《Linux 内核修炼之道》

作者：华清远见

第 4 章 系统初始化

本章简介

当你想要运行程序时，最 Windows 的方式是在 GNOME 或者 KDE 等桌面环境中点击相应的图标，或你采用自己习惯的方式——打开终端，在 shell 中敲入程序的文件名并回车。这样应用程序就能够被加载并运行，但是所有这一切的前提是已经有其他的软件或程序将内核加载并运行，这样的软件或程序也被称为内核引导程序，比如 GRUB 和 LILO。而内核引导程序本身也需要被其他某个软件来加载和运行，这一个递归的过程到系统硬件时终止。

因此说系统的初始化是一个很复杂的过程，这也恰恰验证了“万事开头难”的俗语。本章的内容将陪伴你一起经历这个困难的过程。

4.1 引导过程

Linux 的引导过程包括很多阶段。首先，对 Linux 的正常使用要求，必须有其他某个软件来加载并运行内核，这样的软件被称为内核引导程序，通常是 GRUB 或 LILO。

更进一步，还需要有软件能够加载并运行内核引导程序。而这个软件本身又需要被其他软件来加载和运行，这样一个递归的引导过程最终必须在某个地方终止，这个地方就是系统硬件。

如图 4.1 所示为 X86 PC 的引导过程。

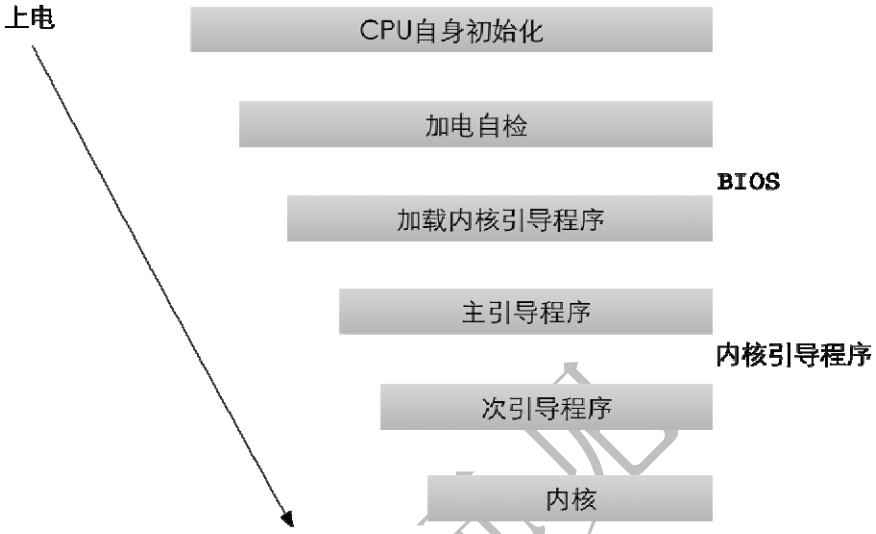

图 4.1　X86 PC 引导过程

（1）CPU 自身初始化。

CPU 自身的初始化是引导过程的第一步，如果有多个 CPU，即多处理器系统，则每个 CPU 都要进行自身初始化。

比如，对于双处理器的 Pentium 系统，一个 CPU 总是为主，另外一个 CPU 总是为辅，主 CPU 执行引导过程的剩余工作，随后内核才会激活辅 CPU。在辅 CPU 被激活之前，我们可以认为该系统中只有一个 CPU 可用，而不必考虑另外一个 CPU。

接下来，CPU 从某个固定位置（一般是 0Xffffffff0）取得指令并执行。该指令为跳转指令，跳转到 BIOS 代码的首部。注意，CPU 并不真正关心 BIOS 是否存在，它只是执行该地址中保存的任何指令。

（2）BIOS。

BIOS 被固化于主板上一个容量相对较小的只读存储器（Read-Only Memory，ROM）中，它的工作主要有两个：加电自检，即进行所谓的 POST（Power On Self Test）；加载内核引导程序。

POST 阶段完成系统硬件的检测，包括内存检测、系统总线检测等。BIOS 在 POST 阶段依据内置的规则，或者用户的手工选择确定启动设备。

POST 完成之后，BIOS 读取启动设备第一个扇区，即首 512 字节的信息，如图 4.2 所示，该扇区又被称之为主引导记录（Master Boot Record，MBR）。MBR 中保存了内核引导程序的开始部分，BIOS 将其加载到内存并执行。

加载内核引导程序之后，POST 部分的代码会被从内存中清理出来，但仍然会有部分的运行时服务保留在内存之中，供目标操作系统使用。

（3）内核引导程序。

内核引导程序分为两个阶段：MBR 中的主引导程序；活动分区引导记录中的次引导程序。

MBR 中的主引导程序是一个 512 字节的映像，如图 4.3 所示，它包含了 446 字节的程序代码和 64 字节的分区表，最后两个字节固定为 0xAA55，用于检查 MBR 是否有效。

