构建属于你自己的内核镜像

利用 "Github Actions" 进行内核云编译

@easterNday

仓库地址： https://github.com/DogDayAndroid/Android-Builder

博观而约取，厚积而薄发

目录

CONTENTS

1 什么是内核?

2 如何获取到设备对应的内核源码?

3 如何选取编译工具链?

4 如何编译内核?

5 如何将内核进行打包?

6 如何利用 Github Action 完成内核编译?

7 如何开启 KernelSU 支持?

8 如何开启 Docker 支持?

9 本项目参考了哪些项目?

Android 内核简介

内核之于 Android

> 一个完整的操作系统，包括内核，功能库，用户界面三个主要部分。

严格来说，Android 并不是 Linux 的一个发行版，但是 Linux 内核对于 Android 来说必不可少。

Android 设备的启动分为三个阶段：

- **Bootloader:** 设备打开电源后，首先会从处理器的片上 `ROM` 的启动引导代码开始执行，寻找 `Bootloader` 代码，并加载到内存并完成硬件的初始化。
- **Linux Kernel:** 硬件初始化后， `Linux` 内核代码加载到内存，初始化各种软硬件环境，加载驱动程序，挂载根文件系统。
- **Android 系统服务:** 并执行 `init.rc` 通知 `Android` 启动。

因此， `Android` 系统实际上是运行在 `Linux Kernel` 之上的一系列系统服务进程。

内核源码寻找

设备代号查询

> 编译内核时设备代号通常是必不可少的内容

在安卓设备终端（`adb shell`）上执行:

```
getprop | grep device
```

在返回的结果中寻找带有 `ro.xx.device` 字样的文本，里面的内容即为你的设备代号，例如：

```
# 手机代号为 thyme
[ro.product.device]: [thyme]
```

例如，我对 **小米 10S** 进行查询：

```
› adb shell
thyme:/ $ getprop | grep device
[bluetooth.device.class_of_device]: [90,2,12]
[bluetooth.profile.hid.device.enabled]: [true]
[cache_key.bluetooth.bluetooth_device_get_bond_state]: [-1781083723495810673]
[cache_key.system_server.device_policy_manager_caches]: [-1749000785485656421]
[debug.tracing.battery_stats.device_idle]: [0]
[debug.tracing.device_state]: [0:DEFAULT]
[ro.boot.boot_devices]: [soc/1d84000.ufshc]
[ro.boot.bootdevice]: [1d84000.ufshc]
[ro.frp.pst]: [/dev/block/bootdevice/by-name/frp]
[ro.lineage.device]: [thyme]
[ro.opa.eligible_device]: [true]
[ro.product.bootimage.device]: [thyme]
[ro.product.device]: [thyme]
[ro.product.mod_device]: [thyme_global]
[ro.product.odm.device]: [thyme]
[ro.product.product.device]: [thyme]
[ro.product.system.device]: [thyme]
[ro.product.system_ext.device]: [thyme]
[ro.product.vendor.device]: [thyme]
[ro.product.vendor_dlkm.device]: [thyme]
[sys.usb.mtp.device_type]: [3]
```

获取设备架构

> 设备架构在编译过程中也是必须的一环

查询设备架构的方法也很简单。

在安卓设备终端（`adb shell`）上执行:

```
uname -m
```

例如，我对 **小米 10S** 进行查询：

```
thyme:/ $ uname -m
aarch64
```

其设备架构显示为 `aarch64`，也即是我的设备是 `aarch64` 架构。

获取设备内核版本

在安卓设备终端（`adb shell`）上执行:

```
uname -r
```

输出内容的格式为：

- [版本].[补丁版本].[子版本号]-[内核标识]-[提交记录]

例如，我对 **小米 10S** 进行查询：

```
thyme:/ $ uname -r
4.19.157-Margatroid-g2b220a0a942c
```

