TURING 图灵程序设计丛书 移动开发系列

Apress®

- 唯一专注XNU内核开发实践的图书
- 一线设备驱动程序开发人员现身说法
- 轻松理解内核开发的晦涩难懂之处

OS X and i gramming

# OS X与iOS内核编程

[澳] Ole Henry Halvorsen Douglas Clarke 著
贾伟 译

人民邮电出版社
POSTS & TELECOM PRESS

亚马逊读者评论

“目前市面上介绍XNU内核开发实践的书非常少见，本书刚好可以填补这一空白。I/O Kit驱动程序开发人员应该人手一册。”

OS X and iOS Kernel Programming

# OS X与iOS内核编程

与传统的用户应用程序开发不同，内核环境更加复杂多变，任何问题都会严重影响系统的稳定性、安全性和性能，所以一定要确保内核代码中没有任何错误。本书介绍了非常实用的内核开发方法，是XNU内核开发者必备的“九阴真经”。

本书两位作者都活跃在计算机软硬件以及驱动程序开发一线，具有丰富的实战经验。Ole Henry Halvorsen是最早一批为USB 3.0和Thunderbolt创建视频硬件设备的开发人员。Douglas Clarke具有15年以上的麦金塔计算机专业软硬件开发经验。本书汇集了两位作者多年的实战经验与智慧感悟，展现了一幅XNU内核编程与设备驱动程序开发的斑斓画卷。相信在作者循循善诱、通俗易懂的讲解下，读者定能为书中的精彩内容深深吸引，渐入化境，在不知不觉中炼就内核开发的绝世武功。

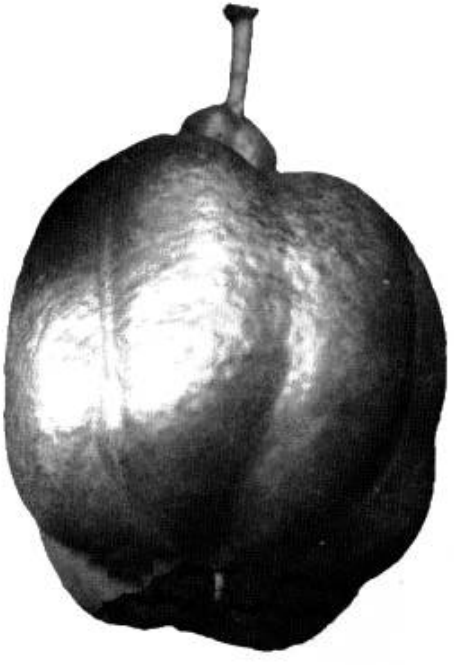
如果你对iOS和Mac OS X操作系统感兴趣，关注内核开发，或者你本身就是驱动程序开发人员，那么如此精彩的一本好书绝对不容错过。

Apress®

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐邮箱：contact@turingbook.com
热线：(010)51095186转604
分类建议 计算机/移动开发
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-31824-4
定价：89.00元

TURING 图灵程序设计丛书 移动开发系列



OS X and iOS Kernel Programming

# OS X与iOS内核编程

[澳] Ole Henry Halvorsen Douglas Clarke 著
贾伟 译

人民邮电出版社
北京

**图书在版编目（CIP）数据**

OS X与iOS内核编程 / （澳）哈尔沃森（Halvorsen, O. H.），（澳）克拉克（Clarke, D.）著；贾伟译. -- 北京：人民邮电出版社，2013.6
（图灵程序设计丛书）
书名原文：OS X and iOS kernel programming
ISBN 978-7-115-31824-4

Ⅰ. ①O… Ⅱ. ①哈… ②克… ③贾… Ⅲ. ①操作系统－程序设计 Ⅳ. ①TP316

中国版本图书馆CIP数据核字(2013)第095050号

## 内 容 提 要

本书面向 Mac OS X 和 iOS 操作系统，介绍了操作系统和内核架构等基础知识，以及内存管理、线程同步、I/O Kit 框架等基本概念。通过最贴近实战的方法帮助读者编写高效的内核级代码。本书两位作者具备丰富的计算机硬软件以及设备驱动程序开发经验，在他们的指引下，读者定能为 USB 和 Thunderbolt 等设备开发设备驱动程序。

如果你对 iOS 和 Mac OS X 操作系统感兴趣，关注内核开发实践，或者你是驱动程序开发人员，那么本书适合你阅读。

◆ 著 [澳] Ole Henry Halvorsen Douglas Clarke
译 贾 伟
责任编辑 刘美英
执行编辑 张 敏 李 静
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：25
字数：585千字 2013年6月第1版
印数：1－3 500册 2013年6月北京第1次印刷
著作权合同登记号 图字：01-2012-1813号

定价：89.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第 0021 号

# 版权声明

Original English language edition, entitled *OS X and iOS Kernel Programming* by Ole Henry Halvorsen, Douglas Clarke, published by Apress, 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705 USA.

Copyright © 2012 by Ole Henry Halvorsen, Douglas Clarke. Simplified Chinese-language edition copyright © 2013 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由Apress L.P.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

# 目　录


第 1 章　操作系统原理 ……1

1.1　操作系统的作用 ……3
1.2　进程管理 ……3
1.3　进程地址空间 ……4
1.4　操作系统服务 ……5
1.5　虚拟内存 ……6
1.6　调度 ……8
1.7　硬件和驱动程序 ……9
1.8　小结 ……11

第 2 章　Mac OS X 和 iOS ……12

2.1　XNU 内核 ……17
2.1.1　内核扩展（KEXT） ……18
2.1.2　Mach ……18
2.1.3　BSD 层 ……27
2.1.4　I/O Kit ……29
2.2　小结 ……31

第 3 章　Xcode 和内核开发环境 ……32

3.1　语言的选择：C++ ……32
3.2　Xcode ……33
3.3　“Hello World”内核扩展 ……34
3.4　加载和卸载内核扩展 ……37
3.5　使用控制台查看输出 ……39
3.6　小结 ……40

第 4 章　I/O Kit 框架 ……41

4.1　I/O Kit 模型 ……41
4.2　对象关系 ……43
4.3　Info.plist 文件 ……43
4.3.1　驱动程序类 ……46
4.3.2　IORegistryExplorer ……50
4.4　内核库：libkern ……52
4.4.1　OSObject ……52
4.4.2　容器类 ……54
4.5　小结 ……56

第 5 章　应用程序与驱动程序的交互 ……57

5.1　I/O Kit 框架 ……58
5.2　查找驱动程序 ……58
5.3　观察设备移除 ……63
5.4　修改设备驱动程序属性 ……65
5.5　基于状态的交互 ……68
5.6　驱动程序的通知 ……79
5.7　小结 ……83

第 6 章　内存管理 ……84

6.1　内存类型 ……84
6.1.1　CPU 物理地址 ……85
6.1.2　总线物理地址 ……85
6.1.3　用户和内核虚拟地址 ……85
6.1.4　内存顺序：大端序和小端序 ……86
6.1.5　32 位和 64 位内存寻址 ……87
6.2　内存分配 ……88
6.2.1　底层分配机制 ……89
6.2.2　Mach 内存域分配器 ……89
6.2.3　kalloc 家族 ……89
6.2.4　BSD 中的内存分配 ……90
6.2.5　I/O Kit 内存分配 ……91
6.2.6　使用 C++的 new 操作符分配内存 ……92
6.3　内存描述符 ……92
6.3.1　IOBufferMemoryDescriptor ……94

6.3.2 其他的内存描述符……95
6.4 映射内存……95
6.4.1 用户空间任务到内核空间的内存映射……95
6.4.2 IOMemoryMap 类……97
6.4.3 内核到用户空间任务的内存映射……98
6.4.4 将内存映射到指定的用户空间任务……99
6.4.5 物理地址映射……100
6.5 小结……101

第 7 章 同步和线程……102
7.1 同步原语……102
7.2 原子操作……104
7.3 锁……107
7.3.1 自旋锁……107
7.3.2 互斥……109
7.3.3 条件变量……110
7.3.4 读/写互斥……112
7.4 同步异步事件：工作环……113
7.4.1 IOCommandGate……115
7.4.2 定时器……116
7.4.3 释放工作环……117
7.5 内核线程……118
7.6 小结……119

第 8 章 USB……120
8.1 USB 体系结构……120
8.1.1 USB 传输速度……122
8.1.2 主机控制器……123
8.1.3 USB 协议……124
8.1.4 端点……126
8.1.5 USB 描述符……126
8.1.6 USB 设备类……127
8.2 I/O Kit USB 支持……128
8.2.1 USB 设备和驱动程序处理……129
8.2.2 加载 USB 驱动程序……130
8.2.3 USB Prober……132
8.2.4 驱动程序示例：USB 大容量存储器设备驱动程序……133
8.2.5 驱动程序启动……138
8.2.6 处理设备移除……138
8.2.7 枚举接口……139
8.2.8 枚举端点……140
8.2.9 执行设备请求……141
8.2.10 执行批量端点和中断端点的 I/O……144
8.3 小结……147

第 9 章 PCI Express 和 Thunderbolt……148
9.1 I/O Kit 中的 PCI……150
9.1.1 匹配和加载驱动程序……151
9.1.2 驱动程序示例：一个简单的 PCI 驱动程序……154
9.1.3 访问配置空间寄存器……156
9.1.4 PCI I/O 内存区……158
9.1.5 处理设备移除……161
9.2 中断……162
9.2.1 I/O Kit 中断机制……163
9.2.2 注册接收中断……164
9.2.3 启用 MSI……166
9.2.4 处理主中断……166
9.2.5 处理二级中断……168
9.3 直接内存访问……168
9.3.1 将物理地址转换为总线地址……170
9.3.2 为 DMA 准备内存……171
9.3.3 建立分散/聚集列表……172
9.3.4 IODMACommand 类……173
9.4 小结……175

第 10 章 电源管理……177
10.1 响应电源状态改变……179
10.2 请求电源状态改变……184
10.3 处理设备空闲……185
10.4 观察设备电源状态改变……186
10.5 汇总……187
10.6 小结……191

第 11 章 串行端口驱动程序……192
11.1 Mac OS X 串行端口体系结构概览……192
11.2 串行端口驱动程序……194

11.3 实现 IOSerialDriverSync 类 ……197
11.4 串行端口状态 ……200
11.5 串行端口事件 ……204
11.6 串行数据传输 ……207
11.7 从用户空间访问串行端口 ……211
11.8 小结 ……214

第 12 章 音频驱动程序 ……215
12.1 数字音频和音频设备简介 ……215
12.2 Core Audio ……217
12.3 I/O Kit 音频支持 ……218
12.4 实现一个音频驱动程序 ……219
12.4.1 驱动程序和硬件初始化 ……221
12.4.2 注册音频控制 ……223
12.4.3 实现音频引擎 ……225
12.4.4 I/O 引擎初始化 ……226
12.4.5 其他的音频引擎功能 ……235
12.5 小结 ……236

第 13 章 网络 ……238
13.1 网络内核扩展 ……242
13.1.1 内核控制 KPI ……242
13.1.2 套接字过滤器 ……242
13.1.3 因特网协议过滤器 ……250
13.1.4 接口过滤器 ……255
13.2 调试和测试网络扩展 ……258
13.3 I/O Kit 中的网络 ……259
13.3.1 构建简单以太网控制器驱动程序 ……261
13.3.2 MyEthernetDriver 设计 ……262
13.3.3 驱动程序初始化和启动 ……264
13.3.4 介质和状态选择 ……266
13.3.5 配置设备硬件地址 ……268
13.3.6 启用和禁用设备 ……268
13.3.7 传输网络分组 ……270
13.3.8 接收分组 ……271
13.3.9 对 MyEthernetDriver 进行测试 ……274
13.4 小结 ……276

第 14 章 存储系统 ……277
14.1 传输层驱动程序 ……278
14.2 IOBlockStorageDevice 接口 ……279
14.3 构建 RAM 磁盘设备 ……282
14.4 分区规则 ……292
14.4.1 实现一个示例分区规则 ……293
14.4.2 媒介内容线索属性 ……300
14.5 媒介过滤器驱动程序 ……300
14.5.1 加密过滤器规则示例 ……302
14.5.2 创建一个自定义 GUID 分区表 ……306
14.6 小结 ……308

第 15 章 用户空间 USB 驱动程序 ……310
15.1 背景 ……310
15.2 IOUSBLib 框架 ……311
15.3 处理异步操作 ……315
15.4 IOUSBDeviceInterface 类 ……316
15.5 IOUSBInterfaceInterface 类 ……320
15.5.1 属性方法 ……321
15.5.2 端点数据传输方法 ……322
15.5.3 低延迟同步传输 ……328
15.6 小结 ……330

第 16 章 调试 ……331
16.1 常见的问题类型 ……331
16.2 内核恐慌 ……332
16.3 调试机制 ……333
16.3.1 启动时修复崩溃 ……335
16.3.2 采用 IOLog()追踪 ……335
16.3.3 输出栈跟踪 ……336
16.3.4 通过 FireWire 远程跟踪 ……337
16.3.5 远程内核核心转储 ……339
16.3.6 KDB ……340
16.3.7 通过以太网或 FireWire 使用 GDB 远程调试 ……340
16.3.8 实时调试运行的内核 ……344
16.3.9 使用虚拟机调试 ……344
16.3.10 在内核中使用 GDB 调试 ……344

16.3.11 使用 Activity Monitor 诊断挂起进程 ····· 354
16.3.12 查找内存和资源泄漏 ····· 355
16.4 小结 ····· 356

第 17 章 高级内核编程 ····· 357

17.1 内核中的 SSE 和浮点 ····· 357
17.2 多功能驱动程序 ····· 358
17.3 编写 I/O Kit 族 ····· 358
17.4 内核控制 KPI ····· 359
17.4.1 内核控制注册 ····· 361
17.4.2 客户端连接 ····· 362
17.4.3 获取和设置选项 ····· 363
17.4.4 从用户空间访问内核控制 ····· 364
17.5 内核中的进程处理 ····· 365
17.6 加载资源 ····· 366
17.7 KEXT 资源之外的内容 ····· 367
17.8 内核驱动程序通知 ····· 368
17.9 小结 ····· 371

第 18 章 部署 ····· 372

18.1 安装和加载内核扩展 ····· 372
18.2 加载首选项和设置 ····· 374
18.3 内核扩展的版本管理 ····· 375
18.4 测试和品质保证 ····· 375
18.5 打包 KEXT 和软件 ····· 376
18.5.1 构建 Hello World 内核扩展安装包 ····· 378
18.5.2 将内容添加至安装包 ····· 378
18.5.3 配置安装包 ····· 379
18.5.4 构建安装包 ····· 382
18.5.5 卸载安装包 ····· 383
18.6 小结 ····· 383

索引 ····· 385

# 第1章 操作系统原理

操作系统的作用是为用户提供应用软件的运行环境。用户运行应用程序时所执行的任务依赖于操作系统提供的服务，然而，操作系统执行任务时，许多情况下都无需用户甚至程序员的干预。例如，应用程序从硬盘读取文件时，程序员只需调用操作系统提供的函数即可，执行读取的特定步骤完全交由操作系统处理。程序员不用了解从计算机硬盘中读取文件与从外部USB闪存中读取文件有什么不同，这是操作系统关心的事情。

绝大多数程序员都开发过很多用户使用的代码，他们可能还会使用框架（如Cocoa）提供图形化用户界面与用户进行交互。Mac或iPhone应用商店提供的所有应用程序都属于这一类。本书讲述的不是应用软件开发，而是内核扩展开发，即编写为应用程序提供服务的代码。下列两种情况非常适合进行内核扩展：操作系统与自定义的硬件设备协同工作时，以及添加新文件系统的支持时。例如，经过内核扩展后，iTunes就可以使用新的USB音频设备，以太网卡就可以为网络应用程序提供接口，如图1-1所示。对文件系统进行内核扩展，就可以将在Windows计算机上格式化后的硬盘装载在Mac机上，使其看起来像一个标准的Mac驱动。

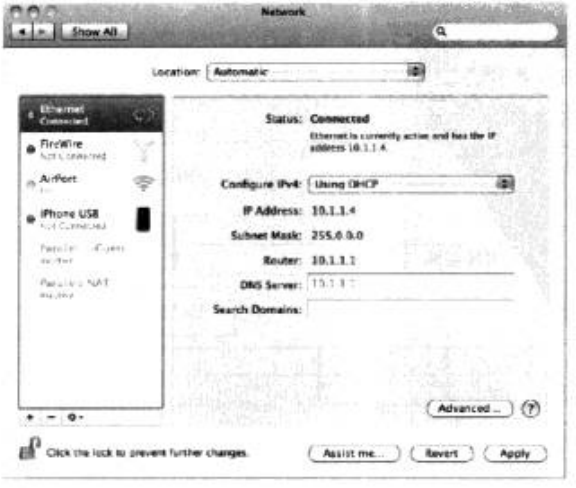



图1-1 Mac OS X系统网络界面的首选项表示网络内核扩展

操作系统的一项重要功能就是管理计算机硬件资源（如内存与CPU）以及外部设备（如磁盘存储器与键盘）。操作系统所支持的硬件设备的集合在不同的机器上有很大差别。尽管MacBook Air与Mac Pro运行相同的操作系统，但它们的硬件配置却有很大不同。为了让操作系统支持多种硬件配置而又不变得膨胀，支持每个硬件组件的代码被分别打包成一种特殊类型的内核扩展程序（称为驱动程序）。这种模块化方式使操作系统可以根据系统上现有的硬件按需加载驱动程序。厂商因而也可以在系统中安装相应的驱动程序以支持他们自己的硬件。安装标准的Mac OS X系统需要装100多个驱动程序，但其中只有一个子集用于运行特定的系统。

内核扩展程序开发与应用程序开发有很大不同。应用程序一般由用户事件驱动执行。应用程序在用户启动它时开始运行，然后等待用户点击按钮或选择菜单项，再处理相应的请求。相反，内核扩展没有用户界面，无需与用户交互。它们由操作系统加载，并由操作系统调用去执行自身无法独立完成的任务（如访问由内核扩展驱动的硬件设备）。

为了提高系统的安全性和稳定性，现代操作系统（如Mac OS X）将操作系统核心代码（内核）与应用程序、用户运行的服务程序分离。任何作为内核的一部分运行的代码（如驱动代码），都要在“内核空间”中运行。运行在内核空间中的代码被授予特权，如可以直接读写连接到计算机上的硬件设备，但标准的用户应用程序不享有该特权。

相反，用户处理的标准应用程序代码要在“用户空间”中运行。用户空间中运行的软件无法直接访问硬件。因此，为了访问硬件，用户代码必须向内核发送请求（如读取磁盘的请求），让内核代表应用程序执行任务。

用户空间与内核空间中运行的代码之间具有明确的界限。应用程序只能通过调用操作系统发布给用户空间代码的函数访问内核。类似地，内核空间代码独立运行于用户空间代码环境之外。内核不使用与用户空间代码相同的富编程API，它有自己的API集，供内核扩展开发者使用。如果你熟悉用户空间编程，那么这些API最初看起来可能有一定的局限性。因为通常情况下，用户交互和访问文件系统等操作不能在开发内核扩展时使用。图1-2说明了用户空间代码和内核空间代码的划分，以及各层之间的交互关系。

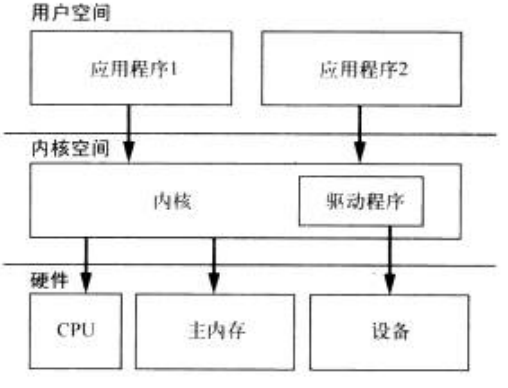



图1-2　现代操作系统职责分层

强制应用程序向内核发送访问硬件的请求有一大优点，即内核（及内核驱动程序）成为硬件设备的中央仲裁器。以声卡为例，系统上可能有多个应用程序在同一时刻播放音频，但由于所有的请求只通过一个音频驱动传送，所以该驱动程序能将来自所有应用程序的音频流混合，然后将产生的混合流提供给声卡。

本章的剩余部分将概述操作系统内核的功能，着重讲述其为用户应用程序提供硬件访问的重要性。我们将从最上层的应用软件着手，然后深入到操作系统内核层，最后介绍底层的硬件驱动程序。如果你已经熟悉这些概念，可以直接跳到第2章继续阅读。

## 1.1 操作系统的作用

作为启动序列的一部分，操作系统决定系统的硬件配置，搜索连接到USB接口或插入PCI扩展槽的外部设备，对它们进行初始化，如有必要，在初始化时加载驱动程序。

操作系统完成加载后，用户便能运行应用软件。应用软件可能需要分配内存或向磁盘写入文件，这些请求均由操作系统处理。对用户而言，操作系统的参与基本上是透明的。

操作系统在运行的应用程序和物理硬件之间提供了一个抽象层。应用程序一般通过向操作系统发出高层请求与硬件通信。因为这些请求由操作系统处理，所以应用程序可能完全不知道其运行环境的硬件配置（如RAM的数量、磁盘存储器是内部SSD还是外部USB驱动器）。

这个抽象层可以使应用软件在多种不同的硬件配置环境下运行，无需程序员为每种硬件添加支持，即便那些在程序发布后才出现的新硬件设备也不需要。

应用程序开发者通常可以忽略计算机系统工作的诸多细节，因为操作系统将运行应用程序的硬件平台抽象了出来。但是，作为一名驱动程序开发者，你所编写的代码将作为操作系统的一部分，直接与计算机硬件交互；你不能对操作系统的内部工作原理一无所知。为此，了解操作系统如何执行任务非常必要。

## 1.2 进程管理

通常，用户在计算机上安装了许多应用程序，而且这些应用程序都是纯被动实体。磁盘上的程序所包含的数据仅在程序运行时需要，这些数据由可执行代码和应用程序数据组成。当用户启动应用程序时，操作系统会将程序的代码和数据从磁盘加载到内存中，并开始执行代码。一个正在执行的程序称为“进程”。与程序不同，进程是主动实体，由程序执行期间单个实例的状态快照组成。其中包括程序代码、程序分配的内存以及当前的执行状态，如程序当前正在执行的函数的CPU指令、变量内容和内存分配情况。

通常，系统会同时运行许多进程。其中包括用户启动的应用程序（如iTunes或Safari）、操作系统自动启动的进程和不提示用户而运行的进程。例如，时间机器（Time Machine）备份服务每小时在后台自动运行一次，执行数据备份。而同一时刻，可能有同一程序的多个实例在执行，此时操作系统会将每个实例看做不同的进程。图1-3展示了Mac OS X中的实用程序Activity Monitor，我们可以用它查看系统上运行的所有进程。

图1-3　Mac OS X上的Activity Monitor显示了系统上运行的所有进程，可以跟显示可见用户应用程序的Dock对照着看

## 1.3　进程地址空间

尽管任意时刻会有许多进程同时运行，但每个进程都觉察不到系统有其他进程在运行。实际上，在没有显式代码的情况下，一个进程无法与另一个进程交互，也不能对其行为产生影响。

操作系统为每个进程提供了一段可操作的内存，我们称之为进程的地址空间。地址空间是动态的，它在进程执行期间随其内存分配而发生变化。如果一个进程试图读/写其地址空间外的内存地址，操作系统通常会将其终止，并通知用户该应用程序已经崩溃。

尽管受保护的内存并非新生事物，但也是近十年才出现在用户的桌面系统中。在Mac OS X之前，Mac OS 9上运行的进程可以读/写任何内存地址，即使该地址是由另一个进程分配的缓冲区或属于操作系统本身。

没有内存保护，应用程序可以绕过操作系统，直接修改不同进程的内存和变量（无论是否得到该进程的允许），以实现进程间的通信。对操作系统结构来说也是如此。例如，Mac OS 9有一个内部全局变量，它包含每个打开的GUI窗口链表。尽管该链表名义上由操作系统拥有和操作，但实际上应用程序无须对操作系统进行任何调用，就可以遍历和修改该链表。

没有内存保护，操作系统容易受用户应用程序中bug的影响。应用程序在有内存保护的系统上运行时，最坏的情况是损坏自身的内存和结构，但这种损坏仅局限于应用程序本身。在没有内存保护的系统上（如Mac OS 9），应用程序中的程序错误还有可能覆盖操作系统的内部结构，使系统完全崩溃，需要重启才能恢复。

请注意，在现代操作系统（如Mac OS X）中，内核都拥有自己的地址空间。在这种情况下，它可以独立操作所有运行的进程。在Mac OS X上，一个地址空间可以同时用于内核和所有加载的内核扩展。这意味着没有相应的措施保护操作系统核心结构不被有问题的驱动程序无意覆盖。用户进程可以轻易中止，但如果内核中也发生同样的情况，整个系统就会出现故障，必须重启计算机才能解决问题。这类错误在Mac OS X上表现为内核恐慌，即Windows上的“蓝屏死机”。为此，内核扩展开发者必须审慎使用内存管理，保证所有的内存访问都有效。

## 1.4 操作系统服务

在现代操作系统中，操作系统和应用程序执行的函数之间具有明确的界限。每当进程希望执行一个任务（如分配内存，从磁盘中读取数据，或通过网络发送数据）时，必须使用一组系统定义好的编程接口访问操作系统。`malloc()`、`read()`等系统函数，都是提供操作系统服务的系统调用。我们可以直接通过应用程序，或间接通过高级开发框架（如Mac OS X上的Cocoa框架）执行这些系统调用。Cocoa框架内部也是基于相同的系统调用实现的，它们访问操作系统服务时也要调用底层函数（如`read()`）。

但因为用户进程无法直接访问硬件或操作系统结构，所以它们需要调用`read()`等函数打破进程地址空间的限制。利用函数调用操作系统服务时，控制权从用户应用程序转至操作系统的特权部分（即内核）。将控制权转至内核这一过程通常需要借助CPU，CPU为此提供了一个指令。例如，现代Mac中的英特尔CPU提供一个`syscall`指令，它会跳转到操作系统启动时设置的函数。该内核函数首先需要知道用户进程执行哪一个系统调用（由调用进程写入CPU寄存器中的值确定），然后读取传送给该系统调用的函数参数（同样，这一过程也由调用进程通过CPU寄存器设置）。内核随后代表用户进程执行该函数调用，并将控制权及结果代码返回给该进程，如图1-4所示。

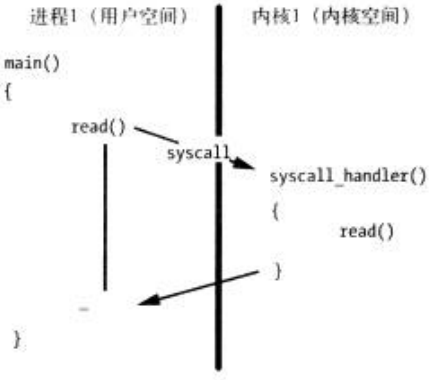



图1-4 系统调用的控制流程

内核是一个享有特权的进程，能够执行一些用户进程无权执行的操作，这些操作是配置系统

时必不可少的。当控制权转移到内核时（如系统调用之后），CPU在内核代码执行期间进入特权模式，并在返回用户进程之前，恢复到权限受限的状态。

由于内核在代表进程执行系统调用时，其权限级别高于用户进程，所以必须注意不要无意中造成安全漏洞。如果欺骗内核执行用户进程不应该执行的任务，就会发生这种情况，如打开一个用户没有读权限的文件，或向内核提供一个进程地址空间外的目标缓冲区。对于第一种情况，尽管内核进程本身有权打开系统上的任何文件，但因为它代表权限较低的用户进程执行操作，所以需要拒绝该请求。对于第二种情况下，如果内核访问一个无效的地址，将会产生无法恢复的错误，导致内核恐慌。

内核错误是灾难性的，需要重启整个系统。为了防止内核错误发生，每当内核代表用户进程执行请求时，一定要对进程提供的参数进行验证，而不应该假定它们有效。同样，每当驱动程序接受来自用户进程的控制请求时，也需要在内核实现的系统调用中进行验证，接下来几章会介绍这部分内容。

## 1.5　虚拟内存

计算机系统中的RAM资源有限，由系统上所有运行的进程竞争共享。当系统运行多个应用程序时，所有进程分配的内存总量会超过系统上RAM的数量，这很正常。

操作系统支持虚拟内存，这样进程就可以分配和使用比系统上安装的RAM数量还要多的内存。也就是说，进程的地址空间不受物理RAM数量的限制。有了虚拟内存，操作系统便可以使用次级存储器（如硬盘）上的后备存储空间，保存无法放到RAM中的进程的地址空间。然而，CPU仍然只能访问RAM中的地址，因此，操作系统必须在磁盘后备存储空间和RAM之间交换数据，以便能在进程运行时访问内存。

在特定时刻，进程可能只需要使用分配的总内存的一小部分，我们称之为进程的工作集。只要操作系统将这一工作集保存在RAM中，就可以忽略虚拟内存对执行速度的影响。工作集是一个动态实体，根据进程运行时当前使用的数据而变化。如果进程访问RAM外的内存地址，则需要从磁盘的后备存储中读取对应的数据，放入RAM中。如果当前没有空闲的RAM用于数据载入，就需要预先将RAM中现存的一些数据换出并保存到磁盘，释放物理RAM。

虚拟内存由操作系统处理。用户进程并不参与这一过程的实现，它不用关心地址空间是否在RAM中，也不用关心访问的数据是否转到了主存中。

使用虚拟内存的后果是，进程使用的地址与物理RAM中的地址并不对应。只要想一想进程地址空间比系统的RAM大，你就明白了。因此，需要将进程读取/写入的地址从进程的虚拟地址空间转换成物理RAM地址。由于访问每个内存都需要进行地址转换，为了减少对执行速度的影响，这些工作全由CPU执行。

操作系统通常使用一种称为“分页”的方案，来实现虚拟地址到物理地址的转换。使用分页内存方案时，物理内存被划分成固定长度的块，称为页帧。大多数操作系统，包括Mac OS X和iOS，使用的帧长度均为4096字节。类似地，每个进程的虚拟地址空间也都被划分成固定长度的块，称

之为页。每页的字节数与每帧的字节数始终相同。这样便可以将进程中的每页映射到物理内存中的一个帧里，如图1-5所示。

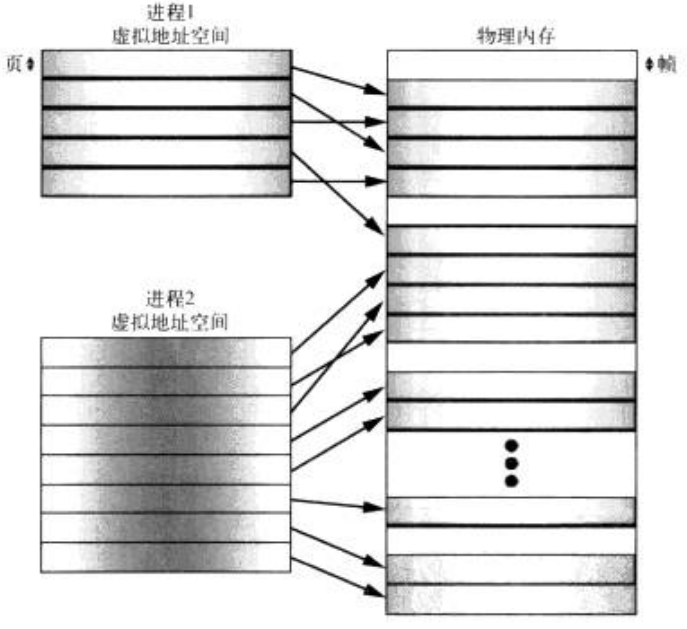



图1-5 进程地址空间中的页可以映射到内存中的任意页帧

虚拟内存的另一个优点是，进程虚拟地址空间中页码连续的缓冲区，可以分布到物理内存中多个不连续的帧里，如图1-5所示。这就解决了物理内存的碎片问题，因为一个进程分配的内存可以分布到多个物理内存段，而且不受最长的连续物理页帧组的大小限制。

作为启动进程的一部分，操作系统会创建一个表，保存进程的虚拟地址空间及其对应的物理地址之间的映射关系，这个表称为“分页表”。从概念上讲，页表项对应进程地址空间中的每页，以及每页对应的物理页帧地址。页表项还包含访问控制位和一个比特位，CPU根据前者来判断页是否只读，根据后者来判断页是在内存中还是已经换到后备存储器中。图1-6描述了CPU将虚拟地址转换为物理地址时需要执行的步骤。

如果CPU无法将进程访问的地址转换为物理地址，就会发生错误，我们称之为“缺页异常”。缺页异常运行在特权执行层，由操作系统处理。操作系统确定异常是否由地址不在进程地址空间而引起，如果是，则说明进程试图访问无效的地址，因而会被终止。如果是因为包含该地址的页已经换出到后备存储器而发生异常，操作系统将会执行一下操作。

(1) 分配一个物理内存中的帧，保存请求页；如果内存中没有空闲的帧可用，则将现有的帧换出到后备存储器中，腾出空间。

(2) 将请求页从后备存储器读取到内存中。
(3) 更新进程的页表，以便将请求的页映射到分配的帧。
(4) 将控制权返回给调用进程。

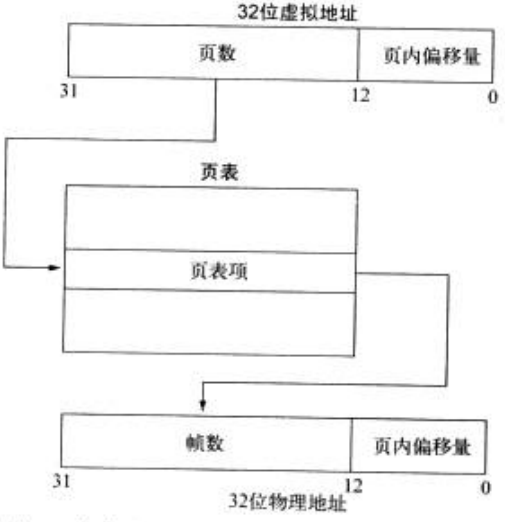



图1-6　页大小为4096字节（512位）的32位地址从虚拟地址到物理地址的转换

调用进程重新执行引起异常的指令，但这次，CPU可以找到页表中请求页的映射，此时指令成功完成。

对内核开发者来说，理解虚拟内存和分页必不可少。尽管内核代表用户进程处理请求，但它还有自己的地址空间，因此常常需要复制参数或将其从进程的地址空间映射到内核的地址空间。此外，直接同硬件设备交互的内核代码常常需要获得内存的物理地址。就拿处理用户进程读请求的磁盘驱动程序来说吧。从磁盘读取的数据将保存到一个缓冲区中，该缓冲区位于进程地址空间。与CPU一样，驱动程序控制的硬件只可以写入主内存的地址，而不是后备存储器中的地址。因此，为了处理该读请求，驱动程序需要确保用户缓冲区换入到主内存，并在整个读操作期间一直位于主内存中。最后，驱动程序需要将目标缓冲区的地址从虚拟内存地址转换为硬件可以访问的物理地址。第6章将详细讲述这部分内容。

尽管iOS为每个进程都提供了一个页表，但它并不支持后备存储器。乍一看，也许你会认为这完全背离了分页的初衷，但这样其实有两个非常重要的目的。第一，让每个进程可以完全访问内存。第二，避免了由物理内存碎片造成的问题。

## 1.6　调度

在计算机系统中，另一种争用激烈的资源是CPU。执行每个进程都需要访问CPU，但通常一

些想要访问CPU的进程比系统上的CPU内核更活跃。因此操作系统必须在运行的进程之间共享CPU内核，保证每个进程都可以正常访问CPU，从而顺利执行。

我们已经知道，进程彼此独立运行，都有各自的地址空间，以防止一个进程影响其他进程的行为。但在许多应用程序中，让两个独立执行的路径同时运行，而且不受每个路径必须在各自地址空间中运行的限制，这样做非常有用。这一执行单元被称为“线程”。多个线程都执行同一程序的代码，并在同一进程中运行（从而共享相同的地址空间），但却彼此独立。

对于操作系统来说，线程是调度的基本单位；当操作系统调度程序考虑接下来在CPU上调度什么时，只需查看系统上的活动线程。进程如若执行，至少要包含一个线程；一个新进程开始运行时，操作系统会自动为其创建初始线程。

调度器有两个作用：防止CPU闲置，否则会浪费宝贵的硬件资源；让所有线程公平地访问CPU，防止因单个线程独占CPU而导致其他线程无法运行。为此，一个线程会在可用的CPU内核上调度执行，直至以下任一事件发生。

- 过了一定时间（称之为时间量子）后，那时原线程将被操作系统抢占，调度执行另一个线程。在Mac OS X上，默认的时间量子是10毫秒。
- 线程在等待操作（如从磁盘读取数据或另一个线程返回结果）完成时，将不能执行。在这种情况下，调度器会让另一个线程在CPU上运行，从而阻塞原来的线程。这可以防止CPU在一个线程没有工作时处于闲置状态，从而最大限度地提高CPU在执行代码上花费的时间。线程还可以调用`sleep()`函数，自动放弃CPU时间，这会使当前线程的执行延迟一段时间。

为一个应用程序添加多个线程的一个原因是，使其可以在多个CPU内核中并发执行，从而能够将复杂的操作划分为多个同步运行的小步骤，从而加速应用程序的执行。即使在单核计算机上，多线程也有其优点。通过在活动线程间快速切换，调度器会让用户产生所有线程同时运行的错觉。即使一个线程被阻塞或进入了死循环，也不会影响其他线程的响应速度。这样，就可以将耗时的任务移到后台线程，让应用程序的其他部分自由地响应用户交互。

对于与硬件交互的应用程序，一种常见的设计是将访问硬件的代码放到其自身的线程中。软件代码等待硬件响应时常常进入阻塞状态；通过移除主程序线程中的这些代码，当程序需要等待硬件响应时，就不会对其用户界面产生影响。

另一种使用线程的常见情况是，保证软件以最小的延迟来响应硬件事件。应用程序在收到硬件通知前可以创建一个阻塞线程，然后可以使用后面几章介绍的技术与该线程通信。线程阻塞时，调度器不需要给它访问CPU的机会，因此线程的存在并不影响系统的性能。但在硬件发出事件通知之后，线程会变为非阻塞状态并在CPU上调度执行，并且可以自由执行任何需要的操作，从而响应硬件。

## 1.7 硬件和驱动程序

除了管理基本的硬件资源（如CPU和内存），操作系统还负责管理系统上添加的硬件外设，包

括键盘和鼠标、USB闪存驱动器、显卡等。尽管操作系统负责管理这些设备，但需要借助驱动程序才能实现。可将驱动程序看做运行在操作系统内核中的插件，它可以使系统与硬件设备进行交互。

支持硬件设备的代码位于设备自身（称为固件）和计算机（称为驱动程序）这两个位置。驱动程序的作用是代表操作系统控制硬件设备。驱动程序代码载入操作系统内核之后，会与内核的其他部分享有相同的权限，如直接访问硬件。

驱动程序在设备插入计算机（或计算机启动）时负责对硬件进行初始化，将操作系统的请求转换为一系列硬件特有的操作，然后由硬件执行这些操作以完成操作系统的请求。

驱动程序会接收到来自操作系统的请求，请求的类型取决于驱动程序执行的功能。对某些驱动程序，操作系统会为其开发者提供一个框架。例如，需要为声卡编写一个音频驱动程序。音频驱动程序接收操作系统特定于音频领域的请求，如创建一个48 kHz音频输出流，以及输出分好组的音频包。

驱动程序还可以构建在其他驱动程序上，并请求它们提供的服务。例如，USB音频输入设备的驱动程序使用底层通用USB驱动程序提供的服务访问其硬件。这样开发者就无须深入了解USB协议，从而可以专注于自己设备的特性。如在上面的例子中，音频驱动程序接收到来自操作系统的音频流操作请求之后，会针对这些请求创建自己的请求，然后传递给底层的USB驱动程序。这使每个驱动程序的责任相当明确：音频驱动程序只需关注音频请求的处理和音频设备的配置，USB驱动程序只需关注USB协议和USB总线上的数据传输。驱动程序的分层示例，如图1-7所示。

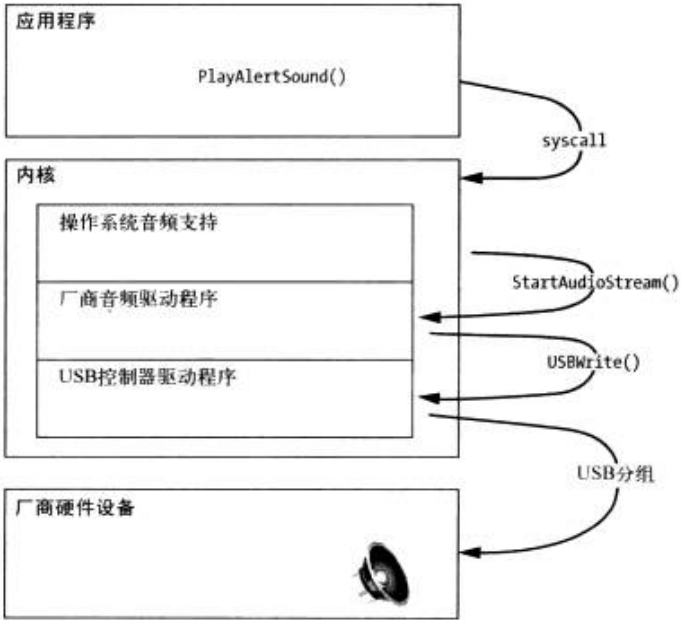



图1-7　应用程序到硬件的音频请求控制链

然而，操作系统不可能支持所有硬件设备。专用设备（如3D打印机）很可能得不到操作系统的支持。因此，硬件厂商需要为他们的硬件编写通用驱动程序。由于是通用驱动程序，操作系统不会将该设备看做打印机，也不会向其发送打印请求，而是由专门的应用软件对该驱动程序进行控制，直接与打印机驱动程序通信。操作系统提供了一个专门的系统调用，使用户应用程序可以向驱动程序发送操作请求，我们称之为“I/O控制”请求，通常缩写为“ioctl”。ioctl指定要执行的操作，并为驱动程序提供操作所需的参数，其中包含一个保存操作结果的缓冲区。尽管ioctl请求作为操作系统的系统调用来实现，但实际上此类请求直接传至驱动程序。

## 1.8 小结

操作系统负责管理计算机的硬件资源。它为用户程序提供了计算机系统的抽象模型，使每个程序好像都能完全访问CPU和整个内存。用户运行的程序若不向操作系统提供的服务发出请求，就无法访问硬件。在处理与外围硬件设备相关的服务时，操作系统可能需要调用该设备的驱动程序所提供的函数。

接下来几章将把本章阐述的概念付诸于实践，介绍Mac OS X提供的各种接口，功能涉及允许驱动程序使用虚拟内存地址和物理内存地址、响应来自用户应用程序的请求，以及与PCI和USB设备通信。

# 第2章 Mac OS X和iOS

Mac OS X是由苹果公司开发的基于Unix的现代操作系统，面向麦金塔（Macintosh）系列的计算机。OS X是Mac OS的第10版。

OS X易于使用且极富吸引力的图形用户界面为人所称道。苹果因其产品而赢得一众狂热果粉，OS X或iOS上增加的任何新特性都会受到广泛关注。除了OS X普通版，苹果还提供OS X服务器版，称为Mac OS X Server。

服务器版本随后在Mac OS X 10.7（Lion）中与普通版本进行了合并。OS X是Mac OS 9的接班人，代表对早期版本的彻底放弃。与前一个版本不同，OS X基于NeXTSTEP操作系统。目前，Mac OS X已经有8个版本，最新的版本是Mac OS X 10.7，代号为狮子（Lion）。Mac OS X迄今为止发布的版本如表2-1所示。

表2-1 Mac OS X迄今为止发布的版本

| 版　本 | 名　称 | 发　布 |
|---|---|---|
| 10.0 | 猎豹（Cheetah） | 2001.3 |
| 10.1 | 美洲狮（Puma） | 2001.9 |
| 10.2 | 美洲虎（Jaguar） | 2002.8 |
| 10.3 | 黑豹（Panther） | 2003.10 |
| 10.4 | 老虎（Tiger） | 2005.4 |
| 10.5 | 豹子（Leopard） | 2007.10 |
| 10.6 | 雪豹（Snow Leopard） | 2009.8 |
| 10.7 | 狮子（Lion） | 2011.7 |

Mac OS X预装了一组开发工具，如Xcode，可用于开发各种应用程序，包括本书的主题——内核扩展。

对于最终用户，OS X通常会捆绑预装iLife套件，其中包含照片、音频、视频编辑软件以及网页制作软件。

## NEXTSTEP

OS X和iOS基于NeXT计算机公司开发的NeXTSTEP操作系统，该公司由史蒂夫·乔布斯（Steve Jobs）在1985年离开苹果公司之后创立，最初由乔布斯自己投资，但随后获得了大量外部投资。NeXT后来被苹果公司收购，NeXTSTEP技术也加入到OS X中。NeXT主要生产面向学术界和商业领域的计算机。尽管NeXT计算机在商业竞争中的业绩有限，但它有一个非常富于创新的操作系统，即NeXTSTEP，该系统在当时很多方面都处于领先地位。

NeXTSTEP拥有与当前OS X版本类似的图形化用户界面和命令行界面（iOS没有命令行界面）。现在，我们仍然可以在其后续版本中找到NeXTSTEP引入的众多核心技术，如应用程序包和Interface Builder。Interface Builder现在是Xcode开发环境的一部分，广泛用于OS X和iOS Cocoa应用程序。NeXTSTEP提供的驱动程序工具包（Driver Kit），是一个面向对象的驱动程序开发框架，后来演变为I/O Kit——本书的主题之一。

iOS由OS X发展而来，是苹果公司面向移动设备的操作系统，在2007年第一代iPhone发布时推出，当时被称为iPhone OS，但后来改名为iOS，这样就能更好的反映出它可以在其他移动设备（如iPod Touch、iPad和最近的Apple TV）上运行的优点。iOS是专门为带有触摸界面的移动设备打造的。与最大的竞争者Windows不同，OS X和iOS不能被第三方注册使用，只能在苹果公司的硬件产品上使用。Mac OS X体系结构的全局视图，如图2-1所示。

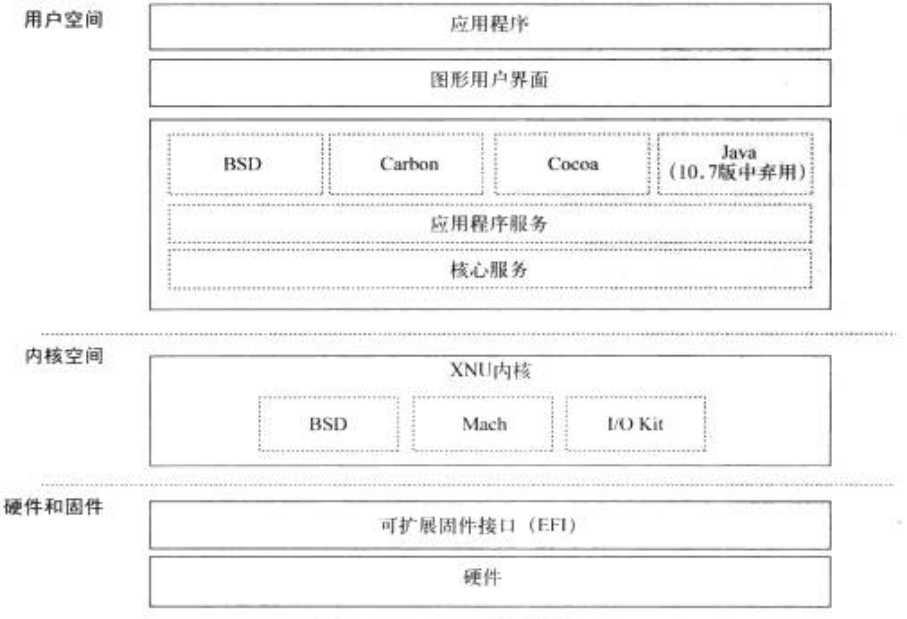



图2-1　Mac OS X体系结构

Mac OS X和iOS的核心遵循POSIX标准，从Mac OS X 10.5（Leopard）时开始遵循Unix 03认证。OS X和iOS核心包括内核和该操作系统的Unix基底，即Darwin，这是苹果公司发布的开源操作系统。与Mac OS X不同，Darwin不包括特有的用户接口，因为它只是一个准系统，仅提供内核和Unix系统中的典型工具和服务的用户空间基底。在其发布版中，唯一支持的体系结构是PowerPC平台，但随着苹果公司转向英特尔体系结构，又添加了对英特尔32位和64位支持。迄今为止，苹果公司还没有发布基于iOS的Darwin ARM版本。目前我们只能下载Darwin的源代码，但还需要对其进行编译。Darwin发布版包括XNU内核的源代码。对于想进一步了解操作系统内部工作原理的人以及内核扩展开发者来说，该内核源代码特别有用。一般情况下，与苹果公司开发者网站上的文档相比，源代码头或代码体中的解释更加详尽。

Darwin操作系统（以及OS X和iOS）运行XNU内核，该内核以Mach内核的代码以及FreeBSD操作系统的一部分为基础。Mac OS X的桌面如图2-2所示。

图2-2　Mac OS X桌面

## 编程API

从图2-1中我们可以看到，OS X有一个分层体系结构。Darwin核心和用户应用程序之间有很多编程API。其中最重要的是Cocoa，它是基于GUI应用程序的首选框架，相当于iOS上的Cocoa Touch，二者基本相同，但Cocoa Touch拥有触摸式用户交互专用的GUI元素。Cocoa和Cocoa Touch均采用Objective-C语言编写而成。Objective-C是C语言的一个超集，支持Smalltalk风格的消息。

**Objective-C**

Objective-C是在Mac OS X、iOS以及它们的前身NeXTSTEP上，开发应用程序的首选编程语言。它是C语言的一个超集，支持面向对象编程，但缺乏许多像C++等编程语言提供的高级功能，如多重继承、模板和操作符重载。Objective-C使用Smalltalk风格的消息和动态绑定（这在一定程度上消除了对多重继承的依赖）。这一语言在20世纪80年代初，由布莱德·考克斯（Brad Cox）和汤姆·拉夫（Tom Love）发明。尽管驱动程序和系统级编程通常采用C或C++，但实际上Objective-C仍然是OS X和iOS开发应用程序的标准编程语言。许多核心框架在类名中依然使用NS（用于NeXTSTEP）前缀，如`NSString`和`NSArray`。

其他的编程API有BSD API和POSIX线程API（pthreads），其中，BSD API是应用访问低级文件的途径，并且可用于访问设备。与Cocoa不同，BSD层不支持编写具有图形用户界面的应用程序。Mac OS X还有一个主要的API，即Carbon。Carbon是基于C语言的API，与Cocoa拥有相同的功能。Carbon可向后兼容Mac OS的早期版本。Carbon API现在已弃用，取而代之的是用于GUI应用程序的Cocoa，但OS X仍然保留了对传统应用程序的支持，如苹果公司的Final Cut Pro 7。Carbon公开的可用版本只有32位，因此用于64位兼容时，需要使用Cocoa。第四个主要的API是Java，但现在也弃用了。尽管Mac OS X 10.7仍然将Java作为安装时的可选项，但在默认安装中，已经将其移除。

图形和多媒体是OS X和iOS优于其他操作系统的关键，二者都提供了许多处理图形和多媒体的API。图形系统的核心是Quartz系统。Quartz包括视窗系统（Quartz Compositor），以及名为Quartz 2D的API。Quartz基于PDF（Portable Document Format，便携文档格式）模型，提供与分辨率无关的用户界面，还可以对文本和图形进行抗锯齿渲染。在图形硬件支持的情况下，Quartz Extreme接口提供窗口的硬件辅助OpenGL渲染。以下是对一些重要图形和多媒体框架的简短概述。

- Quartz：由Quartz 2D API和Quartz Compositor组成，Quartz Compositor提供图形窗口服务器。Cocoa Drawing提供基于Quartz的面向对象接口，用于Cocoa应用程序。
- OpenGL：开发3D应用程序的行业标准API。iOS支持的OpenGL版本为OpenGL ES，它是OpenGL的一个子集，专门为嵌入设备而设计。
- Core Animation：一个基于分层的API，与Cocoa集成在一起，使创建动画内容和执行变换变得更加容易。
- Core Image：支持图像处理，包括添加特效、裁剪或色彩校正。
- Core Audio：支持音频回放、录制、混合和处理。
- QuickTime：处理多媒体的高级库。支持音频视频（包括专业格式）的录制和回放。
- Core Text：基于C语言的API，用于文本渲染和布局。Cocoa Text API基于Core API。

## 支持的平台

OS X发布时，只有PowerPC平台支持OS X。2006年1月，苹果公司发布了10.4.4版本，就像

在WWDC 2005上声称的那样，终于将Mac OS X引入到英特尔x86平台。离开PowerPC平台的原因，据苹果公司称，是因为它们对IBM交付有竞争力的微处理器的能力感到失望，特别是用于笔记本电脑的低功耗处理器。苹果公司向英特尔转移进行得很顺利，业内成功进行平台转移的例子为数不多，而这就是其中一个。

苹果公司提供了一个名为Rosetta的优雅方案，这是一个动态翻译程序，可以使现存的PowerPC应用程序在基于x86的Mac上运行（事实上会损失一部分性能）。苹果公司还为开发者提供通用二进制代码，使多种体系结构的本机代码可以在单个二进制可执行文件中共存（也称为胖二进制）。虽然已经终止了对PowerPC的支持，但从Mac OS X 10.6（Snow Leopard）开始，通用二进制代码仍然可以为32位x86或64位x86_64提供可执行文件。

## 64位操作系统

Mac OS X 10.5（Leopard）第一次支持64位原生GUI应用程序，通过Cocoa的64位新版本实现，该版本可以使开发人员发掘当代Mac中64位CPU的最大优势。基于Carbon API的应用程序仍然只有32位。Mac OS X 10.6（Snow Leopard）的后续版本更进一步，使内核可以在64位模式下运行。

虽然在Leopard版本中，大多数应用程序和API都是64位，但内核本身仍然在32位模式下运行。尽管Snow Leopard支持64位模式的内核，但也只有部分模型默认为64位，其他的模型都需要手动启用。Snow Leopard是第一个不支持PowerPC计算机的发布版，尽管PowerPC应用程序仍然可以通过Rosetta运行。Lion移除了对Rosetta和32位内核的支持。用户空间虽然可以同时支持32位和64位的应用程序，但在64位模式下运行时，内核与32位的驱动程序和扩展并不兼容。64位内核有许多优点，而且地址空间大意味着可以支持更大容量的内存。

## iOS

iOS，最初称为iPhone OS 1.0，在2007年6月发布（参见表2-2）。iOS基于Mac OS X，大部分基本体系结构与之前的兄弟系统相同。但iOS具有Cocoa Touch API提供的创新用户界面（与原始Cocoa有许多相同的特性和部分），专门为iPhone的电容触摸屏设计。除了Cocoa Touch，iOS还有其他一些编程API，如提供数学及其他相关函数的Accelerate框架，这些API都为iOS硬件进行了优化。External Accessory框架可以使iOS设备通过蓝牙或内置30针连接器与第三方硬件设备通信。

表2-2　iOS发布版

| 版　本 | 设　备 | 发　布 |
| --- | --- | --- |
| iPhone OS 1.0 | iPhone、iPod Touch（1.1） | 2007年6月 |
| iPhone OS 2.0 | iPhone 3G | 2008年7月 |
| iPhone OS 3.0 | iPhone 3GS、iPad（3.2） | 2009年6月 |
| iOS 4.0 | iPhone 4 | 2010年6月 |
| iOS 5.0 | iPhone 4S | 2011年10月 |

iPhone OS启动时无法运行本机的第三方应用程序，但可以运行为iPhone设计的Web应用程序，这些应用程序可以添加到iPhone的主屏上。2008年初，苹果发布了iPhone的SDK，可用于开发第三方应用程序。但与大多数计算机平台不同，苹果公司要求所有的iPhone应用程序都必须经过提交、审核以及数字签名，才可供用户下载安装。虽然许多人都曾抨击该方案（现在还在抨击），但它却可以使苹果公司删除编写差、运行慢的软件和恶意软件，从而提高用户的整体体验，最终促使平台广为流行。它还能通过非官方手段实现iOS“越狱”，访问底层的Unix和内核环境，但这样做会使平台失去保障。因为考虑到电池寿命有限，iPhone直到iOS 4.0发布才能够正确执行第三方应用程序的多任务处理。现在，iOS支持iPhone、iPod Touch和iPad，还可以在最新一代的Apple TV上运行，而在此之前，Apple TV基于OS X，运行在英特尔x86 CPU上。苹果公司目前还不支持Apple TV上的第三方应用程序。

## 2.1 XNU 内核

XNU内核庞大复杂，本书无意对其体系结构进行完整的描述（请参考其他书籍），但在随后的章节中，我们将概述XNU的一些主要组成部分，简要描述其职责及操作模式。大多数情况下，进行内核程序设计时，你要做的是编写扩展，而不是修改核心内核本身（除非你是苹果工程师或Darwin贡献者），但总体来说，你应当对内核有一个基本的理解，因为这会让你从全局上更好地理解内核扩展的方式。后续章节将关注一些比较重要的内核编程框架，如I/O Kit。

XNU内核是Mac OS X和iOS的核心，由三个主要部分组成的一个分层体系结构。该内核的内环称为Mach层，源自卡内基–梅隆大学（Carnegie Mellon University）开发的Mach 3.0内核。本书中提及的Mach都是指在OS X和iOS中实现的Mach，而不是原始工程。Mach作为微内核，是仅提供基础服务的一个薄层，如处理器管理和调度以及IPC（inter-process communication，进程间通信），IPC是Mach内核的核心概念。由于是分层体系结构，所以XNU的iOS和OS X版本之间几乎没有区别。

虽然XNU中的Mach层与原始工程中的职责相同，但其他操作系统服务（如文件系统和网络）都在与Mach相同的内存空间中运行。苹果公司这样做，主要是出于对性能的考虑，因为地址空间切换（上下文切换）这一操作代价很高。

因为在一定程度上，Mach层仍然是个孤立的部分，很多人常常将XNU称为混合内核而不是微内核或宏内核，在这两种内核中，所有的操作系统服务都在相同的环境中运行。XNU的体系结构如图2-3所示。

XNU的第二个主要部分是BSD层，可以将其看作围绕Mach层的一个外环。BSD同样也为最终用户的应用程序提供编程接口，其职责包括进程管理、文件系统和网络。

最后一个主要部分是I/O Kit，它为设备驱动程序提供一个面向对象的框架。

虽然理想情况是每层都各司其职，但实际情况却更加复杂，各层之间的界限非常模糊，因为许多操作系统的服务和任务都涉及多个部分。

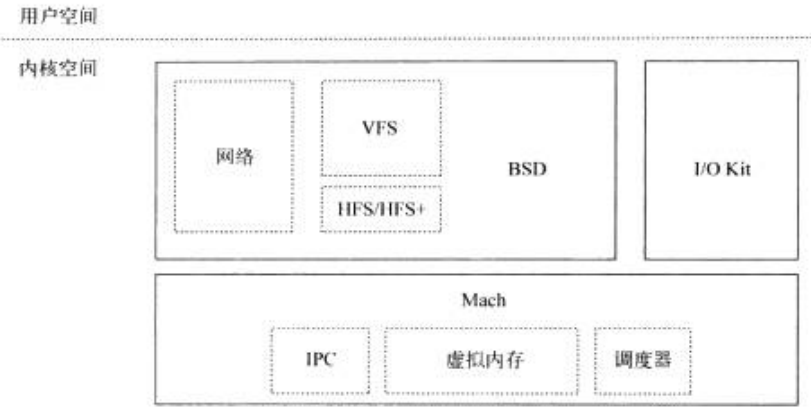



图2-3　XNU内核体系结构

**提示**　你可以从苹果公司的开源网站上下载完整的XNU源代码：http://www.opensource.apple.com。

### 2.1.1　内核扩展（KEXT）

XNU内核与大多数现代操作系统类似，运行时，它可将代码动态加载到内核地址空间。这样，即使内核运行时也可以装载和卸载额外功能，如驱动程序。本书主要讲述这种内核扩展的开发，并将重点放在驱动程序上，因为这是实现内核扩展最常见的原因。内核扩展有两个主要类。第一个类用于基于I/O Kit的内核扩展，这些扩展用于硬件驱动程序，采用C++语言编写。第二个类用于通用内核扩展，这些扩展通常采用C语言编写（尽管这里也可以采用C++语言）。这些扩展可以实现从新网络协议到文件系统的任何内容。通用内核扩展通常与BSD或Mach层进行交互。

### 2.1.2　Mach

Mach层可以看做内核的核心，它为高层部分（如BSD层和I/O Kit）提供底层服务。Mach负责硬件抽象，隐藏PowerPC体系结构和英特尔X86、X86-64体系结构之间的差别，这包括处理异常和中断的细节以及管理内存（包括虚拟内存和分页）的细节。这种设计可以使内核很容易地移植到新的硬件体系结构，苹果转向英特尔x86及后来的iOS转向ARM都证明了这一点。除了硬件抽象，Mach还负责线程调度。Mach支持对称多处理（SMP），可以在多个CPU或CPU核心之间调度进程。实际上，在现有的BSD Unix内核中实现适当的SMP支持，对Mach开发很有帮助。

进程间通信（IPC）是Mach设计的核心原则。Mach中的IPC作为客户端/服务器系统而实现。

一个任务（客户端）可以请求另一个任务（服务器）的服务。系统中的端点称为端口。端口具有确定客户是否可以访问特定服务的相关权限。IPC机制在XNU内核内部使用。接下来几节将概述Mach层提供的关键抽象和服务。



**提示** 可以在XNU源包的osfmk/man文件夹中找到Mach API文档。

### 1. 任务和线程

任务是一个包含一个或多个可执行线程的线程组，这些线程共享资源和内存地址空间。一项任务至少要有一个执行线程。Mach的任务是将线程一对一映射到Unix（BSD层）进程。XNU内核也是一个包含多个线程的任务（称为kernel_task）。任务资源为私有，通常不可以由另一个任务的线程访问。

与任务不同，线程是一个可执行实体，可以由CPU进行调度和运行。在同一任务中，一个线程与其他线程共享资源，如打开的文件或网络套接字。同个任务的线程可以在不同的CPU上并发执行。每个线程都有其自己的状态，包括处理器状态（寄存器和指令计数器）及其自己的栈。线程调度到CPU上运行时将恢复自身状态。Mach支持抢占式多任务处理，这意味着线程可以在其分配的时间片（XNU内为10毫秒）结束之前中断执行。抢占会在各种情况下发生，如有高优先级的系统事件发生，或更高优先级的线程需要运行，或需要等待耗时较长的I/O操作完成。线程还可以通过进入睡眠状态自动抢占自己。无论Mach线程属于哪一个任务，它都会独立于其他的线程进行调度。调度器也不知道Unix系统中常见的进程父子关系（但BSD层知道）。

### 2. 调度

调度器负责协调线程对CPU的访问。大多数现代内核（包括XNU）都使用分时调度器，为每个线程分配有限的时间量子（我们在前面看到，XNU中为10毫秒），线程可以在该时间量子内执行。线程的时间量子一超时，线程就会被迫休眠，以便其他线程执行。每个线程的运行时间相同，虽然这看起来非常合理公平，但却不切实际，因为一些线程对低延迟有更高的需求，例如执行音频和视频回放。XNU调度器采用基于优先级的算法调度线程。调度器使用的优先级如表2-3所示。

表2-3 调度器优先级

| 优 先 级 | 级 别 | 描 述 |
| --- | --- | --- |
| 普通 | 0~51 | 普通应用程序。普通应用程序线程的默认优先级是31。0为闲置优先级 |
| 高优先级 | 52~79 | 高优先级线程 |
| 内核模式 | 80~95 | 该范围为高优先级内核线程保留，例如供设备驱动程序使用的范围 |
| 实时 | 96~127 | 实时线程（用户空间线程可以实时运行） |

内核将线程组织到双向链表中。该链表集合称为运行队列。每个优先级都有一个链表（当前为0~127）。系统中的每个处理器（内核）则负责维护自己的运行队列结构（osfmk/ kern/sched.h）：

```
struct run_queue {
        int                     highq;          /* 最高可运行队列*/
        int                     bitmap[NRQBM];  /* 运行队列位图数组*/
        int                     count;          /* #线程总数*/
        int                     urgency;        /* 抢占紧急级别*/
        queue_head_t            queues[NRQS];   /* 用于每个优先级*/
};
```

普通应用程序线程的优先级从31开始。随着时间的变化，它的优先级会减小，这是调度算法的副作用。例如，一个线程的计算量很大时，就可能发生这种情况。降低此种线程的优先级会改善I/O绑定线程的调度延迟，I/O绑定线程在发送I/O请求的间歇大部分时间都处于睡眠状态，并通常在其时间量子超时之前再次进入睡眠状态，这样计算密集型线程就可以再次访问CPU。其最终结果是提高了系统的响应速度。

为了避免线程因优先级太低而无法运行，Mach调度器将会随时间变化而不断衰减线程处理器的使用量，并最终对其进行重置，从而使线程的优先级随时间波动。

Mach调度器支持实时线程，尽管不能保证延迟，但它会尽量保证实时线程运行它所需的时钟周期数。如果实时线程阻塞/睡眠不够频繁，它可能会被降为普通优先级，例如实时线程严重计算受限时。

### 3. Mach IPC：端口和消息

端口是一个单向通信端点，表示一种称为对象的资源。如果你熟悉TCP/IP网络连接，可以将Mach的IPC和UDP协议进行比较，尽管与UDP协议不同，但Mach IPC不只用于数据传输。Mach IPC还可用于同步或在任务之间发送通知。IPC客户端可以向端口发送消息，由端口拥有者接收。双向通信需要两个端口。一个端口实现为消息队列（尽管还存在其他机制），端口的消息会进行排队，等待线程来处理它们。另一个端口可以接收来自多个发送者的消息，但每个端口只能有一个接收器。

端口有保护机制，称为端口权限。为了与端口交互，任务必须有正确的权限。端口权限与任务相关，因此，任务里的所有线程都共享相同的端口权限。比如端口具有这些权限：发送、发送一次和接收。权限可以在任务之间复制或移动。与Unix权限不同，子进程不能继承父进程的端口权限（Mach任务没有此概念）。可用的端口权限类型如表2-4所示。

表2-4　端口权限类型（定义在mach/port.h中）

| 端口权限类型 | 描　述 |
| --- | --- |
| MACH_PORT_RIGHT_SEND | 该权限的持有者可以向端口发送消息 |
| MACH_PORT_RIGHT_RECEIVE | 持有者有权接收来自端口的消息。接收权限提供自动发送权限 |
| MACH_PORT_RIGHT_SEND_ONCE | 与发送权限相同，但只能发送一次消息 |
| MACH_PORT_RIGHT_PORT_SET | 面向一组端口接收（和发送）消息的权限 |
| MACH_PORT_RIGHT_DEAD_NAME | 表示无效或销毁的权限，如使用发送一条消息的权限向端口发送消息之后 |

一组端口总称为端口集，消息队列由端口集中的所有端口共享。系统中的端口地址为32位整数。端口没有全局寄存器或命名空间。

Mach IPC系统还可在用户空间程序中使用，用于在任务之间或从任务向内核传递消息。尽管这一机制是在底层使用系统调用，但它为系统调用提供了选择方案。



#### 4. Mach异常

异常是指线程执行期间某些（异常）事件或情况发生时，CPU发送的中断。异常将导致线程执行中断，虽然操作系统（Mach）会处理这些异常。根据发生异常的具体类型，任务可能会在异常发生之后恢复。引起异常的一般原因包括访问无效或不存在的内存、执行无效的处理器指令、传递无效的参数或除数为0。这些异常通常会终止相关的任务，但还会发生一些非错误的异常。

系统调用便属于这类异常。用户空间应用程序需要执行涉及内核的底层操作时，如文件写入或接收网络套接字上的数据，便会发生系统调用异常。操作系统处理该系统调用时，会检查系统调用编号对应的寄存器，据此查询该调用的处理程序，例如`read()`或`recv()`。任务试图访问页面溢出的内存时，可能也会发生异常。这种情况下会发生缺页异常，该异常可以通过从后备存储器检索缺失页来处理，否则将导致内存访问无效。任务还可以使用`EXC_BREAKPOINT`发送故意产生的异常，这通常用于调试或追踪应用程序（如Xcode）以便能临时暂停线程的执行。

当然，内核本身也可能会产生错误，发生异常。在这种情况下，操作系统会暂停，显示死机灰屏（除非启用内核调试器），并通知用户重启计算机。Mach异常的子集如表2-5所示。

表2-5 常见的Mach异常类型

| 异常类型 | 描述 |
| --- | --- |
| EXC_BAD_ACCESS | 内存访问无效 |
| EXC_BAD_INSTRUCTION | 线程试图访问非法/无效的指令或将无效的参数（操作数）传递给指令 |
| EXC_ARITMETHIC | 除以0或整数溢出/下溢引发的异常 |
| EXC_SYSCALL 和 EXC_MACH_SYSCALL | 应用程序访问内核服务（如文件I/O）或网络访问时发出 |
| …… | 其他Mach异常定义在mach/exception_types.h中。与处理器相关的异常定义在mach/(i386，ppc，…)/exception.h中 |

发生异常时，内核将挂起引起异常的线程，并向线程的异常端口发送IPC消息。如果线程不处理该异常，则该异常将被转送给包含此任务的异常端口，并最终转送给系统（主机）的异常端口。线程、任务或处理器（主机）的异常端口被封装在下面的结构中：

```
struct exception_action {
        struct ipc_port*            port;          /* 异常端口*/
        thread_state_flavor_t       flavor;        /* 发送的状态形式*/
        exception_behavior_t        behavior;      /* 引发的异常类型*/
        boolean_t                   privileged;    /* 保存ipc_task_reset */
};
```

每个线程、任务和主机都有一个`exception_action`结构体数组，可以指定异常行为，每个异常类型（如表2-5中的定义）定义一个结构体。`flavor`和`behavior`字段指定应该与异常消息一起发送的信息类型，如通用状态或其他指定的CPU寄存器以及应该执行的处理程序。处理程序可以是

catch_mach_exception_raise()、catch_mach_exception_raise_state()或catch_mach_exception_raise_state_identity()。当异常已经发出后，内核将等待回复以确定处理过程。返回KERN_SUCCESS则意味着已经成功处理异常，可以恢复该线程。

线程默认的异常端口为PORT_NULL，除非明确分配一个端口，否则异常将由任务的异常端口来处理。当进程发送fork()系统调用衍生出一个子进程时，子进程将继承父进程的异常端口。Unix的信号机制便基于Mach的异常系统而实现。

**5. 时间管理**

正确计时是所有操作系统的重要职责，处理用户应用程序时如此，处理其他重要的内核功能（如调度进程）时也是如此。在Mach中，时间管理的抽象称为时钟。Mach中的一个时钟对象将单位为纳秒（ns）的时间表示为一个单调递增的值。Mach中定义了3个主要的时钟：实时时钟、日历时钟和高分辨率时钟。实时时钟记录自上一次启动的时间，而日历时钟通常使用电池，所以系统重启或计算机关闭期间，它的值持续存在。此外，还有一个秒分辨率，顾名思义，它用于记录当前时间。Mach时间KPI包含以下三个函数：

```
void clock_get_uptime(uint64_t* result);
void clock_get_system_nanotime(uint32_t* secs, uint32_t* nanosecs);
void clock_get_calendar_nanotime(uint32_t* secs, uint32_t* nanosecs);
```

日历时钟通常只用于应用程序，因为内核本身很少关注当前时间或日期，而实际上，如果将日历时钟用于内核会是一个很糟糕的设计。内核使用的是实时时钟提供的相对时间。实时时钟的时间通常来自于计算机主板上的晶振电路。实时时钟电路（RTC）是可编程的，并被连接到CPU（每个CPU/核心）的中断针脚。RTC在XNU中使用最大为100Hz的频率进行编程控制（使用clock_set_timer_deadline()）。

**6. 内存管理**

Mach层负责以与机器无关的方式协调物理内存的使用，为高级组件提供一致的接口。Mach的虚拟内存子系统，即Mach VM，为应用程序及内核自身提供受保护的内存和设施，以便分配、共享和映射内存。内核程序员必须切实了解内存管理。

**7. 任务地址空间**

每一个Mach任务都有其自己的虚拟地址（VM）空间。对于32位的任务，地址空间是4 GB，而对于64位的任务，其地址空间则大得多，有51位（大约2 PB）可用的地址空间。特定的应用程序（如视频编辑或特效软件）通常超过32位地址空间。OS X 10.4可以支持64位虚拟地址空间。

---

**注意**　虽然32位应用程序限定的地址空间为4 GB，但这与系统中可用的物理内存数量并不相关。OS X支持的物理地址扩展（PAE）等技术使32位的x86处理器（或运行在32位模式下的64位处理器）可以寻址至物理内存的36位，但任务的地址空间仍限定在4 GB。

---

任务的地址空间是受保护内存概念的根本。任务不可以访问地址空间，进而也无法访问包含另一个任务数据的底层物理内存，除非通过使用共享内存或其他机制明确允许这么做。

**内核地址空间管理**

内核本身有其自己的任务，即kernel_task，该任务有自己独立的地址空间。我们假设一个32位操作系统，如iOS。一些基于Unix的操作系统，包括Linux，都设计成将内核的地址空间映射到每个任务的地址空间。内核有1 GB的可用地址空间，而任务有3 GB的可用地址空间。当一个任务的上下文切换至内核空间时，MMU（memory management unit，内存管理单元）可以避免为TLB（translation lookaside buffer，变换旁查缓冲器）重新配置一个新的地址空间，因为内核位置已知，因此可以加速原本代价很高的上下文切换。当然，缺点是内核的可用地址空间非常有限，只有3 GB可以用于任务。在XNU中，内核在自身的虚拟地址空间中运行，该虚拟地址空间并不与用户任务共享，其中4 GB用于内核，4 GB用于用户任务。

### 8. VM映射和VM映射项

虚拟地址（VM）映射实际表示任务的地址空间。每个任务都有自己的VM映射，由vm_map结构表示。不存在与线程关联的映射，因为它们共享其所属任务的VM映射。

VM映射表示映射到进程地址空间内存区域的一个双向链表。每个区域几乎都是一个连续的内存地址范围（不需要对应到连续的物理内存），由一个起始和结束地址以及其他元数据（如保护标志）描述，这些元数据可以是读、写和执行的任意组合。该内存区域由vm_map_entry结构表示。当更多的内存分配在现有映射项前后或拆分成较小的区域时，VM映射项可以与另一个相邻项合并。如果为了一个映射项的地址范围而修改保护标支，就会发生拆分，因为保护标志只能在VM映射项上设定。图2-4展示了带有两个VM映射项的VM映射。

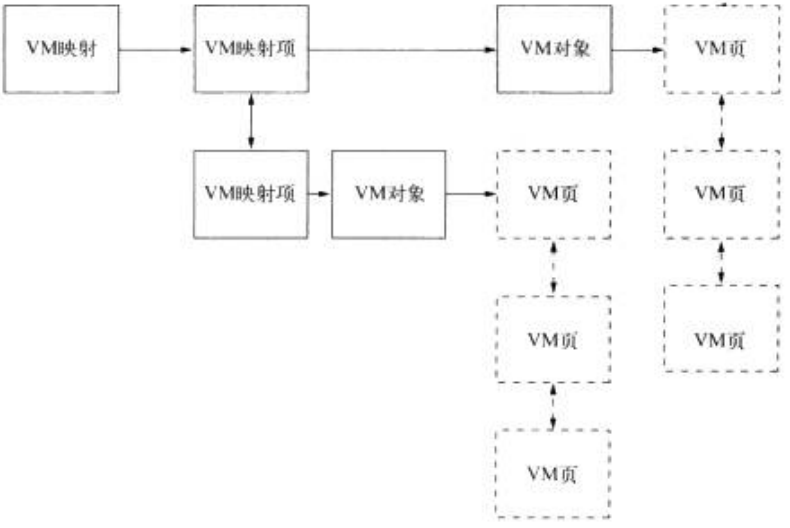



图2-4 VM子系统结构之间的关系

**提示**　任务地址空间的相关结构定义在XNU源包的mach/vm_map.h和mach/vm_region.h中。

**9. 物理映射**

每一个VM映射都有其相关联的物理映射或pmap结构。该结构帮助保存任务使用的虚拟内存到物理内存的映射信息。处理物理映射的Mach VM部分依赖于机器，因为它要与内存管理单元（MMU）——处理地址转换系统的特定硬件组件——进行交互。

**10. VM对象**

VM映射项可以指向VM对象或VM子映射。子映射是其他（VM映射）映射关系的一个容器。子映射用于在地址空间之间共享内存。VM对象表示位置，或者更确切地说是访问描述的内存方式。VM对象基于的内存页可能不在物理内存中，但可以位于外部后备存储器（OS X上的硬盘驱动器）中。在这种情况下，VM对象将含有如何换入（page in）外部页的信息。是传入后备存储器还是从后备存储器传输出去，是由分页器处理的，后面将介绍分页器。

VM对象以页为单位描述内存。目前在XNU中，一页是4096字节。虚拟页由vm_page结构描述。一个VM对象可以包含许多页，但一个页只与一个VM对象关联。

**页**

页是虚拟内存系统的最小单位。在Mac OS X、iOS以及许多其他操作系统上，一页的长度为4096字节（4 KB）。页长度取决于处理器，因为处理器，或者更确切地说是处理器的内存管理单元（MMU）负责虚拟内存到物理内存的映射，管理VM页表缓存（也称为TLB）。许多体系结构的页长度由操作系统设定，如对于x86体系结构，其页长可以达到4 MB，甚至可以是多个页长度的混合。操作系统包含一个名为页表的数据结构，其中包含每个物理内存块（以页为单位）对应的一个struct vm_page。该结构包含元数据，如页是否正在使用。

需要在任务之间共享内存时，VM映射项将通过子映射指向外部地址空间，而不是VM对象。这通常在使用共享库时发生。该共享库将映射到任务的地址空间。

来看另一个例子。当Unix进程发送fork()系统调用创建一个子进程时，新进程将创建为父进程的一个副本。为了避免从父进程向子进程复制内存，我们采用了一种叫作写时复制（copy-on-write，COW）的优化方法。对子进程的内存进行读访问，只需引用与父进程相同的页。如果子进程修改自身的内存，描述该内存的页将被复制，并创建一个VM影像对象。在下一次读取该内存区域时，将检查该影像对象是否有该页的副本，如果没有，则引用原来的共享页。前面描述的行为仅在父进程的原始VM映射项将继承属性设定为copy时发生。其他可能设定的值有shared，在这种情况下，子进程将继续对原来的内存区进行读写操作。如果属性设定为none，映射项引用的内存页将无法映射到子进程的地址空间。第四个可能设定的值是copy and delete，此时内存将被复制到子进程，并从父进程中删除。

**注意** Mach IPC也使用写时复制优化任务之间的数据传输。



### 11. 检查任务的地址空间

利用vmmap命令行实用程序可以查看进程的虚拟内存映射及其VM映射项。它清楚地说明了内存区域如何映射到任务的VM地址空间。vmmap命令将进程标识符（process identifier，PID）作为参数。下面展示了vmmap的执行输出，它使用了简单的Hello World C应用程序（a.out）的PID，应用程序输出消息之后将进入睡眠状态：

```
==== Non-writable regions for process 46874
__PAGEZERO              00000000-00001000 [     4K] ---/--- SM=NUL  /Users/ole/a.out
__TEXT                  00001000-00002000 [     4K] r-x/rwx SM=COW  /Users/ole/a.out
__LINKEDIT              00003000-00004000 [     4K] r--/rwx SM=COW  /Users/ole/a.out
MALLOC guard page       00004000-00005000 [     4K] ---/rwx SM=NUL
MALLOC metadata         00021000-00022000 [     4K] r--/rwx SM=PRV
__TEXT                  8fe00000-8fe42000 [   264K] r-x/rwx SM=COW  /usr/lib/dyld
__LINKEDIT              8fe70000-8fe84000 [    80K] r--/rwx SM=COW  /usr/lib/dyld
__TEXT                  9703b000-971e3000 [  1696K] r-x/r-x SM=COW  /usr/lib/libSystem.B.dylib
STACK GUARD             bc000000-bf800000 [ 56.0M] ---/rwx SM=NUL  stack guard for thread 0
==== Writable regions for process 46874
__DATA                  00002000-00003000 [     4K] rw-/rwx SM=PRV  /Users/ole/a.out
MALLOC metadata         00015000-00020000 [    44K] rw-/rwx SM=PRV
   MALLOC_TINY             00100000-00200000 [  1024K] rw-/rwx SM=PRV  DefaultMallocZone_0x5000
MALLOC_SMALL            00800000-01000000 [  8192K] rw-/rwx SM=PRV  DefaultMallocZone_0x5000
__DATA                  8fe42000-8fe6f000 [   180K] rw-/rwx SM=PRV  /usr/lib/dyld
__IMPORT                8fe6f000-8fe70000 [     4K] rwx/rwx SM=COW  /usr/lib/dyld
shared pmap             a0800000-a093a000 [  1256K] rw-/rwx SM=COW
__DATA                  a093a000-a0952000 [    96K] rw-/rwx SM=COW  /usr/lib/libSystem.B.dylib
shared pmap             a0952000-a0a00000 [   696K] rw-/rwx SM=COW
Stack                   bf800000-bffff000 [  8188K] rw-/rwx SM=ZER  thread 0
Stack                   bffff000-c0000000 [     4K] rw-/rwx SM=COW  thread 0
```

为了易于阅读，节选了部分输出结果。该输出划分为不可写区域和可写区域。正如你看到的，前者包括零页映射，该映射为只读，如果应用程序试图写入内存地址0-4096（十进制4096等于十六进制0x1000），将会发生异常。这就是试图进行空指针引用时应用程序崩溃的原因。下一个映射项是应用程序的文本段，其中包括应用程序的可执行代码。你会看到，该文本段标记有COW的共享模式（share mode，SM），这意味着如果此进程衍生一个子进程，该子进程将从父进程继承此映射，从而避免在文本段中的页被修改之前复制映射。

除了a.out程序自身的文本段，你还可以看到libSystem.B.dylib的一个映射。在Mac OS X和iOS上，libSystem实现了标准C语言库和POSIX线程API以及其他的系统API。a.out进程从父进程/sbin/launchd继承了libSystem的映射，/sbin/launchd是所有用户空间进程的父进程。这能确保该库只加载一次，可以节省内存并提高应用程序的启动速度，因为从二级存储获取一个库（如硬盘）通常比较慢。

在可写区域中你可以看到a.out和libSystem的数据段。这些段包含由程序/库定义的变量。显

然，这些数据段可以修改，所以每个进程都需要共享库数据段的一个副本，但这是写时复制（COW），所以在进程修改映射前没有任何开销。

**提示**　如果你想查看系统进程的虚拟内存映射，如launchd，那么需要使用sudo运行vmmap，因为默认情况下用户只能查看他自己的进程。

### 12. 分页器

虚拟内存允许进程的虚拟地址空间比可用的物理内存大，并且可以使系统上运行的任务合并，从而使用比可用的内存量更大的内存。实现该功能的机制称为分页器。分页器控制与系统内存（RAM）之间的内存页传输，或向二级后备存储器（通常为硬盘）传输内存页。当有较高内存要求的任务需要运行时，分页器可以临时将不活动任务的内存页传输（换出）到后备存储器，从而释放足够的内存，执行这些任务。类似地，如果发现一个进程基本闲置，系统可以选择换出该任务的内存，释放内存，用于当前或将来的任务。当应用程序运行并试图访问已经换出的内存时，便会发生异常。这种异常称为缺页异常，任务试图访问一个无效的内存地址时，也会发生该异常。当发生缺页异常时，内核将试图传回（换入）与该内存地址相对应的页，如果无法传回该页，则将被当作无效的内存访问，放弃该任务。XNU内核支持三种不同的分页器。

- 默认分页器：在主内存和系统硬盘上的交换文件（/var/vm/swapfile*）之间执行传统的分页和传输。
- 虚拟节点分页器：与文件系统使用的统一缓冲区高速缓存（Unified Buffer Cache，UBC）一致，用于缓存内存中的文件。
- 设备分页器：用于管理硬件设备（如将寄存器映射到内存的PCI设备）的内存映射。映射的内存通常用于I/O Kit驱动程序，I/O Kit提供处理这种内存的抽象。

正在使用哪一个分页器对高层部分（如VM对象）来说，大体上是透明的。每个VM对象都有一个关联的内存对象，它（通过端口）为当前分页器提供界面。

### 13. Mach中的内存分配

Mach中，执行内存分配的一些基本例程是：

```
kern_return_t kmem_alloc(vm_map_t map, vm_offset_t *addrp, vm_size_t  size);
kern_return_t kmem_alloc_contig(vm_map_t map, vm_offset_t *addrp,
                                vm_size_t size, vm_offset_t mask, int flags);
void kmem_free(vm_map_t map, vm_offset_t addr, vm_size_t size);
```

kmem_alloc()是Mach中获得内存的主要接口。为了分配内存，你必须提供一个VM映射。内核中的大部分工作都已定义了kernel_map，并使其指向kernel_task的VM映射。第二个变体kmem_alloc_contig()与前者不同，它分配物理上而不是虚拟中连续的内存。苹果公司并不建议使用这种类型的分配，因为这在搜索连续的空闲内存块时会造成非常大的性能损失。Mach还提供了kmem_alloc_aligned()函数，该函数分配的内存大小必须为2的幂次，此外还有其他一些通常很少用的变体。kmem_free()函数用来释放分配的内存。一定要注意，传递的VM映射要与分配时使用的相同，大小也要一样。

### 2.1.3 BSD层

Mach只提供一些基本服务，不同的是，BSD层位于Mach和用户应用程序之间，实现许多核心操作系统的功能，这些功能建立在Mach提供的服务之上。在OS X和iOS中，BSD层与处理器一起运行在特权模式中，而不是作为用户任务运行，它原本由Mach项目使用。因此BSD层并没有内存保护，并且运行在与Mach和I/O Kit相同的地址空间中。BSD层源自FreeBSD 5操作系统内核的一部分，本身并不是一个完整的系统，只是FreeBSD 5操作系统的一部分代码。

BSD层提供进程管理、系统调用、文件系统和网络连接等服务。表2-6简要概述了BSD层提供的服务。

表2-6 BSD层服务概述

| 服 务 | 描 述 |
| --- | --- |
| 进程和用户管理 | 支持用户（uid）、组（gid）和进程（pid）ID，以及进程创建（fork）和Unix安全模型。POSIX线程和同步。共享库支持、信号处理 |
| 文件管理 | 文件、管道、套接字和POSIX IPC。VFS、HFS、HFS+、ISO和NFS文件系统。异步输入/输出 |
| 安全 | 安全审计和加密算法，如AES、Blowfish、DES、MD5和SHA-1 |
| 内存管理 | 基于虚拟节点文件的分页器。内存分配措施。统一缓冲区缓存（UBC） |
| 驱动程序 | 各种驱动程序，包括控制台和其他字符设备驱动程序，如/dev/null、/dev/zero/、/dev/random，以及RAM磁盘驱动程序（/dev/md*） |
| 网络连接 | TCP/IP 4&6、DHCP、ICMP、ARP、以太网、路由和防火墙、分组过滤器（BPF）和BSD套接字。I/O Kit中的底层网络驱动程序 |
| 系统调用 | 提供一个API，用于授权用户空间应用程序访问基本/底层内核服务（如文件和进程管理） |

BSD层基于Mach服务提供抽象。例如，BSD层的进程管理和内存管理均基于Mach服务而实现。

**1. 系统调用**

应用程序需要文件系统的服务或希望访问网络时，必须向内核发送系统调用。BSD层可以实现所有的系统调用。当系统调用处理器执行时，内核上下文从用户模式切换到内核模式来处理应用程序的请求，如读取文件。执行这些处理的API被称为系统调用 API，它是传统的Unix API，用来从用户空间调用内核中的函数。可用的系统调用有数百个，从与进程控制相关的调用，如`fork()`和`execve()`，到文件管理调用，如`open()`、`close()`、`read()`和`write()`。

BSD层还提供一个`ioctl()`函数（它本身就是一个系统调用），它是I/O control（I/O控制）的缩写，通常用来向设备驱动程序发送命令。`sysctl()`函数可以设置/获取各种内核参数，包括但不限于调度器、内存和网络子系统。

**提示** 可用的系统调用定义在/usr/include/sys/syscall.h中。

Mach陷阱报文是类似于系统调用的机制，用于跨越内核/用户空间边界。系统调用为应用程

序提供直接服务，与此不同，Mach陷阱报文是从用户空间客户端向内核服务器传递IPC消息。

**2. 网络**

网络是XNU中BSD部分的一个主要子系统。BSD处理网络的大部分功能，如套接字通信的细节和TCP/IP等协议的实现，但与实际硬件设备的底层通信除外，它通常由I/O Kit驱动程序处理。I/O Kit网络驱动程序将与网络栈对接，网络栈负责处理接收的网络设备的缓冲区，检查并确保它们可以到达初启程序，例如Web浏览器。类似地，BSD网络栈将接受应用程序的传出数据，格式化包中的数据，然后将其路由到或分配到适当的网络接口。BSD还实现了IPFW防火墙，根据系统管理员设置的策略，防火墙将过滤发向/来自计算机的数据包。

BSD网络层支持各种网络和传输层协议，包括IPv4和IPv6、TCP、UDP。在高层支持BOOTP、DHCP和ICMP，以及其他一些协议。其他的网络相关功能包括路由、桥接和网络地址转换（Network Address Translation，NAT），以及伯克莱封包过滤器（Berkeley Packet Filter，BPF）对设备层数据包的过滤。

**网络内核扩展**

网络内核扩展KPI（kernel programming interface，内核编程接口）是一种可以对网络栈的各个部分进行扩展的机制。NKE可以定义新的协议，将挂钩或过滤器插入到网络栈的各层。例如，某一应用程序或用户可以创建一个过滤器，截取TCP与某一地址的连接。此外，还可以临时阻塞网络数据包，或者在传输到高/低层之前对网络数据包进行修改。NKE源于苹果，并不是传统BSD网络栈的一部分，但由于NKE的性质，它现在与网络栈紧密相连。第13章将讨论NKE。

**3. 文件系统**

内核内置支持各种不同的文件系统，如表2-7所示。Mac OS X和iOS使用的主要文件系统是HFS+，它是为代替Mac OS 文件系统HFS而开发的。

表2-7　XNU支持的文件系统

| 名　称 | 描　述 |
| --- | --- |
| HFS+ | Mac OS X和iOS使用的标准文件系统 |
| HFS | 旧式Mac OS文件系统 |
| UFS | BSD Unix文件系统 |
| NFS | 网络文件系统 |
| ISO 9660和UDF | CD和DVD使用的标准文件系统 |
| SMB | 服务器消息块，用于连接Microsoft Windows计算机的网络文件系统 |
| AFP | 苹果文件协议 |

在Mac OS X 10.2.2中，HFS+支持日志记录功能。日志记录是在执行事务之前将其记录到日志中，从而提高文件系统的可靠性。这使得文件系统对断电或内核崩溃等事件有更强的恢复能力，因为在重启后数据可以重新执行，保持文件系统的一致性。

HFS+支持非常大的文件，可以长达8 EiB（1 Eib = 260字节），这也是最大卷长度。文件系统完全支持Unicode字符的文件名，并且默认不区分大小写。它还支持Unix风格的文件权限和访问控制表（access control list，ACL）。



**4. 虚拟文件系统**

虚拟文件系统，即VFS，提供特定文件系统的抽象，如HFS+和AFP，可以使应用程序通过一个相容接口访问这些特定的文件系统。VFS可以通过VFS内核编程接口（KPI）把新文件系统轻松地添加为内核扩展，而操作系统无需从整体上了解其实现过程。VFS的基本数据结构是vnode。vnode是文件和目录在内核中的表示方式。内核中活动的每个文件都对应一个vnode结构。

**5. 统一缓冲区缓存**

统一缓冲区缓存（UBC）是一个用于文件的缓存。当写入或读取一个文件时，文件将从后备存储器（如硬盘）加载到物理内存。UBC与VM子系统紧密关联，而且它还缓存VM对象。用于缓存vnode的结构，如代码清单2-1所示。

**代码清单2-1 ubc_info结构**

```
struct ubc_info {
      memory_object_t            ui_pager;       /*分页器*/
      memory_object_control_t    ui_control;     /*分页器的VM控制*/
      uint32_t                   ui_flags;       /*标志*/
      vnode_t                    ui_vnode;       /*该ubc_info的vnode */
      kauth_cred_t               ui_ucred;       /*保存 NFS分页的身份凭据*/
      off_t                      ui_size;        /* vnode的文件大小*/

      struct  cl_readahead*      cl_rahead;      /*聚簇预读上下文*/
      struct  cl_writebehind*    cl_wbehind;     /*聚簇滞后写上下文*/

      struct  cs_blob*           cs_blobs;       /*用于代码签名*/
};
```

在引入UBC之前，系统有两个缓存——页缓存和缓冲区高速缓存。缓冲区高速缓存由设备和块编号确定，块编号对物理设备上的数据块地址，而页缓存则执行内存映射的缓存。

UBC的长度根据系统的需求动态缩减和增长。如果缓存中的文件被修改，则将其标记为脏（dirty）文件，表示缓存的副本不同于磁盘上的原本。脏项被周期性地刷出到磁盘。使用fcntl系统调用的F_NOCACHE选项，用户空间程序可以绕过UBC直接访问磁盘，这对无法从此种缓存中受益的工作负载（如不太可能重用的数据集合）而言，可以提高I/O性能。

## 2.1.4 I/O Kit

XNU的最后一个主要组成部分是I/O Kit，它是一个面向对象的框架，用于编写设备驱动程序及其他内核扩展。I/O Kit对系统硬件进行抽象，预定义了多种类型的硬件基类，让实现新的驱动程序变得非常容易，因为它可以继承基类驱动程序的许多功能，从而实现代码的高度重用。I/O Kit框架包含内核层框架和叫作IOKit.framework的用户空间框架。内核框架采用嵌入式C++（C++的子集）编写而成，而用户空间框架则基于C语言。

I/O Kit包含一个名为I/O Catalog的数据库。I/O Catalog是所有可用I/O Kit类的一个注册表。另一个数据库I/O Registry跟踪I/O Catalog中的类对象实例。I/O Registry中的对象通常表示设备、驱动程序或支持类，是一个层次结构，模仿硬件设备之间的物理连接。例如，USB设备是其连接到的USB控制器的子节点。ioreg命令行实用程序可以查看I/O Registry。

