



Linux GKI 开发指南

版本号: 2.2
发布日期: 2021.8.23

版本历史

版本号	日期	制/修订人	内容描述
1.0	2020.04.15	AWA1132	初版
1.1	2020.08.31	AWA1132	补充 GKI 镜像编译及获取，增加 bootloader gki 开发说明，增加 GKI 常见问题调试
1.2	2020.11.12	AWA1132	修正部分 GKI 开发说明，增加 GKI 官方镜像启动失败调试说明
2.0	2020.11.18	AWA1132	根据评审建议更新文档
2.1	2021.6.4	XAA0193	根据 android12 的升级过程，1: 增加了 android12 相关的环境搭建指南；2: 更新了 gki 的测试 faq
2.2	2021.8.23	XAA0193	根据 android12 升级过程的实际案例，更新了部分资料，新增了案例章节并补充了案例；

目 录

1 概述	1
1.1 编写目的	1
1.2 适用范围	1
1.3 适用人员	1
1.4 术语介绍	1
2 GKI 介绍	2
2.1 什么是 GKI	2
2.2 为什么要用 GKI	3
2.3 GKI Roadmap	5
3 GKI 开发	6
3.1 boot partition	6
3.2 bootloader 开发	7
3.2.1 Android 10 与 Android 11 头部组织结构的变化	7
3.2.2 ramdisk 拼接	8
3.3 驱动模块开发	8
3.3.1 增加驱动版本号管理	9
3.3.2 不使用 sys_config	9
3.3.3 可使用 MODULE 变量	9
3.3.4 不使用 #ifdef 来判断配置是否打开	9
3.3.5 不使用解析 cmdline 参数的接口	9
3.3.6 不使用 OF_DECLARE 接口	9
3.3.7 不使用 debugfs 接口	10
3.3.8 不使用 vfs_read/vfs_write/kernel_read/kernel_write 接口	10
3.3.9 模块 ko 加载过程中不允许失败	10
3.3.10 配置 Android ko 加载列表	10
3.4 GKI whitelist	11
3.4.1 whitelist 介绍	11
3.4.2 export 接口被优化问题	12
3.4.3 AW 模块 whitelist 需求收集	12
3.4.4 如何向 Google 提交 whitelist	15
3.4.5 android11 更新 abi_gki_aarch64.xml	16
3.4.6 android12 更新 abi_gki_aarch64.xml	16
3.5 GKI defconfig	17
3.5.1 gki_defconfig 介绍	17
3.5.2 如何更新 gki_defconfig	17
3.6 android11-BUILD	17
3.6.1 环境搭建	17
3.6.2 使用 Google 源码编译 GKI 镜像	18
3.7 android12-BUILD	18
3.7.1 环境搭建	18

3.7.2 使用 Google 源码编译 GKI 镜像	19
3.8 GKI 启动流程	19
3.8.1 GKI 基本启动流程	19
4 GKI 兼容性测试	20
4.1 使用自编译 GKI 镜像 (仅供调试用)	20
4.2 使用 Google 官方发布的 GKI 镜像	20
4.3 GKI 兼容性测试常见问题	21
4.3.1 symbol magic 不一致导致 ko 加载失败	21
4.3.2 whitelist 缺失导致 ko 加载失败	22
4.3.3 烧写 google 官方 GKI 镜像后, 设备不断重启, 并且没有任何内核打印	24
4.3.4 放开 ko 加载调试信息	25
4.3.5 boot.img 编译失败	26
4.3.6 常见相关文档	27
5 GKI tools	28
6 GKI 补丁提交	29
7 GKI 相关问题案例库	30
7.1 android12 升级过程 GKI 启动失败	30
7.1.1 问题现象	30
7.1.2 分析过程	31

插 图

2-1 GKI 框架	2
2-2 AOSP common kernel 老的发展模式	3
2-3 AOSP common kernel 新的发展模式	4
3-1 gki boot partition 示意图	6
3-2 of declare	10
3-3 gki whitelsit detect	13
3-4 gki whitelsit detect	13
3-5 gki whitelsit detect	14
3-6 gki whitelsit detect	14
3-7 gki whitelsit detect	15
3-8 gki whitelist submitt	16
4-1 gki vmlinux	25
4-2 android first stage console	25
4-3 kernel printk ratelimit	26

1 概述

1.1 编写目的

本文档介绍了 Google GKI，用于在 Linux-5.4 以及更新版本内核的升级项目中，指导模块驱动进行 GKI 开发，以及指导 GKI 兼容测试及维护。本文档以 AW A133 B3 平台为例子进行介绍。

1.2 适用范围

本文档适用于所有需要支持 Google GKI 的平台（Linux-5.4 以及更新版本）。

1.3 适用人员

Linux 模块驱动开发人员。

1.4 术语介绍

术语	解释
GKI	Generic Kernel Image
KMI	Kernel Module Interface
ACK	Android Common Kernels
LTS	Long Term Supported linux kernel
vendor	芯片厂商
kernel fragmentation	内核代码碎片化
GSI	Generic System Image

2 GKI 介绍

2.1 什么是 GKI

GKI, Generic Kernel Image, 也就是通用内核镜像, 从字面上理解就是一个通用的 kernel 镜像, 可以在不同 SoC 厂商的不同设备上运行。GKI 的目的是为了统一内核代码, 把 SoC 板级驱动从内核镜像中剥离出来, 并将其都编译为 ko。也就是说, Google GKI 要求 SoC 厂商在新的 android 内核代码中, 不要随意去修改内核通用代码, 只增加并维护好自己的 SoC 板级驱动 ko, 驱动代码要与内核框架层解耦。如果需要修改内核通用代码或者提交 bug-fix, 必须先提交到 linux mainline 内核分支中, 再 merge 到 google 分支。

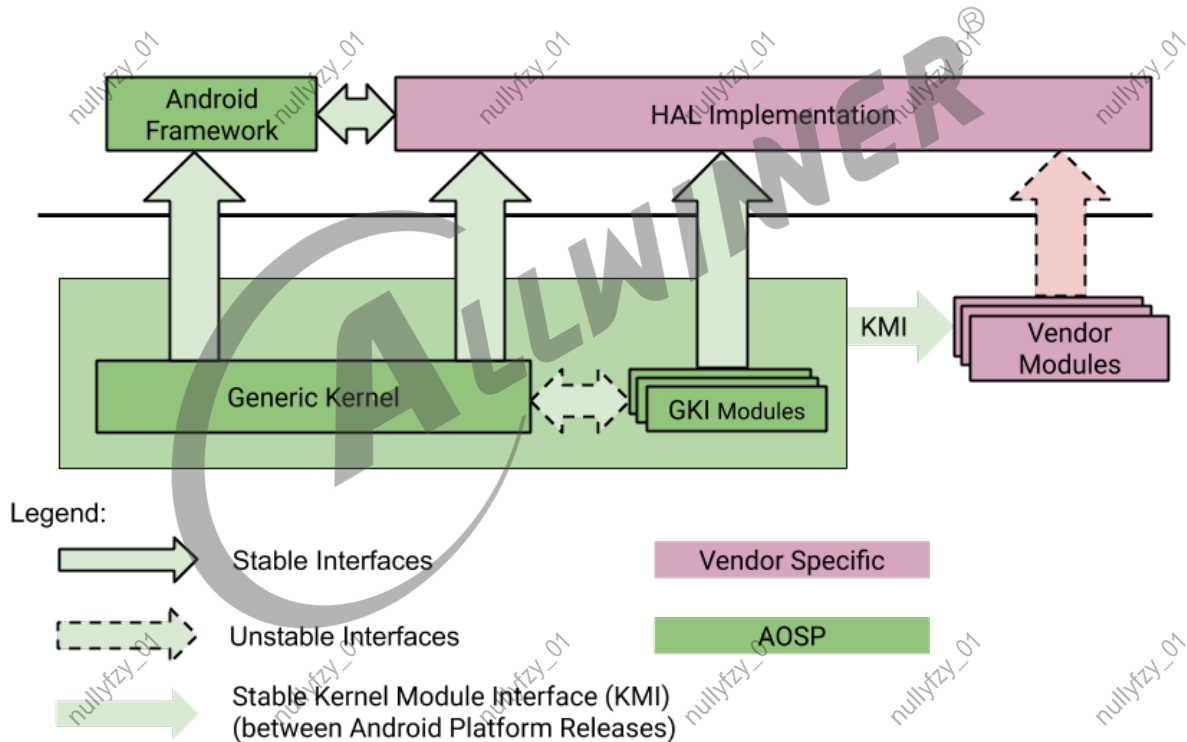


图 2-1: GKI 框架

上图是一个 Android 设备搭载 GKI 的框架图, 从图中可以看到, Google 会提供 KMI 接口, 用于 vendor modules 和 GKI 的通讯。几个关键词的解析如下:

- Generic Kernel: 通用内核代码
- GKI Modules: 通用内核模块, 与 Generic Kernel 一起组成 GKI;
- KMI: GKI export 给模块 ko 使用的接口;
- Vendor Modules: 芯片厂商驱动, 比如 sunxi iommu 驱动;

