Linux设备树语法详解 - Abnor

概念

Linux内核从3.x开始引入设备树的概念，用于实现驱动代码与设备信息相分离。在设备树出现以前，所有关于设备的具体信息都要写在驱动里，一旦外围设备变化，驱动代码就要重写。引入了设备树之后，驱动代码只负责处理驱动的逻辑，而关于设备的具体信息存放到设备树文件中，这样，如果只是硬件接口信息的变化而没有驱动逻辑的变化，驱动开发者只需要修改设备树文件信息，不需要改写驱动代码。比如在ARM Linux内，一个.dts(device tree source)文件对应一个ARM的machine，一般放置在内核的"arch/arm/boot/dts/"目录内，比如exynos4412参考板的板级设备树文件就是"arch/arm/boot/dts/exynos4412-origen.dts"。这个文件可以通过`$make dtbs`命令编译成二进制的.dtb文件供内核驱动使用。

基于同样的软件分层设计的思想，由于一个SoC可能对应多个machine，如果每个machine的设备树都写成一个完全独立的.dts文件，那么势必相当一些.dts文件有重复的部分，为了解决这个问题，Linux设备树目录把一个SoC公用的部分或者多个machine共同的部分提炼为相应的.dtsi文件。这样每个.dts就只有自己差异的部分，公有的部分只需要"include"相应的.dtsi文件，这样就是整个设备树的管理更加有序。我这里用`Linux4.8.5源码自带的dm9000网卡为例来分析设备树的使用和移植。这个网卡的设备树节点信息在"Documentation/devicetree/bindings/net/davicom-dm9000.txt"有详细说明，其网卡驱动源码是"drivers/net/ethernet/davicom/dm9000.c"。

```
Davicom DM9000 Fast Ethernet controller

Required properties:
- compatible = "davicom,dm9000";
- reg : physical addresses and sizes of registers, must contain 2 entries:
    first entry : address register,
    second entry : data register.
- interrupt-parent : interrupt controller to which the device is connected
- interrupts : interrupt specifier specific to interrupt controller

Optional properties:
- davicom,no-eeprom : Configuration EEPROM is not available
- davicom,ext-phy : Use external PHY
- reset-gpios : phandle of gpio that will be used to reset chip during probe
- vcc-supply : phandle of regulator that will be used to enable power to chip

Example:

        ethernet@18000000 {
                compatible = "davicom,dm9000";
                reg = <0x18000000 0x2 0x18000004 0x2>;
                interrupt-parent = <&gpn>;
                interrupts = <7 4>;
                local-mac-address = [00 00 de ad be ef];
                davicom,no-eeprom;
                reset-gpios = <&gpf 12 GPIO_ACTIVE_LOW>;
                vcc-supply = <&eth0_power>;
        };
```

设备树框架

设备树用树状结构描述设备信息，它有以下几种特性

1. 每个设备树文件都有一个根节点，每个设备都是一个节点。
2. 节点间可以嵌套，形成父子关系，这样就可以方便的描述设备间的关系。
3. 每个设备的属性都用一组key-value对(键值对)来描述。
4. 每个属性的描述用;结束

所以，一个设备树的基本框架可以写成下面这个样子，一般来说，/表示板子，它的子节点node1表示SoC上的某个控制器，控制器中的子节点node2表示挂接在这个控制器上的设备(们)。

```
/{                                  //根节点
    node1{                          //node1是节点名，是/的子节点
        key=value;                  //node1的属性
        ...
        node2{                      //node2是node1的子节点
            key=value;              //node2的属性
            ...
        }
    }                               //node1的描述到此为止
    node3{
```

```
        key=value;
        ...
    }
}
```

节点名

理论个节点名只要是长度不超过31个字符的ASCII字符串即可，此外
Linux内核还约定设备名应写成形如`<name>[@<unit_address>]`的形式，其中name就是设备名，最长可以是31个字符长度。unit_address一般是设备地址，用来唯一标识一个节点，下面就是典型节点名的写法

```
34          firmware@0203F000 {
35                  compatible = "samsung,secure-firmware";
36                  reg = <0x0203F000 0x1000>;
37          };
```

