attribute 用法 section 部分

1. gcc 的__attribute__ 编译属性

要了解 Linux Kernel 代码的分段信息，需要了解一下 gcc 的__attribute__ 的编绎属性，

__attribute__ 主要用于改变所声明或定义的函数或 数据的特性，它有很多子项，用于改变

作用对象的特性。比如对函数， noline 将禁止进行内联扩展、 noreturn 表示没有返回值、

pure 表明函数除 返回值外， 不会通过其它 （如全局变量、 指针） 对函数外部产生任何影响。

但这里我们比较感兴趣的是对代码段起作用子项 section 。

__attribute__ 的 section 子项的使用格式为：

__attribute__((section("section_name")))

其作用是将作用的函数或数据放入指定名为 "section_name" 输入段。

这里还要注意一下两个概念：输入段和输出段

输入段和输出段是相对于要生成最终的 elf 或 binary 时的 Link 过程说的， Link 过程的输入

大都是由源代码编绎生成的目标文件 .o，那么这些 .o 文件中包含的段相对 link 过程来说就

是输入段，而 Link 的输出一般是可执行文件 elf 或库等，这些输出文件中也包含有段，这

些输出文件中的段就叫做输 出段。输入段和输出段本来没有什么必然的联系， 是互相独立，

只是在 Link 过程中， Link 程序会根据一定的规则（这些规则其实来源于 Link Script ），将

不同的输入段重新组合到不同的输出段中， 即使是段的名字， 输入段和输出段可以完全不同。

其用法举例如下：

```
int var __attribute__((section(".xdata"))) = 0;
```

这样定义的变量 var 将被放入名为 .xdata 的输入段, （注意： __attribute__ 这种用法中的括号好像很严格，这里的几个括号好象一个也不能少。）

```
static int __attribute__((section(".xinit"))) functionA(void)

{

.....

}
```

这个例子将使函数 functionA 被放入名叫 .xinit 的输入段。

需要着重注意的是, __attribute__ 的 section 属性只指定对象的输入段, 它并不能影响所指定对象最终会放在可执行文件的什么段。

2. linux Kernel 源代码中与段有关的重要宏定义

A．关于 __init 、__initdata 、__exit 、__exitdata 及类似的宏

打开 Linux Kernel 源代码树中的文件： include/init.h ，可以看到有下面的宏定议：

```
#define __init __attribute__ ((__section__ (".init.text"))) __cold

#define __initdata __attribute__ (( __section__ (".init.data")))

#define __exitdata __attribute__ (( __section__ (".exit.data")))
```

```
#define __exit_call __attribute_used__ __attribute__ (( __section__ (".exitcall.exit")))

#define __init_refok oninline __attribute__ ((__section__ (".text.init.refok")))

#define __initdata_refok __attribute__ ((__section__ (".data.init.refok")))

#define __exit_refok noinline __attribute__ ((__section__ (".exit.text.refok")))

.....

#ifdef MODULE

#define __exit __attribute__ (( __section__ (".exit.text"))) __cold

#else

#define __exit __attribute_used__ __attribute__ ((__section__ (".exit.text"))) __cold

#endif
```

对于经常写驱动模块或翻阅 Kernel 源代码的人，看到熟悉的宏了吧： __init, __initdata, __exit, __exitdata 。

__init 宏最常用的地方是驱动模块初始化函数的定义处， 其目的是将驱动模块的初始化函数放入名叫 .init.text 的输入段。对于 __initdata 来说，用 于数据定义，目的是将数据放入名叫.init.data 的输入段。其它几个宏也类似。 另外需要注意的是， 在以上定意中， 用__section__ 代替了 section 。还有其它一些类似的宏定义，这里不一一列出，其作用都是类似的。

B．关于 initcall 的一些宏定义

在该文件中，下面这条宏定议更为重要，它是一条可扩展的宏：

__attribute__ ((__section__(".initcall" level ".init"))) = fn

这条宏带有 3 个参数： level,fn, id ，分析该宏可以看出：

1 .其用来定义类型为 initcall_t 的 static 函数指针，函数指针的名称由参数 fn 和 id 决定：__initcall_##fn##id ，这 就是函数指针的名称，它其实是一个变量名称。从该名称的定义方法我们其学到了宏定义的一种高级用法，即利用宏的参数产生名称，这要借助于 "##" 这一符号 组合的作用。

2．这一函数指针变量放入什么输入段呢，请看 __attribute__ ((__section__ (".initcall" levle ".init"))) ，输入段的名称由 level 决定，如果 level="1" ，则输入段是 .initcall1.init ，如果 level="3s" ，则输入段是 .initcall3s.init 。这一函数指针变量就是放在用这种方法决定的输入段中的。