2.2 为什么要用 GKI

先看一张在 2019 以及之前的 Google 内核发展模式图：

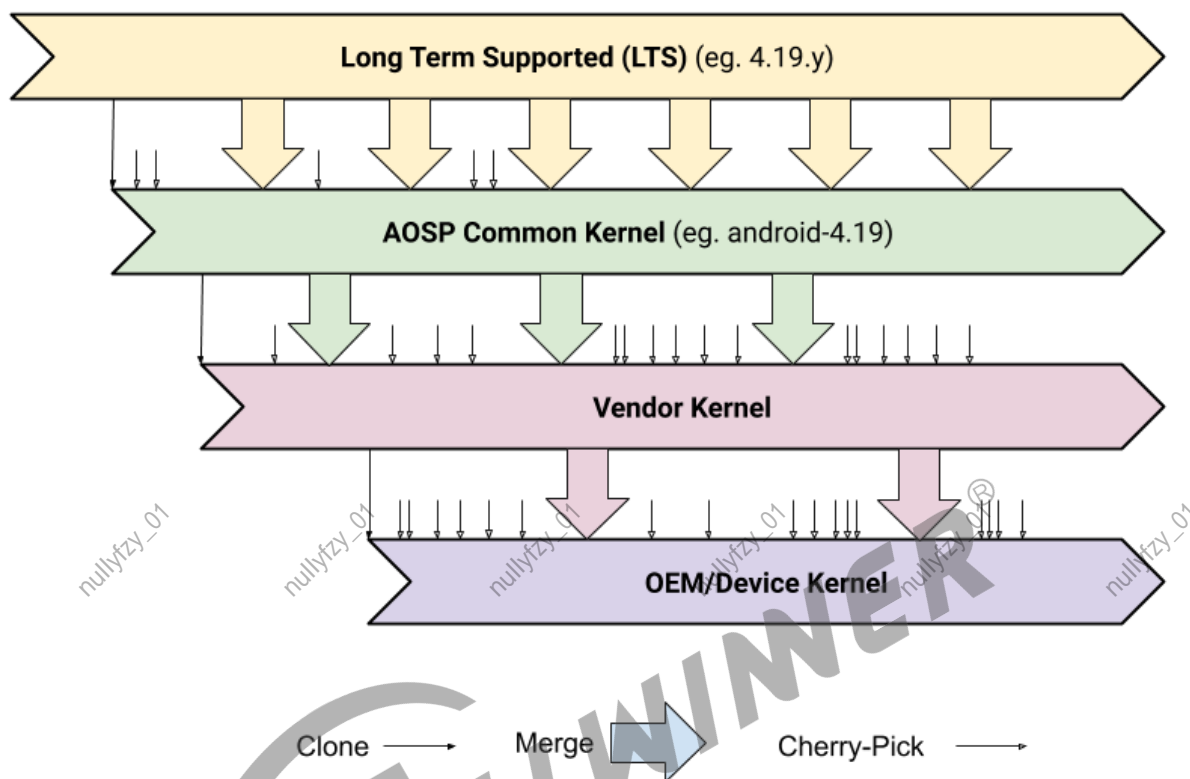


图 2-2: AOSP common kernel 老的发展模式

以 AW Android 10 的开发为例，android 内核分支发展如下：

```
LTS linux-4.9 release
--> android-4.9
    --> android-4.9-q(Android10 feature complete)
        --> sunxi-dev linux-4.9-q
            --> AW clinet linux-4.9-q
```

在这个过程中会形成 kernel fragmentation，也即是内核代码碎片化，因为几乎每个 Android 设备都有一个定制化的内核代码，使得很多代码都是 out-of-tree 的状态（游离在上游 linux 内核和 android 内核之外），很难将这些代码提交到 LTS 中，同时也大大增加了补丁同步的难度，最直观的体现就是我们的内核代码往往没有及时同步上游分支中的 bug-fix 补丁、安全补丁。

为了解决 kernel fragmentation，Google 推出了新的 android 内核发展模式，如下图：

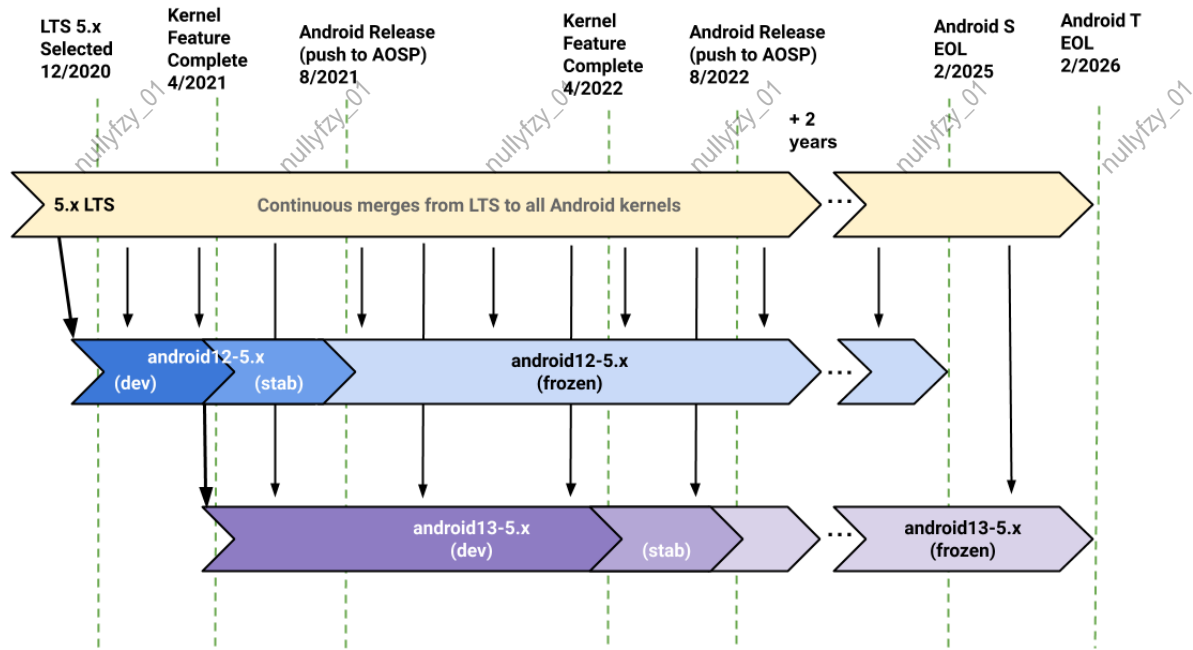


图2-3: AOSP common kernel 新的发展模式

以目前正在进行 AW Android 11 + linux-5.4 的开发为例，android 内核分支发展如下：

```
LTS linux kernel
--> android-mainline
--> android12-5.4(Linux-5.4 feature complete)
--> android11-5.4(Android11 feature complete)
--> sunxi-dev linux-5.4
--> AW clinet linux-5.4
```

这里需要注意几个关键点：

- android-mainline：android 内核主线分支会持续从 LTS merge 补丁，并充分测试；
- android11-5.4：Android 11(R) 对应的内核分支；
- android12-5.x：Android 12(S) 对应的内核分支；
- android13-5.x：Android 13(T) 对应的内核分支；
- 所以在新的模式下，将不会有所谓的 desert-kernel 分支，比如 android-4.9-o、android-4.9-q 等；
- 假如 AW 支持了 GKI，那 sunxi-dev linux-5.4 可以毫不费劲地从 android11-5.4 上同步补丁。

在新的模式下支持 GKI，就可以解决 kernel fragmentation。

2.3 GKI Roadmap

Gogole GKI 分为下面两个阶段推进：

- I. GKI 兼容性：Android 11(R) + linux-5.4 require GKI compatibility test.
- II. GKI 产品化：Android 12(S) + linux-5.x 及之后 require GKI kernel.

第一阶段是要求 SoC 厂商要通过 GKI 兼容性测试，这也是我们现阶段的目标，具体的 GKI 兼容性测试的方式会在下文介绍；第二阶段是要求 SoC 厂商要直接使用 Google 提供的 GKI 镜像出货。

其实到了第二阶段，有个明显的好处就是我们从 linux-5.4 开始，就不需要再为 android 产品做内核升级了，Google 会帮我们做。

Google 将这两个阶段称为 GKI1.0 和 GKI2.0，详细的区别如下：

在 GKI 1.0 中，使用 Android 11 和 Linux-5.4 启动的设备必须通过 VTS 和 CTS-on-GSI + GKI 测试。GKI 1.0 目标包括以下内容。

- 用 GKI 内核替换产品内核时，避免在 VTS 或 CTS 中进行回归。
- 减轻合作伙伴的负担，使他们的内核与 AOSP 通用内核保持最新。
- 在内核中包含 Android 的核心更改，以便通过新的 Android 版本升级和启动设备。
- 不要破坏 Android 用户空间。
- 将特定于硬件的组件与核心内核分开，作为可加载模块。

在 GKI 2.0 中，使用 Android 12 和 Linux-5.x（其中 5.x 是 5.4 的下一个版本，将在 2020 年底被选定为长期支持（LTS）的内核）启动的设备必须随附 GKI 内核。签名的启动映像可用，并定期通过 LTS 和严重的错误修复进行更新。因为 KMI 保持二进制稳定性，所以您可以安装这些引导映像而无需更改供应商映像。GKI 2.0 目标包括以下内容。

