**linux子系统的初始化\_subsys\_initcall()：那些入口函数【转】**

     内核选项的解析完成之后，各个子系统的初始化即进入第二部分—入口函数的调用。通常USB、PCI这样的子系统都会有一个名为subsys\_initcall的入口，如果你选择它们作为研究内核的切入点，那么就请首先找到它。

朱德庸在《关于上班这件事》里说，要花前半生找入口，花后半生找出口。可见寻找入口对于咱们这一生，对于看内核代码这件事儿都是无比重要的。

但是很多时候，入口并不仅仅只有subsys\_initcall一个，比如PCI。

以下代码来自 linux内核源码中 include/linux/init.h 文件

1. 117 #define pure\_initcall(fn) \_\_define\_initcall("0",fn,1)
2. 118
3. 119 #define core\_initcall(fn) \_\_define\_initcall("1",fn,1)
4. 120 #define core\_initcall\_sync(fn) \_\_define\_initcall("1s",fn,1s)
5. 121 #define postcore\_initcall(fn) \_\_define\_initcall("2",fn,2)
6. 122 #define postcore\_initcall\_sync(fn) \_\_define\_initcall("2s",fn,2s)
7. 123 #define arch\_initcall(fn) \_\_define\_initcall("3",fn,3)
8. 124 #define arch\_initcall\_sync(fn) \_\_define\_initcall("3s",fn,3s)
9. 125 #define subsys\_initcall(fn) \_\_define\_initcall("4",fn,4)
10. 126 #define subsys\_initcall\_sync(fn) \_\_define\_initcall("4s",fn,4s)
11. 127 #define fs\_initcall(fn) \_\_define\_initcall("5",fn,5)
12. 128 #define fs\_initcall\_sync(fn) \_\_define\_initcall("5s",fn,5s)
13. 129 #define rootfs\_initcall(fn) \_\_define\_initcall("rootfs",fn,rootfs)
14. 130 #define device\_initcall(fn) \_\_define\_initcall("6",fn,6)
15. 131 #define device\_initcall\_sync(fn) \_\_define\_initcall("6s",fn,6s)
16. 132 #define late\_initcall(fn) \_\_define\_initcall("7",fn,7)
17. 133 #define late\_initcall\_sync(fn) \_\_define\_initcall("7s",fn,7s)
18. 134
19. 135 #define \_\_initcall(fn) device\_initcall(fn)

这些入口有个共同的特征，它们都是使用\_\_define\_initcall宏定义的。它们的调用也不是随便的，而是按照一定顺序的，这个顺序就取决于\_\_define\_initcall宏。\_\_define\_initcall宏用来将指定的函数指针放到.initcall.init节里。

**.initcall.init节**

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偏移量处。在vmlinux.lds文件里查找initcall.init就可以看到下面的内容

1. \_\_inicall\_start = .;
2. .initcall.init : AT(ADDR(.initcall.init) – 0xC0000000) {
3. \*(.initcall1.init)
4. \*(.initcall2.init)
5. \*(.initcall3.init)
6. \*(.initcall4.init)
7. \*(.initcall5.init)
8. \*(.initcall6.init)
9. \*(.initcall7.init)
10. }
11. \_\_initcall\_end = .;

这就告诉我们.initcall.init节又分成了7个子节，而xxx\_initcall入口函数指针具体放在哪一个子节里边儿是由xxx\_initcall的定义中，\_\_define\_initcall宏的参数决定的，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子节等等。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，等等。不同的入口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。

**注意：设备驱动程序中常见的module\_init(x)函数，查看init.h文件发现，**

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

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

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

**这样推断 module\_init 调用优先级为6低于subsys\_initcall调用优先级4.**

**do\_initcalls()函数**

那些入口函数的调用由do\_initcalls函数来完成。

do\_initcall函数通过for循环，由\_\_initcall\_start开始，直到\_\_initcall\_end结束，依次调用识别到的初始化函数。而位于\_\_initcall\_start和\_\_initcall\_end之间的区域组成了.initcall.init节，其中保存了由xxx\_initcall形式的宏标记的函数地址，do\_initcall函数可以很轻松的取得函数地址并执行其指向的函数。

.initcall.init节所保存的函数地址有一定的优先级，越前面的函数优先级越高，也会比位于后面的函数先被调用。

由do\_initcalls函数调用的函数不应该改变其优先级状态和禁止中断。因此，每个函数执行后，do\_initcalls会检查该函数是否做了任何变化，如果有必要，它会校正优先级和中断状态。

另外，这些被执行的函数有可以完成一些需要异步执行的任务，flush\_scheduled\_work函数则用于确保do\_initcalls函数在返回前等待这些异步任务结束。

1. 666 static void \_\_init do\_initcalls(void)
2. 667 {
3. 668 initcall\_t \*call;
4. 669 int count = preempt\_count();
5. 670
6. 671 for (call = \_\_initcall\_start; call < \_\_initcall\_end; call++) {
7. 672 ktime\_t t0, t1, delta;
8. 673 char \*msg = NULL;
9. 674 char msgbuf[40];
10. 675 int result;
11. 676
12. 677 if (initcall\_debug) {
13. 678 printk("Calling initcall 0x%p", \*call);
14. 679 print\_fn\_descriptor\_symbol(": %s()",
15. 680 (unsigned long) \*call);
16. 681 printk("/n");
17. 682 t0 = ktime\_get();
18. 683 }
19. 684
20. 685 result = (\*call)();
21. 686
22. 687 if (initcall\_debug) {
23. 688 t1 = ktime\_get();
24. 689 delta = ktime\_sub(t1, t0);
25. 690
26. 691 printk("initcall 0x%p", \*call);
27. 692 print\_fn\_descriptor\_symbol(": %s()",
28. 693 (unsigned long) \*call);
29. 694 printk(" returned %d./n", result);
30. 695
31. 696 printk("initcall 0x%p ran for %Ld msecs: ",
32. 697 \*call, (unsigned long long)delta.tv64 >> 20);
33. 698 print\_fn\_descriptor\_symbol("%s()/n",
34. 699 (unsigned long) \*call);
35. 700 }
36. 701
37. 702 if (result && result != -ENODEV && initcall\_debug) {
38. 703 sprintf(msgbuf, "error code %d", result);
39. 704 msg = msgbuf;
40. 705 }
41. 706 if (preempt\_count() != count) {
42. 707 msg = "preemption imbalance";
43. 708 preempt\_count() = count;
44. 709 }
45. 710 if (irqs\_disabled()) {
46. 711 msg = "disabled interrupts";
47. 712 local\_irq\_enable();
48. 713 }
49. 714 if (msg) {
50. 715 printk(KERN\_WARNING "initcall at 0x%p", \*call);
51. 716 print\_fn\_descriptor\_symbol(": %s()",
52. 717 (unsigned long) \*call);
53. 718 printk(": returned with %s/n", msg);
54. 719 }
55. 720 }
56. 721
57. 722 /\* Make sure there is no pending stuff from the initcall sequence \*/
58. 723 flush\_scheduled\_work();
59. 724 }

**目前研究Linux驱动程序的启动流程，这篇文章对Linux子系统调用顺序进行了详细的讲解，同时也说明了设备驱动程序的调用顺序，很值得收藏。**

本文来自：<http://blog.csdn.net/yimiyangguang1314/article/details/7312209>