《Linux 内核修炼之道》

作者：华清远见

第 3 章 浏览内核源代码

本章简介

阅读本章的内容之前，我们有必要首先端正一下自己的认识：学习内核并不等同于学习内核书籍。LKD、ULK 等确实经典，但整日地抱着它们用功地啃，最多只是说明你是一个很有上进心很应该得到表彰的好青年、好同志，不过也就仅此而已。

毫不夸张地说，学习内核说的就是学习内核代码，内核代码本身就是最好的参考资料，其他任何经典或非经典的书笔者都只是起辅助作用，不能也不应该取代内核代码在我们学习过程中的主导地位。

有了正确的认识之后，本章接下来的内容会使你在学习内核源码时，不会迷失在无尽的代码海洋里。

3.1 内核学习的技术基础

内核的学习是一项浩大的工程，需要首先掌握以下几个基础。

（1）熟练使用 Linux 操作系统。

Linux 操作系统是 Linux 内核在用户层面的具体体现，只有熟练掌握 Linux 的基本操作，才能在内核学习的过程中达到事半功倍的效果。

（2）掌握操作系统理论基础。

只需要掌握操作系统中比较基础的理论，比如分时（time-shared）和实时（real-time）的区别，进程的概念，CPU 和系统总线、内存的关系等。

（3）掌握 C 语言基础。

不需要很精通 C 语言，但能够理解链表、散列表等数据结构的 C 实现，用过 GCC 编译器。当然，越熟悉 C 语言就会越有帮助。

3.2 内核体系结构

如图 3.1 所示是一个完整操作系统最基本的视图，它向我们传递了这样的信息——内核将应用程序和硬件分离开来。

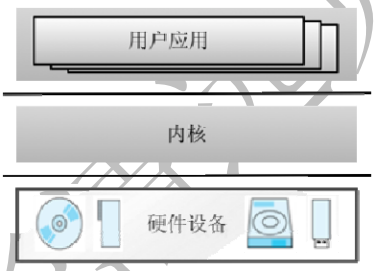

图 3.1　操作系统结构的基本视图

内核一方面负责与计算机硬件进行交互，实现对硬件的编程控制和接口操作，调度对硬件资源的访问，另一方面为用户应用程序提供一个高级的执行环境和访问硬件的虚拟接口。

提供硬件的兼容性是内核的设计目标之一，几乎所有的硬件，只要不是为其他操作系统所定制，都可以得到 Linux 的支持。

与硬件兼容性相关的是可移植性，即在不同的硬件平台上运行 Linux 的能力。从最初只支持标准 IBM 兼容机上的 Intel X86 架构到现在可以支持 Alpha、ARM、MIPS、PowerPC 等几乎所有硬件平台，如此广泛的平台支持之所以能够成功，部分原因在于内核清晰地划分为了体系相关部分和体系无关部分。

因此，相比图 3.1，Linux 操作系统更为普遍的视图是图 3.2。我们从中可知：部分内核为体系结构和硬件所特有，即体系相关部分；部分内核是可移植的，即体系无关部分。

图 3.2　Linux 操作系统结构的基本视图

体系无关部分通常会定义与体系相关部分的接口，这样，内核向新的体系结构移植的过程就变成确认这些接口的特性并将它们加以实现的过程。

同时，用户应用程序和内核之间的联系，一般是通过它和内核的中间层——标准 C 库来实现，而标准 C 库函数本身，则是建立在内核提供的系统调用基础之上。

通过标准 C 库，以及内核体系无关部分与体系相关部分的接口，用户应用程序和部分内核都成为可移植的。图 3.3 所示为 Linux 操作系统更为标准的视图。

图 3.3　Linux 操作系统结构的标准视图

（1）系统调用接口。

为了与用户应用程序进行交互，内核提供了一组系统调用接口，通过这组接口，应用程序可以访问系统硬件和各种操作系统资源。

系统调用接口层在用户应用程序和内核之间添加了一个中间层，形象地说，它扮演了一个函数调用多路复用和多路分解器的角色。

（2）进程管理。

进程管理负责创建和销毁进程，并处理它们之间的互相联系（进程间通信），同时负责安排调度它们去分享 CPU。

进程管理部分实现了一个进程世界的抽象，这个进程世界类似于我们的人类世界，只不过我们人类世界里的个体是人，而在进程世界里则是一个一个的进程，人与人之间通过书信、手机、网络等进行交互，而各个进程之间则是通过不同方式的进程间通信，我们所有人都在分享同一个地球，而所有进程都在分享一个或多个 CPU。

（3）内存管理。

在进程世界里，内存是重要的资源之一，就好比我们的土地。因此，管理内存的策略与方式是决定系统性能的一个关键因素。

内核的内存管理部分根据不同的需要，提供了包括 malloc/free 在内的许多简单或者复杂的接口，并为每个进程都提供了一个虚拟的地址空间，基本上实现虚拟内存对进程的按需分配。

（4）虚拟文件系统。

虚拟文件系统为用户空间提供了文件系统接口，同时又为各个具体的文件系统提供了通用的接口抽象。在 VFS 上面，是对诸如 open、close、read 和 write 之类函数的一个通用 API 抽象，在 VFS 下面则是具体的文件系统，它们定义了上层函数的实现方式。

通过虚拟文件系统，我们可以利用标准的 Linux 文件系统调用对不同介质上的不同文件系统进行操作。应该说，VFS 是内核在各种具体的文件系统上建立的一个抽象层，它提供了一个通用的文件系统模型，而该模型囊括了我们所能想到的所有文件系统的行为。

（5）网络功能。

网络子系统处理数据包的收集、标识、分发，路由和地址的解析等所有网络有关的操作。socket 层是网络子系统的标准 API，它为各种网络协议提供了一个用户接口。

（6）设备驱动程序。

操作系统的目的是为用户提供一种方便访问硬件的途径，因此，几乎每一个系统操作最终都会映射到物理的硬件设备上。除了 CPU、内存等有限的几个对象，所有设备的访问控制操作都要由相关的代码来完成，这些代码就是所谓的设备驱动程序。

（7）依赖体系结构的代码。

前面讲到，部分内核代码是体系相关的，./linux/arch 子目录定义了内核源代码中依赖于体系结构的部分，其中包含了对应各种特定体系结构的子目录。比如，对于一个典型的桌面系统来说，使用的是 i386 目录。

每个特定体系结构对应的子目录又包含了很多下级子目录，分别关注内核中的一个特定方面，比如引导、内核、内存管理等。

3.3 内核源码目录结构

浏览内核代码之前，有必要知道内核源码的整体分布情况，按照惯例，内核代码安装在/usr/src/linux 目录下，该目录下的每一个子目录都代表了一个特定的内核功能性子集，下面针对 2.6.23 版本进行简单描述。

（1）Documentation。

这个目录下面没有内核代码，只有很多质量参差不齐的文档，但往往能够给我们提供很多的帮助。

（2）arch。

所有与体系结构相关的代码都在这个目录以及 include/asm-*/目录中，Linux 支持的每种体系结构在 arch 目录下都有对应的子目录，而在每个体系结构特有的子目录下又至少包含 3 个子目录。

kernel：存放支持体系结构特有的诸如信号量处理和 SMP 之类特征的实现。

lib：存放体系结构特有的对诸如 strlen 和 memcpy 之类的通用函数的实现。

mm：存放体系结构特有的内存管理程序的实现。

除了这 3 个子目录之外，大多数体系结构在必要的情况下还有一个 boot 子目录，包含了在这种硬件平台上启动内核所使用的部分或全部平台特有代码。

此外，大部分体系结构所特有的子目录还根据需要包含了供附加特性使用的其他子目录。比如，i386 目录包含一个 math-emu 子目录，其中包括了在缺少数学协处理器（FPU）的 CPU 上运行模拟 FPU 的代码。

（3）drivers。

这个目录是内核中最庞大的一个目录，显卡、网卡、SCSI 适配器、PCI 总线、USB 总线和其他任何 Linux 支持的外围设备或总线的驱动程序都可以在这里找到。