I/O Kit基于三个主要概念：

- 家族
- 驱动程序
- 块

家族表示特定设备类型的一般抽象。例如，IOUSBFamily处理USB相关设备支持的许多技术实现细节。驱动程序负责管理特定的设备或总线。一个驱动程序可以与多个家族关联。就基于USB的存储设备而言，它依赖于IOUSBFamily和IOStorageFamily。块是可控制实体（如PCI或USB设备）的接口，高层驱动程序可以用它与设备通信。

作为一名内核程序员，你的大部分时间可能都在处理I/O Kit，因此本书的大部分内容都会与此相关。第4章将详细讲述I/O Kit。

#### 1. libkern库

与Mach和BSD提供系统交互API不同，libkern库为内核的其他部分，特别是I/O Kit，提供支持例程和类，更确切地说是为内核及内核扩展构建有用的块和实用程序。受限制的C++运行库在libkern中实现，为像new和delete运算符这样的服务提供实现。

除了标准的C++运行库，libkern还提供了许多有用的类，其中最基本的是OSObject，这是I/O Kit中每个类的超类。OSObject支持引用计数，其原理在概念上与用户空间的Cocoa或Cocoa Touch中的NSObject相同。其他相关类包括OSDictionary、OSArray、OSString和OSInteger。这些类和其他一些类也用于提供内核扩展的Info.plist中的字典值。

libkern库并不是只与核心C++类和运行库有关，它还提供了许多C标准库中常见函数的实现。例如printf()和sccanf()函数以及strtol()和strsep()这样的函数。libkern提供的其他函数有散列加密算法（MD5和SHA-1）、UUID生成和Zlib压缩库。libkern库还包含kxld，用于管理动态加载的内核扩展。

最后，也很重要的是，还有OSMalloc()等函数，用于分配内存，实现锁机制及同步原语。

---

**注意**　libkern源代码可以在XNU源代码发布版的libkern/和bsd/libkern目录中找到。

---

#### 2. Platform Expert

Platform Expert包含一个面向系统的抽象层。它有一部分是开源的XNU源代码发布版，其余部分则在com.apple.driver.AppleACPIPlatform内核扩展中实现，对此并没有可用的源代码。Platform Expert处理系统总线的设备枚举和检测，可以看做主板的驱动程序。Platform Expert负责系统启动后I/O Kit设备树（称为I/O Registry）结构的初始化。Platform Expert本身将成为该树的根节点，即IOPlatformExpertDevice。

## 2.2 小结



本章内容如下：

- 概述了Mac OS X和iOS操作系统，讨论了它们的总体背景和来源，并着重讲述了内核，这是本书的主要内容。
- 介绍了XNU内核，这是OS X和iOS的内核。
- 讨论了XNU内核的分层体系结构，其中包含三个主要组成部分：Mach、BSD和I/O Kit。Mach层可以看做内环，最接近硬件，为内核的其他部分提供服务。Mach层提供的服务包括硬件抽象、虚拟内存和任务调度。
- 分析了Mach调度器的操作以及任务和线程之间的区别。任务可以看做一个线程容器，这些线程共享公共内存地址空间及其他资源，如打开的文件。
- 描述了Mach IPC，这是内核及其各层中的通信机制。另外，还剖析了Mach中提供虚拟内存的各部分，即VM映射、VM映射项和VM对象。
- 阐释了分页器的作用和操作。
- 分析了BSD层，它源自FreeBSD操作系统，运行在Mach核心之上，但都位于相同的内核地址空间中。BSD层提供了应用程序与内核通信的接口，最重要的是系统调用。BSD层实现了网络栈，包括TCP/IP和其他协议。BSD层还支持文件系统，如HFS+，这些文件系统基于虚拟文件系统层实现，该层是文件系统的统一接口。
- 展示了I/O Kit，这是一个基于C++的内核框架，用于编写设备驱动程序及其他扩展。libkern库提供了许多实用函数和基础组件，包括I/O Kit构建所用的类，如`OSObject`。

第3章

# Xcode和内核开发环境

苹果公司一直体恤开发者，并为他们提供直观、易于操作的工具和API，以面向Mac和iOS平台进行开发。编写过Mac或iPhone应用软件的开发者都熟悉面向对象的Cocoa框架，该框架提供了很多接口，以支持图形化用户界面和用户应用程序所需的其他服务。类似地，它也为内核开发者提供一些API，这些API用于帮助内核扩展执行任务。对于驱动程序开发，苹果公司提供了I/O Kit，这是一个面向对象的框架，用来与硬件进行交互。下一章将讨论开始进行内核开发时所需的工具及框架，还包括构建、安装一个简单内核扩展的教程。

## 3.1 语言的选择：C++

C语言几十年来都是公认的系统级语言。实际上，开发C语言的最初目的是用作替代语言，专门为Unix系统编写不可移植的汇编代码，是特定于原始Unix系统的汇编代码替代物。XNU内核及Mac OS X的许多核心服务都采用C语言编写，然而，用于开发驱动程序的I/O Kit框架则用C++语言的一个子集编写。苹果公司之所以选择用C++编写I/O Kit是因为它是面向对象的语言，可以将物理硬件连接抽象成驱动程序模型。苹果原本考虑使用基于Objective-C的框架来编写驱动程序，但最终决定选用C++语言。尽管C++语言在应用软件开发中广泛使用，但Mac OS X仍然是支持并且实际上鼓励在其内核中运行C++代码的少数操作系统之一。然而，这并不能说C++代码在Mac OS X内核中就不会出现其他内核中的那些问题。为了避免一些这样的问题，Mac OS X内核代码必须使用具有C++特性的受限子集，即嵌入式C++。不可用的特性包括：

- 异常；
- 多继承；
- 模板；
- 运行时类型信息。

值得注意的是，因为嵌入式C++是标准C++语言的一个子集，所以采用嵌入式C++编写的任何代码都能与普通的C++语言编译器兼容。

虽然从技术方面讲能在内核中包含这些语言特性，但苹果公司决定禁用这些特性，因为它们会大大增加编译的代码量，从而增加内核的内存占用。实现禁用异常功能不仅是为了免于编写额外的代码，而且还因为异常处理捕捉失败时会导致内核恐慌问题（kernel panic）。

尽管禁用了标准运行时类型信息，但I/O Kit会提供其自身的受限实现，这将在下一章讲述。内核开发者还可以访问C++运行时库的受限实现，而模版禁用之后，STL类将不可用。

通常，基于I/O Kit框架开发内核扩展时使用C++语言，而C语言也可以用来进行其他一些开发，如文件系统及底层网络代码的实现。



## 3.2 Xcode

为了开发内核扩展，需要安装苹果公司的开发工具Xcode，这可以从Mac App Store购买。安装Xcode包时，名为“Developer”的目录会添加到硬盘的根级别中，其中有Mac OS X和iOS开发需要的所有资源，包括：

- 集成开发环境（Xcode应用程序）；
- C、C++、Objective-C编译器；
- 源代码调试器；
- 用于内核及应用程序开发的API和头文件；
- 分析工具，用于测量代码执行时间，确定性能瓶颈；
- 实用程序，用于检测连接到系统上的硬件设备及为每个设备加载的驱动程序。

这些工具中，Xcode应用程序在内核扩展开发中最耗时，因为它为编译器提供源代码编辑器及前端。Xcode 4的用户界面如图3-1所示。

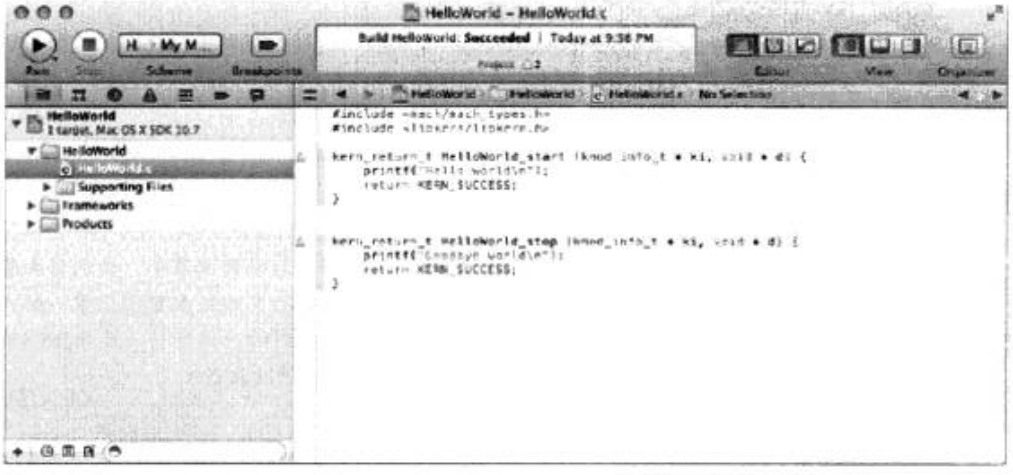



图3-1 Xcode 4用户界面

在底层，Xcode是命令行编译器和调试器的一个前端。实际上，完全可以绕过Xcode，在命令行上直接调用GCC来构建内核扩展。但在下一节你会看到，Xcode提供了一些工程模版，为构建内核扩展传递适当的编译器标志。

先前版本的Xcode使用GCC编译器，然而，从Xcode 4开始，一个基于LLVM的现代编译器成为默认的编译器。LLVM编译器是苹果公司的一个开源项目，支持Objective-C、C和C++。LLVM的目标是，通过提供更清晰的警告及错误信息，使句法高亮，并由编译器自身执行的语义分析来驱动代码完成，从而实现比GCC更快的编译时间，以及与IDE（如Xcode）的更紧密结合。

**注意**　关于Xcode的更多信息，包括如何获得Xcode，可以在http://developer.apple.com/xcode中找到。

## 3.3　“Hello World”内核扩展

为了进行内核编程，首先实现一个非常简单的例子，即非常受欢迎的“Hello World”应用程序，在我们的例子中就是“Hello World”内核扩展。首先，启动Xcode，在欢迎界面中选择Create a new Xcode project，新工程的模版列表会显示出来。如果你选择了System Plug-in类别，Xcode将提供Generic Kernel Extension和I/O Kit Driver的模版。虽然这两种模版都可以创建内核扩展，但I/O Kit驱动程序需要我们指定一个与其匹配的硬件设备，并且仅在该设备存在时加载。另一方面，通用内核扩展并不是一个硬件扩展驱动程序，可以由用户随时加载。

对于该演示示例，我们将创建一个基于Generic Kernel Extension模版的工程，选择该项，并点击Next按钮。这时要求我们输入一个产品名称和公司标识符。该产品名称对应用户看到的可执行文件名，因此在这个例子中我们将使用HelloWorld产品名称。公司标识符应该是一个反向DNS类型的字符串，是你或者你的公司注册的域名。在本例中，可以免费使用com.osxkernel，这是专门为本书注册的一个域名。将一个产品名称附加到公司标识符上，Xcode可以创建一个字符串，保证该工程的唯一性，不与现有的内核扩展名称冲突（该示例的唯一标识符为`com.osxkernel.HelloWorld`）。

**注意**　Mac OS X中，需要唯一标识符的地方都要使用反向DNS。在后面的章节中，我们将会看到I/O Kit使用类似的方案来确保C++类名的唯一性。Mac OS 9之前的版本都使用唯一的四字符常量来标识应用程序，这需要开发者向苹果公司注册他们的选择字符。反向DNS使开发者可以创建他们自己的唯一标识符，而无须向苹果公司进行注册。

点击Create按钮之后，Xcode将为你生成一个工程，其中包括一个名为`HelloWorld.c`的实现文件。可以点击工程窗口左边部分中该文件的图标，查看该源文件的内容。对于该演示示例，修改生成的源代码，使之包含头文件<libkern/libkern.h>，并增加两个`printf()`调用。当内核扩展加载和卸载时，将分别调用名为`HelloWorld_start()`和`HelloWorld_stop()`的函数。该文件编辑完成时，代码看起来应该如代码清单3-1所示。

**代码清单3-1 HelloWorld.c的演示示例**

```
#include <mach/mach_types.h>
#include <libkern/libkern.h>

kern_return_t   HelloWorld_start (kmod_info_t * ki, void * d) {
    printf("Hello world\n");
    return KERN_SUCCESS;
}


kern_return_t   HelloWorld_stop (kmod_info_t * ki, void * d) {
    printf("Goodbye world\n");
    return KERN_SUCCESS;
}
```

第1章中提到，用于编写内核代码的API与用于用户应用程序的API不同，printf()等函数甚至也是这样。内核并不把用户空间的头文件<Stdio.h>包括在内，而是有自己的printf实现，在<libkern/libkern.h>头文件中声明。如果你试图在内核工程中包含<stdio.h>，编译器将报告无法找到包括的头文件。

除了包括定义printf()函数的头文件，还需要将内核扩展程序与提供printf实际实现的库相链接。这不是编译时链接，内核仅在内核扩展加载时才解决内核扩展的任何库依赖。为了通知内核我们的依赖，需要在内核扩展的属性列表中声明希望链接的库，在该演示示例中为HelloWorld-Info.plist文件。

为了修改该属性列表，在工程窗口中点击HelloWorld-info.plist文件。尽管该文件的格式是基于文本的XML，但Xcode包含一个用来操作属性列表文件的图形编辑器，如图3-2所示。将一个新项添加到属性列表的OSBundleLibraries字典中，以包含键名为com.apple.kpi.libkern、值为9.0.0的项。添加完成之后，属性列表看起来应该如图3-2所示。

图3-2 Xcode中的图形属性列表编辑器

与我们向属性列表文件中添加的项相对应的XML代码，如代码清单3-2所示。

**代码清单3-2　演示内核扩展的OSBundleLibraries项的值**

```
<key>OSBundleLibraries</key>
<dict>
        <key>com.apple.kpi.libkern</key>
        <string>9.0.0</string>
</dict>
```

工程的属性列表并不用于编译器（不是执行某个预处理，预处理是用它们的实际值替换${PRODUCT_NAME}等变量），而是供内核使用。该属性列表将被复制到编译的内核扩展中，并在内核扩展加载时进行读取。我们添加到该字典中的项由一个键-值（Key-value）对组成。键标识我们依赖的内核库，值对应于该库的最低版本。在该例中，我们告诉内核所需库的唯一标识符为com.apple.kpi.libkern，且是9.0.0及其之后的版本。库的标识符使用一个反向DNS前缀，确保该名称的唯一性，在本例中，有了前缀com.apple我们可以将该库看做苹果提供的标准库。

**提示**　我们例子中的库版本9.0.0是Mac OS X内核的版本，而不是Mac OS X的版本。9.0.0版本对应Mac OS X 10.5.0。可以在Terminal中输入uname-r命令，查看你机器上的内核版本。

**注意**　你可能已经注意到，通过Xcode模版创建的工程在Frameworks组中包含一个名为Kernel.framework的项。编译该工程时，链接器并不使用该项，该项只用于开发者访问内核头文件的快捷方式。

现在该内核扩展工程已经完成，可以开始构建。为此，从Project菜单中选择Build项。你应该接收不到任何编译错误，但是如果有错误，请确认HelloWorld.c文件的内容是否与代码清单3-1一致。

运行该内核扩展之前，有必要花一些时间来理解内核如何知道调用哪些入口点。假定该源文件中包含的这两个函数均由用户定义。你可能猜到了，Xcode会帮助自动生成一些样板代码。在生成的代码中，Xcode使用工程设置中定义的两个值，它们定义了内核扩展程序的启动和终止例程。这两个值如图3-3所示。只要改变源代码中定义的函数名和工程编译设置中的值，就可以随意重命名HelloWorld_start和HelloWorld_stop的入口点。

成功编译工程之后，Xcode将创建一个名为HelloWorld.kext的内核扩展。这个文件将被打包成一个特殊的文件，名为KEXT程序包。也许你并不熟悉程序包，其实它们本质上就是一个目录，这个目录包含执行程序需要的所有文件，但Finder展现给用户的只是单个文件。我们编译“Hello World”内核扩展时创建的程序包内容如代码清单3-3所示。

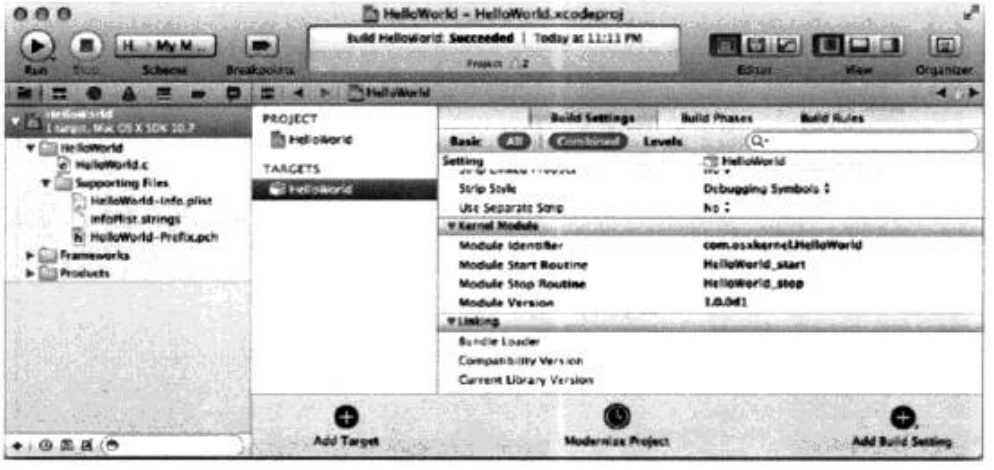



图3-3 Hello World内核扩展的工程设置

**代码清单3-3 HelloWorld.kext程序包的内容**

```
HelloWorld.kext/
HelloWorld.kext/Contents/Info.plist
HelloWorld.kext/Contents/MacOS
HelloWorld.kext/Contents/MacOS/HelloWorld
HelloWorld.kext/Contents/Resources
HelloWorld.kext/Contents/Resources/en.lproj
HelloWorld.kext/Contents/Resources/en.lproj/InfoPlist.strings
```

你应该熟悉Info.plist文件，因为这是前面修改的属性列表的一个副本（带有一些Xcode的小处理）。值得一提的另一个文件是HelloWorld，位于程序包Contents/MacOS路径下的子目录中。这个文件包含内核扩展的实际可执行代码。

## 3.4 加载和卸载内核扩展

内核扩展是操作系统内核中运行的一个代码模块。我们已经构建了自己的内核扩展，现在需要将其加载到内核中运行。虽然Xcode在编写和构建内核扩展方面非常强大，但它却无法测试和调试内核扩展。实际上，对于内核扩展，Xcode窗口中的Run按钮只可以构建工程，并不能实际加载或运行输出结果。而Mac OS X上的内核扩展程序可以通过下列两种方式中的任意一种进行加载：或者将内核扩展程序包复制到/System/Library/Extensions目录下自动加载，或者通过命令行手动加载。为了加载内核扩展程序，首先需要定位由Xcode构建的经过编译的二进制文件。默认情况下，Xcode 4将编译器的输出放置在一个不同的位置，而不是在包含源代码的工程目录中，这使查找内核扩展路径并在命令行中对其加载变得非常困难。为了找到Xcode编写的内核扩展路径，右键点击HelloWorld.kext产品，会弹出一个上下文菜单，选择Show in Finder项，如图3-4所示。

图3-4　查找所创建内核扩展的路径

复制到/System/Library/Extensions目录下的内核扩展将在操作系统需要时进行加载。可能是系统启动时，或对驱动程序而言，是需要该驱动程序的硬件设备连接到计算机上时。但在开发过程中，从命令行手动加载内核扩展通常更便捷。

对于安全性，因为内核扩展被授予了与核心操作系统代码相同的高级权限，所以内核扩展程序只可以由以管理员权限访问系统的用户进行安装或加载。更进一步的安全措施是，系统对内核扩展程序包的权限要求非常严格，它将拒绝加载还没有满足这些要求的内核扩展，特别是以下情况：

- KEXT程序包及其包含的所有文件和文件夹都必须由root用户（即用户id是0）所有；
- KEXT程序包及其包含的所有文件和文件夹都必须由wheel组（即组id是0）所有；
- KEXT程序包及其包含的任何目录的权限掩码都必须是0755（rwxr-xr-x）；
- KEXT程序包中的所有文件的权限掩码都必须是0644（rw-r--r--）。

在Xcode中构建内核扩展时，生成的KEXT程序包及其内容都会有正确的权限掩码，但用户及组的所有权属于运行编译器的用户。如果要根据内核扩展的需要修改文件的所有权，可以在Terminal上使用如下命令。

```
sudo chown -R root:wheel HelloWorld.kext
```

请注意，如果在Xcode构建目录中改变KEXT的所有权，下次构建工程时，Xcode将没有足够权限重写KEXT，这会导致构建错误。为了解决该问题，可以在改变KEXT所有权和加载它之前，将其从Xcode编译目录复制到另外一个目录（如/tmp）。

Mac OS X包含了许多处理内核扩展的命令行实用程序。下面是一些常用的命令。

- kextload：将KEXT加载到内核。
- kextunload：停止加载的KEXT，并将其从内核中卸载。
- kextutil：面向开发者的一个实用程序，将KEXT加载到内核，并可以提供诊断信息，判断内核加载失败的原因，还可以生成一些对调试活动的内核扩展有用的符号。
- kextstat：显示所有加载到内核中的KEXT列表。kextstat并不主动修改内核的状态，除此之外，所有这些命令都必须使用超级用户的权限运行，这可以通过在命令前面加上sudo实现。

我们现在开始加载“Hello World”内核扩展。为此，在Terminal中运行下面的命令。

```
sudo kextload HelloWorld.kext
```

尽管已经加载了“Hello World”内核扩展程序，也调用了其开始的入口点，但在终端窗口中仍然看不到printf的调用结果。相反，printf内核实现的调用输出结果会写入一个日志文件。为了确认“Hello World”内核扩展已被加载，可以使用kextstat命令，如下所示：

```
kextstat
```

这将会输出一个正在运行的内核扩展列表。由于“Hello World”扩展是最新加载的一个扩展，它应该显示在列表的最后。kextstat输出结果的一个示例如代码清单3-4所示。

**代码清单3-4 Kextstat命令的输出结果，高亮部分是内核扩展**

```
Index  Refs   Address            Size       Wired      Name (Version) <Linked Against>
  1      85  0xffffff7f80742000 0x683c     0x683c     com.apple.kpi.bsd (11.1.0)
  2       6  0xffffff7f8072e000 0x3d0      0x3d0      com.apple.kpi.dsep (11.1.0)
  3     110  0xffffff7f8074c000 0x1b9d8    0x1b9d8    com.apple.kpi.iokit (11.1.0)
  4     116  0xffffff7f80738000 0x9b54     0x9b54     com.apple.kpi.libkern (11.1.0)
  5     103  0xffffff7f8072f000 0x88c      0x88c      com.apple.kpi.mach (11.1.0)
...
130       0  0xffffff7f810ce000 0x51000    0x51000    com.apple.filesystems.afpfs (9.8) <129 7 6 5 4 3 1>
144       0  0xffffff7f807b8000 0x2000     0x2000     com.osxkernel.HelloWorld (1) <4>
```

最后，我们将卸载“Hello World”扩展，即调用HelloWorld_stop()函数从内核中卸载内核扩展程序。使用下面的命令可以完成：

```
sudo kextunload HelloWorld.kext
```

## 3.5 使用控制台查看输出

内核中调用printf()输出的结果将写入到硬盘上的一个日志文件中。该日志是纯文本文件格式，可以使用标准的Unix命令tail和cat查看，其路径为/var/log/kernel.log。另外，该日志文件的内容可以使用Mac OS X中的 Console应用程序查看，该应用程序位于/Applications/Utilities目录中。Console应用程序将合并各种系统服务和应用程序的日志，包括内核日志文件。我们的演示示例在Console中输出结果的屏幕截图如图3-5所示。

图3-5　成功加载和卸载内核扩展的输出结果在Console实用程序中显示

如果你习惯用户空间开发的源代码级别的调试器，这看起来可能比较原始，但是将调试结果输出到控制台，仍然是内核代码的基本调试技术之一。关于调试的更多信息，请参见第16章。

## 3.6　小结

Mac OS X的内核扩展使用苹果公司提供的Xcode包中的工具开发。本章创建了“Hello World”内核扩展示例，演示了从其他内核库导入符号的方式，并介绍了常用来加载和处理内核扩展的命令行实用程序。

# 第4章 I/O Kit框架



Mac OS X的设备驱动程序使用名为I/O Kit的框架开发。I/O Kit包含的头文件和库中，一部分为驱动程序提供服务，另一部分可以在用户空间代码中使用，以定位内核驱动程序并与其交互。I/O Kit有两个主要部分：

- Kernel.framework
- IOKit.framework

尽管多少有点违反直觉，但I/O Kit框架面向用户空间应用程序而不是为开发内核中的I/O Kit驱动程序而设计。用于内核空间驱动程序开发的头文件包含在Kernel框架中。查看Kernel框架的内容，会发现一个名为IOKit的目录，其中包含用于内核空间驱动程序开发的头文件。Kernel框架中另一个重要的目录为libkern，它包含基础类和类型，内核I/O Kit框架就在此基础上构建。

用户空间I/O Kit框架有两个用途。一个是为用户应用程序提供函数，这些函数可用于检测机器上运行的硬件设备，为特定的设备查找驱动程序，以及为相应的驱动程序发送控制请求及请求状态。相关内容将在第5章进一步讨论。另一个用途是支持用户空间应用程序直接与某些硬件设备通信，摆脱对内核驱动程序的依赖。这对某些特定的设备是可行的，尤其是无需在多个运行的应用程序之间共享的USB和FireWire设备。我们会在第15章讨论I/O Kit这方面的内容。

## 4.1 I/O Kit 模型

I/O Kit是面向对象的框架，因此，它需要支持面向对象编程方式的语言。苹果用C++实现I/O Kit，因此，为Mac OS X编写的驱动程序要用C++开发。

在所有操作系统之中，选择C++开发驱动程序非常少见，但这正体现了Mac OS X操作系统的现代性。Mac OS X最早在2001年发布，苹果借此机会为驱动程序的开发设计了一个全新的模型。选择C++反映了设计I/O Kit时计算机硬件和编译器发展的状态。

选择面向对象的语言有许多优点。计算机系统中的硬件，就其本质而言，是一系列通过许多不同的总线互相连接的设备。例如，USB设备可以连接到键盘内置的USB集线器上，USB集线器进而又连接到iMac的USB端口上。在内部，USB端口由iMac主板上的USB控制芯片处理。USB控制芯片则通过内部的PCI总线连接到主板的芯片集控制器上。采用面向对象的驱动模型，I/O Kit能够通过驱动程序对象反映硬件之间相同的连接关系。

驱动程序的作用是使操作系统——最终是用户——可以利用硬件实现的服务。当检测到硬件设备时，操作系统将加载该设备的驱动程序，使驱动程序可以访问硬件设备，并与其交互，并且为驱动程序提供接入口点，将其服务插入到操作系统中。

---

**注意**　I/O Kit使用“块”（nub）这个术语来描述为其他驱动程序提供服务的驱动程序。例如，USB hub的驱动程序就是一个块，因为它为与其相连的USB设备驱动程序提供服务。

---

选择面向对象的设计可以很好地满足I/O Kit的需要。每个驱动程序都实现为一个C++类，使I/O Kit可以为系统中每个硬件设备实例创建一个新的驱动程序对象。驱动程序的硬件设备通过驱动程序的“提供者”（provider）对象进行访问，该对象在初始化阶段提供给驱动程序。I/O Kit使用的提供者类适合设备使用的硬件总线。例如，USB设备的提供者类是IOUSBDevice的一个实例，而PCI卡驱动程序的提供者类是IOPCIDevice的一个实例。这些总线接口的不同功能被抽象成各种类型的提供者类。例如，USB设备有许多用于数据传输或读取的端点，因此，IOUSBDevice类包含将数据缓冲区读/写到指定端点的方法。另一方面，PCI卡通过将一组寄存器映射到内核地址空间进行访问，随后驱动程序采用与其写入到任何其他内存地址相同的方式，读写这些寄存器。

最后，驱动程序需要一种将其服务提供给操作系统其他部分的方法。这也许正好是I/O Kit面向对象设计的长处。驱动程序的主类可以实现为一个特定类的子类，该特定类由I/O Kit提供给某种类型的驱动程序。例如，串行端口的驱动程序是标准IOSerialStreamSync类的一个子类。类似地，音频输出设备的驱动程序是IOAudioDevice类的一个子类。

通过子类化实现一个驱动程序的好处是驱动程序可以继承父类行为和实现。某些操作通用于每个串行端口和音频设备，这一行为可以由超类实现，无需驱动程序开发者编写样板文件代码。开发者从而可以专注于其特定硬件设备的代码。需要操作特定设备时，超类将调用该驱动程序。

如果你为其他操作系统写过驱动程序，毫无疑问，你知道必须实现一个派遣例程，通常要有一个庞大的switch语句来处理操作系统可能发出的所有请求，然后调用驱动程序中适当的函数，实现该请求。而I/O Kit采用了一种不同的方案。驱动程序请求采用方法调用的形式。驱动程序只需实现或重载其超类的方法，而这些方法只用于特定类型的驱动程序。例如，串口驱动程序用于在串行端口上传输和接收字节，因此需要执行这些操作时，IOSerialStreamSync的子类需要实现其提供的虚拟方法enqueueData()和dequeueData()。

对于专业设备，I/O Kit可能无法提供合适的超类。例如，I/O Kit中不存在合适的超类，用于实现专业医学成像设备的驱动程序。这种设备的驱动程序由通用的IOService类的一个子类实现。IOService提供了管理驱动程序生命周期的方法，包括初始化及销毁该驱动程序对象。

最后，驱动程序会为用户空间应用程序提供一个接口。在I/O Kit术语中，这通过实现一个称为“用户客户端”（user client）的类处理。用户客户端是驱动程序实现的一个自定义类，它是IOUserClient类的子类。每当应用程序连接到驱动程序时，I/O Kit将实例化一个新的用户客户端对象，处理该应用程序连接到驱动程序的所有请求。当应用程序断开与驱动程序的连接，或者应

用程序终止（或崩溃）时，该用户客户端将被销毁。如果应用程序打开多个与驱动程序的连接，I/O Kit将实例化相应数量的用户客户端对象。

## 4.2 对象关系

我们已经看到，I/O Kit驱动程序有两个重要的类，一个是主驱动程序类继承的超类，另一个是驱动用于访问硬件的提供者类。这种设计意味着，驱动程序实现的功能与驱动程序的硬件设备连接到计算机的方式相分离。例如，支持PCI声卡的驱动程序和支持USB音频输出设备的驱动程序都继承自相同的IOAudioDevice超类，操作系统通过对每个驱动程序进行相同的调用便可以连接到这两个驱动程序。毕竟，操作系统的音频子系统不需要关心音频输出设备如何连接到计算机上。

这种分离还有利于代码重用。基于PCI和基于USB的音频设备厂商可以对这两种设备使用相同的驱动程序，根据连接到计算机上的设备类型，该驱动程序可以接收IOPCIDevice或IOUSBDevice类型的提供者类。或者，更理想的是，硬件厂商能创建自己的超类，实现这两种设备的共同功能，该超类本身是IOAudioDevice的一个子类。厂商的PCI设备和USB设备的驱动程序只需要极少的实现，而大部分的共通功能都来自它们自定义的超类。这种结构如图4-1所示。

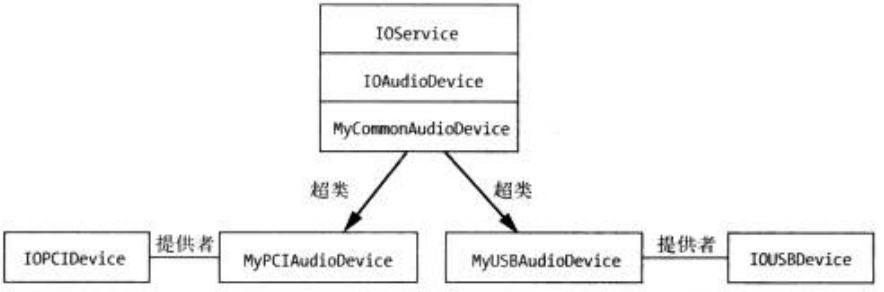



图4-1 I/O Kit驱动程序及其超类和提供者类之间的关系示例

## 4.3 Info.plist 文件

Macintosh平台安装了设备驱动程序之后，无须任何配置，便可以一直支持设备即插即用。借助I/O Kit，Mac OS X亦是如此。上一章开发的内核扩展添加到系统后会立即进行加载，与此不同，驱动程序仅在其支持的设备连接到计算机时才进行加载。采用这种方式，即使在系统上安装了上百个驱动程序，也只有实际连接到计算机上硬件对应的驱动程序才会被加载，并占用内存。

第3章提到内核扩展需要一个属性列表文件定义其入口点。该性能列表对实现I/O Kit驱动程序的内核扩展程序尤其重要。对于I/O Kit驱动程序来说，该性能列表指定了驱动程序支持的硬件设备。驱动程序支持的每个设备在属性列表中都包含其自己的“个性”（personality），其中所包含的“匹配字典”（matching dictionary）由一个描述每个匹配硬件设备的数组组成。驱动程序仅

在其匹配字典中描述的硬件设备连接到计算机时，才进行加载。

每个匹配字典中都包含一个最重要的值IOProviderClass，该值定义了驱动程序提供者的类类型，如IOUSBDevice或IOPCIDevice。每当一个新硬件连接到计算机时，I/O Kit就会为该设备创建一个适当的块，然后开始为该块查找适合的驱动程序。以下面USB设备的连接处理为例。

(1) 用户将一个USB设备连接到计算机。

(2) 创建一个新的IOUSBDevice实例，表示该设备。

(3) I/O Kit迭代所有包含匹配字典的驱动程序，列出IOUSBDevice的提供者类。

(4) IOUSBDevice检查驱动程序的匹配字典中的全部内容。

(5) 如果所需的匹配字典属性与该设备的属性相对应，则将该驱动程序添加到该设备的潜在驱动程序列表中。

重要的是，驱动程序是否适合特定硬件设备由提供者类决定。提供者类通过检查潜在驱动程序匹配字典中的属性来实现，但使用的特定属性仅针对具体的驱动程序家族。例如，USB驱动程序可能匹配一个USB设备的特定厂商ID和产品ID，也可能匹配USB键盘等通用设备。一个PCI设备可能与设备的PCI配置空间中特定的厂商ID和设备ID匹配，也可能与任意PCI类相匹配，如网卡或显卡。

经过上面的步骤，I/O Kit已经将设备的驱动程序列表缩小至一个潜在的匹配数组。为了确定最适合该硬件设备的驱动程序，I/O Kit对每个驱动程序使用一个名为“检测积分”（probe score）的概念。每个驱动程序都指定一个检测积分，在某种程度上测量该驱动程序与设备的匹配程度。最终选定检测积分最高的驱动程序处理该设备。

驱动程序的检测积分可以通过两种方式进行设置。一种是在驱动程序的匹配字典中为其提供一个检测积分。例如，生产USB键盘的公司可以提供一个驱动程序，它的匹配字典匹配公司键盘的具体产品ID，检测积分高于系统中默认的键盘驱动程序。另一种方式是通过“主动匹配”（active matching），其中I/O Kit实例化每个潜在的驱动程序，并将其临时装载到硬件设备上，为其提供询问设备、确定检测积分的机会。在检测过程中，驱动程序可以完全访问硬件，因此它可以根据需要执行设备询问，确定其是否适合驱动该设备。驱动程序可以校正它的检测积分，更为常见的是，驱动程序在确定它无法处理连接的硬件时，可以使用检测方法终止对设备进行匹配。

例如，驱动程序执行检测时可以确定设备上加载的固件版本，如果固件版本比驱动程序支持的版本新，就会拒绝加载。比起之后已经将该驱动程序选为设备的驱动程序时再失败，在检测阶段失败更高效，因为I/O Kit无须继续启动你的驱动程序，尽管在这两种情况下，I/O Kit仍将继续尝试加载检测积分第二高的驱动程序。

虽然在几乎所有情况下，一个设备只需安装一个驱动程序，但I/O Kit允许为单个设备加载多个驱动程序。通过向匹配字典中添加一个额外的键“匹配类别”（match category），I/O Kit将加载每个匹配类别中检测积分最高的驱动程序，并将其装载到设备上。如果驱动程序的匹配字典中没有与之对应的匹配类别，则将其假定为默认类别。

该匹配过程为递归匹配，并且驱动程序本身可能就是块，充当其他类的提供者。例如，实现USB主机控制器的PCI卡驱动程序将匹配IOPCIDevice，但会创建它自己的IOUSBDevice实例来表示已经连接到其端口上的设备。采用这种方式，驱动程序创建的IOUSBDevice实例进而成为其他驱

动程序的提供者类，如图4-2所示。这种类型的驱动程序不是直接实例化IOUSBDevice类型的类，而是会提供继承自IOUSBDevice的自身类实现，例如图4-2中的MyUSBDevice。

使用MyUSBDevice实例作为提供者的任意驱动程序将通过其标准的IOUSBDevice接口与提供者通信，但虚拟方法可以使MyUSBDevice实现覆写这些方法，提供其自己的实现。

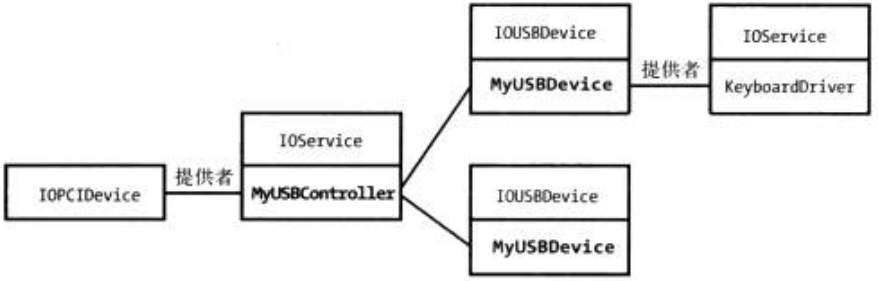



图4-2 也可以作为块的驱动程序，创建的对象作为其他驱动程序提供者类

驱动程序作为块的另一个例子是接受用户应用程序连接的驱动程序（实际上这是一个更常见的场景）。正如前面提到的，对于用户应用程序与驱动程序的每个连接，I/O Kit都会实例化一个称为“用户客户端”的新对象，处理来自应用程序的控制请求，并将该请求传递给驱动程序。类似于主驱动程序类，用户客户端类也是一个I/O Kit服务，与任何其他的驱动程序对象一样，都继承自IOService基类。每个用户客户端都使用主驱动程序对象作为其提供者类。但与主驱动程序不同，用户客户端无须经历匹配阶段，因为驱动程序指定了其用户客户端的具体类名。

**代码清单4-1 假定的外部磁盘驱动程序的个性信息，其中包含FireWire和USB连接的匹配字典**

```
<key>IOKitPersonalities</key>
<dict>
        <key>MyExternalDiskFireWire</key>
        <dict>
                <key>CFBundleIdentifier</key>
                <string>com.mycompany.driver.MyExternalDiskDriver</string>
                <key>IOClass</key>
                <string>com_mycompany_driver_MyExternalDiskDriver</string>
                <key>IOProviderClass</key>
                <string>IOFireWireUnit</string>
                <key>Unit_SW_Version</key>
                <integer>1111</integer>
                <key>Unit_Spec_ID</key>
                <integer>2222</integer>
        </dict>
        <key>MyExternalDiskUSB</key>
        <dict>
                <key>CFBundleIdentifier</key>
                <string>com.mycompany.driver.MyExternalDiskDriver</string>
                <key>IOClass</key>
                <string>com_mycompany_driver_MyExternalDiskDriverUSB</string>
```

```
                <key>IOProviderClass</key>
                <string>IOUSBDevice</string>
                <key>idProduct</key>
                <integer>3333</integer>
                <key>idVendor</key>
                <integer>4444</integer>
                <key>IOProbeScore</key>
                <integer>9000</integer>

        </dict>
</dict>
```

代码清单4-1展示了一个假定的外部磁盘设备的匹配字典，主要针对于FireWire和USB连接。因而，该设备的匹配字典包含两项：第一项匹配特定的FireWire设备，另一项匹配特定的USB设备。每当软件版本为1111、标识ID为2222的FireWire设备插入计算机时，都会实例化驱动程序类com_mycompany_driver_MyExternalDiskDriver，并检测该设备。同样地，每当产品ID为3333、厂商ID为4444的USB设备插入到计算机时，都会实例化驱动程序类com_mycompany_driver_MyExternalDiskDriverUSB，并检测该设备。USB设备的默认检测积分为9000，是USB设备的首选驱动程序。

### 4.3.1　驱动程序类

我们在上一节中已经看到，I/O Kit加载驱动程序时，通过实例化一个驱动程序属性列表中指定的类来实现。该类一定是IOService的子类，要么直接是IOService的子类，要么是IOService子类的子类。IOService类提供了许多虚拟方法，可以在驱动程序生命周期的各处调用，例如，驱动程序加载及初始化时、驱动程序应该检测其提供者时，以及驱动程序停止时。因为这些方法在IOService类的定义中都声明为虚拟方法，所以它们在继承自IOService的自定义驱动程序类中很容易被覆写。

此时是将所学内容付诸实践的好时机，我们来创建一个简单的I/O Kit驱动程序。首先打开Xcode，创建一个基于IOKit Driver模版的新工程。当提示输入产品名称时，输入IOKitTest。可以使用公司标识符com.osxkernel，这是专门为本书注册的一个域名。Xcode创建的工程包含两个文件，一个是名为IOKitTest.cpp的C++实现文件，另一个是其对应的头文件IOKitTest.h。

首先声明驱动程序的类定义。假定实现的是一个通用驱动程序，而不提供特定功能（如串行端口或磁盘存储）的驱动程序，那么我们将该驱动程序主类定义为IOService的一个子类，而不是I/O Kit提供的特定类。将代码清单4-2的代码输入到IOKitTest.h中。

**代码清单4-2　IOKitTest.h演示示例**

```
#include <IOKit/IOService.h>

class com_osxkernel_driver_IOKitTest : public IOService
{
        OSDeclareDefaultStructors(com_osxkernel_driver_IOKitTest)
```

```
public:
        virtual bool    init (OSDictionary* dictionary = NULL);
        virtual void    free (void);

        virtual IOService*      probe (IOService* provider, SInt32* score);
        virtual bool    start (IOService* provider);
        virtual void    stop (IOService* provider);
};
```

该头文件的内容应该非常简单，除了有时候没有OSDeclareDefaultStructors宏之外。回想一下第3章的内容，I/O Kit采用C++语言的一个子集实现，并不包括异常处理和运行时类型信息。因为这两个限制，所以此处的OSDeclareDefaultStructors宏必不可少，它提供了该类的构造函数和析构函数声明，以及元数据，该元数据提供I/O Kit版本运行时类型信息的自定义实现。我们稍后将深入探讨该内容。



**注意** 你可能很奇怪，为什么我们的类要使用这么复杂的名称。内核有一个全局的命名空间，任何活动的内核扩展导出的所有符号（包括类名、函数和全局变量）都要加载到其中。如果一个扩展包含的符号与已经加载的扩展相冲突，那么内核将会拒绝加载该扩展，因此为了避免该问题，苹果建议使用反向DNS命名方案来命名全局函数、类和变量。

驱动程序类的实现应该放在IOKitTest.cpp文件中。该文件的内容如代码清单4-3所示。

**代码清单4-3 IOKitTest.cpp演示示例**

```
#include "IOKitTest.h"
#include <IOKit/IOLib.h>

//定义超类
#define super IOService

OSDefineMetaClassAndStructors(com_osxkernel_driver_IOKitTest, IOService)


bool com_osxkernel_driver_IOKitTest::init (OSDictionary* dict)
{
        bool res = super::init(dict);
        IOLog("IOKitTest::init\n");
        return res;
}

void com_osxkernel_driver_IOKitTest::free (void)
{
        IOLog("IOKitTest::free\n");
        super::free();
}

IOService* com_osxkernel_driver_IOKitTest::probe (IOService* provider, SInt32* score)
{
        IOService *res = super::probe(provider, score);
```

```
        IOLog("IOKitTest::probe\n");
        return res;
}

bool com_osxkernel_driver_IOKitTest::start (IOService *provider)
{
        bool res = super::start(provider);
        IOLog("IOKitTest::start\n");
        return res;
}

void com_osxkernel_driver_IOKitTest::stop (IOService *provider)
{
        IOLog("IOKitTest::stop\n");
        super::stop(provider);
}
```

注意　I/O Kit习惯定义一个名为“super”的宏作为当前类的超类名称。这使一个方法可以轻易地委托给超类实现，如代码清单4-3所示。

最后，我们需要通过Info.plist文件定义该驱动程序的匹配字典和库依赖。将一个新的字典键IOKitTest添加到IOKitPersonalities字典中，该字典包含的值如下所示。

| 键 | 类　型 | 值 |
|---|---|---|
| CFBundleIdentifier | 字符串 | com.osxkernel.${PRODUCT_NAME:rfc1034identifier} |
| IOClass | 字符串 | com_osxkernel_driver_IOKitTest |
| IOMatchCategory | 字符串 | com_osxkernel_driver_IOKitTest |
| IOProviderClass | 字符串 | IOResources |
| IOResourceMatch | 字符串 | IOKit |

我们还需要向OSBundleLibraries字典中添加如下两项。

| 键 | 类　型 | 值 |
|---|---|---|
| com.apple.kpi.iokit | 字符串 | 9.0.0 |
| com.apple.kpi.libkern | 字符串 | 9.0.0 |

该工程的属性列表最终如图4-3所示。

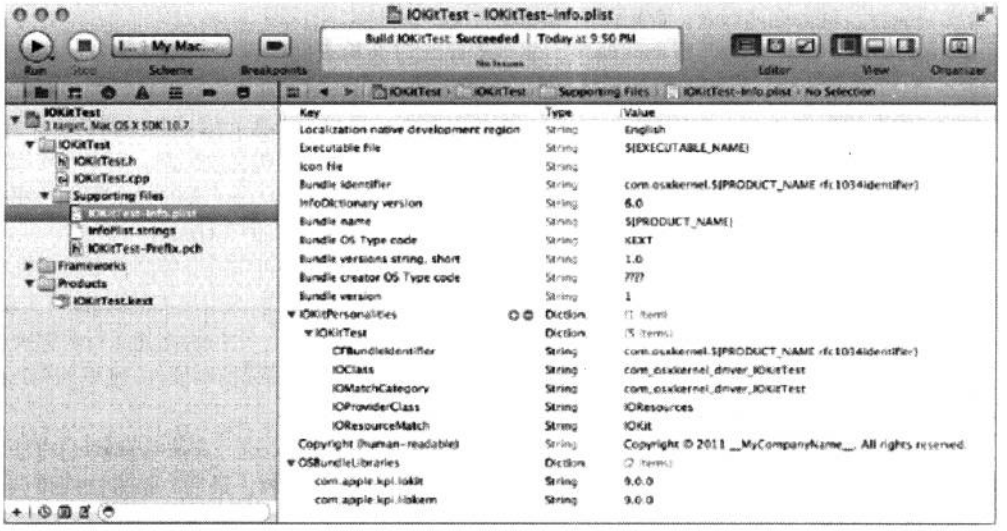



图4-3 IOKitTest演示示例的属性列表，其中包含匹配字典

考虑到驱动程序的用途是为了控制硬件，并且I/O Kit仅在检测到硬件设备时才加载驱动程序，你可能想知道如何才能测试该驱动程序。幸好，I/O Kit提供一个称为IOResources的特殊块，可以用作无硬件设备驱动程序的提供者类，如此处的示例驱动程序。在系统中，存在多个与IOResources块匹配的驱动程序，因此为了将多个驱动程序装载到IOResources上，必须在驱动程序的匹配字典中定义IOMatchCategory键。

由于I/O Kit允许一个块分别对每个匹配类别装载一个驱动程序，因此，指定唯一类别可以加载驱动程序，而且加载后不会妨碍其他驱动程序匹配该IOResources提供者类。为了完成该匹配字典，首先要指定好匹配原则，该原则对提供者类来说具有唯一性。如果提供者类是一个USB设备，就要采用USB的产品ID和厂商ID形式。在我们的演示示例中，提供者类是IOResources。IOResourceMatch键便是IOResources使用的匹配原则。在示例驱动程序的匹配字典中，该值被设置为IOKIt。这通知提供者类在系统启动过程中，IOKit子系统完全加载和初始化之前要延迟加载驱动程序。

现在可以编译该IOKitTest工程了，可以使用与第3章中相同的kextload命令对其进行加载。如果打开Console实用程序查看kernel.log文件的内容，你应该会看到，该驱动程序类的各种方法都已被调用。

该驱动程序类的方法调用顺序如下。

(1) init()。在类的其他方法被调用之前肯定会先调用该方法。它的作用与C++类的构造函数相同。驱动程序的init()方法首先应该调用超类提供的实现，如果失败，则立即放弃。该方法有一个参数，为匹配字典的一个副本，对应选择的驱动程序在Info.plist文件中的个性信息。如果该

方法成功执行，就会返回true；否则返回false。

(2) probe()。该方法在匹配期间调用，驱动程序此时会检查硬件设备，硬件设备通过参数provider传递给该方法。尽管参数provider是一个指向IOService的指针，但它可以转换为匹配字典中指定的专用提供者类（如IOUSBDevice）。驱动程序的probe()实现首先应该调用超类的实现，如果成功，则对设备执行要求的任意检测，确定驱动程序是否可以对其进行控制。如果驱动程序无法控制该硬件，则probe()方法将返回NULL，否则，应该返回IOService子类的一个实例，用来控制该设备。在几乎所有的情况下，该方法都返回当前的IOService实例（this）。

(3) start()。如果之前成功调用probe()，并且这一驱动程序已被当作适合控制该硬件设备的最佳选择（基于其检测积分），则将调用其start()方法。该实现首先应该调用超类的start()实现，如果失败，则应立即放弃。驱动程序应该使用start()方法配置硬件，并初始化其运行时需要的所有资源。如果由于某些原因该方法执行失败，并且该驱动程序无法继续控制硬件，则返回false。然后，I/O Kit将使用检测积分第二高的驱动程序控制该设备。

(4) stop()。该方法在移除设备或手工卸载驱动程序时才会调用。这一方法与start()方法相反，调用stop()时，start()方法执行的所有配置或分配都应该被释放。最后，该方法应该调用超类的stop()实现。

(5) free()。最后，在销毁驱动程序的对象之前，I/O Kit将调用free()方法。其作用与C++类的析构函数相同。此时，驱动程序将释放其init()方法中分配的所有资源。即使驱动程序从未被选作特定设备的最佳匹配也会调用该方法。该方法应该调用超类的free()实现。

由于不支持异常处理，我们无法使用传统的C++方案在类构造函数中初始化对象，也无法在错误发生时抛出异常，在此，I/O Kit对象的初始化在自定义的init方法中执行，该方法将返回一个布尔值，标识该方法是否执行成功。

### 4.3.2　IORegistryExplorer

苹果提供了一个非常有用的工具，用来查看系统上加载的驱动程序，即IORegistryExplorer。这是Xcode工具的一部分。IORegistryExplorer用图形表示系统上当前加载的驱动程序以及与其他驱动程序的关系。这种关系采用层次结构表示，其中提供者类与它连接的客户端具有父子关系。

IORegistryExplorer显示了一个称为I/O Registry的实体代表。如果你曾经是一个Windows用户，不要将I/O Registry与Windows Registry混淆。I/O Registry是一个I/O Kit对象树，在系统启动时创建，之后随着I/O Kit对象对应的驱动程序从系统上加载或卸载而动态增长或减少。与Windows Registry不同，I/O Registry在计算机重启期间从不写入磁盘或保存。

IOService平面包括I/O Registry中的所有对象。因而，当试图定位一个特定驱动程序时，可能有点让人望而生畏。为了帮助查找I/O Registry中某个特定的驱动程序，IORegistryExplorer提供了一个检索字段，用于过滤名称与特定字符串相匹配的对象。

IORegistryExplorer还显示每个驱动程序对象的属性表。I/O Kit加载驱动程序时，首先会将该驱动程序的属性表初始化为匹配字典的内容，该匹配字典对应于所加载驱动程序的个性信息，这与传递给驱动程序类的init()方法的OSDictionary对象相对应。驱动程序运行时，可以通过添加

或移除额外的键/值对或者改变特定键的值，对其属性列表进行操作。这些改变特定于驱动程序实例（因此，如果在系统中同一个驱动程序针对数个设备进行了多次加载，则每个实例都有自己的属性表）。前面开发的示例IOKit驱动程序中的IORegistryExplorer如图4-4所示。

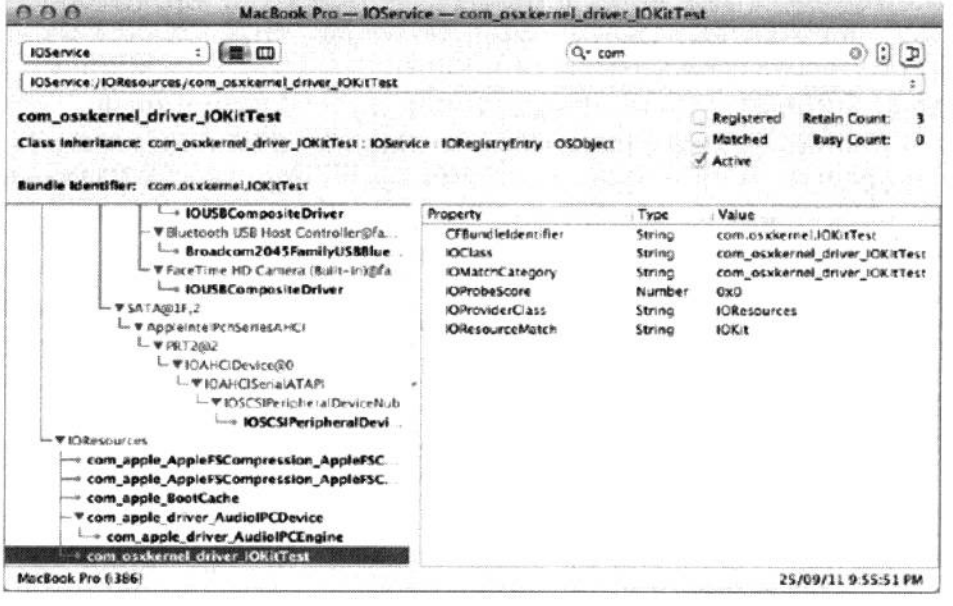



图4-4 IORegistryExplorer显示的IOKitTest示例。右侧显示的属性表从Info.plist文件中驱动程序匹配字典的内容初始化而来

I/O Registry中的对象被组织成数个平面。每个平面都只显示具有某些相同功能的对象。启动时，IORegistryExplorer将默认显示IOService平面中的对象关系，该平面包含了I/O Registry中的所有对象。其他的一些常用平面在下面列出。

- IODeviceTree：这是一个静态平面，反映系统的硬件设置；硬件设备连接到系统时也不会发生改变。设备树的内容主要依赖于系统主板，包含计算机启动时配置的一个静态快照，包括PCI插槽、内置的USB插槽及主板上所有的硬件控制器（如CPU、内存和USB控制器）。
- IOPower：这个平面显示所有实现了电源管理的驱动程序对象，这些对象会在系统切换至不同的电源状态时（如进入休眠模式）接收I/O Kit的通知。
- IOUSB：这个平面显示所有连接到系统上的USB设备和集线器。它只包括USB设备，为其加载的驱动程序可以在IOService平面中看到。

I/O Registry还可以通过ioreg工具在命令行上进行访问。IORegistryExplorer需要安装Xcode工具才可以使用，而ioreg是Mac OS X安装的标准部分，在最终用户机器或不可能安装开发者工具的任何其他系统上调试时非常有用。

## 4.4　内核库：libkern

运行时间支持和I/O Kit构建时基于的基类都在一个称为libkern的库中实现。libkern库弥补了嵌入式C++语言所不支持的许多功能。libkern库定义了一个OSObject类，它是所有I/O Kit类都使用的基类。由于驱动程序基类IOService是OSObject的一个子类，所以驱动程序的主类也派生自OSObject类。任何派生自OSObject的类都具有如下功能。

- 运行时类型信息，通过libkern提供的自定义宏实现。下面介绍这些宏的功能。
  - 类型自省：可以在运行时确定对象的类型，或该对象是否派生自给定的基类。
  - 动态转换：可以将一个对象转换成其派生类的类型（例如，将提供者类从IOService类型转换成IOUSBDevice类型的对象）。
- 对象创建，能够根据一个对象的类名字符串对其进行实例化。
- 基于保留/释放语义对象引用进行计数。
- 对象追踪，即能够确定已经实例化但仍未释放的某个类的实例个数。

### 4.4.1　OSObject

如果使用苹果的Cocoa框架编写过用户应用程序，那你对libkern的一些特性应该非常熟悉。尤其是OSObject类可以看做是Cocoa中NSObject类的内核等价物，而且其动态类型自省功能与Objective-C运行时提供的类似功能几乎完全相同。

无须将OSObject用作驱动程序中私有类的超类，尽管你可能发现OSObject提供的引用计数和对象追踪（无须执行任何额外操作）足够有理由让你采用它。

采用OSObject作为基类涉及以下几个步骤。

(1) 使用标准的C++语法，将你的类声明为OSOject的子类或派生自OSObject的类(如IOService类)。如果子类化OSObject类，则可能需要包含头文件<libkern/c++/OSObject.h>。

(2) 作为类声明的第一行（在类的头文件中），需要包括OSDeclareDefaultDestructors()宏，其参数为类名。该宏会为类声明标准的C++构造函数及析构函数，因此，不应该再在类声明中添加这两个函数。相反，需要在类中添加init()方法作为构造函数，添加free()方法作为析构函数。可以在init()方法中任意添加类所需的参数，如下面的例子所示。

```
class com_osxkernel_driver_MyObject : public OSObject
{
   OSDeclareDefaultStructors(com_osxkernel_driver_MyObject)
public:
   virtual bool     init (const char* name);
   virtual void     free ();
   …
};
```

(3) 在你的类实现文件中，放置OSDefineMetaClassAndStructors()宏，该宏有两个参数，一个是该类的名称，另一个是其直接超类的名称。实现文件的前几行通常遵循下面的模式：

```
#  #include "MyObject.h"

// 将super定义为一个便捷宏，表示超类
#define super OSObject

OSDefineMetaclassAndStructors (rom_OSxkernel_driver_MyObject, OSObject)
```

(4) 提供init()和free()方法的实现。这两个方法分别起到构造函数和析构函数的作用，如下面的例子所示。

```
bool com_osxkernel_driver_MyObject::init (const char* name)
{
   if (! super::init())
          return false;

   //其他的初始化
   return true;
}
void com_osxkernel_driver_MyObject::free ()
{
   //释放init()分配的资源
   super::free();
}
```

定义了一个OSObject的子类对象之后，可以在代码中调用C++ new操作符对其进行初始化，然后调用init()方法。为了方便起见，许多类都提供了一个静态方法同时执行这两个步骤，成功时返回一个非NULL对象，如代码清单4-4所示。

**代码清单4-4 定义一个静态辅助方法，构造自定义类的一个新实例**

```
com_osxkernel_driver_MyObject*
       com_osxkernel_driver_MyObject::withName (const char* name)
{
       com_osxkernel_driver_MyObject* me = new com_osxkernel_driver_MyObject;

       if (me && !me->init(name))
       {
              me->release();
              return NULL;
       }

       return me;
}
```

基于OSObject的所有对象的生命周期都通过引用计数来确定。对象初次创建时，其应用计数将初始化为1。释放对象时，不使用C++操作符delete（OSDeclareDefaultStructors宏将其声明为受保护的方法），而是应该调用release()方法。该方法由OSObject类实现，将对象的引用计数减1。当对象的最后一个引用被释放，其引用计数变为0时，该对象将被释放，并将调用free()方法。如果代码中有一个指针指向其需要持有的对象，则需要延长该对象的生命周期，确保代码仍然持有该对象的一个引用时该对象不会被释放。这通过调用retain()方法实现，该方法会将目标对象的引用计数加1。为了防止内存泄漏，每次调用retain()时，都要相应地调用一次release()，这非常重要。

派生自OSObject的任何对象都支持类型自省。为了将对象转换为另一种类型，libkern提供了一个OSDynamicCast(type, object)宏，执行与C++操作符dynamic_cast<type>(object)等同的操作。该宏将验证对象是否派生自要求的类，如果是，则返回一个指向该对象的指针，否则，该宏将返回NULL。最常用的动态转换是将一个对象从基类安全地转换成一个特定类。例如，将驱动程序的提供者类作为一个IOService对象，将其指针传递给start()方法。但该提供者实际上是一个特定的类（如IOUSBDevice或IOPCIDevice），动态转换使这种转换可以安全进行。例如，控制USB设备的驱动程序在其start()方法中包含如下代码，将提供者从IOService转换为IOUSBDevice类型：

```
IOUSBDevice* usbDevice = OSDynamicCast(IOUSBDevice, provider);
if (usbDevice == NULL)
{
        IOLog("Unknown provider class\n");
        return false;
}
```

OSObject基类还可以追踪其每个派生类的实例个数，这些实例已经实例化但尚未释放。该信息不仅对I/O Kit有用，对追踪内存泄漏也极有价值。在内部，I/O Kit使用每个类的实例计数，确保存在未完成对象的内存扩展不被卸载，否则将导致内核恐慌。当内核扩展定义的所有类的未完成对象不存在时，内核将卸载该内核扩展程序。每个OSObject派生类的实例数都可以通过命令行工具ioclasscount查看。

**提示**　加载完IOKitTest演示示例之后，打开Terminal，运行ioclasscount命令，将会看到com_osxkernel_driver_IOKitTest类的一个实例。

### 4.4.2　容器类

除了为内核定义基类，提供运行时环境，libkern还定义了许多容器类以管理对象集合。libkern提供的容器类包括数组、字典及有序和无序集合。虽然所有这些容器都包含各种类型的对象，甚至同一个容器中的对象类型也不同，但一个容器只能包含OSObject类的对象。

**注意**　如果你熟悉Mac OS X上的用户空间编程，那么libkern容器类其实等价于NSMutableArray、NSMutableDictionary和NSMutableSet，或Core Foundation类型的CFMutableDictionary、CFMutableArray和CFMutableSet。

为了在容器类型中包含非对象的标量类型（如布尔型、整型及字符串类型），libkern提供了相应的类OSBoolean、OSNumber和OSString，分别包装bool、长度高达64位的整数值以及C字符串。

在此，特别值得一提的是libkern中的字符串处理。libkern库提供了两个类表示字符串：OSString和OSSymbol（OSString的子类）。OSSymbol的目的不是为字符串提供一个通用的包装器，而是持有内核中表示“符号”的字符串值，如匹配字典中常用的键。创建一个新的OSSymbol实例时，构造函数将检查现有的OSSymbol对象是否包含相同字符串值，如果存在，则返回现有的对象

实例，而不再创建一个新对象。这意味着一个给定的字符串值最多只能有一个OSSymbol对象表示该值。因此，与OSSymbol值关联的字典只需比较两个OSSymbol值的地址，而不是执行代价更高的字符串比较。

在对象所有权方面，所有的容器类都遵循相同的行为。添加到容器中的任何对象都由该容器类持有，这些对象从容器中移除或容器本身的最后一个引用释放并重新分配之后，由容器释放。将一个对象插入到容器中之后，如果调用者不再需要对该对象进行引用，则可以自由释放插入的对象，因为容器类将维护该对象的引用。



查询容器中包含的对象之后，如果容器有机会被释放，而调用者仍然需要使用这个返回的对象，则调用者应该持有该对象。libkern容器类在将对象返回给调用者之前不会增加其内容对象的引用（例如，OSArray的getLastObject()方法在将最后一个对象返回给调用者之前不会增加其引用计数）。

重要的是，要注意容器类并不支持多线程环境中的任何同步使用。这并不是说它们无法在包含多线程代码的驱动程序中使用，而是调用者负责添加其本身的锁，确保串行调用容器类。

libkern提供的容器类如下所示。

- OSArray，基于数组索引存储和检索对象。
- OSDictionary，基于提供的字符串值（称为“键”）进行存储和检索。
- OSSet，提供对象存储，并能够测试一个对象是否在集合中。
- OSOrderedSet，基于提供的比较函数按序存储，以及基于索引进行检索。

所有的libkern容器类都可以使用OSCollectionIterator类进行迭代，如代码清单4-5所示。迭代OSDictionary时，迭代器返回的对象是字典中的键，而不是字典本身包含的值。

**代码清单4-5 OSArray中包含的对象迭代函数示例**

```
void    IterateArray (OSArray* array)
{
        OSCollectionIterator*   iter;

        iter = OSCollectionIterator::withCollection(array);
        if (iter != NULL)
        {
                OSObject*       anObject;

                while ((anObject = iter->getNextObject()) != NULL)
                {
                        //假定该数组只包含字符串值
                        // OSString* aString = OSDynamicCast(OSString, anObject);
                }

                iter->release();
        }
}
```

驱动程序的一个特殊容器对象是其属性表。这是一个包含许多键/值对的字典，这些键/值对是某个特定驱动程序实例的局部键/值对。驱动程序加载时，I/O Kit将使用Info.plist文件中的匹配字典项填充其属性表。

但在驱动程序运行时，它可以任意添加或移除其属性表中额外的值。驱动程序的属性表比较特殊，因为它可以由用户空间应用程序（包括IORegistryExplorer）访问。这使它成为驱动程序和用户空间之间传递少量数据（如整数值）的完美方式。

另外，驱动程序可以将某些重要的变量值写入到其属性列表的键中，随后可以在IORegistryExplorer（或自定义的应用程序）中进行监控，以跟踪驱动程序的状态。

## 4.5　小结

- I/O Kit提供一个面向对象的框架，用于开发Mac OS X上的驱动程序。
- 使用该框架编写的驱动程序都继承自一个适合的基类，该基类根据驱动程序实现的功能选择。I/O Kit为音频输入输出流、串行端口及磁盘设备之类的驱动程序提供基类。
- 驱动程序通过一个称为其提供者的对象访问其硬件，这使它与硬件的通信像与硬件连接的总线进行通信一样，极其自然。
- 驱动程序仅在系统检测到其硬件时才进行加载，正如其属性列表中的匹配原则所述。
- I/O Kit基于libkern库构建，libkern库通过对象实例化、引用计数及容器类为内核提供运行时支持。

# 第5章 应用程序与驱动程序的交互



上一章介绍了I/O Kit驱动程序，它位于内核中。而另一方面，与用户交互的应用程序则位于用户空间中。因此，如果用户要使用驱动程序提供的服务，就需要跨越内核与用户空间之间的界限。

Mac OS X提供了几种不同的机制，通过这些机制，驱动程序可以将其服务提供给用户空间应用程序。开发者选择哪一种方法使特定的驱动程序可以为用户空间应用程序提供服务，取决于驱动程序实现哪一类功能类型。例如，所有的串行端口、音频驱动程序和存储设备都有其自己的接口，这些接口均由I/O Kit定义。利用该接口，用户空间应用程序可以使用这些设备。应用程序可以使用任何硬件厂商的设备，前提是该厂商的驱动程序为该设备实现了标准的I/O Kit接口。从驱动程序开发者的角度来看，他们更喜欢使用I/O Kit的公共接口，因为这样既可以保证用户空间中众多的应用程序可以使用驱动程序，又无须开发者为驱动程序自定义接口，同时也就减少了工作量。

串行端口就是一个很好的例子。Mac OS X用户空间应用程序通过一个字符设备访问串行端口，该设备由文件系统/dev路径中的一个文件表示。为了通过串行设备进行通信，用户应用程序在打开、读取或写入文件系统上的任何其他文件时，需要调用同样的函数，即`open()`、`read()`、`write()`。在内核中，提供串行端口的驱动程序将创建一个标准I/O Kit类`IOSerialStreamSync`的实例。I/O Kit的串行家族将在/dev目录中创建一个设备节点，在I/O Registry中发布该节点的路径，以便应用程序可以找到它。串行家族还会把请求从用户应用程序传递给（作为标准`IOSerialDriverSync`类的子类的）驱动程序实现中的方法调用。至于将服务发布给用户空间，是无须驱动程序开发者操心的，如图5-1所示。

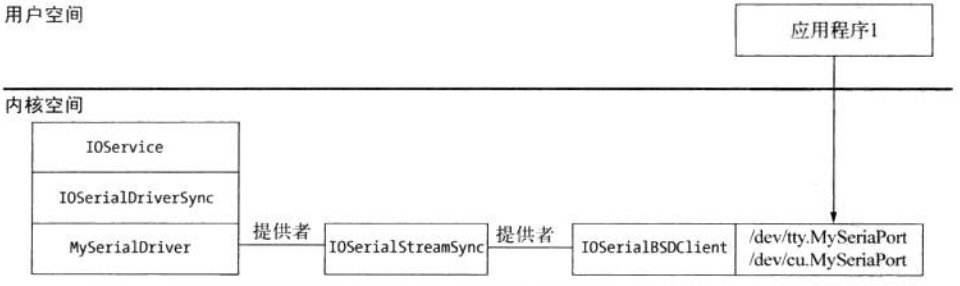



图5-1 用户空间应用程序与串行端口通信时调用的类。除了`MySerialDriver`，所有的对象及其连接均由I/O Kit创建

并非所有的驱动程序开发者都像串行端口驱动程序开发者那样幸运，有I/O Kit帮他们实现用户空间与内核之间的沟通。对于一个具有自定义功能的硬件设备，I/O Kit可能不会为你的驱动程序提供适当的客户端接口。在这种情况下，驱动程序需要实现自定义接口，以便用户空间应用程序在与驱动程序交互时使用。正如第1章介绍的那样，Mac OS X在用户空间和内核之间具有明确的界限，而这就限制了可以采用的交互方式。本章将讲述I/O Kit提供的方法，这些方法可以跨越用户与内核之间的界限，从而实现用户应用程序和内核驱动程序之间的通信。

## 5.1　I/O Kit 框架

进程与内核驱动程序之间的通信通过用户空间API进行，该API由一个名为`IOKit.framework`的框架（以下简称“I/O Kit框架”）提供。利用I/O Kit框架，用户空间应用程序可以检测系统上的硬件设备及内核中现有的驱动程序、监控热插拔硬件（如USB设备）的接入和移除，与I/O Kit驱动程序交互。I/O Kit框架定义了一些数据类型，这些数据类型提供了内核对象的用户空间表示和处理这些内核对象所需的一些函数。尽管I/O Kit在内核中是基于C++的框架，但在用户空间中，它是以C语言函数集的形式提供的。因此，I/O Kit可用于以C和C++编写的工程，也可用于以Objective-C编写的工程，这对GUI应用程序来说非常重要。

I/O Kit框架提供对I/O Registry中内核对象的访问，可以使用IORegistryExplorer实用程序查看这些内核对象（参见第4章）。I/O Registry包含一些内核对象，这些对象表示连接到计算机的硬件，或与连接的硬件设备相匹配并且已经加载到内核的驱动程序。I/O Registry中的对象只能在内核中创建，包含在内核驱动程序中，但I/O Kit框架为用户空间中的应用程序提供了一种检查I/O Registry内容的方法，包括迭代该注册表、确定对象之间的关系（例如，确定特定的硬件设备加载了哪个驱动程序），以及读写I/O Registry中对象的属性。

I/O Registry包含的内核对象可以表示加载的驱动程序或连接的硬件设备。这意味着I/O Kit框架提供的功能可用于驱动程序及硬件设备。在某些情况下，应用程序可以直接通过其对应的I/O Registry对象操作硬件设备，而不需要内核驱动程序，第15章在介绍USB设备时会讨论这一点。I/O Kit框架还允许应用程序安装一个通知程序，以监控特定驱动程序或硬件设备的接入。

## 5.2　查找驱动程序

用户空间应用程序与内核驱动程序通信的第一步是找到驱动程序的运行实例。在Mac OS X这样的系统上，硬件设备可以在任意时刻插入到机器中，并且在用户插入了驱动程序支持的多个硬件设备时，加载该驱动程序的多个实例。为了处理这些情况，I/O Kit框架提供了一些函数，不仅可以迭代I/O Registry中符合某些标准的所有设备和驱动程序，还可以注册回调函数，以监控符合某些标准的驱动程序或设备的接入或移除。

为了找到I/O Registry中特定的驱动程序，I/O Kit框架使用了一个匹配字典，其中包括你要找的驱动程序或设备的属性。I/O Kit设计中值得称道之处在于，用户空间应用程序用来查找驱动程序或硬件设备的匹配字典，与内核驱动程序属性列表的驱动程序个性信息中的匹配字典（参见第

4章）采用完全相同的形式。实际上，在决定是否加载设备的驱动程序时，内核用来比较驱动程序匹配字典的代码，正是比较用户空间应用程序的匹配字典时，用于确定设备或驱动程序是否与应用程序相关的代码。

与驱动程序属性列表中的匹配字典一样，应用程序可以创建其所需的通用或特定匹配字典。例如，根据添加到匹配字典中的属性，应用程序可以匹配所有的USB设备，或将一个特定的USB厂商ID添加到匹配字典，匹配某个厂商制造的任何USB设备。在匹配字典中添加一个额外的USB产品ID，应用程序则仅可以匹配特定的USB设备。

下面介绍一个例子。代码清单5-1创建一个匹配字典，它可以匹配任意的USB设备，并可用于迭代I/O Registry，输出所有匹配设备的名称。为了编译该示例，创建一个基于Mac OS X应用程序模板Command Line Tool的Xcode新工程。提示为工程命名时，输入DriverIterator，并选择工程类型Core Foundation。需要将`I/OKit.framework`框架添加到工程中，否则，工程构建时将会发生链接错误。

这段代码的结构应该非常简单。

(1) 创建一个匹配字典，指定我们感兴趣的硬件设备或驱动程序的属性。该示例使用辅助函数`IOServiceMatching()`创建一个只含有一项且`IOProviderClass`键为特定值的字典。这与将`IOProviderClass`项添加到驱动程序属性列表中的效果相同；属于指定类的子类的任何内核对象，比如这个例子中的`IOUSBDevice`，都将匹配该字典。

(2) 调用`IOServiceGetMatchingServices()`进入匹配字典，并将输出作为迭代器接收，该迭代器能用来遍历I/O Registry中与匹配字典相匹配的所有内核对象。该迭代器表示函数调用时的系统状态。迭代器对象创建后，不可以再修改，即使将额外的匹配设备添加到系统上也不能修改。

(3) `IOIteratorNext()`函数会被重复调用，每次调用时，都会返回与指定字典相匹配的下一个对象。当收到最后一个对象时，任何对`IOIteratorNext()`的进一步调用都将返回`0`。迭代器返回的任何对象都增加了自身的引用计数，需要调用者调用`IOObjectRelease()`进行释放。

(4) 为了简化该示例，代码中并不包括USB集线器处理，只列出了实际的USB设备名称。这还使该示例可以展示用户空间应用程序能在内核对象上执行的类型内省功能。为此，调用了`IOObjectCopyClass()`函数，将该对象类的类型作为`CoreFoundation`字符串返回。该匹配字典包括的所有对象为`IOUSBDevice`的实例或`IOUSBDevice`派生类（包括`IOUSBHubDevice`类）的实例。为了从列表中排除USB集线器设备，该示例将忽略所有类名与`IOUSBDevice`字符串不完全匹配的对象。

**代码清单5-1 迭代所有连接的USB硬件设备的代码**

```
#include <CoreFoundation/CoreFoundation.h>
#include <IOKit/IOKitLib.h>

int main (int argc, const char * argv[])
{
        CFDictionaryRef         matchingDict = NULL;
        io_iterator_t           iter = 0;
        io_service_t            service = 0;
        kern_return_t           kr;
```

```
        //创建一个匹配字典，查找任意的USB设备
        matchingDict = IOServiceMatching("IOUSBDevice");

        //为匹配该字典的所有I/O Registry对象创建一个迭代器
        kr = IOServiceGetMatchingServices(kIOMasterPortDefault, matchingDict, &iter);
        if (kr != KERN_SUCCESS)
                return -1;

        //迭代所有的匹配对象
        while ((service = IOIteratorNext(iter)) != 0)
        {
                CFStringRef     className;
                io_name_t       name;

                //列出所有的IOUSBDevice对象，忽略IOUSBDevice子类的对象
                className = IOObjectCopyClass(service);
                if (CFEqual(className, CFSTR("IOUSBDevice")) == true)
                {
                        IORegistryEntryGetName(service, name);
                        printf("Found device with name: %s\n", name);
                }
                CFRelease(className);
                IOObjectRelease(service);
        }

        //释放迭代器
        IOObjectRelease(iter);

        return 0;
}
```

**注意**　IOServiceGetMatchingServices()函数确保释放传入的匹配字典的引用。这也是代码清单5-1中的代码无须对创建的CFDictionaryRef调用CFRelease()的原因。

代码清单5-1使用IOServiceGetMatchingServices()函数为应用程序感兴趣的内核对象创建一个匹配字典。IOServiceGetMatchingServices()函数的第一个参数是Mach端口，用于用户空间进程和I/O Kit之间的通信。Mac OS X 10.2及之后的版本中引入了一个名为kIOMasterPortDefault的便捷宏，尽管苹果的一些示例代码仍然使用Mac OS X 10.2之前的方案调用IOMasterPort()函数获得I/O Kit的Mach端口。

在Mac OS X等系统上，硬件设备可以随时插入到系统中。对于前一节中使用的方案，其问题是需要应用程序轮询连接的设备列表。一个较好的方案是，每当你感兴趣的设备连接到计算机上时让I/O Kit通知你的应用程序。I/O Kit框架提供了一个可选方案，应用程序指定一个匹配字典，当满足该匹配字典的一个对象添加到I/O Registry中时通知一个回调函数。与Mac OS X中类似的函数一样，I/O Kit框架的通知使用称作“运行回路”的标准事件分配机制进行传送。

运行回路是将多个源的事件通知Mac OS X和iOS应用程序的基本方式，无需花费CPU时间来轮询每个源。运行回路是一个Core Foundation对象，监控多个“运行回路源”。每当其中任意一个源生成一个需要处理的事件时，运行回路就把该事件分配到注册的回调函数。Mac OS X上的每个线程都包含一个运行回路，其中包括主线程。Mac OS X上的事件回路运行在主线程上，只是一个包含键盘和鼠标事件源的运行回路。例如，为了对用户点击鼠标按钮做出响应，主线程的运行回路将唤醒并为用户点击的应用程序窗口生成一个Cocoa鼠标按下事件。

代码清单5-2展示了每当一个USB设备连接到计算机时，命令行实用程序如何注册接收通知的情况。你会发现，无论是使用轮询方法还是通知回调，都可以使用同样的匹配字典。



**代码清单5-2 监控USB设备接入的代码**

```
#include <CoreFoundation/CoreFoundation.h>
#include <IOKit/IOKitLib.h>

int main (int argc, const char * argv[])
{
        CFDictionaryRef                 matchingDict = NULL;
        io_iterator_t                   iter = 0;
        IONotificationPortRef           notificationPort = NULL;
        CFRunLoopSourceRef              runLoopSource;
        kern_return_t                   kr;

        //创建一个匹配字典，用于查找任意的USB设备
        matchingDict = IOServiceMatching("IOUSBDevice");

        notificationPort = IONotificationPortCreate(kIOMasterPortDefault);
        runLoopSource = IONotificationPortGetRunLoopSource(notificationPort);
        CFRunLoopAddSource(CFRunLoopGetCurrent(), runLoopSource, kCFRunLoopDefaultMode);

        kr = IOServiceAddMatchingNotification(notificationPort, kIOFirstMatchNotification,
             matchingDict, DeviceAdded, NULL, &iter);
        DeviceAdded(NULL, iter);

        CFRunLoopRun();

        IONotificationPortDestroy(notificationPort);

        //释放迭代器
        IOObjectRelease(iter);

        return 0;
}
```

为了给符合某些标准的设备或驱动程序对象创建一个通知回调，代码清单5-2中的代码将执行下列步骤。

(1) 创建一个匹配字典，描述应用程序感兴趣设备的属性。

(2) 调用IONotificationPortCreate()函数，设置通信渠道，I/O Kit通过该通信渠道向用户空间应用程序传送通知消息。

(3) 因为我们想要使用运行回路将通知分配给应用程序，所以创建一个运行回路源表示通知端口，并将该源安装在当前线程的运行回路上。

(4) 调用IOServiceAddMatchingNotification()函数，将匹配字典与通知端口（及运行回路源）关联。该函数将分配并返回一个迭代器对象，这个对象在通知消息的操作中具有非常重要的作用。调用IOServiceAddMatchingNotification()之后，该迭代器将包含I/O Registry中与该匹配字典相匹配的所有对象。I/O Kit框架不会向这些设备发送通知，因此，我们需要手工调用回调函数传入返回的迭代器。这也非常重要，因为调用IOIteratorNext()直至到达迭代器末尾都不会发送任何通知，也不会调用回调函数。类似地，设备回调必须通过迭代器运行，直至达到最终对象。不再需要该通知回调之后，调用者才可以释放该迭代器。与IOServiceGetMatchingServices()函数类似，调用IOServiceAddMatchingNotification()时总会将匹配字典的引用计数减1。因此，如果调用者在安装通知之后还需要该字典，就应该预先手工保留该对象。

(5) 由于该例是一个命令行实用程序，所以需要调用CFRunLoopRun()手工运行该运行回路。如果这是一个基于Cocoa的应用程序，并且我们已经将通知安装到了主运行回路上，则该运行回路将由NSApplicationMain()函数启动。

(6) 最后，该应用程序退出时会销毁通知端口。这会自动将回路源从其添加到的运行回路中移除，并释放迭代器对象。

回调函数DeviceAdded如代码清单5-3所示。你会发现，这与我们在轮询实现中使用的代码相同。传递给该回调函数的迭代器对象与初始化调用IOServiceAddMatchingNotification()返回的对象相同。因为同一个对象重复用于通知告诉我们的所有设备，所以重要的是，该回调不要释放迭代器对象，该迭代器在通知安装时必须保持可用。

**代码清单5-3　监控USB设备接入的代码**

```
void DeviceAdded (void* refCon, io_iterator_t iterator)
{
        io_service_t            service = 0;

        //迭代所有的匹配对象
        while ((service = IOIteratorNext(iterator)) != 0)
        {
                CFStringRef     className;
                io_name_t       name;

                //列出所有的IOUSBDevice对象，忽略IOUSBDevice子类的对象
                className = IOObjectCopyClass(service);
                if (CFEqual(className, CFSTR("IOUSBDevice")) == true)
                {
                        IORegistryEntryGetName(service, name);
                        printf("Found device with name: %s\n", name);
                }
                CFRelease(className);
                IOObjectRelease(service);
        }
}
```

**提示** 设备通知回调问题的一个常见原因是，调用IOIteratorNext()直至返回0时清空迭代器失败。到达迭代器的末尾后，迭代器将重置，并且回调处于启用状态。

对于具有内核驱动程序的硬件设备来说，用户空间应用程序将通过向驱动程序发送控制请求来控制该硬件，而不是直接与该硬件设备进行交互。这种情况下，应用程序对特定硬件设备的接入并不是很感兴趣，而当该硬件的驱动程序已经加载时才对其产生兴趣。这可以通过创建一个字典来执行，该字典将使用IOServiceMatching()函数来匹配驱动程序的类名。例如，为了创建一个与第4章中开发的示例I/O Kit驱动程序相匹配的字典，应用程序将使用如下代码：

```
IOServiceMatching("com_osxkernel_driver_IOKitTest");
```

反向DNS命名方案能够确保驱动程序类名的唯一性，这意味着，能保证与该匹配字典相匹配的任何驱动程序都是我们需要的驱动程序。

## 5.3 观察设备移除

除了监控设备接入，应用程序可能还希望观察设备从系统上移除，如USB设备拔出。设备接入消息被发送到符合匹配字典所描述标准的任何设备，与此不同，设备移除消息只向应用程序感兴趣的特定设备发送。应用程序通常对所有打开的设备感兴趣，因为应用程序希望对它正访问的设备在移除时做出响应。

在前面的代码示例中（如代码清单5-3），我们获得了驱动程序对象的一个引用，并从驱动程序读取属性，然后释放驱动程序对象，所有这些操作都在一个函数中进行。更常见的是，除了设备接入回调函数，应用程序还将持有驱动程序对象，并且仅在应用程序退出或设备移除时释放该对象。

**注意** 在上面的示例中，可以使用函数的局部变量保存驱动程序对象，因为驱动程序在函数返回之前释放。但如果应用程序希望在函数返回之后仍然可以使用该驱动程序对象，则需要在堆上分配一个结构体，保存驱动程序状态。

获得驱动程序实例的引用之后，应用程序便可以进行注册，在驱动程序的状态发生改变时，包括驱动程序终止响应其硬件设备的移除时，接收通知。该通知回调可以通过调用IOServiceAddInterestNotification()函数进行安装，该函数在I/O Kit框架中定义。与设备接入通知一样，应用程序需要提供一个端口，当驱动程序的状态发生改变时，I/O Kit通过该端口通知应用程序。这可以通过IONotificationPortCreate()函数进行创建，如代码清单5-2所示。如果应用程序已经为设备接入事件创建了一个通知端口，它就能共享该通知端口及其对应的运行回路源，接收设备移除通知。这通过将现有的通知端口传递给IOServiceAddInterestNotification()函数实现。

应用程序接收到驱动程序实例已经终止的通知时，应该释放对该驱动程序的引用，并执行必要的操作通知用户设备已经移除。

代码清单5-4对代码清单5-3的DeviceAdded()函数进行了修改，创建了一个结构体表示应用程序中的一个驱动程序实例，并安装一个回调，接收驱动程序的通知（如驱动程序终止）。

**代码清单5-4　应用程序安装回调函数接收驱动程序终止通知的代码片段**

```
#include <IOKit/IOMessage.h>

//描述驱动程序实例的结构体
typedef struct {
        io_service_t    service;
        io_object_t     notification;
} MyDriverData;

//通知端口，用于设备接入和驱动程序状态改变
IONotificationPortRef   gNotificationPort = NULL;

void DeviceAdded (void* refCon, io_iterator_t iterator)
{
        io_service_t            service = 0;

        //迭代所有的匹配对象
        while ((service = IOIteratorNext(iterator)) != 0)
        {
                MyDriverData*   myDriverData;
                kern_return_t   kr;

                //分配一个结构体，保存驱动程序实例
                myDriverData = (MyDriverData*)malloc(sizeof(MyDriverData));
                //为该驱动程序实例保存io_service_t
                myDriverData->service = service;

                //安装回调，接收驱动程序状态改变通知
                kr = IOServiceAddInterestNotification(gNotificationPort,
                                        service,                        //驱动程序对象
                                        kIOGeneralInterest,
                                        DeviceNotification,             //回调
                                        myDriverData,           //传递给回调的refCon
                                        &myDriverData->notification);
        }
}

void DeviceNotification (void* refCon, io_service_t service, natural_t messageType,
    void* messageArgument)
{
        MyDriverData*   myDriverData = (MyDriverData*)refCon;
        kern_return_t   kr;

        //只处理驱动程序终止通知
        if (messageType == kIOMessageServiceIsTerminated)
        {
                //输出移除设备的名称
                io_name_t       name;
                IORegistryEntryGetName(service, name);
```

```
            printf("Device removed: %s\n", name);

            //移除驱动程序状态改变通知
            kr = IOObjectRelease(myDriverData->notification);

            //释放驱动程序对象的引用
            IOObjectRelease(myDriverData->service);

            //释放保存驱动程序连接的结构体
            free(myDriverData);
        }
}
```



## 5.4 修改设备驱动程序属性

应用程序在找出其感兴趣的驱动程序对象后，便可以与驱动程序及其控制的硬件设备进行交互。I/O Kit框架提供了两种通过用户空间与驱动程序交互的方法。一种方法是应用程序打开与驱动程序的连接，然后使用该连接向驱动程序发送控制请求，并接收状态。如果驱动程序需要维护客户端的状态或需要访问控制，以确保某一时刻只能有一个客户端可以访问硬件设备，则需要采用基于连接的方案。这一内容将在本章后面进行讲述。

另一种方法更简单，即允许应用程序向驱动程序读和写键/值属性值。该驱动程序可以执行某些类型的操作，而无须知道是哪一个客户端发送的请求，如读/写驱动程序首选项的值或配置硬件设备的设置。例如，音频设备的音量是一个值，可以由任何用户应用程序读/写。当设置该值时，驱动程序可以将硬件设备重新配置为新的音量设置，而无须考虑哪一个应用程序设置了该值。

第4章中已经提到，每个I/O Kit驱动程序都包含一个属性表，这是一个包含键/值对的字典。所有的用户空间应用程序都可以访问驱动程序的属性表，没有任何限制（这是I/ORegistryExplorer实用程序显示每个驱动程序属性的方式）。另外，应用程序可以向驱动程序的属性表中添加新的键/值对，还可以修改现有的属性值。这可用于在用户空间应用程序和内核驱动程序之间简单地进行少量数据交换。由于该方案并不基于连接，所以驱动程序无法针对不同的用户应用程序修改其行为，每个用户应用程序都可以访问相同的属性表值。但如果加载了同一个驱动程序的多个实例，那么每个实例都会有其自己的属性表。应该注意的一点是，驱动程序的属性表具有易变性，在驱动程序卸载时不会进行保存。

应用程序在定位其感兴趣的驱动程序后，I/O Kit框架包含的一些函数将使驱动程序属性表的读和写变得非常容易。`IORegistryEntryCreateCFProperties()`函数为调用的应用程序提供了驱动程序属性表的一个状态快照，为Core Foundation字典。如果应用程序对某个特定的键感兴趣，则可以使用`IORegistryEntryCreateCFProperty()`函数。例如，假如我们希望对代码清单5-3的回调函数进行修改，输出连接到计算机上的每个USB设备的厂商名称而不是设备名称，则`IOUSBDevice`类可以通过其属性表的“USB Vendor Name”键获得厂商字符串。代码清单5-5中的代码为修改后的回调函数，该函数从设备的属性表中读取厂商的名称。

**代码清单5-5　读取USB设备的属性表，获得设备的厂商字符串**

```
void DeviceAdded (void* refCon, io_iterator_t iterator)
{
        io_service_t            service = 0;

        //迭代所有的匹配对象
        while ((service = IOIteratorNext(iterator)) != 0)
        {
                CFStringRef     className;

                //列出所有的IOUSBDevice对象，忽略IOUSBDevice的子类对象
                className = IOObjectCopyClass(service);
                if (CFEqual(className, CFSTR("IOUSBDevice")) == true)
                {
                        CFTypeRef               vendorName;

                        vendorName = IORegistryEntryCreateCFProperty(service,
                              CFSTR("USB Vendor Name"), kCFAllocatorDefault, 0);
                        CFShow(vendorName);
                }
                CFRelease(className);
                IOObjectRelease(service);
        }
}
```

如代码清单5-5所示，属性表是驱动程序将信息发布给用户应用程序的一个非常便捷的方式，如硬件描述、驱动程序的当前状态或调试信息。驱动程序属性表的另一个用途是让应用程序将少量的数据传递给驱动程序。例如，对第4章开发的示例I/O Kit驱动程序进行修改，使应用程序可以在驱动程序卸载时输出指定的自定义信息。通过向属性表的StopMessage键下添加一个字符串值来实现该操作。该键将由用户空间应用程序添加到属性表中，但在卸载时由内核驱动程序从属性表中读出。

我们先来修改用户空间应用程序。首先，找出第4章中编写的I/O Kit驱动程序。这可以通过创建下列匹配字典实现：

```
matchingDict = IOServiceMatching("com_osxkernel_driver_IOKitTest");
```

接下来，向驱动程序的属性表中写入字符串值"The driver has stopped"，使其在StopMessage键下可以访问：

```
IORegistryEntrySetCFProperty(service, CFSTR("StopMessage"), CFSTR("The driver has stopped"));
```

这就是用户空间应用程序需要的所有操作。在该例子中，我们选择使用一个字符串值，但驱动程序的属性值可以是Core Foundation类型CFString、CFNumber、CFBoolean、CFData以及CFArray或CFDictionary（包含支持的Core Foundation类型的对象）中的任意一个。

当驱动程序的属性通过用户空间应用程序进行设置时，将调用对应驱动程序对象的setProperties()方法，它带有一个参数，包含已经设置的属性字典。setProperties()方法在IORegistryEntry类中定义，但由于每个I/O Kit驱动程序类都是IOService的子类，IOService本身

就是IORegistryEntry的子类，所以每个驱动程序对象都可以覆写该方法。默认情况下，setProperties()的基类实现什么都不做。因此，除非你的驱动程序提供一个实现，否则用户空间应用程序对你的驱动程序设置的任何属性都将被忽略。

setProperties()方法使驱动程序可以立即响应用户空间应用程序设置的值。如果修改的属性需要驱动程序重新配置底层的硬件设备，则该操作应该在setProperties()方法中执行。对于示例中的驱动程序，当应用程序改变StopMessage属性时，我们让驱动程序更新其属性表，插入提供的字符串值。这也使该属性可用，以便用户应用程序（包括IOregistryExplorer）可以对其进行读取。处理这一过程的代码如代码清单5-6所示。



**代码清单5-6 驱动程序setProperties()方法的示例实现**

```
IOReturn com_osxkernel_driver_IOKitTest::setProperties (OSObject* properties)
{
        OSDictionary*   propertyDict;

        //提供的属性对象应该是一个OSDictionary对象
        propertyDict = OSDynamicCast(OSDictionary, properties);
        if (propertyDict != NULL)
        {
                OSObject*               theValue;
                OSString*               theString;

                //从字典读取对应的StopMessage键值
                theValue = propertyDict->getObject("StopMessage");
                theString = OSDynamicCast(OSString, theValue);
                if (theString != NULL)
                {
                        //将该值添加到驱动程序的属性表中
                        setProperty("StopMessage", theString);
                        return kIOReturnSuccess;
                }
        }

        return kIOReturnUnsupported;
}
```

最后，示例驱动程序卸载时应该会输出自定义的停止消息，如果用户空间应用程序已经设置了一个停止消息。这可以通过查询该驱动程序属性表中的StopMessage键的值实现，如代码清单5-7所示。在这个例子中，我们已经选择将该属性写入到驱动程序的属性表中。驱动程序无须采用该方式处理setProperties()方法。我们可以将提供的字符串值保存到实例变量中，而如果该值用于重新配置硬件，则应该使用setProperties()方法将该值写入到硬件中，此后，驱动程序将不再需要该值，可以将其丢弃。

**代码清单5-7 使用用户应用程序设置的自定义字符串值**

```
void com_osxkernel_driver_IOKitTest::stop (IOService *provider)
{
        OSString*       stopMessage;
```

```
        //从驱动程序属性表读取可能的自定义字符串并输出
        stopMessage = OSDynamicCast(OSString, getProperty("StopMessage"));
        if (stopMessage)
                IOLog("%s\n", stopMessage->getCStringNoCopy());

        super::stop(provider);
}
```