- 用 GKI 内核替换产品内核时，不要引入明显的性能或功耗下降。
- 使合作伙伴无需供应商参与即可提供内核安全修复程序和错误修复程序。
- 降低将设备的主要内核版本更新的成本（例如，从 v5.x 到 v5.y）。
- 通过使用清晰的升级过程更新内核版本，可以为每个体系结构维护一个 GKI 内核二进制文件。

注意：Android 11 + Linux-5.4 属于 GKI 1.0 阶段，只要求 GKI 兼容性，并且 Android 12 + 5.4 也是如此。

3 GKI 开发

3.1 boot partition

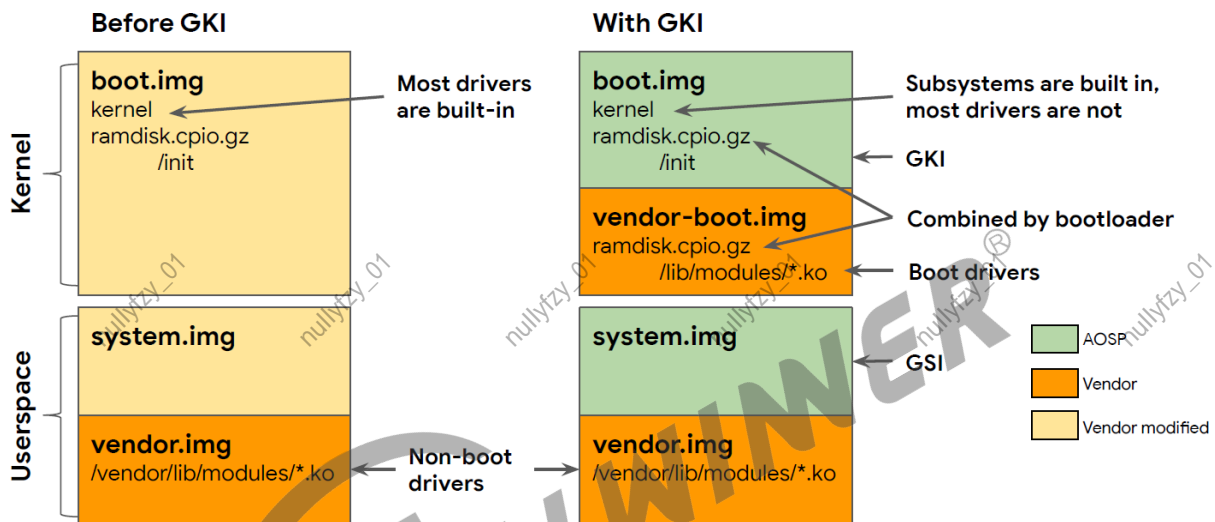


图 3-1: gki boot partition 示意图

上图是 boot 分区支持 GKI 的变化情况，这张图很有指导意义，其中关键点有：

- boot.img 变为 boot.img + vendor-boot.img，其中 boot.img 中放的是 GKI 镜像 + ramdisk，vendor-boot.img 中放的是需要启动加载的 vendor 模块 ko；
- 启动过程中，需要 bootloader 加载并整合 boot.img 和 vendor-boot.img 中的两个 ramdisk；
- 与 GKI 对应的，Google 还推出 GSI，也就是通用 system 镜像；
- vendor.img 可以存放不需要要启动加载的 ko，这个跟以前是一样的。

bootloader 开发需要注意的地方有：

- boot header V3.0；
- DTB 的存放位置从 boot.img 改到了 vendor-boot；
- 启动过程中 boot 需要加载 boot.img 和 vendor-boot.img 中的 ramdisk 并做整合，而且 boot.img 的 ramdisk 优先级高。

这里插讲一下 GKI 兼容性测试：

GKI 兼容性测试中，Google 会提供 boot.img（包含了 GKI 镜像），替换掉我们固件中的 boot.img，并将该固件烧到机器中，如果机器能成功启动，vendor-boot 中的模块 ko 能顺利加载并运行，系统功能正常，并且最终要能通过 VTS 和 CTS-on-GSI + GKI 测试。所以为了能通过测试，我们需要做到以下三点：

- build：我们的模块 ko 编译环境要跟 Google 编译 GKI 的环境保持一致；
- bootloader：bootloader 需要成功加载并整合 ramdisk；
- device driver：模块驱动代码需要与框架层解耦。

3.2 bootloader 开发

3.2.1 Android 10 与 Android 11 头部组织结构的变化

- dtb 从 boot.img 变到了 vendor_boot.img 中。
- 删除了 second stage, recovery dtbo。
- 新增了 vendor_ramdisk。
- boot.img 包括 boot_header, kernel, ramdisk, second stage, recovery dtbo, dtb 的数据

```
+-----+
| boot header      | 1 page
+-----+
| kernel           | n pages
+-----+
| ramdisk          | m pages
+-----+
| second stage     | o pages
+-----+
| recovery dtbo/acpio | p pages
+-----+
| dtb              | q pages
+-----+
n = (kernel_size + page_size - 1) / page_size
m = (ramdisk_size + page_size - 1) / page_size
o = (second_size + page_size - 1) / page_size
p = (recovery_dtbo_size + page_size - 1) / page_size
q = (dtb_size + page_size - 1) / page_size
```

- Android 11 时，boot.img 变成了 boot header, kernel, ramdisk 的数据。

```
+-----+
| boot header      | 1 page
+-----+
| kernel           | n pages
```

```
+-----+
| ramdisk          | m pages
+-----+
n = (kernel_size + page_size - 1) / page_size
m = (ramdisk_size + page_size - 1) / page_size
```

- Android 11, 增加了 vendor_boot, 将开发者的 KO 放在 vendor_ramdiskz 中, vendor_boot.img 包括 vendor_boot_header, vendor_ramdisk, dtb

```
+-----+
| vendor boot header | n page
+-----+
| vendor ramdisk     | o pages
+-----+
| DTB                | p pages
+-----+
n = (2108 + page_size - 1) / page_size
o = (vendor_ramdisk_size + page_size - 1) / page_size
p = (dtb_size + page_size - 1) / page_size
```

3.2.2 ramdisk 拼接

- Android 11 中, ramdisk 分为两份, 一份为 boot_ramdisk, 存放在 boot.img 中一份为 vendor_boot_ramdisk 存放在 vendor_boot.img 中
- 在 bootloader 启动时需要先后加载 boot_ramdisk, vendor_boot_ramdisk 并进行前后拼接。
- ramdisk 用的是 cpio.lz4 格式, 可以进行简单的首尾拼接, 但是 bootloader 需要注意两个 ramdisk 中间必须紧密拼接, 不能对齐再拼接, 否则会导致内核解压时失败, 同时拼接后改变 ramdisk 的大小。

3.3 驱动模块开发

Google GKI 要求 SoC 板级驱动做到以下几点:

- 模块 ko 化;
- 与框架层解耦, 能成功在 Google 通用内核中运行。
- 提交初版驱动时, 原则上只添加源文件, 不修改内核原生代码;

模块 ko 化的方式就不在这里讲了, 遇到问题请自行 google 解决。下面介绍驱动开发中需要注意的地方, 该章节会持续更新...

3.3.1 增加驱动版本号管理

Linux-5.4 模块驱动要求版本号管理，各个模块都要有自己的版本号记录，最好能相应有的 version 节点。

3.3.2 不使用 sys_config

Linux-5.4 需要完全剥离 sys_config.fex，设备驱动配置一律放在 dts 中。

3.3.3 可使用 MODULE 变量

当模块被编译为 ko 时，MODULE 变量为 true，built-in 时为 false。所以可以使用 #ifdef MODULE 和 #ifndef MODULE 来进行 ko 和 built-in 的区分。

3.3.4 不使用 #ifdef 来判断配置是否打开

Linux-5.4 驱动代码中不能使用 #ifdef 或者 #if define 来判断 menuconfig 配置是否打开，必须使用：#if IS_ENABLED()。

比如模块 AAA 编译为 ko，对应的 CONFIG_AAA=m，驱动代码中不能用 #ifdef CONFIG_AAA 来判断，而必须使用 #if IS_ENABLED(CONFIG_AAA)，因为用 #ifdef 无法判断编译为 ko 的配置，会导致判断出错。

所以，在驱动代码中，原则上编译为 KO 的模块必须使用：#if IS_ENABLED()，而 built-in 的模块可以使用 #ifdef。但是，为了避免疏忽，建议统一使用 #if IS_ENABLED()。

3.3.5 不使用解析 cmdline 参数的接口

模块驱动编译为 ko，驱动代码中不能使用 early_param_setup_setup_param 这类接口来解析 cmdline 参数，不然要么编译出错，要么得到的结果为 0。这类驱动通常是需要从 bootloader 中获取参数，建议把该参数放在 dts 中，由 uboot 更新 dts 对应的参数，最后在驱动中直接解析 dts。请模块负责人兼顾好 uboot 和内核驱动的适配，如果需要，请找 bootloader 同事支持。

