内核开发入门

- https://www.kernel.org/doc/html/latest/
  - linux内核开发官方文档
- https://linuxtools-rst.readthedocs.io/zh_CN/latest/base/index.html
  - 官方文档的一部分，**快速教程**
- https://elixir.bootlin.com/
  - 源码阅读
- https://command-not-found.com/
- https://wiki.osdev.org/Main_Page

容易忘记的tips

1. **grub —— GRand Unified Bootloader**
   - BIOS先加载grub，grub选择OS
   - 选择**不同的OS**以及**向内核提供启动参数**
2. 内核启动的参数
   - **single:单用户模式，正常模式进不去的时候可以尝试一下单用户模式**
     - **只加载部分内容**
   - **quiet:静默模式，不会显示log** and **splash:闪屏**
     - **这俩一起用的，没有log就相当于有一个进入动画**
3. **启动log**观察
   - **journalctl**
4. **linux 内核 log级别**
   - ref
     - https://www.linuxtopia.org/online_books/linux_kernel/kernel_configuration/re06.html
       1. 0 (KERN_EMERG)
       2. 1 (KERN_ALERT)
       3. 2 (KERN_CRIT)
       4. 3 (KERN_ERR)
       5. 4 (KERN_WARNING)
       6. 5 (KERN_NOTICE)
       7. 6 (KERN_INFO)
       8. 7 (KERN_DEBUG)
   - 越小越危险，**比指定的小就能打印出来**

linux基础知识

Filesystem Hierarchy Standard

- linux各个目录的用法标准
- https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.pdf
  - /etc
    - Host-specific (特定于主机的) system configuration
      - /etc hierarchy contains configuration files
      - A **configuration file** is a local file used to control the operation of a program; it must be **static** and **cannot be an executable binary**
      - It is recommended that files be stored in subdirectories of /etc rather than directly in /etc

- /media
  - Mount point for **removable media**
- /mnt
  - Mount point for a temporarily mounted filesystem
  - This directory is provided so that the system administrator may temporarily mount a filesystem as needed
    - 比如创建一个 `/mnt/ramdisk`，然后挂载一个 NOVA fs
- /home : User home directories (optional)
- /root : Home directory for the root user (optional)

apt 包管理器

- `apt` 命令是 Debian Linux 发行版中的APT**软件包管理工具**。所有**基于Debian的发行**都使用这个包管理系统
  - 我们常用的 ubuntu 就是 Debian 的一个发行版本
- 包是哪里来的呢
  - 使用 apt-get 命令的第一步就是引入必需的软件库，Debian 的软件库也就是所有 Debian 软件包的集合，它们存在互联网上的一些公共站点上。把它们的地址加入 `/etc/apt/sources.list`，apt-get就能搜索到我们想要的软件。`/etc/apt/sources.list` 是存放这些地址列表的配置文件
  - `/etc/apt/source.list`
    - deb官方统一管理包的位置，其实内含的就是一些**资源网站**
      - 每一个**linux的发行版**都会维护一个自己软件仓库
      - 类似于苹果的 app 商店或 Ubuntu 的 Ubuntu software
    - **换源或添加源**说的都是对该文件的操作

apt-get update

- 检索最新的软件包并更新到本地
  - 比如 linux source 更新到 5.0 了。但是本地的软件 list 如果不更新的话只能显示到 4.8
  - 如果不 update 将无法看到最新的软件发布情况，**所以需要定期执行**

apt-get upgrade

- 就是执行更新的操作
- 如果当前软件版本比 source list 中的版本旧，则执行更新
  - 所以 upgrade 之前都要执行 update。否则可能什么都不会更新

apt-get install

- 基本步骤
  1. 先读 `/etc/apt/source.list` 查看目标软件是否存在
     - 没有或太慢就可能需要**换源**操作
  2. 亲切的告诉你当前机器中可能有很多东西用不着了
     - `sudo apt-get autoremove`
  3. 构建依赖
  4. 安装依赖，并最终安装软件
- apt-get被锁的解决方案
  - 如果遇到错误 `could not get lock /var/lib/dpkg/lock`，则说明有进程在使用 `apt-get`，需要找到它并kill它
    - `ps -ef|grep apt`
  - 但是有时候并不能解决问题，需要继续执行以下的命令
    - `sudo rm /var/cache/apt/archives/lock`
    - `sudo rm /var/lib/dpkg/lock`

apt-cache，queries and displays information

- `apt-cache show`
- `apt-cache search`

apt-get --help 善用

- archive files 指的是安装包

aptitude

- 实际上就安装与删除来说，**aptitude**是一个更好的选择，其在处理依赖关系上非常好。其基本命令与apt无异

dpkg

- dpkg 本身是一个底层的工具。上层的工具，如 apt 或 aptitude，被用于从远程获取软件包以及处理复杂的软件包关系
- `dpkg` 是 **Debian Package** 的简写
- dpkg -l
  - list 所有安装的包
- `.deb` 的安装
  - deb是Debian软件包格式，文件扩展名为 `.deb` ，处理这些包的经典程序是**dpkg**，经常是通过apt来运作
  - `dpkg -i xxx.deb`
    - `-i means install`

常用操作以及一些tips

tips

- Linux内核是模块化组成的，允许内核在运行时动态的向其中**插入**或**删除**代码
- 安装与加载是两回事
  - 安装只是将 `.so` 或可执行的 elf 文件挪到恰当的位置，之后可以直接命令行启动而已
  - 加载是将一个模块 load 到 run time 的内核中，runtime
- 支持内核开发的源码树位于 `/usr/src/` 下
- 发行版是没有内核源码的，内核源码树仅包含头文件，**Makefile以及kconfig等**
  - 利用这个头文件包已经 **足够** 为内核开发第三方的模块
    - 如果需要 include 发行版未提供的 `.h` 文件，就要自己考虑一下了
    - 把源码拷贝到仅含头文件（以及makefile与kconfig）的源码树下就有点**日怪**了
- 如果自己开发环境的 `linux-header` **GG了**就需要重新安装**linux header文件**
  - ref
    - https://stackoverflow.com/questions/50361990/how-to-solve-kernel-configuration-is-invalid-issues
      - 我貌似知道我是怎么把它干崩的了，尝试用其它路径下的makefile去编译ko
  - `apt install --reinstall linux-headers-$(uname -r)`
    - 这个操作完 `/lib/modules/5.15.0-60-generic` 下面没有build的符号链接，需要额外的操作
      - `sudo apt-get install build-essential`
  - `reboot`

常用操作

- 查看当前已经内置的模块
  - `cat /lib/modules/$(uname -r)/modules.builtin | more`
  - `grep "=y" /boot/config-$(uname -r) | more`
- 查看当前系统已安装的版本

- `dpkg --list | grep linux-image`
- `dpkg --get-selection | grep linux`
  - 只能看到发行版的，源码安装的是无法看到的

Kconfig

```
# drivers 的顶层 Kconfig
# 实际上就是在 source 底层的 Kconfig
# menu会对应 menuconfig 中的 -->，对应一个目录
# SPDX-License-Identifier: GPL-2.0
menu "Device Drivers"
...
source "drivers/pci/Kconfig"
...
source "drivers/char/Kconfig"
endmenu
```

