ChatGPT

Linux 内核模块开发入门手册

1. 为什么使用内核模块

在 Linux 操作系统中，从内核角度可分为**宏内核**和**微内核**架构。**宏内核**将大部分功能（如进程管理、线程管理、内存管理、文件系统、设备驱动、网络协议等）都集成在内核态运行，这样性能高，但稳定性较差：驱动中的一个 bug 可能导致整个系统崩溃 [1]。做驱动开发时，如果驱动程序出现错误，经常需要通过按电源键强制重启系统来恢复。而**微内核**只保留最基本的功能（如调度、内存管理），驱动程序和文件系统运行在用户态的守护进程中，这种架构稳定性更高：即使驱动崩溃，也只会影响相应进程，不会导致内核挂掉，开发调试时可以直接 kill 进程然后重启 [1]。缺点是效率较低，因为内核态与用户态之间要频繁进行消息传递。

Linux 虽然是宏内核，但汲取了微内核的优点——采用**模块化设计**。Linux 支持可抢占、支持内核线程，并允许动态装载和卸载内核模块 [2]。这样做既保持了宏内核的高性能（内核内部直接调用函数，无需复杂通信），又能在需要时按需加载功能模块，提高灵活性。此外，使用内核模块还能**缩小内核镜像大小**，节省内存：不需要把所有驱动都编译到内核中，只在运行时以模块形式加载所需驱动 [3]。

要点回顾：
- **宏内核**：所有功能编译进内核（高效，但稳定性低） [1]；**微内核**：只保留基本功能，驱动在用户态（稳定，高容错） [1]。
- Linux 是宏内核，但支持动态模块加载（实用主义设计），兼顾效率与灵活性 [2]。
- 内核模块可按需加载，减少内存占用，并且便于开发调试。

2. 内核模块是什么？

内核模块（kernel module）是运行在内核空间的可加载驱动或功能代码。通俗地说，许多硬件驱动和功能都是以模块形式存在，它们相互独立互不干扰。例如，一个 LED 驱动和一个蜂鸣器驱动，可以分别编译成独立的模块 [4]。用户态应用程序通过操作设备文件（如 `/dev/led_drv`、`/dev/beep_drv`）来调用对应模块提供的功能，而这两个模块本身并不直接交互，如下例所示：

```
#include <stdio.h>
#include <fcntl.h>

int main(void) {
    int fd_led = open("/dev/led_drv", O_WRONLY);
    int fd_beep = open("/dev/beep_drv", O_WRONLY);
    // 通过文件操作控制 LED 和蜂鸣器
    // ……
    close(fd_led);
    close(fd_beep);
    return 0;
}
```

内核模块编译成功后会生成以 `.ko` 结尾的文件（kernel object） [5]。常用命令：
- `insmod led_drv.ko`：将模块加载到内核中；

- `rmmod led_drv`：从内核中卸载模块；
- `lsmod`：查看当前已加载的模块列表 [5]。

要点回顾：
- **模块文件**：内核模块编译后为 `*.ko` 文件 [5]。
- **加载/卸载**：使用 `insmod` 加载模块，`rmmod` 卸载，`lsmod` 查看状态 [5]。
- **设备接口**：应用程序通过打开 `/dev` 下的设备文件与模块交互；模块彼此独立 [4]。

3. 设计一个简单的内核模块

一个基本的内核模块至少需要包含模块入口函数（初始化函数）和模块出口函数（清理函数），并使用宏 `module_init` 和 `module_exit` 将它们注册。下面是一个 LED 驱动模块的简化示例（以 ARM 平台的 `imx6ull_led` 为例）：

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

// 模块入口函数：加载模块时调用
static int __init imx6ull_led_init(void)
{
  printk(KERN_INFO "imx6ull led init\n"); // 在内核日志中打印信息
  return 0; // 返回 0 表示初始化成功
}

// 模块出口函数：卸载模块时调用
static void __exit imx6ull_led_exit(void)
{
  printk(KERN_INFO "imx6ull led exit\n"); // 在内核日志中打印退出信息
}

// 指定加载时调用的入口函数
module_init(imx6ull_led_init);
// 指定卸载时调用的出口函数
module_exit(imx6ull_led_exit);