上面的节点名是`firmware`，节点路径是`/firmware@0203f000`,这点要注意，因为根据节点名查找节点的API的参数是不能有"@xxx"这部分的。

Linux中的设备树还包括几个特殊的节点，比如chosen，chosen节点不描述一个真实设备，而是用于firmware传递一些数据给OS，比如bootloader传递内核启动参数给内核

```
29          chosen {
30                  bootargs ="console=ttySAC2,115200";
31                  stdout-path = &serial_2;
32          };
```

引用

当我们找一个节点的时候，我们必须书写完整的节点路径，这样当一个节点嵌套比较深的时候就不是很方便，所以，设备树允许我们用下面的形式为节点标注引用(起别名)，借以省去冗长的路径。这样就可以实现类似函数调用的效果。编译设备树的时候，相同的节点的不同属性信息都会被合并，相同节点的相同的属性会被重写，使用引用可以避免移植者四处找节点，直接在板级.dts增改即可。

```
576                 gpx0: gpx0 {
577                         gpio-controller;
578                         #gpio-cells = <2>;
579
580                         interrupt-controller;
581                         interrupt-parent = <&gic>;
582                         interrupts = <0 16 0>, <0 17 0>, <0 18 0>, <0 19 0>,
583                                      <0 20 0>, <0 21 0>, <0 22 0>, <0 23 0>;
584                         #interrupt-cells = <2>;
585                 };
```

下面的例子中就是直接引用了dtsi中的一个节点，并向其中添加/修改新的属性信息

```
 99 &cpu0 {
100         cpu0-supply = <&buck2_reg>;
101 };
```

KEY

在设备树中，键值对是描述属性的方式，比如，Linux驱动中可以通过设备节点中的"compatible"这个属性查找设备节点。
Linux设备树语法中定义了一些具有规范意义的属性，包括：compatible, address, interrupt等，这些信息能够在内核初始化找到节点的时候，自动解析生成相应的设备信息。此外，还有一些Linux内核定义好的，一类设备通用的有默认意义的属性，这些属性一般不能被内核自动解析生成相应的设备信息，但是内核已经编写的相应的解析提取函数，常见的有 "mac_addr"，"gpio"，"clock"，"power"。"regulator" 等等。

compatible

设备节点中对应的节点信息已经被内核构造成struct platform_device。驱动可以通过相应的函数从中提取信息。compatible属性是用来查找节点的方法之一，另外还可以通过节点名或节点路径查找指定节点。dm9000驱动中就是使用下面这个函数通过设备节点中的"compatible"属性提取相应的信息，所以二者的字符串需要严格匹配。
在下面的这个dm9000的例子中，我们在相应的板级dts中找到了这样的代码块：

```
87                  ethernet@5000000 {
88                          compatible = "davicom,dm9000";
```

然后我们取内核源码中找到dm9000的网卡驱动，从中可以发现这个驱动是使用的设备树描述的设备信息(这不废话么，显然用设备树好处多多)。我们可以找到它用来描述设备信息的结构体，可以看出，驱动中用于匹配的结构使用的compatible和设备树中一模一样，否则就可能无法匹配，这里另外的一点是struct of_device_id数组的最后一个成员一定是空，因为相关的操作API会读取这个数组直到遇到一个空。

```
1786 #ifdef CONFIG_OF
1787 static const struct of_device_id dm9000_of_matches[] = {
1788         { .compatible = "davicom,dm9000", },
1789         { /* sentinel */ }
1790 };
1791 MODULE_DEVICE_TABLE(of, dm9000_of_matches);
1792 #endif
1793
1794 static struct platform_driver dm9000_driver = {
1795         .driver = {
1796                 .name    = "dm9000",
1797                 .pm      = &dm9000_drv_pm_ops,
1798                 .of_match_table = of_match_ptr(dm9000_of_matches),
1799         },
1800         .probe   = dm9000_probe,
1801         .remove  = dm9000_drv_remove,
1802 };
```

address

(几乎)所有的设备都需要与CPU的IO口相连，所以其IO端口信息就需要在设备节点节点中说明。常用的属性有

- #address-cells，用来描述子节点"reg"属性的地址表中用来描述首地址的cell的数量，
- #size-cells，用来描述子节点"reg"属性的地址表中用来描述地址长度的cell的数量。