（4）fs。

虚拟文件系统（VFS，Virtual File System）的代码，和各个不同文件系统的代码都在这个目录中。Linux 支持的所有文件系统在 fs 目录下面都有一个对应的子目录。比如 ext2 文件系统对应的是 fs/ext2 目录。

一个文件系统是存储设备和需要访问存储设备的进程之间的媒介。存储设备可能是本地的物理上可访问的，比如硬盘或 CD-ROM 驱动器，它们分别使用 ext2/ext3 和 isofs 文件系统；也可能是通过网络访问的，使用 NFS 文件系统。

还有一些虚拟文件系统，比如 proc，它以一个标准文件系统出现，然而，它其中的文件只存在于内存中，并不占用磁盘空间。

（5）include。

这个目录包含了内核中大部分的头文件，它们按照下面的子目录进行分组。

include/asm-*/，这样的子目录有多个，每一个都对应着一个 arch 的子目录，比如 include/asm-alpha、include/asm-arm、include/asm-i386 等。每个子目录中的文件都定义了支持给定体系结构所必须的预处理器宏和内联函数，这些内联函数多数都是全部或部分使用汇编语言实现的。

编译内核时，系统会建立一个从 include/asm 目录到目标体系结构特有的目录的符号链接。比如对于 arm 平台，就是 include/asm-arm 到 include/asm 的符号链接。因此，体系结构无关部分的内核代码可以使用如下形式包含体系相关部分的头文件。

```
#include <asm/some-file>
```

include/linux/，与平台无关的头文件都在这个目录下面，它通常会被链接到目录/usr/include/linux（或者它里面的所有文件会被复制到/usr/include/linux 目录下面）。因此用户应用程序里和内核代码里的语句：

```
#include <linux/some-file>
```

包含的头文件的内容是一致的。

include 目录下的其他子目录，在此不做赘述。

（6）init。

内核的初始化代码。包括 main.c、创建早期用户空间的代码以及其他初始化代码。

（7）ipc。

IPC，即进程间通信（interprocess communication）。它包含了共享内存、信号量以及其他形式 IPC 的代码。

（8）kernel。

内核中最核心的部分，包括进程的调度（kernel/sched.c），以及进程的创建和撤销（kernel/fork.c 和 kernel/exit.c）等，和平台相关的另外一部分核心的代码在 arch/*/kernel 目录。

（9）lib。

库代码，实现了一个标准 C 库的通用子集，包括字符串和内存操作的函数（strlen、mmcpy 和其他类似的函数）以及有关 sprintf 和 atoi 的系列函数。与 arch/lib 下的代码不同，这里的库代码都是使用 C 编写的，在内核新的移植版本中可以直接使用。

（10）mm。

包含了体系结构无关部分的内存管理代码，体系相关的部分位于 arch/*/mm 目录下。

（11）net。

网络相关代码，实现了各种常见的网络协议，如 TCP/IP、IPX 等。

（12）scripts。

该目录下没有内核代码，只包含了用来配置内核的脚本文件。当运行 make menuconfig 或者 make xconfig 之类的命令配置内核时，用户就是和位于这个目录下的脚本进行交互的。

（13）block。

block 层的实现。最初 block 层的代码一部分位于 drivers 目录，一部分位于 fs 目录，从 2.6.15 开始，block 层的核心代码被提取出来放在了顶层的 block 目录。

（14）crypto。

内核本身所用的加密 API，实现了常用的加密和散列算法，还有一些压缩和 CRC 校验　算法。

（15）security。

这个目录包括了不同的 Linux 安全模型的代码，比如 NSA Security-Enhanced Linux。

（16）sound。

声卡驱动以及其他声音相关的代码。

（17）usr。

实现了用于打包和压缩的的 cpio 等。

3.4 浏览代码的工具

工欲善其事，必先利其器。一个功能强大，同时又使用方便的代码浏览工具对于学习内核代码是很有帮助的。

3.4.1 Source Insight

Windows 下最为方便快捷的代码浏览工具是 Source Insight（商业软件）。

打开 Source Insight，创建一个工程，将内核代码加入到该工程中，并进行文件同步，然后就可以在代码之间进行关联阅读。Windows XP 中的 Source Insight 界面如图 3.4 所示。

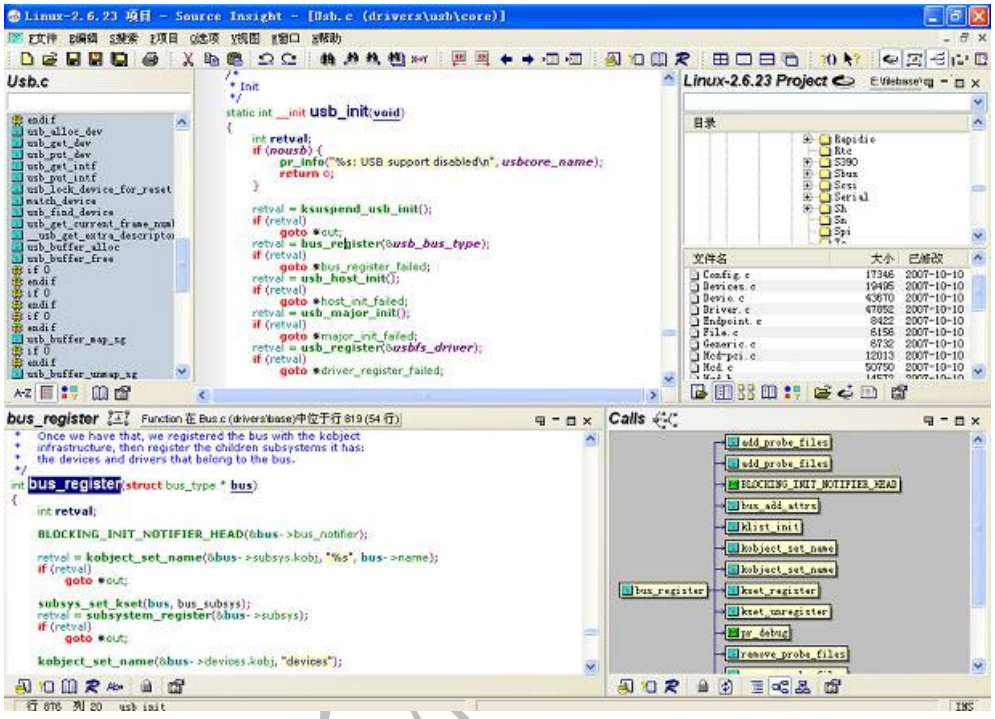

图 3.4 Windows XP 中的 Source Insight 界面

3.4.2 Vim+Cscope

Source Insight 并没有对应 Linux 的版本。因此对于很多 Linux 初学者来说，在一个完全的 Linux 环境下进行学习，首先要解决的一个问题就是寻找一个可以取代 Source Insight 的代码浏览工具。

这个工具就是 Vim，各种 Linux 发行版都会默认进行安装。虽然 Vim 默认的编辑界面很普通，甚至说丑陋，但是可以通过配置文件.vimrc 添加不同的界面效果。同时还可以配合 TagList、WinManager 等很多好用的插件或工具，将 Vim 打造成一个不次于 Source Insight 的代码浏览编辑工具。

在配合 Vim 适用的众多插件和工具当中，Cscope 无疑是最受注目的一个，因此下面重点介绍 Vim 如何配合 Cscope 进行使用。

1．Cscope 介绍

通过 Cscope，可以方便地获知某个函数定义的确切位置以及被哪些函数所调用，某个变量在哪里被赋值，某个符号曾经在哪些地方使用过等。

如果安装了 Vim 的中文帮助手册，就可以使用下面的命令阅读 Cscope 的详细介绍。

```
:help if_cscop.txt
```

2．安装 Cscope

通常，系统安装时并不会默认安装 Cscope，所以如果你的 Linux 系统中没有 Cscope 命令，就首先需要进行安装。

如果希望从源码进行安装，需要登录 Cscope 的主页（http://cscope.sourceforge.net）下载一个源码包，解压后编译安装。

```
# ./configure
# make
# make install
```

