FreeBSD 系统结构手册

FreeBSD 系统结构手册

修订：43126

2000 年 8 月 由 .

2005 年 12 月 由 .

摘要

欢迎您阅读《FreeBSD系统结构手册》。 这本手册还在不断由许多人继续书写。 许多章节还是空白，有的章节亟待更新。 如果您对这个项目感兴趣并愿意有所贡献，请发信给 FreeBSD 文档计划邮件列表。

本文档的最新英文原始版本可从 FreeBSD Web 站点 获得， 由 FreeBSD 中文计划 维护的最新译本可以在 FreeBSD 中文计划 快照 Web 站点 和 FreeBSD 中文计划 文档快照 处获得， 这一译本会不断向主站同步。 此外， 您也可以从 FreeBSD FTP 服务器 或众多的 镜像站点 得到这份文档的各种其他格式以及压缩形式的版本。

1. 再分发源代码 (SGML DocBook) 必须不加修改的保留上述版权告示、 本条件清单和下述弃权书作为该文件的最先若干行。

2. 再分发编译的形式 (转换为其它DTD、 PDF、 PostScript、 RTF 或其它形式), 必须将上述版权告示、本条件清单和下述弃权书复制到与分发品一同提供的文件, 以及其它材料中。

> **重要**
>
> 本文档由 FREEBSD DOCUMENTATION PROJECT “按现状条件” 提供, 并在此明示不提供任何明示或暗示的保障, 包括但不限于对商业适销性、 对特定目的的适用性的暗示保障。 任何情况下, FREEBSD DOCUMENTATION PROJECT 均不对任何直接、 间接、 偶然、 特殊、 惩罚性的, 或必然的损失 (包括但不限于替代商品或服务的采购、 使用、 数据或利益的损失或营业中断) 负责, 无论是如何导致的并以任何有责任逻辑的, 无论是否是在本文档使用以外以任何方式产生的契约、 严格责任或是民事侵权行为(包括疏忽或其它)中的, 即使已被告知发生该损失的可能性。

Redistribution and use in source (SGML DocBook) and "compiled" forms (SGML, HTML, PDF, PostScript, RTF and so forth) with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code (SGML DocBook) must retain the above copyright notice, this list of conditions and the following disclaimer as the first lines of this file unmodified.

2. Redistributions in compiled form (transformed to other DTDs, converted to PDF, PostScript, RTF and other formats) must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

> **重要**
>
> THIS DOCUMENTATION IS PROVIDED BY THE FREEBSD DOCUMENTATION PROJECT "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FREEBSD DOCUMENTATION PROJECT BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

> **重要**
>
> 从 FreeBSD 和 FreeBSD 中文化项目官方网站获取的本文档 PostScript# 和 PDF 格式文件中内嵌的中文字体，由文鼎科技开发股份有限公司按《文鼎公众授权书》授权提供。

文鼎公众授权书

前言

绝大多数软件的授权书是用来剥夺您共享和修改软件的自由。相反， 如果您将我们所授与您的一切权利授与您的接受者， 并且确保他们能得到本软件的修改物， 文鼎公众授权书特别许可并鼓励您使用本软件。

法律性条款

0. 定义

本授权书中，「字型」是指「文鼎PL细上海宋」、「文鼎PL中楷」（BIG-5码） 和「文鼎PL简报宋」、「文鼎PL简中楷」（GB码）等TrueType字型， 以及经修改而产生的该等字型的衍生物，包括修改字型样式、 更动编码顺序、 转换格式、 变更字型名称， 或者在 glyph table 中增减一些文字。

「PL」是指「Public License」（公众授权）。

「版权人」是指在字型版权声明中被称为版权所有者之人。

「您」是指被授权者或复制、再散布、修改字型的人。

「自由使用」是指您在所接受的相同条件下， 有复制、 修改字型及散布字型复制品的自由，而非指价格。 如果您愿意， 可以对此项服务收取费用。

1. 复制及分发

如果在所有复制品维持本授权书文件（ARPHICPL.TXT）不变， 您可以在任何媒体无限制复制及分发本字型之复制品。

2．　修改

如果满足以下条件，　您可以用任何方式修改本字型的复制品，　包括修改字型样式、　更换编码顺序、　转换格式、　变更字型名称，　或者在 glyph table 中增减一些文字，　并且依据上述第一条的规定复制及分发该修改物。

a)您必须在每个修改过的文件中插入明显的声明，　表示您如何及何时修改了该文件。

b)您必须允许所有第三方在本授权书条款下，　得以「自由使用」该修改物的全部，　例如通过提供指定地点的复制路径，或在通常作为软件交换的媒体上发布该修改物。

c)如果修改过的字型在执行时以交谈方式读取命令，　您必须使它在开始进入一般的交谈使用方式时打印或显示声明，　包括适当的版权声明、　无担保声明　（或者其他您提供担保的声明），　以及使用者可以依据这些条件再分发本字型的声明，　并且告知使用者如何看到本授权书的复本。

这些要求适用于整个修改过的作品。　如果作品的可辨识部分并非衍生自本字型，　且可以合理地认其为独立而个别的作品，　则当您将它当作个别作品分发时，　本授权书及其条款将不适用于该部分。　因此，　仅将非基于本字型之其他作品和本字型一起放入贮存或分发媒体的同一卷上，　并不会使该其他作品纳入本授权书的适用范围。

3．　解除条件

除非依照本授权书的规定，　否则您不能复制、　修改、　再授权或分发本字型。　任何用其他方式复制、　修改、　再授权或分发本字型的意图，　将使本授权书所授与您的权利自动失效。　不过，从您那依据本授权书取得复制品和权利的人，　只要他们完全遵行条款，其权利将维持有效。

4．　同意

因您未签订本授权书，　所以没必要遵行它。　然而，　没有任何其他文件授与您复制、　修改、再授权或分发本字型之权利，　如果您不同意本授权书，　这些行为是法律所禁止的。　因此一旦复制、　修改、　再授权或分发本字型，　就表示您接受本授权书及其一切条款和条件。

5．　自动取得授权

每当您再散布本字型时，　收受者便自动从原始授权者处取得依据这些条款和条件复制、　分发或修改本字型的授权。　您不可对在此所授与收受者之权利实行强加其他限制，　您也没有使第三方遵行本授权书的义务。

6．　抵触

若因法院判决、　违反专利宣称或任何其他原因（不限于专利纠纷）的结果，　使强加于您的条件（不论是法院命令、协议或其他）和授权书的条件有所冲突时，　也不能让您违背授权书的条件。　如果您不能同时满足本授权书所定的义务及其他相关义务来分发时，　那麼您根本不可分发本字型。　例如，　如果专利授权不许所有直接或间接从您那里取得复制品者，　在免授权费的情况下再分发本字型时，　唯一能同时满足双方要求的方法是完全避免分发本字型。

如果本条款之任何部分在特定情况下无效或无法实施，　条款的其他部分将继续适用，　并作为整体适用于其他情况。

7． 无担保

由于本字型是免费授权， 在法律允许的范围内， 本字型将不附带任何担保。 除非另有书面说明， 版权人或其他提供本字型者「同样」不提供任何种类的担保， 不论是明示还是暗示， 包括但不限于销售性和适于特定目的的暗示担保。 有关本字型品质和执行的一切危险均由您承担。如果本字型出现瑕疵， 您应承担一切所需服务、修复和改正的费用。

8． 损害赔偿请求权的放弃

除非法律规定或以书面形式同意， 在任何情况下， 任何版权人或以上被允许得以复制、修改或再分发本字型者， 均不对您因使用或不能使用本字型（包括但不限于替代商品或服务的采购； 使用、 数据或利益之损失； 或者营业妨碍） 所引致的任何直接、 间接、 必然、 偶然、特殊或惩罚性损失负责， 即使版权人或其他人已知该损失的可能性时亦不例外。

目录

插图清单

表格清单

范例清单

部分 I. 内核

目 录

第 1 章 引导过程与内核初始化

供稿：Sergey Lyubka.
翻译：.

1.1. 概述

这一章是对引导过程和系统初始化过程的总览。这些过程始于BIOS(固件)POST，直到第一个用户进程建立。由于系统启动的最初步骤是与硬件结构相关的、是紧配合的， 这里用IA-32(Intel Architecture 32bit)结构作为例子。

1.2. 总览

一台运行FreeBSD的计算机有多种引导方法。这里讨论其中最通常的方法， 也就是从安装了操作系统的硬盘上引导。引导过程分几步完成：

- BIOS POST
- boot0阶段
- boot2阶段
- loader阶段
- 内核初始化

boot0和boot2阶段在手册 boot(8)中被称为bootstrap stages 1 and 2， 是FreeBSD的3阶段引导过程的开始。在每一阶段都有各种各样的信息显示在屏幕上， 你可以参考下表识别出这些步骤。请注意实际的显示内容可能随机器的不同而有一些区别：

视不同机器而定	BIOS(固件)消息
F1 FreeBSD F2 BSD F5 Disk 2	boot0
>>FreeBSD/i386 BOOT Default: 1:ad(1,a)/boot/loader boot:	boot2[a]
BTX loader 1.0 BTX version is 1.01	loader

BIOS drive A: is disk0 BIOS drive C: is disk1 BIOS 639kB/64512kB available memory FreeBSD/i386 bootstrap loader, Revision 0.8 Console internal video/keyboard (jkh@bento.freebsd.org, Mon Nov 20 11:41:23 - GMT 2000) /kernel text=0x1234 data=0x2345 - syms=[0x4+0x3456] Hit [Enter] to boot immediately, or any other - key for command prompt Booting [kernel] in 9 seconds..._	
Copyright (c) 1992-2002 The FreeBSD Project. Copyright (c) 1979, 1980, 1983, 1986, 1988, - 1989, 1991, 1992, 1993, 1994 The Regents of the University of - California. All rights reserved. FreeBSD 4.6-RC #0: Sat May 4 22:49:02 GMT - 2002 devnull@kukas:/usr/obj/usr/src/sys/DEVNULL Timecounter -"i8254" frequency 1193182 Hz	内核

[a] 这种提示仅在boot0阶段用户选择操作系统后 仍按住键盘上某一键时才出现。

1.3. BIOS POST

当PC加电后，处理器的寄存器被设为某些特定值。在这些寄存器中， 指令指针寄存器被设为32位值0xfffffff0。 指令指针寄存器指向处理器将要执行的指令代码。cr1， 一个32位控制寄存器，在刚启动时值被设为0。cr1的PE(Protected Enabled， 保护模式使能)位用来指示处理器是处于保护模式还是实地址模式。 由于启动时该位被清位，处理器在实地址模式中引导。在实地址模式中， 线性地址与物理地址是等同的。

值0xfffffff0略小于4G,因此计算机没有4G字节物理内存， 这就不会是一个有效的内存地址。计算机硬件将这个地址转指向BIOS存储块。

BIOS表示Basic Input Output System (基本输入输出系统)。在主板上，它被固化在一个相对容量较小的 只读存储器(Read-Only Memory，ROM)。BIOS包含各种各样为主板硬件 定制的底层例程。就这样，处理器首先指向常驻BIOS存储器的地址0xfffffff0。通常这个位置包含一条跳转指令，指向BIOS的POST例程。

POST表示Power On Self Test(加电自检)。 这套程序包括内存检查，系统总线检查和其它底层工具， 从而使得CPU能够初始化整台计算机。这一阶段中有一个重要步骤， 就是确定引导设备。现在所有的BIOS都允许手工选择引导设备。 你可以从软盘、光盘驱动器、硬盘等设备引导。

POST的最后一步是执行INT 0x19指令。 这个指令从引导设备第一个扇区读取512字节，装入地址0x7c00。 第一个扇区的说法最早起源于硬盘的结构， 硬盘面被分为

若干圆柱形轨道。给轨道编号，同时又将轨道分为 一定数目(通常是64)的扇形。0号轨道是硬盘的最外圈，1号扇区， 第一个扇区(轨道、柱面都从0开始编号，而扇区从1开始编号) 有着特殊的作用，它又被称为主引导记录(Master Boot Record，MBR)。 第一轨剩余的扇区常常不使用[1]。

1.4. boot0阶段

让我们看一下文件/boot/boot0。 这是一个仅512字节的小文件。如果在FreeBSD安装过程中选择 “bootmanager” ，这个文件中的内容将被写入硬盘MBR

如前所述， INT 0x19 指令装载 MBR， 也就是 boot0 的内容至内存地址 0x7c00。再看文件 sys/boot/i386/boot0/boot0.S， 可以猜想这里面发生了什么 – 这是引导管理器， 一段由 Robert Nordier书写的令人起敬的程序片段。

MBR里，也就是boot0里， 从偏移量0x1be开始有一个特殊的结构，称为 分区表。其中有4条记录 (称为分区记录)，每条记录16字节。 分区记录表示硬盘如何被划分，在FreeBSD的术语中， 这被称为slice(d)。16字节中有一个标志字节决定这个分区是否可引导。 有仅只能有一个分区可设定这一标志。否则， boot0的代码将拒绝继续执行。

一个分区记录有如下域：

- 1字节 文件系统类型
- 1字节 可引导标志
- 6字节 CHS格式描述符
- 8字节 LBA格式描述符

一个分区记录描述符包含某一分区在硬盘上的确切位置信息。 LBA和CHS两种描述符指示相同的信息，但是指示方式有所不同：LBA (逻辑块寻址，Logical Block Addressing)指示分区的起始扇区和分区长度， 而CHS(柱面 磁头 扇区)指示首扇区和末扇区

引导管理器扫描分区表，并在屏幕上显示菜单，以便用户可以 选择用于引导的磁盘和分区。在键盘上按下相应的键后， boot0进行如下动作：

- 标记选中的分区为可引导，清除以前的可引导标志
- 记住本次选择的分区以备下次引导时作为缺省项
- 装载选中分区的第一个扇区，并跳转执行之

[1]有些工具如disklabel(8) 会使用这一区域存储信息，主要是在第二扇区里。

什么数据会存在于一个可引导扇区(这里指FreeBSD扇区)的第一扇区里呢? 正如你已经猜到的,那就是boot2。

1.5. boot2阶段

也许你想知道,为什么boot2是在 boot0之后,而不是在boot1之后。事实上, 也有一个512字节的文件boot1存放在目录 /boot里,那是用来从一张软盘引导系统的。从软盘引导时,boot1起着 boot0对硬盘引导相同的作用:它找到 boot2并运行之。

你可能已经看到有一文件/boot/mbr。 这是boot0的简化版本。 mbr中的代码不会显示菜单让用户选择, 而只是简单的引导被标志的分区。

实现boot2的代码存放在目录 sys/boot/i386/boot2/里,对应的可执行文件在 /boot里。在/boot里的文件 boot0和boot2不会在引导过程中使用, 只有boot0cfg这样的工具才会使用它们。 boot0的内容应在MBR中才能生效。 boot2位于可引导的FreeBSD分区的开始。 这些位置不受文件系统控制,所以它们不可用ls 之类的命令查看。

boot2的主要任务是装载文件 /boot/loader,那是引导过程的第三阶段。 在boot2中的代码不能使用诸如 open()和read() 之类的例程函数,因为内核还没有被加载。而应当扫描硬盘, 读取文件系统结构,找到文件/boot/loader, 用BIOS的功能将它读入内存,然后从其入口点开始执行之。

除此之外,boot2还可提示用户进行选择, loader可以从其它磁盘、系统单元、分区装载。

boot2 的二进制代码用特殊的方式产生:

```
sys/boot/i386/boot2/Makefile
boot2: boot2.ldr boot2.bin ${BTX}/btx/btx
 btxld --v --E ${ORG2} --f bin --b ${BTX}/btx/btx --l boot2.ldr \
  -o boot2.ld --P 1 boot2.bin
```

这个Makefile片断表明btxld(8)被用来链接二进制代码。 BTX表示引导扩展器(BooT eXtender)是给程序(称为客户(client) 提供保护模式环境、并与客户程序相链接的一段代码。所以 boot2是一个BTX客户, 使用BTX提供的服务。

工具btxld是链接器, 它将两个二进制代码链接在一起。btxld(8)和ld(1) 的区别是ld通常将两个目标文件 链接成一个动态链接库或可执行文件,而btxld 则将一个目标文件与BTX链接起来,产生适合于放在分区首部的二进制代码, 以实现系统引导。

boot0执行跳转至BTX的入口点。 然后,BTX将处理器切换至保护模式,并准备一个简单的环境, 然后调用客户。这个环境包括:

- 虚拟8086模式。这意味着BTX是虚拟8086的监视程序。 实模式指令，如pushf，popf，cli，sti，if，均可被客户调用。

- 建立中断描述符表(Interrupt Descriptor Table，IDT)， 使得所有的硬件中断可被缺省的BIOS程序处理。 建立中断0x30，这是系统调用关口。

- 两个系统调用exec和 exit的定义如下：

```
sys/boot/i386/btx/lib/btxsys.s:
  .set INT_SYS,0x30  # ###
#
# System call: exit
#
__exit:  xorl %eax,%eax   # BTX####0x0
  int $INT_SYS   #
#
# System call: exec
#
__exec:  movl $0x1,%eax   # BTX####0x1
  int $INT_SYS   #
```

BTX建立全局描述符表(Global Descriptor Table，GDT)：

```
sys/boot/i386/btx/btx/btx.s:
gdt:  .word 0x0,0x0,0x0,0x0  # #####
  .word 0xffff,0x0,0x9a00,0xcf # SEL_SCODE
  .word 0xffff,0x0,0x9200,0xcf # SEL_SDATA
  .word 0xffff,0x0,0x9a00,0x0 # SEL_RCODE
  .word 0xffff,0x0,0x9200,0x0 # SEL_RDATA
  .word 0xffff,MEM_USR,0xfa00,0xcf# SEL_UCODE
  .word 0xffff,MEM_USR,0xf200,0xcf# SEL_UDATA
  .word _TSSLM,MEM_TSS,0x8900,0x0 # SEL_TSS
```

客户的代码和数据始于地址MEM_USR(0xa000)，选择符(selector) SEL_UCODE指向客户的数据段。选择符 SEL_UCODE 拥有第3级描述符权限 (Descriptor Privilege Level， DPL)，这是最低级权限。但是 INT 0x30 指令的处理程序存储于另一个段里， 这个段的选择符SEL_SCODE (supervisor code)由有着管理级权限。 正如代码建立IDT(中断描述符表)时进行的操作那样：

```
  mov $SEL_SCODE,%dh  # ####
init.2:  shr %bx    # #########
  jnc init.3    # #
  mov %ax,(%di)   # #########
  mov %dh,0x2(%di)  # #########
  mov %dl,0x5(%di)  # ## P:DPL:type
  add $0x4,%ax    # #########
```

所以，当客户调用 __exec()时，代码将被以最高权限执行。 这使得内核可以修改保护模式数据结构，如分页表(page tables)、全局描述符表(GDT)、 中断描述符表(IDT)等。

boot2 定义了一个重要的数据结构： struct bootinfo。这个结构由 boot2 初始化，然后被转送到loader，之后又被转入内核。 这个结构的部分项目由boot2设定，其余的由loader设定。 这个结构中的信息包括内核文件名、BIOS提供的硬盘柱面/磁头/扇区数目信息、 BIOS提供的引导设备的驱动器编号，可用的物理内存大小，envp 指针(环境指针)等。定义如下：

/usr/include/machine/bootinfo.h

```
struct bootinfo {
 u_int32_t bi_version;
 u_int32_t bi_kernelname;  /* ####### * */
 u_int32_t bi_nfs_diskless; /* struct nfs_diskless * */
  /* ###### */
#define bi_endcommon bi_n_bios_used
 u_int32_t bi_n_bios_used;
 u_int32_t bi_bios_geom[N_BIOS_GEOM];
 u_int32_t bi_size;
 u_int8_t bi_memsizes_valid;
 u_int8_t bi_bios_dev;  /* #####BIOS#### */
 u_int8_t bi_pad[2];
 u_int32_t bi_basemem;
 u_int32_t bi_extmem;
 u_int32_t bi_symtab;  /* struct symtab * */
 u_int32_t bi_esymtab;  /* struct symtab * */
  /* #######bootloader## */
 u_int32_t bi_kernend;  /* ###### */
 u_int32_t bi_envp;  /* ## */
 u_int32_t bi_modulep;  /* ###### */
};
```

boot2 进入一个循环等待用户输入，然后调用 load()。如果用户不做任何输入，循环将在一段时间后结束， load() 将会装载缺省文件(/boot/loader)。 函数 ino_t lookup(char *filename)和 int xfsread(ino_t inode, void *buf, size_t nbyte) 用来将文件内容读入内存。/boot/loader是一个ELF格式二进制文件， 不过它的头部被换成了a.out格式中的struct exec结构。 load()扫描loader的ELF头部，装载/boot/loader 至内存，然后跳转至入口执行之：

sys/boot/i386/boot2/boot2.c:

```
    __exec((caddr_t)addr, RB_BOOTINFO -| (opts & RBX_MASK),
    MAKEBOOTDEV(dev_maj[dsk.type], 0, dsk.slice, dsk.unit, dsk.part),
    0, 0, 0, VTOP(&bootinfo));
```

1.6. loader阶段

loader也是一个 BTX 客户，在这里不作详述。 已有一部内容全面的手册 loader(8) ，由Mike Smith书写。 比loader更底层的BTX的机理已经在前面讨论过。

loader 的主要任务是引导内核。当内核被装入内存后，即被loader调用：

sys/boot/common/boot.c:
```
-/* #loader#########exec## */
module_formats[km->m_loader]->l_exec(km);
```

1.7. 内核初始化

让我们来看一下链接内核的命令。 这能帮助我们了解 loader 传递给内核的准确位置。 这个位置就是内核真实的入口点。

sys/conf/Makefile.i386:
```
ld --elf --Bdynamic --T -/usr/src/sys/conf/ldscript.i386  --export-dynamic \
-dynamic-linker -/red/herring --o kernel --X locore.o \
<lots of kernel -.o files>
```

在这一行中有一些有趣的东西。首先，内核是一个ELF动态链接二进制文件， 可是动态链接器却是/red/herring，一个莫须有的文件。 其次，看一下文件sys/conf/ldscript.i386， 可以对理解编译内核时ld的选项有一些启发。 阅读最前几行，字符串

sys/conf/ldscript.i386:
```
ENTRY(btext)
```

表示内核的入口点是符号 `btext'。这个符号在locore.s 中定义：

sys/i386/i386/locore.s:
```
 .text
/**********************************************************************
 *
 * This is where the bootblocks start us, set the ball rolling...
 * ##
 */
NON_GPROF_ENTRY(btext)
```

首先将寄存器EFLAGS设为一个预定义的值0x00000002， 然后初始化所有段寄存器：

sys/i386/i386/locore.s
```
/* ####BIOS###EFLAGS# */
 pushl $PSL_KERNEL
 popfl

/*
 * ####BIOS###%fs#%gs###########%cs#%ds#%es#%ss#
 */
 mov %ds, %ax
 mov %ax, %fs
 mov %ax, %gs
```

btext调用例程recover_bootinfo()，identify_cpu()，create_pagetables()。 这些例程也定在locore.s之中。这些例程的功能如下：

recover_bootinfo	这个例程分析由引导程序传送给内核的参数。引导内核有3种方式：由loader引导(如前所述)，由老式磁盘引导块引导,无盘引导方式。 这个函数决定引导方式，并将结构struct bootinfo 存储至内核内存。
identify_cpu	这个函数侦测CPU类型，将结果存放在变量 _cpu中。
create_pagetables	这个函数为分页表在内核内存空间顶部分配一块空间，并填写一定内容

下一步是开启VME(如果CPU有这个功能)：

```
testl $CPUID_VME, R(_cpu_feature)
jz 1f
movl %cr4, %eax
orl $CR4_VME, %eax
movl %eax, %cr4
```

然后，启动分页模式：

```
/* Now enable paging */
 movl R(_IdlePTD), %eax
 movl %eax,%cr3   /* load ptd addr into mmu */
 movl %cr0,%eax   /* get control word */
 orl $CR0_PE|CR0_PG,%eax  /* enable paging */
 movl %eax,%cr0   /* and let's page NOW! */
```

由于分页模式已经启动，原先的实地址寻址方式随即失效。 随后三行代码用来跳转至虚拟地址：

```
 pushl $begin    /* jump to high virtualized address */
 ret

/* #####KERNBASE################### */
begin:
```

函数init386()被调用；随参数传递的是一个指针， 指向第一个空闲物理页。随后执行mi_startup()。 init386是一个与硬件系统相关的初始化函数， mi_startup()是个与硬件系统无关的函数（前缀'mi_'表示Machine Independent，不依赖于机器）。 内核不再从mi_startup()里返回； 调用这个函数后，内核完成引导：

```
sys/i386/i386/locore.s:
 movl physfree, %esi
 pushl %esi  /* ##init386()###### */
 call _init386 /* ##386######UNIX## */
 call _mi_startup /* ################ */
 hlt  /* ######## */
```

1.7.1. init386()

init386()定义在 sys/i386/i386/machdep.c中， 它针对Intel 386芯片进行低级初始化。loader已将CPU切换至保护模式。 loader已经建立了最早的任务。

> **译者注**
>
> 每个"任务"都是与其它“任务”相对独立的执行环境。 任务之间可以分时切换，这为并发进程/线程的实现提供了必要基础。 对于Intel 80x86任务的描述，详见Intel公司关于80386 CPU及后续产品的资料， 或者在清华大学图书馆 馆藏记录中用"80386"作为关键词所查找到的系统结构方面的书目。

在这个任务中，内核将继续工作。在讨论其代码前， 我将处理器对保护模式必须完成的一系列准备工作一并列出：

- 初始化内核的可调整参数，这些参数由引导程序传来
- 准备GDT(全局描述符表)
- 准备IDT(中断描述符表)
- 初始化系统控制台
- 初始化DDB(内核的点调试器)，如果它被编译进内核的话
- 初始化TSS(任务状态段)
- 准备LDT(局部描述符表)
- 建立proc0(0号进程，即内核的进程)的pcb(进程控制块)

init386()首先初始化内核的可调整参数， 这些参数由引导程序传来。先设置环境指针(environment pointer, envp)调用， 再调用init_param1()。 envp指针已由loader存放在结构bootinfo中：

```
sys/i386/i386/machdep.c:
  kern_envp = (caddr_t)bootinfo.bi_envp + KERNBASE;

 /* #########,#hz# */
 init_param1();
```

init_param1()定义在 sys/kern/subr_param.c之中。 这个文件里有一些sysctl项，还有两个函数， init_param1()和init_param2()。 这两个函数从init386()中调用：

```
sys/kern/subr_param.c
 hz = HZ;
 TUNABLE_INT_FETCH("kern.hz", &hz);
```

TUNABLE_<typename>_FETCH用来获取环境变量的值:

```
/usr/src/sys/sys/kernel.h
#define TUNABLE_INT_FETCH(path, var) getenv_int((path), (var))
```

Sysctlkern.hz是系统时钟频率。同时， 这些sysctl项被init_param1()设定: kern.maxswzone, kern.maxbcache, kern.maxtsiz, kern.dfldsiz, kern.maxdsiz, kern.dflssiz, kern.maxssiz, kern.sgrowsiz。

然后init386() 准备全局描述符表 (Global Descriptors Table, GDT)。在x86上每个任务都运行在自己的虚拟地址空间里， 这个空间由"段址:偏移量"的数对指定。举个例子，当前将要由处理器执行的指令在 CS:EIP，那么这条指令的线性虚拟地址就是“代码段虚拟段地址CS” + EIP。 为了简便，段起始于虚拟地址0，终止于界限4G字节。所以，在这个例子中， 指令的线性虚拟地址正是EIP的值。段寄存器，如CS、DS等是选择符， 即全局描述符表中的索引(更精确的说，索引并非选择符的全部， 而是选择符中的INDEX部分)。

> ### 译者注
>
> 对于80386， 选择符有16位，INDEX部分是其中的高13位。

FreeBSD的全局描述符表为每个CPU保存着15个选择符:

```
sys/i386/i386/machdep.c:
union descriptor gdt[NGDT * MAXCPU]; /* ###### */

sys/i386/include/segments.h:
/*
 * ######(GDT)####
 */
#define GNULL_SEL 0 /* #### */
#define GCODE_SEL 1 /* ####### */
#define GDATA_SEL 2 /* ####### */
#define GPRIV_SEL 3 /* #####(SMP)######## */
#define GPROC0_SEL 4 /* Task state process slot zero and up, ###### */
#define GLDT_SEL 5 /* ########### */
#define GUSERLDT_SEL 6 /* ############ */
#define GTGATE_SEL 7 /* ######## */
#define GBIOSLOWMEM_SEL 8 /* BIOS######(#####8###) */
#define GPANIC_SEL 9 /* ################# */
#define GBIOSCODE32_SEL 10 /* BIOS##(32###) */
```

```
#define GBIOSCODE16_SEL 11 /* BIOS##(16###) */
#define GBIOSDATA_SEL 12 /* BIOS##(##) */
#define GBIOSUTIL_SEL 13 /* BIOS##(##) */
#define GBIOSARGS_SEL 14 /* BIOS##(######) */
```

请注意，这些#defines并非选择符本身，而只是选择符中的INDEX域， 因此它们正是全局描述符表中的索引。 例如，内核代码的选择符(GCODE_SEL)的值为0x08。

下一步是初始化中断描述符表(Interrupt Descriptor Table，IDT)。 这张表在发生软件或硬件中断时会被处理器引用。例如，执行系统调用时， 用户应用程序提交INT 0x80 指令。这是一个软件中断， 处理器用索引值0x80在中断描述符表中查找记录。这个记录指向处理这个中断的例程。 在这个特定情形中，这是内核的系统调用关口。

> **译者注**
>
> Intel 80386支持“调用门”，可以使得用户程序只通过一条call指令 就调用内核中的例程。可是FreeBSD并未采用这种机制， 也许是因为使用软中断接口可免去动态链接的麻烦吧。另外还有一个附带的好处： 在仿真Linux时，当遇到FreeBSD内核不支持的而又并非关键性的系统调用时， 内核只会显示一些出错信息，这使得程序能够继续运行； 而不是在真正执行程序之前的初始化过程中就因为动态链接失败而不允许程序运行。

中断描述符表最多可以有256 (0x100)条记录。内核分配NIDT条记录的内存给中断描述符表， 这里NIDT=256，是最大值：

sys/i386/i386/machdep.c:

```
static struct gate_descriptor idt0[NIDT];
struct gate_descriptor *idt = &idt0[0]; /* ###### */
```

每个中断都被设置一个合适的中断处理程序。 系统调用关口INT 0x80也是如此：

sys/i386/i386/machdep.c:

```
  setidt(0x80, &IDTVEC(int0x80_syscall),
   SDT_SYS386TGT, SEL_UPL, GSEL(GCODE_SEL, SEL_KPL));
```

所以当一个用户应用程序提交INT 0x80指令时， 全系统的控制权会传递给函数_Xint0x80_syscall， 这个函数在内核代码段中，将被以管理员权限执行。

然后，控制台和DDB(调试器)被初始化：

sys/i386/i386/machdep.c:

```
 cninit();
/* ############DDB#### */
#ifdef DDB
 kdb_init();
 if (boothowto & RB_KDB)
  Debugger("Boot flags requested debugger");
#endif
```

任务状态段(TSS)是另一个x86保护模式中的数据结构。当发生任务切换时， 任务状态段用来让硬件存储任务现场信息。

局部描述符表(LDT)用来指向用户代码和数据。系统定义了几个选择符， 指向局部描述符表，它们是系统调用关口和用户代码、用户数据选择符：

```
/usr/include/machine/segments.h
#define LSYS5CALLS_SEL 0 /* Intel BCS##### */
#define LSYS5SIGR_SEL 1
#define L43BSDCALLS_SEL 2 /* ## */
#define LUCODE_SEL 3
#define LSOL26CALLS_SEL 4 /* Solaris >=2.6####### */
#define LUDATA_SEL 5
/* separate stack, es,fs,gs sels -? #####es#fs#gs#### */
/* #define LPOSIXCALLS_SEL 5*/ /* notyet, ## */
#define LBSDICALLS_SEL 16 /* BSDI system call gate, BSDI###### */
#define NLDT  (LBSDICALLS_SEL + 1)
```

然后，proc0(0号进程，即内核所处的进程)的进程控制块(Process Control Block) (struct pcb)结构被初始化。proc0是一个 struct proc 结构，描述了一个内核进程。 内核运行时，该进程总是存在，所以这个结构在内核中被定义为全局变量：

```
sys/kern/kern_init.c:
   struct proc proc0;
```

结构struct pcb是proc结构的一部分， 它定义在/usr/include/machine/pcb.h之中， 内含针对i386硬件结构专有的信息，如寄存器的值。

1.7.2. mi_startup()

这个函数用冒泡排序算法，将所有系统初始化对象，然后逐个调用每个对象的入口：

```
sys/kern/init_main.c:
 for (sipp = sysinit; *sipp; sipp++) {

  /* -... ## -... */

  /* #### */
  (*((*sipp)->func))((*sipp)->udata);
  /* -... ## -... */
 }
```

尽管sysinit框架已经在《FreeBSD开发者手册》中有所描述， 我还是在这里讨论一下其内部原理。

每个系统初始化对象(sysinit对象)通过调用宏建立。 让我们以announce sysinit对象为例。 这个对象打印版权信息：

```
sys/kern/init_main.c:
static void
print_caddr_t(void *data __unused)
{
 printf("%s", (char *)data);
}
SYSINIT(announce, SI_SUB_COPYRIGHT, SI_ORDER_FIRST, print_caddr_t, copyright)
```

这个对象的子系统标识是SI_SUB_COPYRIGHT(0x0800001)， 数值刚好排在SI_SUB_CONSOLE(0x0800000)后面。 所以，版权信息将在控制台初始化之后就被很早的打印出来。

让我们看一看宏SYSINIT()到底做了些什么。 它展开成宏C_SYSINIT()。宏C_SYSINIT()然后展开成一个静态结构 struct sysinit。结构里申明里调用了另一个宏 DATA_SET：

```
/usr/include/sys/kernel.h:
     #define C_SYSINIT(uniquifier, subsystem, order, func, ident) \
     static struct sysinit uniquifier ## _sys_init = { \ subsystem, \
     order, \ func, \ ident \ -}; \ DATA_SET(sysinit_set,uniquifier ##
     _sys_init);

#define SYSINIT(uniquifier, subsystem, order, func, ident) \
 C_SYSINIT(uniquifier, subsystem, order,   \
 (sysinit_cfunc_t)(sysinit_nfunc_t)func, (void *)ident)
```

宏DATA_SET()展开成MAKE_SET()， 宏MAKE_SET()指向所有隐含的sysinit幻数：

```
/usr/include/linker_set.h
#define MAKE_SET(set, sym)      \
 static void const * const __set_##set##_sym_##sym = &sym; \
 __asm(".section -.set." #set -",\"aw\"");    \
 __asm(".long -" #sym);      \
 __asm(".previous")
#endif
#define TEXT_SET(set, sym) MAKE_SET(set, sym)
#define DATA_SET(set, sym) MAKE_SET(set, sym)
```

回到我们的例子中，经过宏的展开过程，将会产生如下声明：

```
static struct sysinit announce_sys_init = {
 SI_SUB_COPYRIGHT,
 SI_ORDER_FIRST,
```

```
 (sysinit_cfunc_t)(sysinit_nfunc_t)  print_caddr_t,
 (void *) copyright
};

static void const *const __set_sysinit_set_sym_announce_sys_init =
   &announce_sys_init;
__asm(".section -.set.sysinit_set" -",\"aw\"");
__asm(".long -" -"announce_sys_init");
__asm(".previous");
```

第一个__asm指令在内核可执行文件中建立一个ELF节(section)。 这发生在内核链接的时候。这一节将被命令为.set.sysinit_set。 这一节的内容是一个32位值##announce_sys_init结构的地址，这个结构正是第二个 __asm指令所定义的。第三个__asm指令标记节的结束。 如果前面有名字相同的节定义语句，节的内容(那个32位值)将被填加到已存在的节里， 这样就构造出了一个32位指针数组。

用objdump察看一个内核二进制文件， 也许你会注意到里面有这么几个小的节：

```
% objdump --h -/kernel
  7 -.set.cons_set 00000014  c03164c0  c03164c0  002154c0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  8 -.set.kbddriver_set 00000010  c03164d4  c03164d4  002154d4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  9 -.set.scrndr_set 00000024  c03164e4  c03164e4  002154e4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 10 -.set.scterm_set 0000000c  c0316508  c0316508  00215508  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 11 -.set.sysctl_set 0000097c  c0316514  c0316514  00215514  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 12 -.set.sysinit_set 00000664  c0316e90  c0316e90  00215e90  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

这一屏信息显示表明节.set.sysinit_set有0x664字节的大小， 所以0x664/sizeof(void *)个sysinit对象被编译进了内核。 其它节，如.set.sysctl_set表示其它链接器集合。

通过定义一个类型为struct linker_set的变量， 节.set.sysinit_set将被“收集”到那个变量里：

```
sys/kern/init_main.c:
    extern struct linker_set sysinit_set; -/* XXX */
```

struct linker_set定义如下：

```
/usr/include/linker_set.h:
  struct linker_set {
 int ls_length;
 void *ls_items[1];  /* ls_length#####, #NULL## */
};
```

译者注

实际上是说， 用C语言结构体linker_set来表达那个ELF节。

第一项是sysinit对象的数量，第二项是一个以NULL结尾的数组， 数组中是指向那些对象的指针。

回到对mi_startup()的讨论， 我们清楚了sysinit对象是如何被组织起来的。 函数mi_startup()将它们排序， 并调用每一个对象。最后一个对象是系统调度器：

```
/usr/include/sys/kernel.h:
enum sysinit_sub_id {
 SI_SUB_DUMMY  = 0x0000000, /* ############ */
 SI_SUB_DONE  = 0x0000001, /* ####*/
 SI_SUB_CONSOLE  = 0x0800000, /* ###*/
 SI_SUB_COPYRIGHT = 0x0800001, /* ########## */
...
 SI_SUB_RUN_SCHEDULER = 0xfffffff /* ###:### */
};
```

系统调度器sysinit对象定义在文件sys/vm/vm_glue.c中， 这个对象的入口点是scheduler()。 这个函数实际上是个无限循环，它表示那个进程标识(PID)为0的进程##swapper进程。 前面提到的proc0结构正是用来描述这个进程。

第一个用户进程是init， 由sysinit对象init建立：

```
sys/kern/init_main.c:
static void
create_init(const void *udata __unused)
{
 int error;
 int s;

 s = splhigh();
 error = fork1(&proc0, RFFDG -| RFPROC, &initproc);
 if (error)
  panic("cannot fork init: %d\n", error);
 initproc->p_flag -|= P_INMEM -| P_SYSTEM;
 cpu_set_fork_handler(initproc, start_init, NULL);
 remrunqueue(initproc);
 splx(s);
}
SYSINIT(init,SI_SUB_CREATE_INIT, SI_ORDER_FIRST, create_init, NULL)
```

create_init()通过调用fork1() 分配一个新的进程，但并不将其标记为可运行。当这个新进程被调度器调度执行时， start_init()将会被调用。 那个函数定义

在init_main.c中。 它尝试装载并执行二进制代码init， 先尝试/sbin/init，然后是/sbin/oinit， /sbin/init.bak，最后是/stand/sysinstall：

sys/kern/init_main.c:

```
static char init_path[MAXPATHLEN] =
#ifdef INIT_PATH
    __XSTRING(INIT_PATH);
#else
    -"/sbin/init:/sbin/oinit:/sbin/init.bak:/stand/sysinstall";
#endif
```

第 2 章 内核中的锁

翻译：.

这一章由 FreeBSD SMP Next Generation Project 维护。 请将评论和建议发送给FreeBSD 对称多处理 (SMP) 邮件列表.

这篇文档提纲挈领的讲述了在FreeBSD内核中的锁，这些锁使得有效的多处理成为可能。 锁可以用几种方式获得。数据结构可以用mutex或lockmgr(9)保护。 对于为数不多的若干个变量，假如总是使用原子操作访问它们，这些变量就可以得到保护。

> **译者注**
>
> 仅读本章内容，还不足以找出“mutex” 和“共享互斥锁”的区别。似乎它们的功能有重叠之处， 前者比后者的功能选项更多。它们似乎都是lockmgr(9)的子集。

2.1. Mutex

Mutex就是一种用来解决共享/排它矛盾的锁。 一个mutex在一个时刻只可以被一个实体拥有。如果另一个实体要获得已经被拥有的mutex， 就会进入等待，直到这个mutex被释放。在FreeBSD内核中，mutex被进程所拥有。

Mutex可以被递归的索要，但是mutex一般只被一个实体拥有较短的一段时间， 因此一个实体不能在持有mutex时睡眠。如果你需要在持有mutex时睡眠， 可使用一个 lockmgr(9) 的锁。

每个mutex有几个令人感兴趣的属性：

变量名
: 在内核源代码中struct mtx变量的名字

逻辑名
: 由函数mtx_init指派的mutex的名字。 这个名字显示在KTR跟踪消息和witness出错与警告信息里。 这个名字还用于区分标识在witness代码中的各个mutex

类型
: Mutex的类型，用标志MTX_*表示。 每个标志的意义在mutex(9)有所描述。

    MTX_DEF
    : 一个睡眠mutex

MTX_SPIN
    一个循环mutex

MTX_RECURSE
    这个mutex允许递归

保护对象
    这个入口所要保护的数据结构列表或数据结构成员列表。 对于数据结构成员，将按照 ###.###的形式命名。

依赖函数
    仅当mutex被持有时才可以被调用的函数

表 2.1. Mutex列表

变量名	逻辑名	类型	保护对象	依赖函数
sched_lock	“sched lock”（调度器锁）	MTX_SPIN | MTX_RECURSE	_gmonparam，cnt.v_swtch，cp_time，curpriority，mtx.mtx_blocked，mtx.mtx_contested，proc.p_procq，proc.p_slpq，proc.p_sflag，proc.p_stat，proc.p_estcpu，proc.p_cpticks proc.p_pctcpu，proc.p_wchan，proc.p_wmesg，proc.p_swtime，proc.p_slptime，proc.p_runtime，proc.p_uu，proc.p_su，proc.p_iu，proc.p_uticks，proc.p_sticks，proc.p_iticks，proc.p_oncpu，proc.p_lastcpu，proc.p_rqindex，	setrunqueue，remrunqueue，mi_switch，chooseproc，schedclock，resetpriority，updatepri，maybe_resched，cpu_switch，cpu_throw，need_resched，resched_wanted，clear_resched，aston， astoff，astpending，calcru，proc_compare

变量名	逻辑名	类型	保护对象	依赖函数
			proc.p_heldmtx, proc.p_blocked, proc.p_mtxname, proc.p_contested, proc.p_priority, proc.p_usrpri, proc.p_nativepri, proc.p_nice, proc.p_rtprio, pscnt, slpque, itqueuebits, itqueues, rtqueuebits, rtqueues, queuebits, queues, idqueuebits, idqueues, switchtime, switchticks	
vm86pcb_lock	“vm86pcb lock”(虚拟8086模式进程控制块锁)	MTX_DEF	vm86pcb	vm86_bioscall
Giant	“Giant”(巨锁)	MTX_DEF | MTX_RECURSE	几乎可以是任何东西	许多
callout_lock	“callout lock”(延时调用锁)	MTX_SPIN | MTX_RECURSE	callfree, callwheel, nextsoftcheck, proc.p_itcallout, proc.p_slpcallout, softticks, ticks	

2.2. 共享互斥锁

这些锁提供基本的读/写类型的功能，可以被一个正在睡眠的进程持有。 现在它们被统一到lockmgr(9)之中。

表 2.2. 共享互斥锁列表

变量名	保护对象
allproc_lock	allproc zombproc pidhashtbl proc.p_list proc.p_hash nextpid
proctree_lock	proc.p_children proc.p_sibling

2.3. 原子保护变量

原子保护变量并非由一个显在的锁保护的特殊变量，而是： 对这些变量的所有数据访问都要使用特殊的原子操作(atomic(9))。 尽管其它的基本同步机制(例如mutex)就是用原子保护变量实现的， 但是很少有变量直接使用这种处理方式。

- mtx.mtx_lock

第 3 章 内核对象

翻译：.

内核对象，也就是Kobj，为内核提供了一种面向对象 的C语言编程方式。被操作的数据也承载操作它的方法。 这使得在不破坏二进制兼容性的前提下，某一个接口能够增/减相应的操作。

3.1. 术语

对象
: 数据集合–数据结构–数据分配的集合

方法
: 某一种操作—函数

类
: 一种或多种方法

接口
: 一种或多种方法的一个标准集合

3.2. Kobj的工作流程

> **译者注**
>
> 这一小节两段落中原作者的用词有些含混， 请参考我在括号中的注释阅读。

Kobj工作时，产生方法的描述。每个描述有一个唯一的标识和一个缺省函数。 某个描述的地址被用来在一个类的方法表里唯一的标识方法。

构建一个类，就是要建立一张方法表，并将这张表关联到一个或多个函数(方法)；这些函数(方法)都带有方法描述。使用前，类要被编译。编译时要为这个类分配一些缓存。 在方法表中的每个方法描述都会被指派一个唯一的标识， 除非已经被其它引用它的类在编译时指派了标识。对于每个将要被使用的方法， 都会由脚本生成一个函数(方法查找函数)，以解析外来参数， 并在被查询时给出方法描述的地址。被生成的函数(方法查找函数) 凭着那个方法描述的唯一标识按Hash的方法查找对象的类的缓存。 如果这个方法不在缓存中，函数会查找使用类的方法表。如果这个方法被找到了， 类里的相关函数(也就是某个方法的实现代码)就会被使用。 否则，这个方法描述的缺省函数将被使用。

这些过程可被表示如下：

```
##->##<->#
```

3.3. 使用Kobj

3.3.1. 结构

```
struct kobj_method
```

3.3.2. 函数

```
void kobj_class_compile(kobj_class_t cls);
void kobj_class_compile_static(kobj_class_t cls, kobj_ops_t ops);
void kobj_class_free(kobj_class_t cls);
kobj_t kobj_create(kobj_class_t cls, struct malloc_type *mtype, int mflags);
void kobj_init(kobj_t obj, kobj_class_t cls);
void kobj_delete(kobj_t obj, struct malloc_type *mtype);
```

3.3.3. 宏

```
KOBJ_CLASS_FIELDS
KOBJ_FIELDS
DEFINE_CLASS(name, methods, size)
KOBJMETHOD(NAME, FUNC)
```

3.3.4. 头文件

```
<sys/param.h>
<sys/kobj.h>
```

3.3.5. 建立一个接口的模板

使用Kobj的第一步是建立一个接口。建立接口包括建立模板的工作。 建立模板可用脚本src/sys/kern/makeobjops.pl完成， 它会产生申明方法的头文件和代码，脚本还会生成方法查找函数。

在这个模板中如下关键词会被使用：#include， INTERFACE， CODE， METHOD， STATICMETHOD，和 DEFAULT.

#include语句的整行内容将被一字不差的 复制到被生成的代码文件的头部。

例如：

```
#include <sys/foo.h>
```

关键词INTERFACE用来定义接口名。 这个名字将与每个方法名接合在一起，形成 [interface name]_[method name]。 语法是：INTERFACE [接口名]；

例如：

```
INTERFACE foo;
```

关键词CODE会将它的参数一字不差的复制到代码文件中。 语法是CODE { [####] };

例如：

```
CODE {
 struct foo * foo_alloc_null(struct bar *)
 {
  return NULL;
}
};
```

关键词METHOD用来描述一个方法。语法是： METHOD [#####] [###] { [## [, ####]] };

例如：

```
METHOD int bar {
 struct object *;
 struct foo *;
 struct bar;
};
```

关键词DEFAULT跟在关键词METHOD之后， 是对关键词METHOD的补充。它给这个方法补充上缺省函数。语法是： METHOD [#####] [###] { [##; [####]] }DEFAULT [####];

例如：

```
METHOD int bar {
 struct object *;
 struct foo *;
 int bar;
} DEFAULT foo_hack;
```

关键词STATICMETHOD类似关键词METHOD。 对于每个Kobj对象，一般其头部都有一些Kobj专有的数据。 METHOD定义的方法就假设这些专有数据位于对象头部； 假如对象头部没有这些专有数据，这些方法对这个对象的访问就可能出错。而STATICMETHOD定义的对象可以不受这个限制： 这样描述出的方法，其操作的数据不由这个类的某个对象实例给出， 而是全都由调用这个方法时的操作数(译者注：即参数)给出。 这也对于在某个类的方法表之外调用这个方法有用。

> **译者注**
>
> 这一段的语言与原文相比调整很大。 静态方法是不依赖于对象实例的方法。 参看C++类中的“静态函数”的概念。

其它完整的例子：

```
src/sys/kern/bus_if.m
src/sys/kern/device_if.m
```

3.3.6. 建立一个类

使用Kobj的第二步是建立一个类。一个类的组有名字、方法表； 假如使用了Kobj的“对象管理工具”（Object Handling Facilities）， 类中还包含对象的大小。建立类时使用宏DEFINE_CLASS()。 建立方法表时，须建立一个kobj_method_t数组，用NULL项结尾。 每个非NULL项可用宏KOBJMETHOD()建立。

例如：

```
DEFINE_CLASS(fooclass, foomethods, sizeof(struct foodata));

kobj_method_t foomethods[] = {
 KOBJMETHOD(bar_doo, foo_doo),
 KOBJMETHOD(bar_foo, foo_foo),
 { NULL, NULL}
};
```

类须被“编译”。根据该类被初始化时系统的状态， 将要用到一个静态分配的缓存和“操作数表”（ops table， 译者注：即“参数表”）。这些操作可通过声明一个结构体 struct kobj_ops并使用 kobj_class_compile_static()， 或是只使用kobj_class_compile()来完成。

3.3.7. 建立一个对象

使用Kobj的第三步是定义对象。Kobj对象建立程序假定Kobj 专有数据在一个对象的头部。如果不是如此，应当先自行分配对象， 再使用kobj_init()初始化对象中的Kobj专有数据； 其实可以使用kobj_create()分配对象， 并自动初始化对象中的Kobj专有内容。kobj_init() 也可以用来改变一个对象所使用的类。

将Kobj的数据集成到对象中要使用宏KOBJ_FIELDS。

例如

```
struct foo_data {
 KOBJ_FIELDS;
 foo_foo;
 foo_bar;
};
```

3.3.8. 调用方法

使用Kobj的最后一部就是通过生成的函数调用对象类中的方法。 调用时，接口名与方法名用'_'接合，而且全部使用大写字母。

例如，接口名为foo，方法为bar，调用就是：

```
[### = -] FOO_BAR(## [, ####]);
```

3.3.9. 善后处理

当一个用kobj_create()不再需要被使用时， 可对这个对象调用kobj_delete()。 当一个类不再需要被使用时， 可对这个类调用kobj_class_free()。

第 4 章 Jail子系统

翻译：.
Evan Sarmiento

<evms@cs.bu.edu>

在大多数UNIX®系统中，用户root是万能的。这也就增加了许多危险。 如果一个攻击者获得了一个系统中的root，就可以在他的指尖掌握系统中所有的功能。 在FreeBSD里，有一些sysctl项削弱了root的权限， 这样就可以将攻击者造成的损害减小到最低限度。这些安全功能中，有一种叫安全级别。 另一种在FreeBSD 4.0及以后版本中提供的安全功能，就是jail(8)。 Jail将一个运行环境的文件树根切换到某一特定位置， 并且对这样环境中叉分生成的进程做出限制。例如， 一个被监禁的进程不能影响这个jail之外的进程、不能使用一些特定的系统调用， 也就不能对主计算机造成破坏。

> **译者注**
>
> 英文单词“jail”的中文意思是“囚禁、监禁”。

Jail已经成为一种新型的安全模型。 人们可以在jail中运行各种可能很脆弱的服务器程序，如Apache、 BIND和sendmail。 这样一来，即使有攻击者取得了jail中的root， 这最多让人们皱皱眉头，而不会使人们惊慌失措。 本文主要关注jail的内部原理(源代码)。 如果你正在寻找设置Jail的指南性文档， 我建议你阅读我的另一篇文章，发表在Sys Admin Magazine, May 2001，《Securing FreeBSD using Jail》。

4.1. Jail的系统结构

Jail由两部分组成：用户级程序， 也就是jail(8)；还有在内核中Jail的实现代码：jail(2) 系统调用和相关的约束。我将讨论用户级程序和jail在内核中的实现原理。

4.1.1. 用户级代码

Jail的用户级源代码在/usr/src/usr.sbin/jail， 由一个文件jail.c组成。这个程序有这些参数：jail的路径， 主机名，IP地址，还有需要执行的命令。

4.1.1.1. 数据结构

在jail.c中，我将最先注解的是一个重要结构体 struct jail j;的声明，这个结构类型的声明包含在 /usr/include/sys/jail.h之中。

jail结构的定义是:

```
/usr/include/sys/jail.h:

struct jail {
        u_int32_t       version;
        char            *path;
        char            *hostname;
        u_int32_t       ip_number;
};
```

正如你所见，传送给命令jail(8)的每个参数都在这里有对应的一项。 事实上，当命令jail(8)被执行时，这些参数才由命令行真正传入:

```
/usr/src/usr.sbin/jail.c
char path[PATH_MAX];
...
if(realpath(argv[0], path) == NULL)
    err(1, -"realpath: %s", argv[0]);
if (chdir(path) -!= 0)
    err(1, -"chdir: %s", path);
memset(&j, 0, sizeof(j));
j.version = 0;
j.path = path;
j.hostname = argv[1];
```

4.1.1.2. 网络

传给jail(8)的参数中有一个是IP地址。这是在网络上访问jail时的地址。jail(8)将IP地址翻译成网络字节顺序，并存入j(jail类型的结构体)。

```
/usr/src/usr.sbin/jail/jail.c:
struct in_addr in;
...
if (inet_aton(argv[2], &in) == 0)
    errx(1, -"Could not make sense of ip-number: %s", argv[2]);
j.ip_number = ntohl(in.s_addr);
```

函数inet_aton(3) “将指定的字符串解释为一个Internet地址， 并将其转存到指定的结构体中” 。inet_aton(3)设定了结构体in， 之后in中的内容再用ntohl(3)转换成主机字节顺序， 并置入jail结构体的ip_number成员。

4.1.1.3. 囚禁进程

最后，用户级程序囚禁进程。现在Jail自身变成了一个被囚禁的进程， 并使用execv(3)执行用户指定的命令。

```
/usr/src/usr.sbin/jail/jail.c
i = jail(&j);
...
if (execv(argv[3], argv + 3) -!= 0)
    err(1, -"execv: %s", argv[3]);
```

正如你所见，函数jail()被调用，参数是结构体jail中被填入数据项， 而如前所述，这些数据项又来自jail(8)的命令行参数。 最后，执行了用户指定的命令。下面我将开始讨论jail在内核中的实现。

4.1.2. 相关的内核源代码

现在我们来看文件/usr/src/sys/kern/kern_jail.c。 在这里定义了jail(2)的系统调用、相关的sysctl项，还有网络函数。

4.1.2.1. sysctl项

在kern_jail.c里定义了如下sysctl项:

