回顾：
笔试题：自己编写一个strcmp或者stcpy函数
内核源码/lib/string.c提供相关的参考代码

1.linux内核设备驱动开发相关基础内容
  1.1.用户空间和内核空间
  1.2.linux内核程序编程基本框架
  1.3.linux内核程序编译
  1.4.linux内核程序操作
  1.5.linux内核程序命令行传参
  1.6.linux内核程序符号导出
  1.7.linux内核程序打印函数printk
  1.8.linux内核GPIO操作库函数
  1.9.linux内核系统调用实现过程

2.linux内核字符设备驱动开发相关内容
  2.1.linux系统理念
  2.2.linux设备驱动分类
  2.3.设备文件
      字符设备文件
      块设备文件
      /dev/
      包含主，次设备号
      mknod
      利用系统调用函数进行访问
  2.4.设备号
      包括主，次设备号
      dev_t
      12
      20
      MKDEV
      MAJOR
      MINOR
      主设备号：应用根据主设备号找驱动
      次设备号：驱动根据次设备号区分硬件个体
      设备号是一种宝贵的资源
      申请：alloc_chrdev_region
      释放：unregister_chrdev_region

  2.5.linux内核描述字符设备驱动的数据结构
      struct cdev {
        dev_t dev; //保存申请的设备号
        int count; //保存硬件设备的个数
        const struct file_operations *ops;//保存字符设备驱动具有硬件操作接口
        ...
      };

      linux内核字符设备驱动的硬件操作接口的数据结构:
      struct file_operations {
        int (*open) (struct inode *, struct file *);
        int (*release) (struct inode *, struct file *);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ...
      };
  2.6.问：如何实现一个字符设备驱动程序呢？

答：编程步骤是定死的！
具体编程步骤：
1.定义初始化一个struct file_operations硬件操作接口对象
2.定义初始化一个struct cdev字符设备对象
  cdev_init用来初始化字符设备对象
3.调用cdev_add将字符设备对象注册到内核中
  至此内核就有一个新的字符设备驱动
4.调用cdev_del将字符设备对象从内核中卸载
  一旦卸载,内核就不存在这个字符设备驱动

2.7.切记应用程序的系统调用函数和底层驱动硬件操作接口之间的联系：
应用程序调用open->C库open->软中断->内核sys_open->驱动open接口
应用程序调用read->C库read->软中断->内核sys_read->驱动read接口
应用程序调用write->C库write->软中断->内核sys_write->驱动write接口
应用程序调用close->C库close->软中断->内核sys_close->驱动release

案例：应用程序打开LED设备时,开灯
      应用程序关闭LED设备时,关灯
分析：
0.将两个LED灯作为一个硬件设备
1.应用程序使用open/close
  驱动提供open/release接口
2.两个接口只需要利用GPIO操作库函数分别操作LED即可
3.前提是需要将file_operations和cdev进行定义初始化和注册

实施步骤：
虚拟机执行：
```
mkdir /opt/drivers/day03/1.0 -p
cd /opt/drivers/day03/1.0
vim led_drv.c
vim Makfile
make
cp led_drv.ko /opt/rootfs/home/drivers
vim led_test.c //测试的应用程序
arm-linux-gcc -o led_test led_test.c
cp led_test /opt/rootfs/home/drivers
```

ARM板执行：
```
insmod /home/drivers/led_drv.ko
cat /proc/devices //查看申请的主设备号
Character devices: //当前系统的字符设备
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  ...
  250 tarena
  ...
```
  第一列：申请的主设备号
  第二列：设备名称

```
mknod /dev/myled c 250 0
/home/drivers/led_test
```

3.编写一个字符设备驱动的步骤：
1.写头文件

2.写入口和出口函数
3.先声明和定义初始化硬件信息
4.然后定义初始化软件信息
  file_operations
  cdev
  dev
5.填充入口和出口函数
  先写注释流程
  再写代码
6.最后将给用户提供的接口函数完善

4.linux内核字符设备驱动硬件操作接口之write
  回忆：应用write系统调用函数的使用
  char msg[1024] = {....};
  write(fd, msg, 1024); //向设备写入数据

  对应的底层驱动的write接口：
  struct file_operations {
        ssize_t (*write) (struct file *file,
                          const char __user *buf,
                          size_t count,
                          loff_t *ppos);
  };
  调用关系：
  应用调write->C库write->软中断->内核sys_write->驱动write接口
  功能：此接口用于将数据写入硬件设备
  参数：
  file:文件指针
  buf：此指针变量用__user修饰,表明此指针变量保存的地址位于
       用户空间(0~3G),也就是保存用户缓冲区的首地址(例如msg)
       虽然底层驱动可以访问buf获取用户的数据,但是内核不允许驱动
       直接操作这个用户缓冲区(例如：int data = *(int *)buf),
       这么操作时相当危险的,如果要从buf缓冲区读取数据到驱动中
       必须要利用内核提供的内存拷贝函数copy_from_user将用户缓冲区的
       数据拷贝到内核缓冲区(3G~4G)中
  count:要写入的字节数
  ppos：记录了上一次的写位置,参考代码：
         loff_t pos = *ppos //获取上一次的写位置
         这一次写了512字节,此接口返回之前,要记得更新写位置：
         *ppos = pos + 512