- kconfig 是一个层次式的关系
  - menu 会对应一个目录
  - menuconfig 会对应一个配置的依赖关系
- Kconfig 有错的话，会报在 make menuconfig 的时候
- source
  - **excute commands from a file in the current shell**

kernel Makefile (kbuild)

- **obj-y list**
  - 就是最终的目标 vmlinux 需要 link 的 **.o** list
- **obj-m list**

overview

- The Makefiles have five parts:
  - `Makefile`
    - the top Makefile
  - `.config`
    - the kernel configuration file
  - `arch/$(SRCARCH)/Makefile`
    - the arch Makefile
  - `scripts/Makefile.*`
    - common rules etc. for all kbuild Makefiles
  - `kbuild Makefiles`
    - exist in every subdirectory
    - 子目录下的那个 Makefile 是按照 kbuild 的语法规则编写的
      - The preferred name for the kbuild files are **Makefile** but **Kbuild** can be used and if both a **Makefile** and a **Kbuild** file exists, then the **Kbuild** file will be used
- top makefile 根据当前的体系结构，根据用户配置的 config，递归到每一个子目录根据 kbuild 最终生成俩结果
  - vmlinux (`exe`)
  - modules (各个 `.so`)

obj-y and obj-m

```
obj-y += foo.o
# vmlinux will link foo.o

obj-$(CONFIG_FOO) += foo.o
# $(CONFIG_FOO) evaluates to either y (for built-in) or m (for module). If CONFIG_FOO is neither y nor m, then the file will not b

#fs/ext2/Makefile
obj-$(CONFIG_EXT2_FS) += ext2.o
ext2-y := balloc.o dir.o file.o ialloc.o inode.o ioctl.o namei.o super.o symlink.o
ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o xattr_user.o xattr_trusted.o
# In this example, xattr.o, xattr_user.o and xattr_trusted.o are only part of the composite object ext2.o if $(CONFIG_EXT2_FS_XAT
```

- kbuild会生成所有的 `.o` in obj-y list 然后 link 到 **built-in.a**
  - 不包含符号表
  - link 的顺序很重要，**说白了内核启动的顺序是依赖这个的**
    - obj-y 本质上是一个 list
      - `obj-y = a.o b.o c.o d.o`
    - because certain functions **(module_init() / __initcall)** will be called during boot **in the order they appear**
- **$(obj-m)** specifies object files which are built as loadable kernel modules
  - 一个模块可能会包含多个源文件，这种情况下，kbuild允许指定模块名
  - 模块名为 ext2，但是没有一个 `.c` 叫 `ext2.c`，就是这个意思

Descending（下降） down in directories

- The build system will automatically invoke make recursively in subdirectories, provided you let it know of them. To do so, obj-y and obj-m are used. ext2 lives in a separate directory, and the Makefile present in fs/ tells kbuild to descend down using the following assignment.

```
#fs/Makefile
obj-$(CONFIG_EXT2_FS) += ext2/
# If CONFIG_EXT2_FS is set to either 'y' (built-in) or 'm' (modular) the corresponding obj- variable will be set, and kbuild will
```

- When Kbuild descends into the directory with **y**
  - 下面的内容依然可以配置，可以进入 vmlinux，也可以进入 modules
- When Kbuild descends into the directory with **m**
  - 下面的内容只能进入 modules
- kernel top makefile
  - http://blog.sina.com.cn/s/blog_87c063060101l23z.html
    - top makefile 的一个中文解析

```
# make, make vmlinux and modules
# make modules, make modules
# make module_install, 将相应的模块拷贝到对应的目录下，如果是外部模块的话，就是把`.ko`拷贝到`/lib/modules/$(uname -r)/extra`下面
# make install, 该命令的作用是将.config, vmlinuz, initrd.img, System.map文件到/boot/目录、更新grub。且新内核为默认启动内核

# 这是第一个目标
_all
...
# 如果 make 的 commandline 中有 M, 则
KBUILD_EXTMOD:=$(M)
...
ifeq($(KBUILD_EXTMOD),)  # 参数相等为true, 否则为false
_all:all  # KBUILD_EXTMOD 为空
else
_all:modules
endif
...
all: vmlinux
all: modules  # by default, build modules as well
...
vmlinux:...
modules:...
```

linux内核编译安装相关操作

```
# download linux source in ~/kernelDevTree
mkdir ~/kernelDevTree
tar xvJf linux-***.tar.xz
cd linux-***
make mrproper # 拿到一个新的内核环境需要clear的时候可以执行该命令 即 make mrproper
make O=~/kernelDevTree/build/kernel defconfig  # 功能略少, 最好 cp 当前的 config
make O=~/kernelDevTree/build/kernel menuconfig
make O=~/kernelDevTree/build/kernel -jX
sudo make O=~/kernelDevTree/build/kernel modules_install install  # sudo
sudo update-grub  # 安装会自动执行的可选
```

内核源码的安装

- `/usr/src` 下包含了不同版本的是**内核源码树**
  - 内核源代码通常都会安装到 `/usr/src/linux` 下，但在开发的时候最好不要使用这个源代码树
    - 因为当前系统在用
      - `../($shell uname-r)/..`

下载源码以得到一个内核源码树

- apt-get
  - `sudo apt-get install linux-source`
    - 安装与当前运行环境一致的内核版本
    - 下载一个 `***.tar.bz2` 压缩包到 `/usr/src` 目录下
      - `sudo tar jxkf linux-source-4.15.0.tar.bz2`
- 自行下载
  - https://www.kernel.org/
  - 直接下载一个最新的稳定版 `***.tar.xz` 解压到 `~/kernelDevTree`
    - `tar xvJf ***.tar.xz`
    - `xz -d ***.tar.xz` and `tar -xvf ***.tar`

- 得到开发环境的**源码树**之后可以立刻执行一下 `make mrproper`
  - **Delete the current configuration, and all generated files**
  - 删除所有的编译生成文件、内核配置文件(**.config文件**)以及各种备份文件，几乎只在**第一次**执行内核编译前才用这条命令

配置内核

- 编译的过程需要源码目录下有一个 `.config` 文件用于指导如何编译内核。生成的方式比较多，挑几个常用的说明一下
  1. `make menuconfig`
     - 调用内核配置的图形界面， `make menuconfig` requires the **ncurses libraries**
       - `sudo apt-get install libncurses5-dev`
     - `using defaults found in /boot/config-5.15.0-60-generic`
       - 所以make menuconfig是采用当前系统的config在配置，如果源码版本与当前系统版本差异较大可能需要付出更多的努力
         - **所以编译源码如果版本不一致，则不要一上来就这样操作**
  2. `make defconfig`
     - 在 `arch/xxx/configs/xxx_defconfig` 会有各个体系结构默认的一个配置
     - 例如，当前 `.config` 文件会由 `x86_64_defconfig` 文件生成
       - `arch/x86/configs/x86_64_defconfig`
  3. `cp /boot/config-$(uname -r) .config`
     - 得到当前系统所用的配置
  4. `make oldconfig`
     - 在内核配置完成之后，可以选择 `make oldconfig` 进行备份
     - 重新配置完成后一般会自行备份生成 `.config.old`
- `.config`
  - 包含 `CONFIG_XXX=y` 或 `CONFIG_XXX=n` 或 `CONFIG_XXX=m` 等内容
  - 关于能否手动修改 `.config` 的问题
    - 有简单的去做了一下实验，并不是说 `.config` 文件不能修改，但是如果修改的时候格式有问题，**make的时候会自动恢复**
      - 可以认为 `.config` 是需要满足**既定的格式**的
- kconfig文件中包含了所有配置项的说明
  - **经常看看**
- 如果这一步报错，一般就是缺少编译内核所需的最基本条件
  - 搜一下缺啥 `apt-get` 一下即可
- 部分功能只能配置成 **Y 或 N**，无法配置为模块，比如 DAX
  - 只有部分功能支持 **M**，当然这部分也支持 **Y**
  - 取决于kconfig中可配置选项的类型
    - bool
    - tristate（三态）

