(2 条消息) module_init 机制的理解_芋圆 - 钰源的博客 - CSDN 博客_module_init

我们在学习 Linux 驱动开发时，首先需要了解 Linux 的模块化机制（module），但是 module 并不仅仅用于支撑驱动的加载和卸载。一个最简单的模块例子如下：

```
// filename: HelloWorld.c #include <linux/module.
```

模块代码有两种运行方式，一是静态编译连接进内核，在系统启动过程中进行初始化；一是编译成可动态加载的 module，通过 insmod 动态加载重定位到内核。这两种方式可以在 Makefile 中通过 obj-y 或 obj-m 选项进行选择。

而一旦可动态加载的模块目标代码（.ko）被加载重定位到内核，其作用域和静态链接的代码是**完全等价**的。所以这种运行方式的优点显而易见：

1. 可根据系统需要运行动态加载模块，以扩充内核功能，不需要时将其卸载，以释放内存空间；

2. 当需要修改内核功能时，只需编译相应模块，而不必重新编译整个内核。

因为这样的优点，在进行设备驱动开发时，基本上都是将其编译成可动态加载的模块。但是需要注意，有

些模块必须要编译到内核，随内核一起运行，从不卸载，如 vfs、platform_bus 等。

那么同样一份 C 代码如何实现这两种方式的呢？

答案就在于 **module_init** 宏！下面我们一起来分析 module_init 宏。（这里所用的 Linux 内核版本为 3.10.10）

定位到 Linux 内核源码中的 include/linux/init.h，可以看到有如下代码：

```
#ifndef MODULE// 省略#define module_init(x)  __in
```

显然，MODULE 是由 Makefile 控制的。上面部分用于将模块静态编译连接进内核，下面部分用于编译可动态加载的模块。接下来我们对这两种情况进行分析。

方式一：#ifndef MODULE

代码梳理：

```
#define module_init(x)  __initcall(x);|--> #defir
```

即 **module_init(hello_init)** 展开为：

```
static initcall_t __initcall_hello_init6 __used \
```

这里的 **initcall_t** 是函数指针类型，如下：

```
typedef int (*initcall_t)(void);
```

GNU 编译工具链支持用户自定义 section，所以我们阅读 Linux 源码时，会发现大量使用如下一类用法：

```
__attribute__((__section__("section-name")))
```

__attribute__用来指定变量或结构位域的特殊属性，其后的双括弧中的内容是属性说明，它的语法格式为：__attribute__ ((attribute-list))。它有位置的约束，通常放于声明的尾部且 “;” 之前。

这里的 attribute-list 为 __section__(“.initcall6.init”)。通常，编译器将生成的代码存放在. text 段中。但有时可能需要其他的段，或者需要将某些函数、变量存放在特殊的段中，section 属性就是用来指定将一个函数、变量存放在特定的段中。

所以这里的意思就是：定义一个名为 __initcall_hello_init6 的函数指针变量，并初始化为 hello_init (指向 hello_init) ；并且该函数指针变量存放于 .initcall6.init 代码段中。

接下来，我们通过查看链接脚本 (arch/$(ARCH)/kernel/vmlinux.lds.S) 来了解 .initcall6.init 段。

可以看到，.init 段中包含 INIT_CALLS，它定义在 include/asm-generic/vmlinux.lds.h。INIT_CALLS 展开后可得：

```
#define INIT_CALLS                                      \
```

进一步展开为：

```
__initcall_start = .;            \         *(.in
```

```
Freeing unused kernel memory: 124K (80312000 - 8
```

```
start_kernel|--> rest_init     |     --> kernel_th
```

那么存放于 .initcall6.init 段中的 __initcall_hello_init6 是怎么样被调用的呢？我们看文件 init/main.c，代码梳理如下：

```
static void __init do_initcalls(void){     int le
```

kernel_init 这个函数是作为一个内核线程被调用的（该线程最后会启动第一个用户进程 init）。

我们着重关注 do_initcalls 函数，如下：

```
static void __init do_initcall_level(int level){
```

函数 do_initcall_level 如下：

```
int __init_or_module do_one_initcall(initcall_t f
```

函数 do_one_initcall 如下：

```
static initcall_t *initcall_levels[] __initdata =
```

initcall_levels 的定义如下：

```
extern initcall_t __initcall_start[];extern initc
```

initcall_levels[] 中的成员来自于 INIT_CALLS 的展开，如 “__initcall0_start = .;”，这里的 __initcall0_start 是一个变量，它跟代码里面定义的变量的作用是一样的，所以代码里面能够使用 __initcall0_start。因此在 init/main.c 中可以通过 extern 的方法将这些变量引入，如下：

```
    #define module_init(initfn)                    \
```

到这里基本上就明白了，在 **do_initcalls** 函数中会遍历 **initcalls** 段中的每一个**函数指针**，然后执行这个函数指针。因为编译器根据链接脚本的要求将各个函数指针链接到了指定的位置，所以可以放心地用 do_one_initcall(*fn) 来执行相关初始化函数。

我们例子中的 **module_init(hello_init)** 是 level6 的 initcalls 段，比较靠后调用，很多外设驱动都调用 module_init 宏，如果是静态编译连接进内核，则这些函数指针会按照编译先后顺序插入到 **initcall6.init** 段中，然后等待 **do_initcalls** 函数调用。

方式二：#else

相关代码：

```
    #define module_init(initfn)                    \
```

__inittest 仅仅是为了检测定义的函数是否符合 **initcall_t** 类型，如果不是 __inittest 类型在编译时将会报错。所以真正的宏定义是：

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

因此，用动态加载方式时，可以不使用 module_init 和 module_exit 宏，而直接定义 **init_module** 和 **cleanup_module** 函数，效果是一样的。

alias 属性是 gcc 的特有属性，将定义 init_module 为函数 initfn 的别名。所以 module_init(hello_init) 的作用就是定义一个变量名 **init_module**，其地址和 **hello_init** 是一样的。

上述例子编译可动态加载模块过程中，会自动产生 HelloWorld.mod.c 文件，内容如下：

```
insmod_main|--> bb_init_module     |     --> init_
```

可知，其定义了一个类型为 module 的全局变量 **__this_module**，成员 **init** 为 init_module（即 hello_init），且该变量链接到 **.gnu.linkonce.this_module** 段中。

编译后所得的 HelloWorld.ko 需要通过 **insmod** 将其加载进内核，由于 insmod 是 busybox 提供的用户层命令，所以我们需要阅读 busybox 源码。代码梳理如下：（文件 busybox/modutils/insmod.c）

```
#define init_module(mod, len, opts) syscall(__NR_
```

而 init_module 定义如下：（文件 busybox/modutils/modutils.c ）

```
SYSCALL_DEFINE3(init_module, ...)|-->load_module
```

因此，该系统调用对应内核层的 **sys_init_module** 函数。

回到 Linux 内核源代码（ kernel/module.c ），代码梳理：

```
#define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINE
```

文件（ include/linux/syscalls.h ）中，有：

```
#define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINE
```

从而形成 **sys_init_module** 函数。

全文完