3．生成 Cscope 的数据库

使用 Cscope 的功能前，必须先为内核代码生成一个 Cscope 的数据库。在内核代码树的根目录运行下面的命令。

```
# cd /usr/src/linux
# cscope -Rbq
```

然后会生成以下 3 个文件。

```
# ll cscope.*
-rw-rw-r-- 1 root root  19030016 2008-07-18 10:56 cscope.in.out
-rw-rw-r-- 1 root root 147895939 2008-07-18 10:56 cscope.out
-rw-rw-r-- 1 root root 105656256 2008-07-18 10:56 cscope.po.out
```

打开内核文件，将刚才生成的 Cscope 文件导入 Vim 中，如下所示。

```
# vim init/main.c
# cs add /usr/src/linux/cscope.out /usr/src/linux
```

也可以将下面的语句添加到 Vim 配置文件.vimrc 中，这样就可以不用每次打开 Vim 都去执行上面的命令。

```
" add any cscope database in current directory
if filereadable("cscope.out")
"  cs add cscope.out
" else add the database pointed to by environment variable
elseif $CSCOPE_DB != ""
    cs add $CSCOPE_DB
endif
```

4．Cscope 的主要功能

Cscope 的主要功能通过它的子命令“find”来实现。

```
cs find c|d|e|f|g|i|s|t name
```

s：查找本 C 代码符号。

g：查找本定义。

d：查找本函数调用的函数。

c：查找调用本函数的函数。

t：查找本字符串。

e：查找本 egrep 模式。

f：查找本文件。

I：查找包含本文件的文件。

如果不愿意每次都要输入一长串的命令，还可以在.vimrc 文件中添加下面的快捷键。

```
nmap <C-\>s :cs find s <C-R>=expand("<cword>")<CR><CR>
nmap <C-\>g :cs find g <C-R>=expand("<cword>")<CR><CR>
```

```
nmap <C-\>c :cs find c <C-R>=expand("<cword>")<CR><CR>
nmap <C-\>t :cs find t <C-R>=expand("<cword>")<CR><CR>
nmap <C-\>e :cs find e <C-R>=expand("<cword>")<CR><CR>
nmap <C-\>f :cs find f <C-R>=expand("<cfile>")<CR><CR>
nmap <C-\>i :cs find i ^<C-R>=expand("<cfile>")<CR>$<CR>
nmap <C-\>d :cs find d <C-R>=expand("<cword>")<CR><CR>
```

具体到使用时，当光标停留在要查找的某个对象时，按下<C-\>g，即先按“Ctrl+\”，然后很快再按“g”，将会查找该对象的定义。

5．使用 Cscope

实用 Cscope 查找函数 start_kernel 的定义的指令如下：

```
:cs f g start_kernel
```

结果如图 3.5 所示。

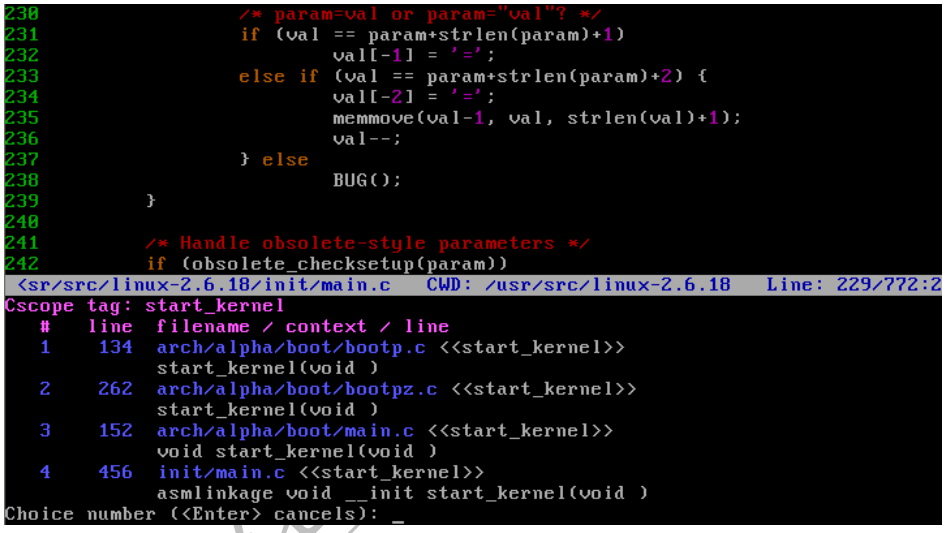

图 3.5　使用 Cscope 查找函数定义

图 3.5 命令栏里显示找到了多个 start_kernel 的定义，在光标提示处输入“4”，按回车键，将会跳转到 init/main.c 文件中定义 start_kernel 的位置。

3.4.3　LXR

LXR （Linux Cross Reference）也是比较流行的源代码浏览工具，它的下载及安装可参见：http://lxr.linux.no/。

如果目的只是浏览 Linux 内核代码，我们并不需要去安装 LXR。因为网站 http://lxr.linux.no/上已经提供了几乎所有版本的 Linux 内核代码，我们只需要登录该网站，选择某一特定的内核版本，就可以在内核代码之间进行关联阅读。

比如，登录网站并选择内核版本后，在查找框内输入要查找的内核代码符号名称，然后就可以搜索得到所有以超链接形式给出的对该符号定义和引用的确切位置。以 2.6.23 内核版本为例，查找 usb_hub_init 函数，得到下面的信息。

```
Code search: usb_hub_init
Function
       drivers/usb/core/hub.c, line 2771 [usage...]
Function prototype or declaration
       drivers/usb/core/usb.h, line 27 [usage...]
Freetext search: usb_hub_init (3 estimated hits)
```

```
drivers/usb/core/hub.c, line 2038 (100%)
drivers/usb/core/usb.c, line 894 (96%)
drivers/usb/core/usb.h, line 27 (83%)
```

3.5 内核代码的特点

Linux 内核同时使用 C 语言和汇编语言实现，C 语言编写的代码移植性较好、易于维护，而汇编语言编写的代码相当于针对特定的平台做了优化，速度较快，所以需要在它们之间寻找一定的平衡。

一般而言，在体系结构的底层或对执行时间要求严格的地方，会使用汇编语言，比如中断和异常处理的底层代码，初始化有关的部分代码等。内核其他部分则是用 C 语言编写。

3.5.1 GCC 扩展

Linux 内核必须使用 GNU 的 GCC 编译器来编译，而 GCC 提供了很多的 C 语言扩展，这些扩展对优化、目标代码布局、更安全的检查等提供了很强的支持。因此，内核代码所使用的 C 语法并不完全符合 ANSI C 标准，实际上，只要有可能，内核开发者总是要用到 GCC 提供的 C 语言扩展部分。

这就为我们学习内核增加了一定的困难，因此，为了能够比较顺利地阅读内核代码，有必要对 GCC 扩展进行了解。

下面详细介绍 Linux 内核中常出现的、主要的 GCC 扩展。

1．语句表达式（statement-embedded expression）

GCC 把包含在括号中的复合语句看作是一个表达式，称为语句表达式，它允许在一个表达式内使用循环、跳转、局部变量，并可以出现在任何允许表达式出现的地方。

位于括号中的复合语句的最后一句必需是一个以分号结尾的表达式，它的值将成为这个语句表达式的值。

计算最大值和最小值通常被定义为：

```
#define max(x, y) ((x) > (y) ? (x) : (y))
#define min(x, y) ((x) < (y) ? (x) : (y))
```

但是其中的 x 和 y 可能会分别被计算两次。当参数 x 和 y 带有副作用时，将会产生错误的结果。而内核则使用语句表达式将其定义为：

```
++++ include/linux/kernel.h
305 #define min_t(type,x,y) \
306     ({ type __x = (x); type __y = (y); __x < __y ? __x: __y; })
307 #define max_t(type,x,y) \
308     ({ type __x = (x); type __y = (y); __x > __y ? __x: __y; })
```