         友情提示：如果写操作只进行一次,无需记录写位置!
         所以此参数用于多次写入操作

   内存拷贝函数copy_from_user
   unsigned long copy_from_user(void *to,
                                const void __user *from,
                                unsigned long n)
   函数功能：将用户空间缓冲区的数据拷贝到内核空间的缓冲区中
   参数：
   to:目标,内核缓冲区的首地址
   from:源,用户缓冲区的首地址
   n:要拷贝的字节数
   切记：此函数仅仅做了将用户内存的数据拷贝到内核的内存
         此时还没有进行硬件操作,硬件操作还需要进行额外的操作

案例：应用向设备写1,开灯
应用写设备写0,关灯
实施步骤：
mkdir /opt/drivers/day03/2.0
cd /opt/drivers/day03/2.0
vim led_drv.c
vim Makfile
make
cp led_drv.ko /opt/rootfs/home/drivers
vim led_test.c //测试的应用程序
arm-linux-gcc -o led_test led_test.c
cp led_test /opt/rootfs/home/drivers

ARM板执行：
insmod /home/drivers/led_drv.ko
cat /proc/devices //查看申请的主设备号
Character devices: //当前系统的字符设备
1 mem
2 pty
3 ttyp
4 /dev/vc/0
...
250 tarena
...
第一列：申请的主设备号
第二列：设备名称

mknod /dev/myled c 250 0
/home/drivers/led_test on
/home/dirvers/led_test off

总结：底层驱动write编写三步曲：
1.定义内核缓冲区
2.从用户缓冲区拷贝数据到内核缓冲区
3.根据用户的需求进行硬件数据写入操作

5. linux内核字符设备驱动硬件操作接口之read
回忆：应用read系统调用函数的使用
```
char msg[1024];
read(fd, msg, 1024); //从硬件设备读取数据到msg缓冲区
```

对应的底层驱动的read接口：
```
struct file_operations {
      ssize_t (*read) (struct file *file,
                       char __user *buf,
                       size_t count,
                       loff_t *ppos);
};
```
调用关系：
应用调read->C库read->软中断->内核sys_read->驱动read接口
功能：从硬件设备读取数据到用户缓冲区
参数：
file:文件指针
buf：此指针变量用__user修饰,表明此指针变量保存的地址位于
用户空间(0~3G),也就是保存用户缓冲区的首地址(例如msg)
虽然底层驱动可以访问buf向用户缓冲区写入数据,但是内核不允许驱动

直接操作这个用户缓冲区(例如：*(int *)buf=0x5555),
这么操作时相当危险的,如果驱动要想用户缓冲区写入数据
必须要利用内核提供的内存拷贝函数copy_to_user将内核缓冲区的
数据拷贝到用户缓冲区中

count:要读取的字节数

ppos：记录了上一次的读位置,参考代码：

```
loff_t pos = *ppos //获取上一次的读位置
这一次读了512字节,此接口返回之前,要记得更新读位置：
*ppos = pos + 512
```

友情提示：如果读操作只进行一次,无需记录读位置!
所以此参数用于多次读操作

内存拷贝函数copy_to_user

```
unsigned long copy_to_user(void __user *to,
                           const void  *from,
                           unsigned long n)
```

函数功能：将内核空间缓冲区的数据拷贝到用户空间的缓冲区中

参数：

to:目标,用户缓冲区的首地址

from:源,内核缓冲区的首地址

n:要拷贝的字节数

切记：此函数仅仅做了将内核内存的数据拷贝到用户的内存
此时还没有进行硬件操作,硬件操作还需要进行额外的操作

总结：数据流走向
对于write：
用户数据->经过一次拷贝到内核空间->经过二次拷贝到硬件
对于read：
硬件经过一次数据拷贝->内核,经过二次拷贝->用户

案例：获取灯的开关状态
总结：底层驱动read编写三步曲：
1.定义内核缓冲区
2.从硬件获取数据拷贝到内核缓冲区
3.将内核缓冲区数据最终拷贝到用户缓冲区

案例：应用能够任意开关某个灯
提示：应用应该向设备写入两个信息：开或者关和灯的编号(1或者2)
应用只需向驱动写入一个结构体即可：

```
struct led_cmd {
        int cmd; //开：1;关：0
        int index;//第一个灯：1；第二个灯：2
};
write(fd, 结构体,结构体大小);
```

底层驱动write接口用结构体来接收数据，
接收完解析判断结构体成员