```
/usr/src/sys/kern/kern_jail.c:

int     jail_set_hostname_allowed = 1;
SYSCTL_INT(_security_jail, OID_AUTO, set_hostname_allowed, CTLFLAG_RW,
    &jail_set_hostname_allowed, 0,
    -"Processes in jail can set their hostnames");
    -/* Jail############# */

int     jail_socket_unixiproute_only = 1;
SYSCTL_INT(_security_jail, OID_AUTO, socket_unixiproute_only, CTLFLAG_RW,
    &jail_socket_unixiproute_only, 0,
    -"Processes in jail are limited to creating UNIX/IPv4/route sockets only");
    -/* Jail###########UNIX####IPv4########## */

int     jail_sysvipc_allowed = 0;
SYSCTL_INT(_security_jail, OID_AUTO, sysvipc_allowed, CTLFLAG_RW,
    &jail_sysvipc_allowed, 0,
    -"Processes in jail can use System V IPC primitives");
    -/* Jail########System V####### */

static int jail_enforce_statfs = 2;
SYSCTL_INT(_security_jail, OID_AUTO, enforce_statfs, CTLFLAG_RW,
    &jail_enforce_statfs, 0,
    -"Processes in jail cannot see all mounted file systems");
    -/* jail ####################### */

int     jail_allow_raw_sockets = 0;
SYSCTL_INT(_security_jail, OID_AUTO, allow_raw_sockets, CTLFLAG_RW,
    &jail_allow_raw_sockets, 0,
    -"Prison root can create raw sockets");
    -/* jail ## root ######## raw socket */
```

```
int    jail_chflags_allowed = 0;
SYSCTL_INT(_security_jail, OID_AUTO, chflags_allowed, CTLFLAG_RW,
    &jail_chflags_allowed, 0,
    -"Processes in jail can alter system file flags");
    -/* jail ################# */

int    jail_mount_allowed = 0;
SYSCTL_INT(_security_jail, OID_AUTO, mount_allowed, CTLFLAG_RW,
    &jail_mount_allowed, 0,
    -"Processes in jail can mount/unmount jail-friendly file systems");
    -/* jail ##############jail####### */
```

这些sysctl项中的每一个都可以用命令sysctl(8)访问。在整个内核中， 这些sysctl项按名称标识。例如，上述第一个sysctl项的名字是security.jail.set_hostname_allowed。

4.1.2.2. jail(2)系统调用

像所有的系统调用一样，系统调用jail(2)带有两个参数， struct thread *td和struct jail_args *uap。 td是一个指向thread结构体的指针，该指针用于描述调用jail(2)的线程。 在这个上下文中，uap指向一个结构体，这个结构体中包含了一个指向从用户级 jail.c传送过来的jail结构体的指针。 在前面我讲述用户级程序时，你已经看到过一个jail结构体被作为参数传送给系统调用 jail(2)。

```
/usr/src/sys/kern/kern_jail.c:
/*
 * struct jail_args {
 *     struct jail *jail;
 * -};
 */
int
jail(struct thread *td, struct jail_args *uap)
```

于是uap->jail可以用于访问被传递给jail(2)的jail结构体。 然后，jail(2)使用copyin(9)将jail结构体复制到内核内存空间中。 copyin(9)需要三个参数：要复制进内核内存空间的数据的地址 uap->jail，在内核内存空间存放数据的j， 以及数据的大小。uap->jail指向的Jail结构体被复制进内核内存空间， 并被存放在另一个jail结构体j里。

```
/usr/src/sys/kern/kern_jail.c:
error = copyin(uap->jail, &j, sizeof(j));
```

在jail.h中定义了另一个重要的结构体型prison。 结构体prison只被用在内核空间中。 下面是prison结构体的定义。

```
/usr/include/sys/jail.h:
struct prison {
```

```
        LIST_ENTRY(prison) pr_list;              -/* (a) all prisons */
        int          pr_id;                  -/* (c) prison id */
        int          pr_ref;                 -/* (p) refcount */
        char          pr_path[MAXPATHLEN];        -/* (c) chroot path */
        struct vnode    *pr_root;                -/* (c) vnode to rdir */
        char          pr_host[MAXHOSTNAMELEN];      -/* (p) jail hostname */
        u_int32_t      pr_ip;                  -/* (c) ip addr host */
        void          *pr_linux;               -/* (p) linux abi */
        int          pr_securelevel;            -/* (p) securelevel */
        struct task     pr_task;                -/* (d) destroy task */
        struct mtx      pr_mtx;
        void          **pr_slots;              -/* (p) additional data */
};
```

然后，系统调用jail(2)为一个prison结构体分配一块内存，并在jail和prison结构体之间复制数据。

```
/usr/src/sys/kern/kern_jail.c:
MALLOC(pr, struct prison *, sizeof(*pr), M_PRISON, M_WAITOK -| M_ZERO);
...
error = copyinstr(j.path, &pr->pr_path, sizeof(pr->pr_path), 0);
if (error)
   goto e_killmtx;
...
error = copyinstr(j.hostname, &pr->pr_host, sizeof(pr->pr_host), 0);
if (error)
      goto e_dropvnref;
pr->pr_ip = j.ip_number;
```

下面，我们将讨论另外一个重要的系统调用jail_attach(2)，它实现了将进程监禁的功能。

```
/usr/src/sys/kern/kern_jail.c
/*
 * struct jail_attach_args {
 *     int jid;
 * -};
 */
int
jail_attach(struct thread *td, struct jail_attach_args *uap)
```

这个系统调用做出一些可以用于区分被监禁和未被监禁的进程的改变。要理解jail_attach(2)为我们做了什么，我们首先要理解一些背景信息。

在FreeBSD中，每个对内核可见的线程是通过其thread结构体来识别的，同时，进程都由它们自己的proc结构体描述。你可以在/usr/include/sys/proc.h中找到thread和proc结构体的定义。例如，在任何系统调用中，参数td实际上是个指向调用线程的thread结构体的指针，正如前面所说的那样。td所指向的thread结构体中的td_proc成员是一个指针，这个指针指向td所表示的线程所属进程的proc结构体。结构体proc包含的成员可以描述所有者的身份（p_ucred），进程资源限制（p_limit），

等等。在由proc结构体的p_ucred成员所指向的ucred结构体的定义中， 还有一个指向prison结构体的指针(cr_prison)。

```
/usr/include/sys/proc.h:
struct thread {
   -...
   struct proc *td_proc;
   -...
};
struct proc {
   -...
   struct ucred *p_ucred;
   -...
};
/usr/include/sys/ucred.h
struct ucred {
   -...
   struct prison *cr_prison;
   -...
};
```

在kern_jail.c中，函数jail()以给定的jid 调用函数jail_attach()。随后jail_attach()调用函数change_root()以改变 调用进程的根目录。接下来，jail_attach()创建一个新的ucred结构体，并在 成功地将prison结构体连接到这个ucred结构体后，将这个ucred结构体连接 到调用进程上。从此时起，这个调用进程就会被识别为被监禁的。 当我们以新创建的这个ucred结构体为参数调用内核路径jailed()时， 它将返回1来说明这个用户身份是和一个jail相连的。 在jail中叉分出来的所有进程的的公共祖先进程就是这个执行了jail(2)的进程， 因为正是它调用了jail(2)系统调用。当一个程序通过execve(2)而被执行时， 它将从其父进程的ucred结构体继承被监禁的属性， 因而它也会拥有一个被监禁的ucred结构体。

```
/usr/src/sys/kern/kern_jail.c
int
jail(struct thread *td, struct jail_args *uap)
{
...
   struct jail_attach_args jaa;
...
   error = jail_attach(td, &jaa);
   if (error)
      goto e_dropprref;
...
}

int
jail_attach(struct thread *td, struct jail_attach_args *uap)
{
   struct proc *p;
   struct ucred *newcred, *oldcred;
   struct prison *pr;
```

```
...
   p = td->td_proc;
...
   pr = prison_find(uap->jid);
...
   change_root(pr->pr_root, td);
...
   newcred->cr_prison = pr;
   p->p_ucred = newcred;
...
}
```

当一个进程被从其父进程叉分来的时候， 系统调用fork(2)将用crhold()来维护其身份凭证。 这样，很自然的就保持了子进程的身份凭证于其父进程一致，所以子进程也是被监禁的。

```
/usr/src/sys/kern/kern_fork.c:
p2->p_ucred = crhold(td->td_ucred);
...
td2->td_ucred = crhold(p2->p_ucred);
```

4.2. 系统对被囚禁程序的限制

在整个内核中，有一系列对被囚禁程序的约束措施。 通常，这些约束只对被囚禁的程序有效。如果这些程序试图突破这些约束， 相关的函数将出错返回。例如:

```
if (jailed(td->td_ucred))
   return EPERM;
```

4.2.1. SysV进程间通信(IPC)

System V 进程间通信 (IPC) 是通过消息实现的。 每个进程都可以向其它进程发送消息， 告诉对方该做什么。 处理消息的函数是: msgctl(3)、msgget(3)、msgsnd(3) 和 msgrcv(3)。前面已经提到，一些 sysctl 开关可以影响 jail 的行为， 其中有一个是 security.jail.sysvipc_allowed。 在大多数系统上， 这个 sysctl 项会设成0。 如果将它设为1， 则会完全失去 jail 的意义： 因为那样在 jail 中特权进程就可以影响被监禁的环境外的进程了。 消息与信号的区别是：消息仅由一个信号编号组成。

/usr/src/sys/kern/sysv_msg.c:

- msgget(key, msgflg)：msgget返回(也可能创建)一个消息描述符， 以指派一个在其它函数中使用的消息队列。

- msgctl(msgid, cmd, buf)：通过这个函数， 一个进程可以查询一个消息描述符的状态。

- msgsnd(msgid, msgp, msgsz, msgflg)：msgsnd向一个进程发送一条消息。
- msgrcv(msgid, msgp, msgsz, msgtyp, msgflg)：进程用这个函数接收消息。

在这些函数对应的系统调用的代码中，都有这样一个条件判断：

```
/usr/src/sys/kern/sysv_msg.c:
if (!jail_sysvipc_allowed && jailed(td->td_ucred))
    return (ENOSYS);
```

信号量系统调用使得进程可以通过一系列原子操作实现同步。 信号量为进程锁定资源提供了又一种途径。 然而，进程将为正在被使用的信号量进入等待状态，一直休眠到资源被释放。 在jail中如下的信号量系统调用将会失效： semget(2)，semctl(2) 和semop(2)。

/usr/src/sys/kern/sysv_sem.c：

- semctl(semid, num, cmd, ...)： semctl对在信号量队列中用semid标识的信号量执行cmd指定的命令。
- semget(key, nsems, flag)：semget建立一个对应于key的信号量数组。

  ##key#flag####msgget()######

- setop(semid, array, nops)：semop对semid标识的信号量完成一组由array所指定的操作。

System V IPC使进程间可以共享内存。进程之间可以通过它们虚拟地址空间 的共享部分以及相关数据读写操作直接通讯。这些系统调用在被监禁的环境中将会失效：shmdt(2)、shmat(2)、shmctl(2)和shmget(2)

/usr/src/sys/kern/sysv_shm.c：

- shmctl(shmid, cmd, buf)：shmctl对id标识的共享内存区域做各种各样的控制。
- shmget(key, size, flag)：shmget建立/打开size字节的共享内存区域。
- shmat(shmid, addr, flag)：shmat将shmid标识的共享内存区域指派到进程的地址空间里。
- shmdt(addr)：shmdt取消共享内存区域的地址指派。

4.2.2. 套接字

Jail以一种特殊的方式处理socket(2)系统调用和相关的低级套接字函数。 为了决定一个套接字是否允许被创建，它先检查sysctl项

security.jail.socket_unixiproute_only是否被设置为1。 如果被设为1，套接字建立时将只能指定这些协议族： PF_LOCAL， PF_INET， PF_ROUTE。否则，socket(2)将会返回出错。

```
/usr/src/sys/kern/uipc_socket.c:
int
socreate(int dom, struct socket **aso, int type, int proto,
    struct ucred *cred, struct thread *td)
{
    struct protosw *prp;
...
    if (jailed(cred) && jail_socket_unixiproute_only &&
        prp->pr_domain->dom_family -!= PF_LOCAL &&
        prp->pr_domain->dom_family -!= PF_INET &&
        prp->pr_domain->dom_family -!= PF_ROUTE) {
        return (EPROTONOSUPPORT);
    -}
...
}
```

4.2.3. Berkeley包过滤器

Berkeley包过滤器提供了一个与协议无关的，直接通向数据链路层的低级接口。现在BPF是否可以在监禁的环境中被使用是通过devfs(8)来控制的。

4.2.4. 网络协议

网络协议TCP，UDP，IP和ICMP很常见。IP和ICMP处于同一协议层次：第二层， 网络层。当参数nam被设置时， 有一些限制措施会防止被囚禁的程序绑定到一些网络接口上。 nam是一个指向sockaddr结构体的指针， 描述可以绑定服务的地址。一个更确切的定义：sockaddr “是一个模板，包含了地址的标识符和地址的长度” 。 在函数in_pcbbind_setup()中sin是一个指向sockaddr_in结构体的指针， 这个结构体包含了套接字可以绑定的端口、地址、长度、协议族。 这就禁止了在jail中的进程指定不属于这个进程所存在于的jail的IP地址。

```
/usr/src/sys/kern/netinet/in_pcb.c:
int
in_pcbbind_setup(struct inpcb *inp, struct sockaddr *nam, in_addr_t *laddrp,
    u_short *lportp, struct ucred *cred)
{
    -...
    struct sockaddr_in *sin;
    -...
    if (nam) {
        sin = (struct sockaddr_in *)nam;
        -...
        if (sin->sin_addr.s_addr -!= INADDR_ANY)
            if (prison_ip(cred, 0, &sin->sin_addr.s_addr))
```

```
            return(EINVAL);
        -...
        if (lport) {
            -...
            if (prison && prison_ip(cred, 0, &sin->sin_addr.s_addr))
                return (EADDRNOTAVAIL);
            -...
        -}
    -}
    if (lport == 0) {
        -...
        if (laddr.s_addr -!= INADDR_ANY)
            if (prison_ip(cred, 0, &laddr.s_addr))
                return (EINVAL);
        -...
    -}
...
    if (prison_ip(cred, 0, &laddr.s_addr))
        return (EINVAL);
...
}
```

你也许想知道函数prison_ip()做什么。 prison_ip()有三个参数，一个指向身份凭证的指针(用cred表示)， 一些标志和一个IP地址。当这个IP地址不属于这个jail时，返回1； 否则返回0。正如你从代码中看见的，如果，那个IP地址确实不属于这个jail， 就不再允许向这个网络地址绑定协议。

```
/usr/src/sys/kern/kern_jail.c:
int
prison_ip(struct ucred *cred, int flag, u_int32_t *ip)
{
    u_int32_t tmp;

    if (!jailed(cred))
        return (0);
    if (flag)
        tmp = *ip;
    else
        tmp = ntohl(*ip);
    if (tmp == INADDR_ANY) {
        if (flag)
            *ip = cred->cr_prison->pr_ip;
        else
            *ip = htonl(cred->cr_prison->pr_ip);
        return (0);
    -}
    if (tmp == INADDR_LOOPBACK) {
        if (flag)
            *ip = cred->cr_prison->pr_ip;
        else
            *ip = htonl(cred->cr_prison->pr_ip);
        return (0);
```

```
    -}
    if (cred->cr_prison->pr_ip -!= tmp)
        return (1);
    return (0);
}
```

4.2.5. 文件系统

如果完全级别大于0，即便是jail里面的root， 也不允许在Jail中取消或更改文件标志，如“不可修改”、“只可添加”、“不可删除”标志。

```
/usr/src/sys/ufs/ufs/ufs_vnops.c:
static int
ufs_setattr(ap)
    -...
{
    -...
        if (!priv_check_cred(cred, PRIV_VFS_SYSFLAGS, 0)) {
            if (ip->i_flags
                & (SF_NOUNLINK -| SF_IMMUTABLE -| SF_APPEND)) {
                    error = securelevel_gt(cred, 0);
                    if (error)
                        return (error);
            -}
            -...
        -}
}
/usr/src/sys/kern/kern_priv.c
int
priv_check_cred(struct ucred *cred, int priv, int flags)
{
    -...
    error = prison_priv_check(cred, priv);
    if (error)
        return (error);
    -...
}
/usr/src/sys/kern/kern_jail.c
int
prison_priv_check(struct ucred *cred, int priv)
{
    -...
    switch (priv) {
    -...
    case PRIV_VFS_SYSFLAGS:
        if (jail_chflags_allowed)
            return (0);
        else
            return (EPERM);
    -...
    -}
```

```
    -...
}
```

第 5 章 SYSINIT框架

翻译：.

SYSINIT是一个通用的调用排序与分别执行机制的框架。 FreeBSD目前使用它来进行内核的动态初始化。 SYSINIT使得FreeBSD的内核各子系统可以在内核或模块动态加载链接时被重整、 添加、删除、替换，这样，内核和模块加载时就不必去修改一个静态的有序初始化 安排表甚至重新编译内核。这个体系也使得内核模块(现在称为KLD可以与内核不同时编译、链接、 在引导系统时加载，甚至在系统运行时加载。这些操作是通过 “内核链接器”(kernel linker)和“链接器集合”(linker set)完成的。

5.1. 术语

链接器集合(Linker Set)

一种链接方法。这种方法将整个程序源文件中静态申明的数据收集到 一个可邻近寻址的数据单元中。

5.2. SYSINIT操作

SYSINIT要依靠链接器获取遍布整个程序源代码多处申明的静态数据 并把它们组成一个彼此相邻的数据块。这种链接方法被称为 “链接器集合”(linker set)。SYSINIT使用两个链接器集合以维护两个数据集合， 包含每个数据条目的调用顺序、函数、一个会被提交给该函数的数据指针。

SYSINIT按照两类优先级标识对函数排序以便执行。 第一类优先级的标识是子系统的标识， 给出SYSINIT分别执行子系统的函数的全局顺序， 定义在<sys/kernel.h>中的枚举 sysinit_sub_id内。第二类优先级标识在子系统中的元素的顺序， 定义在<sys/kernel.h>中的枚举 sysinit_elem_order内。

有两种时刻需要使用SYSINIT：系统启动或内核模块加载时， 系统析构或内核模块卸载时。内核子系统通常在系统启动时使用SYSINIT 的定义项以初始化数据结构。例如，进程调度子系统使用一个SYSINIT 定义项来初始化运行队列数据结构。设备驱动程序应避免直接使用 SYSINIT()，对于总线结构上的物理真实设备应使用 DRIVER_MODULE()调用的函数先侦测设备的存在， 如果存在，再进行设备的初始化。这一系统过程中， 会做一些专门针对设备的事情，然后调用SYSINIT()本身。 对于非总线结构一部分的虚设备，应改用DEV_MODULE()。

5.3. 使用SYSINIT

5.3.1. 接口

5.3.1.1. 头文件

```
<sys/kernel.h>
```

5.3.1.2. 宏

```
SYSINIT(uniquifier, subsystem, order, func, ident)
SYSUNINIT(uniquifier, subsystem, order, func, ident)
```

5.3.2. 启动

宏SYSINIT()在SYSINIT启动数据集合中 建立一个SYSINIT数据项，以便SYSINIT在系统启动或模块加载时排序 并执行其中的函数。SYSINIT()有一个参数uniquifier， SYSINIT用它来标识数据项，随后是子系统顺序号、子系统元素顺序号、 待调用函数、传递给函数的数据。所有的函数必须有一个恒量指针参数。

例 5.1. SYSINIT()的例子

```
#include <sys/kernel.h>

void foo_null(void *unused)
{
        foo_doo();
}
SYSINIT(foo, SI_SUB_FOO, SI_ORDER_FOO, foo_null, NULL);

struct foo foo_voodoo = {
        FOO_VOODOO;
}

void foo_arg(void *vdata)
{
        struct foo *foo = (struct foo *)vdata;
        foo_data(foo);
}
SYSINIT(bar, SI_SUB_FOO, SI_ORDER_FOO, foo_arg, &foo_voodoo);
```

注意，SI_SUB_FOO和SI_ORDER_FOO 应当分别在上面提到的枚举sysinit_sub_id和 sysinit_elem_order之中。既可以使用已有的枚举项， 也可以将自己的枚举项添加到这两个枚举的定义之中。 你可以使用数学表达式微调SYSINIT的

执行顺序。 以下的例子示例了一个需要刚好要在内核参数调整的SYSINIT之前执行的SYSINIT。

例 5.2. 调整SYSINIT()顺序的例子

```
static void
mptable_register(void *dummy __unused)
{

 apic_register_enumerator(&mptable_enumerator);
}

SYSINIT(mptable_register, SI_SUB_TUNABLES -- 1, SI_ORDER_FIRST,
    mptable_register, NULL);
```

5.3.3. 析构

宏SYSUNINIT()的行为与SYSINIT()的相当, 只是它将数据项填加至SYSINIT的析构数据集合。

例 5.3. SYSUNINIT()的例子

```
#include <sys/kernel.h>

void foo_cleanup(void *unused)
{
        foo_kill();
}
SYSUNINIT(foobar, SI_SUB_FOO, SI_ORDER_FOO, foo_cleanup, NULL);

struct foo_stack foo_stack = {
        FOO_STACK_VOODOO;
}

void foo_flush(void *vdata)
{
}
SYSUNINIT(barfoo, SI_SUB_FOO, SI_ORDER_FOO, foo_flush, &foo_stack);
```

第 6 章 TrustedBSD MAC 框架

Chris Costello 和 Robert Watson.
翻译：.

6.1. MAC 文档版权声明

本文档是作为 DARPA CHATS 研究计划的一部分，由供职于 Security Research Division of Network Associates 公司Safeport Network Services and Network Associates Laboratories 的Chris Costello依据 DARPA/SPAWAR 合同 N66001-01-C-8035（“CBOSS”），为 FreeBSD 项目编写的。

6.2. 术语解析

FreeBSD 以一个内核安全扩展性框架(TrustedBSD MAC 框架)的方式，为若干强制访问控制策略 (也称 “集权式访问控制策略”) 提供试验性支持。MAC 框架是一个插入式的访问控制框架，允许新的安全策略更方便地融入内核:安全策略可以静态链入内核,也可以 在引导时加载,甚至在运行时动态加载。该

框架所提供的标准化接口,使得运行在其上的安全策略模块能对系统对象的安全属性进行诸如标记等一系列操作。 MAC 框架的存在，简化了这些操作在策略模块中的实现，从而显著降低了新安全策略模块的开发难度。

本章将介绍 MAC 策略框架，为读者提供一个示例性的 MAC 策略模块文档。

6.3. 概述

TrustedBSD MAC 框架提供的机制,允许在其上运行的内核模块在内核编译或者运行时，对内核的访问控制模型进行扩展。 新的系统安全策略作为一个内核模块实现，并被链接到内核中；如果系统中同时存在多个安全策略模块，则它们的决策结果将以某种确定的方式组合。 为了给简化新安全策略的开发，MAC 向上提供了大量用于访问控制的基础设施，特别是，对临时的或者持久的、策略无关的对象安全标记的支持。 该支持目前仍是试验性质的。

本章所提供的信息不仅将使在 MAC 使能环境下工作的潜在用户受益， 也可以为需要了解 MAC 框架是如何支持对内核访问控制进行扩展的策略模块开发人员所用。

6.4. 安全策略背景知识

强制访问控制（简称 MAC），是指由操作系统强制实施的一组针对用户的访问控制策略。 在某些情况下,强制访问控制的策略可能会与自主访问控制（简称 DAC）所提供的保护措施发生冲突， 后者是用来向非管理员用户对数据采取保护措施提供支持的。在传统的 UNIX 系统中， DAC 保护措施包括文件访问模式和访问控制列表；而 MAC 则提供进程控制和防火墙等。 操作系统设计者和安全机制研究人员对许多经典的 MAC 安全策略作了形式化的表述，比如， 多级安全（MLS)机密性策略，Biba 完整性策略，基于角色的访问控制策略（RBAC），域和型裁决策略(DTE),以及型裁决策略(TE)。 安全策略的形式化表述被称为安全模型。每个模型根据一系列条件做出安全相关的决策，这些条件包括， 用户的身份、角色和安全信任状，以及对象的安全标记(用来代表该对象数据的机密性/完整性级别)。

TrustedBSD MAC 框架所提供的对策略模块的支持，不仅可以用来实现上述所有策略， 还能用于实现其他利用已有安全属性(如，用户和组ID、文件扩展属性等）决策的系统安全强化策略。 此外，因为具体策略模块在访问授权方面所拥有的高度灵活性和自主性，所以MAC 框架同样可以用来实现完全自主式的安全策略.

6.5. MAC 框架的内核体系结构

TrustedBSD MAC 框架为大多数的访问控制模块提供基本设施，允许它们以内核模块的形式灵活地扩展系统中实施的安全策略。 如果系统中同时加载了多个策略，MAC 框架将负责将各个策略的授权结果以一种（某种程度上）有意义的方式组合，形成最后的决策。

6.5.1. 内核元素

MAC 框架由下列内核元素组成:

- 框架管理接口
- 并发与同步原语
- 策略注册
- 内核对象的扩展性安全标记
- 策略入口函数的组合操作
- 标记管理原语
- 由内核服务调用的入口函数 API
- 策略模块的入口函数 API
- 入口函数的实现 (包括策略生命周期管理、标记管理和访问控制检查三部分)
- 管理策略无关标记的系统调用
- 复用的mac_syscall() 系统调用
- 以 MAC 的策略加载模块形式实现的各种安全策略

6.5.2. 框架管理接口

对 TrustedBSD MAC 框架进行直接管理的方式有三种:通过 sysctl 子系统、通过 loader 配置，或者使用系统调用。

多数情况下, 与同一个内核内部变量相关联的 sysctl 变量和 loader 参数的名字是相同的, 通过设置它们, 可以控制保护措施的实施细节, 比如, 某个策略在各个内核子系统中的实施与否等等。 另外, 如果在内核编译时选择支持 MAC 调试选项, 内核将维护若干计数器以跟踪标记的分配使用情况。 通常不建议在实用环境下通过在不同子系统上设置不同的变量或参数来实施控制, 因为这种方法将会作用于系统中所有的活跃策略。 如果希望对具体策略实施管理而不相影响其他活跃策略,则应当使用策略级别的控制, 因为这种方法的控制粒度更细, 并能更好地保证策略模块的功能一致性。

与其他内核模块一样, 系统管理员可以通过系统的模块管理系统调用和其他系统接口, 包括 boot loader 变量, 对策略模块执行加载与卸载操作; 策略模块可以在加载时,设置加载标志,来指示系统对其加载、卸载操作进行相应控制, 比如阻止非期望的卸载操作。

6.5.3. 策略链表的并发与同步

在运行时,系统中活跃的策略集合可能发生变化,然而对策略入口函数的使用操作并不是原子性的,因此,当某一个入口函数正被使用时, 系统需要提供额外的同步机制来阻止对该策略模块的加载与卸载,以确保当前活跃的策略集合不会在此过程中发生改变。 通过使用"框架忙" 计数器,就可以做到这一点:一旦某个入口函数被调用,计数器的值被增加1;而每当一个入口函数调用结束时,计数器的值被减少1。 检查计数器的值,如果其值为正,框架将阻止对策略链表的修改操作,请求操作的线程将被迫进入睡眠,直到计数器的值重新减少到0为止。 计数器本身由一个互斥锁保护,同时结合一个条件变量(用于唤醒等待对策略链表进行修改操作的睡眠线程)。 采用这种同步模型的一个副作用是,在同一个策略模块内部,允许嵌套地调用框架,不过这种情况其实很少出现。

为了减少由于采用计数器引入的额外开销,设计者采用了各种优化措施。其中包括,当策略链表为空或者其中仅含有静态表项 (那些只能在系统运行之前加载而且不能动态卸载的策略)时,框架不对计数器进行操作,其值总是为0,从而将此时的同步开销减到0。 另一个极端的办法是,使用一个编译选项来禁止在运行时对加载的策略链表进行修改,此时不再需要对策略链表的使用进行同步保护。

因为 MAC 框架不允许在某些入口函数之内阻塞,所以不能使用普通的睡眠锁。 故而,加载或卸载操作可能会为等待框架空闲而被阻塞相当长的一段时间。

6.5.4. 标记同步

MAC 框架必须对其负责维护的安全属性标记的存储访问提供同步保护。下列两种情形,可能导致对安全属性标记的不一致访问: 第一,作为安全属性标记的持有者,内核对象本身可能同时被多个线程访问;第二,MAC 框架代码是可重入的,即允许多个线程同时在框架内执行。通常,MAC 框架使用内核对象数据上已有的内核同步机制来保护该其上附加的 MAC 安全标记。 例如,套接字上的 MAC 标记由已有的套接字互斥锁保护。类似的,对于安全标记的并发访问的过程与对其所在对象进行的并发访问在语义上是一样的, 例如,信任状安全标记,将保持与该数据结构中其他内容一致的"写时复制"的更新过程。 MAC 框架在引用一个内核对象时,将首先对访问该对象上的标记需要用到的锁进行断言。 策略模块的编写者必须了解这些同步语义, 因为它们可能会限制对安全标记所能进行的访问类型。 举个例子,如果通过入口函数传给策略模块的是对某个信任状的只读引用,那么在策略内部,只能读该结构对应的标记状态。

6.5.5. 策略间的同步与并发

FreeBSD 内核是一个可抢占式的内核,因此,作为内核一部分的策略模块也必须是可重入的,也就是说, 在开发策略模块时必须假设多个内核线程可以同时通过不同的入口函数进入该模块。 如果策略模块使用可被修改的内核状态,那么还需要在策略内部使用恰当的同步原语,确保在策略内部的多个线程不会因此观察到不一致的内核状态, 从而避免由此产生的策略误操作。为此,策略可以使用 FreeBSD 现有的同步原语,包括互斥锁、睡眠锁、条件变量和计数信号量。 对这些同步

原语的使用必须慎重，需要特别注意两点：第一，保持现有的内核上锁次序； 第二，在非睡眠的入口函数之内不要使用互斥锁和唤醒操作。

为避免违反内核上锁次序或造成递归上锁，策略模块在调用其他内核子系统之前，通常要释放所有在策略内部申请的锁。 这样做的结果是，在全局上锁次序形成的拓朴结构中，策略内部的锁总是作为叶子节点， 从而保证了这些锁的使用不会导致由于上锁次序混乱造成的死锁。

6.5.6. 策略注册

为了记录当前使用的策略模块集合，MAC 框架维护两个链表：一个静态链表和一个动态链表。 两个链表的数据结构和操作基本相同，只是动态链表还额外使用了一个"引用计数"以同步对其的访问操作。 当包含 MAC 框架策略的内核模块被加载时，该策略模块会通过 SYSINIT 调用一个注册函数； 相对应的，每当一个策略模块被卸载，SYSINIT 也会调用一个注销函数。 只有当遇到下列情况之一时，注册过程才会失败： 一个策略模块被加载多次，或者系统资源不足不能满足注册过程的需要（ 例如，策略模块需要对内核对象添加标记而可用资源不足），或者其他的策略加载前提条件不满足（有些策略要求只能在系统引导之前加载）。 类似的，如果一个策略被标记为不可卸载的，对其调用注销过程将会失败。

6.5.7. 入口函数

内核服务与 MAC 框架之间进行交互有两种途径： 一是，内核服务调用一系列 API 通知 MAC 框架安全事件的发生； 二是，内核服务向 MAC 框架提供一个指向安全对象的策略无关安全标记数据结构的指针。 标记指针由 MAC 框架经由标记管理入口函数进行维护， 并且，只要对管理相关对象的内核子系统稍作修改，就可以允许 MAC 框架向策略模块提供标记服务。 例如，在进程、进程信任状、套接字、管道、Mbuf、网络接口、IP 重组队列和其他各种安全相关的数据结构中均增加了指向安全标记的指针。 另外，当需要做出重要的安全决策时，内核服务也会调用 MAC 框架，以便各个策略模块根据其自己的标准（可以使用存储在安全标记中的数据）完善这些决策。 绝大多数安全相关的关键决策是显式的访问控制检查； 也有少数涉及更加一般的决策函数，比如，套接字的数据包匹配和程序执行时刻的标记转换。

6.5.8. 策略组合

如果内核中同时加载了多个策略模块，这些策略的决策结果将由框架使用一个合成运算子来进行组合汇总，得出最终的结果。 目前，该算子是硬编码的，并且只有当所有的活跃策略均对请求表示同意时才会返回成功。 由于各个策略返回的出错条件可能并不相同（成功、访问被拒绝、请求对象不存在等等）， 需要使用一个选择子先从各个策略返回的错误条件集合中选择出一个作为最终返回结果。 一般情况下，与“访问被拒绝”相比，将更倾向于选择“请求对象不存在”。 尽管不能从理论上保证合成结果的有效性与安全性，但试验结果表明，对于许多实用的策略集合来说，事实的确如此。 例如，传统的可信系统常常采用类似的方法对多个安全策略进行组合。

6.5.9. 标记支持

与许多需要给对象添加安全标记的访问控制扩展一样，MAC 框架为各种用户可见的对象提供了一组用于管理策略无关标记的系统调用。 常用的标记类型有，partition标识符、机密性标记、完整性标记、区间（非等级类别）、域、角色和型。 “策略无关”的意思是指，标记的语法与使用它的具体策略模块无关，而同时策略模块能够完全独立地定义和使用与对象相关联的元数据的语义。 用户应用程序提供统一格式的基于字符串的标记，由使用它的策略模块负责解析其内在含义并决定其外在表示。 如果需要，应用程序可以使用多重标记元素。

内存中的标记实例被存放在由 slab 分配的struct label数据结构中。 该结构是一个固定长度的数组，每个元素是由一个 void * 指针和一个 long组成的联合结构。 申请标记存储的策略模块在向 MAC 注册时，将被分配一个“slot”值，作为框架分配给其使用的策略标记元素在整个标记存储结构中的位置索引。 而所分配的存储空间的语义则完全由该策略模块来决定：MAC 框架向策略模块提供了一系列入口函数用于对内核对象生命周期的各种事件进行控制，包括， 对象的初始化、标记的关联/创建和对象的注销。使用这些接口，可以实现诸如访问计数等存储模型。 MAC 框架总是给入口函数传入一个指向相关对象的指针和一个指向该对象标记的指针，因此，策略模块能够直接访问标记而无需知悉该对象的内部结构。 唯一的例外是进程信任状结构，指向其标记的指针必须由策略模块手动解析计算。今后的 MAC 框架实现可能会对此进行改进。

初始化入口函数通常有一个睡眠标志位，用来表明一个初始化操作是否允许中途睡眠等待； 如果不允许，则可能会失败返回，并要求撤销此次标记分配操作（乃至对象分配操作）。 例如，如果在网络栈上处理中断时因为不允许睡眠或者调用者持有一个互斥锁，就可能出现这种情况。 考虑到在处理中的网络数据包（Mbufs）上维护标记的性能损失太大，策略必须就自己对 Mbuf 进行标记的要求向 MAC 框架做出特别声明。 动态加载到系统中而又使用标记的策略必须为处理未被其初始化函数处理过的对象作好准备， 这些对象在策略加载之前就已经存在，故而无法在初始化时调用策略的相关函数进行处理。 MAC 框架向策略保证，没有被初始化的标记 slot 的值必为0或者 NULL，策略可以借此检测到未初始化的标记。 需要注意的是，因为对 Mbuf 标记的存储分配是有条件的，因此需要使用其标记的动态加载策略还可能需要处理 Mbuf 中值为 NULL 的标记指针。

对于文件系统对象的标记，MAC 框架在文件的扩展属性中为其分配永久存储。 只要可能，扩展属性的原子化的事务操作就被用于保证对 vnode 上安全标记的复合更新操作的一致性－－目前，该特性只被 UFS2 文件系统支持。 为了实现细粒度的文件系统对象标记（即每个文件系统对象一个标记），策略编写者可能选择使用一个（或者若干）扩展属性块。 为了提高性能， vnode 数据结构中有一个标记 (v_label)字段，用作磁盘标记的缓冲； vnode 结构实例化时，策略可以将标记值装入该缓冲，并在需要时对其进行更新。 如此，不必在每次进行访问控制检查时，均无条件地访问磁盘上的扩展属性。

> **注意**
>
> 目前，如果一个使用标记的策略允许被动态卸载，则卸载该模块之后,其状态 slot 尚无法被系统回收重用， 由此导致了 MAC 框架对标记策略卸载－重载操作数目上的严格限制。

6.5.10. 相关系统调用

MAC 框架向应用程序提供了一组系统调用：其中大多数用于向进行查询和修改策略无关标记操作的应用 API提供支持。

这些标记管理系统调用，接受一个标记描述结构， struct mac，作为输入参数。 这个结构的主体是一个数组，其中每个元素包含了一个应用级的 MAC 标记形式。每个元素又由两部分组成:一个字符串名字，和其对应的值。 每个策略可以向系统声明一个特定的元素名字，这样一来，如果需要，就可以将若干个相互独立的元素作为一个整体进行处理。 策略模块经由入口函数，在内核标记和用户提供的标记之间作翻译转换的工作，这种实现提供了标记元素语义上的高度灵活性。 标记管理系统调用通常有对应的库函数包装，这些包装函数可以提供内存分配和错误处理功能，从而简化了用户应用程序的标记管理工作。

目前的FreeBSD 内核提供了下列 MAC 相关的系统调用：

- mac_get_proc() 用于查询当前进程的安全标记。
- mac_set_proc() 用于请求改变当前进程的安全标记。
- mac_get_fd() 用于查询由文件描述符所引用的对象（ 文件、 套接字、 管道文件等等） 的安全标记。
- mac_get_file() 用于查询由文件系统路径所描述的对象的安全标记。
- mac_set_fd() 用于请求改变由文件描述符所引用的对象（ 文件、套接字、 管道文件等等） 的安全标记。
- mac_set_file() 用于请求改变由文件系统路径所描述的对象的安全标记。
- mac_syscall() 通过复用该系统调用,策略模块能够在不修改系统调用表的前提下创建新的系统调用； 其调用参数包括：目标策略名字、 操作编号和将被该策略内部使用的参数。
- mac_get_pid() 用于查询由进程号指定的另一个进程的安全标记。

- mac_get_link() 与 mac_get_file() 功能相同， 只是当路径参数的最后一项为符号链接时， 前者将返回该符号链接的安全标记， 而后者将返回其所指文件的安全标记。
- mac_set_link() 与 mac_set_file() 功能相同， 只是当路径参数的最后一项为符号链接时， 前者将设置该符号链接的安全标记， 而后者将设置其所指文件的安全标记。
- mac_execve() 与 execve() 功能类似， 只是前者还可以在开始执行一个新程序时，根据传入的请求参数,设置执行进程的安全标记。 由于执行一个新程序而导致的进程安全标记的改变,被称为 “转换” 。
- mac_get_peer()， 通过一个套接字选项自动实现， 用于查询一个远程套接字对等实体的安全标记。

除了上述系统调用之外， 也可以通过 SIOCSIGMAC 和 SIOCSIFMAC 网络接口的 ioctl 类系统调用来查询和设置网络接口的安全标记。

6.6. MAC策略模块体系结构

安全策略可以直接编入内核，也可以编译成独立的内核模块，在系统引导时或者运行时使用模块加载命令加载。 策略模块通过一组预先定义好的入口函数与系统交互。通过它们，策略模块能够掌握某些系统事件的发生，并且在必要的时候影响系统的访问控制决策。 每个策略模块包含下列组成部分：

- 可选：策略配置参数
- 策略逻辑和参数的集中实现
- 可选：策略生命周期事件的实现，比如，策略的初始化和销毁
- 可选：对所选内核对象的安全标记进行初始化、维护和销毁的支持
- 可选：对所选对象的使用进程进行监控以及修改对象安全标记的支持
- 策略相关的访问控制入口函数的实现
- 对策略标志、模块入口函数和策略特性的声明

6.6.1. 策略注销

策略模块可以使用 MAC_POLICY_SET() 宏来声明。 该宏完成以下工作：为该策略命名（向系统声明该策略提供的名字）；提交策略定义的 MAC 入口函数向量的地址； 按照策略的要求设置该策略的加载标志位，保证 MAC 框架将以策略所期望的方式对其进行操作； 另外，还可能请求框架为策略分配标记状态 slot 值。

```
static struct mac_policy_ops mac_policy_ops =
{
```

```
        -.mpo_destroy = mac_policy_destroy,
        -.mpo_init = mac_policy_init,
        -.mpo_init_bpfdesc_label = mac_policy_init_bpfdesc_label,
        -.mpo_init_cred_label = mac_policy_init_label,
/* -... */
        -.mpo_check_vnode_setutimes = mac_policy_check_vnode_setutimes,
        -.mpo_check_vnode_stat = mac_policy_check_vnode_stat,
        -.mpo_check_vnode_write = mac_policy_check_vnode_write,
};
```

如上所示，MAC 策略入口函数向量，mac_*policy*_ops，将策略模块中定义的功能函数挂接到特定的入口函数地址上。 在稍后的“入口函数参考”小节中，将提供可用入口函数功能描述和原型的完整列表。 与模块注册相关的入口函数有两个：.mpo_destroy和.mpo_init。 当某个策略向模块框架注册操作成功时，.mpo_init将被调用，此后其他的入口函数才能被使用。 这种特殊的设计使得策略有机会根据自己的需要，进行特定的分配和初始化操作，比如对特殊数据或锁的初始化。 卸载一个策略模块时，将调用 .mpo_destroy 用来释放策略分配的内存空间或注销其申请的锁。 目前，为了防止其他入口函数被同时调用，调用上述两个入口函数的进程必须持有 MAC 策略链表的互斥锁：这种限制将被放开， 但与此同时，将要求策略必须谨慎使用内核原语,以避免由于上锁次序或睡眠造成死锁。

之所以向策略声明提供模块名字域，是为了能够唯一标识该模块，以便解析模块依赖关系。选择使用恰当的字符串作为名字。 在策略加载和卸载时，策略的完整字符串名字将经由内核日志显示给用户。另外，当向用户进程报告状态信息时也会包含该字符串。

6.6.2. 策略标志

在声明时提供标志参数域的机制，允许策略模块在作为模块被加载时，就自身特性向 MAC 框架提供说明。 目前，已经定义的标志有三个：

MPC_LOADTIME_FLAG_UNLOADOK
: 表示该策略模块可以被卸载。 如果未提供该标志，则表示该策略模块拒绝被卸载。 那些使用安全标记的状态，而又不能在运行时释放该状态的模块可能会设置该标志。

MPC_LOADTIME_FLAG_NOTLATE
: 表示该策略模块必须在系统引导过程时进行加载和初始化。 如果该标志被设置，那么在系统引导之后注册该模块的请求将被 MAC 框架所拒绝。 那些需要为大范围的系统对象进行安全标记初始化工作，而又不能处理含有未被正确初始化安全标记的对象的策略模块可能会设置该标志。

MPC_LOADTIME_FLAG_LABELMBUFS
: 表示该策略模块要求为 Mbuf 指定安全标记，并且为存储其标记所需的内存空间总是提前分配好的。 缺省情况下，MAC 框架并不会为 Mbuf 分配标记存储，除非系统中注册的策略模块中至少有一个设置了该标志。 这种做法在没有策略需要对 Mbuf 做标记时，显著地提升了系统网络性能。另外，在某些特殊环

境下，可以通过设置内核选项， MAC_ALWAYS_LABEL_MBUF，强制 MAC 框架为 Mbuf 的安全标记分配存储，而不论上述标志如何设置。

> **注意**
>
> 那些使用了 MPC_LOADTIME_FLAG_LABELMBUFS 标志但没有设置 MPC_LOADTIME_FLAG_NOTLATE 标志的 策略模块必须能够正确地处理通过入口函数传入的值为 NULL 的 Mbuf 安全标记指针。 这是因为那些没有分配标记存储的处理中的 Mbuf 在一个需要 Mbuf 安全标记的策略模块加载之后， 其安全标记的指针将仍然为空。 如果策略在网络子系统活跃之前被加载（即，该策略不是被推迟加载的），那么所有的 Mbuf 的标记存储的分配就可以得到保证。

6.6.3. 策略入口函数

MAC 框架为注册的策略提供四种类型的入口函数： 策略注册和管理入口函数； 用于处理内核对象声明周期事件，如初始化、 创建和销毁，的入口函数； 处理该策略模块感兴趣的访问控制决策事件的入口函数； 以及用于管理对象安全标记的调用入口函数。 此外， 还有一个 mac_syscall() 入口函数， 被策略模块用于在不注册新的系统调用的前提下， 扩展内核接口。

策略模块的编写人员除了必须清楚在进入特定入口函数之后， 哪些对象锁是可用的之外， 还应该熟知内核所采用的加锁策略。 编程人员在入口函数之内应该避免使用非叶节点锁， 并且遵循访问和修改对象时的加锁规程， 以降低导致死锁的可能性。 特别地， 程序员应该清楚， 虽然在通常情况下， 进入入口函数之后， 已经上了一些锁， 可以安全地访问对象及其安全标记， 但是这并不能保证对它们进行修改（ 包括对象本身和其安全标记） 也是安全的。 相关的上锁信息，可以参考 MAC 框架入口函数的相关文档。

策略入口函数把两个分别指向对象本身和其安全标记的指针传递给策略模块。 这样一来，即使策略并不熟悉对象内部结构，也能基于标记作出正确决策。 只有进程信任状这个对象例外：MAC 框架总是假设所有的策略模块是理解其内部结构的。

6.7. MAC策略入口函数参考

6.7.1. 通用的模块入口函数

6.7.1.1. **mpo_init**

void **mpo_init(conf)**;

struct mac_policy_conf ***conf**;

参数	说明	锁定
conf	MAC 策略定义	

策略加载事件。当前进程正持有策略链表上的互斥锁，因此是非睡眠的，对其他内核子系统的调用也须慎重。　如果需要在策略初始化阶段进行可能造成睡眠阻塞的存储分配操作，可以将它们放在一个单独的模块 SYSINIT() 过程中集中进行。

6.7.1.2. mpo_destroy

void **mpo_destroy**(**conf**);

struct mac_policy_conf ***conf**;

参数	说明	锁定
conf	MAC 策略定义	

策略加载事件。必须持有策略链表互斥锁，因此需要慎重行事。

6.7.1.3. mpo_syscall

int **mpo_syscall**(**td**, **call**, **arg**);

struct thread ***td**;
int **call**;
void ***arg**;

参数	说明	锁定
td	调用	

参数	说明	锁定
td	线程	
call	策略特有的系统调用编号	
arg	系统调用参数的指针	

该入口函数提供策略复用的系统调用，这样策略模块不需要为其向用户进程提供的每一个额外服务而注册专用的系统调用。 由应用程序提供的策略注册名字来确定提供其所申请服务的特定策略，所有参数将通过该入口函数传递给被调用的策略。当实现新服务时，安全模块必须在必要时通过 MAC 框架调用相应的访问控制检查机制。 比方说，假如一个策略实现了某种额外的信号功能，那么它应该调用相关的信号访问控制检查，以接受 MAC 框架中注册的其他策略的检查。

> **注意**
>
> 不同的模块需要并发地手动进行copyin()拷贝系统调用数据。

6.7.1.4. **mpo_thread_userret**

void **mpo_thread_userret**(**td**);

struct thread ***td**;

参数	说明	锁定
td	返回线程	

使用该入口函数，策略模块能够在线程返回用户空间时（系统调用返回、异常返回等等）进行 MAC 相关的处理工作。 使用动态进程标记的策略需要使用该入口函数，因为在处理系统调用的过程中，并不是在任意时刻都能申请到进程锁的； 进程的标记可能表示传统的认证信息、进程历史记录或者其他数据。为使用该入口函数，对进程信任状所作的修改 可能被存放在 p_label ,该域受一个进程级自旋锁的保护；接下来，设置线程级的TDF_ASTPENDING 标志位和进程级的PS_MACPENDM标志位，表明将调度一个对 userret 入口函数的调用。通过该入口函数， 策略可以在相对简单的同步上下文中创建信任状的替代品。策略编程人员必须清楚，需要保证与调度一个 AST 相关的事件执行次序， 同时所执行的 AST 可能很复杂，而且在处理多线程应用程序时可能被重入。

6.7.2. 操作标记

6.7.2.1. **mpo_init_bpfdesc_label**

void **mpo_init_bpfdesc_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	将被应用的新标记	

为一个新近实例化的 bpfdesc（BPF 描述子）初始化标记。可以睡眠。

6.7.2.2. **mpo_init_cred_label**

void **mpo_init_cred_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	将被初始化的新标记	

为一个新近实例化的用户信任状初始化标记。可以睡眠。

6.7.2.3. mpo_init_devfsdirent_label

void **mpo_init_devfsdirent_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	将被应用的新标记	

为一个新近实例化的 devfs 表项初始化标记。可以睡眠。

6.7.2.4. mpo_init_ifnet_label

void **mpo_init_ifnet_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	将被应用的	

新标记

为一个新近实例化的网络接口初始化标记。可以睡眠。

6.7.2.5. **mpo_init_ipq_label**

void **mpo_init_ipq_label**(**label**, **flag**);

struct label ***label**;
int **flag**;

label	将被应用的新标记
flag	睡眠/不睡眠 malloc(9)；参见下文

为一个新近实例化的 IP 分片重组队列初始化标记。其中的*flag*域可能取M_WAITOK或M_NOWAIT之一，用来避免在该初始化调用中因为 malloc(9) 而进入睡眠。IP 分片重组队列的分配操作通常是在 对性能有严格要求的环境下进行的，因此实现代码必须小心地避免睡眠和长时间的操作。IP 分片重组队列分配操作失败时上述入口函数将失败返回。

6.7.2.6. **mpo_init_mbuf_label**

void **mpo_init_mbuf_label**(**flag**, **label**);

int **flag**;
struct label ***label**;

参数	说明	锁定
flag	睡眠/不睡眠 malloc(9); 参见下文	
label	将被初始化的策略标记	

为一个新近实例化的 mbuf 数据包头部（*mbuf*）初始化标记。 其中的*flag*的值可能取M_WAITOK和M_NOWAIT之一， 用来避免在该初始化调用中因为 malloc(9) 而进入睡眠。Mbuf 头部的分配操作常常在对性能有严格要求的环境下被频繁执行， 因此实现代码必须小心地避免睡眠和长时间的操作。上述入口函数在 Mbuf 头部分配操作失败时将失败返回。

6.7.2.7. mpo_init_mount_label

void **mpo_init_mount_label**(**mntlabel**, **fslabel**);

struct label ***mntlabel**;
struct label ***fslabel**;

参数	说明	锁定
mntlabel	将被	

参数	说明	锁定
	初始化的 mount 结构策略标记	
fslabel	将被初始化的文件系统策略标记	

为一个新近实例化的 mount 点初始化标记。可以睡眠。

6.7.2.8. mpo_init_mount_fs_label

void **mpo_init_mount_fs_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	将被初始化的	

参数	说明	锁定
	标记	

为一个新近加载的文件系统初始化标记。可以睡眠。

6.7.2.9. mpo_init_pipe_label

void **mpo_init_pipe_label**(**label**);

struct label***label**;

参数	说明	锁定
label	将被填写的标记	

为一个刚刚实例化的管道初始化安全标记。可以睡眠。

6.7.2.10. mpo_init_socket_label

void **mpo_init_socket_label**(**label**, **flag**);

struct label ***label**;
int **flag**;

参数	说明	锁定
label	将被初始化的新标记	

参数	说明	锁定
flag	malloc(9) flags	

为一个刚刚实例化的套接字初始化安全标记。其中的 *flag* 域的值必须被指定为M_WAITOK和M_NOWAIT之一，以避免在该初始化程中使用可能睡眠的malloc(9) 。

6.7.2.11. mpo_init_socket_peer_label

void **mpo_init_socket_peer_label**(**label**, **flag**);

struct label ***label**;
int **flag**;

参数	说明	锁定
label	将被初始化的新标记	
flag	malloc(9) flags	

为刚刚实例化的套接字对等体进行标记的初始化。其中的 *flag* 域的值必须被指定为 M_WAITOK 和 M_NOWAIT 之一，以避免在该初始化程中使用可能睡眠的malloc(9)。

6.7.2.12. mpo_init_proc_label

void **mpo_init_proc_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	将被	

参数	说明	锁定
	初始化的新标记	

为一个刚刚实例化的进程初始化安全标记。可以睡眠。

6.7.2.13. **mpo_init_vnode_label**

void **mpo_init_vnode_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	将被初始化的新标记	

为一个刚刚实例化的 vnode 初始化安全标记。可以睡眠。

6.7.2.14. **mpo_destroy_bpfdesc_label**

void **mpo_destroy_bpfdesc_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	bpfdesc 标记	

销毁一个 BPF 描述子上的标记。在该入口函数中，策略应当释放所有在内部分配与 *label* 相关联的存储空间，以便销毁该标记。

6.7.2.15. mpo_destroy_cred_label

void **mpo_destroy_cred_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	将被销毁的标记	

销毁一个信任状上的标记。在该入口函数中，策略应当释放所有在内部分配的与 *label* 相关联的存储空间，以便销毁该标记。

6.7.2.16. mpo_destroy_devfsdirent_label

void **mpo_destroy_devfsdirent_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	将被销毁的标记	

销毁一个 devfs 表项上的标记。在该入口函数中，策略应当释放所有在内部分配的与 *label* 相关联的存储空间，以便销毁该标记。

6.7.2.17. mpo_destroy_ifnet_label

void **mpo_destroy_ifnet_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	将被销毁的标记	

销毁与一个已删除接口相关联的标记。在该入口函数中，策略应当释放所有在内部分配的与 *label* 相关联的存储空间，以便销毁该标记。

6.7.2.18. mpo_destroy_ipq_label

void **mpo_destroy_ipq_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	将被销毁的标记	

销毁与一个 IP 分片队列相关联的标记。在该入口函数中，策略应当释放所有在内部分配的与 *label* 相关联的存储空间，以便销毁该标记。

6.7.2.19. mpo_destroy_mbuf_label

void **mpo_destroy_mbuf_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	将被销毁的	

参数	说明	锁定
标记		

销毁与一个 Mbuf 相关联的标记。在该入口函数中，策略应当释放所有在内部分配的与 *label* 相关联的存储空间，以便销毁该标记。

6.7.2.20. mpo_destroy_mount_label

void **mpo_destroy_mount_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	将被销毁的Mount点标记	

销毁与一个 mount 点相关联的标记。在该入口函数中，策略应当释放所有在内部分配的与 *mntlabel* 相关联的存储空间，以便销毁该标记。

6.7.2.21. mpo_destroy_mount_label

void **mpo_destroy_mount_label**(**mntlabel**, **fslabel**);

struct label ***mntlabel**;
struct label ***fslabel**;

参数	说明	锁定
mntlabel	将被销毁的Mount点	

参数	说明	锁定
	标记	
fslabel	file system label being destroyed>	

销毁与一个 mount 点相关联的标记。在该入口函数中，策略应当释放所有在内部分配的，与 *mntlabel* 和*fslabel* 相关联的存储空间，以便销毁该标记。

6.7.2.22. mpo_destroy_socket_label

void **mpo_destroy_socket_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	将被销毁的套接字标记	

销毁与一个套接字相关联的标记。在该入口函数中，策略应当释放所有在内部分配的，与 *label* 相关联的存储空间，以便销毁该标记。

6.7.2.23. mpo_destroy_socket_peer_label

void **mpo_destroy_socket_peer_label**(**peerlabel**);

struct label ***peerlabel**;

参数	说明	锁定
peerlabel	将被	

参数	说明	锁定
	销毁的套接字对等实体标记	

销毁与一个套接字相关联的对等实体标记。在该入口函数中，策略应当释放所有在内部分配的，与 *label* 相关联的存储空间，以便销毁该标记。

6.7.2.24. mpo_destroy_pipe_label

void **mpo_destroy_pipe_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	管道标记	

销毁一个管道的标记。在该入口函数中，策略应当释放所有在内部分配的，与 *label* 相关联的存储空间，以便销毁该标记。

6.7.2.25. mpo_destroy_proc_label

void **mpo_destroy_proc_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	进程	

参数	说明	锁定
	标记	

销毁一个进程的标记。在该入口函数中，策略应当释放所有在内部分配的，与 *label* 相关联的存储空间，以便销毁该标记。

6.7.2.26. mpo_destroy_vnode_label

void **mpo_destroy_vnode_label**(**label**);

struct label ***label**;

参数	说明	锁定
label	进程标记	

销毁一个 vnode 的标记。在该入口函数中，策略应当释放所有在内部分配的，与 *label* 相关联的存储空间，以便销毁该标记。

6.7.2.27. mpo_copy_mbuf_label

void **mpo_copy_mbuf_label**(**src**, **dest**);

struct label ***src**;
struct label ***dest**;

参数	说明	锁定
src	源标记	
dest	目标标记	

将 *src* 中的标记信息拷贝到 *dest*中。

6.7.2.28. **mpo_copy_pipe_label**

void **mpo_copy_pipe_label**(**src**, **dest**);

struct label ***src**;
struct label ***dest**;

参数	说明	锁定
src	源标记	
dest	目标标记	

将 *src* 中的标记信息拷贝至 *dest*。

6.7.2.29. **mpo_copy_vnode_label**

void **mpo_copy_vnode_label**(**src**, **dest**);

struct label ***src**;
struct label ***dest**;

参数	说明	锁定
src	源标记	
dest	目标标记	

将 *src* 中的标记信息拷贝至 *dest*。

6.7.2.30. **mpo_externalize_cred_label**

int **mpo_externalize_cred_label**(**label**, **element_name**, **sb**, ***claimed**);

struct label ***label**;
char ***element_name**;
struct sbuf ***sb**;

int ***claimed**;

参数	说明	锁定
label	将要用外部形式表示的标记	
element_name	需要外部表示标记的策略的名字	
sb	用来存放标记的文本表示形式的	

参数	说明	锁定
	字符buffer	
claimed	如果可以填充element_data域，则其数值递增	

根据传入的标记结构，产生一个以外部形式表示的标记。 一个外部形式标记，是标记内容的文本表示，它由用户级的应用程序使用，是用户可读的。 目前的MAC实现方案将依次调用策略的相应入口函数，因此， 具体策略的实现代码，需要在填写sb之前，先检查element_name中指定的名字。 如果element_name中的内容与你的策略名字不相符，则直接返回0。 仅当转换标记数据的过程中出现错误时，才返回非0值。 一旦策略决定填写element_data，递增*claim的数值。

6.7.2.31. mpo_externalize_ifnet_label

int **mpo_externalize_ifnet_label**(**label**, **element_name**, **sb**, ***claimed**);

struct label ***label**;
char ***element_name**;
struct sbuf ***sb**;
int ***claimed**;

参数	说明	锁定
label	将用外部形式表	

参数	说明	锁定
	示的标记	
element_name	需要外部表示标记的策略的名字	
sb	用来存放标记的文本表示形式的字符 buffer	
claimed	如果可以填充	

参数	说明	锁定
	element_data域，则其数值递增	

根据传入的标记结构，产生一个以外部形式表示的标记。 一个外部形式标记，是标记内容的文本表示，它由用户级的应用程序使用，是用户可读的。 目前的MAC实现方案将依次调用策略的相应入口函数，因此， 具体策略的实现代码，需要在填写sb之前，先检查element_name中指定的名字。 如果element_name中的内容与你的策略名字不相符，则直接返回0。 仅当转换标记数据的过程中出现错误时，才返回非0值。 一旦策略决定填写element_data，递增*claim的数值。

6.7.2.32. mpo_externalize_pipe_label

int **mpo_externalize_pipe_label**(**label**, **element_name**, **sb**, ***claimed**);

struct label ***label**;
char ***element_name**;
struct sbuf ***sb**;
int ***claimed**;

参数	说明	锁定
label	将用外部形式表示的标记	
element_name	需要外部表	

参数	说明	锁定
	示标记的策略的名字	
sb	用来存放标记的文本表示形式的字符 buffer	
claimed	如果可以填充 element_data 域，则其数值递增	

根据传入的标记结构，产生一个以外部形式表示的标记。 一个外部形式标记，是标记内容的文本表示，它由用户级的应用程序使用，是用户可读的。 目前的MAC实现方案将依次调用策略的相应入口函数，因此， 具体策略的实现代码，需要在填写sb之前，先检查element_name中指定的名字。 如果element_name中的内容与你的策略名字不相符，则直接返回0。 仅当转换标记数据的过程中出现错误时，才返回非0值。 一旦策略决定填写element_data，递增*claim的数值。

6.7.2.33. **mpo_externalize_socket_label**

int **mpo_externalize_socket_label**(**label**, **element_name**, **sb**, ***claimed**);

struct label ***label**;
char ***element_name**;
struct sbuf ***sb**;
int ***claimed**;

参数	说明	锁定
label	将要用外部形式表示的标记	
element_name	需要外部表示标记的策略的名字	

参数	说明	锁定
sb	用来存放标记的文本表示形式的字符buffer	
claimed	如果可以填充element_data域，则其数值递增	

根据传入的标记结构，产生一个以外部形式表示的标记。 一个外部形式标记，是标记内容的文本表示，它由用户级的应用程序使用，是用户可读的。 目前的MAC实现方案将依次调用策略的相应入口函数，因此， 具体策略的实现代码，需要在填写sb之前，先检查element_name中指定的名字。 如果element_name中的内容与你的策略名字不相符，则直接返回0。 仅当转换标记数据的过程中出现错误时，才返回非0值。 一旦策略决定填写element_data，递增*claim的数值。

6.7.2.34. mpo_externalize_socket_peer_label

int **mpo_externalize_socket_peer_label(label**, **element_name**, **sb**, ***claimed**);

struct label ***label**;
char ***element_name**;
struct sbuf ***sb**;
int ***claimed**;

参数	说明
label	将用外部形式表示的标记
element_name	需要外部表示标记的策略的名字
sb	用来存放标记的文本表示

参数	说明	锁定
	形式的字符buffer	
claimed	如果可以填充element_data域，则其数值递增	

根据传入的标记结构，产生一个以外部形式表示的标记。 一个外部形式标记，是标记内容的文本表示，它由用户级的应用程序使用，是用户可读的。 目前的MAC实现方案将依次调用策略的相应入口函数，因此， 具体策略的实现代码，需要在填写sb之前，先检查element_name中指定的名字。 如果element_name中的内容与你的策略名字不相符，则直接返回0。 仅当转换标记数据的过程中出现错误时，才返回非0值。 一旦策略决定填写element_data，递增*claim的数值。

6.7.2.35. mpo_externalize_vnode_label

int **mpo_externalize_vnode_label**(**label**, **element_name**, **sb**, ***claimed**);

struct label ***label**;
char ***element_name**;
struct sbuf ***sb**;
int ***claimed**;

参数	说明	锁定
label	将用外部	

[illegible] 形式表示的标记

element_name 需要外部表示标记的策略的名字

sb 用来存放标记的文本表示形式的字符 buffer

claimed 如果可

[illegible]
以填充element_data域，则其数值递增

根据传入的标记结构，产生一个以外部形式表示的标记。 一个外部形式标记，是标记内容的文本表示，它由用户级的应用程序使用，是用户可读的。 目前的MAC实现方案将依次调用策略的相应入口函数，因此， 具体策略的实现代码，需要在填写sb之前，先检查element_name中指定的名字。 如果element_name中的内容与你的策略名字不相符，则直接返回0。 仅当转换标记数据的过程中出现错误时，才返回非0值。 一旦策略决定填写element_data，递增*claim的数值。

6.7.2.36. **mpo_internalize_cred_label**

int **mpo_internalize_cred_label**(**label**, **element_name**, **element_data**, **claimed**);

struct label ***label**;
char ***element_name**;
char ***element_data**;
int ***claimed**;

[illegible]
label 将被填充的标记
element_name 需要内部

[illegible]	表示标记的策略的名字
element_data	需要被转换的文本数据
claimed	如果数据被正确转换，则其数值递增

根据一个文本形式的外部表示标记数据，创建一个内部形式的标记结构。 目前的MAC方案将依次调用所有策略的相关入口函数，来响应标记的内部转换请求， 因此，实现代码必须首先通过比较element_name中的内容和自己的策略名字， 来确定是否需要转换element_data中存放的数据。 类似的，如果名字不匹配或者数据转换操作成功，该函数返回0，并递增*claimed的值。

6.7.2.37. mpo_internalize_ifnet_label

int **mpo_internalize_ifnet_label**(**label**, **element_name**, **element_data**, **claimed**);

struct label ***label**;
char ***element_name**;
char ***element_data**;
int ***claimed**;

参数	说明	锁定
label	将被填充的标记	
element_name	需要内部表示标记的策略的名字	
element_data	需要被转换的文本数据	
claimed	如果	

参数	说明	锁定
	数据被正确转换，则其数值递增	

根据一个文本形式的外部表示标记数据，创建一个内部形式的标记结构。 目前的MAC方案将依次调用所有策略的相关入口函数，来响应标记的内部转换请求， 因此，实现代码必须首先通过比较`element_name`中的内容和自己的策略名字， 来确定是否需要转换`element_data`中存放的数据。 类似的，如果名字不匹配或者数据转换操作成功，该函数返回0，并递增`*claimed`的值。

6.7.2.38. mpo_internalize_pipe_label

int **mpo_internalize_pipe_label**(**label**, **element_name**, **element_data**, **claimed**);

struct label ***label**;
char ***element_name**;
char ***element_data**;
int ***claimed**;

参数	说明	锁定
label	将被填充的标记	
element_name	需要内部	

参数	说明	锁定
	表示标记的策略的名字	
element_data	需要被转换的文本数据	
claimed	如果数据被正确转换，则其数值递增	

根据一个文本形式的外部表示标记数据，创建一个内部形式的标记结构。 目前的MAC方案将依次调用所有策略的相关入口函数，来响应标记的内部转换请求， 因此，实现代码必须首先通过比较element_name中的内容和自己的策略名字， 来确定是否需要转换element_data中存放的数据。 类似的，如果名字不匹配或者数据转换操作成功，该函数返回0，并递增*claimed的值。

6.7.2.39. mpo_internalize_socket_label

int **mpo_internalize_socket_label**(**label**, **element_name**, **element_data**, **claimed**);

struct label ***label**;
char ***element_name**;
char ***element_data**;
int ***claimed**;

参数	说明	锁定
label	将被填充的标记	
element_name	需要内部表示标记的策略的名字	
element_data	需要被转换的文本数据	
claimed	如果	

参数	说明	锁定
	数据被正确转换，则其数值递增	

根据一个文本形式的外部表示标记数据，创建一个内部形式的标记结构。 目前的MAC方案将依次调用所有策略的相关入口函数，来响应标记的内部转换请求， 因此，实现代码必须首先通过比较element_name中的内容和自己的策略名字， 来确定是否需要转换element_data中存放的数据。 类似的，如果名字不匹配或者数据转换操作成功，该函数返回0，并递增*claimed的值。

6.7.2.40. mpo_internalize_vnode_label

int **mpo_internalize_vnode_label**(**label**, **element_name**, **element_data**, **claimed**);

struct label ***label**;
char ***element_name**;
char ***element_data**;
int ***claimed**;

参数	说明	锁定
label	将被填充的标记	
element_name	需要内部	

[illegible]	
	表示标记的策略的名字
element_data	需要被转换的文本数据
claimed	如果数据被正确转换，则其数值递增

根据一个文本形式的外部表示标记数据，创建一个内部形式的标记结构。 目前的MAC方案将依次调用所有策略的相关入口函数，来响应标记的内部转换请求， 因此，实现代码必须首先通过比较element_name中的内容和自己的策略名字， 来确定是否需要转换element_data中存放的数据。 类似的，如果名字不匹配或者数据转换操作成功，该函数返回0，并递增*claimed的值。

6.7.3. 标记事件

策略模块使用MAC 框架提供的“标记事件”类入口函数，对内核对象的标记进行操作。策略模块将感兴趣的被标记内核对象的相关生命周期事件 注册在恰当的入口点上。对象的初始化、创建和销毁事件均提供了钩子点。在某些对象上还可以实现重新标记，即，允许用户进程改变对象上的标记值。 对某些对象可以实现其特定的对象事件，比如与 IP 重组相关的标记事件。一个典型的被标记对象在其生命周期中将拥有下列入口函数：