编译安装内核

```
# 编译完成之后的目录
double_d@dd:~/kernelDevTree/build/kernel$ ls
arch   crypto   include  kernel    mm                       modules.order   scripts   source      usr      vmlinux.o
block  drivers  init     lib       modules.builtin          Module.symvers  security  System.map  virt     vmlinux.symvers
certs  fs       ipc      Makefile  modules.builtin.modinfo  net             sound     tools       vmlinux

# 将module安装到系统中后
double_d@dd:/lib/modules/5.10.14$ ls
build   modules.alias      modules.builtin      modules.builtin.modinfo  modules.dep.bin  modules.order    modules.symbols      sc
kernel  modules.alias.bin  modules.builtin.bin  modules.dep              modules.devname  modules.softdep  modules.symbols.bin

# 内核 image 被安装到系统中后
double_d@dd:/boot$ ls -al
total 167264
drwxr-xr-x  3 root root     4096 2月   9 15:44 .
drwxr-xr-x 24 root root     4096 2月   9 01:03 ..
...
-rw-r--r--  1 root root   125924 2月   9 15:44 config-5.10.14
drwxr-xr-x  5 root root     4096 2月   9 15:45 grub
...
-rw-r--r--  1 root root 15292864 2月   9 15:44 initrd.img-5.10.14
...
-rw-r--r--  1 root root  5203720 2月   9 15:44 System.map-5.10.14
...
-rw-r--r--  1 root root  9261120 2月   9 15:44 vmlinuz-5.10.14
```

- **一个重要的编译目标**
  - bzImage

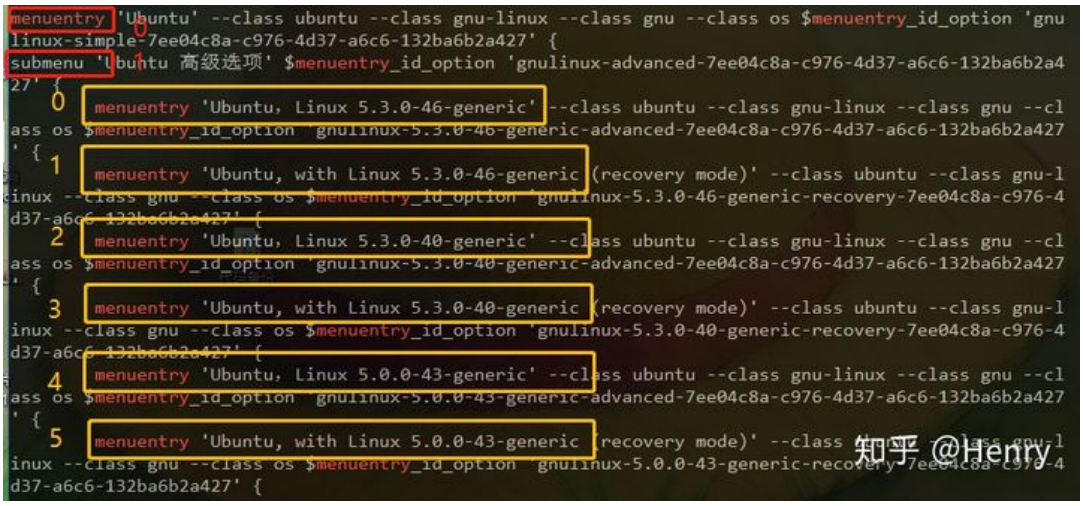
```
# Default kernel to build
all: bzImage

# KBUILD_IMAGE specify target image being built
KBUILD_IMAGE := $(boot)/bzImage

bzImage: vmlinux
ifeq ($(CONFIG_X86_DECODER_SELFTEST),y)
        $(Q)$(MAKE) $(build)=arch/x86/tools posttest
endif
        $(Q)$(MAKE) $(build)=$(boot) $(KBUILD_IMAGE)
        $(Q)mkdir -p $(objtree)/arch/$(UTS_MACHINE)/boot
        $(Q)ln -fsn ../../x86/boot/bzImage $(objtree)/arch/$(UTS_MACHINE)/boot/$@
```

- When compiling the kernel, all output files will per default be **stored together with the kernel source code**. Using the option `make O=output/dir` allows you to specify an alternate place for the output files (including `.config`)
  - kernel source code
    - `~/kernelDevTree/linux-xxx`
  - build directory
    - `~/kernelDevTree/build/kernel`
- To configure and build the kernel, use
  - `cd ~/kernelDevTree/linux-xxx`
  - `make O=~/kernelDevTree/build/kernel menuconfig`
    - `.config` 以及其它的一些相关文件位于 `~/kernelDevTree/build/kernel` 中
  - `make O=~/kernelDevTree/build/kernel -jX`
    - 如果编译有报错大都是缺少某些库或包，搜一下 `apt-get` 一下即可
    - 编译的结果均位于 `~/kernelDevTree/build/kernel` 目录下，还未出现在任何的系统位置，如上所示
      - vmlinux 原始的linux kernel 可执行文件，我自己编译的 `linux-5.10` vmlinux 约为 **52MB**
    - `~/kernelDevTree/build/kernel/arch/x86/boot/bzImage`