3.3.6 不使用 OF_DECLARE 接口

当驱动被编译为 ko 时，OF_DECLARE 就相当于一个空操作，所以 TIMER_OF_DECLARE、CLK_OF_DECLARE、IRQCHIP_DECLARE 等均不允许在驱动为 ko 时使用。

google 官方提供的适配示例如下，使用 MODULE 变量来区分：

```
#ifdef MODULE
static int xxx_timer_probe(struct platform_device *pdev)
{
    struct device_node *np = pdev->dev.of_node;
    return xxx_timer_init(np);
}

static const struct of_device_id xxx_timer_match_table[] = {
    { .compatible = "XXX,xxx-timer" },
    {}
};

MODULE_DEVICE_TABLE(of, xxx_timer_match_table);

static struct platform_driver xxx_timer_driver = {
    .probe = xxx_timer_probe,
    .driver = {
        .name = "xxx-timer",
        .of_match_table = xxx_timer_match_table,
    },
};

module_platform_driver(xxx_timer_driver);

#else
TIMER_OF_DECLARE(xxx_timer, "XXX,xxx-timer", xxx_timer_init);
#endif
```

图 3-2: of declare

3.3.7 不使用 debugfs 接口

Android 11 不允许打开 CONFIG_DEBUG_FS，debugfs 类接口都不能调用，为了满足调试需求，可以用 CONFIG_DEBUG_FS 宏定义来做兼容。

3.3.8 不使用 vfs_read/vfs_write/kernel_read/kernel_write 接口

5.4.43 内核版本移除了 sdcardfs 驱动，连带 vfs_read/vfs_write/kernel_read/kernel_write 接口也取消了 export，所以在驱动代码中，不能调用这几个接口去操作文件节点。

3.3.9 模块 ko 加载过程中不允许失败

启动过程中模块 ko 加载失败，会导致设备会重启，所以要做好模块启动顺序及依赖关系的梳理，制定好启动加载流程，同时要求模块驱动不要有太强的模块依赖性。

3.3.10 配置 Android ko 加载列表

A133 B3 的 ko 加载列表文件为如下：

```
android/device/softwinner/ceres-b3/configs/vendor_ramdisk.modules
```


只需在该文件中增加 ko 的名称，列表的先后顺序等于 ko 加载顺序。出现在该列表上的模块会在 android 启动的 frist init stage 阶段被加载，如果想要把某些模块加载延后，比如放在 late-fs 阶段加载，则可以在 init.xxx.rc 文件中配置。

3.4 GKI whitelist

3.4.1 whitelist 介绍

Google GKI 镜像中不可能包含所有 SoC 设备驱动所需要的代码，肯定会有很多 SoC 驱动 ko 没法直接在 GKI 上运行，那怎么办呢？为此，Google 提供了一份 gki_defconfig 和 GKI whitelist，让 SoC 厂商根据自身的需求，往 gki_defconfig 中增加模块配置，并在 whitelist 中添加自己需要的 symbol(函数、数据成员等)，Google 会把 gki_defconfig 中的配置项全部编译到 GKI 镜像中，并将这份 whitelist 对应的 symbol 全部 export 出来，供模块 ko 使用。同时，Google 也会利用这份 whitelist 来监测及维护 KMI 的 ABI 稳定性，ABI 稳定性是 GKI 得以实现的基础。

当然，Google 会有一份原始 GKI whitelist，有个别芯片厂商目前也提交了自己的 whitelist。如果这些 whitelist 加起来可以满足我们驱动的开发，那是再好不过了。如果不满足，我们就需要增加自己的 whitelist，并将其提交到 google 内核分支中。

对于 arm64 来说，这份基础白名单为 abi_gki_aarch64_whitelist，位于内核顶层目录下。目前 Goole GKI 正在开发中，我们可以看到这份白名单几乎是空的，因为 Google 前期是基于高通 db845c 平台开发的，具体可以看 abi_gki_aarch64_db845c_whitelist。google 编译 GKI 内核时，会把所有内核顶层目录下的所有 whitelist 合并为一份，并且将其 export 出来。

现阶段的 whitelist 如下：

```
~/google/common/android$  
├─ abi_gki_aarch64  
├─ abi_gki_aarch64_cuttlefish  
├─ abi_gki_aarch64_db845c  
├─ abi_gki_aarch64_exynos  
├─ abi_gki_aarch64_galaxy  
├─ abi_gki_aarch64_goldfish  
├─ abi_gki_aarch64_hikey960  
├─ abi_gki_aarch64_oneplus  
├─ abi_gki_aarch64_oplus  
├─ abi_gki_aarch64_qcom  
├─ abi_gki_aarch64_sunxi  
├─ abi_gki_aarch64_unisoc  
├─ abi_gki_aarch64_vivo  
├─ abi_gki_aarch64_workarounds  
└─ abi_gki_aarch64.xml
```


3.4.2 export 接口被优化问题

Google 在编译 GKI 时打开了 CONFIG_TRIM_UNUSED_KSYMS，这个选项会在编译 GKI 镜像时对 export symbol 进行优化，使得很多本来是 export 出来的接口变成了非 export，而 GKI whitelist 则是强制 export，不让其优化 whitelist 中的接口。而我们一般不打开这个选项，即使打开了，由于编译内容不同，也会存在差异。

这就可以解释以下现象：明明有些接口已经使用 EXPORT_SYMBOL、EXPORT_SYMBOL_GPL 透出来了，在 GKI 镜像中却没有被 export 出来。

在我们公司的开发方案中，会从 aosp 将这个abi_gki_whitelist文件拷贝至我们的 SDK 的如下目录，然后进行在编译过程使用。

```
android-S/longan/kernel/linux-5.4/abi_gki_whitelist
```

比如在 android12 上这个文件同步的补丁如下：

```
commit dc9576a26ce610ba6a185565725b7952132f4522
Author: liyong <liyong@allwinnertech.com>
Date:   Sun Apr 25 09:27:45 2021 +0800

    K1:sunxi:P2:gki:add abi_gki_whitelist

    Built from aosp official branch android12-5.4
    File copied from: out/android12-5.4/common/abi_symbollist.raw

-----
    commit a5631d493c290fee4c2102804d71c21e304fdecf
    Author: Alexander Potapenko <glider@google.com>
    Date:   Thu Apr 15 17:48:38 2021 +0200

        ANDROIDID: gki_defconfig: temporarily disable KFENCE in GKI

-----

    Module Version: NULL
    PMS TaskID: 43409

    Change-Id: I1501a68210cf2e090da5e2c02adb839d8771d746
```

3.4.3 AW 模块 whitelist 需求收集

在模块完成 ko 之后，可以利用 google 提供的工具及 GKI 对应的 vmlinux 文件，检测自己的模块 ko 所需要的接口是否有在 GKI 中 export 出来，具体步骤如下。

- 1. 拷贝 extract_symbols 工具和 GKI vmlinux；

工具和 vmlinux 请到以下路径取（也可以自行下载 google 源码编译后获取，编译见后面章节）：

192.168.200.10 ▶ ShareForAll ▶ huangshuosheng ▶ GKI ▶ whitelist_detect_4-15			
名称	修改日期	类型	
extract_symbols	2020/4/13 18:39	文件	
vmlinux	2020/4/15 19:42	文件	
sunxi_iommu_whitelist	2020/4/15 16:45	文件	
gki_defconfig	2020/4/15 19:45	文件	

图 3-3: gki whitelsit detect

其中, gki_defconfig 为编译 google GKI 镜像生成的.config, 不直接使用 gki_defconfig 文件, 是因为在编译 GKI 时, 会对 defonfig 进行更新, 所以必须以.config 为准。

编译后产生的.config 路径为: out/android11-5.4/common/.config

- 2. 拷贝模块 ko 到同一路径下, 并运行 extract_symbols;

执行命令: ./extract_symbols --whitelist out.whitelist

其中 out.whitelist 为输出文件, 可以自己命名, 保存了符合要求的模块接口, 即 GKI 已经 export 了这些接口, 可以不用关注。

下面以 sunxi_iommu.ko 为例:

```
Hss:~/gki/google$ ls
extract_symbols  sunxi-iommu.ko  vmlinux
Hss:~/gki/google$
Hss:~/gki/google$ ./extract_symbols --whitelist out.whitelist
Symbol __dynamic_dev_dbg required by sunxi-iommu.ko but not provided
Symbol __dynamic_pr_debug required by sunxi-iommu.ko but not provided
Symbol iommu_device_sysfs_add required by sunxi-iommu.ko but not provided
Symbol iommu_device_sysfs_remove required by sunxi-iommu.ko but not provided
Symbol iommu_group_alloc required by sunxi-iommu.ko but not provided
Symbol iommu_group_register_notifier required by sunxi-iommu.ko but not provided
Hss:~/gki/google$
Hss:~/gki/google$ ls
extract_symbols  out.whitelist  sunxi-iommu.ko  vmlinux
```

图 3-4: gki whitelsit detect

可以看到 google GKI 没有 export 出 sunxi-iommu.ko 所需要的所有接口, “not provided” 表示 GKI 没有 export 出该接口, 但是 sunxi-iommu.ko 需要用到。最直接的解决方法就是将其添加到 gki whitelist 中, 让 google 编译 GKI 时把这些接口 export 出来。当然我们也要先考虑是否一定需要调用这些接口, 看是否能够优化。

其中 `__dynamic_dev_dbg` 和 `__dynamic_pr_debug` 可以忽略，因为我们打开了 `CONFIG_DYNAMIC_DEBUG`，而 google GKI 是没打开的，后续 GKI 测试时，我们也需要关闭该选项。类似的接口还有：`mutex_destroy` 等。