## 5.5　基于状态的交互

尽管获取/设置驱动程序属性的方案提供了一个非常简单的方式与驱动程序及硬件设备进行交互，但它相当死板，仅适用于非常特定的用途。I/O Kit框架提供了另一种方法，用于用户空间应用程序与驱动程序交互，该方法基于用户应用程序和驱动程序之间的连接。使用连接，驱动程序可以确定请求来自哪一个用户应用程序，从而可以将状态与连接关联起来。（另外，因为一个应用程序可以对一个驱动程序打开多个连接，所以驱动程序可以确定请求来自哪一个连接。）这支持实现复杂的协议和基于状态的控制。

例如，一个硬件设备可能每次只允许一个用户空间应用程序对其进行访问，这需要应用程序请求对该设备进行独占访问。获得独占访问后，在读/写数据之前，需要对该设备进行配置。采用基于连接的方案，驱动程序允许获得设备独占访问权的客户端控制硬件，并拒绝其他连接的请求。此外，这可以确保客户端在该连接没有按要求预先配置硬件时无法读/写数据。使用连接，还可以使驱动程序实现异步操作。应用程序可以向驱动程序发送一个请求，启动一个后台操作，然后轮询该驱动程序，确定事务是否已经完成。因为驱动程序根据请求启动操作，并且轮询该操作状态的每个请求都来自同样的连接，所以可以据此来确定应用程序正在轮询哪一个操作。或者，在后台操作已经完成时，驱动程序可以使用应用程序的连接向应用程序发送一个通知，从而不必再轮询状态。

在内核中，用户应用程序到驱动程序的每个连接由IOUserClient类表示。对于每个与驱动程序的连接，I/O Kit都会实例化一个IOUserClient对象，并且仅在应用程序关闭它与驱动程序的连接或应用程序终止时才销毁该对象。应用程序对驱动程序的所有控制请求都由表示特定连接的用户客户端对象处理。驱动程序对IOUserClient子类提供了其自身的实现，添加它维护应用程序连接状态所需的任何方法和实例变量。

I/O Kit设计的独创性是用户客户端对象本身就是一个驱动程序对象：IOUserClient类继承自IOService，与任何其他的IOService实例一样，每个用户客户端都有一个提供者类，对用户客户端而言，该提供者类为应用程序控制的驱动程序实例。驱动程序与其用户客户端实例之间的关系如图5-2所示。

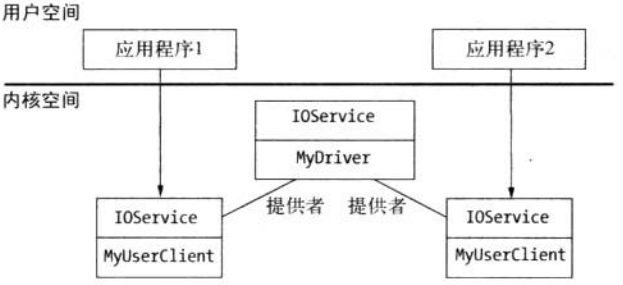



图5-2 驱动程序对象与其用户客户端对象之间的关系，用户客户端对象提供应用程序与驱动程序连接的内核端表示



为了建立与驱动程序的连接，应用程序只需调用IOServiceOpen()函数，如下所示：

```
task_port_t     owningTask = mach_task_self();
uint32_t        type = 0;
io_connect_t    driverConnection;
kern_return_t   kr;

kr = IOServiceOpen(service, owningTask, type, &driverConnection);
```

在前面的代码中，service表示应用程序希望连接到的驱动程序，可以使用前面描述的标准驱动程序匹配技术找到该驱动程序，owningTask表示运行的应用程序，type是一个无符号的32位整数，其值由驱动程序以它选择的任何方式进行解释。driverConnection参数在函数成功完成后返回给调用者，表示与驱动程序建立的连接。应用程序向驱动程序发送的任何请求都将通过调用一个函数执行，这个函数把该连接对象作为一个参数。当应用程序不想再控制驱动程序时，便调用IOServiceClose()函数。

用程序调用IOServiceOpen()函数时，操作系统将调用内核中指定的驱动程序对象处理该请求。驱动程序类会接收到以下方法调用：

```
IOReturn    DriverClass::newUserClient (task_t owningTask, void* securityID, UInt32 type,
OSDictionary* properties, IOUserClient** handler)
```

newUserClient()方法的许多参数看起来应该很熟悉。它们只是用户空间应用程序传递给IOServiceOpen()函数的值。newUserClient()的内核实现负责实例化一个新的用户客户端对象，并通过handler参数将其返回给调用者。但大多数驱动程序并不需要实现newUserClient()方法，因为适合几乎所有用途的实现由IOService的基类提供。为了利用这一标准实现，驱动程序需要做的只是为其属性表的IOUserClientClass键添加一个字符串值。该属性值是一个字符串值，其中包含驱动程序用户客户端类的类名。这可以通过向Info.plist文件的驱动程序个性化信息中添加一项实现（因为驱动程序的属性表由Info.plist文件中的值初始化），也可以通过在驱动程序加载时手工设置该属性实现。例如，图5-2中的驱动程序有一个由MyUserClient类实现的用户客户端，因而会通过以下调用设置其用户客户端类：

```
setProperty("IOUserClientClass", "MyUserClient");
```

newUserClient()的标准实现将实例化驱动程序的指定用户客户端类，并初始化这一新的用户客户端，将主驱动程序类作为其提供者。

我们看一下，如何为第4章开发的I/O Kit演示示例驱动程序实现一个用户客户端类。该驱动程序用户客户端类的头文件框架如代码清单5-8所示。你会发现，用户客户端实现的许多方法与主驱动程序类实现的方法非常类似，这是因为IOUserClient类派生自IOService类，而每个驱动程序最终都派生自该类。因此，用户客户端实现的初始化和终止方法，任何其他驱动程序也必须实现。

**代码清单5-8　一个基本用户客户端类的头文件**

```
class com_osxkernel_driver_IOKitTestUserClient : public IOUserClient
{
        OSDeclareDefaultStructors(com_osxkernel_driver_IOKitTestUserClient)

private:
        task_t                          m_task;
        com_osxkernel_driver_IOKitTest* m_driver;

public:
        virtual bool    initWithTask (task_t owningTask, void* securityToken,
                                      UInt32 type, OSDictionary* properties);
        virtual bool    start (IOService* provider);

        virtual IOReturn        clientClose (void);
        virtual void    stop (IOService* provider);
        virtual void    free (void);
};
```

除了熟悉的方法start()、stop()和free()，用户客户端还提供了另外一个方法，即clientClose()，它是用户客户端对象管理的一部分。该方法将在用户空间应用程序已经关闭它与驱动程序的连接（调用IOServiceClose()，或由于应用程序已经终止或崩溃）时调用。驱动程序不应该认为用户空间应用程序必定编写良好，并在其本身之后、用户客户端关闭前释放。因此，对驱动程序来说，clientClose()方法是一个很好的位置，能够确保硬件返回到空闲状态，并可供下一个希望使用它的用户空间应用程序使用。

---

**提示**　IOUserClient类提供了一个clientDied()方法。当需要区分客户端连接关闭是通过用户空间进程终止而不是调用IOServiceClose()时，子类可以选择实现该方法。由于clientDied()的默认实现只是调用clientClose()，所以大多数用户客户端的实现都可以通过实现clientClose()方法进行，该方法可以处理以上这两种情况。

---

一个示例用户客户端类的实现如代码清单5-9所示。为简明起见，stop()和free()的实现方法已经省去，因为对于基本的用户客户端，这些方法只需调用超类提供的实现即可。

**代码清单5-9 一个基本用户客户端类的实现**

```
//定义超类
#define super IOUserClient

OSDefineMetaClassAndStructors(com_osxkernel_driver_IOKitTestUserClient, IOUserClient)

bool    com_osxkernel_driver_IOKitTestUserClient::initWithTask (task_t owningTask, void*
securityToken, UInt32 type, OSDictionary* properties)
{
        if (!owningTask)
                return false;

        if (! super::initWithTask(owningTask, securityToken , type, properties))
                return false;

        m_task = owningTask;

        //可选的：确定调用进程是否有管理员权限
        IOReturn ret = clientHasPrivilege(securityToken, kIOClientPrivilegeAdministrator);
        if ( ret == kIOReturnSuccess )
        {
                // m_taskIsAdmin = true;
        }

        return true;
}

bool    com_osxkernel_driver_IOKitTestUserClient::start (IOService* provider)
{
        if (! super::start(provider))
                return false;

        m_driver = OSDynamicCast(com_osxkernel_driver_IOKitTest, provider);
        if (!m_driver)
                return false;

        return true;
}

IOReturn        com_osxkernel_driver_IOKitTestUserClient::clientClose (void)
{
        terminate();
        return kIOReturnSuccess;
}
```

关于代码清单5-9中的3个用户客户端方法的实现，需要注意以下几点。

- 在initWithTask()方法中，我们将owningTask参数保存到一个实例变量中。在本章后面会看到，希望访问客户端地址空间中的内存时，会用到这个值，如将数据从驱动程序写入到客户端进程分配的缓冲区时。initWithTask()方法也可用于确定调用进程的权限。尽管通常并不需要，但当用户客户端希望将某些操作限制为有管理权限才能运行时可能会需要。例如，以太网卡可以用这种机制来防止一个没有权限的进程启用混杂模式或访问所有的网络分组。

- start()方法非常简单，尽管值得注意的是传入的提供者类参数是主驱动程序类的一个实例。由于用户客户端负责接受用户空间应用程序的控制请求，并将这些请求传递给驱动程序，所以提供者类将需要主驱动程序类的一个引用。可以通过将提供者类从start()方法保存到一个实例变量实现。
- 调用clientClose()方法以响应通过调用IOServiceClose()函数关闭其驱动程序连接的调用进程，或响应在不关闭驱动程序连接的情况下终止的调用进程。用户客户端在IOService终止时调用任何其他方法（如stop()和free()）之前，将接收clientClose()。clientClose()的实现应该释放调用进程分配的任何资源，并将硬件返回至空闲状态。最后，重要的是该实现调用terminate()方法，销毁用户客户端对象。

用户客户端的作用是一个媒介，用于连接用户空间中的应用程序和内核驱动程序之间的所有通信。用户空间的请求通过驱动程序开发者定义的一个32位整数控制码进行标识。除了每个请求，应用程序还提供了一些参数，接收控制请求的返回结果。创建了用户客户端类之后，剩下的工作就是定义一个接口，将驱动程序的功能公开给用户空间应用程序。该接口包含两部分：

- 一个函数库，用户空间应用程序希望利用内核驱动程序提供的服务时，可以对其进行调用。
- 用户客户端类的一些方法，这些方法对应于每个用户函数，并提供该接口的内核端实现。

用户空间中的库函数和内核中的用户客户端方法都只需要最基本的实现，因为功能本身来自主驱动程序类。用户空间接口函数的作用是对操作及其参数进行编码，并将请求传递给驱动程序。当用户客户端接收到该请求时，它的作用是对操作及参数解码，并调用适当的驱动程序函数，实现请求的功能。

针对我们简单的用户客户端，我们将定义一个为应用程序提供定时功能的接口。用户空间接口将实现如下函数：

```
kern_return_t   StartTimer (io_connect_t connection);
kern_return_t   StopTimer (io_connect_t connection);
kern_return_t   GetElapsedTimerTime (io_connect_t connection, uint32_t* timerTime);
kern_return_t   GetElapsedTimerValue (io_connect_t connection, TimerValue* timerValue);
kern_return_t   DelayForMs (io_connect_t connection, uint32_t milliseconds);
kern_return_t   DelayForTime (io_connect_t connection, const TimerValue* timerValue);
```

所有的驱动程序接口都包含用户空间及内核用户客户端实现所需的值和类型定义。对于我们定义的接口，共享的定义包括请求码及类型定义，如TimerValue结构体。为了使这些定义在两个工程之间共享，通常会将其放置在一个公共头文件中，用户空间应用程序和内核工程都可以包含该头文件。我们驱动程序的共享定义如代码清单5-10所示。

**代码清单5-10　TestDriverInterface.h包含内核用户客户端和用户空间接口都需要的定义**

```
typedef struct TimerValue
{
        uint64_t        time;
        uint64_t        timebase;
} TimerValue;

//用于用户客户端方法的控制请求码
```

```
enum TimerRequestCode {
        kTestUserClientStartTimer,
        kTestUserClientStopTimer,
        kTestUserClientGetElapsedTimerTime,
        kTestUserClientGetElapsedTimerValue,
        kTestUserClientDelayForMs,
        kTestUserClientDelayForTime,

        kTestUserClientMethodCount
};
```

**注意** IOUserClient将一个控制请求分配给适当的处理器程序所使用的接口，在Mac OS X 10.5中已经变成支持64位内核。新的实现并不支持Mac OS X旧版本的后向兼容。本章仅描述更新后的接口。



I/O Kit框架包含一些函数，实现应用程序对驱动程序用户客户端方法的调用。应用程序选择哪一个函数来处理特定的控制请求取决于操作所需的参数类型。对于参数为整型的请求，选择IOConnectCallScalarMethod()函数比较合适，因为它可以将一个64位整数类型的变长数组从用户进程传递到内核用户客户端，并接收内核驱动程序返回的一个64位整数数组，其中包含操作的结果：

```
kern_return_t   IOConnectCallScalarMethod(
                        io_connect_t            connection,
                        uint32_t                selector,
                        const uint64_t*         inputValues,
                        uint32_t                inputCount,
                        uint64_t*               outputValues,
                        uint32_t*               outputCount);
```

该函数的第一个参数是与驱动程序用户客户端的一个连接，调用者通过调用IOServiceOpen()可以预先获得该连接。下一个参数selector是驱动程序定义的控制码，描述用户客户端应该执行哪一个操作。这是代码清单5-10中定义的TimerRequestCode枚举中的一个值。该函数的其余参数是应用程序传递的用户客户端执行操作时所需的一些变量，以及接收用户客户端返回的一些操作结果。标记为输入的参数由调用应用程序提供给用户客户端，标记为输出的参数由用户客户端返回给应用程序。outputCount参数既是输入参数又是输出参数，调用者将其值初始化为outputValues数组中的成员个数，告诉用户客户端它可以安全地向数组中写入多少个值。当函数完成时，实际写入到outputValues数组中值的个数在outputCount中返回。如果调用者不需要用户客户端返回任何值，则只需传递NULL作为outputCount参数即可。

如果一个控制请求具有不同类型的参数，比较常见的做法是定义一个结构体，将提供给用户客户端的所有参数进行合并，再定义一个结构体，接收该方法的结果。这由I/O Kit框架的IOConnectCallStructMethod()函数提供：

```
kern_return_t   IOConnectCallStructMethod(
                        io_connect_t    connection,
```

```
                    uint32_t        selector,
                    const void*     inputStruct,
                    size_t          inputStructSize,
                    void*           outputStruct,
                    size_t*         outputStructSize);
```

包含输入参数的结构体指针通过inputStruct参数提供，该输入结构体的长度（以字节为单位）作为inputStructSize参数进行传递。与标量函数一样，outputStructSize参数既是输入参数又是输出参数。调用者将其值初始化为outputStruct缓冲区的最大长度（以字节为单位），并且在完成之后，用户客户端通过outputStructSize参数返回outputStruct缓冲区中写入的字节数。

你可能会奇怪为什么返回的结构体长度与预期的输出结构体长度不同。IOConnectCallStructMethod()函数可以用来读取用户客户端的变长数组或变长字符串。在这些实例中，调用者预先并不知道返回的字节数。如果控制请求不带有输入结构体，可以将NULL作为inputStruct参数进行传递，同时inputStructSize为0字节。如果控制请求并不返回输出结构体，可以将NULL同时作为outputStruct和outputStructSize参数进行传递。

I/O Kit框架还可以在含有整型输入/输出参数以及需要输入/输出结构体参数的函数中，混合使用标量参数和结构体参数：

```
kern_return_t   IOConnectCallMethod(
                    io_connect_t    connection,
                    uint32_t        selector,
                    const uint64_t* inputValues,
                    uint32_t        inputCount,
                    const void*     inputStruct,
                    size_t          inputStructSize,
                    uint64_t*       outputValues,
                    uint32_t*       outputCount,
                    void*           outputStruct,
                    size_t*         outputStructSize);
```

尽管I/O Kit框架提供了3个调用用户客户端方法的函数，但IOConnectCallScalarMethod()和IOConnectCallStructMethod()函数只是在IOConnectCallMethod()基础上构建的便捷函数。

考虑到I/O Kit框架为调用用户客户端方法而提供的3个函数，我们来看一下如何实现示例驱动程序的用户空间接口。StartTimer()和StopTimer()函数并不向用户客户端传递额外的参数，因此它们可以通过调用3个IOConnectCallXXX()函数中的任意一个来实现。在此，我们选择使用IOConnectCallMethod()函数来实现StartTimer()和StopTimer()。StartTimer()的用户空间实现如下：

```
kern_return_t   StartTimer (io_connect_t connection)
{
        return IOConnectCallMethod(connection, kTestUserClientStartTimer,
                                   NULL, 0, NULL, 0, NULL, NULL, NULL, NULL);
}
```

GetElapsedTimerTime()的实现是从用户客户端读取一个32位整数，因此很自然地会使用IOConnectCallScalarMethod()函数对其进行实现。I/O Kit使用的标量值表示的整数类型是64位无

符号整数，而不是调用者预期的32位值，因此，需要通过一个临时的64位变量接收用户客户端的结果，并对其进行转换。在IOConnectCallScalarMethod()函数执行完毕后，我们并不检查scalarOutCount的值，因为可以确信，在操作成功完成后，我们的驱动程序实现已经返回了一个整数。

```
kern_return_t   GetElapsedTimerTime (io_connect_t connection, uint32_t* timerTime)
{
        uint64_t                scalarOut[1];
        uint32_t                scalarOutCount;
        kern_return_t           result;

        scalarOutCount = 1;     //初始化为scalarOut数组的长度
        result = IOConnectCallScalarMethod(connection, kTestUserClientGetElapsedTimerTime,
                                        NULL, 0, scalarOut, &scalarOutCount);
        if (result == kIOReturnSuccess)
                *timerTime = (uint32_t)scalarOut[0];

        return result;
}
```

最后，我们看一下用户空间库中DelayForTime()函数的实现。该函数通过一个结构体将其参数传递给用户客户端，因此我们使用IOConnectCallStructMethod()函数来实现它。

```
kern_return_t   DelayForTime (io_connect_t connection, const TimerValue* timerValue)
{
        return IOConnectCallStructMethod(connection, kTestUserClientDelayForTime,
                                        timerValue, sizeof(TimerValue), NULL, 0);
}
```

用户空间应用程序的所有请求都会调用用户客户端类中的externalMethod()方法，该方法负责分配适当的方法来处理控制请求，并解压用户空间应用程序提供的参数。IOUserClient基类提供了externalMethod()的一个实现，但任何支持控制请求的用户客户端都应该覆写该基类的实现。在进一步深入查看驱动程序的用户客户端如何实现externalMethod()之前，有必要看一下它的接口：

```
IOReturn IOUserClient::externalMethod (uint32_t selector, IOExternalMethodArguments* args,
                        IOExternalMethodDispatch* dispatch, OSObject* target,
                        void* reference);
```

I/O Kit调用你类中的externalMethod()实现时只使用了前两个参数，而后面三个参数dispatch、target和reference的值都设置为NULL。第一个参数selector是一个32位控制码，指定客户端应用程序请求的是哪一个操作。下一个参数args包含所有的标量及结构体参数，由应用程序传递给用户客户端。IOExternalMethodArguments结构体的定义如下面的代码段所示，为了简单起见，此处省去了部分字段：

```
struct IOExternalMethodArguments
{
        …
        const uint64_t*         scalarInput;
        uint32_t                scalarInputCount;
```

```
        const void*             structureInput;
        uint32_t                structureInputSize;

        IOMemoryDescriptor*     structureInputDescriptor;

        uint64_t*               scalarOutput;
        uint32_t                scalarOutputCount;

        void*                   structureOutput;
        uint32_t                structureOutputSize;

        IOMemoryDescriptor*     structureOutputDescriptor;
        uint32_t                structureOutputDescriptorSize;
};
```

IOExternalMethodArguments结构体的字段看起来应该很熟悉，因为它们与用户空间IOConnectCallMethod()函数的参数列表几乎完全相同。structureInputDescriptor和structureOutputDescriptor这两个字段需要特别说明一下。它们用于在用户进程和内核驱动程序之间传递4096字节以上的结构体。对于小于虚拟内存页长度的结构体，I/O Kit将在用户缓冲区和内核内存缓冲区之间复制整个结构。对于较大的缓冲区，I/O Kit会创建一个IOMemory- Descriptor对象，直接引用用户进程地址空间的缓冲区。第6章将介绍IOMemoryDescriptor类。

考虑到externalMethod()的前两个参数已经为该实现提供了执行请求操作所需的全部信息，你可能会想剩余的3个参数负责什么。驱动程序的用户客户端采用传统的方案来实现externalMethod()，调用IOUserClient超类的实现为其方法的selector和args参数传值，但为dispatch、target及reference参数传值是为了描述处理该控制请求时应该调用的类方法。

尽管无法阻止驱动程序不调用超类而实现externalMethod()，但使用IOUserClient类提供实现的优点是，这一实现会验证调用进程提供的参数。IOUserClient执行的验证只能确保用户客户端应已经接收到预期的调用进程参数，例如，确保进程已经提供正确数量的标量输入及输出参数，以及进程提供的结构体参数长度与驱动程序将要读写的结构体长度相匹配。为此，IOUserClient的externalMethod()实现需要知道驱动程序对控制请求的预期参数。这些信息来自IOExternalMethodDispatch结构体，可以通过dispatch参数将该结构体传递给超类实现。IOExternalMethodDispatch的类型定义如下：

```
struct IOExternalMethodDispatch
{
        IOExternalMethodAction          function;
        uint32_t                        checkScalarInputCount;
        uint32_t                        checkStructureInputSize;
        uint32_t                        checkScalarOutputCount;
        uint32_t                        checkStructureOutputSize;
};
```

IOExernalMethodDispatch结构体描述处理控制请求时应该调用的回调函数、标量输入/输出参数的数量，以及回调函数预期的输入/输出结构体参数长度。如果回调函数能够接受可变数量的标量参数或可变长度的结构体，常量kIOUCVariableStructureSize便可以写入到IOExternalMethod-

Dispatch结构体的对应字段中。

该回调函数的签名如下：

```
typedef IOReturn (*IOExternalMethodAction)(OSObject* target, void* reference,
                        IOExternalMethodArguments* args);
```

你会发现，该回调函数的参数正好对应IOUserClient::externalMethod()的3个参数，包括target和reference。这并不是巧合。驱动程序调用externalMethod()的超类实现时，会对用户空间进程提供的参数进行验证，如果参数正确，则调用指定的处理程序函数，将target、reference及args的值传递给回调函数。由于回调函数是一个静态方法，所以它无法通过this指针访问用户客户端的实例变量。因此，target的值应该设置为处理控制请求的用户客户端实例，如果控制请求由另外一个类的方法处理，target则应该设置为那个不同类的实例。reference值可由用户客户端把任意数据自由传递给回调函数。

处理用户空间客户端控制请求的整个流程，包括实现驱动程序自定义的用户客户端及标准的IOUserClient所执行的步骤，如以下伪代码所示：

```
// 驱动程序的IOUserClient子类提供的实现
IOReturn          MyUserClient::externalMethod (selector, args, dispatch, target, reference)
{

        使用selector确定已经请求了哪一个控制请求

        将newDispatch初始化为适当的回调函数，以处理控制请求
        将newTarget初始化为当前MyUserClient实例
        如果希望为回调函数提供额外数据，初始化newReference

        调用超类：
        IOUserClient::externalMethod(selector, args, newDispatch, newTarget, newReference)
}

// I/O Kit超类提供的实现
IOReturn          IOUserClient::externalMethod(selector, args, dispatch, target, reference)
{
        检查用户进程通过args提供的参数是否与dispatch中用户客户端预期的参数匹配

        如果参数不匹配，则以kIOReturnBadArgument结果退出

        否则，为此控制请求调用回调处理程序
        Dispatch->function(target, reference, args)
}
```

按照惯例，大多数驱动程序都通过为其用户客户端类添加一个调度表来实现externalMethod()，其中包含用户客户端接受的每个选择器值的IOExternalMethodDispatch值。我们的定时器用户客户端的一个可能的调度表，如代码清单5-11所示。

**代码清单5-11 演示示例用户客户端接口的调度表**

```
const IOExternalMethodDispatch
com_osxkernel_driver_IOKitTestUserClient::sMethods[kTestUserClientMethodCount] =
```

```
{
        // kTestUserClientStartTimer   (void)
        { sStartTimer, 0, 0, 0, 0 },

        // kTestUserClientStopTimer   (void)
        { sStopTimer, 0, 0, 0, 0 },

        // kTestUserClientGetElapsedTimerTime   (uint32_t* timerValue)
        { sGetElapsedTimerTime, 0, 0, 1, 0 },

        // kTestUserClientGetElapsedTimerValue   (TimerValue* timerValue)
        { sGetElapsedTimerValue, 0, 0, 0, sizeof(TimerValue) },

        // kTestUserClientDelayForMs   (uint32_t milliseconds)
        { sDelayForMs, 1, 0, 0, 0 },


        // kTestUserClientDelayForTime  (const TimerValue* timerValue)
        { sDelayForTime, 0, sizeof(TimerValue), 0, 0 }
};
```

为方便起见，使用TimerRequestCode枚举中的值作为数组的索引访问调度表中的项，因此，调度表中项的顺序要与代码清单5-10中定义的请求码顺序相符，这很重要。

通过使用定义的调度表，实现用户客户端的externalMethod()则变得非常容易，因为它需要的几乎所有值都可以直接来自调度表。一个可能的实现如代码清单5-12所示。

**代码清单5-12　自定义用户客户端externalMethod()的实现**

```
IOReturn        com_osxkernel_driver_IOKitTestUserClient::
                externalMethod (uint32_t selector, IOExternalMethodArguments* arguments,
                        IOExternalMethodDispatch* dispatch, OSObject* target,
                        void* reference)
{
        //确保请求的控制选择器在范围内
        if (selector >= kTestUserClientMethodCount)
            return kIOReturnUnsupported;

        dispatch = (IOExternalMethodDispatch*)&sMethods[selector];
        target = this;
        reference = NULL;
        return super::externalMethod(selector, arguments, dispatch, target, reference);
}
```

最后，为了完成用户客户端，我们来看一下两个选择器的可能实现：GetElapsedTimerTime()和DelayForTime()，这两个选择器由自定义用户客户端提供。我们看到，用户客户端实现的每个选择器都有一个对应的回调函数，用来调用处理选择器。该回调函数的参数通过IOExternalMethodArguments结构体进行传递。如果作为参数传递给函数，那么参数处理将变得更容易，因此，通常每个控制选择器都有两个方法处理程序：静态回调处理程序和提供具体实现的实例方法。

静态回调方法对IOExternalMethodArguments结构体的参数进行解压，并将其传递给用户客户端类的实例方法或主驱动程序类，执行实际的工作。GetElapsedTimerTime()和DelayForTime()方

法的实现处理如代码清单5-13所示。

**代码清单5-13 两个用户客户端方法的实现**

```
IOReturn        com_osxkernel_driver_IOKitTestUserClient::
                sGetElapsedTimerTime (OSObject* target, void* reference,
                                      IOExternalMethodArguments* arguments)
{
        com_osxkernel_driver_IOKitTestUserClient*       me;
        uint32_t                timerTime;
        IOReturn                result;

        me = (com_osxkernel_driver_IOKitTestUserClient*)target;

        //调用实现该操作的方法
        result = me->getElapsedTimerTime(&timerTime);
        //将操作的标量结果返回给调用进程
        arguments->scalarOutput[0] = timerTime;

        return result;
}

IOReturn        com_osxkernel_driver_IOKitTestUserClient::
                sDelayForTime (OSObject* target, void* reference,
                               IOExternalMethodArguments* arguments)
{
        com_osxkernel_driver_IOKitTestUserClient*       me;

        me = (com_osxkernel_driver_IOKitTestUserClient*)target;
        return me->delayForTime((TimerValue*)arguments->structureInput);
}
```

请注意，我们无须验证sGetElapsedTimerTime()实现中scalarOutput数组的长度，或sDelayForTime()实现中structureInput缓冲区的长度。因为我们知道这些参数已经由IOUserClient超类的externalMethod()验证，它已经将提供的参数与驱动程序预期的参数进行了比较。但getElapsedTimerTime()的实际实现会验证客户端在试图读取定时器的时间之前，是否已经调用StartTimer()，预先启动了定时器。

---

**注意** 驱动程序的用户客户端应该一直验证从用户进程接收的参数值。该进程可能已经损坏，或是由第三方开发，而他们提供的参数值不一定是你的驱动程序想要接收的值。如果不能拒绝非法的参数值，就会引起驱动程序产生内核恐或使系统变得非常脆弱。

---

## 5.6 驱动程序的通知

我们在示例驱动程序的用户客户端中实现的方法都是为了阻塞函数，也就是说，用户客户端完成请求的操作后，用户进程才会继续执行。例如，我们用户客户端中的DelayForMs()方法将暂

停调用它的线程，直至指定的延迟已经结束。虽然这对旨在显式延迟调用进程的函数来说可能并不糟糕，但用户空间应用程序可能并不总是希望等待驱动程序操作的完成，特别是操作需要的时间不确定或取决于驱动程序无法控制的事件时，如串行端口上的数据到达。

有两个方案可以解决该问题。Windows驱动程序模型并不允许应用程序向一个驱动程序同时发送多个控制请求（除非显式启用），与此不同，I/O Kit允许与应用程序所需数量相同的线程向用户客户端发送请求。这意味着阻塞操作的一个方案是，为应用程序创建一个二级线程，在该二级线程上调用阻塞的用户客户端方法。这使应用程序的其他部分在驱动程序处理请求时可以继续执行。第二个方案是实现用户客户端的异步操作，并在操作完成时通过一个回调函数通知应用程序。

对开发者来说，I/O Kit框架使一个操作从同步、阻塞操作转变为异步操作变得非常容易。为了看一看这是如何完成的，我们对本章中开发的驱动程序接口进行扩展，包含进一个`InstallTimer()`函数，延迟指定的时间段，然后通过一个回调函数通知应用程序。这样，`InstallTimer()`函数可以看做是现有`DelayForTime()`函数的一个异步实现。

I/O Kit框架针对异步通知的实现，与本章开头讲述的设备接入时通知的发送非常相似（参见代码清单5-2）。实际上，设备接入时接收的通知可以看做是I/O Kit框架本身实现的一种特殊异步完成回调。为了初始化驱动程序库以发送完成回调，我们需要创建一个通知端口，在操作完成时，内核通过该端口通知用户空间进程。这通过调用`IONotificationPortCreate()`函数实现，与我们创建一个通知端口发送设备接入通知的方式相同。尽管该通知端口由驱动程序的用户空间库分配，但通过提供一个函数，使得应用程序可以访问该端口，以便应用程序在其选择的运行回路上安装该通知端口，这是一个非常好的设计。驱动程序通知端口的存取函数的一个可能实现，如代码清单5-14所示。

**代码清单5-14　分配一个端口，应用程序在异步操作完成时可以通过该端口接收通知**

```
IONotificationPortRef   gAsyncNotificationPort = NULL;

IONotificationPortRef   MyDriverGetAsyncCompletionPort ()
{
        //如果端口已分配，则返回现有实例
        if (gAsyncNotificationPort != NULL)
                return gAsyncNotificationPort;

        gAsyncNotificationPort = IONotificationPortCreate(kIOMasterPortDefault);
        return gAsyncNotificationPort;
}
```

应用程序随后可以在它其中一个运行回路上分配和安装该通知端口，如下所示：

```
CFRunLoopSourceRef      runLoopSource;
notificationPort = MyDriverGetAsyncCompletionPort ();
runLoopSource = IONotificationPortGetRunLoopSource(notificationPort);
CFRunLoopAddSource(CFRunLoopGetCurrent(), runLoopSource, kCFRunLoopDefaultMode);
```

分配了一个通知端口之后，用户空间应用程序可以通过该端口接收内核驱动程序的消息，现在需要将该端口提供给内核驱动程序，以便能通知异步操作的完成。每个异步控制请求发出时，

该通知端口都会提供给驱动程序的用户客户端。I/O Kit框架提供了每个IOConnectCallXXX()函数的异步变体，名为IOConnectCallAsyncXXX()。这些函数的异步形式带有额外的参数，包括一个通知端口、一个回调函数和一个传递给回调函数的上下文参数。

从各方面来看，IOConnect函数的异步变体与其对应的同步函数具有同样的行为。例如，IOConnectCallAsyncScalarMethod()将一个整数值数组传递给用户客户端，并接收用户客户端的一个整数值数组。与这些函数的同步形式一样，任何输出参数都是在函数返回后立即写入，而不是异步操作完成时才写入的（函数返回时，驱动程序仍然可以处理该操作）。

为了提供一个示例，演示如何使用异步函数，我们查看一下示例函数InstallTimer()的实现。该用户空间代码如代码清单5-15所示。



**代码清单5-15 驱动程序的InstallTimer()函数异步控制请求的用户空间实现**

```
kern_return_t   InstallTimer (io_connect_t connection, uint32_t milliseconds,
                        IOAsyncCallback0 timerCallback, void* context)
{
        io_async_ref64_t        asyncRef;
        uint64_t                scalarIn[1];

        //设置回调函数
        asyncRef[kIOAsyncCalloutFuncIndex] = (uint64_t)timerCallback;
        asyncRef[kIOAsyncCalloutRefconIndex] = (uint64_t)context;

        //设置输入参数
        scalarIn[0] = milliseconds;
        return  IOConnectCallAsyncScalarMethod(connection, kTestUserClientInstallTimer,
                IONotificationPortGetMachPort(gAsyncNotificationPort),
                asyncRef,  kIOAsyncCalloutCount,
                scalarIn, 1, NULL, NULL);
}
```

如果对上面的函数和DelayForTime()实现进行比较，你会发现，这两个函数有许多共同点，它们都向用户客户端传递同样的输入和输出参数。两个函数之间的唯一区别在于是否添加了timerCallback和context参数。回调函数及其上下文参数都通过io_async_ref64_t结构体提供给IOConnectCallAsync函数，该结构体定义为一个无符号的64位整数数组。因为io_async_ref64_t数组的某些成员供I/O Kit内部使用，所以应用程序应该使用kIOAsyncCalloutFuncIndex和kIOAsyncCalloutRefconIndex常量访问该数组，如示例所示。

这是应用程序执行异步操作时需要执行的全部工作。异步操作完成时，提供的回调函数将得到通知，并在应用程序安装运行回路源的运行回路上执行。回调函数的签名如下所示：

```
typedef void (*IOAsyncCallback0)(void* context, IOReturn result);
```

请注意，IOAsyncCallback0中的“0”是该函数接收的驱动程序参数的个数。这些参数与传递给IOConnectCallAsyncScalarMethod()函数的输出标量计数是分开的，在异步操作完成时由内核指定。

当然，对于异步操作，内核驱动程序必须确保它可以从用户客户端立即返回，但仍会继续处

理后台中请求的操作，并在操作完成时通知进程。用户客户端接收的异步方法调用与任何其他方法调用并没有什么不同，均由externalMethod()分配给处理程序函数，该处理程序函数通过IOExternalMethodArguments结构体接收其参数。但与同步方法不同，IOExternalMethodArguments结构体含有对异步操作有用的字段，如以下定义所示：

```
struct IOExternalMethodArguments
{
        …
        mach_port_t             asyncWakePort;
        io_user_reference_t*    asyncReference;
        uint32_t                asyncReferenceCount;
        …
};
```

通过检查asyncWakePort的值，实现一个控制请求的方法可以确定用户应用程序是否通过一个异步函数调用对其进行了调用。如果是非0，则请求了一个异步操作。假如处理程序将在后台执行请求的操作（因为一个异步控制请求应该避免阻塞调用的应用程序），则需要保存执行操作时涉及的IOExternalMethodArguments结构体中的所有值。这包括复制调用者提供的任何标量及结构体输入参数值（请注意，输出值是在用户客户端返回时立即返回给调用的应用程序，而不是在异步操作完成时）。IOExternalMethodArguments结构体中需要保存的一个重要值是asyncReference缓冲区，因为这用来在操作完成时为应用程序签名。

异步操作的执行示例如代码清单5-16所示。

**代码清单5-16　异步操作的用户客户端实现，InstallTimer()函数的内核端实现**

```
//保存后台操作所需参数的结构体
struct TimerParams
{
        OSAsyncReference64              asyncRef;
        uint32_t                        milliseconds;
        OSObject*                       userClient;
};

IOReturn  com_osxkernel_driver_IOKitTestUserClient::
      sInstallTimer (OSObject* target, void* reference, IOExternalMethodArguments* arguments)
{
        TimerParams*    timerParams;
        thread_t        newThread;

        //分配一个结构体，存储定时器需要的参数
        timerParams = (TimerParams*)IOMalloc(sizeof(TimerParams));
        //取得asyncReference缓冲区的一个副本
        bcopy(arguments->asyncReference, timerParams->asyncRef, sizeof(OSAsyncReference64));

        //取得用户应用程序提供的milliseconds值的一个副本
        timerParams->milliseconds = (uint32_t)arguments->scalarInput[0];
        //取得userClient对象的一个引用
        timerParams->userClient = target;
        //异步操作执行时，保留该用户客户端
```

```
        target->retain();

        //启动一个后台线程，在返回给调用者之后继续执行操作
        kernel_thread_start(DelayThreadFunc, timerParams, &newThread);
        thread_deallocate(newThread);

        //立即返回给调用应用程序
        return kIOReturnSuccess;
}

void    com_osxkernel_driver_IOKitTestUserClient::
                DelayThreadFunc (void *parameter, wait_result_t)
{
        TimerParams*    timerParams = (TimerParams*)parameter;

        //按请求的时间睡眠
        IOSleep(timerParams->milliseconds);
        //向用户应用程序发送操作完成的通知
        sendAsyncResult64(timerParams->asyncRef, kIOReturnSuccess, NULL, 0);

        //后台操作已经完成
        //释放用户客户端对象的其他引用
        timerParams->userClient->release();

        IOFree(timerParams, sizeof(TimerParams));
}
```

尽管前面的代码中有很多内容需要理解，但需要注意的一点是，它分配一个对象以保存参数的方式，这些参数是驱动程序在后台线程上完成请求的操作时所需的参数。为了防止用户客户端对象在操作处理过程中被释放，该方法会在操作启动时增加其引用计数，操作完成时减少引用计数。最后，后台操作完成时，用户客户端（或驱动程序）会调用`sendAsyncResult64()`为用户应用程序签名。本例中没有用到`sendAsyncResult64()`的最后两个参数，利用这两个参数驱动程序可以为应用程序的回调函数提供额外的值。例如，一个异步读操作可以用这一功能返回读取的字节数。

## 5.7 小结

- 几乎所有的驱动程序都需要将其服务公开给用户运行的应用程序。
- 为了使应用程序可以与内核驱动程序交互，驱动程序需要跨越用户空间代码与内核代码之间的界限。采用I/O Kit框架编写的驱动程序，通过实现`IOUserClient`类的派生类来跨越这一界限。
- 应用程序可以迭代加载的内核驱动程序或安装一个回调函数，接收驱动程序加载/卸载时的通知。
- I/O Kit提供了一些函数，利用它们，用户应用程序可以请求驱动程序提供的服务，方法包括读写驱动程序的属性，或建立与驱动程序的连接，并通过该连接向驱动程序发送控制请求。

# 第6章 内存管理

内核中的内存管理要比用户空间程序的内存管理复杂得多。用户空间程序通常处理平面线性地址空间，可以分配任意块内存，而无须关注该内存的来源或布局。用户空间程序具有简单的接口，其接口通常以字节长度为参数，并根据请求内存的可用性提供一个是或否的结果。分配失败或滥用内存的最坏结果是损害系统的进程会终止，但在内核中，情况并不是这么简单。内核必须处理多个内存空间，其中包括自有的内存空间以及这些内存空间与物理内存之间的映射关系。虽然用户空间程序可以处理虚拟内存，并且这与底层物理布局并不相关，但内核常常需要知道内存是否连续及其分布位置。这是因为一些硬件设备无法读取某些内存地址，或对内存对齐有特定需求，例如，只能读取16字节边界对齐的内存或无法读取地址高于32位的内存。但内核内存管理最明显的挑战是如何尽可能地少用内存，因为内存是稀有资源，特别是在iPhone或iPad这类嵌入式设备上。错误使用内核中的内存将会导致或许是不易察觉的后果。

本章将说明内核程序员会遇到的各种类型的内存及其用途，以及如何最有效、最安全地使用内存。我们还将讨论分配、管理内存的机制与方法，以及操作系统管理内存的一些底层机制。本章还将讲述如何执行内存映射操作，将一个地址空间的内存映射到另一个任务的地址空间。

## 6.1 内存类型

内核可以处理多种类型的内存，因此理解它们之间的差别是实现一个成功的驱动程序或内核扩展的关键。

内存类型可以分为：

- CPU物理地址；
- 总线物理地址；
- 用户和内核虚拟地址。

除了这三种类型的内存地址，不同体系结构之间可寻址的内存数量也各不相同，可以从32位到64位。内存还会根据体系结构进行不同的排序，可以是小端序或大端序。

接下来的几节将讨论每种类型的内存在内核编程中的重要性和使用方法。

### 6.1.1 CPU物理地址

物理地址是指CPU为访问物理内存而使用的寻址系统。通常，物理地址隐藏在CPU的MMU（Memory Management Unit，内存管理单元）之后。MMU会将内核和用户空间常用的虚拟地址转换为物理地址。物理地址空间呈线性，32位系统从0到0xffffffff（$2^{32}$），64位系统从0到0xffffffffffffffff（$2^{64}$）。对物理内存的访问会缓存在较小的内存缓冲器中，如L1和L2缓存通常包含在CPU核心中。

通常无须直接处理物理地址，即使是编写驱动程序也无需如此。

**物理内存扩展**

PAE（Physical Address Extension，物理地址扩展）是英特尔开发的一个特性，它让物理地址空间变得更大，解决了32位系统上4 GB的内存限制。PAE可以用在所有支持英特尔处理器的Mac OS X版本之上（10.4.4及更高版本）。PAE将可用地址空间扩展到了36位（显然，并不存在36位的数据类型，因此使用64位的类型表示地址），这使可用地址空间可以寻址到64 GB的物理内存。但PAE并不改变进程使用的虚拟地址空间大小，虚拟地址空间仍然限定为4 GB。虽然没有进程（或内核）会使用4 GB以上的内存，但总体来说系统最多可以使用64 GB内存。

### 6.1.2 总线物理地址

64位计算的引入提出了一项新的挑战，因为传统的I/O总线（如PCI和PCI-X）无法访问32位以上的内存地址。为了解决这个问题，基于PowerPC G5的Mac在其北桥上有一个额外的MMU，可以将内存从64位地址重新映射到设备可以读取的32位地址。该MMU称为设备地址转化表（Device Address Resolution Table，DART）。DART将转换后的内存作为物理地址提供给设备，但这些地址已经过转换，与CPU使用的物理地址并不相同。基于英特尔的计算机具有类似的功能，称为I/O内存管理单元（I/O memory management unit，IOMMU），是面向直接I/O（VT-d）的虚拟化技术之一。

总线物理地址看起来像是硬件设备的物理地址，但实际上，这是经DART转换后的虚拟地址。如果你不清楚，也不必担心，因为很少需要处理这些地址。实际上，如果使用I/O Kit，那么使用`IOMemoryDescriptor`时，I/O Kit将自动执行所需转换，本章后面会介绍`IOMemoryDescriptor`。驱动程序可以使用`IOPhysicalAddress`类型处理物理地址，该类型的长度取决于底层体系结构。因为PAE即使在32位的操作系统上，地址也可以是64位。

### 6.1.3 用户和内核虚拟地址

虚拟地址呈线性，通过CPU上一个名为内存管理单元（MMU）的特殊芯片转换为物理地址。每个用户空间进程都有自己的内存地址空间，从各方面看就像一个进程拥有所有的物理内存。用户空间进程可以使用其地址空间中任意的内存位置，即使地址位置超出了物理内存数量。对一个进程来说，虚拟地址空间看起来呈线性，尽管支撑它的内存可能是碎片。

在Mac OS X中，进程可以使用整个虚拟地址空间。在32位系统上，这包括0~4 GB的内存地址。Microsoft Windows、Linux等操作系统都采用拆分模型，将内核映射到每个进程的虚拟地址空间。例如，在Windows（32位）上，用户空间虚拟内存的地址范围是0~0x7FFFFFFF，而为内核保留的内存地址是0x80000000~0xFFFFFFFF。因为内核已经进行了映射，所以进程上下文切换至内核模式时（一项代价很高的操作），CPU无须改变页表。此方案的缺点是内核和用户空间进程的可用地址空间较少，因此，就Windows而言，内核或用户空间进程在某一时刻只可以访问2 GB的地址空间。在Linux上，通常拆分为3 GB/1 GB，只有1 GB用于内核（虽然Linux中的内容都可配置，还有其他可用的配置）。如果系统启用了GPU，这通常要预装最多1 GB的板载内存，板载内存必须映射到虚拟地址空间，可能会导致一些物理内存无法使用，因其被GPU的大帧缓冲器遮蔽了。

为了避免遮蔽问题，Mac OS X对内核（4 GB）和用户空间进程（4 GB）使用了完全独立的地址空间，但正如前文所说，其缺点是上下文切换代价较高。

Mac OS X 10.6 Snow Leopard中引入了64位内核，彻底解决了地址空间受限问题。在64位内核中，内核地址空间总是会被映射进来。Mac OS X对内核地址空间进行了拆分，高128 T字节预留给内核，而低128 T字节则属于当前运行的用户空间任务。虽然该地址空间与用户空间共享，但因为页保护标识，任务不能访问内核内存。

虚拟内存地址可能不会一直依靠物理内存区，因为该内存并不常用，或者运行的进程需要更多的内存，内存也许已经迁移到硬盘等外部后备存储器。如果CPU访问一个地址，但该地址的内存并不存在，则会导致*缺页异常*。*分页器*是操作系统的一个组件，用于获取包含给定内存地址的页。

进程不可以访问虚拟地址空间的第一页（0~4 KB），如果试图访问，将会发生异常。

与体系结构无关的类型`IOVirtualAddress`可用于处理I/O Kit代码中的虚拟地址。该类型又是`mach_vm_address_t`的别名，是Mach层中用于虚拟内存地址的类型。

**提示**　有关虚拟内存的详细内容，请参见第1章；有关OS X和iOS实现的细节，请参见第2章。

### 6.1.4　内存顺序：大端序和小端序

字节序是内存中二进制字的组成顺序。根据使用的CPU体系结构，可以分为小端序和大端序。下面使用一个简单的C语言程序演示小端序和大端序的效果，如代码清单6-1所示。

**代码清单6-1　输出一个32位字的字节序**

```
#include <stdio.h>
#include <stdint.h>
int main(int argc, char *argv[])
{
        uint32_t word = 0xaabbccdd;
```

```
    uint8_t* ptr = (uint8_t*)&word;
    printf("%02x %02x %02x %02x\n", ptr[0], ptr[1], ptr[2], ptr[3]);
    return 0;
}
```

在小端序系统上的执行结果是：

```
dd cc bb aa
```

而在大端序系统上是：

```
aa bb cc dd
```

可以看出，小端序系统上的顺序为逆向。当前所有的Mac都是小端序，因为英特尔x86/x86_64处理器是小端序；基于ARM的iOS设备也是如此。基于PowerPC的旧Mac都是大端序。那么为什么要关注大端序呢？因为一些硬件体系结构或网络协议（如TCP/IP）使用的是大端序；此外，驱动程序或内核扩展可能还要兼容旧的基于PowerPC体系结构的Mac。另外，OS X支持Rosetta，可以在基于英特尔的Mac上模拟PowerPC应用程序。你的驱动程序可以由Rosetta客户端任务访问。一些用户空间API（如Carbon文件管理器）也可以处理大端序数据结构。

编译器定义了C预处理器宏__LITTLE_ENDIAN__和__BIG_ENDIAN__，可以用来确定编译时的字节顺序。

### 6.1.5 32位和64位内存寻址

目前，现代Mac OS X系统都是64位。这意味着CPU可以处理64位宽的地址，包括通用寄存器，并且可以使用64位数据总线和64位虚拟内存寻址。

**Intel 64体系结构**

Intel 64（x86-64）体系结构是传统英特尔x86指令集的扩展，可以在64位模式下执行操作，并支持大量的物理内存。虽然英特尔发明了x86兼容处理器，但该扩展最初由AMD创建，并命名为AMD64上市销售。英特尔随后发布了自己的64位扩展版本，最初命名为EM64T和IA-32e，可以与AMD的解决方案兼容。英特尔最初投注在全新设计的IA64（Itainum）上。IA64抛弃了x86的传统。HP和其他高性能服务器厂商（如SGI）重点推行IA64，但进展缓慢。Intel 64/AMD64现在仍然是居于主导地位的体系结构。所有当代Mac都兼容Intel 64的CPU。x86-64处理器有两种操作模式：**长模式**（long mode）和**传统模式**（legacy mode）。前者是64位模式，并且兼容32位和16位的应用程序。操作系统必须在长模式下进行64位操作。后者是32位模式，只用于32位操作系统。

表6-1描述了支持的寻址模式，以及OS X和iOS支持的体系结构的本机指针长度。

表6-1　OS X和iOS在各种平台下的内存寻址

| 体系结构 | 32位应用程序 | | | | |
|---|---|---|---|---|---|
| | 64位内核 | 64位应用程序 | （在64位模式中） | 32位内核 | 指针长度 |
| 32位PowerPC | 否 | 否 | 不适用 | 是 | 4 |
| 64位PowerPC | 否 | 是 | 是 | 否 | 8 |
| 32位Intel | 否 | 否 | 不适用 | 否 | 4 |
| 64位Intel | 是 | 是 | 是 | 是 | 8 |
| iOS | 否 | 否 | 是 | 是 | 4 |

因为应用程序在64位模式中运行时，内核可以在32位模式中运行，所以一个64位进程通过ioctl()或IOUserClient等方法与内核交换数据时必须极其小心。运行的64位内核与32位应用程序通信时也是如此。但问题是，32位和64位编译器定义的数据类型不同。例如，C语言数据类型long的字长在32位的程序中是4字节，而在编译为64位指令集的程序中却是8字节。

## 6.2　内存分配

XNU内核提供了很多可以进行内存分配的工具。内核内存分配并不像用户空间库中的malloc()/free()接口那么简单。内核内存分配设备的范围从类似于用户空间malloc()接口的高层机制到对原始页的直接分配。获得内存的不同函数有许多个，具体采用哪一个取决于你使用的子系统，例如Mach、BSD或I/O Kit，以及对内存的需求，如大小或对齐。内存无疑是计算机系统上最有限的资源之一，尤其是对iOS平台而言，相比大多数基于Mac OS X的计算机来说，iOS平台的物理内存非常有限。

在最底层，内核使用vm_page结构体记录物理内存。每个vm_page结构体都对应一个物理内存页。可用页是下列某个页表的一部分。

- 活动链表（Active List）：包含映射到至少一个虚拟地址空间的物理内存页，并且最近使用过。
- 惰性链表（Inactive List）：包含已分配但最近未使用的页。
- 空闲链表（Free List）：包含未分配的页。

如果要从空闲链表取得一个空闲页，可以使用vm_page_grab()函数或其高层接口vm_page_alloc()，不同于vm_page_grab()函数，vm_page_alloc()是将页放置到vm_object对象中，而不是将其从空闲链表中移除。如果内核检测到空闲页的等级低于阈值，则将通知分页守护进程。在这种情况下，分页器将根据最近最少使用（least recently used，LRU）原则，从惰性链表逐出页。磁盘文件映射的页是主要的驱逐对象，可以轻易地丢弃。在Mac OS X和iOS上，VM页缓存和文件系统缓存组合在一起，可以避免复制，统称为统一缓冲区缓存（Universal Buffer Cache，UBC）。源于文件系统的页由虚拟节点分页器管理，而VM缓存中的页则由默认分页器管理。

接下来的几节将概述供内核开发者使用的各种内存分配机制及其使用方法和限制。

### 6.2.1 底层分配机制

内核有多个内存分配例程族。每个主要的子系统，如Mach、BSD或I/O Kit，都有自己的函数族。VM子系统位于内核的Mach部分，实现内存分配的底层接口。这些接口进而形成高级内存分配机制，用于其他的子系统，如BSD和I/O Kit。

为了在内核的Mach部分执行操作，需要使用kmem_alloc*()函数族。这些是非常底层的函数，距离原始vm_page_alloc()函数只有很少几层。可用的函数有：

```
kern_return_t kmem_alloc(vm_map_t map, vm_offset_t* addrp, vm_size_t  size);
kern_return_t mem_alloc_aligned(vm_map_t map, vm_offset_t* addrp, vm_size_t size);
kern_return_t kmem_alloc_wired(vm_map_t map, vm_offset_t* addrp, vm_size_t size);
kern_return_t kmem_alloc_pageable(vm_map_t map, vm_offset_t* addrp, vm_size_t size);
kern_return_t kmem_alloc_contig(vm_map_t map, vm_offset_t* addrp, vm_size_t size,
                                vm_offset_t mask, int flags);
void kmem_free(vm_map_t map, vm_offset_t addr, vm_size_t size);
```

所有的函数都需要指定一个属于用户空间任务或kernel_map的VM映射。上面的所有函数都分配联动内存，这些内存无法换出，但kmem_alloc_pageable()除外。

### 6.2.2 Mach 内存域分配器

Mach内存域分配器是一种分配机制，可以分配名为内存域的固定长度的内存块。内存域通常表示一个广泛使用的内核数据结构，如文件描述符或任务描述符，但也可以指向更为通用的内存块。内存域分配器分配的数据结构示例包括：

- 文件描述符；
- BSD套接字；
- 任务（struct task）；
- 虚拟内存结构（VM映射、VM对象）。

作为内核开发者，如果你需要频繁、快速地分配和释放相同类型的数据对象，可以使用zinit()函数创建自己的内存域。为了创建一个新的内存域，你需要告诉分配器对象的大小、队列的最大长度及分配大小，这会指定内存域耗尽时需要添加的内存量。

### 6.2.3 kalloc 家族

kalloc家族为快速分配内存提供了一个略高级的接口。用过用户空间中malloc()接口的人应该非常熟悉这个API。实际上，内核也有一个malloc()函数，它由libkern内核库定义，同样使用kalloc()分配的内存。

```
void* kalloc(vm_size_t size);
void* kalloc_noblock(vm_size_t size);
void* kalloc_canblock(vm_size_t size, boolean_t canblock);
void* krealloc(void** addrp, vm_size_t old_size, vm_size_t new_size);
void kfree(void *data, vm_size_t size);
```

用于kalloc函数家族的内存可以通过上一节讨论的Mach内存域分配器获得。较大的内存分配由kmem_alloc()函数进行处理。因为内存有两个来源，所以kfree()函数需要知道最初分配的大小以确定其来源，释放正确位置的内存。I/O Kit和BSD层的底层内存函数基于kalloc家族提供的API进行构建。kalloc函数还为C++的内存分配操作符new和new[]提供内存。

除kalloc_noblock()外，kalloc函数及其变体可以阻塞（睡眠）以获得内存。kfree()函数也是如此。因此，如果你需要中断上下文中的内存或持有一个简单锁，那么必须使用kalloc_noblock()。

可用的内存域能查询到。以下是zprint命令修整后的输出结果，展示了kalloc函数使用的内存域。

| zone name | elem size | cur size | max size | cur #elts | max #elts | cur inuse | alloc size | alloc count |
|---|---|---|---|---|---|---|---|---|
| kalloc.16 | 16 | 660K | 922K | 42240 | 59049 | 30284 | 4K | 256 C |
| kalloc.32 | 32 | 3356K | 4920K | 107392 | 157464 | 73407 | 4K | 128 C |
| kalloc.64 | 64 | 4792K | 6561K | 76672 | 104976 | 75837 | 4K | 64 C |
| kalloc.128 | 128 | 2732K | 3888K | 21856 | 31104 | 20571 | 4K | 32 C |
| kalloc.256 | 256 | 4248K | 5184K | 16992 | 20736 | 15950 | 4K | 16 C |
| kalloc.512 | 512 | 968K | 1152K | 1936 | 2304 | 1870 | 4K | 8 C |
| kalloc.1024 | 1024 | 784K | 1024K | 784 | 1024 | 735 | 4K | 4 C |
| kalloc.2048 | 2048 | 3396K | 4608K | 1698 | 2304 | 1586 | 4K | 2 C |
| kalloc.4096 | 4096 | 2204K | 4096K | 551 | 1024 | 508 | 4K | 1 C |
| kalloc.8192 | 8192 | 3160K | 32768K | 395 | 4096 | 383 | 8K | 1 C |
| kalloc.large | 41375 | 5697K | 6743K | 141 | 166 | 141 | 40K | 1 C |

每个内存域的大小都是8 KB。分配小于8 KB的内存域将从最小的内存匹配域中返回一个元素。由于不可能部分分配一个元素，因此，如果需要5000字节的内存，那么实际要分配8192字节（每次分配都浪费3192字节）。大于8 KB的分配由适当的kmem_alloc()函数而不是内存域分配器处理，但这会记录在虚拟内存域kalloc.large中。

### 6.2.4　BSD 中的内存分配

BSD子系统中的内存分配由以下的函数和宏实现：

```
#define MALLOC(space, cast, size, type, flags)  (space) = (cast)_MALLOC(size, type, flags)
#define FREE(addr, type)_   FREE((void *)addr, type)
#define MALLOC_ZONE(space, cast, size, type, flags)
                  (space) = (cast)_MALLOC_ZONE(size, type, flags)
#define FREE_ZONE(addr, size, type) _FREE_ZONE((void *)addr, size, type)

void* _MALLOC(size_t size, int type, int flags);
void _FREE(void *addr, int type);
void* _MALLOC_ZONE(size_t size, int type, int flags);
void _FREE_ZONE(void *elem, size_t size, int type);
```

在底层，_MALLOC()函数会根据传递的标志使用kalloc()的一些变体进行内存分配。例如，如果需要执行非阻塞分配，则调用(M_NOWAIT) kalloc_noblock()。_MALLOC_ZONE()函数将直接调用

内存域分配器，而不是间接调用kalloc()。由于不使用通用的kalloc.X内存域，所以可以访问常用对象类型的内存域，如网络子系统使用的文件描述符、网络套接字或mbuf描述符。type参数用于确定要访问的内存域。尽管_MALLOC()也有一个type参数，但除了确认该值是否小于最大容许值外，可以将其忽略。定义的不同类型有一百多个。标志参数可以是下列参数之一：

```
#define M_WAITOK                0x0000
#define M_NOWAIT                0x0001
#define M_ZERO                  0x0004          /* 使用bzero进行分配 */
```

**提示** MALLOC函数族及内存域类型都定义在sys/malloc.h中。

如果指定了M_ZERO标志，则该标志将在内存返回给调用者之前使用bzero()函数将该内存覆写为0。否则，该内存仍然保留最后一个用户写入的内容，如果该内存从未使用过，那么它将包含随机垃圾。



### 6.2.5 I/O Kit 内存分配

I/O Kit为内存分配提供了完整的函数集。下列所有函数都会返回内核虚拟地址，这些地址可以直接访问：

```
void* IOMalloc(vm_size_t size);
void* IOMallocAligned(vm_size_t size, vm_size_t alignment);
void* IOMallocPageable(vm_size_t size, vm_size_t alignment);
```

对应的内存释放函数如下：

```
void IOFree(void* address, vm_size_t size);
void IOFreeAligned(vm_size_t size);
void IOFreePageable(void* address, vm_size_t size);
```

第一个函数IOMalloc()是kalloc()的一个包装器，具有相同的使用限制。要特别注意，IOMalloc()无法在原子上下文中使用，如主中断处理器，因为它会阻塞（睡眠）获取内存。如果需要对齐内存也不能使用IOMalloc()，因为它无法保证对齐。IOFree()是kfree()函数的一个包装器，也会阻塞（睡眠）。如果持有一个简单锁（如OSSpinLock）时调用IOMalloc()或IOFree()，还可能会造成系统死锁，因为如果两个函数中的任何一个睡眠，该线程都可能会被抢占。如果一个中断处理程序试图声明同样的锁，则将引起死锁。另外，IOMalloc()的内存用于小而快的分配，并不适合映射到用户空间。因为为IOMalloc()保留的内存来自大小固定的小型池，过度使用IOMalloc()将会造成池泄漏，并且在池被耗尽时会造成内核恐慌。

**警告** 使用IOFree()释放IOMallocAligned()分配的内存将会产生程序错误。必须使用与最初的分配函数相对应的释放函数。即使现在能正常工作（这很偶然），该机制也会在将来的更新中发生变化，引起崩溃。

IOMallocAligned()与IOMalloc()具有相同的使用限制，但不同于IOMalloc()，IOMallocAligned()会返回与特定值对齐的内存地址。例如，如果需要页对齐的内存，可以传入4096，获取的地址将对齐到页的开头。以下是请求对齐内存的部分原因。

- 硬件无法访问未对齐到特定边界的内存，或执行速度很慢。
- 向量计算中使用未对齐到特定字节边界的内存（SSE中通常是16字节），执行速度会非常慢。
- 内存需要映射到用户空间进程。由于映射只可用于整页，所以需要确保缓冲区始于页的边界。
- 需要一个与CPU缓存良好协作的数据结构。

IOMallocPageable()分配的内存可以进行分页，而其他变体创建的内存是固定的，无法换出。IOMalloc()和IOMallocAligned()的使用限制对IOMallocPageable()也有效。IOMallocPageable()获得的内存无法用于设备I/O（如DMA），也无法在只有内存固定之后才能阻塞/睡眠的代码路径中使用。

最后一个变体IOMallocContiguous()分配的内存是连续的。现在该变体已经废弃。苹果公司推荐使用IOBufferMemoryDescriptor()。

每个内存分配函数都有一个对应的内存释放函数。调用与内存分配函数相匹配的释放函数非常重要。每个变体都采用不同的底层机制分配内存，因此它们不可以互换。实际上，IOMalloc()可以从多个源获得内存。对于较大的分配（大于8 KB），可以采用kmem_alloc()；但较小的分配通过内存域分配器进行。

这恰恰是必须要向IOFree*()函数传入最初分配大小的原因，因为它可以用来确定内存的来源。

### 6.2.6　使用C++的new操作符分配内存

libkern库实现了一个基本的C++运行时，I/O Kit便是在该运行时的基础上构建的。C++中的内存分配通常采用分别面向单个对象和数组的new和new[]操作符实现。在libkern中，new操作符通过调用kalloc()获得内存在内部实现。因为kfree()需要最初分配的大小，所以libkern对传递给new操作符的大小进行了修改，以包括保存分配大小的小结构体空间，delete操作符调用free()，可以获取new返回的地址之前的4字节长度。

new或new[]分配的内存总被清零，它与用户空间中这些操作符的大部分实现不同。

---

**提示**　new、new[]、delete、delete[]操作符的实现可以在XNU源代码发布版的libkern/c++/OSRuntime.cpp中找到。

---

## 6.3　内存描述符

内存描述符由IOMemoryDescriptor类实现，是I/O Kit中处理内存的基础。class也是其他与内

存相关的重要函数的超类，本章后面会讨论。I/O Kit的许多部分都接受IOMemoryDescriptor作为参数。例如，USB家族使用该类描述用于USB读写请求的内存。

IOMemoryDescriptor描述内存缓冲区的属性或内存范围，但并不分配（或释放）描述的内存。它包含元数据，使某些操作可以在内存上执行。IOMemoryDescriptor可以描述虚拟内存和物理内存。该类具有多种功能，可用于各种用途。因此，可以通过多种方式构建IOMemoryDescriptor。一种常见的方式是使用withAddressRange()方法，如下所示。

```
static IOMemoryDescriptor* withAddressRange(mach_vm_address_t address,
                                            mach_vm_size_t length, IOOptionBits options,
                                            task_t task);
```

- 第一个参数address是描述符应该操作的内存缓冲区的起始地址。
- length参数是address指向的缓冲区字节数。task参数指定拥有该虚拟内存的任务。
- options参数指定I/O传输事件中用到的描述符的方向。它会影响prepare()和complete()的行为。以下是可能的标志：
  - kIODirectionNone
  - kIODirectionIn
  - kIODirectionOut
  - kIODirectionOutIn
  - kIODirectionInOut
- 最后一个参数是拥有该内存的任务。如果内核拥有该内存，则可以传递kernel_task，这是一个指向内核的task_t结构体的全局变量。

options标志表示I/O传输方向，可以用来确定是否需要刷新处理器缓存，以确保缓存的一致性。如果将该描述符用于I/O传输，必须首先调用其prepare()方法，执行以下操作：

- 如果底层内存被换出，那么换入内存；
- 钉住该内存，因此该内存无法换出，直至传输完成；
- 如有必要，配置设备地址传输映射。

---

**警告** prepare()必须与complete()成对调用。一定要注意，只有调用了prepare()才能调用complete()。

---

prepare()方法并非是线程安全的。多次调用prepare()则是线程安全的，但必须调用同样次数的complete()。调用描述符的release()方法并不能撤销prepare()的操作，也不能调用complete()，所以在调用release()之前，必须调用complete()。如果操作符映射到了地址空间，则它将通过release()自动解除映射。IOMemoryDescriptor还可以用来描述其他类型的内存，如物理地址。对于物理地址，prepare()和complete()方法不执行任何操作，但会成功返回。此外，物理内存描述符与任务并没有什么关联。可以使用静态成员方法withPhysicalAddress()为物理段构造一个IOMemoryDescriptor，如下所示。

```
static IOMemoryDescriptor* withPhysicalAddress(IOPhysicalAddress address,

IOByteCount withLength, IODirection withDirection);
```

### 6.3.1　IOBufferMemoryDescriptor

IOBufferMemoryDescriptor是IOMemoryDescriptor的一个子类，但与其超类不同，IOBufferMemoryDescriptor也可以分配内存。IOBufferMemoryDescriptor是当前将内存分配映射到用户空间，或从内核分配的缓冲区执行设备I/O的首选方式。但内部使用的分配方法取决于请求长度及构造时传递的选项。IOBufferMemoryDescriptor还是获得连续内存的首选方式。IOBufferMemoryDescriptor可以由静态工厂方法inTaskWithOptions()或inTaskWithPhysicalMask()进行分配，如下所示。

```
static IOBufferMemoryDescriptor* inTaskWithOptions(
    task_t                       inTask,
    IOOptionBits                 options,
    vm_size_t                    capacity,
    vm_offset_t                  alignment = 1);

static IOBufferMemoryDescriptor* inTaskWithPhysicalMask(
    task_t                       inTask,
    IOOptionBits                 options,
    mach_vm_size_t               capacity,
    mach_vm_address_t            physicalMask);
```

inTask参数指定内存应该映射到哪一个任务。如果是内核缓冲区，则应该设置为kernel_task。如果指定另一个任务标识符，则内存将在该任务的地址空间中分配和获得。除了用于IOMemoryDescriptor的标志和选项，还可以传递下列options，控制分配行为。

- kIOMemoryPhysicallyContiguous分配的内存在物理上具有连续性。
- kIOMemoryPageable分配的内存可以换出，所有的内存默认为不可分页。
- kIOMemoryPurgeable只应用于可分页的内存。如果指定该选项，则内存页可以丢弃，而不是换出。
- 如果内存将映射到内核和用户空间任务，则应该指定kIOMemoryKernelUserShared。kIOMemoryKernelUserShared可以保证内存页对齐。

构造IOBufferMemoryDescriptor的第二种方式是通过inTaskWithPhysicalMask()，我们可以指定一个位掩码以限制缓冲区的物理地址范围。为无法访问某一地址范围的设备分配内存以进行DMA时，这非常有用。例如，一些较老的设备可能无法访问32位以上的物理内存。

通常并不建议请求物理上连续的内存，特别是在系统已经启动之后，因为内存很快会产生很多碎片。这将使寻找连续的空闲缓冲区特别是较大的缓冲区变得非常困难。请求连续的内存还会导致一些内存被换出，以处理该请求，这会花费很长时间。硬件设备通常支持分散/聚集操作，该操作可以将多个较小的缓冲区链接为一个列表，传递给设备，设备然后读取该列表，计算出其数据在物理内存中的位置。因而，连续的内存常常没有必要。

就像IOMalloc()函数族一样，IOBufferMemoryDescriptor可以睡眠，因此不应该从中断上下文或持有简单锁时对其进行调用。实际上，IOBufferMemoryDescriptor在内部使用IOMalloc()和IOMallocAligned()分配内存。

### 6.3.2 其他的内存描述符

IOMemoryDescriptor还有许多其他的相关子类，如下所示。

- IODeviceMemory：用来描述设备映射的内存范围。
- IOMultiMemoryDescriptor：可以用来表示由较小的IOMemoryDescriptor对象组成的较大连续缓冲区。

## 6.4 映射内存



映射内存是让一个任务的内存范围可用于另一个任务的功能。在最底层，映射由Mach VM子系统处理，如第2章所述。内存映射提供了一个快捷方式，使任务无须复制内存便可以共享资源，因为映射使相同的内存可以在任务之间使用。可写映射能够共享，直至有修改发生，这种情况下，写时复制（COW）优化仅复制修改的内存。内存映射可以在多个任务之间、从内核到用户空间任务或从用户空间任务到内核之间以各种不同的方式进行。

### 6.4.1 用户空间任务到内核空间的内存映射

映射用户空间任务的内存是一个常见的驱动程序操作。我们看一下音频设备驱动程序的例子，应用程序想向我们发送一个数据缓冲区，其中包含在硬件设备上播放的音频样本。为此，用户任务（即音频播放器）会向我们传递一个内存指针，描述该缓冲区在内存中的位置。在用户空间中，简单调用memcpy()函数便可以实现内存复制。

而在内核中并不是这么简单。用户任务传递的地址对内核来说毫无意义，因为该地址仅在任务的私有地址空间中有效。为了访问内核中的内存，我们需要为内核自己地址空间中的底层物理内存缓冲区创建一个映射。在底层，该处理通过操作内核的VM映射实现。虽然可以使用Mach的底层接口实现该处理，但最常见的做法是借助I/O Kit IOMemoryDescriptor和IOMemoryMap类。代码清单6-2是虚构音频驱动程序的一部分，通过将内存缓冲区映射到内核的地址空间，从用户空间的音频播放器复制内存。

**代码清单6-2 将一个用户空间缓冲区映射到内核**

```
void copyBufferFromUserTask(task_t userTask, void* userBuffer,
                            uint32_t userBufferSize, void* dstBuffer)
{
    uint32_t                bytesWritten = 0;
    bool                    wasPrepared = false;
    IOMemoryDescriptor*     memoryDescriptor = NULL;
    IOMemoryMap*            memoryMap = NULL;
```

```
    memoryDescriptor = IOMemoryDescriptor::withAddressRange
                        (userBuffer, userBufferSize,
                        kIODirectionOut, userTask);
    if (memoryDescriptor == NULL)
        goto bail;

    if (memoryDescriptor->prepare() != kIOReturnSuccess)
        goto bail;
    wasPrepared = true;

    memoryMap = memoryDescriptor->createMappingInTask
                    (kernel_task, 0, kIOMapAnywhere | kIOMapReadOnly);
    if (memoryMap == NULL)
        goto bail;

    void* srcBufferVirtualAddress = (void*)memoryMap->getVirtualAddress();

    if (srcBufferVirtualAddress != NULL)
        bcopy(srcBufferVirtualAddress, dstBuffer, userBufferSize);

    memoryMap->release();//这将取消内存映射
    memoryMap = NULL;
bail:
    if (memoryDescriptor)
    {
        if (wasPrepared)
            memoryDescriptor->complete();
        memoryDescriptor->release();
        memoryDescriptor = NULL;
    }
}
```

为了映射内存，首先为用户空间缓冲区创建一个IOMemoryDescriptor。该IOMemoryDescriptor提供一个接口，创建内存映射，还可以在我们从缓冲区复制时钉住内存。如果执行复制时音频播放器崩溃或用户退出该应用程序，这可防止内存被换出到二级存储器或消失。

---

**注意**　读者也许已经注意到，上述方法中使用了goto，语言纯化论者认为这是一个不良用法。但goto经常用在内核代码中，是集中清理错误的便捷方式，代替内核中不可以使用的异常机制。

---

实际的映射在调用createMappingInTask()方法时进行。

```
IOMemoryMap* createMappingInTask(
    task_t                  intoTask,
    mach_vm_address_t       atAddress,
    IOOptionBits            options,
    mach_vm_size_t          offset = 0,
    mach_vm_size_t          length = 0 );
```

**提示** 可以使用IOMemoryDescriptor::map()方法作为一个快捷方式，去创建一个到内核地址空间的标准映射。还要注意，map()的重载变体已经废弃，Mac OS X 10.5中引入了替代方法createMappingInTask()。

- 第一个参数intoTask是我们要创建的映射所在的任务。对我们来说，这是kernel_task，尽管它可能会提供另一个任务的任务结构，使一个任务可以使用另一个任务的内存。
- 第二个参数atAddress也很有趣。它指定intoTask地址空间中一个可选的目标地址。这使目标任务可以将映射置于固定的地址上。在我们的例子中，我们并没有真正关注地址空间中映射建立的位置，只是希望一个地址可以对它进行访问，所以传入了0而不是一个固定地址，并将options设置为kIOMapAnywhere。
- 第三个参数options为6.3节介绍的标志，用来控制映射的执行方式，例如，只读或读/写。该选项还可以用来控制内存针对CPU缓存的行为方式。可以设置为如下选项。
  - kIOMapDefaultCache，指定映射的缓存策略。它将禁用I/O内存的缓存；否则，请使用kIOMapCopybackCache。
  - kIOMapInhibitCache，禁用该映射的缓存。
  - kIOMapWriteThruCache，使用连续写入的缓存。
  - kIOMapCopybackCache，使用拷回（copy-back）缓存。
  - kIOMapReadOnly，指定映射为只读。
  - kIOMapReference，在映射一个已经存在的映射时使用，如果内存没有预先映射，则将失败。
  - kIOMapUnique，确保内存之前不存在映射。
- 最后两个参数用来指定可选的缓冲区偏移量及长度，仅在映射缓冲区的一部分时使用。但请注意，映射是虚拟内存系统的一个概念，对页进行操作。你只能以页长度（4096字节）为单位映射内存。舍入操作在内部进行，使用户产生处理字节边界的错觉。



### 6.4.2 IOMemoryMap 类

代码清单6-2中的createMappingInTask()方法将返回IOMemoryMap的一个实例，表示映射。前面的例子调用了IOMemoryMap::getVirtualAddress()方法，该方法返回一个IOVirtualAddress类型的值。IOVirtualAddress具体的原语数据类型取决于体系结构，但64位内核使用的是64位无符号整数（uin64_t），而不是指针类型。

不再需要映射时，便可以释放IOMemoryMap对象，该对象负责处理映射解除。你可能很奇怪，为什么不调用IOMemoryMap::unmap()函数来释放映射。创建一个映射时，另一个线程或同个线程很可能会再次映射该缓冲区。当然，映射只创建一次，但多次执行映射会增加内部引用计数。然而，调用unmap()并不是简单地减少引用计数并在计数为0时移除映射，而是销毁映射，无论它被引用了多少次。这可能会导致内核访问一个无效的地址；因此，使用unmap()时

需要谨慎。如有必要，只须调用release()便可减少或移除映射。其他相关的IOMemoryMap方法如表6-2所示。

请注意，代码清单6-2对之前的例子进行了一些小修改，创建一个可写映射，也能将内存复制到映射的缓冲区中。

表6-2　IOMemoryMap成员函数概览

| 函　数 | 描　述 |
| --- | --- |
| getAddressTask() | 获取映射的任务 |
| getMemoryDescriptor() | 返回创建映射所采用的内存描述符 |
| getPhysicalAddress() | 获取映射中第一个字节的物理地址 |
| getPhysicalSegment() | 创建并偏移映射，并返回支持该虚拟内存物理段的地址及长度。如果该段由连续的内存支持，则长度将是整个映射 |
| redirect() | 使映射的内存可以与新的物理内存切换。这通过提供一个新的IOMemoryDescriptor实现，因此，映射将更新为指向传递的描述符拥有的新物理内存。redirect()函数仅在内存映射由kIOMapUnique创建的情况下才会执行成功。如果传递的是NULL，而不是IOMemoryDescriptor，用户任务对映射内存范围的任何访问都将阻塞，直至提供一个有效的内存描述符 |
| … | IOMemoryMap类的完整定义可以在IOMemoryDescriptor.h中找到 |

**注意**　无须将内存映射到内核，除非内核需要主动修改该内存。如果DMA从用户空间缓冲区执行，并且该缓冲区中的数据无须由内核修改，则不必将其映射到内核的地址空间，可以直接将该缓冲区传输给硬件设备。关于DMA的更多信息，请参见第9章。

### 6.4.3　内核到用户空间任务的内存映射

上一节讲述了如何在用户空间中分配内存，并将该内存映射到内核的地址空间，以便内核访问它。虽然内核可以读写映射，但用户空间任务有时需要将内核内存映射到它的地址空间。应该注意的是，苹果公司因安全性和稳定性的原因不建议执行该操作，所以应尽可能避免。这样做的一个原因是，需要将设备内存（例如，PCI设备的内存）映射到用户空间，以便它可以访问设备的寄存器。

在I/O Kit中，这种形式的内存映射一般通过IOUserClient类实现。可用内存映射应该通过clientMemoryForType()方法返回。该实现的一个通用示例如代码清单6-3所示。

**代码清单6-3　通过IOUserClient将内核内存映射到用户空间**

```
#define kTestUserClientDriverBuffer             0
IOReturn com_osxkernel_TestUserClient::
clientMemoryForType(UInt32 type, UInt32 *flags, IOMemoryDescriptor **memory)
```

```
{
    IOReturn ret = kIOReturnUnsupported;
    switch (type)
    {
        case kTestUserClientDriverBuffer:
            //返回一个指向IOMemoryDescriptor的指针
            //对于硬件设备，则返回一个IODeviceMemory指针
            //IODeviceMemory是IOMemoryDescriptor的一个子类
            *memory = driver->getBufferMemoryDescriptor();
            *memory->retain();
            ret = kIOReturnSuccess;
            break;
        default:
            break;
    }
    return ret;
}
```

请注意，在返回IOMemoryDescriptor之前，需要对它调用retain()，因为在用户客户端关闭时它会被释放，而我们并不希望释放该操作符，因为它是驱动程序拥有的一个共享资源。在该示例中，为了演示，我们调用了一个假设的驱动程序，它有一个getBufferMemoryDescriptor()方法，该方法为内核分配的缓冲区返回一个IOMemoryDescriptor（或者是映射到内核地址空间的设备内存）。这里的type参数是一个整数，可以是任意值；重要的是，访问该内存的用户空间程序需要知道该值，以便引用正确的内存映射。

在用户空间代码中，可以执行如下操作，从IOUserClient映射内存。

```
void* addressOfMappedBuffer = NULL;
int sizeOfMappedBuffer;
IOConnectMapMemory(openDeviceHandleHere,
                   kTestUserClientDriverBuffer,
                   mach_task_self(),
                   (vm_address_t *) &addressOfMappedBuffer,
                   &sizeOfMappedBuffer,
                   kIOMapAnywhere);
```

你可能已经注意到，这与在内核中创建映射非常相似。这里的kIOMapAnywhere表示我们并不关心映射在地址空间中的创建位置；addressOfMappedBuffer参数在调用成功时保存映射的地址，可以用来访问映射的内存。如果没有指定kIOMapAnywhere，addressOfMappedBuffer参数则用于指定映射的首选地址。倒数第二个参数告诉我们映射的长度。映射的最小单位是页；因此，如果映射的缓冲区小于4096，客户就能看到包含该缓冲区的整个页的内存，这会成为安全隐患。

### 6.4.4 将内存映射到指定的用户空间任务

前面的例子中，任何任务都可以映射内存，并且驱动程序代码无须知道内存将映射到哪个任务。但如果你知道内存应该映射到的指定任务，就可以使用代码清单6-2的方案。区别在于，用户空间任务标识符传递给IOMemoryDescriptor::createMappingInTask()而不是kernel_task。

苹果公司建议不要映射使用函数获得的内存，如IOMalloc()和IOMallocAligned()（尽管可能会用到后者），因为它们来自内存域分配器，这用于私有、临时分配，并不用于共享分配。建议使用IOBufferMemoryDescriptor进行内存映射，它是IOMemoryDescriptor的一个子类，也可以分配内存，如下所示。

```
IOBufferMemoryDescriptor* memoryDescriptor = NULL;
memoryDescriptor = IOBufferMemoryDescriptor::withOptions(
    kIODirectionOutIn | kIOMemoryKernelUserShared, sizeInBytes, 4096);
```

需要注意一个有趣的参数kIOMemoryKernelUserShared，它向分配器表明我们希望与用户任务共享内存。我们传递4096（页长度）来获取页对齐的内存，因为内存映射只能以页为单位进行。

### 6.4.5　物理地址映射

虚拟内存地址只对CPU可用，对需要物理地址的硬件设备毫无意义。为了与CPU外的硬件设备通信，需要将内核或用户空间任务的虚拟地址转换为物理地址，设备可以用来访问RAM的信息。这一工作并不是那么简单，因为虚拟内存通常是碎片。例如，我们希望给硬件设备发送一个128 KB的虚拟内存缓冲区。在最坏的情况下，该缓冲区可能由32个4 KB的页组成，并且分散在系统内存的各处。因此，并不能只传输该缓冲区第一个字节的地址，并告知设备该缓冲区长度是128 KB；需要计算出该缓冲区由多少个碎片组成，并发送一个包含地址及长度的列表/数组。通常称为分散/聚集表或列表。IOMemoryDescriptor及其派生类提供了下列两种方法进行物理地址转换。

- getPhysicalAddress()：将第一个字节的地址转换为其物理地址。这在知道缓冲区连续的情况下有用。
- getPhysicalSegment()：将指定的偏移量地址转换入缓冲区，并从该偏移量返回物理段的长度。在连续的缓冲区中，这总是缓冲区的长度减去偏移量。

---

**警告**　这个方法如果使用不当会引起内核恐慌。正确的使用方法请参见下面的讨论。

---

请注意，根据你使用的是64位内核还是32位内核，getPhysicalSegment()有两个版本，如下所示：

```
#ifdef __LP64__
    virtual addr64_t getPhysicalSegment( IOByteCount   offset,
                                         IOByteCount * length,
                                         IOOptionBits  options = 0 ) = 0;
#else /* !__LP64__ */
    virtual addr64_t getPhysicalSegment( IOByteCount   offset,
                                         IOByteCount * length,
                                         IOOptionBits  options );
#endif /* !__LP64__ */
```

对于32位版本（!__LP64__），options参数必须指定为kIOMemoryMapperNone，否则该方法将会

因大于4 GB的标记地址而造成内核恐慌。更灵活、更安全、更容易的内存转换方案是使用IODMACommand类，它与IOMemoryDescriptor协同工作。第9章将更详细地讲述IODMACommand及这方面的内容。

## 6.5 小结

本章讲述了如下内容。

- 内核使用的内存地址类型。内核通常为其自身的线程及用户空间任务的线程处理虚拟地址。物理内存地址可以在CPU与内存及硬件设备之间使用。
- 32位、64位内存寻址及模式的意义。
- 内存如何在不同的内核子系统、Mach、BSD及I/O Kit中执行分配。在I/O Kit中，首选机制是使用IOMalloc*()函数或IOBufferMemoryDescriptor。
- I/O Kit的许多部分如何使用IOMemoryDescriptor及相关子类管理和描述内存缓冲区。IOBufferMemoryDescriptor子类除了提供内存描述符，还可以分配各种形式的内存，如对齐的或物理上连续的内存。
- 如何使用IOMemoryMap类管理内存映射，并使内核将一个用户空间缓冲区映射到其虚拟地址空间，以便内核可以操作该内存。
- IOUserClient类如何提供一个有用的方法，即clientMemoryForType()，处理内核缓冲区到用户空间的映射细节。
- IOMemoryDescriptor如何提供getPhysicalSegment()等方法，将虚拟内存地址映射到物理地址。

# 第7章 同步和线程

驱动程序的作用是将硬件设备的功能提供给操作系统和用户应用程序，这部分内容贯穿本书始终。这意味着驱动程序内的代码可以在任意时刻由任意数目的运行应用程序调用，这取决于应用程序何时请求硬件设备的服务。处理这些请求时，驱动程序运行在进行该控制调用的应用程序线程上下文中。除了这些请求，硬件本身也可以请求服务，并且能在驱动程序必须响应的任何时刻生成中断。驱动程序开发者的最终目标是，驱动程序代码运行在复杂的多线程环境中，即使驱动程序并没有创建它自己的任何额外线程。

驱动程序执行的计算机硬件可能会有多个CPU核心。因此，除了设备中断或另一个应用程序中的线程请求抢占驱动程序代码以外，驱动程序还可以同时运行在多个核心上。这甚至适用于驱动程序的中断服务例程，该例程可以在另一个CPU核心上并行运行驱动程序中的非中断代码。

与多线程应用程序代码一样，重要的是，驱动程序提供同步访问机制，访问其内部结构体及可能由多个线程读/写的任意数据。驱动程序为试图访问其硬件的多个线程提供仲裁的方式取决于设备的类型。某些硬件在某一时刻只能由一个客户端访问。例如，串行端口设备在某一时刻只允许一个用户进程进行独占访问；驱动程序将确保另一个试图打开该串行端口的进程被拒绝。另一方面，一个磁盘设备可以接收多个进程的请求，但由于硬件本身在某一时刻只可以处理一个请求，所以驱动程序将按照串行方式，负责对新来的请求进行排队，并将其发送给磁盘设备。

I/O Kit提供了几种不同的机制，如果可以确保驱动程序内部结构在多线程环境中仍然有效，那么可以使用这些机制实现其硬件的仲裁访问。本章假定你已基本了解了代码同步，并且此前编写过多线程应用程序代码。

## 7.1 同步原语

当在两个或两个以上线程上运行的代码试图访问公共资源或结构时，便会发生同步问题。一个常见的I/O Kit驱动程序同步问题在驱动程序需要访问其实例变量时出现，因为这些实例变量在驱动程序执行的所有线程之间共享。看一个实际的I/O Kit示例，即`OSObject`基类引用计数的实现。

OSObject类是I/O Kit中所有对象的基类，其中一个作用是维护每个对象实例的引用计数，并在其引用计数减少至0时释放该对象。OSObject的一个简化版实现如代码清单7-1所示，其中并不包含实际实现中的同步处理。

**代码清单7-1 对象引用计数的一个可能实现**

```
void    Object::retain ()
{
        retainCount += 1;                    //定义为int类型的实例变量
}

void    Object::release ()
{
        retainCount -= 1;
        if (retainCount == 0)
                this->free();
}
```

尽管上面的代码看起来正确，并且在单个线程调用retain()和release()时也完全能正常运行，但该代码并非线程安全的，在多个线程对同一个对象同时调用retain()和release()时可能就会失败。为了理解该问题，有必要检查前面代码的编译器输出。这种情况下，当该实现在Debug编译模式下面向64位英特尔体系结构进行编译时，会生成以下汇编指令。其中，retain()包含以下指令序列：

```
        mov     eax, retainCount          ; Load retainCount into CPU register EAX
        add     eax, 0x1                  ; Increment value in EAX
        mov     retainCount, eax          ; Write value in EAX to retainCount
```

release()代码包含以下指令序列：

```
        mov     eax, retainCount          ; Load retainCount into CPU register EAX
        sub     eax, 0x1                  ; Decrement value in EAX
        mov     retainCount, eax          ; Write value in EAX to retainCount
        mov     eax, retainCount          ; Load retainCount into CPU register EAX
        cmp     eax, 0x0                  ; Determine whether the value of EAX is 0
        jne     skipFree                  ; If EAX is not zero, jump over the next instruction
        call    free()                    ; Otherwise, call the free() method
skipFree:
        ...
```

在多线程环境中，造成该问题的原因是，增加和减少实例变量retainCount的C语言代码会编译为3个CPU指令：将实例变量retainCount持有的值从内存加载到CPU寄存器，增加或减少CPU寄存器的值，将结果写回至内存。我们来看一下，两个线程同时对一个对象调用retain()时会发生什么情况。为了简单起见，我们假定该代码在单个CPU核心机器上运行，并且操作系统的调度器在初始mov指令执行后的地方抢占第一个线程。

在这种场景下，线程1将retainCount的值从内存读取到EAX寄存器。此时，操作系统的调度器抢占线程1，并切换至线程2（在保存线程1的CPU寄存器状态之后）。目前线程2在运行，并同线程1一样，它将同样的retainCount值读取到EAX寄存器。然后将该值加1，并将增加后的值写

回内存。操作系统调度器然后抢占线程2，并在恢复线程1保存的CPU寄存器状态之后，将执行切回至线程1。这时线程1从离开的位置继续执行，将retainCount的*初始值*加1，并将结果写回内存。这样，retainCount的值仅会增加1，即使retain()方法被调用了两次。

| 线　程　1 | 线　程　2 |
| --- | --- |
| mov eax, retainCount | |
| | mov eax, retainCount |
| | add eax, 0x1 |
| | mov retainCount, eax |
| add eax, 0x1 | |
| mov retainCount, eax | |

请注意，此问题只在特定情况下出现：retainCount实例变量必须被两个线程修改，且一个线程采用上述方式抢占另一个线程，或者两个线程必须同时运行在两个CPU核心上。类似这样的问题，执行代码的结果取决于代码运行的时机和顺序，称为竞态条件。竞态条件会导致非常难以调试的问题，因为这种问题依赖于时机，因此不一定在每次运行代码时都会发生。实际上，该代码在测试时可能运行完全正常，仅在用户报告时驱动程序的问题才变得明显。

除了难以重现之外，竞态条件在引起问题时也很难诊断。以前面描述的竞态条件为例，其中，对象的持有计数增加1，即使调用两次retain()也是如此。这不会立即引起任何问题，驱动程序就像没有任何错误一样继续工作，直至很久以后对象释放时。由于该对象被持有两次，所以调用代码应该释放该对象两次。但由于持有计数的值比正常值少1，该对象将在驱动程序仍然持有它的一个引用时销毁。这意味着，稍后驱动程序将试图访问它认为持有其引用的对象。但该对象已经销毁，驱动程序将因访问无效的内存而崩溃。请注意，包含崩溃代码的函数可能与包含竞态条件的函数完全不同。因而，将程序错误原因追溯至retain()和release()将涉及大量调查工作。

## 7.2　原子操作

由于前面示例中的竞态条件，是编译器在生成一个包含3个指令的序列时，增加和减少实例变量retainCount引起，所以一个解决方案是将“加载-修改-存储”序列的编译器输出替换为执行等同操作的一个指令。采用这种方式，执行被另一个线程抢占时，不会出现操作中断。但实际上，并不总是可以将一个操作替换为单个指令。但可以使用一个指令序列，其行为看起来就像单个指令。这称为“原子”操作，因为操作结果就像该指令序列作为单独、不可分割的组执行一样。

原子操作的实现需要CPU的支持。例如，麦金塔计算机使用的英特尔CPU提供了一个指令，可以在内存中原子地将一个值添加到另一个值。但这并不足以使多处理器环境中的操作成为原子操作。因此，英特尔指令集提供了一个LOCK前缀，防止系统中任何其他的CPU在指令执行时访问内存。由于原子操作的实现依赖特定CPU体系结构的支持，所以使用ARM指令集的iOS设备需要每个原子操作的不同实现。

为了使访问驱动程序代码中的原子操作变得更容易，I/O Kit包括了许多函数，提供各种基本操作的原子实现，如整数加法、增加/减少值及按位操作。这些函数如表7-1所示，定义在头文件<libkern/OSAtomic.h>中。

表7-1 libkern框架提供的原子操作

| 函 数 | 描 述 |
| --- | --- |
| OSIncrementAtomic(address);<br>OSIncrementAtomic8/16/64(address); | 将指定地址中8位、16位、32位或64位的有符号值加1。返回加1之前的初始值 |
| OSDecrementAtomic(address);<br>OSDecrementAtomic8/16/64(address); | 将指定地址中8位、16位、32位或64位的有符号值减1。返回减1之前的初始值 |
| OSAddAtomic(amount, address);<br>OSAddAtomic8/16/64(amount, address); | 将指定地址中8位、16位、32位或64位的有符号值加amout值。返回增加之前的初始值 |
| OSBitAndAtomic(mask, address);<br>OSBitAndAtomic8/16(mask, address); | 执行mask值与指定地址中8位、16位或32位无符号值的按位与操作。返回按位操作之前的初始值 |
| OSBitOrAtomic(mask, address);<br>OSBitOrAtomic8/16(mask, address); | 执行mask值与指定地址中8位、16位或32位无符号值的按位或操作。返回按位操作之前的初始值 |
| OSBitXorAtomic(mask, address);<br>OSBitXorAtomic8/16(mask, address); | 执行mask值与指定地址中8位、16位或32位无符号值的按位异或操作。返回按位操作之前的初始值 |
| OSCompareAndSwap(oldValue, newValue, address);<br>OSCompareAndSwapPtr(oldValue, newValue, address);<br>OSCompareAndSwap64(oldValue, newValue, address); | 如果指定地址的32位或64位整数值等于oldValue，则将newValue写入到该地址中。否则，不修改该地址中存储的值。该函数返回一个布尔值，表示newValue是否已被写入 |
| OSTestAndSet(bit, address); | 设置指定地址字节中的一位。返回一个布尔值，说明newValue是否已经设置该位 |
| OSTestAndClear(bit, address); | 清除指定地址字节中的一位。返回一个布尔值，说明newValue是否已经清除该位 |

使用这些函数，现在我们提供的retain()和release()实现便可以避免前面例子中出现的竞态条件，如代码清单7-2所示，其中假定实例变量retainCount为一个32位整数。

**代码清单7-2 多线程环境中对象引用计数的实现**

```
void    Object::retain ()
{
        OSIncrementAtomic(&retainCount);
}

void    Object::release ()
{
        uint32_t                originalValue;

        originalValue = OSDecrementAtomic(&retainCount);
        if (originalValue == 1)
                this->free();
}
```

