linux 3.6 启动源码分析(七) do_initcalls

原创 2013年12月18日 14:16:50 2458

do_initcalls()将按顺序从由__initcall_start开始，到__initcall_end结束的section中以函数指针的形式取出这些编译到内核的驱动模块中初始化函数起始地址，来依次完成相应的初始化。而这些初始化函数由__define_initcall(level,fn)指示编译器在编译的时候，将这些初始化函数的起始地址值按照一定的顺序放在这个section中。

由于内核某些部分的初始化需要依赖于其他某些部分的初始化的完成，因此这个顺序排列常常非常重要。先看看调用源码：

```cpp
static void __init do_initcalls(void)
{
    int level;

    for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1; level++)
        do_initcall_level(level);//依次调用不同等级的初始化函数
}
```

```cpp
extern initcall_t __initcall_start[];
extern initcall_t __initcall0_start[];
extern initcall_t __initcall1_start[];
extern initcall_t __initcall2_start[];
extern initcall_t __initcall3_start[];
extern initcall_t __initcall4_start[];
extern initcall_t __initcall5_start[];
extern initcall_t __initcall6_start[];
extern initcall_t __initcall7_start[];
extern initcall_t __initcall_end[];

static initcall_t *initcall_levels[] __initdata = {
    __initcall0_start,
    __initcall1_start,
    __initcall2_start,
    __initcall3_start,
    __initcall4_start,
    __initcall5_start,
    __initcall6_start,
    __initcall7_start,
    __initcall_end,
};

/* Keep these in sync with initcalls in include/linux/init.h */
static char *initcall_level_names[] __initdata = {
    "early",
    "core",
    "postcore",
    "arch",
    "subsys",
    "fs",
    "device",
    "late",
};

static void __init do_initcall_level(int level)
{
    extern const struct kernel_param __start___param[], __stop___param[];
    initcall_t *fn;

    strcpy(static_command_line, saved_command_line);
    parse_args(initcall_level_names[level],
           static_command_line, __start___param,
           __stop___param - __start___param,
           level, level,
           &repair_env_string);

    for (fn = initcall_levels[level]; fn < initcall_levels[level+1]; fn++)
        do_one_initcall(*fn);
}
```

调用其实很简单，下面看看linux是怎么样构造这个section的：

1、分析 __define_initcall(level,fn) 宏定义

1) 这个宏的定义位于inlclude\linux\init.h中：

```cpp
#define __define_initcall(level,fn,id) \
    static initcall_t __initcall_##fn##id __used \
    __attribute__((__section__(".initcall" level ".init"))) = fn
```

其中 initcall_t 是一个函数指针类型：

typedef int (*initcall_t)(void);

而属性 __attribute__((__section__())) 则表示把对象放在一个这个由括号中的名称所指代的section中。

所以这个宏定义的的含义是：

1) 声明一个名称为__initcall_##fn##id的函数指针(其中##表示替换连接，)；linux3.6比2.6多添加了一个id作为扩展。

2) 将这个函数指针初始化为fn；

3) 编译的时候需要把这个函数指针变量放置到名称为 ".initcall" level ".init" 的section中(比如level="1"，代表这个section的名称是 ".initcall1.init")。

2) 举例：__define_initcall(6, pci_init,1s)

qing_ping
关注
原创 30 粉丝 31 喜欢 1 评论 1
等级： 博客 3 访问量： 3万+
积分： 650 排名： 7万+

上述宏调用的含义是：1) 声明一个函数指针__initcall_pic_init_is = pci_init； 这个指针变量__initcall_pic_init_1s 需要放置到名称为 .initcall6.init的section中(其实质就是将 这个函数pic_init的首地址放置到了这个section中)。

3) 这个宏一般并不直接使用，而是被定义成下述其他更简单的若干个衍生宏
这些衍生宏宏的定义也位于 inlclude\linux\Init.h 中:

```cpp
#define early_initcall(fn)      __define_initcall("early",fn,early)

/*
 * A "pure" initcall has no dependencies on anything else, and purely
 * initializes variables that couldn't be statically initialized.
 *
 * This only exists for built-in code, not for modules.
 * Keep main.c:initcall_level_names[] in sync.
 */
#define pure_initcall(fn)       __define_initcall("0",fn,0)

#define core_initcall(fn)       __define_initcall("1",fn,1)
#define core_initcall_sync(fn)      __define_initcall("1s",fn,1s)
#define postcore_initcall(fn)       __define_initcall("2",fn,2)
#define postcore_initcall_sync(fn)  __define_initcall("2s",fn,2s)
#define arch_initcall(fn)       __define_initcall("3",fn,3)
#define arch_initcall_sync(fn)      __define_initcall("3s",fn,3s)
#define subsys_initcall(fn)     __define_initcall("4",fn,4)
#define subsys_initcall_sync(fn)    __define_initcall("4s",fn,4s)
#define fs_initcall(fn)         __define_initcall("5",fn,5)
#define fs_initcall_sync(fn)        __define_initcall("5s",fn,5s)
#define rootfs_initcall(fn)     __define_initcall("rootfs",fn,rootfs)
#define device_initcall(fn)     __define_initcall("6",fn,6)
#define device_initcall_sync(fn)    __define_initcall("6s",fn,6s)
#define late_initcall(fn)       __define_initcall("7",fn,7)
#define late_initcall_sync(fn)      __define_initcall("7s",fn,7s)
```