```
#####          o
#########       \
####             o
                  \
#######            o--<--.
########           -|    -|
######             ~-->--o
                          \
####                       o
```

使用标记初始化入口函数，策略可以以一种统一的、与对象使用环境无关的方式设置标记的初始值。 分配给一个策略的缺省 slot 值为0，这样不使用标记的策略可能并不需要执行专门的初始化操作。

标记的创建事件发生在将一个内核数据结构同一个真实的内核对象相关联（内核对象实例化）的时刻。 例如，在真正被使用之前，在一个缓冲池内已分配的 mbuf 数据结构，将保持为“未使用”状态。 因此，mbuf 的分配操作将导致针对该 mbuf 的标记初始化操作，而 mbuf 的创建操作则被推迟到该 mbuf 真正与一个数据报相关联的时刻。 通常，调用者将会提供创建事件的上下文，包括创建环境、创建过程中涉及的其他对象的标记等。例如，当一个套接字创建一个 mbuf 时， 除了新创建的 mbuf 及其标记之外，作为创建者的套接字与其标记也被提交给策略检查。 不提倡在创建对象时就为其分配内存的原因有两个：创建操作可能发生在对性能有严格要求的内核接口上； 而且，因为创建调用不允许失败，所以无法报告内存分配失败。

对象特有的事件一般不会引发其他的标记事件，但是在对象上下文发生改变时，策略使用它们可以对相关标记进行修改或更新操作。 例如，在MAC_UPDATE_IPQ 入口函数之内，某个 IP 分片重组队列的标记可能会因为队列中接收了新的 mbuf 而被更新。

访问控制事件将在后续章节中详细讨论。

策略通过执行标记销毁操作，释放为其分配的存储空间或维护的状态，之后内核才可以重用或者释放对象的内核数据结构。

除了与特定内核对象绑定的普通标记之外，还有一种额外的标记类型：临时标记。 这些标记用于存放由用户进程提交的更新信息。 它们的初始化和销毁操作与其他标记一样，只是创建事件，MAC_INTERNALIZE，略有不同： 该函数接受用户提交的标记，负责将其转化为内核表示形式。

6.7.3.1. 文件系统对象标记事件操作

6.7.3.1.1. **mpo_associate_vnode_devfs**

void **mpo_associate_vnode_devfs**(**mp**, **fslabel**, **de**, **delabel**, **vp**, **vlabel**);

struct mount ***mp**;
struct label ***fslabel**;
struct devfs_dirent ***de**;
struct label ***delabel**;
struct vnode ***vp**;
struct label ***vlabel**;

参数	说明	锁定
mp	Devfs 挂载点	
fslabel	Devfs 文件系统标记（mp->mnt_fslabel）	
de	Devfs 目录项	
delabel	与 *de* 相关联的策略标记	

参数	说明	锁定
vp	与 *de* 相关联的 vnode	
vlabel	与 *vp* 相关联的策略标记	

根据参数 *de* 传入的 devfs 目录项及其标记信息，为一个新近创建的 devfs vnode 填充标记（*vlabel*）。

6.7.3.1.2. mpo_associate_vnode_extattr

int **mpo_associate_vnode_extattr**(**mp**, **fslabel**, **vp**, **vlabel**);

struct mount ***mp**;
struct label ***fslabel**;
struct vnode ***vp**;
struct label ***vlabel**;

参数	说明	锁定
mp	文件系统挂载点	
fslabel	文件	

参数	说明	锁定
	系统标记	
vp	将被标记的 vnode	
vlabel	与 vp 相关联的策略标记	

从文件系统扩展属性中读取 *vp* 的标记。成功，返回 0。 不成功，则在 errno 指定的相应的错误编码。 如果文件系统不支持扩展属性的读取操作，则可以考虑将 *fslabel* 拷贝至 *vlabel*。

6.7.3.1.3. mpo_associate_vnode_singlelabel

void **mpo_associate_vnode_singlelabel**(**mp**, **fslabel**, **vp**, **vlabel**);

struct mount ***mp**;
struct label ***fslabel**;
struct vnode ***vp**;
struct label ***vlabel**;

参数	说明	锁定
mp	文件系统挂	

参数	说明	锁定
	载点	
fslabel	文件系统标记	
vp	将被标记的 vnode	
vlabel	与 *vp* 相关联的策略标记	

在非多重标记文件系统上，使用该入口函数，根据文件系统标记，*fslabel*，为 *vp* 设置策略标记。

6.7.3.1.4. mpo_create_devfs_device

void **mpo_create_devfs_device**(**dev**, **devfs_dirent**, **label**);

dev_t **dev**;
struct devfs_dirent ***devfs_dirent**;
struct label ***label**;

参数	说明	锁定
devfs_dirent	对应	

参数	说明	锁定
	的设备	
devfs_dirent	将被标记的 Devfs 目录项	
label	将被填写的 *devfs_dirent* 标记	

为传入设备新建的 devfs_dirent 填写标记。该函数将在设备文件系统加载、重构或添加新设备时被调用。

6.7.3.1.5. mpo_create_devfs_directory

void **mpo_create_devfs_directory**(**dirname**, **dirnamelen**, **devfs_dirent**, **label**);

char ***dirname**;
int **dirnamelen**;
struct devfs_dirent ***devfs_dirent**;
struct label ***label**;

参数	说明	锁定
dirname	新建目录的名字	

参数	说明	锁定
namelen	字符串 *dirname* 的长度	
de	devfs_dirent 新建目录在 Devfs 中对应的目录项	

为传入目录参数的新建 devfs_dirent 填写标记。该函数将在加载、重构设备文件系统，或者添加一个需要指定目录结构的新设备时被调用。

6.7.3.1.6. mpo_create_devfs_symlink

void **mpo_create_devfs_symlink**(**cred**, **mp**, **dd**, **ddlabel**, **de**, **delabel**);

struct ucred ***cred**;
struct mount ***mp**;
struct devfs_dirent ***dd**;
struct label ***ddlabel**;
struct devfs_dirent ***de**;
struct label ***delabel**;

参数	说明	锁定
cred	主体信任状	

参数	说明	锁定
mp	devfs 挂载点	
dd	链接目标	
ddlabel	与 *dd* 相关联的标记	
de	符号链接项	
delabel	与 *de* 相关联的策略标记	

为新近创建的 devfs(5) 符号链接项填写标记（*delabel*）。

6.7.3.1.7. mpo_create_vnode_extattr

int **mpo_create_vnode_extattr**(**cred**, **mp**, **fslabel**, **dvp**, **dlabel**, **vp**, **vlabel**, **cnp**);

struct ucred ***cred**;
struct mount ***mp**;
struct label ***fslabel**;

struct vnode ***dvp**;
struct label ***dlabel**;
struct vnode ***vp**;
struct label ***vlabel**;
struct componentname ***cnp**;

参数	说明	锁定
cred	主体信任状	
mount	文件系统挂载点	
label	文件系统标记	
dvp	父目录 vnode	
dlabel	与 *dvp* 相关联的策略标记	
vp	新创	

参数	说明
	建的 vnode
vlabel	与 *vp* 相关联的策略标记
cnp	*dvp* 中的子域名字

将 *vp* 的标记写入文件扩展属性。成功，将标记填入 *vlabel*，并返回 0。否则，返回对应的错误编码。

6.7.3.1.8. **mpo_create_mount**

void **mpo_create_mount**(**cred**, **mp**, **mnt**, **fslabel**);

struct ucred ***cred**;
struct mount ***mp**;
struct label ***mnt**;
struct label ***fslabel**;

参数	说明
cred	主体信任状
mp	客体；将

参数	说明	锁定
	被挂载的文件系统	
mntlabel	将被填写的*mp*的策略标记	
fslabel	将被挂载到*mp*的文件系统的策略标记。	

为传入的主体信任状所创建的挂载点填写标记。该函数将在文件系统挂载时被调用。

6.7.3.1.9. mpo_create_root_mount

void **mpo_create_root_mount(cred**, **mp**, **mntlabel**, **fslabel**);

struct ucred ***cred**;
struct mount ***mp**;
struct label ***mntlabel**;
struct label ***fslabel**;

参数	说明	锁定
见第6.7.3.1.8节“mpo_create_mount”.		

为传入的主体信任状所创建的挂载点填写标记。该函数将在挂载根文件系统时，mpo_create_mount；之后被调用。

6.7.3.1.10. mpo_relabel_vnode

void **mpo_relabel_vnode**(**cred**, **vp**, **vnodelabel**, **newlabel**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***vnodelabel**;
struct label ***newlabel**;

参数	说明	锁定
cred	主体信任状	
vp	将被重新标记的vnode	
vnodelabel	现	

参数	说明	锁定
	有的策略标记	
newlabel	将取代*vnodelabel*的新（可能只是部分）标记	

根据传入的新标记和主体信任状，更新参数 vnode 的标记。

6.7.3.1.11. mpo_setlabel_vnode_extattr

int **mpo_setlabel_vnode_extattr**(**cred**, **vp**, **vlabel**, **intlabel**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***vlabel**;
struct label ***intlabel**;

参数	说明	锁定
cred	主体信任状	
vp	写出标记	

参数	说明	锁定
	所对应的 vnode	
vplabel	vp 的策略标记	
intlabel	将被写入磁盘的标记	

将参数 *intlabel* 给出的标记信息写入指定 vnode 的扩展属性。该函数被 vop_stdcreatevnode_ea 所调用。

6.7.3.1.12. mpo_update_devfsdirent

void **mpo_update_devfsdirent**(**devfs_dirent**, **direntlabel**, **vp**, **vnodelabel**);

struct devfs_dirent ***devfs_dirent**;
struct label ***direntlabel**;
struct vnode ***vp**;
struct label ***vnodelabel**;

参数	说明	锁定
devfs_dirent	对象体; devfs 目录项	

direntlabel	将被更新的*devfs_dirent*的策略标记	
vp	父vnode	锁定
vnodelabel	*vp*的策略标记	

根据所传入的 devfs vnode 标记，对 ***devfs_dirent*** 的标记进行更新。 重新标记一个 devfs vnode 的操作成功之后，将调用该函数来确认标记的改变，如此，即使相应的 vnode 数据结构被内核回收重用， 也不会丢失标记的新状态。另外，在 devfs 中新建一个符号链接时，紧接着mac_vnode_create_from_vnode， 也将调用该函数，对 vnode 标记进行初始化操作。

6.7.3.2. IPC 对象标记事件操作

6.7.3.2.1. mpo_create_mbuf_from_socket

void **mpo_create_mbuf_from_socket**(**so**, **socketlabel**, **m**, **mbuflabel**);

struct socket ***so**;
struct label ***socketlabel**;
struct mbuf ***m**;
struct label ***mbuflabel**;

socket	套接字	锁

参数	说明	锁定
		锁定 WIP
socketlabel	*socket*的策略标记	
m	客体；mbuf	
mbuflabel	将被填写的*m*的策略标记	

根据传入的套接字标记为新创建的mbuf头部设置标记。 每当套接字产生一个新的数据报或者消息，并将其存储在参数 mbuf 中时，将调用该函数。

6.7.3.2.2. mpo_create_pipe

void **mpo_create_pipe**(**cred**, **pipe**, **pipelabel**);

struct ucred ***cred**;
struct pipe ***pipe**;
struct label ***pipelabel**;

参数	说明	锁定
cred	主体信任状	

参数	说明	锁定
pipe	管道	
pipelabel	pipe 的策略标记	

根据传入的主体信任状参数，设置新建管道的标记。每当一个新管道被创建，该函数将被调用。

6.7.3.2.3. mpo_create_socket

void **mpo_create_socket**(**cred**, **so**, **socketlabel**);

struct ucred ***cred**;
struct socket ***so**;
struct label ***socketlabel**;

参数	说明	锁定
cred	主体信任状	不可改变
so	实体；将被标记的套接字	
socketlabel	将被填写	

参数	说明	锁定
	的 *so* 的标记	

根据传入的主体信任状参数，设置新建套接字的标记。每当新建一个套接字，该函数将被调用。

6.7.3.2.4. mpo_create_socket_from_socket

void **mpo_create_socket_from_socket**(**oldsocket**, **oldsocketlabel**, **newsocket**, **newsocketlabel**);

struct socket ***oldsocket**;
struct label ***oldsocketlabel**;
struct socket ***newsocket**;
struct label ***newsocketlabel**;

参数	说明	锁定
oldsocket	监听套接字	
oldsocketlabel	*oldsocket* 的策略标记	
newsocket	新建套接字	
newsocketlabel	*newsocket* 的策略	

参数	说明	锁定
	标记	

根据 listen(2) 套接字 *oldsocket*， 为新建 accept(2) 的套接字 *newsocket*，设置标记。

6.7.3.2.5. **mpo_relabel_pipe**

void **mpo_relabel_pipe**(**cred**, **pipe**, **oldlabel**, **newlabel**);

struct ucred ***cred**;
struct pipe ***pipe**;
struct label ***oldlabel**;
struct label ***newlabel**;

参数	说明	锁定
cred	主体信任状	
pipe	管道	
oldlabel	*pipe* 的当前策略标记	
newlabel	将为*pipe*设置的新的策略	

参数	说明	锁定
	标记	

为*pipe*设置新标记*newlabel*。

6.7.3.2.6. mpo_relabel_socket

void **mpo_relabel_socket**(**cred**, **so**, **oldlabel**, **newlabel**);

struct ucred ***cred**;
struct socket ***so**;
struct label ***oldlabel**;
struct label ***newlabel**;

参数	说明	锁定
cred	主体凭证	不可改变状
so	客体；套接字	
oldlabel	so的当前标记	
newlabel	将为*socket*设置的新标记	

根据传入的标记参数，对套接字的当前标记进行更新。

6.7.3.2.7. **mpo_set_socket_peer_from_mbuf**

void **mpo_set_socket_peer_from_mbuf**(**mbuf**, **mbuflabel**, **oldlabel**, **newlabel**);

struct mbuf ***mbuf**;
struct label ***mbuflabel**;
struct label ***oldlabel**;
struct label ***newlabel**;

参数	说明	锁定
mbuf	从套接字接收到的第一个数据报	
mbuflabel	mbuf的标记	
oldlabel	套接字的当前标记	
newlabel	将为套接字	

填写的策略标记	

根据传入的 mbuf 标记，设置某个 stream 套接字的对等标志。 除Unix域的套接字之外，每当一个 stream 套接字接收到第一个数据报时，该函数将被调用。

6.7.3.2.8. mpo_set_socket_peer_from_socket

void **mpo_set_socket_peer_from_socket**(**oldsocket**, **oldsocketlabel**, **newsocket**, **newsocketpeerlabel**);

struct socket ***oldsocket**;
struct label ***oldsocketlabel**;
struct socket ***newsocket**;
struct label ***newsocketpeerlabel**;

oldsocket	本地套接字
oldsocketlabel	*oldsocket*的策略标记
newsocket	对等套接字
newsocketpeerlabel	将为*newsocket*填

参数	说明	锁定
	写的策略标记	

根据传入的远程套接字端点，为一个 stream UNIX 与套接字设置对等标记。 每当相应的套接字对之间进行连接时，该函数将在两端分别被调用。

6.7.3.3. Network Object Labeling Event Operations

6.7.3.3.1. **mpo_create_bpfdesc**

void **mpo_create_bpfdesc**(**cred**, **bpf_d**, **bpflabel**);

struct ucred ***cred**;
struct bpf_d ***bpf_d**;
struct label ***bpflabel**;

参数	说明	锁定
cred	主体凭证	不可改变状
bpf_d	客体；bpf描述子	
bpf	将为*bpf_d*填写的策略标记	

根据传入的主体信任状参数，为新建的 BPF 描述子设置标记。当进程打开 BPF 设备节点时，该函数将被调用。

6.7.3.3.2. mpo_create_ifnet

void **mpo_create_ifnet**(**ifnet**, **ifnetlabel**);

struct ifnet ***ifnet**;
struct label ***ifnetlabel**;

参数	说明	锁定
ifnet	网络接口	
ifnetlabel	将为*ifnet*填写的策略标记	

为新建的网络接口设置标记。该函数在以下情况下被调用： 当一个新的物理接口变为可用时，或者当一个伪接口在引导时或由于某个用户操作而实例化时。

6.7.3.3.3. mpo_create_ipq

void **mpo_create_ipq**(**fragment**, **fragmentlabel**, **ipq**, **ipqlabel**);

struct mbuf ***fragment**;
struct label ***fragmentlabel**;
struct ipq ***ipq**;
struct label ***ipqlabel**;

参数	说明	锁定
fragment	第一个被	

参数	说明	锁定
	接收的 IP 分片	
fragmentlabel	的策略标记	
ipq	将被标记的 IP 重组队列	
ipqlabel	将为*ipq*填写的策略标记	

根据第一个接收到的分片的 mbuf 头部信息，为新建的 IP 分片重组队列设置标记。

6.7.3.3.4. **mpo_create_datagram_from_ipq**

void **mpo_create_create_datagram_from_ipq**(**ipq**, **ipqlabel**, **datagram**, **datagramlabel**);

struct ipq ***ipq**;
struct label ***ipqlabel**;

struct mbuf ***datagram**;
struct label ***datagramlabel**;

参数	说明	锁定
ipq	IP 重组队列	
ipqlabel	*ipq* 的策略标记	
datagram	将被标记的数据报	
datagramlabel	将为*datagramlabel*填写的策略标记	

根据 IP 分片重组队列，为刚刚重组完毕的 IP 数据报设置标记。

6.7.3.3.5. mpo_create_fragment

void **mpo_create_fragment**(**datagram**, **datagramlabel**, **fragment**, **fragmentlabel**);

struct mbuf ***datagram**;
struct label ***datagramlabel**;
struct mbuf ***fragment**;
struct label ***fragmentlabel**;

参数	说明	锁定
datagram	数据报	
datagramlabel	*datagram*的策略标记	
fragment	将被标记的分片	
fragmentlabel	将为*datagram*填写的策略标记	

根据数据报所对应的 mbuf 头部信息，为其新建的分片的 mbuf 头部设置标记。

6.7.3.3.6. mpo_create_mbuf_from_mbuf

void **mpo_create_mbuf_from_mbuf**(**oldmbuf**, **oldmbuflabel**, **newmbuf**, **newmbuflabel**);

struct mbuf *__oldmbuf__;
struct label *__oldmbuflabel__;
struct mbuf *__newmbuf__;
struct label *__newmbuflabel__;

参数	说明	锁定
oldmbuf	已有	

参数	说明	锁定
	的（源）mbuf	
oldmbuflabel	的策略标记	
newmbuf	将被标记的新建 mbuf	
newmbuflabel	将为*newmbuf*填写的策略标记	

根据某个现有数据报的 mbuf 头部信息，为新建数据报的 mbuf 头部设置标记。在许多条件下将会调用该函数， 比如，由于对齐要求而重新分配某个 mbuf 时。

6.7.3.3.7. mpo_create_mbuf_linklayer

void **mpo_create_mbuf_linklayer**(**ifnet**, **ifnetlabel**, **mbuf**, **mbuflabel**);

struct ifnet ***ifnet**;
struct label ***ifnetlabel**;
struct mbuf ***mbuf**;
struct label ***mbuflabel**;

参数	说明	锁定
ifnet	网络	

参数	说明	锁定
	接口	
ifnetlabel	*ifnet*的策略标记	
mbuf	新建数据报的mbuf头部	
mbuflabel	将为*mbuf*填写的策略标记	

为在给定接口上由于某个链路层响应而新建的数据报的mbuf头部设置标记。 该函数将在若干条件下被调用，比如当IPv4和IPv6协议栈在响应ARP或者ND6时。

6.7.3.3.8. mpo_create_mbuf_from_bpfdesc

void **mpo_create_mbuf_from_bpfdesc**(**bpf_d**, **bpflabel**, **mbuf**, **mbuflabel**);

struct bpf_d ***bpf_d**;
struct label ***bpflabel**;
struct mbuf ***mbuf**;
struct label ***mbuflabel**;

参数	说明	锁定
bpf_d	BPF 描	

参数	说明	锁定
	述子	
bpflabel	的策略标记	
mbuf	将被标记的新建 mbuf	
mbuflabel	将为*mbuf*填写的策略标记	

为使用参数 BPF 描述子创建的新数据报的 mbuf 头部设置标记。 当对参数 BPF 描述子所关联的 BPF 设备进行写操作时，该函数将被调用。

6.7.3.3.9. mpo_create_mbuf_from_ifnet

void **mpo_create_mbuf_from_ifnet**(**ifnet**, **ifnetlabel**, **mbuf**, **mbuflabel**);

struct ifnet ***ifnet**;
struct label ***ifnetlabel**;
struct mbuf ***mbuf**;
struct label ***mbuflabel**;

参数	说明	锁定
ifnet	网络	

锁定说明	
接口	
ifnetlabel	的策略标记
mbuf	新建数据报的 mbuf 头部
mbuflabel	将为*mbuf*填写的策略标记

为从网络接口参数创建的数据报的 mbuf 头部设置标记。

6.7.3.3.10. **mpo_create_mbuf_multicast_encap**

void **mpo_create_mbuf_multicast_encap**(**oldmbuf**, **oldmbuflabel**, **ifnet**, **ifnetlabel**, **newmbuf**, **newmbuflabel**);

struct mbuf ***oldmbuf**;
struct label ***oldmbuflabel**;
struct ifnet ***ifnet**;
struct label ***ifnetlabel**;
struct mbuf ***newmbuf**;
struct label ***newmbuflabel**;

参数	说明	锁定
oldmbuf	现有数据报的 mbuf 头部	
oldmbuflabel	*oldmbuf* 的策略标记	
ifnet	网络接口	
ifnetlabel	*ifnet* 的策略标记	
newmbuf	将被标记的新建数据报 mbuf 头部	

参数	说明	锁定
newmbuflabel	将为*newmbuf*填写的策略标记	

当传入的已有数据报被给定多播封装接口（multicast encapsulation interface）处理时被调用，为新创建的数据报所在 mbuf 头部设置标记。每当使用该虚拟接口传递一个mbuf时，将调用该函数。

6.7.3.3.11. mpo_create_mbuf_netlayer

void **mpo_create_mbuf_netlayer**(**oldmbuf**, **oldmbuflabel**, **newmbuf**, **newmbuflabel**);

struct mbuf ***oldmbuf**;
struct label ***oldmbuflabel**;
struct mbuf ***newmbuf**;
struct label ***newmbuflabel**;

参数	说明	锁定
oldmbuf	接收的数据报	
oldmbuflabel	*oldmbuf*的策略标记	
newmbuf	新建数据报	

参数	说明	锁定
newmbuflabel	*newmbuf* 的策略标记	

为由 IP 堆栈因为响应接收数据报（*oldmbuf*）而新建的数据报设置其 mbuf 头部的标记。 许多情况下需要调用该函数，比如，响应 ICMP 请求数据报时。

6.7.3.3.12. mpo_fragment_match

int **mpo_fragment_match**(**fragment**, **fragmentlabel**, **ipq**, **ipqlabel**);

struct mbuf ***fragment**;
struct label ***fragmentlabel**;
struct ipq ***ipq**;
struct label ***ipqlabel**;

参数	说明	锁定
fragment	IP 数据报分片	
fragmentlabel	*fragment* 的策略标记	
ipq	IP 分片重组队列	

参数	说明	锁定
ipqlabel	的策略标记	

根据所传入的 IP 分片重组队列（*ipq*）的标记， 检查包含一个 IP 数据报（*fragment*）的 mbuf 的头部是否符合其要求。 符合，则返回1。否则，返回0。每当 IP 堆栈尝试将一个刚刚接收到的分片放入某个已有的分片重组队列中时，将调用该函数进行安全检查； 如果失败，将为分片重新实例化一个新的分片重组队列。 策略可以利用该入口函数，根据标记或者其他信息阻止不期望的 IP 分片重组。

6.7.3.3.13. **mpo_relabel_ifnet**

void **mpo_relabel_ifnet**(**cred**, **ifnet**, **ifnetlabel**, **newlabel**);

struct ucred ***cred**;
struct ifnet ***ifnet**;
struct label ***ifnetlabel**;
struct label ***newlabel**;

参数	说明	锁定
cred	主体信任状	
ifnet	客体；网络接口	
ifnetlabel	的策略	

参数	说明	锁定
	标记	
newlabel	将为*ifnet*设置的新标记	

根据所传入的新标记，*newlabel*，以及主体信任状， *cred*，对网络接口的标记进行更新。

6.7.3.3.14. mpo_update_ipq

void **mpo_update_ipq**(**fragment**, **fragmentlabel**, **ipq**, **ipqlabel**);

struct mbuf ***fragment**;
struct label ***fragmentlabel**;
struct ipq ***ipq**;
struct label ***ipqlabel**;

参数	说明	锁定
mbuf	IP 分片	
***mbuf**label*	的策略标记	
ipq	IP 分片重组队列	

参数	说明	锁定
ipqlabel	将被更新的*ipq*的当前策略标记	

根据所传入的 IP 分片 mbuf 头部（*mbuf*）为接收 它的 IP 分片重组队列（*ipq*）的标记进行更新。

6.7.3.4. 进程标记事件操作

6.7.3.4.1. mpo_create_cred

void **mpo_create_cred**(**parent_cred**, **child_cred**);

struct ucred ***parent_cred**;
struct ucred ***child_cred**;

参数	说明	锁定
parent_cred	父主体信任状	
child_cred	子主体信任状	

根据所传入的主体信任状，为新建的主体信任状设置标记。 每当为一个新建的 struct ucred调用 crcopy(9) 时，将调用此函数。 该函数不应与进程复制（forking）或者创建事件混为一谈。

6.7.3.4.2. mpo_execve_transition

void **mpo_execve_transition**(**old**, **new**, **vp**, **vnodelabel**);

struct ucred ***old**;
struct ucred ***new**;
struct vnode ***vp**;
struct label ***vnodelabel**;

参数	说明	锁定
old	已有的主体信任状	不可改变
new	将被标记的新主体信任状	
vp	将被执行的文件	被锁定
vnodelabel	*vp* 的策略标记	

一个拥有信任状*old*的主体由于执行(*vp*文件而导致标记转换时， 该函数根据vnode标记为该主体重新标记为*new*。 每当一个进程请求执行vnode文件，而通过 入口函数mpo_execve_will_transition 有成功返回的策略时，将调用该函数。 策略模块可以通过传入两个主体信任状和简单地调用 mpo_create_cred 来实现该入口函数， so as not to implement a transitioning event. 一旦策略实现了mpo_create_cred函数，即使没有实现 mpo_execve_will_transition，也应该实现该函数。

6.7.3.4.3. mpo_execve_will_transition

int **mpo_execve_will_transition**(**old**, **vp**, **vnodelabel**);

struct ucred ***old**;
struct vnode ***vp**;
struct label ***vnodelabel**;

参数	说明	锁定
old	在执行execve(2)之前的主体信任状	不可改变
vp	将被执行的文件	
vnodelabel	*vp*的策略标记	

由策略决定，当参数主体信任状执行参数 vnode 时，是否需要进行一个标记转换操作。如果需要，返回1； 否则，返回0。即使一个策略返回0，它也必须为自己不

期望的对 mpo_execve_transition的调用作好准备，因为只要有其他任何一个策略要求转换，就将执行此函数。

6.7.3.4.4. **mpo_create_proc0**

void **mpo_create_proc0**(**cred**);

struct ucred ***cred**;

参数	说明	锁定
cred	将被填写的主体信任状	

为进程0，所有内核进程的祖先，创建主体信任状。

6.7.3.4.5. **mpo_create_proc1**

void **mpo_create_proc1**(**cred**);

struct ucred ***cred**;

参数	说明	锁定
cred	将被填写的主体信任状	

为进程1，所有用户进程的祖先，创建主体信任状。

6.7.3.4.6. mpo_relabel_cred

void **mpo_relabel_cred**(**cred**, **newlabel**);

struct ucred ***cred**;
struct label ***newlabel**;

参数	说明	锁定
cred	主体信任状	
newlabel	将被应用到 *cred* 上的新标记	

根据传入的新标记，对主体信任状上的标记进行更新。

6.7.4. 访问控制检查

通过访问控制入口函数，策略模块能影响内核的访问控制决策。 通常情况下，不是绝对，一个访问控制入口函数的参数有，一个或者若干个授权信任状，和相关操作涉及的其他任何对象的信息（其中可能包含标记）。 访问控制入口函数返回0，表示允许该操作；否则，返回一个 errno(2) 错误编码。调用该入口函数，将遍历所有系统注册的策略模块，逐一进行 策略相关的检查和决策，之后按照下述方法组合不同策略的返回结果：只有当所有的模块均允许该操作时，才成功返回。 否则，如果有一个或者若干模块失败返回，则整个检查不通过。如果有多个模块的检查出错返回，将由定义在kern_mac.c 中的 error_select() 函数从它们返回的错误编码中，选择一个合适的，返回给用户。

最高优先级	EDEADLK
	EINVAL
	ESRCH

	EACCES
最低优先级	EPERM

如果所有策略模块返回的错误编码均没有出现在上述优先级序列表中，则任意选择一个返回。 选择错误编码的一般次序为：内核错误，无效的参数，对象不存在，访问被拒绝，和其他错误。

6.7.4.1. mpo_check_bpfdesc_receive

int **mpo_check_bpfdesc_receive**(**bpf_d**, **bpflabel**, **ifnet**, **ifnetlabel**);

struct bpf_d ***bpf_d**;
struct label ***bpflabel**;
struct ifnet ***ifnet**;
struct label ***ifnetlabel**;

参数	说明	锁定
bpf_d	主体；BPF描述子	
bpflabel	*bpf_d*的策略标记	
ifnet	客体；网络接口	
ifnetlabel	*ifnet*的策略标记	

决定 MAC 框架是否应该允许将由参数接口接收到的数据报传递给由 BPF 描述子所对应的缓冲区。成功，则返回0； 否则，返回错误编码信息errno。建议使用的错误编码有：EACCES，用于标记不符的情况； EPERM，用于缺少特权的情况。

6.7.4.2. **mpo_check_kenv_dump**

int **mpo_check_kenv_dump**(**cred**);

struct ucred ***cred**;

参数	说明	锁定
cred	主体信任状	

决定相关主体是否应该被允许查询内核环境状态（参考 kenv(2)）。

6.7.4.3. **mpo_check_kenv_get**

int **mpo_check_kenv_get**(**cred**, **name**);

struct ucred ***cred**;
char ***name**;

参数	说明	锁定
cred	主体信任状	
name	内核的环境变量名字	

决定相关主体是否可以查询内核中给定环境变量的状态。

6.7.4.4. mpo_check_kenv_set

int **mpo_check_kenv_set**(**cred**, **name**);

struct ucred ***cred**;
char ***name**;

参数	说明	锁定
cred	主体信任状	
name	内核的环境变量名字	

决定相关主体是否有权设置给定内核环境变量的值。

6.7.4.5. mpo_check_kenv_unset

int **mpo_check_kenv_unset**(**cred**, **name**);

struct ucred ***cred**;
char ***name**;

参数	说明	锁定
cred	主体信任状	
name	内核的环境	

参数	说明	锁定
	变量名字 Kernel environment variable name	

决定相关主体是否有权清除给定的内核环境变量的设置。

6.7.4.6. mpo_check_kld_load

int **mpo_check_kld_load**(**cred**, **vp**, **vlabel**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***vlabel**;

参数	说明	锁定
cred	主体信任状	
vp	内核模块的 vnode	
vlabel	vp 的策略标记	

决定相关主体是否有权加载给定的模块文件。

6.7.4.7. mpo_check_kld_stat

int **mpo_check_kld_stat**(**cred**);

struct ucred ***cred**;

参数	说明	锁定
cred	主体信任状	

决定相关主体是否有权访问内核的加载模块文件链表以及相关的统计数据。

6.7.4.8. mpo_check_kld_unload

int **mpo_check_kld_unload**(**cred**);

struct ucred ***cred**;

参数	说明	锁定
cred	主体信任状	

决定相关主体是否有权卸载一个内核模块。

6.7.4.9. mpo_check_pipe_ioctl

int **mpo_check_pipe_ioctl**(**cred**, **pipe**, **pipelabel**, **cmd**, **data**);

struct ucred ***cred**;
struct pipe ***pipe**;
struct label ***pipelabel**;
unsigned long **cmd**;
void ***data**;

参数	说明	锁定
cred	主体信任状	

参数	说明	锁定
pipe	管道	
pipelabel	管道的策略标记	
cmd	ioctl(2) 命令	
data	ioctl(2) 数据	

决定相关主体是否有权调用指定的 ioctl(2) 系统调用。

6.7.4.10. mpo_check_pipe_poll

int **mpo_check_pipe_poll**(**cred**, **pipe**, **pipelabel**);

struct ucred ***cred**;
struct pipe ***pipe**;
struct label ***pipelabel**;

参数	说明	锁定
cred	主体信任状	
pipe	管道	
pipelabel	管道的策略标记	

决定相关主体是否有权对管道*pipe*执行poll操作。

6.7.4.11. **mpo_check_pipe_read**

int **mpo_check_pipe_read**(**cred**, **pipe**, **pipelabel**);

struct ucred ***cred**;
struct pipe ***pipe**;
struct label ***pipelabel**;

参数	说明	锁定
cred	主体信任状	
pipe	管道	
pipelabel	*pipe*的策略标记	

决定该主体是否有权读取*pipe*。

6.7.4.12. **mpo_check_pipe_relabel**

int **mpo_check_pipe_relabel**(**cred**, **pipe**, **pipelabel**, **newlabel**);

struct ucred ***cred**;
struct pipe ***pipe**;
struct label ***pipelabel**;
struct label ***newlabel**;

参数	说明	锁定
cred	主体信任状	
pipe	管道	

参数	说明	锁定
pipelabel	*pipe*的当前策略标记	
newlabel	将为*pipelabel*设置的新标记	

决定该主体是否有权为*pipe*重新设置标记。

6.7.4.13. **mpo_check_pipe_stat**

int **mpo_check_pipe_stat**(**cred**, **pipe**, **pipelabel**);

struct ucred ***cred**;
struct pipe ***pipe**;
struct label ***pipelabel**;

参数	说明	锁定
cred	主体信任状	
pipe	管道	
pipelabel	*pipe*的策略标记	

决定该主体是否有权查询与*pipe*相关的统计信息。

6.7.4.14. mpo_check_pipe_write

int **mpo_check_pipe_write**(**cred**, **pipe**, **pipelabel**);

struct ucred ***cred**;
struct pipe ***pipe**;
struct label ***pipelabel**;

参数	说明	锁定
cred	主体信任状	
pipe	管道	
pipelabel	*pipe*的策略标记	

决定该主体是否有权写*pipe*。

6.7.4.15. mpo_check_socket_bind

int **mpo_check_socket_bind**(**cred**, **socket**, **socketlabel**, **sockaddr**);

struct ucred ***cred**;
struct socket ***socket**;
struct label ***socketlabel**;
struct sockaddr ***sockaddr**;

参数	说明	锁定
cred	主体信任状	
socket	将被绑	

参数	说明	锁定
	定的套接字	
socketlabel	的策略标记	
sockaddr	的地址	

6.7.4.16. mpo_check_socket_connect

int **mpo_check_socket_connect**(**cred**, **socket**, **socketlabel**, **sockaddr**);

struct ucred ***cred**;
struct socket ***socket**;
struct label ***socketlabel**;
struct sockaddr ***sockaddr**;

参数	说明	锁定
cred	主体信任状	
socket	将被连接的套接字	
socketlabel	的策略	

参数	说明	锁定
	标记	
sockaddr	地址	

决定该主体（*cred*）是否有权将套接字（*socket*）绑定到地址 *sockaddr*。成功，返回0，否则返回一个错误编码errno。 建议采用的错误编码有：EACCES，用于标记不符的情况；EPERM，用于特权不足的情况。

6.7.4.17. mpo_check_socket_receive

int **mpo_check_socket_receive**(**cred**, **so**, **socketlabel**);

struct ucred ***cred**;
struct socket ***so**;
struct label ***socketlabel**;

参数	说明	锁定
cred	主体信任状	
so	套接字	
socketlabel	该策略标记	

决定该主体是否有权查询套接字*so*的相关信息。

6.7.4.18. mpo_check_socket_send

int **mpo_check_socket_send**(**cred**, **so**, **socketlabel**);

struct ucred ***cred**;
struct socket ***so**;
struct label ***socketlabel**;

参数	说明	锁定
cred	主体信任状	
so	套接字	
socketlabel	的策略标记	

决定该主体是否有权通过套接字*so*发送信息。

6.7.4.19. **mpo_check_cred_visible**

int **mpo_check_cred_visible**(**u1**, **u2**);

struct ucred ***u1**;
struct ucred ***u2**;

参数	说明	锁定
u1	主体信任状	
u2	对象信任状	

确定该主体信任状*u1*是否有权 “see” 具有信任状*u2* 的其他主体。 成功，返回0；否则，返回错误编码errno。建议采用的错误编码有： EACCES，用于标记不符的情况；EPERM，用于特权不足的情况；ESRCH， 用来提供不可见性。该函数可在许多环境下使用，包括命令ps所使用的进程间的状态 sysctl，以及通过procfs 的状态查询操作。

6.7.4.20. mpo_check_socket_visible

int **mpo_check_socket_visible**(**cred**, **socket**, **socketlabel**);

struct ucred ***cred**;
struct socket ***socket**;
struct label ***socketlabel**;

参数	说明	锁定
cred	主体信任状	
socket	客体；套接字	
socketlabel	socket 的策略标记	

6.7.4.21. mpo_check_ifnet_relabel

int **mpo_check_ifnet_relabel**(**cred**, **ifnet**, **ifnetlabel**, **newlabel**);

struct ucred ***cred**;
struct ifnet ***ifnet**;
struct label ***ifnetlabel**;
struct label ***newlabel**;

参数	说明	锁定
cred	主体信任状	
ifnet	客体；网	

参数	说明	锁定
	络接口	
ifnetlabel	现有的策略标记	
newlabel	将被应用到*ifnet*上的新的策略标记	

决定该主体信任状是否有权使用传入的标记更新参数对给定的网络接口的标记进行重新设置。

6.7.4.22. mpo_check_socket_relabel

int **mpo_check_socket_relabel**(**cred**, **socket**, **socketlabel**, **newlabel**);

struct ucred ***cred**;
struct socket ***socket**;
struct label ***socketlabel**;
struct label ***newlabel**;

参数	说明	锁定
cred	主体信任状	

参数	说明	锁定
socket	客体；套接字	
socketlabel	现有的策略标记	
newlabel	将被应用到*socketlabel*上的更新标记	

决定该主体信任状是否有权采用传入的标记对套接字参数的标记进行重新设置。

6.7.4.23. mpo_check_cred_relabel

int **mpo_check_cred_relabel**(**cred**, **newlabel**);

struct ucred ***cred**;
struct label ***newlabel**;

参数	说明	锁定
cred	主体信任状	
newlabel	将被应	

参数	说明	锁定
	用到*cred*上的更新标记	

决定该主体信任状是否有权将自己的标记重新设置为给定的更新标记。

6.7.4.24. **mpo_check_vnode_relabel**

int **mpo_check_vnode_relabel**(**cred**, **vp**, **vnodelabel**, **newlabel**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***vnodelabel**;
struct label ***newlabel**;

参数	说明	锁定
cred	主体信任状	
vp	客体；vnode	被锁定
vnodelabel	*vp*现有的策略标记	
newlabel	将被应用到*vp*上	

参数	说明	锁定
	的策略标记	

决定该主体信任状是否有权将参数 vnode 的标记重新设置为指定标记。

6.7.4.25. mpo_check_mount_stat

int **mpo_check_mount_stat**(**cred**, **mp**, **mountlabel**);

struct ucred ***cred**;
struct mount ***mp**;
struct label ***mountlabel**;

参数	说明	锁定
cred	主体信任状	
mp	客体；文件系统挂载	
mountlabel	*mp* 的策略标记	

确定相关主体信任状是否有权查看在给定文件系统上执行 statfs 的结果。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。 该函数可能在下列情况下被调用： 在 statfs(2) 和其他相关调用期间，或者当需要从文件系统列表中选择排除哪个文件系统时，比如， 调用 getfsstat(2)时。

6.7.4.26. mpo_check_proc_debug

int **mpo_check_proc_debug**(**cred**, **proc**);

struct ucred ***cred**;
struct proc ***proc**;

参数	说明	锁定
cred	主体信任状	不可变
proc	客体；进程	

确定相关主体信任状是否有权 debug 给定进程。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配；EPERM，用于权限不够； ESRCH，用于隐瞒目标的存在。 ptrace(2) 和 ktrace(2) API，以及某些 procfs 操作将调用该函数。

6.7.4.27. mpo_check_vnode_access

int **mpo_check_vnode_access**(**cred**, **vp**, **label**, **flags**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;
int **flags**;

参数	说明	锁定
cred	主体信任状	
vp	客体；vnode	
label	vp的策	

参数	说明	锁定
	策略标记	
flags	access(2) 标志	

根据相关主体信任状决定其对给定 vnode 以给定访问标志执行的 access(2) 和其他相关调用的返回值。一般，应采用与mpo_check_vnode_open 相同的语义来实现该函数。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

6.7.4.28. mpo_check_vnode_chdir

int **mpo_check_vnode_chdir**(**cred**, **dvp**, **dlabel**);

struct ucred ***cred**;
struct vnode ***dvp**;
struct label ***dlabel**;

参数	说明	锁定
cred	主体信任状	
dvp	客体；chdir(2) 的目的 vnode	
dlabel	dvp 的策略标记	

确定相关主体信任状是否有权将进程工作目录切换到给定 vnode。成功，则返回 0； 否则，返回一个 errno值。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

6.7.4.29. mpo_check_vnode_chroot

int **mpo_check_vnode_chroot**(**cred**, **dvp**, **dlabel**);

struct ucred ***cred**;
struct vnode ***dvp**;
struct label ***dlabel**;

参数	说明	锁定
cred	主体信任状	
dvp	目录 vnode	
dlabel	与 *dvp* 相关联的策略标记	

确定相关主体是否有权 chroot(2) 到由（*dvp*)给定的目录。

6.7.4.30. mpo_check_vnode_create

int **mpo_check_vnode_create**(**cred**, **dvp**, **dlabel**, **cnp**, **vap**);

struct ucred ***cred**;
struct vnode ***dvp**;
struct label ***dlabel**;
struct componentname ***cnp**;
struct vattr ***vap**;

参数	说明	锁定
cred	主体信	

参数	说明	锁定
	任状	
dvp	父体；vnode	
dlabel	dvp的策略标记	
cnp	dvp中的成员名	
vap	vp的vnode属性	

确定相关主体信任状是否有权在给定父目录，以给定的名字和属性， 常见一个 vnode。成功，则返回 0；否则， 返回一个errno值。 建议使用的错误编码：EACCES 来表示用于标记不匹配， 而用 EPERM，用于权限不足。 以O_CREAT为参数调用 open(2)， 或对 mknod(2)， mkfifo(2) 等的调用将导致该函数被调用。

6.7.4.31. mpo_check_vnode_delete

int **mpo_check_vnode_delete**(**cred**, **dvp**, **dlabel**, **vp**, **label**, **cnp**);

struct ucred ***cred**;
struct vnode ***dvp**;
struct label ***dlabel**;
struct vnode ***vp**;
void ***label**;
struct componentname ***cnp**;

参数	说明	锁定
cred	主体信	

参数	说明	锁定
cred	主体信任状	
dvp	父目录 vnode	
dlabel	与 dvp 关联的策略标记	
vp	客体；将被删除的 vnode	
label	与 vp 关联的策略标记	
cnp	vp 中的成员名	

确定相关主体信任状是否有权从给定的父目录中，删除给定名字的 vnode。成功，则返回 0；否则，返回一个errno值。建议使用的错误编码：EACCES，用于标记不匹配；EPERM，用于权限不够。使用 unlink(2) 和 rmdir(2)，将导致该函数被调用。提供该入口函数的策略还必须实现一个 mpo_check_rename_to，用来授权由于重命名操作导致的目标文件的删除。

6.7.4.32. mpo_check_vnode_deleteacl

int **mpo_check_vnode_deleteacl**(**cred**, **vp**, **label**, **type**);

struct ucred ***cred**;

struct vnode ***vp**;
struct label ***label**;
acl_type_t **type**;

参数	说明	锁定
cred	主体信任状	
vp	对象；vnode	被锁定
label	vp的策略标记	
type	ACL类型	

确定相关主体信任状是否有权删除给定 vnode 的给定类型的 ACL。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

6.7.4.33. mpo_check_vnode_exec

int **mpo_check_vnode_exec**(**cred**, **vp**, **label**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;

参数	说明	锁定
cred	主体信任状	
vp	客体；	

参数	说明	锁定
	将被执行的vnode	
label	vp的策略标记	

确定相关主体信任状是否有权执行给定 vnode。 对于执行特权的决策与任何瞬时事件的决策是严格分开的。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

6.7.4.34. mpo_check_vnode_getacl

int **mpo_check_vnode_getacl**(**cred**, **vp**, **label**, **type**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;
acl_type_t **type**;

参数	说明	锁定
cred	主体信任状	
vp	客体；vnode	
label	vp的策略标记	

参数	说明	锁定
type	ACL 类型	

确定相关主体信任状是否有权查询给定 vnode 上的给定类型的 ACL。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

6.7.4.35. **mpo_check_vnode_getextattr**

int **mpo_check_vnode_getextattr**(**cred**, **vp**, **label**, **attrnamespace**, **name**, **uio**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;
int **attrnamespace**;
const char ***name**;
struct uio ***uio**;

参数	说明	锁定
cred	主体信任状	
vp	客体；vnode	
label	vp的策略标记	
attrnamespace	扩展属性名字	

参数	说明	锁定
	空间	
name	扩展属性名	
uio	UIO结构指针；参见 uio(9)	

确定相关主体信任状是否有权查询给定 vnode 上给定名字空间和名字的扩展属性。使用扩展属性实现标记存储的策略模块可能会需要对这些扩展属性的操作进行特殊处理。成功，则返回 0；否则，返回一个errno值。建议使用的错误编码：EACCES，用于标记不匹配；EPERM，用于权限不够。

6.7.4.36. mpo_check_vnode_link

int **mpo_check_vnode_link**(**cred**, **dvp**, **dlabel**, **vp**, **label**, **cnp**);

struct ucred ***cred**;
struct vnode ***dvp**;
struct label ***dlabel**;
struct vnode ***vp**;
struct label ***label**;
struct componentname ***cnp**;

参数	说明	锁定
cred	主体信任状	

参数	说明	锁定
dvp	目录vnode	
dlabel	与*dvp*相关联的策略标记	
vp	链接目的vnode	
label	与*vp*相关联的策略标记	
cnp	将被创建的链接对应的成员名	

确定相关主体是否有权为参数*vp*给定的 vnode 创建一个由参数*cnp*给定名字的链接。

6.7.4.37. **mpo_check_vnode_mmap**

int **mpo_check_vnode_mmap**(**cred**, **vp**, **label**, **prot**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;
int **prot**;

参数	说明	锁定
cred	主体信任状	
vp	将被映射的 vnode	
label	与 *vp* 相关联的策略标记	
prot	mmap 保护 (参见 mmap(2))	

确定相关主体是否有权将给定 vnode *vp* 以 *prot*指定的保护方式进行映射.

6.7.4.38. **mpo_check_vnode_mmap_downgrade**

void **mpo_check_vnode_mmap_downgrade**(**cred**, **vp**, **label**, **prot**);

struct ucred ***cred**;
struct vnode ***vp**;

struct label ***label**;
int ***prot**;

参数	说明	锁定
cred *vp* *label*	见第 6.7.4.37 节 “mpo_check_vnode_mmap” .	
prot	将被降级的 mmap protections	

根据主体和客体标记，降低 mmap protections。

6.7.4.39. mpo_check_vnode_mprotect

int **mpo_check_vnode_mprotect**(**cred**, **vp**, **label**, **prot**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;
int **prot**;

参数	说明	锁定
cred	主体信任状	
vp	映射的 vnode	
prot	存储保护	

确定相关主体是否有权将给定 vnode*vp* 映射内存空间的存储保护参数设置为指定值。

6.7.4.40. mpo_check_vnode_poll

int **mpo_check_vnode_poll**(**active_cred**, **file_cred**, **vp**, **label**);

struct ucred ***active_cred**;
struct ucred ***file_cred**;
struct vnode ***vp**;
struct label ***label**;

参数	说明	锁定
active_cred	主体信任状	
file_cred	struct file 相关联的信任状	
vp	将被执行 poll 操作的 vnode	
label	*vp* 相关联的策略	

标记

确定相关主体是否有权对给定 vnode *vp*执行 poll 操作。

6.7.4.41. **mpo_check_vnode_rename_from**

int **mpo_vnode_rename_from**(**cred**, **dvp**, **dlabel**, **vp**, **label**, **cnp**);

struct ucred ***cred**;
struct vnode ***dvp**;
struct label ***dlabel**;
struct vnode ***vp**;
struct label ***label**;
struct componentname ***cnp**;

参数	说明	锁定
cred	主体信任状	
dvp	目录 vnode	
dlabel	与 *dvp* 相关联的策略标记	
vp	将被重命名的 vnode	

参数	说明	锁定
label	与*vp*相关联的策略标记	
cnp	*vp*中的成员名	

确定相关主体是否有权重命名给定vnode，*vp*。

6.7.4.42. **mpo_check_vnode_rename_to**

int **mpo_check_vnode_rename_to**(**cred**, **dvp**, **dlabel**, **vp**, **label**, **samedir**, **cnp**);

struct ucred ***cred**;
struct vnode ***dvp**;
struct label ***dlabel**;
struct vnode ***vp**;
struct label ***label**;
int **samedir**;
struct componentname ***cnp**;

参数	说明	锁定
cred	主体信任状	
dvp	目录vnode	
dlabel	与*dvp*相关联	

锁定的数据对象的策略标记
被 *vp* 覆盖的 vnode
与 *vp* *label* 相关联的策略标记
samedir 布尔型变量；如果源和目的目录是相同的，则被置为1

参数	说明	锁定
cnp	目标 component 名	

确定相关主体是否有权重命名给定 vnode *vp*，至指定目录 *dvp*，或更名为*cnp*。如果无需覆盖已有文件， 则*vp* 和 *label* 的值将为 NULL.

6.7.4.43. mpo_check_socket_listen

int **mpo_check_socket_listen**(**cred**, **socket**, **socketlabel**);

struct ucred ***cred**;
struct socket ***socket**;
struct label ***socketlabel**;

参数	说明	锁定
cred	主体信任状	
socket	客体；套接字	
socketlabel	的策略标记	

确定相关主体是否有权监听给定套接字。 成功，则返回0；否则，返回错误编码值errno。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

6.7.4.44. mpo_check_vnode_lookup

int **mpo_check_vnode_lookup**(, , , **cnp**);

struct ucred *cred;

struct vnode *dvp;
struct label *dlabel;
struct componentname ***cnp**;

参数	说明	锁定
cred	主体信任状	
dvp	客体；vnode	
dlabel	dvp的策略标记	
cnp	被检查的成员名	

确定相关主体信任状是否有权在给定的目录 vnode 中为查找给定名字执行lookup 操作。 成功，则返回 0；否则，返回一个 errno值。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

6.7.4.45. mpo_check_vnode_open

int **mpo_check_vnode_open**(**cred**, **vp**, **label**, **acc_mode**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;
int **acc_mode**;

参数	说明	锁定
cred	主体	

参数	说明	锁定
	信任状	
vp	客体; vnode	
label	vp 的策略标记	
acc_mode	open(2) 访问模式	

确定相关主体信任状是否有权在给定 vnode 上以给定的访问模式执行 open 操作。 如果成功，则返回 0；否则，返回一个错误编码。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

6.7.4.46. mpo_check_vnode_readdir

int **mpo_check_vnode_readdir**(, ,);

struct ucred *cred;
struct vnode *dvp;
struct label *dlabel;

参数	说明	锁定
cred	主体信任状	
dvp	客体; 目录 vnode	

参数	说明	锁定
dlabel	dvp 的策略标记	

确定相关主体信任状是否有权在给定的目录 vnode 上执行 readdir 操作。 成功，则返回 0；否则，返回一个错误编码 errno。 建议使用的错误编码：EACCES，用于标记不匹配；EPERM，用于权限不够。

6.7.4.47. mpo_check_vnode_readlink

int **mpo_check_vnode_readlink**(**cred**, **vp**, **label**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;

参数	说明	锁定
cred	主体信任状	
vp	客体；vnode	
label	vp 的策略标记	

确定相关主体信任状是否有权在给定符号链接 vnode 上执行 readlink 操作。成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配；EPERM，用于权限不够。 该函数可能在若干环境下被调用，包括由用户进程显式执行的 readlink 调用， 或者是在进程执行名字查询时隐式执行的 readlink 。

6.7.4.48. mpo_check_vnode_revoke

int **mpo_check_vnode_revoke**(**cred**, **vp**, **label**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;

参数	说明	锁定
cred	主体信任状	
vp	客体；vnode	
label	vp的策略标记	

确定相关主体信任状是否有权撤销对给定 vnode 的访问。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配；EPERM，用于权限不够。

6.7.4.49. mpo_check_vnode_setacl

int **mpo_check_vnode_setacl**(**cred**, **vp**, **label**, **type**, **acl**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;
acl_type_t **type**;
struct acl ***acl**;

参数	说明	锁定
cred	主体信任状	
vp	客体；vnode	

参数	说明	锁定
label	的策略标记	
type	ACL类型	
acl	ACL	

确定相关主体信任状是否有权设置给定 vnode 的给定类型的 ACL。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配；EPERM，用于权限不够。

6.7.4.50. mpo_check_vnode_setextattr

int **mpo_check_vnode_setextattr**(**cred**, **vp**, **label**, **attrnamespace**, **name**, **uio**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;
int **attrnamespace**;
const char ***name**;
struct uio ***uio**;

参数	说明	锁定
cred	主体信任状	
vp	客体；vnode	
label	的策略标记	

参数	说明	锁定
attrnamespace	扩展属性名字空间	
name	扩展属性名	
uio	I/O 结构指针；参见 uio(9)	

确定相关主体信任状是否有权设置给定 vnode 上给定名字空间中给定名字的扩展属性的值。 使用扩展属性备份安全标记的策略模块可能需要对其使用的属性实施额外的保护。另外， 由于在检查和实际操作时间可能存在的竞争， 策略模块应该避免根据来自*uio*中的数据做出决策。 如果正在执行一个删除操作，则参数 *uio* 的值也可能为 NULL。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配；EPERM，用于权限不够。

6.7.4.51. mpo_check_vnode_setflags

int **mpo_check_vnode_setflags**(**cred**, **vp**, **label**, **flags**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;
u_long **flags**;

参数	说明	锁定
cred	主体信任状	
vp	客体；vnode	
label	*vp* 的策略标记	
flags	文件标志；参见 chflags(2)	

确定相关主体信任状是否有权为给定的 vnode 设置给定的标志。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

6.7.4.52. mpo_check_vnode_setmode

int **mpo_check_vnode_setmode**(**cred**, **vp**, **label**, **mode**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;
mode_t **mode**;

参数	说明	锁定
cred	主体信任状	

参数	说明	锁定
vp	客体；vnode	
label	vp的策略标记	
mode	文件模式；参见 chmod(2)	

确定相关主体信任状是否有权将给定 vnode 的模式设置为给定值。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

6.7.4.53. mpo_check_vnode_setowner

int **mpo_check_vnode_setowner**(**cred**, **vp**, **label**, **uid**, **gid**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;
uid_t **uid**;
gid_t **gid**;

参数	说明	锁定
cred	主体信任状	
vp	客体；vnode	

参数	说明	锁定
label	vp的策略标记	
uid	用户ID	
gid	组ID	

确定相关主体信任状是否有权将给定 vnode 的文件 uid 和文件 gid 设置为给定值。如果无需更新， 相关参数值可能被设置为(-1)。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

6.7.4.54. **mpo_check_vnode_setutimes**

int **mpo_check_vnode_setutimes**(, , , ,);

struct ucred *cred;
struct vnode *vp;
struct label *label;
struct timespec atime;
struct timespec mtime;

参数	说明	锁定
cred	主体信任状	
vp	客体；vp	
label	vp的策略标记	

参数	说明	锁定
atime	访问时间；参见 utimes(2)	
mtime	修改时间；参见 utimes(2)	

确定相关主体信任状是否有权将给定 vnode 的访问时间标签设置为给定值。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

6.7.4.55. mpo_check_proc_sched

int **mpo_check_proc_sched(ucred**, **proc)**;

struct ucred ***ucred**;
struct proc ***proc**;

参数	说明	锁定
cred	主体信任状	
proc	客体；进程	

确定相关主体信任状是否有权改变给定进程的调度参数。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配；EPERM，用于权限不够； ESRCH，用于提供不可见性质。

See setpriority(2) for more information.

6.7.4.56. mpo_check_proc_signal

int **mpo_check_proc_signal**(**cred**, **proc**, **signal**);

struct ucred ***cred**;
struct proc ***proc**;
int **signal**;

参数	说明	锁定
cred	主体信任状	
proc	客体；进程	
signal	信号；参见 kill(2)	

确定相关主体信任状是否有权向给定进程发送给定信号。成功，则返回 0；否则，返回一个errno值。建议使用的错误编码：EACCES，用于标记不匹配；EPERM，用于权限不够；ESRCH，用于提供不可见性质。

6.7.4.57. mpo_check_vnode_stat

int **mpo_check_vnode_stat**(**cred**, **vp**, **label**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***label**;

参数	说明	锁定
cred	主体信	

参数	说明	锁定
	任状	
vp	客体；vnode	
label	*vp* 的策略标记	

确定相关主体信任状是否有权在给定 vnode 上执行 stat 操作。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

See stat(2) for more information.

6.7.4.58. mpo_check_ifnet_transmit

int **mpo_check_ifnet_transmit**(**cred**, **ifnet**, **ifnetlabel**, **mbuf**, **mbuflabel**);

struct ucred ***cred**;
struct ifnet ***ifnet**;
struct label ***ifnetlabel**;
struct mbuf ***mbuf**;
struct label ***mbuflabel**;

参数	说明	锁定
cred	主体信任状	
ifnet	网络接口	
ifnetlabel	*ifnet* 的策略	

参数	说明	锁定
	标记	
mbuf	客体; 将被发送的 mbuf	
mbuflabel	mbuf 的策略标记	

确定相关网络接口是否有权传送给定的 mbuf。成功，则返回 0； 否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

6.7.4.59. mpo_check_socket_deliver

int **mpo_check_socket_deliver**(**cred**, **ifnet**, **ifnetlabel**, **mbuf**, **mbuflabel**);

struct ucred ***cred**;
struct ifnet ***ifnet**;
struct label ***ifnetlabel**;
struct mbuf ***mbuf**;
struct label ***mbuflabel**;

参数	说明	锁定
cred	主体信任状	
ifnet	网络接口	

参数	说明	锁定
ifnetlabel	的策略标记	
mbuf	客体；将被传送的mbuf	
mbuflabel	的策略标记	

确定相关套接字是否有权从给定的 mbuf 中接收数据报。 成功，则返回 0；否则，返回一个errno值。 建议使用的错误编码：EACCES，用于标记不匹配； EPERM，用于权限不够。

6.7.4.60. mpo_check_socket_visible

int **mpo_check_socket_visible**(**cred**, **so**, **socketlabel**);

struct ucred ***cred**;
struct socket ***so**;
struct label ***socketlabel**;

参数	说明	锁定
cred	主体凭证	不可改变状
so	客体；	

参数	说明	锁定
socketlabel	套接字的策略标记	

确定相关主体信任状cred 是否有权使用系统监控函数，比如，由netstat(8) 和 sockstat(1)使用的程序来观察 给定的套接字(*socket*)。成功，则返回 0；否则，返回一个errno值。建议使用的错误编码：EACCES，用于标记不匹配；EPERM，用于权限不够；ESRCH，用于提供不可见性质。

6.7.4.61. mpo_check_system_acct

int **mpo_check_system_acct**(**ucred**, **vp**, **vlabel**);

struct ucred ***ucred**;
struct vnode ***vp**;
struct label ***vlabel**;

参数	说明	锁定
ucred	主体信任状	
vp	审计文件；acct(5)	
vlabel	与vp相关联的标记	

根据主体标记和审计日志文件的标记，确定该主体是否有权启动审计。

6.7.4.62. **mpo_check_system_nfsd**

int **mpo_check_system_nfsd**(**cred**);

struct ucred ***cred**;

参数	说明	锁定
cred	主体信任状	

确定相关主体是否有权调用 nfssvc(2)。

6.7.4.63. **mpo_check_system_reboot**

int **mpo_check_system_reboot**(**cred**, **howto**);

struct ucred ***cred**;
int **howto**;

参数	说明	锁定
cred	主体信任状	
howto	来自 reboot(2) 的 *howto* 参数	

确定相关主体是否有权以指定方式重启系统。

6.7.4.64. **mpo_check_system_settime**

int **mpo_check_system_settime**(**cred**);

struct ucred ***cred**;

参数	说明	锁定
cred	主体信任状	

确定相关用户是否有权设置系统时钟。

6.7.4.65. mpo_check_system_swapon

int **mpo_check_system_swapon**(**cred**, **vp**, **vlabel**);

struct ucred ***cred**;
struct vnode ***vp**;
struct label ***vlabel**;

参数	说明	锁定
cred	主体信任状	
vp	swap设备	
vlabel	与*vp*相关联的标记	

确定相关主体是否有权增加一个作为swap设备的 *vp* 。

6.7.4.66. mpo_check_system_sysctl

int **mpo_check_system_sysctl**(**cred**, **name**, **namelen**, **old**, **oldlenp**, **inkernel**, **new**, **newlen**);

struct ucred ***cred**;
int ***name**;
u_int ***namelen**;

void ***old**;
size_t ***oldlenp**;
int **inkernel**;
void ***new**;
size_t **newlen**;

参数	说明	锁定
cred	主体信任状	
name	参见 sysctl(3)	
namelen		
old		
oldlenp		
inkernel	布尔型变量；如果从内核被调用，其值被置为1	
new	参见 sysctl(3)	
newlen		

确定相关主体是否应该被允许执行指定的 sysctl(3) 事务。

6.7.5. 标记管理调用

当用户进程请求对某个对象的标记进行修改时，将引发重新标记事件。对应的更新操作分两步进行： 首先，进行访问控制检查，确认此次更新操作是有效且被允许的；然后，调用另一个独立的入口函数对标记进行修改。 重新标记入口函数通常接收由请求进程提交的对象、对象标记指针和请求新标记，作为输入参数。 对象重新标记操作的失败将由先期的标记检查报告，所以，不允许在接下来的标记修改过程中报告失败，故而不提倡在此过程中新分配内存。

6.8. 应用层体系结构

TrustedBSD MAC 框架包含了一组策略无关的组成元素，包括管理抽象标记的 MAC 接口库， 对系统信任状管理体系的修改，为用户分配 MAC 标记提供支持的 login 库函数， 以及若干负责维护和更新内核对象(进程、文件和网络接口等)安全标记的工具。 不久，将有更多关于应用层体系结构的详细信息被包含进来。

6.8.1. 策略无关的标记管理 API

TrustedBSD MAC 提供的大量库函数和系统调用，允许应用程序使用一种统一的、策略无关的接口来处理对象的 MAC 标记。 如此，应用程序可以轻松管理各种策略的标记，无需为增加对某个特定策略的支持而重新编码。许多通用工具，比如 ifconfig(8)， ls(1) 和 ps(1)，使用这些策略无关的接口查询网络结构、文件和进程的标记信息。 这些 API 也被用于支持 MAC 管理工具，比如，getfmac(8)，getpmac(8)， setfmac(8)， setfsmac(8)，和 setpmac(8)。 MAC API的设计细节可参考 mac(3).

应用程序处理的 MAC 标记有两种存在形式：内部形式，用来返回和设置进程和对象的标记（mac_t）； 基于 C 字符串的外部形式，作为标记在配置文件中的存放形式，用于向用户显示或者由用户输入。 每一个 MAC 标记由一组标记元素组成，其中每个元素是一个形如（名字，值）的二元组。 内核中的每个策略模块分别被指定一个特定的名字，由它们对标记中与该名字对应的值采用其策略特有的方式进行解析。 采用外部形式表示的标记，其标记元素表示为名字 / 值，元素之间以逗号分隔。 应用程序可以使用 MAC 框架提供的 API 将一个安全标记在内部形式和文本形式之间进行转换。 每当向内核查询某个对象的安全标记时，内部形式的标记必须针对所需的元素集合作好内部标记存储准备。 为此，通常采用下面两种方式之一：使用 mac_prepare(3) 和一个包含所需标记元素的任意列表；或者， 使用从mac.conf(5) 配置文件中加载缺省元素集合的某个系统调用。在对象级别设置缺省标记，将允许应用程序在不确定 系统是否采用相关策略的情况下，也能向用户返回与对象相关联的有意义的安全标记。

> **注意**
>
> 目前的 MAC 库不支持直接修改内部形式的标记元素，所有的修改必须按照下列的步骤进行： 将内部形式的标记转换成文本字符串，对字符串进行编辑，最后将其转换成内部形式标记。如果应用程序的作者证明确实有需要， 可以在将来的版本中加入对内部形式标记进行直接修改的接口。

6.8.2. 为用户指定标记

用户上下文管理的标记接口， setusercontext(3) ，的行为已经被修改为，从 login.conf(5) 中查询与某个用户登录类别相关联的 MAC 安全标记。 当 LOGIN_SETALL 被设置，或者当 LOGIN_SETMAC 被明确指定时，这些安全标记将和其他用户上下文参数一起被设置。

> **注意**
>
> 可以预期，在今后的某个版本中，FreeBSD 将把 MAC 标记从 login.conf 的用户类别数据库中抽出，为其维护一个独立的数据库。 不过在此前后，setusercontext(3) API应该保持不变。

6.9. 小结

TrustedBSD MAC 框架使得内核模块能以一种集中的方式，完善系统的安全策略。它们既可利用现有的内核对象属性，又能使用由 MAC 框架协助维护的安全标记数据，来实施访问控制。 框架提供的灵活性使得开发人员可以在其上实现各种策略，如利用 BSD 现有的信任状（credential） 与文件保护机制的策略，以及信息流安全策略（如 MLS 和 Biba）。 实现新安全服务的策略编程人员，可以参考本文档，以了解现有安全模块的信息。

第 7 章 虚拟内存系统

供稿：Matthew Dillon.
翻译：.

7.1. 物理内存的管理—vm_page_t

物理内存通过结构体vm_page_t以页为基础进行管理。 物理内存的页由它们各自对应的结构体vm_page_t所代表， 这些结构体存放在若干个页管理队列中的一个里面。

一页可以处于在线(wired)、活动(active)，去活(inactive)、缓存(cache)、 自由(free)状态。除了在线状态，页一般被放置在一个双向链表队列里， 代表了它所处的状态。在线页不放置在任何队列里。

FreeBSD为缓存页和自由页实现了一个更为复杂的页队列机制， 以实现对页的分类管理。每一种状态都对应着多个队列， 队列的安排对应着处理器的一级、二级缓存。当需要分配一个新页时， FreeBSD会试图把一个按一级、二级缓存对齐的页面分配给虚拟内存对象。

此外，一个页可以有一个引用计数，可以被一个忙计数锁定。 虚拟内存系统也实现了“终极锁定”(ultimate locked)状态， 一个页可以用页标志PG_BUSY表示这一状态。

总之，每个页队列都按照LRU(Least-Recently Used)的原则工作。

> **译者注**
>
> 短语Least-Recently Used有两种理解方式： 1.将“least-recently”理解为反向比较级，意义为“最早”，整个短语理解为“最近的使用时间最早”；2.将“least”和“recently”理解为副词， 都修饰“used”，整个短语理解为“最近最少使用”。 这两种理解方式的实际意义基本相同。

一个页常常最初处于在线或活动状态。在线时，页常常关联于某处的页表。 虚拟内存系统通过扫描在一个较活跃的页队列(LRU)确定页的年龄， 以便将他们移到一个较不活跃的页队列中。 移动到缓存中的页依然与一个VM对象关联，但被作为立即再用的候选。 在自由对列中的页是真正未被使用的。FreeBSD尽量不将页放在自由队列中， 但是必须保持一定数量的自由页，以便响应中断时分配。

如果一个进程试图访问一个不在页表中而在某一队列中的页 (例如去活队列或缓存队列)，一个相对耗费资源少的页错误发生， 导致页被重激活。如果页根本不存在于系统内存之中，进程必须被阻塞， 此时页被从磁盘中载入。

> ### 译者注
>
> Intel等厂商的CPU工作在保护模式时，可用来实现虚拟内存。当寻址的地址空间对应着真实内存时，则正常读写； 当寻址的地址空间没有对应的真实内存时，CPU会产生一个“错误”， 通知操作系统与磁盘等设备进行交换，读寻址则调入存储内容， 写寻址则写出存储内容。这个“错误” 并非操作系统或应用程序开发人员犯下的错误， 尽管在CPU硬件实现中这与应用程序或操作系统内核崩溃的错误的发生机制相同。参见Intel的CPU保护模式开发手册。

FreeBSD动态的调整页队列，试图将各个队列中的页数维护在一个适当的比例上，同时管理程序崩溃的已清理和未清理页。重新平衡的比例数值决定于系统内存的负担。 这种重新平衡由pageout守护进程实现，包括清理未清理页(与他们的后备存储同步)、 监视页被引用的活跃程度 (重置它们在LRU队列中的位置或在不同活跃程度的页队列间移动)、 当比例不平衡时在队列间迁移页，如此等等。 FreeBSD的VM系统会将重激活页而产生的错误频率调低到一个合理的数值， 由此确定某一页活跃/闲置的实际程度。 这可以为更好的决定何时清理/分配一个页做出决策。

7.2. 统一的缓存信息结构体—vm_object_t

FreeBSD实现了统一的“虚拟内存对象”(VM对象)的设计思想。 VM对象可以与各种类型的内存使用方式相结合—直接使用(unbacked)、 交换(swap)、物理设备、文件。 由于文件系统使用相同的VM对象管理核内数据—文件的缓存， 所以这些缓存的结构也是统一的。

VM对象可以被影复制(shadowed)。 它们可以被堆放到其它类别VM对象堆栈的顶端。例如，可以有一个交换VM对象， 放置在文件VM对象堆栈的顶端，以实现MAP_PRIVATE的mmap()操作。 这样的入栈操作也可以用来实现各种各样的共享特性， 包括写入时复制(copy-on-write，用于日志文件系统)，以派生出地址空间。

应当注意，一个vm_page_t 结构体在任一个时刻只能与一个VM对象相关联。 VM对象影复本可以实现跨实例的共享相同的页。

7.3. 文件系统输入/输出—buf结构体

vnode VM对象，比如文件VM对象，一般需要维护它们自己的清理(clean)/ 未清理(dirty)信息，而不依赖于文件系统的清理/未清理维护。 例如，当VM系统要同步一个物理页和其对应的实际存储器， VM系统就需要在写入到实际存储器前将该页标记为已清理。 另外，文件系统要能够将文件或文件元数据的各部分映射到内核虚拟内存 (KVM)中以便操作。

用来进行这些管理的实体就是众所周知的文件系统缓存， struct buf或bp。 当文件系统需要对一个VM对象的一部分操作时， 它常会将这个对象的这部分映射到struct buf， 并且将struct buf中页映射到内核虚拟内存(KVM)中。 同样的，磁盘输入/输出通常要先将VM对象的各部分映射到buf结构体中， 然后对buf结构体进行输入/输出操作。 下层的vm_page_t在输入/输出期间通常被标记为 “忙” 。 文件系统缓存也会 “忙” ，这对于文件系统驱动程序非常有用， 对文件系统缓存操作比对VM真实页(hard)操作更好。

FreeBSD保留一定数量的内核虚拟内存来存放struct buf的映射， 但是这些buf结构体应该是被清理过的。这些内核虚拟内存仅用来存放映射， 并不限制缓存数据的能力。严格的说，物理数据缓存是 vm_page_t的一个功能，不是文件系统缓存的功能。 然而，由于文件系统缓存被用来处理输入/输出， 他们固有的限制了同时进行输入/输出可能的数量。 但是，由于通常有数千文件系统缓存可供使用，所以这并不会造成问题。

7.4. 映射页表—vm_map_t, vm_entry_t

FreeBSD将物理页表结构从VM系统中分离了出来。各进程的所有页表可以脱离进程(on the fly)重建，并且通常被认为是一次性的。特殊的页表，如内核虚拟内存(KVM)， 常常是被永久性预分配的；这些页表不是一次性的。

FreeBSD通过vm_map_t和vm_entry_t 结构将虚拟内存中vm_objects的各地址范围部分关联起来。 页表被直接的从 vm_map_t/vm_entry_t/vm_object_t 中有层次的合成出来。这里需要重申一下，我曾提到的“物理页仅直接与 vm_object相关联”并不很正确。vm_page_t 也被会被链接到正在与之相关联的页表中。当页表被调用时，一个vm_page_t结构体可以被链接到几个pmaps。 然而，由于有了层次的关联，因此在对象中所有对同一页的引用会引用同一 vm_page_t结构体，这样就实现了跨区域(board)的缓存的统一。

7.5. KVM存储映射

FreeBSD使用KVM存放各种各样的内核结构体。在KVM中最大的单个实体是文件系统缓存。 那是与struct buf实体有关的映射。

不像Linux，FreeBSD不将所有的物理内存映射到KVM中。 这意味着FreeBSD可以在32位平台上管理超过4GB的内存配置。事实上， 如果mmu(译者注：可能是指“内存管理单元”，“Memory Management Unit”）有足够的能力，FreeBSD理论上可以在32位平台上管理最多8TB的内存配置。 然而，大多数32平台只能映射4GB内存，这只能是一个争论点。

有几种机制可以管理KVM。管理KVM的主要机制是区域分配器 (zone allocator)。区域分配器管理着KVM的大块，再将大块切分为恒定大小的小块， 以便按照某一种类型的结构体分配。你可以使用命令vmstat -m 一览当前KVM分区使用情况。

7.6. 调整FreeBSD的虚拟内存系统

开发者的协同努力使得FreeBSD可以自行动态调整内核。一般来说， 除了内核配置选项maxusers和NMBCLUSTERS， 你不需要做任何杂乱的事情。这些内核编译配置选项(一般)被指定在 /usr/src/sys/i386/conf/CONFIG_FILE 之中。所有可用内核配置选项的描述可在 /usr/src/sys/i386/conf/LINT中找到。

在一个大系统的配置中，你可能需要增加maxusers的值。 数值范围通常在10到128。注意，过度增加maxusers 的值可能导致系统从实际可用的KVM中溢出，从而引起无法预知的操作。 最好将maxusers设为一个合理的数值，并且添加其它选项，如NMBCLUSTERS，来增加特定的资源。

如果你的系统要被重负荷的使用网络，你需要增加 NMBCLUSTERS的值。数值范围通常在1024到4096。

NBUF也是传统的规划系统的参数。 这个参数决定系统可用来映射文件系统输入/输出缓存的KVM的数量。 注意：这个参数与统一的缓存没有任何关系。这个参数可在3.0-CURRENT 和以后的内核中被动态的调节，通常不应当被手动的调节。 我们推荐你不要指定NBUF。 让系统自行确定它。太小的值会导致非常低效的文件系统操作； 太大的值会使用页队列中缺少页面，而大量的页处于在线状态。

缺省情况下，FreeBSD内核编译时是不被优化的。 你可以在内核配置文件中用makeoptions 指定排错(debugging)和优化标志。注意，你一般不应使用-g， 除非你能够应付由此产生的大内核(典型的是7MB或更多)。