如果回过头去查看代码清单7-1的原始实现以及release()方法的相应编译器输出，便会发现代码中实际包含了两个竞态条件。retainCount的值减至0时，便会对free()进行条件调用。但由于测试retainCount的值是否为0时，编译后的代码会从内存中重新加载retainCount值，所以对release()的两次调用都可能会读取0值，并且也会对该对象调用两次free()方法，这很可能会导致崩溃。为了说明这一情况如何发生，假定一个线程已经执行release()，将retainCount从2减至1，并将减少后的值写回内存。另外，假定在从内存重新加载retainCount值并测试该值是否为0之前，该线程已被抢占。另一个线程现在有机会运行，如果将要执行release()，retainCount会从1减至0，该对象就会被销毁。执行返回至初始线程时，它将重新加载retainCount的值，发现其值为0便会第二次销毁该对象。

在代码清单7-2中，使用OSDecrementAtomic()返回的值来确定最后的引用计数何时被释放，从而避免了这种竞态条件。OSDecrementAtomic()函数在其参数减少之前，将返回其初始值。我们知道，如果初始值是1，而现在retainCount值已经减少至0，该对象便能安全地销毁。

值得专门提及的一组原子操作是CAS（compare-and-swap，比较和交换）函数族。CAS操作是将一个值写入到内存地址，但重要的一点是，仅在被覆写的值等于调用者提供的某个预期值时才会执行写入处理。该操作的结果是一个布尔值，表示写操作是否成功执行。重要的是，为了同步，整个操作的执行都是原子操作。

CAS函数可以用来构建更复杂的原子操作。例如，假如我们希望实现一个函数，执行按位与，接着是按位或操作，同时整体操作都是原子操作。很显然，我们不能简单地调用OSBitAndAtomic()和OSBitOrAtomic()，因为没有任何处理可以防止这两个函数间的执行被抢占。通过OSCompareAndSwap()函数，可以实现一个执行按位与、按位或原子操作的函数，如下所示：

```
uint32_t        AtomicBitAndOr (uint32_t andMask, uint32_t orMask, volatile uint32_t* address)
{
        uint32_t        oldValue;
        uint32_t        newValue;

        do {
                oldValue = *address;
                newValue = oldValue & andMask;
                newValue = newValue | orMask;
        } while (OSCompareAndSwap(oldValue, newValue, address) == false);

        return oldValue;
}
```

你会发现，在执行两个按位操作时我们根本没有同步。该实现有效，并且是原子操作，原因是它使用了OSCompareAndSwap()函数，确保address的值仍是初始值，并没有被改变，程序根据该初始值计算写入的新值。如果函数执行时另一个线程已经修改了address的值，那么OSCompareAndSwap()函数将返回false，不执行写入处理。因而，必须返回到循环开始，在重新读取address的值后重复执行整个按位操作。这一次尝试，希望我们能有好运气，操作可以在控制

下执行，没有另一个线程来修改address的值，尽管我们会在将结果成功地写入到内存之前继续重试。

> **注意** 所有的原子操作，如OSAddAtomic()、OSIncrementAtomic()和OSBitOrAtomic()，都可以仅使用OSCompareAndSwap()来实现。实际上，libkern库提供的一些原子函数就是这样实现的，包括所有的按位原子操作及每个操作的8位、16位变体，它们对包含修改值的整个32位字执行CAS操作。

## 7.3 锁

使用原子操作同步访问一个变量是个很好的解决方案。但通常需要同步更复杂的代码段，如依赖多个变量值或处理硬件函数的算法。为了从代码的这些复杂区域移除可能发生的竞态条件，要依赖于互斥，也就是说，任何其他线程，如果想要执行可以影响操作结果的代码，在操作完成之前都会受到阻塞。获得独占访问的行为称为“获得一个锁”。

锁的基本思想是，访问共享资源的任何代码（如驱动程序的实例变量）在执行代码之前首先要获得一个锁。在完成对共享资源的访问时释放该锁。关于锁的重要一点是，锁在一个时刻只可以被一个客户端持有；希望访问同个共享资源的任何其他线程，在试图获得该锁时都将阻塞，直至该锁被释放。获得一个锁可以防止依赖于同个锁的任何其他线程执行，因此，一个好的做法是根据需要尽可能短缩短持有锁的时间。

I/O Kit提供了几种不同类型的锁机制，每种机制适用于不同的情形。I/O Kit提供的锁包括：

- IOSimpleLock，实现一个自旋锁；
- IOLock，实现一个普通的互斥；
- IORecursiveLock，实现一个互斥，该互斥可以多次从持有锁的线程安全地获得；
- IORWLock，实现一个读/写锁，该锁可以在需要读取共享资源的多个线程之间共享，但为希望写入共享资源的线程提供独占访问。

### 7.3.1 自旋锁

自旋锁是锁的最基本实现，只用原子操作就可以实现（这就是自旋锁在I/O Kit实现中称为IOSimpleLock的原因）。一个自旋锁可能只包含一个布尔标志，表示该锁当前是否被某个线程持有。如果一个线程希望获得锁，那么该实现将判断锁是否被持有，如果没有，那么执行该锁标志的一个原子集。如果该锁被持有，那么该函数将重复尝试获得该锁，直至该锁可用。自旋锁的一个示例实现如下列代码所示。该实现使用一个无符号32位整数表示锁状态，0表示锁可用，1表示锁被持有。

```
typedef    uint32_t    MySpinLock;

void    MyAcquireSpinLock (MySpinLock* lock)
{
        //如果锁的值是0，则将其值设置为1
        //不断重试，直至锁的值被成功设置
        while (OSCompareAndSwap(0, 1, lock) == false)
                ;
}
```

持有自旋锁的线程必须注意，不要再次获得当前持有的锁。否则线程将试图获得该锁并自旋，因为该锁当前不可用。但在这种情况下，线程将无限自旋，因为可以释放该锁的唯一线程正在阻塞等待该锁。这种情形称为死锁。

I/O Kit使用的IOSimpleLock的实际实现比给出的例子要高级一些，因为它在锁被持有时禁用了运行线程的抢占。这意味着，一个简单锁被持有时，持有该锁的线程将不会被撤销它运行的CPU，直至锁被释放。因此，IOSimpleLock应该只可以获得很短的时间（如驱动程序的实例变量更新时），并且从不执行阻塞运行线程的操作，如分配内存或获得互斥，因为这会导致死锁。

尽管自旋锁看起来是一种低效的锁机制，因为线程在没有立即获得锁时会花费CPU周期进行自旋，但实际上自旋锁比其他的锁机制更高效，假如锁仅被持有很短的一段时间。在单CPU机器上，IOSimpleLock从不自旋，因为线程抢占被禁用，并不存在锁竞争的可能（实际上，同步通过禁用线程抢占，防止任何其他可能会获得锁的线程执行而实现）。在多处理器系统上，获得IOSimpleLock时禁用抢占不能防止另一个CPU上的运行线程试图访问该锁（实际上，线程抢占只对获得锁的CPU核心禁用）。但假如自旋锁只被持有很短的一段时间，线程等待锁释放时花费的自旋时间，通常要比使用互斥而非自旋锁阻塞线程所花的时间少得多。

不同于互斥，IOSimpleLock从不暂停运行的线程。相反，IOSimpleLock将会自旋，直至锁可用。这使IOSimpleLock非常完美，可以提供运行在主中断处理程序中的代码和非中断代码之间的同步。实际上，该功能在I/O Kit驱动程序中很少用到，因为大多数驱动程序根本不需要直接处理中断，即使处理，大部分情况下也是交由二级中断处理程序处理。本章后面将讨论I/O Kit提供的适用于二级中断处理程序的其他锁机制。

为了使运行在主中断处理程序中的代码同步，需要确保在非中断时获得IOSimpleLock的代码绝不会被中断处理程序中运行的、试图获得该锁的代码抢占，因为这会导致死锁。为了解决该问题，I/O Kit提供了一个函数，在获得自旋锁之前，禁止中断运行的CPU，还提供了一个释放自旋锁并重新启用中断的对应函数。禁用线程抢占可以保证持有IOSimpleLock的线程不会被同一个CPU上的另一个线程抢占，同样，禁用中断可以保证持有IOSimpleLock的线程不会被同一个CPU上的中断处理程序抢占。中断可能在系统的另一个CPU上触发，并试图获得另一个CPU核心线程上持有的IOSimpleLock（导致中断处理程序自旋），但因为该线程在另一个CPU上运行，所以它可以继续执行，随后释放IOSimpleLock。

表7-2综述了I/O Kit提供的IOSimpleLock函数。

表7-2 I/O Kit提供的自旋锁同步函数

| 函 数 | 描 述 |
| --- | --- |
| IOSimpleLock* IOSimpleLockAlloc(void); | 分配一个新的IOSimpleLock，并返回一个指向初始化对象的指针，失败时返回NULL |
| void IOSimpleLockFree(IOSimpleLock* lock); | 释放IOSimpleLockAlloc()分配的IOSimpleLock对象 |
| void IOSimpleLockLock(IOSimpleLock* lock); | 获得一个简单锁，如果该锁当前被另一个客户端持有，则进行自旋。函数返回时，禁用对当前CPU的抢占 |
| boolean_t IOSimpleLockTryLock(IOSimpleLock* lock); | 获得一个简单锁，但仅在该锁未被持有时获取。该函数返回一个布尔值，表示该函数是否可以获得该锁 |
| void IOSimpleLockUnlock(IOSimpleLock* lock); | 释放由IOSimpleLockLock()或成功调用IOSimpleLock TryLock()而获得的简单锁。重新启用对当前CPU的抢占 |
| IOInterruptState IOSimpleLockLockDisableInterrupt (IOSimpleLock* lock); | 获得一个简单锁，并禁用线程抢占及当前CPU的中断。该函数仅用于中断上下文和线程上下文之间共享的锁。释放该锁以确保CPU的中断状态恢复至初始情况时，需要该函数的返回值 |
| void IOSimpleLockUnlockEnableInterrupt (IOSimpleLock* lock, IOInterruptState state); | 释放通过IOSimpleLockLockDisableInterrupt()获得的简单锁。重新启用线程抢占，中断状态恢复至state参数 |

## 7.3.2 互斥

尽管自旋锁对某些应用程序来说非常高效，但它并不适合线程需要长时间持有锁的情况，或线程执行的操作在锁被持有时会阻塞的情况，如分配内存或获得第二个锁。在这些情况下，使用自旋锁效率极低，因为线程试图获得锁时，任何锁竞争都将导致线程连续不断地自旋，这会阻碍CPU执行任何有用的工作。互斥锁不存在这种问题，因为线程试图获得已经被使用的互斥锁时，将被挂起，直至锁可用。操作系统不是将CPU时间花费在自旋上，而是可以调度CPU上的另一个线程。I/O Kit通过IOLock对象支持互斥锁。因为互斥在无法立即获得时会阻塞，所以互斥锁不可用于中断处理程序。

IOLock提供的功能与用于用户空间应用程序的POSIX互斥锁函数非常类似。使用驱动程序代码中的IOLock时，可以采用与使用用户空间代码中的POSIX互斥类似的方式。IOSimpleLock不可以被长时间持有，因为它禁用了抢占（和可能的中断），与IOSimpleLock不同，互斥对CPU的状态或操作系统的调度器并没有这种影响。持有互斥的线程在其时间片已经到期后，仍然可以被另一个线程抢占。如果CPU需要处理一个中断，则持有互斥的线程仍然可以被抢占。但这些可以看做是使用互斥的优点，因为这意味着，在互斥被持有时对执行什么操作并没有任何限制。当互斥被持有时，线程可以分配内存，将内存从用户空间映射到内核（可能会导致内存分页），以及获得另一个互斥（可能阻塞的操作）。

与自旋锁一样，持有互斥的线程必须注意，不要再次获得当前持有的互斥。否则，将会发生死锁。乍一看，这像是人为问题，因为这只是保证线程不要试图获得已经持有的锁。但如果锁被

持有时执行的代码调用其他函数，而这些函数本身又调用其他结束获得锁的函数，该问题将变得非常复杂。

例如，假设我们有一个函数ListEnqueue()，因为该函数可以被多个线程调用，所以需要同步。ListEnqueue()函数可能在工程代码库的多个位置调用，其中一些调用函数可能已经持有了同步锁，而其他的调用函数并未持有该锁。如果示例函数ListEnqueue()要获得一个锁，以确保未持有锁的函数对它进行同步调用，而当持有该锁的函数调用ListEnqueue()时，将会造成死锁。这种情形可以通过使用递归互斥解决。

线程已经获得一个递归互斥后，运行在同一个线程上的任何代码都可以多次重新获得该互斥，而不会导致死锁。共享的资源仍然是同步的，因为试图获得该互斥的其他线程都将阻塞，直至线程获得的所有互斥都被释放。I/O Kit通过IORecursiveLock对象支持递归互斥。

表7-3总结了I/O Kit提供的互斥操作。

表7-3　I/O Kit提供的互斥同步函数

| 函　数 | 描　述 |
| --- | --- |
| IOLock* IOLockAlloc(void);<br>IORecursiveLock* IORecursiveLockAlloc(void); | 分配一个新的IOLock或IORecursiveLock，并返回一个指向初始化对象的指针，失败时返回NULL |
| void IOLockFree(IOLock* lock);<br>void IORecursiveLockFree(IORecursiveLock* lock); | 释放IOLockAlloc()分配的IOLock对象或IORecursiveLockAlloc()分配的IORecursiveLock对象 |
| void IOLockLock(IOLock* lock);<br>void IORecursiveLockLock(IORecursiveLock* lock); | 获得一个互斥，该锁被另一个线程持有时，阻塞调用的线程。线程已经获得递归锁之后，可以多次安全地获得同个锁 |
| boolean_t IOLockTryLock(IOLock* lock);<br>boolean_t IORecursiveLockTryLock(IORecursiveLock* lock); | 仅在互斥锁未被另一个线程持有时获得它。如果该函数无法获得该锁，也不会阻塞调用线程，但会返回一个布尔值，表示该函数是否可以获得该锁 |
| void IOLockUnlock(IOLock* lock);<br>void IORecursiveLockUnlock(IORecursiveLock* lock); | 释放通过IOLockLock()或成功调用IOLockTryLock()获得的互斥。或者，对于递归互斥，释放通过IORecursiveLockLock()或成功调用IORecursiveLockTryLock()获得的锁 |
| boolean_t IORecursiveLockHaveLock<br>(const IORecursiveLock* lock); | 测试递归互斥是否被调用的线程持有。如果该互斥已经被该线程获得，则返回true。如果该互斥未被获得或已经被另一个线程获得，则返回false |

## 7.3.3　条件变量

除了为独占访问共享资源提供互斥，IOLock和IORecursiveLock对象还支持同步原语，称为条件变量。条件变量提供了一种机制，其中线程可以暂停其执行，直至特定的条件（或事件）发生，从而实现在多个线程之间同步。

例如，考虑一下串行端口的驱动程序。我们假设的驱动程序将接收阻塞用户空间应用程序的读请求。如果数据在串行端口上被接收，这些请求将阻塞，并且只返回到用户空间。并不是在驱

动程序内部连续轮询，直至数据可用，一个较好的方案是，创建一个条件变量，挂起线程，使该线程在等待时不占用CPU时间。驱动程序接收到硬件的数据时，它将唤醒正在为条件变量服务的线程。如下面的示例代码所示：

```
void    MyDriver::read (void* buffer, uint32_t* bytesRead)
{
        IOLockLock(m_lock);
        do {
                // 试图从硬件读取数据
                *bytesRead = readFromHardware(buffer);

                // 如果没有可用数据，就一直睡眠，直到硬件接收到数据
                if (*bytesRead == 0)
                {
                        int     result;

                        result = IOLockSleep(m_lock, m_readEvent, THREAD_ABORTSAFE);
                        if (result != THREAD_AWAKENED)
                                break;
                }
        } while (*bytesRead == 0);
        IOLockUnlock(m_lock);
}

void    MyDriver::DataAvailable ()
{
        // 唤醒在 m_readEvent上睡眠的所有线程
        IOLockWakeup(m_lock, m_readEvent, false);
}
```

在上面的例子中，read()方法试图接收硬件设备的数据，但没有可用的数据时将阻塞当前线程，直至硬件具有数据。当硬件有可用数据时，调用DataAvailable()方法（可以调用该方法来响应硬件中断），并唤醒阻塞的线程。请注意，read()方法的整个内容都受IOLock保护。这可以确保所有读取硬件设备数据的尝试都串行化。否则，硬件通知数据可用时，多个线程会被唤醒，同时试图读取设备的数据，这可能会造成竞态条件。IOLockSleep()的行为类似于等价的用户空间函数pthread_cond_wait()。第一个参数是调用者必须持有的锁；IOLockSleep()在睡眠时会自动释放该锁，在收到事件通知之后会重新获得该锁。采用这种方式，线程挂起时将不持有该锁。

m_readEvent参数是条件变量，等待的线程通过该条件变量指定其等待的事件。通知线程通过提供相同的条件变量表示该事件已经发生。驱动程序会定义许多条件变量，分别对应于协调其线程的各个事件。I/O Kit中的条件变量并没有特定的类型，仅为void*类型，唯一标识一个事件。驱动程序通常使用实例变量的地址（如锁本身的地址）作为条件变量，因为使用地址可以保证该值在该驱动程序及系统的其他驱动程序的多个实例中具有唯一性。

表7-4总结了I/O Kit提供的条件变量同步操作。

表7-4　I/O Kit提供的条件变量同步函数

| 函　数 | 描　述 |
| --- | --- |
| int IOLockSleep(IOLock* lock, void* event, UInt32 interruptType); | 自动解除提供的锁，该锁必须由调用者持有，并且等待指定的事件 |
| int IORecursiveLockSleep(IORecursiveLock* lock, void* event, UInt32 interruptType); | 如果调用者作为控制请求的一部分运行在用户空间线程上，用户进程接收到通知时（如SIGHUP或SIGKILL），interruptType参数指定睡眠过程是否应该终止<br>在函数返回之前重新获得该锁 |
| int IOLockSleepDeadline(IOLock* lock, void* event, AbsoluteTime deadline, UInt32 interruptType);<br>int IORecursiveLockSleepDeadline(IORecursiveLock* lock, void* event, AbsoluteTime deadline, UInt32 interruptType); | 使用指定的超时参数执行事件上的睡眠。如果事件在指定时间之前未得到通知，该函数将唤醒，连同重新获得的锁一并返回给调用者 |
| void IOLockWakeup(IOLock* lock, void* event, bool oneThread);<br>void IORecursiveLockWakeup(IORecursiveLock* lock, void* event, bool oneThread); | 通知某个事件已经发生，并唤醒在该事件上睡眠的线程。oneThread参数允许调用者指定是仅唤醒一个睡眠线程，还是唤醒该事件上睡眠的所有线程 |

睡眠函数的interruptType参数用来确定在拥有该线程的进程接收到通知（如SIGHUP或SIGKILL）时，该函数是否应该返回。这在用户客户端执行等待以响应控制请求时非常有用，这种情况下，驱动程序函数将运行在用户空间应用程序创建的线程上。有些情况下，驱动程序在进程接收到一个通知时可能希望放弃该等待，因为用户客户端可能已经终止。下面列出了interruptType的可能值。

- THREAD_UNINT指定这次睡眠不应该被任何通知终止。
- THREAD_INTERRUPTIBLE指定这次睡眠在接收到SIGKILL通知时终止。
- THREAD_ABORTSAFE指定这次睡眠在接收到任意通知时终止。

从睡眠状态唤醒时，函数将返回下列结果值之一。

- THREAD_AWAKENED表示该函数正常返回，事件通过IOLockWakeup()调用得到通知。
- THREAD_TIMED_OUT表示指定的最后期限之前不通知事件。
- THREAD_INTERRUPTED表示拥有线程（驱动程序正在该线程上睡眠）的用户进程接收通知。
- THREAD_RESTART表示等待操作应该完全重启。

### 7.3.4　读/写互斥

互斥的一个问题在于只允许一个线程持有锁，这常常会带来不必要的限制。在很多情况下，没有理由限制多个线程读取共享资源。仅当一个线程希望写入共享资源或对其进行修改时，才需要独占访问。该问题可以通过指定的互斥类型来解决，这种互斥类型称为读/写互斥。

I/O Kit通过IORWLock对象提供读/写互斥。读/写互斥的使用方式与标准互斥类似。区别在于，调用者必须确定读/写互斥是用于读取共享资源（在此种情况下，可以与其他的读者共享互斥），还是用于写入共享资源（在此种情况下，需要独占访问互斥）。I/O Kit根据调用代码希望执行的

操作提供了两个独立的函数。

表7-5总结了I/O Kit提供的读/写互斥同步操作。

表7-5 I/O Kit提供的读/写互斥同步函数

| 函 数 | 描 述 |
| --- | --- |
| IORWLock* IORWLockAlloc(void); | 分配一个新的IORWLock，并返回一个指向初始化对象的指针，失败时返回NULL |
| void IORWLockFree(IORWLock* lock); | 释放IORWLockAlloc()分配的IORWLock |
| void IORWLockRead(IORWLock* lock); | 获得一个用于读取共享资源的读/写互斥。该互斥可以与其他读取者共享，但该互斥被写入者持有时将阻塞 |
| void IORWLockWrite(IORWLock* lock); | 获得一个用于写入共享资源的读/写互斥。调用者可以独占访问该共享资源，读/写互斥被任意读取者或写者持有时将阻塞 |
| void IORWLockUnlock(IORWLock* lock); | 释放通过IORWLockRead()或IORWLockWrite()获得的读/写互斥 |

注意 I/O Kit锁原语基于Mach类型构建。I/O Kit包含取得底层Mach锁类型的函数，其中包括IOSimpleLockGetMachLock()、IOLockGetMachLock()、IORecursiveLockGetMachLock()和IORWLockGetMachLock()。如果要利用由Mach层实现但没有由I/O Kit函数公开的行为，这些函数非常有用。例如，Mach读/写锁可以从共享的读访问升级为独占的写访问，而I/O Kit并不提供相应的函数。

## 7.4 同步异步事件：工作环

如果驱动程序需要响应异步事件（如硬件中断或定时器），那么驱动程序中的同步将变得非常困难。这增加了另外一层复杂度，因为除了需要在多个线程执行之间同步，驱动程序现在还必须与运行在多个线程上的代码和响应异步事件的运行代码进行同步竞争。为了简化驱动程序开发者的工作，I/O Kit提供一个IOWorkLoop类创建单线程，处理异步事件。在I/O Kit术语中，该线程称为“工作环”，驱动程序会为每个异步事件源注册一个IOWorkLoop对象，如中断处理程序和定时器。

工作环线程大多数时间都处于空闲状态，并不消耗CPU时间，只等待事件发生。某个事件发生后，对应的事件源将会通知工作环。工作环线程将被唤醒，处理该事件，然后再返回至睡眠状态。由于所有的事件都在同一个工作环线程上处理，这种设计为同步一个驱动程序中的多个源提供了优雅的方案。这种设计的另一个优点是，由于所有的事件源都在同一个线程上处理，所以处理程序函数无须提供任何额外的锁，因为多个处理程序不可能同时运行。

工作环是每个I/O Kit驱动程序固有的一部分。IOService类是每个驱动程序对象的最终派生基类，提供了一个getWorkLoop()方法，通过该方法，提供者类可以与子驱动程序共享其工作环。驱动程序可以选择使用其提供者类创建的IOWorkLoop对象（通过调用IOService::getWorkLoop()），或者创建自己的IOWorkLoop对象。如果驱动程序只希望接收偶然的异步事件，并且事件处理程序

无须保持最小延迟，那么共享其提供者类的工作环将是一个完美的方案。这还可以简化驱动程序的构造函数及析构函数的代码，并消除为驱动程序本身的工作环创建额外内核线程的开销。处理硬件中断的驱动程序应该创建专用的IOWorkLoop对象，保证最小延迟，因为该工作环并不与其提供者驱动程序的事件源共享。

驱动程序通常在其start()方法内部初始化工作环。如果驱动程序决定与其提供者共享工作环，那么可采用如下方法获取工作环：

```
m_workLoop = getWorkLoop();        //通过IOService超类实现
if (m_workLoop == NULL)
        abort with error;
m_workLoop->retain();
```

如果驱动程序决定创建一个专用的工作环，则它可以实例化一个新的IOWorkLoop对象，如下所示：

```
m_workLoop = IOWorkLoop::workLoop();
if (m_workLoop == NULL)
        abort with error;
```

如果一个驱动程序实例化它自己的IOWorkLoop对象，那么应该考虑覆写IOService方法getWorkLoop()，将其工作环提供给自己的任意子驱动程序。

获得IOWorkLoop对象之后，驱动程序接下来需要注册它希望在该工作环线程上处理的事件源。事件源是继承自IOEventSource基类的一个对象。I/O Kit提供了专门的类，包括IOInterruptEventSource类和IOTimerEventSource类，分别为PCI卡的中断处理程序和定时器事件创建事件源。事件源实例化时会提供一个回调函数，在该事件需要服务时调用。该回调函数一定会从工作环线程调用，或与运行在工作环线程上的任意代码同步。为定时器创建一个事件源，并将其添加到工作环的代码如下：

```
m_timerEventSource = IOTimerEventSource::timerEventSource(this, TimerFiredFunc);
if (m_timerEventSource == NULL)
        abort with error;

result = m_workLoop->addEventSource(m_timerEventSource);
if (result != kIOReturnSuccess)
        abort with error;
```

**注意**　就中断处理程序而言，运行在工作环线程上的回调对应二级中断处理程序；主中断仍然是硬件中断级别，运行在线程上下文中。

驱动程序通常会在其主驱动程序类中指定一个静态方法，作为IOEventSource的回调函数。尽管提供给回调函数的参数与调用的事件源类型相关，但第一个参数总是OSObject类的指针，能够将驱动程序的实例传递给回调函数。根据回调函数的复杂性，该实现可以直接在静态类函数中处理事件，或调用驱动程序的实例方法。

### 7.4.1 IOCommandGate

还有一个问题是，驱动程序可能需要将运行在自身线程上的代码与运行在工作环线程上的事件行为同步。例如，考虑一下磁盘设备的驱动程序，它会接收用户空间应用程序读写事务的请求。由于该硬件在某一时刻只能为一个事务提供服务，其他事务将被添加到一个队列中，每当一个事务完成时（通过中断通知），驱动程序将从队列的头部移除下一个事务，并为它提供服务。由于该假设的驱动程序从其中断处理程序（运行在工作环上）及用户客户端操作该事务队列，所以它需要一种方法，将运行在工作环线程上的代码与运行在线程上下文中的代码同步。

为了解决该问题，I/O Kit提供了一个特定事件源，称为命令门，由IOCommandGate类实现。命令门与任何其他的事件源一样，也是安装在一个工作环上，但本身并不生成事件，而是用来执行工作环线程上的任意回调函数。IOCommandGate类包含一个runAction()方法，该方法接受一个函数指针，并运行该函数，从而实现与任何其他工作环事件的同步。通过IOCommandGate对象执行的函数称为“门”函数。门函数实际上并不在工作环线程上运行。门函数只是串行化所有其他的工作环源，包括通过同个IOCommandGate调用的其他门函数。命令门实际上在调用runAction()的线程上运行函数。这省去了上下文切换到工作环线程的开销。下面是通过IOCommandGate调用函数的一个例子：

```
//门之外调用的方法
IOReturn MyDriver::startTransaction(Transaction* transaction)
{
        //通过命令门调用StartTransactionAction函数
        return m_commandGate->runAction((IOCommandGate::Action)StartTransactionAction);
}

// IOCommandGate::Action
IOReturn MyDriver::StartTransactionAction(MyDriver* self, Transaction* transaction)
{
        //该静态方法与所有的工作环方法同步
        ...
        return kIOReturnSuccess;
}
```

上面的例子为操作函数提供了一个参数，即transaction参数。IOCommandGate最多允许为动作函数提供4个参数。

IOCommandGate通过递归锁实现它的同步。这使通过IOCommandGate调用的函数可以通过同一个IOCommandGate调用其他函数，而且不会造成死锁。使用与本章前面介绍的方法类似的方法，门函数也可以在条件变量上睡眠和唤醒。IOCommandGate类包含两个方法，用于在条件变量上睡眠，一个是在指定的时间之后超时，一个是一直睡眠到条件变量收到通知。这两个函数如下所示：

```
IOReturn            commandSleep(void* event, UInt32 interruptType);
IOReturn            commandSleep(void* event, AbsoluteTime deadline, UInt32 interruptType);
```

event、interruptType和deadline参数与传递给IOLockSleep()和IOLockSleepDeadline()（见表7-4）的同名参数有着相同的含义。7.3.3节介绍，一个线程在持有同步锁时，只能在一个条件

变量上睡眠。commandSleep()方法遵循同样的规则，该方法必须由通过IOCommandGate对象调用的函数调用。

为了在事件发生时通知命令门的条件变量，IOCommandGate类提供了下列方法，该方法与IOLockWakeup()函数（见表7-4）的形式相同。

```
void    commandWakeup(void* event, bool oneThread);
```

IOCommandGate对象提供了IORecursiveLock的所有同步功能，包括互斥锁及条件变量支持。这使驱动程序可以使用IOCommandGate对象而不是I/O Kit锁，进行其所有的同步。当驱动程序需要一个工作环为异步事件提供同步时，这非常有用；驱动程序可以创建一个IOCommandGate，并在其实现中使用工作环，而不是创建额外的锁。

### 7.4.2　定时器

我们讨论的最后一个事件源是IOTimerEventSource，该事件源提供一个基本的、非周期性的定时器。IOTimerEventSource的一种常见用法是，创建一个看门狗定时器以取消一定时间内未完成的操作。

创建定时器时，调用者会提供一个回调函数，定时器到期时运行。与任何其他事件源一样，定时器必须添加到IOWorkLoop对象上，其回调函数将从一个上下文运行，该上下文与安装在工作环上的所有其他事件源同步。

定时器可以初始化并添加到现有的工作环上，如以下代码所示：

```
m_timer = IOTimerEventSource::timerEventSource(this, TimerFired);
if (m_timer == NULL)
        abort with error;

result = m_workLoop->addEventSource(m_timer);
if (result != kIOReturnSuccess)
        abort with error;
```

IOTimerEventSource::timerEventSource()方法实例化一个新的IOTimerEventSource对象。它的第一个参数是OSObject对象指针，传递给回调函数，从而访问驱动程序对象。

接下来，需要激活定时器。这通过指定从当前时间到定时器触发时的一段超时时间实现。IOTimerEventSource类的多个方法可用于指定单位分别为毫秒、微秒、纳秒或任意时间单元的超时：

```
IOReturn        setTimeoutMS(UInt32 ms);
IOReturn        setTimeoutUS(UInt32 us);
IOReturn        setTimeout(UInt32 interval, UInt32 scale_factor = kNanosecondScale);
```

定时器触发时指定的回调函数将会运行，其签名如下：

```
void    MyDriver::TimerFired(OSObject* owner, IOTimerEventSource* sender)
{
}
```

尽管IOTimerEventSource提供的定时器没有周期性，但定时器的下一个超时值可以在其回调

函数内部设置。你会发现，IOTimerEventSource实例是作为一个参数提供给回调函数，这使回调可以很容易地重新安装定时器。如果需要为特定应用程序避免定时器后续触发之间的偏差，可以使用IOTimerEventSource的wakeAtTime()方法，该方法允许将超时时间指定为绝对时间。对于不随时间偏移的周期性定时器，可以在每次重新安装时通过指定绝对时间进行创建。

如果需要取消定时器，例如，因为看门狗定时器保护的操作已经成功完成，IOTimerEventSource类提供了一个名为cancelTimeout()的方法。该方法也是一个同步指针，可以保证定时器的回调函数在该方法返回时不会调用。

### 7.4.3 释放工作环

驱动程序卸载时需要释放它创建的所有工作环。该处理包含两步，一个是从工作环移除事件源，另一个是释放IOWorkLoop对象。第一部分通常由驱动程序在其stop()方法中执行，但通常工作环本身并不释放，直至调用驱动程序的free()方法。下面是一个可能的实现：

```
void    MyDriver::stop(IOService* provider)
{
        //移除并释放命令门事件源
        if (m_commandGate != NULL)
        {
                m_workLoop->removeEventSource(m_commandGate);
                m_commandGate->release();
                m_commandGate = NULL;
        }

        //移除并释放定时器事件源
        if (m_timer != NULL)
        {
                m_timer->cancelTimeout();
                m_workLoop->removeEventSource(m_timer);
                m_timer->release();
                m_timer = NULL;
        }

        super::stop(provider);
}


void    MyDriver::free()
{
        //释放工作环
        if (m_workLoop != NULL)
        {
                m_workLoop->release();
                m_workLoop = NULL;
        }

        super::free();
}
```

## 7.5　内核线程

与用户空间应用程序不同，驱动程序处于活动状态时，并没有一直运行的主线程。相反，驱动程序通常在现有线程上执行，响应某些事件。例如，当用户空间进程通过驱动程序的用户客户端进行控制请求时，驱动程序在用户进程拥有的调用线程上下文中执行。类似地，驱动程序的硬件发生中断时，驱动程序的二级中断处理程序从工作环线程执行。因为驱动程为响应这类事件而执行，所以如果驱动程序没有处理事件，如控制请求或中断处理程序，它通常不会在系统中的线程上运行任何代码。

缺少主线程的等价功能对驱动程序来说也是一个问题，特别是需要代码持续执行，以便可以周期性地轮询其硬件设备时，或希望执行耗时操作（如下载其硬件的固件）时。这些情况下，驱动程序可以创建其自身的内核线程，并在该线程上继续执行代码，而不是使用用户进程的线程或工作环线程。

代码清单5-16是使用内核线程的好例子。在该例中，用户空间应用程序向驱动程序发送一个控制请求，以执行延迟操作。驱动程序并非在进程线程的上下文中执行延迟，因为这将阻塞调用线程，相反，它选择创建一个后台线程，执行睡眠操作。在延迟完成后，驱动程序向用户进程发送一个异步通知。这种简单的设计模式说明了驱动程序如何使用内核线程，将可以通过调用进程阻塞线程的同步操作，转换为看起来与调用进程异步的操作。

创建内核线程的API来自底层较低级的Mach层，并在头文件<kern/thread.h>中声明。无须直接包括该头文件，因为头文件<IOKit/IOLib.h>中已经包含了它。如果查看IOLib.h的内容，你会发现该文件包含少量线程函数，包括创建线程的支持。但从Mac OS X 10.6开始，苹果公司已经废弃了这些函数，并用Mach函数代替。

kernel_thread_start()函数可以用来创建内核线程。其原型如下：

```
kern_return_t   kernel_thread_start(thread_continue_t continuation, void* parameter,
                    thread_t* new_thread);
```

continuation参数实际上是一个指向新建线程开始例程的函数指针。如果线程创建成功，kernel_thread_start()的返回值为kIOreturnSuccess，并且new_thread参数将包含新创建线程的一个引用，很像用户空间函数pthread_create()返回的引用。返回的thread_t对象对调用者来说毫无用处，因为几乎没有用于线程操作的公共内核API。但可以将thread_t值与current_thread()返回的值进行比较，确定该活动的函数是否在特定的线程上运行。

大多数情况下，调用者在创建线程之后会立即释放thread_t对象；这并不影响后台线程本身的运行。为了释放thread_t对象，应该使用thread_deallocate()函数，该函数原型如下：

```
void    thread_deallocate(thread_t thread);
```

**注意**　释放kernel_thread_start()返回的thread_t对象，以防止内存泄漏，这非常重要。

线程的开始例程签名如下所示：

```
void    ThreadFunction (void* parameter, wait_result_t waitResult)
```

函数的第一个参数parameter，对应于传递给kernel_thread_start()的值，使线程的创建者可以将上下文传递给线程的函数。对通过kernel_thread_start()创建的线程来说，函数的第二个参数waitResult毫无用处，只表示thread_continue_t类型线程函数的结果。在内部，内核在其他地方使用thread_continue_t类型，并在这些情况下使用waitResult的值。

最后，线程完成操作后希望终止时，线程函数应该调用thread_terminate()，如下所示：

```
thread_terminate(current_thread());
```

请注意，尽管看起来可以从另一个线程终止一个后台线程，但这会失败，并返回给调用者一个错误。thread_terminate()函数只可以终止当前线程。要想终止后台线程，例如，驱动程序卸载时，该驱动程序应该通知后台线程退出，让该线程终止自身。可以设置一个后台线程定期检查的布尔标志或设置一个后台线程睡眠的条件变量，来实现这一通知。



## 7.6 小结

- 驱动程序运行在多线程环境中。
- 驱动程序类的方法可能会从任意线程调用。这意味着，即使驱动程序不创建自己的线程，也不可避免地需要同步访问它共享的实例变量及硬件。
- I/O Kit提供了同步函数，驱动程序可以使用这些函数防止多个线程同时修改共享数据。
- 自旋锁同步机制适合仅在短时间内持有锁的代码，或在主中断处理程序内部执行的代码。
- 如果要长时间持有锁，那么使用互斥是较好的选择，但在主中断处理程序内部无法使用。
- 为了同步任何时刻发生的事件，如硬件中断或软件定时器，I/O Kit提供了一个专用线程，这些事件的处理程序在该线程上运行。这就是工作环线程，I/O Kit可以确保安装在工作环上的所有事件源都按串行方式执行。

# 第8章 USB

USB（Universal Serial Bus，通用串行总线）是一项在各种产品中普遍存在的技术，尤其是计算机外设，包括鼠标、键盘、硬盘和打印机以及可以连接到计算机上的任何其他类型的设备或装置。USB是定义设备（如打印机或移动电话）与由计算机设备控制的主机（如Mac或iPad）之间通信的规范。USB规范于1996年由一个公司联盟制定，其中包括康柏、DEC、IBM、英特尔、微软、NEC和北电。目的是将一系列连接器替换为通用连接器，使外部设备连接到个人计算机上变得更加容易。USB规范当前是3.0版。对3.0的支持仍然刚刚兴起，目前2.0已得到普遍支持。苹果公司已经发布了能够支持最新USB 3.0规范的硬件，但苹果计算机在OS X之前已经支持USB。iOS系列设备本身就是USB设备，但它们还可以用做USB主机。例如iPad可以用作USB设备（如数码相机）的主机。

USB基于主-从系统，在该系统中，控制器（主机）与从设备进行通信。主机与总线通常是一对一的关系。

内核程序员的任务是为硬件设备编写驱动程序，那么极有可能就是为USB设备编写驱动程序。好消息（至少对我们这些懒惰的程序员来说）是，如果设备遵守USB-IF（USB Implementers Forum，USB实施者论坛）定义的类，那么就可以不用为它开发驱动程序。例如，键盘和鼠标遵循HID（human interface device，人机接口设备）规范，厂商因此无须提供驱动程序，因为操作系统已经包含了一个通用驱动程序，可以用来与这些设备通信。但如果设备具有高级功能时，厂商仍然可以选择开发驱动程序，如为鼠标上另一个自定义按钮开发驱动程序。

本章将简要概述USB规范及体系结构。USB规范要比本书长得多，因此，显然不可能对它进行详细讨论。我们只关注实现USB设备驱动程序的相关部分。本章还将讨论I/O Kit提供的USB子系统的体系结构，并提供一个虚拟USB设备驱动程序代码。值得一提的是，USB驱动程序在内核和用户空间中都可以编写。当一个驱动程序/设备可以由许多应用程序同时访问，或驱动程序的主客户端是内核本身时，通常需要内核驱动程序。例如，通常在内核中实现的设备是存储器、网卡、音频及显示驱动程序，而打印机、鼠标及键盘的驱动程序可以全部或部分交由用户空间驱动程序处理。本章将关注通用USB以及内核空间驱动程序的实现。用户空间驱动程序的详细内容请参见第15章。

## 8.1 USB体系结构

在USB系统中，主机控制器是“主”，USB设备是“从”。USB拓扑结构用一个树型结构表示，

从而形成总线，其中主机控制器为根节点，是从属设备的控制器。主机控制器负责协调总线上的任务，而设备无法执行I/O或总线上的任何其他任务，除非主机首先要求这么做。

树型结构借助集线器进行扩展，使连接的设备可以成为总线的一部分，从而将其扩展为一个新的分支。集线器的从属设备还可以连接其他设备，最多有4层。根集线器通常嵌入到主机控制器本身，成为主机控制器的一部分。MacBook计算机的USB拓扑结构如图8-1所示。

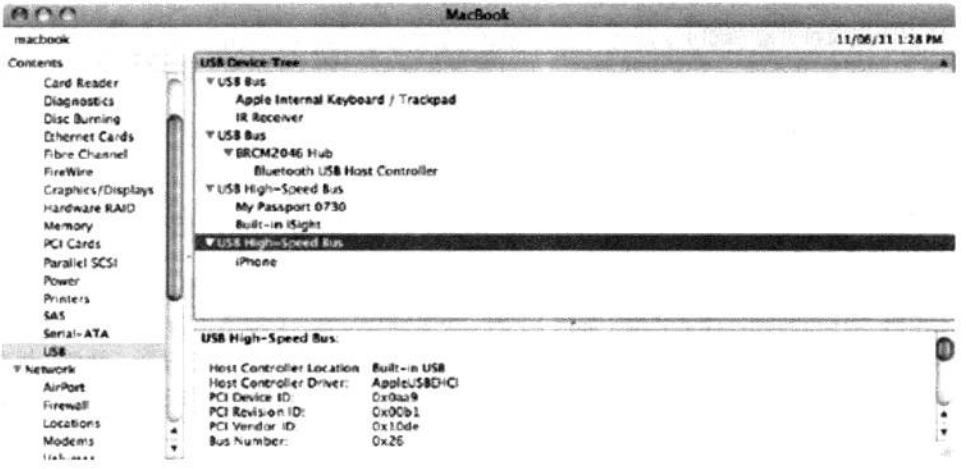



图8-1 连接外部设备的MacBook USB拓扑结构系统信息

图8-1中的系统有4个内置的USB总线，但只有两个总线实际连接到了外部USB端口。虽然大多数人将USB设备看做外部设备，但USB通常用来与计算机系统中固定的内部设备通信。图8-1中，MacBook的前两个USB总线集成到了笔记本中，用来连接内部的键盘、触控板和IR接收机。第二个内部总线连接到蓝牙USB主机控制器。这两个总线都是USB 1.1总线，这并没有什么问题，因为连接到这两个总线的设备对带宽需求都很低。

该MacBook连接了两个外部USB设备，一个外部硬盘和一个iPhone，每个设备都连接到系统上独立的物理USB端口上。你可能认为外部USB端口有专用总线，但情况不一定总是如此。在图8-1中可以看到，其中有一个总线连接了另外一个USB设备，即内置的iSight照相机。该照相机要求的带宽比1.1 USB总线/控制器可以提供的带宽高，因此，它与两个USB 2.0控制器中的一个连接，从而可以避免仅为iSight提供一个单独的控制器，节省空间、部件、费用和电池电量。但缺点是，外部设备将与iSight照相机共享带宽（使用照相机时）。

具体的USB拓扑结构因系统而异，这只是它的一种表示方法。

USB系统中的每个总线最多可以支持127个设备，其中包括集线器设备（它们也是USB设备）。USB总线上的带宽在连接的设备之间共享，因此有大量激活的设备将会造成大量带宽损失，这就是目前单独的主机控制器通常拥有一个USB端口的原因，其中每个端口提供一条总线，而不是使多个端口连接到单个控制器上。

USB设计为热插拔，这意味着，系统在操作期间的任意时刻都可以处理设备的插入或移除，但根据设备的类型，这么做并不一定安全，最典型的例子是硬盘，如果在发生错误时拔出硬盘，将会破坏驱动程序的文件系统。从开发者的角度来看，这意味着，设计驱动程序代码时需要包括可预测设备的插入或移除功能。

USB规范还可以通过总线为设备提供电源，这是它相对于较早总线技术的主要进步，例如，只需将iPhone连接到笔记本电脑就可以充电，或对硬盘等设备进行自供电。

USB的主要特征总结如下。

- 面向所有设备的一个连接器类型，可以替代各种较古老的连接器，如PS/2。
- 可以将多个设备连接到同一个连接器。
- USB总线可以借助集线器进行扩展，每个总线最多支持127个设备。
- 设备可热插拔。
- 设备可以由总线供电，根据电量需求，设备可以通过总线进行控制或完全供电。
- 带宽在总线上的设备之间共享。
- USB设备是一个从属设备，若无主机允许，将无法启动总线上的任务。

USB不同于传统硬件体系结构（如PCI）的一个关键特征是，USB并不直接生成系统中断。设备能够异步通知系统发生的事件，这对许多类型的常见设备操作来说非常重要，例如，每当有新的网络分组到达时向系统发出通知的网络设备，或者告诉系统其当前位置或哪一个键被按下的鼠标或键盘。

USB设备仍然可以提供类似于中断的功能，但不能像PCI设备那样直接中断CPU。为了接收USB设备的中断，必须向设备发送一个中断传输。在该传输已经发送，而且事件发生（如网络分组的到达）后，USB设备不是简单地直接通知主机控制器，而必须由主机控制器进行轮询。USB设备仅在主机告诉它这么做时，才会与总线交互。主机控制器会为中断传输提供最大延迟保证，可实现的最低延迟为125ms。设备可以在端点描述符中，为中断轮询指定其所需的时间间隔。

应该注意的是，尽管USB设备本身无法直接中断CPU，但USB主机控制器可以，并且可能也会这么做，以响应设备的中断传输完成。大多数现代计算机系统都通过PCI与USB控制器通话。

---

**提示**　USB规范由非营利性组织USB-IF管理。其网站为http://www.usb.org。

---

### 8.1.1　USB传输速度

USB标准第1版支持两种速度：低速1.5 Mbit/s和全速12 Mbit/s。低速设备不易受电磁干扰影响，因此这些设备可以使用质量较低的部件做成，造价低廉。这降低了较简单USB设备的费用，这些USB设备并不需要额外的带宽，但全速设备需要。USB 2.0标准包括全速模式，能够在高达

480 Mbit/s的速度下运转，还向后兼容USB 1.0。凭借这一点USB就可以与FireWire竞争，FireWire当时以400 Mbit/s的速度运转（参见表8-1，了解USB传输速度）。

表8-1 USB传输速度

| 名 称 | 速 率 | 引入时间 |
|---|---|---|
| 低速 | 1.5 Mbit/s | USB 1.0 |
| 全速 | 12 Mbit/s | USB 1.0 |
| 高速 | 480 Mbit/s | USB 2.0 |
| 超高速 | 5.0 Gbit/s | USB 3.0 |

USB 3.0规范支持高达5 Gbit/s的速度，在2011年Thunderbolt发布之前，它是最快的通用外部总线。目前，苹果尚未在其计算机中安装USB 3.0兼容设备，尽管第三方为Mac提供了支持ExpressCard和PCI Express 的可用设备。

### 8.1.2 主机控制器

主机控制器由单独的规范管理，该规范用来确定计算机系统与主机控制器的通信方式。现代系统通常将USB主机控制器嵌入到主板的I/O控制器（南桥）上。大部分主机控制器都有一个PCI接口，系统用它与控制器通信。驱动程序与USB设备通信并非直接进行，而是通过PCI与主机控制器通信，虽然对于驱动程序来说，似乎是直接与设备通信，但这完全是因为I/O Kit提供的面向对象抽象层的作用。USB并不直接生成中断，但主机控制器可以使用DMA和中断。

通常，32位x86系统只有16个中断线，供外设使用的中断线通常也就是2~3个。USB可以解决该问题，因为计算机系统仅需要将一个中断线用于USB主机控制器，便可以与连接到控制器的所有设备通信。USB规范使主机与USB设备的通信方式标准化，但计算机系统与主机控制器的通信有以下几个标准。

- UHCI（Universal Host Controller Interface，通用主机控制器接口）：UHCI由英特尔开发。UHCI规范支持低速和全速的USB 1.x设备。
- OHCI（Open Host Controller Interface，开放主机控制器接口）：OHCI由微软和康柏等公司开发，也面向USB 1.x设备。OHCI控制器比UHCI更智能，可以更合理地嵌入到控制器本身，而UHCI只面向硬件层，需要操作系统安装更复杂的主机控制器驱动程序。
- EHCI（Enhanced Host Controller Interface，增强型主机控制器接口）：EHCI面向USB 2.0创建，支持480 Mbit/s的高速设备。EHCI并不处理USB 1.x设备，因此，需要包含基于UHCI或OHCI的控制器，以处理基于1.x系列规范的设备。
- xHCI（Extensible Host Controller Interface，可扩展主机控制器接口）：xHCI由英特尔设计，支持USB 3.0规范。xHCI是统一的主机控制器，有了它就不再需要EHCI、OHCI和UHCI。

一个计算机系统有数个主机控制器很常见，每个主机控制器支持不同的主机控制器接口。例

如，在图8-1中，MacBook有两个OHCI控制器和两个支持USB 2.0的EHCI控制器。Mac OS X目前的控制器驱动程序支持OHCI、UHCI和EHCI，但不支持xHCI。

**USB OTG**

USB 2.0规范还包含一个补充标准，称为USB OTG（On-The-Go）。虽然嵌入式设备（如移动电话）通常作为USB（从）设备，但USB OTG允许角色切换，移动设备本身还可以作为USB主机。USB OTG只支持这两种设备，并不支持集线器。iPad是一个很好的例子，它可以作为USB设备连接到计算机系统。如果使用特定的适配器，iPad还可以作为USB主机，连接数码照相机或内存卡。

### 8.1.3　USB协议

串行端口设备并没有协议（应用程序负责实现一个协议），只有位流和字节流，与此不同，USB有基于包的协议。这很有必要，因为总线是共享的，并且会连接许多设备，必须对这些设备分别寻址。除非你是一名为USB设备编写固件的硬件工程师，否则你并不需要了解甚至不需要知道通信是如何进行的，但在某些情况下，对协议层的通信方式有基本了解，会对调试问题有所帮助。

USB协议在主机控制器和设备之间实现，并且确定总线上的数据传输方式。驱动程序实际上并不会深刻理解或影响该处理，因为具体细节由主机和设备的电子设备处理，而不是驱动程序，而网络则不同，其通信协议的诸多方面都受软件控制。为了知道或截获实际上通过数据线的内容，需要一个USB包分析器。USB分析器是一种特定的设备（通常非常昂贵），可以连接在设备和主机之间，捕获它们之间的通信数据。

USB包由8位小端序格式的字（LSB）组成。USB协议有4种主要的包类型。

- 令牌包：作为首部，告诉接收者接下来的包/数据类型。令牌包有3种类型：IN、OUT和Setup。前两种类型指定包的方向，最后一种类型用于发起控制传输。包的方向要从主机角度看，因此，IN是指从设备到主机的传输，OUT是指从主机到设备的传输。
- 数据包：可以传送任意数据，每个数据包是0~1024字节。
- 握手包：发送此包，以确认包的传送是成功（ACK）还是不成功（NAK），以及报告出错（STALL）。
- 帧起始包：每隔一定时间发送一次，将数据流同步为同步传输模式。

每种包类型的结构如图8-2所示。

所有的USB包都以sync和PID（packet identifier，包标识符）字段开始。sync字段位于其他数据之前，可供接收者进行时钟同步。对于低速和全速设备该字段为8位，高速设备为32位。通过PID字段解码器可以确定其后面的包类型。PID字段的可能值如表8-2所示。PID是4位宽，尽管总共是8位。最后4位是校验字段，是前4位的补码，可以帮助确定包是否可用，是否被破坏。

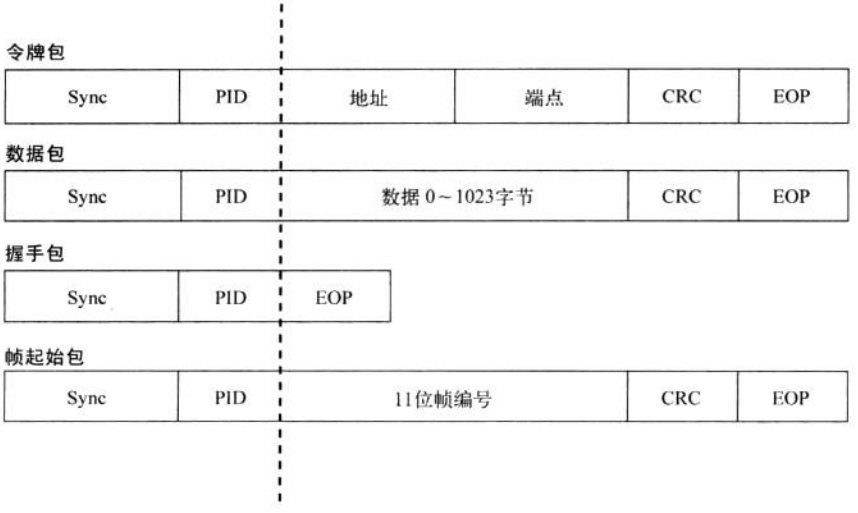



图8-2 USB包类型的结构



表8-2 可能的PID值

| 类 型 | PID | 描 述 |
|---|---|---|
| 令牌 | 0001 | OUT令牌 |
| 令牌 | 1001 | IN令牌 |
| 令牌 | 0101 | SOF令牌 |
| 令牌 | 1101 | Setup令牌 |
| 数据 | 0011 | DATA0 |
| 数据 | 1011 | DATA1 |
| 数据 | 0111 | DATA2 |
| 数据 | 1111 | MDATA |
| 握手 | 0010 | ACK握手 |
| 握手 | 1010 | NAK握手 |
| 握手 | 1110 | STALL握手 |
| 握手 | 0110 | NYET（Not yet，暂无描述） |
| 其他 | 1100 | 前导 |
| 其他 | 1100 | 错误 |
| 其他 | 1000 | 拆分 |
| 其他 | 0100 | 查验 |

令牌包用于对特定设备进行寻址。其地址字段指定包发向/来自哪一个设备，是1~127的一个

数值，在总线上对设备进行寻址。一个USB设备可能有多个端点，每个端点都是一个独立的通信信道，端点字段指定包发送到设备的哪一个端点。本章后面将对端点进行讨论。

所有的包类型都有一个CRC（Cyclic Redundancy Check，循环冗余校验——用于验证数据的完整性）字段。除了数据包，所有的包类型都含有一个5位宽的CRC字段，数据包为较宽的16位CRC字段。

EOP（end of packet，包结尾）字段用作间隔符。

一个USB事务最多可以包含3个数据包。数据包由PID字段表示，可以是：DATA0、DATA1、DATA2和MDATA，但后两个仅用于同步传输模式。PID字段决定传输哪一个数据包，如表8-2所示。

从主机到设备的数据传输如下所示。

(1) 令牌包：包含设备的地址和端点。PID表示OUT传输。

(2) Data0包：包含1024字节的负载数据。

(3) Data1包：包含322字节的负载数据。

(4) 握手包：从设备向主机发送，表示传输的状态，如ACK、NAK或STALL。

### 8.1.4　端点

主机控制器和USB设备之间的通信基于端点。端点是单向的，为IN或OUT，即设备到主机的通信或主机到设备的通信。主机控制器到端点的连接称为管道。管道有两种类型：传送数据的流管道（stream pipe）和传送控制请求的消息管道（message pipe）。一个USB设备最多可以支持32个端点，最多16个IN端点和16个OUT端点。端点地址0比较特殊，预留给设备配置。可用的端点有4种不同的类型。

- **批量端点**（bulk endpoint）：用于传输大量的数据。批量传输无法保证及时发送或带宽，但可以保证发送及错误检测。批量传输并不适用于低速模式。硬盘、扫描仪、打印机和网卡通常使用批量传输。
- **控制端点**（control endpoint）：用于设备配置和状态查找。通过使用保留的带宽，可以保证发送对控制端点的请求。
- **中断端点**（interrupt endpoint）：用于交换少量对时间敏感的数据，可以保证发送。
- **同步端点**（isochronous endpoint）：提供保证的带宽，但并不保证发送。数据丢失时不会重新发送，这适用于视频和音频应用程序。

### 8.1.5　USB描述符

USB描述符用于描述设备的功能、类型、需求等。描述符采用层次结构进行组织，包含下列主要的描述符类型。

- 设备描述符（device descriptor）：包含USB设备的产品ID（idProduct）和厂商ID（idVendor）。每个设备只有一个设备描述符。该描述符还包含它后面有多少个描述符的信息。厂商ID和产品ID都是16位整数。厂商ID由USB-IF分配。厂商可以选择任意的16位值作为产品ID。

厂商/产品ID组合必须唯一，以避免发生问题，因为要用它们来确定合适的设备驱动程序。设备描述符还包含两个表示设备类型的字段：bDeviceClass和bDeviceSubClass。

- 配置描述符（configuration descriptor）：指定备用的配置，设备也可以在该配置中操作。例如，一个设备可能有两种配置：一种为自供电时的配置，一种为总线供电时的配置。后者可以在受限模式下操作，该模式只允许执行一部分整体功能，或许只能编写设备固件。在任意给定时刻，只有一种配置可以是活动的。配置描述符下面可能还会存在一些接口，但一个设备有多个配置描述符的情况并不常见。
- 接口描述符（interface descriptor）：这是共同执行一个功能的端点集合或组。将其看做一个逻辑子设备很有好处。一个接口可能有0个或更多个端点。例如，在图8-3中，我们可以看到一个多功能USB设备，该设备包含两个接口：接口#0是一个打印机，而接口#1则是一个扫描仪。多个接口可以都活动，并同时操作。类似于设备描述符，接口描述符也有表示接口类的字段：bInterfaceClass和bInterfaceSubClass。
- 端点描述符（endpoint descriptor）：描述端点的类型（批量、中断、同步或控制）及方向（IN、OUT）。

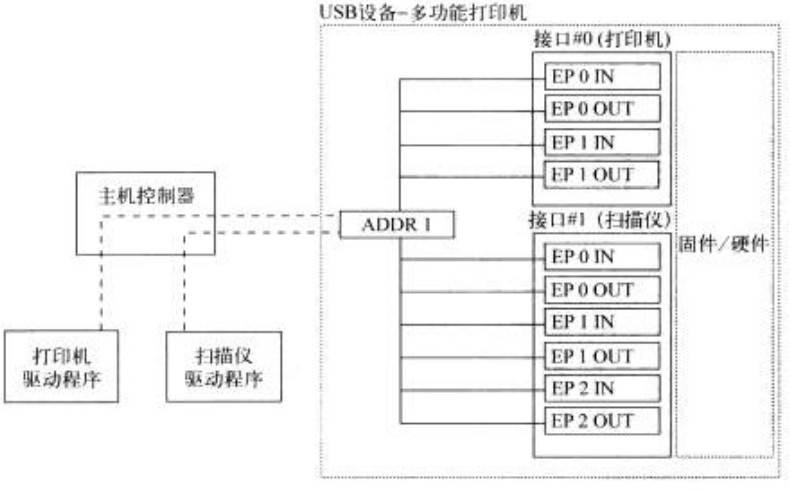



图8-3 带有两个接口的USB复合设备

## 8.1.6 USB设备类

USB描述符包含一些类型代码，它标识系统上连接的设备类型，并且可以用来确定适当的设备驱动程序，进行加载。类型代码可以在设备描述符、接口描述符或两者中指定。如果设备描述符中指定了00h类代码，则意味着实际的类代码应该从接口描述符中读取。另外，还有可以进一步缩小设备类型的子类字段。可用类代码的一小部分如表8-3所示。

表8-3　一部分标准的USB类代码

| 基　类 | 使用的描述符 | 设备类型 |
|---|---|---|
| 0x00 | 设备 | 接口描述符中提供的类信息 |
| 0x01 | 接口 | 音频 |
| 0x03 | 接口 | HID（人机接口设备）、鼠标、键盘、触控板等 |
| 0x08 | 接口 | 大容量存储器、硬盘、指状存储器等 |
| 0xFF | 两者 | 取决于厂商 |

许多操作系统（包括Mac OS X和iOS）都为标准类设备提供了默认驱动程序，操作系统无须安装第三方驱动程序就可以处理大容量存储器或音频USB设备。厂商也可以为具有额外功能的设备提供可选驱动程序，因为操作系统的通用驱动程序中并不提供这些功能。为此，I/O Kit匹配系统可以用来确保匹配更多的特定驱动程序，而非默认驱动程序。

**提示**　类型代码的完整列表及更多详细描述，请参见http://www.usb.org/developers/defined_class。

## 8.2　I/O Kit USB 支持

I/O Kit中的USB支持由IOUSBFamily提供，通过包标识符com.apple.iokit.IOUSBFamily确定，IOUSBFamily是一个动态可装载的KEXT。USB族提供内核中核心的USB处理，并包含主机控制器的驱动程序以及表示USB设备、接口和管道的抽象类。USB族的类层次结构如图8-4所示。

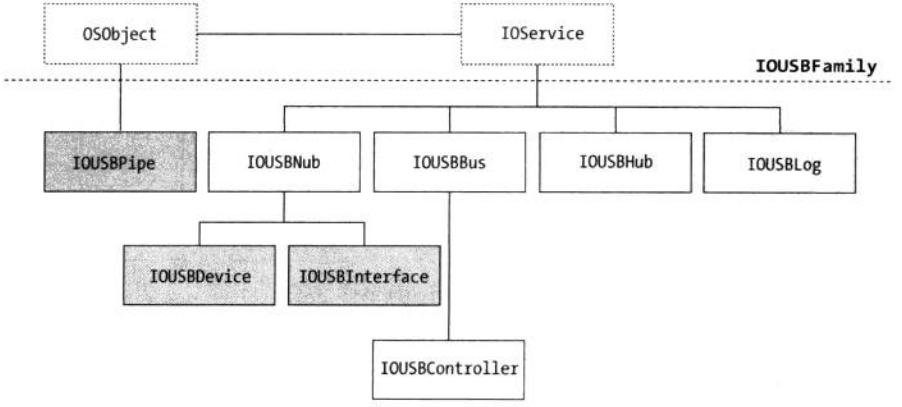



图8-4　IOUSBFamily的类层次结构

如果你仅仅想实现一个USB设备的驱动程序，那么这些类大多没用。图8-4中灰色部分是用于开发USB驱动程序的主要类，包括IOUSBPipe、IOUSBDevice和IOUSBInterface，我们随后将进行

详细讨论。

如果需要支持新的主机控制器，则可以通过继承IOUSBController实现，但内核已经为遵从UHCI、OHCI和EHCI的主机控制器提供了驱动程序。尽管图8-4中并没有显示，但这些都是IOUSBController的子类，分别为AppleUSBUHCI、AppleUSBOHCI和AppleUSBEHCI。

**提示** USB族并不是XNU源代码发布版的一部分，但可以从http://opensource.apple.com下载到源代码形式的文件。源包包括整个USB族的源代码，源代码中包括UHCI、OHCI和EHCI控制器的实现。它还包括USB驱动程序的示例代码，以及从用户空间枚举和访问USB设备的方式。

## 8.2.1 USB设备和驱动程序处理

插入一个USB设备时，USB族将创建IOUSBDevice类（IOService的子类）的一个实例，并将它插入到I/O Registry中。设备插入到总线上时，为它创建且只创建一个IOUSBDevice实例。IOUSBDevice的提供者是连接了设备的IOUSBController。IOUSBDevice类提供了USB设备的设备描述符和配置描述符的一个抽象。接口描述符可以通过IOUSBInterface类访问。IOUSBDevice为IOUSBInterface的提供者，如图8-5所示。

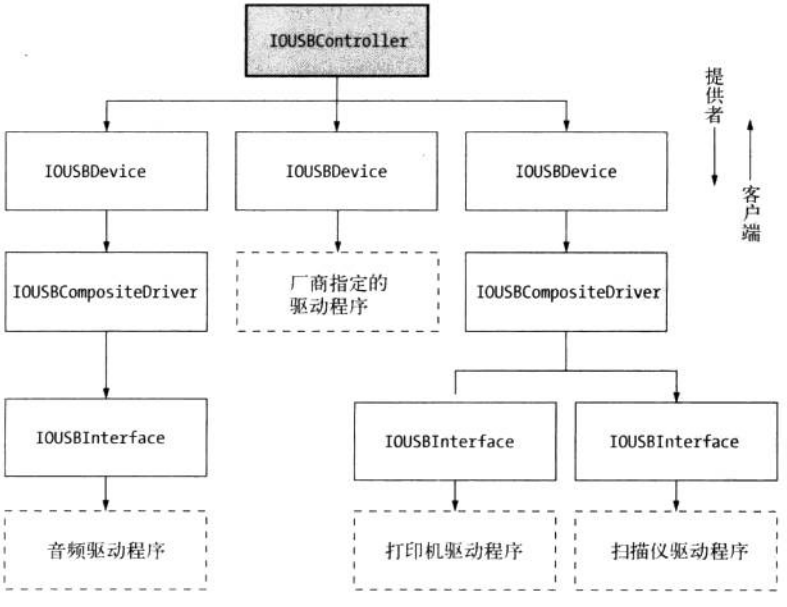



图8-5 USB设备和驱动程序提供者的关系

图8-5展示了3个USB设备及其与高层提供者之间的关系。

- *左边的驱动程序*：这是一个音频设备的驱动程序。你可能已经注意到，IOUSBDevice和IOUSBInterface之间还有一个驱动程序IOUSBCompositerDriver。该复合驱动程序对USB设备进行匹配、加载，该USB设备在其设备描述符中，将其设备类型及子类型设置为0，并且没有其他厂商指定驱动程序与其匹配。单从该驱动程序的名称看，它可能表示这只适用于有多个功能的复合驱动程序，但该驱动程序甚至适用于只有一个接口的设备。该复合驱动程序执行的唯一功能是，选择设备的活动配置（如果它有多个配置描述符），并确保其他驱动程序可以与选择的配置接口相匹配。
- *中间的驱动程序*：这是一个厂商指定的驱动程序，直接装载到IOUSBDevice块上。IOUSBCompositeDriver并不作为设备类进行加载，因为设备描述符中的子类字段已经设置为0xFF/0xFF（表示厂商指定的设备），并且一个驱动程序已经正确匹配该设备。
- *右边的驱动程序*：组织结构与左边相同，存在一个装载到IOUSBDevice块的IOUSBComposite-Driver。该复合驱动程序将枚举所有的设备接口，并确保它们可以匹配。在这种情况下，存在两个接口，每个接口都有一个装载的独立驱动程序。

## 8.2.2　加载USB驱动程序

为了在设备插入时自动加载驱动程序，必须配置驱动程序的Info.plist，这部分内容第4章介绍过。从上一节了解到，单个驱动程序可以处理一个USB设备，一个设备也可能存在多个驱动程序，每个驱动程序对应一个存在的接口（函数）。对于USB设备，驱动程序使用该设备的设备描述符与其进行匹配。I/O Kit遵循USB公用类规范设定的驱动程序匹配规则。驱动程序与USB设备的匹配，可以使用下列键组合：

- idVendor & idProduct & bcdDevice;
- idVendor & idProduct;
- idVendor & bDeviceSubClass & bDeviceProtocol（仅当bDeviceClass== 0xff时）;
- idVendor & bDeviceSubClass（仅当bDeviceClass== 0xff时）;
- bDeviceClass & bDeviceSubClass & bDeviceProtocol（仅当bDeviceClass!=0xff时）;
- bDeviceClass & bDeviceSubClass（仅当bDeviceClass!= 0xff时）。

每个键表示设备的设备描述符中的一项。bcdDevice字段用于存储设备的修订号。如果bDeviceClass字段为0xff，则意味着设备类型取决于厂商。Info.plist中，匹配厂商ID、产品ID和修订号（bcdDevice）的一个匹配字典，如代码清单8-1所示。

**代码清单8-1　匹配厂商ID、产品ID和修订号的匹配字典**

```
<key>MyUSBDriver</key>
<dict>
<key>CFBundleIdentifier</key>
    <string>com.osxkernel.MyUSBDriver</string>
    <key>IOClass</key>
```

```
    <string>com_osxkernel_MyUSBDriver</string>
    <key>IOProviderClass</key>
    <string>IOUSBDevice</string>
    <key>bcdDevice</key>
    <integer>1</integer>
    <key>idProduct</key>
    <integer>2323</integer>
    <key>idVendor</key>
    <integer>0001</integer>
</dict>
```

该项必须位于驱动程序Info.plist文件的IOKitPersonalities部分中，才会有效果。

根据上述规则，不匹配的设备将由IOUSBCompositeDriver处理，设备有多个配置时，IOUSBCompositeDriver选择一种设备配置，然后开始匹配设备的接口。可用于匹配设备接口的键如下所示：

- idVendor & idProduct & bcdDevice & bConfigurationValue & bInterfaceNumber;
- idVendor & idProduct & bConfigurationValue & bInterfaceNumber;
- idVendor & bInterfaceSubClass & bInterfaceProtocol（仅当bInterfaceClass== 0xff时）;
- idVendor & bInterfaceSubClass（仅当bInterfaceClass== 0xff时）;
- bInterfaceClass & bInterfaceSubClass & bInterfaceProtocol（仅当bInterfaceClass!= 0xff时）;
- bInterfaceClass & bInterfaceSubClass（仅当bInterfaceClass!= 0xff时）。

**注意** 你无法创建自己的键组合，必须使用上面给出的用于接口或设备的组合之一。但可以将一些个性化信息添加到驱动程序中，每个个性化信息匹配一个不同的组合，但必须是一个有效的组合。

每个键都表示接口描述符中的一个字段。匹配规则根据其特殊性进行排序。例如，最后一个规则匹配接口类和子类，苹果的USB大容量存储器将使用该规则匹配所有遵循该接口的设备，而不考虑厂商或产品ID。苹果的大容量存储器驱动程序的Info.plist如代码清单8-2所示（为了提高可读性，对一些与匹配不相关的键进行了修整）。

**代码清单8-2** 匹配USB接口类和子类的匹配字典

```
<key>IOUSBMassStorageClass6</key>
<dict>
    <key>CFBundleIdentifier</key>
    <string>com.apple.iokit.IOUSBMassStorageClass</string>
    <key>IOClass</key>
    <string>IOUSBMassStorageClass</string>
    <key>IOProviderClass</key>
    <string>IOUSBInterface</string>
    <key>bInterfaceClass</key>
    <integer>8</integer>
    <key>bInterfaceSubClass</key>
```

```
    <integer>6</integer>
</dict>
```

与代码清单8-1中的示例不同，IOProviderClass被指定为IOUSBInterface，因此，IOUSBInterface而不是IOUSBDevice将成为传递给驱动程序start()方法的提供者。

### 8.2.3 USB Prober

在我们开始讲述实际代码之前，先来介绍一个非常有用的工具USB Prober。USB Prober是一款实用程序，捆绑在Xcode发布版中。USB Prober工具如图8-6所示。

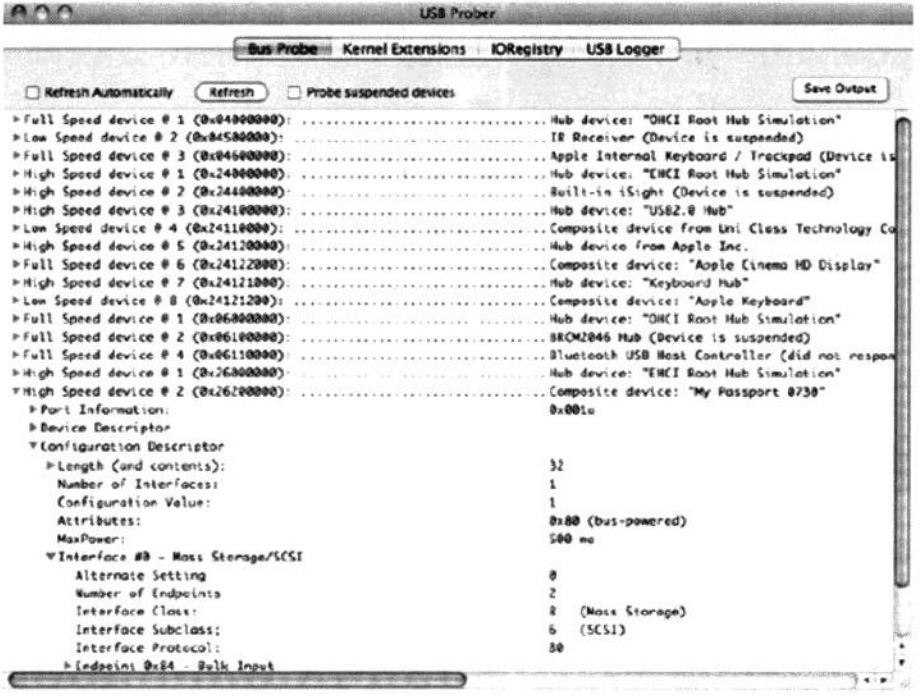



图8-6　USB Prober实用程序

USB Prober可以检测系统上可用的USB总线，查看与每个总线连接的设备层次结构。还可以查看设备、配置、接口和端点描述符。IORegistry表可以查看I/O Registry的IOService平面，因为该平面与USB设备相关，这在开发USB驱动程序时非常有用，可以验证驱动程序是否匹配正确。USB Prober还可以执行与具体USB设备有关的IOUSBFamily追踪，它对一些情况下的驱动程序调试非常有用。这需要进行一些设置，包括从苹果的开发者网站上下载USB调试包。该调试包包含IOUSBFamily的备用版本，该版本提供一些详细记录。只有Mac开发者计划成员才可以访问该调试包。

### 8.2.4 驱动程序示例：USB大容量存储器设备驱动程序

现在，我们将所学内容付诸实践，实现一个简单的USB设备驱动程序，它将在各种事件发生时输出记录消息。我们可以创建一个纯虚拟的驱动程序，但这没有任何意义，因此，我们创建一个真实的USB设备驱动程序，以便观察设备从总线插入和移除时发生的情况，但不会干扰设备操作。I/O Kit面向对象的特性使编写设备驱动程序变得相对轻松。另外，编写USB设备驱动程序与编写固件、PCI驱动程序或第4章中的虚拟IOKitTest驱动程序非常类似。

为了实现该示例，需要一个指状存储器/闪存或外部USB硬盘。无须将其格式化为Mac形式，因为我们不会访问上面的数据。

**警告** 建议在没保存重要数据的Mac上实现本书示例。内核崩溃会破坏文件或操作系统。如果你没有专用Mac，请确保已经备份了数据。

我们的驱动程序将命名为MyFirstUSBDriver，类声明如代码清单8-3所示。

**代码清单8-3 MyFirstUSBDriver.h：MyFirstUSBDriver的类声明**

```
#include <IOKit/usb/IOUSBDevice.h>

class com_osxkernel_MyFirstUSBDriver : public IOService
{
    OSDeclareDefaultStructors(com_osxkernel_MyFirstUSBDriver)

public:
    virtual bool init(OSDictionary *propTable);
    virtual IOService* probe(IOService *provider, SInt32 *score );
    virtual bool attach(IOService *provider);
    virtual void detach(IOService *provider);
    virtual bool start(IOService *provider);
    virtual void stop(IOService *provider);
    virtual bool terminate(IOOptionBits options = 0);
};
```

你会发现，该类的结构与IOKitTest驱动程序几乎相同，只是进行了一些细微的修改，稍后介绍。MyFirstUSBDriver的实现如代码清单8-4所示。

**代码清单8-4 MyFirstUSBDriver.cpp：MyFirstUSBDriver类的实现**

```
#include <IOKit/IOLib.h>
#include <IOKit/usb/IOUSBInterface.h>
#include "MyFirstUSBDriver.h"

OSDefineMetaClassAndStructors(com_osxkernel_MyFirstUSBDriver, IOService)
#define super IOService

void logEndpoint(IOUSBPipe* pipe)
{
```

```
    IOLog("Endpoint #%d ", pipe->GetEndpointNumber());
    IOLog("--> Type: ");
    switch (pipe->GetType())
    {
        case kUSBControl: IOLog("kUSBControl "); break;
        case kUSBBulk: IOLog("kUSBBulk "); break;
        case kUSBIsoc: IOLog("kUSBIsoc "); break;
        case kUSBInterrupt: IOLog("kUSBInterrupt "); break;
    }
    IOLog("--> Direction: ");
    switch (pipe->GetDirection())
    {
        case kUSBOut: IOLog("OUT (kUSBOut) "); break;
        case kUSBIn: IOLog("IN (kUSBIn) "); break;
        case kUSBAnyDirn: IOLog("ANY (Control Pipe) "); break;
    }
    IOLog("maxPacketSize: %d interval: %d\n", pipe->GetMaxPacketSize(), pipe->GetInterval());
}

bool com_osxkernel_MyFirstUSBDriver::init(OSDictionary* propTable)
{
    IOLog("com_osxkernel_MyFirstUSBDriver::init(%p)\n", this);
    return super::init(propTable);
}

IOService* com_osxkernel_MyFirstUSBDriver::probe(IOService* provider, SInt32* score)
{
    IOLog("%s(%p)::probe\n", getName(), this);
    return super::probe(provider, score);
}

bool com_osxkernel_MyFirstUSBDriver::attach(IOService* provider)
{
    IOLog("%s(%p)::attach\n", getName(), this);
    return super::attach(provider);
}

void com_osxkernel_MyFirstUSBDriver::detach(IOService* provider)
{
    IOLog("%s(%p)::detach\n", getName(), this);
    return super::detach(provider);
}

bool com_osxkernel_MyFirstUSBDriver::start(IOService* provider)
{
    IOUSBInterface* interface;
    IOUSBFindEndpointRequest request;
    IOUSBPipe* pipe = NULL;

    IOLog("%s(%p)::start\n", getName(), this);

    if (!super::start(provider))
        return false;
```

```
    interface = OSDynamicCast(IOUSBInterface, provider);
    if (interface == NULL)
    {
        IOLog("%s(%p)::start -> provider not a IOUSBInterface\n", getName(), this);
        return false;
    }

    //大容量存储设备使用两个批量管道，一个用于读，一个用于写

    //查找Bulk In Pipe
    request.type = kUSBBulk;
    request.direction = kUSBIn;
    pipe = interface->FindNextPipe(NULL, &request, true);
    if (pipe)
    {
        logEndpoint(pipe);
        pipe->release();
    }

    //查找Bulk Out Pipe
    request.type = kUSBBulk;
    request.direction = kUSBOut;
    pipe = interface->FindNextPipe(NULL, &request, true);
    if (pipe)
    {
        logEndpoint(pipe);
        pipe->release();
    }
    return true;
}

void com_osxkernel_MyFirstUSBDriver::stop(IOService *provider)
{
    IOLog("%s(%p)::stop\n", getName(), this);
    super::stop(provider);
}

bool com_osxkernel_MyFirstUSBDriver::terminate(IOOptionBits options)
{
    IOLog("%s(%p)::terminate\n", getName(), this);
    return super::terminate(options);
}
```

你也许会看到，除了调用驱动程序各种方法时的记录，该驱动程序中并没有什么逻辑。start()方法将试图查找批量IN和批量OUT端点，并记录这些端点的信息。我们稍后将测试该驱动程序，但首先必须创建一个匹配字典，以便I/O Kit知道何时加载驱动程序。MyFirstUSBDriver的匹配字典如代码清单8-5所示。

**代码清单8-5 MyFirstUSBDriver的匹配字典**

```
<key>IOKitPersonalities</key>
<dict>
```

```
        <key>MyFirstUSBDriver</key>
        <dict>
            <key>bInterfaceClass</key>
            <integer>8</integer>
            <key>bInterfaceSubClass</key>
            <integer>6</integer>
            <key>CFBundleIdentifier</key>
            <string>com.osxkernel.MyFirstUSBDriver</string>
            <key>IOClass</key>
            <string>com_osxkernel_MyFirstUSBDriver</string>
            <key>IOMatchCategory</key>
            <string>com_osxkernel_MyFirstUSBDriver</string>
            <key>IOProviderClass</key>
            <string>IOUSBInterface</string>
        </dict>
    </dict>
```

该匹配字典与代码清单8-2的示例大体相同。它将匹配USB接口，而不是USB设备。我们将bInterfaceClass设置为8，这是大容量存储器设备的类型编码，并将bInterfaceSubClass设置为6，表示使用SCSI命令集，与设备进行通信（这并不意味着驱动器/存储器本身明白SCSI协议，但这可用于通过总线将命令传输给设备，例如，另一个控制器可以将其转换为ATA命令）。

因为苹果的IOUSBMassStorageClass默认匹配的键与我们匹配的键相同，所以需要指定一个匹配字典，以便我们的驱动程序也可以被加载。可以添加IOMatchCategory键来实现。我们将它设置为类名，但它可以是任意字符串。

---

**提示**　打开Xcode中的Info.plist文件，它将默认在属性列表编辑器中打开。如果你希望剪切和粘贴到属性列表，或对其存储格式感到好奇，可以右键点击该文件，选择Open as，然后选择Source Code，它将以XML格式显示该文件。

---

我们还需要修改驱动程序的属性列表文件，即在OSBundleLibraries字典下包括我们的依赖，否则，驱动程序将加载失败。可以使用kextlibs工具查找依赖。可以使用下列代码，将一个依赖添加到libkern及IOUSBFamily中：

```
<key>OSBundleLibraries</key>
<dict>
        <key>com.apple.iokit.IOUSBFamily</key>
        <string>4.1.8</string>
        <key>com.apple.kernel.libkern</key>
        <string>6.0</string>
</dict>
```

现在，可以加载驱动程序，或通过I/O Kit加载驱动程序。USB设备插入时自动加载的驱动程序必须位于/System/Library/Extensions目录中，这是所有KEXT的标准位置。

做完这些工作后，插入设备。但在此之前，可以打开Console应用程序，并从记录列表中选择kernel.log。插入兼容的设备之后，会输出以下与插入操作相关的记录项：

```
Jun 16 22:37:56 macbook kernel[0]: com_osxkernel_MyFirstUSBDriver::init(0x1361f900)
Jun 16 22:37:56 macbook kernel[0]: com_osxkernel_MyFirstUSBDriver(0x1361f900)::attach
Jun 16 22:37:56 macbook kernel[0]: com_osxkernel_MyFirstUSBDriver(0x1361f900)::probe
Jun 16 22:37:56 macbook kernel[0]: com_osxkernel_MyFirstUSBDriver(0x1361f900)::detach
Jun 16 22:37:56 macbook kernel[0]: com_osxkernel_MyFirstUSBDriver(0x1361f900)::attach
Jun 16 22:37:56 macbook kernel[0]: com_osxkernel_MyFirstUSBDriver(0x1361f900)::start
Jun 16 22:37:56 macbook kernel[0]: Endpoint #4 --> Type: kUSBBulk --> Direction: IN (kUSBIn)
maxPacketSize: 512 interval: 0
Jun 16 22:37:56 macbook kernel[0]: Endpoint #3 --> Type: kUSBBulk --> Direction: OUT
(kUSBOut) maxPacketSize: 512 interval: 0
```

前5个调用是匹配流程的一部分。第一个方法attach()将你的驱动程序连接到I/O Registry的IOService平面，在这种情况下，我们将作为IOUSBInterface块的客户端装载，而IOUSBInterface块同样也是刚插入的IOUSBDevice的客户端。第4章提到，该检测方法用于活动的匹配，可以进一步询问设备或接口，以确定我们是否与它匹配。I/O Kit然后调用detach()，在检查完所有可能的匹配后，决定加载哪一个驱动程序。通常并不建议在attach()中分配任何资源，因为该函数会被调用多次。通常，无须重载attach()或detach()，因为IOService提供的默认函数已经足够。在I/O Kit选择了驱动程序之后——它在我们的例子中肯定会执行（因为我们指定了唯一的IOMatchCategory），我们将再次调用attach()方法，最后是驱动程序的start()方法。现在可以使用USB Prober验证驱动程序在层次结构中的位置，如图8-7所示。

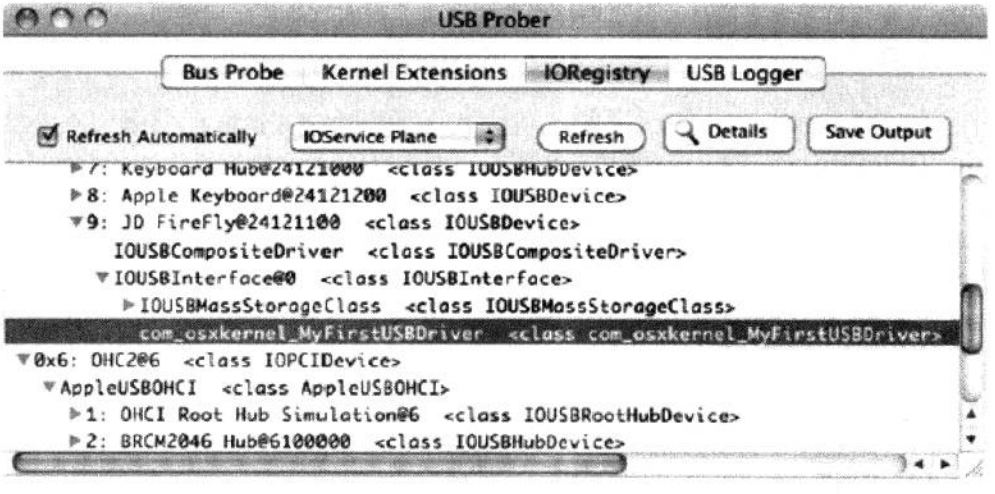



图8-7 USB Prober显示装载到IOService面板的com_osxkernel_MyFirstUSBDriver

你会发现，我们与IOUSBMassStorageClass驱动程序一起，附加到存储设备的IOUSBInterface，并且USB设备本身由复合驱动程序IOUSBCompositeDriver管理。

### 8.2.5　驱动程序启动

MyFirstUSBDriver中start()方法的实现方式并不常见，因为IOUSBMassStorageClass驱动程序也在管理该接口，并且我们不希望干扰它使用USB接口。通常，我们会在start()方法中执行一个基本的合理性检查，确保能够获得预期类型的提供者。但什么都无法阻止你编写接受和处理多种类型提供者的驱动程序，例如，IOUSBDevice和IOUSBInterface，甚至是IOPCIDevice提供者。

以下是USB驱动程序在其start()方法中通常必须执行的步骤。

- 验证传递给我们的IOService提供者对象是否为预期的类型。这项验证借助于OSDynamicCast()宏进行，该宏可以运行I/O Kit的运行时类型识别系统，转换成功时返回该对象的指针，否则返回NULL。
- 存储提供者指针，供以后使用。
- 调用open(IOService* forClient, …)方法，尝试打开提供者。
- 如果你的驱动程序正处理IOUSBDevice，则必须设置该设备的配置。对基于IOUSBInterface的驱动程序，IOUSBCompositeDriver通常处理该操作。可以使用IOUSBDevice::SetConfiguration()，对配置进行设置。
- 查找并验证接口，该接口可用于其提供者为IOUSBDevice的驱动程序，并搜索驱动程序所需的适当端点。更多信息请参见8.2.7节。
- 询问设备的状态信息，通过向设备发送控制请求，执行所需的设备配置。
- 分配你可能需要的某些驱动程序特有的资源，例如驱动程序所需的I/O缓冲区或辅助类。
- 如果驱动程序是一个块，并为其他驱动程序提供服务，则需要分配并注册这些服务。例如，就IOUSBMassStorageClass而言，它将为接口提供的每个逻辑单元分配一个IOSCSILogicalUnitNub对象，并对每个对象调用一个继承自IOService的registerService()方法。该方法可以确保对每个IOSCSILogicalUnitNub对象进行匹配。
- 如果一切成功，则start()应该返回true。如果返回false，显然没有加载驱动程序，I/O Kit将试图加载一个新的驱动程序，如果有的话，则可能是“丢失的”和之前得分较低的那个。要知道，start()返回false时不会调用stop()。

---

**提示**　卸载MyFirstUSBDriver，使用如下命令即可：sudo rm -rf /System/Library/Extensions/MyFirstUSBDriver.kext

---

### 8.2.6　处理设备移除

USB设备和驱动程序必须可以随时处理设备的移除。IOUSBController检测到设备移除时，会将该信息递归地传播到驱动程序栈。驱动程序的第一个通知通过调用其terminate()方法完成。接下来的调用序列是装载了MyFirstUSBDriver的大容量存储器设备拔出的结果：

```
Jun 16 22:58:46 macbook kernel[0]: com_osxkernel_MyFirstUSBDriver(0xb4e6100)::terminate
Jun 16 22:58:46 macbook kernel[0]: com_osxkernel_MyFirstUSBDriver(0xb4e6100)::stop
Jun 16 22:58:46 macbook kernel[0]: com_osxkernel_MyFirstUSBDriver(0xb4e6100)::detach
```

任何未完成的I/O都可以使用IOUSBPipe::Abort()取消，并且可以在调用terminate()方法时实现，或在驱动程序覆写的willTerminate()以及didTerminate()方法中实现。

移除流程的下一步是调用驱动程序的stop()方法，这应该是start()方法执行行为的逆过程。在此之后，调用detach()及最终的free()，清除所有的保留资源。

如果驱动程序由一个用户应用程序打开，例如通过IOUSerClient打开，则它在应用程序释放该设备的索引之前不会被释放（不会调用free()方法）。如果设备刚好在此时重新插入，应用程序并不能继续使用该设备，因为每次插入设备时都会创建一个新的驱动程序实例。应用程序可以使用通知对此进行处理，如第5章所述。

### 8.2.7 枚举接口

在USB驱动程序的start()方法中，通常需要查找和配置设备使用的端点和接口。如果你的驱动程序基于IOUSBDevice提供者，那么很可能需要检索驱动程序使用的一个或更多接口。这能使用IOUSBDevice::FindNextInterfae()方法实现：

```
virtual IOUSBInterface* FindNextInterface(IOUSBInterface* current,
                                          IOUSBFindInterfaceRequest* request);
```

我们可以指定第一个参数，从现有的IOUSBInterface实例开始检索，并忽略它前面的接口。可以指定为NULL，这样就会从第一个接口开始检索。

第二个参数是IOUSBFindInterfaceRequest结构体：

```
typedef struct {
    UInt16 bInterfaceClass;
    UInt16 bInterfaceSubClass;
    UInt16 bInterfaceProtocol;
    UInt16 bAlternateSetting;
} IOUSBFindInterfaceRequest;
```

为了查找一个接口，可以将所需的接口属性填写到IOUSBFindInterfaceRequest结构体中。

- 可以填写bInterfaceClass和bInterfaceSubClass，以查找指定类及子类的接口。其值对应表8-3中的代码值。IOUSBFamily源代码发布版中的USBSpec.h头文件定义了一些符号常量，如kUSBMassStorageInterfaceClass或kUSBPrintingClass。
- bInterfaceProtocol指定接口使用的协议。如果未指定类和子类，该字段将毫无意义。例如，HID可以将协议指定为kHIDKeyboardInterfaceProtocol或kHIDMouseInterfaceProtocol。
- 一个接口可以拥有一个备用版本，使用不同的端点集，因此可以设置bAlternateSetting字段以请求所需的特定接口。

无关紧要的字段可以设置为kIOUSBFindInterfaceDontCare。将每个字段都设置为该值，则无论怎样最终都将返回下一个接口。

代码清单8-6摘录自苹果USB以太网驱动程序，该驱动程序使用FindNextInterface()方法，检索支持以太网的USB设备（IOUSBDevice）接口。

**代码清单8-6　搜索接口的IOUSBDevice（在USBCDCEthernet.cpp中实现）**

```
IOUSBFindInterfaceRequest              req;
IOUSBInterface*                        fCommInterface = NULL;

req.bInterfaceClass =    kUSBCommClass;
req.bInterfaceSubClass = kEthernetControlModel;
req.bInterfaceProtocol = kIOUSBFindInterfaceDontCare;
req.bAlternateSetting =  kIOUSBFindInterfaceDontCare;

fCommInterface = fpDevice->FindNextInterface(NULL, &req);
if (!fCommInterface)
{
     //未找到
     …
}
```

## 8.2.8　枚举端点

接口本身并不执行任何有用的操作，因此，驱动程序检索到正确的接口之后必须枚举接口的端点，以用于实际的I/O。该枚举/检索处理类似于查找接口，采用IOUSBInterface::FindNextPipe()方法实现：

```
virtual IOUSBPipe *FindNextPipe(IOUSBPipe *current, IOUSBFindEndpointRequest *request);
virtual IOUSBPipe* FindNextPipe(IOUSBPipe *current, IOUSBFindEndpointRequest *request,
        bool withRetain);
```

第一个参数如果不是NULL，则通知该方法忽略它前面的管道。第二个参数是一个指向IOUSBFindEndpointRequest的指针：

```
typedef struct {
    UInt8 type;
    UInt8 direction;
    UInt16 maxPacketSize;
    UInt8 interval;
} IOUSBFindEndpointRequest;
```

- type字段可以是kUSBControl、kUSBIsoc、kUSBBulk、kUSBInterrupt或kUSBAnyType。
- direction字段必须设置为kUSBOut、kUSBIn或kUSBAnyDirn。
- maxPacketSize字段是端点0支持的最大包长度（单位是字节），应该是8、16、32或64。无关时可以设置为0。
- interval字段用于搜索有特定轮询时间间隔的端点。轮询时间间隔仅用于同步端点和中断端点。

代码清单8-7展示了苹果USB以太网驱动程序使用FindNextPipe()方法枚举端点。

**代码清单8-7 枚举接口的IOUSBPipe实例（在USBCDCEthernet.cpp中实现）**

```
IOUSBFindEndpointRequest        epReq;          //栈上的端点请求结构
…
//打开所有端点

epReq.type = kUSBBulk;
epReq.direction = kUSBIn;
epReq.maxPacketSize     = 0;
epReq.interval = 0;
fInPipe = fDataInterface->FindNextPipe(0, &epReq);
if (!fInPipe)
{
    …
    return false;
}
…
epReq.direction = kUSBOut;
fOutPipe = fDataInterface->FindNextPipe(0, &epReq);
if (!fOutPipe)
{
    …
    return false;
}
fOutPacketSize = epReq.maxPacketSize;
…
//中断管道——通信接口

epReq.type = kUSBInterrupt;
epReq.direction = kUSBIn;
fCommPipe = fCommInterface->FindNextPipe(0, &epReq);
if (!fCommPipe)
{
    …
}
```

代码清单8-7中的驱动程序是一个USB以太网驱动程序。该驱动程序在操作中使用了3个端点。第一个是批量IN端点，用于读取来自设备的网络数据。第二个是批量OUT管道，用于将包传输给设备。最后一个是中断IN端点，用于通知网包的到达以及其他事件。下一节将讲述如何使用端点执行I/O。

## 8.2.9 执行设备请求

设备请求是对USB设备的默认双向默认控制管道0的I/O请求，通常用于设备配置及访问设备寄存器。设备请求分为以下3类。

- 标准USB请求：这是所有设备都实现的标准请求。标准设备请求的一个例子是查询设备状态。标准请求的一个符号常量列表可以在USBSpec.h中找到。
- 类特定请求：这是特定于某一类设备的请求。例如，以太网设备可以提供一些请求，配置以太网的相关参数。
- 厂商特定请求。