其对应的内核版本为显示为 `4.19.157`

内核源码获取

内核源码的一般格式为 [android_]kernel_设备厂商_cpu/代号。

例如，小米 10S 的代号为 thyme，CPU 型号为 sm8250，生产厂商为 xiaomi，则搜索格式应为下面几种：

- kernel_xiaomi_thyme
- kernel_xiaomi_sm8250
- android_kernel_xiaomi_thyme
- android_kernel_xiaomi_sm8250

用以上关键词在 Github 上进行搜索一般都可以找到对应的源码。

除此之外,我们还可以通过各手机厂商开源的代码来进行获取。

途径	具体介绍
小米内核开源	小米内核开源
华为开源代码	华为开源代码
去手机社区找源码	XDA 论坛

~~值得注意的是，通常来说，各手机厂商代码都会经过阉割，并不能开箱即用。~~

交叉编译器的选择

交叉编译的定义

什么是交叉编译（Cross Compile）？

- 所谓 "**交叉编译**"，是指编译源代码的开发编译平台和执行源代码编译后程序的目标运行平台是两个不同的平台。

为什么要使用交叉编译呢?

- 目标平台上无法实现本地编译(Native Compile)，但有能力实现源代码编译的平台 CPU 架构或操作系统与目标平台不同。

各种 CPU 架构

通过之前的内容，我们获取到了我们设备的 CPU 架构，这些架构可以归类为如下几种：

- **arm:** 32 位的 Arm 架构，包括 `arm`， `arm32`， `armv7`
- **arm64:** 64 位的 Arm 架构，包括 `arm64`， `aarch64`
- **x86:** 32 位的 Intel x86 架构，包括 `x86`， `i386`， `i686`
- **x86_64:** 64 位的 Intel x86 架构，包括 `x86_64`， `amd64`

一般来说，我们的手持设备的型号一般为 `arm` 和 `arm64` 中的某一种，当然，也不排除有其他的情况（凤凰 OS 等电脑上使用的 Android 系统是 x86 或 x86_64 架构）。

交叉编译工具链的三元组格式

一般来说，交叉编译连工具的二进制文件命名规则为： `{arch}-{vendor}-{sys}-{abi}`

他们分别对应如下内容：

- **arch:** 对应的设备架构;
- **vendor:** 供应商名称，一般来说会被省略或没有;
- **sys:** 对应的系统，`Android` 编译时一般都为 `linux`;
- **abi:** 两个程序单元之间的二进制接口，`Android` 编译时一般都为 `gnu` 或 `gnueabi` 等。

有些编译工具链中会省略 `vendor` 或 `abi` 部分，`Android` 编译时一些常见的命名规则是：

- aarch64-linux-gnu-
- arm-linux-gnueabi-
- aarch64-linux-android-
- arm-linux-androideabi-

交叉编译工具链的选取

> 使用不匹配的编译器，轻则无法开机，重则编译失败。

构建安卓内核的工具，只能从安卓源码里面拉取，而且有版本限制，太新不行，太老了也不行。

一般来说，2023 年的机型的编译工具链应该都是 `clang` + `gcc` 的形式，但对于一些老机型，可能只支持 `gcc` 编译，下面是一些设备的说明：

CAF 机型（除谷歌外的高通设备）：

- `3.18`、`4.4`、`4.9` 版本的内核，默认均使用谷歌的 `gcc`
- `4.14` 及以上版本的内核，默认使用 `clang` + `gcc`

谷歌机型（Pixel 系列）：

- 从 `Pixel2` 开始使用 `clang` 编译，`Pixel3` 开始使用 `clang` 的 `LTO` 优化

交叉编译工具链的下载

一般来说，我们可以通过第三方系统的仓库（例如：`LineageOS`、`PixelExperience`）或者 `Google` 官方拉取、下载我们需要的交叉编译工具链。

LineageOS:

- **Clang:** LineageOS/android_prebuilts_clang_kernel_linux-x86_clang-r416183b
- **Gcc for ARM64:** LineageOS/android_prebuilts_gcc_linux-x86_aarch64_aarch64-linux-android-4.9
- **Gcc for ARM:** LineageOS/android_prebuilts_gcc_linux-x86_arm_arm-linux-androideabi-4.9