- 3. 确认接口能否加入到 whitelist

当发现 google GKI 没有 export 出自己模块所需接口时，首先要看 google GKI 是否有编译这些接口所对应的代码，如果有，那我们可以考虑将其加入 whitelist；如果没有，我们优先考虑将其编译为 ko；如果发现无法将其编为 ko，则只能考虑给 google 提需求，但需慎重考虑，因为 google 对与 gki defconfig 的提交很严格，毕竟涉及到 ABI 稳定性问题。

可以通过 google 编译 GKI 所用到的 config，也就是上面提供的 `gki_defconfig`，来确认是否能直接加入 whitelist。

还是以 `sunxi_iommu.ko` 为例子，上面看到它还需要 export 的接口有：`iommu_device_sysfs_add`、`iommu_device_sysfs_remove` 等，找到这些接口对应源文件的编译选项：`CONFIG_IOMMU_API`，在 google `gki_defconfig` 中找：

```
Hss:~/gki/google$ cat gki_defconfig | grep CONFIG_IOMMU_API
CONFIG_IOMMU_API=y
```

图 3-5: gki whitelsit detect

能找到并且为 y，说明可以直接加入 whitelist，

之所以推荐用 `gki_defconfig` 方式进行确认，主要想让大家弄清楚模块 CONFIG 的依赖关系，虽然可能会稍微费点时间。当然，也可以通过直接解析 `vmlinux` 进行快速确认：

```
Hss:~/gki/google$ nm --defined-only vmlinux | grep iommu_device_sysfs_add
ffffffc01075b210 T iommu_device_sysfs_add
ffffffc01162ed44 d __ksym_marker_iommu_device_sysfs_add
```

图 3-6: gki whitelsit detect

上图可以看出 `iommu_device_sysfs_add` 已经被编译进 GKI 镜像，但是没有被 export 出来，因为编译时被优化了。

下面是我把 `iommu_device_sysfs_add` 加入 gki whitelist 之后，在 google 内核环境下编译出 `vmlinux` 的运行结果：

```
Hss:~/gki/google$ nm --defined-only vmlinux | grep iommu_device_sysfs_add
0000000006607bd95 A __crc_iommu_device_sysfs_add
ffffffc01075b25c T iommu_device_sysfs_add
ffffffc010d75838 t iommu_device_sysfs_add.cfi_jt
ffffffc0116510f7 r __kstrtab_iommu_device_sysfs_add
ffffffc01162f044 d __ksymtab_iommu_device_sysfs_add
ffffffc011641038 r __ksymtab_iommu_device_sysfs_add
```

图 3-7: gki whitelsit detect

kstrtab、ksymtab 等字段表示已经 export 出来了。

• 4.whitelist 添加格式

```
[abi_whitelist]

# required by sunxi_iommu.ko
iommu_device_sysfs_add
iommu_device_sysfs_remove
iommu_group_alloc
iommu_group_register_notifier
```

模块负责人可以自己新建一个 whitelist，可以命名为：sunxi_xxx 模块 _whitelist，按照上面的格式添加，并发给 GKI 负责人。

注意：whitelist 需求检测不是做一遍就完事的，当 google GKI whitelist 定下来之后，KMI 也就定了，所以要求模块负责人每次提交代码时都检查一下 whitelist 需求。当然，后续可以考虑在 Gerrit 上做自动化 whitelist 检查。

3.4.4 如何向 Google 提交 whitelist

GKI 负责人在收集完模块 whitelist 需求后，整理为一份 whitelist：abi_gki_aarch64_sunxi_whitelist，充分评估后，可提交到 google 内核分支，步骤如下。

- 增加 abi_gki_aarch64_sunxi_whitelist 到 google kernel 顶层目录；
- 更新 abi_gki_aarch64.xml；
- 修改 build.config.gki.aarch（该步骤仅为首次提交 whitelist 所需）；

```
Hss:~/workspace/google/kernel/common$ git diff build.config.gki.aarch64
diff --git a/build.config.gki.aarch64 b/build.config.gki.aarch64
index 8b681cd..e471ec4 100644
--- a/build.config.gki.aarch64
+++ b/build.config.gki.aarch64
@@ -11,5 +11,6 @@ abi_gki_aarch64_goldfish_whitelist
abi_gki_aarch64_hikey960_whitelist
abi_gki_aarch64_qcom_whitelist
abi_gki_aarch64_unisoc_whitelist
+abi_gki_aarch64_sunxi_whitelist
"
TRIM_NONLISTED_KMI=1
```

图 3-8: gki whitelist submit

其中, abi_gki_aarch64.xml 用于维护 ABI 稳定性, 不能手动修改, 必须利用 google 提供的 abi 工具进行更新。

3.4.5 android11 更新 abi_gki_aarch64.xml

- 下载 google 内核源码 (下载方法见下面章节) ;
- 在 build 仓库中打上补丁: <http://gerrit.allwinnertech.com:8081/#/c/118187/>
- 运行 bootstrap 工具配置编译环境: ./build/abi/bootstrap
- 编译内核以生成 xml 文件: BUILD_CONFIG=common/build.config.gki.aarch64 build/build_abi.sh -update -print-report
- 更新 xml 文件: 用 out_abi/android11-5.4/dist/abi.xml 覆盖 common/abi_gki_aarch64.xml

注意事项:

需要确保 libabigail 和 elfutils 这两个仓库的代码是最新的, 由于这两个仓库不在 AW gerrit 镜像仓库自动更新的组内, 所以得自己手动更新, 需要用到可以访问外放的编译服务器, 用 git merge 的方式同步。

git://sourceware.org/git/libabigail.git	需要更新的branch: mm-next (注意不是master)
git://sourceware.org/git/elfutils.git	需要更新的branch: master

编译后产生的 whitelist 路径为: out/android11-5.4/common/abi_symbolist.raw

3.4.6 android12 更新 abi_gki_aarch64.xml

在 google 源码路径下运行如下命令:

```
BUILD_CONFIG=common/build.config.gki.aarch64 build/build_abi.sh --update --print-report
```

3.5 GKI defconfig

3.5.1 gki_defconfig 介绍

google gki 内核镜像是基于 gki_defconfig 编译的，gki_defconfig 的具体路径如下所示：

```
~/google/common/arch/arm64/configs$  
.  
├─ db845c_gki.fragm          --> vendor modules  
├─ defcon  
├─ gki_defcon                --> GKI kernel  
├─ hikey960_gki.fragm  
└─ rockpi4_deffig
```

以高通 db845c 为例子，db845c_gki.fragment 文件是用于编译 soc 驱动模块，里面的模块全都配置为 KO，gki_defconfig 是 Google 用来编译 GKI 镜像的配置。目前，AW 内部平台还没把 GKI 公共配置和模块配置分开，所以没有.fragment 文件。

3.5.2 如何更新 gki_defconfig

如果由于开发需求限制，必须在 gki_defconfig 中打开某些配置，则需要将改动提交给 google。

介绍一下如何规范地更新 gki_defconfig 文件，而不是直接手动修改：

```
~/google/common/android$ make ARCH=arm64 gki_defconfig  
~/google/common/android$ make ARCH=arm64 menuconfig  
~/google/common/android$ make ARCH=arm64 savedefconfig  
~/google/common/android$ cp defconfig arch/arm64/config/gki_defconfig
```

3.6 android11-BUILD

3.6.1 环境搭建

Google 内核的下载及编译方法可在 AW wiki 上搜：Android 11 开发环境搭建。

- 下载 Google kernel 源码 (AW 内部开发下载)：

```
repo init -u http://gerrit.allwinnertech.com:8081/aosp/kernel/manifest -b common-android11-5.4  
repo sync  
repo start --all android11-5.4
```

- 下载 Google kernel 源码 (外网下载):

```
repo init -u https://android.googlesource.com/kernel/manifest -b common-android11-5.4
repo sync
```

3.6.2 使用 Google 源码编译 GKI 镜像

- 需要确保 google 源码中 gki_defconfig 打开了 ARCH_SUNXI (已经向 google 提交 patch, 尚未合并, 目前需要手动打补丁);
- 从 google 官方网站获取 gki ramdisk 镜像;

<https://ci.android.com/builds/branches/aosp-master-throttled/grid?>

下载 aosp_arm64 中的 ramdisk.img 镜像。

- 在 google 源码顶层目录下运行以下命令:

```
BUILD_BOOT_IMG=1 SKIP_VENDOR_BOOT=1 KERNEL_BINARY=Image GKI_RAMDISK_PREBUILT_BINARY=gki-ramdisk.img BUILD_CONFIG=common/build.config.gki.aarch64 build/build.sh
```

编译后产生的 boot.img 镜像路径为: out/android11-5.4/dist/boot.img

3.7 android12-BUILD

3.7.1 环境搭建

- 内部途径

```
repo init -u http://gerrit.allwinnertech.com:8081/aosp/kernel/manifest -b common-android12-5.4
repo start --all android12-5.4
```

- 外网途径

有外网的权限时, 可以通过如下命令进行环境的搭建, 或者有外网的权限时, 可以通过如下命令进行环境的搭建:


```
mkdir android-kernel
cd android-kernel/
repo init -u https://android.googlesource.com/kernel/manifest -b common-android12-5.4
repo sync
repo start --all android12-5.4
```