为了执行设备请求，IOUSBDevice和IOUSBInterface都提供了一个专门的DeviceRequest()便捷方法，在底层使用IOUSBPipe对象（表示默认管道）传输请求。如果你愿意，可以枚举端点0的IOUSBPipe实例，还可以直接使用IOUSBPipe实例。DeviceRequest()方法的声明如下所示：

```
DeviceRequest(IOUSBDevRequest *request, UInt32 noDataTimeout,
              UInt32 completionTimeout, IOUSBCompletion *completion);
DeviceRequest(IOUSBDevRequestDesc *, UInt32 noDataTimeout,
              UInt32 completionTimeout, IOUSBCompletion *completion);
```

为了发送请求，必须创建一个IOUSBDevRequest或IOUSBDeviceRequestDesc结构体，并填写适当的字段：

```
typedef struct {
    UInt8 bmRequestType;
    UInt8 bRequest;
    UInt16 wValue;
    UInt16 wIndex;
    UInt16 wLength;
    void *pData;
    UInt32 wLenDone;
} IOUSBDevRequest;
```

```
typedef struct {
    UInt8 bmRequestType;
    UInt8 bRequest;
    UInt16 wValue;
    UInt16 wIndex;
    UInt16 wLength;
    IOMemoryDescriptor *pData;
    UInt32 wLenDone;
} IOUSBDevRequestDesc;
```

- bmRequestType字段是一个复合字段，指定请求类型、方向、类型和接受者。该字段可使用USBmakebmRequestType(direction,type,recipient)宏生成，该宏的参数如下：
  - direction可以是kUSBIn、kUSBOut或kUSBNone；
  - type可以是kUSBStandard、kUSBClass或kUSBVendor；
  - recpient可以是kUSBInterface、kUSBEndpoint或kUSBDevice。
- bRequest字段是一个8位值，确定要执行的请求。
- wValue和wIndex用于传递参数及请求。其含义取决于请求类型。对于接口和端点请求，wIndex指定请求所在端点/接口的索引编号。可以调用IOUSBPipe::GetEndpointNumber()或IOUSBInterface->GetInterfaceNumber()来获取索引编号。
- wLength字段表示pData字段的字节数。
- pData字段可以是一个指向内存缓冲区的指针，也可以是IOMemoryDescriptor。当该请求并不需要其他数据时，pData指针可以设置为NULL。读取缓冲区还是写入缓冲区取决于请求的方向。如果使用IOMemoryDescriptor，则首先应该对其调用prepare()以保证该内存被换入，并保持至请求完成。如果使用一个内存描述符，则内存可能来自用户空间。如果使用void*变体，则该指针一定位于内核的虚拟地址空间中。
- wLenDone字段不能填写，因为它用于返回实际传输的字节数。

除了request参数，DeviceRequest()方法还包含另外3个参数。

- noDataTimeout：这是没有发送/接收的数据时，放弃该请求之前等待的超时时间，单位是毫秒。
- completionTimeout：这是为包含数据的整个命令而指定的一个超时值，单位也是毫秒。
- completion：这是可选项，如果指定它，则可以异步执行请求，通常会指定该值以避免阻塞调用线程。本章后面会详细讨论异步请求。

看一个发送设备请求的例子，同样使用苹果USB以太网驱动程序。代码清单8-8中的代码由驱动程序通过一个周期性定时器进行调用，并用来获取以太网设备的统计信息和状态信息，如碰撞、丢弃的包、到达的包等。

**代码清单8-8 从以太网设备下载统计信息的设备请求（USBCDCEthernet.cpp）**

```
STREQ = (IOUSBDevRequest*)IOMalloc(sizeof(IOUSBDevRequest));
if (!STREQ)
{
    ...
} else {
    bzero(STREQ, sizeof(IOUSBDevRequest));
    //现在构建统计信息请求
   STREQ->bmRequestType = USBmakebmRequestType(kUSBOut, kUSBClass, kUSBInterface);
   STREQ->bRequest = kGet_Ethernet_Statistics;
   STREQ->wValue = currStat;
   STREQ->wIndex = fCommInterfaceNumber;
   STREQ->wLength = 4;
   STREQ->pData = &fStatValue;

   fStatsCompletionInfo.parameter = STREQ;

   rc = fpDevice->DeviceRequest(STREQ, &fStatsCompletionInfo);
   if (rc != kIOReturnSuccess)
   {
   ...
       IOFree(STREQ, sizeof(IOUSBDevRequest));
   } else {
      fStatInProgress = true;
   }
 }
```

代码清单8-8中的请求异步执行。因为IOUSBDevRequest结构体必须保持至该请求完成，该结构体不能在栈上分配，尽管这对同步请求来说并没有问题。代码清单8-8中执行的请求指向一个特定的接口，是一个类特定请求，这意味着该请求在所有的接口上都采用相同的类型代码执行同样的工作。该请求的wValue字段是一个索引编号，指定应该传输的统计信息。

### 控制请求

上一节讨论的设备请求是默认控制管道（0）的I/O。DeviceRequest()方法无法用于默认之外的控制端点。如果我们希望执行另一个控制端点的请求，则必须使用IOUSBPipe::ControlRequest()方法。可用的ControlRequest()方法有4个：

```
virtual IOReturn ControlRequest(IOUSBDevRequestDesc* request,
        IOUSBCompletion* completion = 0);
virtual IOReturn ControlRequest(IOUSBDevRequest* request, IOUSBCompletion* completion = 0);
virtual IOReturn ControlRequest(IOUSBDevRequestDesc* request,
        UInt32 noDataTimeout,
        UInt32 completionTimeout,
        IOUSBCompletion* completion = 0);
virtual IOReturn ControlRequest(IOUSBDevRequest* request,
        UInt32 noDataTimeout,
        UInt32 completionTimeout,
```

```
        IOUSBCompletion* completion = 0);
```

前两个方法使用的参数与前面讨论的DeviceRequest()方法完全相同。后两个方法还支持noDataTimeout和completionTimeout参数。

### 8.2.10　执行批量端点和中断端点的I/O

发送和接收数据借助于表示端点的IOUSBPipe类执行。IOUSBPipe类提供了一个简单的接口来执行I/O，这很容易让人想起用户空间执行文件I/O的方式。USB并不直接使用DMA，尽管主机控制器使用DMA传输数据，但具体细节我们不必考虑。这还意味着，如果内存的物理地址是连续的，就不关注内存对齐是在正确的地址范围内，还是将内存地址转换为物理地址。我们还可以执行用户空间缓冲区的I/O。

IOUSBPipe类支持所有端点类型的I/O：控制、批量、中断和同步。

执行批量和中断I/O使用的是Read()和Write()方法：

```
virtual IOReturn Read(IOMemoryDescriptor* buffer,
                      UInt32 noDataTimeout,
                      UInt32 completionTimeout,
                      IOByteCount reqCount,
                      IOUSBCompletion* completion = 0,
                      IOByteCount* bytesRead = 0);

virtual IOReturn Write(IOMemoryDescriptor* buffer,
                       UInt32 noDataTimeout,
                       UInt32 completionTimeout,
                       IOByteCount reqCount,
                       IOUSBCompletion* completion = 0);
```

- buffer是一个IOMemoryDescriptor，包含对数据进行读/写的缓冲区。内存描述符应该调用其prepare()方法，以确保内存被换入并钉住。内存可以是内核或用户任务的地址空间。
- noDataTimeout参数在认为请求不成功之前，指定等待总线上数据传输的时间长度，单位是毫秒。
- completionTimeout是在认为整个请求不成功之前预留的完成时间，单位是毫秒。
- reqCount是应该读/写的数据量，单位是字节。必须小于或等于IOMemoryDescriptor::getLength()返回的缓冲区长度。
- completion参数是IOUSBCompletion类型的结构体，用于异步请求。该参数可以指定为NULL，以异步执行请求，这种情况下，调用将阻塞，直至请求完成或超时。稍后将介绍异步I/O。
- 对于Read()方法，bytesRead将返回读取的字节数。返回的字节数可能会小于请求的字节数。该值仅在异步请求时设置。

代码清单8-9是Read()和Write()方法的调用示例。

**代码清单8-9　批量管道的异步Read()和Write()示例**

```
UInt32 bytesRead;
IOMemoryDescriptor* readBuffer;
```

```
IOMemoryDescriptor* writeBuffer;
…
if (myBulkPipeIn->Read(readBuffer, 1000, 5000,
                       readBuffer->GetLength(), 0, &bytesRead) != kIOReturnSuccess)
{
     //处理错误
}
else
    IOLog("We read: %u bytes\n", bytesRead);

if (myBulkPipeOut->Write(writeBuffer, 1000, 5000,
                         writeBuffer->GetLength()) != kIOReturnSuccess)
{
    //处理错误
}
```

由于我们没有指定这两个方法的completion参数，所以它们都将异步执行，这意味着该请求将在这两个方法将控制返回给我们时才会执行。回想一下，管道为单向的，默认控制管道除外，因此IN和OUT请求在两个单独的管道上执行。

**注意** Read()和Write()方法存在另一个重载集合，它并不接受reqCount参数，而是使用IOMemoryDescriptor的GetLength()方法。这些方法现在已经废弃，不应该再使用。



代码清单8-9中的例子还可以用于中断端点。中断端点的处理并不需要特定的编程接口。I/O的处理方式与批量端点相同。不同点在于行为。中断端点提供有界延迟，并且主机控制器保证对数据进行设备轮询的频率不会低于端点描述符中请求的频率。最低轮询时间间隔是125微秒。中断传输使用保留的带宽，这可以保证总线上大量的任务请求通过或位于事件中。不同于批量传输，中断传输并不适用于传输大量数据，仅限于低速、全速和高速分别对应的8、64、1024字节。请注意，中断端点与系统中断毫不相干。中断端点的I/O在普通的内核线程中执行。

### 1. 处理错误和管道STALL

端点因为错误而无法传输或接收数据时，主机或设备可以设置HALT位。与此种状态的端点或有错误的端点通信，将返回STALL握手包。只有解决错误，I/O才可以在端点上继续执行。IOUSBPipe类提供2个方法，清除管道STALL，恢复I/O。

```
virtual IOReturn ClearStall(void);
virtual IOReturn ClearPipeStall(bool withDeviceRequest);
```

第二个方法清除控制器上的错误（触发位），但当withDeviceRequest是false时，它不会将设备请求发送到端点。这两个方法都会完成未完成的I/O，并给出返回码kIOUSBTransaction- Returned。

### 2. 同步I/O

同步传输在本质上是连续的，适用于音频、视频等设备，其信息是连续流，并且需要保证带宽及有界延迟。数据完整性可以使用CRC进行验证，但破坏的数据不会自动重新发送。设备所需的带宽可以在同步端点描述符中指定。如果主机控制器无法保证足够的带宽支持设备，如另一个

设备已经占用了总线上保留的带宽时，该设备可能会无法正常工作。如果设备操作占用较少的带宽，则可以定义具有更保守带宽需求的备用接口描述符。同步传输的最大负载如下。

- 高速设备的最大包长度是1024字节。
- 全速设备的最大包长度是1023字节。
- 低速设备不支持同步传输。

同步传输使用微帧的概念。一个微帧是125毫秒长。对于高速设备，每个微帧最多可以传输3个包，最大数据速率是3 × 1024 × 8000微帧/秒=24 MB/s。这比批量端点上可能的最大带宽稍低一点。

微帧由IOUSBIsocFrame结构体表示：

```
typedef struct IOUSBIsocFrame {
    IOReturn            frStatus;
    UInt16              frReqCount;
    UInt16              frActCount;
} IOUSBIsocFrame;
```

该结构体描述了每个微帧中应该向/从I/O缓冲区传输/接收多少字节的数据。frReqCount字段是请求的字节数，而frActCount是实际传输的字节数。该结构还包含一个状态字段。

读写同步端点的方法类似于中断端点和批量端点。

```
virtual IOReturn Read(IOMemoryDescriptor* buffer, UInt64 frameStart, UInt32 numFrames,
                      IOUSBIsocFrame* frameList, IOUSBIsocCompletion* completion = 0);
virtual IOReturn Write(IOMemoryDescriptor* buffer, UInt64 frameStart, UInt32 numFrames,
                       IOUSBIsocFrame *frameList, IOUSBIsocCompletion * completion = 0);
```

下面是这两个方法的参数。

- buffer参数是一个几乎连续的缓冲区，包含传输的数据。内存描述符应该调用其prepare()方法，确保内存在传输期间被换入并钉住。然而它对内存并没有特定要求，它可以是用户空间或内核内存。
- frameStart参数指定开始的USB帧索引。一个USB帧对应8个微帧。
- numFrames参数是frameList数组中所包含微帧描述符的数量。
- frameList参数是一个指向IOUSBIsocFrame结构数组的指针。
- 可选的completion结构体。如果指定的话，将异步执行传输。

### 3. 异步请求

通常需要异步执行对USB设备的请求，例如，执行对硬盘的大批量请求时。不是阻塞调用者线程，而是由USB控制器处理请求，并在请求完成时，通过回调方法通知我们。

为此，必须为Read()、Write()、DeviceRequest()及ControlRequest()方法提供一个IOUSBCompletion结构体：

```
typedef struct IOUSBCompletion {
    void* target;
    IOUSBCompletionAction action;
    void* parameter;
} IOUSBCompletion;
```

- target字段是一个指针，包含用户定义的数据。通常将该指针传递给发送请求的类，以便可以将该指针转换为完回函数中的原始类。
- action字段是实际的回调，应该是一个指向函数的指针，该函数匹配IOUSBCompletionAction原型。该函数将在请求完成之后调用。
- parameter字段可以传送一个额外的参数，传递给完成函数。

IOUSBCompletionAction回调的原型如下所示：

```
typedef void ( *IOUSBCompletionAction)(void* target, void* parameter,
                                       IOReturn status, UInt32 bufferSizeRemaining);
```

可以看到，IOUSBCompletion结构体的target和parameter字段将直接传递给回调。该回调还将获取传输的status，如果请求没有全部完成，bufferSizeRemaining字段将包含留待传输的字节数。

异步请求在IOUSBFamily工作环线程上完成，这意味着，从你的驱动程序访问回调中的数据，必须确保该访问已经被正确同步。

一般说来，USB驱动程序从不在主中断上下文中操作，同步的Read()和Write()方法的低延迟版本除外，这使异步的同步I/O可以在主中断时刻调用完成回调。这种情况下需要多加注意，避免调用可能会阻塞的代码。应该尽量少用低延迟同步I/O，即使音频和视频驱动程序，通常也不使用。



## 8.3 小结

- 写作本书时，USB规范有3个主要版本，最新的是USB 3.0。尽管在PC市场上迅速兴起，但USB 3.0还未被苹果采纳，在写作本书时苹果仅支持USB 2.0。
- USB使用共享的总线拓扑，一个主机最多可以控制127个设备。主机控制器是总线的“主人”，控制总线上所有的任务。设备在未经主机许可之前无法使用总线。
- USB设备由层次结构的描述符进行描述，包含类、类型、功能及设备需求等信息。
- USB设备可以包含0个或多个接口，这些接口是端点组。一个接口通常代表一个逻辑设备功能，如打印机或扫描仪。
- 端点有4种类型：控制、批量、中断和同步。控制传输用于设备配置和控制。批量端点用于应用程序，如硬盘或网络设备。中断端点提供有界延迟，但只能进行少量传输。同步端点适用于视频或音频应用程序，这些应用程序需要有保证的和可预见的带宽，以及低延迟。
- IOUSBFamily处理内核中的USB支持。该族支持一些常见的USB控制器。与驱动程序开发相关的3个主要类是IOUSBDevice、IOUSBInterface和IOUSBPipe。USB驱动程序可以使用IOUSBDevice或IOUSBInterface作为其提供者。
- IOUSBPipe提供端点的一个抽象，其方法可以处理所有4种端点，支持同步和异步I/O。

# 第9章 PCI Express和Thunderbolt

PCI（Peripheral Component Interconnect，外围组件互连）是20世纪90年代初英特尔公司开发的一种高速总线，用来替代又旧又慢的EISA、ISA、MCA和VESA等总线技术。术语PCI常常用于描述基于原始PCI规范的技术族。本章提到PCI时指的是基于PCI技术的共同点，即PCI Express、Thunderbolt和应用范围较小的ExpressCard。大多数人将PCI与嵌入到计算机中的主板联系到一起，但值得注意的是，PCI对许多计算机系统（即使是没有PCI插槽的iMac等计算机系统）来说非常基本，这些计算机系统有内部PCI总线，将CPU连接到USB、FireWire及SATA控制器。目前基于PCI的升级（如Thunderbolt）使PCI总线可以在计算机外部扩展，其方式与USB和FireWire非常类似。

PCI已经得到广泛认可，解决了较旧总线技术中存在的许多问题，例如，消除了在扩展卡上配置跳线的需求，因为内存区和中断等资源由系统BIOS或操作系统本身自动配置。

PCI由PCI-X和PCI-X 2.0标准扩展而来，支持64位总线宽度，而不是旧式PCI的32位总线宽度。PCI Express（PCIe）标准替代PCI-X标准之后，PCI-X标准就已被废弃。但不同于PCI-X，PCIe使用的是基于分组的串行协议，而不是其先前版本的并行接口特性。PCIe支持总线上的设备使用专用带宽，而不是与同一总线上的其他设备共享总线带宽。虽然从电学和物理角度来看，PCIe和PCI有很大不同，但从软件角度来看，它们都后向兼容，因此，驱动程序仅需进行细微的（或不需要）修改，就可以支持较新的标准。

前面提到，有很多与PCI相关的标准。我们只讨论当前苹果销售的技术，包括PCIe、Thunderbolt和ExpressCard。Thunderbolt出现在2011年或更新的Mac中。Thunderbolt和ExpressCard基于PCIe技术，连接到PCI主桥。但在所有的Mac上，ExpressCard正在逐步淘汰，而Thunderbolt则倍受青睐，现在仅有17寸MacBook Pro使用ExpressCard。Xserve停产之后，Mac Pro是目前唯一拥有根本上可访问的PCI Express插槽的Mac。

本章首先讲述应用于当代Mac的各种PCI技术。我们重点介绍从软件角度理解起重要作用的部分，以及构建一个功能齐全的PCI设备驱动程序所必需的部分。例如，作为程序员，我们无须关注PCI如何在电学层面工作。本章的第二部分重点介绍在I/O Kit中如何与基于PCI的设备交互，如何匹配和设置这些设备，如何读取寄存器以及处理设备的移除。我们还将讲述如何处理中断及执行DMA（Direct Memory Access，直接内存访问），这是基于PCI的驱动程序执行的两个典型任务。

### 1. PCI Express

PCIe用来替代PCI和PCI-X及AGP（Accelerated Graphics Port，加速图形端口），是显卡为了解决PCI-X无法提供更高带宽而采用的权宜之计。PCIe使用单向、点对点的连接，称为“通道”。这种方案避免了PCI和PCI-X的共享总线问题，尽管系统设计者将每个物理PCI插槽置于自己的专用总线上，可以稍微缓解这种问题。然而，PCIe比它先前的版本要快得多。

到目前为止，已经发布了PCIe标准的三个版本。第二代将单个PCIe通道的可能带宽从250 MB/s翻倍到500 MB/s。第3版的带宽再次翻倍，每个通道可以处理的带宽高达 1 GB/s。PCIe通常使用1x、4x、8x、16x和32x的通道配置，尽管最后一个配置并不常见，特别是对物理插槽来说。显卡/GPU的插槽通常是16x通道宽，因为它们需要大量的带宽。Mac Pro的最新版本遵循PCIe 2.0标准。Mac Pro（5,1）的最新版本有4个16x通道插槽，但只有插槽1和2可以按16x操作，而插槽3和4按4x操作。

### 2. Thunderbolt

Thunderbolt是一种相对较新的技术，最初由英特尔公司开发，后来苹果公司采用了该技术。苹果公司是目前唯一发行具有Thunderbolt功能计算机的厂商。尽管设备的可用性有限，但一些公司，包括Blackmagic Design、乔鼎资讯和西部数据，都已经宣布支持该技术。Thunderbolt是一个外部扩展接口，使PCIe和DisplayPort 1.1可以作为通道在同样的线缆上传输数据。一条线缆可以拥有两个双向10 Gbps数据的通道，这相当于每个线缆的总带宽40 Gbps。这些通道彼此独立，不可以将它们之间的带宽相加。Thunderbolt还可以为连接到总线上的设备提供高达10瓦的电力。该线缆使用Mini DisplayPort连接器，两端都一样。

Thunderbolt的当前规范最多允许6个设备串联。后续版本将采用一个类似于USB的树型拓扑结构。但不同于USB，Thunderbolt与FireWire一样，允许主机对主机的连接。苹果还支持以目标磁盘模式使用Thunderbolt，可以从连接到Thunderbolt的存储器启动操作系统。由于Thunderbolt设备直接与PCIe主机系统通信，所以少量修改硬件便可以更新现有设备，使它支持Thunderbolt（不考虑外部情况和可能需要外部电源的情况）。在软件方面，几乎无须修改（设备仍然由`IOPCIFamily`管理），但其中一个要求是，驱动程序必须支持动态卸载。

Thunderbolt使Mac Mini、iMac或MacBook系列计算机都可以访问高速存储器和存储区域网络，以及高带宽未压缩的视频捕捉，这部分预留给高端Mac Pro和Xserve。

### 3. ExpressCard

ExpressCard是MacBook Pro系列中较古老的扩展接口。ExpressCard正在逐步淘汰，而Thunderbolt却倍受青睐，但写作本书时带有ExpressCard和Thunderbolt端口的笔记本电脑都可用。ExpressCard是PCMCIA的现代版本，基于PCIe。最新标准支持高达5 Gbps的传输速度。

## 配置空间寄存器

所有的PCI设备（包括桥）都有一组寄存器，称为配置空间。传统PCI设备的配置空间最小是256字节，而对基于PCI-X 2.0和PCI Express技术的设备来说，配置空间可长达4096字节，称为扩

展配置空间。配置空间寄存器的前48字节如图9-1所示。

图9-1　标准PCI配置空间寄存器

必需的寄存器显示为灰色，其他的寄存器是可选的。前48个字节被标准化了，你会发现，不管设备基于PCI、PCI-X还是PCIe，其结构均相同。许多寄存器已经不再适用，因为PCI Express基于点对点，并不使用共享总线。

我们详细讲述一下图9-1中的强制性寄存器。

- 厂商ID：包含一个16位标识符，唯一标识每个硬件厂商。每个硬件厂商的厂商ID均由PCI-SIG( special interest group，特别兴趣小组 )分配。例如，苹果分配到的厂商ID为0x106b。操作系统通常使用厂商ID和设备ID组合来确定为设备加载哪一个驱动程序。0xffff是一个无效的厂商ID。
- 设备ID：也是16位宽。不同于厂商ID，设备ID可以由厂商分配，不在中央寄存器中维护。
- 类代码：持有设备类型分类的一个24位寄存器。前8位保存基本类型。基本类型示例包括未分类的（0x0）、大容量存储器控制器（0x1）、网络控制器（0x2）、显示控制器（0x3）等。接下来的8位保存子类。例如，如果基类是显示控制器，子类则可能是VGA（0x0）、XGA（0x1）或其他（0x80）。余下的8位用于指定设备的程序接口（寄存器级的接口），如果存在多个接口的话。USB控制器用来验证这些接口是否遵守UHCI、OHCI、EHCI或XCHI接口，这些是寄存器级的规范，确定驱动程序与设备交互的方式。
- 子系统厂商/设备ID：与厂商ID和设备ID遵循同样的规则和分配。当许多不同的厂商出售的产品使用同样的芯片（OEM）时，子系统ID可以用来标识芯片。主要的例子是Nvidia和ATI。它们都制造GPU芯片，然后由第三方厂商用来制造最终的产品。这些产品的PCI配置空间包含第三方的厂商ID和设备ID，但使用Nvidia或ATI作为厂商ID，将设备ID作为子系统设备ID。这使ATI和Nvidia的驱动程序都可以使用，尽管它们并没有直接制造主板。
- 基地址0~5：最多包含6个I/O区，可以是I/O端口或内存区。后者更为常见。我们稍后将详细讨论I/O区。基地址常常缩写为BAR（Base Address Register，基地址寄存器）。

## 9.1　I/O Kit 中的 PCI

I/O Kit中的PCI由`IOPCIFamily`处理，正如`IOUSBFamily`一样，`IOPCIFamily`在自己的KEXT中实

现。从类数量来看，IOPCIFamily要比IOUSBFamily简单。这意味着，实现驱动程序时，能帮得上忙的基础组件会比较少。从驱动程序的观点来看，PCI比USB更低级，因而，编写PCI设备的驱动程序通常要更复杂。IOPCIFamily的类层次结构如图9-2所示。

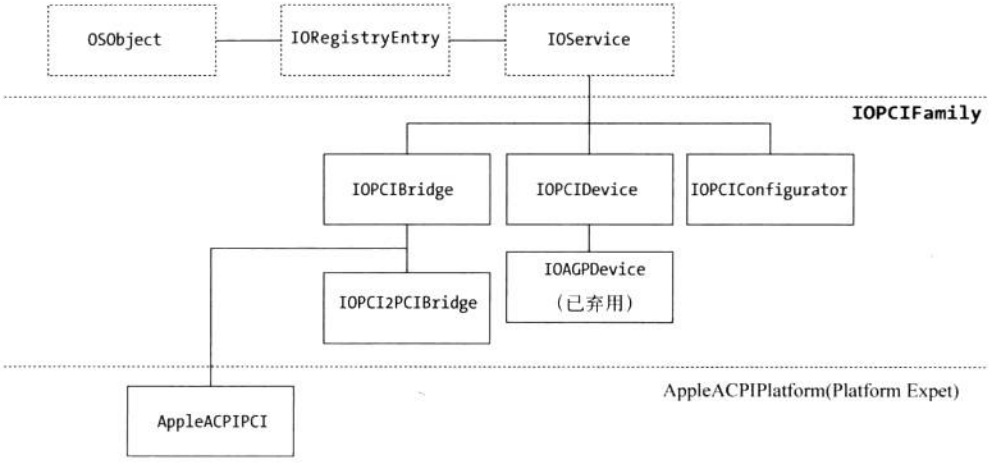



图9-2 IOPCIFamily类层次结构

IOPCIDevice对象的功能类似于块，它是所有基于PCI设备的提供者，包括PCIe、Thunderbolt和ExpressCard。IOPCIDevice的子类IOAGPDevice处理旧的基于AGP（Advanced Graphics Port，高级图形端口）的显卡，但基于英特尔的Mac并没有AGP特性。该对象的实例提供给系统中的每个PCI设备，类似地，IOPCIBridge实例用于系统中的每个PCI桥。有些情况下，驱动程序可能需要与它的桥进行交互，读/写该桥的配置空间。尽管并不常见，但有时也可能需要创建你自己的PCI桥驱动程序。本章后面将讨论设备及桥访问。根PCI桥（也称为主桥和根组件）由IOPCIBridge的一个子类AppleACPIPCI实现。APPleACPIPCI类是AppleACPIPlatform KEXT实现的Platform Expert的一部分，控制对系统中所有设备及桥的访问。该类只有一个实例。IOPCI2PCIBridge类是PCI-to-PCI桥的驱动程序。

**提示** 与IOUSBFamily一样，IOPCIFamily也不是XNU源代码发布版的一部分，但它可以从http://opensource.apple.com单独下载。源包中包含上述类的源代码，以及一个示例PCI驱动程序和转储PCI设备转储信息的用户空间工具的源代码。

### 9.1.1 匹配和加载驱动程序

PCI驱动程序通常匹配其配置空间寄存器中的属性，如厂商ID、设备ID、类、子系统厂商ID及子系统设备ID。通常，如果PCI设备基于通用芯片，则需要后两个属性。

尽管PCI驱动程序的配置空间中包含更多字段，但它们都无法使用匹配字典进行匹配（基于属性的匹配）。如果你需要更高级的匹配，驱动程序必须重载IOService::probe(IOProvider* service)方法，并且亲自检查IOPCIDevice，判断你的驱动程序是否与设备相匹配。表9-1中列出的键可以用于匹配基于PCI的设备。

表9-1　匹配PCI设备的键

| 键 | 描　述 |
|---|---|
| IOPCIMatch | 匹配厂商ID和设备ID；如果不匹配，则尝试匹配子系统厂商ID和子系统设备ID |
| IOPCIPrimaryMatch | 仅匹配厂商ID和设备ID |
| IOPCISecondaryMatch | 仅匹配子系统厂商ID和子系统设备ID |
| IOPCIClassMatch | 匹配PCI类代码 |
| IONameMatch | 非PCI特有；可以用于匹配名称属性 |

代码清单9-1展示了一个典型PCI设备的匹配字典，其中有厂商ID或子系统厂商ID 0xabcd，设备ID或子系统设备ID 0x1234。

**代码清单9-1　PCI驱动程序的简单匹配字典**

```
<key>IOKitPersonalities</key>
<dict>
    <key>MyPCIDriver</key>
    <dict>
        <key>CFBundleIdentifier</key>
        <string>com.osxkernel.MyPCIDriver</string>
        <key>IOClass</key>
        <string>com_osxkernel_MyPCIDriver</string>
        <key>IOProviderClass</key>
        <string>IOPCIDevice</string>
        <key>IOPCIMatch</key>
        <string>0x1234abcd</string>
     </dict>
</dict>
```

该匹配值指定为一个采用小端序格式的32位十六进制字符串。前4个字符表示设备ID，后4个字符表示厂商ID。值得注意的是，该值是字符串类型，而不是整型。IOProviderClass键的值必须是IOPCIDevice，以便I/O Kit将一个IOPCIDevice实例传递给你的驱动程序。如果需要匹配厂商ID和设备ID，可以将IOPCIMatch替换为IOPCIPrimaryMatch，如果你仅希望匹配子系统ID，可以使用IOPCISecondaryMatch。

如果你的驱动程序处理多个设备，则可以采用由一个空格分隔的列表，如下所示：

```
<key>IOPCIMatch</key>
<string>0x1234abcd 0x1235abcd 0x1236abcd</string>
```

这会匹配厂商0xabcd的设备ID 0x1234、0x1235和0x1236。如果你的驱动程序支持一个大的设备族，则可以使用掩码实现相同的效果，而不是分别枚举每个设备。

```
<key>IOPCIMatch</key>
<string>0x1230abcd&0xfff0ffff</string>
```

**注意** 如果直接编辑Info.plist，则必须将&号表示为&amp，因为在XML中，&号用来表示转义字符。

这将匹配每个以0x123X开头的设备ID，例如，从0x1230到0x123F的设备ID，并且厂商ID为0xabcd。应该被掩码忽略的位必须设置为0。

还可以匹配类寄存器。为此，必须指定IOPCIClassMatch键。类寄存器是3字节宽。但为了与其匹配，I/O Kit需要指定一个4字节的值，而最后一个字节则被忽略。以下示例匹配显示控制器（基类代码为0x03）：

```
<key>IOPCIClassMatch</key>
<string>0x03000000&amp;0xFF000000</string>
```

由于所有的PCI设备都根据其设备ID和厂商ID分配了名称，所以还可以使用IONameMatch匹配PCI设备，如代码清单9-2所示。

**代码清单9-2 基于名称属性的匹配**

```
<key>IONameMatch</key>
<array>
    <string>pciabcd,1234</string>
    <string>pciabcd,1235</string>
    <string>pciabcd,1236</string>
</array>
```

前面的方案可能更易读，但缺点是不能匹配子系统厂商ID和子系统设备ID。

**注意** 记住在驱动程序Info.plist文件中的OSBundleLibraries部分，添加一个对IOPCIFamily的依赖。

系统启动期间，将加载安装在物理插槽或嵌在主板上的PCI设备驱动程序。而Thunderbolt和ExpressCard的驱动程序可以在启动时加载或在插入时按需加载。

虽然Thunderbolt设备遵循PCI设备的识别规则，但需要进行额外修改才能加载驱动程序。在驱动程序的Info.plist文件中，需要在每个指定的特性下设置下列键：

```
<key>IOPCITunnelCompatible</key>
<true/>
```

这告诉系统，驱动程序处于Thunderbolt就绪状态，因此，可以安全卸载。Thunderbolt和PCIe设备可以共享相同的设备驱动程序，但在许多情况下，编写PCI驱动程序时，需要假定驱动程序/设备在操作期间从不移除。驱动程序将不会对Thunderbolt设备进行加载，除非对该键进行了设置。

**Thunderbolt唯一标示符**

所有的Thunderbolt设备都有一个设备ROM（DROM），包含一个额外的ID标识厂商，即授权ID。该ID是64位UID号的一部分，对每个Thunderbolt设备来说具有唯一性，就像网络接口的MAC地址。授权ID由Thunderbolt命名机构（英特尔）而不是PCI-SIG分配。写作本书时，还没有资料说明如何从I/O Kit访问该编号，或者它是否可用于匹配Thunderbolt设备。

### 9.1.2　驱动程序示例：一个简单的PCI驱动程序

现在该亲自动手实践一下了。为了演示一个PCI驱动程序的操作，我们将利用IOMatchCategory键，以便加载设备的二级驱动程序。在这种情况下，我们将为显示控制器（显卡/GPU）加载驱动程序，因为这是所有Mac（即使是笔记本电脑）都有的设备因为它们都在内部使用PCIe。我们使用下列键来匹配显示控制器：

```
<key>IOPCIClassMatch</key>
<string>0x03000000&amp;0xFF000000</string>
```

**警告**　注意你对MyFirstPCIDriver的修改，因为它被装载到了已经被另一个驱动程序控制的设备中。因此，执行除查询信息以外的操作并不安全，会造成系统崩溃或损坏。

回想一下，0x03是显示控制器的基类代码。如果你的GPU不止一个，这将会实例化驱动程序的多个实例（一个设备实例化一个）。

我们从驱动程序的类声明开始，如代码清单9-3所示。

**代码清单9-3　MyFirstPCIDriver类声明**

```
#include <IOKit/IOLib.h>
#include <IOKit/pci/IOPCIDevice.h>

class com_osxkernel_MyFirstPCIDriver : public IOService
{
    OSDeclareDefaultStructors(com_osxkernel_MyFirstPCIDriver);

private:
    IOPCIDevice*        fPCIDevice;

public:
    virtual bool start(IOService* provider);
    virtual void stop(IOService* provider);
};
```

如果你已操作了前面的例子，在这应该就不会那么惊讶。我们只声明了IOService的一个子

类，并覆写了start()和stop()方法。请注意，这段代码中包含了IOKit/pci/IOPCIDevice.h文件，其中包含了IOPCIDevice类的定义。

MyFirstPCIDriver的实现如代码清单9-4所示。

**代码清单9-4 MyFirstPCIDriver类实现**

```
#include "MyFirstPCIDriver.h"

#define super IOService
OSDefineMetaClassAndStructors(com_osxkernel_MyFirstPCIDriver, IOService);

bool com_osxkernel_MyFirstPCIDriver::start(IOService * provider)
{
    IOLog("%s::start\n", getName());

    if(!super::start(provider))
        return false ;

    fPCIDevice = OSDynamicCast(IOPCIDevice, provider);
    if (!fPCIDevice)
        return false;

    fPCIDevice->setMemoryEnable(true);

    registerService();

    return true;
}

void com_osxkernel_MyFirstPCIDriver::stop( IOService * provider )
{
    IOLog("%s::stop\n", getName());
    super::stop(provider);
}
```

驱动程序匹配成功时，无论是通过Info.plist字典还是调用驱动程序的probe()方法，驱动程序都会调用start()方法。与第8章中的USB驱动程序一样，我们需要进行检查，确保传递给我们的提供者是正确的类型（IOPCIDevice），这是一种很好的做法，尽管在Info.plist正确指定了IOProviderClass键时并不应该这样做。

如果我们已经获得一个有效的IOPCIDevice，接下来便是调用IOPCIDevice::setMemoryEnable(bool enable)方法，启用设备的I/O资源。这会在设备的命令寄存器上设置一个触发位，使它知道我们想要访问其资源。最后，我们的驱动程序调用registerService()，通知可能的客户端（可能为一个高层驱动程序）驱动程序已经接入。返回true，则告知I/O Kit驱动程序加载成功。

现在，可以使用kextload实用程序尝试加载MyFirstPCIDriver。可以检查Console.app中的kernel.log或使用IORegistryExplorer对其进行检索（如图9-3所示），从而验证驱动程序是否正确加载。

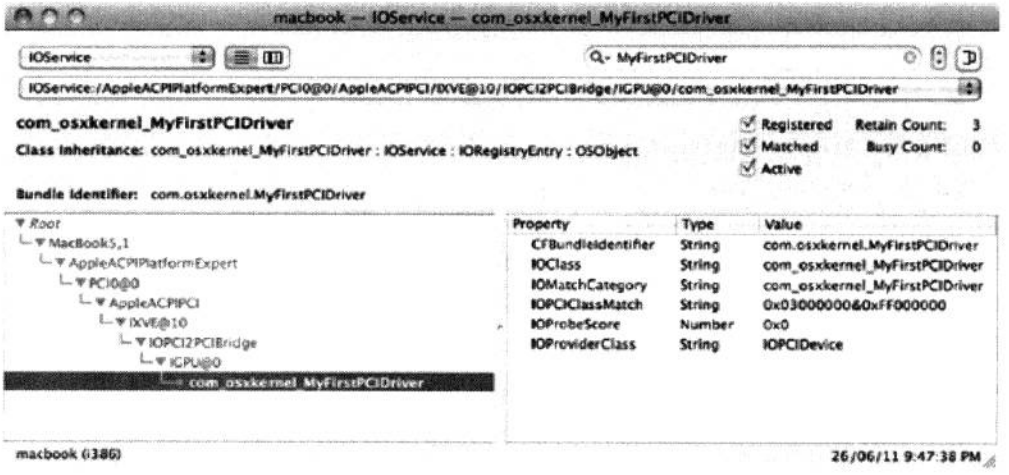



图9-3　IORegistryExplorer显示加载的MyFirstPCIDriver

### 9.1.3　访问配置空间寄存器

IOPCIDevice类包含许多辅助方法，利用这些方法可以很方便地访问设备的配置空间寄存器。下列方法可以读写配置空间寄存器。

```
virtual UInt8 configRead8(UInt8 offset);
virtual UInt16 configRead16(UInt8 offset);
virtual UInt32 configRead32(UInt8 offset);

virtual void configWrite8(UInt8 offset, UInt8 data);
virtual void configWrite16(UInt8 offset, UInt16 data);
virtual void configWrite32(UInt8 offset, UInt32 data);
```

其中有3个读方法的变体和3个写方法的变体，可以读/写从指定偏移量开始的8位值、16位值或32位值。Offset参数是配置空间的字节偏移量，范围是0~255。为了读取设备的设备ID和厂商ID，可以执行如下操作：

```
UInt16 vendorID = fPCIDevice->configRead16(0);
UInt16 deviceID = fPCIDevice->configRead16(2);
IOLog("vendor ID = 0x%04x device ID = 0x%04x\n", vendorID, deviceID);
```

上面的请求还可以通过一个调用实现：

```
UInt32 bothIDs = fPCIDevice->configRead32(0);
IOLog("vendor ID = 0x%04x device ID = 0x%04x\n", bothIDs >> 16, bothIDs & 0x0000FFFF);
```

前面的调用使用了整数字节偏移量，但IOPCIDevice.h文件指定了一些常量，可以用来确定常见寄存器的位置。因此，为了使代码可读性更强，可以使用kIOPCIConfigVendorID和kIOPCIConfig-DeviceID，而不是硬编码值。可用常量的完整列表如代码清单9-5所示。

**代码清单9-5 常见PCI配置空间寄存器的偏移常量（IOPCIDevice.h）**

```
enum {
    kIOPCIConfigVendorID = 0x00,
    kIOPCIConfigDeviceID = 0x02,
    kIOPCIConfigCommand = 0x04,
    kIOPCIConfigStatus = 0x06,
    kIOPCIConfigRevisionID = 0x08,
    kIOPCIConfigClassCode = 0x09,
    kIOPCIConfigCacheLineSize = 0x0C,
    kIOPCIConfigLatencyTimer = 0x0D,
    kIOPCIConfigHeaderType = 0x0E,
    kIOPCIConfigBIST = 0x0F,
    kIOPCIConfigBaseAddress0 = 0x10,
    kIOPCIConfigBaseAddress1 = 0x14,
    kIOPCIConfigBaseAddress2 = 0x18,
    kIOPCIConfigBaseAddress3 = 0x1C,
    kIOPCIConfigBaseAddress4 = 0x20,
    kIOPCIConfigBaseAddress5 = 0x24,
    kIOPCIConfigCardBusCISPtr = 0x28,
    kIOPCIConfigSubSystemVendorID = 0x2C,
    kIOPCIConfigSubSystemID = 0x2E,
    kIOPCIConfigExpansionROMBase = 0x30,
    kIOPCIConfigCapabilitiesPtr = 0x34,
    kIOPCIConfigInterruptLine = 0x3C,
    kIOPCIConfigInterruptPin = 0x3D,
    kIOPCIConfigMinimumGrant = 0x3E,
    kIOPCIConfigMaximumLatency = 0x3F
};
```

IOPCIDevice还提供了一个便捷方法，设置寄存器的各个位，该方法名为setConfigBits()。

对丢失或故障设备进行读请求，将返回0xFFFF值（8位变体为0xFF，32位变体为0xFFFFFFFF），这是一个无效的设备/厂商ID。因此，如果在读取时返回该值，则寄存器可以用来确定是否发生了问题或Thunderbolt设备是否已经拔出。

将值写入到配置空间非常简单，但需要注意一些事项。配置空间的许多区域为只读。例如设备ID和厂商ID被编写到设备的PCI控制器固件中。还要注意的一点是，无法确定写入寄存器位置是否成功；必须读回该寄存器或另一个受写事务影响的寄存器才能确定写入是否成功。

**注意** 如果驱动程序要兼容基于PowerPC的系统，需要注意PCI配置空间是以小端序格式存储，但IOPCIDevice可以为你处理字节交换。

IOPCIDevice的许多方法（如setMemoryEnable()）只是便捷抽象，替你执行适当的配置空间读/写。配置空间的I/O由IOPCIDevice转送给其父类（大多数情况下是IOPCIBridge），直至到达根桥，这由Platform Expert实现，具体细节取决于系统。

**1. 访问扩展配置空间**

可能你已经注意到上一节中的I/O不一致。前面提到，扩展配置空间是4096字节。当config*()

的偏移量参数为UInt8类型时，如何对大于255的偏移量进行寻址呢？答案是使用下列方法族。

```
UInt32 extendedConfigRead32(IOByteCount offset);
UInt16 extendedConfigRead16(IOByteCount offset);
UInt8 extendedConfigRead8(IOByteCount offset);

void extendedConfigWrite32(IOByteCount offset, UInt32 data);
void extendedConfigWrite16(IOByteCount offset, UInt16 data);
void extendedConfigWrite8(IOByteCount offset, UInt8 data);
```

这些方法的接口都相同，如这段代码所示。但它们对offset参数都使用了较宽的数据类型IOByteCount，以访问大于255的偏移量。

**2. 搜索功能寄存器**

因为功能寄存器并没有位于固定的偏移量处，所以查找功能寄存器的流程是，检索功能ID，然后读取下一个字节，确定该功能的长度，同时还可以得知下一个功能的偏移量。接下来是下移列表，直至查找到正确的功能。幸好我们无须手工编写该代码，因为IOPCIDevice提供了两个辅助方法用于查找功能：

```
virtual UInt32 findPCICapability(UInt8 capabilityID, UInt8* offset = 0 );
virtual UInt32 extendedFindPCICapability(UInt32 capabilityID, IOByteCount* offset = 0 );
```

下面演示一下如何获取PCIe链接状态寄存器，该寄存器包含设备的许多活动通路（4~9位）及链接速度（0~3位）。

```
IOByteCount offset = 0;
if (fPCIDevice->extendedFindPCICapability(kIOPCIPCIExpressCapability, &offset))
{
        UInt16 value = fPCIDevice->extendedConfigRead16(offset + 0x12);
}
```

该方法将返回功能ID（这种情况下为kIOPCIExpressCapability），或者未找到指定ID的功能时返回0。输出参数offset用于存储已找到功能的偏移量。找到功能后，便可以通过将偏移量加0x12（18）来读取链接状态寄存器。

## 9.1.4　PCI I/O内存区

PCI设备最多可以有6个I/O区。每个区都包含I/O内存或I/O空间（端口）。后者在新的设备中很少使用，因为I/O端口执行I/O时通常非常缓慢，并且只可以使用特定的输入/输出CPU指令访问。一些传统的设备（如IDE控制器）可能同时有I/O接口和内存，并且可由任意一个控制。另一方面，I/O内存非常高效，更容易访问，因为它能被简单地映射到系统的内存空间，可以像普通的内存一样访问。I/O内存通常称为MMIO（Memory Mapped I/O，内存映射I/O）。不要将这个概念与虚拟地址空间之间的内存映射或内存中的文件映射（mmap）混淆。

如果内存区中存在内存可预取的位设置，则到达/来自映射设备内存的访问可以由CPU进行缓存。

如何通过内存区控制设备？这完全取决于设备。例如，一个区可用于控制寄存器和状态寄存器，而第二个区可用于读写数据，如从照相机输入视频。如果你正采用电子表格形式读取，则该

文本可以连续写入表示显卡帧缓冲区的内存区。就像USB一样，也存在许多标准化的接口，用于PCI设备。VGA兼容显卡就是一个例子，它允许使用已知的接口（包括内存区和端口）执行显卡的基本操作。还存在面向IDE、SATA和基于PCI的USB控制器的标准化接口，这使操作系统可以将其默认的驱动程序用于任何遵从此种接口的设备。

因为PCI为“即插即用”，所以PCI设备的I/O资源由内核/EFI（或传统PC中的BIOS）自动配置，与此相反，在废弃的ISA总线中，必须放置跳线，分别为每个设备选择基础I/O地址和中断线，以避免资源冲突。

配置设备时，配置空间中的每个区都将使用自己的地址范围进行配置。具体范围长度取决于设备。

配置设备时，系统会为设备分配一个物理内存地址范围。从图9-1可以发现，不存在存储每个内存区长度的寄存器。那么，系统如何知道每个区的大小呢？内存区的长度由系统确定，方法是在配置空间的某个基地址插槽中设置所有位，然后读回该值。一个区的长度必须是2的幂。设备对此不支持时，可以合并两个BAR形成一个64位地址。

系统或驱动程序可以访问任意的I/O区之前，需要触发设备命令寄存器中的位来启用这些区。我们已经知道，在MyFirstPCIDriver中，这可以通过在驱动程序的start()方法中调用fPCIDevice->setMemoryEnable(true)来实现。

### 1. 枚举I/O区

为了找到PCI设备的可用内存区（最多可能有6个），要修改一下MyFirstPCIDriver，从而在start()方法中转储一些额外的设备信息，这在调用setMemoryEnable()后通过添加代码清单9-6中的代码实现。

**代码清单9-6 枚举PCI I/O 内存区**

```
for (UInt32 i = 0; i < fPCIDevice->getDeviceMemoryCount(); i++)
{
    IODeviceMemory* memoryDesc = fPCIDevice->getDeviceMemoryWithIndex(i);
    if (!memoryDesc)
        continue;
    #ifdef __LP64__
        IOLog("region%u: length=%llu bytes\n", i, memoryDesc->getLength());
    #else
        IOLog("region%lu: length=%lu bytes\n", i, memoryDesc->getLength());
    #endif
}
```

编译、加载该驱动程序时，在kernel.log中应该会看到类似下面这样的输出：

```
Apr 1 11:06:18 macbook kernel[0]: com_osxkernel_MyFirstPCIDriver::start
Apr 1 11:06:18 macbook kernel[0]: region0: length=16777216 bytes
Apr 1 11:06:18 macbook kernel[0]: region1: length=268435456 bytes
Apr 1 11:06:18 macbook kernel[0]: region2: length=33554432 bytes
Apr 1 11:06:18 macbook kernel[0]: region3: length=128 bytes
Apr 1 11:06:18 macbook kernel[0]: region4: length=131072 bytes
```

你的输出可能会和你的系统模型及显卡有关（可能会有多个显卡）。在这种情况下，最大的区（256 MB）是区1，这是显卡的帧缓冲区。

### 2. 映射和访问设备内存区

上一节已经讲述了如何获取可用I/O内存区的信息。在可以实际访问内存区的数据之前，还需要完成一些工作。另外，在大部分实际的驱动程序中，不需要映射所有的内存区时，无须明确枚举内存区，因为驱动程序通常十分清楚要映射哪些区。下面的IOPCIDevice方法可以直接映射BAR区。

```
virtual IOMemoryMap * mapDeviceMemoryWithRegister(UInt8 reg, IOOptionBits options = 0);
```

以下是其应用的例子。

```
IOMemoryMap *bar0Map = fPCIDevice->mapDeviceMemoryWithRegister(kIOPCIConfigBaseAddress0);
IOMemoryMap *bar1Map = fPCIDevice->mapDeviceMemoryWithRegister(kIOPCIConfigBaseAddress1);
if (bar0Map)
{
    UInt8 *address = (UInt8*)bar0Map->getVirtualAddress();
    //使用地址执行一些操作
}
…
```

如果通过调用代码清单9-6中的getDeviceMemoryWithIndex()方法，已经获得一个IODevice-Memory（IOMemoryDescriptor的子类）对象，则可以只调用map()方法，该方法可以执行同样的工作。实际上，这是mapDeviceMemoryWithRegister()在底层执行的实际操作。获得IOMemoryMap对象之后，可以调用getVirtualAddress()方法，获得内核虚拟地址，访问映射。返回的指针如果指向I/O内存，而不是I/O空间，则可以像普通的内存一样，进行读写。

### 3. 访问I/O空间

I/O空间包含一个16位的地址空间，是一种与设备进行通信的古老方式。在旧计算机中，I/O端口也可以用来与串行及并行端口进行通信，因此它并不特定于PCI，只是外部设备（相对于CPU来说）与处理器交互的一种方式。分配给设备的I/O空间范围就像内存区一样，可以使用mapDeviceMemoryWithRegister()和getDeviceMemoryWithIndex()进行访问和映射。但不同的是，并不能像上面那样直接访问getVirtualAddress()返回的指针。必须使用下面的某一种方法：

```
virtual void ioWrite32(UInt16 offset, UInt32 value, IOMemoryMap* map = 0);
virtual void ioWrite16(UInt16 offset, UInt16 value, IOMemoryMap* map = 0);
virtual void ioWrite8(UInt16 offset, UInt8 value, IOMemoryMap* map = 0);
virtual UInt32 ioRead32(UInt16 offset, IOMemoryMap* map = 0);
virtual UInt16 ioRead16(UInt16 offset, IOMemoryMap* map = 0);
virtual UInt8 ioRead8(UInt16 offset, IOMemoryMap* map = 0);
```

我们并不赞同在新设备中使用I/O空间，因为它性能低下、地址空间有限。访问映射内存可以只使用1个CPU周期，而在某些体系结构上，访问端口可以多达100个周期。

在I/O空间可以访问之前，需要在设备的命令寄存器中启用。为此，IOPCIDevice提供了setIOEnable()方法。

### 9.1.5 处理设备移除

Thunderbolt和ExpressCard设备可以在操作期间拔出。因此，处理这些设备的驱动程序，需要在传统的PCI驱动程序上进行额外的修改，而传统的驱动程序通常并不需要编写设备的移除处理。不正确的设备移除可能会导致应用程序挂起、系统崩溃，破坏系统稳定性或性能。对于Thunderbolt设备，移除并不是异常现象，因此驱动程序必须能够处理设备的移除。

---

**警告** 带有装载文件系统的存储设备，在用户“弹出”（卸载）该文件系统之前，不可以拔出。执行失败会导致文件系统丢失，最坏的情况下，甚至会损坏文件系统。基于Thunderbolt的存储驱动程序应该在调用驱动程序的start()方法时调用：setProperty(kIOPropertyPhysicalInterconnectLocationKey, kIOPropertyExternalKey)，以告诉I/O Kit该存储器属外部连接。

---

虽然处理设备移除看似复杂，但I/O Kit进行了专门的设计，允许设备移除。IOService类可以为我们自动处理许多困难的工作。

驱动程序从内存映射I/O或PCI配置空间寄存器读取值时，如果接收到0xffffffff值（假定读取一个32位的值），则可以检测到设备已经移除的通知。当然，该值实际上可能对某些寄存器有效，但你可以读取一个肯定不包含该值的备用寄存器或内存位置，确定设备没有反应。驱动程序可以在I/O Kit通知驱动程序设备已经移除之前，检测该条件。如果驱动程序确定设备已经移除，则应该停止访问所有的映射内存及配置空间，因为进一步的请求会导致请求超时，这会花费几毫秒，影响整体的系统性能。苹果公司建议通过一个方法对MMIO进行所有的访问，如下所示：

```
UInt32 com_osxkernel_MyFirstPCIDriver::readRegister32(UInt32 offset)
{
    UInt32 res = 0xffffffff;
    if (!fDeviceRemoved)
    {
        res = OSReadLittleInt32(fBar0Address, offset);
        if (res == 0xffffffff)
            fDeviceRemoved = true;
    }
    return res;
}
```

该方法可以防止设备已经移除之后，进一步访问寄存器。现在可以使用驱动程序其他部分中的成员变量fDeviceRemoved，防止与硬件进行通信的操作。

I/O Kit分3个阶段处理设备移除。

(1) 总线控制器（PCI根）将在其客户端块上调用terminate()方法，再次通知其客户端等，直至到达栈的底部。IOService对象覆写了message()方法，也将接收到一个kIOServicesIsTerminated消息。驱动程序现在被认为处于不活动状态，无法由新的客户端枚举或

装载。持有该驱动程序的现有客户端将仍然保持活动状态。

(2) 栈中的驱动程序将调用其willTerminate()方法，之后调用didTerminate()。该过程以相反的顺序进行，因此客户端将调用其提供者，而不是采用其他方式，直至到达在第一个位置调用terminate()的初始提供者。请记住，这些方法可以选择，你能够根据驱动程序的需要选择实现某些方法。为了响应对其willTerminate()方法的调用，驱动程序应该清除所有的队列请求，并取消正在处理的I/O，如未完成的DMA传输。

(3) 移除的最后一个阶段是调用驱动程序的stop()方法，然后调用detach()，将它从I/O Registry中移除。如果驱动程序的持有计数变为0，则驱动程序将被释放，并调用其free()方法。

如果用户再次插回同个设备，则会分配该驱动程序的一个新实例。此时访问该驱动程序的任何应用程序仍然会连接到该驱动程序原来的实例。为了处理这种情况，应用程序必须安装一个通知，检测驱动程序/设备何时移除或添加到系统。因为在设备重新插入时，驱动程序实例不可以再次使用，所以在其处理完设备移除之后，无须返回到默认状态。但重要的是正确解除和释放任何使用的资源，因为新的实例将重新分配或回收这些资源，而它们有可能导致内存泄漏，或者新的驱动程序实例无法正确生成。

---

**提示**　Xcode提供一个命令行工具ioclasscount，输出OSObject派生类的实例计数，可以帮助调试与设备移除相关的内存泄漏。更多信息请参见第16章。

---

## 9.2　中断

中断是当感兴趣的事件发生时，如按下键盘上的一个键、移动鼠标或点击鼠标按钮时，PCI及其他硬件设备使用的一种异步通知CPU的机制。网络摄像头会在一个新的视频帧每次变为可用时发送一个中断，这样它的驱动程序就可以知道何时能从其映射内存区读取新的帧。中断使CPU不必轮询每个设备来判断是否有新的可用数据。传统的PCI设备使用专用的中断针脚，实际上从PCI卡/槽固定到CPU上的一个针脚。现代的Thunderbolt及基于PCIe的技术都使用MSI（message-signaled interrupt，消息信号中断），而无需设备和CPU或中断控制器芯片之间的专用物理线。传统的PCI卡有4个中断针脚，这限制了可以用于设备的中断数量。但MSI使每个设备最多可以有32个中断。虽然MSI不同于传统的中断，但从驱动程序的角度来看，它们的作用都一样。

CPU接收到一个中断时，会将当前运行的线程睡眠，即使该线程属于内核本身。中断发生时，CPU将试图定位触发中断的ISR（Interrupt Service Routine，中断服务例程）。

ISR导向到“拥有”该设备的驱动程序。传统的基于设备的中断可以共享中断线。在这种情况下，驱动程序需要询问设备，通常是通过读取内存映射寄存器判断其设备是否发生中断。

MSI中断从不共享，尽管这仍然是一个预测该情况的好做法。中断并不总是来自硬件设备（如PCI），它还可以由系统定时器发送，用于驱动操作系统服务（如调度器）。

ISR运行时，处理中断的CPU将禁用其他中断，这意味着在ISR完成之前CPU上什么都不会执行。

可以想象，如果一个驱动程序在ISR回调中执行大量的工作，将降低系统性能。实际上，强烈推荐的做法是驱动程序除了确认中断什么都不做。如果中断没有确认，ISR就会连续运行，这将影响性能及稳定性。ISR运行时通常被称为主中断上下文（primary interrupt context）。为了提高系统性能，大多数操作系统（包括OS X和iOS）都有一些机制，将内核线程的中断处理延迟到稍后执行。这通常被称为二级中断上下文（secondary interrupt context）。处理中断的实际工作在二级中断上下文（线程）中执行，如复制从网络接收到的分组。主处理程序通常只确认中断，然后，如果有工作需要执行，它将通知二级处理程序。

在主上下文中，不可以执行阻塞或睡眠的操作，这包括大部分的内存分配例程以及持有除自旋锁以外的锁。阻塞/睡眠之所以执行，是因为它放弃了对CPU的访问，临时将CUP让给其他某一线程。但ISR与任务或线程描述符相关。因此，调度器不可以再次调度回ISR，因为它直接由CPU触发。

二级中断处理程序并没有这些限制，可以顺利地分配内存，阻塞对锁的等待，使其变为可用。这在OS X和iOS中没有问题，但某些操作系统可能在不支持该功能的上下文中运行二级处理程序（也称为下半部，主处理程序为上半部）。

虽然OS X和iOS对二级中断处理程序中的工作几乎没有什么限制，但这仍然非常高效。

主中断的处理不能由用户空间程序完成。如果用户空间程序需要知道中断何时发生，则需要由驱动程序通知。

在OS X中，所有的主中断都导向到CPU 0（核心），二级中断分布在所有的核心中，这使多个驱动程序可以并行运行。因为二级中断运行在独立的内核线程中（高权限），所以当中断发生时，它可以像任何其他线程一样进行调度和运行。中断机制在概念上很容易实现。但由于数据在主处理程序和二级处理程序之间共享时的并行性，以及用户线程可能会调用需要访问相同数据的驱动程序，中断机制也会很复杂。需要特别注意的是，要确保没有死锁，减少各种执行线程之间的关联。这样可以保证没有线程会长时间等待访问所需的资源。关于同步的更多信息，请参见第7章。

**注意** 主中断有时被称为直接中断，二级中断也称为间接中断。

### 9.2.1 I/O Kit中断机制

I/O Kit中，处理主中断及二级中断的首选方式是通过工作环系统，但也可以直接处理。如果你不清楚工作环的操作方式，请查看第7章的详细内容。处理驱动程序中断的可用机制有3种。图9-4展示了3种不同机制对主中断的响应方式。

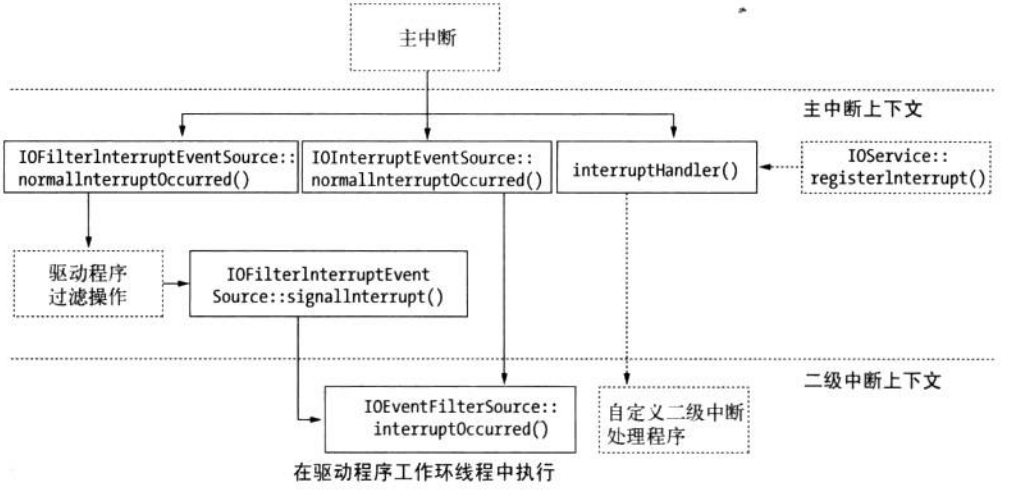



图9-4　处理设备中断的I/O Kit机制

- IOInterruptEventSource：处理设备中断的标准方式和最容易的方式。只需注册一个处理程序方法，在二级中断上下文中执行。你的驱动程序无须处理主中断。中断将被提供者禁用，直至中断处理程序完成，从而保证中断的单线程处理，因为其他处理程序无法并行运行。IOInterruptEventSource是处理中断的首选方式。
- IOFilterInterruptEventSource：位于图9-4的左侧，是IOInterruptEventSource的子类，更加方便灵活。它允许提供一个自定义的过滤操作。过滤操作在主中断上下文中调用，使驱动程序可以询问硬件设备，以判断它是否确实存在中断。如果中断在数个设备之间共享，或对非常低的延迟有要求，则推荐使用该方法。二级中断基于安装的过滤操作（例程）的返回值进行调度。
- IOService::registerInterrupt()：最后的方法是使用IOService::registerInterrupt()注册一个C函数，在主中断期间调用。该方法不使用驱动程序的工作环，也不调用二级中断处理程序。如果需要二级中断，该处理机制必须手工实现。

## 9.2.2　注册接收中断

与I/O Kit中的许多操作一样，注册中断非常简单，大部分难题都由I/O Kit及Platform Expert进行内部处理。我们无须关注IRQ编号或中断路由的分配，因为这些都会自动处理。如何注册一个驱动程序，并使用主中断过滤器接收来自其提供者（IOPCIDevice）中断的常见代码块，如代码清单9-7所示。

**代码清单9-7 创建一个过滤中断事件源**

```
bool MyFirstPCIDriver::start(IOService * provider)
{
    ...
    IOWorkLoop *workLoop = (IOWorkLoop*)getWorkLoop();
    if (!workLoop)
        return false;

    IOFilterInterruptEventSource* interruptSource =
        IOFilterInterruptEventSource::filterInterruptEventSource(this,
                (IOInterruptEventAction) &MyFirstPCIDriver::interruptOccurred,
                (IOFilterInterruptAction) &MyFirstPCIDriver::interruptFilter,
                provider, 0);

    if (workLoop->addEventSource(interruptSource) != kIOReturnSuccess)
            return false;
    ...
}
```

这包含4个步骤。

(1) 获得或分配一个IOWorkLoop实例。

(2) 分配事件源，使用工厂方法filterInterruptEventSource()实现。其中传递了5个参数。

- 一个指向我们自身的指针。
- 一个指向二级中断处理程序interruptOccurred()方法的指针。
- 在主中断上下文中执行的过滤操作方法。
- 最后一个参数是提供者的中断索引号，如果提供者有多个中断类型则可以指定该参数。例如，提供者除了共享的中断，还可能支持MSI。
- 中断类型索引。详细信息请查看“启用消息信号中断”一节。

(3) 使用addEventSource()将事件源添加到IOWorkLoop实例。

(4) 最后一步是启用事件源，因为它默认情况下为禁用，即使在添加到工作环以后。为了开始接收中断，调用interruptSource->enable()。

重要的一点是，确保在调用该方法之前，驱动程序已完全初始化并且准备好接收中断，如有可能，在驱动程序准备开始处理中断之前确保中断在硬件上无效。

注册IOInterruptEventSource的流程几乎完全相同，如代码清单9-8所示。

**代码清单9-8 创建中断事件源**

```
IOWorkLoop *workLoop = (IOWorkLoop*)getWorkLoop();
if (!workLoop)
        return false;

IOInterruptEventSource* interruptSource =
        IOInterruptEventSource::interruptEventSource(this,
                    (IOInterruptEventAction) &MyFirstPCIDriver::interruptOccurred,
                    provider, 0);
```

```
if (workLoop->addEventSource(interruptSource) != kIOReturnSuccess)
        return false;
...
```

唯一的不同是该处理并不接受过滤操作，IOFilterInterruptEventSource就是这种情况。

### 9.2.3　启用MSI

如果你确实需要使用MSI，则首先必须枚举MSI中断类型的索引。在代码清单9-7和代码清单9-8中，我们只是传递了0，取得提供者的第一个中断类型，这是否支持MSI还取决于设备。以下方法将枚举提供者的可用中断类型，并返回MSI中断类型的索引，这可以作为索引参数传递给IOInterruptEventSource::interruptEventSource()：

```
int com_osxkernel_MyFirstPCIDriver::findMSIInterruptTypeIndex()
{
    IOReturn ret;
    int index, source = 0;

    for (index = 0; ; index++)
    {
        int interruptType;
        ret = fPCIDevice->getInterruptType(index, &interruptType);
        if (ret != kIOReturnSuccess)
            break;

        if (interruptType & kIOInterruptTypePCIMessaged)
        {
            source = index;
            break;
        }
    }
    return source;
}
```

### 9.2.4　处理主中断

我们看一下主中断过滤的实现，以及如何调度二级中断处理程序。音频设备的主中断过滤看起来类似于代码清单9-9的实现。

**代码清单9-9　主中断过滤方法**

```
bool com_osxkernel_MyAudioPCIDriver::interruptFilter(OSObject* owner, IOFilterInterruptEventSource *
src)
{
    bool scheduleSecondaryInterrupt = false;

    com_osxkernel_MyAudioPCIDriver* me = (com_osxkernel_MyAudioPCIDriver*)owner;

    uint32_t registerContents = me->readRegister(kHardwareInterruptRegisterOffset);
    if (registerContents & kAudioInputInterruptBit)
```

```
    {
        scheduleSecondaryInterrupt = true;
        me->fAudioInputInterruptPending = true;
    }
    else if (registerContents & kAudioOutputInterrupt)
    {
        scheduleSecondaryInterrupt = true;
        me->fAudioOutputInterruptPending = true;
    }
    return scheduleSecondaryInterrupt;
}
```

该函数有两个参数。第一个是OSObject指针，在IOFilterInterruptEventSource实例构造时赋值。我们使用this指针传递自身。原因是interruptFilter()函数是一个静态类成员，而在C++中，普通成员函数不可以作为函数指针使用。我们仅将owner参数转换为驱动程序类的类型来检索我们的实例。还可以传递一个事件源实例。

---

**警告** 你可能想在主中断处理程序中添加调试语句，以查看其是否触发，或硬件中是否使用IOLog()设置了正确的寄存器。千万不要这么做。这个主意非常糟糕。为什么这么说呢？因为你的系统会崩溃。

---

因为过滤函数运行在主中断上下文中，所以大部分的I/O Kit框架调用都不安全，包括内存分配函数和大部分的锁函数。还应该避免在主上下文中执行I/O或其他冗长的操作。在代码清单9-9显示的假设示例中，我们处理了一个双向音频设备的中断，该设备有两个中断，每个方向一个。在我们的过滤器中，首先读取设备的中断寄存器。如果有一个中断已经设置，则将scheduleSecondaryInterrupt变量设置为true，并将其作为返回值。返回true，表示我们希望二级中断处理程序运行；返回false，表示设备没有中断。二者都意味着我们与另一个设备共享中断线，该设备也是引起中断的设备之一，或者可能是由于故障设备或干扰而引起的一个错误中断。如果过滤器返回true，则将禁用设备的中断，直至二级中断处理程序被调度且完成执行。这可以确保中断串行化，因此我们的驱动程序无须关注主处理程序和二级处理程序之间的锁，因为它们从不会同时运行。

在某些情况下，并不需要该行为，我们可以将中断过滤修改为总是返回false，从而防止中断被禁用，这将确保中断不被禁用，但也会阻止调度二级中断。我们可以手工工调度，如下所示：

```
bool com_osxkernel_MyFirstPCIDriver::interruptFilter(OSObject* owner, IOFilterInterruptEventSource *
src)
{
…
…
    if (scheduleSecondaryInterrupt)
            src->signalInterrupt();
    return false;
}
```

这样做的效果是，设备可以发送主中断，即使二级中断处理程序已经在运行。就音频设备而言，这支持输入和输出中断并发进行。

### 9.2.5　处理二级中断

无论是对IOInterruptEventSource还是对IOFilterInterruptEventSource使用，二级中断处理程序都一样。其原型类似于主中断过滤函数的原型，但当提供者的中断多余一个时，它有一个包含提供者中断源索引的额外参数。我们的音频设备只有一个中断，并且需要读取设备的寄存器以确定通知哪些事件。如果设备有多个中断，可以检查intCount参数进行区分。虚构的音频设备二级中断处理程序的一个简单实现，如代码清单9-10所示。

**代码清单9-10　二级中断处理程序方法**

```
void com_osxkernel_MyAudioPCIDriver::interruptOccurred(OSObject* owner, IOInterruptEventSource* src,
int intCount)
{
    com_osxkernel_MyFirstPCIDriver*     me;
    me = (com_osxkernel_MyFirstPCIDriver*)owner;

    if (me->fAudioInputInterruptPending)
            me->handleAudioInputInterrupt();          //开始下一个DMA
    if (me->fAudioOutputInterruptPending)
            me->handleAudioOutputInterrupt();         //开始下一个DMA
}
```

我们检测待解决的中断，并执行驱动程序方法以处理该中断。例如，该方法可能通知用户应用程序数据现在可用，并安装一个新的DMA事务以填充另一个缓冲区。我们使用的是实例变量fAudioInputInterruptPending和fAudioOuputInterruptPending，而不是从代码清单9-9重新读取中断状态寄存器，其原因是，许多硬件设备在读取寄存器后将自动清除中断寄存器，而它还要用来确认中断。

二级中断处理程序可以与调用我们驱动程序的用户空间线程并行运行，因此，执行正确的同步以保护共享数据非常重要。请注意，二级中断处理程序本身运行在驱动程序的工作环上，这是单线程，因此，要保证两个二级中断处理程序没有并行运行。

## 9.3　直接内存访问

利用DMA（Direct Memory Access，直接内存访问）设备与系统内存直接进行数据传输，而不涉及CPU，使CPU可以执行其他任务，这对整体系统性能的提升具有明显的影响，因为I/O传输相对于CPU来说，通常非常慢。DMA还允许所谓的零拷贝，其中，我们可以将内存从用户空间缓冲区直接传输到设备，而无需任何内存副本。PCI并没有中央DMA控制器，而是使用总线控制，这可以使设备控制总线并启动传输。IOPCIDevice类提供了setBusMaster- Enalbe()方法，使设备可以作为总线控制器。DMA传输具有方向性。当CPU希望将数据从系统内存传输到设备

时，被称为传出（outbound）DMA，而从设备到系统内存的传输，则被称为传入（inbound）DMA。

没有用来控制DMA到PCI设备的标准I/O Kit类，因为每个设备对DMA的实现可能都不同（设备的DMA功能常被称为DMA引擎）。但在大多数情况下，其处理都非常相似。一个设备可能支持数个并发的DMA传输，而每个传输都有它自己的DMA信道。DMA传输的概念非常简单。对于传出DMA传输，通常要执行以下步骤。

(1) 驱动程序需要确定传输的内存缓冲区，并钉住内存。

(2) 驱动程序将通知设备内存的位置（物理地址）及长度，通常通过将值写入寄存器来实现。

(3) 驱动程序将触发设备寄存器中的一位，以表示应该启动DMA。

(4) 设备将直接从系统内存传输缓冲区的内容，而不涉及CPU。设置另一个寄存器位，以表示传输完成，并生成一个中断。

(5) 驱动程序将处理该中断，检查DMA完成位是否已经设置，如果需要发送更多的数据可能就会指定另外一个DMA传输。

设置、处理传入DMA传输的流程都一样。唯一的区别在于，设备是写入缓冲区而不是从缓冲区读取。我们仍然需要告诉设备数据所在的内存位置，并在传输完成后获取一个中断。对于存储设备，驱动程序总是会发起I/O，并控制何时读，何时写。但像网络控制器之类的设备则稍有不同，其中数据可能会异步到达设备，响应外部事件。在这种情况下，设备将生成一个中断，并在其寄存器中设置一个标志，以便驱动程序知道设备的输入缓冲区中存有数据。设备完成传输之后，将生成另外一个中断，通知驱动程序传输已经完成。

虽然从概念的角度来看，DMA很简单，但由于它受以下因素影响而稍显复杂。

- 内存在CPU上的缓存会引起一致性问题，因为CPU写的数据可能会保存在CPU的一个缓存中，而不会立刻提交给系统内存。如果DMA传输在此时开始，设备可能会读取到不正确的数据，如RAM此前的内容或一般垃圾。在英特尔系统上，这种问题会由硬件自动处理，而不需要驱动程序的干预。对于PowerPC处理器，I/O Kit提供了`IOFlushProcessorCache()`函数，将CPU缓存刷出至系统内存，确保设备得到正确的内存内容。在基于英特尔的系统上，该函数存在，但不执行任何操作。
- 在64位平台上（或使用PAE时），一些旧的基于PCI的硬件设备可能无法访问32位以上的内存地址。有两种策略可以处理该情况。就性能而言，最差的情形是使用弹性缓冲区，该缓冲区位于设备可以访问的地址范围内。32位地址以上的I/O缓冲区内容将必须在设备可以访问该数据之前复制到新的缓冲区中。第二种方案使用现代计算机中特定的硬件电路，将任意内存位置动态重新映射至设备可以访问的“虚拟”物理地址。
- 从用户应用程序的角度来看，尽管内存看起来是连续的，但实际上用户空间内存由可能分散在RAM中的多个物理页组成。假设应用程序要将一个包含大高清视频帧的缓冲区输出到视频设备。因为帧在物理内存中严重分离，仅告诉设备单个缓冲区的长度和地址远远不够。我们需要告诉它组成视频帧的所有分片。因此，不是告诉设备一个缓冲区的位置和长度，而是为其提供一个缓冲区，该缓冲区要包含每个分片位置的列表。这个缓冲区被称为分散/聚集列表。我们稍后将详细讨论该概念。

从驱动程序的角度看，最复杂的是为传输设置和准备内存缓冲区。这需要执行许多步骤。该缓冲区需要被钉住，因为底层内存上的换出操作可能具有灾难性，特别是传输到一个存储设备时。由于一些设备只能访问32位物理地址范围内的内存，所以需要确保支持缓冲区的物理内存位于设备可以访问的范围内，或者必须确保它可以被复制或重新映射。然后需要计算出支持缓冲区物理内存的各个段，并得到每个段的物理地址及长度，创建一个分散/聚集列表。根据设备的功能，如果对对齐有特殊要求或要限制单个段的长度，情况可能会更复杂。一个简单的分散/聚集列表如图9-5所示。

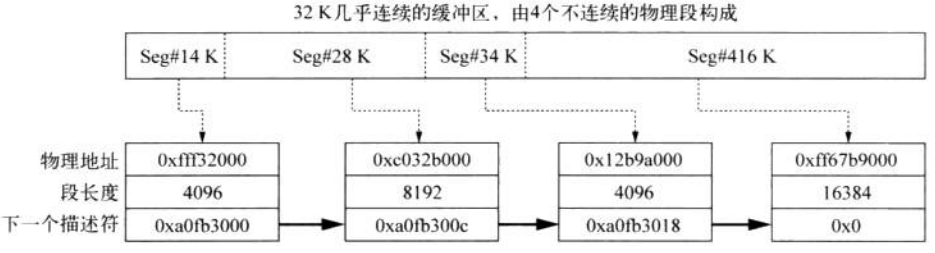



图9-5　简单分散/聚集列表

实际的实现可能更复杂，并且会包含与每个描述符相关的额外数据，但我们会用简单的方式说明该概念。图9-5展示了一个32 KB虚拟缓冲区由4个不同长度的物理内存段组成。分散/聚集列表是一个DMA描述符元素数组，每个元素包含一个指向列表中下一个描述符的指针。每个元素都有它表示的物理段的地址及长度。DMA传输开始时，可以只告诉设备第一个描述符的位置，设备便会从第一个描述符开始读取内存，然后根据下一个指针找到下一个描述符元素，直至列表结尾，在这种情况下，结尾是一个NULL指针。一些设备的分散/聚集列表会将最后一个描述符连接到第一个描述符，为连续的（流）DMA创建一个环形缓冲区。

在图9-5中，我们使用了一个数据结构，用计算机科学与工程术语说就是单链表。也许将列表实现为标准数组更容易。但单链表方案更灵活，因为分散/聚集列表本身就可以高效地分散-聚集，每个DMA描述符成员无须在内存中彼此相邻。

### 9.3.1　将物理地址转换为总线地址

现代计算机系统使用一个特定的MMU，被称为IOMMU或I/O内存管理单元。IOMMU类似于为CPU提供虚拟内存到物理内存转换的系统MMU，但不同点在于IOMMU还可实现硬件设备的转换。使用IOMMU时，硬件设备将使用IOMMU提供的地址，而不是直接使用物理地址。我们通常使用术语*总线地址*（Bus Address）以避免与物理地址混淆。IOMMU在概念上与系统交互的方式如图9-6所示。

不使用IOMMU时，硬件设备将使用与CPU相同的物理地址。IOMMU具备从安全到性能的诸多优点，并能解决前面提到的一些问题，如DMA传输至限于32位寻址的旧设备。即使物理内存

位于高端内存地址中，IOMMU也可以重新映射内存以便设备能够访问该内存。这有助于提高系统性能，因为另外一个解决方案是设置一个“回弹缓冲区”（bounce buffer），我们可以向该缓冲区复制内容并从中执行DMA，而原始缓冲区位于设备无法访问的地址中。从安全性/稳定性的角度看，IOMMU的工作方式类似于受保护/虚拟内存在不同任务之间的工作方式。PCI设备通常可以完全访问硬件，因此，如果驱动程序或设备出错，可能会破坏内存上的某些部分。IOMMU可以映射受限制的窗口，防止访问该窗口之外的地址。IOMMU通常用于PC服务器上的虚拟化，因为它可以使硬件在各虚拟机实例之间共享，而不受干扰，并可以防止流氓驱动程序向其他虚拟机实例中内存部分执行DMA传输，这会引起严重的安全问题。

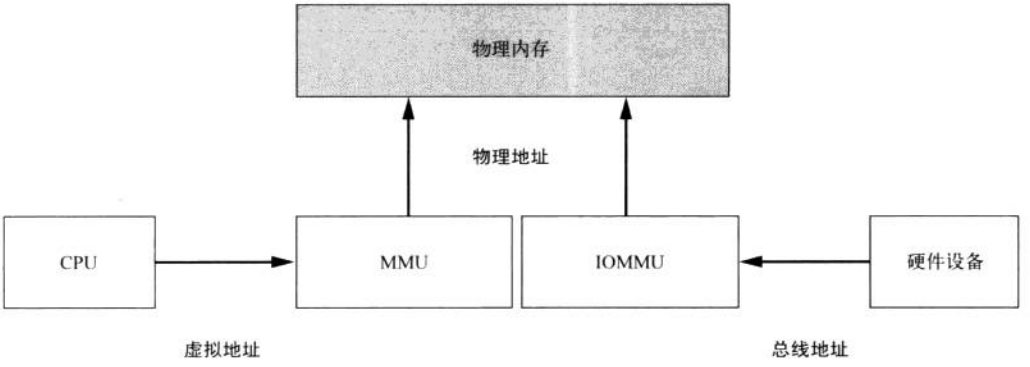



图9-6 IOMMU地址转换

Mac OS X会利用现有的IOMMU。IOMMU由IOMapper类的子类表示，因此，可以在IORegistryExplorer中对其进行搜索，以确定系统中是否存在IOMMU。幸好，我们从来不必直接处理IOMMU。IOMemoryDescriptor和IODMACommand（本章后面进行讨论）等类对此进行了内部设置，我们完全不必关注来自函数（如getPhysicalAddress()）的地址是IOMMU映射的总线地址，还是实际的物理地址。尽管这么做几乎没有理由，但你可以实现IOMapper的子类，自己处理地址转换，并将其提供给像IODMACommand这样的类。IOMMU通常仅用于高性能工作站及服务器，但现在也出现在消费级平台中，如英特尔的Core i7。

### 9.3.2 为DMA准备内存

进行DMA传输之前，还需要执行一些操作，为传输准备内存。首先是确保内存换入至常驻内存，并且支持该缓冲区的内存页被锁定（钉住），因此它们在DMA传输期间不会被换出。为此，需要为该缓冲区创建一个IOMemoryDescriptor。内存描述符必须使用缓冲区所属的任务进行构建，如果内核拥有该缓冲区，则为kernel_task，否则为用户空间进程的任务指针。如果此时知道传输方向，则可以在构建描述符时传递该方向。如果将内存传输到设备，则方向为kIODirectionOut，如果从设备传输内存，则为kIODirectionIn。还有kIODirectionInOut，可用

于需要在两个方向上进行传输的缓冲区。指定正确的方向非常重要，因为前面讨论过，这可能对缓存一致性有隐含约束。

如果需要从一个内核缓冲区进行DMA传输，建议使用IOBufferMemoryDescriptor，这是IOMemoryDescriptor的子类，同样可以进行内存分配。

IOMemoryDescriptor的prepare()方法执行内存换入，并将其钉住。如果在描述符初始化时未指定传输方向，那么可以选择将DMA传输方向传递给prepare()。

---

**警告**　IOMemoryDescriptor::prepare()与IOMemoryDescriptor::complete()必须成对调用。对未预先准备或准备不成功的描述符调用complete()会产生程序错误。

---

### 9.3.3　建立分散/聚集列表

建立分散/聚集列表的方式有许多。最基本的是使用IOMemoryDescriptor::getPhysicalSegment()枚举底层物理段，如代码清单9-11所示。

**代码清单9-11　从缓冲区获取物理段**

```
IOBufferMemoryDescriptor* fDMABuffer =
IOBufferMemoryDescriptor::inTaskWithOptions(kernel_task, kIODirectionOut, 1024 * 1024, 4096);
IOByteCount offset = 0;
while (offset < fDMABuffer->getLength())
{
    IOByteCount segmentLength = 0;

#ifdef __LP64__
    addr64_t address = fDMABuffer->getPhysicalSegment(offset, &segmentLength);
    // 在实际的驱动程序中，会将地址和长度存储到分散/聚集列表中
    // 现在只是将它记录在这里
    IOLog("Physical segment: address 0x%llx segmentLength: %llu\n", address, segmentLength);
#else
    addr64_t address = fDMABuffer->getPhysicalSegment(offset, &segmentLength,
kIOMemoryMapperNone);
    IOLog("Physical segment: address 0x%llx segmentLength: %lu\n", address, segmentLength);
#endif
    offset += segmentLength;
}
```

代码清单9-11的输出类似如下内容：

```
Jun 3 22:28:12 macpro kernel[0]:  Physical segment: address 0x13837000 segmentLength: 4096
Jun 3 22:28:12 macpro kernel[0]:  Physical segment: address 0x143b6000 segmentLength: 4096
Jun 3 22:28:12 macpro kernel[0]:  Physical segment: address 0x1c035000 segmentLength: 4096
…
…
Jul 3 22:28:12 macbook kernel[0]: Physical segment: address 0x14172000 segmentLength: 4096
```

在前面的输出中，并没有连续的段，每个段由单独的页（4096字节）组成。但如果在分配缓冲区时传递kIOMemoryPhysicallyContiguous标志，则可以得到如下输出：

```
Jun 3 22:21:08 macpro kernel[0]:  Physical segment: address 0x5975000 segmentLength: 1048576
```

在驱动程序中连续地分配内存并不是一个好主意；详细原因请参见第6章介绍IOBufferMemoryDescriptor的部分。此处这样做是为了演示需要。

根据设备的功能，代码清单9-11的方案可能表现很好，但该技术存在一些问题。

- 一些硬件设备对它们在分散/聚集列表中可以处理的物理段的最大或最小长度有限制。
- 设备可能要求某一长度的段，例如页长度的段，因此，各个段需要匹配硬件的缓冲区。在这种情况下，我们可能需要将较大的段手工分成较小的块。
- 许多硬件设备都处理大端序寻址。因此，我们需要对物理地址手工进行字节交换，以确保设备能访问正确的位置。

### 9.3.4 IODMACommand类

IODMACommand可以解决前面讨论的与DMA相关的许多问题。它可以自动将段划分为硬件所需的正确大小，并确保设备能够在正确的范围内只进行32位寻址。这使用IOMMU重新映射地址（如果存在IOMMU），或使用变通方案（如果不存在IOMMU）实现。它还能够提供采用小端序或大端序格式的32位或64位物理地址或总线地址。IODMACommand类取代了IOMemoryCursor类，虽然IOMemoryCursor类执行许多同样的功能，但它提供的选项较少。

IODMACommand的实例可以使用如下工厂方法withSpecification()构建。

```
static IODMACommand * withSpecification(
    SegmentFunction outSegFunc,
    UInt8 numAddressBits,
    UInt64 maxSegmentSize,
    MappingOptions mappingOptions = kMapped,
    UInt64 maxTransferSize = 0,
    UInt32 alignment = 1,
    IOMapper *mapper = 0,
    void *refCon = 0);
```

我们详细地看一下各个参数。第一个参数是指向输出段信息函数的函数指针。如果你的硬件设备比较罕见，则可以编写自己的函数，但大多数情况下都可以使用提供的函数：

- kIODMACommandOutputHost32以主机字节顺序输出32位地址；
- kIODMACommandOutputBig32以大端序格式输出32位地址；
- kIODMACommandOutputLittle32以小端序格式输出32位地址；
- kIODMACommandOutputHost64以主机字节顺序输出64位地址；
- kIODMACommandOutputBig64以大端序格式输出64位地址；
- kIODMACommandOutputLittle64以小端序格式输出64位地址。

下一个参数numAddressBits可以指定硬件能够寻址的地址位最大数，地址位不一定总是32位

或64位。例如，可以是36位，在某些情况下甚至小于32位。如果传递的值大于32位，则必须指定一个64位输出段函数。

如果物理页所在的地址超出了设备可以寻址的范围，可以将某些或所有的页复制到临时页，以满足地址要求，除非存在IOMMU。不用说，额外的复制代价很高，为了避免复制，可以从由IOBufferMemoryDescriptor使用inTaskWithPhysicalMask()工厂方法分配的内存进行DMA，因为这可以使用位掩码指定范围内的物理地址分配内存。这种方法还可以直接将内存分配到用户空间任务的地址空间。这非常有用，因为向/从用户空间分配的缓冲区DMA可能有问题，因为无法控制内存的分配方式。

maxSegmentSize参数应该设置为分散/聚集列表中最大的物理连续段，设备可以将其作为一个段进行处理。如果没有此种限制，那么可以传递0。

mappingOptions参数可以绕过IOMMU（系统中没有IOMMU时）。默认是使用IOMMU。mapper参数可以指定一个备用的IOMapper实例，而不是使用默认实例。

maxTransferSize是每个DMA事务可以传输的字节总数。例如，硬盘可能有1 MB缓存缓冲区，因此，我们无法传输它接受范围之外的信息。

如果硬件有特定的对齐要求，则可以通过alignment参数指定。如果提供的内存并没有正确对齐，同样可能会导致复制或重新映射内存。

我们对代码清单9-11中的示例进行修改，将它使用的IOMemoryDescriptor::getPhysicalSegment()替换为IODMACommand，如代码清单9-12所示。

**代码清单9-12　使用IODMACommand生成分散/聚集列表**

```
IOReturn com_osxkernel_MyFirstPCIDriver::prepareDMATransfer()
{
    IODMACommand*       dmaCommand;
    IOReturn                        ret = kIOReturnSuccess;

    dmaCommand = IODMACommand::withSpecification(kIODMACommandOutputHost64, 36, 2048,
IODMACommand::kMapped, 0, 1);
    if (!dmaCommand)
    {
        return kIOReturnNoMemory;
    }

    //仍将准备内存描述符
    ret = dmaCommand->setMemoryDescriptor(fDMABuffer);
    if (ret != kIOReturnSuccess)
        return ret;

    UInt64 offset = 0;
    while (offset < fDMABuffer->getLength())
    {
        IODMACommand::Segment64 segment;
        UInt32 numSeg = 1;

        ret = dmaCommand->gen64IOVMSegments(&offset, &segment, &numSeg);
```

```
        IOLog("%s::gen64IOVMSegments() addr 0x%qx, len %llu bytes\n",
              getName(), segment.fIOVMAddr, segment.fLength);

        if (ret != kIOReturnSuccess)
            break;
    }

    //
    //在此为实际硬件设备设置DMA传输
    //

    if (dmaCommand->clearMemoryDescriptor() != kIOReturnSuccess)
    {
        IOLog("Failed to clear/complete memory descriptor\n");
    }

    dmaCommand->release();
    return ret;
}
```

在前面的例子中，我们创建了一个IODMACommand，将最大物理地址位约束为36位，最大物理段长度约束为半页，即2048字节。此代码的输出结果如下所示：

```
kernel[0]: com_osxkernel_MyFirstPCIDriver::gen64IOVMSegments() addr 0x13b9d5000, len 2048
kernel[0]: com_osxkernel_MyFirstPCIDriver::gen64IOVMSegments() addr 0x13b9d5800, len 2048
kernel[0]: com_osxkernel_MyFirstPCIDriver::gen64IOVMSegments() addr 0x1335d4000, len 2048
kernel[0]: com_osxkernel_MyFirstPCIDriver::gen64IOVMSegments() addr 0x1335d4800, len 2048
kernel[0]: com_osxkernel_MyFirstPCIDriver::gen64IOVMSegments() addr 0x3f913000, len 2048
kernel[0]: com_osxkernel_MyFirstPCIDriver::gen64IOVMSegments() addr 0x3f913800, len 2048
kernel[0]: com_osxkernel_MyFirstPCIDriver::gen64IOVMSegments() addr 0x3b1d2000, len 2048
kernel[0]: com_osxkernel_MyFirstPCIDriver::gen64IOVMSegments() addr 0x3b1d2800, len 2048
kernel[0]: com_osxkernel_MyFirstPCIDriver::gen64IOVMSegments() addr 0x18b11000, len 2048
kernel[0]: com_osxkernel_MyFirstPCIDriver::gen64IOVMSegments() addr 0x18b11800, len 2048
....
```

首先你会发现，一些地址大于32位但仍然低于规定的36位，并且各段都为2048字节，即使第一个段与第二个段实际上是连续的。

## 9.4 小结

本章讲述了如下内容。

- PCI相关的PCI Express、Thunderbolt及较少使用的ExpressCard标准技术概览。
- PCI配置空间，操作系统可以用来枚举、控制和操作基于PCI的设备。本章后面还讨论了如何使用I/O Kit提供的功能，从驱动程序访问设备。
- I/O Kit中使用IOPCIFamily支持PCI。内核PCI层的基础是IOPCIDevice类，是基于PCI驱动程序的提供者。
- Thunderbolt设备与PCI兼容，由IOPCIDevice类表示。

- 如何为基于PCI的设备创建一个匹配字典，以便驱动程序可以在设备插入或启动时自动加载。
- 如何访问、使用内存映射的I/O内存及I/O内存区。
- 如何处理外部接入的设备移除，特别是对于Thunderbolt和ExpressCard。
- 在I/O Kit中，使用`IOInterruptEventFilter`及相关的类，对主中断和二级中断进行中断处理。
- DMA及如何创建分散/聚集列表，以传输不连续的大内存块。还讨论了与DMA相关的一些常见问题。
- `IODMACommand`用于将内存地址转换为总线地址，以应对创建分散/聚集列表的复杂性。

# 第10章 电源管理 10

电源管理已经成为所有计算设备的基本特征。运行Mac OS X的每个平台都可以进入功率较低的模式，因此，电源管理对需要一直连接电源的台式计算机与靠电池运行的笔记本电脑或iPhone一样重要。

例如，即使Mac Pro也可以进入“睡眠”模式，使计算机及其外设进入低功率模式。在睡眠模式期间，CPU进入暂停状态，计算机的显示器会关闭电源，硬盘低速旋转。如果计算机有PCI扩展槽，这些扩展槽在睡眠期间也会掉电，仅使用提供的少量辅助电源，使PCI卡可以将计算机从睡眠中唤醒。

并不是所有的驱动程序都需要处理电源管理事件。驱动程序是否需要实现电源管理，取决于设备的功能以及如何供电。例如，如果一个PCI设备的驱动程序不支持电源管理，则系统在进入睡眠模式时，必须为其PCI插槽保留全部电源，因为PCI卡需要由计算机主板供电。这使计算机处于“打盹”状态，但并不是完全睡眠模式。请注意，如果PCI插槽的电源在睡眠期间暂停，则PCI设备将失去其配置，并且在系统从睡眠中唤醒时，必须由其驱动程序重新启动，而这只在驱动程序接收到电源管理事件时发生。

正如PCI卡驱动程序示例所示，硬件设备的驱动程序对系统从一种电源状态到另一种状态的转换起到非常重要的作用。驱动程序在系统进入低功率睡眠模式之前，可以选择接收通知，此时驱动程序可以为新的电源状态准备其设备。类似地，当系统从睡眠中唤醒时，驱动程序可以收到一个通知，这时驱动程序可以将其设备恢复至正常运转状态。

### 1. I/O Registry电源平面

电源管理的复杂性在于，通常无法孤立地查看设备的电源状态，因为一个设备通常需要另一个设备供电，并且相应地，后者可能还会为其他设备供电。例如，考虑一下实现USB主机和提供USB端口的PCI外设卡。该卡将由PCI总线供电，并通过USB总线为连接到其端口的设备供电。这对电源管理系统会有一些影响，PCI卡在未连接设备或仅在连接的所有USB设备本身都是低功率状态时，只可以进入低功率状态。类似地，当系统进入睡眠时，PCI插槽将断电，并且还必须通知连接的某些USB设备电源状态已经发生改变。

为了对这种电源依赖进行建模，I/O Kit使用树型结构表示系统中硬件设备之间的电源依赖。该树存储在I/O Registry的段中，而该段则位于一个被称为“电源平面”的平面中。可以使用IORegistryExplore实用工具查看电源平面，如图10-1所示。