Google:

- **Clang:** platform/prebuilts/clang/host/linux-x86
- **Gcc for ARM64:** platform/prebuilts/gcc/linux-x86/aarch64/aarch64-linux-android-4.9
- **Gcc for ARM:** platform/prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-4.9

需要注意，以上只是示例，如果内核编译出现问题，那么可能需要您更换具体的 `clang` 和 `gcc` 版本。

内核编译

环境准备

我们假定您本地正在使用 `Ubuntu` 进行内核编译的开发，我们需要安装一些必要的编译工具：

```
sudo apt update
sudo apt-get install -y bc bison build-essential ccache curl flex g++-multilib gcc-multilib git git-lfs gnupg gperf \
    lib32ncurses5-dev lib32readline-dev lib32z1-dev libelf-dev liblz4-tool libncurses5 libncurses5-dev \
    libsdl1.2-dev libssl-dev libxml2 libxml2-utils lzop pngcrush rsync \
    schedtool squashfs-tools xsltproc zip zlib1g-dev imagemagick
```

或者您使用 `ArchLinux` 进行开发，您只需要安装：

```
paru -S aosp-devel linageos-devel
```

内核源码克隆

要编译内核，我们首先就需要我们对应设备的内核源码克隆下来，此处以**小米 10S**为例。

我选择的仓库是 https://codeberg.org/DogDayAndroid/android_kernel_xiaomi_thyme

在终端执行如下指令:

```
git clone https://codeberg.org/DogDayAndroid/android_kernel_xiaomi_thyme
```

等待仓库拉取完毕，即可完成内核源码的克隆。

检查内核源码中设备的 defconfig

在**内核源码寻找**部分中，我们已经获取到了设备的**代号**、**架构**，一般来说配置文件会按照这些内容来定义。

例如，我的**小米 10S**的代号为**thyme**，架构为**aarch64**，也即是**arm64**，具体架构参考下方:

- **arm:** 32 位的 Arm 架构，包括 `arm`， `arm32`， `armv7`
- **arm64:** 64 位的 Arm 架构，包括 `arm64`， `aarch64`

一般来说， `defconfig` 文件位于如下路径中：

- `<内核源码文件夹>/arch/<设备架构>/configs/`
- `<内核源码文件夹>/arch/<设备架构>/configs/vendor/`

我刚克隆的内核源码中，**thyme**对应的 `defconfig` 文件路径为 `arch/arm64/configs/thyme_defconfig`，请确保您能够找到您的 `defconfig` 文件路径。

交叉编译工具链的配置

此处我使用**LineageOS**的交叉编译工具链来进行演示：

- **Clang:** LineageOS/android_prebuilts_clang_kernel_linux-x86_clang-r416183b
- **Gcc for ARM64:** LineageOS/android_prebuilts_gcc_linux-x86_aarch64_aarch64-linux-android-4.9
- **Gcc for ARM:** LineageOS/android_prebuilts_gcc_linux-x86_arm_arm-linux-androideabi-4.9

终端中执行:

```
git clone https://github.com/LineageOS/android_prebuilts_clang_kernel_linux-x86_clang-r416183b clang
git clone https://github.com/LineageOS/android_prebuilts_gcc_linux-x86_aarch64_aarch64-linux-android-4.9 gcc64
git clone https://github.com/LineageOS/android_prebuilts_gcc_linux-x86_arm_arm-linux-androideabi-4.9 gcc32
```

随后将编译工具链临时加入我们的环境变量中：

```
export PATH="$PWD/clang/bin:$PWD/gcc64/bin:$PWD/gcc32/bin:$PATH"
```