        - bzImage 文件是一个特殊的格式，包含了 `bootsect.o + setup.o + misc.o + piggy.o` 串接。 `piggy.o` 包含了一个 `gzip` 格式的 `vmlinux` 文件
            - 对应的 **bzImage** 约为 8.9MB
    - `sudo make O=~/kernelDevTree/build/kernel modules_install install`
        - 安装的本质是将 `.ko` 以及内核 `image` copy 到系统目录下，即执行脚本 `linux/arch/x86/boot/install.sh`
            - 安装内核模块到 `/lib/modules` 下，多了一个目录 `5.10.14`
            - 安装内核到 `/boot` 下，并自动执行 `sudo update-grub` 来更新引导项目，当然也可以自己手动再执行一次。**新的内核安装完成，重启即可选择新内核进入**
            - 调用脚本 `/usr/sbin/update-initramfs` and `/usr/sbin/mkinitramfs` 生成 initrd.img-5.10.14。会打印 `update-initramfs: Generating initrd.img-X.Y.Z` 到标准输出
                - initrd.img-5.10.14
                    - initialized ram disk
                - System.map-5.10.14 —— 它本身的定义与**导出与否**是**无关**的
                    - 符号表，**函数名或全局变量与其地址的对应关系**
                        - 包含了kernel中所有的全局变量与函数名等等
                    - **每次编译内核**都会产生一个**新**的 `System.map` 文件
                    - 如果**内核gg**了，根据地址能够很快的定位到**符号**
                        - **EXPORT_SYMBOL**标签内定义的函数或者符号**对全部内核代码公开**
                    - `/boot` 目录下只有这个，如果自己编译内核的话，除 `System.map` 外，还有一个文件是 `vmlinux.symvers`
                        - 详见 `kernel_tips`
                - vmlinuz-5.10.14
                    - 实际上就是 bzImage 的 copy
                        - `cat $2 > $4/vmlinuz` and cp `$3 $4/System.map`
                            - $2: kernel image file
                                - `arch/x86/boot/bzImage`
                            - $3: kernel map file
                                - `System.map`
                            - $4: default install path (blank if root directory)
                                - `/boot`
        - 好长时间没有回来看，忘了，这个源码是要放在 `/usr/src` 下的，我还下意识的以为还需要安装头文件
            - 源码是要丢在 `/usr/src` 下的
            - 这个没反应过来超级被动，**一直以为是没有安装成功**
- Please note: If the `O=output/dir` option is used, then it must be used for all invocations (调用) of make
    - **之后所有make的调用都要带这个参数，尽管麻烦一些，但是看得清爽**
- BIOS 加载 bootloader，即 grub，grub 再加载 kernel
    - linux启动的时候首先进入的就是引导界面，即 BIOS 将 grub 加载的内存中，供用户选择，具体加载哪个版本的 kernel



- **再看内核安装 grub 相关**

- 关于 grub 有 2 个文件
  - `/etc/default/grub`
    - change it and run `sudo update-grub` afterwards to update `/boot/grub/grub.cfg`
  - `/boot/grub/grub.cfg`
    - kernel 启动的 BootLoader
    - 选择操作系统启动之前的选择界面就是来自于 这里的
      - kernel 启动的 BootLoader
    - 修改 `/etc/default/grub` 需要update或重新生成
- 检查 当前系统有多少的 kernel 的版本，并且它们是 **按照启动顺序排列的**
  - `grep menuentry /boot/grub/grub.cfg`
- 在 grub 中可能存在 submenu 的情况，即一级菜单如下所示
  - ref
    - https://zhuanlan.zhihu.com/p/131542406
  - ubuntu
  - advanced options for ubuntu
    - 4.1.1
    - 4.1.2
    - 5.3.1
    - ...
  - 如果我想要选择 5.3.1 作为内核的启动项，则 `/etc/default/grub` 应该做如下修改
    - `GRUB_DEFAULT=1>2`
      - 就能够选择到对应的内核版本，包括 recovery 模式也会单独占下标
      - **甚至可以直接用字符串表示**
        - `GRUB_DEFAULT="Advanced options for Ubuntu>Ubuntu, with Linux 3.13.0-166-generic"`
          - 原来是这个意思
          - 这个options的首字母是小写的。因为误写成大写的导致**白干**了两个小时，记忆深刻
- **最新安装的内核是否是默认的启动项**，这个我不是太十分确定
  - `make install` 的详细的代码实现过程我没有太找到，只是下面这些是一定会存在的
    - 执行 `arch/x86/boot/install.sh`
    - 调用 `/sbin/installkernel`
      - 这个是一个sh脚本，之前有一个误区，以为 **绿色的可执行文件一定是elf的可执行文件**，但是实际上形式还是比较多的
        - **sh脚本，python脚本** 都可以是绿色的可执行文件的形式
        - 有的绿色的可执行文件是可以 **直接 cat 去 check 其内容的**
    - 然后调用 update-grub 来更新 `/boot/grub/grub.cfg`
      - add 是一个必然的过程，但是我不确定这个 add 是 append 的方式还是直接怼到第一个的
        - **第一顺位启动**

- 有一天炜杰遇到一个安装升级内核的问题，然后我自己试了一下，也GG了
  - 基本表现就是一顿操作都很正常，就是5.15的内核，利用make menuconfig把本机的config文件搞过来，但是最后启动的时候报错 out of memory
  - 然后炜杰搞定了这个问题，即在安装模块的时候用下面这一句 `make INSTALL_MOD_STRIP=1 modules_install`
  - 简单看起来，就是太多了，太大了

ramdisk到底是个啥东西呢

```
# 看看 initrd.img 到底是个啥
mkdir ~/initrd-learn
cd ~/initrd-learn
cp /boot/initrd.img-5.10.14 ./
un initrd.img-5.10.14 ./

# initrd.img 中 kernel 启动的部分内容，其中 init 指的就是 1 号进程的 image
double_d@dd:~/initrd-learn/main$ ls
bin  conf  etc  init  lib  lib64  run  sbin  scripts  usr  var

# 可以用 lsinitrd 来看，需要 sudo apt-get install dracut-core
double_d@dd:~/initrd-learn/main$ lsinitrd /boot/initrd.img-5.10.14
Image: /boot/initrd.img-5.10.14: 17M
================================================================================
Version:

Arguments:
dracut modules:
================================================================================
drwxr-xr-x   3 root     root            0 Nov 28  2018 .
drwxr-xr-x   3 root     root            0 Nov 28  2018 kernel
drwxr-xr-x   3 root     root            0 Nov 28  2018 kernel/x86
drwxr-xr-x   2 root     root            0 Nov 28  2018 kernel/x86/microcode
-rw-r--r--   1 root     root        30546 Nov 28  2018 kernel/x86/microcode/AuthenticAMD.bin
================================================================================

# 任意目录下查看 command 用 which
file `which update-initramfs`
/usr/sbin/update-initramfs: POSIX shell script, ASCII text executable
file `which mkinitramfs`
/usr/sbin/mkinitramfs: POSIX shell script, ASCII text executable
```

- **initrd.img-5.10.14 到底是个什么东西呢**
  - **initrd is init ram disk**
    - tmp rootfs
  - bios 启动，直接加载 grub，grub 加载 kernel，kernel 中是不包括某些 modules 的（取决于编译选项）。kernel 启动过程是必须要 `/sbin/init` 的，但是 init 的 image 是保存在某文件系统中的，如 ext4，如果 ext4 被编译成了moudle。那么 kernel 无法完成初始化工作，无法启动
    - 所以为了 kernel 能够在任何情况下都正常启动，引入了 initrd.img
    - initrd.img 将一段程序打包的 img 中，包括了 kernel 启动所必须的一些模块与内容
      - 1 号 进程 init 的磁盘 image
      - 一些命令 如 insmod
      - 一些脚本，用于调用 insmod 命令加载模块到 kernel 的脚本
      - 一些驱动
      - ...
    - kernel 启动后将 img 读入内存，**释放到内存中执行**
  - `/usr/sbin/update-initramfs` and `/usr/sbin/mkinitramfs` 这两个脚本用于生成 `initrd.img`
    - https://blog.csdn.net/lixiangminghate/article/details/50044033
    - make install 时会调用 update-initramfs，update-initramfs 继而调用 mkinitramfs 生成 initrd.img
      - 是一个往临时 initrd 目录 copy 文件的繁琐过程，mkinitramfs 则用脚本替代了手工操作
        - 在临时 initrd 目录下构建FHS规定的文件系统
        - 按 `/etc/initramfs-tools/module` 和 `/etc/modules` 文件的配置，往 `lib/modules/` 目录拷贝模块，同时生成模块依赖文件 `modules.dep`
          - 实际就是 `/lib/modules/${kernel_version}/` 目录下的一些启动会用到的模块
        - 拷贝 `/etc/initramfs-tools/scripts` 和 `/usr/share/initramfs-tools/scripts` 下的配置文件到 `conf/` 目录下
        - 模块的加载离不开 modprobe 工具集，因此需要拷贝 modprobe 工具集及其他工具到 initrd 目录结构下，同时解决这些工具的依赖关系