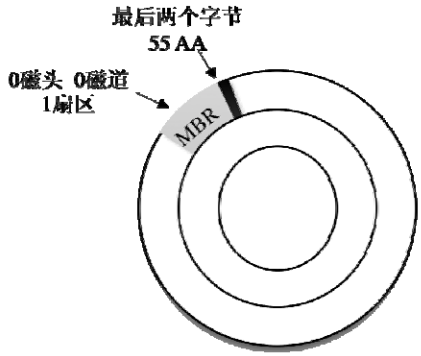

图 4.2　主引导记录 MBR

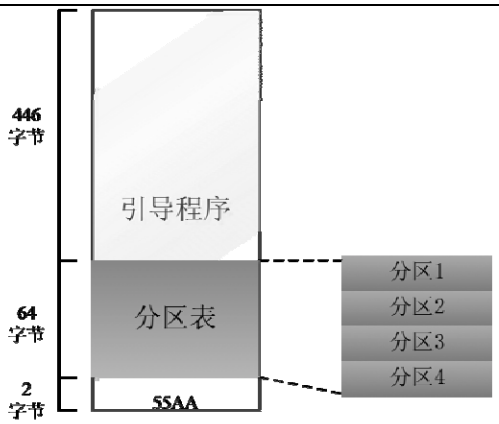

图 4.3　MBR 结构

主引导程序扫描分区表，寻找活动分区，将位于活动分区引导记录中的次引导程序加载到内存中并执行。

次引导程序负责加载 Linux 内核映像，并将控制权转交给内核。

在 PC 环境中，内核引导程序常用 LILO（Linux Loader）和 GRUB（Grand Unified Bootloader），在嵌入式环境中，常用 U-Boot 和 RedBoot。

（4）内核。

内核映像被加载到内存并获得控制权之后，内核阶段开始工作。通常，内核映像以压缩形式存储，并不是一个可执行的内核。因此，内核阶段的首要工作是自解压内核映像。

如图 4.4 所示，内核编译生成 vmlinux 后，通常会对其再进行压缩，成为 zImage（小内核，小于 512KB）或 bzImage（大内核，大于 512KB）。在 zImage 和 bzImage 的头部都内嵌有解压缩程序。

如图 4.5 所示，当用于 i386 映像的 bzImage 被调用时，将首先执行 arch/i386/boot/head.S 的 start 汇编例程，进行一些基本的硬件设置，并调用 arch/i386/boot/compressed/head.S 中的 startup_32 函数。

startup_32 例程设置一个基本的运行环境（堆栈等），并清除 BSS（Block Started by Symbol），然后调用 arch/i386/boot/compressed/misc.c 中的 C 函数 decompress_kernel 来解压内核。

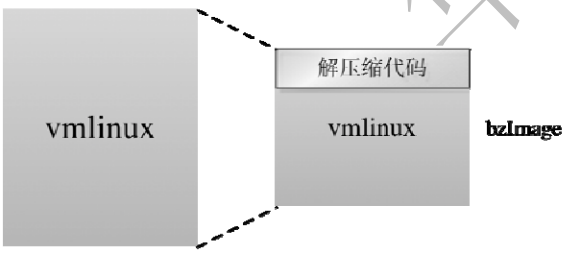

图 4.4　bzImage

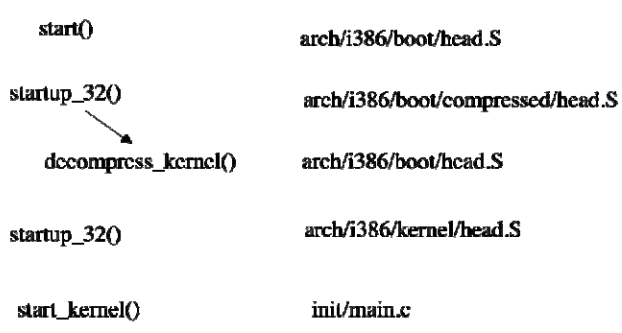

图 4.5　内核阶段的主要函数流程

decompress_kernel 函数打印出信息“Uncompressing Linux...”后，调用 gunzip 函数将内核解压到内存的指定位置。

之后，arch/i386/kernel/head.S 中的另外一个 startup_32 函数被调用。新的 startup_32 函数对页表进行初始化，启用内存分页功能，并为任何可选的浮点单元（FPU）检测 CPU 的类型，将其存储起来供以后使用。

最后，init/main.c 中的 start_kernel 函数被调用，进入体系结构无关的内核部分。自此，内核的引导过程告一段落，进入内核的初始化过程。

正如前面所述，引导过程可归纳为：CPU 加载 BIOS，BIOS 加载内核引导程序，内核引导程序加载压缩内核，压缩内核加载解压内核。其中的每一步都将我们带入更大量、更复杂的代码中，一直到成功地运行了内核。

4.2 内核初始化

如图 4.6 所示，内核的初始化过程由 start_kernel 函数开始，至第一个用户进程 init 结束，调用了一系列的初始化函数对所有的内核组件进行初始化。其中，start_kernel、rest_init、kernel_init、init_post 等 4 个函数构成了整个初始化过程的主线。

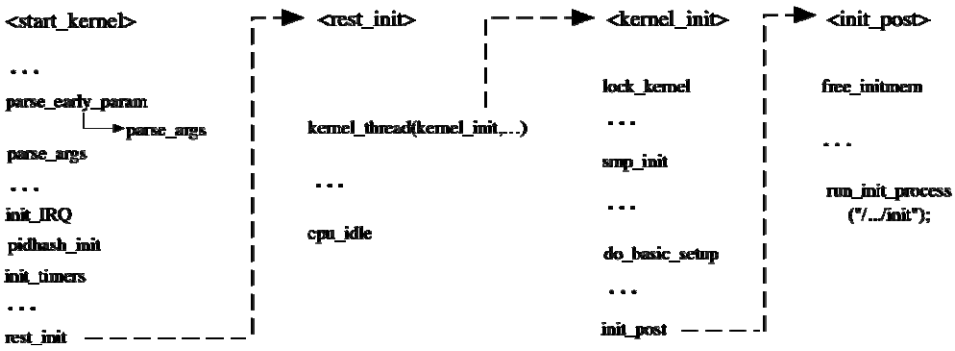

图 4.6　内核初始化

本节接下来的内容会结合内核代码，对内核初始化过程主线上的几个函数进行分析，使读者对该过程有个整体上的认识，以此为基础，读者可以根据自己的兴趣或需要，选择与某些组件相关的初始化函数，进行更进一步的研究分析。

