# 第2章 启动内核

系统上电后，处理器首先运行固件中的引导加载程序，引导加载程序将存储在外部介质中的内核可执行目标文件加载到内存中（内核镜像），将命令行参数和环境变量复制到内存并将地址传递给内核，然后，使CPU程序指针跳转至内核镜像入口地址取指，开始执行内核代码。内核代码首先完成一系列的系统初始化工作，创建内核线程，加载根文件系统，运行第一个用户进程，最后内核自身变换成系统中的空闲进程，内核启动完成。

## 2.1加载内核

内核源文件通过配置、编译、链接后，最后生成单一的可执行目标文件，目标文件是包含内核代码、数据段等的二进制文件。目标文件需要加载到物理内存中才能被处理器执行，因此在运行内核代码前需要有一段程序负责将内核目标文件加载至内存，并将内核代码入口地址赋予CPU程序指针。加载到物理内存的内核目标文件数据称之为内核镜像。

处理器上电后通常从固定的地址取指执行，这个地址是由处理器体系结构定义的。这段固定的地址不能映射到RAM内存，因为此时内存中还没有数据，通常这个固定地址映射到ROM或EPROM等非易性存储器，我们称此类存储器为固件，固件中存储的程序称为引导加载程序。

通常电脑系统中的引导加载程序为BIOS，而嵌入式系统中常见的是U-boot。引导加载程序可理解成一个小型的操作系统，它负责检测并初始化硬件，加载内核目标文件至内存，向内核传递命令行参数和环境变量等，最后将内核代码入口地址赋予处理器PC指针，处理器开始运行内核代码。

系统加载内核目标文件流程如下图所示：



内核目标文件中主要包含存储内核代码（指令）的代码段、已经初始化的数据段、初始化段和未初始化数据段等，这些段中的数据将会加载进物理内存。代码段的开头链接的是head.S文件中的代码，这是内核运行的起点，详见下一节。初始化段中保存的是只在内核启动阶段调用的函数和访问的数据，在内核代码中通常用“\_\_init\*”修饰，内核启动的末期将释放初始化段占用的内存（段中函数和数据不可用）。初始化段中主要链接的是内核初始化函数、设备驱动初始化函数、设备树目标文件、初始文件系统内容（基于物理内存的文件系统）等。

另外，引导加载程序还会将其中设置的命令行参数和环境变量复制到物理内存，并将地址保存到通用寄存器中传递给内核。

内核源代码链接文件为/arch/mips/kernel/vmlinux.lds.S，由文件内容我们可以看出内核目标文件的布局，链接文件内容简列如下：

ENTRY(kernel\_entry) /\*内核执行的入口地址\*/

...

SECTIONS

{

...

. = VMLINUX\_LOAD\_ADDRESS; /\*内核镜像起始地址（虚拟地址）\*/

**\_text** = .; /\*代码段起始地址\*/

.text : {

TEXT\_TEXT /\*宏定义在/include/asm-generic/vmlinux.lds.h头文件\*/

SCHED\_TEXT

LOCK\_TEXT

KPROBES\_TEXT

IRQENTRY\_TEXT

\*(.text.\*)

\*(.fixup)

\*(.gnu.warning)

} :text = 0

**\_etext** = .; /\*代码段结束地址\*/

EXCEPTION\_TABLE(16)

/\* Exception table for data bus errors \*/

\_\_dbe\_table : {

\_\_start\_\_\_dbe\_table = .;

\*(\_\_dbe\_table)

\_\_stop\_\_\_dbe\_table = .;

}

NOTES :text :note

.dummy : { \*(.dummy) } :text

**\_sdata** = .; /\*数据段开始地址\*/

RODATA /\*只读数据段\*/

/\*可写数据段\*/

.data : { /\* Data \*/

. = . + DATAOFFSET; /\* for CONFIG\_MAPPED\_KERNEL \*/

**INIT\_TASK\_DATA(THREAD\_SIZE)**

NOSAVE\_DATA

CACHELINE\_ALIGNED\_DATA(1 << CONFIG\_MIPS\_L1\_CACHE\_SHIFT)

READ\_MOSTLY\_DATA(1 << CONFIG\_MIPS\_L1\_CACHE\_SHIFT)

DATA\_DATA

CONSTRUCTORS

}

**\_gp** = . + 0x8000;

...

.sdata : {

\*(.sdata)

}

**\_edata** = .; /\*数据段结束地址\*/

/\*初始化段起始地址，内核启动后释放\*/

. = ALIGN(PAGE\_SIZE);

**\_\_init\_begin** = .;

INIT\_TEXT\_SECTION(PAGE\_SIZE)

INIT\_DATA\_SECTION(16)

. = ALIGN(4);

.mips.machines.init : AT(ADDR(.mips.machines.init) - LOAD\_OFFSET) {

\_\_mips\_machines\_start = .;

\*(.mips.machines.init)

\_\_mips\_machines\_end = .;

}

.exit.text : {

EXIT\_TEXT

}

.exit.data : {

EXIT\_DATA

}

#ifdef CONFIG\_SMP

PERCPU\_SECTION(1 << CONFIG\_MIPS\_L1\_CACHE\_SHIFT) /\*静态定义percpu变量段\*/

#endif

#ifdef CONFIG\_MIPS\_RAW\_APPENDED\_DTB /\*设备树目标文件段\*/

**\_\_appended\_dtb** = .; /\*设备树目标文件起始地址\*/

. += 0x100000;

#endif

. = ALIGN(0x10000);

**\_\_init\_end** = .; /\*初始化段结束地址\*/

BSS\_SECTION(0, 0x10000, 0) /\*未初始化数据段\*/

**\_end =** . ; /\*内核镜像结束地址\*/

...

}

链接文件中各大段的起始和结束地址标号导出到内核代码，内核代码中可操作和访问段内数据。各段内又划分成许多的小段，在以上链接文件中用宏表示，这些宏定义在/include/asm-generic/vmlinux.lds.h头文件，各小段的段名定义在/include/linux/init.h头文件。在阅读内核源代码时要特别注意显式指明了链接段的函数和变量。

## 2.2内核起点

根据内核链接文件可知，内核目标文件最开始链接的是体系结构实现的head.S文件，这是内核运行的起点。对于MIPS体系结构head.S文件路径为/arch/mips/kernel/head.S，文件内容简介如下。

我们先来看一下head.S文件内定义的几个宏：

.macro **setup\_c0\_status set clr** /\*设置CP0\_STATUS寄存器\*/

.set push

mfc0 t0, CP0\_STATUS /\*读取状态寄存器值\*/

or t0, ST0\_CU0|\set|0x1f|\clr /\*置相应位\*/

xor t0, 0x1f|\clr /\*清零低5位和clr标记的位\*/

mtc0 t0, CP0\_STATUS /\*写状态寄存器\*/

.set noreorder

sll zero,3

.set pop

.endm

setup\_c0\_status宏的作用是清零处理器状态寄存器低5位（内核态、关中断）和clr参数中标记的位，置位标记位ST0\_CU0（使能协处理器0）和set参数标记的位。

.macro **setup\_c0\_status\_pri**

#ifdef CONFIG\_64BIT

setup\_c0\_status ST0\_KX 0

#else

setup\_c0\_status 0 0 /\*设置状态寄存器，设置处理器内核态、关中断\*/

#endif

.endm

MIPS体系结构head.S文件中代码完成的主要工作如下：

1. 设置CP0协处理器状态寄存器值，使处理器处于内核态，关中断。
2. 查找链接到内核目标文件的设备树目标文件，如果存在则将其基址写入a1寄存器。
3. 清零内核未初始化数据段。
4. 将引导加载程序通过a0,a1,a2,a3寄存器传递的参数写入全局变量。
5. 设置内核栈。
6. 跳转至start\_kernel()函数入口地址，运行体系结构无关的内核启动函数。

下面列出head.S文件内主要的程序代码：

#ifndef CONFIG\_NO\_EXCEPT\_FILL /\*处理器配置选项，一般没有选择\*/

.fill 0x400 /\*预留空间用于异常向量\*/

#endif

EXPORT(\_stext)

...

NESTED(**kernel\_entry**, 16, sp) /\*定义kernel\_entry函数，内核入口地址\*/

**kernel\_entry\_setup** /\*默认实现为空，由平台（处理器）相关代码实现\*/

setup\_c0\_status\_pri /\*设置处理器状态寄存器，内核态，关中断\*/

PTR\_LA t0, 0f /\*加载地址至t0，PTR\_LA：la\*/

jr t0 /\*跳转到前面0标号处运行\*/

0:

#ifdef CONFIG\_MIPS\_RAW\_APPENDED\_DTB /\*原始内核目标文件中嵌入设备树目标文件\*/

PTR\_LA t0, **\_\_appended\_dtb**  /\*设备树文件起始地址，/arch/mips/kernel/vmlinux.lds.S\*/

#ifdef CONFIG\_CPU\_BIG\_ENDIAN

li t1, 0xd00dfeed

#else

li t1, 0xedfe0dd0 /\*设备树文件魔数\*/

#endif

lw t2, (t0) /\*设备树文件魔数，在文件开头处\*/

bne t1, t2, not\_found /\*魔数比对不成功表示不是设备树文件，跳转至not\_found\*/

nop

**move a1, t0**  /\*设备树文件查找成功，a1保存设备树文件起始地址\*/

**PTR\_LI a0, -2 /\*a0=-2，PTR\_LI=li\*/**

not\_found:

#endif /\*CONFIG\_MIPS\_RAW\_APPENDED\_DTB结束\*/

PTR\_LA t0, **\_\_bss\_start** /\*未初始化数据段起始地址，清零未初始化数据段\*/

LONG\_S zero, (t0) /\*LONG\_S：sw\*/

PTR\_LA t1, **\_\_bss\_stop** - LONGSIZE

/\*LONGSIZE定义在asm.h，整型的字节数\*/

1:

PTR\_ADDIU t0, LONGSIZE /\*PTR\_ADDIU：addiu\*/

LONG\_S zero, (t0)

bne t0, t1, 1b /\*未初始化数据段清零完成\*/

/\*

\*将引导加载程序传递的参数写入内核全局变量，

\*全局变量fw\_arg0，fw\_arg1,fw\_arg2,fw\_arg3定义在/arch/mips/kernel/setup.c文件内。

\*/

LONG\_S a0, fw\_arg0 /\*命令行参数数量，或为-2（传递了设备树目标文件）\*/

LONG\_S a1, fw\_arg1 /\*命令行参数字符串指针数组，或设备树目标文件地址\*/

LONG\_S a2, fw\_arg2 /\*环境变量指针数组基地址（不是用于用户进程的环境变量）\*/

LONG\_S a3, fw\_arg3 /\*\*/

MTC0 zero, CP0\_CONTEXT /\*清零处理器context寄存器\*/

PTR\_LA $28, **init\_thread\_union** /\***$28**保存内核自身thread\_union实例地址\*/

**PTR\_LI sp, \_THREAD\_SIZE - 32 - PT\_SIZE**

**PTR\_ADDU sp, $28** /\*内核栈顶地址，pt\_regs实例基址\*/

back\_to\_back\_c0\_hazard

**set\_saved\_sp sp, t0, t1**  /\*保存内核栈地址保入全局 kernelsp[cpu]数组项\*/

PTR\_SUBU sp, 4 \* SZREG /\*SZREG=4，以上代码用于设置内核栈sp值\*/

j **start\_kernel** /\*跳转到内核启动函数**start\_kernel()**\*/

END(kernel\_entry) /\*kernel\_entry函数结束\*/

内核中每一个进程/线程，包含内核自身，内核为其创建一个thread\_union联合体实例。联合体定义如下（/include/linux/sched.h（L2383））：

union thread\_union {

struct thread\_info thread\_info; /\*thread\_info结构体实例\*/

unsigned long stack[THREAD\_SIZE/sizeof(long)]; /\*内核栈，通常为8KB\*/

};

thread\_union联合体底部是一个thread\_info结构体实例，表示进程的底层信息，其中包括进程task\_struct结构体实例的指针，如下图所示。联合体大小由stack[}数组表示，通常为8KB，用于进程在内核态运行时的栈，也是内核线程运行栈。内核栈顶部是一个pt\_regs结构体实例，用于保存进程由用户空间进入内核空间运行时的上下文信息，详见5.2节。



内核自身thread\_union联合体实例**init\_thread\_union**定义在/init/init\_task.c文件内，在以上代码中，寄存器**$28**保存init\_thread\_union实例基地址。栈顶预留pt\_regs结构体再加上32字节空间，然后再往下预留16字节，作为内核栈顶地址赋予sp寄存器。

## 2.3启动函数

在体系结构实现的head.S文件中最后调用体系结构无关的初始化函数**start\_kernel()**启动内核。启动函数start\_kernel()内完成引导加载程序传递的命令行参数的处理，内核各系统和系统硬件的初始化工作，创建内核线程，挂载根文件系统，最后运行第一个用户进程，内核启动完成。

**start\_kernel()**函数在/init/main.c文件内实现，代码如下：

asmlinkage \_\_visible void \_\_init start\_kernel(void)

{

char \*command\_line; /\*命令行参数指针，setup\_arch()函数中使用\*/

char \*after\_dashes;

lockdep\_init();

set\_task\_stack\_end\_magic(&init\_task);

/\*在内核栈最低位置保存魔数STACK\_END\_MAGIC，/kernel/fork.c\*/

smp\_setup\_processor\_id(); /\*体系结构代码实现，默认为空操作，/init/main.c\*/

debug\_objects\_early\_init();

boot\_init\_stack\_canary();

cgroup\_init\_early();

local\_irq\_disable(); /\*关闭本地中断\*/

early\_boot\_irqs\_disabled = true;

**boot\_cpu\_init()**; /\*设置启动CPU核在CPU位图中标记位，/init/main.c\*/

**page\_address\_init()**; /\*初始化内核持久映射区管理数据结构，/mm/highmem.c\*/

pr\_notice("%s", linux\_banner);

**setup\_arch(&command\_line)**; /\*体系结构相关启动函数，/arch/mips/kernel/setup.c\*/

mm\_init\_cpumask(&init\_mm); /\*清mm->cpu\_vm\_mask\_var位图，/include/linux/mm\_types.h\*/

**setup\_command\_line(command\_line);** /\*复制命令行参数，/init/main.c\*/

**setup\_nr\_cpu\_ids()**; /\*设置nr\_cpu\_ids变量，表示SMP处理器实际的核数量，/kernel/smp.c\*/

**setup\_per\_cpu\_areas()**; /\*处理静态定义的percpu变量，/mm/percpu.c\*/

**smp\_prepare\_boot\_cpu()**; /\*设置CPU0在CPU位图中标记位，/arch/mips/kernel/smp.c\*/

**build\_all\_zonelists(NULL, NULL);**  /\*借用内存列表初始化，/mm/page\_alloc.c\*/

page\_alloc\_init();  /\*mm/page\_alloc.c\*/

pr\_notice("Kernel command line: %s\n", boot\_command\_line);

**parse\_early\_param()**; /\*处理由early\_param(str, fn)宏声明函数处理的命令行参数\*/

after\_dashes = **parse\_args**("Booting kernel", static\_command\_line, \_\_start\_\_\_param,

\_\_stop\_\_\_param - \_\_start\_\_\_param,-1, -1, NULL, &**unknown\_bootoption**);

/\*处理后期处理及内核未定义的命令行参数，\_\_setup(str, fn)\*/

if (!IS\_ERR\_OR\_NULL(after\_dashes))

**parse\_args**("Setting init args", after\_dashes, NULL, 0, -1, -1,NULL, **set\_init\_arg**);

/\*保存“--”参数之后的命令行参数\*/

jump\_label\_init();

setup\_log\_buf(0);

**pidhash\_init();** /\*upid实例全局散列表创建及初始化，/kernel/pid.c\*/

**vfs\_caches\_init\_early();** /\*虚拟文件系统早期初始化，/fs/dcache.c\*/

sort\_main\_extable();

**trap\_init();** /\*初始化异常（中断）向量（中断处理），见6.1节，/arch/mips/kernel/traps.c\*/

**mm\_init()**; /\*停用bootmem分配器，启用伙伴系统，初始化slab分配器等，/init/main.c\*/

**sched\_init()**; /\*进程调度器初始化，见5.7节，/kernel/sched/core.c\*/

preempt\_disable(); /\*禁止内核抢占\*/

if (WARN(!irqs\_disabled(),"Interrupts were enabled \*very\* early, fixing it\n"))

local\_irq\_disable(); /\*关闭本地CPU核中断\*/

**idr\_init\_cache(**); /\*创建idr数据结构idr\_layer缓存，/lib/idr.c\*/

**rcu\_init()**; /\*RCU机制数据结构初始化，见6.9.6小节，/kernel/rcu/tree.c\*/

trace\_init();

context\_tracking\_init();

**radix\_tree\_init()**; /\*基数树结构初始化，创建缓存，/lib/radix\_tree.c\*/

**early\_irq\_init()**; /\*初始化中断处理irq\_desc[]数组成员，/kernel/irq/irqdesc.c\*/

**init\_IRQ()**; /\*完成irq\_desc[]数组平台相关的初始化，/arch/mips/kernel/irq.c\*/

**tick\_init();**  /\*广播及动态时钟初始化，/kernel/time/tick-common.c**\*/**

**rcu\_init\_nohz(); /\*/**kernel/rcu/tree\_plugin.h\*/

**init\_timers();**  /\*低分辨率定时器初始化，见6.8节，/kernel/time/timer.c**\*/**

**hrtimers\_init();** /\*高分辨率定时器初始化，见6.8节，/kernel/time/hrtimer.c**\*/**

**softirq\_init();** /\*软中断机制初始化，见6.5节，/kernel/softirq.c\*/

**timekeeping\_init();** /\*记时器初始化，见6.8节，/kernel/time/timekeeping.c**\*/**

**time\_init();**  /\*注册时钟源、时钟事件设备，见6.8节，/arch/mips/kernel/time.c\*/

**sched\_clock\_postinit()**; /\*调度时钟初始化\*/

/\*需选择GENERIC\_SCHED\_CLOCK选项，/kernel/time/sched\_clock.c\*/

perf\_event\_init();

profile\_init(); /\*/kernel/profile.c\*/

call\_function\_init(); /\*向CPU通知链注册通知，/kernel/smp.c\*/

WARN(!irqs\_disabled(), "Interrupts were enabled early\n");

early\_boot\_irqs\_disabled = false;

local\_irq\_enable(); /\*开本地CPU核中断\*/

**kmem\_cache\_init\_late();**  /\*slab分配器初始化（后期），/mm/slab.c\*/

**console\_init();** /\*系统控制台初始化，/drivers/tty/tty\_io.c\*/

if (panic\_later)

panic("Too many boot %s vars at `%s'", panic\_later,panic\_param);

lockdep\_info();

locking\_selftest();

#ifdef CONFIG\_BLK\_DEV\_INITRD

if (initrd\_start && !initrd\_below\_start\_ok &&

page\_to\_pfn(virt\_to\_page((void \*)initrd\_start)) < min\_low\_pfn)

{

pr\_crit("initrd overwritten (0x%08lx < 0x%08lx) - disabling it.\n",

page\_to\_pfn(virt\_to\_page((void \*)initrd\_start)), min\_low\_pfn);

initrd\_start = 0;

}

#endif

page\_ext\_init(); /\*需选择SPARSEMEM（稀疏内存）配置选项，否则为空操作\*/

debug\_objects\_mem\_init();

kmemleak\_init();

**setup\_per\_cpu\_pageset()**; /\*为各内存域创建per\_cpu\_pageset实例并初始化，/mm/page\_alloc.c\*/

numa\_policy\_init();

if (late\_time\_init)

late\_time\_init(); /\*空操作，/init/main.c\*/

**sched\_clock\_init();** /\*/kernel/sched/clock.c\*/

calibrate\_delay();

**pidmap\_init();** /\*初始PID命名空间初始化，分配位图，创建pid缓存等，/kernel/pid.c\*/

**anon\_vma\_init()**; /\*初始化匿名映射结构（虚拟内存），/mm/rmap.c\*/

acpi\_early\_init();

...