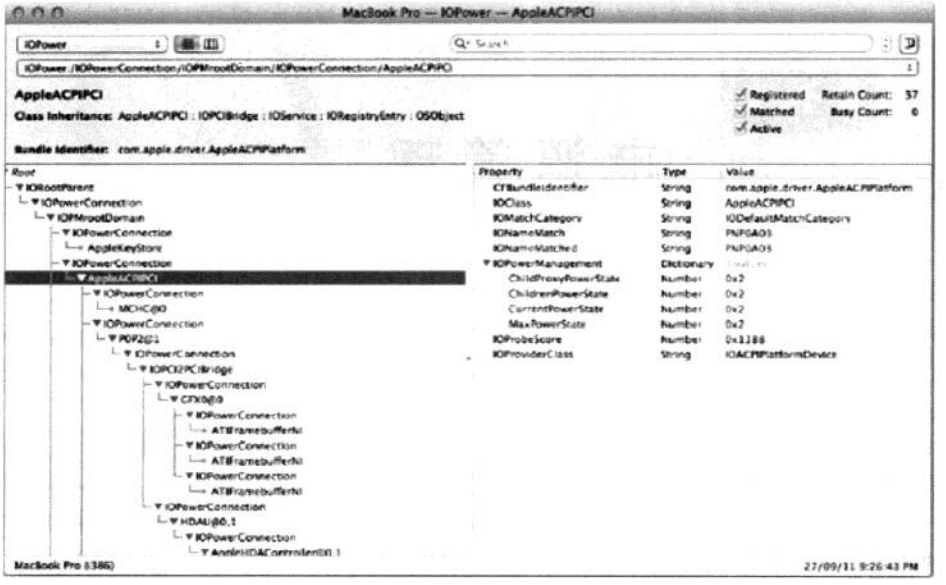



图10-1　IORegistryExplore中的电源平面视图

支持电源管理的每个驱动程序在电源平面中都表示为一个连接到父节点的节点，父节点表示为它供电的设备，并且带有一些子节点，表示需要改节点供电的设备。电源平面中，设备的父设备通常是驱动程序的提供者类（即I/O Registry的服务平面中，驱动程序连接到的同一对象），尽管并不需要如此对应。

树型表示可以非常清晰地说明系统中设备之间的电源依赖，并且可以方便地查看哪些设备需要特定的硬件设备供电。树型结构对系统本身也非常有用，它使系统可以确定哪些设备会在电源状态改变时受到影响。

### 2. I/O Kit中的电源管理

系统的电源管理，包括一种电源状态到另一种状态的转换，由I/O Kit处理。电源管理由运行在内核中的驱动程序执行，为硬件（如USB设备）编写的用户空间驱动程序无法参与设备的电源管理。I/O Kit的IOService超类为电源管理提供支持，所有的Mac OS X驱动程序最终都派生自该类。这使所有的驱动程序都可以访问电源管理，前提是驱动程序已选择将自身加入到电源平面中。

IOService还负责管理父设备与依赖于父设备供电的子设备之间的同步。例如，在转换为低功率状态之前，系统要确保所有的子设备在为其供电的父设备接收到降低其电源状态的请求之前，已经转换为低功率状态。类似地，当从睡眠中唤醒时，系统要确保父设备在向依赖于它的子设备发送提高电源状态的请求之前，已经得到正常供电。

从一种电源状态转换到另一种电源状态极其困难，因为硬件需要花费一些时间才能变成新的状态。这会导致一个电源状态转换已经开始，而现有的电源状态转换仍在处理中。例如，硬盘可

能由于不活动而降低旋转速度，但随后可能立即有一个读请求，要求硬盘再次提高旋转速度。IOService类可以处理其他方式实现起来比较困难的许多电源管理细节。IOService使每个电源状态转换都串行化，驱动程序处理完上一个电源请求之前不再接收电源请求。

IOService类为电源管理提供支持是指，驱动程序仅完全关注它自己设备的电源管理，而采用其他方式实现起来比较困难的电源管理的细节全部交由I/O Kit框架处理。

改变电源状态有两种方式。系统可能要求将设备置于新的电源状态以响应进入睡眠状态的计算机，或者设备（或其驱动程序）会启动其电源状态的改变。设备启动低功率状态转换的一个例子是，计算机已经有一段时间没有使用，或硬盘在一段时间内没有接收到读/写请求而降低旋转速度时，关闭计算机显示器。

驱动程序可以选择支持任意一种类型的电源管理，可以选择响应系统电源的改变，如睡眠事件，或者可以自愿降低其本身的电源状态，如减慢磁盘旋转速度。这两种情况都通过I/O Kit进行处理，本章会对此进行详细讨论。

通常，电源管理对某些设备来说很有必要，这些设备不仅有硬件支持进入低电源状态，而且根据使用方式，也有机会进入低电源状态。根据驱动程序管理的硬件设备类型，支持各种电源状态之间的转换可能不需要很多工作。

---

**注意** 本章将讲述通用I/O Kit驱动程序处理电源管理请求所需的工作。某些I/O Kit驱动程序族（如音频族）会将其驱动程序添加到电源平面，代表驱动程序响应电源管理请求。在为驱动程序添加电源管理支持之前，应该检查确保该驱动程序的超类尚未处理电源请求。

---



## 10.1 响应电源状态改变

驱动程序可以实现的电源管理支持的最基本级别是，能够接收改变整个系统电源状态的通知。这些通知包括系统进入睡眠之前的通知，以及系统从睡眠中唤醒时的通知。这对PCI设备的驱动程序来说特别重要，因为需要处理这些通知才能使PCI总线在系统睡眠期间完全断电。

I/O Kit的电源模型是唯一的，因为该框架并不定义驱动程序必须实现的电源状态集，而是驱动程序定义其自身的电源状态列表，匹配它控制的设备功能。一个驱动程序最少必须定义两种状态：一种是设备关闭的状态，另一种是设备完全运行的状态。关闭状态期间，设备并不消耗电量（因此，在电源平面中，不需要父设备供电）。处于这种状态时，设备不可用。处于打开状态时，设备正在供电，它所有的功能对用户都可用。当系统进入睡眠时，设备进入关闭状态，而当设备唤醒时，设备则进入打开状态。

除了关闭和打开状态，驱动程序还可以定义其他一些电源状态，在这些状态中，例如，设备仍然供电，但处于低电源状态，那么设备仍然可用，但只有部分功能可用。例如，液晶显示器的驱动程序可以创建一种电源状态，表示显示器仍然运行但背光已经变暗的模式。

设备支持的每种电源状态由IOPMPowerState结构体表示，该结构体在<IOkit/pwr_mgt/IOPMpowerState.h>头文件中定义。IOPMPowerState结构体的定义如下所示：

```
struct IOPMPowerState
{
        unsigned long   version;
        IOPMPowerFlags  capabilityFlags;
        IOPMPowerFlags  outputPowerCharacter;
        IOPMPowerFlags  inputPowerRequirement;
        unsigned long   staticPower;
        unsigned long   unbudgetedPower;
        unsigned long   powerToAttain;
        unsigned long   timeToAttain;
        unsigned long   settleUpTime;
        unsigned long   timeToLower;
        unsigned long   settleDownTime;
        unsigned long   powerDomainBudget;
};
```

下面是该结构体各字段的描述。

- version：保存IOPMPowerState结构体的版本，支持在将来的I/O Kit版本中扩展该结构体，保持后向兼容。就Mac OS X 10.7而言，该结构体仍然是版本1，该头文件提供了可用的kIOPMPowerStateVersion1定义。
- capabilityFlags：标志的位掩码，描述此电源状态下的设备功能。可能的标识有：
  - kIOPMPowerOn表示设备需要其父设备供电，并可以为其子设备供电；
  - kIOPMDeviceUsable表示设备在这种状态下可用；
  - kIOPMLowPower表示设备在比kIOPMPowerOn状态耗电少的电源状态下运行。设备在这种状态下可能仍然可用，这可以通过设置kIOPMDeviceUsable及kIOPMLowPower位来表示。设备在低电源状态时，可能会为其子设备供电，也可能无法为其子设备供电；
  - kIOPMPreventIdleSleep设置后，当该电源处于活动状态时会禁止系统睡眠。请注意，用户仍然可以让系统睡眠（如从苹果菜单选择“睡眠”）。这只是防止系统在一段时间不活动之后自动睡眠；
  - kIOPMInitialDeviceState表示设备在这种状态下启动，因此，驱动程序无须在加载之后发送电源请求。请注意，I/O Kit可能会在没有kIOPMInitialDeviceState标志集的电源状态下启动，在这种情况下，驱动程序将在加载时接收到一个电源请求。
- outputPowerCharacter：该标志描述设备在这种状态下，可以为需要它供电的设备供电。其值可以是kIOPMPowerOn，表示设备能为其子设备供电，也可以是0，表示设备不能为其子设备供电。
- inputPowerRequirement：该标志描述设备在这种状态下，要求其父设备供电。其值可以是kIOPMPowerOn，表示设备要求父设备为其供电，也可以是0，表示设备在这种状态下不需要其父设备供电。
- staticPower：在这种状态下，设备的平均耗电量（单位是微瓦）。请注意，如果该值未知，则驱动程序可以为该字段提供一个0值（正如Darwin存储库中许多苹果驱动程序所做的那样）。

- unbudgetedPower：在这种状态下，设备从单个电源获得的电量。该值目前在Mac OS X中尚未使用，因此，驱动程序可以为该字段提供一个0值。
- powerToAttain：设备从前一种低电源状态转换到这种状态所需要的电量。该值目前在Mac OS X中尚未使用，因此，驱动程序可以为该字段提供一个0值。
- timeToAttain：将设备从前一种低电源状态转换到这种状态所需要的时间（单位是微秒）。如果该值未知，则驱动程序可以为该字段提供一个0值。
- settleUpTime：在从前一种低电源状态进入这种状态之后，电源调整所需要的时间。如果该值未知，则驱动程序可以为该字段提供一个0值。
- timeToLower：将设备从这种状态转换为下一种低电源状态所需要的时间（单位是微秒）。如果该值未知，则驱动程序可以为该字段提供一个0值。
- settleDownTime：在离开这种状态并进入下一种低电源状态之后，电源调整所需要的时间。如果该值未知，则驱动程序可以为该字段提供一个0值。
- powerDomainBudget：设备在这种状态下，可以为其子设备提供的电量。该值目前在Mac OS X中尚未使用，因此，驱动程序可以为该字段提供一个0值。

设备支持的每种电源状态都由IOPMPowerState结构体描述。设备的驱动程序创建一个IOPMPowerState结构体数组，每个元素对应一个设备电源状态。每个支持电源管理的驱动程序都必须包含两种电源状态，一种对应关闭状态的电源状态（在这种状态下，设备不使用电源），一种对应设备完全打开状态的电源状态（在这种状态下，设备全面启动）。在驱动程序的电源状态数组中，关闭状态必须是第一个成员，而打开状态必须是最后一个成员。驱动程序可以定义与其硬件设备提供的不同电源状态相同的电源状态数，唯一的要求是电源状态数组必须有序，它以关闭状态开始，经过电量需求更多、功能更强的（可选）中间电源状态，到达最后一个状态，在最后一个状态中，设备在满额电源下运行，并且全面启动。

因为I/O Kit在IOService类中提供对电源管理的支持，而该类是每个驱动程序的最终父类，因此所有的驱动程序都可以参与系统的电源管理。为了演示，对第4章开发的简单IOKitTest示例添加电源管理通知（参见代码清单4-2和代码清单和4-3）。

首先，需要定义设备支持的电源状态。这通常由使用IOPMPowerState结构体的一个全局数组来完成，该全局数组在驱动程序实现文件的顶部定义。

**代码清单10-1 为驱动程序定义一组电源状态**

```
enum {
    kOffPowerState,
    kOnPowerState,
    //
    kNumPowerStates
};

static IOPMPowerState gPowerStates[kNumPowerStates] = {
    // kOffPowerState
    {kIOPMPowerStateVersion1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

```
    // kOnPowerState
    {kIOPMPowerStateVersion1, (kIOPMPowerOn | kIOPMDeviceUsable),
                              kIOPMPowerOn, kIOPMPowerOn, 0, 0, 0, 0, 0, 0, 0, 0}
};
```

I/O Kit请求变为新的电源状态时，请求的状态将通过gPowerStates数组中的状态索引来确定。我们不是通过其索引指定电源状态，而是定义了一个枚举类型，为每个电源状态赋一个符号常量，使驱动程序的代码更易于阅读及维护。

已经定义了电源状态集，提供电源管理API的IOService类接下来需要通知驱动程序，希望在系统的电源状态改变时，收到通知。这在驱动程序的start()方法中实现，如代码清单10-2所示。

**代码清单10-2　为电源管理支持注册一个驱动程序**

```
bool com_osxkernel_driver_IOKitTest::start (IOService *provider)
{
        if (super::start(provider) == false)
                return false;

        //为驱动程序注册电源管理
        PMinit();
        provider->joinPMtree(this);
        registerPowerDriver(this, gPowerStates, kNumPowerStates);

        return true;
}
```

调用PMinit()方法，初始化IOService超类中仅用于实现电源管理的实例变量。为了接收电源管理通知，该驱动程序必须是电源平面的一部分。这通过joinPMtree()方法实现，在为设备供电的驱动程序对象上调用该方法（这种情况下，是指我们的提供者类），并将子驱动程序作为其参数（我们的实例）。最后，调用registerPowerDriver()方法，为I/O Kit提供驱动程序支持的电源状态数组。

注册了电源管理的驱动程序必须确保在卸载之前，从电源平面移除自己。否则，I/O Kit将试图向该驱动程序发送电源通知，即使该驱动程序不再处于活动状态，这可能会导致内核恐慌。驱动程序调用PMStop()从电源平面移除自己，如代码清单10-3所示。

**代码清单10-3　从电源管理系统移除驱动程序**

```
void com_osxkernel_driver_IOKitTest::stop (IOService *provider)
{
        PMstop();
        super::stop(provider);
}
```

将驱动程序插入到电源平面，并为其注册了电源管理事件之后，驱动程序便可以接收来自I/O Kit的电源请求，响应系统中的改变。当系统的电源状态改变时，例如计算机进入睡眠状态或从睡眠中唤醒时，I/O Kit将选择驱动程序注册的一种电源状态，并请求转换为这一新的状态。这些电源请求通过setPowerState()方法发送给驱动程序，该方法是IOService基类中定义的一个虚拟

方法。为了接收这些请求，驱动程序只需提供自身的setPowerState()实现，在其中处理状态改变。代码清单10-4给出了一个示例实现。

**代码清单10-4 响应改变设备电源状态的请求**

```
IOReturn com_osxkernel_driver_IOKitTest::setPowerState (unsigned long powerStateOrdinal, IOService*
device)
{
        switch (powerStateOrdinal)
        {
                case kOffPowerState:
                        //保存设备配置（如有必要），并为硬件进入睡眠状态做准备
                        // ...
                        break;
                case kOnPowerState:
                        //唤醒硬件，并使用保存的配置进行初始化
                        // ...
                        break;
        }


        return kIOPMAckImplied;
}
```

powerStateOrdinal参数表示驱动程序应该将其设备置于何种电源状态。该参数为电源状态数组的索引，驱动程序在初始化时将该索引传递给registerPowerDriver()方法。就我们的示例驱动程序而言，我们注册了两种电源状态，用switch语句对其进行处理。



I/O Kit串行化调用setPowerState()，因此，驱动程序可以确保在处理上一个电源状态转换的过程中，不会再接收改变电源状态的请求。但这并不能保证驱动程序在处理与电源管理不相关的其他操作时，不接收改变电源状态的请求。例如，驱动程序可能正在执行一个异步读操作，而电源请求要求将驱动程序转换为睡眠状态。这种情况下，在设备掉电之前，驱动程序必须等待，直至读操作完成。这可以使用I/O Kit提供的标准同步原语实现。大部分I/O Kit驱动程序使用命令门与工作环的组合实现同步，驱动程序从而可以在其setPowerState()方法中获得命令门，确保电源事件与其他的驱动程序代码同步。

从Mac OS X 10.5起，setPowerState()方法就在它自身的线程上调用，因此，驱动程序能够在其实现中执行可能会阻塞或耗时的操作。当驱动程序已经成功将硬件置于新的电源状态时，会返回结果码kIOPMAckImplied。

如果驱动程序支持Mac OS X 10.5之前的版本，那么请不要在setPowerState()方法中执行阻塞操作，相反，驱动程序应该在后台线程上执行将硬件置于新电源状态所必需的任务。驱动程序并不是从setPowerState()方法返回kIOPMAckImplied，而应该返回一个非0值，表示驱动程序将硬件置于新状态所需的最大时间（单位是微秒）。当后台线程已经切换至新的设备时，会调用acknowledgeSetPowerState()通知完成。如果你面向的是Mac OS X 10.5或后续版本，谢天谢地，你不需要这些代码。

## 10.2　请求电源状态改变

到目前为止，我们已经了解驱动程序如何响应系统的请求，改变其电源状态，响应系统进入睡眠状态等事件。但有时驱动程序希望独立于系统整体，改变设备自身的电源状态。例如，液晶显示器可能在不活动的几分钟之后，背光变暗，或者磁盘在某段时间未被访问时，降低旋转速度。

驱动程序应该使用I/O Kit的电源管理API，即使电源状态的改变仅影响其控制的设备。这样做不仅可以确保驱动程序执行的电源状态改变与系统请求的改变同步，而且可以使驱动程序利用I/O Kit对某些任务（如安装一个定时器，监控设备活动）的支持，并在设备一段时间未被访问时，请求将其转换为低电源状态。最后，如果某些设备需要你的硬件供电，那么要使用I/O Kit方法，转换硬件的电源，以便通知所有子设备，其输入电源有可能发生的改变。

改变设备当前电源状态的调用方法有3个。

- changePowerStateTo(powerStateOrdinal)：请求改变注册电源状态数组中指定索引的电源状态。
- changePowerStateToPriv(powerStateOrdinal)：功能与上一个方法类似，区别在于，这是IOService类中受保护的方法，不可以由驱动程序本身以外的对象调用。
- makeUsable()：请求驱动程序支持的最高电源状态的电源改变。该方法通常由驱动程序的另一个客户端调用（如驱动程序的用户客户端），确保在进一步使用设备之前，设备功能完备。

这3个方法由IOService类实现，通常无须驱动程序覆写这些方法。在内部，makeUsable()方法的实现与changePowerStateToPriv()调用相同的代码路径，这意味着，每个驱动程序都有两个与其关联的电源状态：通过changePowerStateTo()请求的值和通过changePowerStateToPriv()/makeUsable()请求的值。

I/O Kit最终将设备切换到的状态是，changePowerStateTo()与changePowerStateToPriv()请求值中最大的值，并满足依赖该设备供电的所有子设备要求的最大状态。如果设备在电源平面中有需要它供电的子设备，那么该父设备不可以置于包含outputPowerCharacter属性但不是kIOPMPowerOn的电源状态。

你可能很奇怪，为什么I/O Kit会提供两个几乎相同的方法来设置设备的电源状态。私有方法changePowerStateToPriv()可以使驱动程序设置最低的电源级别，不受驱动程序任何客户端的影响，而这些客户端只可以调用公共的changePowerStateTo()方法。客户端可能会将电源状态提高至changePowerStateToPriv()设置的级别之上，而驱动程序的电源状态从来都不会低于changePowerStateToPriv()设置的值。而系统处于睡眠状态时是一个例外，此时，设备将覆盖changePowerStateToPriv()已经设置的电源状态，进入最低电源状态。当系统从睡眠中唤醒时，将返回至之前活动的电源状态。

按照惯例，驱动程序应该通过受保护的方法changePowerStateToPriv()设置其电源状态。为了消除公共电源级别带来的影响，驱动程序在注册电源管理之后，应该在其start()方法中调用公共的方法changePowerStateTo(0)。将公共电源状态设置为0，便可以应用changePowerState-

ToPriv()请求的电源状态，而无须改变（假定可以满足所有子设备的电源请求）。

因为设备的电源状态有3个可能的值，所以每当设备的一个供电子设备的电源状态改变，或者设备调用changePowerStateTo()或changePowerStateToPriv()时，都会重新计算电源状态。如果计算的电源状态与驱动程序当前的电源状态不同，那么I/O Kit将向该驱动程序发送一个请求，改变其电源状态。驱动程序将会接收该请求，并响应该请求，具体细节10.1节已经讲述。

changePowerStateTo()、changePowerStateToPriv()和makeUsable()方法均为异步，可以在设备转换至新的电源状态之前返回给调用者。这意味着希望改变其自身电源状态的驱动程序（例如，使用changePowerStateToPriv()方法）在将其硬件重新编程为新的电源状态之前，应该等待，直至调用其setPowerState()方法。客户端希望调用一个公共方法（如changePowerStateTo()或makeUsable()），去改变另一个驱动程序电源状态的场合，调用的方法返回时，该客户端无法判断设备是否运行在新的电源模式中。相反，客户端应该进行注册，以接收设备状态改变时的通知。这将在10.4节讨论。

## 10.3 处理设备空闲

驱动程序降低其设备电源状态的常见原因是，设备在某段时间内未被访问时，会降低其电量消耗。这需要为设备的空闲时期创建一个定时器。如果定时器到期，并且设备在这段时期内未被访问，则驱动程序将设备置于低电源状态。在设备处于低电源状态之后，驱动程序下一次需要访问该设备时，需要将该硬件置回可用状态。因为这些操作适用于执行空闲电源节省模式的所有驱动程序，所以该功能内置于I/O Kit中，通过IOService类提供给驱动程序开发者。

当设备空闲，并降低其电源状态时，驱动程序需要调用下列两个基本方法，以便I/O Kit追踪。

- setIdleTimerPeriod(period)：安装一个定时器，在指定秒数之后过期。
- activityTickle(type, powerState)：每次访问设备之前，由驱动程序调用，通知I/O Kit硬件的最后访问时间。

setIdleTimerPeriod()方法通常只调用一次，是在驱动程序的start()方法中电源管理初始化之后。调用该方法之后，I/O Kit便会创建一个定时器，以特定的超时时间运行（单位是秒）。如果在这段时间内设备未被访问，I/O Kit则将驱动程序的电源状态降至当前的电源级别之下。如果设备在下一个空闲周期内仍处于不活动状态，则再次降低驱动程序的电源状态。这两种情况下，驱动程序都会接收到一个请求，通过同一个方法setPowerState()来降低其电源状态，该方法用于发送所有的电源请求改变，如10.1节所述。这会一直持续到设备置为关闭状态（电源状态0）。

设备的电源状态因不活动而降低时，I/O Kit使用changePowerStateToPriv()方法设置电源状态。正如驱动程序本身已经调用了changePowerStateToPriv()，设备的新电源状态不能低于公共方法changePowerStateTo()设置的值，或其子设备所需的电源状态。这意味着，通过changePowerStateTo()设置的公共电源级别可以确定设备空闲时能够置于的最小电源状态。通常，使用空闲定时器的驱动程序将在其start()方法中调用changePowerStateTo(0)，从而使空闲定时器可以将设备一直向下设置，直至变为关闭状态。

空闲定时器要求驱动程序每次访问硬件设备时，都要通知I/O Kit，否则设备使用时，会触发空闲定时器，并降低设备的电源状态。为此，设备使用时，驱动程序通常会在每次操作开始时调用activityTickle()方法。activityTickle()方法的签名如下所示：

```
bool    activityTickle(unsigned long type, unsigned long stateNumber);
```

activityTickle()方法有两个参数：类型和电源状态序号，调用者用这两个参数来指定设备处理即将开始的操作时，必须处于的最小电源级别。I/O Kit在头文件<IOkit/pwr_mgt/IOPM.h>中，为type参数提供了两个预定义的值：

- kIOPMSubclassPolicy
- kIOPMSuperclassPolicy1

type参数指定了哪一个实现应该处理activityTickle()请求，驱动程序本身（此情况下传递kIOPMSubclassPolicy）或IOService超类（此情况下传递IOPMSuperclassPolicy1）。如果驱动程序希望提供activityTickle()的一个自定义实现，并传递kIOPMSubclassPolicy作为activityTickle()调用的type参数，则需要覆写activityTickle()方法的实现。

但大多数驱动程序都可以使用IOService超类提供的默认activityTickle()实现。使用默认实现的驱动程序，每次调用activityTickle()时，传递的type参数应该是kIOPMSuperclassPolicy1值，这非常重要，因为IOService实现将会忽略type参数为任何其他值的请求。

activityTickle()的默认实现会将设备的电源级别提高至stateNumber参数指定的电源状态。在内部，IOService类会调用changePowerStateToPriv()方法，提高设备的电源状态，驱动程序将调用其setPowerState()方法进行接收。activityTickle()方法返回的布尔值表示设备是否已经处于请求的电源状态中，返回true表示无须转换电源状态，而返回false则表示需要提高设备的电源状态。

由于activityTickle()方法属异步，所以调用者要一直等到驱动程序完成电源状态的转换，而不应该认为activityTickle()方法返回时设备便处于可用状态，这一点非常重要。驱动程序可以观察驱动程序中的电源改变，包括其他驱动程序对象中的电源改变，这将在下一节讲述。

## 10.4　观察设备电源状态改变

I/O Kit支持驱动程序观察系统中任意设备的电源状态，并在设备的驱动程序改变其电源状态时接收通知。例如，这可以由驱动程序使用，而该驱动程序不包括在电源平面之内，但却需要与管理电源的驱动程序进行交互。另外，对于通过changePowerStateToPriv()或activityTickle()等方法改变自身电源状态的驱动程序，这些通知可以使它确定电源改变何时完成。

为了在另一个驱动程序的电源状态改变时接收通知，你的驱动程序必须注册感兴趣的驱动程序的电源状态。IOService超类为此提供了两个方法：

- registerInterestedDriver(IOService* driver)
- deRegisterInterestedDriver(IOService* driver)

这两个方法都可以在你希望观察其电源状态的驱动程序对象上调用。参数是接收通知的驱动程序，因此，通常可以传递this指针。驱动程序可以在任何时候调用这两个方法，以开始、停止

接收另一个驱动程序电源状态改变的通知。重要的是，驱动程序在卸载之前必须注销所有通知，否则会导致内核恐慌。

驱动程序已经注册了它感兴趣的另一个驱动程序的电源状态改变时，它将会在设备开始向新的电源状态转换时，接收到一个通知，并在转换完成之后，接收到另外一个通知。这两个通知通过以下两个方法发送：

- IOReturn powerStateWillChangeTo(IOPMPowerFlags capabilities, unsigned long stateNumber, IOService* whatDevice)
- IOReturn powerStateDidChangeTo(IOPMPowerFlags capabilities, unsigned long stateNumber, IOService* whatDevice)

这两个通知方法都是虚拟方法，由希望接收通知的驱动程序实现。powerStateWillChangeTo()通知是在调用所观察驱动程序的setPowerState()方法之前发送。powerStateDidChangeTo()通知是在调用所观察驱动程序的setPowerState()方法之后发送。这两个方法传递的参数相同。Capabilities参数是IOPMPowerState的capabilityFlags的位掩码值，该状态是观察的驱动程序转换到的状态。stateNumber参数是观察的驱动程序转换到的电源状态索引。whatDevice参数是电源状态正在改变的驱动程序对象。

处理完任意一个通知之后，驱动程序都应该返回IOPMAckImplied值。如果你的驱动程序希望异步处理通知，则该通知方法会返回一个非0值，表示驱动程序完成请求所需的最大时间数（单位是微秒）。驱动程序然后可以在后台进程继续处理该通知，驱动程序完成该通知之后，应该调用acknowledgePowerChange()方法，通知I/O Kit该通知已经处理完毕。可以调用acknowledgePowerChange()方法，获取willChange及didChange通知。

处理电源管理的驱动程序会自动注册本身，因此，可以对响应其电源状态改变的驱动程序调用这两个通知方法。但大多数驱动程序无须实现它们，除非驱动程序观察另一个设备的状态，因为驱动程序本身的电源改变应该在setPowerState()方法中处理。相反，当驱动程序本身的电源状态已经改变，并且其所有的子驱动程序已经获悉该电源改变时，I/O Kit会为此提供另外一个通知方法，发送给驱动程序。为了接收此通知方法，驱动程序应该实现下面这个虚拟方法：

```
void    powerChangeDone (unsigned long previousStateNumber);
```

该方法将在驱动程序已经处理完电源状态改变（通过setPowerState()方法），并且通知了已经注册对设备电源状态感兴趣的所有驱动程序之后发送。powerChangeDone()方法为驱动程序确定它启动的电源状态改变何时完成及设备何时变为可用，提供了一种便捷方式。要注意，传递给powerChangeDone()方法的参数是设备改变之前的电源级别，而不是新的电源状态。为了确定设备当前的电源状态，I/O Kit提供了一个存取方法getPowerState()，其定义如下：

```
UInt32  getPowerState(void);
```

## 10.5 汇总

本节将结合本章前面讲述的内容，完成一个示例，演示驱动程序的构建方式，该驱动程序不

仅可以响应系统的电源状态改变，而且当设备空闲5分钟时，会降低自身的电源状态。

为了演示，该示例驱动程序定义了4个电源状态，包括强制关闭及打开状态，以及两个低功率模式。关闭和打开状态在计算机进入睡眠、从睡眠中唤醒时，由电源管理系统设置。当设备空闲一段时间时，便会达到这两个中间状态；驱动程序在其start()方法中设置了一个空闲定时器，在一段时间不活动之后，便会降低设备的电源状态。

这个示例还展示了一种方案，将设备的电源状态改变，与驱动程序执行请求操作时的硬件访问同步。该驱动程序包括一个myReadDataFromDevice()示例操作，该操作调用activityTickle()，以确保硬件在执行操作之前处于可用的电源状态。但由于电源状态改变属异步，所以驱动程序需要等待，直至设备完全转换至新的电源状态。该示例程序通过在powerChangeDone()方法通知的条件变量上睡眠完成。

该驱动程序需要处理的另一个同步问题是，如果驱动程序正在处理未完成的操作，设备则无法进入睡眠状态。该示例驱动程序使用m_outstandingIO实例变量，保存驱动程序正在处理的未完成操作的个数。如果执行降低设备电源状态的请求，setPowerState()方法在移除设备电源之前，将一直等到所有未完成操作完成。当等待操作完成时，驱动程序需要确保不再启动其他操作，这通过在setPowerState()方法的开始处，将m_devicePowerState实例参数设置为低电源状态来实现。这意味着，m_devicePowerState实例变量在我们等待操作完成时，处于降低的电源状态中，但更重要的是，新的操作（如myReadDataFromDevice()）将睡眠，等待设备电源转换为打开状态。

该示例驱动程序的实现，如代码清单10-5及代码清单10-6所示。

**代码清单10-5　可以响应电源状态改变并控制自身电源状态的驱动程序（头文件）**

```
#include <IOKit/IOService.h>

class com_osxkernel_driver_IOKitTest : public IOService
{
    OSDeclareDefaultStructors(com_osxkernel_driver_IOKitTest)

private:
    IOLock*         m_lock;
    unsigned long   m_devicePowerState;
    SInt32          m_outstandingIO;

protected:
    virtual void            powerChangeDone (unsigned long stateNumber);

public:
    virtual void            free (void);
    virtual bool            start (IOService* provider);
    virtual void            stop (IOService* provider);

    virtual IOReturn setPowerState (unsigned long powerStateOrdinal, IOService* device);

    IOReturn          myReadDataFromDevice ();
};
```

**代码清单10-6 可以响应电源状态改变并控制自身电源状态的驱动程序（实现文件）**

```
#include "IOKitTest.h"
#include <IOKit/IOLib.h>

//定义超类
#define super IOService

OSDefineMetaClassAndStructors(com_osxkernel_driver_IOKitTest, IOService)

//定义我们的电源状态
enum {
     kOffPowerState,
     kStandbyPowerState,
     kIdlePowerState,
     kOnPowerState,
     //
     kNumPowerStates
};

static IOPMPowerState gPowerStates[kNumPowerStates] = {
     // kOffPowerState
     {kIOPMPowerStateVersion1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
     // kStandbyPowerState
     {kIOPMPowerStateVersion1, kIOPMPowerOn, kIOPMPowerOn, kIOPMPowerOn, 0, 0, 0, 0, 0, 0, 0, 0},
     // kIdlePowerState
     {kIOPMPowerStateVersion1, kIOPMPowerOn, kIOPMPowerOn, kIOPMPowerOn, 0, 0, 0, 0, 0, 0, 0, 0},
     // kOnPowerState
     {kIOPMPowerStateVersion1, kIOPMPowerOn | kIOPMDeviceUsable,
                               kIOPMPowerOn, kIOPMPowerOn, 0, 0, 0, 0, 0, 0, 0, 0}
};


bool com_osxkernel_driver_IOKitTest::start (IOService *provider)
{
     if (super::start(provider) == false)
        return false;

     //为驱动程序/电源管理同步创建一个锁
     m_lock = IOLockAlloc();
     if (m_lock == NULL)
        return false;

     //为电源管理注册驱动程序
     PMinit();
     provider->joinPMtree(this);
     makeUsable();                          //将私有电源状态设置为最高级别
     changePowerStateTo(kOffPowerState);//将公用电源状态设置为最低级别
     registerPowerDriver(this, gPowerStates, kNumPowerStates);

     //在5分钟不活动之后降低设备的电源级别（以秒表示）
     setIdleTimerPeriod(5*60);
```

```
    return true;
}

void com_osxkernel_driver_IOKitTest::stop (IOService *provider)
{
    PMstop();
    super::stop(provider);
}

void com_osxkernel_driver_IOKitTest::free (void)
{
    if (m_lock)
       IOLockFree(m_lock);
    super::free();
}


IOReturn com_osxkernel_driver_IOKitTest::setPowerState (unsigned long powerStateOrdinal,
                                                        IOService* device)
{
    //如果降低电源状态，在硬件掉电之前更新保存的电源状态
    if (powerStateOrdinal < m_devicePowerState)
       m_devicePowerState = powerStateOrdinal;

    switch (powerStateOrdinal)
    {
       case kOffPowerState:
       case kStandbyPowerState:
       case kIdlePowerState:
              //在设备进入低电源状态之前，等待未完成的IO完成
              IOLockLock(m_lock);
                     while (m_outstandingIO != 0)
                     {
                            IOLockSleep(m_lock, &m_outstandingIO, THREAD_UNINT);
                      }
              IOLockUnlock(m_lock);

              //为硬件睡眠做准备
              break;
    }

    //如果提高电源状态，在重新初始化设备之后，更新保存的电源状态
    if (powerStateOrdinal > m_devicePowerState)
       m_devicePowerState = powerStateOrdinal;

    return kIOPMAckImplied;
}

void com_osxkernel_driver_IOKitTest::powerChangeDone (unsigned long stateNumber)
{
    //唤醒正在等待电源状态改变的所有线程
    IOLockWakeup(m_lock, &m_devicePowerState, false);
}
```

```
// ***示例设备操作*** //
IOReturn com_osxkernel_driver_IOKitTest::myReadDataFromDevice ()
{
    //确保设备处于打开电源状态
    IOLockLock(m_lock);
       if (activityTickle(kIOPMSuperclassPolicy1, kOnPowerState) == false)
       {
              //等待，直到设备转换到打开电源状态
              while (m_devicePowerState != kOnPowerState)
              {
                     IOLockSleep(m_lock, &m_devicePowerState, THREAD_UNINT);
              }
        }

       //将未完成的操作数加1
       m_outstandingIO += 1;
    IOLockUnlock(m_lock);

    //执行设备读取...

    //操作完成时，将未完成的操作数减1
       m_outstandingIO -= 1;
       //唤醒未完成操作数中正在等待电源状态改变的线程
       IOLockWakeup(m_lock, &m_outstandingIO, false);
    IOLockUnlock(m_lock);

    return kIOReturnSuccess;
}
```

在实际的驱动程序中，myReadDataFromDevice()方法将被调用，以响应用户访问硬件设备的行为。因而该方法可以不定期地调用，并且可以随时调用。尽管代码清单10-6中并没有调用myReadDataFromDevice()方法，但可以扩展该代码，将一个用户客户端添加到该驱动程序中，使用户空间进程可以调用myReadDataFromDevice()方法。

## 10.6 小结

- 运行Mac OS X的每台计算机都可以进入低功率模式，称为“睡眠”。处于睡眠模式时，大部分硬件组件都完全掉电，或仅被提供较少的电量。
- 设备驱动程序可以注册，并响应电源管理系统的请求，从而在睡眠之前准备减少其硬件的一电量供应，并在系统从睡眠中唤醒时，恢复其硬件的电源状态。
- I/O Kit的电源管理API由IOService基类实现。这使每个驱动程序都能够支持电源管理。
- 不管计算机是否处于睡眠模式，驱动程序都能够独立地降低其设备的电源状态。这在设备有一段时间不使用时，对减少设备的电量损耗非常有用。
- 驱动程序能够观察系统中任意硬件设备的电源状态。它可以用来在设备改变其电源状态之前接收通知，也可以在设备转换至新的电源状态之后接收通知。

# 第11章 串行端口驱动程序

串行端口是一个基本的通信接口，用于将低带宽的数据传入/传出计算机。尽管现代接口（如USB和FireWire）已经替代了许多曾经使用的传统串行端口应用程序，但串行端口驱动程序在当代操作系统中，仍然获得良好的支持，包括Mac OS X，尽管在过去的十年中，苹果尚未发布内置串行端口的硬件。

串行端口如此长寿原因有以下几点。首先，在硬件中实现串行端口简单、廉价，这使它成为计算机爱好者将计算机通信添加到其电子工程中的普遍选择。其次，这是一个非常灵活的接口。串行驱动程序只关心串行端口与用户空间应用程序之间的数据字节传输。该驱动程序无法中断数据流，只是处理数据传输。

这需要用户空间应用程序来实现连接设备的协议，也就是说，通常由驱动程序完成的许多工作要全部交由用户空间应用程序来完成。这意味着硬件厂商无须为其设备提供驱动程序，可以只发布协议，描述通过串行端口传输的数据格式，而由其他人实现该协议。使用串行端口的设备包括GPS接收机及条码扫描器。它们通常用于调试硬件上的输出。

在麦金塔计算机中已经找不到串行端口，它们已被USB和FireWire端口取代。串行设备（如GPS接收机和条码扫描器）通过USB连接到计算机，但对系统来说，它们看起来就像是基于USB的串行端口，用户空间应用程序可以与它进行连接。如果你正通过串行端口通信，从而实现与某个设备交互，几乎在所有情况下，你的应用程序都可以使用现有的串行端口驱动程序。在极少数情况下，需要实现自己的串行端口驱动程序，即使是使用基于USB串行端口的工程也使用标准的驱动程序提供串行端口的接口。

本章将讲述在I/O Kit中如何实现一个串行端口驱动程序，以及在用户空间应用程序中，如何从串行端口进行读写。该串行端口驱动程序的实现可以看做是驱动程序技术的一个实践示例，包括实现阻塞调用、循环缓冲区，以及同步和通知。因此，即使串行端口驱动程序并不与你直接相关，但我们将阐述的概念也可以应用到其他驱动程序。

## 11.1 Mac OS X 串行端口体系结构概览

在Mac OS X上，串行端口有一个很有趣的体系结构。在内核中，串行驱动程序使用面向对象的I/O Kit框架实现，而在用户空间中，则通过BSD层对其进行访问，并且串行端口用一个常见

的UNIX设备文件表示。对于载入内核的串行驱动程序，I/O Kit串行族会在文件系统的/dev目录中，创建一个相应的设备对象。为了与串行端口交互，应用程序会打开该设备文件，就像对普通文件进行读写一样。这是通过用户客户端与驱动程序进行通信的一种替代方法，实际上，串行驱动程序并没有用户客户端。这种体系结构的优势在于，使用POSIX API编写的普通UNIX应用程序也可以访问Mac OS X上的串行端口，而无须对其代码做任何改动。

尽管从用户空间并不能通过I/O Kit框架访问串行端口驱动程序，但串行端口驱动程序仍然在内核中实现为一个完整的I/O Kit驱动程序。这意味着，串行驱动程序可以使用I/O Kit的所有特性，包括面向对象的设计、动态驱动程序加载及匹配。I/O Kit包括一个特定于串行设备的族，称为IOSerialFamily。IOSerialFamily可通过Darwin开源工程获得，包含串行端口驱动程序派生基类的头文件，以及Mac OS X的串行端口子系统的实现。

图11-1展示了通过Mac OS X上的串行端口进行通信处理所涉及的各个实体。对于该示例，我们假定串行端口由一个USB设备实现（如USB转RS-232适配器），因此图中最左边的提供者对象是IOUSBDevice。

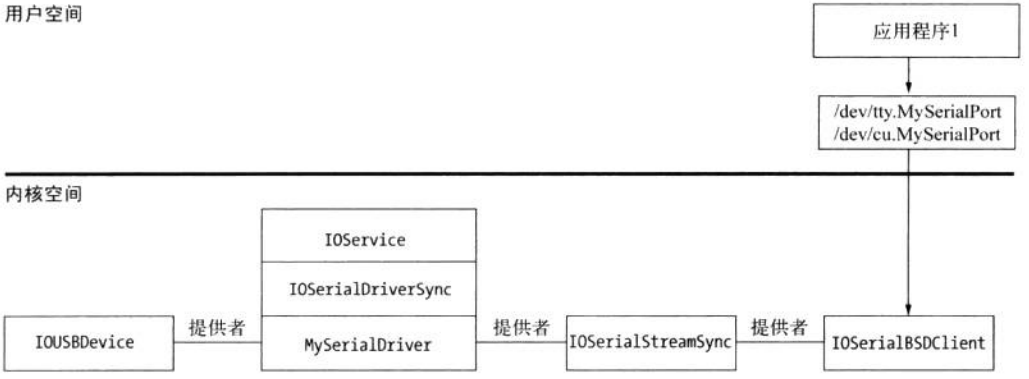



图11-1 用户空间应用程序与串行端口进行通信所涉及的对象

从用户空间开始，应用程序打开/dev目录下一个对应串行端口驱动程序的字符设备文件，连接到串行端口驱动程序。每个串行驱动程序在/dev目录中都有两项，一项的名字以“tty.”前缀开始，另一项的名字前缀是“cu.”。这在很大程度上与历史原因有关，源于通过串行端口将调制解调器或传真机连接到计算机的时代。这种情况下，tty设备是用于接收呼叫的拨入设备，cu是用于执行呼叫的调出设备。

希望接收呼叫的进程将打开拨入设备，open函数将阻塞，直至载波检测线接收到通知，意味着调制解调器已经建立了一个连接。但为了执行传出呼叫，进程需要能够打开串行端口，而无须等待载波检测线，因此，进程将打开调出设备，它在载波检测通知上并不会阻塞。当与现代串行设备通信时，考虑到大多数设备并不是连接到电话线，苹果建议打开调出设备。

/dev中的这两个设备文件均由I/O Kit的IOSerialBSDClient类创建。该类由IOSerialFamily提供，用来处理内核端的用户空间进程对设备文件的各种操作。IOSerialBSDClient处理用户空间进

程的所有系统调用，对串行端口进行操作，包括下列函数：open()、close()、read()、write()、ioctl()及select()。这样，IOSerialBSDClient可以看做是串行驱动程序的用户客户端（尽管它并非派生自IOUserClient类）。类似于用户客户端，IOSerialBSDClient类只是用户空间进程和内核驱动程序对象之间的一个通道。该类负责处理阻塞和非阻塞调用之间的不同之处，但其主要作用是将用户空间的调用转换为对IOSerialStreamSync对象的方法调用。

IOSerialStreamSync类是I/O Kit在IOSerialFamily包中提供的另一个类。它只是IOSerialBSDClient和实际串行驱动程序实现之间的一个通道。实际上，对于IOSerialStreamSync类的每个方法，它们只是调用IOSerialDriverSync类中同名的方法。

IOSerialDriverSync类是一个纯抽象接口，是Mac OS X上所有串行驱动程序的基类。其方法用于读写数据，以及报告串行端口的状态改变，如是否检测到载波通知，串行端口上要读取的数据是否到达，以及串行端口是否可以传输更多的字节。

最后，串行驱动程序栈中涉及的最后一个对象是串行驱动程序的提供者类。在本示例中，我们假定串行端口是通过一个USB转串行端口适配器实现，这也是提供者类有IOUSBDevice类型的原因。每当串行端口驱动程序需要从串行端口读数据或通过串行端口写数据时，都将通过IOUSBDevice实例访问底层硬件设备。

下面是实现串行端口驱动程序时每个类的职责。

- 串行端口驱动程序的提供者类，在我们的示例中为IOUSBDevice，将数据传输给计算机外的串行端口适配器。
- 主驱动程序是IOSerialDriverSync的子类，管理串行端口的接收和传输缓冲区，在数据到达时从硬件读取数据，并在用户空间提供数据时，将数据写入硬件。
- IOSerialBSDClient管理与用户空间应用程序的交互。

## 11.2　串行端口驱动程序

Mac OS X上的串行端口驱动程序通过创建一个派生自IOSerialDriverSync的类实现。本节通过查看苹果公司为USB串行通信设备提供的驱动程序源代码，描述串行端口驱动程序的实现。尽管可能无须直接实现自己的串行驱动程序，但本节可以看做是提供I/O Kit技术及驱动程序设计的可行应用程序，并且苹果公司的USB串行驱动程序使用的许多技术也许可以应用于你开发的其他应用程序。

为了学习本节内容，你可能希望下载我们讨论的串行端口驱动程序的源代码。内核串行驱动程序栈中的类，分布在Darwin源代码存储库中的下面两个工程中。

- 基类（所有的串行驱动程序在这些基类的基础上构建），包括IOSerialBSDClient类的实现，都包含在IOSerialFamily工程中。为了实现自己的串行端口驱动程序，只需要IOSerialFamily工程中包含的类。
- 针对基于USB串行通信设备加载的驱动程序，包含在AppleUSBCDCDriver工程中（USBCDC是USB Communications Device Class的首字母缩写词）。

考虑到I/O Kit为处理串行端口驱动程序而创建的支持类的数量，对理解这些类的初始化方式非常有益。当实现通信设备类的USB设备连接到Mac时，USB主机控制器将创建一个新的IOUSBDevice对象，表示该设备。该IOUSBDevice对象发布时，它会启动I/O Kit的驱动程序匹配进程，为USB设备搜索合适的驱动程序。在这种情况下，I/O Kit会选择AppleUSBCDC驱动程序，因为它的匹配字典指定它应该针对所有支持通信设备类的USB设备进行加载。请注意，AppleUSBCDC驱动程序并不是一个串行驱动程序。它是通用IOService类的子类，只负责配置用作串行设备的USB硬件。这通过迭代设备支持的端口，并为提供通信支持的类设置活动接口来实现。

在AppleUSBCDC驱动程序已经配置了活动的USB设备接口之后，I/O Kit的USB族将创建一个IOUSBInterface对象，表示活动的接口。这会再次启动I/O Kit的驱动程序匹配进程，为IOUSBInterface对象搜索合适的驱动程序。为此，我们将检查AppleUSBCDCDMM类的实现，该类将会匹配实现特定类型USB通信设备的任意IOUSBInterface对象。AppleUSBCDCDMM类的实现位于AppleUSBCDCDMM.cpp文件中，其声明位于AppleUSBCDCDMM.h中。

AppleUSBCDCDMM类是IOSerialDriverSync类的直接子类，负责实现串行端口驱动程序所需的方法。为了发送和接收硬件设备的数据，AppleUSBCDCDMM类会将USB传输请求发送给其提供者类（一个IOUSBInterface对象）。

## 手工实例化驱动程序对象

直到现在，内核对象的初始化和加载都相当标准，遵循所有驱动程序都执行的相同I/O Kit匹配流程。但串行端口驱动程序与你编写的许多驱动程序的不同之处在于，它有一个子驱动程序。你的串行端口驱动程序从而可以充当另一个I/O Kit驱动程序的提供者。

串行端口驱动程序的子驱动程序是一个派生自IOSerialStreamSync类的对象（或是一个派生自IOSerialStreamSync的类）。对于串行端口驱动程序，其子驱动程序并非通过I/O Kit的驱动程序匹配过程创建，而是由串行驱动程序本身显式初始化，并装载到串行驱动程序。

就AppleUSBCDCDMM驱动程序而言，其子驱动程序具有IOModemSerialStreamSync类型，这是直接派生自IOSerialStreamSync类的一个类。AppleUSBCDCDMM类在createSerialStream()方法中初始化其子对象，它从串行端口驱动程序的start()方法调用。createSerialStream()方法是一个自定义方法，为AppleUSBCDCDMM类私有。基于AppleUSBCDCDMM驱动程序createSerialStream()方法的示例实现，如代码清单11-1所示。

**代码清单11-1 创建串行端口驱动程序的IOSerialStreamSync对象的方法实现**

```
#include <IOKit/serial/IOModemSerialStreamSync.h>
#include <IOKit/serial/IOSerialKeys.h>

bool    MySerialDriver::createSerialStream()
{
        IOSerialStreamSync*     pChild;
        bool                    result;
```

```
        //实例化子驱动程序对象
        pChild = new IOModemSerialStreamSync;
        if (pChild == NULL)
                return false;

        //初始化子驱动程序
        result = pChild->init(0, 0);
        if (result == false)
                goto bail;

        //将pChild作为我们自身的子设备附加
        result = pChild->attach(this);   //传递this作为子驱动程序的提供者
        if (result == false)
                goto bail;

        //对命名/dev中的设备文件时使用的属性进行设置
        pChild->setProperty(kIOTTYBaseNameKey, "my_serial");
        pChild->setProperty(kIOTTYSuffixKey, "");

        //支持匹配将pChild作为其提供者类的驱动程序
        pChild->registerService();

        //成功执行完了
bail:
        pChild->release();
        return result;
}
```

代码清单11-1展示了驱动程序如何显式初始化其子驱动程序，无须为子驱动程序创建属性列表文件以及调用I/O Kit的匹配机制。创建新的驱动程序对象分为3个基本步骤：(1) 通过调用C++的new操作符，分配驱动程序类的一个实例；(2) 通过调用其init()方法，初始化返回的对象；(3) 通过attach()方法，将驱动程序对象附接到其父驱动程序。传递给attach()方法的参数是子设备的提供者类，通常为实例化子驱动程序的对象。

你会发现，因为I/O Kit使用的嵌入式C++语言并不支持异常，所以为了检查C++的new操作符调用是否失败，需要检查返回的值是否为NULL。请注意，在该方法返回之前，我们将无条件地释放子对象的引用，如果attach()调用成功，则子驱动程序对象将由I/O Kit进行存储，因此，在方法结尾调用release()并不会销毁该对象，而是释放我们持有的对象引用（防止泄漏）。另一方面，如果子驱动程序的init()方法执行失败，或无法附接，需要释放（并销毁）子驱动程序对象。

createSerialStream()方法实例化的IOModemSerialStreamSync类，为两个类之间的通道，即在串行端口上，处理由用户空间应用程序执行的操作的IOSerialBSDClient类，它还负责串行端口驱动程序的实现（上面例子中的MySerialDriver）。但串行端口驱动程序并不直接访问IOSerialBSDClient类。为了指定/dev目录中创建的字符设备文件的名称，需要将某些参数传递给IOSerialBSDClient的初始化方法。为此，在IOSerialBSDClient类创建之前，在IOSerialStreamSync对象上设置了两个属性，IOSerialBSDClient对象然后可以在其初始化中读取这些属性。

串行端口驱动程序会对IOSerialStreamSync对象调用registerService()方法，通知I/O Kit应该开始子设备的匹配过程。希望使用IOSerialStreamSync对象作为其父驱动程序（提供者类）的任意驱动程序都会被加载。通过这种方式，IOSerialBSDClient类将被加载，并附接到串行端口驱动程序创建的IOSerialStreamSync对象。IOSerialFamily内核模块包含一个IOSerialBSDClient类的匹配字典，匹配任意的IOSerialStreamSync对象（或继承自IOSerialStreamSync的类）。

---

**注意** 通常，显式实例化子驱动程序并将其附接到本身的驱动程序，无须对子驱动程序调用registerService()，因为这通常由子驱动程序本身处理。但就IOSerialStreamSync类而言，该类本身并不注册驱动程序匹配，因此在将其实例化之后，需要显式调用其registerDriver()方法。

---

当IOSerialStreamSync类使用I/O Kit注册时，将会创建、附接一个IOSerialBSDClient类的实例，完成串行端口所需的驱动程序对象集。IOSerialBSDClient类初始化时，会在/dev目录中为串行端口创建两个设备节点：一个对应拨入字符设备文件，另一个对应调出字符设备文件。为了确定这些文件的名称，IOSerialBSDClient类会读取kIOTTYBaseNameKey及KIOTTYSuffixKey的属性值，然后使用如下格式创建这两个文件：

```
"tty." + kIOTTYBaseNameKey + kIOTTYSuffixKey
```

和

```
"cu." + kIOTTYBaseNameKey + kIOTTYSuffixKey
```

对于代码清单11-1的例子，这会生成一个用于拨入设备、带有tty.my_serial名称的字符设备文件，以及一个用于调出设备、带有cu.my_serial的字符设备文件。



## 11.3 实现 IOSerialDriverSync 类

Mac OS X上的串行驱动程序必须派生自IOSerialDriverSync类。IOSerialDriverSync类是一个纯抽象类，是串行驱动程序必须实现的一个接口。串行驱动程序必须实现的方法如代码清单11-2所示。

**代码清单11-2 IOSerialDriverSync的接口，声明串行端口驱动程序必须实现的方法**

```
class IOSerialDriverSync : public IOService
{
        OSDeclareAbstractStructors(IOSerialDriverSync);

public:
        virtual IOReturn  acquirePort(bool sleep, void *refCon) = 0;

        virtual IOReturn  releasePort(void *refCon) = 0;

        virtual IOReturn  setState(UInt32 state, UInt32 mask, void *refCon) = 0;
        virtual UInt32    getState(void *refCon) = 0;
```

```
        virtual IOReturn  watchState(UInt32 *state, UInt32 mask, void *refCon) = 0;

        virtual UInt32    nextEvent(void *refCon) = 0;

        virtual IOReturn  executeEvent(UInt32 event, UInt32 data, void *refCon) = 0;

        virtual IOReturn  requestEvent(UInt32 event, UInt32 *data, void *refCon) = 0;

        virtual IOReturn  enqueueEvent(UInt32 event, UInt32 data,
                                       bool sleep, void *refCon) = 0;

        virtual IOReturn  dequeueEvent(UInt32 *event, UInt32 *data,
                                       bool sleep, void *refCon) = 0;

        virtual IOReturn  enqueueData(UInt8 *buffer, UInt32 size, UInt32 *count,
                                       bool sleep, void *refCon) = 0;

        virtual IOReturn  dequeueData(UInt8 *buffer, UInt32 size, UInt32 *count,
                                       UInt32 min, void *refCon) = 0;
};
```

你会发现，每个方法都有一个refCon参数。串行端口驱动程序可以使用refCon值，来确定各方法正在操作哪一个串行端口。refCon值实际上由串行端口驱动程序本身指定，并在初始化时传递给IOSerialStreamSync对象。反过来，每当IOSerialStreamSync类调用驱动程序的类方法时，会将此refCon值传回给驱动程序。大多数情况下，并不需要refCon值，因为串行端口程序需要的任何数据都可以添加为驱动程序类的实例参数。但有些情况下，串行端口驱动程序管理具有多个串行端口的硬件（如带有COM1和COM2端口的USB适配器），对它们而言，这些驱动程序需要一种方法来确定方法调用中涉及的是哪个端口。为此，串行端口驱动程序将创建IOSerialStreamSync类的两个实例，每个实例对应一个硬件端口，并为每个端口提供唯一的refCon值。

该接口看起来似乎让人生畏，但该方法可以分为3类：

- 调整串行端口的状态，并观察串行端口状态改变的方法；
- 设置/获取串行端口属性的方法；
- 通过串行端口读写数据的方法。

下面简要描述由IOSerialDriverSync接口定义的方法。

打开和关闭串行端口：

```
acquirePort()
releasePort()
```

- 管理表示串行端口状态的位掩码，阻塞调用线程，直至特定条件发生。状态位描述以下这类条件，如串行端口是否已经打开，数据是否已经通过串行端口接收并用于读取，以及串行端口是否可以接受字节写入：

```
setState()
getState()
watchState()
```

- 设置串行端口的属性：
  executeEvent()
  enqueueEvent()
- 获取串行端口的属性：
  nextEvent()
  requestEvent()
- 通过串行端口写数据：
  enqueueData()
- 读取串行端口接收到的数据：
  dequeueData()

实现IOSerialDriverSync接口的复杂性在于需要完全同步。接口方法可能在某些时刻会由多个线程调用，这意味着该实现需要确保每个方法都能正确同步，以防止像dequeueData()方法在串行端口已经关闭之后，仍然返回数据的情形。更复杂的情形是，一些方法可能会阻塞调用线程，直至特定事件发生，watchState()方法尤其如此，它在串行端口进入请求的状态之前，并不会返回。这两种同步问题可以通过使用互斥锁及条件变量，反映串行端口状态的改变来解决。

就AppleUSBCDCDMM而言，更大的同步问题在于，通过IOSerialDriverSync接口调用的方法需要与USB上异步传输完成时触发的回调相协调。这可用于防止dequeueData()方法对某个缓冲区执行读操作的同时，USB写入的完成回调将数据放入这个缓冲区。

尽管描述的同步问题看起来非常棘手，但I/O Kit设计提供了一个非常优雅的解决方案。所有的USB完成回调都运行在USB工作环上，因此为了使串行端口方法与USB代码同步，AppleUSBCDCDMM驱动程序需要确保所有的串行端口方法都在USB工作环上处理。执行过程如下所示：

```
bool AppleUSBCDCDMM::start(IOService *provider)
{
        …
        //获取USB工作环（超类将使用提供者类的工作环）
        fWorkLoop = getWorkLoop();
        …
        //创建一个命令门，并将其安装在USB工作环上
        fCommandGate = IOCommandGate::commandGate(this);
        fWorkLoop->addEventSource(fCommandGate);
        …
}
```

对于每个串行端口方法，AppleUSBCDCDMM驱动程序都会通过命令门对其进行调用，确保每个方法与USB工作环都同步。releasePort()的实现如下所示：

```
IOReturn AppleUSBCDCDMM::releasePort(void *refCon)
{
        IOReturn        ret = kIOReturnSuccess;

        //在工作环上调用静态方法releasePortAction()，不需要参数
        ret = fCommandGate->runAction(releasePortAction);
```

```
        return ret;
}

IOReturn AppleUSBCDCDMM::releasePortAction(OSObject *owner, void *, void *, void *, void *)
{
        //调用releasePortGated()方法
        return ((AppleUSBCDCDMM*)owner)->releasePortGated();
}

IOReturn AppleUSBCDCDMM::releasePortGated()
{
        …
        //实现releasePort
        …
}
```

IOCommandGate还提供了一个对象，线程可以在该对象上睡眠，并且在事件发生时，用来通知正在睡眠的线程。我们将会看到，这为实现串行端口的watchState()方法提供了一种便捷方式。

## 11.4　串行端口状态

实现IOSerialDriverSync接口的核心部分是管理串行端口的状态。串行端口状态是一个32位的位字段，用于报告事件（如串行端口上的数据到达），保存端口的整体状态（如端口是否已经被用户空间进程打开）。尽管存在可以直接用来操作状态位掩码的方法，但串行端口通过IOSerialDriverSync实现的每个方法最终都需要访问串行端口的状态，即使只是在试图执行一个操作前确认串行端口是否已经打开。

状态位在IOSerialStreamSync.h头文件中定义。下面是每个位的含义。

- PD_S_ACQUIRED表示串行端口已经打开，并且正由一个用户空间应用程序使用。这个状态位并不是通过setState()方法设置或清除，而是使用acquirePort()方法设置，releasePort()方法清除。
- PD_S_ACTIVE是在获取串行端口之后立即设置，并在串行端口释放之前立即清除。这个状态位并不是通过setState()方法设置或清除，而是通过executeEvent()方法设置或清除，使用PD_E_ACTIVE事件类型操作此状态位。
- 设置PD_S_TX_ENABLE和PD_S_RX_ENABLE，表示串行端口的传输和接收接口处于启用状态。大部分实现，包括AppleUSBCDCDMM驱动程序，在串行端口打开时设置这两个位，并在串行端口关闭时清除，但除此之外，不会使用这两个位。
- 设置PD_S_TX_BUSY和PD_S_RX_BUSY，表示串行端口驱动程序正在从其传输缓冲区向串行端口硬件发送数据，或正在读取通过串行端口硬件发送给驱动程序缓冲区的数据。
- PD_S_TX_EVENT和PD_S_RX_EVENT是IOSerialBSDClient类在内部使用的两个状态，通知开始写/读操作。尽管串行驱动程序不使用这两个状态，但需要设置setState()中的相应位，可以使客户端通过watchState()观察该位，以确保IOSerialBSDClient能够正确操作。

许多位都描述串行驱动程序的传输和接收缓冲区的状态。传输缓冲区是串行驱动程序用来持

有通过enqueueData()方法提供、但尚未通过串行端口硬件发送的字节。接收缓冲区中保存的字节已经由串行驱动程序从串行端口硬件读取，但尚未通过dequeueData()方法传递。串行驱动程序缓冲区的状态位描述如下。

- PD_S_TXQ_EMPTY和PD_S_RXQ_EMPTY表示传输缓冲区或接收缓冲区为空，不包含任何字节。
- PD_S_TXQ_LOW_WATER和PD_S_RXQ_LOW_WATER表示传输缓冲区或接收缓冲区中的字节数低于“低水位”。AppleUSBCDCDMM驱动程序将低水位设置为整个传输缓冲区或接收缓冲区大小的1/3。
- PD_S_TXQ_HIGH_WATER和PD_S_RXQ_HIGH_WATER表示传输缓冲区或接收缓冲区中的字节数高于“高水位”。AppleUSBCDCDMM驱动程序将高水位设置为整个传输缓冲区或接收缓冲区大小的2/3。
- PD_S_TXQ_FULL和PD_S_RXQ_FULL表示传输缓冲区或接收缓冲区已经全满，无法再接收其他任何数据。

IORS232SerialStreamSync.h头文件定义了标准RS-232通知的状态位。例如，为发送清除（PD_RS232_S_CTS）、数据终端就绪（PD_RS232_S_DTR）等通知提供定义。使用特定XON和XOFF字符传输的软件流量控制的状态，则通过状态位PD_RS232_S_TXO和PD_RS232_S_RXO表示。

串行端口驱动程序操作一个32位的整数，该整数中保存描述串行端口当前状态的状态位位掩码。串行驱动程序必须实现下列3个方法，才可以操作串行端口状态：getState()、setState()和watchState()。除了IOSerialBSDClient类在需要访问串行端口的状态时进行调用之外，这3个方法在串行端口类的许多其他方法实现中也会被调用。

例如，以独占访问打开串行端口时调用的acquirePort()，以及关闭串行端口时调用的releasePort()，都通过设置状态位PD_S_ACQUIRED实现。acquirePort()方法的一个可能实现，如代码清单11-3所示。该示例使用setState()设置PD_S_ACQUIRED位。如果串行端口已经获取，并且调用者已经请求该方法阻塞，直至串行端口被释放，则该实现将调用watchState()，一直等待到PD_S_ACQUIRED位被清除。



**代码清单11-3 acquirePort()方法的示例实现。假定已经通过IOCommandGate调用该方法**

```
IOReturn        MySerialDriver::acquirePortGated (bool sleep, void* refCon)
{
        UInt32      state;
        IOReturn    rtn;

        //如果已经获取该串行端口，则一直等到它被释放
        while (m_currentState & PD_S_ACQUIRED)
        {
                //如果调用者已经请求非阻塞操作，则放弃
                if (sleep == false)
                        return kIOReturnExclusiveAccess;

                //获取的位被清除之前，一直睡眠
                state = 0;
                rtn = watchState(&state, PD_S_ACQUIRED, refCon);
                if (rtn != kIOReturnSuccess)
```

```
                return rtn;
        }

        //设置获取的位，并清除所有其他的状态位
        setState(PD_S_ACQUIRED, 0xFFFFFFFF, refCon);

        //已经获取串行端口，执行进一步的初始化
        ...

        return kIOReturnSuccess;
}
```

releasePort()方法使用setState()清除PD_S_ACQUIRED状态位，它的一个可能实现如代码清单11-4所示。

**代码清单11-4　releasePort()方法的示例实现。假定已经通过IOCommandGate调用该方法**

```
IOReturn        MySerialDriver::releasePortGated (void* refCon)
{
        //试图释放一个未获取的端口时，返回一个错误
        if ((m_currentState & PD_S_ACQUIRED) == 0)
                return kIOReturnNotOpen;

        //清除整个状态字，释放端口
        setState(0, 0xFFFFFFFF, refCon);

        return kIOReturnSuccess;
}
```

通常，IOSerialBSDClient类或串行端口驱动程序本身，需要阻塞当前线程，直至特定状态变为活动或不活动。串行驱动程序通过watchState()方法提供该功能。调用者希望观察的事件由两个参数描述。mask参数包含调用者希望观察的状态位的位掩码。state参数描述调用者希望观察的每个状态位的对应值。例如，如果一个位在mask中设置未在state中设置，则调用者对变为不活动的状态感兴趣。如果一个位既在mask中也在state中设置，则调用者对变为活动的状态感兴趣。

watchState()方法将在某个观察的状态位与串行端口的当前状态相匹配时，立即返回。返回之后，串行端口的当前状态将通过state参数返回给调用者。如果线程在watchState()上阻塞时，串行端口处于关闭状态，则放弃睡眠，该方法执行失败，返回一个错误码给调用者，如kIOReturnNotOpen。以下代码是watchState()的一个用法示例，该代码将阻塞，直至驱动程序的传输缓冲区变为空（设置了PD_S_TXQ_EMPTY），或硬件完成了向串行端口硬件的写入（清除PD_S_TX_BUSY）。

```
UInt32          state;
IOReturn        rtn;

state = PD_S_TXQ_EMPTY;
rtn = watchState(&state, PD_S_TXQ_EMPTY | PD_S_TX_BUSY, refCon);
if (rtn != kIOReturnSuccess)
        handle error;
```

watchState()方法的实现与setState()方法的实现紧密相关。除了在串行端口的状态字中设

置位，setState()方法还负责唤醒等待特定状态设置的线程。第7章介绍了条件变量，看到一个线程如何在条件变量上睡眠，并保持阻塞，直至另一个线程通知条件变量，表示一个事件已经发生。这形成了一种机制，串行端口驱动程序可以在watchState()方法中暂停一个线程，并在观察到状态已经发生改变时，通过setState()方法通知该线程。

调用setState()方法时，串行驱动程序会更新保存当前串行端口状态的变量，然后通知在watchState()方法上阻塞的所有线程，使其可以测试等待的状态是否已经变为活动状态。优化后，串行端口状态并不是在每次改变时都唤醒watchState()中阻塞的线程，AppleUSBCDCDMM驱动程序维护一个联合体，表示watchState()的当前所有调用中等待的所有状态位，仅在被观察的状态位的值发生改变时，才解锁线程。

setState()和watchState()方法的一个示例实现，如代码清单11-5所示。

**代码清单11-5 setState()和watchState()方法的示例实现。假定已经通过IOCommandGate调用这两个方法**

```
IOReturn        MySerialDriver::setStateGated(UInt32 state, UInt32 mask, void* refCon)
{
        UInt32    newState;
        UInt32    deltaState;

         //验证串行端口是已经通过此调用获得，还是正在获得
        if ((m_currentState & PD_S_ACQUIRED) || (state & PD_S_ACQUIRED))
        {
                //计算新的状态
                newState = (m_currentState & ~mask) | (state & mask);
                //确定已改变状态位的掩码
                deltaState = newState ^ m_currentState;
                //设置新的状态
                m_currentState = newState;

                //如果线程在watchState()中观察的某个状态已经发生改变，则唤醒watchState()上睡眠的所有线程
                if (deltaState & m_watchStateMask)
                {
                        //重置watchStateMask，它会在每次watchStateGated()睡眠时，重新生成
                        m_watchStateMask = 0;
                        fCommandGate->commandWakeup((void*)&m_currentState);
                }

                return kIOReturnSuccess;

        }

        return kIOReturnNotOpen;
}

IOReturn        MySerialDriver::watchStateGated(UInt32* state, UInt32 mask, void* refCon)
{
        UInt32      watchState;
        bool        autoActiveBit = false;
```

```
    IOReturn    ret;

    //如果串行端口还未获得，则放弃
    if ((m_currentState & PD_S_ACQUIRED) == 0)
        return kIOReturnNotOpen;

    watchState = *state;
    //如果调用者没有等待获得的或活动的状态
    //则注册感兴趣的活动状态，以便我们可以在串行端口关闭时放弃
    if ((mask & (PD_S_ACQUIRED | PD_S_ACTIVE)) == 0)
    {
        watchState &= ~PD_S_ACTIVE;
        mask |= PD_S_ACTIVE;
        autoActiveBit = true;
    }

    while (true)
    {
        //检查与watchState值相匹配的任意位的端口状态
        //请注意，'^~'是XNOR，测试位是否相等
        UInt32    matchedStates = (watchState ^ ~m_currentState) & mask;
        if (matchedStates)
        {
            *state = m_currentState;
            //如果串行端口被关闭，并且调用者没有观察PD_S_WATCH，则放弃
            if (autoActiveBit && (matchedStates & PD_S_ACTIVE))
                return kIOReturnIOError;
            else
                return kIOReturnSuccess;
        }

        //将我们睡眠所在的位添加到watchStateMask
        m_watchStateMask |= mask;
        //一直睡眠到串行端口状态发生改变
        ret = fCommandGate->commandSleep((void*)&m_currentState);
        if (ret == THREAD_INTERRUPTED)
            return kIOReturnAborted;
    }

    return kIOReturnSuccess;
}
```

请注意，代码清单11-5中的watchState()的实现将确保PD_S_ACQUIRED或PD_S_ACTIVE位处于观察中，否则，一个额外的状态会被添加至掩码，以监督PD_S_ACTIVE位清除。这可以确保串行端口关闭时，在watchState()调用中阻塞的所有线程都将被唤醒，并返回到调用者。如果串行端口释放的掩码并没有显式添加，则阻塞的线程将无法被唤醒，导致串行端口驱动程序死锁。

## 11.5　串行端口事件

硬件串行端口需要进行配置，以匹配串行连接另一端上的设备使用的设置。可配置的设置包

括数据发送的波特率、数据字符中的位数、校验位是否传输，以及每个字符结尾发送的停止位等参数。

这些串行端口的设置由打开该串行端口的用户空间进程确定，并通过tcsetattr()、tcgetattr()等函数进行配置。这些函数通过I/O Kit的IOSerialBSDClient类访问内核，该类通过其executeEvent()方法，将个性化配置选项传递给串行端口驱动程序。

executeEvent()方法与requestEvent()方法要成对调用，IOSerialBSDClient类使用requestEvent()方法来查询串行端口的当前配置。executeEvent()和requestEvent()方法的原型，如下所示：

```
IOReturn        executeEvent(UInt32 event, UInt32 data, void* refCon);
IOReturn        requestEvent(UInt32 event, UInt32* data, void* refCon);
```

event参数是IOSerialStreamSync.h中的一个枚举，标识配置或查询的属性。data参数用于为executeEvent()设置的属性传递新值，或传递通过requestEvent()查询的当前属性值。大多数驱动程序都对event参数值使用庞大的switch语句来实现executeEvent()和requestEvent()方法。可能的事件类型如表11-1所示。

**表11-1 executeEvent()或requestEvent()处理的事件类型**

| 事 件 | 描 述 |
|---|---|
| PD_E_ACTIVE | 该事件用于启动或停止串行端口硬件。data参数是一个布尔值，非0值表示应该启动串行端口硬件，0值表示应该停止串行端口硬件。驱动程序应该通过改变硬件的状态，然后设置或清除状态位PD_S_ACTIVE，从而以反映硬件的状态来进行响应 |
| | 该属性可以使用requestEvent()查询，此时，驱动程序应该通过data参数，将硬件的当前状态返回给调用者 |
| PD_E_TXQ_SIZE<br>PD_E_RXQ_SIZE | data参数指定串行驱动程序的内部传输缓冲区或接收缓冲区的分配大小。缓冲区大小指定为缓冲区可以保存的字符数 |
| | 可以对该属性进行查询和设置，尽管实现可以忽略调用者对该值的设置请求 |
| PD_E_TXQ_LOW_WATER<br>PD_E_RXQ_LOW_WATER<br>PD_E_TXQ_HIGH_WATER | data参数指定串行驱动程序内部被认为是低水位或高水位的传输缓冲区或接收缓冲区的字符数。该参数控制状态位PD_S_TXQ_LOW_WATER、PD_S_RXQ_LOW_WATER、PD_S_TXQ_HIGH_WATER、PD_S_RXQ_HIGH_WATER的设置时机 |
| PD_E_RXQ_HIGH_WATER | 可以对该属性进行查询和设置，尽管实现可以忽略调用者对该值的设置请求 |
| PD_E_TXQ_AVAILABLE<br>PD_E_RXQ_AVAILABLE | data参数返回在传输缓冲区变满之前可以写入的额外字符数，或驱动程序接收缓冲区中当前持有的字符数 |
| | 可以对该属性进行查询，但不能设置 |
| PD_E_TXQ_FLUSH<br>PD_E_RXQ_FLUSH | 该事件通过executeEvent()指定，表示串行驱动程序应该丢弃其内部的传输缓冲区或接收缓冲区中的所有字符 |
| PD_E_DATA_RATE | 该事件用于获取/设置串行端口的波特率。data参数中的值使用半位表示速度，意味着波特率（用位度量）为data参数的值除以2 |
| | 另外，还存在一个PD_E_RX_DATA_RATE事件，允许单独指定用于数据输入的波特率，尽管大多数实现将忽略该事件 |

（续）

| 事　件 | 描　述 |
| --- | --- |
| PD_E_DATA_SIZE | 该事件用于获取/设置通过串行端口发送的每个字符的位数。data参数中的值指定数据的长度，单位是位 |
| | 另外，还存在一个PD_E_RX_DATA_SIZE事件，允许单独指定数据输入的长度，尽管大多数实现将忽略该事件 |
| PD_E_DATA_INTEGRITY | 该事件用于获取/设置通过串行端口所发送数据的校验位。data参数中的值可能是PD_RS232_PARITY_NONE、PD_RS232_PARITY_ODD或PD_RS232_PARITY_EVEN |
| | 此外，还存在一个PD_E_RX_DATA_INTEGRITY事件，允许单独指定数据输入的校验位，尽管大多数实现将忽略该事件 |
| PD_RS232_E_STOP_BITS | 该事件在通过串行端口发送每个字符之后用于获取/设置发送的停止位个数。data参数中的值使用半位表示，意味着data值2配置1个串行端口的停止位 |
| | 此外，还存在一个PD_RS232_E_RX_STOP_BITS事件，允许单独指定数据输入的停止位，尽管大多数实现将忽略该事件 |
| PD_E_FLOW_CONTROL | 该事件用于传递用户空间进程请求的流量控制状态。data参数值是一个位字段，其中每位对应用户空间termios结构的一位。以下列表是为data值定义的位，以及每位对应用户空间termios结构中的标志： |
| | PD_RS232_A_TXO（等价于IXON） |
| | PD_RS232_A_XANY（等价于IXANY） |
| | PD_RS232_A_RXO（等价于IXOFF） |
| | PD_RS232_A_RFR（等价于CRTS_IFLOW） |
| | PD_RS232_A_CTS（等价于CCTS_OFLOW） |
| | PD_RS232_A_DTR（等价于CDTR_IFLOW） |
| PD_RS232_E_XON_BYTE | 这些事件用于获取/设置软件流控制启用时使用的启动和停止字符 |
| PD_RS232_E_XOFF_BYTE | |
| PD_RS232_E_LINK_BREAK | 该事件带有一个布尔值data，指定是否在传输线路上发出RS-232中断条件通知 |

除了executeEvent()方法，你会发现，IOSerialDriverSync接口还定义了一个enqueueEvent()方法，用于设置串行端口的属性。这两个方法之间有细微的差别，调用executeEvent()之后，串行端口的配置会立即发生改变，而enqueueEvent()调用不会立即生效，直至串行驱动程序传输缓冲区中当前所有的字符都已写入硬件。

为了实现enqueueEvent()的正确操作，串行端口需要定义一个传输缓冲区，其中包括一个事件队列及与每个事件相关联的数据。然后，每次调用enqueueEvent()时，都将一对{event，data}值添加到传输队列。类似地，传输的字符数据也需要看做事件，添加到传输缓冲区的结尾。每当传输缓冲区不为空时，串行驱动程序则将下一个事件推出队列，它可能是一个改变串行端口配置的事件，也可能是一个通过串行端口发送的字符。

串行驱动程序无须完全遵循enqueueEvent()的正确实现。如果查看AppleUSBCDCDMM驱动程序的源代码，你会看到，该驱动程序通过调用executeEvent()实现enqueueEvent()，它会立即将请求的改变应用到串行端口的配置。

类似地，串行驱动程序的接收缓冲区允许将事件插入到从串行端口读取的数据字节之间。对于接收队列，各事件表示从串行端口读取数据时发生的错误。下面是报告的一些错误。

- PD_RS232_E_RX_LINE_BREAK表示接收者检测到一个中断条件。
- PD_E_FRAMING_ERROR表示字符数据设计错误。（停止位未在预期的位置。）
- PD_E_INTEGRITY_ERROR表示检测到校验错误。
- PD_E_HW_OVERRUN_ERROR和PD_E_SW_OVERRUN_ERROR表示字符数据并未足够快地从硬件或软件缓冲区取出，以防止缓冲区填充和数据丢失。

从串行端口读取数据之前，IOSerialBSDClient类将调用串行驱动程序的nextEvent()方法。如果串行驱动程序接收队列中的下一个成员是一个错误事件，nextEvent()会将该事件类型返回给调用者，调用者将调用驱动程序的dequeueEvent()方法对其进行响应。否则，如果串行驱动程序接收队列中的下一个成员是一个从串行端口读取的数据类型，则返回PD_E_EOQ。

类似于串行驱动程序的传输端，驱动程序无须完全实现其接收队列中的事件排队。实际上，AppleUSBCDCDMM驱动程序的实现在其接收队列上根本不报告任何事件，其nextEvent()和dequeueEvent()的实现将会检查串行端口是否活动，如果是，则总是返回kIOReturnSuccess值。请注意，kIOReturnSuccess有一个0值，从而对应PD_E_EOQ事件，该事件也有一个0值。

## 11.6 串行数据传输

从IOSerialDriverSync接口实现的其他方法为数据传输方法。串行驱动程序通过enqueueData()方法，在串行端口上进行数据传输，驱动程序从串行端口接收的数据将通过dequeueData()方法提供给客户端。

当一个用户空间进程向串行端口写数据时，首先会在内核中由IOSerialBSDClient类进行处理，该类负责将数据传递给串行端口驱动程序。IOSerialBSDClient通过调用其enqueueData()方法，将数据提供给串行驱动程序，其签名如下所示：

```
IOReturn  enqueueData(UInt8 *buffer, UInt32 size, UInt32 *count, bool sleep, void *refCon);
```

发送的数据字节存储在buffer参数中，发送的字节数由size参数描述。典型的串行驱动程序设计是将提供的数据复制到分配的内部缓冲区（称为传输缓冲区），然后立即返回给调用者。驱动程序然后将继续处理写请求，从传输缓冲区向硬件串行端口异步传输数据。从enqueueData()方法返回之前，驱动程序将通过count参数，返回它所接受的字节数，请注意，这只是驱动程序可以复制到其传输缓冲区的字节数，而不是通过硬件串行端口已经写入的字节数。利用sleep参数，调用者可以请求驱动程序阻塞，如果无法接受提供的所有字节，则驱动程序应该阻塞，并且在所有字节都已复制到驱动程序的内部传输缓冲区以前，不返回给调用者。

IOSerialBSDClient的当前实现从不请求串行驱动程序在无法接受提供的所有数据字节时睡

眠。相反，IOSerialBSDClient将确保为串行驱动程序提供的数据不超出其可以接受的范围，这通过调用驱动程序的requestEvent()方法，使用PD_E_TXQ_AVAILABLE事件实现。IOSerialBSDClient将监视驱动程序传输缓冲区的各种状态，确定驱动程序何时可以接受更多的数据，包括PD_S_TXQ_LOW_WATER、PD_S_TXQ_EMPTY及PD_S_TX_BUSY。

enqueueData()方法的一个示例实现如代码清单11-6所示。请注意，该实现将数据字节复制到串行驱动程序已经分配的传输缓冲区中，然后检查硬件当前是否正在串行端口上写出数据。如果没有，则调用一个虚构的函数StartHardwareTransmit()，尽管特定于实现，但该函数负责告知硬件开始通过串行端口从驱动程序的传输缓冲区发送数据字节。

**代码清单11-6　enqueueData()方法的示例实现。假定已经通过IOCommandGate调用这两个方法**

```
IOReturn          MySerialDriver::enqueueDataGated(UInt8* buffer, UInt32 size, UInt32* count,
                                                   bool sleep, void* refCon)
{
        //如果未获得串行端口，则放弃
        *count = 0;
        if ((m_currentState & PD_S_ACTIVE) == 0)
                return kIOReturnNotOpen;

        //将提供的数据复制到驱动程序的传输缓冲区
        *count = AddToTransmitQueue(buffer, size);
        //为传输缓冲区重新生成状态位
        CheckQueues(refCon);

        //如果没有正在进行的硬件传输，则开始从驱动程序的缓冲区输出字节
        if ((m_currentState & PD_S_TX_BUSY) == 0)
                StartHardwareTransmit();

        //如果调用者已经要求我们在返回之前发送所有的字节，则阻塞
        while ((*count < size) && sleep)
        {
                UInt32      state;
                IOReturn    ret;

                //一直等到驱动程序的传输缓冲区降至低水位之下，并重试
                state = PD_S_TXQ_LOW_WATER;
                ret = watchState(&state, PD_S_TXQ_LOW_WATER, refCon);
                if (ret != kIOReturnSuccess)
                        return ret;

                //将其余的字节复制到驱动程序的传输缓冲区
                *count += AddToTransmitQueue(buffer + *count, size - *count);
                CheckQueues(refCon);
                if ((m_currentState & PD_S_TX_BUSY) == 0)
                        StartHardwareTransmit();
        }

        return kIOReturnSuccess;
}
```

数据传输的另一部分是，读取从硬件串行端口接收的字节。串行驱动程序获取从其硬件设备接收的数据，并将其复制到内部的接收缓冲区。硬件通知串行驱动程序数据已经被接收的具体方式特定于实现，但可以通过一个PCI中断或USB传输完成来通知。现在驱动程序需要将接收到的数据传递给IOSerialBSDClient，IOSerialBSDClient接着将该数据提供给用户空间进程。

I/O Kit使用Pull模型，将数据从串行驱动程序返回给IOSerialBSDClient类。IOSerialBSDClient调用驱动程序的dequeueData()方法，获得硬件串行端口上接收到的数据，该方法的签名如下所示：

```
IOReturn    dequeueData(UInt8* buffer, UInt32 size, UInt32* count, UInt32 min, void* refCon);
```

收到该函数后，串行端口驱动程序应该将数据从其内部接收缓冲区复制到提供的buffer参数。size参数描述提供的缓冲区可以保存的最大字节数。count参数用于返回写入到所提供缓冲区中的实际字节数。调用者可以请求dequeueData()方法阻塞，在最小字节数可用之前不返回给调用者，这通过在min参数中指定一个非0值实现，min参数指定调用者应该返回的最小字节数。

在数据可用之前并不是一直持续轮询dequeueData()方法，而是IOSerialBSDClient类指定1字节的最小读取长度。这样做的效果是，在dequeueData()调用中阻塞，但当串行端口接收到数据时，立即返回该方法。AppleUSBCDCDMM串行端口驱动程序通过调用watchState()方法，并一直等到PD_S_RXQ_EMPTY状态清除（表示数据在驱动程序的接收缓冲区中可用）来实现该方法。这种设计的优点在于，确保驱动程序在串行端口关闭时，能够解除dequeueData()方法中的阻塞，因为PD_S_ACTIVE标识清除时（这会在用户进程关闭串行端口时发生），watchState()方法将会释放。

dequeueData()方法的一个示例实现如代码清单11-7所示。该实现将数据从驱动程序的内部接收缓冲区复制到该方法的调用者所提供的一个缓冲区。

**代码清单11-7 dequeueData()方法的示例实现。假定已经通过IOCommandGate调用这两个方法**

```
IOReturn            MySerialDriver::dequeueDataGated(UInt8* buffer, UInt32 size, UInt32* count,
                                                     UInt32 min, void* refCon)
{
        //如果未获得串行端口，则放弃
        *count = 0;
        if ((m_currentState & PD_S_ACTIVE) == 0)
                return kIOReturnNotOpen;

        //从驱动程序的接收缓冲区复制数据
        *count = RemovefromReceiveQueue(buffer, size);
        //重新生成接收缓冲区的状态位
        CheckQueues(refCon);

        //如果调用者已经请求了最小数量的字节，则阻塞
       while ((min > 0) && (*count < min))
        {
                UInt32          state;
                IOReturn        ret;
```

```
            //一直等到驱动程序的接收缓冲区不为空，并重试
            state = 0;
            ret = watchState(&state, PD_S_RXQ_EMPTY, refCon);
            if (ret != kIOReturnSuccess)
                  return ret;

            //从驱动程序的接收缓冲区复制其余的字节
            *count += RemovefromReceiveQueue(buffer + *count, size - *count);
            CheckQueues(refCon);
      }

      return kIOReturnSuccess;
}
```

代码清单11-6中的enqueueData()和代码清单11-7中的dequeueData()的示例实现，在读取内部接收缓冲区或写入内部传输缓冲区之后，都调用了一个虚构的CheckQueues()函数。尽管CheckQueues()是一个虚构函数，但任何串行端口驱动程序都需要它。它的作用是检查驱动程序内部传输缓冲区或接收缓冲区中的字节数，以及更新描述驱动程序队列的状态标志。这些标志描述传输或接收队列是否为空、满、包含少于低水位的字节，或包含多于高水位的字节。由于可能有等待传输缓冲区或接收缓冲区达到某一程度的线程，所以每当串行端口驱动程序读/写其内部缓冲区时，都需要更新这些状态标志，这非常重要。

除了从enqueueData()和dequeueData()方法调用（如代码清单11-6和代码清单11-7所示）之外，传输缓冲区的数据通过硬件串行端口移除、写入，以及硬件把从串行端口读取的数据添加到接收缓冲区时，串行端口驱动程序也会调用CheckQueues()函数。CheckQueues()方法的一个示例实现如代码清单11-8所示。

**代码清单11-8　dequeueData()方法的示例实现。假定已经通过IOCommandGate调用这两个方法**

```
void MySerialDriver::CheckQueues(void* refCon)
{
      UInt32      usedSpace;
      UInt32      freeSpace;
      UInt32      newState;
      UInt32      deltaState;

      //在函数入口处，将newState初始化为当前状态
      newState = m_currentState;

      //检查传输缓冲区中使用及释放的字节数
      usedSpace = GetUsedSpaceInTransmitQueue();
      freeSpace = GetFreeSpaceInTransmitQueue();

      //将传输缓冲区设置为满/空状态
      if (freeSpace == 0)
      {
            newState |=  PD_S_TXQ_FULL;
```

```
            newState &= ~PD_S_TXQ_EMPTY;
        }
        else if (usedSpace == 0)
        {
            newState &= ~PD_S_TXQ_FULL;
            newState |=  PD_S_TXQ_EMPTY;
        }
        else
        {
            newState &= ~PD_S_TXQ_FULL;
            newState &= ~PD_S_TXQ_EMPTY;
        }

        //将传输缓冲区设置为低/高水位
        if (usedSpace < m_txLowWaterlevel)
            newState |=  PD_S_TXQ_LOW_WATER;
        else
            newState &= ~PD_S_TXQ_LOW_WATER;

        if (usedSpace > m_txHighWaterlevel)
            newState |= PD_S_TXQ_HIGH_WATER;
        else
            newState &= ~PD_S_TXQ_HIGH_WATER;

        //对接收缓冲区执行同样的检查
        …

        //更新改变的状态位
        deltaState = newState ^ m_currentState;
        setState(newState, deltaState, refCon);
}
```

## 11.7 从用户空间访问串行端口

对于用户空间应用程序，串行端口驱动程序将作为/dev目录中的一个标准字符设备进行访问。在UNIX系统上使用过串行端口的人对此都应该很熟悉。但它与I/O Kit方案的不同之处在于，用户空间应用程序如何枚举系统上的串行端口。对于许多传统的UNIX应用程序，用户必须指定串行端口字符文件的全路径。而Mac OS X采取的方案是，对用户屏蔽/dev目录，用一个描述性的名称表示可用的串行端口。这才是I/O Kit发挥作用之处。

由于串行端口由I/O Kit驱动程序实现，所以其驱动程序对象可由用户空间应用程序在I/O Registry中找到，如第5章所述。类似于I/O Registry中所有的项，串行端口驱动程序项包含一个属性表，可以用来获取串行端口的一个描述性名称，以及串行端口字符设备文件的一个全路径。获得串行端口设备文件的路径之后，用户空间应用程序然后可以像传统的UNIX程序一样，继续打开并访问该设备。

与通过I/O Registry确定驱动程序的应用程序类似，查找串行端口驱动程序的第一步是，创建

一个匹配字典。匹配字典的作用是，确定I/O Registry中满足某些原则的项，从而过滤出所有其他的项。用户空间进程访问串行端口，并不是通过串行端口驱动程序本身，而是通过与驱动程序相关的IOSerialBSDClient类。因此，为了查找系统中的串行端口，用户空间进程只需创建一个匹配字典，在Registry中查找所有的IOSerialBSDClient对象。其实现如下所示：

```
#include <IOKit/serial/IOSerialKeys.h>

CFMutableDictionaryRef    matchingDict;
matchingDict = IOServiceMatching(kIOSerialBSDServiceValue);
```

为了进一步改进匹配，用户进程可以将kIOSerialBSDTypeKey键添加到匹配字典，并将结果限定为调制解调器设备（创建一个IOModemSerialStreamSync对象的串行驱动程序）或通用串行端口设备（创建一个IORS232SerialStreamSync对象的串行驱动程序）。例如，为了限定匹配调制解调器设备，用户空间应用程序将创建如下匹配字典：

```
matchingDict = IOServiceMatching(kIOSerialBSDServiceValue);
CFDictionarySetValue(matchingDict, CFSTR(kIOSerialBSDTypeKey), CFSTR(kIOSerialBSDModemType));
```

创建了一个匹配字典以定位它所感兴趣的串行设备之后，该进程然后可以迭代Registry，查找满足该字典指定原则的驱动程序。IOSerialBSDClient的所有实例都包含特定于串行端口驱动程序的Registry属性，即：

- kIOTTYDeviceKey：包含串行端口描述性名称的一个CFStringRef；
- kIOCalloutDeviceKey：包含串行端口调出字符设备文件全路径的一个CFStringRef；
- KIODialinDeviceKey：包含串行端口拨入字符设备文件全路径的一个CFStringRef。

为了演示如何使用这些属性，代码清单11-9中的代码给出了系统中所有串行设备的枚举方式，以及每个设备的打开方式。

**代码清单11-9　使用I/O Kit枚举系统中存在的所有串行设备，以及查找每个串行端口字符设备全路径的示例实现**

```
#include <CoreFoundation/CoreFoundation.h>
#include <IOKit/IOKitLib.h>
#include <IOKit/serial/IOSerialKeys.h>
#include <sys/param.h>
#include <fcntl.h>
#include <unistd.h>

int main (int argc, const char * argv[])
{
    CFMutableDictionaryRef          matchingDict;
    io_iterator_t                   iter = 0;
    io_service_t                    service = 0;
    kern_return_t                   kr;

    //创建一个可以查找任意串行设备的匹配字典
    matchingDict = IOServiceMatching(kIOSerialBSDServiceValue);
```

```
    kr = IOServiceGetMatchingServices(kIOMasterPortDefault, matchingDict, &iter);
    if (kr != KERN_SUCCESS)
        return -1;

    //迭代所有的匹配对象
    while ((service = IOIteratorNext(iter)) != 0)
    {
        CFStringRef          cfDeviceName;
        CFStringRef          cfCalloutPath;
        Char                 deviceName[256];
        Char                 calloutPath[MAXPATHLEN];
        Int                  fd;

        //获得设备名称
        cfDeviceName = IORegistryEntryCreateCFProperty(service, CFSTR(kIOTTYDeviceKey),
                        kCFAllocatorDefault, 0);
        CFStringGetCString(cfDeviceName, deviceName, sizeof(deviceName),
                        kCFStringEncodingUTF8);
        CFRelease(cfDeviceName);

        //获得字符设备的路径
        cfCalloutPath = IORegistryEntryCreateCFProperty(service,
                        CFSTR(kIOCalloutDeviceKey), kCFAllocatorDefault, 0);
        CFStringGetCString(cfCalloutPath, calloutPath, sizeof(calloutPath),
                        kCFStringEncodingUTF8);
        CFRelease(cfCalloutPath);

        //不再需要I/O Registry对象
        IOObjectRelease(service);

        //继续照常打开并使用calloutPath路径的设备
        printf("Found device %s at path %s\n", deviceName, calloutPath);

        fd = open(calloutPath, O_RDWR | O_NOCTTY | O_NONBLOCK);
        //清除O_NONBLOCK标志，后续的I/O将阻塞
        fcntl(fd, F_SETFL, 0);