有了这两个属性，子节点中的"reg"就可以描述一块连续的地址区域。下例中，父节点中指定了#address-cells = <2>;#size-cells = <1>，则子节点dev-bootscs0中的reg中的前两个数表示一个地址,即MBUS_ID(0xf0, 0x01)和0x1045C，最后一个数的表示地址跨度，即是0x4

```
22         soc {
23                 #address-cells = <2>;
24                 #size-cells = <1>;
25                 controller = <&mbusc>;
26
27                 devbus_bootcs: devbus-bootcs {
28                         compatible = "marvell,orion-devbus";
29                         reg = <MBUS_ID(0xf0, 0x01) 0x1046C 0x4>;
30                         ranges = <0 MBUS_ID(0x01, 0x0f) 0 0xffffffff>;
31                         #address-cells = <1>;
32                         #size-cells = <1>;
33                         clocks = <&core_clk 0>;
34                         status = "disabled";
35                 };
```

interrupts

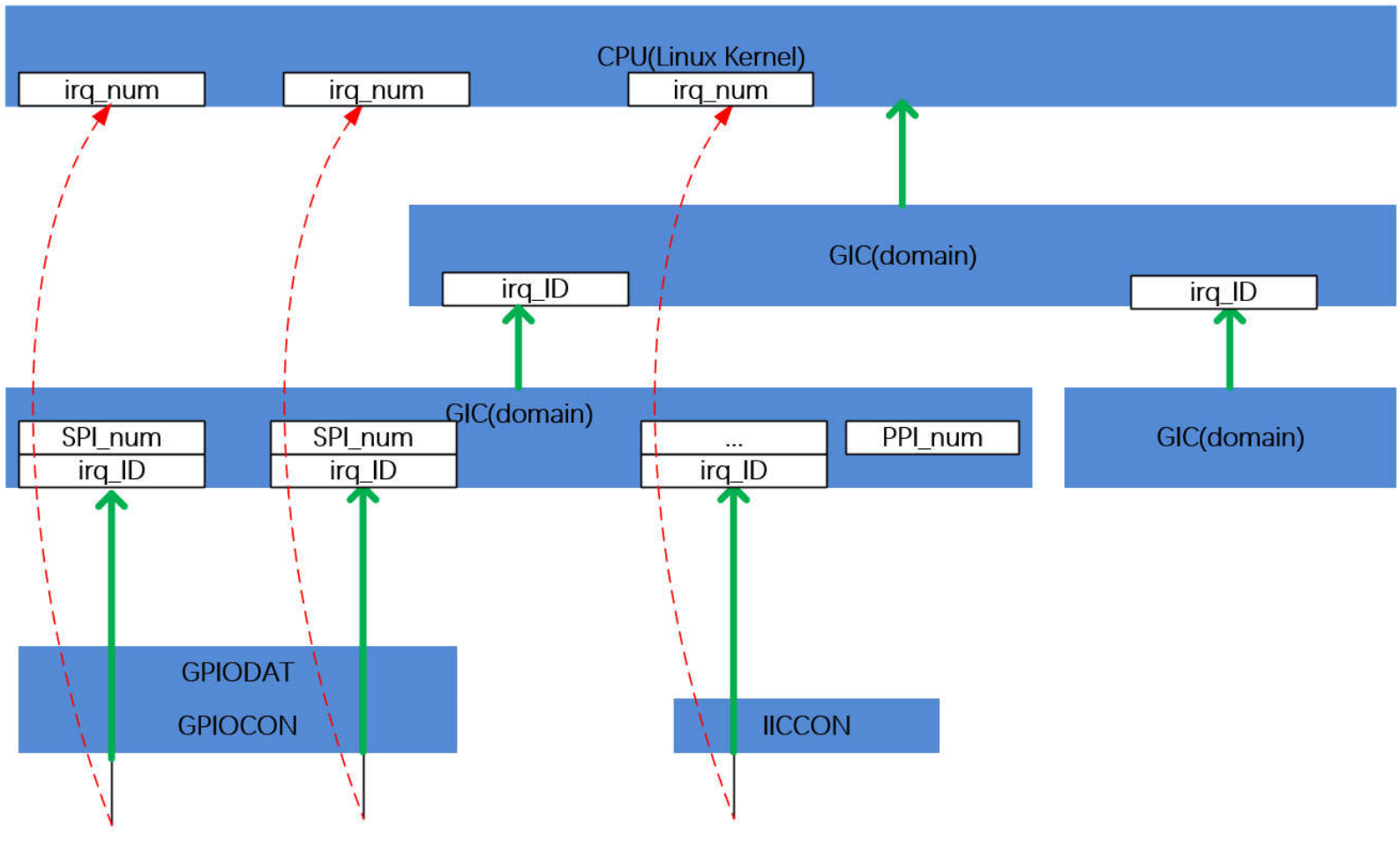
一个计算机系统中大量设备都是通过中断请求CPU服务的，所以设备节点中就需要在指定中断号。常用的属性有