```
makeoptions     DEBUG="-g"
makeoptions     COPTFLAGS="-O --pipe"
```

Sysctl提供了在运行时调整内核的方式。你通常不需要指定任何sysctl变量， 尤其是与VM相关的那些变量。

运行时VM和系统调整的影响相对直接一些。 首先，应当尽可能在UFS/FFS文件系统上使用Soft Updates。 在/usr/src/sys/ufs/ffs/README.softupdates 里有关于如何配置的指示。

其次，应当配置足够多的交换空间。 你应当在每个物理磁盘上配置一个交换分区，最多4个， 甚至在你的“工作”磁盘上。你应当有至少2倍于主内存的交换空间； 假如你没有足够内存的话，交换分区还应更多。 你也应当按照你期望中的最大内存配置决定交换分区的大小， 这样以后就不再需要重新给磁盘分区了。 如果你处理系统崩溃后的内存倾倒(crash dump)， 第一个交换分区必须至少与主内存一样大， /var/crash必须有足够的空间来承装倾倒。

NFS上的交换分区可以很好的被4.X或后来的系统使用， 但是你必须明白NFS服务器将要经受页装载操作很强的冲击。

第 8 章 SMPng 设计文档

写作：John Baldwin 和 Robert Watson.
翻译：.

8.1. 绪论

这份文档对目前 SMPng 架构的设计与实现进行了介绍。 它首先介绍了基本的原语和相关工具， 其后是关于 FreeBSD 内核的同步与执行模型， 接下来讨论了具体系统中的锁策略， 并描述了在各个子系统中引入细粒度的同步和实现并行化的步骤， 最后是详细的实现说明， 用以解释最初做出某些设计决策的动机， 并使读者了解使用特定的原语所可能产生的重大影响。

这份文档仍在撰写当中， 并将不断更新以反映与 SMPng 项目有关的最新设计与实现的情况。 其中有许多小节目前还只是提纲， 但我们会逐渐为其充实内容。 关于这份文档的更新和建议， 请发给文档编辑。

SMPng 的目标是使内核能够并发执行。 基本上， 内核是一个很大而复杂的程序。 要让内核能够多线程地执行， 我们需要使用某些其它多线程程序在实现时所用到的工具， 这包括互斥体(mutex)、 共享/排他锁(shared/exclusive lock)、 信号量(semaphores) 和条件变量(condition variable)。 如果希望了解它们以及其它 SMP 术语， 请参阅本文的 术语表 一节。

8.2. 基本工具与上锁的基础知识

8.2.1. 原子操作指令和内存栅

关于内存栅和原子操作指令已经有很多介绍材料， 因此这一节并不打算对其进行详尽的介绍。 简而言之， 如果有对某一变量上写锁， 就不能在不获得相应的锁时对其进行读取操作。 也就是说， 内存栅的作用在于保证内存操作的相对顺序， 但并不保证内存操作的严格时序。 换言之， 内存栅并不保证 CPU 将本地快取缓存或存储缓冲的内容刷写回内存， 而是在锁释放时确保其所保护的数据， 对于能看到刚释放的那个锁的 CPU 或设备可见。 持有内存栅的 CPU 可以在其快取缓存或存储缓冲中将数据保持其所希望的、 任意长的时间， 但如果其它 CPU 在同一数据元上执行原子操作， 则第一个 CPU 必须保证， 其所更新的数据值， 以及内存栅所要求的任何其它操作， 对第二个 CPU 可见。

例如， 假设在一简单模型中， 认为在主存（或某一全局快取缓存）中的数据是可见的， 当某一 CPU 上触发原子操作时， 其它 CPU 的存储缓冲和快取缓存就必须对同一快取缓存线上的全部写操作， 以及内存栅之后的全部未完成操作进行刷写。

这样一来，在使用由原子操作保护的内存单元时就需要特别小心。例如，在实现 sleep mutex 时，我们就必须使用 atomic_cmpset 而不是 atomic_set 来打开 MTX_CONTESTED 位。这样做的原因是，我们需要把 mtx_lock 的值读到某个变量，并据此进行决策。然而，我们读到的值可能是过时的，也可能在我们进行决策的过程中发生变化。因此，当执行 atomic_set 时，最终可能会对另一值进行置位，而不是我们进行决策的那一个。这就必须通过 atomic_cmpset 来保证只有在我们的决策依据是最新的时，才对相应的变量进行置位。

最后，原子操作只允许一次更新或读一个内存单元。需要原子地更新多个单元时，就必须使用锁来代替它了。例如，如果需要更新两个相互关联的计数器时，就必须使用锁，而不是两次单独的原子操作了。

8.2.2. 读锁与写锁

读锁并不需要像写锁那样强。这两种类型的锁，都需要确保通过它们访问的不是过时的数据。然而，只有写操作必须是排他的，而多个线程则可以安全地读同一变量的值。使用不同类型的锁用于读和写操作有许多各自不同的实现方式。

第一种方法是用 sx 锁，它可以用于实现写时使用的排他锁，而读时则作为共享锁。这种方法十分简单明了。

第二种方法则略显晦涩。可以用多个锁来保护同一数据元。读时，只需锁其中的一个读锁即可。然而，如果要写数据的话，则需要首先上所有的写锁。这会大大提高写操作的代价，但当可能以多种方式访问数据时却可能非常有用。例如，父进程指针是同时受 proctree_lock sx 锁和进程 mutex 保护的。在只希望检查已锁进程的父进程时，用 proc 锁更为方便。但是，其它一些地方，例如 inferior 这类需要通过父指针在进程树上进行搜索，并对每个进程上锁的地方就不能这样做了，否则，将无法保证在对我们所获得的结果执行操作时，之前检查时的状况依旧有效。

8.2.3. 上锁状态和结果

如果您需要使用锁来保持所检查变量的状态，并据此执行某些操作时，是不能仅仅在读变量之前对其上锁，并在执行操作之前解锁的。过早解锁将使变量再次可变，这可能会导致之前所做的决策失效。因此，在所做检测引发的动作结束之前，必须继续保持上锁状态。

8.3. 架构与设计概览

8.3.1. 对中断的处理

与许多其它多线程 UNIX® 内核所采取的模式类似，FreeBSD 会赋予中断处理程序独立的线程上下文，这样做能够让中断线程在遇到锁时阻塞。但为了避免不必要

的延迟， 中断线程在内核中， 是以实时线程的优先级运行的。 因此， 中断处理程序不应执行过久， 以免饿死其它内核线程。 此外， 由于多个处理程序可以分享同一中断线程， 中断处理程序不应休眠， 或使用可能导致休眠的锁， 以避免将其它中断处理程序饿死。

目前在 FreeBSD 中的中断线程是指重量级中断线程。 这样称呼它们的原因在于，转到中断线程需要执行一次完整的上下文切换操作。 在最初的实现中， 内核不允许抢占， 因此中断在打断内核线程之前， 必须等待内核线程阻塞或返回用户态之后才能执行。

为了解决响应时间问题， FreeBSD 内核现在采用了抢占式调度策略。 目前， 只有释放休眠 mutex 或发生中断时才能抢断内核线程， 但最终目标是在 FreeBSD 上实现下面所描述的全抢占式调度策略。

并非所有的中断处理程序都在独立的线程上下文中执行。 相反， 某些处理程序会直接在主中断上下文中执行。 这些中断处理程序， 现在被错误地命名为 “快速” 中断处理程序， 因为早期版本的内核中使用了 INTR_FAST 标志来标记这些处理程序。 目前只有时钟中断和串口 I/O 设备中断采用这一类型。 由于这些处理程序没有独立的上下文， 因而它们都不能获得阻塞性锁， 因此也就只能使用自旋 mutex。

最后， 还有一种称为轻量级上下文切换的优化， 可以在 MD 代码中使用。 因为中断线程都是在内核上下文中执行的， 所以它可以借用任意进程的 vmspace （虚拟内存地址空间）。 因此， 在轻量级上下文切换中， 切换到中断线程并不切换对应的 vmspace， 而是借用被中断线程的 vmspace。 为确保被中断线程的 vmspace 不在中断处理过程中消失， 被中断线程在中断线程不再借用其 vmspace 之前是不允许执行的。 刚才提到的情况可能在中断线程阻塞或完成时发生。 如果中断线程发生阻塞， 则它再次进入可运行状态时将使用自己的上下文， 这样一来， 就可以释放被中断的线程了。

这种优化的坏处在于它们和硬件紧密相关， 而且实现比较复杂， 因此只有在这样做能带来大幅性能改善时才应采用。 目前这样说可能还为时过早， 而且事实上可能会反而导致性能下降， 因为几乎所有的中断处理程序都会立即被全局锁 (Giant) 阻塞， 而这种阻塞将进而需要线程修正。 另外， Mike Smith 提议采用另一种方式来处理中断线程:

1. 每个中断处理程序分为两部分， 一个在主中断上下文中运行的主体 (predicate) 和一个在自己的线程上下文中执行的处理程序 (handler)。

2. 如果中断处理程序拥有主体， 则当触发中断时， 执行该主体。 如果主体返回真， 则认为该中断被处理完毕， 内核从中断返回。 如果主体返回假， 或者中断没有主体， 则调度运行线程式处理程序。

在这一模式中适当地采用轻量级上下文切换可能是非常复杂的。 因为我们可能会希望在未来改变这一模式， 因此现在最好的方案， 应该是暂时推迟在轻量级上下

文切换之上的工作， 以便进一步完善中断处理架构， 随后再考察轻量级上下文切换是否适用。

8.3.2. 内核抢占与临界区

8.3.2.1. 内核抢占简介

内核抢占的概念很简单， 其基本思想是 CPU 总应执行优先级最高的工作。 当然， 至少在理想情况下是这样。 有些时候， 达成这一理想的代价会十分高昂， 以至于在这些情况下抢占会得不偿失。

实现完全的内核抢占十分简单： 在调度将要执行的线程并放入运行队列时， 检查它的优先级是否高于目前正在执行的线程。 如果是这样的话， 执行一次上下文切换并立即开始执行该线程。

尽管锁能够在抢占时保护多数数据， 但内核并不是可以安全地处处抢占的。 例如， 如果持有自旋 mutex 的线程被抢占， 而新线程也尝试获得同一自旋 mutex， 新线程就可能一直自旋下去， 因为被中断的线程可能永远没有机会运行了。 此外， 某些代码， 例如在 Alpha 上的 exec 对进程地址空间编号进行赋值的代码也不能被抢断， 因为它被用来支持实际的上下文切换操作。 在这些代码段中， 会通过使用临界区来临时禁用抢占。

8.3.2.2. 临界区

临界区 API 的责任是避免在临界区内发生上下文切换。 对于完全抢占式内核而言， 除了当前线程之外的其它线程的每个 setrunqueue 都是抢断点。 critical_enter 的一种实现方式是设置一线程私有标记， 并由其对应方清除。 如果调用 setrunqueue 时设置了这个标志， 则无论新线程和当前线程相比其优先级高低， 都不会发生抢占。 然而， 由于临界区会在自旋 mutex 中用于避免上下文切换， 而且能够同时获得多个自旋 mutex， 因此临界区 API 必须支持嵌套。 由于这个原因， 目前的实现中采用了嵌套计数， 而不仅仅是单个的线程标志。

为了尽可能缩短响应时间， 在临界区中的抢占被推迟， 而不是直接丢弃。 如果线程应被抢断， 并被置为可运行， 而当前线程处于临界区， 则会设置一线程私有标志， 表示有一个尚未进行的抢断操作。 当最外层临界区退出时， 会检查这一标志， 如果它被置位， 则当前线程会被抢断， 以允许更高优先级的线程开始运行。

中断会引发一个和自旋 mutex 有关的问题。 如果低级中断处理程序需要锁， 它就不能中断任何需要该锁的代码， 以避免可能发生的损坏数据结构的情况。 目前， 这一机制是透过临界区 API 以 cpu_critical_enter 和 cpu_critical_exit 函数的形式实现的。 目前这一 API 会在所有 FreeBSD 所支持的平台上禁用和重新启用中断。 这种方法并不是最优的， 但它更易理解， 也更容易正确地实现。 理论上， 这一辅助 API 只需要配合在主中断上下文中的自旋 mutex 使用。 然而， 为了让代码更为简单， 它被用在了全部自旋 mutex， 甚至包括所有临界区上。 将其从 MI API 中剥离出来放入 MD API， 并只在需要使用它的 MI API 的自旋 mutex 实现

中使用可能会有更好的效果。 如果我们最终采用了这种实现方式， 则 MD API 可能需要改名， 以彰显其为一单独 API 这一事实。

8.3.2.3. 设计折衷

如前面提到的， 当完全抢占并非总能提供最佳性能时， 采取了一些折衷的措施。

第一处折衷是， 抢占代码并不考虑其它 CPU 的存在。 假设我们有两个 CPU， A 和 B， 其中 A 上线程的优先级为 4， 而 B 上线程的优先级是 2。 如果 CPU B 令一优先级为 1 的线程进入可运行状态， 则理论上， 我们希望 CPU A 切换至这一新线程， 这样就有两个优先级最高的线程在运行了。 然而， 确定哪个 CPU 在抢占时更合适， 并通过 IPI 向那个 CPU 发出信号， 并完成相关的同步工作的代价十分高昂。 因此， 目前的代码会强制 CPU B 切换至更高优先级的线程。 请注意这样做仍会让系统进入更好的状态， 因为 CPU B 会去执行优先级为 1 而不是 2 的那个线程。

第二处折衷是限制对于实时优先级的内核线程的立即抢占。 在前面所定义的抢占操作的简单情形中， 低优先级总会被立即抢断 (或在其退出临界区后被抢断)。 然而， 许多在内核中执行的线程， 有很多只会执行很短的时间就会阻塞或返回用户态。 因此， 如果内核抢断这些线程并执行其它非实时的内核线程， 则内核可能会在这些线程马上要休眠或执行完毕之前切换出去。 这样一来， CPU 就必须调整快取缓存以配合新线程的执行。 当内核返回到被抢断的线程时， 它又需要重新填充之前丢失的快取缓存信息。 此外， 如果内核能够将对将阻塞或返回用户态的那个线程的抢断延迟到这之后的话， 还能够免去两次额外的上下文切换。 因此， 默认情况下， 只有在优先级较高的线程是实时线程时， 抢占代码才会立即执行抢断操作。

启用针对所有内核线程的完全抢占对于调试非常有帮助， 因为它会暴露出更多的竞态条件 (race conditions)。 在难以模拟这些竞态条件的单处理器系统中， 这显得尤其有用。 因此， 我们提供了内核选项 FULL_PREEMPTION 来启用针对所有内核线程的抢占， 这一选项主要用于调试目的。

8.3.3. 线程迁移

简单地说， 线程从一个 CPU 移动到另一个上的过程称作迁移。 在非抢占式内核中， 这只会在明确定义的点， 例如调用 msleep 或返回至用户态时才会发生。 但是， 在抢占式内核中， 中断可能会在任何时候强制抢断， 并导致迁移。 对于 CPU 私有的数据而言这可能会带来一些负面影响， 因为除 curthread 和 curpcb 以外的数据都可能在迁移过程中发生变化。 由于存在潜在的线程迁移， 使得未受保护的 CPU 私有数据访问变得无用。 这就需要在某些代码段禁止迁移， 以获得稳定的 CPU 私有数据。

目前我们采用临界区来避免迁移， 因为它们能够阻止上下文切换。 但是， 这有时可能是一种过于严厉的限制， 因为临界区实际上会阻止当前处理器上的中断线程。 因而， 提供了另一个 API， 用以指示当前进程在被抢断时， 不应迁移到另一 CPU。

这组 API 也叫线程牵制，它由调度器提供。这组 API 包括两个函数：sched_pin 和 sched_unpin。这两个函数用于管理线程私有的计数 td_pinned。如果嵌套计数大于零，则线程将被锁住，而线程开始运行时其嵌套计数为零，表示处于未牵制状态。所有的调度器实现中，都要求保证牵制线程只在它们首次调用 sched_pin 时所在的 CPU 上运行。由于只有线程自己会写嵌套计数，而只有其它线程在受牵制线程没有执行，且持有 sched_lock 锁时才会读嵌套计数，因此访问 td_pinned 不必上锁。sched_pin 函数会使嵌套计数递增，而 sched_unpin 则使其递减。注意，这些函数只操作当前线程，并将其绑定到其执行它时所处的 CPU 上。要将任意线程绑定到指定的 CPU 上，则应使用 sched_bind 和 sched_unbind。

8.3.4. 调出（Callout）

内核机制 timeout 允许内核服务注册函数，以作为 softclock 软件中断的一部分来执行。事件将基于所希望的时钟嘀嗒的数目进行，并在大约指定的时间回调用户提供的函数。

未决 timeout（超时）事件的全局表是由一全局 mutex，callout_lock 保护的；所有对 timeout 表的访问，都必须首先拿到这个 mutex。当 softclock 唤醒时，它会扫描未决超时表，并找出应启动的那些。为避免锁逆序，softclock 线程会在调用所提供的 timeout 回调函数时首先释放 callout_lock mutex。如果在注册时没有设置 CALLOUT_MPSAFE 标志，则在调用调出函数之前，还会抓取全局锁，并在之后释放。其后，callout_lock mutex 会在继续处理前再次获得。softclock 代码在释放这个 mutex 时会非常小心地保持表的一致状态。如果启用了 DIAGNOSTIC，则每个函数的执行时间会被记录，如果超过了某一阈值，则会产生警告。

8.4. 特定数据的锁策略

8.4.1. 凭据

struct ucred 是内核内部的凭据结构体，它通常作为内核中以进程为导向的访问控制的依据。BSD-派生的系统采用一种“写时复制”的模型来处理凭据数据：同一凭据结构体可能存在多个引用，如果需要对其进行修改，则这个结构体将被复制、修改，然后替换该引用。由于在打开时用于实现访问控制的凭据快取缓存广泛存在，这种做法会极大地节省内存。在迁移到细粒度的 SMP 时，这一模型也省去了大量的锁操作，因为只有未共享的凭据才能实施修改，因而避免了在使用共享凭据时额外的同步操作。

凭据结构体只有一个引用时，被认为是可变的；不允许改变共享的凭据结构体，否则将可能导致发生竞态条件。cr_mtxp mutex 用于保护 struct ucred 的引用计数，以维护其一致性。使用凭据结构体时，必须在使用过程中保持有效的引用，否则它就可能在这个不合理的消费者使用过程中被释放。

struct ucred mutex 是一种叶 mutex，出于性能考虑，它通过 mutex 池实现。

由于多用于访问控制决策，凭据通常情况下是以只读方式访问的，此时一般应使用 td_ucred，因为它不需要上锁。当更新进程凭据时，检查和更新过程中必须持有 proc 锁。检查和更新操作必须使用 p_ucred，以避免检查时和使用时的竞态条件。

如果所调系统调用将在更新进程凭据之后进行访问控制检查，则 td_ucred 也必须刷新为当前进程的值。这样做能够避免修改后使用过时的凭据。内核会自动在进程进入内核时，将线程结构体的 td_ucred 指针刷新为进程的 p_ucred，以保证内核访问控制能用到新的凭据。

8.4.2. 文件描述符和文件描述符表

详细内容将在稍后增加。

8.4.3. Jail 结构体

struct prison 保存了用于维护那些通过 jail(2) API 创建的 jail 所用到的管理信息。这包括 jail 的主机名、IP 地址，以及一些相关的设置。这个结构体包含引用计数，因为指向这一结构体实例的指针会在多种凭据结构之间共享。用了一个 mutex，pr_mtx 来保护对引用计数以及所有 jail 结构体中可变变量的读写访问。有一些变量只会在创建 jail 的时刻发生变化，只需持有有效的 struct prison 就可以开始读这些值了。关于每个项目具体的上锁操作的文档，可以在 sys/jail.h 的注释中找到。

8.4.4. MAC 框架

TrustedBSD MAC 框架会以 struct label 的形式维护一系列内核对象的数据。一般来说，内核中的 label（标签）是由与其对应的内核对象同样的锁保护的。例如，struct vnode 上的 v_label 标签是由其所在 vnode 上的 vnode 锁保护的。

除了嵌入到标准内核对象中的标签之外，MAC 框架也需要维护一组包含已注册的和激活策略的列表。策略表和忙计数由一个全局 mutex（mac_policy_list_lock）保护。由于能够同时并行地进行许多访问控制检查，对策略表的只读访问，在增减忙计数时，框架的入口处需要首先持有这个 mutex。MAC 入口操作的过程中并不需要长时间持有此 mutex — 有些操作，例如文件系统对象上的标签操作 — 是持久的。要修改策略表，例如在注册和解除注册策略时，需要持有此 mutex，而且要求引用计数为零，以避免在用表时对其进行修改。

对于需要等待表进入闲置状态的线程，提供了一个条件变量 mac_policy_list_not_busy，但这一条件变量只能在调用者没有持有其它锁时才能使用，否则可能会引发锁逆序问题。忙计数在整个框架中事实上还扮演了某种形式的 共享/排他 锁的作用：与 sx 锁不同的地方在于，等待列表进入闲置状态的线程可以饿死，而不是允许忙计数和其它在 MAC 框架入口（或内部）的锁之间的逆序情况。

8.4.5. 模块

对于模块子系统， 用于保护共享数据使用了一个单独的锁， 它是一个 共享/排他 (SX) 锁， 许多情况需要获得它 (以共享或排他的方式)， 因此我们提供了几个方便使用的宏来简化对这个锁的访问， 这些宏可以在 sys/module.h 中找到， 其用法都非常简单明了。 这个锁保护的主要是 module_t (当以共享方式上锁) 和全局的 modulelist_t 这两个结构体， 以及模块。 要更进一步理解这些锁策略， 需要仔细阅读 kern/kern_module.c 的源代码。

8.4.6. Newbus 设备树

newbus 系统使用了一个 sx 锁。 读的一方应持有共享 (读) 锁 (sx_slock(9)) 而写的一方则应持有排他 (写) 锁 (sx_xlock(9))。 内部函数一般不需要进行上锁， 而外部可见的则应根据需要上锁。 有些项目不需上锁， 因为这些项目在全程是只读的， (例如 device_get_softc(9))， 因而并不会产生竞态条件。 针对 newbus 数据结构的修改相对而言非常少， 因此单个的锁已经足够使用， 而不致造成性能折损。

8.4.7. 管道

...

8.4.8. 进程和线程

- 进程层次结构

- proc 锁及其参考

- 在系统调用过程中线程私有的 proc 项副本， 包括 td_ucred

- 进程间操作

- 进程组和会话

8.4.9. 调度器

本文在其它地方已经提供了很多关于 sched_lock 的参考和注释。

8.4.10. Select 和 Poll

select 和 poll 这两个函数允许线程阻塞并等待文件描述符上的事件 — 最常见的情况是文件描述符是否可读或可写。

...

8.4.11. SIGIO

SIGIO 服务允许进程请求在特定文件描述符的读/写状态发生变化时， 将 SIGIO 信号群发给其进程组。 任意给定内核对象上， 只允许一进程或进程组注册 SIGIO，这个进程或进程组称为属主 (owner)。 每一支持 SIGIO 注册的对象， 都包含一指针字段， 如果对象未注册则为 NULL， 否则是一指向描述这一注册的 struct sigio 的指针。 这一字段由一全局 mutex， sigio_lock 保护。 调用 SIGIO 维护函数时，必须以 “传引用” 方式传递这一字段， 以确保本地注册副本的中这个字段不脱离锁的保护。

每个关联到进程或进程组的注册对象， 都会分配一 struct sigio 结构， 并包括指回该对象的指针、 属主、 信号信息、 凭据， 以及关于这一注册的一般信息。 每个进程或进程组都包含一个已注册 struct sigio 结构体的列表， 对进程来说是 p_sigiolst， 而对进程组则是 pg_sigiolst。 这些表由相应的进程或进程组锁保护。 除了用以将 struct sigio 连接到进程组上的 sio_pgsigio 字段之外， 在 struct sigio 中的多数字段在注册过程中都是不变量。 一般而言， 开发人员在实现新的支持 SIGIO 的内核对象时， 会希望避免在调用 SIGIO 支持函数， 例如 fsetown 或 funsetown 持有结构体锁， 以免去需要在结构体锁和全局 SIGIO 锁之间定义锁序。 通常可以通过提高结构体上的引用计数来达到这样的目的， 例如， 在进行管道操作时， 使用引用某个管道的文件描述符这样的操作， 就可以照此办理。

8.4.12. Sysctl

sysctl MIB 服务会从内核内部， 以及用户态的应用程序以系统调用的方式触发。 这会引发至少两个和锁有关的问题： 其一是对维持命名空间的数据结构的保护， 其二是与那些通过 sysctl 接口访问的内核变量和函数之间的交互。 由于 sysctl 允许直接导出 (甚至修改) 内核统计数据以及配置参数， sysctl 机制必须知道这些变量相应的上锁语义。 目前， sysctl 使用一个全局 sx 锁来实现对 sysctl 操作的串行化； 然而， 这些是假定用全局锁保护的， 并且没有提供其它保护机制。 这一节的其余部分将详细介绍上锁和 sysctl 相关变动的语义。

- 需要将 sysctl 更新值所进行的操作的顺序， 从原先的读旧值、 copyin 和 copyout、 写新值， 改为 copyin、 上锁、 读旧值、 写新值、 解锁、 copyout。 一般的 sysctl 只是 copyout 旧值并设置它们 copyin 所得到的新值， 仍然可以采用旧式的模型。 然而， 对所有 sysctl 处理程序采用第二种模型并避免锁操作方面， 第二种方式可能更规矩一些。

- 对于通常的情况， sysctl 可以内嵌一个 mutex 指针到 SYSCTL_FOO 宏和结构体中。 这对多数 sysctl 都是有效的。 对于使用 sx 锁、 自旋 mutex， 或其它除单一休眠 mutex 之外的锁策略， 可以用 SYSCTL_PROC 节点来完成正确的上锁。

8.4.13. 任务队列 (Taskqueue)

任务队列 (taskqueue) 的接口包括两个与之关联的用于保护相关数据的锁。 taskqueue_queues_mutex 是用于保护 taskqueue_queues TAILQ 的锁。 与这个系

统关联的另一个 mutex 锁是位于 struct taskqueue 结构体上。 在此处使用同步原语的目的在于保护 struct taskqueue 中数据的完整性。 应注意的是， 并没有单独的、 帮助用户对其自身的工作进行锁的细化用的宏， 因为这些锁基本上不会在 kern/subr_taskqueue.c 以外的地方用到。

8.5. 实现说明

8.5.1. 休眠队列

休眠队列是一种用于保存同处一个等待通道 (wait channel) 上休眠线程列表的数据结构。 在等待通道上， 每个处于非睡眠状态的线程都会携带一个休眠队列结构。 当线程在等待通道上发生阻塞时， 它会将休眠队列结构体送给那个等待通道。 与等待通道关联的休眠队列则保存在一个散列表中。

休眠队列散列表中保存了包含至少一个阻塞线程的等待通道上的休眠队列。 这个散列表上的项称作 sleepqueue (休眠队列) 链。 它包含了一个休眠队列的链表， 以及一个自旋 mutex。 此处的自旋 mutex 用于保护休眠队列表， 以及其上休眠队列结构的内容。 一个等待通道上只会关联一个休眠队列。 如果有多个线程在同一等待通道上阻塞， 则休眠队列中将关联除第一个线程之外的全部线程。 当从休眠队列中删除线程时， 如果它不是唯一的阻塞的休眠线程， 则会获得主休眠队列的空闲表上的休眠队列结构。 最后一个线程会在恢复运行时获得主休眠队列。 由于线程有可能以和加入休眠队列不同的次序从其中删除， 因此， 线程离开队列时可能会携带与其进入时不同的休眠队列。

sleepq_lock 函数会锁住指定等待通道上休眠队列链的自旋 mutex。 sleepq_lookup 函数会在主休眠队列散列表中查找给定的等待通道。 如果没有找到主休眠队列， 它会返回 NULL。 sleepq_release 函数会对给定等待通道所关联的自旋 mutex 进行解锁。

将线程加入休眠队列是通过 sleepq_add 来完成的。 这个函数的参数包括等待通道、 指向保护等待通道的 mutex 的指针、 等待消息描述串， 以及一个标志掩码。 调用此函数之前， 应通过 sleepq_lock 为休眠队列链上锁。 如果等待通道不是通过 mutex 保护的 (或者它由全局锁保护)， 则应将 mutex 指针设置为 NULL。 而 flags (标志) 参数则包括了一个类型字段， 用以表示线程即将加入到的休眠队列的类型， 以及休眠是否是可中断的 (SLEEPQ_INTERRUPTIBLE)。 目前只有两种类型的休眠队列： 通过 msleep 和 wakeup 函数管理的传统休眠队列 (SLEEPQ_MSLEEP)， 以及基于条件变量的休眠队列 (SLEEPQ_CONDVAR)。 休眠队列类型和锁指针这两个参数完全是用于内部的断言检查。 调用 sleepq_add 的代码， 应明示地在关联的 sleepqueue 链透过 sleepq_lock 进行上锁之后， 并使用等待函数在休眠队列上阻塞之前解锁所有用于保护等待通道的 interlock。

通过使用 sleepq_set_timeout 可以为休眠设置超时。 这个函数的参数包括等待通道， 以及以相对时钟嘀嗒数为单位的超时时间。 如果休眠应被某个到来的信号打

断， 则还应调用 sleepq_catch_signals 函数， 这个函数唯一的参数就是等待通道。如果此线程已经有未决信号， 则 sleepq_catch_signals 将返回信号编号； 其它情况下， 其返回值则是 0。

一旦将线程加入到休眠队列中， 就可以使用 sleepq_wait 函数族之一将其阻塞了。 目前总共提供了四个等待函数， 使用哪个取决于调用这是否希望允许使用超时、 收到信号， 或用户态线程调度器打断休眠状态。 其中， sleepq_wait 函数简单地等待， 直到当前线程通过某个唤醒 (wakeup) 函数显式地恢复运行； sleepq_timedwait 函数则等待， 直到当前线程被显式地唤醒， 或者达到早前使用 sleepq_set_timeout 设置的超时； sleepq_wait_sig 函数会等待显式地唤醒， 或者其休眠被中断； 而 sleepq_timedwait_sig 函数则等待显式地唤醒、 达到用 sleepq_set_timeout 设置的超时， 或线程的休眠被中断这三种条件之一。 所有这些等待函数的第一个参数都是等待通道。 除此之外， sleepq_timedwait_sig 的第二个参数是一个布尔值， 表示之前调用 sleepq_catch_signals 时是否有发现未决信号。

如果线程被显式地恢复运行， 或其休眠被信号终止， 则等待函数会返回零， 表示休眠成功。 如果线程的休眠被超时或用户态线程调度器打断， 则会返回相应的 errno 数值。 需要注意的是， 因为 sleepq_wait 只能返回 0， 因此调用者不能指望它返回什么有用信息， 而应假定它完成了一次成功的休眠。 同时， 如果线程的休眠时间超时， 并同时被终止， 则 sleepq_timedwait_sig 将返回一个表示发生超时的错误代码。 如果返回错误代码是 0 而且使用 sleepq_wait_sig 或 sleepq_timedwait_sig 来执行阻塞， 则应调用 sleepq_calc_signal_retval 来检查是否有未决信号， 并据此选择合适的返回值。 较早前调用 sleepq_catch_signals 得到的信号编号， 应作为参数传给 sleepq_calc_signal_retval。

在同一休眠通道上休眠的线程， 可以由 sleepq_broadcast 或 sleepq_signal 函数来显式地唤醒。 这两个函数的参数均包括希望唤醒的等待通道、 将唤醒线程的优先级 (priority) 提高到多少， 以及一个标志 (flags) 参数表示将要恢复运行的休眠队列类型。 优先级参数将作为最低优先级， 如果将恢复的线程的优先级比此参数更高 (数值更低) 则其优先级不会调整。 标志参数主要用于函数内部的断言， 用以确认休眠队列没有被当做错误的类型对待。 例如， 条件变量函数不应恢复传统休眠队列的执行。 sleepq_broadcast 函数将恢复所有指定休眠通道上的阻塞线程， 而 sleepq_signal 则只恢复在等待通道上优先级最高的阻塞线程。 在调用这些函数之前， 应首先使用 sleepq_lock 对休眠队列上锁。

休眠线程也可以通过调用 sleepq_abort 函数来中断其休眠状态。 这个函数只有在持有 sched_lock 时才能调用， 而且线程必须处于休眠队列之上。 线程也可以通过使用 sleepq_remove 函数从指定的休眠队列中删除。 这个函数包括两个参数， 即休眠通道和线程， 它只在线程处于指定休眠通道的休眠队列之上时才将其唤醒。 如果线程不在那个休眠队列之上， 或同时处于另一等待通道的休眠队列上， 则这个函数将什么都不做而直接返回。

8.5.2. 十字转门 (turnstile)

- 与休眠队列的比较和不同。

- 查询/等待/释放 (lookup/wait/release) - 介绍 TDF_TSNOBLOCK 竞态条件。

- 优先级传播。

8.5.3. 关于 mutex 实现的一些细节

- 我们是否应要求 mtx_destroy() 持有 mutex, 因为无法安全地断言它们没有被其它对象持有?

8.5.3.1. 自旋 mutex

- 使用一临界区...

8.5.3.2. 休眠 mutex

- 描述 mutex 冲突时的竞态条件

- 为何在持有十字转门链锁时, 可以安全地读冲突 mutex 的 mtx_lock。

8.5.4. Witness

- 它能做什么

- 它如何工作

8.6. 其它话题

8.6.1. 中断源和 ICU 抽象

- struct isrc

- pic 驱动

8.6.2. 其它问题/话题

- 是否应将 interlock 传给 sema_wait?

- 是否应提供非休眠式 sx 锁?

- 增加一些关于正确使用引用计数的介绍。

术语表

原子	当遵循适当的访问协议时, 如果一操作的效果对其它所有 CPU 均可见, 则称其为原子操作。

	狭义的原子操作是机器直接提供的。 就更高的抽象层次而言， 如果结构体的多个成员由一个锁保护， 则如果对它们的操作都是在上锁后、解锁前进行的， 也可以称其为原子操作。 参见操作.
阻塞	线程等待锁、 资源或条件时被阻塞。 这一术语也因此被赋予了太多的意涵。 参见休眠.
临界区	不允许发生抢占的代码段。 使用 critical_enter(9) API 来表示进入和退出临界区。
MD	表示与机器/平台有关。 参见MI.
内存操作	内存操作包括读或写内存中的指定位置。
MI	表示与机器/平台无关。 参见MD.
操作	见内存操作.
主中断上下文	主中断上下文表示当发生中断时所执行的那段代码。 这些代码可以直接运行某个中断处理程序， 或调度一异步终端线程， 以便为给定的中断源执行中断处理程序。
实时内核线程	一种高优先级的内核线程。 目前， 只有中断线程属于实时优先级的内核线程。 参见线程.
休眠	当进程由条件变量或通过 msleep 或 tsleep 阻塞并进入休眠队列时， 称其进入休眠状态。 参见阻塞.
可休眠锁	可休眠锁是一种在进程休眠时仍可持有的锁。 锁管理器 (lockmgr) 锁和 sx 锁是目前 FreeBSD 中仅有的可休眠锁。 最终， 某些 sx 锁， 例如 allproc (全部进程) 和 proctree (进程树) 锁将成为不可休眠锁。 参见休眠.
线程	由 struct thread 所表达的内核线程。 线程可以持有锁， 并拥有独立的执行上下文。
等待通道	线程可以在其上休眠的内核虚拟地址。

部分 II. 设备驱动程序

目录

第 9 章 编写 FreeBSD 设备驱动程序

写作：Murray Stokely.
基础性手册intro(4)：Jörg Wunsch.
翻译：.

9.1. 简介

本章简要介绍了如何为FreeBSD编写设备驱动程序。术语设备在 这儿的上下文中多用于指代系统中硬件相关的东西，如磁盘，打印机， 图形显式器及其键盘。设备驱动程序是操作系统中用于控制特定设备的 软件组件。也有所谓的伪设备，即设备驱动程序用软件模拟设备的行为， 而没有特定的底层硬件。设备驱动程序可以被静态地编译进系统，或者 通过动态内核链接工具‘k1d’在需要时加载。

类UNIX®操作系统中的大多数设备都是通过设备节点来访问的，有时也 被称为特殊文件。这些文件在文件系统的层次结构中通常位于 /dev目录下。在FreeBSD 5.0-RELEASE以前的 发行版中，对devfs(5)的支持还没有被集成到FreeBSD中，每个设备 节点必须要静态创建，并且独立于相关设备驱动程序的存在。系统中大 多数设备节点是通过运行MAKEDEV创建的。

设备驱动程序可以粗略地分为两类，字符和网络设备驱动程序。

9.2. 动态内核链接工具—KLD

k1d接口允许系统管理员从运行的系统中动态地添加和删除功能。 这允许设备驱动程序的编写者将他们的新改动加载到运行的内核中， 而不用为了测试新改动而频繁地重启。

k1d接口通过下面的特权命令使用：

- kldload - 加载新内核模块
- kldunload - 卸载内核模块
- kldstat - 列举当前加载的模块

内核模块的程序框架