3.7.2 使用 Google 源码编译 GKI 镜像

在 google 源码目录下运行如下命令：

```
BUILD_BOOT_IMG=1 SKIP_VENDOR_BOOT=1 KERNEL_BINARY=Image GKI_RAMDISK_PREBUILT_BINARY=gki-ramdisk.img BUILD_CONFIG=common/build.config.gki.aarch64 build/build.sh
```

编译过程中在LTO vmlinux.o阶段可能会卡主一段时间（大概十分钟），属于正常现场，第一次编译会比较耗时。

- 编译完成后，boot.img 存放在：

```
android-kernel/out/android12-5.4/dist/boot.img
```

3.8 GKI 启动流程

3.8.1 GKI 基本启动流程

```
boot0
--> uboot: 加载boot.img中的GKI和ramdisk_1, 加载vendor-boot.img的ramdisk_2, 整合ramdisk_1和ramdisk_2,jump to GKI
--> GKI: core kernel init
--> init: load modules ko
--> android bringup
```


4 GKI 兼容性测试

完整的 google GKI 兼容性测试是指，芯片厂商的设备使用 google 提供的 GKI 镜像能成功启动、功能正常，并且能通过 VTS 和 CTS-on-GSI+GKI tests。

GKI 兼容性测试需要依赖 Android + Kernel 编译环境构建完成，能够编译出 boot.img 和 vendor-boot.img，并且 bootloader 要满足合并两个 ramdisk 的需求。

4.1 使用自编译 GKI 镜像 (仅供调试用)

测试步骤：

- 获取 GKI boot.img，可以参照前面章节，编译得到 GKI boot.img；
- 在设备中打开 OEM unlocking 以解锁设备：Developer options -> OEM unlocking，或者通过命令的方式进行解锁：
- 使用 fastboot 烧写 GKI boot.img 到我们的机器，命令如下：

```
adb reboot bootloader
fastboot oem unlock
fastboot flash boot boot.img
fastboot reboot
```

注意：自编译的 GKI 镜像仅用于调试使用，GKI 兼容性测试必须使用 Google 官方发布的 GKI 镜像。

- 重启后，验证基本功能；
- 进行 VTS-CTS 测试

4.2 使用 Google 官方发布的 GKI 镜像

测试步骤与上面的基本相同，只不过不需要我们自己编译 GKI boot.img，直接到 Google 官方发布网站上拿。

android11-5.4 的 GKI 镜像获取地址为（需要特定权限访问）：

```
https://partner.android.com/build/builds;branch=git_rvc-gsi-release;product=aosp_arm64?a=1122546797
```

需要注意的是：

- 1. 不要使用 boot-debug 镜像，仅适用于 GKI 之前的设备；
- 2. boot-5.4.img 为未压缩的 boot.img（AW 只支持未压缩的 boot.img）；
- 3. boot-5.4-gz、boot-5.4-lz4 分别为 gz、lz4 压缩的 boot.img（AW 不支持压缩 boot.img，不能使用）；

4.3 GKI 兼容性测试常见问题

4.3.1 symbol magic 不一致导致 ko 加载失败

- 问题现象：第一个 ko 就加载失败了，系统不断重启；
- 问题原因：根据 log 提示，我们可以看到有 module_layout 报错，这 SYMBOL MAGIC 不一致问题的典型代表（symbol magic 是个大的范畴，module_layout 是其中一个，而因为 module_layout 是加载 ko 时第一个会检查的接口，所以一般出错都是报在这个接口中），这个参数不一样就代表也就是我们本地环境编译 ko 时生成的 symbol magic 与 GKI 环境编译的不一致，这个参数可以在 Module.symvers 文件中查看，具体的路径如下所示：

```
longan/out/kernel/build/Module.symvers # longan下的文件路径
```

```
androidS-google/out/android12-5.4/common/Module.symvers # aosp环境下的文件路径
```

- 解决办法：symbol magic 跟内核实际编译的内容有关系，所以需要确保我们编译内核的 **defconfig (arch/arm64/configs/sun50iw10p1smp_a100_android_defconfig)** 与 **gki_defconfig** 保持一致。
- 补充问题 log：

```
[ 2.553495] ccu_sunxi_ng: disagrees about version of symbol module_layout
[ 2.560111] init: Failed to insmod '/lib/modules/ccu-sunxi-ng.ko' with args ''
```

```
[ 5.043399][ T1] init: Loading module /lib/modules/arc4.ko with args ''
[ 5.050979][ T1] arc4: disagrees about version of symbol crypto_register_skcipher
[ 5.059306][ T1] arc4: Unknown symbol crypto_register_skcipher (err -22)
[ 5.066802][ T1] arc4: disagrees about version of symbol skcipher_walk_virt
[ 5.074584][ T1] arc4: Unknown symbol skcipher_walk_virt (err -22)
[ 5.081477][ T1] arc4: disagrees about version of symbol skcipher_walk_done
[ 5.089267][ T1] arc4: Unknown symbol skcipher_walk_done (err -22)
[ 5.096212][ T1] arc4: disagrees about version of symbol crypto_unregister_skcipher
[ 5.104764][ T1] arc4: Unknown symbol crypto_unregister_skcipher (err -22)
```

```
CONFIG_FB      #这个是我们自己编为m的，不用关注

CONFIG_KEXEC=y  # crashdump相关的配置，我们自己调试才打开的

CONFIG_CRASH_DUMP=y # crashdump相关配置，我们自己调试才打开的
```

- 问题现象: ko 加载过程中, 某个 ko(不是第一个) 加载失败了, 系统不断重启;
- 问题原因: 根据 log 提示, 可以知道问题原因是 kernel 镜像没有 export 出这些 symbol, 导致 ko 加载时没法使用这些 symbol, 从而加载失败。
- 解决办法: 该类问题一般是 GKI whitelist 不满足我们 ko 的需求, 需要根据 ko 需求更新 whitelist, 并且提交到 google。

```
[ 5.336277] lan78xx: Unknown symbol device_set_wakeup_enable (err -2)
[ 5.343984] lan78xx: Unknown symbol phy_ethtool_ksettings_get (err -2)
[ 5.351439] lan78xx: Unknown symbol genphy_read_status (err -2)
[ 5.358199] lan78xx: Unknown symbol mdiobus_read (err -2)
[ 5.364532] lan78xx: Unknown symbol phy_ethtool_set_wol (err -2)
```

- [illegible]

显而易见是由于 ehci_sunxi 模块所需要的 ehci_suspend 接口未被导出导致的。一般情况这个仅需在白名单文件中加上这个接口即可，但为了梳理白名单文件和编译的过程，接下来重头开始分析这个问题：

- 找到接口的提供函数，及其 Makefile 配置项：

```
~/workspace/S/a100/longan/kernel/linux-5.4$ grep -nr "ehci_suspend"
drivers/usb/host/ehci-hcd.c:1128:EXPORT_SYMBOL_GPL(ehci_suspend);

grep -nr "ehci-hcd.o"
drivers/usb/host/Makefile:40:obj-$(CONFIG_USB_EHCI_HCD) += ehci-hcd.o
```

可以得知这是一个内核模块，所以模块的维护应该由 google 维护（built in），在 google 未打开的情况下考虑在我们的 defconfig 中将其打开。

- 在 google 源码中查看接口函数：

```
~/android-kernel/common$ cat arch/arm64/
boot/          crypto/          Kbuild          Kconfig.debug   kernel/
  lib/          mm/          OWNERS
configs/        include/        Kconfig          Kconfig.platforms  kvm/
  Makefile      net/          xen/

~/android-kernel/common$ cat arch/arm64/configs/gki_defconfig | grep "CONFIG_USB_EHCI_HCD"
CONFIG_USB_EHCI_HCD=y
CONFIG_USB_EHCI_HCD_PLATFORM=y
```

发现在 gki_defconfig 中是打开了这个模块的，所以这个模块是被加载了，只是这个模块提供的 ehci_suspend 接口未被导出，所以需要在白名单中加入这个接口：

```
~/android-kernel/common$ git diff android/abi_gki_aarch64_sunxi
diff --git a/android/abi_gki_aarch64_sunxi b/android/abi_gki_aarch64_sunxi
index 5b031acc3f9d..90e2f66e197a 100644
--- a/android/abi_gki_aarch64_sunxi
+++ b/android/abi_gki_aarch64_sunxi

+#required by ehci-sunxi.ko
+ ehci_init_driver
+ ehci_resume
+ ehci_suspend
```

- 如果在 google 源码中查看接口函数发现 google 中未把接口（ehci_suspend）依赖的模块（ehci-hcd.c）built in，那我们就需要考虑将其这个模块（ehci-hcd.c）在我们公司的自己的 defconfig 中通过编译成模块的方式来解决，同时也不再需要在白名单中添加这个接口

4.3.3 烧写 google 官方 GKI 镜像后，设备不断重启，并且没有任何内核打印

这种问题一般是前面两个问题导致的，但是由于 google 官方 GKI 中没有使用 early_printk，所以还没等到打印输出就重启了，这时可以用以下步骤进行调试。

- 1. 检查 whitelist(快速定位问题)

使用 3.4.3 章节的方法，对所有模块 ko 进行 whitelist 检测。extract_symbols 工具和 vmlinux 的获取路径：

```
#extract_symbols:  
google/build/abi/extract_symbols  
  
# vmlinux: 这里用最新的google内核代码编译所得的vmlinux即可,  
google/out/android11-5.4/dist/vmlinux
```