- interrupt-controller 一个空属性用来声明这个node接收中断信号，即这个node是一个中断控制器。
- #interrupt-cells，是中断控制器节点的属性，用来标识这个控制器需要几个单位做中断描述符,用来描述子节点中"interrupts"属性使用了父节点中的interrupts属性的具体的哪个值。一般，如果父节点的该属性的值是3，则子节点的interrupts一个cell的三个32bits整数值分别为:<中断域 中断 触发方式>,如果父节点的该属性是2，则是<中断 触发方式>
- interrupt-parent,标识此设备节点属于哪一个中断控制器，如果没有设置这个属性，会自动依附父节点的
- interrupts,一个中断标识符列表，表示每一个中断输出信号

设备树中中断的部分涉及的部分比较多，interrupt-controller表示这个节点是一个中断控制器，需要注意的是，一个SoC中可能有不止一个中断控制器，这就会涉及到设备树中断组织的很多概念，下面是在文件"arch/arm/boot/dts/exynos4.dtsi"中对exynos4412的中断控制器(GIC)节点描述：

```
134         gic: interrupt-controller@10490000 {
135                 compatible = "arm,cortex-a9-gic";
136                 #interrupt-cells = <3>;
137                 interrupt-controller;
138                 reg = <0x10490000 0x10000>, <0x10480000 0x10000>;
139         };
```

要说interrupt-parent，就得首先讲讲Linux设备管理中对中断的设计思路演变。随着linux kernel的发展，在内核中将interrupt controller抽象成irqchip这个概念越来越流行，甚至GPIO controller也可以被看出一个interrupt controller chip，这样，系统中至少有两个中断控制器了，另外，在硬件上，随着系统复杂度加大，外设中断数据增加，实际上系统可以需要多个中断控制器进行级联，形成事实上的硬件中断处理结构：