        - 所有步骤完成，调用 cpio 和 gzip 工具打包压缩临时 initrd 目录结构
- https://blog.csdn.net/deggfg/article/details/81537049
    - ramdisk的存在意义，解决先有鸡还是先有蛋的问题
        - 这是因为ramdisk临时文件系统和内核一样，也是由bootloader通过低级读写命令（如uboot用nand read，而不用通过文件系统层提供的高级读写接口）加载进内存，因此内核可以挂载内存里ramdisk文件系统
    - 这个博客写的还是有点东西的，**把这个内容全部copy下来吧先**
        - ramdisk一个作用就是用来解决boot过程中mount根文件系统的**先有鸡还是先有蛋**的问题的
        - 一般来说，根文件系统在形形色色的存储设备上，不同的设备又要不同的硬件厂商的驱动，比如intel的南桥自然需要intel的ide/sata驱动，VIA的南桥需要VIA的ide/sata驱动，根文件系统也有不同的文件系统的可能，比如ubuntu发行版可能一般用ext3，suse可能就不是了，不同的文件系统也需要不同的文件系统模块；假如把所有驱动/模块都编译进内核（注：即编一个通用的、万能的内核），那自然没问题，但是这样就违背了“内核”的精神或本质，所以一般来说驱动/模块都驻留在根文件系统本身上 `/lib/modules/xxx` ，那么**鸡蛋**问题就来了，现在要mount根文件系统却需要根文件系统上的模块文件，怎么办？于是，就想出ramdisk，内核总是能安装ramdisk的（注：这是因为ramdisk临时文件系统和内核一样，也是由bootloader通过低级读写命令（如uboot用nand read，而不用通过文件系统层提供的高级读写接口）加载进内存，因此内核可以挂载内存里ramdisk文件系统），然后把所有可能需要的驱动/模块都放在ramdisk上，首先，让内核将ramdisk当作根文件系统来安装，然后再用这个根文件系统上的驱动来安装真正的根文件系统，就将这个矛盾问题解决了
        - ramdisk还举出一个作用，现在的发行版在boot时一般都是图形界面的，那么，ramdisk就可以放frame buffer驱动和一些图片来做这种简单的动画。前一段时间刚好也在研究ramdisk, 下面是我找到的关于ramdisk的资料，希望对楼主有用在
        - Linux kernel 2.4 中，initrd 大致的处理流程如下：(方括号表示主要的执行单元)
            - `[boot loader]` Boot loader 依据预先设定的条件，将 kernel 与 initrd 这两个 image 载入到 RAM
            - `[boot loader -> kernel]` 完成必要的动作后，准备将执行权交给 Linux kernel
            - `[kernel]` 进行一系列初始化动作，initrd 所在的记忆体被 kernel 对应为 /dev/initrd 装置设备， 透过 kernel 内部的 decompressor (gzip 解压缩) 解开该内容并复制到 /dev/ram0 装置设备上
            - `[kernel]` Linux 以 R/W (可读写) 模式将 /dev/ram0 挂载为暂时性的 rootfs
            - `[kernel-space -> user-space]` kernel 准备执行 /dev/ram0 上的 /linuxrc 程式，并切换执行流程
            - `[user space]` /linuxrc 与相关的程式处理特定的操作，比方说准备挂载 rootfs 等
            - `[user-space -> kernel-space]` /linuxrc 执行即将完毕，执行权转交给 kernel
            - `[kernel]` Linux 挂载真正的 rootfs 并执行 /sbin/init 程式
            - `[user space]` 依据 Linux distribution 规范的流程，执行各式系统与应用程式
        - 值得一提的是，以上**两阶段开机**是 initrd 提出的弹性开机流程，在真实的应用中，也可能从未需要挂载真正的 rootfs，换言之，只是把系统当作都在 RAM disk 上运作，或者永远都在 initrd 所引导执行的 /linuxrc 程序中执行 (注意：kernel 永远保留 PID=1 作为 init process 识别，而 /linuxrc 执行的 PID 必非为 1)，在许多装置如智慧型手机，都是行之有年的，不过这不影响我们后续的探[/quote]里边所说的 initrd 大体上就是指 包含根文件系统的ramdisk

删除内核

- 删除发行版自带的， `dpkg --get-selections|grep linux` 能够显示的内核
    - `apt --purge remove kernel` and `apt autoremove`
    - `aptitude purge remove kernel`
- 源码安装的，依次删除以下内容
    - 源码
    - 模块
    - `/boot` 下与之相关的内容
    - update grub

内核模块

- 内核模块是 Linux 内核向外部提供的一个插口，其全称为 **动态可加载内核模块**
    - **Loadable Kernel Module，LKM**
- Linux 内核之所以提供模块机制，是因为它本身是一个单内核 (**monolithic kernel**)。单内核的最大优点是效率高，因为所有的内容都集成在一起，但其缺点是可扩展性和可维护性相对较差，模块机制就是为了弥补这一缺陷

    - 模块是具有独立功能的程序，它可以被单独编译，但不能独立运行。它在运行时被链接到内核作为内核的一部分在内核空间运行
- python 中，一个 `.py` 就是一个模块，文件夹包起来，加一个 `_init_.py` 就成了一个新的模块。对于 C 语言，一个 `.c` 也是一个模块，编译成 `.o` 之后可以与其它 `.o` link to a new module

```
double_d@dd:/lib/modules/5.10.14$ ls
build          modules.alias.bin    modules.builtin.modinfo  modules.devname  modules.symbols
kernel         modules.builtin      modules.dep              modules.order    modules.symbols.bin
modules.alias  modules.builtin.bin  modules.dep.bin          modules.softdep  source  extra
```

- 编译内核的时候会决定一个模块是否**built-in**
- 该目录下的文件格式如下
    - **ASCII text:就像是个list**
    - **data:二进制文件**
- `/lib/modules/kernel/` 下的的**目录解析**
    1. **build**: **build是一个符号链接**，**连接**到**内核头文件**
    2. 内置模块位于 `modules.builtin`
    3. 可加载模块位于 `modules.dep`
        - 根据 `/lib/modules/4.15.0+/kernel` 目录下的内容执行 depmod 生成的。所以可以认为可加载模块位于 `/lib/modules/4.15.0+/kernel`
            - `make modules_install` 就是将 `.ko` copy 到 `/lib/modules/4.15.0+/kernel` 目录下并执行 `sudo depmod`
            - modprobe会在 `modules.dep` 中寻找，所以如果是自己把 `.ko` copy 到 `/lib/modules/4.15.0+/kernel` 目录下的，需要执行 `sudo depmod`
                - 对应的模块才会出现在 `modules.dep` 中
                - but，**DONT RUN IT**
                    - depmod is smart enough to find the dependencies and write to a file - don't run it as it will overwrite the original file. First take backup of the file.
    4. 外部模块会被安装到 `/lib/modules/4.15.0+/extra` 目录下
        - **没有就创建**

