分区名
rootfsA=rootfs        A系统rootfs分区名
systemB=boot_B        B系统kernel分区名
rootfsB=rootfs_B      B系统rootfs分区名
```

底层实现：

此机制会动态修改 boot_partition 和 root_partition 的值。具体 boot_partition 和 root_partition 变量的作用，请参考本文档其他章节的介绍。

5.2.4.2 系统损坏切换

在 flash 的 env 分区中中设置了 4 个标志变量，upgrade_available，bootlimit，bootcount 和 nAB_damage。

```
upgrade_available=0/1      使用损坏切换功能必须设置，0表示关闭损坏切换功能，1表示开启损坏切换功能
bootlimit=N                使用损坏切换必须设置，系统启动失败多少次后进行系统切换
bootcount=n                启动失败次数计数值，uboot阶段会对它加1，应用正常启动需要对它清零(当bootcount > bootlimit后
      会进行系统切换)
systemAB_damage=A/B        系统损坏后由uboot设置，此标志uboot设置后不会进行清除，需要应用对它进行处理(该标志
      位只表明该系统曾经损坏)
```

如果 uboot 中已经使能了系统损坏功能，那么系统可以通过设置 upgrade_available 标志位动态打开或关闭系统损坏切换功能。

使能 upgrade_available 后每次系统启动时 uboot 阶段都会对 bootcount 加 1，系统正常启动后系统应用需要把 bootcount 清零 (正常启动 bootcount 的值都不会超过 bootlimit)。

如果系统启动失败然后重启 (借由外部手段重启或看门狗重启)，每次 bootcount 都会在 uboot 阶段加 1，但因为系统损坏没有进入到系统，bootcount 标志没有被系统应用及时清零，当启动失败次数到达一定次数后 (bootcount > bootlimit)，uboot 会进行系统切换, 如果是 A 系统损坏会切换到 B 系统，同时 uboot 会在 env 里设置标志 systemAB_damage=A，这样 B 系统起来后可以知道 A 系统曾经损坏了。

同理 B 系统损坏会切换到 A 系统，同时在 env 设置标志 systemAB_damage=B。

如果使能损坏切换功能后，系统应用必须在每次正常启动后使用 fw_setenv(或其他工具) 对 bootcount 进行清 0，否则 uboot 会把该系统视为已损坏。

bootcount 清零的参考脚本：（可以根据需要添加到开机自启动脚本中）

```
#reset bootcount, when bootcount > bootlimit, uboot will switch system AB
bootcount=$(fw_printenv -n bootcount 2>/dev/null)
  [ x"$bootcount" != x"0" ] && {
    fw_setenv bootcount 0
  }
```

如果不想使用损坏切换功能，系统应用把 upgrade_available 动态设为 0 关闭即可。

6 注意事项

6.1 Q & A

Q：系统的哪些部分是可以升级的?

rnel 和 rootfs 是可以升级的，但为了掉电安全，需要搭配一个 recovery 系统或者做 AB 系统。对于 nand 和 emmc 来说，boot0/uboot 存在备份，可以升级。对于 nor 来说，boot0/uboot 没有备份，不能升级。或者说升级有风险，中途掉电会导致无法启动。boot0/uboot 的升级具体可参考本文档中的 ota-burnboot 部分。对于 swupdate 升级方案，可以自行在 sw-description 中配置策略，升级自己的定制分区和文件，但务必考虑升级中途掉电的情况，必要的话需要做备份和恢复机制。

Q：系统的哪些部分是不能升级的?

A：分区表是不可升级的，因为改动分区表后，具体分区对应的数据也要迁移。建议在量产前规划好分区，为每个可能升级的分区预留部分空间，防止后续升级空间不足。nor 方案的 boot0/uboot 是不可升级的，因为没有备份。其余不确定是否能够升级的，请向开发人员确认。

s 如何升级?

A：目前默认跟内核绑定，需要与内核一起升级。

Q：升级过程掉电重启后，是从断点继续升级还是从头升级?

A：从头开始升级，例如定义了在 recovery 系统中升级 boot 分区和 rootfs 分区，则在升级 boot 或 rootfs 过程中断电，重启后均是从 boot 重新开始升级。

7 升级失败问题排查

凡是遇到升级失败问题先看串口 log，如果不行再看/mnt/UDISK/swupdate.log 文件

7.1 分区比镜像文件小引起的失败

[illegible]

```
[ERROR] : SWUPDATE failed [0] ERROR handlers/ubivol_handler.c : update_volume : 171 : "recovery"  will not fit volume
        "recovery"
```

解决方法：增加对应方案 tina/device/config/chips/< 芯片编号 >/configs/< 方案名 >/sys_partition.fex 文件的对应分区的大小

7.2 校验失败

[illegible]

解决方法：重新烧录固件，制作差分包

声明

版权所有 ©2025 珠海全志科技股份有限公司。保留一切权利。

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。