在这种趋势下，内核中原本的中断源直接到中断号的方式已经很难继续发展了，为了解决这些问题，linux kernel的大牛们就创造了irq domain(中断域)这个概念。domain在内核中有很多，除了irqdomain，还有power domain，clock domain等等，所谓domain，就是领域，范围的意思，也就是说，任何的定义出了这个范围就没有意义了。如上所述，系统中所有的interrupt controller会形成树状结构，对于每个interrupt controller都可以连接若干个外设的中断请求（interrupt source，中断源），interrupt controller会对连接其上的interrupt source（根据其在Interrupt controller中物理特性）进行编号（也就是HW interrupt ID了）。有了irq domain这个概念之后，这个编号仅仅限制在本interrupt controller范围内，有了这样的设计，CPU(Linux 内核)就可以根据级联的规则一级一级的找到想要访问的中断。当然，通常我们关心的只是内核中的中断号，具体这个中断号是怎么找到相应的中断源的，我们作为程序员往往不需要关心，除了在写设备树的时候，设备树就是要描述嵌入式软件开发中涉及的所有硬件信息，所以，设备树就需要准确的描述硬件上处理中断的这种树状结构，如此，就有了我们的interrupt-parant这样的概念：用来连接这样的树状结构的上下级，用于表示这个中断归属于哪个interrupt controller，比如，一个接在GPIO上的按键，它的组织形式就是：

> 中断源--interrupt parent-->GPIO--interrupt parent-->GIC1--interrupt parent-->GIC2--...—>CPU

有了parant，我们就可以使用一级一级的偏移量来最终获得当前中断的绝对编号，这里，可以看出，在我板子上的dm9000的的设备节点中，它的"interrupt-parent"引用了"exynos4x12-pinctrl.dtsi"（被板级设备树的exynos4412.dtsi包含）中的gpx0节点：

```
87                      ethernet@5000000 {
88                              compatible = "davicom,dm9000";
89                              reg = <0x05000000 0x2 0x05000004 0x2>;
90                              interrupt-parent = <&gpx0>;
91                              interrupts = <6 4>;
```

而在gpx0节点中，指定了"#interrupt-cells = <2>;"，所以在dm9000中的属性"interrupts = <6 4>;"表示dm9000的的中断在作为irq parant的gpx0中的中断偏移量，即gpx0中的属性"interrupts"中的"<0 22 0>"，通过查阅exynos4412的手册知道，对应的中断号是EINT[6]。

```
576                     gpx0: gpx0 {
577                             gpio-controller;
578                             #gpio-cells = <2>;
579
580                             interrupt-controller;
581                             interrupt-parent = <&gic>;
582                             interrupts = <0 16 0>, <0 17 0>, <0 18 0>, <0 19 0>,
583                                          <0 20 0>, <0 21 0>, <0 22 0>, <0 23 0>;
584                             #interrupt-cells = <2>;
585                     };
```

gpio

gpio也是最常见的IO口，常用的属性有

- "gpio-controller"，用来说明该节点描述的是一个gpio控制器
- "#gpio-cells"，用来描述gpio使用节点的属性一个cell的内容，即 `属性 = <&引用GPIO节点别名 GPIO标号 工作模式>

GPIO的设置同样采用了上述偏移量的思想，比如下面的这个led的设备书，表示使用GPX2组的第7个引脚：

```
97          led@4{
98                  compatible = "xj4412,key";
99                  gpx2_7 = <&gpx2 7 0>;
100                 gpx1_0 = <&gpx1 0 0>;
101                 gpf3_45 = <&gpf3 4 0>, <&gpf3 5 0>;
102         };
```

驱动自定义key

针对具体的设备，有部分属性很难做到通用，需要驱动自己定义好，通过内核的属性提取解析函数进行值的获取，比如dm9000节点中的下面这句就是自定义的节点属性，用以表示配置EEPROM不可用。

```
93                          davicom,no-eeprom;
```

VALUE
dts描述一个键的值有多种方式，当然，一个键也可以没有值

字符串信息

```
77                          compatible = "samsung,clock-xusbxti";
```

32bit无符号整型数组信息

```
89                          reg = <0x05000000 0x2 0x05000004 0x2>;
```

二进制数数组

```
92                          local-mac-address = [00 00 de ad be ef];
```

字符串哈希表

```
23          compatible = "insignal,origen4412", "samsung,exynos4412", "samsung,exynos4";
```