4.2.1　start_kernel 函数

从 start_kernel 函数开始，内核即进入了 C 语言部分，它完成了内核的大部分初始化工作。实际上，可以将 start_kernel 函数看做内核的 main 函数。

代码清单 4.1　start_kernel 函数

```
513 asmlinkage void __init start_kernel(void)
514 {
515     char * command_line;
516     extern struct kernel_param __start___param[], __stop___param[];
517
        /*
         * 当只有一个 CPU 的时候这个函数就什么都不做，但是如果有多个 CPU 的时候那么它就
         * 返回在启动的时候的那个 CPU 的号
         */
518     smp_setup_processor_id();
519
520     /*
521      * Need to run as early as possible, to initialize the
522      * lockdep hash:
523      */
524     unwind_init();
525     lockdep_init();
526
        /* 关闭当前 CPU 的中断 */
527     local_irq_disable();
528     early_boot_irqs_off();
        /*
         * 每一个中断都有一个中断描述符（struct irq_desc）来进行描述，这个函数的
```

```
         * 作用就是设置所有中断描述符的锁
         */
529      early_init_irq_lock_class();
530
531 /*
532  * Interrupts are still disabled. Do necessary setups, then
533  * enable them
534  */
         /* 获取大内核锁，锁定整个内核。 */
535      lock_kernel();
         /* 如果定义了 CONFIG_GENERIC_CLOCKEVENTS，则注册 clockevents 框架 */
536      tick_init();
537      boot_cpu_init();
         /* 初始化页地址，使用链表将其链接起来 */
538      page_address_init();
539      printk(KERN_NOTICE);
         /* 显示内核的版本信息 */
540      printk(linux_banner);
         /*
          * 每种体系结构都有自己的 setup_arch()函数，是体系结构相关的，具体编译哪个
          * 体系结构的 setup_arch()函数,由源码树顶层目录下的 Makefile 中的 ARCH 变量
          * 决定
          */
541      setup_arch(&command_line);
542      setup_command_line(command_line);
543      unwind_setup();
         /* 每个 CPU 分配 pre-cpu 结构内存， 并复制.data.percpu 段的数据 */
544      setup_per_cpu_areas();
545      smp_prepare_boot_cpu();  /* arch-specific boot-cpu hooks */
546
547      /*
548       * Set up the scheduler prior starting any interrupts (such as the
549       * timer interrupt). Full topology setup happens at smp_init()
550       * time - but meanwhile we still have a functioning scheduler.
551       */
         /* 进程调度器初始化 */
552      sched_init();
553      /*
554       * Disable preemption - early bootup scheduling is extremely
555       * fragile until we cpu_idle() for the first time.
556       */
         /* 禁止内核抢占 */
557      preempt_disable();
558      build_all_zonelists();
559      page_alloc_init();
         /* 打印 Linux 启动命令行参数 */
560      printk(KERN_NOTICE "Kernel command line: %s\n", boot_command_line);
         /* 对内核选项的两次解析 */
561      parse_early_param();
562      parse_args("Booting kernel", static_command_line, __start___param,
563            __stop___param - __start___param,
564            &unknown_bootoption);
```

```
            /* 检查中断是否已经打开，如果已经打开，则关闭中断 */
    565     if (!irqs_disabled()) {
    566         printk(KERN_WARNING "start_kernel(): bug: interrupts were "
    567                 "enabled *very* early, fixing it\n");
    568         local_irq_disable();
    569     }
    570     sort_main_extable();
            /*
             * trap_init 函数完成对系统保留中断向量（异常、非屏蔽中断以及系统调用）                * 的初始化，init_IRQ 函数则完成其余中断向量的初始化
             */
    571     trap_init();
            /* 初始化 RCU(Read-Copy Update)机制 */
    572     rcu_init();
    573     init_IRQ();
            /* 初始化 hash 表，便于从进程的 PID 获得对应的进程描述符指针 */
    574     pidhash_init();
            /* 初始化定时器相关的数据结构 */
    575     init_timers();
            /* 对高精度时钟进行初始化 */
    576     hrtimers_init();
            /* 初始化 tasklet_softirq 和 hi_softirq */
    577     softirq_init();
    578     timekeeping_init();
            /* 初始化系统时钟源 */
    579     time_init();
            /* 对内核的 profile（一个内核性能调式工具）功能进行初始化 */
    580     profile_init();
    581     if (!irqs_disabled())
    582         printk("start_kernel(): bug: interrupts were enabled early\n");
    583     early_boot_irqs_on();
    584     local_irq_enable();
    585
    586     /*
    587      * HACK ALERT! This is early. We're enabling the console before
    588      * we've done PCI setups etc, and console_init() must be aware of
    589      * this. But we do want output early, in case something goes wrong.
    590      */
            /*
             * 初始化控制台以显示 printk 的内容，在此之前调用的 printk
             * 只是把数据存到缓冲区里
             */
    591     console_init();
    592     if (panic_later)
    593         panic(panic_later, panic_param);
    594
            /* 如果定义了 CONFIG_LOCKDEP 宏，则打印锁依赖信息，否则什么也不做 */
    595     lockdep_info();
    596
    597     /*
    598      * Need to run this when irqs are enabled, because it wants
    599      * to self-test [hard/soft]-irqs on/off lock inversion bugs
```

```
600      * too:
601      */
602     locking_selftest();
603
604 #ifdef CONFIG_BLK_DEV_INITRD
605     if (initrd_start && !initrd_below_start_ok &&
606             initrd_start < min_low_pfn << PAGE_SHIFT) {
607         printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "
608             "disabling it.\n",initrd_start,min_low_pfn << PAGE_SHIFT);
609         initrd_start = 0;
610     }
611 #endif
        /* 虚拟文件系统的初始化 */
612     vfs_caches_init_early();
613     cpuset_init_early();
614     mem_init();
        /* slab 初始化 */
615     kmem_cache_init();
616     setup_per_cpu_pageset();
617     numa_policy_init();
618     if (late_time_init)
619         late_time_init();
        /*
         * 一个非常有趣的 CPU 性能测试函数，可以计算出 CPU 在 1s 内执行了多少次一个
         * 极短的循环，计算出来的值经过处理后得到 BogoMIPS 值（Bogo 是 Bogus 的意思），
         */
620     calibrate_delay();
621     pidmap_init();
        /* 接下来的函数中，大多数都是为有关的管理机制建立专用的 slab 缓存 */
622     pgtable_cache_init();
        /* 初始化优先级树 index_bits_to_maxindex 数组 */
623     prio_tree_init();
624     anon_vma_init();
625 #ifdef CONFIG_X86
626     if (efi_enabled)
627         efi_enter_virtual_mode();
628 #endif
        /* 根据物理内存大小计算允许创建进程的数量 */
629     fork_init(num_physpages);
        /*
         * proc_caches_init(), buffer_init(), unnamed_dev_init(), key_init()
         *
         */
630     proc_caches_init();
631     buffer_init();
632     unnamed_dev_init();
633     key_init();
634     security_init();
635     vfs_caches_init(num_physpages);
636     radix_tree_init();
637     signals_init();
638     /* rootfs populating might need page-writeback */
```

```
639     page_writeback_init();
640 #ifdef CONFIG_PROC_FS
641     proc_root_init();
642 #endif
643     cpuset_init();
644     taskstats_init_early();
645     delayacct_init();
646
        /*
         * 测试该 CPU 的各种缺陷，记录检测到的缺陷，以便于内核的其他部分以后可以
         * 使用它们的工作。
         */
647     check_bugs();
648
649     acpi_early_init(); /* before LAPIC and SMP init */
650
651     /* Do the rest non-__init'ed, we're now alive */
        /* 创建 init 进程 */
652     rest_init();
653 }
```