**thread\_info\_cache\_init();**

/\*thread\_union大于一页为空操作，否则为thread\_union创建slab缓存，/kernel/fork.c\*/

**cred\_init();** /\*创建cred结构体（用户信息）slab缓存，/kernel/cred.c\*/

**fork\_init();**  /\*创建task\_struct结构缓存等，/kernel/sched/fork.c\*/

**proc\_caches\_init()**; /\*创建信号、文件、内存域等结构slab缓存，/kernel/fork.c\*/

**buffer\_init();**  /\*块缓存初始化，为buffer\_head结构创建slab缓存，/fs/buffer.c\*/

key\_init();

security\_init();

dbg\_late\_init();

**vfs\_caches\_init();** /\*虚拟文件系统早期初始化，/fs/dcache.c\*/

**signals\_init()**; /\*sigqueue结构创建slub缓存（用于实时信号），/kernel/signal.c\*/

page\_writeback\_init(); /\*设置数据回写脏页限制值初始值，/mm/page-writeback.c\*/

**proc\_root\_init()**; /\*proc文件系统初始化，/fs/proc/root.c\*/

nsfs\_init();

cpuset\_init(); /\*/kernel/cpuset.c\*/

**cgroup\_init()**; /\*控制组初始化，/kernel/cgroup.c\*/

taskstats\_init\_early();

delayacct\_init();

check\_bugs();

acpi\_subsystem\_init();

sfi\_init\_late();

if (efi\_enabled(EFI\_RUNTIME\_SERVICES)) {

efi\_late\_init();

efi\_free\_boot\_services();

}

ftrace\_init();

**rest\_init()**; /\*完成剩余初始化工作，/init/main.c\*/

}

start\_kernel()函数内主要是调用各初始化函数完成内核的初始化工作，具体函数将在后面的章节中详细介绍。本章将主要介绍体系结构实现的setup\_arch()函数、内核对命令行参数的处理以及rest\_init()函数的实现。

rest\_init()函数将创建内核线程，并由kernel\_init内核线程完成剩余的内核初始化工作、挂载根文件系统、运行第一个用户进程（覆盖kernel\_init线程）。rest\_init()函数最后进入一个无限循环，成为系统中的空闲进程，也就是说内核路径自身变为空闲进程，内核启动完成。

## 2.4 setup\_arch()

**setup\_arch()**函数是由体系结构相关代码实现的函数，完成特定于体系结构的初始化工作。

MIPS体系结构setup\_arch()函数在/arch/mips/kernel/setup.c文件内实现，代码如下：

void \_\_init setup\_arch(char \*\*cmdline\_p)

/\***cmdline\_p**：start\_kernel()中局部变量cmdline\_p的指针\*/

{

**cpu\_probe()**; /\*探测CPU核信息，/arch/mips/kernel/cpu-probe.c\*/

**prom\_init()**;

/\*平台代码实现，主要完成命令行参数复制等，/arch/mips/loongson32/common/prom.c\*/

setup\_early\_fdc\_console();

#ifdef CONFIG\_EARLY\_PRINTK

setup\_early\_printk();

#endif

cpu\_report(); /\*主要用于CPU信息的输出，/arch/mips/kernel/cpu\_probe.c\*/

check\_bugs\_early(); /\*选择64BIT配置选项时才有定义，arch/mips/include/asm/bugs.h\*/

#if defined(CONFIG\_VT)

#if defined(CONFIG\_VGA\_CONSOLE)

conswitchp = &vga\_con;

#elif defined(CONFIG\_DUMMY\_CONSOLE)

conswitchp = &dummy\_con;

#endif

#endif

**arch\_mem\_init(cmdline\_p)**; /\*物理内存管理初始化，见3.3节，/arch/mips/kernel/setup.c\*/

**resource\_init();**  /\*资源管理初始化，见8.5节，/arch/mips/kernel/setup.c\*/

**plat\_smp\_setup();**  /\*多核处理器初始化，见5.8小节，/arch/mips/include/asm/smp-op.h\*/

prefill\_possible\_map(); /\*设置possible处理器核位图（nr\_cpu\_ids），/arch/mips/kernel/setup.c\*/

**cpu\_cache\_init()**; /\*CPU核缓存初始化，/arch/mips/mm/cache.c\*/

}

这里我们先介绍与平台（处理器）密切相关的cpu\_probe()、prom\_init()和cpu\_cache\_init()函数的实现，其它函数将在后续章节中详细介绍。

cpu\_probe()函数由体系结构代码实现，用于获取MIPS处理器的硬件信息，并保存至cpuinfo\_mips结构体实例中，供后续代码使用。

prom\_init()函数由板级（处理器）代码实现，主要用于获取引导加载程序（固件）传递的参数和初始化必要的硬件设备。

cpu\_cache\_init()函数由体系结构代码实现，主要完成处理器缓存（cache）的初始化以及全局的缓存操作接口函数指针的设置，内核代码中可直接调用这些接口函数对处理器缓存进行操作。

### 2.4.1探测处理器信息

MIPS体系结构代码中定义了cpuinfo\_mips结构体（/arch/mips/include/asm/cpu-info.h）用于保存处理器核的硬件信息，内核为处理器中每个CPU核创建了cpuinfo\_mips结构体实例。

struct cpuinfo\_mips {

unsigned long **asid\_cache**; /\*CPU核进程ASID缓存，进程切换时会用到\*/

unsigned long ases;

unsigned long long **options**; /\*标记CPU核信息，如中断类型、缓存类型等\*/

unsigned int udelay\_val;

unsigned int **processor\_id**; /\*PRID寄存器值，处理器核编号，/arch/mips/include/asm/cpu.h\*/

...

unsigned int **cputype**; /\*CPU类型，/arch/mips/include/asm/cpu.h\*/

int isa\_level;

int **tlbsize**; /\*TLB信息\*/

int **tlbsizevtlb**;

int **tlbsizeftlbsets**;

int **tlbsizeftlbways**;

struct **cache\_desc** icache; /\*指令缓存信息\*/

struct cache\_desc dcache; /\*数据缓存信息\*/

struct cache\_desc scache; /\*二级缓存信息\*/

struct cache\_desc tcache; /\*三级缓存信息\*/

int srsets; /\*影子寄存器组数\*/

int package; /\* physical package number \*/

int **core**; /\*CPU核数量\*/

...

} \_\_attribute\_\_((aligned(SMP\_CACHE\_BYTES)));

cpuinfo\_mips结构体中主要成员简介如下：

●**options：**每个比特位表示CPU核的某一特性，每位含义定义在/arch/mips/include/asm/cpu.h头文件内：

#define MIPS\_CPU\_TLB 0x00000001ull /\* 具有TLB \*/

#define MIPS\_CPU\_4KEX 0x00000002ull /\* "R4K"异常向量模式\*/

#define MIPS\_CPU\_3K\_CACHE 0x00000004ull /\* R3000类型缓存\*/

#define MIPS\_CPU\_4K\_CACHE 0x00000008ull /\* R4000类型缓存\*/

#define MIPS\_CPU\_TX39\_CACHE 0x00000010ull /\* TX3900-style caches，缓存特性\*/

#define MIPS\_CPU\_FPU 0x00000020ull /\* CPU具有FPU，浮点协处理器 \*/

#define MIPS\_CPU\_32FPR 0x00000040ull /\* 2位浮点寄存器\*/

#define MIPS\_CPU\_COUNTER 0x00000080ull /\*具有count/compare 寄存器\*/

#define MIPS\_CPU\_WATCH 0x00000100ull /\*具有watchpoint寄存器 \*/

#define MIPS\_CPU\_DIVEC 0x00000200ull /\* dedicated interrupt vector \*/

#define MIPS\_CPU\_VCE 0x00000400ull /\* virt. coherence conflict possible \*/

#define MIPS\_CPU\_CACHE\_CDEX\_P 0x00000800ull /\* Create\_Dirty\_Exclusive CACHE op \*/

#define MIPS\_CPU\_CACHE\_CDEX\_S 0x00001000ull /\* ... same for seconary cache ... \*/

#define MIPS\_CPU\_MCHECK 0x00002000ull /\* Machine check exception \*/

#define MIPS\_CPU\_EJTAG 0x00004000ull /\* EJTAG exception \*/

#define MIPS\_CPU\_NOFPUEX 0x00008000ull /\* no FPU exception \*/

#define MIPS\_CPU\_LLSC 0x00010000ull /\* CPU has ll/sc instructions \*/

#define MIPS\_CPU\_INCLUSIVE\_CACHES 0x00020000ull /\* P-cache subset enforced \*/

#define MIPS\_CPU\_PREFETCH 0x00040000ull /\* CPU has usable prefetch \*/

#define MIPS\_CPU\_VINT 0x00080000ull /\* CPU支持MIPSR2向量中断模式\*/

#define MIPS\_CPU\_VEIC 0x00100000ull /\* CPU 支持MIPSR2 EIC中断模式\*/

#define MIPS\_CPU\_ULRI 0x00200000ull /\* CPU has ULRI feature \*/

#define MIPS\_CPU\_PCI 0x00400000ull /\* CPU has Perf Ctr Int indicator \*/

#define MIPS\_CPU\_RIXI 0x00800000ull /\* CPU has TLB Read/eXec Inhibit \*/

#define MIPS\_CPU\_MICROMIPS 0x01000000ull /\* CPU has microMIPS capability \*/

#define MIPS\_CPU\_TLBINV 0x02000000ull /\* CPU supports TLBINV/F \*/

#define MIPS\_CPU\_SEGMENTS 0x04000000ull /\* CPU supports Segmentation Control registers \*/

#define MIPS\_CPU\_EVA 0x80000000ull /\* CPU supports Enhanced Virtual Addressing \*/

#define MIPS\_CPU\_HTW 0x100000000ull /\* CPU support Hardware Page Table Walker \*/

#define MIPS\_CPU\_RIXIEX 0x200000000ull /\* CPU具有读写阻碍异常处理程序 \*/

#define MIPS\_CPU\_MAAR 0x400000000ull /\* MAAR(I) registers are present \*/

#define MIPS\_CPU\_FRE 0x800000000ull /\* FRE & UFE bits implemented \*/

#define MIPS\_CPU\_RW\_LLB 0x1000000000ull /\* LLADDR/LLB writes are allowed \*/

#define MIPS\_CPU\_XPA 0x2000000000ull /\* CPU supports Extended Physical Addressing \*/

#define MIPS\_CPU\_CDMM 0x4000000000ull /\* CPU has Common Device Memory Map \*/

#define MIPS\_CPU\_BP\_GHIST 0x8000000000ull /\* R12K+ Branch Prediction Global History \*/

●**cache\_desc结构体成员：**表示处理器缓存信息，结构体定义在/arch/mips/include/asm/cpu-info.h头文件：

struct cache\_desc {

unsigned int waysize; /\*每路字节数 \*/

unsigned short sets; /\*行数\*/

unsigned char ways; /\*路数\*/

unsigned char linesz; /\*每行字节数\*/

unsigned char waybit; /\*Bits to select in a cache set \*/

unsigned char flags; /\*标记\*/

};

内核在/arch/mips/kernel/setup.c文件内定义了静态cpuinfo\_mips结构体数组，每个CPU核对应一个数组项：

**struct cpuinfo\_mips cpu\_data[NR\_CPUS] \_\_read\_mostly**;

获取当前运行CPU核cpuinfo\_mips实例（指针）的宏定义在/arch/mips/include/asm/cpu-info.h头文件：

**#define current\_cpu\_data cpu\_data[smp\_processor\_id()]** /\*返回cpuinfo\_mips实例地址\*/

参数NR\_CPUS定义在/include/linux/threads.h头文件，NR\_CPUS与配置参数CONFIG\_NR\_CPUS相同，表示内核支持的最大CPU内核数量，这并不是实际处理器具有的CPU核数量。通常32位系统NR\_CPUS值为32，64位系统为64。

