Linux驱动工程师必知的三种获取结构体地址的方法

原创 Vincenterr 嵌入式Linux充电站 2025年02月12日 08:16 广东

大家好，我是Vincent。

今天分享一下在Linux驱动中，三种获取结构体地址的方法，包括**container_of**、**filp->private_data**和**dev_get_drvdata**

container_of估计较多人熟悉，因为八股文里经常出现，剩下两个可能做驱动开发的人，会见得比较多了。

container_of宏

对于传入参数中包含结构体中具体成员的，可利用container_of宏来实现结构体地址的获取。

```
#define container_of(ptr, type, member) (type *)((char *)(ptr) - (char *)&((type *)0)->member)
```

container_of需要传入三个参数：

- ptr:表示结构体中member的地址；
- type:表示结构体类型；
- member:表示对应第二个参数里面的结构体里面的成员；

返回结构体的首地址。

分析：((type *)0)将0转换为type类型的结构体指针，也就是让编译器认为这个结构体是开始于程序段起始位置0，开始于0地址的话，我们得到的成员变量的地址就直接等于成员变量的偏移地址。转换成 char * 类型，因为偏移量是以字节为单位，两者相减得到结构体的起始位置，再强制转换成 type 类型。

举例：

```
struct device {
    int id;
    char name[20];
};

struct my_device {
    struct device dev;
    int extra_data;
};

static int __init container_of_example_init(void)
{
    struct my_device my_dev;
    struct device *dev_ptr = &my_dev.dev;  // 获取指向 struct device 的指针
```

```
    // 初始化
    my_dev.dev.id = 123;
    strcpy(my_dev.dev.name, "Test Device");
    my_dev.extra_data = 456;

    // 使用 container_of 从 dev_ptr 获取整个 struct my_device 的指针
    struct my_device *container = container_of(dev_ptr, struct my_device, dev);

    printk(KERN_INFO "Device ID: %d\n", container->dev.id);
    printk(KERN_INFO "Device Name: %s\n", container->dev.name);
    printk(KERN_INFO "Extra Data: %d\n", container->extra_data);

    return 0;
}
 ........
```

在上述例子中，我们使用 `container_of` 宏，将 dev_ptr 指针传入，并通过 `container_of(dev_ptr, struct my_device, dev)` 获取包含该成员 dev 的完整结构体指针 `container` 。 使用 container 指针访问 my_device 中的所有字段，包括 dev.id 和 extra_data。

filp->private_data

传入参数中有结构体struct file *filp时，可利用filp->private_data实现

```
static function(struct file *filp, const char __user *buf, size_t count)
{
 struct sunxi_vir *chip = filp->private_data;
 ........
```

- private_data 数据结构

open函数提供给驱动来做任何的初始化来准备后续的操作。

```
int (*open)(struct inode *inode, struct file *filp);
```

`inode` 参数有我们需要的信息,以它的 i_cdev 成员的形式, 里面包含我们之前建立的cdev 结构. 通常我们不想要 cdev 结构本身, 而是需要包含 cdev 结构的 device_private 结构.

`struct file` 是字符设备驱动相关重要结构，代表一个打开的文件描述符。系统中每一个打开的文件在内核中都有一个关联的 struct file。 它由内核在 open时创建，并传递给在文件上操作的任何函数，知道最后关闭。当文件的所有实例都关闭之后，内核释放这个数据结构。

在 struct filed有个成员 `void *private_data` ;该成员是系统调用时保存状态信息非常有用的资源，在open函数被调用的时候 linux 系统就已经将其幅值为NULL，之后可供用户使用。

这个 `private_data` 其实是用来保存自定义设备结构体的地址的。自定义结构体的地址被保存在private_data后，可以在read ,write 等驱动函数中被传递和调用自定义设备结构体中的成员。

dev_get_drvdata

在probe函数中，malloc完相应的driver data结构体，填充完相应的域后，就会将driver data的地址赋值给driver data。

这样，在实现与其他子系统交互的接口时，就能通过其他子系统传递过来的device指针来找到相应的driver data。

举个例子：

1. 定义私有数据结构

```
struct my_device_data {
    int value;
    struct regmap *regmap;
    spinlock_t lock;
};
```

2. Probe 函数中设置数据

```
static int my_driver_probe(struct platform_device *pdev)
{
    struct device *dev = &pdev->dev;
    struct my_device_data *data;

    // 分配内存并初始化
    data = devm_kzalloc(dev, sizeof(*data), GFP_KERNEL);
    if (!data)
        return -ENOMEM;

    data->value = 42;
    spin_lock_init(&data->lock);

    // 将私有数据绑定到设备
```

```
    dev_set_drvdata(dev, data);

    // 其他初始化操作（如注册设备、配置硬件等）
    return0;
}
```

3. 在操作函数中获取数据

```
static ssize_t my_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    struct my_device_data *data = dev_get_drvdata(file->private_data);
    if (!data)
        return -ENODEV;

    // 使用私有数据
    spin_lock(&data->lock);
    // 读取操作...
    spin_unlock(&data->lock);
    return 0;
}
```

4. Remove 函数中清理数据

```
static int my_driver_remove(struct platform_device *pdev)
{
    struct device *dev = &pdev->dev;
    struct my_device_data *data = dev_get_drvdata(dev);

    // 清理资源（如注销设备、释放内存等）
    // 注意：devm_kzalloc 分配的内存会自动释放，无需手动操作
    return 0;
}
```

end

往期推荐

直到我干了底层开发，才知道不写业务代码有多爽

你解决bug的能力，暴露了你的水平

入职Linux驱动工程师后，我才知道的真相.....

很底层的性能优化：让CPU更快地执行你的代码

薪资倒挂，大家都沉默了...

机遇：我是如何走向Linux驱动的...

当我用几道题考了一遍做Linux驱动的同事.....

嵌入式Linux充电站

作者Vincent，分享一些嵌入式Linux、内核、RISC-V等知识。学习、沉淀、分享，才能有...

100篇原创内容

公众号

Linux驱动 33 linux内核 28

Linux驱动 · 目录

上一篇

揭秘驱动工程师在原厂的硅前验证工作

下一篇

原厂驱动工程师解析Linux启动中驱动加载的全过程（附电子版笔记，建议收藏!）