4.2.2 reset_init 函数

在 start_kernel 函数的最后调用了 reset_init 函数进行后续的初始化。

代码清单 4.2 reset_init 函数

```
438 static void noinline __init_refok rest_init(void)
439     __releases(kernel_lock)
440 {
441     int pid;
442
        /* reset_init()函数最主要的历史使命就是启动内核线程 kernel_init */
443     kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
444     numa_default_policy();
        /* 启动内核线程 kthreadd，运行 kthread_create_list 全局链表中的 kthread */
445     pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
446     kthreadd_task = find_task_by_pid(pid);
447     unlock_kernel();
448
449     /*
450      * The boot idle thread must execute schedule()
451      * at least once to get things moving:
452      */
        /*
         * 增加 idle 进程的 need_resched 标志， 并且调用 schedule 释放 CPU,
         * 将其赋给更应该获取 CPU 的进程。
         */
453     init_idle_bootup_task(current);
454     preempt_enable_no_resched();
455     schedule();
456     preempt_disable();
457
```

```
458     /* Call into cpu_idle with preempt disabled */
        /*
         * 进入 idle 循环以消耗空闲的 CPU 时间片， 该函数从不返回。然而，当有实际工作
         * 要处理时，该函数就会被抢占。
         */
459     cpu_idle();
460 }
```

4.2.3 kernel_init 函数

kernel_init 函数将完成设备驱动程序的初始化，并调用 init_post 函数启动用户空间的 init 进程。

代码清单 4.3 kernel_init 函数

```
813 static int __init kernel_init(void * unused)
814 {
815     lock_kernel();
816     /*
817      * init can run on any cpu.
818      */
        /* 修改进程的 CPU 亲和力 */
819     set_cpus_allowed(current, CPU_MASK_ALL);
820     /*
821      * Tell the world that we're going to be the grim
822      * reaper of innocent orphaned children.
823      *
824      * We don't want people to have to make incorrect
825      * assumptions about where in the task array this
826      * can be found.
827      */
        /* 把当前进程设为接受其他孤儿进程的进程 */
828     init_pid_ns.child_reaper = current;
829
830     __set_special_pids(1, 1);
831     cad_pid = task_pid(current);
832
833     smp_prepare_cpus(max_cpus);
834
835     do_pre_smp_initcalls();
836
        /* 激活 SMP 系统中其他 CPU */
837     smp_init();
838     sched_init_smp();
839
840     cpuset_init_smp();
841
        /*
         * 此时与体系结构相关的部分已经初始化完成，现在开始调用 do_basic_setup 函数
         * 初始化设备，完成外设及其驱动程序（直接编译进内核的模块）的加载和初始化
         */
842     do_basic_setup();
843
844     /*
```

```
845      * check if there is an early userspace init.  If yes, let it do all
846      * the work
847      */
848
849     if (!ramdisk_execute_command)
850         ramdisk_execute_command = "/init";
851
852     if (sys_access((const char __user *) ramdisk_execute_command, 0) != 0) {
853         ramdisk_execute_command = NULL;
854         prepare_namespace();
855     }
856
857     /*
858      * Ok, we have completed the initial bootup, and
859      * we're essentially up and running. Get rid of the
860      * initmem segments and start the user-mode stuff.
861      */
862     init_post();
863     return 0;
864 }
```