- 如果仅仅是 make 完毕之后 的模块位于哪里呢
    - `find ./ -name "*.ko"`
    - 例如 ext4 被编译为内建模块，而 f2fs 被编译为模块
        - 则 fs/ext4 文件夹下只有 `.o` 文件，被用于最后 link 到 vmlinux 中
        - 而 fs/f2fs 文件夹下则生成 了 `.ko` 文件，在 make module_install 阶段会被 移动到 `/lib` 目录下
- lsmod 相当于 `cat /proc/modules` 。显示的是 **被加载的 可加载模块**
    - lsmod 的结果是 `modules.dep` 的子集，与 `modules.builtin` 没有半毛钱的关系
    - used by
        - 这个数字表示是否有其它模块依赖于该模块，如果有，则该模块无法直接卸载，**会报错 in use**，需要先卸载掉使用了该模块的其它模块，解除依赖之后才能卸载该模块
            - **当然也可以 -f 卸载**

```
# 符号加载的顺序
doubled@DESKTOP-18QFI0B:~/double_D/linux$ cat modules.order
fs/efivarfs/efivarfs.ko
drivers/thermal/intel/x86_pkg_temp_thermal.ko
samples/kprobes/kprobe_example.ko
samples/kprobes/kretprobe_example.ko
net/netfilter/nf_log_common.ko
net/netfilter/xt_mark.ko
net/netfilter/xt_nat.ko
net/netfilter/xt_LOG.ko
net/netfilter/xt_MASQUERADE.ko
net/netfilter/xt_addrtype.ko
net/ipv4/netfilter/nf_log_arp.ko
net/ipv4/netfilter/nf_log_ipv4.ko
net/ipv4/netfilter/iptable_nat.ko
net/ipv6/netfilter/nf_log_ipv6.ko

# 模块的依赖关系
doubled@DESKTOP-18QFI0B:~/double_D/linux$ modinfo net/ipv6/netfilter/nf_log_ipv6.ko
filename:       /home/doubled/double_D/linux/net/ipv6/netfilter/nf_log_ipv6.ko
alias:          nf-logger-10-0
license:        GPL
description:    Netfilter IPv6 packet logging
author:         Netfilter Core Team <coreteam@netfilter.org>
depends:        nf_log_common  # 该模块一定先于 nf_log_ipv6.ko 被加载
retpoline:      Y
intree:         Y
name:           nf_log_ipv6
vermagic:       5.11.0+ SMP mod_unload
```

- modinfo
  - 可以来看模块间的依赖关系
- Module.order
  - 模块加载的顺序
  - 模块之间会有符号的依赖关系

```
...
0x11ccaf06      stv0367cab_attach       drivers/media/dvb-frontends/stv0367     EXPORT_SYMBOL
0xf00fde8c      stv0367ddb_attach       drivers/media/dvb-frontends/stv0367     EXPORT_SYMBOL
0x8348e1e7      tcp_ca_openreq_child    vmlinux EXPORT_SYMBOL_GPL
0x6afc46a8      bpf_prog_destroy        vmlinux EXPORT_SYMBOL_GPL
0x6eaad29f      devm_remove_action      vmlinux EXPORT_SYMBOL_GPL
0x0cff4058      devm_clk_put    vmlinux EXPORT_SYMBOL
0xc40796a6      deactivate_super        vmlinux EXPORT_SYMBOL
0x569265e8      vmap    vmlinux EXPORT_SYMBOL
0x2caf7ab6      ww_mutex_lock_interruptible     vmlinux EXPORT_SYMBOL
...
```

- Module.symvers
  - It shows the symbols which are exported
  - **导出符号的宏** 就是 **将符号放到一个指定的section中**

built-in 模块

- 内置模块被静态地编译进了内核。无需动态地使用 `modprobe`、`insmod`、`rmmod`、`modinfo` 以及 `lsmod` 等命令加载、卸载、查询，总是在启动时加载进内核，**不会被这些命令管理**
  - `cat /lib/modules/$(uname -r)/modules.builtin | grep nd_pmem`
  - `grep "=y" /boot/config-$(uname -r) | less`

模块的加载与卸载

加载与卸载	高级	简单
加载	modprobe nova	insmod /lib/modules/4.13.0/kernel/fs/nova/nova.ko
卸载	modprobe -r nova	rmmod nova

- 两个简单命令并不进行依赖性检查以及进一步的错误检查。所以强烈推荐使用更先进的模块载入工具 **modprobe**，它提供了依赖性分析，错误智能检查以及错误报告等功能
- modprobe加载模块**不用指定模块文件的路径**，也不用带文件的后缀 `.o` 或 `.ko`
  - 加载一般都是依靠 modprobe
  - 如果模块被安装到了 `/lib/modules/$(uname -r)` 就会很方便
- insmod 需要的是模块的所在目录的绝对路径，并且一定要带有模块文件名后缀的 `modulefile.o` 或 `modulesfile.ko`
  - 如果模块未安装，只能通过 insmod 加载

内核模块开发

内嵌到内核源码树中编译（字符设备为例）

> **需要当前机器使用目标内核，否则 insmod 编译好的模块的时候，会去当前内核版本目录下找，会找不到。**比如开发机内核版本是4.15，编译安装了5.10的内核，但是依然在4.15下做开发就不行。**此外，新 kernel 的 config 还是 copy 一下当前使用的**

1. create folder in `/drivers/char/` named dd_char
2. create Kconfig and Makefile in dd_char with dd_char.c, all files, including relative files, are shown follows

```
# drivers/char/Kconfig
source "drivers/char/dd_char/Kconfig"

# /drivers/char/dd_char/Kconfig
menuconfig DD_CHAR
    tristate "DD_CHAR Support"
    default m
    help
        double_D kernel module

config DD_CHAR_SUB
    bool "DD_CHAR_SUB Support"
    default y
    depends on DD_CHAR
    help
        sub config

# CONFIG_A配置与否，取决于CONFIG_B是否配置。一旦CONFIG_A配置了，CONFIG_C也自动配置了
config A
    depends on B
    select C
```

- https://blog.csdn.net/supjia/article/details/4273587?utm_medium=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.control&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.control
  - kconfig相关的一个博客

```
# /char/drivers/Makefile
...
obj-$(CONFIG_DD_CHAR) += dd_char/

# /drivers/char/dd_char/Makefile
obj-$(CONFIG_DD_CHAR) += lib_dd_char.o
lib_dd_char-y := dd_char.o
```

- `make` 之后会生成 `lib_dd_char.ko`
- `sudo make modules_install` 后， `lib_dd_char.ko` 被安装到 `/lib/modules/$(uname -r)/kernel` 同时被更新到 `/lib/modules/$(uname -r)/modules.dep` 中