```
/*
 * KLD####
```

```
 * #Andrew Reiter#Daemonnews#######
 */

#include <sys/types.h>
#include <sys/module.h>
#include <sys/systm.h>  -/* uprintf */
#include <sys/errno.h>
#include <sys/param.h>  -/* kernel.h###### */
#include <sys/kernel.h> -/* ########### */

/*
 * ###########KLD#######
 */

static int
skel_loader(struct module *m, int what, void *arg)
{
  int err = 0;

  switch (what) {
  case MOD_LOAD:                -/* kldload */
    uprintf("Skeleton KLD loaded.\n");
    break;
  case MOD_UNLOAD:
    uprintf("Skeleton KLD unloaded.\n");
    break;
  default:
    err = EOPNOTSUPP;
    break;
  -}
  return(err);
}

/* ############ */

static moduledata_t skel_mod = {
  -"skel",
  skel_loader,
  NULL
};

DECLARE_MODULE(skeleton, skel_mod, SI_SUB_KLD, SI_ORDER_ANY);
```

9.2.1. Makefile

FreeBSD提供了一个makefile包含文件，利用它你可以快速地编译 你附加到内核的东西。

```
SRCS=skeleton.c
KMOD=skeleton

.include <bsd.kmod.mk>
```

简单地用这个makefile运行make就能够创建文件 skeleton.ko，键入如下命令可以把它加载到内核：

```
# kldload -v ./skeleton.ko
```

9.3. 访问设备驱动程序

UNIX® 提供了一套公共的系统调用供用户的应用程序使用。当用户访问 设备节点时，内核的上层将这些调用分发到相应的设备驱动程序。脚本 /dev/MAKEDEV为你的系统生成了大多数的设备节点， 但如果你正在开发你自己的驱动程序，可能需要用 mknod创建你自己的设备节点。

9.3.1. 创建静态设备节点

mknod命令需要四个参数来创建设备节点。你必须指定设备节点的名字，设备的类型，设备的主号码和设备的从号码。

9.3.2. 动态设备节点

设备文件系统，或者说devfs，在全局文件系统名字空间中提供对 内核设备名字空间的访问。这消除了由于有设备驱动程序而没有静态 设备节点，或者有设备节点而没有安装设备驱动程序而带来的潜在问题。 Devfs仍在进展中，但已经能够工作得相当好了。

9.4. 字符设备

字符设备驱动程序直接从用户进程传输数据，或传输数据到用户进程。 这是最普通的一类设备驱动程序，源码树中有大量的简单例子。

这个简单的伪设备例子会记住你写给它的任何值，并且当你读取它的时候 会将这些值返回给你。下面显示了两个版本，一个适用于FreeBSD# 4.X， 一个适用于FreeBSD#5.X。

例 9.1. 适用于FreeBSD#4.X的回显伪设备驱动程序实例

```
/*
 * ##‘echo’###KLD
 *
 * Murray Stokely
 */
```

```
#define MIN(a,b) (((a) < (b)) -? (a) -: (b))

#include <sys/types.h>
#include <sys/module.h>
#include <sys/systm.h>  -/* uprintf */
#include <sys/errno.h>
#include <sys/param.h>  -/* kernel.h###### */
#include <sys/kernel.h> -/* ########### */
#include <sys/conf.h>   -/* cdevsw## */
#include <sys/uio.h>    -/* uio## */
#include <sys/malloc.h>

#define BUFFERSIZE 256

/* #### */
d_open_t echo_open;
d_close_t echo_close;
d_read_t echo_read;
d_write_t echo_write;

/* ####### */
static struct cdevsw echo_cdevsw = {
 echo_open,
 echo_close,
 echo_read,
 echo_write,
 noioctl,
 nopoll,
 nommap,
 nostrategy,
 "echo",
 33,            -/* #lkms## -- -/usr/src/sys/conf/majors */
 nodump,
 nopsize,
 D_TTY,
 -1
};

typedef struct s_echo {
 char msg[BUFFERSIZE];
 int len;
} t_echo;

/* ## */
static dev_t sdev;
static int count;
static t_echo *echomsg;

MALLOC_DECLARE(M_ECHOBUF);
MALLOC_DEFINE(M_ECHOBUF, -"echobuffer", -"buffer for echo module");

/*
```

```
 * #####kld[un]load(2)########
 * ###################
 */

static int
echo_loader(struct module *m, int what, void *arg)
{
 int err = 0;

 switch (what) {
 case MOD_LOAD:                -/* kldload */
  sdev = make_dev(&echo_cdevsw,
      0,
      UID_ROOT,
      GID_WHEEL,
      0600,
      -"echo");
  /* kmalloc########### */
  MALLOC(echomsg, t_echo *, sizeof(t_echo), M_ECHOBUF, M_WAITOK);
  printf("Echo device loaded.\n");
  break;
 case MOD_UNLOAD:
  destroy_dev(sdev);
  FREE(echomsg,M_ECHOBUF);
  printf("Echo device unloaded.\n");
  break;
 default:
  err = EOPNOTSUPP;
  break;
 }
 return(err);
}

int
echo_open(dev_t dev, int oflags, int devtype, struct proc *p)
{
 int err = 0;

 uprintf("Opened device \"echo\" successfully.\n");
 return(err);
}

int
echo_close(dev_t dev, int fflag, int devtype, struct proc *p)
{
 uprintf("Closing device \"echo.\"\n");
 return(0);
}

/*
 * read#####echo_write()###buf###########
 * #########
 * uio(9)
```

```
 */

int
echo_read(dev_t dev, struct uio *uio, int ioflag)
{
 int err = 0;
 int amt;

 /*
  * #########
  * #######################
  */
 amt = MIN(uio->uio_resid, (echomsg->len -- uio->uio_offset > 0) -?
     echomsg->len -- uio->uio_offset -: 0);
 if ((err = uiomove(echomsg->msg + uio->uio_offset,amt,uio)) -!= 0) {
  uprintf("uiomove failed!\n");
 }
 return(err);
}

/*
 * echo_write########################
 */

int
echo_write(dev_t dev, struct uio *uio, int ioflag)
{
 int err = 0;

 /* ################### */
 err = copyin(uio->uio_iov->iov_base, echomsg->msg,
     MIN(uio->uio_iov->iov_len, BUFFERSIZE -- 1));

 /* #####null########### */
 *(echomsg->msg + MIN(uio->uio_iov->iov_len, BUFFERSIZE -- 1)) = 0;
 echomsg->len = MIN(uio->uio_iov->iov_len, BUFFERSIZE);

 if (err -!= 0) {
  uprintf("Write failed: bad address!\n");
 }
 count++;
 return(err);
}

DEV_MODULE(echo,echo_loader,NULL);
```

例 9.2. 适用于FreeBSD#5.X回显伪设备驱动程序实例

```
/*
 * ##‘echo’### KLD
 *
 * Murray Stokely
 *
 * ####Søren (Xride) Straarup###5.X
 */

#include <sys/types.h>
#include <sys/module.h>
#include <sys/systm.h>  -/* uprintf */
#include <sys/errno.h>
#include <sys/param.h>  -/* kernel.h###### */
#include <sys/kernel.h> -/* ########### */
#include <sys/conf.h>   -/* cdevsw## */
#include <sys/uio.h>    -/* uio## */
#include <sys/malloc.h>

#define BUFFERSIZE 256


/* #### */
static d_open_t      echo_open;
static d_close_t     echo_close;
static d_read_t      echo_read;
static d_write_t     echo_write;

/* ####### */
static struct cdevsw echo_cdevsw = {
 .d_version = D_VERSION,
 .d_open = echo_open,
 .d_close = echo_close,
 .d_read = echo_read,
 .d_write = echo_write,
 .d_name = -"echo",
};

typedef struct s_echo {
 char msg[BUFFERSIZE];
 int len;
} t_echo;

/* ## */
static struct cdev *echo_dev;
static int count;
static t_echo *echomsg;
```

```
MALLOC_DECLARE(M_ECHOBUF);
MALLOC_DEFINE(M_ECHOBUF, -"echobuffer", -"buffer for echo module");

/*
 * #####kld[un]load(2)#######,
 * ##################.
 */

static int
echo_loader(struct module *m, int what, void *arg)
{
 int err = 0;

 switch (what) {
 case MOD_LOAD:                -/* kldload */
  echo_dev = make_dev(&echo_cdevsw,
      0,
      UID_ROOT,
      GID_WHEEL,
      0600,
      -"echo");
  /* kmalloc############ */
  echomsg = malloc(sizeof(t_echo), M_ECHOBUF, M_WAITOK);
  printf("Echo device loaded.\n");
  break;
 case MOD_UNLOAD:
  destroy_dev(echo_dev);
  free(echomsg, M_ECHOBUF);
  printf("Echo device unloaded.\n");
  break;
 default:
  err = EOPNOTSUPP;
  break;
 }
 return(err);
}

static int
echo_open(struct cdev *dev, int oflags, int devtype, struct thread *p)
{
 int err = 0;

 uprintf("Opened device \"echo\" successfully.\n");
 return(err);
}

static int
echo_close(struct cdev *dev, int fflag, int devtype, struct thread *p)
{
 uprintf("Closing device \"echo.\"\n");
 return(0);
}
```

```
/*
 * read#####echo_write()###buf############
 * #########
 * uio(9)
 */

static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
 int err = 0;
 int amt;

 /*
  * #########
  * ######################
  */
 amt = MIN(uio->uio_resid, (echomsg->len -- uio->uio_offset > 0) -?
      echomsg->len -- uio->uio_offset -: 0);
 if ((err = uiomove(echomsg->msg + uio->uio_offset, amt, uio)) -!= 0) {
  uprintf("uiomove failed!\n");
 }
 return(err);
}

/*
 * echo_write################, #######.
 */

static int
echo_write(struct cdev *dev, struct uio *uio, int ioflag)
{
 int err = 0;

 /* ################### */
 err = copyin(uio->uio_iov->iov_base, echomsg->msg,
     MIN(uio->uio_iov->iov_len, BUFFERSIZE -- 1));

 /* #####null########### */
 *(echomsg->msg + MIN(uio->uio_iov->iov_len, BUFFERSIZE -- 1)) = 0;
 echomsg->len = MIN(uio->uio_iov->iov_len, BUFFERSIZE);

 if (err -!= 0) {
  uprintf("Write failed: bad address!\n");
 }
 count++;
 return(err);
}

DEV_MODULE(echo,echo_loader,NULL);
```

在FreeBSD#4.X上安装此驱动程序，你将首先需要用如下命令在 你的文件系统上创建一个节点：

```
# mknod -/dev/echo c 33 0
```

驱动程序被加载后，你应该能够键入一些东西，如：

```
# echo --n -"Test Data" > -/dev/echo
# cat -/dev/echo
Test Data
```

真正的硬件设备在下一章描述。

补充资源

- Dynamic Kernel Linker (KLD) Facility Programming Tutorial - Daemonnews October 2000
- How to Write Kernel Drivers with NEWBUS - Daemonnews July 2000

9.5. 块设备(消亡中)

其他UNIX®系统支持另一类型的磁盘设备，称为块设备。块设备是内核 为它们提供缓冲的磁盘设备。这种缓冲使得块设备几乎没有用，或者说非常 不可靠。缓冲会重新安排写操作的次序，使得应用程序丧失了在任何时刻及时 知道准确的磁盘内容的能力。这导致对磁盘数据结构（文件系统，数据库等）的 可预测的和可靠的崩溃恢复成为不可能。由于写操作被延迟，内核无法向应用 程序报告哪个特定的写操作遇到了写错误，这又进一步增加了一致性问题。 由于这个原因，真正的应用程序从不依赖于块设备，事实上，几乎所有访问 磁盘的应用程序都尽力指定总是使用字符（或“raw”）设备。 由于实现将每个磁盘（分区）同具有不同语义的两个设备混为一谈，从而致使 相关内核代码极大地复杂化，作为推进磁盘I/O基础结构现代化的一部分，FreeBSD 抛弃了对带缓冲的磁盘设备的支持。

9.6. 网络设备驱动程序

访问网络设备的驱动程序不需要使用设备节点。选择哪个驱动程序是 基于内核内部的其他决定而不是调用open()，对网络设备的使用通常由 系统调用socket(2)引入。

更多细节， 请参见 ifnet(9) 联机手册、 回环设备的源代码， 以及 Bill Paul 撰写的网络驱动程序。

第 10 章 ISA设备驱动程序

写作：Sergey Babkin.
改编为手册：Murray Stokely，Valentino Vaschetto 和 Wylie Stilwell.
翻译：.

10.1. 概述

本章介绍了编写ISA设备驱动程序相关的一些问题。这儿展示的伪代码 相当详细，很容易让人联想到真正的代码，不过这依然仅仅是伪代码。 它避免了与所讨论的主题无关的细节。真实的例子可以在实际驱动程序的 源代码中找到。ep和aha 更是信息的好来源。

10.2. 基本信息

典型的ISA驱动程序需要以下包含文件：

```
#include <sys/module.h>
#include <sys/bus.h>
#include <machine/bus.h>
#include <machine/resource.h>
#include <sys/rman.h>

#include <isa/isavar.h>
#include <isa/pnpvar.h>
```

它们描述了针对ISA和通用总线子系统的东西。

总线子系统是以面向对象的方式实现的，其主要结构通过相关联的 方法函数来访问。

ISA驱动程序实现的总线方法的列表与任何其他总线的很相似。 对于名字为“xxx”的假想驱动程序，它们将是：

- static void xxx_isa_identify (driver_t *, device_t); 通常用于总线驱动程序而不是设备驱动程序。 但对于ISA设备，这个方法有特殊用途：如果设备提供某些设备特定的 （非PnP）方法自动侦测设备，这个例程可以实现它。
- static int xxx_isa_probe (device_t dev); 在已知（或PnP）位置探测设备。对于已经部分配置的 设备，这个例程也能够提供设备特定的对某些参数的自动侦测。
- static int xxx_isa_attach (device_t dev); 挂接和初始化设备。
- static int xxx_isa_detach (device_t dev); 卸载设备驱动模块前解挂设备。

- static int xxx_isa_shutdown (device_t dev); 系统关闭前执行设备的关闭。
- static int xxx_isa_suspend (device_t dev); 系统进入节能状态前挂起设备。也可以中止 切换到节能状态。
- static int xxx_isa_resume (device_t dev); 从节能状态返回后恢复设备的活动状态。

xxx_isa_probe()和 xxx_isa_attach()是必须提供的，其余例程根据设备的 需要可以有选择地实现。

使用下面一组描述符将设备驱动链接到系统。

```
  -/* ######## */
  static device_method_t xxx_isa_methods[] = {
    -/* ################# */
    -/* ######## */
    DEVMETHOD(device_identify,  xxx_isa_identify),
    DEVMETHOD(device_probe,     xxx_isa_probe),
    DEVMETHOD(device_attach,    xxx_isa_attach),
    DEVMETHOD(device_detach,    xxx_isa_detach),
    DEVMETHOD(device_shutdown,  xxx_isa_shutdown),
    DEVMETHOD(device_suspend,   xxx_isa_suspend),
    DEVMETHOD(device_resume,    xxx_isa_resume),

{ 0, 0 -}
  -};

  static driver_t xxx_isa_driver = {
    -"xxx",
    xxx_isa_methods,
    sizeof(struct xxx_softc),
  -};


  static devclass_t xxx_devclass;

  DRIVER_MODULE(xxx, isa, xxx_isa_driver, xxx_devclass,
    load_function, load_argument);
```

此处的结构xxx_softc是一个设备 特定的结构，它包含私有的驱动程序数据和驱动程序资源的描述符。 总线代码会自动按需要为每个设备分配一个softc描述符。

如果驱动程序作为可加载模块实现，当驱动程序被加载或卸载时， 会调用load_function()函数进行驱动程序特定的 初始化或清理工作，并将load_argument作为函数的一个参量传递进去。 如果驱动程序不支持动态加载（换句话说，它必须被链接到内核中），则 这些值应当被设置为0，最后的定义将看起来如下所示：

```
DRIVER_MODULE(xxx, isa, xxx_isa_driver,
    xxx_devclass, 0, 0);
```

如果驱动程序是为支持PnP的设备而写的，那么就必须定义一个包含 所有支持的PnP ID的表。这个表由此驱动程序所支持的PnP ID的列表 和以人可读的形式给出的、与这些ID对应的硬件类型和型号的描述 组成。看起来如下：

```
static struct isa_pnp_id xxx_pnp_ids[] = {
    -/* ######PnP ID### */
    { 0x12345678,   -"Our device model 1234A" -},
    { 0x12345679,   -"Our device model 1234B" -},
    { 0,        NULL -}, -/* ### */
-};
```

如果驱动程序不支持PnP设备，它仍然需要一个空的PnP ID表， 如下所示：

```
static struct isa_pnp_id xxx_pnp_ids[] = {
    { 0,        NULL -}, -/* ### */
-};
```

10.3. Device_t指针

Device_t是为设备结构而定义的指针类型， 这里我们只关心从设备驱动程序编写者的角度看感兴趣的方法。下面的方法 用来操纵设备结构中的值：

- device_t device_get_parent(dev) 获取设备的父总线。
- driver_t device_get_driver(dev) 获取指向其驱动程序结构的指针。
- char *device_get_name(dev) 获取驱动程序的名字，在我们的 例子中为"xxx"。
- int device_get_unit(dev) 获取单元号（与每个驱动程序关联的设备从0开始编号）。
- char *device_get_nameunit(dev) 获取设备名，包括单元号。 例如“xxx0”， “xxx1” 等。
- char *device_get_desc(dev) 获取设备描述。通常它以人可读的 形式描述设备的确切型号。
- device_set_desc(dev, desc)设置描述信息。这使得设备描述指向desc字符串， 此后这个字符串就不能被解除分配。
- device_set_desc_copy(dev, desc) 设置描述信息。描述被拷贝到内部动态分配的 缓冲区，这样desc字符串在以后可以被改变而不会产生有害的结果。
- void *device_get_softc(dev) 获取指向与设备关联的设备描述符 （xxx_softc结构）的指针。
- u_int32_t device_get_flags(dev) 获取配置文件中特定于设备的 标志。

可以使用一个很方便的函数device_printf(dev, fmt, ...)从设备驱动程序中打印讯息。它自动在讯息前添加 单元名和冒号。

device_t的这些方法在文件kern/bus_subr.c 中实现。

10.4. 配置文件与自动配置期间识别和探测的顺序

ISA设备在内核配置文件中的描述如下:

```
device xxx0 at isa? port 0x300 irq 10 drq 5
     iomem 0xd0000 flags 0x1 sensitive
```

端口值、IRQ值和其他值被转换成与设备关联的资源值。根据设备 对自动配置需要和支持程度的不同，这些值是可选的。例如， 某些设备根本不需要读DRQ，而有些则允许设备从设备配置端口读取 IRQ设置。如果机器有多个ISA总线，可以在配置文件中明确指定哪条 总线，如isa0或isa1， 否则将在所有ISA总线上搜索设备。

##(sensitive)是一种资源请求，它指示 必须在所有非敏感设备之前探测设备。此特性虽被支持，但似乎从未 在目前的任何驱动程序中使用过。

对于老的ISA设备，很多情况下驱动程序仍然能够侦测配置参数。 但是系统中配置的每个设备必须具有一个配置行。如果系统中装有同一 类型的两个设备，但对应的驱动程序却只有一个配置行，例如：

```
device xxx0 at isa?
```

那么只有一个设备会被配置。

但对于支持通过PnP或专有协议进行自动识别的设备，一个配置行 就足够配置系统中的所有设备，如上面的配置行，或者简单地:

```
device xxx at isa?
```

如果设备驱动程序既支持能自动识别的设备又支持老设备，并且 两类设备同时安装在一台机器上，那么只要在配置文件中描述老设备 就足够了。自动识别的设备将被自动添加。

如果ISA设备是自动配置的，发生的事件如下:

所有设备驱动程序的识别例程（包括识别所有PnP设备的PnP识别 例程）以随机顺序被调用。他们识别出设备后就把设备添加到ISA总线 上的列表中。通常驱动程序的识别例程将新设备与它们的驱动 程序关联起来。而PnP识别例程并不知道其他驱动程序，因此不能将 驱动程序与它所添加的新设备关联起来。

使用PnP协议让PnP设备进入睡眠，以防止它们被探测为老设备。

被标记为##(sensitive)的非PnP设备的 探测例程被调用。如果探测设备成功，那么就为其调用挂接(attach) 例程。

所有非PnP设备的探测和连接例程以同样的方式被调用。

PnP设备从睡眠中恢复过来，并给它们分配所请求的资源：I/O、 内存地址范围、IRQ和DRQ，所有这些与已连接的老设备不会冲突。

对于每个PnP设备，所有ISA设备驱动程序的探测例程都会被调用。 第一个要求此设备的驱动程序将被连接。多个驱动程序以不同的优先权 要求一个设备的情况是可能的，这种情况下，具有最高优先权的驱动程序 将获胜。探测例程必须调用ISA_PNP_PROBE()将 真实的PnP ID和驱动程序支持的ID列表作比较，如果ID不在表中则返回 失败。这意味着每个驱动程序，包括不支持任何PnP设备的驱动程序， 都必须对未知的PnP设备无条件调用 ISA_PNP_PROBE()，对于未知设备， 至少要用一个 空的PnP ID表调用并返回失败。

探测例程遇到错误时会返回一个正值（错误码），成功时返回 零或负值。

负的返回值用于PnP设备支持多个接口的情况。例如，老的兼容接口 和新的高级接口通过不同的驱动程序来提供支持。两个驱动程序 都侦测设备。在探测例程中返回较高值的驱动程序优先（换句话说， 返回0的驱动程序具有最高的优先级，返回-1的其次，返回-2的更在 其后，如此下去）。如果多个驱动程序返回相同的值，那么最先调用的 获胜。因此，如果驱动程序返回0，就基本能够确信它获得优先权仲裁。

设备特定的识别例程也能够将一类而不是单个驱动程序指派给设备。 就象使用PnP的情况一样，对于某一设备，会探测这一类中所有的驱动程序。 由于这个特性在任何现存的驱动程序中总均未实现，故本文档中不再予以 考虑。

由于探测老设备的时候PnP设备被禁用，它们不会被连接两次 （一次作为老设备，一次作为PnP）。但如果识别例程设备相关的， 这种情况下设备驱动程序有责任确保同一设备不会被设备驱动程序 连接两次：一次作为老的由用户配置的，一次作为自动识别的。

对于自动识别的设备（包括PnP和设备特定的）的另一个实践结论是， 不能从内核配置文件中向它们传递旗标。因此它们必须要么根本不使用 旗标，要么为所有自动识别的设备使用单元号为0的设备的旗标，或者 使用sysctl接口而不是旗标。

通过使用函数族resource_query_*()和 resource_*_value()直接访问配置资源， 从而可以提供其他不常用的配置。它们的实现位于 kern/subr_bus.c。老的IDE磁盘驱动器i386/isa/wd.c包含这样使用的例子。 但必须优先使用配置的标准方法。将解析配置资源这类事情留给 总线配置代码。

10.5. 资源

用户写入到内核配置文件中的信息被作为配置资源处理，并传递到内核。 总线配置代码解析这部分信息并将其转换为结构device_t的值和与之 关联的总线资源。对于复杂情况下的配置，驱动程序可以直接使用 resource_* 函数访问配置资源。然而， 通常既不需要也不推荐这样做， 因此这儿不再进一步讨论这个问题。

总线资源与每个设备相关联。通过类型和类型中的数字标识它们。 对于ISA总线，定义了下面的类型：

- SYS_RES_IRQ - 中断号
- SYS_RES_DRQ - ISA DMA通道号
- SYS_RES_MEMORY - 映射到系统内存空间 的设备内存的范围
- SYS_RES_IOPORT - 设备I/O寄存器的范围

类型内的枚举从0开始，因此如果设备有两个内存区域，它的 SYS_RES_MEMORY 类型的资源编号为0和1。 资源类型与C语言的类型无关， 所有资源值具有C语言 unsigned long 类型，并且必要时必须进行类型强制转换 （cast）。资源号不必连续， 尽管对于ISA它们一般是连续的。ISA设备允许的资源编号为：

```
IRQ: 0-1
DRQ: 0-1
MEMORY: 0-3
IOPORT: 0-7
```

所有资源被表示为带有起始值和计数的范围。对于IRQ和DRQ资源， 计数一般等于1。内存的值引用物理地址。

对资源能够执行三种类型的动作：

- set/get
- allocate/release
- activate/deactivate

Set设置资源使用的范围。Allocation保留出请求的范围，使得 其它设备不能再占用（并检查此范围没有被其它设备占用）。 Activation执行必要的动作使得驱动程序可以访问资源（例如，对于 内存，它将被映射到内核的虚拟地址空间）。

操作资源的函数有：

- int bus_set_resource(device_t dev, int type, int rid, u_long start, u_long count)

  为资源设置范围。成功则返回0，否则返回错误码。 一般此函数只有在type，rid，start或 count之一的值超出了允许的范围才会 返回错误。

  - dev - 驱动程序的设备

- type - 资源类型，SYS_RES_*
- rid - 类型内部的资源号（ID）
- start，count - 资源范围

- int bus_get_resource(device_t dev, int type, int rid, u_long *startp, u_long *countp)

  取得资源范围。成功则返回0，如果资源尚未定义则返回错误码。

- u_long bus_get_resource_start(device_t dev, int type, int rid) u_long bus_get_resource_count (device_t dev, int type, int rid)

  便捷函数，只用来获取start或count。出错的情况下返回0， 因此如果0是资源的start合法值之一，将无法区分返回 的0是否指示错误。幸运的是，对于附加驱动程序，没有ISA资源的 start值从0开始。

- void bus_delete_resource(device_t dev, int type, int rid)

  删除资源，令其未定义。

- struct resource * bus_alloc_resource(device_t dev, int type, int *rid, u_long start, u_long end, u_long count, u_int flags)

  在start和end之间没有被其它设备占用的地方按count值 的范围分配一个资源。不过，不支持对齐。如果资源尚未被设置， 则自动创建它。start为0，end为~0（全1）的这对特殊值 意味着必须使用以前通过 bus_set_resource()设置的固定值： start和count就是它们自己，end=(start+count)，这种情况下， 如果以前资源没有定义，则返回错误。尽管rid通过引用传递， 但它并不被ISA总线的资源分配代码设置（其它总线可能使用不同的 方法并可能修改它）。

旗标是一个位映射，调用者感兴趣的有：

- RF_ACTIVE - 使得资源分配后 被自动激活。
- RF_SHAREABLE - 资源可以同时 被多个驱动程序共享。
- RF_TIMESHARE - 资源可以被多个驱动 程序分时共享，也就是说，被多个驱动程序同时分配，但任何 给定时间只能被其中一个激活。
- 出错返回0。被分配的值可以使用 rhand_*()从返回的句柄获得。
- int bus_release_resource(device_t dev, int type, int rid, struct resource *r)
- 释放资源，r为bus_alloc_resource() 返回的句柄。成功则返回0，否则返回错误码。

- int bus_activate_resource(device_t dev, int type, int rid, struct resource *r) int bus_deactivate_resource(device_t dev, int type, int rid, struct resource *r)
- 激活或禁用资源。成功则返回0，否则返回错误码。如果 资源被分时共享且当前被另一驱动程序激活，则返回 EBUSY。
- int bus_setup_intr(device_t dev, struct resource *r, int flags, driver_intr_t *handler, void *arg, void **cookiep) int bus_teardown_intr(device_t dev, struct resource *r, void *cookie)
- 关联/分离中断处理程序与设备。成功则返回0，否则 返回错误码。
- r - 被激活的描述IRQ的资源句柄。

  flags - 中断优先级，如下之一：

  - INTR_TYPE_TTY - 终端和其它 类似的字符类型设备。使用 spltty()屏蔽它们。
  - (INTR_TYPE_TTY | INTR_TYPE_FAST) - 输入缓冲较小的终端类型 设备，而且输入上的数据丢失很关键（例如老式串口）。 使用spltty()屏蔽它们。
  - INTR_TYPE_BIO - 块类型设备， 不包括CAM控制器上的。使用 splbio()屏蔽它们。
  - INTR_TYPE_CAM - CAM （通用访问 方法Common Access Method）总线控制器。使用 splcam()屏蔽它们。
  - INTR_TYPE_NET - 网络接口 控制器。使用 splimp()屏蔽它们。
  - INTR_TYPE_MISC - 各种其它设备。除了通过 splhigh()没有其它方法屏蔽它们。 splhigh()屏蔽所有中断。

当中断处理程序执行时，匹配其优先级的所有其它中断都被屏蔽， 唯一的例外是MISC级别，它不会屏蔽其它中断，也不会被其它中断 屏蔽。

- handler - 指向处理程序的指针， 类型driver_intr_t被定义为void driver_intr_t(void *)
- arg - 传递给处理程序的参量，标识 特定设备。由处理程序将它从void*转换为任何实际类型。ISA 中断处理程序的旧约定是使用单元号作为参量，新约定（推荐） 使用指向设备softc结构的指针。
- cookie[p] - 从 setup()接收的值，当传递给 teardown() 时用于标识处理程序。

定义了若干方法来操作资源句柄(struct resource *)。设备驱动 程序编写者感兴趣的有：

- u_long rman_get_start(r) u_long rman_get_end(r) 取得被分配的资源范围的起始和结束。
- void *rman_get_virtual(r) 取得 被激活的内存资源的虚地址。

10.6. 总线内存映射

很多情况下设备驱动程序和设备之间的数据交换是通过内存 进行的。有两种可能的变体:

(a) 内存位于设备卡上

(b) 内存为计算机的主内存

情况(a)中,驱动程序可能需要在卡上的内存与主存之间来回 拷贝数据。为了将卡上的内存映射到内核的虚地址空间,卡上内存的 物理地址和长度必须被定义为SYS_RES_MEMORY资源。 然后资源就可以被分配并激活,它的虚地址通过使用 rman_get_virtual()获取。较老的驱动程序 将函数pmap_mapdev()用于此目的,现在 不应当再直接使用此函数。它已成为资源激活的一个内部步骤。

大多数ISA卡的内存配置为物理地位于640KB-1MB范围之间的 某个位置。某些ISA卡需要更大的内存范围,位于16M以下的某个 位置(由于ISA总线上24位地址限制)。这种情况下,如果机器有 比设备内存的起始地址更多的内存(换句话说,它们重叠),则 必须在被设备使用的内存起始地址处配置一个内存空洞。许多BIOS允许在起始于14MB或15MB处配置1M的内存空洞。如果BIOS 正确地报告内存空洞,FreeBSD就能够正确处理它们(此特性 在老BIOS上可能会出问题)。

情况(b)中,只是数据的地址被发送到设备,设备使用DMA实际 访问主存中的数据。存在两个限制:首先,ISA卡只能访问16MB以下 的内存。其次,虚地址空间中连续的页面在物理地址空间中可能不 连续,设备可能不得不进行分散/收集操作。总线子系统为这些问题 提供现成现成的解决办法,剩下的必须由驱动程序自己完成。

DMA内存分配使用了两个结构, bus_dma_tag_t 和 bus_dmamap_t。 标签(tag)描述了DMA内存要求的特性。映射(map)表示按照这些 特性分配的内存块。多个映射可以与同一标签关联。

标签按照对特性的继承而被组织成树型层次结构。子标签继承父 标签的所有要求,可以令其更严格,但不允许放宽要求。

一般地,每个设备单元创建一个顶层标签(没有父标签)。如果 每个设备需要不同要求的内存区,则为每个内存区都会创建一个标签,这些 标签作为父标签的孩子。

使用标签创建映射的方法有两种。

其一，分配一大块符合标签要求的连续内存（以后可以被释放）。 这一般用于分配为了与设备通信而存在相对较长时间的那些内存区。 将这样的内存加载到映射中非常容易：它总是被看作位于适当物理 内存范围的一整块。

其二，将虚拟内存中的任意区域加载到映射中。这片内存的 每一页都被检查，看是否符合映射的要求。如何符合则留在原始位置。 如果不符合则分配一个新的符合要求的 “反弹页面(bounce page)”，用作中间存储。 当从不符合的原始页面写入数据时，数据首先被拷贝到反弹页面， 然后从反弹页面传递到设备。当读取时，数据将会从设备到反弹页面， 然后被拷贝到它们不符合的原始页面。原始和反弹页面之间的拷贝 处理被称作同步。这一般用于单次传输的基础之上：每次传输时 加载缓冲区，完成传输，卸载缓冲区。

工作在DMA内存上的函数有：

- int bus_dma_tag_create(bus_dma_tag_t parent, bus_size_t alignment, bus_size_t boundary, bus_addr_t lowaddr, bus_addr_t highaddr, bus_dma_filter_t *filter, void *filterarg, bus_size_t maxsize, int nsegments, bus_size_t maxsegsz, int flags, bus_dma_tag_t *dmat)

  创建新标签。成功则返回0，否则返回错误码。

  - parent – 父标签或者NULL， NULL用于创建顶层标签。
  - alignment – 对将要分配给标签的内存区的对齐要求。“no specific alignment”时值为1。仅应用于以后的 bus_dmamem_alloc()而不是 bus_dmamap_create()调用。
  - boundary – 物理地址边界， 分配内存时不能穿过。对于“no boundary” 使用0。仅应用于以后的 bus_dmamem_alloc()而不是 bus_dmamap_create()调用。必须为2的乘方。如果计划以非层叠DMA方式使用内存（也就是说， DMA地址由ISA DMA控制器提供而不是设备自身），则由于DMA硬件 限制，边界必须不能大于64KB（64*1024）。
  - lowaddr，highaddr – 名字稍微 有些误导。这些值用于限制可用于内存分配的物理地址的允许 范围。其确切含义根据以后不同的使用而有所不同。
    - 对于bus_dmamem_alloc()， 从0到lowaddr-1的所有地址被视为允许，更高的地址不允许 使用。
    - 对于bus_dmamap_create()， 闭区间[lowaddr; highaddr]之外的所有地址被视为可访问。 范围之内的地址页面被传递给过滤函数，由它决定是否可访问。 如果没有提供过滤函数，则整个区间被视为不可访问。
    - 对于ISA设备，正常值（没有过滤函数）为：

      lowaddr = BUS_SPACE_MAXADDR_24BIT

highaddr = BUS_SPACE_MAXADDR

- filter, filterarg - 过滤函数及其 参数。如果filter为NULL，则当调用bus_dmamap_create()时，整个区间 [lowaddr，highaddr]被视为不可访问。 否则，区间[lowaddr; highaddr]内的每个被试图访问的页面的 物理地址被传递给过滤函数，由它决定是否可访问。过滤函数的 原型为：int filterfunc(void *arg, bus_addr_t paddr)。当页面可以被访问时它必须 返回0，否则返回非零值。

- maxsize - 通过此标签可以分配的 最大内存值（以字节计）。有时这个值很难估算，或者可以任意大， 这种情况下，对于ISA设备这个值可以设为BUS_SPACE_MAXSIZE_24BIT。

- nsegments - 设备支持的分散/收集段 的最大数目。如果不加限制，则使用应当使用值 BUS_SPACE_UNRESTRICTED。 建议对父标签使用这个值，而为子孙标签指定实际限制。 nsegments值等于 BUS_SPACE_UNRESTRICTED 的标签不能用于实际加载映射，仅可以将它们作为父标签。 nsetments 的实际限制大约为250-300，再高的值将导致内核堆栈溢出（硬件 无法正常支持那么多的分散/收集缓冲区）。

- maxsegsz - 设备支持的分散/收集段 的最大尺寸。对于ISA设备的最大值为BUS_SPACE_MAXSIZE_24BIT。

- flags - 旗标的位图。感兴趣的旗标 只有：

  - BUS_DMA_ALLOCNOW - 创建标签时 请求分配所有可能用到的反射页面。

  - BUS_DMA_ISA - 比较神秘的一个标志， 仅用于Alpha机器。i386机器没有定义它。Alpha机器的所有ISA设备 都应当使用这个标志，但似乎还没有这样的驱动程序。

- dmat - 指向返回的新标签的存储的 指针。

- int bus_dma_tag_destroy(bus_dma_tag_t dmat)

  销毁标签。成功则返回0，否则返回错误码。

  dmat - 被销毁的标签。

- int bus_dmamem_alloc(bus_dma_tag_t dmat, void** vaddr, int flags, bus_dmamap_t *mapp)

  分配标签所描述的一块连续内存区。被分配的内存的大小为标签的 maxsize。成功则返回0，否则返回错误码。调用结果被用于获取内存的 物理地址，但在此之前必须用bus_dmamap_load() 将其加载。

  - dmat - 标签

- vaddr - 指向存储的指针，该存储空间 用于返回的分配区域的内核虚地址。
- flags - 旗标的位图。唯一感兴趣的旗标为：
  - BUS_DMA_NOWAIT - 如果内存不能 立即可用则返回错误。如果此标志没有设置，则允许例程 睡眠，直到内存可用为止。
- mapp - 指向返回的新映射的存储的指针。

- void bus_dmamem_free(bus_dma_tag_t dmat, void *vaddr, bus_dmamap_t map)

  释放由bus_dmamem_alloc()分配的内存。 目前，对分配的带有ISA限制的内存的释放没有实现。因此，建议的 使用模型为尽可能长时间地保持和重用分配的区域。不要轻易地 释放某些区域，然后再短时间地分配它。这并不意味着不应当使用 bus_dmamem_free()：希望很快它就会被 完整地实现。

  - dmat - 标签
  - vaddr - 内存的内核虚地址
  - map - 内存的映射（跟 bus_dmamem_alloc()返回的一样）

- int bus_dmamap_create(bus_dma_tag_t dmat, int flags, bus_dmamap_t *mapp)

  为标签创建映射，以后用于 bus_dmamap_load()。成功则返回0，否则 返回错误码。

  - dmat - 标签
  - flags - 理论上是旗标的位图。但还 从未定义过任何旗标，因此目前总是0。
  - mapp - 指向返回的新映射的存储的指针。

- int bus_dmamap_destroy(bus_dma_tag_t dmat, bus_dmamap_t map)

  销毁映射。成功则返回0，否则返回错误码。

  - dmat - 与映射关联的标签
  - map - 将要被销毁的映射

- int bus_dmamap_load(bus_dma_tag_t dmat, bus_dmamap_t map, void *buf, bus_size_t buflen, bus_dmamap_callback_t *callback, void *callback_arg, int flags)

  加载缓冲区到映射中(映射必须事先由 bus_dmamap_create()或者bus_dmamem_alloc())创建。缓冲区的所有 页面都会被检查，看是否符合标签的要求，并为那些不符合的分配 反弹页面。会创建物理段描述符的数组，并将其

传递给回调函数。 回调函数以某种方式处理这个数组。系统中的反弹缓冲区是受限的， 因此如果需要的反弹缓冲区不能立即获得，则将请求入队，当反弹 缓冲区可用时再调用回调函数。如果回调函数立即执行则返回0， 如果请求被排队，等待将来执行，则返回 EINPROGRESS。后一种情况下， 与排队的回调函数之间的同步由驱动程序负责。

- dmat - 标签
- map - 映射
- buf - 缓冲区的内核虚地址
- buflen - 缓冲区的长度
- callback，callback_arg - 回调函数及其参数

回调函数的原型为:

void callback(void *arg, bus_dma_segment_t *seg, int nseg, int error)

- arg - 与传递给 bus_dmamap_load()的callback_arg 相同。
- seg - 段描述符的数组
- nseg - 数组中的描述符个数
- error - 表示段数目溢出：如被设为 EFBIG， 则标签允许的最大数目的段无法容纳缓冲区。 这种情况下数组中的描述符的数目只有标签许可的那么多。 对这种情况的处理由驱动程序决定：根据希望的语义，驱动 程序可以视其为错误，或将缓冲区分为两个并单独处理第二个。

段数组中的每一项包含如下字段:

- ds_addr - 段物理地址
- ds_len - 段长度

- void bus_dmamap_unload(bus_dma_tag_t dmat, bus_dmamap_t map)

  unload the map.

  - dmat - 标签
  - map - 已加载的映射

- void bus_dmamap_sync (bus_dma_tag_t dmat, bus_dmamap_t map, bus_dmasync_op_t op)

与设备进行物理传输前后，将加载的缓冲区与其反弹页面进行同步。 此函数完成原始缓冲区与其映射版本之间所有必需的数据拷贝工作。 进行传输之前和之后必须对缓冲区进行同步。

- dmat - 标签
- map - 已加载的映射
- op - 要执行的同步操作的类型：
- BUS_DMASYNC_PREREAD - 从设备到 缓冲区的读操作之前
- BUS_DMASYNC_POSTREAD - 从设备到 缓冲区的读操作之后
- BUS_DMASYNC_PREWRITE - 从缓冲区到 设备的写操作之前
- BUS_DMASYNC_POSTWRITE - 从缓冲区到 设备的写操作之后

当前PREREAD和POSTWRITE为空操作，但将来可能会改变，因此驱动程序 中不能忽略它们。由bus_dmamem_alloc() 获得的内存不需要同步。

从bus_dmamap_load()中调用回调函数之前， 段数组是存储在栈中的。并且是按标签允许的最大数目的段预先分配 好的。这样由于i386体系结构上对段数目的实际限制约为250-300 （内核栈为4KB减去用户结构的大小，段数组条目的大小为8字节，和 其它必须留出来的空间）。由于数组基于最大数目而分配，因此这个值 必须不能设置成超出实际需要。幸运的是，对于大多数硬件而言， 所支持的段的最大数目低很多。但如果驱动程序想处理具有非常多 分散/收集段的缓冲区，则应当一部分一部分地处理：加载缓冲区的 一部分，传输到设备，然后加载缓冲区的下一部分，如此反复。

另一个实践结论是段数目可能限制缓冲区的大小。如果缓冲区中的 所有页面碰巧物理上不连续，则分片情况下支持的最大缓冲区尺寸 为(nsegments * page_size)。例如，如果支持的段的最大数目为10， 则在i386上可以确保支持的最大缓冲区大小为40K。如果希望更大的 则需要在驱动程序中使用一些特殊技巧。

如果硬件根本不支持分散/收集，或者驱动程序希望即使在严重分片的 情况下仍然支持某种缓冲区大小，则解决办法是：如果无法容纳下原始 缓冲区，就在驱动程序中分配一个连续的缓冲区作为中间存储。

下面是当使用映射时的典型调用顺序，根据对映射的具体使用而不同。 字符->用于显示时间流。

对于从连接到分离设备，这期间位置一直不变的缓冲区：

bus_dmamem_alloc -> bus_dmamap_load -> ...use buffer... -> -> bus_dmamap_unload -> bus_dmamem_free

对于从驱动程序外部传递进去，并且经常变化的缓冲区：