4.2.4 init_post 函数

到 init_post 函数为止，内核的初始化已经进入尾声，第一个用户空间进程 init 将姗姗来迟。

代码清单 4.4 init_post 函数

```
774 static int noinline init_post(void)
775 {
776     free_initmem();
777     unlock_kernel();
778     mark_rodata_ro();
779     system_state = SYSTEM_RUNNING;
780     numa_default_policy();
781
782     if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
783         printk(KERN_WARNING "Warning: unable to open an initial console.\n");
784
785     (void) sys_dup(0);
786     (void) sys_dup(0);
787
788     if (ramdisk_execute_command) {
789         run_init_process(ramdisk_execute_command);
790         printk(KERN_WARNING "Failed to execute %s\n",
791                 ramdisk_execute_command);
792     }
793
794     /*
795      * We try each of these until one succeeds.
796      *
797      * The Bourne shell can be used instead of init if we are
798      * trying to recover a really broken machine.
799      */
```

```
800     if (execute_command) {
801         run_init_process(execute_command);
802         printk(KERN_WARNING "Failed to execute %s.  Attempting "
803                     "defaults...\n", execute_command);
804     }
805     run_init_process("/sbin/init");
806     run_init_process("/etc/init");
807     run_init_process("/bin/init");
808     run_init_process("/bin/sh");
809
810     panic("No init found.  Try passing init= option to kernel.");
811 }
```

第 776 行，到此，内核初始化已经接近尾声了，所有的初始化函数都已经被调用，因此 free_initmem 函数可以舍弃内存的__init_begin 至__init_end（包括.init.setup、.initcall.init 等节）之间的数据。

所有使用__init 标记过的函数和使用__initdata 标记过的数据，在 free_initmem 函数执行后，都不能使用，它们曾经获得的内存现在可以重新用于其他目的。

第 782 行，如果可能，打开控制台设备，这样 init 进程就拥有一个控制台，并可以从中读取输入信息，也可以向其中写入信息。

实际上 init 进程除了打印错误信息以外，并不使用控制台，但是如果调用的是 shell 或者其他需要交互的进程，而不是 init，那么就需要一个可以交互的输入源。如果成功执行 open，/dev/console 即成为 init 的标准输入源（文件描述符 0）。

第 785～786 行，调用 dup 打开/dev/console 文件描述符两次。这样，该控制台设备就也可以供标准输出和标准错误使用（文件描述符 1 和 2）。假设第 782 行的 open 成功执行（正常情况），init 进程现在就拥有 3 个文件描述符——标准输入、标准输出以及标准错误。

第 788～804 行，如果内核命令行中给出了到 init 进程的直接路径（或者别的可替代的程序），这里就试图执行 init。

因为当 kernel_execve 函数成功执行目标程序时并不返回，只有失败时，才能执行相关的表达式。接下来的几行会在几个地方查找 init，按照可能性由高到低的顺序依次是： /sbin/init，这是 init 标准的位置；/etc/init 和/bin/init，两个可能的位置。

第 805～807 行，这些是 init 可能出现的所有地方。如果在这 3 个地方都没有发现 init，也就无法找到它的同名者了，系统可能就此崩溃。因此，第 808 行会试图建立一个交互的 shell（/bin/sh）来代替，希望 root 用户可以修复这种错误并重新启动机器。

第 810 行，由于某些原因，init 甚至不能创建 shell。当前面的所有情况都失败时，调用 panic。这样内核就会试图同步磁盘，确保其状态一致。如果超过了内核选项中定义的时间，它也可能会重新启动机器。

4.3 init 进程

当内核被引导并进行初始化之后，内核启动了自己的第一个用户空间应用程序，即 init。这是调用的第一个使用标准 C 库编译的程序，其进程编号始终为 1。

init 负责触发其他必须的进程，以使系统进入整体可用的状态。init 的这些工作根据/etc/inittab 文件来完成，包括设置 getty 进程接受用户登录，设置键盘、字图，设置网络等。如果没有 init 触发这些进程，内核即使成功启动，也没有多大意义。

基于这种设计模式，init 进程是系统中所有进程的起源，init 进程产生 getty 进程，getty 进程产生 login 进程，login 进程又进而产生 shell 进程，然后我们使用 shell，就可以产生每一个需要执行的进程。

4.4 内核选项解析

各个子系统的初始化是内核整个初始化过程必然要完成的基本任务，这些任务按照固定的模式来处理，可以归纳为两个部分：内核选项的解析以及子系统初始化函数的调用。

本节讲解内核选项的注册及解析机制，下一节将会讲解各个子系统的初始化函数如何被调用。

4.4.1 内核选项

Linux 允许用户传递内核配置选项给内核，内核在初始化过程中调用 parse_args 函数对这些选项进行解析，并调用相应的处理函数。

parse_args 函数能够解析形如“变量名=值”的字符串，在模块加载时，它也会被调用来解析模块参数。

内核选项的使用格式同样为“变量名=值”，打开系统的 grub 文件，然后找到 kernel 行，比如：

```
kernel      /boot/vmlinuz-2.6.18 root=/dev/sda1 ro splash=silent vga=0x314 pci=noacpi
```

其中的“pci=noacpi”等都表示内核选项。

内核选项不同于模块参数，模块参数通常在模块加载时通过“变量名=值”的形式指定，而不是内核启动时。如果希望在内核启动时使用模块参数，则必须添加模块名作为前缀，使用“模块名.参数=值”的形式，比如，使用下面的命令在加载 usbcore 时指定模块参数 autosuspend 的值为 2。

```
$ modprobe usbcore autosuspend=2
```