**cpu\_probe()**函数定义在/arch/mips/kernel/cpu-probe.c文件内，函数内首先读取协处理器PRID寄存器值，以确定处理器产商和处理器版本编号，如下图所示，高8位表示产商编号，8-15位表示处理器版本编号，然后根据这两个编号调用相应的函数以填充cpuinfo\_mips实例。



对于龙芯1x处理器，产商编号为0，处理器版本编号为0x42（PRID\_IMP\_LOONGSON\_32），据此函数cpu\_probe()的调用关系如下：



下面我们来简单看一下decode\_configs()函数的定义（/arch/mips/kernel/cpu-probe.c）：

static void decode\_configs(struct cpuinfo\_mips \*c)

{

int ok;

/\* MIPS32和MIPS64处理器默认设置\*/

c->options = MIPS\_CPU\_4KEX | **MIPS\_CPU\_4K\_CACHE** | MIPS\_CPU\_COUNTER |

MIPS\_CPU\_DIVEC | MIPS\_CPU\_LLSC | MIPS\_CPU\_MCHECK;

c->scache.flags = MIPS\_CACHE\_NOT\_PRESENT;

set\_ftlb\_enable(c, !mips\_ftlb\_disabled);

ok = **decode\_config0(c)**; /\*读取config0寄存器，获取处理器信息\*/

BUG\_ON(!ok); /\* Arch spec violation! \*/

if (ok)

ok = **decode\_config1(c)**;

if (ok)

ok = **decode\_config2(c)**;

if (ok)

ok = **decode\_config3(c)**;

if (ok)

ok = **decode\_config4(c)**;

if (ok)

ok = **decode\_config5(c)**;

mips\_probe\_watch\_registers(c);

if (cpu\_has\_rixi) {

set\_c0\_pagegrain(PG\_IEC);

back\_to\_back\_c0\_hazard();

if (read\_c0\_pagegrain() & PG\_IEC)

c->options |= MIPS\_CPU\_RIXIEX;

}

...

}

decode\_configs()函数首先设置c->options成员的默认标记位，然后依次读取配置寄存器0至5的值，根据配置寄存器值设置cpuinfo\_mips实例，decode\_config\*()函数都定义在/arch/mips/kernel/cpu-probe.c文件内，函数比较简单，请读者自行阅读。

### 2.4.2获取固件信息

prom\_init()函数在板级（处理器）代码实现，主要用于获取固件（引导加载程序）传递的参数信息以及初始化必要的硬件。龙芯1B开发板源代码在/arch/mips/loongson32/common/prom.c文件内实现了该函数。

void \_\_init prom\_init(void)

{

void \_\_iomem \*uart\_base;

prom\_argc = fw\_arg0; /\*命令行参数数量\*/

prom\_argv = (char \*\*)fw\_arg1; /\*命令行参数指针数组基地址\*/

prom\_envp = (char \*\*)fw\_arg2; /\*环境变量指针数组基地址\*/

**prom\_init\_cmdline()**; /\*复制参数字符串到**arcs\_cmdline**[]，/arch/mips/loongson32/common/prom.c\*/

memsize = env\_or\_default("memsize", DEFAULT\_MEMSIZE);

/\*若环境变量中无memsize，则设置默认值60MB\*/

highmemsize = env\_or\_default("highmemsize", 0x0);

/\*若环境变量中无highmemsize，则设置默认值为0x0\*/

if (strstr(arcs\_cmdline, "console=ttyS3"))

uart\_base = ioremap\_nocache(LS1X\_UART3\_BASE, 0x0f);

else if (strstr(arcs\_cmdline, "console=ttyS2"))

uart\_base = ioremap\_nocache(LS1X\_UART2\_BASE, 0x0f);

else if (strstr(arcs\_cmdline, "console=ttyS1"))

uart\_base = ioremap\_nocache(LS1X\_UART1\_BASE, 0x0f);

else

uart\_base = ioremap\_nocache(LS1X\_UART0\_BASE, 0x0f);

setup\_8250\_early\_printk\_port((unsigned long)uart\_base, 0, 0);

}

函数内首先调用prom\_init\_cmdline()函数将引导加载程序传递的命令行参数复制到全局**arcs\_cmdline**[]字符数组（以NULL结束），然后从环境变量中读取参数分别设置**memsize**和**highmemsize**全局变量，这两个变量分别表示物理内存大小和高端内存大小（单位MB），最后设置串口设备IO寄存器基地址并输出信息。

命令行参数是形如**"root=/dev/mtdblock1 console=ttyS2,115200 noinitrd init=/linuxrc"**的“参数名称=值 参数名称=值”的字符串，用于控制内核行为或设置内核参数等。引导加载程序需要将字符串按空格划分成形如“参数名称=值”的单个参数，并将参数基址保存在数组中，将数组基址和命令行参数数量传递给内核，如下图所示。



prom\_init\_cmdline()函数将命令行参数复制到全局字符数组**arcs\_cmdline[]**中，其中参数与参数间用空格字符隔开，数组结尾是NULL字符（0x00）。

### 2.4.3缓存初始化

函数**cpu\_cache\_init()**用于初始化处理器缓存，函数定义在/arch/mips/mm/cache.c文件内，函数内根据处理器缓存类型的不同调用不同的初始化函数。

在前面介绍的decode\_configs()函数中，MIPS32和MIPS64兼容的处理器设置**MIPS\_CPU\_4K\_CACHE**标记位，表示缓存类型为4k\_cache，因此cpu\_cache\_init()函数调用关系如下：

void cpu\_cache\_init(void)

{

...

if (cpu\_has\_4k\_cache) {

extern void \_\_weak r4k\_cache\_init(void);

**r4k\_cache\_init()**; /\*/arch/mips/mm/c-r4k.c\*/

}

...

setup\_protection\_map(); /\*/arch/mips/mm/c-r4k.c\*/

}

r4k\_cache\_init()函数内初始化处理器指令、数据缓存，并且设置全局的缓存操作接口函数指针，内核代码中可通过这些接口函数对处理器核缓存进行操作，函数源代码请读者自行阅读。

## 2.5命令行参数及模块参数

命令行参数是引导加载程序传递给内核的系统参数，是形如“参数名称=值 参数名称=值”的字符串，例如：**"root=/dev/mtdblock1 console=ttyS2,115200 noinitrd init=/linuxrc"。**命令行参数用于控制内核的行为或设置内核参数。在前面介绍的prom\_init()函数中命令行参数将被复制到全局字符数组**arcs\_cmdline[]**，内核在启动阶段将会对命令行参数进行处理。

模块参数是在模块代码中设置的参数（可视为外部变量，对外可见），在内核启动阶段可对内置模块（永久编译进内核的模块）参数进行设置，在命令行参数中嵌入“**模块名.参数名称=值**”的字符可对内置模块参数进行赋值。在使用insmode或modprobe命令加载外部模块时，可设置模块参数值，例如：“insmode 模块名称 **参数名称=值**”。

内核对命令行参数和模块参数的处理采用了相同的函数，本节主要介绍命令行参数和模块参数的定义以及内核处理参数的机制。

### 2.5.1命令行参数

引导加载程序可向内核传递命令行参数，以控制内核的行为和参数，内核需要为各个命令行参数定义处理函数，在启动阶段内核需要扫描命令行参数并调用定义的处理函数对各命令行参数进行处理。

#### 1参数传递

在MIPS体系结构中，引导加载程序需要将命令行参数拆分成形如“参数名称=值”的分量，建立指针数组指向各分量，参数数量保存至a0寄存器，指针数组地址保存至a1寄存器。在前面介绍的板级（平台）实现的prom\_init()函数中会复制引导加载程序传递的命令行参数至**arcs\_cmdline**[]字符数组，各参数之间用空格字符隔开，数组结尾字符为NULL（0x00），如下图所示。



在体系结构相关的文件/arch/mips/kernel/setup.c内定义了字符数组用于保存命令行参数字符串：

static char \_\_initdata **command\_line**[COMMAND\_LINE\_SIZE]; /\*传递给start\_kernel()函数\*/

char \_\_initdata  **arcs\_cmdline**[COMMAND\_LINE\_SIZE]; /\*体系相关内部代码使用\*/

常数COMMAND\_LINE\_SIZE定义在/arch/mips/include/uapi/asm/setup.h头文件内：

#define COMMAND\_LINE\_SIZE 4096

在体系结构无关的/init/main.c文件内定义了下列变量用于保存和指示命令行参数：

char \_\_initdata  **boot\_command\_line**[COMMAND\_LINE\_SIZE]; /\*体系结构无关的字符数组\*/

char \*saved\_command\_line; /\*指向参数字符串指针\*/

static char \*static\_command\_line; /\*指向参数字符串指针\*/

在体系结构相关的arch\_mem\_init()函数（/arch/mips/kernel/setup.c）会将**arcs\_cmdline**[]数组内容复制到**command\_line**[]和**boot\_command\_line**[]字符数组，如下图所示。



启动函数中调用的setup\_command\_line()函数将会为saved\_command\_line和static\_command\_line字符数组指针分配内存空间，并复制命令行参数字符数组到分配的空间中。

内核在**parse\_early\_param()**和**parse\_args()**函数中将会扫描命令行参数并对各参数（含内置模块参数）调用内核定义的处理函数对其进行处理。在加载外置模块时也将调用**parse\_args()**函数对模块参数进行设置。

#### 2注册参数处理函数

在内核代码中需要实现命令行参数的处理函数，并调用**early\_param(str, fn)**或**\_\_setup(str, fn)**宏注册参数处理函数，str为指向参数名称字符串的指针，fn为处理函数指针。

early\_param(str, fn)和\_\_setup(str, fn)宏定义在/include/linux/init.h头文件内：

#define **early\_param**(str, fn) \

\_\_setup\_param(str, fn, fn, 1)

#define **\_\_setup**(str, fn) \

\_\_setup\_param(str, fn, fn, 0)

#define \_\_setup\_param(str, unique\_id, fn, **early**) \

static const char **\_\_setup\_str\_##unique\_id[]** \_\_initconst \_\_aligned(1) = **str**; \ /\*创建字符数组\*/

static struct **obs\_kernel\_param**  **\_\_setup\_##unique\_id** \ /\*创建obs\_kernel\_param实例\*/

\_\_used \_\_section(**.init.setup**) \ /\*链接到指定的**.init.setup**段\*/

\_\_attribute\_\_((aligned((sizeof(long))))) \

= { **\_\_setup\_str\_##unique\_id**, fn, early }

由以上定义可知early\_param(str, fn)和\_\_setup(str, fn)宏都是通过\_\_setup\_param(str, unique\_id, fn, early)宏创建命令行参数名称字符数组和**obs\_kernel\_param**结构体实例并初始化。