混合形式

上述几种的混合形式

设备树/驱动移植实例

设备树就是为驱动服务的，配置好设备树之后还需要配置相应的驱动才能检测配置是否正确。比如dm9000网卡，就需要首先将示例信息挂接到我们的板级设备树上，并根据芯片手册和电路原理图将相应的属性进行配置，再配置相应的驱动。需要注意的是，dm9000的地址线一般是接在片选线上的，所以设备树中就应该归属与相应片选线节点，我这里用的exynos4412，接在了bank1，所以是"<0x50000000 0x2 0x50000004 0x2>"
最终的配置结果是：

```
81          srom-cs1@5000000{
82                  compatible = "simple-bus";
83                  #address-cells = <1>;
84                  #size-cells = <1>;
85                  reg = <0x05000000 0x01000000>;
86                  ranges;
87                  ethernet@5000000 {
88                          compatible = "davicom,dm9000";
89                          reg = <0x05000000 0x2 0x05000004 0x2>;
90                          interrupt-parent = <&gpx0>;
91                          interrupts = <6 4>;
92                          local-mac-address = [00 00 de ad be ef];
93                          davicom,no-eeprom;
94                  };
```

勾选相应的选项将dm9000的驱动编译进内核。

```
make menuconfig
[*] Networking support  --->
    Networking options  --->
        <*> Packet socket
        <*>Unix domain sockets
        [*] TCP/IP networking
        [*]   IP: kernel level autoconfiguration
Device Drivers  --->
    [*] Network device support  --->
        [*]   Ethernet driver support (NEW)  --->
```

```
            <*>   DM9000 support
File systems  --->
    [*] Network File Systems (NEW)  --->
        <*>   NFS client support
        [*]     NFS client support for NFS version 3
        [*]       NFS client support for the NFSv3 ACL protocol extension
        [*]   Root file system on NFS
```

执行`make uImage;make dtbs`,tftp下载，成功加载nfs根文件系统并进入系统，表示网卡移植成功

```
[    1.893876] devtmpfs: mounted
[    1.897492] Freeing unused kernel memory: 1024K (c0a00000 - c0b00000)
[    1.925059] EXT2-fs (mmcblk1p2): warning: mounting unchecked fs, running e2fsck is recommended
                    #----------------------XJ tech-------------------------#
eth0 Not enable

Please press Enter to activate this console.
@(none):/$ ls
bin         lib         mnt         sbin        usr
dev         linuxrc     proc        sys         var
etc         lost+found  root        tmp
@(none):/$
```

Linux是单内核系统，可通用计算平台的外围设备是频繁变化的，不可能将所有的(包括将来即将出现的)设备的驱动程序都一次性编译进内核，为了解决这个问题，Linux提出了可加载内核模块(Loadable Kernel Module,LKM)的概念，允许一个设备驱动通过模块加载的方式，在内核运行起来之后"融入"内核，加载进内核的模块和本身就编译进内核的模块一模一样。

一个程序在编译的地址的相对关系就已经确定了，运行的时候只是进行简单的偏移，为了使模块加载进内核后能够被放置在正确的地址，并正确的调用系统的运行的导出符号表，编译模块的时候必须要使用系统的编译地址，并调用系统编译出得静态的导出符号表。即模块必须使用系统的配置环境:Makefile+.config，一旦这两个文件任意一个发生了变化，都很可能导致模块的编译地址与系统的编译地址不匹配，造成运行时的错误甚至宕机。

导出符号表

从提供系统运行效率的角度，一个模块不是也不应该是完全独立的，即一个模块往往会调用其他模块提供的功能来实现自己的功能，这样做能更好实现系统的分工并提高效率。Linux为了实现模块间的相互调用，设计了导出符号表，每个模块都可以将自己的一个私有的标号导出到系统层级，以使该标号对其他模块可见，系统在编译一个模块的时候会自动导出这个模块的导出符号表到modules.syms文件(如果没有导出任何符号，可以为空)，并在加载一个模块的时候会自动将该模块的导出符号表与系统自身的导出符号表合并。一个系统的源码的导出符号表一般在源码顶层目录的modules.syms文件中，查看正在运行的系统导出符号表使用cat /proc/kallsyms。注意，正如前面解释的，我们的模块之所以能够正常运行，一个重要原因就是编译我们模块使用的符号地址就是编译内核时使用的符号地址，所以运行起来虽然地址会有偏移，但是模块中相关的符号的地址也会和内核地址一起偏移，也就还能找得到。基于这种思想，我们也可以直接查看系统当前运行的地址，将地址赋值给一个函数指针并使用，也是没有问题的，当然，这只是阐述原理，并不建议这么写模块。