相比较来看，使用语句表达式只计算参数一次，避免了可能的错误。

如果对 typeof 加以利用，还可以定义更为通用的宏，针对最大值最小值的计算，内核的另外一种定义为：

```
++++ include/linux/kernel.h
287 #define min(x,y) ({ \
288     typeof(x) _x = (x);  \
289     typeof(y) _y = (y);  \
290     (void) (&_x == &_y);\
291     _x < _y ? _x : _y; })
292
```

```
293 #define max(x,y) ({ \
294     typeof(x) _x = (x);  \
295     typeof(y) _y = (y);  \
296     (void) (&_x == &_y);\
297     _x > _y ? _x : _y; })
```

其中 typeof(x) 表示 x 的类型，第 294 行定义了一个与 x 类型相同的局部变量_x，并将其初始化为 x，注意第 290 行的作用是检查参数 x 和 y 的类型是否相同。

2．零长度数组（flexible array）

在内核代码里经常出现类似下面的结构。

```
++++ include/linux/usb
199 struct usb_interface_cache {
200     unsigned num_altsetting; /* number of alternate settings */
201     struct kref ref;     /* reference counter */
202
203     /* variable-length array of alternate settings for this interface,
204      * stored in no particular order */
205     struct usb_host_interface altsetting[0];
206 };
```

结构的最后一个元素 usb_host_interface altsetting[0]就是 GCC 所谓的零长度数组，也可以称之为可变长数组，它并不占用结构的空间，但它意味着这个结构的长度充满了变数。创建该结构对象时，可以根据实际的需要指定这个可变长数组的长度，并分配相应的空间。

3．可变参数宏

1999 年的 ISO C 标准里规定了可变参数宏，语法和函数类似，比如：

```
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)
```

其中的“…”表示可变参数，实际调用时，它们会替代宏体里的__VA_ARGS__。GCC 支持更复杂的形式，可以给可变参数取个名字，如下所示。

```
#define debug(format, args...) fprintf (stderr, format, args)
```

有了名字之后，代码显得更具有可读性。内核中的例子为：

```
++++ include/linux/kernel.h
244 #define pr_info(fmt,arg...) \
245     printk(KERN_INFO fmt,##arg)
```

其中的 pr_info 和上面的 debug 形式除了“##”外，几近相同。“##”主要针对参数为空的情况。既然称为可变参数，那传递空参数也是可以的。如果没有使用“##”，传递空参数时，比如：

```
debug ("A message");
```

宏展开后，其中的字符串后面会多个多余的逗号，而“##”则会使预处理器去掉这个多余的逗号。

4．标号元素

在标准 C 里，数组或结构变量的初始化值必须以固定的顺序出现，而在 GCC 中，通过指定索引或结构域名，则允许初始化值以任意顺序出现。

指定数组索引的方法是在初始化值前写“[INDEX] =”，还可以使用“[FIRST ... LAST] =”的形式指定一个范围，比如：

```
++++ arch/ia64/kernel/acpi.c
132 int platform_intr_list[ACPI_MAX_PLATFORM_INTERRUPTS] = {
133     [0 ... ACPI_MAX_PLATFORM_INTERRUPTS - 1] = -1
134 };
```

将数组 platform_intr_list 的任何元素都初始化为-1。

对于结构初始化，比如：

```
++++ fs/ext2/file.c
42 const struct file_operations ext2_file_operations = {
43      .llseek     = generic_file_llseek,
44      .read       = do_sync_read,
45      .write      = do_sync_write,
46      .aio_read   = generic_file_aio_read,
47      .aio_write  = generic_file_aio_write,
48      .ioctl      = ext2_ioctl,
49 #ifdef CONFIG_COMPAT
50      .compat_ioctl   = ext2_compat_ioctl,
51 #endif
52      .mmap       = generic_file_mmap,
53      .open       = generic_file_open,
54      .release= ext2_release_file,
55      .fsync      = ext2_sync_file,
56      .splice_read= generic_file_splice_read,
57      .splice_write   = generic_file_splice_write,
58 };
```

将结构 ext2_file_operations 的元素 llseek 初始化为 generic_file_llseek，元素 rea 初始化为 genenric_file_read，依次类推。使用这种形式，当结构体的定义变化导致元素的偏移位置改变时，仍然可以确保已知元素的正确性。对于未出现在初始化中的元素，其初值为 0。

5．特殊属性（__attribute__）

GCC 允许声明函数、变量和类型的特殊属性，以便指示编译器进行特定方面的优化和更仔细的代码检查。使用方式为在声明后加上：

```
__attribute__ (( ATTRIBUTE ))
```

其中 ATTRIBUTE 是属性的说明，多个说明之间以逗号分隔。GCC 可以支持十几个属性，下面介绍一些比较常用的。

noreturn

属性 noreturn 用于函数，表示该函数从不返回。它能够让编译器生成较为优化的代码，消除不必要的警告信息。

format（archetype, string-index, first-to-check）

属性 format 用于函数，表示该函数使用 printf、scanf、strftime 或 strfmon 风格的参数，并可以让编译器检查函数声明和函数实际调用参数之间的格式化字符串是否匹配。

archetype 指定是哪种风格，string-index 指定传入函数的第几个参数是格式化字符串，first-to-check 指定从函数的第几个参数开始按照上述规则进行检查。比如：

```
++++ include/linux/kernel.h
164 static inline int printk(const char *s, ...)
165     __attribute__ ((format (printf, 1, 2)));
```

表示 printk 的第一个参数是格式化字符串，从第二个参数开始根据格式化字符串检查参数。

unused

属性 unused 用于函数和变量，表示该函数或变量可能并不使用，这个属性能够避免编译器产生警告信息。

section ("section-name")

属性 section 用于函数和变量，比如：

```
++++ include/linux/init.h
43 #define __init       __attribute__ ((__section__ (".init.text"))) __cold
44 #define __initdata    __attribute__ ((__section__ (".init.data")))
45 #define __exitdata    __attribute__ ((__section__(".exit.data")))
46 #define __exit_call   __attribute_used__   __attribute__   ((__section__   (".exitcall.
exit")))
```

通常编译器将函数放在.text 节，变量放在.data 或.bss 节，而使用 section 属性，可以让编译器将函数或变量放在指定的节中。因此上面对__init 的定义便表示将__init 修饰的代码放在.init.text 节。

连接器可以把相同节的代码或数据安排在一起，Linux 内核很喜欢使用这种技术，比如__init 修饰的所有代码都会被放在.init.text 节里，初始化结束后就可以释放这部分内存。

aligned (ALIGNMENT)

属性 aligned 用于变量、结构或联合，设定一个指定大小的对齐格式，以字节为单位，比如：

```
++++ drivers/usb/host/ohci.h
181 struct ohci_hcca {
182 #define NUM_INTS 32
183     __hc32  int_table [NUM_INTS];    /* periodic schedule */
184
185 /*
186  * OHCI defines u16 frame_no, followed by u16 zero pad.
187  * Since some processors can't do 16 bit bus accesses,
188  * portable access must be a 32 bits wide.
189  */
190 __hc32  frame_no;       /* current frame number */
191 __hc32  done_head;      /* info returned for an interrupt */
192 u8  reserved_for_hc [116];
193 u8  what [4];       /* spec only identifies 252 bytes :) */
194 } __attribute__ ((aligned(256)));
```

表示结构体 ohci_hcca 的成员以 256 字节对齐。

如果 aligned 后面不紧跟一个指定的数字值，那么编译器将依据目标机器情况使用最大、最有益的对齐方式。

需要注意的是，attribute 属性的效果与你的连接器也有关，如果你的连接器最大只支持 16 字节对齐，那么此时定义 32 字节对齐也是无济于事的。

packed

属性 packed 用于变量和类型，用于变量或结构体成员时表示使用最小可能的对齐，用于枚举、结构体或联合类型时表示该类型使用最小的内存。

属性 packed 多用于定义硬件相关的结构时，使元素之间不会因对齐产生问题。比如：

```
++++ include/asm-i386/desc.h
295 struct usb_interface_descriptor {
296 __u8  bLength;
297 __u8  bDescriptorType;
298
299 __u8  bInterfaceNumber;
300 __u8  bAlternateSetting;
301 __u8  bNumEndpoints;
302 __u8  bInterfaceClass;
```

```
303 __u8  bInterfaceSubClass;
304 __u8  bInterfaceProtocol;
305 __u8  iInterface;
306 } __attribute__ ((packed));
```