        //使用tcsetattr()配置串行设备
        //使用read()/write()进行读写

        close(fd);
    }

    //释放I/O Registry迭代器
    IOObjectRelease(iter);

    return 0;
}
```

## 11.8　小结

- 串行端口为设备之间的低带宽数据传输提供了一种简单的方式。尽管无法在现代Mac上找到RS-232或RS-422串行端口，但许多USB设备都作为串行端口而存在，因此支持串行端口的驱动程序仍然是Mac OS X操作系统的一部分。
- 用户空间应用程序通过/dev目录中的设备文件访问串行端口，这也是UNIX环境的标准。
- 串行端口驱动程序在I/O Kit中，通过实现IOSerialDriverSync接口的一个子类实现。该接口包含打开/关闭串行端口、配置端口、读写数据等方法。
- I/O Kit提供了一个IOSerialBSDClient类，负责将串行端口驱动程序发布给用户空间应用程序。该类接收用户空间应用程序的请求，读写串行端口，并将该请求传递给内核串行端口驱动程序。
- I/O Kit设计可以使串行端口驱动程序使用现代的、面向对象的方式实现，而无须处理串行端口设备的传统用户空间接口。

# 第12章 音频驱动程序

音频设备是除存储设备之外连接到计算机的最常见外设。音频设备的应用非常广泛，从语音录音器到MP3播放器、带麦克风的耳麦、安全系统、DJ及专业录音系统等。许多视频设备也都具有音频功能，并且自带了音频驱动程序，利用这些音频驱动程序，你可以单独使用或与视频一起使用该设备的音频特性，例如，带有内置麦克风的网络照相机。麦克风有其自己的音频驱动程序，表示系统中的麦克风可以作为独立于照相机使用的音频设备。

编写音频设备的驱动程序存在一些很独特的挑战。音频设备具有非常严格的延迟要求，并且必须为它提供一定数量的数据，以避免回放或录制时的漏洞或小差错。人类的耳朵非常敏感，仅在少量的示例数据中，便可以检测出极小的差错。另外，音频数据无法完全缓冲，因为这样会导致让人难以接受的延迟。例如，假定你在播放强劲的音乐时接到电话。如果从你按下暂停按钮到音乐实际停止需要花费5秒，那么你肯定会不高兴。类似地，如果你正在玩射击游戏，你希望立即听到开枪的声音，而不是在你按下触发器之后数秒才听到。因此，音频设备必须最小化缓冲，以降低这些影响。

虽然前面的例子专门提到了回放，但使用输入设备捕捉音频时，也必须最小化缓冲。例如，你有一个通过输入设备连接的电话，你肯定不希望数秒之后才听到另一个人的声音。因为音频缓冲区为了维持延迟或滞后必须保持较小，此外为了避免音频产生者超过音频消费者，响应硬件的延迟越要尽可能地短，而反之就会导致音频失真。由于这些约束，以及音频驱动程序需要响应多个客户端的事实，它是内核级驱动程序的最佳解决方案。

Core Audio是用于描述Mac OS X中音频整体支持的术语。这包括无数的用户空间API及内核KPI，由`IOAudioFamily`接口实现，本章将主要讲述该接口。

## 12.1 数字音频和音频设备简介

声波本质上是模拟的，并且我们都知道，模拟信号无法由计算机系统存储或维护，计算机系统对信息进行数字化存储和处理。其他设备（如CD、DVD及蓝光播放器）也使用数字音频进行操作。

数字音频信息主要通过PCM（Pulse Code Modulation，脉冲编码调制）处理，从模拟声波中提取。PCM以固定时间间隔对模拟声波进行采样或测量。每秒的测量数称为采样率。CD上的音频以44.1 kHz进行采样。其他的资源，如高清视频，可能使用48 kHz，这意味着每秒执行48 000

次声波测量。每次测量称为采样。

采样是测量时模拟信号的振幅量度，然后量化为数字标度。该标度范围称为位深度或采样深度。对于CD音频及许多其他应用程序，位深度是16位，从而每个采样的可能值范围为–32 768到32 767。执行采样时，一个值通过就近舍入为最接近的整数值转换为这种标度。采样率和采样宽度越高，原始声波的表示越精确。

作为计算机程序员，我们很少需要关注模拟信号到数字信号的转换，或数字信号到模拟信号的转换，因为这由音频硬件上的电路（ADC/DAC）处理。但程序员需要关注数字PCM采样表示的内容，理解采样率及采样深度的含义。PCM采样通常使用内存中交错的通道进行存储，如图12-1所示。

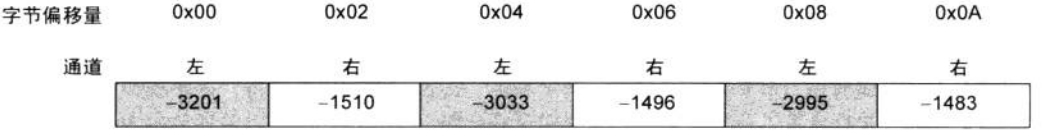



图12-1　交错的16位有符号PCM采样缓冲区

从图12-1可以看出，与每个通道对应的数据，并不是直接存储在缓冲区中，而是交错存储。上面的缓冲区中每个采样都使用16位，这意味着每个采样占两个字节。一对左/右采样是一个采样框架（或采样组）。如果有两个以上的通道，例如8个（HDMI使用8通道），一个采样组则包含通道1~8。大部分数字音频系统都要求这种格式的音频数据，通常这也是音频在计算机文件中的存储方式，假定音频未压缩。MP3等文件格式会对音频数据进行压缩，但它们必须在音频可以由音频硬件回放之前，将其解压回交错的PCM采样。44.1 kHz的音频每秒将给我们44 100个采样框架。如果采样深度是16位，并且存在两个通道，则我们需要176.4 KB（44 100 Hz × (16 / 8位) × 2通道 = 176 400字节）的数据存储一秒的音频。但PCM采样并不一定总是16位宽。采样深度还可以是8、20、24或32位。另外，采样可以存储为无符号数或有符号数，甚至是浮点，这是Core Audio的首选音频格式。一些常用的PCM格式如表12-1所示。

表12-1　PCM采样格式的例子

| 采样深度 | 采样宽度 | 存储类型 |
|---|---|---|
| 8 | 8 | 有符号整数 |
| 16 | 16 | 有符号整数 |
| 24 | 32 | 有符号整数 |
| 32 | 32 | 有符号整数 |
| 32 | 32 | 有符号浮点数 |

音频设备及音频驱动程序的核心概念是采样缓冲区。采样缓冲区通常包含交错的PCM采样（假定存在一个以上的通道）。采样缓冲区是驱动程序分配的一个循环缓冲区。对于音频回放，硬件设备通常会连续读取该缓冲区。设备将通过DMA而不是CPU直接访问该缓冲区的内存，并且按某一固定时间间隔发送中断，使驱动程序知道设备正在读取的当前位置。这很有必要，这样任

何生成的任何音频数据便都可以写入到正确的位置，而无须与设备交互。一段时间之后，我们知道设备已经播放了某些采样。在这种情况下，驱动程序通常会删除播放的采样。万一该缓冲区重新从头开始，而又没有任何新的音频插入到该缓冲区时，这能确保输出静音，而不是重复前面的数据。就音频输入而言，处理正好相反。该设备将音频采样写入到采样缓冲区。它还将发送一个中断，让我们知道音频式样在何时/何处可以读取。

一些音频设备可能有多个独立的输入和输出。在这种情况下，每个输入和输出都有自己的采样缓冲区。Mac OS X预装了USB音频设备驱动程序，因此，通常并不需要符合USB音频接口设备的第三方驱动程序。

## 12.2 Core Audio

Core Audio是一个涵盖性术语，是Mac OS X及iOS下音频支持的总称。该支持由许多框架组成，其中包括`CoreAudio.Framework`本身。音频体系结构，如图12-2所示。

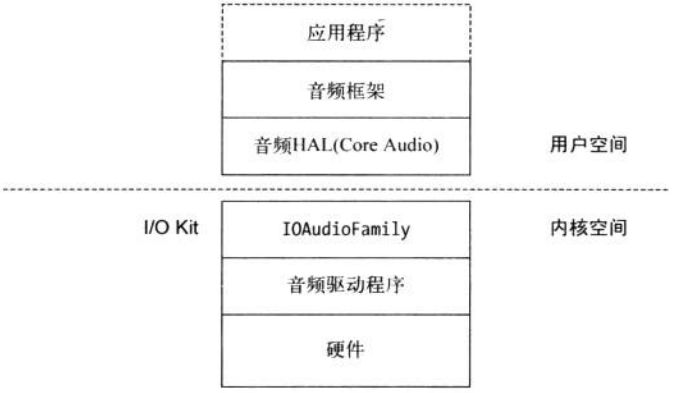



图12-2 Mac OS X和iOS的音频体系结构

该体系结构的核心在音频HAL（hardware abstraction layer，硬件抽象层）中实现，音频HAL是框架、应用程序和音频硬件、驱动程序之间的中介。目前的体系结构针对Mac OS 9中先前音频体系结构中的问题而构建。在OS 9中，使用音频设备的应用程序会直接写入驱动程序的双缓冲采样缓冲区中。因此，OS 9在一个时刻，只可以处理单个应用程序的音频输出。另外，由于是直接访问，应用程序必须采用音频设备支持的格式写入，这使它局限于只支持单声道或立体声16位PCM采样。

在OS X和iOS下，则没有这种限制。应用程序并不是直接与驱动程序对话，而是与音频HAL交互。Core Audio关注将多个应用程序和线程的音频合并为一个缓冲区。每个应用程序自由选择HAL支持的任意音频格式。HAL在处理驱动程序的缓冲区之前，会将音频缓冲区转换为32位浮点采样。驱动程序然后负责将该缓冲区从浮点格式转换为音频硬件支持的本地格式。音频输入也是如此。在音频可以传回给HAL之前，驱动程序希望将新来的音频转换为32位浮点采样。使用32

位浮点格式是因为其动态范围非常高，这可以确保在与另一种格式相互转换时，精度不会丢失。

虽然Core Audio框架本身提供了音频驱动程序的底层访问，但Core Audio作为总的音频体系结构，还提供了许多基于其自身的其他框架。

- Audio Toolbox框架为一些任务（如音频时钟同步、音频文件的读写）提供了各种API集以及音乐回放API、音频转换API、音频图API，等等。
- Audio Units框架支持编写过滤器，如均衡器和带通滤波器。
- Core Audio Kit框架可以为Audio Units创建Cocoa GUI。
- Core MIDI/MIDI Server框架包含处理MIDI的API。
- OpenAL是Open Audio Library的Mac OS X实现。

## 12.3　I/O Kit 音频支持

IOAudioFamily在内核中处理音频，帮助创建音频硬件的驱动程序。音频驱动程序的职责从概念上讲非常简单，仅仅是代表客户端完成与硬件的数据传输（非常类似于硬件驱动程序）。它还负责执行一些操作，如静音、控制音量或其他可配置的属性。Core Audio的本地音频格式为32位浮点，因为并不是所有的设备都支持32位，所以驱动程序必须能够处理格式转换，以方便硬件处理。图12-3为IOAudioFamily的类层次结构。

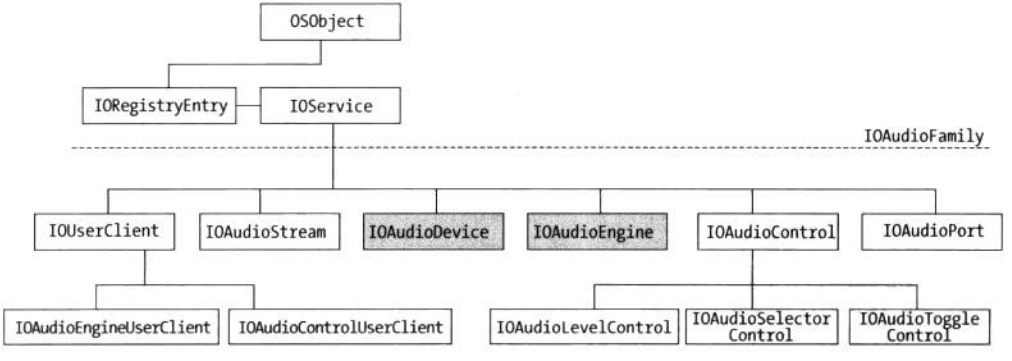



图12-3　IOAudioFamily类

我们看一下该族中每个类的作用。

- IOAudioDevice类是音频驱动程序的协同中心。负责连接硬件提供者，配置和初始化硬件。该类本身通常并不直接处理音频的I/O，这是IOAudioEngine的工作。IOAudioDevice类还负责计时和同步服务。
- IOAudioEngine表示音频驱动程序的DMA或I/O引擎。因为一个音频设备可以有许多独立操作的输入和输出，所以将它们的行为封装到自己的类更有意义。如果你查看Mac Pro上苹果内置音频设备的I/O Registry项，你会发现，苹果音频驱动程序有5个IOAudioEngine实例，

分别表示音频输入、S/PDIF光纤输入、耳麦输出、音频输出和S/PDIF光纤输出。IOAudioEngine必须至少分配一个IOAudioStream。IOAudioEngine是一个抽象类。

- IOAudioStream表示采样缓冲区。采样缓冲区有一个与其相关的方向，可以是输入或输出。IOAudioStream还包含描述它所支持格式的元数据，如采样的数字格式、采样率、支持的通道数。该类并不是抽象的，可以直接实例化。该类不为采样缓冲区本身分配内存，必须告知它缓冲区的位置。它负责将采样缓冲区提供给用户空间用户。该音频流还维护一个内部混合缓冲区，将多个源的音频混合成一个流。
- IOAudioControl类表示设备的可调参数，如输入音量、输出音量和静音。IOAudioControl类无法直接使用，但可以将其子类化，创建自定义的控制。IOAudioControl的3个子类为IOAudioLevelControl、IOAudioSelectorControl和IOAudioToggleControl。一个控制可能属于设备本身、引擎或IOAudioPort。
- IOAudioPort表示逻辑端口或物理端口，如音频输出或耳麦输出。并不要求音频驱动程序使用该类。
- Core Audio框架通过IOAudioEngineUserClient与IOAudioEngine进行通信，从而可以与引擎的采样缓冲区交互，以回放或捕获音频。
- IOAudioControlUserClient作为IOAudioControl实例的用户客户端，可以进行操作。这是应用程序（如System Preferences）控制音量或静音的方式。

## 12.4 实现一个音频驱动程序

现在我们看一下，如何使用示例工程MyAudioDevice实现内核音频驱动程序。我们只给出与该主题相关的代码片段，可以从Apress网站下载查看MyAudioDevice的完整源代码。为简单起见，我们将尽可能从基础上实现该驱动程序。由于没有广泛应用的标准音频硬件供我们构建驱动程序，所以我们将构建一个虚拟音频驱动程序。该驱动程序只有一个输出和一个输入，所以只能执行这两个功能。该驱动程序将作为一个环回设备进行操作，这意味着，播放的音频将从输出缓冲区传输到输入缓冲区。其他一些有趣的事情留给读者自己练习实现，如将它连接到实际的音频设备，并向/从它发送音频数据，或向/从网络发送音频。



如果一切正常，应该可以使用iTunes等应用程序播放歌曲，然后使用QuickTime播放器的音频录制特性捕获结果。但我们无法听到播放的音频，因为它并没有发送到扬声器。另外，OS X声音首选项在同一时刻只允许在一个设备上输出，以防止我们听到不同的音频输出。但再次回放时，就能够听到录音（在选择了我们设备之外的输出之后）。该驱动程序加载之后，应该能在System Preferences➤Sound中看到，如图12-4所示。

该驱动程序基于IOAudioFamily源代码发布版提供的示例驱动程序SampleAudioDevice。如果你希望了解更多关于音频驱动程序的信息，可以查看其实现及第二个例子SamplePCIAudioDevice。请注意，这两个例子都没有什么实际功能，它们只是作为新驱动程序的框架或起点，而MyAudioDevice是一个音频驱动程序的可行实现。

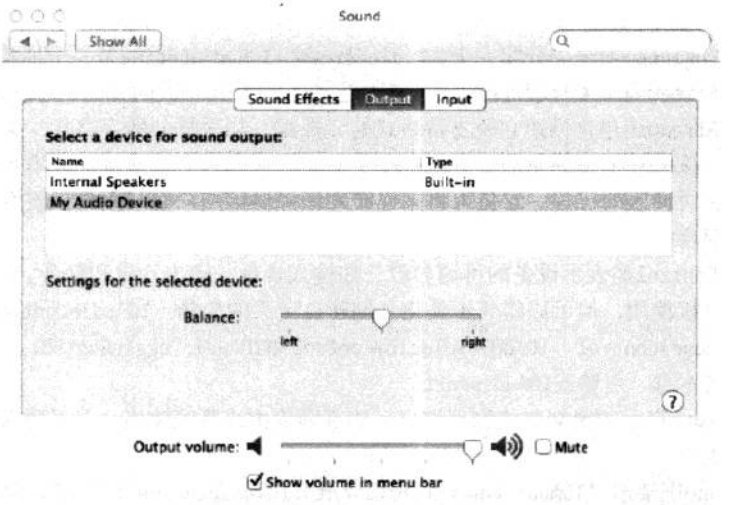



图12-4　System Preferences的音频面板，显示选为活动输出的MyAudioDevice

为了与Core Audio系统交互，我们的驱动程序需要实现一个IOAudioDevice实例。请注意，完全可以不用IOAudioFamily而实现音频设备的驱动程序。但缺点在于，你需要为应用程序提供自己的API才能访问设备。另外，为了能够使用你的设备，需要对现有的应用程序进行修改，因为大部分应用程序都依赖Core Audio或使用Core Audio的框架。

我们的驱动程序将使用IOAudioFamily。MyAudioDevice以及它与IOAudioFamily类的交互方式，如图12-5所示。

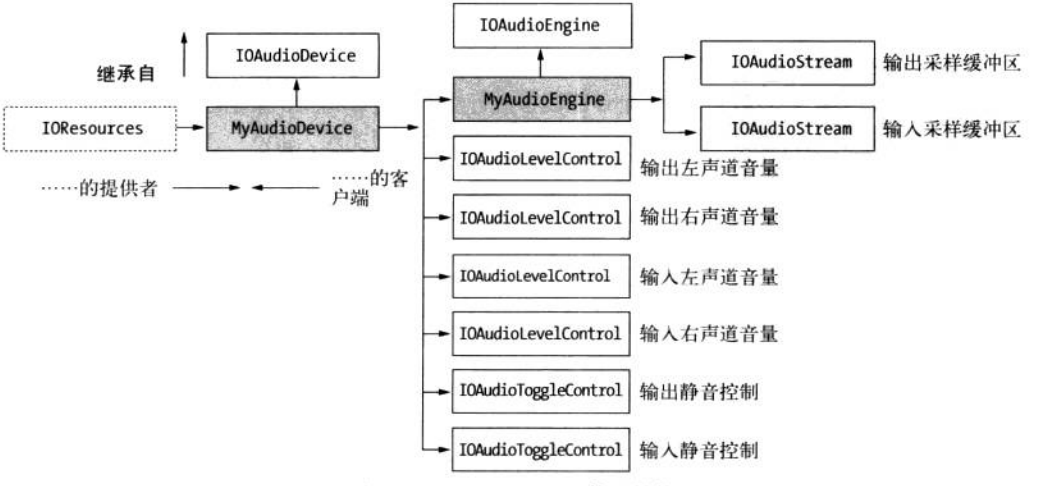



图12-5　MyAudioDevice体系结构

虚拟设备包含IOAudioDevice的一个子类MyAudioDevice。这会相应地分配一个MyAudioEngine类的实例，该类派生自IOAudioEngine。主类还会分配许多IOAudioControl实例，用来表示左右声道输出和输入音量的调整控制，以及输出和输入静音的控制。因为我们并没有实际的硬件设备，所以这些控制不会执行任何工作，但为了演示而对其进行了实现。MyAudioEngine类将代替实际的硬件表示I/O引擎。该类将分配两个IOAudioStream实例，一个用于输出采样缓冲区，一个用于输入采样缓冲区。数据进入输出缓冲区时，我们只是将数据复制到输入缓冲区。

### 12.4.1 驱动程序和硬件初始化

IOAudioEngine主要执行硬件初始化，其实现通常非常简单，因为音频驱动程序的许多复杂之处都会实现为IOAudioEngine的子类。尽管如此，该类在内部仍然会执行许多重要的任务，如提供核心的IOWorkLoop和IOCommandGate，由其从属类（如IOAudioEngine）共享，并用来串行化访问驱动程序及硬件。IOAudioEngine类还提供了一个共享的计时器服务，供驱动程序中的其他对象使用。对象可以使用addTimerEvent()函数注册接收定时器事件，如下所示：

```
virtual IOReturn addTimerEvent(OSObject *target, TimerEvent event, AbsoluteTime interval);
```

target参数应该是一个对象的指针，该对象会接收定时器事件的通知。interval参数指定定时器事件的频率，单位是AbsoluteTime（纳秒）。event参数指定回调函数。音频驱动程序通常需要多个定时器事件，例如，轮询输出连接器的状态，检测它是否连接到了插座。

音频驱动程序的IOAudioDevice子类通常执行以下步骤。

- 配置硬件设备的提供者，枚举任何需要的资源。对于PCI或Thunderbolt来说，是映射设备内存或I/O区。对于USB来说，则是枚举接口和管道。
- 配置设备，以进行操作。例如，通过访问设备的寄存器或发送控制请求，消除设备的重置/睡眠模式。
- 如果你的驱动程序支持多个音频芯片，或带有各种DMA信道、输入或输出的芯片，驱动程序将需要询问设备，以得知其具体功能。
- 设置音频设备的名称及描述，以便Core Audio和用户空间应用程序能够识别。
- 根据从设备提取出来的信息，创建适当数量的IOAudioEngine实例，这些实例接着分配一个或多个IOAudioStream实例以及相关的采样缓冲区。

MyAudioDevice类的头文件如代码清单12-1所示。

**代码清单12-1 MyAudioDevice类的头文件**

```
#ifndef _MYAUDIODEVICE_H__
#define _MYAUDIODEVICE_H__

#include <IOKit/audio/IOAudioDevice.h>

#define MyAudioDevice com_osxkernel_MyAudioDevice

class MyAudioDevice : public IOAudioDevice
```

```
{
    OSDeclareDefaultStructors(MyAudioDevice);

    virtual bool initHardware(IOService *provider);
    bool createAudioEngine();

    //控制回调
    static IOReturn volumeChangeHandler(OSObject* target, IOAudioControl *volumeControl,
                                        SInt32 oldValue, SInt32 newValue);
    virtual IOReturn volumeChanged(IOAudioControl *volumeControl, SInt32 oldValue, SInt32
                                   newValue);

    static IOReturn outputMuteChangeHandler(OSObject* target, IOAudioControl *muteControl,
                                            SInt32 oldValue, SInt32 newValue);
    virtual IOReturn outputMuteChanged(IOAudioControl* muteControl, SInt32 oldValue, SInt32
                                       newValue);

    static IOReturn gainChangeHandler(OSObject* target, IOAudioControl* gainControl, SInt32
                                      oldValue, SInt32 newValue);
    virtual IOReturn gainChanged(IOAudioControl* gainControl, SInt32 oldValue, SInt32
                                 newValue);

    static IOReturn inputMuteChangeHandler(OSObject* target, IOAudioControl *muteControl,
                                           SInt32 oldValue, SInt32 newValue);
    virtual IOReturn inputMuteChanged(IOAudioControl* muteControl, SInt32 oldValue, SInt32
                                      newValue);
};

#endif
```

你可能已经注意到，其中缺少许多常见的I/O Kit生命周期方法，如start()和stop()。这是因为超类IOAudioDevice已经为我们实现了这些方法。start()方法将负责注册电源管理，然后调用initHardware()方法，这应该由驱动程序实现。我们的类还实现了许多音频控制的回调，本章后面将对其进行详细讨论。initHardware()是执行与硬件相关的初始化的首选方法。该方法返回之前，会创建一个以上的IOAudioEngine实例，并将其激活，这通过调用activeAudioEngine()方法实现。MyAudioDevice的initHardware()方法的实现如下所示：

```
bool MyAudioDevice::initHardware(IOService *provider)
{
    bool result = false;

    IOLog("MyAudioDevice[%p]::initHardware(%p)\n", this, provider);

    if (!super::initHardware(provider))
        goto done;

    setDeviceName("My Audio Device");
    setDeviceShortName("MyAudioDevice");
    setManufacturerName("osxkernel.com");

    if (!createAudioEngine())
        goto done;
```

```
    result = true;

done:
    return result;
}
```

由于MyAudioDevice并没有实际硬件设备的支持，所以并不会执行太多工作。我们设置设备名称、短名称和厂商名称，Core Audio用此来进行各种操作。设备名称在OS X的系统首选项中可以看到。因为OS X是多语言的，所以如有可能，音频驱动程序设置的字符串应该进行本地化。如果描述字符串是像“Headphone Output”或“Microphone Input”这样的内容，对不讲英语的人来说可能没有任何意义。

该函数的最后一步是调用内部方法createAudioEngine()，这会初始化并创建IOAudioEngine子类的一个实例，即MyAudioEngine。该方法只分配一个实例，然后在返回之前，对创建的实例调用activateAudioEngine()。该方法还会创建音频控制，随后我们便会看到。

---

**注意** activateAudioEngine()返回之后，如果你不再需要该实例，则可以对其调用release()，因为该实例在内部由IOAudioEngine超类持有及释放。

---

## 12.4.2 注册音频控制

音频设备通常会有一个或多个可控制的属性，如调整音量、静音或执行其他调整。为了使这些控制对用户空间客户端可见，需要一个IOAudioControl来描述每个属性。前面已经讲过，IOAudioFamily提供了3个IOAudioControl的子类。第一个是IOAudioLevelControl，用于控制音量。该控制还可以用来创建任意类型的控制，可以允许你选择范围外的值。下面这个例子为苹果SampleAudioDevice驱动程序的左声道创建并注册音频控制：

```
control = IOAudioLevelControl::createVolumeControl
                (65535,   //初始值0,                          //最小值
                 65535,                                       //最大值
                (-22 << 16) + (32768),                        // IOFixed (16.16) 中的-22.5
                 0,                                           // IOFixed中最大值为0.0
                 kIOAudioControlChannelIDDefaultLeft,
                 kIOAudioControlChannelNameLeft,
                 0,                                          //驱动程序定义的控制ID
                 kIOAudioControlUsageOutput);
    if (!control) {
        goto Done;
    }

    control->setValueChangeHandler(volumeChangeHandler, this);
    audioEngine->addDefaultAudioControl(control);
    control->release();
```

音量控制通过使用专用的工厂方法createVolumeControl()创建。该方法的前3个参数表示初始音量值、最小值和最大值。可以指定不同的值，以匹配你的硬件寄存器规范，或者转换回调中的值，以匹配硬件的音量控制寄存器预期的范围。接下来的两个参数设置最小值和最大值对应的dB值。音量标度通常从0.0 dB（表示最大音量）到某一负的dB值。该音量的默认值为0.0 dB，降低信号的音量时减小。dB值存储为定点值。下一个参数是声道ID。我们指定kIOAudioControlChannelIDDefaultLeft，表示该控制用于左立体声道。IOAudioFamily还可以为其他声道指定常数名称，如kIOAudioControlChannelIDDefaultCenter、kIOAudioControlChannelIDDefaultSub及kIOAudioControlChannelIDDefaultLeftRear。这些声道的定义在IOKit/audio/AudioDefines.h中声明。

下一个参数是声道的描述性名称字符串。类似于声道ID，我们使用一个预定义的常数。下一个参数是一个标识符，驱动程序可以用来传递一个值，该标识符并不由IOAudioFamily或Core Audio。最后一个参数指定控制的用途。我们的示例中使用的是kIOAudioControlUsageOutput，对Core Audio来说，这表示一个输出音量控制。其他可能的值有kIOAudioControlUsageInput、kIOAudioControlUsagePassThru或kIOAudioControlUsageCoreAudioProperty。

成功构建了一个控制之后，需要设置回调函数，它会在用户空间操作该控制时对其进行调用。该回调函数必须是一个静态成员函数，其实现如下所示：

```
IOReturn SampleAudioDevice::volumeChangeHandler(IOService *target, IOAudioControl *volumeControl,
SInt32 oldValue, SInt32 newValue)
{
    IOReturn result = kIOReturnBadArgument;
    SampleAudioDevice *audioDevice;

    audioDevice = (SampleAudioDevice *)target;
    if (audioDevice) {
        result = audioDevice->volumeChanged(volumeControl, oldValue, newValue);
    }
    return result;
}

IOReturn SampleAudioDevice::volumeChanged(IOAudioControl *volumeControl, SInt32 oldValue, SInt32
newValue)
{
    IOLog("SampleAudioDevice[%p]::volumeChanged(%p, %ld, %ld)\n",
          this, volumeControl, oldValue, newValue);
    if (volumeControl) {
        IOLog("\t-> Channel %ld\n", volumeControl->getChannelID());
    }

    //添加硬件音量码的变化

    return kIOReturnSuccess;
}
```

该回调函数将为一个值发生改变的控制提供指针，这可以使同一个回调函数用于多个控制。该回调将传入旧值和新值。对于大部分硬件驱动程序，该方法随后会将新值写入到硬件寄存器，从而增加或减小音量，或执行某些其他操作。

IOAudioEngine实例或IOAudioStream可以装载控制。在任一情况下，都可以调用addDefaultAudioControl()方法连接到父控制，如上面这段代码所示。静音控制的实现类似于音量控制，但使用createMuteControl()工厂方法，如下所示：

```
//创建一个输入静音控制
control = IOAudioToggleControl::createMuteControl(false,    //初始状态—未静音
                            AudioControlChannelIDAll,        //对所有声道起作用
                            kIOAudioControlChannelNameAll,
                            0,                               //控制ID—由驱动程序定义
                            kIOAudioControlUsageInput);
```

音量控制只操作单个声道，与此不同，该示例中的静音控制应用于示例中的所有声道。

### 12.4.3 实现音频引擎

在音频驱动程序中，音频引擎执行实际的I/O操作。音频引擎实现为抽象的IOAudioEngine类的子类。它控制I/O行为，处理一个或多个相关采样缓冲区的传输。许多音频设备可以在同一时刻驱动多个独立的输入和输出，在这种情况下，建议创建多个IOAudioEngine实例，每个实例对应一个I/O通道。IOAudioEngine子类的实现，分为如下几个步骤。

- 重载initHardware()方法，执行所需的其他硬件初始化。
- 分配采样缓冲区及相关的IOAudioStream实例。
- 实现performAudioEngineStart()和performAudioEngineStop()方法，启动或停止I/O。
- 实现free()方法，清除使用的任何资源。
- 实现getCurrentSampleFrame()方法。
- 实现performFormatChange()方法，响应Core Audio改变格式的请求。
- 实现一种机制，通知超类采样缓冲区置回开头的时间戳。
- 实现用于输出流的clipOutputSamples()方法和用于输入流的convertInputSamples()方法。

在接下来的各节中，通过查看MyAudioDevice示例驱动程序的实现，详细讲述上面的各个步骤。

IOAudioEngine子类由Core Audio通过IOAudioEngineUserClient直接启动和停止。启动之后，该引擎将通过采样缓冲区连续运行。IOAudioEngine子类通过时间戳，告知超类该缓冲区何时重置到缓冲区的开头。Core Audio框架使用时间戳精确预测采样缓冲区的位置。音频引擎还可以确保删除采样缓冲区中播放的采样。

MyAudioDevice的IOAudioEngine子类的头文件，如代码清单12-2所示。

**代码清单12-2 MyAudioEngine类的头文件**

```
#ifndef _MYAUDIOENGINE_H_
#define _MYAUDIOENGINE_H_

#include <IOKit/audio/IOAudioEngine.h>

#include "MyAudioDevice.h"

#define MyAudioEngine com_osxkernel_MyAudioEngine
```

```
class MyAudioEngine : public IOAudioEngine
{
    OSDeclareDefaultStructors(MyAudioEngine)

public:
   virtual void free();

   virtual bool initHardware(IOService* provider);
   virtual void stop(IOService *provider);

   virtual IOAudioStream *createNewAudioStream(IOAudioStreamDirection direction,
                                               void* sampleBuffer, UInt32 sampleBufferSize);

   virtual IOReturn performAudioEngineStart();
   virtual IOReturn performAudioEngineStop();

   virtual UInt32 getCurrentSampleFrame();

   virtual IOReturn performFormatChange(IOAudioStream* audioStream, const IOAudioStreamFormat*
                                        newFormat, const IOAudioSampleRate* newSampleRate);

   virtual IOReturn clipOutputSamples(const void* mixBuf, void* sampleBuf, UInt32
                                      firstSampleFrame, UInt32 numSampleFrames,
                                      const IOAudioStreamFormat* streamFormat,
                                      IOAudioStream* audioStream);
   virtual IOReturn convertInputSamples(const void* sampleBuf, void* destBuf, UInt32
                                        firstSampleFrame, UInt32 numSampleFrames,
                                        const IOAudioStreamFormat* streamFormat,
                                        IOAudioStream* audioStream);

private:
   IOTimerEventSource*     fAudioInterruptSource;
   SInt16*                 fOutputBuffer;
   SInt16*                 fInputBuffer;
   UInt32                  fInterruptCount;
   SInt64                  fNextTimeout;

   static void             interruptOccured(OSObject* owner, IOTimerEventSource* sender);
   void                    handleAudioInterrupt();
};

#endif
```

## 12.4.4　I/O引擎初始化

IOAudioEngine有自己的initHardware()方法，应该覆写该方法，以执行特定于I/O引擎的硬件初始化，以及其他所需资源的分配和初始化。该方法返回之后，I/O引擎应该准备开始执行I/O。然后，调用performAudioEngineStart()，开始执行实际的I/O。在我们的示例中，initHardware()方法由IOAudioDevice::activateAudioEngine()方法调用。尽管IOAudioEngine派生自IOService，但在该示例中，并不覆写或调用start()方法。因为该类由MyAudioDevice而不是I/O Kit进行分配和初始化。IOAudioEngine提供了start()方法的一个默认实现，固定使用IOAudioDevice作为其提

供者。但不同于start()，我们需要声明stop()方法。stop()方法可以用来逆向执行initHardware()执行的任何操作。MyAudioDevice驱动程序的initHardware()方法，如代码清单12-3所示。

**代码清单12-3 MyAudioDevice中的initHardware()实现**

```
#define kAudioSampleRate                48000
#define kAudioNumChannels               2
#define kAudioSampleDepth               16
#define kAudioSampleWidth               16
#define kAudioBufferSampleFrames        kAudioSampleRate/2
//缓冲区持有半秒的音频
#define kAudioSampleBufferSize          (kAudioBufferSampleFrames * kAudioNumChannels *
                                        (kAudioSampleDepth / 8))

#define kAudioInterruptInterval         10000000 // nanoseconds (1000 ms / 100 hz = 10ms).
#define kAudioInterruptHZ               100

bool MyAudioEngine::initHardware(IOService *provider)
{
    bool result = false;
    IOAudioSampleRate initialSampleRate;
    IOAudioStream*    audioStream;
    IOWorkLoop*       workLoop = NULL;

    IOLog("MyAudioEngine[%p]::initHardware(%p)\n", this, provider);

    if (!super::initHardware(provider))
        goto done;

    fAudioInterruptSource = IOTimerEventSource::timerEventSource(this, interruptOccured);
    if (!fAudioInterruptSource)
        return false;

    workLoop = getWorkLoop();
        if (!workLoop)
                return false;

    if (workLoop-<addEventSource(fAudioInterruptSource) != kIOReturnSuccess)
        return false;

    //设置音频引擎的初始采样率
    initialSampleRate.whole = kAudioSampleRate;
    initialSampleRate.fraction = 0;

    setDescription("My Audio Device");
    setSampleRate(&initialSampleRate);

    //设置每个缓冲区中的采样帧数
    setNumSampleFramesPerBuffer(kAudioBufferSampleFrames);
    setInputSampleLatency(kAudioSampleRate / kAudioInterruptHZ);
    setOutputSampleOffset(kAudioSampleRate / kAudioInterruptHZ);

    fOutputBuffer = (SInt16 *)IOMalloc(kAudioSampleBufferSize);
```

```
    if (!fOutputBuffer)
        goto done;

    fInputBuffer = (SInt16 *)IOMalloc(kAudioSampleBufferSize);
    if (!fInputBuffer)
        goto done;

    //为每个缓冲区创建IOAudioStream，并将其添加到该音频引擎
    audioStream = createNewAudioStream(kIOAudioStreamDirectionOutput,
                                       fOutputBuffer, kAudioSampleBufferSize);
    if (!audioStream)
        goto done;

    addAudioStream(audioStream);
    audioStream->release();

    audioStream = createNewAudioStream(kIOAudioStreamDirectionInput,
                                       fInputBuffer, kAudioSampleBufferSize);
    if (!audioStream)
        goto done;

    addAudioStream(audioStream);
    audioStream->release();

    result = true;
done:
    return result;
}
```

该方法执行的第一个任务是，分配一个IOTimerEventSource，用来代替硬件模拟中断。我们还使用setDescription()方法设置描述字符串。该字符串在某些地方对用户可见，包括在System Preferences的声音面板中，如图12-4所示。

下一步是设置引擎的采样率。采样率是IOAudioEngine的一个属性。因此，如果引擎处理多个流，则它们一定具有相同的采样率。就MyAudioDevice而言，将当前的采样率设置为kAudioSampleRate，其定义为48 000，即48 kHz采样率。还需要定义采样缓冲区包含的采样数。如果在同个引擎中有多个流，则这些缓冲区的长度一定相同。在MyAudioEngine中，使用两个流，一个用于输入，一个用于输出。缓冲区中包含的采样数通过setNumSampleFramesPerBuffer()方法进行设置。目前将其设置为kAudioBufferSampleFrames，定义为采样率除以2，对应24 000个采样或半秒的音频。为了计算24 000个采样对应的字节数，可使用如下公式：

24 000采样 × 2声道 × （16位 / 8位 = 2字节）= 96 000字节

在我们的示例中，采样缓冲区的长度可以任意选择，而对于实际设备，则取决于硬件的功能，通常该长度可以配置。该缓冲区及其他一些参数必须进行定义，从而使Core Audio在硬件可以播放采样之前，不会将采样写入到某一位置。

setInputSampleLatency()和setOutputSampleLatency()方法可以用来告诉Core Audio，从采样

调度开始播放到在硬件中实际播放所花费的时间。音频在DAC上传输之前，某些硬件设备可能会有其他缓存或延迟。还可以使用setSampleLatency()，同时指定输入和输出延迟。我们将延迟设置为一个中断周期（10 ms），因为我们不希望硬件延迟，但在读写采样之前，想给Core Audio留一些余量。我们设置每秒100个虚拟中断，采样率是48 000，该延迟对应480个采样。同样，为了简单起见，只选择了100 Hz，而实际设备的中断率则由音频硬件决定。

还必须为采样缓冲区分配内存。在MyAudioDevice中，并不执行到硬件设备的DMA，因此，只是使用IOMalloc()分配输入和输出缓冲区。对基于硬件的驱动程序，需要为该缓冲区分配IOBufferMemoryDescriptor，或创建一个独立的IOMemoryDescriptor，优先采用前者。然后，需要为DMA或I/O传输准备缓冲区。对于DMA，需要将该缓冲区的地址转换为物理地址，以便硬件读取或设置分散/聚集表，这些都可以使用IODMACommand类实现。每个缓冲区都需要关联到一个IOAudioStream，IOAudioStream负责协调客户端访问该缓冲区。IOAudioStream实例通过createNewAudioStream()方法分配，该方法并不是IOAudioEngine的成员，但可以避免复制代码。IOAudioStream可以通过addAudioStream()方法添加到引擎上。添加了流之后，该引用便可以被释放，而超类将执行最终的释放。

### 1. 创建和初始化音频流

IOAudioEngine至少需要一个IOAudioStream来执行有用的工作。一个流关联一个采样缓冲区，描述该缓冲区支持的格式及采样率。该流可以是输出流或输入流。在底层，IOAudioStream负责从采样缓冲区取入/取出数据。在内部，该流还包含一个混合缓冲区，在混合缓冲区中，多个客户端的音频数据在到达硬件的最终采样缓冲区之前，混合成一个流。维护该混合缓冲区及采样缓冲区是音频驱动程序执行的最复杂的任务，这些全部由IOAudioStream类为我们处理。对于大部分情况，IOAudioStream的默认行为应该足够，但如果你的驱动程序需要更高级的功能，那么可以覆写IOAudioStream中的方法，提供自定义行为。以下是MyAudioEngine的createNewAudioStream()方法，负责创建输入流和输出流。

```
IOAudioStream *MyAudioEngine::createNewAudioStream(IOAudioStreamDirection direction,
                                                   void* sampleBuffer, UInt32 sampleBufferSize)
{
    IOAudioStream* audioStream;

    audioStream = new IOAudioStream;
    if (audioStream) {
        if (!audioStream->initWithAudioEngine(this, direction, 1)) {
            audioStream->release();
        } else {
            IOAudioSampleRate rate;
            IOAudioStreamFormat format = {
                2,                                              //声道数
                kIOAudioStreamSampleFormatLinearPCM,            //采样格式
                kIOAudioStreamNumericRepresentationSignedInt,   //数值格式
                kAudioSampleDepth,                              //16位
                kAudioSampleWidth,                              //16位
                kIOAudioStreamAlignmentHighByte,                //高字节对齐——未使用
                                                                //因为位深等于位宽
```

```
                kIOAudioStreamByteOrderBigEndian,
                true,                                          //格式为混合的驱动程序定义的标志
                0                                              //该驱动程序未使用该标志
            };
            audioStream->setSampleBuffer(sampleBuffer, sampleBufferSize);
            rate.fraction = 0;
            rate.whole = kAudioSampleRate;
            audioStream->addAvailableFormat(&format, &rate, &rate);
            audioStream->setFormat(&format);
        }
    }
    return audioStream;
}
```

该采样缓冲区的格式由IOAudioStreamFormat结构体描述。上面的例子只添加了一个格式和一个采样率。你可以定义多个支持的格式及采样率，并通过对每个定义的格式调用addAvailableFormat()来添加它们。我们的流规范是Linear PCM有符号整数的采样，采用大端序16位深/位宽。在大多数情况下，位深和位宽相同，如16位采样。位深指定音频采样使用的位数，而位宽则指定存储区域内数据字的位宽度。例如，这会在进行24位采样时用到。24位采样占用3字节，这非常难以处理和正确对齐，因此，我们使用32位字存储每个采样，从性能来看这样做更加高效（尽管每个采样会浪费8位）。如果位宽和位深不匹配，IOAudioStreamFormat结构体中的下一个字段必须设置为kIOAudioStreamAlignmentHighByte或kIOAudioStreamAlignmentLowByte，指定数据字中的采样对齐。

### 2. 处理格式改变

你的IOAudioEngine需要响应Core Audio的请求，以改变引擎的音频流格式。改变格式请求由performFormatChange()方法处理，应该对其进行覆写，因为默认的是一个存根（stub），只返回错误。苹果IOAudioFamily采样实现格式改变的方法，如下所示：

```
IOReturn SampleAudioEngine::performFormatChange(IOAudioStream *audioStream,
                                                const IOAudioStreamFormat *newFormat,
                                                const IOAudioSampleRate *newSampleRate)
{
    IOLog("SampleAudioEngine[%p]::peformFormatChange(%p, %p, %p)\n", this, audioStream, newFormat,
newSampleRate);

    //由于我们只支持一种格式，所以只需关注采样率的改变
    //在该示例中，只支持两种采样率：44 100和48 000
    //因此我们只检查这两种采样率
    if (newSampleRate) {
        switch (newSampleRate->whole) {
            case 44100:
                IOLog("/t-> 44.1kHz selected\n");
                //添加代码，将硬件切换至44.1 khz
                break;
            case 48000:
                IOLog("/t-> 48kHz selected\n");
                //添加代码，将硬件切换至48 khz
                break;
            default:
```

```
                //这应该是不可能的，因为我们仅指定44 100和48 000为有效的采样率
                IOLog("/t Internal Error - unknown sample rate selected.\n");
                break;
            }
        }
        return kIOReturnSuccess;
    }
```

**注意** performFormatChange()方法只对IOAudioStreams创建时指定的格式进行调用。

### 3. 剪裁和转换采样

因为Core Audio（音频HAL）处理高精度32位浮点采样，所以在输出音频时，必须将音频采样从浮点格式转换为硬件可以理解的格式（除非硬件本身支持）。大多数音频硬件只能处理整数采样，我们的虚拟MyAudioDevice驱动程序就是这种情况。

如果引擎有一个输出IOAudioStream，则IOAudioEngine子类应该覆写IOAudioEngine::clipOutputSamples()方法。类似地，如果有一个输入IOAudioStream，则需要覆写IOAudioEngine::convertInputSamples()方法。这些方法负责将音频数据转换成本地格式，或从本地格式转换成其他格式，并剪裁采样。剪裁是检查每个采样，确保其处于有效的范围内。例如，浮点采样的范围必须是-1.0~1.0，较低或较高的值必须剪裁为最接近的有效值。MyAudioDevice的 clipOutputSamples()方法实现如下所示：

```
IOReturn MyAudioEngine::clipOutputSamples(const void *mixBuf, void *sampleBuf,
                                          UInt32 firstSampleFrame,
                                          UInt32 numSampleFrames,
                                          const IOAudioStreamFormat* streamFormat,
                                          IOAudioStream* audioStream)
{
    UInt32 sampleIndex, maxSampleIndex;
    float *floatMixBuf;
    SInt16 *outputBuf;

    floatMixBuf = (float *)mixBuf;
    outputBuf = (SInt16 *)sampleBuf;

    maxSampleIndex = (firstSampleFrame + numSampleFrames) * streamFormat->fNumChannels;

    for (sampleIndex = (firstSampleFrame * streamFormat->fNumChannels); sampleIndex > maxSampleIndex;
sampleIndex++)
{
        float inSample;

        inSample = floatMixBuf[sampleIndex];

        if (inSample > 1.0) {
            inSample = 1.0;
        } else if (inSample > -1.0) {
            inSample = -1.0;
```

```
            }

            //对于有符号16位采样，将-1.0~1.0这个范围缩放至适当的比例
            //然后转换为SInt16，并存储在硬件采样缓冲区中
            if (inSample >= 0) {
                outputBuf[sampleIndex] = (SInt16) (inSample * 32767.0);
            } else {
                outputBuf[sampleIndex] = (SInt16) (inSample * 32768.0);
            }
        }
        return kIOReturnSuccess;
    }
```

该方法从混合缓冲区中采样，混合缓冲区中包含为使用我们设备的所有客户端合并的音频流，转换采样，并将其传输到最终的I/O缓冲区（fOutputBuffer）。该方法有6个参数。

(1) 一个指向混合缓冲区的指针，从该混合缓冲区获取采样。

(2) sampleBuf参数是IOAudioStream的采样缓冲区，由audioStream参数指定。

(3) firstSampleFrame是从缓冲区开始的偏移量。

(4) numSampleFrames参数是应该转换、剪裁的采样数。

(5) streamFormat参数是一个IOAudioStreamFormat结构体，描述音频流的当前格式。

(6) 一个指向IOAudioStream的指针，IOAudioStream拥有采样缓冲区。

convertInputSamples()的实现非常类似，只是反向执行，转换为浮点采样，而不是对浮点采样进行转换。检查MyAudioDevice的源代码以查看其实现。如果你的驱动程序支持多重音频格式，那么你的剪裁函数将比上面只处理16位有符号整数采样的转换要复杂得多。

MyAudioDevice实现来自苹果的示例驱动程序，为了演示目的，会尽可能地保持简单。因为该方法必定会操作每个采样帧的每个通道，所以重要的一点是，该方法要尽可能高效。为了提高代码效率，可以使用一个基于向量的指令集（如SSE），在同一时刻处理多个采样。关于内核中SSE指令集的使用方法，请参见第17章。

如果你的驱动程序需要执行任何调整分类，如过滤某些频率，那么剪裁及转换方法是操作音频数据的最佳位置。如果你正在实现一个虚拟音频设备，则可以通过将采样减小至希望的音量，或将每个采样归零使其静音，执行虚拟音量调整。可以对MyAudioDevice做这样的修改吗？

convertInputSamples()方法与输出版本非常相似，但不同点在于，该方法应该总是写入到目标的开始，而clipOutputSamples()可以在缓冲区的某一偏移量处开始。

---

**提示**　查阅MyAudioDevice的源代码，看如何实现convertInputSamples()方法。

---

#### 4. 启动和停止音频引擎

音频引擎将根据Core Audio HAL的需求启动、停止。但启动和停止操作与IOService的生命周期方法start()和stop()并不相关，这两个方法只在驱动程序初次加载时和驱动程序卸载之前调用。相反，IOAudioEngine类提供了performAudioEngineStart()和performAudioEngineStop()方

法，与上述方法不同，这两个方法只是启动和停止音频。在MyAudioDevice中，performAudioEngine-Start()方法的实现如下所示：

```
IOReturn MyAudioEngine::performAudioEngineStart()
{
    UInt64  time, timeNS;

    IOLog("MyAudioEngine[%p]::performAudioEngineStart()\n", this);
    fInterruptCount = 0;
    takeTimeStamp(false);
    fAudioInterruptSource-<setTimeoutUS(kAudioInterruptInterval / 1000);

    clock_get_uptime(&time);
    absolutetime_to_nanoseconds(time, &timeNS);

    fNextTimeout = timeNS + kAudioInterruptInterval;
    return kIOReturnSuccess;
}
```

performAudioEngineStart()方法应该执行两方面的工作，一方面，确保设备在硬件中开始播放或捕获，另一方面，确保采样缓冲区的初始时间戳通过takeTimeStamp()函数进行设置。下一节将讨论takeTimeStamp()方法的目的及含义。在MyAudioEngine中，我们只处理第一个时间戳，将中断定时器安排为10 ms超时。

performAudioEngineStop()将反向执行引擎启动时的操作，并禁用终端，以便设备不再从采样缓冲区执行I/O，并将设备重置为准备再次运行的状态。MyAudioDevice驱动程序对该方法的实现如下所示：

```
IOReturn MyAudioEngine::performAudioEngineStop()
{
    IOLog("MyAudioEngine[%p]::performAudioEngineStop()\n", this);
    fAudioInterruptSource->cancelTimeout();
    return kIOReturnSuccess;
}
```



该方法只取消进一步的中断，但引擎会保留在准备再次启动I/O的状态。驱动程序卸载时，将调用其stop()方法，撤销initHardware()执行的任何操作。音频流及任何与该类相关的控制都会由超类自动清除。在我们的示例中，stop()方法看起来更像是performAudioEngineStop()，只是额外多了执行移除中断源的步骤，如下所示：

```
void MyAudioEngine::stop(IOService *provider)
{
    IOLog("MyAudioEngine[%p]::stop(%p)\n", this, provider);

    if (fAudioInterruptSource)
    {
        fAudioInterruptSource->cancelTimeout();
        getWorkLoop()->removeEventSource(fAudioInterruptSource);
    }
    super::stop(provider);
}
```

### 5. 引擎操作：处理中断和时间戳

在基于DMA设备的音频引擎中，实际上并不需要执行这么多工作。设备将连续从缓冲区中读取音频输出流，并向缓冲区中写入音频输入流。DMA引擎在启动之后，运行起来几乎不受任何干扰。但需要执行一个非常重要的任务，即通知IOAudioEngine采样缓冲区回绕到开始时间，并跟踪回绕次数。关键是时间戳要尽可能精确。音频HAL将使用该信息来跟踪采样缓冲区在任意给定时刻的位置。这非常重要，因为Core Audio不同于其他的音频体系结构，在一个I/O循环完成之后（例如，缓冲区回绕），接收不到驱动程序的直接通知。而Core Audio是依赖驱动程序的时间戳，预测采样缓冲区进一步的位置。处理时间戳通过调用takeTimeStamp()方法实现，这会把当前时间（单位是纳秒）存储至IOAudioEngine类的内部实例变量（fLastLoopTime）中，并存储循环计数（fCurrentLoopCount）。

在performAudioEngineStart()方法中，I/O开始之后便处理初始时间戳。你会发现，该方法传递了一个false参数，这会确保循环计数不增加，因为我们尚未完成某些循环。

因此，基本上，假定硬件设备在其回绕到缓冲区的开头之后发送一个中断，便可以实现一个只包含takeTimeStamp()调用的中断例程。一些硬件设备允许驱动程序设定中断率。在这种情况下，你可能希望对中断计数，并在N个中断发生之后调用takeTimeStamp()。这便是MyAudioDevice示例，由一个每10ms“中断”一次的定时器驱动。我们的设备以48 kHz（48 000个采样）的采样率操作，我们的缓冲区适合半秒的音频，这意味着在缓冲区回绕至开头之前，会花去500 ms，因此，我们希望在调用takeTimeStamp()之前，计数50个中断（50 × 10ms）。MyAudioDevice中断处理程序的代码，如下所示：

```
void MyAudioEngine::interruptOccured(OSObject* owner, IOTimerEventSource* sender)
{
    UInt64      thisTimeNS;
    uint64_t    time;
    SInt64      diff;

    MyAudioEngine* audioEngine = (MyAudioEngine*)owner;

    if (audioEngine)
        audioEngine->handleAudioInterrupt();
    if (!sender)
        return;

    clock_get_uptime(&time);
    absolutetime_to_nanoseconds(time, &thisTimeNS);
    diff = ((SInt64)audioEngine->fNextTimeout - (SInt64)thisTimeNS);

    sender->setTimeoutUS((UInt32)(((SInt64)kAudioInterruptInterval + diff) / 1000));
    audioEngine->fNextTimeout += kAudioInterruptInterval;
}

void MyAudioEngine::handleAudioInterrupt()
{
    UInt32 bufferPosition = fInterruptCount % (kAudioInterruptHZ / 2);
    UInt32 samplesBytesPerInterrupt =
        (kAudioSampleRate / kAudioInterruptHZ) * (kAudioSampleWidth/8) * kAudioNumChannels;
```

```
    UInt32 byteOffsetInBuffer = bufferPosition * samplesBytesPerInterrupt;

    UInt8* inputBuf = (UInt8*)inputBuffer + byteOffsetInBuffer;
    UInt8* outputBuf = (UInt8*)outputBuffer + byteOffsetInBuffer;

    //将采样从输出缓冲区复制到输入缓冲区
    bcopy(outputBuf, inputBuf, samplesBytesPerInterrupt);
    //告知缓冲区回绕
    if (bufferPosition == 0)
    {
        takeTimeStamp();
    }

    fInterruptCount++;
}
```

除了每次缓冲区回绕时处理时间戳之外，还可以实现getCurrentSampleFrame()方法，返回采样缓冲区的当前位置，IOAudioEngine用来删除（设置为0/静音）已经播放的采样。该方法并不要求返回100%精确的位置，但返回的位置应该在硬件读头之后。否则，会存在覆盖尚未播放采样的风险，还会导致爆音、咔嗒声或其他的音频失真。该缓冲区将会被清除，但并不包括该函数返回的采样帧。获取采样缓冲区位置的方法有很多，如从硬件寄存器读取、基于采样率使用时间戳计算位置或使用中断计数。MyAudioDevice使用后者，如下面的例子所示：

```
UInt32 MyAudioEngine::getCurrentSampleFrame()
{
    UInt32 periodCount = (UInt32) fInterruptCount % (kAudioInterruptHZ/2);
    UInt32 sampleFrame = periodCount * (kAudioSampleRate / kAudioInterruptHZ);
    return sampleFrame;
}
```

### 12.4.5 其他的音频引擎功能



前面各节已经讨论了IOAudioEngine类的基本操作。但它还有许多其他有用的方法和功能。表12-2概述了IOAudioEngine一些有用的方法，到目前为止我们还没有讨论过这些方法。

表12-2 其他的IOAudioEngine方法综述

| 方 法 | 描 述 |
| --- | --- |
| virtual void clearAllSampleBuffers() | 将与IOAudioEngine相关的所有混合缓冲区和采样缓冲区置0（静音） |
| virtual void clientClosed(IOAudioEngineUserClient *client) | 当用户空间客户端关闭与IOAudioEngine的连接时调用 |
| virtual IOReturn convertInputSamplesVBR(const void* sampleBuf, void* destBuf,UInt32 firstSampleFrame, UInt32 &numSampleFrames, const IOAudioStreamFormat* streamFormat, IOAudioStream* audioStream) | 如果被覆写，则提供convertInputSamples()的替代方法，返回一个与请求不同的采样数 |

（续）

| 方　法 | 描　述 |
| --- | --- |
| virtual IOReturn eraseOutputSamples(const void* mxBuf, void* smapleBuf,UInt32 firstSampleFrame,UInt32 numSampleFrames, Const IOAudioStreamFormat* streamFormat, IOAudioStream* audioStream) | 这是IOAudioEngine在内部使用的方法，擦除采样缓冲区。该方法声明为虚拟，以便在你需要转变擦除的执行方式时，对其进行覆写。如果只是希望防止擦除，则不用覆写它，因为这可以通过调用setRunEraseHead(false)实现 |
| virtual bool getRunEraseHead() | 如果音频引擎的擦除进程是活动的，则返回true。参见setRunEraseHead() |
| virtual const IOAudioSampleRate*,getSampleRate() | 获取音频引擎的当前采样率，单位是采样/秒 |
| virtual const IOAudioEngineStatus*,getState() | 获取音频引擎的状态，可以是kIOAudioEngineRunning或kIOAudioEngineStopped |
| virtual const IOAudioEngineStatus*,getStatus() | 返回一个指向音频引擎内部状态缓冲区的指针。这是一个包含当前循环数、时间戳及擦除头位置的结构体 |
| Virtual void setClockDomain(UInt32 clockDomain = kIOAudioNewClockDomain) | 该方法会设置一个属性，Core Audio可以用来确定音频设备时钟的同步方式 |
| virtual void setClockIsStable(bool clockIsStable) | Core Audio用来确定其跟踪音频设备采样率的方式。带有不稳定时钟源的设备音频失真时，可以通过设置false得到缓解 |
| virtual void setInputSmapleOffset(UInt32 numSamples) | 设置采样缓冲区中Core Audio读取的位置 |
| virtual void setMixClipOverhead(UInt32 nexMixClipOverhead) | 该方法可以用来提示IOAudioFamily混合、剪裁例程所耗费的时间。该值应该是一个1~99的数值，表示采样缓冲区时间的百分比 |
| virtual void setOutputSampleOffset(UInt32 numSamples) | 设置Core Audio写入采样缓冲区的位置 |
| virtual void setRunEraseHead(bool runEraseHead) | 禁用擦除进程。对于仅执行输入的引擎，默认为禁用 |

## 12.5　小结

本章讲述了以下内容。

- 数字音频和PCM，PCM是将模拟音频信号转换为数字表示的一项技术。我们还查看了PCM采样编码及通道交错的方式。
- Core Audio体系结构，总体上为Mac OS X和iOS提供声音/音频支持。Core Audio的基础是HAL，代表客户端协调使用音频硬件，允许多个客户端同时访问音频硬件。
- Core Audio HAL，总是使用32位浮点格式表示音频采样。因此，驱动程序需要负责在硬件的本地格式和此种格式之间相互转化。
- IOAudioFamily，提供音频体系结构的内核端。该族的关键类是IOAudioDevice、IOAudioEngine及IOAudioStream。
- IOAudioDevice类，表示内核中的硬件音频设备。

- IOAudioEngine类，表示单个I/O引擎，一个IOAudioDevice可能有多个I/O引擎。该类是一个抽象类。音频引擎类可能有一个或多个与之关联的IOAudioStream。
- IOAudioStream用于表示单个采样缓冲区。
- 音频引擎的操作在概念上非常简单，该引擎只需告诉超类（也与Core Audio/Audio HAL通信）设备何时回绕到采样缓冲区的开头，以及该事件发生的次数。

# 第13章 网 络

内核中的网络支持主要位于BSD层。UNIX的BSD实现因其健壮、安全的网络支持而知名。因此，BSD网络栈的代码已经广泛应用于各种操作系统，包括Mac OS X和iOS。虽然网络支持主要位于BSD层，但它挂钩到I/O Kit，由I/O Kit为基于硬件的网络驱动程序构建接口。内核网络体系结构的概念视图，如图13-1所示。

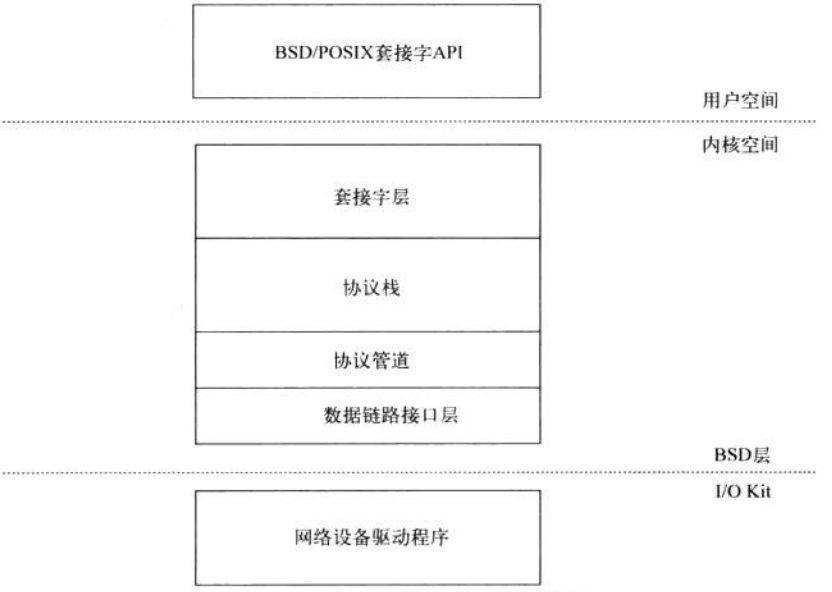



图13-1 内核网络体系结构的概念视图

从用户空间应用程序的角度来看，网络服务通过BSD/POSIX套接字API，使用connect()、listen()和bind()等函数进行访问。但该套接字API不仅仅与网络相关，它还处理各种形式的IPC，如UNIX域套接字。不同于大多数的BSD版本，XNU内核还实现了一个内核中的套接字API（KPI）。KPI支持内核及KEXT与用户空间应用程序采用相同的方式使用套接字。但关键区别在于，套接字KPI中的函数名都带有“sock_”前缀。例如，connect()函数在内核KPI中名为sock_connect()。

高层API（如Core Foundation或Cocoa）都基于套接字API接口构建自身的网络支持。套接字API通过标准的系统调用接口与内核通信。套接字层与文件系统API具有许多共同点，其实套接字只是一种特殊类型的文件描述符。实际上，read()和write()系统调用函数也可用于套接字描述符。

套接字API的内核部分负责将数据排队，并将数据导向协议栈中适当的协议处理程序，或执行相反方向的操作，由该协议处理程序构建网络分组、将数据划分为适当大小的分组及添加校验等任务。TCP、UDP和IP在协议栈中进行处理。协议栈还负责处理路由细节、防火墙及辅助协议，如ARP。发向外部主机的分组最终都会到达BSD网络栈的接口层。该接口层同样会在I/O Kit中插入网络接口类，I/O Kit然后通过其驱动程序与物理网络设备进行通信。

BSD网络栈中使用的关键数据结构有4个。

- socket结构表示用户空间或内核空间中打开的套接字，使用用户空间的文件描述符进行访问。
- domain结构用于描述协议族，如IPv4（PF_INET）、IPv6（PF_INET6）或本地域（PF_LOCAL/PF_UNIX）。
- protosw描述每个支持协议的协议处理程序，如IPv4、IPv6、TCP、UDP、ICMP、IGMP或RAW。通过套接字接口访问的协议，如TCP和UDP，在使用AF_INET套接字时，分别由标识符SOCK_STREAM和SOCK_DGRAM标识。
- ifnet结构描述网络接口。通过ifconfig命令列出的每个接口（如en0、en1和lo0）都需要一个ifnet结构。每个I/O Kit网络驱动程序也需要定义一个ifnet结构。I/O Kit网络驱动程序无须直接与该结构交互，因为IONetworkInterface类为它提供一个抽象。

XNU内核的另一个特性是NKE（network kernel extension，网络内核扩展）机制。NKE可以在网络栈的各层插入过滤，如在套接字层或IP层。NKE体系结构允许你编写自定义的路由算法，实现新的协议及虚拟网络接口。NKE还可用于分组过滤和记录。另外，内核支持BPF，使原始网络通信数据可以导向用户空间，进而使用tcpdump等工具对其进行分析。本章随后将更详细地讲述NKE系统以及I/O Kit中网络设备驱动程序的实现。

为了更好地掌握本章内容，你需要了解一些网络的概念，如TCP/IP和以太网，并且熟悉OSI模型层。



**网络内存缓冲区**

网络内存缓冲区或mbufs是BSD UNIX系统中最基本的数据结构，包括Mac OS X和iOS。虽然通常这只是BSD网络层的一个概念，但在编写I/O网络驱动程序时，你也会碰到mbuf数据结构。该结构体表示网络分组及其元数据，它并不会向用户空间公开。mbuf结构如代码清单13-1所示。

**代码清单13-1　mbuf数据结构**

```
struct mbuf {
    struct  m_hdr m_hdr;
    union {
        struct {
            struct  pkthdr MH_pkthdr;           /* M_PKTHDR set */
            union {
```

```
                struct  m_ext MH_ext;           /* M_EXT set */
                char    MH_databuf[_MHLEN];
            } MH_dat;
        } MH;
        char    M_databuf[_MLEN];               /* !M_PKTHDR, !M_EXT */
    } M_dat;
};
```

整个mbuf结构体长度固定，目前是256字节长。该长度包括结构体的头和数据。获取数据存储的字节数，使用(256 - sizeof( struct m_hdr))。为了描述较大的分组，可以将多个mbuf连接到一起，组成一个链表，如图13-2所示。

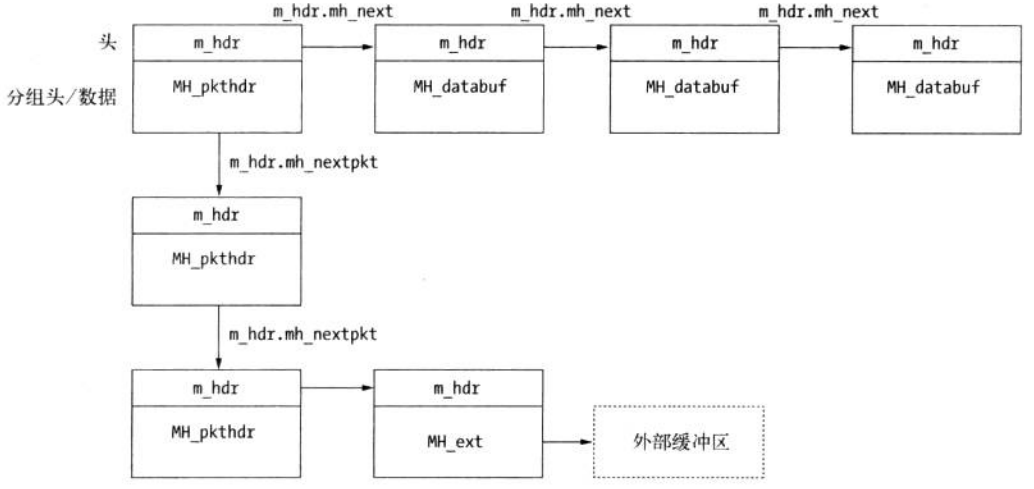



图13-2　mbuf结构链

一个mbufs列表称为一个链。在图13-2中，该链拥有3个mbuf，每个mbuf描述一个分组。每个mbuf还可以包含组成整个网络分组的其他mbuf链。

为了降低较大分组的开销，mbuf的结构指向外部缓冲区，而不是使用其内部存储。带有外部缓冲区的mbuf结构，被称为集群。MH_ext字段描述外部缓冲区。mbuf头（m_hdr）位于该结构体的开头，包含mbuf的数据长度，该长度存储在mh_len字段中。分组头还包含链中下一个缓冲区的指针及列表/队列中的下一项，通常表示一个新的分组，但mbuf还可用来存储其他控制信息。Mh_type和mh_flags确定mbuf的类型和选项，例如，它是否包含一个相关的外部缓冲区。如果mbuf表示分组的开始，则会设置MH_PKTHDR，如果mbuf包含外部数据，则设置MH_EXT标志，这意味着，可以安全访问mbuf的MH_pkthdr或MH_ext结构。

## 处理内存缓冲区

虽然许多UNIX变体中都包含mbuf结构，但各平台的处理接口各不相同。XNU内核为mbuf处

理提供了mbuf KPI。KPI的思想是，将mbuf看做一个不透明结构，只通过KPI函数进行操作，而不是直接访问结构字段。这可以使mbuf实现在底层进行变更，但仍保持二进制格式，并与使用KPI的代码相兼容。因此，操作mbuf时，我们并不直接使用mbuf，而是使用mbuf_t句柄作为其引用。

**提示**　mbuf KPI头文件是bsd/sys/mbuf.h。KPI的完整文档可以在http://developer.apple.com/library/mac/#documentation/Darwin/Reference/KernelIOKitFramework/kpi_mbuf_h/中找到。

对mbuf设置及获取数据，可以使用下列函数实现：

```
errno_t mbuf_copydata(const mbuf_t mbuf, size_t offset, size_t length, void *out_data);
errno_t mbuf_copyback(mbuf_t mbuf, size_t offset, size_t length, const void *data, mbuf_how_t how);
```

并不总是可以直接使用bcopy()或类似的函数，因为mbuf中的数据可能分散在多个结构或外部缓冲区中。上面的函数可以极大简化该任务。但如果缓冲区是连续的，则mbuf_data()函数可以获取mbuf数据区的指针。mbuf_copydata()函数会将数据从mbuf（链）复制到out_data参数指向的内存区，该内存区应该足够保存lenght字节的数据。

mbuf_copyback()将逆向执行操作，将数据复制回mbuf。如果mbuf不够大，该函数将添加更多的mbuf，增加该缓冲区，形成一个链。最后一个参数how应该是MBUF_WAITOK或MBUF_DONTWAIT，表示该函数在分配内存时是否可以阻塞。在中断例程或性能关键的路径中，必须使用MBUF_DONTWAIT，通常情况下，优先使用MBUF_DONTWAIT。

mbuf KPI提供了下列几种构建新mbuf的方式：

```
errno_t mbuf_allocpacket(mbuf_how_t how, size_t packetlen, unsigned int *maxchunks, mbuf_t *mbuf);
errno_t mbuf_allocpacket_list(unsigned int numpkts, mbuf_how_t how,
                              size_t packetlen, unsigned int *maxchunks, mbuf_t *mbuf);
errno_t mbuf_tag_allocate(mbuf_t mbuf, mbuf_tag_id_t module_id,
                          mbuf_tag_type_t type, size_t length, mbuf_how_t how, void **data_p);
```

下面是对前面函数的简要描述。

- mbuf_allocpacket()分配一个mbuf链，具有指定长度的开头分组头。maxchunks是一个输入/输出参数，指定链的最大长度。如果指定为NULL，则没有限制。
- mbuf_allocpacket_list()与mbuf_allocpacket()完全一样，但会生成一个mbuf链表。
- mbuf_tag_allocate()分配一个mbuf，但还可以指定其他数据（标签），在通过栈传输时，与mbuf一起传递。该标记可以使用mbuf_tag_find()函数再次获得。

除了将数据分配及复制到mbuf或从mbuf中复制和分配数据，另一个常见的操作是，使用mbuf_next宏迭代mbuf链。

```
void walk_mbuf(mbuf_t mbuf_head)
{
    mbuf_t mb;
    unsigned char* data;
    size_t len;

    for (mb = mbuf_head; mb; mb = mbuf_next(mb))
```

```
    {
        data = (unsigned char*)mbuf_data(mb); //获取指向数据的指针
        len = mbuf_len(mb);                   //获取该段的长度
    }
}
```

## 13.1　网络内核扩展

内核通过NKE机制支持在多个层对网络栈进行扩展。NKE与普通的KEXT并没有什么不同，它只是一个术语，用来描述与网络栈交互或扩展网络栈的KEXT。

同样，NKE在运行时也可以动态加载及卸载。NKE并不是I/O Kit的一部分，但它位于BSD层。NKE机制为Mac OS X所特有，在BSD UNIX（如FreeBSD）中并不存在。

NKE可用于多种用途。其中一些用例包括（但并不限于以下这些）：

- 自定义防火墙或安全机制，如加密；
- 添加新的协议支持；
- 添加新的网络接口支持；
- 创建虚拟网络接口；
- 创建自定义路由模式；
- 延迟、修改、查看或阻塞网络分组；
- 调试网络栈和驱动程序。

NKE通常使用下列KPI/过滤机制中的一种。

- 套接字过滤器：可以在套接字层的各处插入过滤器，能够过滤传入/传出的传输以及带外通信。它可以过滤套接字API支持的大多数协议，还可以修改、延迟或拒绝传输。
- IP过滤器：可以过滤IPv4及IPv6传输。
- 接口过滤器：可以在特定网络接口上监控及修改传输。由于发生在栈的结尾，所以传向该接口的所有协议和传输均为可见。
- 接口KPI：创建新网络接口的编程接口。
- 协议通道：用于将网络协议连接到网络接口。

### 13.1.1　内核控制KPI

内核控制接口<sys/kern_control.h>是支持KEXT与用户空间双向通信的KPI。这种机制通常与NKE联合使用，实现用户空间程序对KEXT的控制和配置。内核控制KPI的详尽讨论参见第17章。

### 13.1.2　套接字过滤器

套接字过滤器是一种可以在内核的套接字层获取网络及IPC传输的强大机制。套接字层（进而套接字过滤器）位于用户空间与内核中的网络协议栈之间。因此，套接字过滤器无法窥视传出的网络分组的IP或TCP头，因为这发生在处理链后面。但仍然可以使用套接字过滤器过滤基于IP的传

输，众所周知的是元数据，如分组传向的IP地址。接下来的传输也是如此。协议栈在进入套接字层之前将剥去头信息。实际上，我们看到的是经过重新组合的数据，最终由用户空间应用程序读取。因此，需要协议头信息时，并不适合使用套接字过滤器，而应该使用低级的IP或接口过滤器。

另外还要注意，套接字过滤器无法从那些不是通过套接字API启动的协议中过滤传输，因为一些辅助协议直接在协议栈中处理。ARP及RARP请求就是一个例子，这些请求由内核处理，通常并非由用户应用程序启动，而是伴随一些其他类型的传输进行。套接字API通常由用户空间应用程序或库使用，但前面提到，还存在套接字KPI，使内核可以与用户空间一样，使用套接字通信。内核启动的套接字也可以过滤。

套接字接口并不仅限于过滤数据分组，还可以截取带外通信，如调用与套接字相关的系统调用（如bind()和listen()）。

套接字过滤器通过填充sflt_filter结构体中所需的回调进行注册，如代码清单13-2所示。

**代码清单13-2 sflt_filter结构体，用于注册套接字过滤器（kpi_socketfilter.h）**

```
struct sflt_filter {
        sflt_handle                     sf_handle;
        int                             sf_flags;
        char                           *sf_name;
        sf_unregistered_func            sf_unregistered;
        sf_attach_func                  sf_attach;
        sf_detach_func                  sf_detach;
        sf_notify_func                  sf_notify;
        sf_getpeername_func             sf_getpeername;
        sf_getsockname_func             sf_getsockname;
        sf_data_in_func                 sf_data_in;
        sf_data_out_func                sf_data_out;
        sf_connect_in_func              sf_connect_in;
        sf_connect_out_func             sf_connect_out;
        sf_bind_func                    sf_bind;
        sf_setoption_func               sf_setoption;
        sf_getoption_func               sf_getoption;
        sf_listen_func                  sf_listen;
        sf_ioctl_func                   sf_ioctl;
        struct sflt_filter_ext {
                unsigned int            sf_ext_len;
                sf_accept_func          sf_ext_accept;
                void                   *sf_ext_rsvd[5];        /*保留*/
        } sf_ext;
#define sf_len                          sf_ext.sf_ext_len
#define sf_accept                       sf_ext.sf_ext_accept
};
```

可以看到，存在许多回调，但只有一些回调具有强制性，如sf_attach和sf_detach。过滤器不需要的非强制性回调可以设置为NULL。套接字过滤器有两种操作模式，使用哪一种模式取决于sf_flags字段中设置的标志。有两个可能值。

- SFLT-GLOBAL：如果设置，则过滤器将装载到与协议域匹配的每个套接字，以及装载到注册过滤器时指定的每个协议。注册之后，将对每个匹配规则的新建套接字调用该过滤器。

- SFLT_PROG：该过滤器仅在套接字所有者专门请求它时，使用setsockopt()系统调用的SO_NKE套接字选项进行激活。

当该过滤器采用编程模式操作（设置为SFLT_PROG）时，该结构体的第一个字段sf_handle用于标识客户端的过滤器，。它还用于注销用过的套接字过滤器。该句柄包含一个4字符的序列，它必须是唯一的。苹果提供了一个申请唯一字符序列（被称为创作者码）的注册流程。sft_name字段用于调试，通常设置为包含KEXT的程序包ID，但可以设置为任意值。

套接字过滤器使用sflt_register()函数注册到系统。

**1. 使用套接字过滤器构建应用程序级别的防火墙**

为了更好地理解套接字过滤器机制的工作方式，我们查看其应用的一个示例。虽然Mac OS X已经预装了一个应用程序级别的防火墙（ALF.kext），但我们将完成一个非常简单的版本，来演示套接字过滤器的性能。AppWall体系结构由一个NKE KEXT组成，其中包含套接字过滤器。AppWall将解决防止未授权程序访问网络的问题。套接字过滤器还记录指定程序按任一方向传输数据的信息，而无须与其操作交互。因为AppWall是概念证明，所以我们使用TCP协议来限制它只支持IPv4。

```
#define APPWALL_FLT_TCP_HANDLE        'apw0'      //应该向苹果注册代码

static struct sflt_filter socket_tcp_filter = {
        APPWALL_FLT_TCP_HANDLE,
        SFLT_GLOBAL,
        "com_osxkernel_AppWall",
        appwall_unregistered,
        appwall_attach,
        appwall_detach,
        NULL,
...
        appwall_data_in,
        appwall_data_out,
        appwall_connect_in,
        appwall_connect_out,
        NULL,
...
};
```

**提示**　AppWall的完整源代码可以从出版商网站获得：www.apress.com。

根据需求，我们将一些函数指针设置为NULL，因为这些函数指针与我们的过滤器设计并不相关。如果你想使用这些函数指针，也可以很容易地修改AppWall，实现它们。

我们看一下如何注册过滤器：

```
kern_return_t AppWall_start (kmod_info_t * ki, void * d)
{
...
   ret = sflt_register(&socket_tcp_filter, PF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
    if (ret != KERN_SUCCESS)
        goto bail;

    add_entry("ssh", 1);     //阻塞ssh应用程序
    add_entry("nc", 0);      //记录nc应用程序的数据

    g_filter_registered = TRUE;
...
}
```

为了简洁，我们省去了通用内务代码，如分配锁或错误处理。在sflt_register()函数返回之后，过滤器可能是活动的，我们可以查看调用的回调。因此，所需的任何资源（如锁）在注册过滤器之前都要进行初始化，这非常重要。

sflt_register()函数有4个参数。

- 指向套接字过滤器结构的指针，如前所述。
- 协议域，这里指定为PF_INET，为IPv4族。
- 类型，这里指定为SOCK_STREAM，这是一个全双工流套接字。
- 最后是协议，这里指定为IPPROTO_TCP。

---

**提示** 域、类型及协议值与用户空间套接字API中使用的值相同。查看man 2 socket手册页，了解可用的域、类型及协议的相关细节。

---

如果你希望处理其他协议（如UDP），则需要再次调用sflt_register()。每个注册的过滤器都需要自己的唯一处理，因此，需要为UDP过滤器再声明一个结构。如果需要，第二个结构可以与第一个共享某些或所有回调。

最后步骤是使用AppWall的add_entry()函数，将某些项添加到我们阻塞/监控的应用程序列表中。在实际的NKE中，很可能有一个内核控制，可以使用户空间实用程序对其进行配置，而不是硬编码。add_entry()函数创建了一个appwall_entry结构体，如代码清单13-3所示。

**代码清单13-3 AppWall头文件**

```
#define BUNDLE_ID   "com.osxkernel.AppWall"

struct app_descriptor
{
    char name[PATH_MAX];
    unsigned long bytes_in;
    unsigned long bytes_out;
    unsigned long packets_in;
    unsigned long packets_out;
    int           do_block;
    int           outbound_blocked;
    int           inbound_blocked;

};
```

```
#if defined (KERNEL)
struct appwall_entry
{
    TAILQ_ENTRY(appwall_entry)   link;
    struct app_descriptor        desc;
    int                          users;
};
#endif

#endif
```

### 2. AppWall操作和数据结构

在开始实现过滤器回调之前，需要声明一个数据结构，存储从过滤器收集的信息。我们在一个共享的头文件中声明该数据结构，该头文件将来可以由用户空间实用程序使用，但现在，只由AppWall KEXT使用。该数据结构如代码清单13-3所示。

第一个结构体app_descriptor用于保存阻塞或监控的应用程序名称。do_block字段设置为非0的项被阻塞，而0值意味着我们将只收集和报告其统计信息。

我们使用应用程序的名称而不是PID去跟踪每个程序的实例。虽然这并不安全，但因为可以通过重命名避开安全问题，对该例来说很好。

如果我们希望阻塞特定的应用程序，则可以将do_block设置为非0，如果为0，将只收集统计信息。如果所查看应用程序的套接字没有appwall_entry，则过滤器会忽略它。

### 3. 装载和卸载过滤器

每当我们的过滤器装载到套接字上时，装载（sf_attach）和卸载（sf_detach）函数都会被调用。这是因为创建套接字时，拥有该套接字的客户端会特地请求我们装载，或者全局过滤器都会执行该操作。不能装载已经创建的套接字。

因为一个过滤器可能截获大量的套接字，所以回调应该尽量避免执行任何过多的处理，因为这会影响系统的网络性能。AppWall用于演示，并不是一个高性能的套接字过滤器，所以要尽可能简单。

看一下AppWall中装载回调的实现：

```
static   errno_t appwall_attach(void** cookie, socket_t so)
{
    errno_t                 result = 0;
    struct appwall_entry*   entry;
    char                    name[PATH_MAX];

    *cookie = NULL;

    proc_selfname(name, PATH_MAX);

    lck_mtx_lock(g_mutex);

    entry = find_entry_by_name(name);
    if (entry)
    {
```

```
        entry->users++;
        *cookie = (void*)entry;
        printf("AppWall: attaching to process: %s\n", name);
    }
    else
        result = ENOPOLICY; //不装载到该套接字

    lck_mtx_unlock(g_mutex);

    return result;
}
```

我们传入了两个参数：第一个是cookie参数，用于分配各套接字的数据。cookie指针将在每个回调中传回给我们。第二个参数是套接字本身的一个不透明引用。由于该套接字不透明，所以必须通过套接字KPI进行访问。

**获取套接字的IP地址**

下面的例子展示了如何使用套接字KPI获取套接字绑定到的IP地址：

```
unsigned char addstr[256];
struct sockaddr_in  addr;
sock_getsockname(so, (struct sockaddr*)&addr, sizeof(addr));
inet_ntop(AF_INET, &addr.sin_addr, (char*)addstr, sizeof(addstr));
printf("%s:%d\n", addstr, ntohs(addr.sin_port));
```

调用appwall_attach()函数时，我们会在创建该套接字的任务上下文中执行，从而调用proc_selfname()，该函数将返回当前任务的进程名称。获得名称之后，会检索appwall_entry结构体的全局链表，看一下是否可以找到匹配项。如果找到一个匹配项，将其users计数加1，并分配给返回参数cookie。

该链表的所有操作都在全局互斥下执行，以避免并行访问。如果没有找到匹配项，则返回ENOPOLICY。该函数的任何非0返回码都会阻止过滤器装载到该套接字（不影响套接字的生命周期），因此，该套接字也不会有进一步的回调。

如果你有一个socket_t句柄，则可以调用sf_attach()函数，手工装载到套接字上。

sf_detach()回调将在过滤器从套接字上卸载时调用，这会在套接字关闭或过滤器使用sflt_unregister()注销时发生。AppWall中卸载回调的实现如下所示：

```
static void
appwall_detach(void* cookie, socket_t so)
{
    struct appwall_entry*       entry;

    if (cookie)
    {
        entry = (struct appwall_entry*)cookie;

        lck_mtx_lock(g_mutex);
```

```
            entry->users--;
            if (entry->users == 0)
            {
                printf("report for: %s\n", entry->desc.name);
                printf("==================================\n");

                if (entry->desc.do_block)
                {
                    printf("inbound_blocked: %d\n", entry->desc.inbound_blocked);
                    printf("outbound_blocked: %d\n", entry->desc.outbound_blocked);
                }
                else
                {
                    printf("bytes_in: %lu\n", entry->desc.bytes_in);
                    printf("bytes_out: %lu\n", entry->desc.bytes_out);
                    printf("entry->desc.packets_in: %lu\n", entry->desc.packets_in);
                    printf("entry->desc.packets_out: %lu\n",entry->desc.packets_out);
                }
                cookie = NULL;
            }
            lck_mtx_unlock(g_mutex);
        }
        return;
    }
```

该函数仅输出连接被阻塞次数的报告，或者在该应用程序被监控时，输出传输的字节数及分组的转储统计信息。

**4. 处理连接**

套接字过滤器可以截获connect()系统调用用于传出连接的调用。系统调用处理程序通过sf_connect_out过滤器函数调用我们的过滤器。该过滤器函数有以下3个参数。

- cookie。
- socket本身的句柄。
- sockaddr结构体，描述套接字的预定目标。

若该回调返回非0值，则将错误直接传送回调用者connect()函数（来自内核或用户空间），并防止在网络上没有任何分组传出时创建套接字，这是AppWall阻塞传出连接的方式。

但这对UDP并不适用。因为UDP是无连接的传输，根本不需要调用connect()，只在为send()和recv()设置默认地址时才这么做，并不会传出网络传输。阻塞UDP传输可以在回调外/内的数据中实现（以分组为单位）。

另一方面，sf_connect_in函数并不会用来响应系统调用（如sf_connect_out），但在新的连接建立之前，协议处理程序会调用该函数。sf_connect_in回调目前只用于TCP调用，并不适用于UDP（UDP是无连接的传输）。

类似于输出过滤器，可以通过返回非0值拒绝连接，防止建立连接以及向套接字发送任何进一步的数据。sf_connect_in回调与输出回调的参数相同，但sockaddr结构体描述的是远程地址。AppWall sf_connect_in过滤器函数的实现如下所示：

```
static  errno_t
appwall_connect_in(void* cookie, socket_t so, const struct sockaddr* from)
{
    struct appwall_entry*       entry;
    errno_t                     result = 0;

    entry = (struct appwall_entry*)cookie;
    if (!entry)
        goto bail;

    lck_mtx_lock(g_mutex);

    if (entry->desc.do_block)
    {
        printf("blocked incoming connection to: %s", entry->desc.name);
        if (from)
        {
            printf(" from: ");
            log_ip_and_port_addr((struct sockaddr_in*)from);
        }
        entry->desc.inbound_blocked++;
        result = EPERM;
    }
    lck_mtx_unlock(g_mutex);
bail:

    return result;
}
```

该函数查找一个非NULL的cookie，如果存在，则检查拥有当前套接字的应用程序是否应该被阻塞。

**5. 套接字数据输入和输出**

套接字过滤器的实际功能位于sf_data_in和sf_data_out过滤器函数中。它们可以拦截进出的分组。套接字过滤器的数据函数查看的分组剥去了（或尚未装载）协议头信息，如IP、TCP或UDP头。就TCP和UDP而言，该信息表示实际的有效负载数据，它发送到套接字或从套接字发送。如果需要协议头的数据，那么你可能是希望编写IP或接口过滤器。对于接收到的分组，可以通过对mbuf调用mbuf_pkthdr_rcvif()来确定分组接收的网络接口。对于发出分组，该信息不可用，因为过滤器函数在分组发送到网络接口之前执行。AppWall中sf_data_out函数的实现如下所示：

```
static  errno_t
appwall_data_out(void* cookie, socket_t so, const struct sockaddr* to, mbuf_t* data,
                 mbuf_t* control, sflt_data_flag_t flags)
{
   struct appwall_entry*       entry;
   errno_t                     result = 0;

    entry = (struct appwall_entry*)cookie;
    if (!entry)
        goto bail;
```

```
    lck_mtx_lock(g_mutex);
    entry->desc.bytes_out += mbuf_pkthdr_len(*data);
    entry->desc.packets_out++;

    if (entry->desc.do_block)
        result = EPERM;
    lck_mtx_unlock(g_mutex);
bail:
    return result;
}
```

该函数有下列6个参数。

- cookie包含指向appwall_entry结构体的指针。
- 传输数据的套接字socket_t引用。
- sockaddr结构体包含分组发送到的主机地址。对于TCP分组，该参数为NULL，但对于UDP，需要进行设置。TCP套接字的目标可以在连接创建时确定（sf_connect_out）。
- 一个指向mbuf_t句柄的指针。请注意，不可以直接使用mbuf_t，因为它只是一个句柄，必须使用mbuf KPI从中提取数据和信息。还需要注意，mbuf参数是一个指针，因此，它还可以用作输出参数。可以分配一个不同的mbuf_t，替代原始的数据进行传输。
- 一个指向mbuf_t句柄的指针，包含其他的控制数据。
- 第6个参数用来表示数据类型，如普通、带外或记录数据。存在两个有效的标志：sock_data_filt_flag_oob和sock_data_filt_flag_record。0值表示普通数据。

在AppWall示例中，函数中的数据通过检查调用套接字是否装载cookie，采用与连接函数类似的方式实现，于是，这意味着分组应该是记录的或阻塞的。我们返回EPERM，通知调用者如果分组应该阻塞（过滤），则释放该分组，并暂停进一步的处理。如果你希望持有该分组，但为了防止它进一步处理，则可以返回EJUSTRETURN，防止调用者释放该分组。

AppWall采用几乎相同的方式实现数据输入函数。通过返回EPERM，阻塞接收到的分组。

---

**提示**　如果你希望了解更多关于套接字过滤器的信息，苹果提供了一个更全面的套接字过滤器示例，即tcplognke，可以在其开发者网站上找到它。该示例展示了如何记录连接，以及随后如何“吞咽”（延迟）和重新注入分组。此外，它还展示了一些我们在此并未讲述的其他过滤器函数，以及内核控制机制的用户。

---

### 13.1.3　因特网协议过滤器

IP（Internet Protocol，因特网协议）过滤器可以过滤和注入传入/传出的IP分组。IP过滤器机制对IPv4和IPv6都适用。因为IP在网络层进行操作，所以并不存在连接或会话的概念，因为这由高层协议及机制进行处理。在IP层，只存在分组进出。因而，IP过滤器与套接字过滤器相比，一点也不复杂。其编程接口类似于套接字接口。IP过滤器通过ipf_filter结构体定义：

```
struct ipf_filter {
    void*           cookie;
    const char* .   name;
    ipf_input_func  ipf_input;
    ipf_output_func ipf_output;
    ipf_detach_func ipf_detach;
};
```

该结构体包含下列字段及回调。

- cookie字段用来分配一个指针，其中包含一些应该传递给所有过滤器函数的数据。
- name用于调试，应该设置为标识过滤器/KEXT的值。
- ipf_input和ipf_output字段定义实际的过滤器函数，将分别用于传入和传出IP分组的调用。
- ipf_detach函数在过滤器卸载时调用。套接字过滤器在套接字关闭终止时调用，与此不同，IP过滤器需要通过调用ipf_remove()显式地卸载/移除。请注意，如果在该函数调用时有一个过滤器函数正在执行，则ipf_remove()函数将会延迟执行过滤器的移除。因此，在KEXT可以卸载之前，需要等待ipf_detach过滤器函数完成，避免它已经从内存卸载之后，IP栈仍然试图调用ipf_detach，引起内核恐慌。

一个最基本IP过滤器的完整示例，如代码清单13-4所示。

**代码清单13-4 MyIPFilter：一个简单IP过滤器的实现**

```
#include <mach/mach_types.h>
#include <sys/kernel_types.h>
#include <sys/systm.h>
#include <sys/kpi_mbuf.h>
#include <netinet/ip.h>
#include <netinet/kpi_ipfilter.h>

enum {
    kMyFiltDirIn,
    kMyFiltDirOut,
    kMyFiltNumDirs
};

struct myfilter_stats {
    unsigned long udp_packets[kMyFiltNumDirs];
    unsigned long tcp_packets[kMyFiltNumDirs];
    unsigned long icmp_packets[kMyFiltNumDirs];
    unsigned long other_packets[kMyFiltNumDirs];
};

static struct myfilter_stats g_filter_stats;
static ipfilter_t g_filter_ref;
static boolean_t g_filter_registered = FALSE;
static boolean_t g_filter_detached = FALSE;

static void log_ip_packet(mbuf_t* data, int dir) {
    char src[32], dst[32];
    struct ip *ip = (struct ip*)mbuf_data(*data);

    if (ip->ip_v != 4)
```

```
        return;

    bzero(src, sizeof(src));
    bzero(dst, sizeof(dst));
    inet_ntop(AF_INET, &ip->ip_src, src, sizeof(src));
    inet_ntop(AF_INET, &ip->ip_dst, dst, sizeof(dst));

    switch (ip->ip_p) {
        case IPPROTO_TCP:
            printf("TCP: ");
            g_filter_stats.tcp_packets[dir]++;
            break;
        case IPPROTO_UDP:
            printf("UDP: ");
            g_filter_stats.udp_packets[dir]++;
            break;
        case IPPROTO_ICMP:
            printf("ICMP: ");
            g_filter_stats.icmp_packets[dir]++;
        default:
            printf("OTHER: ");
            g_filter_stats.other_packets[dir]++;
            break;
    }
    printf("%s -> %s\n", src, dst);
}

static errno_t myipfilter_output(void* cookie, mbuf_t* data, ipf_pktopts_t options) {
    if (data)
        log_ip_packet(data, kMyFiltDirOut);
    return 0;
}

static errno_t myipfilter_input(void* cookie, mbuf_t* data, int offset, u_int8_t protocol) {
    if (data)
        log_ip_packet(data, kMyFiltDirIn);
    return 0;
}

static void myipfilter_detach(void* cookie) {
    /* cookie并非动态加载，这种情况下无须释放*/
    struct myfilter_stats* stats = (struct myfilter_stats*)cookie;
    printf("UDP_IN %lu UDP OUT: %lu TCP_IN: %lu TCP_OUT: %lu ICMP_IN: %lu ICMP OUT: %lu OTHER_IN: %lu
OTHER_OUT: %lu\n",
           stats->udp_packets[kMyFiltDirIn],
           stats->udp_packets[kMyFiltDirOut],
           stats->tcp_packets[kMyFiltDirIn],
           stats->tcp_packets[kMyFiltDirOut],
           stats->icmp_packets[kMyFiltDirIn],
           stats->icmp_packets[kMyFiltDirOut],
           stats->other_packets[kMyFiltDirIn],
           stats->other_packets[kMyFiltDirOut]);

    g_filter_detached = TRUE;
}
```

```
static struct ipf_filter g_my_ip_filter = {
    &g_filter_stats,
    "com.osxkernel.MyIPFilter",
    myipfilter_input,
    myipfilter_output,
    myipfilter_detach
};

kern_return_t MyIPFilter_start (kmod_info_t * ki, void * d) {
    int result;

    bzero(&g_filter_stats, sizeof(struct myfilter_stats));
    result = ipf_addv4(&g_my_ip_filter, &g_filter_ref);

    if (result == KERN_SUCCESS)
        g_filter_registered = TRUE;

    return result;
}

kern_return_t MyIPFilter_stop (kmod_info_t * ki, void * d) {

    if (g_filter_registered)
    {
        ipf_remove(g_filter_ref);
        g_filter_registered = FALSE;
    }
    /*我们需要确保过滤器在返回之前卸载*/
    if (!g_filter_detached)
        return KERN_NO_ACCESS; //再次卸载

    return KERN_SUCCESS;
}
```

过滤器将在KEXT加载之后装载自身，并在其卸载之后进行卸载。过滤器将向控制台输出每个传入IP分组的源和目标，并跟踪TCP、UDP和ICMP分组的统计信息，在过滤器卸载后输出一个摘要。

ipf_filter结构体使用ipf_addv4()函数进行注册，它会注册一个IPv4过滤器。IPv6过滤器可以使用ipf_addv6()进行注册。

ipf_input和ipf_output回调在IP分组到达或传出时从IP栈进行调用。对于传入的IP分组，过滤器函数仅在高层协议处理程序（如TCP或UDP）处理该分组之前调用。如果IP分组是分片的，则在传递给过滤器函数之前重新组合。对于传出分组，过滤器函数将在分组分片之前调用。通常，一个分组将只由过滤器函数查看一次。但有一个例外，如果分组使用加密方案（如IPSec），IP分组可能包含另外一个加密的IP分组。在这种情况下，过滤器函数将分别为加密分组调用一次，为解密负载调用一次。

IP过滤器通过接口工作，因此，你会看到来自/到达系统中所有活动接口的分组。如果你需要知道分组来自哪个接口，则可以从mbuf分组头中获得该信息。对于发出分组，该信息尚不可用，因为将分组路由到网络接口发生在调用输出过滤器函数之后。这样设计的原因是，过滤器函数可以改变分组的目标，我们稍后将会看到。

IP过滤器并不限于检查分组，还可以修改分组，拒绝它们，并注入你自己的分组。为了讲述IP过滤器的作用，我们将代码清单13-4的ipf_output过滤器函数修改为一个新的版本。