下面这个例子可以看出编译出的地址和运行时的地址是不一样的：

```
build $cat Module.symvers |grep  -w dev_printk --color
0xe5065023      dev_printk      vmlinux EXPORT_SYMBOL
build $sudo cat /proc/kallsyms |grep -w dev_printk --color
ffffffff815096f0 T dev_printk
```

导出符号表可以大大的提高系统的运行效率，这也是只有开源系统才能提供的一个强大的功能，但是，导出符号表的引入会导致一个小小的麻烦--模块的依赖，当我们使用lsmod的时候，就可以查看系统当前的模块，其最后两列分别是该模块被引用的次数以及引用该模块的内核模块，当一个模块被其他模块引用时，我们是不能进行卸载的，同样，如果模块A依赖于模块B，那么如果模块B不加载的时候模块A也加载不了。在编写多模块的时候尤其要注意这个问题，可以写一个脚本管理多个依赖模块。Linux内核使用两个宏来导出一个模块的符号

```
EXPORT_SYMBOL(符号名)
EXPORT_SYMBOL_GPL(符号名)
```

模块编译方法

借助内核的Makefile，编译出的XXX.ko（Kernel Object）就是可加载到该内核的外部模块，为了利用内核的Makefile，我们可以将编译外部模块的Makefile写成如下的格式：

```
ifneq ($(KERNELRELEASE),)
    export-objs = demo.o
    obj-m = extern.o
else
    KERNELDIR :=  /lib/modules/$(shell uname -r)/build
    PWD       := $(shell pwd)
```

```
all:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

clean:
    $(RM) .tmp_versions Module.symvers modules.order .tmp_versions .*.cmd *.o *.ko *.mod.c
endif
```

这个简单的Makfile是利用ubuntu主机的源码Makefile来编译模块，学习模块编程的开始阶段在主机进行编译调试更方便一点，下面我解释一下这个Makefile，首先，我们的思路还是通过内核的Makefile来准备我们的模块，而内核的Makefile一旦执行，就会给KERNELRELEASE这个变量赋值，所以第一次进入我们这个Makfile的时候，这个变量还是空，所以执行else的部分——给相关的变量赋值，make默认编译第一个目标all，make -C $(KERNELRELEASE)就是进入到KERNELRELEASE指定的目录并执行里面的Makefile，显然，这就是我们内核源码的顶层Makefile，接下来的选项M=$(PWD) modules都是传入这个顶层Makefile的参数，表示我要编译一个模块，这个模块位于M指定的目录，所以内核会进行相关的配置并最终进入到"这个模块所在的目录"，此时，我们的这个Makefile会再被进入一次，这一次是从内核Makefile中跳入这里的，，KERNELRELEASE已经被定义过，内核Makefile想要的就是obj-m后面指定的要编译的目标文件，所以内核Makfile就会找到我们写的模块源文件进行编译。如此我们就得到了能在ubuntu下执行的xxx.ko文件，如果需要在开发板上运行，只需要将内核路径改成开发板运行系统的源码路径即可，同时记得要导出相关的环境变量(ARCH, CROSS_COMPILE)