特别需要注意的是**obs\_kernel\_param**实例链接到内核目标文件**.init.setup**段中（嵌入到初始化段），也就是说所有的obs\_kernel\_param实例都链接在一起，组成一个数组。

在/include/asm-generic/vmlinux.lds.h头文件中定义了**.init.setup**段的起始、结束地址的标号：

#define INIT\_SETUP(initsetup\_align) \

. = ALIGN(initsetup\_align); \

VMLINUX\_SYMBOL(**\_\_setup\_start**) = .; \

\*(**.init.setup**) \

VMLINUX\_SYMBOL(**\_\_setup\_end**) = .;

下面看一下obs\_kernel\_param结构体的定义（/include/linux/init.h）：

struct obs\_kernel\_param {

const char \*str; /\*指向参数名称字符串\*/

int (\*setup\_func)(char \*); /\*处理函数指针\*/

int early; /\*是否在启动早期处理，1表示是，0表示启动后期处理\*/

};

\_\_setup\_param()宏中创建的obs\_kernel\_param实例str成员指向命令行参数字符串，setup\_func保存参数处理函数指针（入口地址），early值为0或1，1表示参数需要在内核启动初期处理，0表示在后期处理。

\_\_setup(str, fn)和early\_param(str, fn)宏创建的obs\_kernel\_param实例，如下图所示：



内核在启动阶段对传递进来的每个命令行参数扫描**.init.setup**段的obs\_kernel\_param实例，比对参数名称与实例str成员指向的字符串，如果相同则调用实例中的setup\_func指向的函数进行处理，并将参数值字符数组指针传递给参数处理函数作为函数参数。

内核代码中定义的参数处理函数形式如下，例如，在/arch/mips/kernel/setup.c内定义了“rd\_size”参数的处理函数为rd\_size\_early()：

static int \_\_init rd\_size\_early(char \*p) /\*p指向参数值字符串\*/

{

initrd\_end += memparse(p, &p);

return 0;

}

**early\_param("rd\_size", rd\_size\_early)**; /\*早期处理命令行参数\*/

在/init/main.c内定义了“init”参数的处理函数为init\_setup()：

static int \_\_init init\_setup(char \*str) /\*str指向参数值字符串\*/

{

unsigned int i;

**execute\_command = str**; /\*execute\_command指向参数值字符数组\*/

for (i = 1; i < MAX\_INIT\_ARGS; i++)

argv\_init[i] = NULL;

return 1;

}

**\_\_setup("init=", init\_setup)**; /\*后期处理命令行参数\*/

#### 3常用命令行参数

下面简要介绍几个常用的命令行参数及其用途：

●**root=\*\***\*：设置挂载外部根文件系统的设备文件名称，处理函数定义在/init/do\_mounts.c文件内，负责将参数值字符数组复制到saved\_root\_name[]全局字符数组。

●**rootwait**：设置加载根文件系统时是否等待，以便外部设备初始化完成，处理函数在/init/do\_mounts.c文件内，不需要参数值。

●**rw/ro**：设置挂载外部根文件系统的读写/只读属性，系统默认为只读挂载，不需要参数值，处理函数定义在/init/do\_mounts.c文件内。

●**rootfstype=\*\*\***：设置rootfs根文件系统的类型，处理函数定义在/init/do\_mounts.c文件内，参数值为文件系统类型名称。

●**rootdelay=\*\*\***：设置加载根文件系统前等待的时间，以便外部介质初始化完成。处理函数定义在文件/init/do\_mounts.c内，参数值为整数值。

●**noinitrd**：表示采用ramfs不采用ramdisk，不需要参数值，处理函数定义在/init/do\_mounts\_initrd.c文件内。

### 2.5.2模块参数

模块参数可理解成模块代码中导出至外部的变量，对于内置模块的模块参数，可将参数赋值字符数组嵌入到命令行参数中，形如“**模块名.参数名称=值**”，对于外置的模块参数，可以在insmode或modprobe命令行中添加对参数的赋值，形如“**参数名称=值**”。内核在启动阶段或加载模块时，会根据外部传入的参数赋值字符串，对模块参数赋值。

模块参数其实就是模块代码中的全局变量，只不过在加载模块时可从外部对其赋初值。模块参数的类型包括：byte，short，ushort，int，uint，long，ulong，charp(字符指针)，bool，invbool（反bool）等。内核在/include/linux/moduleparam.h头文件内定义了声明变量为模块参数的宏，例如，如下是声明book\_num为int类型模块参数的代码（其它声明模块参数的宏请读者自行阅读）：

static int book\_num=4000; /\*参数初值\*/

**module\_param**(book\_num,int,S\_IRUGO);

**module\_param()**宏定义如下：

#define **module\_param**(name, type, perm) \ /\*name：名称，type：类型，perm：读写权限\*/

**module\_param\_named**(name, name, type, perm)

#define **module\_param\_named**(name, **value**, type, perm) \

param\_check\_##type(name, &(value)); \ /\*参数类型检查\*/

**module\_param\_cb**(name, &**param\_ops\_##type**, **&value**, perm); \ /\*&value：变量指针（地址）\*/

\_\_MODULE\_PARM\_TYPE(name, #type) /\*模块信息\*/

#define **module\_param\_cb**(name, **ops**, **arg**, perm) \ **\_\_module\_param\_call**(MODULE\_PARAM\_PREFIX, name, ops, arg, perm, **-1, 0**)

/\*ops：kernel\_param\_ops实例指针，arg：变量指针\*/

#define **\_\_module\_param\_call**(prefix, name, ops, arg, perm, level, flags) \ /\*level=-1，flags=0\*/

static const char **\_\_param\_str\_##name[] = prefix #name**; \ /\*变量名称字符数组\*/

static struct **kernel\_param** \_\_moduleparam\_const **\_\_param\_##name** \ /\*kernel\_param实例\*/

\_\_used \_\_attribute\_\_ ((unused,**\_\_section\_\_ ("\_\_param")**,aligned(sizeof(void \*)))) \

= { **\_\_param\_str\_##name**, THIS\_MODULE, **ops**, \

VERIFY\_OCTAL\_PERMISSIONS(perm), level, flags, { **arg** } }

**module\_param()**宏最终的效果是创建kernel\_param结构体实例，并将实例链接到指定的**"\_\_param"**段内，其实是kernel\_param实例数组。

kernel\_param结构体定义如下（/include/linux/moduleparam.h）：

struct kernel\_param {

const char \***name**; /\*参数名称字符数组指针\*/

struct module \*mod; /\*模块指针\*/

const struct **kernel\_param\_ops \*ops**; /\*参数操作结构体\*/

const u16 perm; /\*读写权限\*/

s8 level; /\*等级\*/

u8 flags; /\*标记\*/

union {

**void \*arg**; /\*指向变量的指针\*/

const struct kparam\_string \*str;

const struct kparam\_array \*arr;

};

};

kernel\_param\_ops结构体定义如下，表示对模块参数的操作：

struct kernel\_param\_ops {

unsigned int flags; /\*标记\*/

int (\***set**)(const char \*val, const struct kernel\_param \*kp); /\*设置模块参数（变量赋值）\*/

int (\*get)(char \*buffer, const struct kernel\_param \*kp); /\*获取模块参数\*/

void (\*free)(void \*arg); /\*释放模块参数\*/

};

set()成员函数用于获取命令行参数（或加载模块命令行参数）中传递的模块参数字符数组，val指向参数值字符数组，set函数需要将字符数组转换成参数类型变量值并赋予模块内部的变量。

内核在/kernel/params.c文件内定义了各模块参数类型对应的kernel\_param\_ops实例，实例名称为param\_ops\_*type*，例如，charp类型对应实例为param\_ops\_charp。

**module\_param()**宏中定义的kernel\_param实例**ops**成员指向param\_ops\_*type*实例，**arg**为变量指针，**name**成员指向参数名称字符数组，对于内置模块，参数名称为：模块名称**.**变量名称，对于外置模块，参数名称与模块中的变量名称相同。**module\_param()**宏创建kernel\_param实例如下图所示：



kernel\_param实例链接到内核目标文件的只读数据段（/include/asm-generic/vmlinux.lds.h）：

#define **RO\_DATA\_SECTION**(align) \ /\*只读数据段\*/

. = ALIGN((align)); \

.rodata : AT(ADDR(.rodata) - LOAD\_OFFSET) {

...

\_\_param : AT(ADDR(\_\_param) - LOAD\_OFFSET) { \

VMLINUX\_SYMBOL(**\_\_start\_\_\_param**) = .; \

**\*(\_\_param)** \

VMLINUX\_SYMBOL(**\_\_stop\_\_\_param**) = .; \

} \

...

}

\_\_start\_\_\_param和\_\_stop\_\_\_param为kernel\_param实例数组的起止地址。

### 2.5.3参数处理

内核对命令行参数和模块参数采用了相同的处理函数，处理机制是扫描命令行/模块参数，根据名称扫描内核目标文件中的obs\_kernel\_param实例/kernel\_param实例数组，比对实例中指示的名称字符数组和参数名称字符数组，如果相同则调用实例的**setup\_func()**/**set()**函数处理参数或设置模块变量值。

#### 1早期处理命令行参数

内核在体系结构相关的arch\_mem\_init()函数或启动函数start\_kernel()中将调用**parse\_early\_param()**函数处理需要在早期处理的命令行参数（**early\_param(str, fn)**宏声明的参数处理函数），随后调用**parse\_args()**函数处理其它的命令行参数和内置模块参数。下面我们先来看一下parse\_early\_param()函数的实现。

parse\_early\_param(void)函数定义在/init/main.c文件内，代码如下：

void \_\_init parse\_early\_param(void)

{

static \_\_initdata int done = 0; /\*静态变量，保证此函数只被执行一次\*/

static \_\_initdata char tmp\_cmdline[COMMAND\_LINE\_SIZE]; /\*保存命令行参数的局部变量\*/

if (done)

return;

strlcpy(tmp\_cmdline, boot\_command\_line, COMMAND\_LINE\_SIZE); /\*复制字符串至局部变量\*/

**parse\_early\_options(tmp\_cmdline)**; /\*拆分字符数组并处理各个参数，/init/main.c\*/

done = 1; /\*标记函数已经执行过\*/

}

parse\_early\_param(void)函数内部将保证只被执行一次，函数内调用**parse\_early\_options()**函数负责处理命令行参数，函数定义如下：

void \_\_init parse\_early\_options(char \*cmdline)

{

**parse\_args**("early options", cmdline, NULL, 0, 0, 0, NULL,**do\_early\_param**); /\*/kernel/params.c\*/

}

parse\_args()函数是处理命令行参数，包括内核参数的公用函数，函数定义在/kernel/params.c文件内，代码如下，这里我们只关心对命令行参数的处理：

char \***parse\_args**(const char \*doing,char \***args**,const struct kernel\_param \***params**,unsigned num,

s16 min\_level,s16 max\_level,void \***arg**,

int (\***unknown**)(char \*param, char \*val,const char \*doing, void \*arg))

/\*

\*doing：字符串，只是用于显示信息，**args**：命令行参数字符数组指针，num：NULL，min\_level：0

\*max\_level：0，arg：NULL，**unknown**：处理单个命令行参数的函数指针，**do\_early\_param()。**

\*/

{

char \*param, \*val;

args = skip\_spaces(args); /\*跳过字符串开头的空格\*/

...

while (\*args) { /\*扫描命令行参数字符数组，直至结束（扫描到NULL）\*/

int ret;

int irq\_was\_disabled;

args = **next\_arg**(args, &param, &val); /\*/kernel/params.c\*/

/\*逐个取出参数，参数间由空格隔开，param指向参数名称字符数组，val指向值字符数组\*/

if (!val && strcmp(param, "--") == 0) /\*遇到"--"参数，返回"--"参数（没有值）之后参数指针\*/

return args;

irq\_was\_disabled = irqs\_disabled();

ret = **parse\_one**(param, val, doing, **params**, **num**,min\_level, max\_level, arg, **unknown**);

/\*处理单个参数，/kernel/params.c\*/

...

switch (ret) {

...