其中__attribute__ ((packed))告诉编译器，usb_interface_descriptor 的元素为 1 字节对齐，不要再添加填充位。因为这个结构的定义和 usb spec 里是完全一致的，包括各个字段的长度，如果不给编译器这个暗示，编译器就会依据平台的类型在结构的每个元素之间添加一定的填充位，使用这个结构时就不能达到预期的结果。

6．内建函数

GCC 提供了大量的内建函数，其中很多是标准 C 库函数的内建版本，比如 memcpy，它们与对应的 C 库函数功能相同，这里就不再进行介绍。

还有很多内建函数的名字通常以 __builtin 开始，下面只对__builtin_expect 进行分析并示例，其他__builtin_xxx 函数的分析可参考这个分析过程。

__builtin_expect (long exp, long c)

遇到这类很陌生的使用方式，GCC 参考手册是我们最好的参考资料。下面是 GCC 参考手册里对__builtin_expect 的介绍。

```
long __builtin_expect (long exp, long c)
You may use __builtin_expect to provide the compiler with branch prediction information. In general, you should prefer to use actual profile feedback for this (‘-fprofile-arcs’), as programmers are notoriously bad at predicting how their programs actually perform. However, there are applications in which this data is hard to collect.
The return value is the value of exp, which should be an integral expression. The value of c must be a compile-time constant. The semantics of the built-in are that it is expected that exp == c. For example:
    if (__builtin_expect (x, 0))
        foo ();
    would indicate that we do not expect to call foo, since we expect x to be zero. Since
    you are limited to integral expressions for exp, you should use constructions such
    as
    if (__builtin_expect (ptr != NULL, 1))
        error ();
    when testing pointer or floating-point values.
```

大意是，由于大部分代码在分支预测方面做的比较糟糕，所以 GCC 提供了这个内建的函数来帮助处理分支预测、优化程序。它的第一个参数 exp 为一个整型的表达式，返回值也是这个 exp。它的第二个参数 c 的值必须是一个编译期的常量。这个内建函数的意思就是 exp 的预期值为 c，编译器可以根据这个信息适当地重排条件语句块的顺序，将符合这个条件的分支放在合适的地方。

具体到内核里，比如：

```
++++ include/linux/compiler.h
60 #define likely(x) __builtin_expect(!!(x), 1)
61 #define unlikely(x) __builtin_expect(!!(x), 0)
```

unlikely(x)用于告诉编译器，条件 x 发生的可能性不大，likely 则相反。它们一般用在条件语句里，if 语句不变，只是当 if 条件为 1，即为真的可能性非常小的时候，可以在条件表达式外面包装一个 unlikely()，那么这个条件块里语句的目标码可能就会被放在一个比较远的位置，以保证经常执行的目标码更紧凑。如果可能性非常大，则使用 likely()包装。

3.5.2 内嵌汇编

用汇编语言编写内核代码中的部分代码，大体上是出于如下几个方面的考虑。

（1）Linux 内核中的底层代码直接和硬件打交道，需要一些专用的指令，而这些指令在 C 语言中并无对应的语言成分。

（2）内核中实现某些操作的过程、代码段或函数，在运行时会很频繁地被调用，这时用汇编语言编写，其时间效率会有大幅度提高。

（3）在某些特别的场合，一段代码的空间效率也很重要，比如操作系统的引导程序一定要容纳在磁盘的第一个扇区中，多一个字节都不行。这时只能用汇编语言编写。

在 Linux 内核代码中，以汇编语言编写的代码，有两种不同的形式。一是完全的汇编代码，这样的代码采用.s 作为文档名的后缀。第二种是嵌入在 C 代码中的汇编语言片段。

对于新接触 Linux 内核源码的读者，即使比较熟悉 i386 汇编语言，在理解内核中的汇编代码时都会感到困难。其原因是，在内核的汇编代码中采用的是不同于常用 Intel i386 汇编语言的 AT&T 格式的汇编语言，而在嵌入 C 代码的汇编语言片段中，更是增加了一些指导汇编工具如何分配使用寄存器、如何与 C 代码中定义的变量相结合的语言成分。这些成分使得嵌入 C 代码的汇编语言片断实际上变成了一种介于汇编和 C 之间的一种中间语言。

3.6 内核中的链表

鉴于链表在内核中的特殊地位，有必要在此对其做一番陈述。内核中链表的实现位于 include/linux/list.h 文件，链表数据结构的定义也很简单。

```
21 struct list_head {
22      struct list_head *next, *prev;
23 };
```

list_head 结构包含两个指向 list_head 结构的指针 prev 和 next，由此可见，内核中的链表实际上都是双链表（通常都是双循环链表）。

通常，我们在数据结构课堂上所了解的链表定义方式是这样的（以单链表为例）：

```
struct list_node {
    struct list_node *next;
    ElemTypedata;
};
```

通过这种方式使用链表，对每一种数据类型，都要定义它们各自的链表结构。而内核中的链表却与此不同，它并没有数据域，不是在链表结构中包含数据，而是在描述数据类型的结构中包含链表。

比如在 hub 驱动中使用 struct usb_hub 来描述 hub 设备，hub 需要处理一系列的事件，比如当探测到一个设备连进来时，就会执行一些代码去初始化该设备，所以 hub 就创建了一个链表来处理各种事件，这个链表的结构如图 3.6 所示。

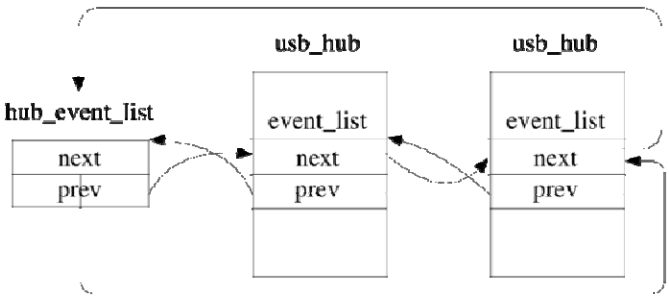

图 3.6　hub 链表结构

（1）声明与初始化。

链表的声明可以使用两种方式，一种为使用 LIST_HEAD 宏在编译时静态初始化，一种为使用 INIT_LIST_HEAD()在运行时进行初始化。

```
25 #define LIST_HEAD_INIT(name) { &(name), &(name) }
26
27 #define LIST_HEAD(name) \
28      struct list_head name = LIST_HEAD_INIT(name)
29
30 static inline void INIT_LIST_HEAD(struct list_head *list)
31 {
32      list->next = list;
33      list->prev = list;
34 }
```

无论采用哪种方式，新生成的链表头的两个指针 next、prev 都初始化为指向自己。

（2）判断链表是否为空。

```
298 static inline int list_empty(const struct list_head *head)
299 {
300     return head->next == head;
301 }
```

（3）插入。

有了链表，自然就要对其进行操作，就要向里面加东西。list_add()和 list_add_tail()这两个函数可以完成这个工作。

```
67 static inline void list_add(struct list_head *new, struct list_head *head)
68 {
69      __list_add(new, head, head->next);
70 }
84 static inline void list_add_tail(struct list_head *new, struct list_head *head)
85 {
86      __list_add(new, head->prev, head);
87 }
```

其中，list_add()将数据插入在 head 之后，list_add_tail()将数据插入在 head->prev 之后。其实对于循环链表来说，表头的 next、prev 分别指向链表中的第一个和最后一个节点，所以，list_add()和 list_add_tail()的区别并不大。

（4）删除。

链表里的元素不能只加不减，没用了的元素就应该删除掉。

```
224 static inline void list_replace_init(struct list_head *old,
225                     struct list_head *new)
226 {
227     list_replace(old, new);
228     INIT_LIST_HEAD(old);
229}
```

list_replace_init()从链表里删除一个元素，并且将其初始化。

（5）遍历。

内核中的链表仅仅保存了 list_head 结构的地址，我们如何通过它获取一个链表节点真正的数据项？这就要提到链表的所有操作里面最为重要的 list_entry 宏了，我们可以通过它很容易地获得一个链表节点的数据。

```
425 #define list_entry(ptr, type, member) \
426     container_of(ptr, type, member)
```