独立编译（字符设备为例）

- 学习资料
  - git@github.com:martinezjavier/ldd3.git
    - ldd3: linux device driver 3
  - https://paper.seebug.org/779/
  - http://liuao.tech/post/20161210/
- 基本步骤
  1. 在内核源码树外部创建一个合适的目录
     - `/home/double_D/my-kernel-module`
  2. copy **Makefile** and **source code** shown as follows and `make`
  3. you can choose install the `.ko` to system in `/lib/modules/$(uname -r)/extra` or just let it here with `insmod`
- source code

```
#include<linux/module.h>  // 头文件 init.h 包含了宏 _init 和 _exit
#include<linux/kernel.h>
#include<linux/init.h>

static int __init entry_init(void){
        printk("kernel module entry!\n");
        return 0;
}

static void __exit entry_exit(void){
        printk("say byby to kernel!\n");
}

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("double_D");
module_init(entry_init); // module_init 是模块的入口点，向内核注册模块所提供的功能，内置模块在引导时调用该位置，可加载模块在模块被插入时调
module_exit(entry_exit); // module_exit 是离开点，向内核注销模块所提供的功能，内置模块在该位置do nothing，可加载模块在该位置完成注销
```

- Makefile
  - https://blog.csdn.net/cjluxuwei/article/details/37878021
    - KERNELRELEASE的执行

内核中外部模块驱动的Makefile规范

```
# ifneq 判断参数是否不相等，不相等为 true，相等为 false
# /lib/modules/$(uname -r)/build
# build -> /usr/src/linux-headers-***-generic
# 这个 makefile 会执行两次，第一次执行的时候，KERNELRELEASE 是空的，所以不会执行 obj-m += entry.o
ifneq  ($(KERNELRELEASE),)
obj-m += kvisual.o  # 内核可视化模块，最终 entry.o 与 iovisual.o  会被 link 成为一个 kvisual.o
# -objs 表明 entry.o 与 iovisual.o 无论如何一定会被编译出来
kvisual-objs := \
        entry.o \
        iovisual.o
else
PWD = $(shell pwd)
KERNEL_DIR = /lib/modules/$(shell uname -r)/build
endif

# ifneq  ($(KERNELRELEASE),)
# obj-m += entry.o
# else
# KERNEL_DIR = /lib/modules/$(shell uname -r)/build
# PWD = $(shell pwd)
# endif

# 执行 KERNEL_DIR 下的 makefile
# modules 包括 clean 都是执行的 KERNEL_DIR 下 makefile 的目标，即 kernel 的 top makefile

# -C $(KDIR) 指明跳转到KDIR目录下读取那里的Makefile
# 当用户需要以某个内核为基础编译一个外部模块的话，即本例中的entry
# 需要在make modules 命令中加入 M=dir，程序会自动到你所指定的dir目录中查找模块源码，将其编译，生成 KO 文件
# M=$(PWD) 表明然后返回到当前目录继续读入并执行当前的 Makefile

# make M=dir clean
# make M=dir modules
# make M=dir same as make M=dir modules
# make M=dir modules_install

modules:
        $(MAKE) -C $(KERNEL_DIR) M=$(PWD) modules

modules_install:
        $(MAKE) -C $(KERNEL_DIR) M=$(PWD) modules_install

.PHONY:test
test:
        @echo $(PWD)

.PHONY:clean
clean:
        make -C $(KERNEL_DIR) M=$(PWD) clean
```

- 在 mac 下编写的，vscode 编程的时候，linux 的头文件会有红色波浪线，不指望在 mac 上跑，所以 down 了一个 linux 最新的头文件，并且工程的 vscode 中说明了 header 的搜索路径，在 mac 上就能编写了也就。没有找到 down src header 的地方，直接把服务器上的 copy下来了
  - 有点蠢，而且 `.h` 会有同名的情况，其实不准确
  - vscode include 的递归查找 `/**`，但是跑到其他地方的 `include/module.h` 下面去了就很尴尬了，红色波浪线我忍了
- `/usr/src` 下，有些会带有 **hwe**, hwe 是硬件支持 hardware element，硬件现在的发布速度快赶上软件了，所以一个硬件出来之后需要立刻能够跑在 linux上，所以就有了 hwe。emmmm，这个词和华为没啥关系
- 如果编写外部模块，linux-header 下面的这个 makefile 就是 kernel 的 top makefile
- 成功加载 `.ko` 后
  - `/var/log/syslog` 是能够查看 printk 的内容的
  - 在安装外部模块 `make modules_install` 的过程中，有一个类似 `skip system.dep` 的报错
    - `sudo depmod` 就能够解决

    - modprobe 与 insmod 的比较
        - modprobe 会去 `/lib/modules/$(uname -r)/modules.dep` 下面去找对应的 `.ko`，所以一定要 `make modules_install`
        - 但是用 insmod，编译完，直接在 `.ko` 的目录下 `insmod entry.ko` 就完事了
- KERNELRELEASE 是在内核源码的顶层 Makefile 中定义的一个变量，在第一次读取执行此 Makefile 时，KERNELRELEASE 没有被定义，所以 make 将读取执行 else 之后的内容
    - 如果 make 的目标是 clean，直接执行 clean 操作，然后结束
    - 当 make 的目标为 all 时
        - `-C $(KERNEL_DIR)` 指明跳转到内核源码目录下读取那里的 Makefile
        - `M=$(PWD)` 表明 **返回到当前目录继续读入、执行当前的 Makefile**
    - 当从内核源码目录返回时，KERNELRELEASE 已被定义
        - kbuild 也被启动去解析 kbuild 语法的语句
        - make 将继续读取 else 之前的内容
            - else 之前的内容为 kbuild 语法的语句，**指明模块源码中各文件的依赖关系，以及要生成的目标模块名**
                - `obj-m := param.o` 表示编译连接后将生成 `param.o` 模块
                - `param-objs := file1.o file2.o` 表示 `param.o` 由 `file1.o` 与 `file2.o` **连接**生成

构建外部模块的相关模块

- `linux/Documentation/kbuild/makefiles.txt`
- `linux/Documentation/kbuild/modules.txt`
    - To include a header file located under `include/linux/`, simply use
        - `#include <linux/module.h>`
    - kbuild will add options to **gcc** so the relevant directories are searched
    - External modules tend to place header files in a separate `include/` directory where `their source is located`
        - `drivers/nvme/host/nvme.h`
    - Although this is `not the usual kernel style`. To inform kbuild of the directory\
        - use either `ccflags-y` or `CFLAGS_<filename>.o`
- 参数 ccflags-y 即可解决问题
    - `ccflags-y := -Iinclude`
        - Note that in the assignment **there is no space** between -I and the path
            - This is a limitation of kbuild
                - there must be no space present
        - Note: Flags with the same behaviour were previously named
            - `EXTRA_CFLAGS`
            - They are still supported but their usage is deprecated