}

}

return NULL;

}

parse\_args()函数对**args**参数传递进来的命令行参数字符数组按空格进行拆分，分别对每个参数调用函数**parse\_one()**进行处理，参数params指向参数名称字符数组（‘=’字符改为NULL），val指向值字符数组（数组结尾空格改成NULL）。如果在命令行参数中扫描到了“--”参数（没有值），parse\_args()函数将返回，args指向“--”参数之后的参数。

**parse\_one()**函数处理早期命令行参数时（params参数为NULL，num参数为0），主要是调用unknown参数指向的函数处理参数，此处为指向**do\_early\_param()**函数。

**do\_early\_param()**函数定义在init/main.c文件内，用于处理单个参数，代码如下：

static int \_\_init do\_early\_param(char \*param, char \*val,const char \*unused, void \*arg)

{

const struct obs\_kernel\_param \*p;

/\*扫描obs\_kernel\_param实例段，找到相应参数的处理函数并调用执行\*/

for (p = \_\_setup\_start; p < \_\_setup\_end; p++) /\*扫描obs\_kernel\_param实例段\*/

{

if ((**p->early** && **parameq(param, p->str))** || /\*名称相同且early为1的实例\*/

(strcmp(param, "console") == 0 &&

strcmp(p->str, "earlycon") == 0))

{

if (**p->setup\_func(val)** != 0) /\*调用处理函数，val为参数值字符数组地址\*/

pr\_warn("Malformed early option '%s'\n", param);

}

}

return 0;

}

**do\_early\_param()**函数比较简单，函数内将扫描obs\_kernel\_param实例，比对实例str指向的字符数组与参数名称字符数组，如果相同且实例early成员值不为0，则调用obs\_kernel\_param实例中处理函数处理参数。

#### 2其它参数处理

下面讨论内核如何处理由**\_\_setup**(str, fn)宏声明处理函数的命令行参数和内置模块参数的处理。在介绍其它参数的处理之前，先来看一下内核在/init/main.c文件内定义的两个字符指针数组，它们用于向第一个用户进程（init进程）传递命令行参数和环境变量。

static const char \*argv\_init[MAX\_INIT\_ARGS+2] = { "init", NULL, }; /\*命令行参数\*/

const char \*envp\_init[MAX\_INIT\_ENVS+2] = { "HOME=/", "TERM=linux", NULL, }; /\*环境变量\*/

static const char \*panic\_later, \*panic\_param;

用户可以通过引导加载程序的命令行参数向内核第一个用户进程（init进程）添加命令行参数和环境变量，argv\_init[]和envp\_init[]字符指针数组设置了初始的命令行参数和环境变量。例如，引导加载程序中可写入字符串“HOME=\*\*\*”用于覆盖内核初始设置的init进程HOME环境变量，其余未定义处理函数的命令行参数或模块参数，且与envp\_init[]环境变量名称不匹配的命令行参数（参数值为NULL）将作为init进程的命令行参数，添加到argv\_init[]指针数组。

在启动函数start\_kernel()内，在调用完parse\_early\_param()函数后接着调用两次**parse\_args()**函数，完成对其它命令行参数和内置模块参数的处理，函数调用如下：

after\_dashes = parse\_args("Booting kernel", static\_command\_line, \_\_start\_\_\_param,

\_\_stop\_\_\_param - \_\_start\_\_\_param,-1, -1, NULL, &**unknown\_bootoption**);

/\*处理“--”参数之前的命令行参数（内置模块参数）\*/

if (!IS\_ERR\_OR\_NULL(after\_dashes))

parse\_args("Setting init args", after\_dashes, NULL, 0, -1, -1,NULL, **set\_init\_arg**);

/\*保存“--”参数之后的命令行参数\*/

parse\_args()函数在前面介绍过了，但是此处params和num\_params参数不为空，调用parse\_one()函数时实现有所不同，下面列出parse\_one()函数的实现：

static int parse\_one(char \*param,char \*val,const char \*doing,const struct kernel\_param \***params**,

unsigned **num\_params**,s16 min\_level,s16 max\_level,void \*arg,

int (\***handle\_unknown**)(char \*param, char \*val,const char \*doing, void \*arg))

/\*

\***params：**kernel\_param实例（数组）指针，**num\_params：**kernel\_param实例数量，

\***min\_level、max\_level**：-1。

\*/

{

unsigned int i;

int err;

for (i = 0; i < num\_params; i++) { /\*扫描kernel\_param实例数组，处理内置模块参数\*/

if (**parameq(param, params[i].name**)) { /\*比对模块参数名称，“模块名**.**参数名称”\*/

if (params[i].level < min\_level|| params[i].level > max\_level)

return 0;

if (!val &&!(params[i].ops->flags & KERNEL\_PARAM\_OPS\_FL\_NOARG))

return -EINVAL;

pr\_debug("handling %s with %p\n", param,params[i].ops->set);

kernel\_param\_lock(params[i].mod);

param\_check\_unsafe(&params[i]);

err = **params[i].ops->set(val, &params[i])**; /\*调用设置模块参数\*/

kernel\_param\_unlock(params[i].mod);

return err; /\*函数返回\*/

}

}

/\*处理**\_\_setup**(str, fn)宏声明处理函数的命令行参数，以及尚未处理的内置模块参数\*/

if (handle\_unknown) {

pr\_debug("doing %s: %s='%s'\n", doing, param, val);

return **handle\_unknown**(param, val, doing, arg);

}

pr\_debug("Unknown argument '%s'\n", param);

return -ENOENT;

}

函数start\_kernel()在第一次调用parse\_args()函数时将处理命令行参数中“--”参数之前的命令行参数和内置模块参数。内置模块参数的处理将扫描kernel\_param实例数组，比对参数名称字符数组，相同则调用set()函数设置模块变量值，命令行参数（以及没有处理的内置模块参数）则调用**unknown\_bootoption()**函数进行处理。

函数start\_kernel()在第二次调用parse\_args()函数时将处理命令行参数中“--”参数之后的命令行参数，处理函数为**set\_init\_arg()**，下面将介绍这两个命令行参数处理的函数。

##### ■处理命令行参数

**unknown\_bootoption()**函数用于处理由**\_\_setup**(str, fn)宏声明处理函数的命令行参数，函数定义如下：

static int \_\_init unknown\_bootoption(char \*param, char \*val,const char \*unused, void \*arg) /\*/init/main.c\*/

{

**repair\_env\_string**(param, val, unused, NULL); /\*参数名称设置‘=’字符\*/

/\*处理过时的参数，**\_\_setup**(str, fn)宏声明处理函数的命令行参数\*/

if (**obsolete\_checksetup**(param)) /\*处理命令行参数，/init/main.c\*/

return 0;

/\*没有使用的内置模块参数，返回0\*/

if (strchr(param, '.') && (!val || strchr(param, '.') < val))

return 0;

if (panic\_later)

return 0;

/\*在内核中没有定义处理函数的命令行参数用于设置init进程命令行参数或环境变量\*/

if (val) { /\*参数值不为空\*/

unsigned int i;

for (i = 0; envp\_init[i]; i++) {

if (i == MAX\_INIT\_ENVS) {

panic\_later = "env";

panic\_param = param;

}

if (!**strncmp(param, envp\_init[i], val - param)**) /\*如果参数名称等于环境变量名称\*/

break;

}

**envp\_init[i] = param**; /\*添加环境变量\*/

} else { /\*参数值为空，其余尚未处理的命令行参数默认为init进程的命令行参数\*/

unsigned int i;

for (i = 0; argv\_init[i]; i++) {

if (i == MAX\_INIT\_ARGS) {

panic\_later = "init";

panic\_param = param;

}

}

**argv\_init[i] = param**; /\*添加命令行参数\*/

}

return 0;

}

**unknown\_bootoption()**函数处理由\_\_setup(str, fn)宏声明处理函数的命令行参数时，调用参数名称匹配的obs\_kernel\_param实例中的**setup\_func()**函数处理参数，对于没有匹配kernel\_param实例的内置模块参数不进行处理。如果命令行参数值不为空，且内核没有定义处理函数也不是模块参数，将作为init进程的环境变量，如果命令行参数值为空，且内核没有定义处理函数也不是模块参数，将作为init进程的命令行参数。

##### ■处理“--”之后参数

start\_kernel()函数在第二次调用parse\_args()函数时将处理命令行参数中“--”参数之后的命令行参数（不处理内置模块参数），处理函数为**set\_init\_arg()**，函数定义在/init/main.c文件内。

static int \_\_init set\_init\_arg(char \*param, char \*val,const char \*unused, void \*arg)

{

unsigned int i;

if (panic\_later) /\*命令行参数或环境变量指针数组是否已满\*/

return 0;

repair\_env\_string(param, val, unused, NULL); /\*参数名称设置‘=’字符\*/

for (i = 0; argv\_init[i]; i++) {

if (i == MAX\_INIT\_ARGS) {

panic\_later = "init";

panic\_param = param;

return 0;

}

}

**argv\_init[i] = param**; /\*添加命令行参数\*/

return 0;

}

set\_init\_arg()函数将引导加载程序传递的命令行参数中“--”参数之后的参数设为init进程的命令行参数。内核在运行第一个用户进程时会将argv\_init[]和envp\_init[]指示的命令行参数和环境变量传递给进程。