开始构建内核

首先我们需要了解一下我们构建的参数：

参数	说明	一般参数
CC	指定使用的编译器，因为 make 默认使用 gcc，因此实际上只有你在使用 clang 进行编译的时候才会使用该参数	clang
CROSS_COMPILE	您的主要交叉编译链工具，如果你只使用 gcc 进行编译，请指定参数为 aarch64-linux-android-，32 位同理	aarch64-linux-gnu-
CLANG_TRIPLE	只在使用 clang 进行编译的时候才需要使用，用于指定当 clang 不生效时候使用的工具链，但在使用上一节我们提到的工具中基本不用设置该参数	aarch64-linux-gnu-
CROSS_COMPILE_ARM32	只在编译 32 位内核或者带 vdso 补丁的内核时需要指定该参数	arm-linux-gnueabi-
CROSS_COMPILE_COMPAT	类似于参数 CROSS_COMPILE_ARM32 ，但内核版本为 4.19 及更新版本应使用本参数而非 CROSS_COMPILE_ARM32	arm-linux-gnueabi-

开始构建内核

下面我给出一个一般的构建脚本：

```
#!/bin/bash
args="-j$(nproc --all) O=out ARCH=arm64 CC=clang CLANG_TRIPLE=aarch64-linux-gnu- CROSS_COMPILE=aarch64-linux-android- \
    CROSS_COMPILE_ARM32=arm-linux-androideabi- LD=ld.lld AR=llvm-ar NM=llvm-nm OBJCOPY=llvm-objcopy OBJDUMP=llvm-objdump READELF=llvm-readelf \
    OBJSIZE=llvm-size STRIP=llvm-strip LDGOLD=aarch64-linux-gnu-ld.gold LLVM_AR=llvm-ar LLVM_DIS=llvm-dis"
make ${args} <configName>
make ${args}
```

其中 `ARCH=arm64` 是指定架构为 `arm64` 架构，其余参数作用均与上述表格中的内容对应。

其中 `<copnfigName>` 是我们的 `defconfig` 文件，例如：

- **小米 10S**对应的 `defconfig` 文件路径为 `arch/arm64/configs/thyme_defconfig`，此处便填写 `thyme_defconfig`;
- 部分项目可能会位于 `arch/arm64/configs/vendor/thyme_defconfig`，此时应该填写 `vendor/thyme_defconfig`。

这个构建脚本制定了架构为 `arm64` ，使用 `clang` 进行编译，并且指定了对应的交叉编译工具。

值得注意的是，这个脚本并不是通用的，具体的内核需要自己具体的分析和修改。

内核打包

获取内核编译产物

通过上节内容，我们可以完成内核编译。这时候我们需要考虑如何将内核进行打包。

一般来说，内核编译后的产物位于 `arch/<设备架构>/boot/` 文件夹中。

- 在编译产物中，可能存在`Image`、`Image.gz`、`Image.gz-dtb`等文件，而我们需要的是那个带 `dtb` 的文件，也就是 `Image.gz-dtb`。不过编译产物也可能是 `Image.lz4-dtb` 抑或 `Image-dtb` 这种，但总之，带上 `dtb` 总没错。
- 对于一些内核源码，其编译产物可能是 `Image` + `独立的 dtb 文件 / dtbo.img` 的形式，如果是这两个文件也是可以的。

Anykernel3

`Anykernel` 是一个最初由 `koush` 编写，后被 `osm0sis` 接手并多次迭代的内核刷写工具。

其项目地址为：https://github.com/osm0sis/AnyKernel3

我们首先将项目克隆下来并对脚本进行一些修改：

```
git clone https://github.com/osm0sis/AnyKernel3 AnyKernel3
# 修改脚本
sed -i 's/do.devicecheck=1/do.devicecheck=0/g' AnyKernel3/anykernel.sh
sed -i 's!block=/dev/block/platform/omap/omap_hsmmc.0/by-name/boot;!block=auto;!g' AnyKernel3/anykernel.sh
sed -i 's/is_slot_device=0;/is_slot_device=auto;/g' AnyKernel3/anykernel.sh
```