若是在内核启动时指定，则必须使用下面的形式：

```
usbcore.autosuspend=2
```

从 Documentation/kernel-parameters.txt 文件里可以查询到某个子系统已经注册的内核选项，比如 PCI 子系统注册的内核选项为：

```
pci=option[,option...]   [PCI] various PCI subsystem options:
    off     [X86-32] don't probe for the PCI bus
    bios    [X86-32] force use of PCI BIOS, don't access
            the hardware directly. Use this if your machine
            has a non-standard PCI host bridge.
    nobios      [X86-32] disallow use of PCI BIOS, only direct
            hardware access methods are allowed. Use this
            if you experience crashes upon bootup and you
            suspect they are caused by the BIOS.
    conf1       [X86-32] Force use of PCI Configuration
            Mechanism 1.
    conf2       [X86-32] Force use of PCI Configuration
            Mechanism 2.
    nommconf[X86-32,X86_64] Disable use of MMCONFIG for PCI
            Configuration
    nomsi       [MSI] If the PCI_MSI kernel config parameter is
            enabled, this kernel boot option can be used to
            disable the use of MSI interrupts system-wide.
    nosort      [X86-32] Don't sort PCI devices according to
            order given by the PCI BIOS. This sorting is
            done to get a device order compatible with
            older kernels.
    biosirq     [X86-32] Use PCI BIOS calls to get the interrupt
            routing table. These calls are known to be buggy
            on several machines and they hang the machine
            when used, but on other computers it's the only
            way to get the interrupt routing table. Try
```

```
                this option if the kernel is unable to allocate
                IRQs or discover secondary PCI buses on your
                motherboard.
        rom     [X86-32] Assign address space to expansion ROMs.
                Use with caution as certain devices share
                address decoders between ROMs and other
                resources.
        irqmask=0xMMMM  [X86-32] Set a bit mask of IRQs allowed to be
                assigned automatically to PCI devices. You can
                make the kernel exclude IRQs of your ISA cards
                this way.
        pirqaddr=0xAAAAA [X86-32] Specify the physical address
                of the PIRQ table (normally generated
                by the BIOS) if it is outside the
                F0000h-100000h range.
        lastbus=N   [X86-32] Scan all buses thru bus #N. Can be
                useful if the kernel is unable to find your
                secondary buses and you want to tell it
                explicitly which ones they are.
        assign-busses   [X86-32] Always assign all PCI bus
                numbers ourselves, overriding
                whatever the firmware may have done.
        usepirqmask [X86-32] Honor the possible IRQ mask stored
                in the BIOS $PIR table. This is needed on
                some systems with broken BIOSes, notably
                some HP Pavilion N5400 and Omnibook XE3
                notebooks. This will have no effect if ACPI
                IRQ routing is enabled.
        noacpi      [X86-32] Do not use ACPI for IRQ routing
                or for PCI scanning.
        routeirq    Do IRQ routing for all PCI devices.
                This is normally done in pci_enable_device(),
                so this option is a temporary workaround
                for broken drivers that don't call it.
        firmware    [ARM] Do not re-enumerate the bus but instead
                just use the configuration from the
                bootloader. This is currently used on
                IXP2000 systems where the bus has to be
                configured a certain way for adjunct CPUs.
        noearly     [X86] Don't do any early type 1 scanning.
                This might help on some broken boards which
                machine check when some devices' config space
                is read. But various workarounds are disabled
                and some IOMMU drivers will not work.
        bfsort      Sort PCI devices into breadth-first order.
                This sorting is done to get a device
                order compatible with older (<= 2.4) kernels.
        nobfsort    Don't sort PCI devices into breadth-first order.
        cbiosize=nn[KMG]    The fixed amount of bus space which is
                reserved for the CardBus bridge's IO window.
                The default value is 256 bytes.
        cbmemsize=nn[KMG]   The fixed amount of bus space which is
```

```
                reserved for the CardBus bridge's memory
                window. The default value is 64 megabytes.
```

4.4.2 注册内核选项

我们不必理解 parse_args 函数的实现细节，但必须知道如何注册内核选项。模块参数使用 module_param 系列的宏注册，内核选项则使用__setup 宏来注册。

__setup 宏在 include/linux/init.h 文件中定义。

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

__setup 需要两个参数，其中 str 是内核选项的名字，fn 是该内核选项关联的处理函数。__setup 宏告诉内核，在启动时如果检测到内核选项 str，则执行函数 fn。str 除了包括内核选项名字之外，必须以“=”字符结束。

不同的内核选项可以关联相同的处理函数，比如内核选项 netdev 和 ether 都关联了 netdev_boot_setup 函数。

除了__setup 宏之外，还可以使用 early_param 宏注册内核选项。它们的使用方式相同，不同的是，early_param 宏注册的内核选项必须要在其他内核选项之前被处理。

如图 4.7 所示，__setup 宏和 early_param 宏都由__setup_param 宏实现。__setup_param 宏将__setup 宏和 early_param 宏注册的内核选项所关联的函数存放到.init.setup 节。

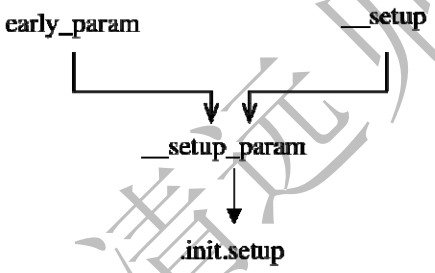

图 4.7 setup_param 宏及其封装

4.4.3 两次解析

相应于__setup 宏和 early_param 宏两种注册形式，内核在初始化时，调用了两次 parse_args 函数进行解析。

如图 4.8 所示，parse_args 函数第一次被调用是在 parse_early_param 函数中，用于处理 early_param 宏注册的高优先级的内核选项。parse_early_param 函数执行结束之后，parse_args 函数被第二次调用，处理其他的选项。