```
bus_dmamap_create -->
--> bus_dmamap_load --> bus_dmamap_sync(PRE...) --> do transfer -->
--> bus_dmamap_sync(POST...) --> bus_dmamap_unload -->
-...
--> bus_dmamap_load --> bus_dmamap_sync(PRE...) --> do transfer -->
--> bus_dmamap_sync(POST...) --> bus_dmamap_unload -->
--> bus_dmamap_destroy
```

当加载由bus_dmamem_alloc()创建的映射时，传递 进去的缓冲区的地址和大小必须和 bus_dmamem_alloc()中使用的一样。这种情况下就 可以保证整个缓冲区被作为一个段而映射（因而回调可以基于此假设）， 并且请求被立即执行（永远不会返回EINPROGRESS）。这种情况下回调函数 需要作的只是保存物理地址。

典型示例如下：

```
 static void
alloc_callback(void *arg, bus_dma_segment_t *seg, int nseg, int error)
{
 *(bus_addr_t *)arg = seg[0].ds_addr;
-}

 -...
 int error;
 struct somedata {
  -....
 -};
 struct somedata *vsomedata; -/* ### */
 bus_addr_t psomedata; -/* ######### */
 bus_dma_tag_t tag_somedata;
 bus_dmamap_t map_somedata;
 -...

 error=bus_dma_tag_create(parent_tag, alignment,
  boundary, lowaddr, highaddr, -/*filter*/ NULL, -/*filterarg*/ NULL,
  -/*maxsize*/ sizeof(struct somedata), -/*nsegments*/ 1,
  -/*maxsegsz*/ sizeof(struct somedata), -/*flags*/ 0,
  &tag_somedata);
 if(error)
 return error;

 error = bus_dmamem_alloc(tag_somedata, &vsomedata, -/* flags*/ 0,
   &map_somedata);
 if(error)
   return error;

 bus_dmamap_load(tag_somedata, map_somedata, (void *)vsomedata,
   sizeof (struct somedata), alloc_callback,
   (void *) &psomedata, -/*flags*/0);
```

代码看起来有点长，也比较复杂，但那是正确的使用方法。实际结果是：　如果分配多个内存区域，则总将它们组合成一个结构并作为整体分配　（如果对齐和边界限制允许的话）是一个很好的主意。

当加载任意缓冲区到由bus_dmamap_create()　创建的映射时，由于回调可能被延迟，因此必须采取特殊措施与回调 函数进行同步。代码看起来像下面的样子：

```
{
 int s;
 int error;

 s = splsoftvm();
 error = bus_dmamap_load(
     dmat,
     dmamap,
     buffer_ptr,
     buffer_len,
     callback,
     -/*callback_arg*/ buffer_descriptor,
     -/*flags*/0);
 if (error == EINPROGRESS) {
     -/*
      * #################
      * ############splx()#tsleep()######
      */
    -}
 splx(s);
-}
```

处理请求的两种方法分别是：

1． 如果通过显式地标记请求已经结束来完成请求（例如CAM请求），则　将所有进一步的处理放入回调驱动程序中会比较简单，回调结束后会　标记请求。之后不需要太多额外的同步。由于流控制的原因，冻结请求　队列直到请求完成才释放可能是个好主意。

2． 如果请求是在函数返回时完成（例如字符设备上传统的读写请求），　则需要在缓冲区描述符上设置同步标志，并调用　tsleep()。后面当回调函数被调用时，它将 执行处理并检查同步标志。如果设置了同步标志，它应该发出一个 唤醒操作。在这种方法中，回调函数或者进行所由必需的处理（就像　前面的情况），或者简单在缓冲区描述符中存储段数组。回调完成　后，回调函数就能使用这个存储的段数组并进行所有的处理。

10.7. DMA

ISA总线中Direct Memory Access (DMA)是通过DMA控制器（实际上是它们 中的两个，但这只是无关细节）实现的。为了使以前的ISA设备简单便宜，　总线控制和地

址产生的逻辑都集中在DMA控制器中。幸运的是，FreeBSD 提供了一套函数，这些函数大多把DMA控制器的繁琐细节对设备驱动程序 隐藏了起来。

最简单情况是那些比较智能的设备。就象PCI上的总线主设备一样， 它们自己能产生总线周期和内存地址。它们真正从DMA控制器需要的 唯一事情是总线仲裁。所以为了此目的，它们假装是级联从DMA控制器。 当连接驱动程序时，系统DMA控制器需要做的唯一事情就是通过调用 如下函数在一个DMA通道上激活级联模式。

void isa_dmacascade(int channel_number)

所有进一步的活动通过对设备编程完成。当卸载驱动程序时，不需要 调用DMA相关的函数。

对于较简单的设备，事情反而变得复杂。使用的函数包括：

- int isa_dma_acquire(int chanel_number)

  保留一个DMA通道。成功则返回0，如果通道已经被保留或被其它 驱动程序保留则返回EBUSY。大多数的ISA设备都不能共享DMA通道， 因此这个函数通常在连接设备时调用。总线资源的现代接口使得 这种保留成为多余，但目前仍必须使用。如果不使用，则后面其它 DMA例程将会panic。

- int isa_dma_release(int chanel_number)

  释放先前保留的DMA通道。释放通道时必须不能有正在进行中的 传输（另外，释放通道后设备必须不能再试图发起传输）。

- void isa_dmainit(int chan, u_int bouncebufsize)

  分配由特定通道使用的反弹缓冲区。请求的缓冲区大小不能超过 64KB。以后，如果传输缓冲区碰巧不是物理连续的，或超出ISA 总线可访问的内存范围，或跨越64KB的边界，则会自动使用反弹 缓冲区。如果传输总是使用符合上述条件的缓冲区（例如，由 bus_dmamem_alloc()分配的那些），则 不需要调用isa_dmainit()。但使用此函数 会让通过DMA控制器传输任意数据变得非常方便。

  - chan - 通道号

  - bouncebufsize - 以字节计数的反弹 缓冲区的大小

- void isa_dmastart(int flags, caddr_t addr, u_int nbytes, int chan)

  准备启动DMA传输。实际启动设备上的传输之前必需调用此函数 来设置DMA控制器。它检查缓冲区是否连续的且在ISA内存范围 之内，如果不是则自动使用反弹缓冲区。如果需要反弹缓冲区， 但反弹缓冲区没有用isa_dmainit() 设置，或对于请求的传输大小来说太小，则系统将panic。 写请求且使用反弹缓冲区的情况下，数据将被自动拷贝到反弹 缓冲区。

- flags - 位掩码，决定将要完成的操作的类型。方向位B_READ和 B_WRITE互斥。
  - B_READ - 从ISA总线读到内存
  - B_WRITE - 从内存写到ISA总线上
  - B_RAW - 如果设置则DMA控制器将会记住缓冲区，并在传输结束后 自动重新初始化它自己，再次重复传输同一缓冲区（当然，驱动 程序可能发起设备的另一个传输之前改变缓冲区中的数据）。 如果没有设置，参数只对一次传输有效，在发起下一次传输之前 必须再次调用isa_dmastart()。只有在不 使用反弹缓冲区时使用B_RAW才有意义。
- addr - 缓冲区的虚地址
- nbytes - 缓冲区长度。必须小于等于64KB。不允许长度为0：因为 DMA控制器将会理解为64KB，而内核代码把它理解为0，那样就会导致 不可预测的效果。对于通道号等于和高于4的情况，长度必需为偶数， 因为这些通道每次传输2字节。奇数长度情况下，最后一个字节不被 传输。
- chan - 通道号
- void isa_dmadone(int flags, caddr_t addr, int nbytes, int chan)

  设备报告传输完成后，同步内存。如果是使用反弹缓冲区的读操作， 则将数据从反弹缓冲区拷贝到原始缓冲区。参量与 isa_dmastart()的相同。允许使用B_RAW标志， 但它一点也不会影响isa_dmadone()。
- int isa_dmastatus(int channel_number)

  返回当前传输中剩余的字节数。在 isa_dmastart()中设置了B_READ的情况下， 返回的数字一定不会等于零。传输结束时它会被自动复位到缓冲区的 长度。正式的用法是在设备发信号指示传输已完成时检查剩余的字节数。 如果字节数不为0，则此次传输可能有问题。
- int isa_dmastop(int channel_number)

  放弃当前的传输并返回剩余未传输的字节数。

10.8. xxx_isa_probe

这个函数探测设备是否存在。如果驱动程序支持自动侦测设备配置的 某些部分（如中断向量或内存地址），则自动侦测必须在此例程中完成。

对于任意其他总线，如果不能侦测到设备，或者侦测到但自检失败， 或者发生某些其他问题，则应当返回一个正值的错误。如果设备不 存在则必须返回值

ENXIO。 其他错误值可能表示其他条件。零或负值 意味着成功。大多数驱动程序返回零表示成功。

当PnP设备支持多个接口时使用负返回值。例如，不同驱动程序支持 老的兼容接口和较新的高级接口。则两个驱动程序都将侦测设备。 在探测例程中返回较高值的驱动程序获得优先（换句话说，返回0的 驱动程序具有最高的优先级，返回-1的其次，返回-2的更后，等等）。 这样，仅支持老接口的设备将被老驱动程序处理（其应当从探测例程中 返回-1），而同时也支持新接口的设备将由新驱动程序处理（其应当从 探测例程中返回0）。

设备描述符结构xxx_softc由系统在调用探测例程之前分配。如果 探测例程返回错误，描述符会被系统自动取消分配。因此如果出现 探测错误，驱动程序必须保证取消分配探测期间它使用的所有资源， 且确保没有什么能够阻止描述符被安全地取消分配。如果探测成功 完成，描述符将由系统保存并在以后传递给例程xxx_isa_attach()。如果驱动程序返回负值， 就不能保证它将获得最高优先权且其连接例程会被调用。因此这种 情况下它也必须在返回前释放所有的资源，并在需要的时候在连接 例程中重新分配它们。当xxx_isa_probe() 返回0时，在返回前释放资源也是一个好主意，而且中规中矩的驱动 程序应当这样做。但在释放资源会存在某些问题的情况下，允许驱动 程序在从探测例程返回0和连接例程的执行之间保持资源。

典型的探测例程以取得设备描述符和单元号开始：

```
struct xxx_softc *sc = device_get_softc(dev);
 int unit = device_get_unit(dev);
 int pnperror;
 int error = 0;

 sc->dev = dev; -/* #### */
 sc->unit = unit;
```

然后检查PnP设备。检查是通过一个包含PnP ID列表的表进行的。此表 包含这个驱动程序支持的PnP ID和以人工可读形式给出的对应这些ID的 设备型号的描述。

```
pnperror=ISA_PNP_PROBE(device_get_parent(dev), dev,
xxx_pnp_ids); if(pnperror == ENXIO) return ENXIO;
```

ISA_PNP_PROBE的逻辑如下：如果卡（设备单元）没有被作为PnP侦测到， 则返回ENOENT。如果被作为PnP侦测到，但侦测到的ID不匹配表中的 任一ID，则返回ENXIO。最后，如果设备能支持PnP且匹配表中的一个 ID，则返回0，并且由device_set_desc()从 表中取得适当的描述进行设置。

如果设备驱动程序仅支持PnP设备，则情况看起来如下：

```
if(pnperror -!= 0)
   return pnperror;
```

对于不支持PnP的驱动程序不需要特殊处理，因为驱动程序会传递空的 PnP ID表，且如果在PnP卡上调用会得到ENXIO。

探测例程通常至少需要某些最少量的资源，如I/O端口号，来发现并探测卡。 对于不同的硬件，驱动程序可能会自动发现其他必需的资源。PnP设备的 所有资源由PnP子系统预先设置，因此驱动程序不需要自己发现它们。

通常访问设备所需要的最少信息就是端口号。然后某些设备允许从设备 配置寄存器中取得其余信息（尽管不是所有的设备都这样）。因此首先 我们尝试取得端口起始值：

```
sc->port0 = bus_get_resource_start(dev,
    SYS_RES_IOPORT, 0 -/*rid*/); if(sc->port0 == 0) return ENXIO;
```

基端口地址被保存在softc结构中，以便将来使用。如果需要经常使用 端口，则每次都调用资源函数将会慢的无法忍受。如果我们没有得到 端口，则返回错误即可。相反，一些设备驱动程序相当聪明，尝试探测 所有可能的端口，如下：

```
-/* #########I/O##### */
static struct xxx_allports {
   u_short port; -/* #### */
   short used; -/* ################ */
-} xxx_allports = {
   { 0x300, 0 -},
   { 0x320, 0 -},
   { 0x340, 0 -},
   { 0, 0 -} -/* ### */
-};

-...
int port, i;
-...

port =  bus_get_resource_start(dev, SYS_RES_IOPORT, 0 -/*rid*/);
if(port -!=0 -) {
   for(i=0; xxx_allports[i].port!=0; i++) {
      if(xxx_allports[i].used -|| xxx_allports[i].port -!= port)
         continue;

      -/* ### */
      xxx_allports[i].used = 1;
      -/* ######## */
      return xxx_really_probe(dev, port);
   -}
   return ENXIO; -/* ############ */
-}

-/* ################# */
for(i=0; xxx_allports[i].port!=0; i++) {
```

```
        if(xxx_allports[i].used)
            continue;

        -/* ####### -- ###############
         * ############
         */
         xxx_allports[i].used = 1;

        error = xxx_really_probe(dev, xxx_allports[i].port);
        if(error == 0) -/* ########### */
            return 0;
    -}
    -/* ################# */
    return ENXIO;
```

当然，做这些事情通常应该使用驱动程序的 identify()例程。但可能有一个正当的理由来 说明为什么在函数probe()中完成更好：如果 这种探测会让一些其他敏感设备发疯。探测例程按旗标 sensitive排序：敏感设备首先被探测，然后是 其他设备。但identify()例程在所有探测之前 被调用，因此它们不会考虑敏感设备并可能扰乱这些设备。

现在，我们得到起始端口以后就需要设置端口数（PnP设备除外），因为 内核在配置文件中没有这个信息。

```
    if(pnperror -/* ###PnP## */
    && bus_set_resource(dev, SYS_RES_IOPORT, 0, sc->port0,
    XXX_PORT_COUNT)<0)
        return ENXIO;
```

最后分配并激活一片端口地址空间（特殊值start和end意思是说 “使用我们通过bus_set_resource() 设置的那些值”）：

```
    sc->port0_rid = 0;
    sc->port0_r = bus_alloc_resource(dev, SYS_RES_IOPORT,
    &sc->port0_rid,
        -/*start*/ 0, -/*end*/ ~0, -/*count*/ 0, RF_ACTIVE);

    if(sc->port0_r == NULL)
        return ENXIO;
```

现在可以访问端口映射的寄存器后，我们就可以以某种方式向设备写入 数据并检查设备是否如我们期望的那样作出反应。如果没有，则说明 可能其他的设备在这个地址上，或者这个地址上根本没有设备。

通常驱动程序直到连接例程才会设置中断处理函数。这之前我们替代以 轮询模式进行探测，超时则以DELAY()实现。 探测例程必须确保不能永久挂起，设备上的所有等待必须在超时内完成。 如果设备不在这段时间内响应，则可能设备出故障或配置错误，驱动程序 必须返回错误，当确定超时间隔时，给设备一些额外时间

以确保可靠： 尽管假定DELAY()在任何机器上都延时相同数量的 时间，但随具体CPU的不同，此函数还是有一定的误差幅度。

如果探测例程真的想检查中断是否真的工作，它可以也配置和探测中断。 但不建议这样。

```
-/* ############### */
if(error = xxx_probe_ports(sc))
    goto bad; -/* ####### */
```

依赖于所发现设备的确切型号，函数 xxx_probe_ports()也可能设置设备描述。但 如果只支持一种设备型号，则也可以硬编码的形式完成。当然，对于 PnP设备，PnP支持从表中自动设置描述。

```
if(pnperror)
    device_set_desc(dev, -"Our device model 1234");
```

探测例程应当或者通过读取设备配置寄存器来发现所有资源的范围， 或者确保由用户显式设置。我们将假定一个带板上内存的例子。 探测例程应当尽可能是非插入式的，这样分配和检查其余资源功能性 的工作就可以更好地留给连接例程来做。

内存地址可以在内核配置文件中指定，或者对应某些设备可以在非易失性 配置寄存器中预先配置。如果两种做法均可用却不同，那么应当用 哪个呢？可能用户厌烦在内核配置文件中明确设置地址，但他们知道 自己在干什么，则应当优先使用这个。一个实现的例子可能是这样的：

```
-/* ########## */
sc->mem0_p = bus_get_resource_start(dev, SYS_RES_MEMORY, 0 -/*rid*/);
if(sc->mem0_p == 0) { -/* ######## */
    sc->mem0_p = xxx_read_mem0_from_device_config(sc);


if(sc->mem0_p == 0)
        -/* ############## */
        goto bad;
-} else {
    if(xxx_set_mem0_address_on_device(sc) < 0)
        goto bad; -/* ######## */
-}

-/* ############
 * ################
 * #########################
 * ################“msize”#####
 * #ISA######
 */
```

```
        if(pnperror) { -/*###PnP## */
          sc->mem0_size = bus_get_resource_count(dev, SYS_RES_MEMORY, 0 -/*rid*/);
          if(sc->mem0_size == 0) -/* ###### */
              sc->mem0_size = xxx_read_mem0_size_from_device_config(sc);

          if(sc->mem0_size == 0) {
              -/* ########################
               * #####################
               * ####################
               */
              sc->mem0_size = 8*1024;
          -}

          if(xxx_set_mem0_size_on_device(sc) < 0)
              goto bad; -/*######### */

          if(bus_set_resource(dev, SYS_RES_MEMORY, -/*rid*/0,
                  sc->mem0_p, sc->mem0_size)<0)
              goto bad;
      -} else {
          sc->mem0_size = bus_get_resource_count(dev, SYS_RES_MEMORY, 0 -/*rid*/);
      -}
```

类似，很容易检查IRQ和DRQ所用的资源。

如果一切进行正常，然后就可以释放所有资源并返回成功。

```
      xxx_free_resources(sc);
      return 0;
```

最后，处理棘手情况。所有资源应当在返回前被释放。我们利用这样一个　事实：softc结构在传递给我们以前被零化，因此我们能够找出是否分配了 某些资源：如果分配则这些资源的描述符非零。

```
      bad:

      xxx_free_resources(sc);
      if(error)
          return error;
      else -/* ###### */
        return ENXIO;
```

这是完整的探测例程。资源的释放从多个地方完成，因此将它挪到一个　函数中，看起来可能像下面的样子：

```
static void
        xxx_free_resources(sc)
          struct xxx_softc *sc;
      {
          -/* ##########0### */

          -/* ###### */
          if(sc->intr_r) {
```

```
        bus_teardown_intr(sc->dev, sc->intr_r, sc->intr_cookie);
        bus_release_resource(sc->dev, SYS_RES_IRQ, sc->intr_rid,
            sc->intr_r);
        sc->intr_r = 0;
    -}

    -/* ############# */
    if(sc->data_p) {
        bus_dmamap_unload(sc->data_tag, sc->data_map);
        sc->data_p = 0;
    -}
     if(sc->data) { -/* sc->data_map##0##### */
        -/* the map will also be freed */
        bus_dmamem_free(sc->data_tag, sc->data, sc->data_map);
        sc->data = 0;
    -}
    if(sc->data_tag) {
        bus_dma_tag_destroy(sc->data_tag);
        sc->data_tag = 0;
    -}

    -... ############## -...

    if(sc->parent_tag) {
        bus_dma_tag_destroy(sc->parent_tag);
        sc->parent_tag = 0;
    -}

    -/* ######## */
    if(sc->mem0_r) {
        bus_release_resource(sc->dev, SYS_RES_MEMORY, sc->mem0_rid,
            sc->mem0_r);
        sc->mem0_r = 0;
    -}
    -...
    if(sc->port0_r) {
        bus_release_resource(sc->dev, SYS_RES_IOPORT, sc->port0_rid,
            sc->port0_r);
        sc->port0_r = 0;
    -}
-}
```

10.9. xxx_isa_attach

如果探测例程返回成功并且系统选择连接那个驱动程序，则连接例程 负责将驱动程序实际连接到系统。如果探测例程返回0 ，则连接例程期望 接收完整的设备结构softc，此结构由探测例程设置。同时，如果探测例程 返回0，它可能期望这个设备的连接例程应当在将来的某点被调用。如果 探测例程返回负值，则驱动程序可能不会作此假设。

如果成功完成，连接例程返回0，否则返回错误码。

连接例程的启动跟探测例程相似，将一些常用数据取到一些更容易 访问的变量中。

```
struct xxx_softc *sc = device_get_softc(dev);
int unit = device_get_unit(dev);
int error = 0;
```

然后分配并激活所需资源。由于端口范围通常在从探测返回前就 被释放，因此需要重新分配。我们希望探测例程已经适当地设置了 所有的资源范围，并将它们保存在结构softc中。如果探测例程留下了 一些被分配的资源，就不需要再次分配（重新分配被视为错误）。

```
sc->port0_rid = 0;
sc->port0_r = bus_alloc_resource(dev, SYS_RES_IOPORT, &sc->port0_rid,
   -/*start*/ 0, -/*end*/ ~0, -/*count*/ 0, RF_ACTIVE);

if(sc->port0_r == NULL)
    return ENXIO;

-/* #### */
sc->mem0_rid = 0;
sc->mem0_r = bus_alloc_resource(dev, SYS_RES_MEMORY, &sc->mem0_rid,
   -/*start*/ 0, -/*end*/ ~0, -/*count*/ 0, RF_ACTIVE);

if(sc->mem0_r == NULL)
     goto bad;

-/* ##### */
sc->mem0_v = rman_get_virtual(sc->mem0_r);
```

DMA请求通道(DRQ)以相似方式被分配。使用 isa_dma*()函数族进行初始化。例如：

isa_dmacascade(sc->drq0);

中断请求线(IRQ)有点特殊。除了分配以外，驱动程序的中断处理 函数也应当与它关联。在古老的ISA驱动程序中，由系统传递给中断处理 函数的参量是设备单元号。但在现代驱动程序中，按照约定，建议传递 指向结构softc的指针。一个很重要的原因在于当结构softc被动态分配后， 从softc取得单元号很容易，而从单元号取得softc很困难。同时，这个 约定也使得用于不同总线的应用程序看起来统一，并允许它们共享代码： 每个总线有其自己的探测，连接，分离和其他总线相关的例程，而它们 之间可以共享大块的驱动程序代码。

```
sc->intr_rid = 0;
sc->intr_r = bus_alloc_resource(dev, SYS_RES_MEMORY, &sc->intr_rid,
     -/*start*/ 0, -/*end*/ ~0, -/*count*/ 0, RF_ACTIVE);

if(sc->intr_r == NULL)
   goto bad;
```

```
-/*
 * ###XXX_INTR_TYPE##############
 * ##INTR_TYPE_CAM##CAM#####
 */
error = bus_setup_intr(dev, sc->intr_r, XXX_INTR_TYPE,
    (driver_intr_t *) xxx_intr, (void *) sc, &sc->intr_cookie);
if(error)
    goto bad;
```

如果驱动程序需要与内存进行DMA，则这块内存应当按前述方式分配:

```
error=bus_dma_tag_create(NULL, -/*alignment*/ 4,
    -/*boundary*/ 0, -/*lowaddr*/ BUS_SPACE_MAXADDR_24BIT,
    -/*highaddr*/ BUS_SPACE_MAXADDR, -/*filter*/ NULL, -/*filterarg*/ NULL,
    -/*maxsize*/ BUS_SPACE_MAXSIZE_24BIT,
    -/*nsegments*/ BUS_SPACE_UNRESTRICTED,
    -/*maxsegsz*/ BUS_SPACE_MAXSIZE_24BIT, -/*flags*/ 0,
    &sc->parent_tag);
if(error)
    goto bad;

-/* #############
 * ##sc->data########################
 * struct {
 *   u_short rd_pos;
 *   u_short wr_pos;
 *   char    bf[XXX_RING_BUFFER_SIZE]
 * -} *data;
 */
error=bus_dma_tag_create(sc->parent_tag, 1,
    0, BUS_SPACE_MAXADDR, 0, -/*filter*/ NULL, -/*filterarg*/ NULL,
    -/*maxsize*/ sizeof(* sc->data), -/*nsegments*/ 1,
    -/*maxsegsz*/ sizeof(* sc->data), -/*flags*/ 0,
    &sc->data_tag);
if(error)
    goto bad;

error = bus_dmamem_alloc(sc->data_tag, &sc->data, -/* flags*/ 0,
    &sc->data_map);
if(error)
     goto bad;

-/* #&sc->data_p#####xxx_alloc_callback()#######
 * #################
 * ###################
 * #########
 *
 * static void
 * xxx_alloc_callback(void *arg, bus_dma_segment_t *seg,
 *     int nseg, int error)
```

```
 * {
 *     *(bus_addr_t *)arg = seg[0].ds_addr;
 * -}
 */
bus_dmamap_load(sc->data_tag, sc->data_map, (void *)sc->data,
    sizeof (* sc->data), xxx_alloc_callback, (void *) &sc->data_p,
    -/*flags*/0);
```

分配了所有的资源后，设备应当被初始化。初始化可能包括测试 所有特性，确保它们起作用。

```
if(xxx_initialize(sc) < 0)
    goto bad;
```

总线子系统将自动在控制台上打印由探测例程设置的设备描述。但 如果驱动程序想打印一些关于设备的额外信息，也是可能的，例如：

```
device_printf(dev, -"has on-card FIFO buffer of %d bytes\n", sc->fifosize);
```

如果初始化例程遇到任何问题，建议返回错误之前打印有关信息。

连接例程的最后一步是将设备连接到内核中的功能子系统。完成 这个步骤的精确方式依赖于驱动程序的类型：字符设备、块设备、网络 设备、CAM SCSI总线设备等等。

如果所有均工作正常则返回成功。

```
error = xxx_attach_subsystem(sc);
if(error)
    goto bad;

return 0;
```

最后，处理棘手情况。返回错误前，所有资源应当被取消分配。 我们利用这样一个事实：结构softc传递给我们之前被零化，因此我们 能找出是否分配了某些资源：如果分配则它们的描述符非零。

```
bad:

xxx_free_resources(sc);
if(error)
    return error;
else -/* exact error is unknown */
    return ENXIO;
```

这就是连接例程的全部。

10.10. xxx_isa_detach

如果驱动程序中存在这个函数，且驱动程序被编译为可加载模块，则 驱动程序具有被卸载的能力。如果硬件支持热插拔，这是一个很重要的 特性。但ISA总线不支持热插拔，因此这个特性对于ISA设备不是特别 重要。卸载驱动程序的能力可能在调试时有用，但很多情况下只有在 老版本的驱动程序莫名其妙地卡住系统的情况下才需要安装新版本的 驱动程序，并且无论如何都需要重启，这样使得花费精力写分离例程 有些不值得。另一个宣称卸载允许在用于生产的机器上升级驱动程序的 论点看起来似乎更多的只是理论而已。升级驱动程序是一项危险的操作， 决不不应当在用于生产的机器上实行（并且当系统运行于安全模式时这 也是不被允许的）。然而，出于完整性考虑，还是会提供分离例程。

如果驱动程序成功分离，分离例程返回0，否则返回错误码。

分离逻辑是连接的镜像。要做的第一件事情就是将驱动程序从内核 子系统分离。如果设备当前正打开着，驱动程序有两个选择：拒绝分离 或者强制关闭并继续进行分离。选用哪种方式取决于特定内核子系统 执行强制关闭的能力和驱动程序作者的偏好。通常强制关闭似乎是 更好的选择。

```
struct xxx_softc *sc = device_get_softc(dev);
int error;

error = xxx_detach_subsystem(sc);
if(error)
   return error;
```

下一步，驱动程序可能希望复位硬件到某种一致的状态。包括停止任何 将要进行的传输，禁用DMA通道和中断以避免设备破坏内存。对于大多数 驱动程序而言，这正是关闭例程所做的，因此如果驱动程序中包括关闭 例程，我们只要调用它就可以了。

xxx_isa_shutdown(dev);

最后释放所有资源并返回成功。

```
xxx_free_resources(sc);
return 0;
```

10.11. xxx_isa_shutdown

当系统要关闭的时候调用此例程。通过它使硬件进入某种一致的状态。 对于大多数ISA设备而言不需要特殊动作，因此这个函数并非真正必需， 因为不管怎样重启动时设备会被重新初始化。但有些设备必须按特定 步骤关闭，以确保在软重启后能被正确地检测到（对于很多使用私有 识别协议的设备特别有用）。很多情况下，在设备寄存器中禁用DMA和 中断，并停止将要进行的传输是个好主意。确切动作取决于硬件，因此 我们无法在此详细讨论。

10.12. xxx_intr

当收到来自特定设备的中断时就会调用中断处理函数。ISA总线不支持 中断共享（某些特殊情况例外），因此实际上如果中断处理函数被调用， 几乎可以确信中断是来自其设备。然而，中断处理函数必须轮询设备 寄存器并确保中断是由它的设备产生的。如果不是，中断处理函数应当 返回。

ISA驱动程序的旧约定是取设备单元号作为参量。现在已经废弃，当 调用bus_setup_intr()时新驱动程序接收任何 在连接例程中为他们指定的参量。根据新约定，它应当是指向结构 softc的指针。因此中断处理函数通常像下面那样开始：

```
static void
xxx_intr(struct xxx_softc *sc)
{
```

它运行在由bus_setup_intr()的中断类型参数指定 的中断优先级上。这意味着禁用所有其他同类型的中断和所有软件中断。

为了避免竞争，中断处理例程通写成循环形式：

```
while(xxx_interrupt_pending(sc)) {
   xxx_process_interrupt(sc);
   xxx_acknowledge_interrupt(sc);
-}
```

中断处理函数必须只向设备应答中断，但不能向中断控制器应答，后者由 系统负责处理。

第 11 章 PCI设备

翻译：.

本章将讨论FreeBSD为了给PCI总线上的设备编写驱动程序而提供的机制。

11.1. 探测与连接

这儿的信息是关于PCI总线代码如何迭代通过未连接的设备，并查看新 加载的kld是否会连接其中一个。

11.1.1. 示例驱动程序源代码(**mypci.c**)

```
/*
 * #PCI#########KLD
 *
 * Murray Stokely
 */

#include <sys/param.h>  /* kernel.h###### */
#include <sys/module.h>
#include <sys/systm.h>
#include <sys/errno.h>
#include <sys/kernel.h>  /* ########### */
#include <sys/conf.h>  /* cdevsw## */
#include <sys/uio.h>  /* uio## */
#include <sys/malloc.h>
#include <sys/bus.h>  /* pci########## */

#include <machine/bus.h>
#include <sys/rman.h>
#include <machine/resource.h>

#include <dev/pci/pcivar.h> /* ####get_pci#! */
#include <dev/pci/pcireg.h>

/* softc############ */
struct mypci_softc {
 device_t my_dev;
 struct cdev *my_cdev;
};

/* #### */
static d_open_t  mypci_open;
static d_close_t mypci_close;
static d_read_t  mypci_read;
static d_write_t mypci_write;

/* ####### */
```

```
static struct cdevsw mypci_cdevsw = {
 .d_version = D_VERSION,
 .d_open = mypci_open,
 .d_close = mypci_close,
 .d_read = mypci_read,
 .d_write = mypci_write,
 .d_name = "mypci",
};

/*
 * #cdevsw###########cdev####si_drv1#####softc#
 * #####/dev###############
 * #############softc#
 */

int
mypci_open(struct cdev *dev, int oflags, int devtype, d_thread_t *td)
{
 struct mypci_softc *sc;

 /* Look up our softc. */
 sc = dev->si_drv1;
 device_printf(sc->my_dev, -"Opened successfully.\n");
 return (0);
}

int
mypci_close(struct cdev *dev, int fflag, int devtype, d_thread_t *td)
{
 struct mypci_softc *sc;

 /* Look up our softc. */
 sc = dev->si_drv1;
 device_printf(sc->my_dev, -"Closed.\n");
 return (0);
}

int
mypci_read(struct cdev *dev, struct uio *uio, int ioflag)
{
 struct mypci_softc *sc;

 /* Look up our softc. */
 sc = dev->si_drv1;
 device_printf(sc->my_dev, -"Asked to read %d bytes.\n", uio->uio_resid);
 return (0);
}

int
mypci_write(struct cdev *dev, struct uio *uio, int ioflag)
{
 struct mypci_softc *sc;
```

```
 /* Look up our softc. */
 sc = dev->si_drv1;
 device_printf(sc->my_dev, -"Asked to write %d bytes.\n", uio->uio_resid);
 return (0);
}

/* PCI#### */

/*
 * ########################
 * #################
 */
static int
mypci_probe(device_t dev)
{

 device_printf(dev, -"MyPCI Probe\nVendor ID -: 0x%x\nDevice ID -: 0x%x\n",
     pci_get_vendor(dev), pci_get_device(dev));

 if (pci_get_vendor(dev) == 0x11c1) {
  printf("We've got the Winmodem, probe successful!\n");
  device_set_desc(dev, -"WinModem");
  return (BUS_PROBE_DEFAULT);
 }
 return (ENXIO);
}

/* ############### */

static int
mypci_attach(device_t dev)
{
 struct mypci_softc *sc;

 printf("MyPCI Attach for -: deviceID -: 0x%x\n", pci_get_devid(dev));

 /* Look up our softc and initialize its fields. */
 sc = device_get_softc(dev);
 sc->my_dev = dev;

 /*
  * Create a -/dev entry for this device.  The kernel will assign us
  * a major number automatically.  We use the unit number of this
  * device as the minor number and name the character device
  * -"mypci<unit>".
  */
 sc->my_cdev = make_dev(&mypci_cdevsw, device_get_unit(dev),
     UID_ROOT, GID_WHEEL, 0600, -"mypci%u", device_get_unit(dev));
 sc->my_cdev->si_drv1 = sc;
 printf("Mypci device loaded.\n");
 return (0);
}
```

```
/* ##### */

static int
mypci_detach(device_t dev)
{
 struct mypci_softc *sc;

 /* Teardown the state in our softc created in our attach routine. */
 sc = device_get_softc(dev);
 destroy_dev(sc->my_cdev);
 printf("Mypci detach!\n");
 return (0);
}

/* #######sync##### */

static int
mypci_shutdown(device_t dev)
{

 printf("Mypci shutdown!\n");
 return (0);
}

/*
 * #######
 */
static int
mypci_suspend(device_t dev)
{

 printf("Mypci suspend!\n");
 return (0);
}

/*
 * #############
 */
static int
mypci_resume(device_t dev)
{

 printf("Mypci resume!\n");
 return (0);
}

static device_method_t mypci_methods[] = {
 /* #### */
 DEVMETHOD(device_probe,  mypci_probe),
 DEVMETHOD(device_attach, mypci_attach),
 DEVMETHOD(device_detach, mypci_detach),
 DEVMETHOD(device_shutdown, mypci_shutdown),
 DEVMETHOD(device_suspend, mypci_suspend),
```

```
 DEVMETHOD(device_resume, mypci_resume),

 { 0, 0 -}
};

static devclass_t mypci_devclass;

DEFINE_CLASS_0(mypci, mypci_driver, mypci_methods, sizeof(struct mypci_softc));
DRIVER_MODULE(mypci, pci, mypci_driver, mypci_devclass, 0, 0);
```

11.1.2. 示例驱动程序的Makefile

```
# ####mypci#Makefile

KMOD= mypci
SRCS= mypci.c
SRCS+= device_if.h bus_if.h pci_if.h

.include <bsd.kmod.mk>
```

如果你将上面的源文件和 Makefile放入一个目录，你可以运行 make编译示例驱动程序。 还有，你可以运行make load 将驱动程序装载到当前正在运行的内核中，而make unload可在装载后卸载驱动程序。

11.1.3. 更多资源

- PCI Special Interest Group
- PCI System Architecture, Fourth Edition by Tom Shanley, et al.

11.2. 总线资源

FreeBSD为从父总线请求资源提供了一种面向对象的机制。几乎所有设备 都是某种类型的总线（PCI，ISA，USB，SCSI等等）的孩子成员，并且这些设备 需要从他们的父总线获取资源（例如内存段，中断线，或者DMA通道）。

11.2.1. 基地址寄存器

为了对PCI设备做些有用的事情，你需要从PCI配置空间获取 Base Address Registers (BARs)。获取BAR时的 PCI特定的细节被抽象在函数bus_alloc_resource()中。

例如，一个典型的驱动程序可能在attach() 函数中有些类似下面的东西：

```
    sc->bar0id = PCIR_BAR(0);
```

```
sc->bar0res = bus_alloc_resource(dev, SYS_RES_MEMORY, &sc->bar0id,
 0, ~0, 1, RF_ACTIVE);
if (sc->bar0res == NULL) {
  printf("Memory allocation of PCI base register 0 failed!\n");
  error = ENXIO;
  goto fail1;
-}

sc->bar1id = PCIR_BAR(1);
sc->bar1res = bus_alloc_resource(dev, SYS_RES_MEMORY, &sc->bar1id,
 0, ~0, 1, RF_ACTIVE);
if (sc->bar1res == NULL) {
  printf("Memory allocation of PCI base register 1 failed!\n");
  error =  ENXIO;
  goto fail2;
-}
sc->bar0_bt = rman_get_bustag(sc->bar0res);
sc->bar0_bh = rman_get_bushandle(sc->bar0res);
sc->bar1_bt = rman_get_bustag(sc->bar1res);
sc->bar1_bh = rman_get_bushandle(sc->bar1res);
```

每个基址寄存器的句柄被保存在softc 结构中，以便以后可以使用它们向设备写入。

然后就能使用这些句柄与bus_space_*函数一起 读写设备寄存器。例如，驱动程序可能包含如下的快捷函数，用来读取板子 特定的寄存器：

```
uint16_t
board_read(struct ni_softc *sc, uint16_t address)
{
  return bus_space_read_2(sc->bar1_bt, sc->bar1_bh, address);
}
```

类似的，可以用下面的函数写寄存器：

```
void
board_write(struct ni_softc *sc, uint16_t address, uint16_t value)
{
  bus_space_write_2(sc->bar1_bt, sc->bar1_bh, address, value);
}
```

这些函数以8位，16位和32位的版本存在，你应当相应地使用 bus_space_{read|write}_{1|2|4}。

注意

在 FreeBSD 7.0 和更高版本中， 可以用 bus_* 函数来代替 bus_space_*。 bus_* 函数使用的参数是 struct resource * 指针， 而不是 bus tag 和句柄。 这样， 您就可以将 softc 中的

bus tag 和 bus 句柄这两个成员变量去掉， 并将 board_read() 函数改写为：

```
uint16_t
board_read(struct ni_softc *sc, uint16_t address)
{
 return (bus_read(sc->bar1res, address));
}
```

11.2.2. 中断

中断按照和分配内存资源相似的方式从面向对象的总线代码分配。首先， 必须从父总线分配IRQ资源，然后必须设置中断处理函数来处理这个IRQ。

再一次，来自设备attach()函数的例子比文字 更具说明性。

```
/* ##IRQ## */

   sc->irqid = 0x0;
   sc->irqres = bus_alloc_resource(dev, SYS_RES_IRQ, &(sc->irqid),
     0, ~0, 1, RF_SHAREABLE -| RF_ACTIVE);
   if (sc->irqres == NULL) {
 printf("IRQ allocation failed!\n");
 error = ENXIO;
 goto fail3;
   -}

   -/* ############## */

   error = bus_setup_intr(dev, sc->irqres, INTR_TYPE_MISC,
     my_handler, sc, &(sc->handler));
   if (error) {
 printf("Couldn't set up irq\n");
 goto fail4;
   -}
```

在设备的分离例程中必须注意一些问题。你必须停顿设备的中断流， 并移除中断处理函数。一旦bus_teardown_intr() 返回，你知道你的中断处理函数不会再被调用，并且所有可能已经执行了 这个中断处理函数的线程都已经返回。由于此函数可以睡眠，调用此函数时 你必须不能拥有任何互斥体。

11.2.3. DMA

本节已废弃，只是由于历史原因而给出。处理这些问题的适当方法是 使用bus_space_dma*()函数。当更新这一节以反映 那样用法时，这段就可能被去掉。然而，目前API还不断有些变动，因此一旦 它们固定下来后，更新这一节来反映那些改动就很好了。

在PC上，想进行总线主控DMA的外围设备必须处理物理地址，由于 FreeBSD使用虚拟内存并且只处理虚地址，这仍是个问题。幸运的是，有个 函数，vtophys()可以帮助我们。

```
#include <vm/vm.h>
#include <vm/pmap.h>

#define vtophys(virtual_address) (...)
```

然而这个解决办法在alpha上有点不一样，并且我们真正想要的是一个 称为vtobus()的函数。

```
#if defined(__alpha__)
#define vtobus(va)      alpha_XXX_dmamap((vm_offset_t)va)
#else
#define vtobus(va)      vtophys(va)
#endif
```

11.2.4. 取消分配资源

取消attach()期间分配的所有资源非常重要。 必须小心谨慎，即使在失败的条件下也要保证取消分配那些正确的东西， 这样当你的驱动程序去掉后系统仍然可以使用。

第 12 章 通用访问方法SCSI控制器

写作：Sergey Babkin.
改编为手册：Murray Stokely.
翻译：.

12.1. 提纲

本文档假定读者对FreeBSD的设备驱动程序和SCSI协议有大致了解， 本文档中很多信息是从以下驱动程序中：

- ncr（/sys/pci/ncr.c）由Wolfgang Stanglmeier and Stefan Esser编写
- sym（/sys/dev/sym/sym_hipd.c）由Gerard Roudier编写
- aic7xxx（/sys/dev/aic7xxx/aic7xxx.c）由Justin T. Gibbs编写

和从CAM的代码本身（作者 Justin T. Gibbs， 见/sys/cam/*）中摘录。当一些解决方法看起来 极具逻辑性，并且基本上是从 Justin T. Gibbs 的代码中一字不差地摘录时， 我将其标记为“recommended”。

本文档以伪代码例子进行说明。尽管有时例子中包含很多细节，并且 看起来很像真正代码，但它仍然只是伪代码。这样写是为了以一种可理解 的方式来展示概念。对于真正的驱动程序，其它方法可能更模块化，并且 更加高效。文档也对硬件细节进行抽象，对于那些会模糊我们所要展示的 概念的问题，或被认为在开发者手册的其他章节中已有描述的问题也做同样 处理。这些细节通常以调用具有描述性名字的函数、注释或伪语句的形式展现。 幸运的是，具有实际价值的完整例子，包括所有细节，可以在真正的驱动 程序中找到。

12.2. 通用基础结构

CAM代表通用访问方法（Common Access Method）。它以类SCSI方式寻址 I/O总线。这就允许将通用设备驱动程序和控制I/O总线的驱动程序分离开来： 例如磁盘驱动程序能同时控制SCSI、IDE、且/或任何其他总线上的磁盘， 这样磁盘驱动程序部分不必为每种新的I/O总线而重写（或拷贝修改）。 这样，两个最重要的活动实体是：

- 外围设备模块 - 外围设备（磁盘， 磁带， CD-ROM等）的驱动程序

- SCSI接口模块(SIM) – 连接到I/O总线，如SCSI或IDE，的主机总线适配器驱动程序。

外围设备驱动程序从OS接收请求，将它们转换为SCSI命令序列并将 这些SCSI命令传递到SCSI接口模块。SCSI接口模块负责将这些命令传递给 实际硬件（或者如果实际硬件不是SCSI，而是例如IDE，则也要将这些SCSI 命令转换为硬件的native命令）。

由于这儿我们感兴趣的是编写SCSI适配器驱动程序，从此处开始我们 将从SIM的角度考虑所有的事情。

典型的SIM驱动程序需要包括如下的CAM相关的头文件：

```
#include <cam/cam.h>
#include <cam/cam_ccb.h>
#include <cam/cam_sim.h>
#include <cam/cam_xpt_sim.h>
#include <cam/cam_debug.h>
#include <cam/scsi/scsi_all.h>
```

每个SIM驱动程序必须做的第一件事情是向CAM子系统注册它自己。 这在驱动程序的xxx_attach()函数（此处和以后的 xxx_用于指带唯一的驱动程序名字前缀）期间完成。 xxx_attach()函数自身由系统总线自动配置代码 调用，我们在此不描述这部分代码。

这需要好几步来完成：首先需要分配与SIM关联的请求队列：

```
  struct cam_devq *devq;

  if(( devq = cam_simq_alloc(SIZE) -)==NULL) {
    error; -/* ######### */
  -}
```

此处 SIZE 为要分配的队列的大小， 它能包含的最大请求数目。 它是 SIM 驱动程序在 SCSI 卡上能够并行处理的请求的数目。一般可以如下估算：

```
SIZE = NUMBER_OF_SUPPORTED_TARGETS * -
MAX_SIMULTANEOUS_COMMANDS_PER_TARGET
```

下一步为我们的SIM创建描述符：

```
  struct cam_sim *sim;

  if(( sim = cam_sim_alloc(action_func, poll_func, driver_name,
        softc, unit, max_dev_transactions,
        max_tagged_dev_transactions, devq) -)==NULL) {
    cam_simq_free(devq);
    error; -/* ######## */
  -}
```

注意如果我们不能创建SIM描述符，我们也释放 devq，因为我们对其无法做任何其他事情， 而且我们想节约内存。

如果SCSI卡上有多条SCSI总线，则每条总线需要它自己的 cam_sim 结构。

一个有趣的问题是，如果SCSI卡有不只一条SCSI总线我们该怎么做， 每个卡需要一个devq结构还是每条SCSI总线？ 在CAM代码的注释中给出的答案是：任一方式均可，由驱动程序的作者 选择。

参量为：

- action_func – 指向驱动程序 xxx_action 函数的指针。

  static void **xxx_action** (**struct cam_sim *sim**, **union ccb *ccb**);

  struct cam_sim *sim, , **union ccb *ccb** ;

- poll_func – 指向驱动程序 xxx_poll()函数的指针。

  static void **xxx_poll** (**struct cam_sim *sim**);

  struct cam_sim *sim ;

- driver_name – 实际驱动程序的名字，例如 “ncr” 或“wds” 。

- softc – 指向这个SCSI卡 驱动程序的内部描述符的指针。这个指针以后被驱动程序用来获取 私有数据。

- unit – 控制器单元号，例如对于控制器 “wds0”的此数字将为0。

- max_dev_transactions – 无标签模式下每个SCSI目标的 最大并发（simultaneous）事务数。这个值一般几乎总是等于1，只有非 SCSI卡才可能例外。此外，如果驱动程序希望执行一个事务的同时准备另 一个事务，可以将其设置为2，但似乎不值得增加这种复杂性。

- max_tagged_dev_transactions – 同样的东西，但是 在标签模式下。标签是SCSI在设备上发起多个事务的方式：每个事务 被赋予一个唯一的标签，并被发送到设备。当设备完成某些事务，它 将结果连同标签一起发送回来，这样SCSI适配器（和驱动程序）就能知道 哪个事务完成了。此参量也被认为是最大标签深度。它取决于SCSI 适配器的能力。

最后我们注册与我们的SCSI适配器关联的SCSI总线。

```
if(xpt_bus_register(sim, bus_number) -!= CAM_SUCCESS) {
   cam_sim_free(sim, -/*free_devq*/ TRUE);
   error; -/* ######## */
-}
```

如果每条SCSI总线有一个devq结构（即， 我们将带有多条总线的卡看作多个卡，每个卡带有一条总线），则总线号 总是为0，否则SCSI卡上的每条总线应当有不同的号。每条总线需要 它自己单独的cam_sim结构。

这之后我们的控制器完全挂接到CAM系统。现在 devq的值可以被丢弃：在所有以后从CAM发出的 调用中将以sim为参量，devq可以由它导出。

CAM为这些异步事件提供了框架。有些事件来自底层（SIM驱动程序）， 有些来自外围设备驱动程序，还有一些来自CAM子系统本身。任何驱动 程序都可以为某些类型的异步事件注册回调，这样那些事件发生时它就 会被通知。

这种事件的一个典型例子就是设备复位。每个事务和事件以 “path”的方式区分它们所作用的设备。目标特定的事件 通常在与设备进行事务处理期间发生。因此那个事务的路径可以被重用 来报告此事件（这是安全的，因为事件路径的拷贝是在事件报告例程中进行的， 而且既不会被deallocate也不作进一步传递）。在任何时刻，包括中断例程中， 动态分配路径也是安全的，尽管那样会导致某些额外开销，并且这种方法 可能存在的一个问题是碰巧那时可能没有空闲内存。对于总线复位事件， 我们需要定义包括总线上所有设备在内的通配符路径。这样我们就能提前为 以后的总线复位事件创建路径，避免以后内存不足的问题：

```
struct cam_path *path;

if(xpt_create_path(&path, -/*periph*/NULL,
        cam_sim_path(sim), CAM_TARGET_WILDCARD,
        CAM_LUN_WILDCARD) -!= CAM_REQ_CMP) {
    xpt_bus_deregister(cam_sim_path(sim));
    cam_sim_free(sim, -/*free_devq*/TRUE);
    error; -/* ######## */
-}

softc->wpath = path;
softc->sim = sim;
```

正如你所看到的，路径包括：

- 外围设备驱动程序的ID（由于我们一个也没有，故此处为空）
- SIM驱动程序的ID （cam_sim_path(sim)）
- 设备的SCSI目标号（CAM_TARGET_WILDCARD的意思指 “所有devices”）
- 子设备的SCSI LUN号（CAM_LUN_WILDCARD的意思指 “所有LUNs”）

如果驱动程序不能分配这个路径，它将不能正常工作，因此那样情况下 我们卸除（dismantle）那个SCSI总线。

我们在softc结构中保存路径指针以便以后 使用。这之后我们保存sim的值（或者如果我们愿意，也可以在从 xxx_probe()退出时丢弃它）。

这就是最低要求的初始化所需要做的一切。为了把事情做正确无误， 还剩下一个问题。

对于SIM驱动程序，有一个特殊感兴趣的事件：何时目标设备被认为 找不到了。这种情况下复位与这个设备的SCSI协商可能是个好主意。因此我们 为这个事件向CAM注册一个回调。通过为这种类型的请求来请求CAM控制块上 的CAM动作，请求就被传递到CAM：（译注：参看下面示例代码和原文）