还是 hub 驱动的那个例子，当我们要处理 hub 的事件的时候，我们当然需要知道具体是哪个 hub 触发了这起事件。而 list_entry 的作用就是，从 struct list_head event_list 得到它所对应的 struct usb_hub 结构体变量。比如以下 4 行代码：

```
struct list_head *tmp;
struct usb_hub *hub;
tmp = hub_event_list.next;
hub = list_entry(tmp, struct usb_hub, event_list);
```

从全局链表 hub_event_list 中取出一个来，叫做 tmp，然后通过 tmp，获得它所对应的 struct usb_hub。

3.7 Kconfig 和 Makefile

毫不夸张地说，Kconfig 和 Makefile 是我们浏览内核代码时最为依仗的两个文件。基本上，Linux 内核中每一个目录下边都会有一个 Kconfig 文件和一个 Makefile 文件。Kconfig 和 Makefile 就好似一个城市的地图，地图引导我们去认识一个城市，而 Kconfig 和 Makefile 则可以让我们了解一个内核目录下面的结构。在希望研究内核的某个子系统、某个驱动或其他某个部分时，都有必要首先仔细阅读一下相关目录下的 Kconfig 和 Makefile 文件。

3.7.1 Kconfig 结构

每种平台对应的目录下面都有一个 Kconfig 文件，比如 arch/i386/Kconfig，该文件通过 source 语句构建出一个 Kconfig 树。文件 arch/i386/Kconfig 的内容片段如下：

```
mainmenu "Linux Kernel Configuration"

config X86_32
    bool
    default y
    help
      This is Linux's home port.  Linux was originally native to the Intel
      386, and runs on all the later x86 processors including the Intel
      486, 586, Pentiums, and various instruction-set-compatible chips by
      AMD, Cyrix, and others.
……
source "init/Kconfig"

menu "Processor type and features"

source "kernel/time/Kconfig"
……
config KTIME_SCALAR
    bool
    default y
```

Kconfig 的详细语法规则可以参看内核文档 Documentation/kbuild/kconfig-language.txt，下面对其简单介绍。

（1）菜单项。

config 关键字可以定义一个新的菜单项，比如：

```
config MODVERSIONS
    bool "Set version information on all module symbols"
    depends on MODULES
    help
      Usually, modules have to be recompiled whenever you switch to a new
      kernel.  ...
```

后面的几行定义了该菜单项的属性，包括类型、依赖关系、选择提示、帮助信息和缺省值等。

类型包括 bool、tristate、string、hex 和 int。bool 类型的只能选中或不选中，tristate 类型的菜单项多了编译成内核模块的选项。

依赖关系通过“depends on”或“requires”定义，指出此菜单项是否依赖于另外一个菜单项。

帮助信息需要使用“help”或“---help---”指出。

（2）菜单组织结构。

菜单选项通过两种方式组成树状结构。

使用关键字“menu”显式声明为菜单，比如：

```
menu "Bus options (PCI, PCMCIA, EISA, MCA, ISA)"

config PCI
……

endmenu
```

通过依赖关系确定菜单结构，比如：

```
config MODULES
    bool "Enable loadable module support"

config MODVERSIONS
    bool "Set version information on all module symbols"
    depends on MODULES

comment "module support disabled"
    depends on !MODULES
```

MODVERSIONS 菜单项依赖于 MODULES，所以它就是一个子菜单项。这要求菜单项和它的子菜单项同步显示或不显示。

（3）Kconfig 关键字。

Kconfig 文件描述了一系列的菜单选项，除帮助信息外，文件中的每一行都以一个关键字开始，主要有 config、menuconfig、choice/endchoice、comments、menu/endmenu、if/endif、source 等，它们都可以用于结束一个菜单项，只有前 5 个可以用在菜单项定义的开始。

3.7.2 利用 Kconfig 和 Makefile 寻找目标代码

以学习 Linux U 盘驱动的实现为例，因为 U 盘是一种 storage 设备，所以我们应该先进入 drivers/usb/storage/目录。但是该目录下的文件很多，究竟哪些文件才是我们需要关注的？

为了解决这个问题，有必要先去阅读 Kconfig 和 Makefile 文件。对于 Kconfig 文件，我们可以看到下面的选项。

```
34 config USB_STORAGE_DATAFAB
35      bool "Datafab Compact Flash Reader support (EXPERIMENTAL)"
36      depends on USB_STORAGE && EXPERIMENTAL
37      help
38        Support for certain Datafab CompactFlash readers.
39        Datafab has a web page at <http://www.datafabusa.com/>.
```

显然，这个选项和我们的目的没有关系。首先它专门针对 Datafab 公司的产品，其次虽然 CompactFlash reader 是一种 Flash 设备，但显然不是 U 盘。因为 drivers/usb/storage 目录下的代码是针对 usb mass storage 这一类设备，而不是针对某一种特定的设备。U 盘只是 usb mass storage 设备中的一种。再比如：

```
101 config USB_STORAGE_SDDR55
102     bool "SanDisk SDDR-55 SmartMedia support (EXPERIMENTAL)"
103     depends on USB_STORAGE && EXPERIMENTAL
104     help
105       Say Y here to include additional code to support the Sandisk SDDR-55
106       SmartMedia reader in the USB Mass Storage driver.
```

很显然，这个选项是有关 SanDisk 产品的，并且针对的是 SM 卡，同样不是 U 盘，所以我们也不需要去关注。

事实上，很容易确定，只有选项 CONFIG_USB_STORAGE 才是我们真正需要关注的。

```
9 config USB_STORAGE
```

```
10  tristate "USB Mass Storage support"
11      depends on USB && SCSI
12  ---help---
13    Say Y here if you want to connect USB mass storage devices to your
14    computer's USB port. This is the driver you need for USB
15    floppy drives, USB hard disks, USB tape drives, USB CD-ROMs,
16    USB flash devices, and memory sticks, along with
17    similar devices. This driver may also be used for some cameras
18    and card readers.
19
20    This option depends on 'SCSI' support being enabled, but you
21       probably also need 'SCSI device support: SCSI disk support'
22    (BLK_DEV_SD) for most USB storage devices.
23
24    To compile this driver as a module, choose M here: the
25    module will be called usb-storage.
```

接下来阅读 Makefile 文件。

```
0 #
1 # Makefile for the USB Mass Storage device drivers.
2 #
3 # 15 Aug 2000, Christoph Hellwig <hch@infradead.org>
4 # Rewritten to use lists instead of if-statements.
5 #
6
7 EXTRA_CFLAGS   := -Idrivers/scsi
8
9 obj-$(CONFIG_USB_STORAGE)  += usb-storage.o
10
11 usb-storage-obj-$(CONFIG_USB_STORAGE_DEBUG)   += debug.o
12 usb-storage-obj-$(CONFIG_USB_STORAGE_USBAT)   += shuttle_usbat.o
13 usb-storage-obj-$(CONFIG_USB_STORAGE_SDDR09)  += sddr09.o
14 usb-storage-obj-$(CONFIG_USB_STORAGE_SDDR55)  += sddr55.o
15 usb-storage-obj-$(CONFIG_USB_STORAGE_FREECOM) += freecom.o
16 usb-storage-obj-$(CONFIG_USB_STORAGE_DPCM)    += dpcm.o
17 usb-storage-obj-$(CONFIG_USB_STORAGE_ISD200)  += isd200.o
18 usb-storage-obj-$(CONFIG_USB_STORAGE_DATAFAB) += datafab.o
19 usb-storage-obj-$(CONFIG_USB_STORAGE_JUMPSHOT)+= jumpshot.o
20 usb-storage-obj-$(CONFIG_USB_STORAGE_ALAUDA)  += alauda.o
21 usb-storage-obj-$(CONFIG_USB_STORAGE_ONETOUCH)+= onetouch.o
22 usb-storage-obj-$(CONFIG_USB_STORAGE_KARMA)   += karma.o
23
24 usb-storage-objs :=   scsiglue.o protocol.o transport.o usb.o \
25          initializers.o $(usb-storage-obj-y)
26
27 ifneq ($(CONFIG_USB_LIBUSUAL),)
28  obj-$(CONFIG_USB)    += libusual.o
29 endif
```