## 2.6剩余初始化

内核启动函数start\_kernel()最后调用rest\_init()函数完成剩余的初始化工作。rest\_init()函数内将创建一批内核线程，其中kernel\_init线程继续完成内核初始化工作并最后转变成第一个用户进程。内核路径将继续往下执行rest\_init()函数，最后进入一个无限循环，变成系统中的空闲进程（实际为内核线程），在系统无事可做时运行该进程。

### 2.6.1内核路径

#### 1创建内核线程

rest\_init()函数执行流程简列如下图所示，其中实线表示函数调用，虚线表示并行执行：



rest\_init()函数内首先创建kernel\_init内核线程，kernel\_init线程并不会立即运行，而是进入睡眠等待，rest\_init()函数随后又通过创建kthreadd线程来创建一批内核线程，并唤醒睡眠等待的kernel\_init线程，rest\_init()函数最后执行函数cpu\_startup\_entry()，进入无限循环，内核路径转为空闲进程。

rest\_init()函数调用之后，系统中存在多个并行运行的内核线程，其中kernel\_init线程继续完成内核各子系统的初始化工作，挂载外部存储介质根文件系统，最后选择用户进程目标文件运行第一个用户进程。

rest\_init()函数在/init/main.c文件内实现，代码如下：

static noinline void \_\_init\_refok rest\_init(void)

{

int pid;

rcu\_scheduler\_starting();

/\*/include/linux/rcutiny.h，没有配置DEBUG\_LOCK\_ALLOC则为空操作\*/

smpboot\_thread\_init();

**kernel\_thread(kernel\_init, NULL, CLONE\_FS)**; /\*创建kernel\_init内核线程，见下一小节\*/

numa\_default\_policy();

**pid = kernel\_thread(kthreadd, NULL, CLONE\_FS | CLONE\_FILES)**;

/\*创建kthreadd内核线程，用于创建内核线程，见5.4节\*/

rcu\_read\_lock();

**kthreadd\_task** = find\_task\_by\_pid\_ns(pid, &init\_pid\_ns);

/\*由pid查找kthreadd线程task\_struct实例\*/

rcu\_read\_unlock();

complete(&**kthreadd\_done**); /\*唤醒kernel\_init线程\*/

init\_idle\_bootup\_task(current);

/\*设置内核自身（空闲进程）调度器类为idle\_sched\_class，/kernel/sched/core.c\*/

schedule\_preempt\_disabled();

**cpu\_startup\_entry(CPUHP\_ONLINE)**; /\*空闲进程，/kernel/sched/idle.c\*/

}

rest\_init()函数内先后调用kernel\_thread()函数创建kernel\_init和kthreadd内核线程，在创建完kthreadd线程后唤醒kernel\_init线程。kernel\_init线程执行的工作将在下一小节中介绍，kthreadd内核线程用于创建其它的内核线程，它是一个用来创建内核线程的内核线程，详见5.4节。

rest\_init()函数最后调用cpu\_startup\_entry()函数进入一个无限循环，内核路径最终转为系统的空闲进程，这是内核自身最终的归宿。

#### 2空闲进程

cpu\_startup\_entry()函数函数定义在/kernel/sched/idle.c文件内，代码如下：

void cpu\_startup\_entry(enum cpuhp\_state state)

{

...

arch\_cpu\_idle\_prepare(); /\*体系结构代码定义，否则为空操作，MIPS为空，/kernel/sched/idle.c\*/

**cpu\_idle\_loop()**; /\*空闲进程执行函数，/kernel/sched/idle.c\*/

}

空闲进程最终的执行函数为cpu\_idle\_loop()，函数代码如下：

static void cpu\_idle\_loop(void)

{

while (1) {

\_\_current\_set\_polling(); /\*设置进程TIF\_POLLING\_NRFLAG标记位\*/

**tick\_nohz\_idle\_enter()**; /\*激活动态时钟，/kernel/time/tick-sched.c\*/

while (!need\_resched()) { /\*是否需要重调度\*/

check\_pgt\_cache();

rmb();

if (cpu\_is\_offline(smp\_processor\_id())) {

rcu\_cpu\_notify(NULL, CPU\_DYING\_IDLE, (void \*)(long)smp\_processor\_id());

smp\_mb(); /\* all activity before dead. \*/

this\_cpu\_write(cpu\_dead\_idle, true);

arch\_cpu\_idle\_dead();

}

**local\_irq\_disable()**; /\*关中断\*/

arch\_cpu\_idle\_enter(); /\*空操作\*/

if (cpu\_idle\_force\_poll || tick\_check\_broadcast\_expired())

cpu\_idle\_poll();

else

**cpuidle\_idle\_call()**; /\*空闲进程主要的执行函数，/kernel/sched/idle.c\*/

arch\_cpu\_idle\_exit(); /\*空操作\*/

}

**preempt\_set\_need\_resched()**; /\*设置需要重调度\*/

**tick\_nohz\_idle\_exit()**; /\*退出动态时钟，/kernel/time/tick-sched.c\*/

\_\_current\_clr\_polling();

smp\_mb\_\_after\_atomic();

sched\_ttwu\_pending();

schedule\_preempt\_disabled();

} /\*无限循环\*/

}

空闲进程中将激活动态时钟（见6.8节），执行其它的一些内核工作，并检测是否需要重调度，如果需要则执行进程调度，使处理器运行其它的就绪进程。

### 2.6.2 kernel\_init线程

kernel\_init内核线程由rest\_init()函数创建，并在kthreadd内核线程创建后被唤醒。kernel\_init内核线程执行函数为kernel\_init()，函数定义在/init/main.c文件内。kernel\_init()函数执行流程简列如下图所示：



kernel\_init\_freeable()函数将继续初始化内核各子系统，为线程打开标准输入输出文件（将传递给第一个用户进程），并挂载外部存储介质作为内核根文件系统（如果需要）。

free\_initmem()函数将释放内核镜像中初始化段占用的内存，将其释放给伙伴系统，供系统分配使用。

kernel\_init线程最后根据命令行参数等选择第一个用户进程的目标文件，并加载目标文件至线程用户空间，线程转而运行目标文件中代码，kernel\_init线程转变在第一个用户进程，内核启动最终完成。

在介绍kernel\_init()函数前，先看两个定义在/init/main.c文件内的全局变量：

static char \*execute\_command;

static char \*ramdisk\_execute\_command;

execute\_command指向保存命令行参数"init=\*\*\*"值的字符串，ramdisk\_execute\_command指向保存命令行参数"rdinit=\*\*\*"值的字符串，这两个指针变量在处理命令行参数时设置。前者用于指定在外部介质根文件系统中查找第一个用户进程目标文件的路径，后者用于指示在内核初始挂载的rootfs文件系统中查找第一个用户进程目标文件的路径。

内核在文件系统初始化过程中将会挂载初始的rootfs文件系统作为根文件系统，这是一个基于RAM内存的文件系统（详见7.12节），文件系统内容由用户设置。kernel\_init线程首先在rootfs文件系统中查找"rdinit=\*\*\*"参数指示的目标文件或/init是否存在，存在则加载运行，作为第一个用户进程，不再挂载外部存储介质文件系统。

如果不能在rootfs文件系统中运行"rdinit=\*\*\*"参数指示的目标文件或/init目标文件，kernel\_init线程将挂载外部存储介质（由命令行参数“root=\*\*\*”指示）作为根文件系统，并查找由"init=\*\*\*"参数指示的目标文件，如果查找到则加载运行，作为第一个用户进程。如果没有查找到"init=\*\*\*"参数指示的目标文件，则查找内核默认的目标文件并加载运行，作为第一个用户进程。

下面来看一下kernel\_init()函数的实现，代码如下（/init/main.c）：

static int \_\_ref kernel\_init(void \*unused)

{

int ret;

**kernel\_init\_freeable();**  /\*初始化各子系统，挂载外部根文件系统等，/init/main.c\*/

async\_synchronize\_full(); /\*/kernel/async.c\*/

**free\_initmem()**; /\*释放初始化段占用内存，详见3.6.4小节，/arch/mips/mm/init.c\*/

mark\_rodata\_ro();

system\_state = SYSTEM\_RUNNING;

numa\_default\_policy();

flush\_delayed\_fput();

/\*如果rootfs文件系统中存在"rdinit=\*\*\*"指示的目标文件或/init目标文件，则加载运行\*/

if (ramdisk\_execute\_command) {

ret = **run\_init\_process**(ramdisk\_execute\_command); /\*运行新进程，/init/main.c\*/

if (!ret)

return 0;

pr\_err("Failed to execute %s (error %d)\n",ramdisk\_execute\_command, ret);

}

/\*如果ramdisk\_execute\_command为NULL，尝试运行"init=\*\*\*"指示的目标文件\*/

if (execute\_command)

{

ret = run\_init\_process(execute\_command); /\*运行新进程，目标文件位于外部根文件系统\*/

if (!ret)

return 0;

panic("Requested init %s failed (error %d).",execute\_command, ret);

}

/\*如果还是运行不成功，尝试运行默认的目标文件，以先到者为先\*/

if (!**try\_to\_run\_init\_process**("/sbin/init") ||

!try\_to\_run\_init\_process("/etc/init") ||

!try\_to\_run\_init\_process("/bin/init") ||

!try\_to\_run\_init\_process("/bin/sh"))

return 0;

panic("No working init found. Try passing init= option to kernel. "

"See Linux Documentation/init.txt for guidance.");

}

kernel\_init线程内调用**kernel\_init\_freeable()**函数完成内核各子系统的初始化、外部存储介质文件系统的挂载（如果需要），随后调用free\_initmem()函数释放内核镜像初始化段占用的内存，最后选择目标文件运行第一个进程。

需要注意的是如果run\_init\_process()或try\_to\_run\_init\_process()函数执行成功，将会用目标文件内容覆盖kernel\_init线程地址空间，kernel\_init线程原代码不再执行，也就是说这两个函数返回到目标文件代码中运行了，不再执行函数之后的代码。全局字符指针数组argv\_init[]和envp\_init[]指示的命令行参数和环境变量将传递给第一个用户进程。

#### 1继续初始化

kernel\_init线程中调用kernel\_init\_freeable()函数完成内核剩余的初始化工作，并按需挂载外部存储介质中文件系统作为内核根文件系统。函数定义在/init/main.c文件内，代码如下：

static noinline void \_\_init kernel\_init\_freeable(void)

{

**wait\_for\_completion(&kthreadd\_done)**; /\*等待rest\_init()函数创建kthreadd线程完成\*/

gfp\_allowed\_mask = \_\_GFP\_BITS\_MASK;

set\_mems\_allowed(node\_states[N\_MEMORY]); /\*可在所有内存结点中分配内存\*/

**set\_cpus\_allowed\_ptr(current, cpu\_all\_mask)**;

/\*设置当前进程与所有CPU核的亲和性，/kernel/sched/core.c\*/

cad\_pid = task\_pid(current);