// 模块元数据
MODULE_AUTHOR("作者邮箱@example.com");       // 模块作者
MODULE_DESCRIPTION("IMX6ULL LED 驱动模块");    // 模块描述
MODULE_LICENSE("GPL");                  // 指明许可证为 GPL
```

- **`__init` 和 `__exit`**：`__init` 标记的函数只在模块加载时调用一次，执行完成后内核会回收其占用的内存 [6]；`__exit` 标记的函数只在模块卸载时调用，调用后释放内存 [6]。如果模块被编译进内核而无法卸载，则 `__exit` 函数会被忽略。
- **`module_init/exit`**：`module_init(func)` 宏将 `func` 注册为模块的入口函数（相当于 insmod 时执行）；`module_exit(func)` 注册模块的出口函数（rmmod 时执行） [7]。
- **`MODULE_*` 宏**：提供模块的元信息，如许可证、作者、描述等。最重要的是 `MODULE_LICENSE("GPL")`，表明模块遵循 GPL 协议，以便使用导出的 GPL 符号。

要点回顾：
- 模块必须有入口函数和出口函数，分别用 `module_init` 和 `module_exit` 宏注册 [7]。
- `__init` 修饰的初始化函数只执行一次，用完即回收内存；`__exit` 修饰的退出函数只在卸载时运行 [6]。
- 模块中只可使用内核提供的函数和宏（如 `printk`），不能使用标准 C 库函数（如 `printf`）。
- 使用 `MODULE_LICENSE("GPL")` 等宏声明模块的许可证和元信息。

4. 内核模块与应用程序的区别

- **运行空间不同**：内核模块运行在**内核空间**，拥有最高权限，直接访问硬件资源，但一旦出错可能导致整个系统崩溃；而应用程序运行在**用户空间**，权限低，无法直接访问硬件，出错只影响自身，不会影响内核 [8]。
- **程序结构不同**：内核模块有入口函数和出口函数，由 `module_init` / `module_exit` 指定；而普通应用程序只有 `main` 一个入口函数 [8]。
- **可用函数不同**：内核模块只能调用内核提供的函数（头文件位于内核源码的 include 目录），不能使用标准 C 库函数。例如，模块中只能用 `printk` 打印日志，而不能用 `printf` [8]。
- **耦合方式不同**：内核模块之间相互独立，通过导出符号或编译链接共享功能，遵循“高内聚、低耦合”原则，不同模块之间没有直接联系 [9]。
- **编译与版本**：编译内核模块时需要指定内核源码路径、交叉编译器前缀等；最重要的是，模块所用的内核源代码版本必须与目标系统的内核版本一致，否则模块无法加载 [10]。可以通过 `uname -r` 命令查看目标平台的内核版本。
- **内核特性限制**：内核栈一般只有 4KB 或 8KB，容量很小，分配大块内存要使用专门的内存分配函数（如 `kmalloc`、`vmalloc`） [11]。此外，`printk` 不支持浮点数格式（使用会编译成功但不输出正确结果） [12]。

要点回顾：
- 模块在内核空间执行，应用在用户空间；模块崩溃影响系统，应用崩溃只影响自己 [8]。
- 模块有入口/出口函数，应用只有 `main`。 [8]
- 模块只能使用内核函数（如 `printk`、`kmalloc`），不能用标准库 [8] [11]。
- 编译时要指定正确的内核源代码版本和交叉编译器。

5. Makefile 的编写

为了编译内核模块，需要编写特殊的 Makefile，让内核自带的构建系统来完成模块编译。一般需要定义模块名和内核源码路径。例如，一个简单的 Makefile 可能如下：

```
# 定义模块目标文件
obj-m += led_drv.o

# 内核源码所在目录（可用 uname -r 自动获取当前内核版本）
KERNELDIR := /lib/modules/$(shell uname -r)/build
# 当前模块所在目录
PWD := $(shell pwd)

all:
  $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

clean:
  $(MAKE) -C $(KERNELDIR) M=$(PWD) clean
```

说明：
- **obj-m**：指定要编译的模块源文件，不含扩展名。例如 `obj-m += led_drv.o` 表示要生成 `led_drv.ko` 模块。对多个模块，可使用多个 `obj-m += ...`。[13]
- **KERNELDIR/M=$(PWD)**：`make -C $(KERNELDIR) M=$(PWD) modules` 命令会调用内核源代码目录下的 Makefile，使用当前目录下的模块代码编译生成模块[14]。
- **clean**：同理 `make -C $(KERNELDIR) M=$(PWD) clean` 会清除编译产生的临时文件。
- **变量赋值**：常见操作符 `:=` 是立即展开赋值，`+=` 是追加内容，`?=` 是条件赋值（如果未定义时赋值）等[15]。

使用示例命令行：
```bash
make      # 执行模块编译
make clean  # 清理编译生成的文件
```