请注意，这些修改使得打包后的刷机包可以被任何设备刷入，是存在一定风险的，如果您需要安全的使用该刷机包，请自行查阅 `Anykernel3` 的官方文档。

刷机包打包及刷入

按照**获取内核编译产物**中的指导，我们将 `Image-dtb` 或者 `Image` + `dtb` 或者 `Image` + `dtbo.img` 复制到 `Anykernel3` 的文件夹中，随后利用 `zip` 进行压缩即可完成刷机的打包了。压缩命令如下：

```
cd AnyKernel3/
zip -q -r "AnyKnerl3.zip" *
```

打包完成后，我们就可以进入设备的 `TWRP` 将该压缩包刷入。

利用 Github Action 云编译内核

项目地址

https://github.com/DogDayAndroid/Android-Builder

目前，该项目将支持的内容如下：

- 利用 `Github Actions` 来进行内核构建
- 克隆本项目到本地后，您可以用脚本自动构建 `LineageOS`
- 利用 `Github Actions` 来进行 `TWRP` 构建

每个项目的具体内容会在每个文件夹下单独显示。 您可以进入这些文件夹查看其各自的自述文件以了解如何使用它们。

许可证

使用方法

- 在 GitHub 上 `fork` 本项目
- 通过 Github 网页或者拉取到本地修改 `config/*.config.json` 文件，并提交修改
- 查看 Github 网页的 `Action` 页面，找到 `Build kernels` 并 `Run workflow`
- 等待编译完成，即可进入对应页面下载编译产物

编译流程

配置参数解析

每个配置模板均由以下几个部分组成：

字段名称	描述
kernelSource	内核源代码的相关信息，包括名称、仓库地址、分支和设备类型。
toolchains	一个数组，包含了需要用到的工具链的相关信息，包括仓库地址、分支和名称。
enableCcache	一个布尔值，表示是否使用了名为 `ccache` 的编译工具来加速编译。
params	一个对象，包含了构建参数的相关信息，其中包括了架构类型、交叉编译器、编译器等信息。
AnyKernel3	一个对象，包含了构建内核刷机包的相关信息，其中包括了使用的 `AnyKernel3` 仓库地址、分支等信息。
enableKernelSU	一个布尔值，表示是否使用了名为 `KernelSU` 的内核补丁。
enableLXC	一个布尔值，表示是否开启 `Docker` 支持。

内核源码配置(kernelSource)

```
"kernelSource": {
  "name": "",     // 你喜欢的名称，无任何影响，一般设定为 设备名字+编译工具 链版本
  "repo": "",     // 内核源码的仓库地址
  "branch": "",   // 对应内核源码仓库的 分支 名称
  "device": "",   // 对应的设备编号
  "defconfig": ""   // 对应的 defconfig 文件相对路径
}
```

`name` 部分对于整个编译流程来说是没有影响的，因此理论上你可以随意设定。

`repo`，`branch` 用于克隆内核源码，我们会默认克隆源码下的所有子模块来保证内核的完整性。

`defconfig` 中填写的内容是您的 `defconfig` 文件相对于 `arch/arm64/configs` 或 `arch/arm/configs` 文件夹的相对路径，这样做的原因是因为部分 `defconfig` 文件可能会存在于子目录中，`make` 的时候我们需要显示指定他的相对路径。

内核源码配置(kernelSource)

下面是一个基本的例子:

```
"kernelSource": {
  "name": "Mi6X",
  "repo": "https://github.com/Diva-Room/Miku_kernel_xiaomi_wayne",
  "branch": "TDA",
  "device": "wayne",
  "defconfig": "vendor/wayne_defconfig"
}
```

这个内核是**小米 6X**的内核源码，网页打开其 `Github` 地址后，我们看到它的主要分支是 `TDA`，同时其 `defconfig` 文件位于 `/arch/arm64/configs/vendor/wayne_defconfig` 内，因此设定 `defconfig` 为 `vendor/wayne_defconfig` 。