将 extract_symbols、vmlinux、所有模块的 ko 放到同一个目录下，执行命令：

```
./extract_symbols --whitelist out.whitelist
```

如果出现提示 “required but not provided”（注意，开头为 __ 的函数不算），则为 4.3.2 问题；如果没有出现，则进行第二步调试（DS5 调试）。

- 2. 使用 earlycon 查看 log

如果 whitelist 检查没问题，则需要查看内核 log 以确认问题原因，最直接方便的方式是在 uboot 阶段打开 serial 8250(GKI 中使能) 的 earlycon，步骤如下：

```
a. 进入uboot shell  
  
b. 将bootargs中的earlyprintk替换 为earlycon，具体命令如下：  
setenv setargs_mmc setenv bootargs earlycon=uart8250,mmio32,0x05000000 clk_ignore_unused  
initcall_debug=${initcall_debug} console=${console} loglevel=${loglevel} root=${  
mmc_root} init=${init} cma=${cma} snum=${snum} mac_addr=${mac} wifi_mac=${wifi_mac}  
bt_mac=${bt_mac} specialstr=${specialstr} gpt=1 androidboot.force_normal_boot=${  
force_normal_boot} androidboot.slot_suffix=${}  
  
c. 执行boot命令启动，可以看到使用earlycon打印的内核log
```

- 3.DS5 调试

如果要更进一步调试，可以则需要用 DS5 进行调试，但要先获取改 google GKI 镜像对应的 vmlinux。

vmlinux 获取方法如下：

- 1. 确定 GKI boot-5.4.img 对应的内核版本号：grep -a "Linux version" boot-5.4.img

```
hss:~/gki/osp_arm64-img-6892010-userdebug-RJ1L.201808.001$ grep -a "Linux version" boot-5.4.img
teforward_delay timerhci Linux version 5.4.61-android11 (build-user@build-host) (Android (6443878 based on r383902) clang version 11.0.1 (https://android.googlesource.com/toolchain/llvm-project b397f81060ced7910426782172ed13bee898b79)) #1 SMP PREEMPT 2020-10-05 01:27:15
```

图 4-1: gki vmlinux

如上图所示，6888953 为内核版本号。

- 2. 打开对应的 google 内核构建页面，下载 vmlinux

链接地址如下，其中 kernel_version 为内核版本号：

https://ci.android.com/builds/submitted/<kernel_version>/kernel_aarch64/latest

以步骤 1 为例，即打开 https://ci.android.com/builds/submitted/6888953/kernel_aarch64/latest

4.3.4 放开 ko 加载调试信息

当 GKI ko 加载失败时，log 信息过少而不足以判断问题原因，这时可以把 ko 加载的调试信息放开，步骤如下（以 A133 B3 为例）：

- 在 android/device/softwinner/ceres-b3/BoardConfig.mk 中增加以下配置：

BOARD_KERNEL_CMDLINE += androidboot.first_stage_console=1

```
hss:~/workspace/androidR/android/device/softwinner/ceres-b3$ git diff
diff --git a/BoardConfig.mk b/BoardConfig.mk
index 6a75753..a386ee1 100644
--- a/BoardConfig.mk
+++ b/BoardConfig.mk
@@ -38,6 +38,7 @@ BOARD_BOOTIMAGE_PARTITION_SIZE := $(call get_partition_size,boot_a,$(PARTITION_C
BOARD_VENDOR_BOOTIMAGE_PARTITION_SIZE := $(call get_partition_size,vendor_boot_a,$(PARTITION_CFG_FILE))
BOARD_KERNEL_CMDLINE += androidboot.selinux=enforcing
BOARD_KERNEL_CMDLINE += androidboot.dtbo_idx=0,1,2
+BOARD_KERNEL_CMDLINE += androidboot.first_stage_console=1
BOARD_KERNEL_CMDLINE += firmware_class.path=/vendor/etc/firmware
BOARD_FLASH_BLOCK_SIZE := 4096
# prebuilt dtbo
```

图 4-2: android first stage console

- 屏蔽内核 printk 中的 ratelimit:


```

Hss:~/workspace/androidR/longan/kernel/linux-5.4$ git diff kernel/printk/printk.c
diff --git a/kernel/printk/printk.c b/kernel/printk/printk.c
index 971197f..9c920e5 100644
--- a/kernel/printk/printk.c
+++ b/kernel/printk/printk.c
@@ -832,10 +832,12 @@ static ssize_t devkmsg_write(struct kiocb *iocb, struct iov_iter *from)
     return len;

    /* Ratelimit when not explicitly enabled. */
+   /*
    if (!(devkmsg_log & DEVKMSG_LOG_MASK_ON)) {
        if (!__ratelimit(&user->rs, current->comm))
            return ret;
    }
+   */

```

图 4-3: kernel printk ratelimit

- 重新编译内核和 Android。

注意：打开这些调试选项后，android 只会启动到 frist init stage 阶段，不会再往下跑，仅适合用来调试模块加载问题，解决后需要关闭这些调试选项重新编译烧写固件验证。

4.3.5 boot.img 编译失败

- 问题现象：在 google 源码的白名单文件中加上了一些接口之后，编译失败，并有以下的打印报错：

```

~/android-kernel$ BUILD_BOOT_IMG=1 SKIP_VENDOR_BOOT=1 KERNEL_BINARY=Image
GKI_RAMDISK_PREBUILT_BINARY=gki-ramdisk.img BUILD_CONFIG=common/build.config.gki.
aarch64 build/build.sh

```

.....

Comparing the KMI and the symbol lists:

```

~/android-kernel/build/abi/compare_to_symbol_list /home/huangshuosheng/android-kernel/out/
android12-5.4/common/Module.symvers /home/huangshuosheng/android-kernel/out/android12
-5.4/common/abi_symbollist.raw

```

ERROR: Differences between ksymtab and symbol list detected!

Symbols missing from ksymtab:

```
- fb_notifier_call_chain
```

Symbols missing from symbol list:

- 排查过程：明显这个错误来自于 fb_notifier_call_chain 接口，从这个点入手，在 google 源码中 grep 寻找这个接口，得知这个接口来自文件：drivers/video/fbdev/core/fb_notify.c

```

~/android-kernel/common$ grep -nr "EXPORT_SYMBOL_GPL(fb_notifier_call_chain)"
drivers/video/fbdev/core/fb_notify.c:47:EXPORT_SYMBOL_GPL(fb_notifier_call_chain);

```

- 进一步找这个文件是否被 google 源码编译进去：

```
~/android-kernel/common$ cat arch/arm64/configs/ | grep "CONFIG_FB_NOTIFY"  
db845c_gki.fragment      defconfig                gki_defconfig            hikey960_gki.fragment  
rockpi4_defconfig
```

最终发现这个模块未被 google 提供，相应的这个接口也不会被提供。

- 解决办法：在 whitelist 中去掉这个接口，同时由于我们自己的驱动模块需要这个接口，则需要将这个模块 ko 化，对应的解决 patch 如下：

```
http://gerrit.allwinnertech.com:8081/#/c/lichee/linux-5.4/+155058/
```

4.3.6 常见相关文档

android 的 ko 加载文件：

```
android-S/device/softwinner/ceres/ceres-b4/system/vendor_ramdisk.modules
```

分区的大小文件：

```
android-S/device/softwinner/ceres/ceres-b4/system/sys_partition.fex
```


5 GKI tools

- ABI Monitoring for Andoird Kernels:

```
~/google/kernel/build/abi$  
.  
├─ abitool.py  
├─ bootstrap  
├─ diff_abi  
├─ dump_abi  
├─ extract_symbols  
├─ flatten_whitelist  
├─ gki_check  
├─ kmi_defines.py  
├─ __pycache__  
└─ README.md
```

- GKI Tools for Android Kernels:

```
~/google/kernel/build/gki$  
.  
├─ add_EXPORT_SYMBOL_GPL  
├─ add_MODULE_LICENSE  
├─ device_snapshot  
├─ find_circular  
├─ instrument_module_init  
└─ README.md
```