```
**要点回顾：**
- 在 Makefile 中通过 `obj-m += 模块名.o` 定义要编译的模块[13]。
- 使用 `make -C <内核源码路径> M=$(PWD) modules` 来编译外部模块[14]。
- 确保 `KERNELDIR` 指向目标系统对应的内核源码目录（版本要一致）。
- `make clean` 调用内核构建系统的 clean 规则，删除 `.o`、`.mod.c` 等临时文件。

## 6. printk 函数
在内核代码中不能使用用户空间的 `printf`，而应使用 `printk` 函数打印日志信息。`printk` 的使用格式类似于 `printf`，但可以指定日志级别。典型用法：
```c
printk(KERN_INFO "Value = %d\n", val);
printk(KERN_WARNING "Warning: %s is reserved\n", dev_name);
printk(KERN_ERR "Error: failed to register device\n");
```
```

其中 `KERN_INFO`、`KERN_WARNING`、`KERN_ERR` 等宏定义了不同的日志级别[16]。这些宏实际上是特定格式的字符串前缀，例如：
- `KERN_EMERG "<0>"` 致命级：紧急事件，系统不可用[17]；
- `KERN_ALERT "<1>"` 警戒级：必须立即处理的消息；
- `KERN_ERR "<3>"` 错误级：一般错误条件，如设备注册失败[18]；
- `KERN_WARNING "<4>"` 警告级：警告信息；
- `KERN_INFO "<6>"` 通知级：普通信息，如初始化完成日志[18]；
- `KERN_DEBUG "<7>"` 调试级：调试信息。

需要注意：如果在控制台上看不到 `printk` 的输出，可能是因为当前日志级别设置较高，可用 `dmesg` 命令查看内核日志缓冲区。`printk` 输出会记录在内核日志中，可通过 `dmesg | tail` 等命令查看最近的内核日志。

要点回顾：
- 内核中用 `printk` 打印日志，不能用 `printf`[19]。
- `printk` 需要带上日志级别宏（如 `KERN_INFO`、`KERN_ERR`），日志级别从 0（紧急）到 7（调试）[18]。
- 日志输出通常可通过 `dmesg` 命令查看。

7. 内核模块参数

内核模块可以像应用程序一样，支持在加载时传递参数。例如，如果编写串口驱动，需要在加载时指定波特率，就可以定义一个模块参数来接收该值。内核提供了 `module_param` 和 `module_param_array` 宏来声明参数

[20]。支持的数据类型包括：`bool`、`charp`（字符串指针）、整型（short、int、long、ushort、uint、ulong）等[21]。

使用示例：

```
#include <linux/module.h>
#include <linux/kernel.h>

static int baud = 9600;       // 默认波特率
static char *name = "vcom";    // 默认名字
static int ports[4] = {0,1,2,3}; // 数组参数
static int ports_cnt = 0;      // 数组长度

// 定义模块参数
module_param(baud, int, 0644);          // 定义 int 类型参数 baud，可读可写（rw- r-- r--）
module_param_array(ports, int, &ports_cnt, 0644); // 定义 int 数组参数 ports，ports_cnt 保存元素个数
module_param(name, charp, 0644);        // 定义字符串参数 name

static int __init mymod_init(void)
{
   printk(KERN_INFO "Init: baud=%d, name=%s\n", baud, name);
   printk(KERN_INFO "Ports:");
   for (int i = 0; i < ports_cnt; i++)
      printk(KERN_CONT " %d", ports[i]);
   printk(KERN_CONT "\n");
   return 0;
}

static void __exit mymod_exit(void)
{
   printk(KERN_INFO "Exit module\n");
}

module_init(mymod_init);
module_exit(mymod_exit);
```

说明：
- `module_param(name, type, perm)`：用于注册单个参数，其中`name`是变量名，`type`是类型，`perm`是在`/sys/module/`下对应参数文件的权限[20]。权限值类似 Unix 文件权限，如`0644`表示可读可写。
- `module_param_array(name, type, nump, perm)`：用于定义参数数组，`nump`是一个指针，保存数组实际传入的元素个数[20]。
- 加载时传参示例：`insmod mymod.ko baud=115200 name="ttyUSB0" ports=4,5,6`。内核会自动将这些值填充到对应变量。

要点回顾：
- 使用`module_param`宏在模块加载时接受参数[20]；支持整型、字符串、布尔等类型。
- 参数通过类似命令行方式传入：`insmod mod.ko param=value`。
- `perm`参数指定该模块参数在 sysfs 下的文件权限（读写权限）。[22]

8. 编译多个内核模块

如果一个项目有多个内核模块源文件，可以在同一个 Makefile 中同时编译。例如，假设有 `led_drv.c` 和 `sum.c` 两个模块，在 Makefile 中写：