**smp\_prepare\_cpus(setup\_max\_cpus)**; /\*完成多核处理器准备工作，/arch/mips/kernel/smp.c\*/

**do\_pre\_smp\_initcalls()**; /\*调用early\_initcall(fn)注册的初始化函数，见下文，/init/main.c\*/

lockup\_detector\_init();

**smp\_init()**; /\*多核处理器数据初始化，见5.8.2小节，/kernel/smp.c\*/

**sched\_init\_smp()**; /\*多核处理器调度初始化，见5.8.2小节，/kernel/sched/core.c\*/

**page\_alloc\_init\_late()**;

/\*没有选择DEFERRED\_STRUCT\_PAGE\_INIT为空操作，/include/linux/gfp.h\*/

**do\_basic\_setup();** /\*初始化各子系统，/init/main.c\*/

/\*打开位于rootfs文件系统中的/dev/console作为线程标准输入文件\*/

if (**sys\_open((const char \_\_user \*) "/dev/console", O\_RDWR, 0) < 0**)

pr\_err("Warning: unable to open an initial console.\n");

(void) sys\_dup(0); /\*标准输出文件，指向/dev/console\*/

(void) sys\_dup(0); /\*标准错误输出文件，指向/dev/console\*/

if (!ramdisk\_execute\_command) /\*如果没有传递“rdinit=\*\*\*”参数，则赋值"/init"\*/

ramdisk\_execute\_command = **"/init"**;

if (sys\_access((const char \_\_user \*) ramdisk\_execute\_command, 0) != 0)

{ /\*访问ramdisk\_execute\_command 指向文件不成功则挂载外部介质根文件系统\*/

**ramdisk\_execute\_command = NULL**;

**prepare\_namespace()**;

/\*挂载外部文件系统作为根文件系统，详见7.12小节，/init/do\_mounts.c\*/

}

integrity\_load\_keys();

**load\_default\_modules()**; /\*加载块设备驱动IO调度器模块，/init/main.c\*/

}

kernel\_init\_freeable()函数的主要工作是初始化内核各子系统，并判断是否需要挂载外部存储介质作为新的根文件系统，如果需要则调用**prepare\_namespace()**函数挂载，不需要则不挂载。

需要挂载外部根文件系统的条件是：在初始rootfs根文件系统中不能打开（或不存在）“rdinit=\*\*\*”参数指示的目标文件，也不能打开（或不存在）/init目标文件（没有传递“rdinit=\*\*\*”参数时）。

kernel\_init\_freeable()函数中还将打开初始rootfs文件系统中的**/dev/console**设备文件作为线程（第一个用户进程）的标准输入、标准输出、标准错误输出文件，随后就可以向用户空间（终端）输出信息了。

继续初始化内核各子系统的工作主要由**do\_basic\_setup()**函数完成，函数定义在/init/main.c文件内：

static void \_\_init do\_basic\_setup(void)

{

cpuset\_init\_smp(); /\*设置CPU核位图等，/kernel/cpuset.c\*/

usermodehelper\_init();

shmem\_init(); /\*初始化tmpfs文件系统，/mm/shmem.c\*/

**driver\_init()**; /\*驱动模型初始化，见8.2节\*/

**init\_irq\_proc()**; /\*将irq信息导出到proc文件系统/irq目录，/kernel/irq/proc.c\*/

do\_ctors();

usermodehelper\_enable();

**do\_initcalls()**; /\*调用各子系统、驱动程序注册的初始化函数，/init/main.c\*/

random\_int\_secret\_init();

}

这里我们主要关注一下**do\_initcalls()**函数完成各子系统、设备驱动程序等注册的初始化函数的机制。

内核各子系统、驱动程序等需要在内核启动阶段调用的初始化函数，通常用下列宏注册：

#define **early\_initcall**(fn) \_\_define\_initcall(fn, early) /\*/include/linux/init.h\*/

#define pure\_initcall(fn) \_\_define\_initcall(fn, 0)

#define core\_initcall(fn) \_\_define\_initcall(fn, 1)

#define core\_initcall\_sync(fn) \_\_define\_initcall(fn, 1s)

#define postcore\_initcall(fn) \_\_define\_initcall(fn, 2)

#define postcore\_initcall\_sync(fn) \_\_define\_initcall(fn, 2s)

#define arch\_initcall(fn) \_\_define\_initcall(fn, 3)

#define arch\_initcall\_sync(fn) \_\_define\_initcall(fn, 3s)

#define subsys\_initcall(fn) \_\_define\_initcall(fn, 4)

#define subsys\_initcall\_sync(fn) \_\_define\_initcall(fn, 4s)

#define fs\_initcall(fn) \_\_define\_initcall(fn, 5)

#define fs\_initcall\_sync(fn) \_\_define\_initcall(fn, 5s)

#define rootfs\_initcall(fn) \_\_define\_initcall(fn, rootfs)

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

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

#define late\_initcall(fn) \_\_define\_initcall(fn, 7)

#define late\_initcall\_sync(fn) \_\_define\_initcall(fn, 7s)

以上宏内fn参数表示初始化函数指针（函数入口地址），函数类型如下：

typedef int (\***initcall\_t**)(void); /\*函数不能带参数，/include/linux/init.h\*/

**\_\_define\_initcall**(fn, id)宏定义在/include/linux/init.h头文件：

#define \_\_define\_initcall(fn, id) \

static **initcall\_t** \_\_initcall\_##fn##id \_\_used \ /\*initcall\_t为函数指针类型\*/

\_\_attribute\_\_((\_\_section\_\_(**".initcall" #id ".init"**))) = **fn**; \

LTO\_REFERENCE\_INITCALL(\_\_initcall\_##fn##id)

**\_\_define\_initcall**(fn, id)宏的效果是创建**initcall\_t**类型的函数指针变量并赋值fn，fn为注册的初始化函数入口地址，并将变量链接到指定的内核目标文件**".initcall" #id ".init"**段内，参数id嵌入到段名中，表示段的等级。\_\_define\_initcall()宏定义的函数指针变量都链接到同一个段内，参数id又将此段划分成小段，id值表示小段在**".initcall\*.init"**段内的排序，值越小的段排在越前面。例如：pure\_initcall(fn)宏定义的变量排在core\_initcall(fn)宏定义的变量前面。

以下是段名在链接文件中的表示（/include/asm-generic/vmlinux.lds.h），链接到初始化段：

#define INIT\_CALLS \

VMLINUX\_SYMBOL(**\_\_initcall\_start**) = .; \

\*(**.initcallearly.init**) \

INIT\_CALLS\_LEVEL(0) \ /\*等级0\*/

INIT\_CALLS\_LEVEL(1) \

INIT\_CALLS\_LEVEL(2) \

INIT\_CALLS\_LEVEL(3) \

INIT\_CALLS\_LEVEL(4) \

INIT\_CALLS\_LEVEL(5) \

INIT\_CALLS\_LEVEL(rootfs) \

INIT\_CALLS\_LEVEL(6) \

INIT\_CALLS\_LEVEL(7) \

VMLINUX\_SYMBOL(**\_\_initcall\_end**) = .;

**do\_initcalls()**函数执行的工作如下图所示，从等级0开始扫描各段，依次调用执行**initcall\_t**变量指示的函数，完成内核代码中注册的初始化函数。

**early\_initcall**(fn)宏注册的初始化函数将在调用do\_basic\_setup()函数前由**do\_pre\_smp\_initcalls()**函数处理，处理方式与**do\_initcalls()**函数相同，只不过它只处理**.initcallearly.init**段中注册的函数。



#### 2运行第一个用户进程

前面大致介绍了kernel\_init线程在完成初始化工作后转为第一个用户进程的流程。这里有必要再梳理一下，以使读者对外部根文件系统的挂载和运行第一个用户进程有更清晰的认识，更详细的内容见第7章和第13章。

启动函数start\_kernel()中将调用vfs\_caches\_init()函数挂载内核初始的根文件系统，这是一个基于内存盘的ramfs或tmpfs文件系统，统称为rootfs，初始状态此文件系统内容为空。

在配置内核时，如果选择了**BLK\_DEV\_INITRD** 配置选项（且不选择BLK\_DEV\_RAM选项），编译内核时将会编译源文件**./us**r目录下的内容为**.cpio**格式的目标文件，并将链接到内核目标文件的初始化段。在初始化各子系统的do\_initcalls()函数中将会调用注册的初始化函数**populate\_rootfs()**将内核镜像初始化段中.cpio格式目标文件的内容释放到初始rootfs类型的内核根文件系统中，如下图所示。



在前面介绍的kernel\_init\_freeable()函数中将会判断在初始根文件系统中能否打开"**rdinit=\*\*\***"参数值指示的目标文件，或“**/init**”目标文件（没有传递"rdinit=\*\*\*"参数时），如果可以打开将不挂载外部介质文件系统作为内核根文件系统，kerne\_init线程运行此打开的目标文件作为第一个用户进程。

如果不能打开以两个文件中的任意一个，kernel\_init\_freeable()函数将调用**prepare\_namespace()**函数挂载**"root=\*\*\*"**命令行参数（参数值为块设备文件路径名）指示的块设备中的文件系统作为内核根文件系统，如上图所示。随后，判断如果传递了**"init=\*\*\*"**命令行参数（参数值为第一个用户进程目标文件路径名），则运行参数值指示的目标文件作为第一个用户进程，目标文件位于刚挂载的外部介质文件系统中。如果没有传递**"init=\*\*\*"**命令行参数，内核将依次在挂载的外部介质文件系统中查找以下目标文件，以先到者为先，运行第一个用户进程，内核启动大功告成。

●**/sbin/init**

●**/etc/init**

●**/bin/init**

●**/bin/sh**

## 2.7小结

本章粗略地介绍了从加载内核目标文件到物理内存至运行第一个用户进程的过程。处理器启动时首先运行的引导加载程序负责将内核目标文件加载至内存，并向内核传递参数，如命令行参数等。内核从体系结构相关的head.S文件中的代码开始运行，随后运行体系结构无关的启动函数start\_kernel()完成内核的启动。

start\_kernel()函数完成处理器硬件的设置（初始化），内核各子系统的初始化等工作，最后创建内核线程，内核路径自身转换成系统内的空闲进程。内核线程kernel\_init完成剩余初始化工作后，按需要加载外部介质根文件系统，运行第一个用户进程，内核启动完成。

start\_kernel()函数是理解内核各子系统的引子，在后续章节介绍各子系统时将详细介绍各子系统初始化函数的实现。