```
struct ccb_setasync csa;

xpt_setup_ccb(&csa.ccb_h, path, -/*###*/5);
csa.ccb_h.func_code = XPT_SASYNC_CB;
csa.event_enable = AC_LOST_DEVICE;
csa.callback = xxx_async;
csa.callback_arg = sim;
xpt_action((union ccb *)&csa);
```

现在我们看一下xxx_action() 和xxx_poll()的驱动程序入口点。

static void **xxx_action** (**struct cam_sim *sim**, **union ccb *ccb**);

struct cam_sim *sim, , **union ccb *ccb** ;

响应CAM子系统的请求采取某些动作。Sim描述了请求的SIM，CCB为 请求本身。CCB代表“CAM Control Block”。它是很多特定 实例的联合，每个实例为某些类型的事务描述参量。所有这些实例共享 存储着参量公共部分的CCB头部。（译注：这一段不很准确，请自行参考原文）

CAM既支持SCSI控制器工作于发起者(initiator)(“normal”) 模式，也支持SCSI控制器工作于目标(target)（模拟SCSI设备）模式。这儿 我们只考虑与发起者模式有关的部分。

定义了几个函数和宏（换句话说，方法）来访问结构sim中公共数据：

- cam_sim_path(sim) - 路径ID （参见上面）
- cam_sim_name(sim) - sim的名字
- cam_sim_softc(sim) - 指向softc（驱动程序私有数据）结构的指针
- cam_sim_unit(sim) - 单元号
- cam_sim_bus(sim) - 总线ID

为了识别设备，xxx_action()可以使用这些 函数得到单元号和指向它的softc结构的指针。

请求的类型被存储在 ccb->ccb_h.func_code。因此，通常 xxx_action()由一个大的switch组成：

```
struct xxx_softc *softc = (struct xxx_softc *) cam_sim_softc(sim);
struct ccb_hdr *ccb_h = &ccb->ccb_h;
int unit = cam_sim_unit(sim);
int bus = cam_sim_bus(sim);

switch(ccb_h->func_code) {
case -...:
   -...
default:
   ccb_h->status = CAM_REQ_INVALID;
   xpt_done(ccb);
   break;
-}
```

从default case语句部分可以看出（如果收到未知命令），命令的返回码 被设置到 ccb->ccb_h.status 中，并且通过 调用xpt_done(ccb)将整个CCB返回到CAM中。

xpt_done()不必从 xxx_action()中调用：例如I/O请求可以在SIM驱动程序 和/或它的SCSI控制器中排队。（译注：它指I/O请求？） 然后，当设备传递(post)一个中断信号，指示对此请求的处理已结束时， xpt_done()可以从中断处理例程中被调用。

实际上，CCB状态不是仅仅被赋值为一个返回码，而是始终有某种状态。 CCB被传递给xxx_action()例程前，其取得状态 CCB_REQ_INPROG，表示其正在进行中。/sys/cam/cam.h 中定义了数量惊人的状态值，它们应该能非常详尽地表示请求的状态。更有趣的是，状态实际上是一个枚举状态值（低6位）和一些可能出现的附加 类(似)旗标位（高位）的“位或(bitwise or)”。枚举值会在以后 更详细地讨论。对它们的汇总可以在错误概览节(Errors Summary section) 找到。可能的状态旗标为：

- CAM_DEV_QFRZN – 当处理CCB时， 如果SIM驱动程序得到一个严重错误（例如，驱动程序不能响应选择或违反 了SCSI协议），它应当调用xpt_freeze_simq()冻结 请求队列，把此设备的其他已入队但尚未被处理的CCB返回到CAM队列， 然后为有问题的CCB设置这个旗标并调用 xpt_done()。这个旗标会使得CAM子系统处理错误后 解冻队列。

- CAM_AUTOSNS_VALID – 如果设备 返回错误条件，且CCB中未设置旗标CAM_DIS_AUTOSENSE，SIM驱动程序 必须自动执行REQUEST SENSE命令来从设备抽取sense（扩展错误信息） 数据。如果这个尝试成功，sense数据应当被保存在CCB中且设置此旗标。

- CAM_RELEASE_SIMQ – 类似于 CAM_DEV_QFRZN，但用于SCSI控制器自身出问题(或资源短缺)的情况。 此后对控制器的所有请求会被xpt_freeze_simq() 停止。SIM驱动程序克服这种短缺情况，并通过返回设置了此旗标的CCB 通知CAM后，控制器队列将会被重新启动。

- CAM_SIM_QUEUED – 当SIM将一个 CCB放入其请求队列时应当设置此旗标（或当CCB出队但尚未返回给CAM时 去掉）。现在此旗标还没有在CAM代码的任何地方使用过，因此其目的 纯粹用于诊断）。

函数xxx_action()不允许睡眠，因此对资源 访问的所有同步必须通过冻结SIM或设备队列来完成。除了前述的旗标外， CAM子系统提供了函数xpt_release_simq()和xpt_release_devq()来直接解冻队列，而不必将 CCB传递到CAM。

CCB头部包含如下字段：

- path - 请求的路径ID
- target_id - 请求的目标设备ID
- target_lun - 目标设备的LUN ID
- timeout - 这个命令的超时间隔，以毫秒计
- timeout_ch - 一个为SIM驱动 程序存储超时处理函数的方便之所（CAM子系统自身并不对此作任何假设）
- flags - 有关请求的各个 信息位
- spriv_ptr0，spriv_ptr1 - SIM驱动程序保留私用的字段 （例如链接到SIM队列或SIM私有控制块）；实际上，它们作为联合存在： spriv_ptr0和spriv_ptr1具有类型(void *)，spriv_field0和 spriv_field1具有类型unsigned long，sim_priv.entries[0].bytes和 sim_priv.entries[1].bytes为与联合的其他形式大小一致的字节数组， sim_priv.bytes为一个两倍大小的数组

使用CCB的SIM私有字段的建议方法是为它们定义一些有意义的名字， 并且在驱动程序中使用这些有意义的名字，就像下面这样：

```
#define ccb_some_meaningful_name    sim_priv.entries[0].bytes
#define ccb_hcb spriv_ptr1 -/* ####### */
```

最常见的发起者模式的请求是：

- XPT_SCSI_IO - 执行I/O事务

  联合ccb的“struct ccb_scsiio csio”实例用于传递参量。 它们是：

  - cdb_io - 指向SCSI命令缓冲区的指针或缓冲区本身
  - cdb_len - SCSI命令长度
  - data_ptr - 指向数据缓冲区的指针（如果使用分散/集中会复杂一点）
  - dxfer_len - 待传输数据的长度
  - sglist_cnt - 分散/集中段的计数
  - scsi_status - 返回SCSI状态的地方

- sense_data – 命令返回错误时保存SCSI sense信息的缓冲区（这种情况下，如果没有 设置CCB的旗标CAM_DIS_AUTOSENSE，则假定SIM驱动程序会自动运行REQUEST SENSE命令）

- sense_len – 缓冲区的长度（如果碰巧大于sense_data的大小，SIM驱动程序必须 悄悄地采用较小值）（译注：一点改动，参考原文及代码）

- resid，sense_resid – 如果数据传输或SCSI sense返回错误，则它们 就是返回的剩余（未传输）数据的计数。它们看起来并不是特别有意义， 因此当很难计算的情况下（例如，计数SCSI控制器FIFO缓冲区中的字节 数），使用近似值也同样可以。对于成功完成的传输，它们必须被设置 为0。

- tag_action – 使用的标签的种类有：

  - CAM_TAG_ACTION_NONE – 事务不使用标签

  - MSG_SIMPLE_Q_TAG，MSG_HEAD_OF_Q_TAG，MSG_ORDERED_Q_TAG – 值等于适当的标签信息 （见/sys/cam/scsi/scsi_message.h）；仅给出标签类型，SIM驱动程序 必须自己赋标签值

处理请求的通常逻辑如下：

要做的第一件事情是检查可能的竞争条件，确保命令位于队列中时 不会被中止：

```
struct ccb_scsiio *csio = &ccb->csio;

if ((ccb_h->status & CAM_STATUS_MASK) -!= CAM_REQ_INPROG) {
    xpt_done(ccb);
    return;
-}
```

我们也检查我们的控制器完全支持设备：

```
if(ccb_h->target_id > OUR_MAX_SUPPORTED_TARGET_ID
-|| cch_h->target_id == OUR_SCSI_CONTROLLERS_OWN_ID) {
    ccb_h->status = CAM_TID_INVALID;
    xpt_done(ccb);
    return;
-}
if(ccb_h->target_lun > OUR_MAX_SUPPORTED_LUN) {
    ccb_h->status = CAM_LUN_INVALID;
    xpt_done(ccb);
    return;
-}
```

然后分配我们处理请求所需的数据结构（如卡相关的硬件控制块等）。 如果我们不能分配则冻结SIM队列，记录下我们有一个挂起的操作，返回 CCB，

请求CAM将CCB重新入队。以后当资源可用时，必须通过返回其 状态中设置 CAM_SIMQ_RELEASE 位的ccb来解冻SIM队列。否则，如果所有 正常，则将CCB与硬件控制块（HCB）链接，并将其标志为已入队。

```
struct xxx_hcb *hcb = allocate_hcb(softc, unit, bus);

if(hcb == NULL) {
   softc->flags -|= RESOURCE_SHORTAGE;
   xpt_freeze_simq(sim, -/*count*/1);
   ccb_h->status = CAM_REQUEUE_REQ;
   xpt_done(ccb);
   return;
-}

hcb->ccb = ccb; ccb_h->ccb_hcb = (void *)hcb;
ccb_h->status -|= CAM_SIM_QUEUED;
```

从CCB中提取目标数据到硬件控制块。检查是否要求我们分配一个 标签，如果是则产生一个唯一的标签并构造SCSI标签信息。SIM驱动程序 也负责与设备协商设定彼此支持的最大总线宽度、同步速率和偏移。

```
hcb->target = ccb_h->target_id; hcb->lun = ccb_h->target_lun;
generate_identify_message(hcb);
if( ccb_h->tag_action -!= CAM_TAG_ACTION_NONE -)
   generate_unique_tag_message(hcb, ccb_h->tag_action);
if( -!target_negotiated(hcb) -)
   generate_negotiation_messages(hcb);
```

然后设置SCSI命令。可以在CCB中以多种有趣的方式指定命令的存储， 这些方式由CCB中的旗标指定。命令缓冲区可以包含在CCB中或者用指针 指向，后者情况下指针可以指向物理地址或虚地址。由于硬件通常需要 物理地址，因此我们总是将地址转换为物理地址。

不太相关的提示：通常这是通过调用vtophys()来完成的，但由于 特殊的Alpha怪异之处，为了PCI设备（它们现在占SCSI控制器的大多数） 驱动程序向Alpha架构的可移植性，转换必须替代以 vtobus() 来完成。［IMHO 提供两个单独的函数 vtop() 和 ptobus()，而 vtobus() 只是它们的 简单叠代，这样做要好得多。］在请求物理地址的情况下，返回带有状态 CAM_REQ_INVALID 的CCB是可以的，当前的驱动程序就是那样做的。但也 可能像这个例子（驱动程序中应当有不带条件编译的更直接做法）中那样 编译Alpha特定的代码片断。如果需要物理地址也能转换或映射回虚地址， 但那样代价很大，因此我们不那样做。

```
if(ccb_h->flags & CAM_CDB_POINTER) {
   -/* CDB is a pointer */
   if(!(ccb_h->flags & CAM_CDB_PHYS)) {
      -/* CDB###### */
      hcb->cmd = vtobus(csio->cdb_io.cdb_ptr);
   -} else {
      -/* CDB###### */
```

```
#if defined(__alpha__)
          hcb->cmd = csio->cdb_io.cdb_ptr -| alpha_XXX_dmamap_or -;
#else
          hcb->cmd = csio->cdb_io.cdb_ptr -;
#endif
       -}
    -} else {
       -/* CDB#ccb(###)# */
       hcb->cmd = vtobus(csio->cdb_io.cdb_bytes);
    -}
    hcb->cmdlen = csio->cdb_len;
```

现在是设置数据的时候了，又一次，可以在CCB中以多种有趣的方式 指定数据存储，这些方式由CCB中的旗标指定。首先我们得到数据传输的 方向。最简单的情况是没有数据需要传输的情况：

```
    int dir = (ccb_h->flags & CAM_DIR_MASK);

    if (dir == CAM_DIR_NONE)
       goto end_data;
```

然后我们检查数据在一个chunk中还是在分散/集中列表中，并且是 物理地址还是虚地址。SCSI控制器可能只能处理有限数目有限长度的 大块。如果请求到达到这个限制我们就返回错误。我们使用一个特殊 函数返回CCB，并在一个地方处理HCB资源短缺。增加chunk的函数是 驱动程序相关的，此处我们不进入它们的详细实现。对于地址翻译问题 的细节可以参看SCSI命令(CDB)处理的描述。如果某些变体对于特定的卡 太困难或不可能实现，返回状态 CAM_REQ_INVALID 是可以的。实际上， 现在的CAM代码中似乎哪儿也没有使用分散/集中能力。但至少必须实现 单个非分散虚拟缓冲区的情况，CAM中这种情况用得很多。

```
    int rv;

    initialize_hcb_for_data(hcb);

    if((!(ccb_h->flags & CAM_SCATTER_VALID)) {
       -/* ##### */
       if(!(ccb_h->flags & CAM_DATA_PHYS)) {
          rv = add_virtual_chunk(hcb, csio->data_ptr, csio->dxfer_len, dir);
          -}
       -} else {
          rv = add_physical_chunk(hcb, csio->data_ptr, csio->dxfer_len, dir);
       -}
    -} else {
       int i;
       struct bus_dma_segment *segs;
       segs = (struct bus_dma_segment *)csio->data_ptr;

       if ((ccb_h->flags & CAM_SG_LIST_PHYS) -!= 0) {
          -/* SG######## */
          rv = setup_hcb_for_physical_sg_list(hcb, segs, csio->sglist_cnt);
       -} else if (!(ccb_h->flags & CAM_DATA_PHYS)) {
```

```
        -/* SG######### */
        for (i = 0; i < csio->sglist_cnt; i++) {
            rv = add_virtual_chunk(hcb, segs[i].ds_addr,
                segs[i].ds_len, dir);
            if (rv -!= CAM_REQ_CMP)
                break;
        -}
    -} else {
        -/* SG######### */
        for (i = 0; i < csio->sglist_cnt; i++) {
            rv = add_physical_chunk(hcb, segs[i].ds_addr,
                segs[i].ds_len, dir);
            if (rv -!= CAM_REQ_CMP)
                break;
        -}
    -}
-}
if(rv -!= CAM_REQ_CMP) {
    -/* ########chunk#####add_*_chunk()####
     * CAM_REQ_CMP###############chunks#
     * ###CAM_REQ_TOO_BIG, #######CAM_REQ_INVALID#
     */
    free_hcb_and_ccb_done(hcb, ccb, rv);
    return;
-}
end_data:
```

如果这个CCB不允许断开连接，我们就传递这个信息到hcb：

```
if(ccb_h->flags & CAM_DIS_DISCONNECT)
    hcb_disable_disconnect(hcb);
```

如果控制器能够完全自己运行REQUEST SENSE命令，则也应当将旗标CAM_DIS_AUTOSENSE的值传递给它，这样可以在CAM子系统不想REQUEST SENSE 时阻止自动REQUEST SENSE。

剩下的唯一事情是设置超时，将我们的hcb传递给硬件并返回，余下的 由中断处理函数（或超时处理函数）完成。

```
ccb_h->timeout_ch = timeout(xxx_timeout, (caddr_t) hcb,
    (ccb_h->timeout * hz) -/ 1000); -/* ######### */
put_hcb_into_hardware_queue(hcb);
return;
```

这儿是返回CCB的函数的一个可能实现：

```
static void
free_hcb_and_ccb_done(struct xxx_hcb *hcb, union ccb *ccb, u_int32_t status)
{
    struct xxx_softc *softc = hcb->softc;

    ccb->ccb_h.ccb_hcb = 0;
```

```
    if(hcb -!= NULL) {
        untimeout(xxx_timeout, (caddr_t) hcb, ccb->ccb_h.timeout_ch);
        -/* #####hcb############### */
        if(softc->flags & RESOURCE_SHORTAGE)  {
            softc->flags &= ~RESOURCE_SHORTAGE;
            status -|= CAM_RELEASE_SIMQ;
        -}
        free_hcb(hcb); -/* ############hcb */
    -}
    ccb->ccb_h.status = status -|
        (ccb->ccb_h.status & ~(CAM_STATUS_MASK|CAM_SIM_QUEUED));
    xpt_done(ccb);
-}
```

- XPT_RESET_DEV – 发送SCSI “BUS DEVICE RESET” 消息到设备

  除了头部外CCB中没有数据传输，其中最让人感兴趣的参量为target_id。 依赖于控制器硬件，硬件控制块就像XPT_SCSI_IO请求中那样被创建 （参看XPT_SCSI_IO请求的描述）并被发送到控制器，或者立即编程让SCSI 控制器发送RESET消息到设备，或者这个请求可能只是不被支持 （并返回状态CAM_REQ_INVALID）。而且请求完成时，目标的所有已断开 连接(disconnected)的事务必须被中止（可能在中断例程中）。

  而且目标的所有当前协商在复位时会丢失，因此它们也可能被清除。 或者清除可能被延迟，因为不管怎样目标将会在下一次事务时请求重新协商。

- XPT_RESET_BUS – 发送RESET信号到SCSI总线

  CCB中并不传递参量，唯一感兴趣的参量是由指向结构sim的指针标识 的SCSI总线。

  最小实现会忘记总线上所有设备的SCSI协商，并返回状态 CAM_REQ_CMP。

  恰当的实现实际上会另外复位SCSI总线（可能也复位SCSI控制器）并 将所有在硬件队列中的和断开连接的那些正被处理的CCB的完成状态标记为CAM_SCSI_BUS_RESET。像这样：

```
int targ, lun;
struct xxx_hcb *h, *hh;
struct ccb_trans_settings neg;
struct cam_path *path;

-/* SCSI################################
 * ########################
 */
reset_scsi_bus(softc);

-/* #######CCB */
for(h = softc->first_queued_hcb; h -!= NULL; h = hh) {
    hh = h->next;
    free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
```

```
-}

-/* ###################### */
neg.bus_width = 8;
neg.sync_period = neg.sync_offset = 0;
neg.valid = (CCB_TRANS_BUS_WIDTH_VALID
   -| CCB_TRANS_SYNC_RATE_VALID -| CCB_TRANS_SYNC_OFFSET_VALID);

-/* #########CCB#############clean# */
for(targ=0; targ <= OUR_MAX_SUPPORTED_TARGET; targ++) {
   clean_negotiations(softc, targ);

   -/* ######## */
   if(xpt_create_path(&path, -/*periph*/NULL,
         cam_sim_path(sim), targ,
         CAM_LUN_WILDCARD) == CAM_REQ_CMP) {
      xpt_async(AC_TRANSFER_NEG, path, &neg);
      xpt_free_path(path);
   -}

   for(lun=0; lun <= OUR_MAX_SUPPORTED_LUN; lun++)
      for(h = softc->first_discon_hcb[targ][lun]; h -!= NULL; h = hh) {
         hh=h->next;
         free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
      -}
-}

ccb->ccb_h.status = CAM_REQ_CMP;
xpt_done(ccb);

-/* #### */
xpt_async(AC_BUS_RESET, softc->wpath, NULL);
return;
```

将SCSI总线复位作为函数来实现可能是个好主意，因为如果事情出了差错， 它会被超时函数作为最后的报告来重用。

- XPT_ABORT - 中止指定的CCB

  参量在联合ccb的实例 “struct ccb_abort cab” 中传输。其中唯一的参量字段为：

  abort_ccb - 指向被中止的ccb的指针

  如果不支持中断就返回CAM_UA_ABORT。这也是最小化实现这个调用的 简易方式，任何情况下都返回CAM_UA_ABORT。

  困难方式则是真正地实现这个请求。首先检查应用到SCSI事务的中止：

```
struct ccb *abort_ccb;
abort_ccb = ccb->cab.abort_ccb;
```

```
if(abort_ccb->ccb_h.func_code -!= XPT_SCSI_IO) {
   ccb->ccb_h.status = CAM_UA_ABORT;
   xpt_done(ccb);
   return;
-}
```

然后需要在我们的队列中找到这个CCB。这可以通过遍历我们所有硬件 控制块列表，查找与这个CCB关联的控制块来完成：

```
struct xxx_hcb *hcb, *h;

hcb = NULL;

-/* ####softc->first_hcb#########HCB#######
 * ###########################
 */
for(h = softc->first_hcb; h -!= NULL; h = h->next) {
   if(h->ccb == abort_ccb) {
      hcb = h;
      break;
   -}
-}

if(hcb == NULL) {
   -/* ##########CCB */
   ccb->ccb_h.status = CAM_PATH_INVALID;
   xpt_done(ccb);
   return;
-}

hcb=found_hcb;
```

现在我们来看一下HCB当前的处理状态。它可能或呆在队列中正等待 被发送到SCSI总线，或此时正在传输中，或已断开连接并等待命令结果， 或者实际上已由硬件完成但尚未被软件标记为完成。为了确保我们不会 与硬件产生竞争条件，我们将HCB标记为中止(aborted)，这样如果这个 HCB要被发送到SCSI总线的话，SCSI控制器将会看到这个旗标并跳过它。

```
int hstatus;

-/* ################################
 */
set_hcb_flags(hcb, HCB_BEING_ABORTED);

abort_again:

hstatus = get_hcb_status(hcb);
switch(hstatus) {
case HCB_SITTING_IN_QUEUE:
   remove_hcb_from_hardware_queue(hcb);
   -/* #### */
case HCB_COMPLETED:
```

```
        -/* ######### */
        free_hcb_and_ccb_done(hcb, abort_ccb, CAM_REQ_ABORTED);
        break;
```

如果CCB此时正在传输中，我们一般会以某种硬件相关的方式发信号 给SCSI控制器，通知它我们希望中止当前的传输。SCSI控制器会设置 SCSI ATTENTION信号，并当目标对其进行响应后发送ABORT消息。我们也复位 超时，以确保目标不会永远睡眠。如果命令不能在某个合理的时间，如 10秒内中止，超时例程就会运行并复位整个SCSI总线。由于命令会在某个 合理的时间后被中止，因此我们现在可以只将中止请求返回，当作成功完成， 并将被中止的CCB标记为中止（但还没有将它标记为完成）。

```
    case HCB_BEING_TRANSFERRED:
        untimeout(xxx_timeout, (caddr_t) hcb, abort_ccb->ccb_h.timeout_ch);
        abort_ccb->ccb_h.timeout_ch =
            timeout(xxx_timeout, (caddr_t) hcb, 10 * hz);
        abort_ccb->ccb_h.status = CAM_REQ_ABORTED;
        -/* #######CCB############
         */
        if(signal_hardware_to_abort_hcb_and_stop(hcb) < 0) {
            -/* ########################
             * ###################
      * #######getoff#*/
            goto abort_again;
        -}

        break;
```

如果CCB位于断开连接的列表中，则将它设置为中止请求，并在硬件 队列的前端将它重新入队。复位超时，并报告中止请求完成。

```
    case HCB_DISCONNECTED:
        untimeout(xxx_timeout, (caddr_t) hcb, abort_ccb->ccb_h.timeout_ch);
        abort_ccb->ccb_h.timeout_ch =
            timeout(xxx_timeout, (caddr_t) hcb, 10 * hz);
        put_abort_message_into_hcb(hcb);
        put_hcb_at_the_front_of_hardware_queue(hcb);
        break;
    -}
    ccb->ccb_h.status = CAM_REQ_CMP;
    xpt_done(ccb);
    return;
```

这就是关于ABORT请求的全部，尽管还有一个问题。由于ABORT消息 清除LUN上所有正在进行中的事务，我们必须将LUN上所有其他活动事务 标记为中止。那应当在中断例程中完成，且在中止事务之后。

将CCB中止作为函数来实现可能是个很好的主意，因为如果I/O事务超时 这个函数能够被重用。唯一的不同是超时事务将为超时请求返回状态 CAM_CMD_TIMEOUT。于是XPT_ABORT的case语句就会很小，像下面这样：

```
case XPT_ABORT:
    struct ccb *abort_ccb;
    abort_ccb = ccb->cab.abort_ccb;

    if(abort_ccb->ccb_h.func_code -!= XPT_SCSI_IO) {
        ccb->ccb_h.status = CAM_UA_ABORT;
        xpt_done(ccb);
        return;
    -}
    if(xxx_abort_ccb(abort_ccb, CAM_REQ_ABORTED) < 0)
        -/* no such CCB in our queue */
        ccb->ccb_h.status = CAM_PATH_INVALID;
    else
        ccb->ccb_h.status = CAM_REQ_CMP;
    xpt_done(ccb);
    return;
```

- XPT_SET_TRAN_SETTINGS – 显式设置SCSI传输设置的值

在联合ccb的实例“struct ccb_trans_setting cts” 中传输的参量：

- valid – 位掩码，显示应当更新那些设置：
- CCB_TRANS_SYNC_RATE_VALID – 同步传输速率
- CCB_TRANS_SYNC_OFFSET_VALID – 同步位移
- CCB_TRANS_BUS_WIDTH_VALID – 总线宽度
- CCB_TRANS_DISC_VALID – 设置启用/禁用断开连接
- CCB_TRANS_TQ_VALID – 设置启用/禁用带标签的排队
- flags – 由两部分组成，两元参量和子操作标识。两元参量为：
  - CCB_TRANS_DISC_ENB – 启用断开连接
  - CCB_TRANS_TAG_ENB – 启用带标签的排队
- 子操作为：
  - CCB_TRANS_CURRENT_SETTINGS – 改变当前的协商
  - CCB_TRANS_USER_SETTINGS – 记住希望的用户值
  - sync_period, sync_offset – 自解释的，如果sync_offset==0则请求同步模式
  - bus_width – 总线带宽，以位计（而不是字节）

译注

参考原文和源码

支持两组协商参数，用户设置和当前设置。用户设置在SIM驱动程序中 实际上用得不多，这通常只是一片内存，供上层存储（并在以后恢复）其关于 参数的一些主张。设置用户参数并不会导致重新协商传输速率。但当SCSI 控制器协商时，它必须永远不能设置高于用户参数的值，因此它实质上是 上限。

当前设置，正如其名字所示，指当前的。改变它们意味着下一次传输时 必须重新协商参数。又一次，这些“new current settings” 并没有被假定为强制用于设备上，它们只是用作协商的起始步骤。此外， 它们必须受SCSI控制器的实际能力限制：例如，如果SCSI控制器有8位总线， 而请求要求设置16位传输，则在发送给设备前参数必须被悄悄地截取为8位。

一个需要注意的问题就是总线宽度和同步两个参数是针对每目标的而言的， 而断开连接和启用标签两个参数是针对每lun而言的。

建议的实现是保持3组协商参数（总线宽度和同步传输）：

- user - 用户的一组，如上
- current - 实际生效的那些
- goal - 通过设置“current”参数所请求的那些

代码看起来像：

```
struct ccb_trans_settings *cts;
int targ, lun;
int flags;

cts = &ccb->cts;
targ = ccb_h->target_id;
lun = ccb_h->target_lun;
flags = cts->flags;
if(flags & CCB_TRANS_USER_SETTINGS) {
   if(flags & CCB_TRANS_SYNC_RATE_VALID)
      softc->user_sync_period[targ] = cts->sync_period;
   if(flags & CCB_TRANS_SYNC_OFFSET_VALID)
      softc->user_sync_offset[targ] = cts->sync_offset;
   if(flags & CCB_TRANS_BUS_WIDTH_VALID)
      softc->user_bus_width[targ] = cts->bus_width;
```

```
    if(flags & CCB_TRANS_DISC_VALID) {
        softc->user_tflags[targ][lun] &= ~CCB_TRANS_DISC_ENB;
        softc->user_tflags[targ][lun] -|= flags & CCB_TRANS_DISC_ENB;
    -}
    if(flags & CCB_TRANS_TQ_VALID) {
        softc->user_tflags[targ][lun] &= ~CCB_TRANS_TQ_ENB;
        softc->user_tflags[targ][lun] -|= flags & CCB_TRANS_TQ_ENB;
    -}
-}
if(flags & CCB_TRANS_CURRENT_SETTINGS) {
    if(flags & CCB_TRANS_SYNC_RATE_VALID)
        softc->goal_sync_period[targ] =
            max(cts->sync_period, OUR_MIN_SUPPORTED_PERIOD);
    if(flags & CCB_TRANS_SYNC_OFFSET_VALID)
        softc->goal_sync_offset[targ] =
            min(cts->sync_offset, OUR_MAX_SUPPORTED_OFFSET);
    if(flags & CCB_TRANS_BUS_WIDTH_VALID)
        softc->goal_bus_width[targ] = min(cts->bus_width, OUR_BUS_WIDTH);

    if(flags & CCB_TRANS_DISC_VALID) {
        softc->current_tflags[targ][lun] &= ~CCB_TRANS_DISC_ENB;
        softc->current_tflags[targ][lun] -|= flags & CCB_TRANS_DISC_ENB;
    -}
    if(flags & CCB_TRANS_TQ_VALID) {
        softc->current_tflags[targ][lun] &= ~CCB_TRANS_TQ_ENB;
        softc->current_tflags[targ][lun] -|= flags & CCB_TRANS_TQ_ENB;
    -}
-}
ccb->ccb_h.status = CAM_REQ_CMP;
xpt_done(ccb);
return;
```

此后当下一次要处理I/O请求时，它会检查其是否需要重新协商， 例如通过调用函数target_negotiated(hcb)。它可以如下实现：

```
int
target_negotiated(struct xxx_hcb *hcb)
{
    struct softc *softc = hcb->softc;
    int targ = hcb->targ;

    if( softc->current_sync_period[targ] -!= softc->goal_sync_period[targ]
    -|| softc->current_sync_offset[targ] -!= softc->goal_sync_offset[targ]
    -|| softc->current_bus_width[targ] -!= softc->goal_bus_width[targ] -)
        return 0; -/* FALSE */
    else
        return 1; -/* TRUE */
-}
```

重新协商这些值后，结果值必须同时赋给当前和目的(goal)参数， 这样对于以后的I/O事务当前和目的参数将相同，且 target_negotiated()会返回TRUE。当初始化卡 （在xxx_attach()中）当前协商值必须被初始化为 最窄同步模式，目的和当

前值必须被初始化为控制器所支持的最大值。 （译注：原文可能有误，此处未改）

- XPT_GET_TRAN_SETTINGS - 获得SCSI传输设置的值

  此操作为XPT_SET_TRAN_SETTINGS的逆操作。用通过旗标CCB_TRANS_CURRENT_SETTINGS或CCB_TRANS_USER_SETTINGS（如果同时设置则 现有驱动程序返回当前设置）所请求而得的数据填充CCB实例 “struct ccb_trans_setting cts”.

- XPT_CALC_GEOMETRY - 计算磁盘的逻辑（BIOS）结构(geometry)

  参量在联合ccb的实例“struct ccb_calc_geometry ccg” 中传输：

  - block_size - 输入，以字节计的块大小（也称为扇区）
  - volume_size - 输入，以字节计的卷大小
  - cylinders - 输出，逻辑柱面
  - heads - 输出，逻辑磁头
  - secs_per_track - 输出，每磁道的逻辑扇区

  如果返回的结构与SCSI控制器BIOS所想象的差别很大，并且SCSI 控制器上的磁盘被作为可引导的，则系统可能无法启动。从aic7xxx 驱动程序中摘取的典型计算示例：

```
struct    ccb_calc_geometry *ccg;
u_int32_t size_mb;
u_int32_t secs_per_cylinder;
int   extended;

ccg = &ccb->ccg;
size_mb = ccg->volume_size
    -/ ((1024L * 1024L) -/ ccg->block_size);
extended = check_cards_EEPROM_for_extended_geometry(softc);

if (size_mb > 1024 && extended) {
    ccg->heads = 255;
    ccg->secs_per_track = 63;
-} else {
    ccg->heads = 64;
    ccg->secs_per_track = 32;
-}
secs_per_cylinder = ccg->heads * ccg->secs_per_track;
ccg->cylinders = ccg->volume_size -/ secs_per_cylinder;
ccb->ccb_h.status = CAM_REQ_CMP;
xpt_done(ccb);
return;
```

这给出了一般思路，精确计算依赖于特定BIOS的癖好(quirk)。如果 BIOS没有提供方法设置EEPROM中的“extended translation” 旗标，则此旗标通常应当假定等于1。其他流行结构有：

```
128 heads, 63 sectors -- Symbios###
16 heads, 63 sectors -- #####
```

一些系统BIOS和SCSI BIOS会相互竞争，胜负不定，例如Symbios 875/895 SCSI和Phoenix BIOS的结合在系统加电时会给出结构128/63， 而当冷启动或软启动后会是255/63。

- XPT_PATH_INQ – 路径问询， 换句话说，获得SIM驱动程序和SCSI控制器（也称为HBA – 主机总线适配器） 的特性。

  特性在联合ccb的实例“struct ccb_pathinq cpi” 中返回：

  - version_num – SIM驱动程序号，当前所有驱动程序使用1
  - hba_inquiry – 控制器所支持特性的位掩码：
  - PI_MDP_ABLE – 支持MDP消息（来自SCSI3的一些东西?）
  - PI_WIDE_32 – 支持32位宽SCSI
  - PI_WIDE_16 – 支持16位宽SCSI
  - PI_SDTR_ABLE – 可以协商同步传输速率
  - PI_LINKED_CDB – 支持链接的命令
  - PI_TAG_ABLE – 支持带标签的命令
  - PI_SOFT_RST – 支持软复位选择 （硬复位和软复位在SCSI总线中是互斥的）
  - target_sprt – 目标模式支持的旗标，如果不支持则为0
  - hba_misc – 控制器特性杂项：
  - PIM_SCANHILO – 从高ID到低ID的总线扫描
  - PIM_NOREMOVE – 可移除设备不包括在扫描之列
  - PIM_NOINITIATOR – 不支持发起者角色
  - PIM_NOBUSRESET – 用户禁用初始BUS RESET
  - hba_eng_cnt – 神秘的HBA引擎计数，与压缩有关的一些 东西，当前总是置为0

- vuhba_flags - 供应商唯一的旗标，当前未用
- max_target - 最大支持的目标ID（对8位总线为7， 16位总线为15，光纤通道为127）
- max_lun - 最大支持的LUN ID（对较老的SCSI控制器 为7，较新的为63）
- async_flags - 安装的异步处理函数的位掩码，当前未用
- hpath_id - 子系统中最高的路径ID，当前未用
- unit_number - 控制器单元号，cam_sim_unit(sim)
- bus_id - 总线号，cam_sim_bus(sim)
- initiator_id - 控制器自己的SCSI ID
- base_transfer_speed - 异步窄传输的名义传输速率， 以KB/s计，对于SCSI等于3300
- sim_vid - SIM驱动程序的供应商ID，以0结束的字符串， 包含结尾0在内的最大长度为SIM_IDLEN
- hba_vid - SCSI控制器的供应商ID，以0结束的字符串， 包含结尾0在内的最大长度为HBA_IDLEN
- dev_name - 设备驱动程序名字，以0结尾的字符串， 包含结尾0在内的最大长度为DEV_IDLEN，等于cam_sim_name(sim)

设置字符串字段的建议方法是使用strncpy，如：

```
strncpy(cpi->dev_name, cam_sim_name(sim), DEV_IDLEN);
```

设置这些值后将状态设置为CAM_REQ_CMP，并将CCB标记为完成。

12.3. 轮询

static void **xxx_poll (struct cam_sim *sim)**;

struct cam_sim *sim ;

轮询函数用于当中断子系统不起作用时（例如，系统崩溃或正在创建 系统转储）模拟中断。CAM子系统在调用轮询函数前设置适当的中断级别。 因此它所需做全部的只是调用中断例程（或其他方法，轮询例程来 进行实际动作， 而中断例程只是调用轮询例程）。那么为什么要找麻烦 弄出一个单独的函数来呢？这是由于不同的调用约定。 xxx_poll例程取结构cam_sim的指针作为参量， 而PCI中断例程按照

普通约定取的是指向结构 xxx_softc的指针，ISA中断例程只是取设备号， 因此轮询例程一般看起来像：

```
static void
xxx_poll(struct cam_sim *sim)
{
    xxx_intr((struct xxx_softc *)cam_sim_softc(sim)); -/* for PCI device */
}
```

or

```
static void
xxx_poll(struct cam_sim *sim)
{
    xxx_intr(cam_sim_unit(sim)); -/* for ISA device */
}
```

12.4. 异步事件

如果建立了异步事件回调，则应当定义回调函数。

```
static void
ahc_async(void *callback_arg, u_int32_t code, struct cam_path *path, void *arg)
```

- callback_arg - 注册回调时提供的值
- code - 标识事件类型
- path - 标识事件作用于其上的设备
- arg - 事件特定的参量

单一类型事件的实现，AC_LOST_DEVICE，看起来如下：

```
    struct xxx_softc *softc;
    struct cam_sim *sim;
    int targ;
    struct ccb_trans_settings neg;

    sim = (struct cam_sim *)callback_arg;
    softc = (struct xxx_softc *)cam_sim_softc(sim);
    switch (code) {
    case AC_LOST_DEVICE:
        targ = xpt_path_target_id(path);
        if(targ <= OUR_MAX_SUPPORTED_TARGET) {
            clean_negotiations(softc, targ);
            -/* send indication to CAM */
            neg.bus_width = 8;
            neg.sync_period = neg.sync_offset = 0;
```

```
        neg.valid = (CCB_TRANS_BUS_WIDTH_VALID
          -| CCB_TRANS_SYNC_RATE_VALID -| CCB_TRANS_SYNC_OFFSET_VALID);
        xpt_async(AC_TRANSFER_NEG, path, &neg);
      -}
      break;
  default:
      break;
  -}
```

12.5. 中断

中断例程的确切类型依赖于SCSI控制器所连接到的外围总线的类型（PCI， ISA等等）。

SIM驱动程序的中断例程运行在中断级别splcam上。因此应当在驱动 程序中使用splcam()来同步中断例程与驱动程序 剩余部分的活动（对于能察觉多处理器的驱动程序，事情更要有趣，但 此处我们忽略这种情况）。本文档中的伪代码简单地忽略了同步问题。 实际代码一定不能忽略它们。一个较笨的办法就是在进入其他例程的 入口点处设splcam()，并在返回时将它复位，从而 用一个大的临界区保护它们。为了确保中断级别总是会被恢复，可以定义 一个包装函数，如：

```
static void
xxx_action(struct cam_sim *sim, union ccb *ccb)
{
   int s;
   s = splcam();
   xxx_action1(sim, ccb);
   splx(s);
-}

static void
xxx_action1(struct cam_sim *sim, union ccb *ccb)
{
   -... process the request -...
-}
```

这种方法简单而且健壮，但它存在的问题是中断可能会被阻塞相对 很长的事件，这会对系统性能产生负面影响。另一方面， spl()函数族有相当高的额外开销，因此大量 很小的临界区可能也不好。

中断例程处理的情况和其中细节严重依赖于硬件。我们考虑 “典型(typical)”情况。

首先，我们检查总线上是否遇到了SCSI复位（可能由同一SCSI总线上 的另一SCSI控制器引起）。如果这样我们丢弃所有入队的和断开连接的 请求，报告事件并重新初始化我们的SCSI控制器。初始化期间控制器 不会发出另一个复位，这对我们十分重要，否则同一SCSI总线上的两个控制器 可能会一直来回地复位下去。控制

器致命错误/挂起的情况可以在同一　地方进行处理，但这可能需要发送RESET信号到SCSI总线来复位与SCSI 设备的连接状态。

```
int fatal=0;
struct ccb_trans_settings neg;
struct cam_path *path;

if( detected_scsi_reset(softc)
-|| (fatal = detected_fatal_controller_error(softc)) -) {
   int targ, lun;
   struct xxx_hcb *h, *hh;

   -/* #######CCB */
   for(h = softc->first_queued_hcb; h -!= NULL; h = hh) {
      hh = h->next;
      free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
   -}

   -/* ########## */
   neg.bus_width = 8;
   neg.sync_period = neg.sync_offset = 0;
   neg.valid = (CCB_TRANS_BUS_WIDTH_VALID
      -| CCB_TRANS_SYNC_RATE_VALID -| CCB_TRANS_SYNC_OFFSET_VALID);

   -/* #########CCB##### */
   for(targ=0; targ <= OUR_MAX_SUPPORTED_TARGET; targ++) {
      clean_negotiations(softc, targ);

      -/* report the event if possible */
      if(xpt_create_path(&path, -/*periph*/NULL,
            cam_sim_path(sim), targ,
            CAM_LUN_WILDCARD) == CAM_REQ_CMP) {
         xpt_async(AC_TRANSFER_NEG, path, &neg);
         xpt_free_path(path);
      -}

      for(lun=0; lun <= OUR_MAX_SUPPORTED_LUN; lun++)
         for(h = softc->first_discon_hcb[targ][lun]; h -!= NULL; h = hh) {
            hh=h->next;
            if(fatal)
               free_hcb_and_ccb_done(h, h->ccb, CAM_UNREC_HBA_ERROR);
            else
               free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
         -}
   -}

   -/* #### */
   xpt_async(AC_BUS_RESET, softc->wpath, NULL);

   -/* ############################
    * ################ -- #########
    * #######
    */
```

```
        if(!fatal) {
            reinitialize_controller_without_scsi_reset(softc);
        -} else {
            reinitialize_controller_with_scsi_reset(softc);
        -}
        schedule_next_hcb(softc);
        return;
    -}
```

如果中断不是由控制器范围的条件引起的，则很可能当前硬件控制块 出现了问题。依赖于硬件，可能有非HCB相关的事件，此处我们指示不考虑 它们。然后我们分析这个HCB发生了什么：

```
    struct xxx_hcb *hcb, *h, *hh;
    int hcb_status, scsi_status;
    int ccb_status;
    int targ;
    int lun_to_freeze;

    hcb = get_current_hcb(softc);
    if(hcb == NULL) {
        -/* ####(stray)################
         * #############
         */
        #######;
        return;
    -}

    targ = hcb->target;
    hcb_status = get_status_of_current_hcb(softc);
```

首先我们检查HCB是否完成，如果完成我们就检查返回的SCSI状态。

```
    if(hcb_status == COMPLETED) {
        scsi_status = get_completion_status(hcb);
```

然后看这个状态是否与REQUEST SENSE命令有关，如果有关则简单 地处理一下它。

```
        if(hcb->flags & DOING_AUTOSENSE) {
            if(scsi_status == GOOD) { -/* autosense## */
                hcb->ccb->ccb_h.status -|= CAM_AUTOSNS_VALID;
                free_hcb_and_ccb_done(hcb, hcb->ccb, CAM_SCSI_STATUS_ERROR);
            -} else {
        autosense_failed:
                free_hcb_and_ccb_done(hcb, hcb->ccb, CAM_AUTOSENSE_FAIL);
            -}
            schedule_next_hcb(softc);
            return;
        -}
```

否则命令自身已经完成，把更多注意力放在细节上。如果这个CCB 没有禁用auto-sense并且命令连同sense数据失败，则运行REQUEST SENSE 命令接收那些数据。

```
    hcb->ccb->csio.scsi_status = scsi_status;
    calculate_residue(hcb);

    if( (hcb->ccb->ccb_h.flags & CAM_DIS_AUTOSENSE)==0
    && ( scsi_status == CHECK_CONDITION
            -|| scsi_status == COMMAND_TERMINATED) -) {
        -/* ##auto-SENSE */
        hcb->flags -|= DOING_AUTOSENSE;
        setup_autosense_command_in_hcb(hcb);
        restart_current_hcb(softc);
        return;
    -}
    if(scsi_status == GOOD)
        free_hcb_and_ccb_done(hcb, hcb->ccb, CAM_REQ_CMP);
    else
        free_hcb_and_ccb_done(hcb, hcb->ccb, CAM_SCSI_STATUS_ERROR);
    schedule_next_hcb(softc);
    return;
-}
```

属于协商事件的一个典型事情：从SCSI目标（回答我们的协商企图或　　由目标发起的）接收到的协商消息，或目标无法协商（拒绝我们的协商消息　　或不回答它们）。

```
switch(hcb_status) {
case TARGET_REJECTED_WIDE_NEG:
    -/* ###8-bit## */
    softc->current_bus_width[targ] = softc->goal_bus_width[targ] = 8;
    -/* #### */
    neg.bus_width = 8;
    neg.valid = CCB_TRANS_BUS_WIDTH_VALID;
    xpt_async(AC_TRANSFER_NEG, hcb->ccb.ccb_h.path_id, &neg);
    continue_current_hcb(softc);
    return;
case TARGET_ANSWERED_WIDE_NEG:
    {
        int wd;

        wd = get_target_bus_width_request(softc);
        if(wd <= softc->goal_bus_width[targ]) {
            -/* ###### */
            softc->current_bus_width[targ] =
            softc->goal_bus_width[targ] = neg.bus_width = wd;

            -/* #### */
            neg.valid = CCB_TRANS_BUS_WIDTH_VALID;
            xpt_async(AC_TRANSFER_NEG, hcb->ccb.ccb_h.path_id, &neg);
        -} else {
            prepare_reject_message(hcb);
        -}
    -}
    continue_current_hcb(softc);
    return;
```

```
case TARGET_REQUESTED_WIDE_NEG:
    {
        int wd;

        wd = get_target_bus_width_request(softc);
        wd = min (wd, OUR_BUS_WIDTH);
        wd = min (wd, softc->user_bus_width[targ]);

        if(wd -!= softc->current_bus_width[targ]) {
            -/* ####### */
            softc->current_bus_width[targ] =
            softc->goal_bus_width[targ] = neg.bus_width = wd;

            -/* #### */
            neg.valid = CCB_TRANS_BUS_WIDTH_VALID;
            xpt_async(AC_TRANSFER_NEG, hcb->ccb.ccb_h.path_id, &neg);
        -}
        prepare_width_nego_rsponse(hcb, wd);
    -}
    continue_current_hcb(softc);
    return;
-}
```

然后我们用与前面相同的笨办法处理auto-sense期间可能出现的任何　　错误。否则，我们再一次进入细节。

```
if(hcb->flags & DOING_AUTOSENSE)
    goto autosense_failed;

switch(hcb_status) {
```

我们考虑的下一事件是未预期的连接断开，这个事件在ABORT或 BUS DEVICE RESET消息之后被看作是正常的，其他情况下是非正常的。

```
case UNEXPECTED_DISCONNECT:
    if(requested_abort(hcb)) {
        -/* #######LUN###############LUN##
         * #######HCB######
         */
        for(h = softc->first_discon_hcb[hcb->target][hcb->lun];
                h -!= NULL; h = hh) {
            hh=h->next;
            free_hcb_and_ccb_done(h, h->ccb, CAM_REQ_ABORTED);
        -}
        ccb_status = CAM_REQ_ABORTED;
    -} else if(requested_bus_device_reset(hcb)) {
        int lun;

        -/* ######################LUN##
         * #######HCB#####
         */

        for(lun=0; lun <= OUR_MAX_SUPPORTED_LUN; lun++)
```

```
            for(h = softc->first_discon_hcb[hcb->target][lun];
                    h -!= NULL; h = hh) {
                hh=h->next;
                free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
            -}

        -/* #### */
        xpt_async(AC_SENT_BDR, hcb->ccb->ccb_h.path_id, NULL);

        -/* ##CAM_RESET_DEV######### */
        ccb_status = CAM_REQ_CMP;
    -} else {
        calculate_residue(hcb);
        ccb_status = CAM_UNEXP_BUSFREE;
        -/* request the further code to freeze the queue */
        hcb->ccb->ccb_h.status -|= CAM_DEV_QFRZN;
        lun_to_freeze = hcb->lun;
    -}
    break;
```

如果目标拒绝接受标签，我们就通知CAM，并返回此LUN的所有命令：

```
case TAGS_REJECTED:
    -/* #### */
    neg.flags = 0 & ~CCB_TRANS_TAG_ENB;
    neg.valid = CCB_TRANS_TQ_VALID;
    xpt_async(AC_TRANSFER_NEG, hcb->ccb.ccb_h.path_id, &neg);

    ccb_status = CAM_MSG_REJECT_REC;
    -/* ########### */
    hcb->ccb->ccb_h.status -|= CAM_DEV_QFRZN;
    lun_to_freeze = hcb->lun;
    break;
```

然后我们检查一些其他情况，处理(processing)基本上仅限于设置CCB状态：

```
case SELECTION_TIMEOUT:
    ccb_status = CAM_SEL_TIMEOUT;
    -/* request the further code to freeze the queue */
    hcb->ccb->ccb_h.status -|= CAM_DEV_QFRZN;
    lun_to_freeze = CAM_LUN_WILDCARD;
    break;
case PARITY_ERROR:
    ccb_status = CAM_UNCOR_PARITY;
    break;
case DATA_OVERRUN:
case ODD_WIDE_TRANSFER:
    ccb_status = CAM_DATA_RUN_ERR;
    break;
default:
    -/*############# */
    ccb_status = CAM_REQ_CMP_ERR;
    -/* ########### */
```

```
        hcb->ccb->ccb_h.status -|= CAM_DEV_QFRZN;
        lun_to_freeze = CAM_LUN_WILDCARD;
        break;
    -}
```

然后我们检查是否错误严重到需要冻结输入队列，直到它得到处理方可 解冻，如果是这样那么就这样来处理：

```
    if(hcb->ccb->ccb_h.status & CAM_DEV_QFRZN) {
        -/* #### */
        xpt_freeze_devq(ccb->ccb_h.path, -/*count*/1);

*       -/* ########/LUN###########CAM */

        for(h = softc->first_queued_hcb; h -!= NULL; h = hh) {
            hh = h->next;

            if(targ == h->targ
            && (lun_to_freeze == CAM_LUN_WILDCARD -|| lun_to_freeze == h->lun) -)
                free_hcb_and_ccb_done(h, h->ccb, CAM_REQUEUE_REQ);
        -}
    -}
    free_hcb_and_ccb_done(hcb, hcb->ccb, ccb_status);
    schedule_next_hcb(softc);
    return;
```

这包括通用中断处理，尽管特定处理器可能需要某些附加处理。

12.6. 错误总览

当执行I/O请求时很多事情可能出错。可以在CCB状态中非常详尽地 报告错误原因。使用的例子散布于本文档中。为了完整起见此处给出 对典型错误条件的建议响应的一个总览：

- CAM_RESRC_UNAVAIL - 某些资源 暂时不可用，并且当其变为可用时SIM驱动程序不能产生事件。这种资源 的一个例子就是某些控制器内部硬件资源，当其可用时控制器不会为其 产生中断。
- CAM_UNCOR_PARITY - 发生不可恢复的奇偶校验错误
- CAM_DATA_RUN_ERR - 数据外溢或未预期的数据状态(phase)(跑在另一个方向上而不是 CAM_DIR_MASK指定的方向)，或对于宽传输出现奇数传输长度
- CAM_SEL_TIMEOUT - 发生选择超时（目标不响应）
- CAM_CMD_TIMEOUT - 发生命令超时（超时函数运行）
- CAM_SCSI_STATUS_ERROR - 设备返回的错误

- CAM_AUTOSENSE_FAIL - 设备返回的错误且REQUEST SENSE命令失败
- CAM_MSG_REJECT_REC - 收到MESSAGE REJECT消息
- CAM_SCSI_BUS_RESET - 收到SCSI总线复位
- CAM_REQ_CMP_ERR - 出现“不可能(impossible)”SCSI状态(phase) 或者其他怪异事情，或者如果进一步的信息不可用则只是通用错误
- CAM_UNEXP_BUSFREE - 出现未预期的断开连接
- CAM_BDR_SENT - BUS DEVICE RESET消息被发送到目标
- CAM_UNREC_HBA_ERROR - 不可恢复的主机总线适配器错误
- CAM_REQ_TOO_BIG - 请求对于控制器太大
- CAM_REQUEUE_REQ - 此请求应当被重新入队以保持事务的次序性。这典型地出现在下列 时刻：SIM识别出了应当冻结队列的错误，并且必须在sim级别上将目标的 其他入队请求放回到XPT队列。这些错误的典型情况有选择超时、命令 超时和其他类似情况。这些情况下出问题的命令返回状态来指示错误， 此命令和其他还没有被发送到总线的命令被重新入队。
- CAM_LUN_INVALID - SCSI控制器不支持请求中的LUN ID
- CAM_TID_INVALID - SCSI控制器不支持请求中的目标ID

12.7. 超时处理

当HCB的超时期满时，请求就应当被中止，就像处理XPT_ABORT请求 一样。唯一区别在于被中止的请求的返回状态应当为CAM_CMD_TIMEOUT 而不是CAM_REQ_ABORTED（这就是为什么中止的实现最好由函数来完成）。 但还有一个可能的问题：如果中止请求自己出了麻烦怎么办？这种情况下 应当复位SCSI总线，就像处理XPT_RESET_BUS请求一样（并且将其实现为 函数，从两个地方调用的想法也适用于这儿）。而且如果设备复位请求出了 问题，我们应当复位整个SCSI总线。因此最终超时函数看起来像下面样子：