工具链配置(toolchains)

交叉编译工具链是我们编译内核时的重要工具，但是编译工具链的下载形式五花八门，可以使用 `git` 拉取下载，也可以通过下载得到，因此对于不同的获取方式，我们分别作了适配：

- 使用 `Git` 拉取编译工具链
- 使用 `Wget` 下载编译工具链

工具链配置(toolchains) - 使用 Git 拉取编译工具链

```
"toolchains": [
  {
    "name": "proton-clang",
    "repo": "https://github.com/kdrag0n/proton-clang",
    "branch": "master",
    "binaryEnv": ["./bin"]
  }
]
```

这部分的配置其实类似于内核源码的配置，我们同样会使用如下命令来从仓库中拉取源码：

```
git clone --recursive --depth=1 -j $(nproc) --branch <branch> <repo> <name>
```

但是这部分中新增了一个 `binaryEnv`, 这是用于给我们的编译工具链添加全局环境变量设定的，例如此处的 `./bin`，添加内容后，环境变量中会增加编译工具链下的 `bin` 文件夹。

工具链配置(toolchains) - 使用 Wget 下载编译工具链

通过这种方式我们可以获取到 .zip | .tar | .tar.gz | .rar 格式的编译工具链压缩包。

```
"toolchains": [
  {
    "name": "clang",
    "url": "https://android.googlesource.com/platform/prebuilts/clang/host/linux-x86/+archive/refs/heads/master-kernel-build-2022/clang-r450784d.tar.gz",
    "binaryEnv": ["./bin"]
  },
  {
    "name": "gcc",
    "url": "https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/aarch64/aarch64-linux-android-4.9/+archive/refs/tags/android-12.1.0_r27.tar.gz",
    "binaryEnv": ["bin"]
  }
]
```

Action 会下载并解压他们，同时这部分中也会有 binaryEnv, 其作用和上述作用类似，因此不再赘述。

工具链配置(toolchains)

这两种方式并非鱼与熊掌不可得兼的，如果我们既需要 `Git` 拉取又需要 `Wget` 下载的编译工具链，我们也可以像如下配置一样将其混用：

```
"toolchains": [
  {
    "name": "clang",
    "repo": "https://gitlab.com/ThankYouMario/android_prebuilts_clang-standalone/",
    "branch": "11",
    "binaryEnv": ["bin"]
  },
  {
    "name": "gcc",
    "url": "https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/aarch64/aarch64-linux-android-4.9/+archive/refs/tags/android-12.1.0_r27.tar.gz",
    "binaryEnv": ["bin"]
  }
]
```

编译参数(params)

通常我们在本地进行内核编译的时候，会使用形似如下的编译命令：

```
make -j$(nproc --all) \
    O=out \
    ARCH=arm64 \
    CC=clang \
    CLANG_TRIPLE=aarch64-linux-gnu- \
    CROSS_COMPILE=aarch64-linux-gnu- \
    CROSS_COMPILE_ARM32=arm-linux-gnueabi-
```

编译参数(params)

因此，我们的编译参数配置也以类似的方式来进行配置：

```
"params": {
  "ARCH": "arm64",
  "CC": "clang",
  "externalCommands": {
    "CLANG_TRIPLE": "aarch64-linux-gnu-",
    "CROSS_COMPILE": "proton-clang/bin/aarch64-linux-gnu-",
    "CROSS_COMPILE_ARM32": "proton-clang/bin/arm-linux-gnueabi-"
  }
}
```

其中 `-j` 和 `O=out` 这一部分会由编译脚本自动配置好，因此配置中并不用进行设置。`ARCH` 以及 `CC` 部分对应上面的指令部分，其他的更多参数则对应 `externalCommands` 部分。