6 GKI 补丁提交

GKI 补丁提交包括 gki whitelist、gki_defconfig、SoC 驱动代码的提交，如果有相关需求请联系 AW。



7 GKI 相关问题案例库

7.1 android12 升级过程 GKI 启动失败

7.1.1 问题现象

在 A133 android12 升级过程中，出现了 GKI 无法校验通过的问题（此前在 android12 BSP 预研阶段，已经支持了 GKI 部分的全部工作），log 如下所示：

```
[ 4.694885][ T1] Failed to find error inject entry at (__ptrval__)
[ 4.702332][ T1] Failed to find error inject entry at (__ptrval__)
[ 4.709778][ T1] Failed to find error inject entry at (__ptrval__)
[ 4.717225][ T1] Failed to find error inject entry at (__ptrval__)
[ 4.724671][ T1] Failed to find error inject entry at (__ptrval__)
[ 4.732117][ T1] Failed to find error inject entry at (__ptrval__)
[ 4.738899][ T1] Failed to find error inject entry at (__ptrval__)
[ 4.746347][ T1] Failed to find error inject entry at (__ptrval__)
[ 4.753794][ T1] Failed to find error inject entry at (__ptrval__)
[ 4.761240][ T1] Failed to find error inject entry at (__ptrval__)
[ 4.768686][ T1] Failed to find error inject entry at (__ptrval__)
[ 4.775465][ T1] Failed to find error inject entry at (__ptrval__)
[ 4.782244][ T1] Failed to find error inject entry at (__ptrval__)
[ 4.789691][ T1] Failed to find error inject entry at (__ptrval__)
[ 4.797132][ T126] hw perfevents: unable to count PMU IRQs
[ 4.803254][ T126] hw perfevents: /pmu: failed to register PMU devices!
[ 4.811233][ T44] hw perfevents: unable to count PMU IRQs
[ 4.816874][ T44] hw perfevents: /pmu: failed to register PMU devices!
[ 4.824482][ T1] cfg80211: Loading compiled-in X.509 certificates for regulatory
database
[ 4.835951][ T1] cfg80211: Loaded X.509 cert 'sforshee: 00b28ddf47aef9cea7'
[ 4.844092][ T32] platform regulatory.0: Direct firmware load for regulatory.db failed
with error -2
[ 4.849072][ T1] clk: Not disabling unused clocks
[ 4.853914][ T32] platform regulatory.0: Falling back to sysfs fallback for:
regulatory.db
[ 4.859364][ T1] ALSA device list:
[ 4.872602][ T1] No soundcards found.
[ 4.876823][ T1] Warning: unable to open an initial console.
[ 4.884190][ T1] Freeing unused kernel memory: 2048K
[ 4.890111][ C3] cryptomgr_probe (127): highest shadow stack usage: 136 bytes
[ 4.917486][ T1] Run /init as init process
[ 4.926370][ T1] init: init first stage started!
[ 4.932444][ T1] init: alias lines in modules.alias must have 3 entries, not 9
[ 4.940884][ T1] init: alias lines in modules.alias must have 3 entries, not 4
[ 4.950272][ T1] init: Loading module /lib/modules/rtc-sunxi.ko with args ''
[ 4.958436][ T1] rtc_sunxi: disagrees about version of symbol module_layout
[ 4.966401][ T1] init: Failed to insmod '/lib/modules/rtc-sunxi.ko' with args '':
Exec format error
[ 4.975719][ T1] init: LoadWithAliases was unable to load rtc_sunxi
```

```
[ 4.982650][ T1] init: Copied ramdisk prop to /second_stage_resources/system/etc/ramdisk/build.prop
[ 4.996840][ T1] init: Switching root to '/first_stage_ramdisk'
[ 5.004490][ T1] init: [libfs_mgr]ReadFstabFromDt(): failed to read fstab from dt
[ 5.013817][ T1] init: Using Android DT directory /proc/device-tree/firmware/android/
```

7.1.2 分析过程

- 1、**排查 symbol magic**: 从 log 得知，第一个 ko 就加载失败，我们首先考虑是 symbol magic 不一致导致（这部分的详细分析可以看“symbol magic 不一致导致 ko 加载失败”章节），然后首先对 gki_defconfig 文件和我们自己的 android defconfig 文件进行了对比，根据 gk_defconfig 文件的修改，更新 android defconfig 文件，修改后比对 Module.symvers 文件的 module_layout 参数，结果还是不一样；
- 2、**更新 abi_gki_whitelist**: 继续排查 symbol magic，为了和 Google GKI 编译内容保持一致，defconfig 中打开如下配置：

```
CONFIG_TRIM_UNUSED_KSYMS
```

```
CONFIG_UNUSED_KSYMS_WHITELIST="./abi_gki_whitelist"
```

这部分的内容在“export 接口被优化问题”章节有如下的介绍：

Google在编译GKI时打开了CONFIG_TRIM_UNUSED_KSYMS，这个选项会在编译GKI镜像时对export symbol进行优化，使得很多本来是export出来的接口变成了非export，而GKI whitelist则是强制export，不让其优化whitelist中的接口。而我们一般不打开这个选项，即使打开了，由于编译内容不同，也会存在差异。

然后我们将 aosp 的abi_symlist.raw文更改更新至我们的abi_gki_whitelist文件中，然后测试启动即可，对应的 patch 如下：

```
http://gerrit.allwinnertech.com:8081/#/c/lichee/linux-5.4/+162417/2/abi\_gki\_whitelist
```

然后启动验证 GKI，出现如下字样的 log：

```
[ 5.336277] ehci-hcd.ko: Unknown symbol ehci_cf_port_reset_rwsem (err -2)
[ 5.343984] ehci-hcd.ko: Unknown symbol usb_for_each_dev (err -2)
[ 5.351439] ehci-hcd.ko: Unknown symbol usb_hcd_end_port_resume (err -2)
[ 5.358199] ehci-hcd.ko: Unknown symbol usb_hcd_start_port_resume (err -2)
[ 5.364532] ehci-hcd.ko: Unknown symbol usb_hub_clear_tt_buffer (err -2)
```

这是典型的白名单缺失问题，继续分析如下：

- 3、**排查白名单**: 我们进一步利用白名单检查工具检查接口是否全部被 export，结果如下：

```
Symbol ehci_cf_port_reset_rwsem required by ehci-hcd.ko but not provided
Symbol usb_for_each_dev required by ehci-hcd.ko but not provided
Symbol usb_hcd_end_port_resume required by ehci-hcd.ko but not provided
```

```
Symbol usb_hcd_start_port_resume required by ehci-hcd.ko but not provided
Symbol usb_hub_clear_tt_buffer required by ehci-hcd.ko but not provided
```

```
# 进一步查找了涉及的文件对应的defconfig
drivers/usb/core/hub.c      usbcore-y
drivers/usb/core/port.c     usbcore-y
drivers/usb/core/hcd.c      usbcore-y
drivers/usb/core/hcd.c      usbcore-y
drivers/usb/core/hub.c      usbcore-y
```

继续查找这些文件，发现全部被 built-in 编译进内核，这时候按照常规的做法，我们会将这五个被 ehci-hcd.ko 需要的接口添加到白名单，但是此次有所不同，我们查看了近期的 gki_defconfig 修改记录得知，这个 ehci-hcd.ko 被 aosp 提了补丁 built-in 了，patch 如下：

```
commit 95f26af6ee7dc5cc09a13c787291ae6edd8c6932
Author: zhang sanshan <pete.zhang@nxp.com>
Date: Tue Jun 30 16:21:38 2020 +0800
```

```
ANDROID: GKI: support CONFIG_USB_EHCI_HCD
```

```
it use CHIPIDEA usb controller on imx8mm-evk board.
Need enable device/host mode for android device by default.
host mode depend on CONFIG_USB_CHIPIDEA_HOST which need enable CONFIG_USB_EHCI_HCD by default.
CONFIG_USB_EHCI_HCD is used for USB 2.0 "high speed" protocol.
```

```
Bug: 160193672
Change-Id: Ic635ae6c12397ed7d2060ab68fec3853fadfa6ec
Signed-off-by: zhang sanshan <pete.zhang@nxp.com>
Signed-off-by: Greg Kroah-Hartman <gregkh@google.com>
```

```
diff --git a/arch/arm64/configs/gki_defconfig b/arch/arm64/configs/gki_defconfig
index 177c40babb05..edc1c97ba2d1 100644
--- a/arch/arm64/configs/gki_defconfig
+++ b/arch/arm64/configs/gki_defconfig
@@ -363,6 +363,7 @@ CONFIG_HID_SONY=y
 CONFIG_HID_STEAM=y
 CONFIG_USB_HIDDEV=y
 CONFIG_USB_OTG=y
+CONFIG_USB_EHCI_HCD=y
 CONFIG_USB_GADGET=y
 CONFIG_USB_CONFIGFS=y
 CONFIG_USB_CONFIGFS_UEVENT=y
diff --git a/arch/x86/configs/gki_defconfig b/arch/x86/configs/gki_defconfig
index 7fafb9e80aaf..c5465c20199e 100644
--- a/arch/x86/configs/gki_defconfig
+++ b/arch/x86/configs/gki_defconfig
@@ -318,6 +318,7 @@ CONFIG_HID_PLANTRONICS=y
 CONFIG_HID_SONY=y
 CONFIG_HID_STEAM=y
 CONFIG_USB_HIDDEV=y
+CONFIG_USB_EHCI_HCD=y
 CONFIG_USB_GADGET=y
 CONFIG_USB_CONFIGFS=y
 CONFIG_USB_CONFIGFS_UEVENT=y
```

由于白名单是为未编译进内核的 ko 驱动提供的，所以我们不再需要将这个函数添加到白名单，导出函数，仅需要同步这个补丁到我们的 android defconfig 之中，将这个模块 built-in 即可，详

见如下补丁：

http://gerrit.allwinnertech.com:8081/#/c/lichee/linux-5.4/+162416/2/arch/arm64/configs/sun50iw10p1smp_a100_android_defconfig

同时在将模块 built-in 之后，需要将这个模块从 android 的 ko 加载文件中去除，对应的 patch 如下：

<http://gerrit.allwinnertech.com:8081/#/c/platform/tidy/device/softwinner/ceres/+162414/>

然后启动验证 GKI，启动成功，验证通过。




著作权声明

版权所有 © 2022 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

、 **全志科技** （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。