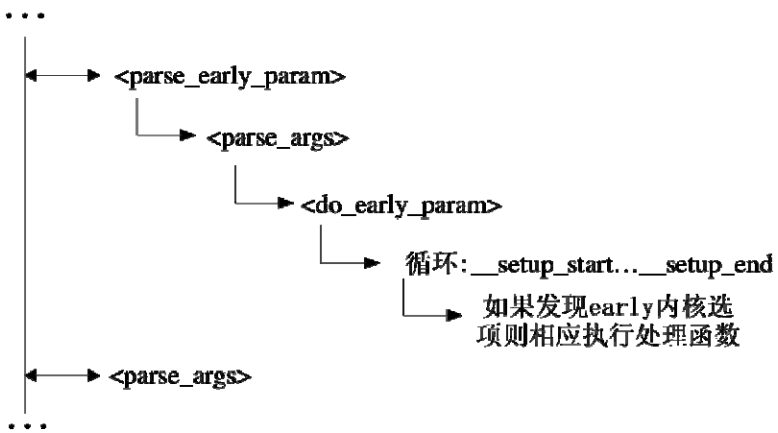

图 4.8 内核选项的两次解析

之前提到，__setup 宏和 early_param 宏注册的内核选项所关联的处理函数存放在.init.setup 节。打开内核连接器脚本文件 arch/i386/kernel/vmlinux.lds，查找.init.setup，然后便会看到：

```
__setup_start = .;
.init.setup : AT(ADDR(.init.setup) - 0xC0000000) { *(.init.setup) }
__setup_end = .;
```

其中的__setup_start 指向.init.setup 节的开始，__setup_end 指向.init.setup 节的结尾。解析时，会在__setup_start 和__setup_end 之间查找内核选项，当识别有内核选项时，即会调用相应的处理函数。

在内核启动之后，.init.setup 节会被释放，其中存放的内存选项不再需要，用户不能够在系统运行时查看或修改它们。

4.5 子系统的初始化

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——初始化函数的调用。这由 4.2 节提到的，内核初始化过程主线上的第三个函数 kernel_init，通过 do_basic_setup 函数再去调用 do_initcalls 函数来完成。

4.5.1 do_initcalls()函数

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 函数可以很轻松地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，这些被执行的函数又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。

代码清单 4.5 do_initcalls 函数

```
666 static void __init do_initcalls(void)
667 {
668     initcall_t *call;
669     int count = preempt_count();
670
671     for (call = __initcall_start; call < __initcall_end; call++) {
672         ktime_t t0, t1, delta;
673         char *msg = NULL;
674         char msgbuf[40];
675         int result;
676
677         if (initcall_debug) {
678             printk("Calling initcall 0x%p", *call);
679             print_fn_descriptor_symbol(": %s()",
680                     (unsigned long) *call);
681             printk("\n");
682             t0 = ktime_get();
683         }
684
```

```
685         result = (*call)();
686
687         if (initcall_debug) {
688             t1 = ktime_get();
689             delta = ktime_sub(t1, t0);
690
691             printk("initcall 0x%p", *call);
692             print_fn_descriptor_symbol(": %s()",
693                     (unsigned long) *call);
694             printk(" returned %d.\n", result);
695
696             printk("initcall 0x%p ran for %Ld msecs: ",
697                 *call, (unsigned long long)delta.tv64 >> 20);
698             print_fn_descriptor_symbol("%s()\n",
699                 (unsigned long) *call);
700         }
701
702         if (result && result != -ENODEV && initcall_debug) {
703             sprintf(msgbuf, "error code %d", result);
704             msg = msgbuf;
705         }
706         if (preempt_count() != count) {
707             msg = "preemption imbalance";
708             preempt_count() = count;
709         }
710         if (irqs_disabled()) {
711             msg = "disabled interrupts";
712             local_irq_enable();
713         }
714         if (msg) {
715             printk(KERN_WARNING "initcall at 0x%p", *call);
716             print_fn_descriptor_symbol(": %s()",
717                     (unsigned long) *call);
718             printk(": returned with %s\n", msg);
719         }
720     }
721
722     /* Make sure there is no pending stuff from the initcall sequence */
723     flush_scheduled_work();
724 }
```

4.5.2 .initcall.init 节

如图 4.8 所示描述了内核初始化代码的内存分布。内核使用了各式各样的宏来标识函数或结构所具有的初始化属性，这些宏定义位于 include/linux/init.h 文件，用于通知连接器将具有这些属性的函数或结构存放到专用的内存节。通过这种方式，我们能够很便利地访问某一类具有同样属性的对象。

图 4.9 的左边一列为每个内存节的开始和结束的指针名，右边一列则表示用于将函数或结构存放到相关内存节的宏。

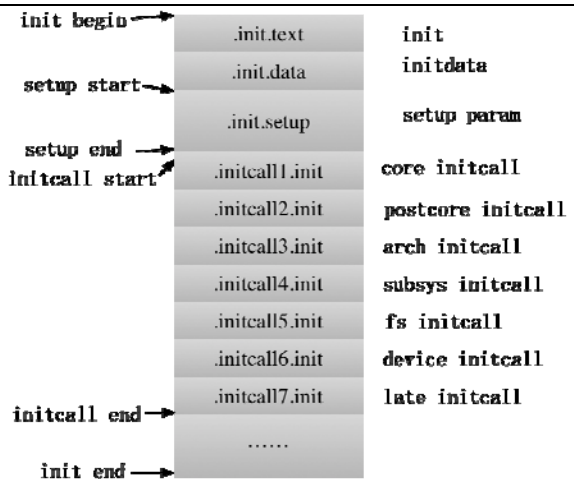

图 4.9　初始化代码的内存分布

__init 宏我们已不再陌生，它所修饰的函数所占用内存在初始化结束后便会释放。__initdata 用于启动时已经初始化的结构。__setup_param 宏正是我们上节所讲述的内容之一。

余下即是位于__initcall_start 和__initcall_end 之间.initcall.init 节。.initcall.init 节又分为 7 个子节，每个子节都对应一个形如 xxx_initcall 的宏，core_initcall 宏将函数指针放在.initcall1.init 子节，postcore_initcall 宏将函数指针放在了.initcall2.init 子节，依此类推。