因此通过宏 core_initcall() 来声明的函数指针，将放置到名称为.initcall1.init的section中，而通过宏postcore_initcall() 来声明的函数指针，将放置到名称为.initcall2.init的section中，依次类推。

2、与初始化调用有关section--initcall.init被分成了7或14个子section
1) 它们依次是.initcall1.init、.initcall2.init、...、.initcall7.init，linux3.0之后增加了一个扩展s.
2) 按照先后顺序依次排列
3) linux各个版本定义的形式也可能所改变，但作用是一致的。

linux3.0 ARM平台在include\asm-generic\vmlinux.lds.h里面有

```cpp
#define INIT_CALLS_LEVEL(level)                     \
        VMLINUX_SYMBOL(__initcall##level##_start) = .;      \
        *(.initcall##level##.init)              \
        *(.initcall##level##s.init)             \

#define INIT_CALLS                          \
        VMLINUX_SYMBOL(__initcall_start) = .;           \
        *(.initcallearly.init)                  \
        INIT_CALLS_LEVEL(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) = .;
```

而在makefile 中有
LDFLAGS_vmlinux += -T arch/$(ARCH)/kernel/vmlinux.lds.s

4) 在这7或14个section总的开始位置被标识为__initcall_start，而在结尾被标识为__initcall_end。

3、 内核初始化函数do_basic_setup(): do_initcalls() 将从.initcall.init 中，也就是这几个section中依次取出所有的函数指针，并调用这些函数指针所指向的函数，来完成内核的一些相关的初始化。

4、 总结
1) __define_initcall(level,fn)的作用就是指示编译器把一些初始化函数的指针(即:函数起始地址)按照顺序放置一个名为 .initcall.init 的section中，这个section又被分成了若干个子section，它们按顺序排列。在内核初始化阶段，这些放置到这个section中的函数指针将供do_initcalls() 按顺序依次调用，来完成相应初始化。
2) 函数指针放置到的子section由宏定义的level确定，对应level较小的子section位于较前面。而位于同一个子section内的函数指针顺序不定，由编译器按照编译的顺序随机指定。
3) 因此，如果你希望某个初始化函数在内核初始化阶段就被调用，那么你 就应该使用宏__define_initcall(level,fn) 或 其几个衍生宏 把这个函数fn的对应的指针放置到按照初始化的顺序放置到相关的 section中。如果某个初始化函数fn_B需要依赖于另外一个初始化函数fn_A的完成，那么你应该把fn_B放在比fn_A对应的level值较大的子section中， 这样，do_initcalls()将在fn_A之后调用fn_B。

如果你希望某个初始化函数在内核初始化阶段就被调用，那么你就应该使用宏__define_initcall(level,fn) 或其7个衍生宏来把这个初始化函数fn的起始地址按照初始化的顺序放置到相关的section 中。 内核初始化时的do_initcalls()将从这个section中按顺序找到这些函数来执行。如果你希望某个初始化函数在内核初始化阶段就被调用，那么你就应该使用宏__define_initcall(level,fn) 或 其几个衍生宏来把这个初始化函数fn的起始地址按照初始化的顺序放置到相关的section 中。 内核初始化时的do_initcalls()将从这个section中按顺序找到这些函数来执行。所以在源码中只要是使用这几个宏或者是衍生宏声明的函数，只要被编译到内核中，就会在do_initcalls被调用，从而初始化设备。

linux内核的子系统（或者说功能模块的）初始化

内核启动过程中需要完成各个部分的初始化，比如中端、页面管 goodluckwhh 2013年10月16日 22:42 3224
理、slab分配器、任务调度器、网络、PCI设备等等的初始化，这
些初始化大概可以分为两种： 一种是关键的，必须完成的而且必须以特定的顺序来完成...

宏__define_initcall(level,fn)的作用 和 do_initcalls()

前言 宏定义__define_initcall(level,fn)对于内核的初始化很重 xiaoxiaozhu2010 2013年12月30日 22:46 867
要，它指示 编译器在编译的时候，将一系列初始化函数的起
始地址值按照一定的顺序 放在一个sec...

Linux内核很吊之 module_init解析（下） Richard_LiuJH 2015年07月04日 22:00 3189

分析module_init 的作用。和module_init注册函数过程，以及注册函数被执行的过程。...

Linux do_initcall_level() chenliang0224 2017年12月07日 23:56 55

1. do_initcalls() 路径：linux-3.10.x\init\main.c start_kernel()-->rest_init()-->kernel_init()-->kerne...

