下面我按“原理→代码→Makefile→文件传输（xmodem）→面试介绍”五个部分，给你一个从底层到高层、面试可讲的全景式剖析。

**一、Linux 内核模块原理概览**

1. **模块（Module）是什么？**
   * Linux 内核通过把一些驱动或功能编译成“可加载模块”（.ko），在运行时按需插入（insmod）或移除（rmmod），而不必重启或重新编译整个内核。
   * 模块之间可以“导出”符号（函数或变量），供其他模块引用。
2. **加载与卸载流程**
   * insmod foo.ko
     1. 将镶嵌的机器码加载到内存。
     2. 调用 foo\_init()（由 module\_init() 宏注册）。
     3. 为导出的符号在全局符号表（ksyms）注册地址。
   * rmmod foo
     1. 如果没有其他模块依赖它，则卸载；否则失败。
     2. 调用 foo\_exit()（由 module\_exit() 宏注册）。
     3. 从符号表注销。
3. **符号导出（EXPORT\_SYMBOL）**
   * A 模块要调用 B 模块的函数，就得在 B 中用 EXPORT\_SYMBOL(getsum) 把 getsum 放到全局表里；A 中只要 extern int getsum(int,int)，链接器就能在运行时解析到。

**二、代码逐行拆解**

**1. sum.c（“库”模块）**

#include <linux/init.h>

#include <linux/module.h>

// 真正的功能函数

int getsum(int a, int b) {

return a + b;

}

// 导出符号，供别的模块（demo4）调用

EXPORT\_SYMBOL(getsum);

// 我这里其实不需要 init/exit，

// 如果只是做“库”，就可以去掉下面两段

static int \_\_init sum\_init(void){

return 0;

}

static void \_\_exit sum\_exit(void){

printk(KERN\_EMERG "sum exit\n");

}

module\_init(sum\_init);

module\_exit(sum\_exit);

//可以不写module\_init/module\_exit

// 如果你只是为了让别的模块调用 getsum()，那么 **不需要 module\_init(sum\_init) 和**

//**module\_exit(sum\_exit)**；

//但**如果你希望加载/卸载时打印提示、初始化资源或校验逻辑，就可以保留它们**。

MODULE\_AUTHOR("LIXIAOFAN");

MODULE\_DESCRIPTION("this is a sum driver");

MODULE\_LICENSE("GPL");

* \_\_init、\_\_exit 修饰符：标明函数只在加载/卸载时用，其他时候可被丢弃以节省内存。
* MODULE\_\* 宏：元信息，帮助 modinfo 展示作者、许可等。

**2. demo4.c（主业务模块）**

#include <linux/init.h>

#include <linux/module.h>

// 引用外部符号

extern int getsum(int a, int b);

static int \_\_init demo4\_init(void){

printk(KERN\_INFO "myled\_init\n");

printk(KERN\_INFO "%d\n", getsum(100,200));

return 0;

}

static void \_\_exit demo4\_exit(void){

printk(KERN\_INFO "myled\_exit\n");

}

module\_init(demo4\_init);

module\_exit(demo4\_exit);

MODULE\_AUTHOR("LIXIAOFAN");

MODULE\_DESCRIPTION("this is a led driver");

MODULE\_LICENSE("GPL");

* printk(KERN\_INFO…)：内核里的 printf，不同优先级（INFO、EMERG、ERR……）。
* 这里它调用了 getsum(100, 200)，如果你先插 sum.ko、再插 demo4.ko 就能打印 300。

**三、Makefile 深度解析**

# 1. 要生成哪些模块

# obj-m += sum.o demo4.o # 如果想做两个独立模块，用这种方式

obj-m+= demo4.o # 这里声明：要编译 demo4.ko

demo4-objs := sum.o demo4.o

* obj-m：告诉内核树：“把后面的 .o 链接成 .ko”。
* demo4-objs：指定 demo4.ko 由哪些目标文件打包，等价于 ld -r -o demo4.o sum.o demo4.o 然后生成 demo4.ko。
* **千万注意**：不要把 demo4.o 递归地加到自己两次；上面 := 是一次赋值，替换早于展开；如果用 += 则会把原有值追加。

# 2. 内核源码路径 + 交叉编译器

KERNEL\_DIR := /home/gec/6818GEC/kernel

CROSS\_COMPILE := /home/gec/6818GEC/prebuilts/gcc/linux-x86/arm/arm-eabi-4.8/bin/arm-eabi-

PWD:= $(shell pwd)

# 3. 默认目标：进入内核源码树，调用其顶层 Makefile

default:

$(MAKE) ARCH=arm CROSS\_COMPILE=$(CROSS\_COMPILE) \

-C $(KERNEL\_DIR) M=$(PWD) modules

clean:

rm -rf \*.o \*.order .\*.cmd \*.mod.c \*.symvers .tmp\_versions

* ARCH=arm：告诉内核你在编译 ARM 平台模块。
* -C $(KERNEL\_DIR)：切换到内核源码目录执行 make；
* M=$(PWD)：告诉它“外部模块”目录，也就是你当前文件夹。
* modules：构建所有在 obj-m 里注册过的模块。

**= vs := vs +=**

* =：递归展开（延迟展开），引用变量时才替换。
* :=：立即展开，定义时就把右侧求值。推荐用在编译路径等“常量”。
* +=：向已有变量追加一段文本，能给 obj-m 追加更多模块。

**四、文件传输：用 xmodem（rx）上板**

你在串口终端里看到：

# rx sum.ko

# rx demo4.ko

* rx 文件名：用 **xmodem** 协议从主机（你的 PC）往嵌入式板卡里“拉”一个文件。
* 在 PC 端用 sx sum.ko 来发送，终端会打印进度、速度和“Errors”。
* 物理串口不稳定时偶尔会校验错误，但 xmodem 简单可靠，常见于 U-Boot、裸机调试。

**五、面试汇报要点**

当被问到“你这个模块是怎么写的/编译的”时，可按下面脉络去组织语言：

1. **模块架构**
   * “我把公共算法 getsum 写成了一个单独的模块 sum.ko 并用 EXPORT\_SYMBOL 导出，再在主模块 demo4.ko 中用 extern 引入，职责分离、利于复用。”
2. **加载顺序与符号解析**
   * “先 insmod sum.ko 注册符号表，再 insmod demo4.ko 链接，这样就能在 demo4\_init 中直接调用 getsum 并打印结果。”
3. **Makefile 机制**
   * 解释 obj-m、demo4-objs：如何把多个 .o 合并打包；
   * 讲清 ARCH=arm、CROSS\_COMPILE、-C $(KERNEL\_DIR) M=$(PWD) 的作用。
   * 提到 =/:=/+= 的区别，显示你对 Make 的深刻理解。
4. **文件传输与调试**
   * “开发板没有网络，只能用串口 xmodem 协议（rx/sx）传模块文件，然后在 /lib/modules/... 目录下 insmod，用 dmesg 检查打印。”
5. **扩展思考**
   * “如果模块更多、依赖复杂，可以写一个自动化脚本或用 kmod-tools 管理加载顺序，还能结合 depmod/modprobe 做依赖解析。”
   * “生产环境下可把这些模块打包进 initramfs，开机自动加载。”

以上就是从基础原理、源码设计、Makefile 细节、开发板传输到面试汇报思路的全套脉络。希望你能串成一条“简洁明了”、“重点突出”的技术故事，在面试官面前清晰呈现！祝你面试顺利！