编译参数(params)

```
# Template for markw(Xiaomi4)
"params": {
  "ARCH": "arm64",
  "CC": "clang",
  "externalCommands": {
    "CLANG_TRIPLE": "aarch64-linux-gnu-",
    "CROSS_COMPILE": "aarch64-linux-android-",
    "CROSS_COMPILE_ARM32": "arm-linux-androideabi-",
    "LD": "ld.lld",
    "AR": "llvm-ar",
    "NM": "llvm-nm",
    "OBJCOPY": "llvm-objcopy",
    "OBJDUMP": "llvm-objdump",
    "READELF": "llvm-readelf",
    "OBJSIZE": "llvm-size",
    "STRIP": "llvm-strip",
    "LDGOLD": "aarch64-linux-gnu-ld.gold",
    "LLVM_AR": "llvm-ar",
    "LLVM_DIS": "llvm-dis",
    "CONFIG_THINLTO": ""
  }
}
```

开启 ccache 加速编译

内核编译过程中，如果反复的编译会非常耗费我们的时间，ccache 使得我们可以可以复用以前编译时的一些中间件的缓存从而加快编译的速度，例如上一小节中的编译命令开启 ccache 后应为:

```
make -j$(nproc --all) \
    O=out \
    ARCH=arm64 \
    CC="ccache clang" \
    CLANG_TRIPLE=aarch64-linux-gnu- \
    CROSS_COMPILE=aarch64-linux-gnu- \
    CROSS_COMPILE_ARM32=arm-linux-gnueabi-
```

开启 ccache 加速编译

这样我们就引出了一个单独的配置参数 `enableCcache`，我们只需要在配置的时候将 `enableCcache` 设置为 `true` 即可实现同样的命令：

```
"enableCcache": true,
"params": {
  "ARCH": "arm64",
  "CC": "clang",
  "externalCommands": {
    "CLANG_TRIPLE": "aarch64-linux-gnu-",
    "CROSS_COMPILE": "aarch64-linux-android-",
    "CROSS_COMPILE_ARM32": "arm-linux-androideabi-"
  }
}
```

内核刷机包配置(AnyKernel3)

目前本项目仅支持 `AnyKernel3`，其配置如下:

```
"AnyKernel3": {
  "use": true,
  "release": true,
  "custom": {
    "repo": "https://github.com/easterNday/AnyKernel3/",
    "branch": "thyme"
  }
}
```

内核刷机包配置(AnyKernel3)

这段配置中，我使用了自定义的 `AnyKernel3` 来进行打包，如果您不想额外的 `fork` 一个仓库来实现的话，可以选择删除 `custom` 字段来使用原版的 `AnyKernel3` 来打包您的内核，删除后的配置如下：

```
"AnyKernel3": {
  "use": true,
  "release": true
}
```

配置中的 `use` 表示您是否使用 `AnyKernel3` 来进行打包， `release` 表示您是否将打包后的刷机包发布出来，`release` 当且仅当 `AnyKernel3` 设置为 `true` 的时候才生效，否则默认为 `false`。

额外的编译参数设定

KernelSU

使用 `"enableKernelSU": true,` 来控制是否启用 `KernelSU`，设置为 `false` 则不启用。

LXC Docker

使用 `"enableLXC": false` 来控制是否启用 `Docker` 支持，设置为 `true` 则启用。

教程及参考内容

教程

- 自己编译定制一个牛逼的安卓内核
- 让 Android 手机更省电流畅，你可以试试「刷内核」
- [内核向] 交叉编译器的选择
- [白话文版] ClangBuiltLinux Clang 的使用
- Neutron-clang 的编译说明
- [内核向] 论如何优雅的刷入内核

参考

- DogDayAndroid/KSU_Thyme_BuildBot：我自己编译的内核使用的本地编译脚本。
- UtsavBalar1231/Drone-scripts：一个很多人使用的编译脚本，我的部分代码也是参考自这里。
- EndCredits/kernel_xiaomi_sm7250：同样的一个编译脚本，但并未提供编译链，但是其中的脚本流程我也有参考。
- xiaoleGun/KernelSU_Action：`KernelSU` 的编译脚本，同样有参考。

～ 欢迎交流 ～