内核模块 include header path 所引发的一个思考

- 在内核函数 **nvme_complete_rq** 处注册 hook 之后，需要调用 3 个宏函数在完成对应的功能，最开始错误的认为以下 3 个宏函数位于 `drivers/nvme/host/nvme.h`
    - **rq_data_dir**
    - **blk_rq_pos**
    - **blk_rq_bytes**
- 所以在linux发行版中尝试迁移时，在 Makefile 中利用 `ccflags-y := -I/lib/modules/$(shell uname -r)/build` 来指定 include 路径，但是始终提示没有该路径
    - 一开始认为是 **ccflags-y** 的用法有问题，但是后面才发现这个路径
        - **确实没有**
        - **确实没有**
        - **确实没有**
    - 这个内容本身是 nvme 驱动中的内容，所以在发行版的头文件中确实是不会存在的
- 最后想明白了，以上 3 个宏的定义实际上依然是位于 `linux/include/linux/nvme.h`，即如下 include 即可顺利编译

    - `#include <linux/nvme.h>`
    - `#include <linux/blkdev.h>`
- linux不可能这么蠢，如上所说 header 是十分规范的
    - 发行版中所暴露的仅仅只有路径 `linux/include/linux/...` 下的内容

后续一系列小问题

ubuntu发行版与linux内核版本的对应关系

- 源于自己想要找一个目标内核版本的ubuntu发行版，但是发现搜索的效果不尽人意，查询这里即可
    - https://ubuntu.com/about/release-cycle#ubuntu-kernel-release-cycle
        - The Ubuntu lifecycle and release cadence

examples

NOVA

- 源码树中开发

```
double_d@dd:~$ dmesg|grep BIOS-e820
[    0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009d7ff] usable
[    0.000000] BIOS-e820: [mem 0x000000000009d800-0x000000000009ffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000000e0000-0x00000000000fffff] reserved
[    0.000000] BIOS-e820: [mem 0x0000000000100000-0x000000001fffffff] usable
[    0.000000] BIOS-e820: [mem 0x0000000020000000-0x00000000201fffff] reserved
[    0.000000] BIOS-e820: [mem 0x0000000020200000-0x000000003fffffff] usable
[    0.000000] BIOS-e820: [mem 0x0000000040000000-0x00000000401fffff] reserved
[    0.000000] BIOS-e820: [mem 0x0000000040200000-0x00000000b2a81fff] usable
[    0.000000] BIOS-e820: [mem 0x00000000b2a82000-0x00000000b2c83fff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000b2c84000-0x00000000c2c9efff] usable
[    0.000000] BIOS-e820: [mem 0x00000000c2c9f000-0x00000000cae9efff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000cae9f000-0x00000000caf9efff] ACPI NVS
[    0.000000] BIOS-e820: [mem 0x00000000caf9f000-0x00000000caffefff] ACPI data
[    0.000000] BIOS-e820: [mem 0x00000000cafff000-0x00000000cfffffff] usable
[    0.000000] BIOS-e820: [mem 0x00000000cb000000-0x00000000cf9fffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000f8000000-0x00000000fbffffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000fe800000-0x00000000fe80ffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000fec00000-0x00000000fec00fff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000fed08000-0x00000000fed08fff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000fed10000-0x00000000fed19fff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000fed1c000-0x00000000fed1ffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000fee00000-0x00000000fee00fff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000ffc20000-0x00000000ffffffff] reserved
[    0.000000] BIOS-e820: [mem 0x0000000100000000-0x000000022e5fffff] usable
[    0.000000] BIOS-e820: [mem 0x000000022e600000-0x000000022effffff] reserved
```

1. NOVA 位于内核源码树中，随内核一起编译。在安装该内核之前，注意对该内核进行的配置选项，关键选项必须选择
    - NVDIMM(Non-Volatile Memory Device
    - DAX(Direct Access)
    - 非标准的 NVDIMM 设备并启用 ADR 内存保护
2. 在确定分配多少内存空间用于模拟 PM 之前，先查看哪些部分的内存是可以使用的
    - `dmesg | grep BIOS-e820`
        - 这一部分内容后面标出usable的内存区域是可以使用的
3. 在 `/etc/default/grub` 中添加的内核启动项如下
    - `GRUB_CMDLINE_LINUX="memmap=4G!4G"`
4. 然后更新或重新配置生成 `/boot/grub/grub.cfg` 文件
    - update `/boot/grub/grub.cfg` 文件

        - 更新文件
        - `sudo update-grub`
    - 重新配置生成 `/boot/grub/grub.cfg` 文件
        - 重新生成配置文件
        - `sudo grub-mkconfig -o /boot/grub/grub.cfg`
5. 重启计算机之后，执行 `dmesg | grep user` 可以显示出有一部分内存区域被标记为 `persistent (type 12` 且会新出现设备文件 `/dev/pmem0`
6. 将模块载入内核
    - `modprobe nova`
        - 不需要指定模块nova的位置
    - `insmod /lib/nodules/4.13.0/kernel/fs/nova/nova.ko`
        - 需要指定出模块nova所在的具体位置
7. 在设备上做文件系统
    - `mkfs.nova /dev/pmem0`
8. 在 `/mnt` 下创建一个目录作为挂载点
    - `mdkir /mnt/randisk`
9. 挂载nova
    - `mount -t NOVA -o init /dev/pmem0 /mnt/ramdisk`

编译 libnvdimm 以支持的 ndctl's unit test

- git@github.com:pmem/ndctl.git

```
make -C tools/testing/nvdimm
```

- **编译测试模块用 -C**
    - 即进入到 `-C` 所指定的目录下执行 `Makefile`
        - 如果模块不在内核源码目录下，则需要 `-M` 指定，如编译外部模块
        - **测试代码模块是在源码树下的，所以无需-M**

sample下内容的编译与学习

- 目录 `linux/samples/...` 下的很多内容都是**类似驱动**一样的内核模块，编译它们就是在内核源码的一级目录下执行一下内容
    - `make clean M=samples/kprobes/`
    - `make M=samples/kprobes/`
- 当然这里需要在 `.config` 中开启对应的配置
- 具体需要开启哪些配置，直接查阅对应sample模块的makefile
    - `CONFIG_SAMPLES=y`
        - 所有sample的前提
    - `obj-$(CONFIG_SAMPLE_KPROBES) += kprobe_example.o`
    - `obj-$(CONFIG_SAMPLE_KRETPROBES) += kretprobe_example.o`
- 无论是直接编译完整内核还是单独执行 `make M=samples/kprobes/` ，都可以看到下面的结果

```
doubled@DESKTOP-18QFI0B:~/double_D/linux$ make M=samples/kprobes/
  CC [M]  samples/kprobes//kprobe_example.o
  CC [M]  samples/kprobes//kretprobe_example.o
  MODPOST samples/kprobes//Module.symvers
  CC [M]  samples/kprobes//kprobe_example.mod.o
  LD [M]  samples/kprobes//kprobe_example.ko
  CC [M]  samples/kprobes//kretprobe_example.mod.o
  LD [M]  samples/kprobes//kretprobe_example.ko
```

- **直接编译到kernel中这个就别想了，还是按照模块的方式来编译**，cp到busybox的ramdisk.img中，就可以很方便的调试了，主要是遇到了下面的这个报错
    - `This feature depends on another which has been configured as a module. As a result, this feature will be built as a module`