前面通过 Kconfig 文件的分析，我们确定了只需要去关注 CONFIG_USB_STORAGE 选项。在 Makefile 文件里查找 CONFIG_USB_STORAGE，从第 9 行得知，该选项对应的模块为 usb-storage。

因为 Kconfig 文件里的其他选项我们都不需要关注，所以 Makefile 的 11～22 行可以忽略。第 24 行意味着我们只需要关注 scsiglue.c、protocol.c、transport.c、usb.c、initializers.c 以及它们同名的.h 头文件。

Kconfig 和 Makefile 很好地帮助我们定位到了所要关注的目标，就像我们到一个陌生的地方要随身携带地图，当我们学习 Linux 内核时，也要谨记寻求 Kconfig 和 Makefile 的帮助。

3.8 代码分析示例

学习内核，不免会经常对内核代码进行分析，如果抱着走马观花、得过且过的态度，结果极有可能就是一边看一边丢，没有多大的收获。学习内核应该遵循严谨的态度，去理解每一段代码的实现，多问多想多记。

下面以内核中 USB 子系统的部分代码为标本进行分析示例。

3.8.1 分析 README

内核中 USB 子系统的代码位于目录 drivers/usb，进入该目录，执行命令 ls，结果显示如下：

```
atm  class  core  gadget  host  image  misc  mon  serial  storage Kconfig
Makefile  README usb-skeleton.c
```

目录 drivers/usb 共包含 10 个子目录和 4 个文件，为了理解每个子目录的作用，有必要首先阅读 README 文件。

```
23 Here is a list of what each subdirectory here is, and what is contained in
24 them.
25
26 core/    - This is for the core USB host code, including the
27            usbfs files and the hub class driver ("khubd").
28
29 host/    - This is for USB host controller drivers.  This
30            includes UHCI, OHCI, EHCI, and others that might
31            be used with more specialized "embedded" systems.
32
33 gadget/      - This is for USB peripheral controller drivers and
34            the various gadget drivers which talk to them.
35
36
37 Individual USB driver directories.  A new driver should be added to the
38 first subdirectory in the list below that it fits into.
39
40 image/       - This is for still image drivers, like scanners or
41            digital cameras.
42 input/       - This is for any driver that uses the input subsystem,
43            like keyboard, mice, touchscreens, tablets, etc.
44 media/       - This is for multimedia drivers, like video cameras,
45            radios, and any other drivers that talk to the v4l
46            subsystem.
47 net/     - This is for network drivers.
48 serial/      - This is for USB to serial drivers.
49 storage/ - This is for USB mass-storage drivers.
50 class/       - This is for all USB device drivers that do not fit
51            into any of the above categories, and work for a range
52            of USB Class specified devices.
53 misc/    - This is for all USB device drivers that do not fit
54            into any of the above categories.
```

README 文件描述了各个子目录的作用。

（1）core。

针对部分核心的功能，内核开发者特意写了一些代码，用于为其他的设备驱动程序提供服务，比如申请内存，实现一些所有的设备都会需要的公共函数，并命名为 USB core。

（2）host。

早期的内核，其结构并不像现在这般富有层次感，几乎所有的文件都直接堆砌在 drivers/usb/目录下面，包括 usb core 和其他各种设备驱动程序的代码。

后来，drivers/usb/目录下面单独列出了一个 core 子目录，用于存放一些比较核心的代码，比如整个 USB 子系统的初始化、root hub 的初始化、host controller 的初始化代码。

再后来，针对 host controller，单独创建了子目录 host，存放与其相关的代码。原因是随着技术的发展，出现了多种 USB host controller，于是内核开发者把 host controller 有关的公共代码保留在 core 目录下，而其他各种 host controller 对应的特定代码则移到 host 目录下面，让相应的负责人去维护。

（3）gadget。

存放 USB gadget 的驱动程序，控制外围设备如何作为一个 USB 设备和主机通信。比如，嵌入式开发板通常会支持 SD 卡，使用 USB 连接线将开发板连接到 PC 时，通过 USB gadget 架构的驱动，可以将该 SD 卡模拟成 U 盘。

core、host 和 gadget 之外，其他几个目录分门别类存放了各种 USB 设备的驱动，比如，U 盘的驱动位于 storage 子目录，触摸屏和 USB 键盘鼠标的驱动位于 input 子目录。

因为我们的目的是研究内核对 USB 子系统的实现，而不是特定设备或 host controller 的驱动，所以通过对 README 文件的分析，应该进一步关注 core 子目录。

3.8.2 分析 Kconfig 和 Makefile

进入 drivers/usb/core 目录，执行命令 ls，结果显示如下：

```
Kconfig  Makefile  buffer.c  config.c  devices.c  devio.c  driver.c
endpoint.c  file.c  generic.c  hcd-pci.c  hcd.c  hcd.h  hub.c  hub.h
inode.c  message.c  notify.c  otg_whitelist.h  quirks.c  sysfs.c  urb.c
usb.c  usb.h
```

然后执行 wc 命令，如下所示。

```
# wc -l ./*
   148 buffer.c
   607 config.c
   706 devices.c
  1677 devio.c
  1569 driver.c
   357 endpoint.c
   248 file.c
   238 generic.c
  1759 hcd.c
   458 hcd.h
   433 hcd-pci.c
  3046 hub.c
   195 hub.h
   758 inode.c
   144 Kconfig
    21 Makefile
  1732 message.c
    68 notify.c
   112 otg_whitelist.h
```

```
   161 quirks.c
   710 sysfs.c
   589 urb.c
   984 usb.c
   160 usb.h
 16880 total
```

drivers/usb/core 目录共包括 24 个文件、16 880 行代码。依照 3.7 节的介绍，为了更准确地定位我们的目标，有必要对 Kconfig 和 Makefile 文件进行分析。

首先分析 Kconfig 文件，可以看到下面的选项。

```
15 config USB_DEVICEFS
16       bool "USB device filesystem"
17       depends on USB
18       ---help---
19         If you say Y here (and to "/proc file system support" in the "File
20         systems" section, above), you will get a file /proc/bus/usb/devices
21         which lists the devices currently connected to your USB bus or
22         busses, and for every connected device a file named
23         "/proc/bus/usb/xxx/yyy", where xxx is the bus number and yyy the
24         device number; the latter files can be used by user space programs
25         to talk directly to the device. These files are "virtual", meaning
26         they are generated on the fly and not stored on the hard drive.
27
28         You may need to mount the usbfs file system to see the files, use
29         mount -t usbfs none /proc/bus/usb
30
31         For the format of the various /proc/bus/usb/ files, please read
32         <file:Documentation/usb/proc_usb_info.txt>.
33
34         Usbfs files can't handle Access Control Lists (ACL), which are the
35         default way to grant access to USB devices for untrusted users of a
36         desktop system. The usbfs functionality is replaced by real
37         device-nodes managed by udev. These nodes live in /dev/bus/usb and
38         are used by libusb.
```

选项 USB_DEVICEFS 与 usbfs 文件系统有关。usbfs 文件系统挂载在/proc/bus/usb 目录，显示了当前连接的所有 USB 设备及总线的各种信息，每个连接的 USB 设备在其中都会有一个对应的文件进行描述。比如文件/proc/bus/usb/xxx/yyy，xxx 表示总线的序号，yyy 表示设备所在总线的地址。

因为 usbfs 文件系统并不属于 USB 子系统实现的核心部分，与之相关的代码我们可以不必关注。针对 Kconfig 文件其余选项的分析类似于 USB_DEVICEFS。

