int request\_threaded\_irq(unsigned int irq, irq\_handler\_t handler, irq\_handler\_t thread\_fn,

                                                      unsigned long irqflags,const char \*devname, void \*dev\_id);

分析request\_threaded\_irq()函数中的各个形参

1>：irq:表示申请的中断号。

2>：handler:表示中断服务例程

3.> thread\_fn：中断线程化，此处传递的是NULL。NULL表示没有中断线程化。

    在 Linux 中，中断具有最高的优先级。不论在任何时刻，只要产生中断事件，内核将立即执行相应的中断处理程序，等到所有挂起的中断和软中断处理完毕后才能执行正常的任务，因此有可能造成实时任务得不到及时的处理。中断线程化之后，中断将作为内核线程运行而且被赋予不同的实时优先级，实时任务可以有比中断线程更高的优先级。这样，具有最高优先级的实时任务就能得到优先处理，即使在严重负载下仍有实时性保证。but,并不是所有的中断都可以被线程化，比如时钟中断，主要用来维护系统时间以及定时器等，其中定时器是操作系统的脉搏，一旦被线程化，就有可能被挂起，这样后果将不堪设想，所以不应当被线程化。

4>.irqflags:表示中断标志位。

5>.devname:表示请求中断的设备的名称。

6>.dev\_id: 对应于request\_irq()函数中所传递的第五个参数，可取任意值，但必须唯一能够代表发出中断请求的设备，通常取描述该设备的结构体。 共享中断时所用。

int devm\_request\_threaded\_irq(struct device \*dev, unsigned int irq,

         irq\_handler\_t handler, irq\_handler\_t thread\_fn,

         unsigned long irqflags, const char \*devname, void \*dev\_id)

{

     struct irq\_devres \*dr;

     int rc;

    dr = devres\_alloc(devm\_irq\_release, sizeof(struct irq\_devres),GFP\_KERNEL);

    if (!dr)

       return -ENOMEM;

   rc = request\_threaded\_irq(irq, handler, thread\_fn, irqflags,devname, dev\_id);

   if (rc) {

      devres\_free(dr);

      return rc;

   }

   dr->irq = irq;

   dr->dev\_id = dev\_id;

   devres\_add(dev, dr);

   return 0;

}

# init

————————————————

## \_\_init 的设计哲学

\_\_init 的定义在include/linux/init.h

#define \_\_init \_\_attribute\_\_ ((\_\_section\_\_ (".init.text")))

那么\_\_attribute\_\_是什么？Linux内核代码使用了大量的GNU C扩展，

以至于GNU C成为能够编译内核的唯一编译器，GNU C的这些扩展对代码优化、目标代码

布局、安全检查等方面也提供了很强的支持。而\_\_attribute\_\_就是这些扩展中的一个，

它主要被用来声明一些特殊的属性，这些属性主要被用来指示编译器进行特定方面的优化和

更仔细的代码检查。GNU C支持十几个属性，section是其中的一个，我们查看gcc的手册

可以看到下面的描述

通常编译器将函数放在.text 节，变量放在.data 或 .bss 节，使用section 属性，可以让

编译器将函数或变量放在指定的节中。那么前面对\_\_init 的定义便表示将它修饰的代码放在.init.text 节。连接器可以把相同节的代码或数据安排在一起，比如\_\_init 修饰的所有代码都会被放在.init.text 节里，初始化结束后就可以释放这部分内存。

## 那内核又是如何调用到这些\_\_init

那内核又是如何调用到这些\_\_init 修饰的初始化函数那？要回答这个问题，还需要回顾一下上面已经提到subsys\_initcall也是一个宏，它也在include/linux/init.h里定义

#define subsys\_initcall(fn) module\_init(fn)

#define module\_init(x) \_\_initcall(x);

#define \_\_initcall(fn) device\_initcall(fn)

#define device\_initcall(fn) \_\_define\_initcall(fn, 6)

#define \_\_define\_initcall(fn, id) \  
 static initcall\_t \_\_initcall\_##fn##id \_\_used \  
 \_\_attribute\_\_((\_\_section\_\_(".initcall" #id ".init"))) = fn; \  
 LTO\_REFERENCE\_INITCALL(\_\_initcall\_##fn##id)

将指定的函数指针fn 放到initcall.init 节里，对于具体的subsys\_initcall 宏，则是把fn 放到.initcall.init 的子节.initcall6.init 里

要弄清楚.initcall.init、.init.text 和.initcall6.init，我们还需要了解一点内核可执行文件相关的概念，内核可执行文件由许多链接在一起的对象文件组成。对象文件有许多节，如文本、数据、

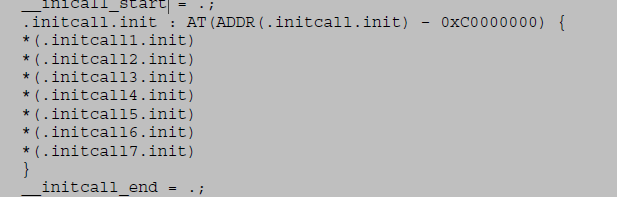
init 数据、bass 等。这些对象文件都是由一个称为链接器脚本的文件链接并装入的。这个链接器脚本的功能是将输入对象文件的各节映射到输出文件中

换句话说，它将所有输入对象的文件都链接到单一的可执行文件中，将该可执行文件的各节装入到指定地址处。 vmlinux.lds 是存在于arch/<target>/ 目录中的内核链接器脚本，它负责链接内核的各个节并将它们装入内存中特定的偏移量处。

E:\K\kernel\msm-android-msm-marlin-3.18-pie-qpr2\arch\arm64\kernel\vmlinux.lds.S

要看懂vmlinux.lds 这个文件是需要花一番工夫的，不过大家都

是聪明人，聪明人做聪明事，所以你需要做的只是搜索initcall.init



这里的\_\_initcall\_start 指向.initcall.init 节的开始，\_\_initcall\_end 指向它的结尾。而.initcall.init节又被分为了7 个子节，分别如下。

我们的subsys\_initcall 宏便是将指定的函数指针放在了.initcall4.init 子节。其他的比如

core\_initcall 将函数指针放在.initcall1.init 子节，device\_initcall 将函数指针放在了.initcall6.init 子节等，都可以从include/linux/init.h 文件找到它们的定义。各个子节的顺序是确定的，即先调

用.initcall1.init 中的函数指针，再调用.initcall2.init 中的函数指针等。\_\_init 修饰的初始化函数在内核初始化过程中调用的顺序和.initcall.init 节里函数指针的顺序有关，不同的初始化函数被放在不同的子节中，因此，也就决定了它们的调用顺序。

至于实际执行函数调用的地方，就在/init/main.c 文件中，内核的初始化不在那里还能在哪里？do\_initcalls 函数会直接用到这里的\_\_initcall\_start、\_\_initcall\_end 来进行判断。不多说了