```
obj-m += led_drv.o
obj-m += sum.o
```

然后使用同样的 `make -C <KERNELDIR> M=$(PWD) modules` 命令就会编译 `led_drv.ko` 和 `sum.ko` 两个模块[23]。注意这里每个模块名都用 `obj-m +=` 添加一行[23]。

要点回顾：
- 在 Makefile 中对每个模块名使用 `obj-m += 模块名.o` 来编译多个模块[23]。
- 执行一次 make，可以同时生成多个 `.ko` 文件。

9. 内核符号表（EXPORT_SYMBOL）

内核符号表记录了内核中所有导出的函数和全局变量。若要让一个模块提供的函数或变量被其他模块使用，需要使用 `EXPORT_SYMBOL` 或 `EXPORT_SYMBOL_GPL` 宏导出它们[24]。例如，在模块 A 中：

```
int myadd(int a, int b) { return a + b; }
unsigned int g_count = 0x100000;
EXPORT_SYMBOL_GPL(myadd);
EXPORT_SYMBOL_GPL(g_count);
```

在模块 B 中可以这样使用：

```
extern int myadd(int, int);
extern unsigned int g_count;
```

然后直接调用 `myadd()` 或访问 `g_count`。`EXPORT_SYMBOL_GPL` 导出的符号仅允许 GPL 许可证的模块使用，而 `EXPORT_SYMBOL` 不限制许可证[24]。内核在编译时会将导出符号的信息放入符号表中，运行时模块加载时即可解析使用。

要点回顾：
- 使用 `EXPORT_SYMBOL(符号名)` 将函数或全局变量导出，供其他模块使用；`EXPORT_SYMBOL_GPL` 只对 GPL 模块可见[24]。
- 模块 B 使用导出的符号时，需要声明 `extern`。
- 符号表（如 `/proc/kallsyms`）记录了所有导出的内核符号。

10. 多个源文件组合为单个模块

有时希望将多个源文件编译成一个模块，但又不想使用符号表导出。可以在 Makefile 中使用 `<module>_objs` 规则。例如，将 `led.c` 和 `sum.c` 链接为一个模块 `led_drv.ko`，Makefile 写法：

```
obj-m += led_drv.o
led_drv-objs := led.o sum.o

KERNELDIR := /path/to/kernel
CROSS_COMPILE := arm-linux-gnueabihf-
PWD := $(shell pwd)

all:
  $(MAKE) ARCH=arm CROSS_COMPILE=$(CROSS_COMPILE) -C $(KERNELDIR) M=$(PWD) modules
```

说明：`obj-m += led_drv.o` 指定模块名为 `led_drv.ko`，而 `led_drv-objs := led.o sum.o` 指定这个模块包含 `led.c` 和 `sum.c` 两个源文件 [25]。链接后，模块内部可以直接调用这两个文件中的符号，无需导出。最终 `make` 会生成 `led_drv.ko`。

要点回顾：
- 使用 `<modulename>-objs := file1.o file2.o ...` 让多个源文件链接成一个模块 [25]。
- 这样编译时将 `file1.c` 和 `file2.c` 编译并打包到同一个 `.ko`，内部可直接调用彼此的函数。
- Makefile 中的模块名（如 `led_drv`）与文件名前缀一致，用于生成模块。

总结： 本手册介绍了内核模块的基本概念、编写流程和常用技巧。从宏/微内核角度说明了使用模块的好处，提供了一个简单的模块模板和 Makefile 示例，并详细讲解了模块参数、日志输出、符号导出等关键概念。掌握这些内容，读者即可着手入门 Linux 内核模块开发，在理解原理的基础上编写自己的驱动模块。

参考资料： 上述内容参考了相关技术文档和资料 [1] [5] [8] [20] [24] [25]。

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] 内核模块-CSDN博客
https://blog.csdn.net/weixin_42832472/article/details/113528876