然后分析 Makefile 文件。

```
5 usbcore-objs   := usb.o hub.o hcd.o urb.o message.o driver.o \
6                   config.o file.o buffer.o sysfs.o endpoint.o \
7                   devio.o notify.o generic.o quirks.o
8
9 ifeq ($(CONFIG_PCI),y)
10       usbcore-objs    += hcd-pci.o
11 endif
12
13 ifeq ($(CONFIG_USB_DEVICEFS),y)
14       usbcore-objs    += inode.o devices.o
15 endif
16
```

```
17 obj-$(CONFIG_USB)      += usbcore.o
18
19 ifeq ($(CONFIG_USB_DEBUG),y)
20 EXTRA_CFLAGS += -DDEBUG
21 endif
```

因为选项 USB_DEVICEFS 和选项 CONFIG_PCI 不需要关注，所以从第 9 行和第 13 行可知，inode.c、device.c 和 hcd-pci.c 文件都可以忽略。

结合第 5 行和第 17 行可知，为了理解内核对 USB 子系统的实现，我们需要研究 buffer.c、config.c、driver.c、endpoint.c、file.c、generic.c、hcd.c hcd.h、hub.c、message.c、notify.c、otg_whitelist.h、quirks.c、sysfs.c、urb.c 和 usb.c 文件。

3.8.3 寻找初始化函数

对 Kconfig 和 Makefile 文件的分析，帮助我们在庞大复杂的内核代码中定位以及缩小了目标代码的范围。

现在，为了研究内核对 USB 子系统的实现，我们还需要在目标代码中找到一个突破口，即 USB 子系统的初始化代码。

针对某个子系统或某个驱动，内核使用 subsys_initcall 或 module_init 宏指定初始化函数。在 drivers/usb/core/usb.c 文件中，我们发现下面的代码。

```
940 subsys_initcall(usb_init);
941 module_exit(usb_exit);
```

subsys_initcall 可以理解为 module_init，只不过因为该部分代码比较核心，开发者们把它看做一个子系统，而不仅仅是一个模块。Linux 中，类似这样一个类别的设备驱动被归结为一个子系统，比如 PCI 子系统，比如 SCSI 子系统。基本上，drivers/目录下面第一层的每个目录都代表一个子系统，因为它们分别代表了一类设备。

subsys_initcall(usb_init)表示，usb_init 函数是 USB 子系统的初始化函数，而 module_exit 则表示，usb_exit 函数是 USB 子系统结束时的清理函数。于是为了研究 USB 子系统在内核中的实现，我们需要从 usb_init 函数开始分析。

```
865 static int __init usb_init(void)
866 {
867     int retval;
868     if (nousb) {
869         pr_info("%s: USB support disabled\n", usbcore_name);
870         return 0;
871     }
872
873     retval = ksuspend_usb_init();
874     if (retval)
875         goto out;
876     retval = bus_register(&usb_bus_type);
877     if (retval)
878         goto bus_register_failed;
879     retval = usb_host_init();
880     if (retval)
881         goto host_init_failed;
882     retval = usb_major_init();
883     if (retval)
884         goto major_init_failed;
885     retval = usb_register(&usbfs_driver);
886     if (retval)
887         goto driver_register_failed;
888     retval = usb_devio_init();
889     if (retval)
890         goto usb_devio_init_failed;
891     retval = usbfs_init();
892     if (retval)
```

```
893         goto fs_init_failed;
894     retval = usb_hub_init();
895     if (retval)
896         goto hub_init_failed;
897     retval = usb_register_device_driver(&usb_generic_driver, THIS_MODULE);
898     if (!retval)
899         goto out;
900
901     usb_hub_cleanup();
902 hub_init_failed:
903     usbfs_cleanup();
904 fs_init_failed:
905     usb_devio_cleanup();
906 usb_devio_init_failed:
907     usb_deregister(&usbfs_driver);
908 driver_register_failed:
909     usb_major_cleanup();
910 major_init_failed:
911     usb_host_cleanup();
912 host_init_failed:
913     bus_unregister(&usb_bus_type);
914 bus_register_failed:
915     ksuspend_usb_cleanup();
916 out:
917     return retval;
918 }
```

（1）__init 标记。

关于 usb_init，第一个问题是，第 865 行的__init 标记具有什么意义？

前面讲解 GCC 扩展的特殊属性 section 时提到，__init 修饰的所有代码都会被放在.init.text 节，初始化结束后就可以释放这部分内存。问题可以到此为止，也可以更深入，即内核是如何调用到__init 所修饰的这些初始化函数？

为了回答这个问题，需要回顾 subsys_initcall 宏，它在 include/linux/init.h 文件中定义为：

```
125 #define subsys_initcall(fn)     __define_initcall("4",fn,4)
```

出现了一个新的宏__define_initcall，它用来将指定的函数指针 fn 存放到.initcall.init 节。对于 subsys_initcall 宏，则表示把 fn 存放到.initcall.init 的子节.initcall4.init。

为了理解.initcall.init、.init.text 和.initcall4.init 这样的符号，我们还需要了解一些内核可执行文件相关的概念。

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。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。

打开 arch/i386/kernel/vmlinux.lds 文件，要完全理解这个文件并不容易，不过我们需要做的只是搜索 initcall.init，然后便会看到：

```
__inicall_start = .;
.initcall.init : AT(ADDR(.initcall.init) - 0xC0000000) {
*(.initcall1.init)
*(.initcall2.init)
*(.initcall3.init)
*(.initcall4.init)
*(.initcall5.init)
*(.initcall6.init)
*(.initcall7.init)
}
__initcall_end = .;
```

其中的__initcall_start 指向.initcall.init 节的开始，__initcall_end 指向.initcall.init 节的结尾。而.initcall.init 节又被分为了 7 个子节，分别是：

```
.initcall1.init
.initcall2.init
.initcall3.init
.initcall4.init
.initcall5.init
.initcall6.init
.initcall7.init
```

subsys_initcall 宏即是将指定的函数指针放在了.initcall4.init 子节。其他类似的宏，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。

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。__init 修饰的初始化函数在内核初始化过程中调用的顺序和.initcall.init 节里函数指针的顺序有关，不同的初始化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。

因为属于内核的初始化，所以实际执行函数调用的位置在/init/main.c 文件，其中的 do_initcalls 函数会直接用到这里的__initcall_start、__initcall_end 并进行判断。

（2）模块参数。

关于 usb_init 函数，第二个问题是，第 868 行的 nousb 表示什么？

nousb 在 drivers/usb/core/usb.c 文件中定义为：

```
static int nousb;    /* Disable USB when built into kernel image */
module_param_named(autosuspend, usb_autosuspend_delay, int, 0644);
MODULE_PARM_DESC(autosuspend, "default autosuspend delay");
```

从中可知 nousb 是个模块参数，用于在内核启动的时候禁止 USB 子系统。

关于模块参数，我们都知道可以在加载模块的时候可以指定，但是如何在内核启动的时候指定？

打开系统的 grub 文件，然后找到 kernel 行，比如：

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

其中的 root、splash、vga 等都表示内核参数。当某一模块被编译进内核的时候，它的模块参数便需要在 kernel 行来指定，格式为“模块名.参数=值”，比如：

```
modprobe usbcore autosuspend=2
```

对应到 kernel 行，即为：

```
usbcore.autosuspend=2
```

通过命令“modinfo -p ${modulename}”可以得知一个模块有哪些参数可以使用。同时，对于已经加载到内核里的模块，它们的模块参数会列举在/sys/module/$ {modulename}/parameters/目录下面，可以使用“echo -n ${value} > /sys/module/$ {modulename}/parameters/${parm}”这样的命令去修改。

（3）可变参数宏。

关于 usb_init 函数，第 3 个问题是 pr_info 如何实现与使用？

前面讲解 GCC 扩展的可变参数宏时，已经对其进行分析，在此不再赘述。

关于 usb_init 函数，上面的 3 个问题之外，余下的代码分别完成 usb 各部分的初始化，接下来就需要围绕它们分别进行深入分析。因为本节只是进行简单的代码分析示例，具体的深入分析就留给读者来完成。

联系方式

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

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