注册/注销模块

Linux为每个模块都预留了相应的地址，注册模块即让该模块对内核可见，这也是模块工作的先决条件。注册之后，我们就可以通过查看内核输出信息dmesg命令来查看模块的运行情况。经常使用内核函数printk()来输出系统信息进行打印调试。使用insmod XXX.ko加载一个模块,使用rmmod XXX.ko卸载一个模块，使用lsmod查看当前系统中的模块及其引用情况
insmod使用的是init_module()系统调用，这个系统调用的实现是sys_init_module()
rmmod使用delete_module()系统调用，这个系统调用的实现是sys_delete_module()

模块的程序框架

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/errno.h>

/* 构造/析构函数 */
static int __init mydemo_init(void)
{
    //构造设备/驱动对象
    //初始化设备/驱动对象
    //注册设备/驱动对象
    //必要的硬件初始化
}
static void __exit mydemo_exit(void)
{
    //回收资源
    //注销设备/驱动对象
}

/* 加载/卸载模块 */
module_init(mydemo_init);
module_exit(mydemo_exit);

/* 授权 */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("XJ");
MODULE_DESCRIPTIPON("mymydemo");

/* 导出符号 */
EXPORT_SYMBOL(data);
```

注意这里的授权是必须的，如果一个模块没有授权，那么很多需要该授权的函数甚至都不能使用，同理，不合适的授权也会导致模块运行或加载的错误，所以初学者一定不要忽视这个授权，相关授权的选项在"linux/module.h"中，这里我把相关的说明贴出来供大家参考

```
/*
 * The following license idents are currently accepted as indicating free
 * software modules
```

```
*
*  "GPL"               [GNU Public License v2 or later]
*  "GPL v2"            [GNU Public License v2]
*  "GPL and additional rights" [GNU Public License v2 rights and more]
*  "Dual BSD/GPL"          [GNU Public License v2
*                   or BSD license choice]
*  "Dual MIT/GPL"          [GNU Public License v2
*                   or MIT license choice]
*  "Dual MPL/GPL"          [GNU Public License v2
*                   or Mozilla license choice]
*
* The following other idents are available
*
*  "Proprietary"           [Non free products]
*/
```

另一个细节是Linux内核源码的默认头文件路径是顶层目录的include目录，所以包含头文件的时候include可以省略，

第一个Linux模块

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>

static int __init demo_init(void)
{
    printk(KERN_INFO"demo_init:%s,%s,%d"__FILE__,__func__,__LINE__);
    return 0;
}

static void __exit demo_exit(void)
{
    printk(KERN_INFO"demo_exit:%s,%s,%d"__FILE__,__func__,__LINE__);
}

module_init(demo_init);
module_exit(demo_exit);

MODULE_LICENSE("GPL");
```

执行insmod xjDemo.ko，查看执行结果

```
xjMod_1 $sudo insmod xjDemo.ko
xjMod_1 $lsmod |grep xjDemo
xjDemo                 16384  0
xjMod_1 $sudo rmmod xjDemo
xjMod_1 $dmesg|tail -n 5
[ 1681.801345] r8169 0000:02:00.0 eth0: link down
[ 1681.801354] r8169 0000:02:00.0 eth0: link down
[ 1683.427123] r8169 0000:02:00.0 eth0: link up
[ 1805.901066] demo_init:demo_init,(null),368047/home/jiang/Desktop/DriveLearn/ModMake/xjMod_1/xjDemo.c
[ 1828.298404] demo_exit:demo_exit,(null),5/home/jiang/Desktop/DriveLearn/ModMake/xjMod_1/xjDemo.c
```

模块传参

我们编写的模块还可以在insmod的时候传入参数，Linux提供了几个宏（函数）用于接收外部的参数。模块内部使用这些函数，只需执行insmod xjDemo.ko num=2，insmod mydemo.ko i=10，insmod mydemo.ko extstr="hello" 等命令就可以将参数传入模块

```
module_param(num,type,perm);     //接收一个传入的int数据
module_param(num,type,perm);     //接收一个传入的charp数据
module_param_array(num,type,nump,perm);                 //接收一个数组
module_param_string(name,string,len,perm);              //接收一个字符串
MODULE_PARAM_DESC("parameter description");
```