3．这一定义的函数指针变量的初始值是什么叫，其实就是宏参数 fn，实际使用中， fn 其实就是真实定义好的函数。

该宏定义并不直接使用，请看接下来的这些宏定义：

```
#define pure_initcall(fn) __define_initcall("0",fn,0)

#define core_initcall(fn) __define_initcall("1",fn,1)

#define core_initcall_sync(fn) __define_initcall("1s",fn,1s)

#define postcore_initcall(fn) __define_initcall("2",fn,2)

#define postcore_initcall_sync(fn) __define_initcall("2s",fn,2s)

#define arch_initcall(fn) __define_initcall("3",fn,3)

#define arch_initcall_sync(fn) __define_initcall("3s",fn,3s)

#define subsys_initcall(fn) __define_initcall("4",fn,4)

#define subsys_initcall_sync(fn) __define_initcall("4s",fn,4s)

#define fs_initcall(fn) __define_initcall("5",fn,5)

#define fs_initcall_sync(fn) __define_initcall("5s",fn,5s)

#define rootfs_initcall(fn) __define_initcall("rootfs",fn,rootfs)

#define device_initcall(fn) __define_initcall("6",fn,6)

#define device_initcall_sync(fn) __define_initcall("6s",fn,6s)

#define late_initcall(fn) __define_initcall("7",fn,7)

#define late_initcall_sync(fn) __define_initcall("7s",fn,7s)
```

这些宏定义出来是为了方便的使用 __define_initcall 宏定义的, 上面每条宏第一次使用时都会产生一个新的输入段。

```
接下来还有一条
```

#define __initcall(fn) device_initcall(fn)

这一条其实只是定义了另一个别名，即平常使用的 __initcall 其实就是这儿的 device_initcall ，用它定义的函数指定位于段 .initcall6.init 中。

C. __setup 宏的来源及使用

__setup 这条宏在 Linux Kernel 中使用最多的地方就是定义处理 Kernel 启动参数的函数及数据结构，请看下面的宏定义：

```
#define __setup_param(str, unique_id, fn, early) \
static char __setup_str_##unique_id[] __initdata __aligned(1) = str; \
static struct obs_kernel_param __setup_##unique_id \
__used __section(.init.setup) \
__attribute__((aligned((sizeof(long))))) \
= { __setup_str_##unique_id, fn, early }

#define __setup(str, fn) \
__setup_param(str, fn, fn, 0)
```

使用 Kernel 中的例子分析一下这两条定义：

```
__setup("root=",root_dev_setup);
```

这条语句出现在 init/do_mounts.c 中，其作用是处理 Kernel 启动时的像 root=/dev/mtdblock3 之类的参数的。

分解一下这条语句，首先变为：

```
__setup_param("root=",root_dev_setup,root_dev_setup,0);
```

继续分解，将得到下面这段代吗：

```
static char __setup_str_root_dev_setup_id[] __initdata __aligned(1) = "root=";

static struct obs_kernel_param __setup_root_dev_setup_id

__used __section(.init.setup)

__attribute__((aligned((sizeof(long)))))

= { __setup_str_root_dev_setup_id, root_dev_setup, 0 };
```

这段代码定义了两个变量：字符数组变量 __setup_str_root_dev_setup_id ，其初始化内容为"root=" ，由于该变量用 __initdata 修饰，它将被放入 .init.data 输入段；另一变量是结构变量 __setup_root_dev_setup_id ，其类型为 struct obs_kernel_param ， 该变理被放入输入段 .init.setup 中。结构 struct obs_kernel_param 也在该文件中定义如下：

```
struct obs_kernel_param {

const char *str;

int (*setup_func)(char *);

int early;

};
```

变量 __setup_root_dev_setup_id 的三个成员分别被初始化为：

__setup_str_root_dev_setup_id　　－－ > 前面定义的字符数组变量，初始内容为　　"root="　。

root_dev_setup　　－－ > 通过宏传过来的处理函数。

0 －－ > 常量 0，该成员的作用以后分析。

现在不难想像内核启动时怎么处理启动参数的了：　　通过 __setup 宏定义 obs_kernel_param 结构变量都被放入 .init.setup 段中，　这样一来实际是使 .init.setup 段变成一张表，　Kernel 在处理每一个启动参数时，　都会来查找这张表，　与每一个数据项中的成员　　str 进行比较，　如果完全相同，就会调用该数据项的函数指针成员　　setup_func 所指向的函数（该函数是在使用__setup 宏定义该变量时传入的函数参数），并将启　　动参数如 root= 后面的内容传给该处理函数

可参考： http://ohse.de/uwe/articles/gcc-attributes.html#func-fastcall