Linux内核中的初始化initcall crazycoder8848 2016年03月10日 15:49 1823

由于在下水平相当有限，不当之处，还望大家批评指正^_^ 基于Linux 2.6.32 include\asm-generic\Vmlinux.lds.h中有如下
定义： 这等于是安排了名称如.in...

linux 3.6 启动源码分析(七) do_initcalls qing_ping 2013年12月18日 14:16 2458

do_initcalls()将按顺序从由__initcall_start开始，到__initcall_end结束的section中以函数指针的形式取出这些编译到内核的
驱动模块中初始化函数起始地址，来依...

Linux内核驱动加载过程 yanlinembed 2016年01月29日 11:30 2455

Linux内核驱动加载过程 驱动加载分为两种情况：静态加载和动态加载。 1. 静态加载 静态加载的方法是把驱动程序直接编译
进内核，然后内核在启动过程中由do_initcall()函数加载。 d...

linux 3.6 启动源码分析(一) qing_ping 2013年12月16日 13:02 2080

作为需要和硬件打交道的工程师来说，比较关注的是驱动和CPU初始化这一块。所以我沿着启动的路线，重点学习一下和硬
件相关的代码。就从linux解压的入口说起。学习阶段，基本是参考大神文章http://bl...

linux 3.6 启动源码分析(二) start_kernel qing_ping 2013年12月16日 13:38 2910

在构架相关的汇编代码运行完之后，程序跳入了构架无关的内核C语言代码：init/main.c中的start_kernel函数，在这个函数
中Linux内核开始真正进入初始化阶段， 进行一系列与内核相关的初...

Linux 系统启动过程详解 yyt8yyt8 2012年09月06日 17:36 2672

以RedHat9.0和i386平台为例---- BIOS 第一步：PC在上电以后，CPU从地址FFFF:0000开始执行(这个地址在ROM BIOS
中，ROM BIOS一般是在FEOOOh...

Linux内核驱动加载过程 sunjing_ 2018年02月23日 16:14 12

http://blog.csdn.net/yanlinembed/article/details/50606112 Linux内核驱动加载过程 驱动加载分为两种情况：静态加载
和动态加载...

linux 3.6 启动源码分析(四) rest_init qing_ping 2013年12月16日 14:25 1311

在内核初始化函数start_kernel执行到最后，就是调用rest_init函数，这个函数的主要使命就是创建并启动内核线程init。这个
函数虽然意思为剩下的初始化，但是这个“剩下”的可是内容颇多，下...

linux 3.6 启动源码分析(五) kernel_init进程

qing_ping 2013年12月16日 14:58 3061

在start_kernel最后的rest_init函数中内核创建了两个内核线程，一个是内核线程的管理者，另一个是内核初始化线程kernel_init. kernel_init它将完成设备驱动程序的初始...

linux 3.6 启动源码分析(三) setup_arch

qing_ping 2013年12月16日 14:05 1894

setup_arch()函数是start_kernel阶段最重要的一个函数，每个体系都有自己的setup_arch()函数，是体系结构相关的，具体编译哪个体系的setup_arch()函数，由顶层Ma...

linux 3.6 启动源码分析(六) do_basic_setup

qing_ping 2013年12月16日 16:20 1685

在内核init线程中调用了do_basic_setup，这个函数也做了很多内核和驱动的初始化工作 /*好了,设备现在已经初始化完成。但是还没有一个设备被初始化过， 但是 CPU 的子系统已经启动...

Juce源码分析（八）强引用与弱引用

Skilla 2014年11月03日 15:58 1552

Juce中有一对强弱引用，分别

学习Linux-4.12内核网路协议栈（1.1）——系统的初始化(do_initcalls)

我们知道，不管在什么样的平台上启动linux，它的开始都是以start_kernel（ ）进行系统的初始化，当然网络协议栈的初始化也是在这个过程中完成，下面从start_kernel()开始跟踪： st...

lee244868149 2017年07月03日 17:42 494

【Python】将Linux 上的Python2.7 升级成Python3.6

wwwdaan5com 2017年10月12日 18:38 460

最近为了测试一些功能，所以装了一台Linux虚拟机。 主要是想写些python程式，但是装完虚拟机器后，发现内建python的版本都是2.7.5 。 据我所知，最新版已经到3.6.3了，看来是自己...

Spring Security源码分析七：Spring Security 记住我

dandandeshangni 2018年01月17日 22:53 80

有这样一个场景——有个用户初访并登录了你的网站，然而第二天他又来了，却必须再次登录。于是就有了“记住我”这样的功能来方便用户使用，然而有一件不言自明的事情，那就是这种认证状态的”旷日持久“早已超出了用...

ARM多核引导过程

cs0301lm 2014年11月13日 14:31 6347

引导过程概述 (这个硕士论文得到过ARM公司Catalin Marinas的认可)