各个子节的顺序是固定的，位于前面的子节具有更高的优先级，即 do_initcalls 函数执行时，会先调用.initcall1.init 中的函数指针，再调用.initcall2.init 中的函数指针。

内核使用各式各样的宏来标识函数或结构所具有的初始化属性，除了使我们能够方便地访问某一类具有同样属性的对象之外，同样还为了优化对内存的使用。

不同于用户空间的代码和数据，内核的代码和数据会一直保留在内存之中，因此在内核运行时尽可能减少对内存的浪费变得非常必要。而图 4.9 中.initcall.init 节的内容在初始化结束后，即使用 free_initmem 函数释放。xxx_initcall 宏和__init 宏等，一起协作完成了对内存的优化使用。

4.5.3　分析示例

这里以 PCI 子系统为例，分析一下它的初始化都使用了上述的哪些宏标记。

与很多子系统不同，PCI 子系统的实现代码分布在内核代码树的两个地方，除去 drivers/pci 存放了体系结构无关部分的代码之外，还有 arch/i386/pci 存放了体系结构相关部分的代码，因此我们需要在这两个地方分别查找它的初始化代码。

如表 4.1 所示为 PCI 子系统的初始化代码分布情况。

PCI 子系统的初始化几乎使用了本节所描述的所有 xxx_initcall 宏，它的初始化也严格按照表 4.1 所描述的内存存放顺序执行。

表 4.1　　PCI 子系统初始化代码分布情况

文　　件	初始化函数	宏　标　记	内 存 位 置
arch/i386/pci/acpi.c	pci_acpi_init	subsys_initcall	.initcall4.init
arch/i386/pci/common.c	pcibios_init	subsys_initcall	.initcall4.init
arch/i386/pci/i386.c	pcibios_assign_resources	fs_initcall	.initcall5.init
arch/i386/pci/legacy.c	pci_legacy_init	subsys_initcall	.initcall4.init
drivers/pci/pci-acpi.c	acpi_pci_init	arch_initcall	.initcall3.init
drivers/pci/pci- driver.c	pci_driver_init	postcore_initcall	.initcall2.init
drivers/pci/pci- sysfs.c	pci_sysfs_init	late_initcall	.initcall7.init
drivers/pci/pci.c	pci_init	device_initcall	.initcall6.init
drivers/pci/proc.c	pci_proc_init	__initcall	.initcall6.init

arch/i386/pci/init.c	pci_access_init	arch_initcall	.initcall3.init

但是，我们可以从表 4.1 中发现，PCI 子系统的一些初始化函数位于同一子节，前面只是讲述了不同子节之间的函数按照子节的优先级顺序执行，并没有讲述同一子节函数之间的调用顺序。

当然，我们可以指出一个事实，同一子节之间，地址位于最前面的函数会首先被调用。但是我们并不知道哪个函数位于前边、哪个函数位于后边，比如同样位于.initcall2.init 子节的 pcibus_class_init 函数和 pci_driver_init 函数，有没有一个简单的方法来进行判断？

下面是 GCC 手册中的一段话。

```
    the linker searches and processes libraries and object files in the order they are specified.
Thus, ‘foo.o -lz bar.o’ searches library ‘z’ after file ‘foo.o’ but before ‘bar.o’.
```

即是说，连接器按照库文件和目标文件被指定的顺序进行处理，打开 pcibus_class_init 函数和 pci_driver_init 函数所在目录 drivers/pci/下的 Makefile 文件，可以看到：

```
5 obj-y         += access.o bus.o probe.o remove.o pci.o quirks.o \
6                       pci-driver.o search.o pci-sysfs.o rom.o setup-res.o
```

probe.o 在 pci- driver.o 之前被指定，因此 probe.c 文件中的 pcibus_class_init 函数将在 pci- driver.c 文件中的 pci_driver_init 函数之前被调用。

对于 pcibus_class_init 函数和 pci_driver_init 函数这样位于同一目录位置的可以通过该目录 Makefile 文件指定的链接顺序来判断，而对于.initcall3.init 子节中的 acpi_pci_init 函数和 pci_access_init 函数则不能使用这个方法。

acpi_pci_init 函数位于 drivers/pci/pci-acpi.c 文件，而 pci_access_init 函数位于 arch/i386/pci/init.c 文件，它们位于不同的目录，此时问题即转化为 arch/i386/pci 下的 Makefile 和 drivers/pci 下的 Makefile 谁先谁后的问题，这就涉及 kbuild 构建内核的运行机制。

内核中的 Makefile 主要有如下 3 种。

内核源码树根目录里的 Makefile。虽说只有一个，但地位远远高于其他 Makefile，其中定义了所有与体系结构无关的变量和目标。

arch/*/Makefile。与特定体系结构相关，它会被根目录下的 Makefile 包含，为 kbuild 提供体系结构的特定信息。而它又包含了 arch/*/目录下面各级子目录下的那些 Makefile。

drivers/等各个子目录下的那些 Makefile。

kbuild 构建内核时，首先从根目录 Makefile 开始执行，从中获得与体系结构无关的变量和依赖关系，并同时从 arch/*/Makefile 中获得体系结构特定的变量等信息，用来扩展根目录 Makefile 所提供的变量。

此时，kbuild 已经拥有了构建内核需要的所有变量和目标。然后，kbuild 进入各个子目录，把部分变量传递给子目录里的 Makefile，子目录 Makefile 根据配置信息决定编译哪些源文件，从而构建出一个需要编译的文件列表。

之后，即是内核编译的漫长过程。现在，很明显，arch/i386/pci 下的 Makefile 是在 drivers/pci 下的 Makefile 之前被执行，即是说，pci_access_init 函数在 acpi_pci_init 函数之前被执行。

掌握这些规则，我们在研究某个子系统时，即可获得初始化函数的执行顺序，并按照该顺序进行深入的分析。

联系方式

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218