```
static void
xxx_timeout(void *arg)
{
    struct xxx_hcb *hcb = (struct xxx_hcb *)arg;
    struct xxx_softc *softc;
    struct ccb_hdr *ccb_h;

    softc = hcb->softc;
    ccb_h = &hcb->ccb->ccb_h;

    if(hcb->flags & HCB_BEING_ABORTED
```

```
    -|| ccb_h->func_code == XPT_RESET_DEV) {
        xxx_reset_bus(softc);
    -} else {
        xxx_abort_ccb(hcb->ccb, CAM_CMD_TIMEOUT);
    -}
}
```

当我们中止一个请求时，同一目标/LUN的所有其他断开连接的请求 也会被中止。因此出现了一个问题，我们应当返回它们的状态 CAM_REQ_ABORTED还是CAM_CMD_TIMEOUT? 当前的驱动程序使用 CAM_CMD_TIMEOUT。这看起来符合逻辑，因为如果一个请求超时，则可能 设备出现了某些的确很糟的事情，因此如果它们没有被扰乱则它们自己 应当超时。

第 13 章 USB设备

写作：Nick Hibma.
改编为手册：Murray Stokely.
翻译：.

13.1. 简介

通用串行总线(USB)是将设备连接到个人计算机的一种新方法。总线 结构突出了双向通信的特色，并且其开发充分考虑到了设备正逐渐智能化 和需要与host进行更多交互的现实。对USB的支持包含在当前所有芯片中， 因此在新近制造的PC中都可用。苹果(Apple)引入仅带USB的iMac对硬件 制造商生产他们USB版本的设备是一个很大的激励。未来的PC规范指定 PC上的所有老连接器应当由一个或多个USB连接器取代，提供通用的 即插即用能力。对USB硬件的支持在NetBSD的相当早期就有了，它是由 Lennart Augustsson为NetBSD项目开发的。代码已经被移植到FreeBSD上，我们目前维护着一个底层共享代码。对USB子系统的实现来说，许多USB的 特性很重要。

Lennart Augustsson已经完成了NetBSD项目中USB支持的 大部分实现。十分感谢这项工作量惊人的工作。也十分感谢Ardy和Dirk 对本文稿的评论和校对。

- 设备直接连接到计算机上的端口，或者连接到称为 集中器的设备，形成树型设备结构。
- 设备可在运行时连接或断开。
- 设备可以挂起自身并触发host系统的重新投入运行。
- 由于设备可由总线供电，因此host软件必须跟踪每个 集中器的电源预算。
- 不同设备类型需要不同的服务质量，并且同一总线 可以连接最多126个设备，这就需要恰当地调度总线上的传输以充分 利用12Mbps的可用带宽。（USB 2.0超过400Mbps）
- 设备智能化并包含很容易访问到的关于自身的信息。

为USB子系统以及连接到它的设备开发驱动程序受已开发或将要开发的 规范的支持。这些规范可以从USB主页公开获得。苹果(Apple)通过使得 通用类驱动程序可从其操作系统MacOS中获得，而且不鼓励为每种新设备 使用单独的驱动程序来强烈推行基于标准的驱动程序。本章试图整理基本 信息以便对FreeBSD/NetBSD中USB栈的当前实现有个基本的了解。然而， 建议将下面参考中提及的相关规范与本章同时阅读。

13.1.1. USB栈的结构

FreeBSD中的USB支持可被分为三层。最底层包含主控器，向硬件 及其调度设施提供一个通用接口。它支持硬件初始化，对传输进行调度， 处理已完成/失败的传输。每个主控器驱动程序实现一个虚拟hub， 以硬件无关方式提供对控制机器背面根端口的寄存器的访问。

中间层处理设备连接和断开，设备的基本初始化，驱动程序的选择， 通信通道（管道）和资源管理。这个服务层也控制默认管道和其上传输的 设备请求。

顶层包含支持特定（类）设备的各个驱动程序。这些驱动程序实现 除默认管道外的其他管道上使用的协议。他们也实现额外功能，使得设备 对内核或用户空间是可见的。他们使用服务层暴露出的USB驱动程序接口（USBDI）。

13.2. 主控器

主控器（HC）控制总线上包的传输。使用1毫秒的帧。在每帧开始 时，主控器产生一个帧开始（SOF，Start of Frame）包。

SOF包用于同步帧的开始和跟踪帧的数目。包在帧中被传输，或由host 到设备（out），或由设备到host（in）。传输总是由host发起（轮询传输）。 因此每条USB总线只能有一个host。每个包的传输都有一个状态阶段， 数据接收者可以在其中返回ACK（应答接收），NAK（重试），STALL（错误 条件）或什么也没有（混乱数据阶段，设备不可用或已断开）。USB规范 USB specification的第8.5节更详细地解释了包的细节。USB总线 上可以出现四中不同类型的传输：控制(control)，大块(bulk)， 中断 (interrupt)和同步(isochronous)。传输的类型和他们的特性在下面 描述（`管道'子节中）。

USB总线上的设备和设备驱动程序间的大型传输被主控器或HC 驱动程序分割为多个包。

到默认端点的设备请求（控制传输）有些特殊。它们由两或三个阶段 组成：启动（SETUP），数据（DATA，可选）和状态（STATUS）。设置（set-up） 包被发送到设备。如果存在数据阶段，数据包的方向在设置包中给出。 状态阶段中的方向与数据阶段期间的方向相反，或者当没有数据阶段时 为IN。主控器硬件也提供寄存器，用于保存根端口的当前状态和自从 状态改变寄存器最后一次复位以来所发生的改变。USB规范[2]建议使用一个 虚拟hub来提供对这些寄存器的访问。虚拟hub必须符合规范第11章中给出的 hub设备类。它必须提供一个默认管道使得设备请求可以发送给它。它返回 标准和hub类特定的一组描述符。它也应当提供一个中断管道用来报告其 端口发生的变化。当前可用的主控器规范有两个： 通用主控器接口（UHCI；英特尔）和 开放主控器接口（OHCI；康柏，微软，国家半导体）。 UHCI规范的设计通过要求主控器驱动程序为每帧的传输提供完整的调度，从而减少了硬件复杂性。OHCI类型的控制器自身提供一个更抽象的接口来 完成很多工作，从而更加独立。

13.2.1. UHCI

UHCI主控器维护着带有1024个指向每帧数据结构的帧列表。 它理解两种不同的数据类型：传输描述符（TD）和队列头（QH）。每个 TD表示表示与设备端点进行通信的一个包。QH是将一些TD（和QH）划分 成组的一种方法。

每个传输由一个或多个包组成。UHCI驱动程序将大的传输分割成 多个包。除同步传输外，每个传输都会分配一个QH。对于每种类型的 传输，都有一个与此类型对应的QH，所有这些QH都会被集中到这个QH上。 由于有固定的时延需求，同步传输必须首先执行，它是通过帧列表中的 指针直接引用的。最后的同步TD传输引用那一帧的中断传输的QH。中断 传输的所有QH指向控制传输的QH，控制传输的QH又指向大块传输的QH。 下面的图表给出了一个图形概览：

这导致下面的调度会在每帧中运行。控制器从帧列表中取得当前帧 的指针后，首先为那一帧中的所有的同步(isochronous)包执行TD。 这些TD的最后一个 引用那一帧的中断传输的QH。然后主控器将从那个QH下行到各个 中断传输的QH。完成那一队列后，中断传输的QH会将控制器指向到所有 控制传输的QH。它将执行在那儿等待调度的所有子队列，然后是在大块QH中 排队的所有传输。为了方便处理已完成或失败的传输，硬件会在每帧末尾 产生不同类型的中断。在传输的最后一个TD中，HC驱动程序设置 Interrupt-On-Completion位来标记传输完成时的一个中断。如果TD达到了 其最大错误数，就标记错误中断。如果在TD中设置短包侦测位，且传输了 小于所设置的包长度（的包），就会标记此中断以通知控制器驱动程序传输 已完成。找出哪个传输已完成或产生错误是主控器驱动程序的任务。 当中断服务例程被调用时，它将定位所有已完成的传输并调用它们的回调。

更详尽的描述请看 UHCI specification。

13.2.2. OHCI

对OHCI主控器进行编程要容易得多。控制器假设有一组端点(endpoint)可用， 并知道帧中不同传输类型的调度优先级和排序。主控器使用的主要 数据结构是端点描述符（ED），它上面连接着一个传输描述符（TD）的队列。 ED包含端点所允许的最大的包大小，控制器硬件完成包的分割。每次传输 后都会更新指向数据缓冲区的指针，当起始和终止指针相等时，TD就退归 到完成队列(done-queue)。四种类型的端点各有其自己的队列。控制和 大块(bulk)端点分别在它们自己的队列排队。中断ED在树中排队，在树中的深度 定义了它们运行的频度。

帧列表 中断 同步(isochronous) 控制 大块(bulk)

主控器在每帧中运行的调度看起来如下。控制器首先运行非 周期性控制和大块队列，最长可到HC驱动程序设置的一个时间限制。 然后以帧编号低5位作为中断ED树上深度为0的那一层中的索引，运行 那个帧编号的中断传输。在这个树的末尾，同步ED被连接，并随后被 遍历。同步TD包含了传输应当运行其中的第一个帧的帧编号。所有周期 性的传输运行过以后，控制和大块队列再次被遍历。中断服务例程

会被 周期性地调用，来处理完成的队列，为每个传输调用回调，并重新调度 中断和同步端点。

更详尽的描述请看 OHCI specification。服务层，即中间层，提供了以可控的方式 对设备进行访问，并维护着由不同驱动程序和服务层所使用的资源。 此层处理下面几方面：

- 设备配置信息
- 与设备进行通信的管道
- 探测和连接设备，以及从设备分离(detach)。

13.3. USB设备信息

13.3.1. 设备配置信息

每个设备提供了不同级别的配置信息。每个设备具有一个或多个 配置，探测/连接期间从其中选定一个。配置提供功率和带宽要求。 每个配置中可以有多个接口。设备接口是端点的汇集(collection)。 例如，USB扬声器可以有一个音频接口（音频类），和对旋钮(knob)、 拨号盘(dial)和按钮的接口（HID类）。 一个配置中的所有接口可以同时有效，并可被不同的 驱动程序连接。每个接口可以有备用接口，以提供不同质量的服务参数。 例如，在照相机中，这用来提供不同的帧大小以及每秒帧数。

每个接口中可以指定0或多个端点。端点是与设备进行通信的单向 访问点。它们提供缓冲区来临时存储从设备而来的，或外出到设备的数据。 每个端点在配置中有唯一地址，即端点号加上其方向。默认端点，即 端点0，不是任何接口的一部分，并且在所有配置中可用。它由服务层 管理，并且设备驱动程序不能直接使用。

Level 0 Level 1 Level 2 Slot 0

Slot 3 Slot 2 Slot 1

(只显示了32个槽中的4个)

这种层次化配置信息在设备中通过标准的一组描述符来描述（参看 USB规范[2]第9.6节）。它们可以通过Get Descriptor Request来请求。 服务层缓存这些描述符以避免在USB总线上进行不必要的传输。对这些 描述符的访问是通过函数调用来提供的。

- 设备描述符：关于设备的通用信息，如供应商，产品 和修订ID，支持的设备类、子类和适用的协议，默认端点的最大包大小 等。
- 配置描述符：此配置中的接口数，支持的挂起和 恢复能力，以及功率要求。
- 接口描述符：接口类、子类和适用的协议，接口备用 配置的数目和端点数目。

- 端点描述符：端点地址、方向和类型，支持的最大包 大小，如果是中断类型的端点则还包括轮询频率。默认端点（端点0） 没有描述符，而且从不被计入接口描述符中。

- 字符串描述符：在其他描述符中会为某些字段提供 字符串索引。它们可被用来检取描述性字符串，可能以多种语言 的形式提供。

类说明(specification)可以添加它们自己的描述符类型，这些描述符 也可以通过GetDescriptor Request来获得。

管道与设备上端点的通信，流经所谓的管道。驱动程序将到端点的 传输提交到管道，并提供传输（异步传输）失败或完成时调用的回调， 或等待完成（同步传输）。到端点的传输在管道中被串行化。传输或者完成， 或者失败，或者超时（如果设置了超时）。对于传输有两种类型的超时。 超时的发生可能由于USB总线上的超时（毫秒）。这些超时被视为失败， 可能是由于设备断开连接引起的。另一种超时在软件中实现，当传输没有 在指定的时间（秒）内完成时触发。这是由于设备对传输的包否定应答引起的。 其原因是由于设备还没有准备好接收数据，缓冲区欠载或超载，或协议错误。

如果管道上的传输大于关联的端点描述符中指定的最大包大小，主 控器（OHCI）或HC驱动程序（UHCI）将按最大包大小分割传输，并且最后 一个包可能小于最大包的大小。

有时候对设备来说返回少于所请求的数据并不是个问题。例如， 到调制解调器的大块in传输可能请求200字节的数据，但调制解调器 那时只有5个字节可用。驱动程序可以设置短包(SPD)标志。它允许主 控器即使在传输的数据量少于所请求的数据量的情况下也接受包。 这个标志只在in传输中有效，因为将要被发送到设备的数据量总是事先 知道的。如果传输过程中设备出现不可恢复的错误，管道会被停顿。 接受或发送更多数据以前，驱动程序需要确定停顿的原因，并通过在 默认管道上发送清除端点挂起设备请求(clear endpoint halt device request)来清除端点停顿条件。

有四种不同类型的端点和对应的管道： -

- 控制管道/默认管道： 每个设备有一个控制管道，连接到默认端点（端点0）。 此管道运载设备 请求和关联的数据。默认管道和其他管道上的传输的区别在于传输所 使用的协议，协议在USB规范[2]中描述。这些请求用于复位和配置设备。 每个设备必须支持USB规范[2]的第9章中提供的一组基本命令。管道上 支持的命令可以通过设备类规范扩展，以支持额外的功能。

- 大块(bulk)管道：这是USB与原始传输媒体对应的等价物。

- 中断管道：host向设备发送数据请求，如果设备没有 东西发送，则将NAK（否定应答）数据包。中断传输按创建管道时指定的 频率被调度。

- 同步管道：这些管道用于具有固定时延的同步数据， 例如视频或音频流，但不保证一定传输。当前实现中已经有对这种类型 管道的某些支持。当传输期间出

现错误，或者由于，例如缺乏缓冲区空间　来存储进入的数据而引起的设备否定应答包（NAK）时，控制、大块和中断 管道中的包会被重试。而同步包在传递失败或对包NAK时不会重试，因为 那样可能违反同步约束。

所需带宽的可用性在管道的创建期间被计算。传输在1毫秒的帧内 进行调度。帧中的带宽分配由USB规范的第5.6节规定。同步和中断传输被 允许消耗帧中多达90%的带宽。控制和大块传输的包在所有同步和中断包　之后进行调度，并将消耗所有剩余带宽。

关于传输调度和带宽回收的更多信息可以在USB规范[2]的第5章，　UHCI规范[3]的的第1.3节，OHCI规范[4]的3.4.2节中找到。

13.4. 设备的探测和连接

集中器(hub)通知新设备已连接后，服务层给端口加电(switch on)，　为设备提供100mA的电流。　此时设备处于其默认状态，并监听设备地址0。服务层会通过默认管道继续检取各种描述符。此后它将向设备发送Set Address请求，将设备 从默认设备地址(地址0)移开。可能有多个设备驱动程序支持此设备。例如，　一个调制解调器可能通过AT兼容接口支持ISDN TA。然而，特定型号的ISDN 适配器的驱动程序可能提供对此设备的更好支持。为了支持这样的灵活性，　探测会返回优先级，指示他们的支持级别。支持产品的特定版本会具有最高　优先级，通用驱动程序具有最低优先级。如果一个配置内有多个接口，也可能　多个驱动程序会连接到一个设备。每个驱动程序只需支持所有接口的一个子集。

为新连接的设备探测驱动程序时，首先探测设备特定的驱动程序。　如果没有发现，则探测代码在所有支持的配置上重复探测过程，直到　在一个配置中连接到一个驱动程序。为了支持不同接口上使用多个驱动　程序的设备，探测会在一个配置中的所有尚未被驱动程序声明(claim)的　接口上重复进行。超出集中器功率预算的配置会被忽略。连接期间，驱动　程序应当把设备初始化到适当状态，但不能复位，因为那样会使得设备将　它自己从总线上断开，并重新启动探测过程。为了避免消耗不必要的带宽，　不应当在连接时声明中断管道，而应当延迟分配管道，直到打开文件并真的　使用数据。当关闭文件时，管道也应当被再次关闭，尽管设备可能仍然 连接着。

13.4.1. 设备断开连接(disconnect)和分离(detach)

设备驱动程序与设备进行任何事务期间，应当预期会接收到错误。 USB的设计支持并鼓励设备在任何点及时断开连接。驱动程序应当确保　当设备不在时做正确的事情。

此外，断开连接(disconnect)后又重新连接(reconnect)的设备不会　被重新连接(reattach)为相同的设备实例。　将来当更多的设备支持序列号（参看设备描述符），　或开发出其他定义设备标识的方法的时候，这种情况可能会改变。

设备断开连接是由集中器在传递到集中器驱动程序的中断包中发 信号通知(signal)的。状态改变信息指示哪个端口发现了连接改变。 连接到那个端口上的设备的所有设备驱动程序共用的设备分离方法被调用, 结构被彻底清理。如果端口状态指示同时一个设备已经连接(connect)到那个 端口,则探测和连接设备的过程将被启动。设备复位将在集中器上产生 一个断开-连接序列,并将按上面所述进行处理。

13.5. USB驱动程序的协议信息

USB规范没有定义除默认管道外其他管道上使用的协议。这方面的信息 可以从各种来源获得。最准确的来源是USB主页[1]上的开发者部分。从这些 页面上可以得到数目不断增长的设备类的规范。这些规范指定从驱动程序 角度看起来兼容设备应当怎样,它需要提供的基本功能和通信通道上使用的 协议。USB规范[2]包括了集中器类的描述。人机界面设备(HID)的类规范已经 创建出来,以迎合对键盘、数字输入板、条形码阅读器、按钮、旋钮(手柄knob)、 开关等的要求。另一个例子是用于大容量存储设备的类规范。设备类的完整列表 参看USB主页[1]的开发者部分。

然而, 许多设备的协议信息还没有被公布。关于所用协议的信息 可能可以从制造设备的公司获得。一些公司会在给你规范之前要求你签署 保密协议(Non-Disclosure Agreement,NDA)。大多数情况下,这会阻止 将驱动程序开放源代码。

另一个信息的很好来源是Linux驱动程序源代码,因为很多公司已经 开始为他们的设备提供Linux下的驱动程序。联系那些驱动程序作者询问 他们的信息来源总是一个好主意。

例子:人机界面设备。人机界面设备,如键盘、鼠标、数字输入板、 按钮、拨号盘等的规范被其他设备类规范引用,并在很多设备中使用。

例如,音频扬声器提供到数模转换器的端点,可能还提供额外管道 用于麦克风。它们也为设备前面的按钮和拨号盘在单独的接口中提供HID 端点。监视器控制类也是如此。通过可用的内核和用户空间的库,与HID 类驱动程序或通用驱动程序一起可以简单直接地创建对这些接口的支持。 另一个设备可以作为在一个配置中的多个接口由不同的设备驱动程序驱动 的例子,这个设备是一种便宜的键盘,带有老的鼠标接口。为了避免在 设备中为USB集中器包括一个硬件而导致的成本上升,制造商将从键盘背面的 PS/2端口接收到的鼠标数据与来自键盘的按键组合成在同一个配置中的 两个单独的接口。鼠标和键盘驱动程序各自连接到适当的接口,并分配到 两个独立端点的管道.

例子:固件下载。已经开发出来的许多设备是基于通用目的处理器, 并将额外的USB核心加入其中。由于驱动程序的开发和USB设备的固件仍然 非常新,许多设备需要在连接(connect)之后下载固件。

下面的步骤非常简明直接。设备通过供应商和产品ID标识自身。第一 个驱动程序探测并连接到它,并将固件下载到其中。此后设备自己软复位, 驱动程序分离。

短暂的暂停之后设备宣布它在总线上的存在。设备将改变 其供应商/产品/版本的ID以反映其提供有固件的事实，因此另一个驱动程序 将探测它并连接(attach)到它。

这些类型的设备的一个例子是基于EZ-USB的ActiveWire I/O板。这个 芯片有一个通用固件下载器。下载到ActiveWire板子上的固件改变版本ID。 然后它将执行EZ-USB芯片的USB部分的软复位，从USB总线上断开，并再次 重新连接。

例子：大容量存储设备。对大容量存储设备的支持主要围绕现有的 协议构建。Iomega USB Zip驱动器是基于SCSI版本的驱动器。SCSI命令和 状态信息被包装到块中，在大块(bulk)管道上传输到/来自设备，在USB线 上模拟SCSI控制器。ATAPI和UFI命令以相似的方式被支持。

大容量存储规范支持两种不同类型的对命令块的包装。最初的尝试 基于通过默认管道发送命令和状态信息，使用大块传输在host和设备之间 移动数据。在经验基础上设计出另一种方法，这种方法基于包装命令和 状态块，并在大块out和in端点上发送它们。规范精确地指定了何时必须 发生什么，以及在碰到错误条件的情况下应该做什么。为这些设备编写 驱动程序的最大挑战是协调基于USB的协议，让它适合已有的对大容量存储设备 的支持。CAM提供了钩子，以相当直接了当的方式来完成这个。ATAPI就 没有这么简单了，因为历史上IDE接口从未有过多种不同的表现方式。

来自Y-E Data的对USB软盘的支持也不是那么直观，因为设计了一套 新的命令集。

第 14 章 Newbus

写作：Jeroen Ruigrok van der Werven (asmodai) 和 Hiten Pandya.
翻译：.

特别感谢Matthew N. Dodd, Warner Losh, Bill Paul, Doug Rabson, Mike Smith, Peter Wemm and Scott Long.

本章详细解释了Newbus设备框架。

14.1. 设备驱动程序

14.1.1. 设备驱动程序的目的

设备驱动程序是软件组件，它在内核关于外围设备（例如，磁盘、网络 适配卡）的通用视图和外围设备的实际实现之间提供了接口。 设备驱动程序接口(DDI)是内核与设备驱动程序组件 之间定义的接口。

14.1.2. 设备驱动程序的类型

在UNIX®那个时代，FreeBSD也从中延续而来，定义了四种类型的 设备：

- 块设备驱动程序
- 字符设备驱动程序
- 网络设备驱动程序
- 伪设备驱动程序

块设备以使用固定大小的[数据]块的方式运行。 这种类型的驱动程序依赖所谓的缓冲区缓存(buffer cache)，其目的 是在内存中的专用区域缓存访问过的数据块。这种缓冲区缓存常常基于后台写 (write-behind)，这意味着数据在内存中被修改后，当系统进行其周期性 磁盘刷新时才会被同步到磁盘，从而优化写操作。

14.1.3. 字符设备

然而，在FreeBSD 4.0版本以及后续版本中， 块设备和字符设备的区别变得不存在了。

14.2. Newbus概览

Newbus实现了一种基于抽象层的新型总线结构， 可以在FreeBSD 3.0中看到这种总线结构的介绍，当时Alpha的移植被导入到 代码树中。直到4.0它才成为设备驱动程序使用的默认系统。其目的是为主机 系统提供给操作系统的各种总线和设备的互连提供更加 面向对象的方法。

其主要特性包括：

- 动态连接
- 驱动程序容易模块化
- 伪总线

最显著的改变之一是从平面和特殊系统演变为设备树布局。

顶层驻留的是“根”设备，它作为 父设备，所有其他设备挂接在它上面。对于每个结构，通常“根” 只有单个孩子，其上连接着诸如host-to-PCI桥 等东西。对于x86，这种“根”设备为 “nexus”设备，对于Alpha，Alpha的各种 不同型号有不同的顶层设备，对应不同的硬件芯片组，包括 lca，apecs， cia和tsunami。

Newbus上下文中的设备表示系统中的单个硬件实体。例如，每个PCI设备被 表示为一个Newbus设备。系统中的任何设备可以有孩子；有孩子的设备通常被 称为“bus”。系统中常用总线的例子就是 ISA和PCI，他们各自管理连接到ISA和PCI总线上的设备列表。

通常，不同类型的总线之间的连接被表示为 “桥”设备，它的孩子就是它所连接的 总线。一个例子就是PCI-to-PCI桥，它在父PCI总线上被 表示为**pcibN**，而用它的孩子 **pciN**表示连接在它上面的 总线。这种布局简化了PCI总线树的实现，允许公共代码同时用于顶层和桥接的 总线。

Newbus结构中的每个设备请求它的父设备来为其映射资源。父设备接着请求 它的父设备，直到到达nexus。因此，基本上nexus是Newbus系统中唯一知道所有 资源的部分。

提示

ISA设备可能想在0x230映射其IO端口，因此它向其 父设备请求，这种情况下是ISA总线。ISA总线将它交给PCI-to-ISA桥，PCI-to-ISA 桥接着请求PCI总线，PCI总线到达host-to-PCI桥，最后到达nexus。这种向上 过渡的优美之处在于可以有空间来变换请求。对0x230IO端口 的请求在MIPS机器上可以被PCI桥变成 0xb0000230处的内存映射。

资源分配可以在设备树的任何地方加以控制。例如，在很多Alpha平台上， ISA中断与PCI中断是单独管理的，对ISA中断的资源分配是由Alpha的ISA总线设备 管理

的。在IA-32上，ISA和PCI中断都由顶层的nexus设备管理。对于两种移植， 内存和端口地址空间由单个实体管理 - 在IA-32上是nexus，在Alpha（例如，CIA 或tsunami）上是相关的芯片组驱动程序。

为了规范化对内存和端口映射资源的访问，Newbus整合了NetBSD的 bus_space API。他们提供了单一的API来代替inb/outb 和直接内存读写。这样做的优势在于单个驱动程序就可以使用内存映射寄存器 或端口映射寄存器（有些硬件支持两者）。

这种支持被合并到了资源分配机制中。分配资源时，驱动程序可以从资源 中检取关联的bus_space_tag_t和 bus_space_handle_t。

Newbus也允许在专用于此目的的文件中定义接口方法。这些是 .m文件，可以在src/sys 目录树中找到。

Newbus系统的核心是可扩展的“基于对象编程(object-based programming)”的模型。系统中的每个设备具有它所支持的一个方法表。 系统和其他设备使用这些方法来控制设备并请求服务。设备所支持的不同方法 被定义为多个“接口”。“接口”只是 设备实现的一组相关的方法。

在Newbus系统中，设备方法是通过系统中的各种设备驱动程序提供的。当 自动配置(auto-configuration)期间设备被连接(attach) 到驱动程序，它使用驱动程序声明的方法表。以后设备可以从其驱动程序 分离(detach)，并 重新连接(re-attach)到具有新方法表的新驱动程序。这就 允许驱动程序的动态替换，而动态替换对于驱动程序的开发非常有用。

接口通过与文件系统中用于定义vnode操作的语言相似的接口定义语言来 描述。接口被保存在方法文件中（通常命名为foo_if.m）。

例 14.1. Newbus的方法

```
# Foo ###/#######...#

INTERFACE foo

    METHOD int doit {
     device_t dev;
    };

    # ######DEVMETHOD()########DEFAULT#########

    METHOD void doit_to_child {
     device_t dev;
     driver_t child;
    } DEFAULT doit_generic_to_child;
```

当接口被编译后，它产生一个头文件 “foo_if.h” ，其中包含函数声明：

```
int FOO_DOIT(device_t dev);
int FOO_DOIT_TO_CHILD(device_t dev, device_t child);
```

伴随自动产生的头文件，也会创建一个源文件 “foo_if.c” ；其中包含一些函数的实现， 这些函数用于在对象方法表中查找相关函数的位置并调用那个函数。

系统定义了两个主要接口。第一个基本接口被称为 “设备(device)” ，并包括与所有设备相关 的方法。“设备(device)” 接口中的方法 包括 “探测(probe)” ，“连接(attach)” 和 “分离(detach)” ，他们用来控制硬件的侦测， 以及 “关闭(shutdown)” ， “挂起(suspend)” 和 “恢复(resume)” ，他们用于关键事件通知。

另一个，更加复杂接口是 “bus” 。 此接口包含的方法适用于带有孩子的设备，包括访问总线特定的每设备信息 [1]，事件通知 （**child_detached**，**driver_added**）和响应管理 （**alloc_resource**，**activate_resource**，**deactivate_resource**，**release_resource**） 。

“bus” 接口中的很多方法为总线设备的某些孩子执行服务。 这些方法通常使用前两个参量指定提供服务的总线和请求服务的子设备。为了 简化设备驱动程序代码，这些方法中的很多都有访问者(accessor)函数，访问者 函数用来查找父设备并调用父设备上的方法。例如，方法 BUS_TEARDOWN_INTR(device_t dev, device_t child, ...) 可以使用函数 bus_teardown_intr(device_t child, ...)来调用。

系统中的某些总线类型提供了额外接口以提供对总线特定功能的访问。 例如，PCI总线驱动程序定义了 “pci” 接口，此接口有两个方法 **read_config**和**write_config**，用于访问PCI设备 的配置寄存器。

14.3. Newbus API

由于Newbus API非常庞大，本节努力将它文档化。本文档的下一版本会 带来更多信息。

14.3.1. 源代码目录树中的重要位置

src/sys/[arch]/[arch] － 特定机器结构的 内核代码位于这个目录。例如i386结构或SPARC64结构。

[1] bus_generic_read_ivar(9) and bus_generic_write_ivar(9)

src/sys/dev/[bus] – 支持特定 [bus]的设备位于这个目录。

src/sys/dev/pci – PCI总线支持代码位于 这个目录。

src/sys/[isa|pci] – PCI/ISA设备驱动程序 位于这个目录。FreeBSD4.0版本中，PCI/ISA支持代码 过去存在于这个目录中。

14.3.2. 重要结构和类型定义

devclass_t – 这是指向 struct devclass的指针的类型定义。

device_method_t – 与 kobj_method_t相同（参看 src/sys/kobj.h）。

device_t – 这是指向 struct device的指针的类型定义。 device_t 表示系统中的设备。它是内核对象。 实现细节参看src/sys/sys/bus_private.h。

driver_t – 这是一个类型定义，它引用 struct driver。 driver结构是一类 device(##)内核对象；它也保存着驱动程序的私有数据。

```
struct driver {
    KOBJ_CLASS_FIELDS;
     void *priv;   /* ######## */
-};
```

图 14.1. driver_t实现

device_state_t是一个枚举类型，即 device_state。它包含Newbus设备在自动配置前后可能的状态。

```
-/*
 * src/sys/sys/bus.h
 */
typedef enum device_state {
 DS_NOTPRESENT, /* ######## */
   DS_ALIVE,  /* #### */
   DS_ATTACHED, /* ####### */
   DS_BUSY   /* ##### */
-} device_state_t;
```

图 14.2. 设备状态device_state_t

第 15 章 声音子系统

供稿：Jean-Francois Dockes.
翻译：.

15.1. 简介

FreeBSD声音子系统清晰地将通用声音处理问题与设备特定的问题分离 开来。这使得更容易加入对新设备的支持。

pcm(4)框架是声音子系统的中心部分。它主要实现下面的组件：

- 一个到数字化声音和混音器函数的系统调用接口（read， write， ioctls）。ioctl命令集合兼容老的OSS 或Voxware接口，允许一般多媒体应用程序 不加修改地移植。
- 处理声音数据的公共代码（格式转换，虚拟通道）。
- 一个统一的软件接口，与硬件特定的音频接口模块接口
- 对某些通用硬件接口（ac97）或共享的硬件特定代码 （例如：ISA DMA例程）的额外支持。

对特定声卡的支持是通过硬件特定的驱动程序来实现的，这些驱动程序 提供通道和混音器接口，插入到通用pcm代码中。

本章中，术语pcm将指声音驱动程序的 中心，通用部分，这是对比硬件特定的模块而言的。

预期的驱动程序编写者当然希望从现有模块开始，并使用那些代码作为 最终参考。但是，由于声音代码十分简洁漂亮，这也基本上免除了注释。 本文档试图给出框架接口的一个概览，并回答改写现有代码时可能出现的 一些问题。

作为另外的途径，或者说除了从一个可工作的范例开始的办法之外， 你可以从http://people.FreeBSD.org/~cg/template.c找到一个注释过的 驱动程序模板。

15.2. 文件

除/usr/src/sys/sys/soundcard.h中的公共 ioctl接口定义外，所有的相关代码当前(FreeBSD 4.4)位于 /usr/src/sys/dev/sound/。

在/usr/src/sys/dev/sound/下面， pcm/目录中保存着中心代码， 而isa/和pci/目录中有ISA和PCI板的驱动程序。

15.3. 探测，连接等

声音驱动程序使用与任何硬件驱动程序模块相同的方法探测和连接（设备）。 你可能希望浏览一下手册中ISA或PCI章节的内容来获取更多信息。

然而，声音驱动程序在某些方面又有些不同：

- 他们将自己声明为pcm类设备，带有一个 设备私有结构struct snddev_info：

  ```
          static driver_t xxx_driver = {
              -"pcm",
              xxx_methods,
              sizeof(struct snddev_info)
          -};

          DRIVER_MODULE(snd_xxxpci, pci, xxx_driver, pcm_devclass, 0, 0);
          MODULE_DEPEND(snd_xxxpci, snd_pcm, PCM_MINVER, -
  PCM_PREFVER,PCM_MAXVER);
  ```

  大多数声音驱动程序需要存储关于其设备的附加私有信息。私有数据 结构通常在连接例程中分配。其地址通过调用 pcm_register()和 mixer_init()传递给 pcm。后面pcm 将此地址作为调用声音驱动程序接口时的参数传递回来。

- 声音驱动程序的连接例程应当通过调用mixer_init() 向pcm声明它的MIXER或AC97接口。对于MIXER接口，这会接着引起调用 xxxmixer_init()。

- 声音驱动程序的连接例程通过调用 pcm_register(dev, sc, nplay, nrec) 向pcm声明其通用CHANNEL配置，其中 sc是设备数据结构的地址， 在pcm以后的调用中将会用到它， nplay和nrec是播放和录音 通道的数目。

- 声音驱动程序的连接例程通过调用 pcm_addchan()声明它的每个通道对象。这会在 pcm中建立起通道合成，并接着会引起调用 xxxchannel_init() （译注：请参考原文）。

- 声音驱动程序的分离例程在释放其资源之前应当调用 pcm_unregister()。

有两种可能的方法来处理非PnP设备：

- 使用device_identify()方法 （范例：sound/isa/es1888.c）。 device_identify()方法在已知地址探测硬件， 如果发现支持的设备就会创建一个新的pcm设备，这个pcm设备接着 会被传递到probe/attach。

- 使用定制内核配置的方法，为pcm设备设置适当的hints（范例： sound/isa/mss.c）。

pcm驱动程序应当实现 device_suspend， device_resume和 device_shutdown例程，这样电源管理和模块卸载就能 正确地发挥作用。

15.4. 接口

pcm核心与声音驱动程序之间的接口以术语 内核对象的叫法来定义。

声音驱动程序通常提供两种主要的接口： CHANNEL以及 MIXER或AC97。

AC97是一个很小的硬件访问（寄存器读/写） 接口，由驱动程序为带AC97编码解码器的硬件来实现。这种情况下，实际的 MIXER接口由pcm中共享的AC97代码提供。

15.4.1. CHANNEL接口

15.4.1.1. 函数参数的通常注意事项

声音驱动程序通常用一个私有数据结构来描述他们的设备，驱动 程序所支持的播放和录音数据通道各有一个。

对于所有的CHANNEL接口函数，第一个参数是一个不透明的指针。

第二个参数是指向私有的通道数据结构的指针， channel_init()是个例外，它的指针指向私有 设备结构（并返回由pcm以后使用的通道指针）。

15.4.1.2. 数据传输操作概览

对于声音数据传输，pcm核心与声音驱动 程序是通过一个由struct snd_dbuf描述的共享内存区域进行通信的。

struct snd_dbuf是 pcm私有的，声音驱动程序通过调用访问者 函数（sndbuf_getxxx()）来获得感兴趣的值。

共享内存区域的大小等于 sndbuf_getsize()，并被分割为大小固定，且等于sndbuf_getblksz()字节的很多块。

当播放时，常规的传输机制如下（将意思反过来就是录音）：

- pcm开始时填充缓冲区，然后以 参数PCMTRIG_START调用声音驱动程序的 xxxchannel_trigger() 。
- 声音驱动程序接着安排以 sndbuf_getblksz()字节大小为块，重复将 整个内存区域（sndbuf_getbuf()， sndbuf_getsize()）传输到设备。对于每个 传输块回调pcm函数 chn_intr()（这通常在中断时间发生）。
- chn_intr()安排将新数据拷贝到那些 数据已传输到设备（现在空闲）的区域，并对 snd_dbuf结构进行适当的更新。

15.4.1.3. channel_init

调用xxxchannel_init()来初始化每个播放 和录音通道。这个调用从声音驱动程序的连接例程中发起。（参看 探测和连接一节）。

```
static void *
xxxchannel_init(kobj_t obj, void *data,
  struct snd_dbuf *b, struct pcm_channel *c, int dir)❶
{
   struct xxx_info *sc = data;
   struct xxx_chinfo *ch;
    -...
   return ch;❷
 -}
```

❶ b为通道 struct snd_dbuf的地址。它应当在 函数中通过调用sndbuf_alloc()来初始化。 所用的缓冲区大小通常是设备'典型'传输大小的一个较小的倍数。

c为 pcm通道控制结构的指针。这是个不透明 指针。函数应当将它保存到局部通道结构中，在后面调用 pcm函数（例如： chn_intr(c)）时会使用它。

dir指示通道方向 （PCMDIR_PLAY或 PCMDIR_REC）。

❷ 函数应当返回一个指针，此指针指向用于控制此通道的私有 区域。它将作为参数被传递到对其他通道接口的调用。

15.4.1.4. channel_setformat

xxxchannel_setformat()应当按特定通道， 特定声音格式设置硬件。

```
static int
xxxchannel_setformat(kobj_t obj, void *data, u_int32_t format)❶
{
   struct xxx_chinfo *ch = data;
    -...
   return 0;
 -}
```

❶ format为 AFMT_XXX value值之一 （soundcard.h）。

15.4.1.5. channel_setspeed

xxxchannel_setspeed()按指定的取样速度 设置通道硬件，并返回返回可能调整过的速度。

```
static int
xxxchannel_setspeed(kobj_t obj, void *data, u_int32_t speed)
{
   struct xxx_chinfo *ch = data;
    -...
   return speed;
 -}
```

15.4.1.6. channel_setblocksize

xxxchannel_setblocksize()设置块大小， 这是pcm与声音驱动程序，以及声音驱动 程序与设备之间的传输单位的大小。传输期间，每次传输这样大小的 数据后，声音驱动程序都应当调用pcm的 chn_intr()。

大多数驱动程序只注意这儿的块大小，因为当实际传输开始时应该 使用这个值。

```
static int
xxxchannel_setblocksize(kobj_t obj, void *data, u_int32_t blocksize)
{
   struct xxx_chinfo *ch = data;
     -...
   return blocksize;❶
 -}
```

❶ 函数返回可能调整过的块大小。如果块大小真的变化了， 这种情况下应当调用sndbuf_resize()调整 缓冲区的大小。

15.4.1.7. channel_trigger

xxxchannel_trigger()由 pcm来控制驱动程序中的实际传输操作。

```
static int
xxxchannel_trigger(kobj_t obj, void *data, int go)❶
{
   struct xxx_chinfo *ch = data;
    -...
   return 0;
 -}
```

❶ go定义当前调用的动作。可能的值为：

- PCMTRIG_START：驱动程序应当 启动从/到通道缓冲区的数据传输。如果需要，应当通过 sndbuf_getbuf()和 sndbuf_getsize()检取缓冲区的 基地址和大小。
- PCMTRIG_EMLDMAWR / PCMTRIG_EMLDMARD：告诉驱动程序， 输入或输出缓冲区可能已被更新过了。大多数驱动程序只是 忽略这些调用。
- PCMTRIG_STOP / PCMTRIG_ABORT：驱动程序应当停止当前 的传输。

> **注意**
>
> 如果驱动程序使用ISA DMA，则应当在设备上执行动作前 调用sndbuf_isadma()，并处理DMA芯片一方的 事情。

15.4.1.8. channel_getptr

xxxchannel_getptr()返回传输缓冲区中 当前的缓冲。它通常由chn_intr()调用，而且这也是为什么pcm知道它应当往哪儿传送 新数据。

15.4.1.9. channel_free

调用xxxchannel_free()来释放通道资源， 例如当驱动程序卸载时，并且如果通道数据结构是动态分配的，或者 如果不使用sndbuf_alloc()进行缓冲区分配， 则应当实现这个函数。

15.4.1.10. channel_getcaps

```
struct pcmchan_caps *
xxxchannel_getcaps(kobj_t obj, void *data)
{
   return &xxx_caps;❶
-}
```

❶ 这个例程返回指向（通常静态定义的） pcmchan_caps结构的指针（在 sound/pcm/channel.h中定义）。这个结构 保存着最小和最大采样频率和被接受的声音格式。任何声音驱动 程序都可以作为一个范例。

15.4.1.11. 更多函数

channel_reset()，channel_resetdone()和 channel_notify()用于特殊目的，未与权威人士(Cameron Grant)进行探讨之前不应当在驱动程序中实现它。

不赞成使用channel_setdir()。

15.4.2. MIXER接口

15.4.2.1. mixer_init

xxxmixer_init()初始化硬件，并告诉 pcm什么混音器设备可用来播放和录音。

```
static int
xxxmixer_init(struct snd_mixer *m)
{
```

```
    struct xxx_info   *sc = mix_getdevinfo(m);
    u_int32_t v;

    [#####]

    [#######v#####]❶
    mix_setdevs(m, v);
    [#######v#####]
    mix_setrecdevs(m, v)

    return 0;
-}
```

❶ 设置一个整数值中的位，并调用 mix_setdevs()和 mix_setrecdevs()来告诉pcm存在什么设备。

混音器的位定义可以在soundcard.h中 找到。（SOUND_MASK_XXX值和SOUND_MIXER_XXX移位）。

15.4.2.2. mixer_set

xxxmixer_set()为混音器设备设置音量级别（1eve1）。

```
static int
xxxmixer_set(struct snd_mixer *m, unsigned dev,
             unsigned left, unsigned right)❶
{
    struct sc_info *sc = mix_getdevinfo(m);
    [######(level)]
    return left -| (right << 8);❷
-}
```

❶ 设备被指定为 SOUND_MIXER_XXX 值

在范围[0–100]之间指定音量值。零值应当让设备静音。

❷ 由于硬件(音量)级别(1eve1)可能不匹配输入比例，会出现 某些圆整，例程返回如上面所示的实际级别值（范围0–100内）。

15.4.2.3. mixer_setrecsrc

xxxmixer_setrecsrc()设定录音源设备。

```
static int
xxxmixer_setrecsrc(struct snd_mixer *m, u_int32_t src)❶
{
    struct xxx_info *sc = mix_getdevinfo(m);

    [##src#####, ####]

    [##src######]
```

```
        return src;❷
-}
```

❶ 期望的录音设备由一个位域指定.
❷ 返回设置用来录音的实际设备。一些驱动程序只能设置一个 录音设备。如果出现错误，函数应当返回-1。

15.4.2.4. mixer_uninit，mixer_reinit

xxxmixer_uninit()应当确保不会发出任何 声音，并且如果可能则应当让混音器硬件断电。

xxxmixer_reinit()应当确保混音器硬件 加电，并且恢复所有不受mixer_set()或mixer_setrecsrc()控制的设置。

15.4.3. AC97接口

AC97由带有AC97编码解码器的驱动程序实现。 它只有三个方法:

- xxxac97_init()返回找到的 ac97编码解码器的数目。
- ac97_read()与 ac97_write()读写指定的寄存器。

The AC97接口由 pcm中的AC97代码来执行高层操作。参看 sound/pci/maestro3.c或sound/pci/下很多其他内容作为范例。

第 16 章 PC Card

翻译：.

本章将讨论FreeBSD为编写PC Card或CardBus设备的驱动程序而提供的机制。 但目前本文只记录了如何向现有的pccard驱动程序中添加驱动程序。

16.1. 添加设备

向所支持的pccard设备列表中添加新设备的步骤已经与系统在FreeBSD 4 中使用的方法不同了。在以前的版本中，需要编辑 /etc中的一个文件来列出设备。从FreeBSD 5.0开始， 设备驱动程序知道它们支持什么设备。现在内核中有一个受支持设备的表， 驱动程序用它来连接设备。

16.1.1. 概览

可以有两种方法来识别PC Card，他们都基于卡上的 CIS信息。第一种方法是使用制造商和产品的数字编号。 第二种方法是使用人可读的字符串，字符串也是包含在CIS中。PC Card总线 使用集中式数据库和一些宏来提供一个易用的设计模式，让驱动程序的编写 者很容易地确定匹配其驱动程序的设备。

一个很普遍的实际情况是，某个公司为一款PC Card产品开发出参考 设计，然后把这个设计卖给另外的公司，以便在市场上出售。那些公司改进 原设计，把向他们的目标客户群或地理区域出售产品，并将他们自己的名字 放到卡中。然而所谓的对现有卡的改进，即使做过任何修改，这些修改通常 也微乎其微。然而，为了强化他们自己版本的品牌，这些供货商常常会把他们 公司的名字放入CIS空间的可读字符串中，却不会改动制造商和产品的ID。

鉴于以上情况，对于FreeBSD来说使用数字ID可以减小工作量。同时也 会减小将ID加入到系统的过程中所带来的复杂性。必须仔细检查谁是卡的 真正制造者，尤其当提供原卡的供货商在中心数据库中已经有一个不同的ID 时。Linksys，D-Link和NetGear是经常出售相同设计的几个美国制造商。 相同的设计可能在日本以诸如Buffalo和Corega的名字出售。然而，这些 设备常常具有相同的制造商和产品ID。

PC Card总线在其中心数据库 /sys/dev/pccard/pccarddevs中保存了卡的信息， 但不包含哪个驱动程序与它们关联的信息。它也提供了一套宏，以允许在 驱动程序用来声明设备的表中容易地创建简单条目。

最后，某些非常低端的设备根本不包含制造商标识。这些设备需要使用 可读CIS字符串来匹配它们。如果我们不需要这种应急办法该有多好，但对于 某些非常低端却非常流行的CD-ROM播放器来说却是必需的。通常应当避免 使用这种方法，但本节中还是列出了很多设备，因为它们是在认识到PC Card商业的OEM本质之前加入的，应当优先使用 数字方法。

16.1.2. pccarddevs的格式

pccarddevs文件有四节。第一节为使用 它们的那些供货商列出了制造商号码。本节按数字排序。下一节包含了 这些供货商使用的所有产品，包括他们的产品ID号码和描述字符串。 描述字符串通常不会被使用（相反，即使我们可以匹配数字版本号，我们 仍然基于人可读的CIS设置设备的描述）。然后为使用字符串匹配方法的那些设备重复这两节的东西。最后，文件任何地方可以使用C风格的注释。

文件的第一节包含供货商ID。请保持列表按数字排序。此外，为了 能有一个通用清晰的保存地来方便地保存这些信息，我们与NetBSD共享此 文件，因此请协调对此文件的任何更改。例如：

```
vendor FUJITSU   0x0004  Fujitsu Corporation
vendor NETGEAR_2  0x000b  Netgear
vendor PANASONIC  0x0032 Matsushita Electric Industrial Co.
vendor SANDISK   0x0045 Sandisk Corporation
```

显示了几个供货商ID。很凑巧的是NETGEAR_2 实际上是NETGEAR从其购买卡的OEM，对那些卡提供支持的作者那时并不知道 NETgear使用的是别人的ID。这些条目相当直接易懂。每行上都有供货商 关键字来指示本行的类别。也有供货商的名字。名字将会在pccarddevs文件 的后面重复出现，名字也会用在驱动程序的匹配表中，因此保持它的短小 并且是有效的C标识符。还有一个给供货商的十六进制数字ID。不要添加 0xffffffff或0xffff形式的ID， 因为它们是保留ID（前者是'空ID集合'，而后者有时会在质量极其差的卡中 看到，用来指示none）。最后还有关于制卡公司的描述字符串。这个字符串 在FreeBSD中除了用于注释目的外并没有被使用过。

文件的第二节包含产品．如你在下面例子中看到的：

```
/* Allied Telesis K.K. */
product ALLIEDTELESIS LA_PCM 0x0002 Allied Telesis LA-PCM

/* Archos */
product ARCHOS ARC_ATAPI 0x0043 MiniCD
```

格式与供货商的那些行相似。其中有产品关键字。然后是供货商名字， 由上面重复而来。后面跟着产品名字，此名字在驱动程序中使用，且应当 是一个有效C标识符，但可以以数字开头。然后是卡的十六进制产品ID。 供货商通常对0xffffffff和0xffff有相同的约定。最后是关于设备自身的字符串 描述。由于FreeBSD的pccard总线驱动程序会从人可读的CIS条目创建一个 字符串，因此这个字符串在FreeBSD中通常不被使用，但某些CIS条目不能 满足要求的情况下还可能使用。产品按制造商的字母顺序排序，然后再按 产品ID的数字排序。每个制造商条目前有一条C注释，条目之间有一个空行。

第三节很象前面的供货商一节，但所由的制造商ID为 -1。-1在FreeBSD pccard总线代码中意味着“匹配发现的任何东西”。由于它们是C标识符， 它们的名字必须唯一。除此之外格式等同于文件的第一节。

最后一节包含那些必须用字符串匹配的卡。这一节的格式与通用 节的格式有点不同：

```
product ADDTRON AWP100
 { -"Addtron", -"AWP-100&spWireless&spPCMCIA", -"Version&sp01.02", NULL -}
product ALLIEDTELESIS WR211PCM { -"Allied&spTelesis&spK.K.", -"WR211PCM", NULL, -
NULL -} Allied Telesis WR211PCM
```

我们已经熟悉了产品关键字，后跟供货商名字，然后再跟卡的名字， 就象在文件第二节中那样。然而，这之后就与那格式不同了。有一个 {}分组，后跟几个字符串。这些字符串对应CIS_INFO三元组中定义的 供货商，产品和额外信息。这些字符串被产生 pccarddevs.h的程序过滤，将 &sp替换为 实际的空格。空条目意味着条目的这部分应当被忽略。在我选择的例子中 有一个错误的条目。除非对卡的操作来说至关重要，否则不应当在其中 包含版本号。有时供货商在这个字段中会有卡的很多不同版本，这些版本 都能工作，这种情况下那些信息只会让那些拥有相似卡的人在FreeBSD中 更难以使用。有时当供货商出于市场考虑（可用性，价格等等），希望出售 同一品牌下的很多不同部分时，这也是有必要的。如果这样，则在那些 供货商仍然保持相同的制造商/产品对的少见情况下，能否区分开卡至关重要．此时不能使用正则表达式匹配。

16.1.3. 探测例程样例

要懂得如何向所支持的设备列表中添加设备，就必须懂得很多驱动程序 都有的探测和/或匹配例程。由于也为老卡提供了一个兼容层，这在 FreeBSD 5.x中有一点复杂。由于只是window-dressing不同，这儿给出了 一个理想化的版本。

```
static const struct pccard_product wi_pccard_products[] = {
 PCMCIA_CARD(3COM, 3CRWE737A, 0),
 PCMCIA_CARD(BUFFALO, WLI_PCM_S11, 0),
 PCMCIA_CARD(BUFFALO, WLI_CF_S11G, 0),
 PCMCIA_CARD(TDK, LAK_CD011WL, 0),
 { NULL -}
};

static int
wi_pccard_probe(dev)
 device_t dev;
{
 const struct pccard_product *pp;

 if ((pp = pccard_product_lookup(dev, wi_pccard_products,
     sizeof(wi_pccard_products[0]), NULL)) -!= NULL) {
  if (pp->pp_name -!= NULL)
   device_set_desc(dev, pp->pp_name);
  return (0);
 }
 return (ENXIO);
}
```

这儿我们有一个可以匹配少数几个设备的简单pccard探测例程。如上面 所提到，名字可能不同（如果不是 foo_pccard_probe()则就是 foo_pccard_match()）。函数pccard_product_lookup()是一个通用函数，它遍历 表并返回指向它所匹配的第一项的指针。一些驱动程序可能使用这个机制来 将某些卡的附加信息传递到驱动程序的其它部分，因此表中可能有些变体。 唯一的要求就是如果你有一个不同的表，则让表的结构的第一个元素为 结构pccard_product。

观察一下表wi_pccard_products就会发现， 所有条目都是 PCMCIA_CARD(*foo*# *bar*# *baz*)的形式。 *foo*部分为来自 pccarddevs的制造商ID。 *bar*部分为产品。 *baz*为此卡所期望的功能号。许多pccards 可以有多个功能，需要有办法区分开功能1和功能0。你可以看一下 PCMCIA_CARD_D，它包括了来自 pccarddevs文件的设备描述。你也可以看看 PCMCIA_CARD2和 PCMCIA_CARD2_D，当你需要按 “使用默认描述” 和 “从pccarddevs中取得” 做法，同时匹配CIS字符串和制造商号码时就会用到它们。

16.1.4. 将它合在一起

因此，为了一个增加新设备，必须进行下面步骤。首先，必须从设备 获得标识信息。完成这个最容易的方法就是将设备插入到PC Card或CF槽中， 并发出devinfo -v。你可能会看到一些类似下面的 东西：

```
      cbb1 pnpinfo vendor=0x104c device=0xac51 subvendor=0x1265 subdevice=0x0300 -
class=0x060700 at slot=10 function=1
        cardbus1
        pccard1
          unknown pnpinfo manufacturer=0x026f product=0x030c cisvendor="BUFFALO" -
cisproduct="WLI2-CF-S11" function_type=6 at function=0
```

作为输出的一部分。制造商和产品为产品的数字ID。而cisvender和 cisproduct为CIS中提供的描述本产品的字符串。

由于我们首先想优先使用数字选项，因此首先尝试创建基于此的条目。 为了示例，上面的卡已经被稍稍虚构化了。我们看到的供货商为BUFFALO， 它已经有一个条目了：

```
vendor BUFFALO   0x026f BUFFALO (Melco Corporation)
```

这样我们就可以了。为这个卡查找一个条目，但我们没有发现。但我们 发现：

```
/* BUFFALO */
product BUFFALO WLI_PCM_S11 0x0305 BUFFALO AirStation 11Mbps WLAN
product BUFFALO LPC_CF_CLT 0x0307 BUFFALO LPC-CF-CLT
product BUFFALO LPC3_CLT 0x030a BUFFALO LPC3-CLT Ethernet Adapter
product BUFFALO WLI_CF_S11G 0x030b BUFFALO AirStation 11Mbps CF WLAN
```

我们就可以向pccarddevs中添加：

```
product BUFFALO WLI2_CF_S11G 0x030c BUFFALO AirStation ultra 802.11b CF
```

目前，需要一个手动步骤来 重新产生pccarddevs.h，用来将这些标识符转换 到客户驱动程序。你在驱动程序中使用它们之前必须完成下面步骤：

```
# cd src/sys/dev/pccard
# make --f Makefile.pccarddevs
```

一旦完成了这些步骤，你就可以向驱动程序中添加卡了。这只是一个 添加一行的简单操作：

```
static const struct pccard_product wi_pccard_products[] = {
 PCMCIA_CARD(3COM, 3CRWE737A, 0),
 PCMCIA_CARD(BUFFALO, WLI_PCM_S11, 0),
 PCMCIA_CARD(BUFFALO, WLI_CF_S11G, 0),
+ PCMCIA_CARD(BUFFALO, WLI_CF2_S11G, 0),
 PCMCIA_CARD(TDK, LAK_CD011WL, 0),
 { NULL -}
};
```

注意，我在我添加的行前面包含了'+'，但这只是 用来强调这一行。不要把它添加到实际驱动程序中。一旦你添加了这行，就 可以重新编译内核或模块，并试着看它是否能识别设备。如果它识别出设备 并能工作，请提交补丁。如果它不工作，请找出让它工作所需要的东西并 提交一个补丁。如果它根本不识别设备，那么你可能做错了什么，应当重新 检查每一步。

如果你是一个FreeBSD源代码的committer，并且所有东西看起来都 正常工作，则你应当把这些改变提交到树中。然而有些小技巧的东西你 需要考虑。首先，你必须提交pccarddevs文件到 树中。完成后，你必须重新产生pccarddevs.h 并将它作为另一次提交来提交（这是为了确保正确的 $FreeBSD$标签会留在后面的文件中）。最后，你需要把 其它东西提交到驱动程序。

16.1.5. 提交新设备

很多人直接把新设备的条目发送给作者。请不要那样做。请将它们作为 PR来提交，并将PR号码发送给作者用于记录。这样确保条目不会丢失。提交 PR时，补丁中没有必要包含pccardevs.h的diff， 因为那些东西可以重新产生。包含设备的描述和客户驱动程序的补丁是必要 的。如果你不知道名字，使用OEM99作为名字，作者将会调查后相应地调整 OEM99。提交者不应当提交OEM99，而应该找到最高的OEM条目并提交高于那个 的一个。

部分 III. 附录

目录

参考书目

[1] Marshall Kirk McKusick、Keith Bostic、Michael J Karels和John S Quarterman. 版权 © 1996 Addison-Wesley Publishing Company, Inc.. 0-201-54979-4. Addison-Wesley Publishing Company, Inc.. The Design and Implementation of the 4.4 BSD Operating System. 1-2.

索引