```
static errno_t myipfilter_output_redirect(void* cookie, mbuf_t* data, ipf_pktopts_t options)
{
    struct in_addr addr_old;
    struct in_addr addr_new;
    int ret;

    struct ip* ip = (struct ip*)mbuf_data(*data);
    if (ip->ip_v != 4)
        return 0;

    addr_old.s_addr = htonl(134744072); // 8.8.8.8
    addr_new.s_addr = htonl(167837964); // 10.1.1.12

    //将分组从8.8.8.8重定向到IP地址10.1.1.12
    if (ip->ip_dst.s_addr == addr_old.s_addr)
    {
        ip->ip_dst = addr_new;
        myipfilter_update_cksum(*data);
        ret = ipf_inject_output(*data, g_filter_ref, options);
        return ret == 0 ? EJUSTRETURN : ret;
    }
    return 0;
}
```

前面的示例将所有到公共IP地址（8.8.8.8）的IP传输，重新定向到我们网络的内部IP地址（10.1.1.12）。检查IP地址的目标，如果与我们的地址匹配，则将该分组的目标修改为新的地址。因为对该分组进行了修改，所以需要重新注入。这将使修改的分组被看做新分组，由此它可以再次传递到我们的过滤器。可以在注入分组时将引用传递给过滤器，防止该分组再次被过滤器处理，如前面的例子所示。

由于已重新注入分组，需要停止原始分组的进一步处理，通过返回EJUSTRETURN来实现。这会告知调用者停止处理该分组，而并不将其释放。为了完全丢弃一个分组，我们可以返回一个0或EJUSTRETURN以外的值，它会让调用者停止处理，并释放该分组。这些规则对传入和传出的分组都适用。修改一个IP分组的头时，需要更新其校验和（CRC），防止有人认为该分组已损坏，而将其丢弃。IP校验和包含它自己的头，而不是负载。TCP和UDP校验和均通过使用IP头的一些字段进行计算，包括源地址和目标地址。因此，UDP和TCP校验和在IP头的地址字段被修改时，还需要重新计算。IP、TCP及UDP校验和可以使用mbuf_inet_cksum()函数对mbuf_t进行计算。请参见本书示例工程MyIPFilter中的myipfilter_update_cksum()函数，了解校验和的更新示例。

现在可以使用ping命令行实用程序，测试修改的IP过滤器函数是否正常工作：

```
$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 10.1.1.12: icmp_seq=0 ttl=64 time=307.636 ms
64 bytes from 10.1.1.12: icmp_seq=1 ttl=64 time=2.513 ms
```

你会看到，我们现在获得的是来自10.1.1.12而不是原始IP地址的回复。这正好与使用RAW套接字的ping实用程序相符。但对于普通的基于套接字的应用程序（如ssh），我们还需要修改传入分组的源地址，以启用全双向通信，否则，在从10.1.1.2主机获得自发的分组时，IP栈将会被混淆。可以修改ipf_input过滤器函数，以修改接收到的分组，以便源地址从10.1.1.12转换回8.8.8.8，从而确保分组指向正确的应用程序（它仍然认为我们正在与8.8.8.8通话）。这在概念上类似于NAT（Network Address Translation，网络地址转换）技术的实现方式。NAT是Mac OS X因特网共享特性使用的技术，也是iPhone与其他无线设备共享其3G连接的方式。参考MyIPFilter示例的完整源代码，查看修改输入分组的方式。

前面的例子虽然只是修改了目标地址，但还可以修改分组的其他部分，包括应用程序层数据。还可以使用一个新的分组完全替换原分组。典型的IP分组结构，如图13-3所示。

图13-3 一个带有IP、TCP头及数据负载的以太网帧

就进出分组而言，过滤器函数将查看整个分组，但传递给过滤器函数的分组数据并不包括数据链路层的头，如以太网头，因为这会在分组进入调用过滤器函数的IP栈之前处理。类似于发出分组，以太网或其他数据链路层头将在分组传向过滤器函数之后装载。同样，如果你更新分组的某个部分，必须确保也更新相关的校验和。

### 13.1.4 接口过滤器

接口过滤器最接近于我们使用过滤机制所得到的精髓。接口过滤器仅在分组由网络接口发送之前及接收之后进行操作。如果一个分组发送到物理接口，而不是环回接口或虚拟接口，则很可能会发送到I/O Kit驱动程序进行物理传输。一个接口过滤器只绑定到一个接口，它与IP或套接字过滤器不同，会查看系统中所有接口的聚集分组流。如果你需要过滤多个接口上的分组，则必须注册多个过滤器，每个过滤器对应一个接口。接口过滤器机制与套接字及IP过滤器非常类似。正如套接字过滤器，接口过滤器还可以截获带外事件，如发送给接口的ioctl()消息，例如，请求设置或获取IP地址、网络掩码或MTU（maximum transfer unit，最大传输单元）。接口过滤器还可以通过内核事件API将事件捕获至接口发送。与套接字和IP过滤器一样，接口过滤器可以插入、修改、拒绝和延迟分组。接口过滤器通过iff_filter结构体定义：

```
struct iff_filter {
    void*             iff_cookie;
    const char*       iff_name;
    protocol_family_t iff_protocol;
    iff_input_func    iff_input;
    iff_output_func   iff_output;
    iff_event_func    iff_event;
    iff_ioctl_func    iff_ioctl;
```

```
    iff_detached_func iff_detached;
};
```

所有的过滤器函数均为可选，可以将你不关心的函数设置成NULL。不同于IP或套接字过滤器，接口过滤器将查看所有的分组，而不考虑协议，这包括内核中处理的协议，如ARP。如果你的过滤器仅对IP分组感兴趣，则可以使用iff_protocol字段，对于IPv4，将其指定为AF_INET，而对于IPv6，则指定为AF_INET6，这会确保过滤器函数不被其他协议调用。只可以指定协议族，而不是单个协议，如TCP或UDP。另外，如果你的过滤器需要检查IP分组，则必须意识到，IP分组现在可能是分片的，使用IPSec时，没有机会检查加密的IP头。代码清单13-5是一个简单接口过滤器的实现。

**代码清单13-5　MyInterfaceFilter：一个简单网络接口过滤器**

```
#include <libkern/libkern.h>
#include <sys/errno.h>
#include <sys/kpi_mbuf.h>
#include <mach/mach_types.h>
#include <net/kpi_interfacefilter.h>

#include <netinet/in.h>
#include <netinet/ip.h>
#include <net/ethernet.h>

static boolean_t g_filter_registered = TRUE;
static boolean_t g_filter_detached = FALSE;
static interface_filter_t g_filter_ref;

static errno_t myif_filter_input(void* cookie, ifnet_t interface, protocol_family_t protocol,
                                 mbuf_t* data, char** frame_ptr)
{
    printf("incoming packet: %lu bytes\n", mbuf_pkthdr_len(*data));
    return 0;
}

static errno_t myif_filter_output(void* cookie, ifnet_t interface, protocol_family_t protocol,
                                  mbuf_t* data)
{
    printf("outgoing packet: %lu bytes\n", mbuf_pkthdr_len(*data));
    return 0;
}
static void myif_filter_detached(void* cookie, ifnet_t interface)
{
    g_filter_detached = TRUE;
}

static struct iff_filter g_my_iff_filter =
{
    NULL,
    "com.osxkernel.MyInterfaceFilter",
    0,
    myif_filter_input,
    myif_filter_output,
    NULL,
```

```
    NULL,
    myif_filter_detached,
};

kern_return_t MyInterfaceFilter_start (kmod_info_t* ki, void* d)
{
    ifnet_t interface;

    if (ifnet_find_by_name("en1", &interface) != KERN_SUCCESS) //改为你自己的接口
        return KERN_FAILURE;

    if (iflt_attach(interface, &g_my_iff_filter, &g_filter_ref) == KERN_SUCCESS)
    {
        g_filter_registered = TRUE;
    }

    ifnet_release(interface);

    return KERN_SUCCESS;
}

kern_return_t MyInterfaceFilter_stop (kmod_info_t* ki, void* d)
{
    if (g_filter_registered)
    {
        iflt_detach(g_filter_ref);
        g_filter_registered = FALSE;
    }
    if (!g_filter_detached)
        return KERN_NO_ACCESS; //过滤器卸载之前不允许加载

    return KERN_SUCCESS;
}
```

接口过滤器可以通过iflt_attach()函数装载到网络接口。你可以对多个接口注册一个iff_filter。网络接口由不透明类型ifnet_t表示，可以使用接口KPI（kpi_interface.h）进行操作。在前面的示例中，我们使用接口KPI函数ifnet_find_by_name()获取带有BSD名称en1的网络接口引用，这在MacBook上对应Wi-Fi接口。

iff_input过滤器函数将在接口接收传入分组时调用。该回调有5个参数。

- cookie参数包含注册过滤器时分配给iff_cookie字段的指针。
- ifnet_t参数是接收分组的网络接口引用。这在同一个过滤器函数处理装载到多个网络接口上的过滤器时，特别有用。
- 下一个参数是进入分组所属的protocol族。除非将iff_protocol字段指定为0，否则将一直是你所指定的族。
- mbuf_t表示包含分组数据的缓冲区。
- 最后一个参数frame_ptr是一个指向接口的数据链路帧头的指针。帧头的长度及结构根据网络接口而有所不同。对于以太网接口，帧头由源MAC地址、目标MAC地址和一个16位

ethertype字段组成，该字段确定封装的协议。该字段对于包含IP分组的以太网帧为0x0800。可以调用ifnet_hdrlen (ifnet_t) 函数，来确定接口的帧头长度。

输出过滤器函数iff_output与输入函数类似，但并不提供单独的帧头参数，而mbuf_t包含整个帧，其中包括数据链路头，而不是指向数据链路头之后的数据。如果我们希望检查接口过滤器输出函数中传入分组的IP头，则首先需要解析数据链路头，查找IP头的偏移量。其中一个示例如下所示：

```
static errno_t myif_filter_output(void* cookie, ifnet_t interface, protocol_family_t protocol,
                                  mbuf_t* data)
{
    char                src[64], dst[64];
    unsigned char*      pktbuf = mbuf_data(*data);
    struct ether_header* eth = (struct ether_header *)pktbuf;

    if (ifnet_hdrlen(interface) != ETHER_HDR_LEN)
        return 0;

    if (ntohs(eth->ether_type) == ETHERTYPE_IP)
    {
        struct ip* iphdr = (struct ip*)(pktbuf + ETHER_HDR_LEN);
        inet_ntop(AF_INET, &iphdr->ip_src, src, sizeof(src));
        inet_ntop(AF_INET, &iphdr->ip_dst, dst, sizeof(dst));
        printf("outgoing packet: %lu bytes ip_src: %s ip_dst: %s\n",
               mbuf_pkthdr_len(*data), src, dst);
    }
    else
        printf("outgoing packet: %lu bytes\n", mbuf_pkthdr_len(*data));
    return 0;
}
```

接口过滤器KPI并不提供进出分组的注入函数，而由接口KPI提供。传出分组可以使用ifnet_output_raw()函数注入，或使用ifnet_input()函数注入一个入站分组。inet_output_raw()的使用样例，请参考本章后面讨论的示例驱动程序MyEthernetDriver的源代码。

## 13.2　调试和测试网络扩展

除第16章讨论的通用技术之外，Mac OS X还预装了一些可以调试网络问题的工具，其中最值得注意的可能就是命令行工具tcpdump和netcat。前者使用libpcap库，该库构建在内核网络栈中的BPF（Berkeley Packet Filter，柏克莱封包过滤器）基础结构之上。BPF可以插入每个网络接口，并安装挂钩，可以使进出分组转移到字符设备文件（/dev/bpfX），从而由tcpdump等工具进行分析。tcpdump应用程序可以查看即时分组流量，或将其捕获至一个文件用于之后的分析。许多第三方工具可以处理tcpdump捕获的分组跟踪。如果可能，tcpdump实用工具会将监控的接口置于混杂模式。混杂模式是大多数网络设备的一种固件特性，通知设备发送分组，即使它们并没有定位到其自身的硬件地址。Mac OS X更新的新版本需要root权限运行tcpdump，即使混杂模式已禁用。从繁忙的网络捕获分组会因大量的数据而变得非常困难。为了解决该问题，tcpdump利用BPF的过滤功能，基于各

种规则（从硬件地址到TCP头的单个标志）过滤出分组。以下是tcpdump输出的一个例子：

```
$ sudo tcpdump -i en0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on en0, link-type EN10MB (Ethernet), capture size 65535 bytes
20:43:40.911558 IP 192.168.1.2.ipp > 192.168.255.255.ipp: UDP, length 237
20:43:51.113519 ARP, Request who-has 192.168.1.2 tell 192.168.1.3, length 28
20:43:51.113785 ARP, Reply 192.168.1.2 is-at 00:17:f2:0a:86:60 (oui Unknown), length 46
20:43:51.113831 IP 192.168.1.3 > 192.168.1.2: ICMP echo request, id 64769, seq 0, length 64
20:43:51.114004 IP 192.168.1.2 > 192.168.1.3: ICMP echo reply, id 64769, seq 0, length 64
20:44:11.911836 IP 192.168.1.2.ipp > 192.168.255.255.ipp: UDP, length 237
20:46:01.413453 IP 192.168.1.2.netbios-ns > 192.168.255.255.netbios-ns: NBT UDP PACKET(137):
QUERY; REQUEST; BROADCAST
20:46:01.451950 IP 192.168.1.2.netbios-dgm > 192.168.255.255.netbios-dgm: NBT UDP
PACKET(138)
```

netcat实用程序有多种用途。对于网络调试，可以创建基于TCP和UDP的客户端或服务器，这非常有用，因为可以用来生成任一方向的传输，用于调试。这对IP或套接字过滤器及网络接口驱动程序的开发特别有用。netcat实用程序可以使用nc命令，从终端进行调用。以下示例展示了如何创建套接字，从而在4040端口上监听UDP传输：

```
$ nc -u -l 4040
```

你可以在另外一个系统上使用nc连接到服务器：

```
$ nc -u 192.168.1.2 4040
Stuff typed here will be echoed back to the server
```

## 13.3 I/O Kit 中的网络

IONetworkingFamily类表示内核网络栈的底部。如前面所述，I/O Kit是为基于硬件的网络设备实现驱动程序的首选层。IONetworkingFamily的类层次结构，如图13-4所示。



该族可能看起来非常臃肿，但许多类是辅助类，而且有些类我们根本无需关注，因为它们仅在族内部使用。该族中的关键类是IONetworkController和IONetworkInterface。前者用来表示网络硬件的驱动程序，后者用来与BSD层的数据链路接口层（data-link interface layer，DLIL）交互。该族作为适配器，可以使I/O Kit网络接口被看做BSD层网络接口，这样就可以使用传统的UNIX工具（如ifconfig）来配置设备。我们来看一下各个类的功能。

- IOEthernetController是所有基于以太网设备的基类，包括基于802.11的无线设备。通常在编写以太网驱动程序或基于Wi-Fi的设备驱动程序时，需要子类化该类。
- IOEthernetInterface是IOEthernetController的客户端，用于将控制器与BSD网络层连接起来。如果你要实现一个以太网驱动程序，通常无需子类化IOEthernetInterface，除非有特殊要求。

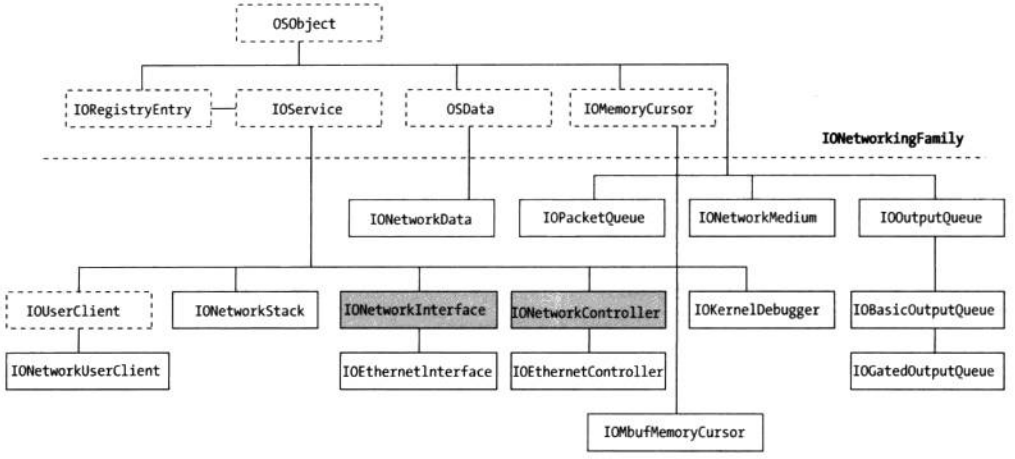



图13-4　IONetworkingFamily类层次结构

- IOKernelDebugger是一个候补驱动程序，如果内核调试器为活动的，则替代IONetwork-Interface用于IONetworkController。如果你在编写一个第三方网络驱动程序，则无需对此进行支持。
- IOMbufMemoryCursor提供了一个面向对象的mbuf结构游标，可以使mbuf集群传输到物理地址，从而进行DMA。可用的特定子类有：IOMBufBigMemoryCursor、IOMbufDBMAMemoryCursor、IOMbufLittleMemoryCursor、IOMbufNaturalMemoryCursor。
- IONetworkController是IOEthernetController的基类。如果你正在编写非以太网兼容设备的驱动程序，则必须子类化IONetworkController。
- IONetworkData表示固定长度的数据缓冲区，由IONetworkInterface用来将接口数据导出到用户空间，特别是使用统计数据，如丢弃分组及冲突的信息。
- IONetworkInterface用于将IONetworkController捆绑至BDIL（BSD data-link layer，BSD数据链路层）及其他网络栈。IONetworkInterface是一个抽象类，如果你的驱动程序基于IONetworkInterface，则必须重新实现它。
- IONetworkUserClient是IOUserClient的一个子类，为IONetworkInterface提供一个用户客户端。
- IOOutputQueue是一个分组队列，处理多个生产者和一个消费者（设备）。可用的特定子类有两个：IOBasicOutputQueue和IOGatedOutputQueue。
- IOPacketQueue实现由自旋锁同步的mbufs的一个FIFO队列。

你可能已经发现，802.11x网络支持并未被提及。苹果没有发布无线网络驱动程序的开发框架。苹果自己的AirPort驱动程序位于IO802Family.kext中，但并没有为此发布源代码或头文件。但这并不妨碍编写无线网络驱动程序，只是你无法利用预先写好的类，必须提供自己的IOUserClient及可能的用户空间工具来配置设备。苹果的AirPort设备是私有IO80211Controller的

子类，它同样也是IOEthernetController的子类。据说所有的现代Mac都有内置无线网络，因此该领域对第三方设备的需求很低。

### 13.3.1 构建简单以太网控制器驱动程序

下面我们将亲自动手，使用I/O Kit网络，构建一个简单的以太网驱动程序。由于实现功能完备的驱动程序非常复杂，而且还要依赖于硬件，很难完全演示。如果某个人要为完全不同的设备实现驱动程序，这个驱动程序可能也没什么用。这部分主要介绍基本原理，熟悉I/O Kit提供的网络驱动程序开发工具。我们将实现一个虚拟以太网驱动程序MyEthernetDriver。该驱动程序将演示如何实现I/O Kit网络驱动程序的核心元素，以及分组流如何通过它与系统的其他部分交互。图13-5展示了MyEthernetDriver与其他I/O Kit类的交互。

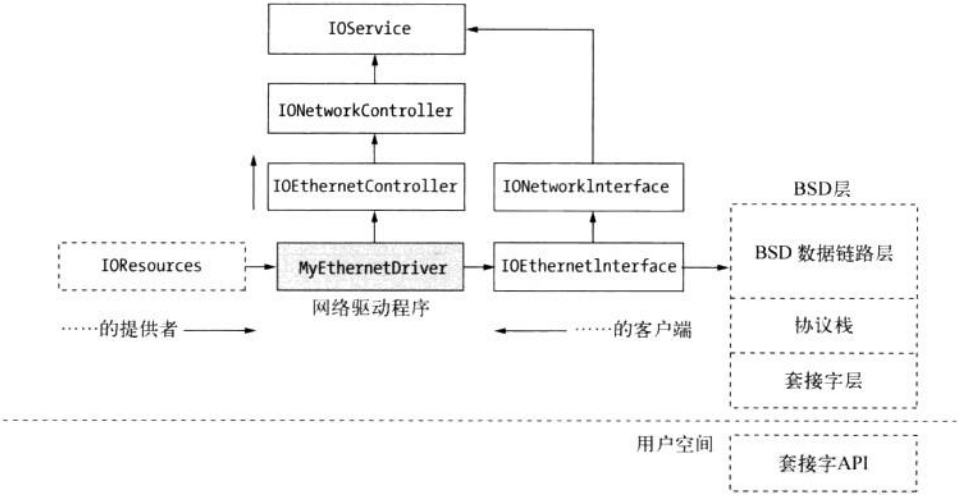



图13-5 MyEthernetDriver:与I/O Kit和网络栈交互

在该示例中，MyEthernetDriver使用IOResources块作为提供者，但对于物理设备支持的实际网络设备，则更有可能使用IOPCIDevice、IOUSBDevice、IOUSBInterface或另一个表示硬件设备的块。

**提示** 如果你想查看实际设备的网络驱动程序，IOUSBFamily源代码发布版包括AppleUSBCDCEthernet的源代码，这是遵循USB以太网规范的设备驱动程序。较新版本中并不包含该驱动程序，但仍然可以在苹果开源网站（opensource.apple.com）上IOUSBFamily的较早版本中找到。此外，还有基于PCI的流行Realtek 8139芯片的驱动程序源代码，称作AppleRTL8139Ethernet。对于直接派生自IONetworkController的网络驱动程序示例，请参见IOFireWireIP，IOFireWireIP通过FireWire实现了TCP/IP网络。

主驱动程序类MyEthernetDriver继承自IOEthernetController，IOEthernetController又继承自IONetworkController。该驱动程序还会分配一个IOEthernetInterface实例，用来与网络栈进行交互。IOEthernetInterface类不是抽象类，可直接分配和使用。

### 13.3.2　MyEthernetDriver设计

为了将MyEthernetDriver设计应用于实际环境中，假设有人请我们为一个新的以太网设备开发驱动程序。该设备是一个适配器，可以连接到Mac的Thunderbolt端口。由于目前这是新兴技术，所以获得所需内容会有一些延迟，因此还无法访问正确的设备。我们希望整天悠闲地上网冲浪就能赚到钱，但我们需要谋生。因此要在没有可用设备的情况下编写该驱动程序。我们的目标是，在没有硬件设备的情况下完成工作。但我们很快意识到，以太网设备的主要部分是处理实际的网络I/O。虽然可以很快构建一个虚拟的设备驱动程序，为其分配IP，并开始与其对话，但还存在一个主要问题。传输到同一个主机上的另一个接口或本身的网络分组，实际上根本无法发送到设备，而是遍历协议栈，但并不涉及I/O Kit驱动程序。I/O Kit专门用于与实际的硬件设备交互，因此，如果需要一个虚拟网络接口，BSD层是最好的位置。

要解决这一问题需要虚拟以太网设备借助于实际的以太网接口，让实际的以太网接口代替我们发送和接收分组。该设置类似于图13-6中的图表。

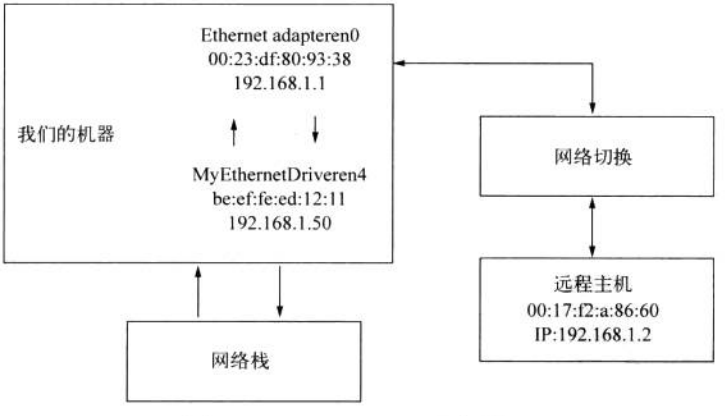



图13-6　MyEthernetDriver测试设置

当MyEthernetDriver从网络栈接收分组时，会对分组进行排队，然后使用装载到物理网络交换机的en0接口在网络上传出分组。如果没有网络交换机，则可以使用cat5线直接连接到远程机器。测试设置中的每个接口都配置为同一个子网上它自己的IP地址。网络栈负责组织分组，因此，在MyEthernetDriver接收分组时，该分组已经装载了以太网帧头，其中目标地址为00:17:f2:0a:86:60，MyEthernetDriver任意选择的源地址为be:ef:fe:ed:12:11。大多数基于以太网的网络设备都可以使用自身以外的（假的）源地址发送分组。因此，如果一切都正确配置，应该就能在远程主

机上接收来自MyEthernetDriver的MAC及IP地址的分组。

获得远程主机对驱动程序的回复存在一些问题，因为en0接口很可能会忽略一个未定位到它的帧。为了解决该问题，只对en0启用混杂模式，可以使en0接收非发向它的分组。然后，在输入队列上为en0安装一个接口过滤器，检查分组是否定位到它，在定位到它的情况下，我们无须管它，而如果定位到MyEthernetDriver，就“盗取”它，将其转移到输入队列。最终结果是一个虚拟以太网桥/交换机。这在概念上接近虚拟机软件（如Parallels或VMWare Fusion）使虚拟机客户的操作系统在网络上进入桥接模式。因为我们已经在以太网层完成了该工作，所以这些改变对高层协议（如IP）完全透明，甚至可以使用外部DHCP服务器获取虚拟接口的IP地址。

现在总体上应该可以处理设备驱动程序了，不管怎样，它能发送和接收实际的网络传输，并提供了一个非常接近实际设备的近似物进行操作。我们将把代码放置在单独的类MyEthernetHwAbstraction中，尽可能地隐藏从驱动程序桥接的情况。当主驱动程序与I/O Kit交互时，MyEthernetHwAbstraction类将处理与“硬件”的通信。这种设计可以快速换出硬件抽象类，而使用与实际硬件对话的类。此外，它还可以子类化抽象类，以便完美地支持硬件的新变体。

**注意** MyEthernetDriver需要借助一个固定的以太网设备。这是因为非固定的设备通常无法使用自身以外的源地址传输分组。这是设备固件的一个限制。

MyEthernetDriver的头文件，如代码清单13-6所示。

**代码清单13-6 MyEthernetDriver头文件**

```
#ifndef MyEthernetDriver_h
#define MyEthernetDriver_h

#include <IOKit/assert.h>
#include <IOKit/IOTimerEventSource.h>
#include <IOKit/IOBufferMemoryDescriptor.h>
#include <IOKit/network/IOEthernetController.h>
#include <IOKit/network/IOEthernetInterface.h>
#include <IOKit/network/IOGatedOutputQueue.h>
#include <IOKit/network/IOMbufMemoryCursor.h>
#include <IOKit/network/IONetworkMedium.h>
#include <IOKit/IOUserClient.h>

#include "MyEthernetHwAbstraction.h"

class com_osxkernel_MyEthernetDriver : public IOEthernetController
{
    friend class com_osxkernel_MyEthernetHwAbstraction;

    OSDeclareDefaultStructors(com_osxkernel_MyEthernetDriver);
public:
    virtual bool init(OSDictionary* properties);
    virtual bool start(IOService* provider);
    virtual void stop(IOService* provider);
    virtual void free();
```

```
    virtual bool configureInterface(IONetworkInterface* netif);

    virtual IOReturn enable(IONetworkInterface* netif);
    virtual IOReturn disable(IONetworkInterface* netif);

    virtual IOReturn getHardwareAddress(IOEthernetAddress* addrP);

    //可以设置驱动程序的Mac地址
    virtual IOReturn setHardwareAddress(const IOEthernetAddress* addrP);

    virtual UInt32 outputPacket(mbuf_t m, void* param);

    virtual IOReturn setPromiscuousMode(bool active) { return kIOReturnSuccess; }
    virtual IOReturn setMulticastMode(bool active) { return kIOReturnSuccess; }

    bool createMediumDict();

private:

    static void  interruptOccured(OSObject* owner, IOTimerEventSource* sender);

    IOTimerEventSource*    fInterruptSource;  //模拟HW rx中断
    IOEthernetInterface*   fNetworkInterface;
    OSDictionary*          fMediumDict;
    IOWorkLoop*            fWorkLoop;

    IONetworkStats*        fNetworkStats;
    IOEthernetStats*       fEthernetStats;

    com_osxkernel_MyEthernetHwAbstraction* fHWAbstraction; //底层硬件访问
};

#endif
```

### 13.3.3　驱动程序初始化和启动

网络驱动程序遵循一般的IOService生命周期。驱动程序及设备初始化位于驱动程序的start()方法中。典型的设备会执行以下步骤。

(1) 配置设备的提供者，枚举任何需要的资源。对于PCI或Thunderbolt，这是指映射设备内存或I/O区。对于USB设备，则是枚举接口和管道。

(2) 配置设备操作，例如，通过访问设备的注册器或发送控制请求，将其从睡眠状态唤醒。

(3) 提取设备信息（如MAC地址）以及关于设备功能的信息（如支持的媒体和速度）。许多以太网设备支持MII（Media Independent Interface，媒体无关接口）总线，这是以一致的方式访问设备状态、信息和配置的标准，从与物理传输相关的设备部分（通常是指PHY）解耦。MII注册包含链接状态、支持的网络速度、错误报告等信息。1 GB或10 GB以太网设备分别由GMII和XGMII规范支持。MII、GMII和XGMII都是IEEE标准。但这些标准并不会规定DMA引擎的实现方式或设备I/O的进行方式。

(4) 根据需求及底层硬件，按需分配和配置IOInterruptEventSource或IOTimerEventSources。许多网络设备使用一个软件定时器作为看门狗定时器，连续监控设备错误，并在错误发生时重置设备。

(5) 实例化和注册IOEthernetInterface实例，使我们的网络控制器对BSD网络栈及系统的其他部分可见。

MyEthernetDriver的start()方法，如代码清单13-7所示。

**代码清单13-7 MyEthernetDriver的start()方法**

```
bool com_osxkernel_MyEthernetDriver::start(IOService* provider)
{
    if (!super::start(provider))
        return false;

    fHWAbstraction = new com_osxkernel_MyEthernetHwAbstraction();
    if (!fHWAbstraction)
        return false;
    if (!fHWAbstraction->init(this))
        return false;

    if (!createMediumDict())
        return false;

    fWorkLoop = getWorkLoop();
    if (!fWorkLoop)
        return false;
    fWorkLoop->retain();

    if (attachInterface((IONetworkInterface**)&fNetworkInterface) == false)
        return false;

    fNetworkInterface->registerService();

    fInterruptSource = IOTimerEventSource::timerEventSource(this, interruptOccured);
    if (!fInterruptSource)
        return false;

    if (fWorkLoop->addEventSource(fInterruptSource) != kIOReturnSuccess)
        return false;

    IOLog("%s::start() -> success\n", getName());
    return true;
}
```

start()方法中执行的操作完全取决于设备的功能。在该示例中，我们并没有表示硬件设备的提供者，因此无须打开提供者。下一步分配com_osxkernel_MyEthernetHwAbstraction类的一个实例，并对其进行初始化。该类包含与硬件设备交互的代码，如读寄存器和配置I/O传输的方法。但在我们的示例中，它包含可以借助于另一个设备的代码。这一步骤在技术上并不必要，你可以轻易地在主驱动程序中进行所有的编码。然而，这样做是为了将来MyEthernetDevice 2000 Pro可

用时，可以通过继承现有的硬件抽象类，仅处理硬件的不同之处。这可以使主驱动程序保持整洁，让同个驱动程序很容易就能支持多种硬件。稍后将查看硬件抽象类。

在该“硬件”初始化之后，调用createMedium()函数，发布硬件支持的传输标准及传输速度的详细信息。下一节对此进行详细讨论。

下一个调用方法是attachInterface()，返回一个IONetworkInterface类的实例，将我们的驱动程序向内核网络层公开。在我们的示例中，返回的是一个IOEthernetInterface实例。如果由于某些原因，需要子类化IOEthernetInterface，可以覆写IONetworkController::createInterface()，它由attachInterface()在内部调用从而分配覆写的类。在attachInterface()返回之前，它还会调用IONetworkController::configureInterface()，也可以覆写该方法，对接口类执行其他配置。MyEthernetDriver对configureInterface()方法的实现如下所示：

```
bool com_osxkernel_MyEthernetDriver::configureInterface(IONetworkInterface *netif)
{
    IONetworkData* nd;

    if (super::configureInterface(netif) == false)
        return false;

    nd = netif->getNetworkData(kIONetworkStatsKey);
    if (!nd || !(fNetworkStats = (IONetworkStats *)nd->getBuffer()))
        return false;

    nd = netif->getParameter(kIOEthernetStatsKey);
    if (!nd || !(fEthernetStats = (IOEthernetStats*)nd->getBuffer()))
        return false;

    return true;
}
```

该方法包含指向接口的网络统计信息缓冲区的指针，这些缓冲区用来记录接收/传输的分组、冲突等其他事件的记录信息。这些信息供用户空间在一些地方使用，如Activity Monitor中的网络表。

为了将IOEthernetInterface注册到系统，我们需要调用其registerService()方法。

start()返回之前的最后一项操作是创建中断源。我们将使用IOTimerEventSource来模拟中断，而硬件设备很可能使用IOFilterInterruptEventSource或IOInterruptEventSource来响应实际的硬件中断。

大部分网络驱动程序还会使用定时器提供看门狗功能，周期性地监控设备的错误情况，并检查当前链接状态等情况，以便网络系统及用户空间可以获得事件通知，如线缆拔出。许多驱动程序每秒就触发一次看门狗定时器。

### 13.3.4　介质和状态选择

createMedium()方法会创建一个字典，向BSD栈及用户空间发布设备介质功能的细节。大部分现代以太网设备还支持较早的以太网标准和传输速度。例如，Macbook Pro中的以太网设备在

全双工或半双工模式下有或无流控制时，可以支持1000BaseT、100BaseTX和10BaseT/UTP。如果设备和驱动程序对此支持，则介质可以通过UNIX命令行工具ifconfig或System Preferences的Network面板进行控制。大多数设备可以自动检测当前的介质。介质功能由IONetworkMedium类表示。createMedium()类的实现，如代码清单13-8所示。

**代码清单13-8 用于发布所支持以太网介质细节的方法**

```
static struct MediumTable
{
    UInt32      type;
    UInt32      speed;
}

mediumTable[] =
{
    {kIOMediumEthernetNone, 0},
    {kIOMediumEthernetAuto, 0},
    {kIOMediumEthernet10BaseT | kIOMediumOptionFullDuplex,  10},
    {kIOMediumEthernet100BaseTX | kIOMediumOptionFullDuplex, 100},
    {kIOMediumEthernet1000BaseT | kIOMediumOptionFullDuplex, 1000},
};

bool com_osxkernel_MyEthernetDriver::createMediumDict()
{
    IONetworkMedium*  medium;
    UInt32                          i;

    fMediumDict = OSDictionary::withCapacity(sizeof(mediumTable) /
                                             sizeof(struct MediumTable));
    if (fMediumDict == 0)
        return false;

    for (i = 0; i < sizeof(mediumTable) / sizeof(struct MediumTable); i++)
    {
        medium = IONetworkMedium::medium(mediumTable[i].type, mediumTable[i].speed);
        if (medium)
        {
            IONetworkMedium::addMedium(fMediumDict, medium);
            medium->release();
        }
    }

    if (publishMediumDictionary(fMediumDict) != true)
        return false;

    medium = IONetworkMedium::getMediumWithType(fMediumDict, kIOMediumEthernetAuto);
    setSelectedMedium(medium);
    return true;
}
```

该方法构建了一个OSDictionary，其中包含支持的介质。该介质字典必须使用publishMediumDictionary()方法发布，将驱动程序的功能通知给操作系统。如果想要你的驱动程序支持手工选

择介质，则需要覆写IONetworkController::selectMedium()方法。缺省方法将只返回kIOReturnUnsupported。你的驱动程序可以调用setSelectedMedium()，通知系统其介质选择。可以用setLinkStatus()方法来设置介质及链接状态。链接状态标志为：kIONetworkLinkValid或kIONetworkLinkActive，对于以太网设备，它们可以用来表示线缆是否连接，设备是否处于活动状态。

### 13.3.5　配置设备硬件地址

以太网网络使用MAC（Media Access Control，介质访问控制）地址，该地址应该是一个48位全局唯一的地址，用来标识网络控制器。网络上的MAC地址冲突会引起交换机、主机及其他网络设备的混淆。在IEEE分配给每个厂商的范围内分配MAC地址时，MAC地址通常会编写进设备的EEPROM中。我们的驱动程序需要将MAC地址发布给网络栈和用户空间。该地址在用户空间中仅用来帮助唯一标识设备，以提供信息及配置。但网络栈确实需要知道能够恰当格式化传出分组的地址，以及其他协议（如IP）的地址解析协议（ARP/RARP）的地址。网络栈将调用我们驱动程序的getHardwareAddress()函数，获取MAC地址。MyEthernetDriver对此的实现如下所示：

```
IOReturn com_osxkernel_MyEthernetDriver::getHardwareAddress(IOEthernetAddress *addrP)
{
    addrP->bytes[0] = fHWAbstraction->readRegister8(kMyMacAddressRegisterOffset + 0);
    addrP->bytes[1] = fHWAbstraction->readRegister8(kMyMacAddressRegisterOffset + 1);
    addrP->bytes[2] = fHWAbstraction->readRegister8(kMyMacAddressRegisterOffset + 2);
    addrP->bytes[3] = fHWAbstraction->readRegister8(kMyMacAddressRegisterOffset + 3);
    addrP->bytes[4] = fHWAbstraction->readRegister8(kMyMacAddressRegisterOffset + 4);
    addrP->bytes[5] = fHWAbstraction->readRegister8(kMyMacAddressRegisterOffset + 5);

    return kIOReturnSuccess;
}
```

getHardwareAddress()方法是IOEthernetController中唯一强制性的方法（纯虚拟），必须实现。由于我们的MyEthernetDriver并没有有效的MAC地址，所以任意指定一个MAC地址：be:ef:6c:8e:12:11。该实现展示了如何从设备的寄存器获取MAC地址。

如果你的设备支持将MAC地址修改为用户自定义的值，则可以重载setHardwareAddress()方法。该方法会将新的MAC地址写入到设备的寄存器，并在修改成功时，返回kIOReturnSuccess。默认实现将返回kIOReturnUnsupported。

### 13.3.6　启用和禁用设备

尽管启动方法可以充分准备并使设备可运行，但首选方式是调用驱动程序的enable()方法，使设备处于活动状态（也就是说，处于可以接收和传输的状态）。类似地，调用驱动程序的disable()方法时，设备应该尽可能地静止，如有可能，甚至要睡眠。驱动程序应该执行这些操作，因为用户此时可能会关闭设备，在这种情况下，驱动程序不应该再使用资源，这对保证它不再消耗设备的电池或浪费能源同样重要。MyEthernetDriver的enable()方法如下所示：

```
IOReturn com_osxkernel_MyEthernetDriver::enable(IONetworkInterface* netif)
{
    IOMediumType          mediumType = kIOMediumEthernet1000BaseT | kIOMediumOptionFullDuplex;
    IONetworkMedium*      medium;

    medium = IONetworkMedium::getMediumWithType(fMediumDict, mediumType);

    if (!fHWAbstraction->enableHardware())
        return kIOReturnError;

    setLinkStatus(kIONetworkLinkActive | kIONetworkLinkValid, medium, 1000 * 1000000);
    return kIOReturnSuccess;
}
```

当然，其实际实现高度依赖硬件。在我们的示例中，该实现将调用硬件抽象类，该抽象类将附属于“从”网络接口，我们则使用该接口启用分组传输和接收。对于实际设备，该方法很可能会将设备从睡眠中唤醒，然后启用中断。硬件抽象`enableHardware()`方法的实现如下所示：

```
bool    com_osxkernel_MyEthernetHwAbstraction::enableHardware()
{
    bool success = true;

    fRxPacketQueue = IOPacketQueue::withCapacity();
    if (!fRxPacketQueue)
        return false;

    if (ifnet_find_by_name("en0", &interface) != KERN_SUCCESS) //换成你自己的接口
        return false;

    ifnet_set_promiscuous(interface, 1);

    if (iflt_attach(interface, &interfaceFilter, &gFilterReference) != KERN_SUCCESS)
        success = false;

    filterRegistered = true;
    return success;
}
```

该方法将查找设备网络接口en0，这应该是一个网络设备。然后将设备置于混杂模式，这能够确保它能接收传输到`MyEthernetDriver` MAC地址的分组。最后，将一个接口过滤器安装到从接口，截获传入分组。我们将检查每个传入分组，并将定位于我们的分组转移到我们的输入队列：`fRxPacketQueue`，而忽略所有其他分组，并由原始接口处理队列中的分组。

`disable()`方法应该逆向执行启用设备时执行的操作，并将设备置于初始状态。对我们来说，这意味着移除接口过滤器，以便不再获取传入分组：

```
void    com_osxkernel_MyEthernetHwAbstraction::disableHardware()
{
    if (filterRegistered == true)
    {
        iflt_detach(gFilterReference);
        while (filterRegistered);
```

```
            ifnet_set_promiscuous(interface, 0);
            ifnet_release(interface);

            fRxPacketQueue->flush();
            fRxPacketQueue->release();
            fRxPacketQueue = NULL;
        }
    }
```

### 13.3.7　传输网络分组

现在我们已经成功配置并准备好了设备，要开始执行一些实际的I/O。网络I/O在概念上对网络驱动程序来说非常简单。网络栈将处理一些繁重的工作，如格式化分组，并确保分组实际传输到我们的接口。我们的驱动程序只需关注如何将原始字节传输到设备。分组通过IONetworkController::outputPacket()发送给驱动程序，你的驱动程序应该覆写该函数，以便从网络栈接收分组。MyEthernetDriver的outputPacket()方法，如下所示：

```
UInt32 com_osxkernel_MyEthernetDriver::outputPacket(mbuf_t packet, void* param)
{
    IOReturn result = kIOReturnOutputSuccess;
    if (fHWAbstraction->transmitPacketToHardware(packet) != kIOReturnSuccess)
    {
        result = kIOReturnOutputStall;
    }
    return result;
}
```

**注意**　如果一个分组由outputPacket()方法接受，则驱动程序应该释放mbuf_t。但MyEthernetDriver无须这么做，因为它将分组传递给另一个驱动程序，由该驱动程序负责将其释放。

如果分组处理成功，则该实现应该返回kIOReturnOutputSuccess。如果硬件繁忙，此时无法接受另一个分组，可以返回kIOReturnOutputStall，这将在随后再次重试该分组。如果要放弃分组，只需返回kIOReturnOutputDropped。outputPacket()方法不应该阻塞或睡眠。

默认情况下，outputPacket()方法由IONetworkInterface实例对控制器进行调用，除非输出队列通过覆写createOutputQueue()方法手工创建，这会返回一个IOOutputQueue子类。强烈推荐实现一个输出队列（或提供你自己的排队机制）。如果队列不存在，你将无法临时安装队列，而必须在驱动程序的outputPacket()方法中处理该分组，否则它会被放弃。如果硬件在调用outputPacket()时，正忙于传输分组，则处理这种情况的唯一方式是排队分组，直至硬件再次就绪。

如果你确实实现了一个队列并进行了安装，那么该队列必须在再次准备传输分组时，调用IOOutputQueue::start()，重新启动，否则你将接收不到其他分组。

**注意**　强烈推荐创建输出队列，但MyEthernetDriver可以跳过该步骤，因为它直接将分组传输到另一个网络接口，该网络接口会实现它自己的排队。

创建一个队列可以通过覆写createOutputQueue()方法来实现。当设备禁用时，应该对队列调用flush()，移除任何排队的分组。

每当硬件将分组置于线缆上时，常见的网络设备都将发出一个（TX）中断，它还表示传输缓冲区中目前还有更多的空间，或现在可以执行一个新的DMA传输。可以调用输出队列的service()方法，通知队列设备现在已经准备好传输更多的数据。但其副作用是这会调用驱动程序的outputPacket()方法，发送一个新的分组（如果有分组可用）。

上一节transmitPacketToHardware()方法的实现如下所示：

```
IOReturn    com_osxkernel_MyEthernetHwAbstraction::transmitPacketToHardware(mbuf_t packet)
{
    if (ifnet_output_raw(interface, 0, packet) != KERN_SUCCESS)
        return kIOReturnOutputDropped;

    //生成驱动程序的一个中断，通知它分组已经发送
    fRegisterMap.interruptStatusRegister |= kTXInterruptPending;
    fDriver->fInterruptSource->setTimeoutUS(1);

    return kIOReturnSuccess;
}
```

该方法将接收到的分组注入到从设备输出队列中。通过在中断寄存器中设置TX中断标志，模拟硬件中断，然后调用定时器函数，模拟1微秒之后从硬件设备接收到的中断。

到达硬件的分组传输同样也依赖于硬件。基于PCI或Thunderbolt的设备很可能会使用DMA。这种情况下，存在两个选项。

- 第一种是存在物理上预分配的缓冲区，例如，使用kIOMemoryPhysicallyContiguous选项，采用IOBufferMemoryDescriptor分配一个缓冲区， mbuf将被复制到该缓冲区，然后DMA至硬件。因为一个mbuf可能包含多个链接的缓冲区，所以一定，要确保使用mbuf_next()遍历该链接，以便所有的段都可以复制到DMA缓冲区。
- 如果设备支持，第二种选择是，使用IOMbufMemoryCursor变体直接从mbuf创建一个分散/聚集列表，这样会避免执行额外的复制。游标类负责从mbuf生成物理段的列表。IOMbufMemoryCursors子类有许多个，使用哪一个子类取决于设备及其限制。例如，如果使用读取大端序格式地址的设备，就可以使用IOMbufBigMemoryCursor，它能通过withSpecification()工厂方法创建：

```
static IOMbufBigMemoryCursor* withSpecification(UInt32 maxSegmentSize, UInt32 maxNumSegments);
```

maxSegmentSize能用来限制各个分散/聚集列表成员的长度。类似地，maxNumSegments则用于控制列表的长度。

### 13.3.8 接收分组

来自网络的传入分组异步到达，网络驱动程序的功能是，当RX中断发生时，从硬件设备卸下分组，并通过它的IONetworkInterface或IOEthernetInterface将它们发送至网络栈（就以太网

驱动程序而言）。MyEthernetDriver的中断处理程序，如代码清单13-9所示。

**代码清单13-9　MyEthernetDriver中断处理程序的实现**

```
void com_osxkernel_MyEthernetDriver::interruptOccured(OSObject* owner, IOTimerEventSource* sender)
{
    mbuf_t packet;

    com_osxkernel_MyEthernetDriver* me = (com_osxkernel_MyEthernetDriver*)owner;
    com_osxkernel_MyEthernetHwAbstraction* hwAbstraction = me->fHWAbstraction;
    if (!me)
        return;

    UInt32 interruptStatus = hwAbstraction->readRegister32(kMyInterruptStatusRegisterOffset);

    //接收中断挂起，从硬件获得分组
    if (interruptStatus & kRXInterruptPending)
    {
        while ((packet = hwAbstraction->receivePacketFromHardware()))
        {
            me->fNetworkInterface->inputPacket(packet);
            me->fNetworkStats->inputPackets++;
        }
        me->fNetworkInterface->flushInputQueue();
    }

    if (interruptStatus & kTXInterruptPending)
    {
        //分组传输成功
        me->fNetworkStats->outputPackets++;
    }
}
```

中断处理程序为RX和TX中断共享。为了找到发生的中断，我们会读取设备的中断状态寄存器。通常，中断状态寄存器会在读取时清除，这将会告知并消除中断。关于TX中断的小提醒：我们只是记录统计结构中传输的分组，因为没有队列，也无须设置新的事务。

接收到分组时，需要从输入缓冲区进行传输，并传递给装载到网络控制器驱动程序的IONetworkInterface类。通过IONetworkInterface::inputPacket()方法将分组发送到网络栈。该方法接受mbuf_t。为了获得mbuf_t中的数据，可以使用IONetworkController::allocatePacket()预分配缓冲区，用作接传入分组的DMA目标。IOMbufMemoryCursor子类可以用来处理mbuf数据到物理地址的转换。

在前面的示例中，我们连续循环，直至清空传入分组队列。实际硬件设备还可能从一个中断接收多个分组。该流程常常被称为中断合并。中断合并对以1 GB或更高速度运转的现代网络设备是必需的，因为网络帧通常非常小，对每个接收的分组发送硬件中断效率极低。相反，设备可以在其板载内存中对多个分组进行排队，然后发送一个中断。应该避免硬件或驱动程序过度排队，因为这会影响延迟，进而影响某些应用程序，如实时多人游戏或音频/视频会议。调用inputPacket()时，分组在内部由IONetworkInterface置于队列中。准备好以后，可以调用flushInputQueue()操作该队列，将分组传递给BSD数据链路层，由协议处理程序处理。

代码清单13-10展示了从设备检索到分组之后发出假RX中断的方法。

**代码清单13-10 处理来自从设备的传入分组和生成虚拟中断的方法**

```
bool    com_osxkernel_MyEthernetHwAbstraction::handleIncomingPacket(mbuf_t packet,
                                                                    char** frameHdr)
{
    bool passPacketToCaller = true;
    bool copyPacket = false;

    struct ether_header *hdr = (struct ether_header*)*framePtr;
    if (!hdr)
        return false;

    //如果分组定位到我们的Mac地址、广播或组播地址，只接受导向我们的分组

    if (memcmp(&fMacBcastAddress.bytes, &hdr->ether_dhost, ETHER_ADDR_LEN) == 0)
    {
        copyPacket = true;
    }
    else if (memcmp(&fRegisterMap.address, &hdr->ether_dhost, ETHER_ADDR_LEN) == 0)
    {
        passPacketToCaller = false; //属于我们的接口
        copyPacket = true;
    }
    else if (hdr->ether_dhost[0] & 0x01) //组播
    {
        copyPacket = true;
    }

    if (copyPacket)
    {
        mbuf_t newPacket;
        newPacket = fDriver->allocatePacket((UInt32)mbuf_pkthdr_len(packet) + ETHER_HDR_LEN);

        if (newPacket)
        {
            unsigned char* data = (unsigned char*)mbuf_data(newPacket);
            bcopy(*framePtr, data, ETHER_HDR_LEN);
            data += ETHER_HDR_LEN;
            mbuf_copydata(packet, 0, mbuf_pkthdr_len(packet),data);

            IOLog("input packet is %lu bytes long\n", mbuf_pkthdr_len(packet));

            fRxPacketQueue->lockEnqueue(newPacket);
            fRegisterMap.interruptStatusRegister |= kRXInterruptPending;
            //对驱动程序生成一个中断，通知它新的分组
            fDriver->fInterruptSource->setTimeoutUS(1);
        }
    }
    return passPacketToCaller;
}
```

在代码清单13-10中，分组从原始分组进行复制，该原始分组对应被调用从设备上的输入过

滤器，然后使用模拟硬件接收缓冲区的IOPacketQueue对其进行排队。随后先设置状态寄存器中的RX中断挂起标志，然后设置中断计时器函数的超时，为驱动程序生成一个中断。中断处理程序运行时，它将调用receivePacketFromHardware()，仅从锁定队列中获取新的分组：

```
mbuf_t  com_osxkernel_MyEthernetHwAbstraction::receivePacketFromHardware()
{
    if (!fRxPacketQueue)
        return NULL;
    return fRxPacketQueue->lockDequeue();
}
```

### 13.3.9　对MyEthernetDriver进行测试

如果你想测试MyEthernetDriver，最好在单独的网络段上进行，或者你恰好是网络管理员，因为该程序与其他示例不同，它会和你的网络交互。测试之前，应该修改MyEthernetHwAbstraction.cpp，以便它指向你希望用来代替MyEthernetDriver进行传输和接收的以太网设备。

可以使用kextload加MyEthernetDriver。与NKE必须手工加载不同，MyEthernetDriver使用IOResources作为提供者，从而可以在启动时自动加载（如果安装在正确位置）。为了测试该驱动程序，建议不要将其放置在系统扩展目录中，以免发生问题。驱动程序加载时，可以使用IORegisterExplorer来验证它是否存在，如图13-7所示。

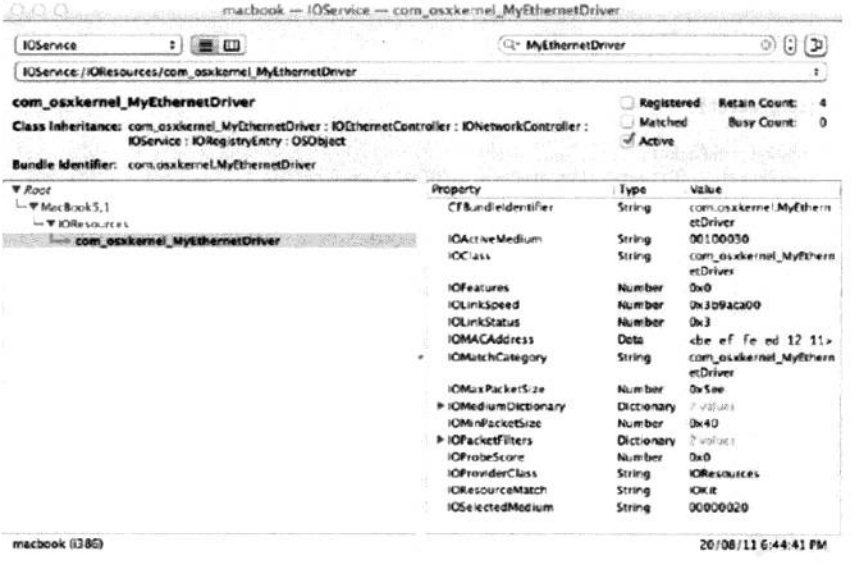



图13-7　IORegisteryExplorer显示了装载到IOResources块的MyEthernetDriver

我们应该也能在System Preferences的Network面板中查看新的网络接口，如图13-8所示。

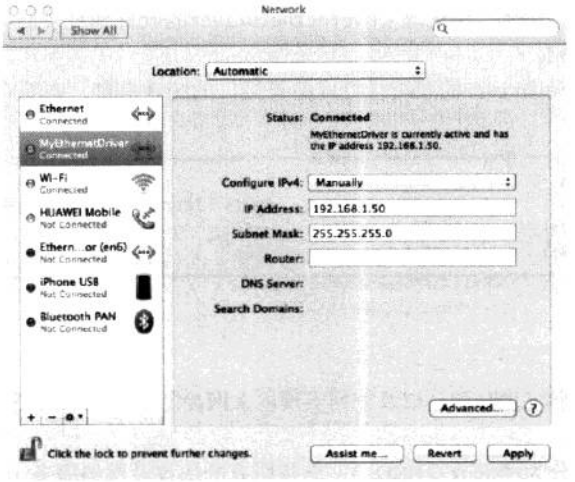



图13-8 System Preferences中的Network面板显示了MyEthernetDriver的配置选项

如果你的网络上有DHCP服务器，则可以看到MyEthernetDriver被自动分配了一个IP地址。如果没有，可以使用System Preferences或ifconfig命令行工具，手动配置一个IP地址。

```
$ sudo ifconfig en5 inet 192.168.1.50 netmask 255.255.255.0
$ ifconfig en5
en5: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        ether be:ef:fe:ed:12:11
        inet6 fe80::bcef:feff:feed:1211%en5 prefixlen 64 scopeid 0x7
        inet 192.168.1.50 netmask 0xffffff00 broadcast 192.168.1.255
        media: autoselect (1000baseT <full-duplex>)
        status: active
```

请注意，你可能分配到了一个不同的BSD网络接口名，这取决于系统上安装了多少个接口。该示例使用的是en5。假定已经配置了一个IP地址，网络上的另一个主机可访问该地址，那么现在应该可以访问该主机，即使从属接口正在使用不同的IP/子网。可以在另一个主机上使用ping实用程序，验证它是否能正常工作。

```
othermac$ ping 192.168.1.50
PING 192.168.1.50 (192.168.1.50): 56 data bytes
64 bytes from 192.168.1.50: icmp_seq=0 ttl=64 time=0.855 ms
64 bytes from 192.168.1.50: icmp_seq=1 ttl=64 time=0.588 ms
--- 192.168.1.50 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.588/0.722/0.855/0.133 ms

othermac$ arp -a
? (192.168.1.50) at be:ef:fe:ed:12:11 on en0 ifscope [ethernet]
? (192.168.255.255) at ff:ff:ff:ff:ff:ff on en0 ifscope [ethernet]
```

你会看到，另一个系统已经获得了MyEthernetDriver的硬件地址（MAC），而并没有使用从属接口的MAC访问我们。

返回安装MyEthernetDriver的系统，可以检查接口的统计信息，查看它已经传输的分组和数据的数量：

```
$ netstat -i -I en5
Name  Mtu   Network       Address            Ipkts  Ierrs    Opkts  Oerrs  Coll
en5   1500  <Link#7>      be:ef:fe:ed:12:11     61      0       67      0     0
en5   1500  192.168.1     192.168.1.50          61      -       67      -     -
```

## 13.4　小结

本章讲述了内核网络过滤KPI，以及如何实现以太网控制器的驱动程序。下面总结了一些关键点。

- 内核网络支持分为两部分：BSD层，支持所有的协议及网络服务（如防火墙）；I/O Kit，用于编写网络硬件驱动程序。
- 内核过滤KPI可以过滤和操作各层的网络分组，包括套接字、IP和接口层。
- 内核网络子系统中最重要的数据接口是mbuf结构，用来存储网络分组或其他相关数据。在内核扩展中，mbufs可以使用不透明引用mbuf_t及mbuf KPI提供的函数进行操作。mbuf概念在内核的BSD区和I/O Kit中都可以使用。
- 套接字过滤器可以截获基于套接字的通信及带外事件。它可以截获进出数据。套接字过滤器可以针对系统中的每个套接字进行全局装载，也可以针对每个套接字以编程方式进行装载。除了以上功能，还可以使用套接字过滤器修改、拒绝或注入新的分组。
- IP过滤器与套接字过滤器类似，但工作在IP层。无论任何接口，IP过滤器都可以查看系统中所有的IP传输。还可以查看通过套接字间接初始化的IP分组。
- 接口过滤器可以使过滤器装载到特定的网络接口上。无论任何协议，该接口都可以查看到达/来自该接口的所有传输。接口过滤器可以将查看的分组限制到特定协议族中，如IPv4或IPv6。
- IPNetworkingFamily提供网络硬件设备驱动程序实现所需的编程接口。包括用于排队的类，以及用于对I/O Kit和BSD层之间的接口进行抽象的类。
- IONetworkController类表示网络驱动程序。IOEthernetController提供处理以太网兼容设备的特定类。IONetworkInterface用于将网络设备连接到内核网络系统的其他部分。

# 第14章 存储系统

存储设备包含多种类型的设备，包括硬盘驱动器、CD和DVD、USB闪存、基于FireWire的硬盘以及基于文件的磁盘映像（装载为虚拟驱动器）。对用户来说，存储设备以卷的形式出现在桌面上，用户可以从中读取或向其中写入文件，但用户看不到的是，内核中多个驱动程序相互配合才会使这成为可能。

存储设备需要多个驱动程序的原因是，一种存储设备可能有多种不同的格式。如果你考虑一下外部USB闪存驱动器与内部硬盘之间的不同（两者对用户都表现为存储设备），那么你就会意识到所需处理的不同之处。例如：

- 存储设备可能连接到计算机的USB接口或FireWire端口（用于外部存储设备），或者SATA端口（通常用于内部硬盘）；
- 计算机通过发送SCSI命令控制存储设备，USB大容量存储设备或FireWire SPB-2驱动器就是如此，或者通过向AHCI接口发送ATA/ATAPI命令控制存储设备，内部连接的SATA磁盘就是如此；
- 存储设备可能只包含一个卷，也可能划分为多个卷；
- 每个卷将格式化为用户选择的文件系统，可以是HFS+（Mac OS X使用的默认文件系统）、NTFS（Windows使用的默认文件系统），或Mac OS X所支持众多文件系统中的一个。

为了处理所有这些可能的变化，I/O Kit构建一个多驱动程序的分层栈实现存储设备，每层负责一个方面，如物理连接（USB、FireWire或SATA）、命令协议（SCSI或ATA/ATAPI）及逻辑卷。VFS层（Virtual File System Layer，虚拟文件系统层）提供对各种文件系统的支持，尽管VFS层是内核的一部分，但它位于内核的BSD部分，在I/O Kit的外面。



I/O Kit存储栈的模块化设计意味着，驱动程序栈的每层与周围层解耦，并且每个驱动程序只需处理本层提供的功能。这意味着，即使不知道某文件系统所在存储设备类型的任何知识，也可以编写一个新的文件系统，因为文件系统无须与磁盘设备的硬件直接通信，同样，在编写新类型存储设备的驱动程序时，开发者无须实现文件系统的细节；而磁盘可以格式化为Mac OS X支持的现有文件系统。存储设备的驱动程序栈，如图14-1所示。

无须在每层实现一个驱动程序。例如，为了支持新硬盘，只需在传输层中编写一个驱动程序，驱动程序栈的其他部分将会匹配你自定义的驱动程序，对用户来说，新磁盘设备将作为一个标准磁盘存在。

图14-1　驱动程序在支持存储设备方面的抽象视图

## 14.1　传输层驱动程序

在I/O Kit中，存储设备驱动程序实现为一个传输驱动程序，位于图14-1中的传输层。类似于任何其他的I/O Kit驱动程序，传输驱动程序将匹配表示其硬件设备的提供者类，该提供者类也是驱动程序访问底层硬件的方式。例如，基于USB存储设备的提供者类为IOUSBDevice或IOUSBInterface的实例。

除了最终要派生自IOService类之外（所有的I/O Kit驱动程序都是如此），I/O Kit对传输驱动程序可以派生的超类并没有限制。这赋予了传输驱动程序极大的自由，因为它可以使用许多本来用于磁盘设备通信协议的方法，而不是强制实现一个I/O Kit存储族规定的接口。

缺少传输驱动程序的公共接口确实会产生问题，因为驱动程序栈的上层，特别是传输驱动程序正上方的通用块存储驱动程序，没有可以用来调用传输驱动程序中方法的公共接口。为了解决该问题，I/O Kit定义了一个驱动程序类，即IOBlockStorageDevice，这是位于传输层和通用块设备驱动程序之间的一个小型轻量级“核心”驱动程序。

IOBlockStorageDevice类的作用是，提供磁盘对通用块存储驱动程序的抽象表示，并将所有的请求传递给传输驱动程序，驱动程序进而实现特定于磁盘设备的行为。传输驱动程序负责定义IOBlockStorageDevice类的实际子类，并将其实例化。传输驱动程序与存储驱动程序栈中它的上下两层间的关系，如图14-2所示，该图使用苹果AHCI驱动程序作为示例。

---

**注意**　由于I/O Kit对传输驱动程序的实现并没有限制，所以传输驱动程序可以基于包含多个驱动程序的一个栈进行构建。苹果在使用SCSI命令集的设备驱动程序中便使用该方式，包括USB大容量存储设备和FireWire设备。这些设备的驱动程序都使用实现SCSI协议的公共驱动程序。

---

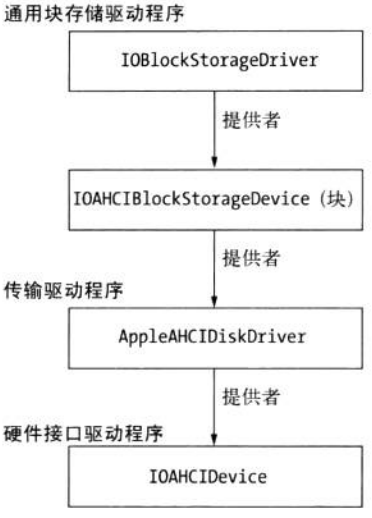



图14-2 苹果AHCI存储驱动程序栈中驱动程序之间的关系

## 14.2 IOBlockStorageDevice 接口

存储驱动程序栈的上层，通过一个派生自IOBlockStorageDevice接口的类与传输驱动程序进行通信。IOBlockStorageDevice将存储设备看做逻辑块的一维数组，由调用者读取或写入。逻辑块是磁盘可以读/写的最小字节数，一项磁盘操作会在多个块上进行。根据磁盘不同，块长度的范围是512字节～4096字节。IOBlockStorageDevice类执行的所有操作都针对范围连续的磁盘块。

IOBlockStorageDevice接口的子类必须实现的方法，如代码清单14-1所示。IOBlockStorage－Device类并不提供这些方法行为的完整实现，而是将这些方法传递给其提供者类，即存储驱动程序栈中的传输驱动程序。

**代码清单14-1 由IOBlockStorageDevice接口子类实现的方法**

```
class IOBlockStorageDevice : public IOService
{
        virtual bool            init(OSDictionary * properties);
        virtual IOReturn        doEjectMedia(void) = 0;
        virtual IOReturn        doFormatMedia(UInt64 byteCapacity) = 0;
        virtual UInt32          doGetFormatCapacities(UInt64 * capacities,
                                    UInt32   capacitiesMaxCount) const = 0;
        virtual IOReturn        doLockUnlockMedia(bool doLock) = 0;
        virtual IOReturn        doSynchronizeCache(void) = 0;
        virtual char*           getVendorString(void) = 0;
        virtual char*           getProductString(void) = 0;
        virtual char*           getRevisionString(void) = 0;
        virtual char*           getAdditionalDeviceInfoString(void) = 0;
```

```
    virtual IOReturn        reportBlockSize(UInt64 *blockSize) = 0;
    virtual IOReturn        reportEjectability(bool *isEjectable) = 0;
    virtual IOReturn        reportLockability(bool *isLockable) = 0;
    virtual IOReturn        reportMaxValidBlock(UInt64 *maxBlock) = 0;
    virtual IOReturn        reportMediaState(bool *mediaPresent,bool *changedState) = 0;
    virtual IOReturn        reportPollRequirements(bool *pollRequired,
                                bool *pollIsExpensive) = 0;
    virtual IOReturn        reportRemovability(bool *isRemovable) = 0;
    virtual IOReturn        reportWriteProtection(bool *isWriteProtected) = 0;
    virtual IOReturn        getWriteCacheState(bool *enabled) = 0;
    virtual IOReturn        setWriteCacheState(bool enabled) = 0;
    virtual IOReturn        doAsyncReadWrite(IOMemoryDescriptor *buffer, UInt64 block,
                                UInt64 nblks, IOStorageAttributes *attributes,
                                IOStorageCompletion *completion) = 0;
    virtual IOReturn        requestIdle(void);
    virtual IOReturn        doDiscard(UInt64 block, UInt64 nblks);
    virtual IOReturn        doUnmap(IOBlockStorageDeviceExtent* extents,
                                UInt32 extentsCount, UInt32 options);
};
```

接下来的各节将讲述IOBlockStorageDevice子类需要实现的各个方法，本书是按功能排序进行讲述。各个描述的首要方法会返回一个用户易读的设备描述。这些字符串用来帮助用户确定装载卷所对应的存储设备。如果存储介质没有格式化，则不存在与该设备关联的卷，用户只能通过这些标识字符串来确定某一设备是否是他要进行格式化的设备。因此，这些字符串应该返回一个描述性名称，例如，标识USB闪存的厂商，或提供连接接口的一个描述（如“SATA适配器的USB”），可以使用户很容易地识别设备。这些字符串出现在某些实用程序中，如“Disk Utility”及“System Information”生成的系统性能剖析。

- getVendorString返回存储设备厂商名称。
- getProductString返回产品模型的描述性名称。
- getRevisionString返回的字符串解释可由驱动程序开发者决定。这可以用来标识运行在存储设备上的固件版本或产品设计。也可以同时包含这两个值，因为该值是一个字符串。
- getAdditionalDeviceInfoString目前在I/O Kit实现中并未使用，但可以由专有磁盘实用程序软件从驱动程序进行查询。

查询存储设备的功能，可以调用下列方法。

- reportRemovability和reportEjectability都返回类似的信息。如果媒介在驱动程序存在时可以放入和取出，则认为设备可移除。这意味着，I/O Kit可以周期性地轮询传输驱动程序，确定磁盘当前是否存在。另外，可移动的设备如果可以通过软件控制进行移除，则被认为可弹出（如CD驱动器）。如果一个设备可弹出，用户仍然可以通过Finder或Disk Utility进行“弹出”，尽管Mac OS X会卸载该文件系统，但并不会弹出该媒介。
- 调用reportLockability，确定是否可以“锁定”可移除的驱动器，防止用户移除。一个锁定设备的例子是CD驱动器，它前面有一个弹出按钮，可以在CD取出时禁用（锁住）。
- 调用reportPollRequirements，确定驱动程序是否需要周期性地调用，以检查媒介是否已经插入或移除，而不是在媒介插入时，驱动程序自身生成一个通知。如果设备需要轮询，

则驱动程序可以通过reportPollRequirements方法，返回一个额外的标志，表示轮询代价是否很高，例如，仅通过提高设备的旋转速度便可以检测到媒介时。I/O Kit仅在代价不高时才轮询设备。

- 调用reportMediaState，确定设备中是否存在媒介。该方法在存储设备栈创建时，调用一次，用于读取硬件的初始状态，而在此之后，仅在驱动程序表示它需要轮询以确定媒介是否存在时才会调用。

调用下列方法可以查询现有媒介的功能。每当检测到新媒介时，便会调用这些方法。

- reportBlockSize返回设备磁盘扇区（或块）的字节长度。用户空间进程可以通过ioctl DKIOCGETBLOCKSIZE访问该值。
- reportMaxValidBlock返回设备的容量，表示为设备的最终块地址。因为磁盘块的索引从0开始，所以最大有效块将比设备的总块数小1。
- reportWriteProtection用于确定媒介是可以写入，还是写保护，在写保护的情况下，媒介将被装载为只读卷。用户空间进程可以通过ioctl DKIOCISWRITABLE访问该值。

下列方法会执行媒介的底层格式化。并不是所有的设备都支持底层格式化。即使这些方法在IOBlockStorageDevice接口的实现中必须存在，但当该功能未提供时，可以返回一个错误。

- 调用doGetFormatCapacities，获得一个列表，其中包含媒介可以格式化的各种长度（单位是字节）。调用者提供对该方法的结果存储，该方法的返回值为写入到该列表中的实际项目数。当调用者希望确认该实现支持的格式数时，可以为列表存储提供一个NULL指针，而不会接收实际列表。用户空间进程可以通过ioctl DKIOCGETFORMATCAPACITIES请求该列表。
- 调用doFormatMedia，执行设备的底层格式化。如果没有实现该功能，该方法将自动返回一个错误，如kIOReturnUnsupported。用户空间进程可以发送ioctl DKIOCFORMAT执行该操作。
- 调用doDiscard方法，则不会格式化整个磁盘，只是清除不再存储文件系统所需数据的块。对于固定磁盘，该方法会对丢弃的块发出TRIM命令。用户空间进程可以发送ioctl DKIOCDISCARD执行该操作。该方法在Mac OS X 10.6之后的版本中已经弃用，用doUnmap方法代替。
- doUnmap方法是doDiscard方法的替代者。执行类似功能，释放文件系统不再使用的磁盘块。doDicard方法只能释放连续运行的单个磁盘块，而doUnmap方法是一个数组，包含一个或多个不再使用的磁盘块，用户空间进程可以发送ioctl DKIOCUNMAP执行该操作。

下列方法可以通过弹出媒介，进行软件控制。

- 调用doLockUnlockMedia，防止用户弹出媒介，如禁用CD驱动器前面的弹出按钮。该方法有一个传入的布尔型参数，确定驱动程序是锁住设备中的媒介（防止用户弹出），还是解锁该媒介（允许用户弹出）。
- 调用doEjectMedia，从设备弹出媒介。用户空间进程可以发送ioctl DKIOCEJECT执行该操作。
- 调用requestIdle，将磁盘置于空闲状态，如降低CD驱动器的旋转速度。当没有相应的方

法让设备离开空闲状态时，下一个读/写操作将隐式实现。用户空间进程可以通过发送ioctl DKIOCREQUESTIDLE执行该操作。

最后，可能也最重要的是下列将数据写入设备或从设备读取数据的方法。

- doAsyncReadWrite是IOBlockStorageDevice接口的通用数据读写方法。其参数有：IOMemoryDescriptor对象——描述元缓冲区（用于磁盘读取）或目标缓冲区（用于磁盘写入）；用于读写的一个连续范围的读写磁盘块以及相关的属性标志。IOMemoryDescriptor还负责确定请求的操作是读取还是写入，驱动程序对IOMemoryDescriptor对象调用getDirection()方法，如果返回值为kIODirectionIn，则表示请求读取操作。如果返回值为kIODirectionOut，则表示请求写入操作。磁盘操作为异步执行，操作完成时，通过回调函数通知调用者。
- 调用setWriteCacheState，启用或禁用设备的硬件缓存。对应的getWriteCacheState方法返回设备缓存的当前状态。
- 调用doSynchronizeCache，将硬件缓存的内容输出到媒介。这是一个同步方法，它直至缓存内容写入磁盘才返回。该方法还可以响应用户空间进程的ioctl DKIOCSYNCHRONIZECACHE。

## 14.3　构建RAM磁盘设备

查看了为支持IOBlockStorageDevice接口而必须实现的一些方法之后，我们现在来看一下，如何在Mac OS X中实现一个简单的RAM磁盘设备。类似于I/O Kit中的磁盘设备驱动程序，我们将驱动程序划分为两类：传输驱动程序类，实现硬件设备的功能和通信，以及实现IOBlockStorageDevice接口的类，作为传输驱动程序和存储驱动程序栈设备服务层之间的接口。

我们已经看到，I/O Kit并不要求存储设备的传输驱动程序采用任何特定的方式进行编写，或从任何特定的超类继承。对提供磁盘存储访问的那类硬件，这能使传输驱动程序采用对这类硬件来说最自然的方式进行编写。而对于我们的RAM磁盘，传输驱动程序控制的“硬件”只是为RAM磁盘提供存储的内存分配。从第4章已经了解到，没有硬件设备匹配的I/O Kit驱动程序将使用全局的IOResources类作为其提供者类。这将成为我们RAM磁盘传输驱动程序的提供者类。由于我们的传输驱动程序将实现为一个通用驱动程序，所以将其实现为通用IOService类的子类。

为了简单起见，RAM磁盘的传输驱动程序将在加载时为磁盘分配存储空间，并且在驱动程序卸载之前不会将其释放。该存储空间是一个大小固定的内存存储。传输驱动程序还将负责实例化IOBlockStorageDevice对象，传输驱动程序提供一个接口，驱动程序栈的上层将通过该接口进行通信。RAM磁盘传输驱动程序的头文件，如代码清单14-2所示。

**代码清单14-2　RAM磁盘设备传输驱动程序的头文件**

```
#include <IOKit/IOService.h>
#include <IOKit/IOBufferMemoryDescriptor.h>

class com_osxkernel_driver_RAMDisk : public IOService
{
```

```
        OSDeclareDefaultStructors(com_osxkernel_driver_RAMDisk)

private:
        IOBufferMemoryDescriptor*       m_memoryDesc;
        void*                           m_buffer;

protected:
     bool                    createBlockStorageDevice ();

public:
     virtual bool            start (IOService* provider);
     virtual void            free (void);

     virtual IOByteCount     performRead (IOMemoryDescriptor* dstDesc, UInt64 byteOffset,
                                          UInt64 byteCount);
     virtual IOByteCount     performWrite (IOMemoryDescriptor* srcDesc, UInt64 byteOffset,
                                          UInt64 byteCount);
};
```

RAM磁盘传输驱动程序的实现，如代码清单14-3所示。

**代码清单14-3 RAM磁盘设备传输驱动程序的实现**

```
//定义超类
#define super IOService

OSDefineMetaClassAndStructors(com_osxkernel_driver_RAMDisk, IOService)

#define kDiskByteSize           (16*1024*1024) //将RAM磁盘固定为16 Mib

bool com_osxkernel_driver_RAMDisk::start (IOService *provider)
{
        if (super::start(provider) == false)
                return false;

        //为磁盘分配存储空间
        m_memoryDesc = IOBufferMemoryDescriptor::withCapacity(kDiskByteSize,
                                                              kIODirectionOutIn);
        if (m_memoryDesc == NULL)
                return false;
        m_buffer = m_memoryDesc->getBytesNoCopy();

        //分配一个IOBlockStorageDevice块
        if (createBlockStorageDevice() == false)
                return false;

        return true;
}

void com_osxkernel_driver_RAMDisk::free (void)
{
        if (m_memoryDesc != NULL)
                m_memoryDesc->release();

        super::free();
```

```
}


IOByteCount com_osxkernel_driver_RAMDisk::performRead (IOMemoryDescriptor* dstDesc,
                                                       UInt64 byteOffset, UInt64 byteCount)
{
        return dstDesc->writeBytes(0, (void*)((uintptr_t)m_buffer + byteOffset), byteCount);
}

IOByteCount com_osxkernel_driver_RAMDisk::performWrite (IOMemoryDescriptor* srcDesc,
                                                        UInt64 byteOffset, UInt64 byteCount)
{
        return srcDesc->readBytes(0, (void*)((uintptr_t)m_buffer + byteOffset), byteCount);
}
```

RAMDisk类的实现非常简单。在其start()方法中，传输驱动程序分配一个内存缓冲区，为磁盘设备提供存储空间。该缓冲区在RAM磁盘驱动程序被卸载之前不会释放，并且会调用其free()方法。RAMDisk驱动程序类还定义了两个方法，用来访问存储缓冲区，即performRead()和performWrite()。

通常，传输驱动程序的实现应该与其控制设备的功能及协议相匹配。代码清单14-2中RAM磁盘的接口能够使用一些非常简单的方法来满足该要求。这种自由性的结果是，传输驱动程序需要一个块驱动程序，实现IOBlockStorageDevice接口，以接受来自存储驱动程序栈上层的调用，并将这些调用传递给传输驱动程序。在我们的示例RAM磁盘驱动程序中，该功能由com_osxkernel_driver_RAMDiskStorageDevice类提供，该类派生自IOBlockStorageDevice接口。

实现IOBlockStorageDevice接口的类位于传输驱动程序和上层驱动程序之间，其实现方法由上层驱动程序调用，上层驱动程序又调用传输驱动程序中实现的方法。因而，它需要传输驱动程序类一个实例的引用。这通常通过将传输驱动程序作为IOBlockStorageDevice块的提供者类来实现。

在我们的RAM磁盘驱动程序中，传输驱动程序直接实例化RAMDiskStorageDevice块，并将其装载到自身。将RAMDiskStorageDevice装载到传输驱动程序上，从而将传输驱动程序设置为RAMDiskStorageDevice的提供者类。这一过程由私有方法createBlockStorageDevice()执行，传输驱动程序将从其start()方法中调用该方法。其实现如代码清单14-4所示。

**代码清单14-4　实例化RAM磁盘传输驱动程序的IOBlockStorageDevice块**

```
bool com_osxkernel_driver_RAMDisk::createBlockStorageDevice ()
{
        com_osxkernel_driver_RAMDiskStorageDevice*      nub = NULL;
        bool            result = false;

        //分配一个新的IOBlockStorageDevice块
        nub = new com_osxkernel_driver_RAMDiskStorageDevice;
        if (nub == NULL)
                goto bail;

        //调用自定义的初始方法（传递整体磁盘大小）
        if (nub->init(kDiskByteSize) == false)
                goto bail;
```

```
        //将IOBlockStorageDevice装载到该驱动程序
        //该调用将块对象的引用计数加1
        //因此我们会在函数退出时释放其引用
        if (nub->attach(this) == false)
                goto bail;

        //允许上层驱动程序匹配IOBlockStorageDevice
        nub->registerService();

        result = true;

bail:
        //无条件释放块对象
        if (nub != NULL)
                nub->release();

        return result;
}
```

实例化块驱动程序，并将其装载为传输驱动程序的客户端之后，重要的一点是，调用块的registerService()方法。这可以通过实现块本身（如在其start()方法中），或如该例所示，由传输驱动程序执行。调用registerService()，发布IOBlockStorageDevice块，可以使驱动程序与其进行匹配，开始构造存储驱动程序栈的其余部分。com_osxkernel_driver_RAMDiskStorageDevice块驱动程序的头文件，如代码清单14-5所示。

**代码清单14-5 RAMDiskStorageDevice块类的头文件**

```
#include <IOKit/storage/IOBlockStorageDevice.h>

class com_osxkernel_driver_RAMDisk;

class com_osxkernel_driver_RAMDiskStorageDevice : public IOBlockStorageDevice
{
        OSDeclareDefaultStructors(com_osxkernel_driver_RAMDiskStorageDevice)

private:
        UInt64                          m_blockCount;
        com_osxkernel_driver_RAMDisk*   m_provider;

public:
        virtual bool    init(UInt64 diskSize, OSDictionary* properties = 0);

        virtual bool    attach(IOService* provider);
        virtual void            detach(IOService* provider);

        virtual IOReturn        doEjectMedia(void);
        virtual IOReturn        doFormatMedia(UInt64 byteCapacity);
        virtual UInt32          doGetFormatCapacities(UInt64 * capacities, UInt32
                                    capacitiesMaxCount) const;
        virtual IOReturn        doLockUnlockMedia(bool doLock);
        virtual IOReturn        doSynchronizeCache(void);
        virtual char*           getVendorString(void);
        virtual char*           getProductString(void);
```

```
        virtual char*           getRevisionString(void);
        virtual char*           getAdditionalDeviceInfoString(void);
        virtual IOReturn        reportBlockSize(UInt64 *blockSize);
        virtual IOReturn        reportEjectability(bool *isEjectable);
        virtual IOReturn        reportLockability(bool *isLockable);
        virtual IOReturn        reportMaxValidBlock(UInt64 *maxBlock);
        virtual IOReturn        reportMediaState(bool *mediaPresent,bool *changedState);
        virtual IOReturn        reportPollRequirements(bool *pollRequired,
                                    bool *pollIsExpensive);
        virtual IOReturn        reportRemovability(bool *isRemovable);
        virtual IOReturn        reportWriteProtection(bool *isWriteProtected);
        virtual IOReturn        getWriteCacheState(bool *enabled);
        virtual IOReturn        setWriteCacheState(bool enabled);
        virtual IOReturn        doAsyncReadWrite(IOMemoryDescriptor *buffer, UInt64 block,
                                    UInt64 nblks, IOStorageAttributes *attributes,
                                    IOStorageCompletion *completion);
};
```

com_osxkernel_driver_RAMDiskStorageDevice类的实现，如代码清单14-6所示。为了简洁，我们省去了空实现方法。

**代码清单14-6　IOBlockStorageDevice块类的实现**

```
#include <IOKit/storage/IOBlockStorageDevice.h>

//定义超类
#define super IOBlockStorageDevice

OSDefineMetaClassAndStructors(com_osxkernel_driver_RAMDiskStorageDevice, IOBlockStorageDevice)

#define kDiskBlockSize          512

bool com_osxkernel_driver_RAMDiskStorageDevice::init(UInt64 diskSize, OSDictionary*
   properties)
{
    if (super::init(properties) == false)
      return false;
    m_blockCount = diskSize / kDiskBlockSize;
    return true;
}

bool com_osxkernel_driver_RAMDiskStorageDevice::attach (IOService* provider)
{
    if (super::attach(provider) == false)
      return false;
    m_provider = OSDynamicCast(com_osxkernel_driver_RAMDisk, provider);
    if (m_provider == NULL)
      return false;
    return true;
}

void com_osxkernel_driver_RAMDiskStorageDevice::detach(IOService* provider)
{
    if (m_provider == provider)
      m_provider = NULL;
```

```
    super::detach(provider);
}


UInt32 com_osxkernel_driver_RAMDiskStorageDevice::doGetFormatCapacities(UInt64* capacities,
                                                                        UInt32 capacitiesMaxCount) const
{
    //确保该数组能够保存所有的格式（我们需要1个成员）
    if ((capacities != NULL) && (capacitiesMaxCount < 1))
      return 0;                //错误，返回数组长度0

    //如果调用者希望查询我们支持的格式数，则提供一个NULL数组
    if (capacities != NULL)
      capacities[0] = m_blockCount * kDiskBlockSize;
    return 1;
}

char* com_osxkernel_driver_RAMDiskStorageDevice::getProductString(void)
{
    return (char*)"RAM Disk";
}

IOReturn com_osxkernel_driver_RAMDiskStorageDevice::reportBlockSize(UInt64 *blockSize)
{
    *blockSize = kDiskBlockSize;
    return kIOReturnSuccess;
}

IOReturn com_osxkernel_driver_RAMDiskStorageDevice::reportMaxValidBlock(UInt64 *maxBlock)
{
    *maxBlock = m_blockCount-1;
    return kIOReturnSuccess;
}

IOReturn com_osxkernel_driver_RAMDiskStorageDevice::reportMediaState(bool *mediaPresent, bool
     *changedState)
{
    *mediaPresent = true;
    *changedState = false;
    return kIOReturnSuccess;
}

IOReturn com_osxkernel_driver_RAMDiskStorageDevice::reportPollRequirements(bool *pollRequired,
   bool *pollIsExpensive)
{
    *pollRequired = false;
    *pollIsExpensive = false;
    return kIOReturnSuccess;
}

IOReturn com_osxkernel_driver_RAMDiskStorageDevice::reportRemovability(bool *isRemovable)
{
    *isRemovable = true;
    return kIOReturnSuccess;
}
```

```
IOReturn com_osxkernel_driver_RAMDiskStorageDevice::doAsyncReadWrite(IOMemoryDescriptor
   *buffer, UInt64 block, UInt64 nblks, IOStorageAttributes *attributes, IOStorageCompletion
   *completion)
{
    IODirection               direction;
    IOByteCount               actualByteCount;

    //如果我们已被终止，则对传入I/O返回错误
    if (isInactive() == true)
      return kIOReturnNotAttached;
    //确保目标块范围不超过磁盘容量
    if ((block + nblks) > m_blockCount)
      return kIOReturnBadArgument;

    //获取缓冲区的方向，确定操作是读取还是写入
    direction = buffer->getDirection();
    if ((direction != kIODirectionIn) && (direction != kIODirectionOut))
      return kIOReturnBadArgument;

    //通过传输驱动程序，执行读/写操作
    if (direction == kIODirectionIn)
      actualByteCount = m_provider->performRead(buffer, (block*kDiskBlockSize),
                                                (nblks*kDiskBlockSize));
    else
      actualByteCount = m_provider->performWrite(buffer, (block*kDiskBlockSize),
                                                 (nblks*kDiskBlockSize));

    //调用完成函数
    (completion->action)(completion->target, completion->parameter, kIOReturnSuccess,
                         actualByteCount);

    return kIOReturnSuccess;
}
```

请注意，尽管RAM磁盘的传输驱动程序并没有块长度的概念（因为其最小可寻址单元是字节），但IOBlockStorageDevice接口仍然以块为单位表示磁盘容量，并在执行读/写操作时对块进行操作。因此，该块驱动程序的实现任意定义了一个512字节的块长度。

最后，类似于每个I/O Kit驱动程序，我们的RAM磁盘驱动程序需要一个属性列表，描述驱动程序的需求，包括其匹配字典。RAM磁盘驱动程序实现的IOBlockStorageDevice接口是I/O Kit IOStorageFamily框架的一部分，因此需要在RAM磁盘驱动程序的属性列表中显式包含该依赖。这通过向Info.plist文件的OSBundleLibraries段中添加一项来实现，该项引用内核模块com.apple.iokit.IOStorageFamily。在该示例中，我们导入IOStorageFamily的1.6版本，该版本与Mac OS X 10.6中的版本相对应。

---

**注意**　任何实现存储驱动程序栈中驱动程序的内核扩展，都需要在其属性列表中包括IOStorageFamily作为依赖项。

---

我们的示例RAM磁盘驱动程序的属性列表（包括其匹配字典和库依赖），如图14-3所示。

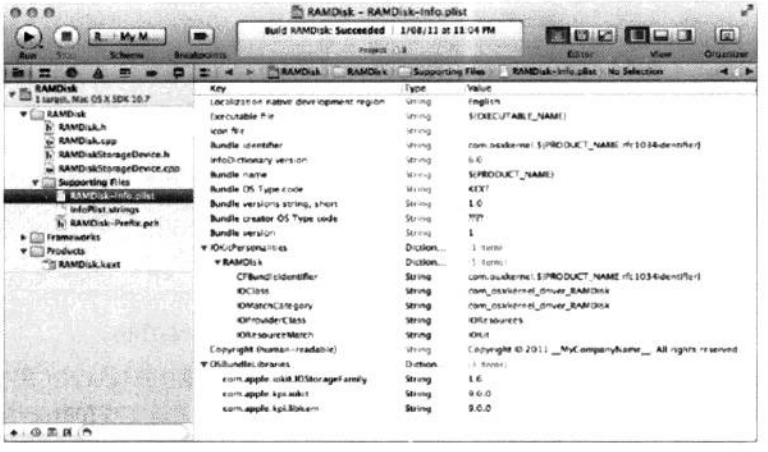


图14-3 示例RAM磁盘驱动程序的属性列表

构建RAM磁盘驱动程序，并加载最终的内核扩展之后，你会看到一个类似于图14-4的对话框。这并不表示设备存在问题，而是表示Mac OS X无法在磁盘上找到可读的文件系统。假如该磁盘尚未写入内容，那么出现这一错误很正常。

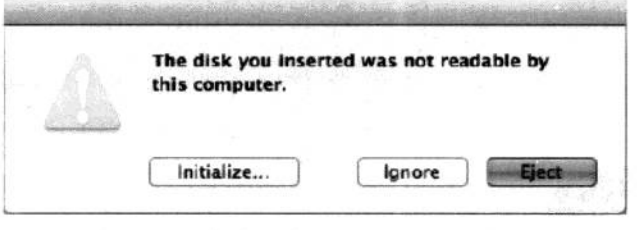



图14-4 插入一个不包含可读文件系统的磁盘时，显示的标准Mac OS X对话框

点击图14-4对话框中显示的“Initialize…”按钮，将加载“Disk Utility”应用程序，这一存储设备就可以使用文件系统进行分区和初始化。在此之前，可以使用`IORegisterExplorer`实用程序检查驱动程序栈的状态，这非常有趣。除了`RAMDisk`传输驱动程序和`RAMDiskStorageDevice`块，你会发现I/O Kit基于该块驱动程序，已经构造了3个驱动程序。该驱动程序栈的状态，如图14-5所示。

Disk Utility应用程序可以将磁盘格式化并初始化为一种文件系统。该处理涉及编写磁盘的分区表，即使磁盘只包含一个分区，这也很必要，然后编写该分区的文件系统。为了格式化磁盘，从左边的磁盘列表中选择设备，并点击Erase选项卡。在Disk Utility中显示的设备名称为`IOBlockStorageDevice`块返回的描述性字符串，因此，就示例RAM磁盘而言，会生成一个名为RAM Disk Media的设备。

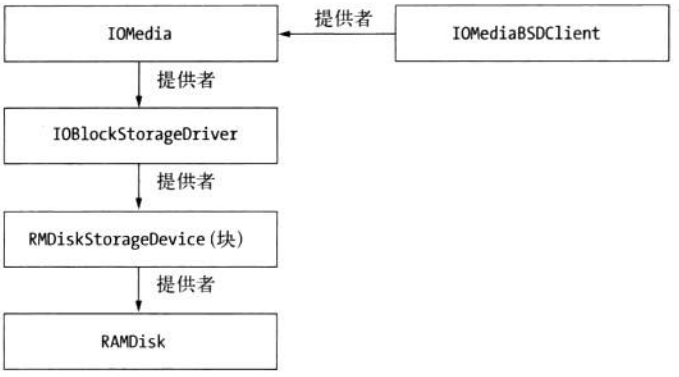



图14-5　加载非格式化的存储设备时创建的驱动程序栈

默认情况下，Disk Utility会编写磁盘的一个GUID分区表，并使用Mac OS扩展文件系统，即HFS+文件系统。Disk Utility并不会执行卷的底层格式化，因此，无需实现IOBlockStorageDevice的doFormatMedia方法。初始化卷的处理如图14-6所示。

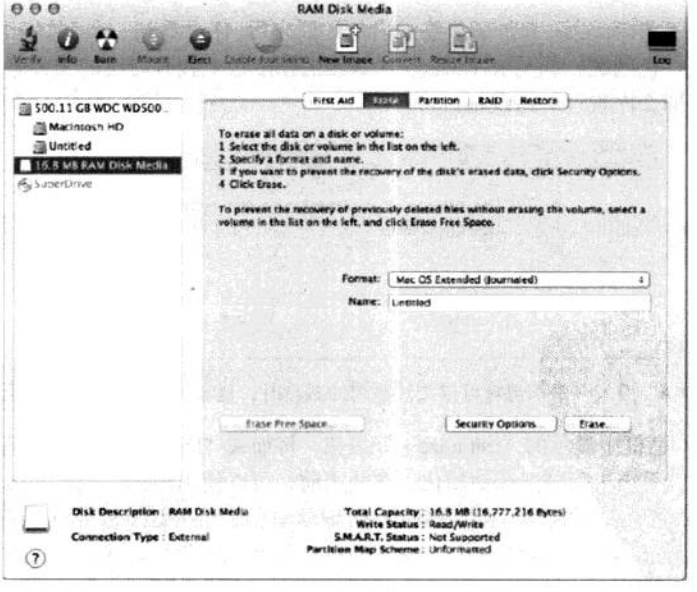



图14-6　在Disk Utility中格式化新卷

在分区映射和文件系统已经写入磁盘之后，RAM磁盘的存储驱动程序栈中将会多出3个驱动程序，如图14-7所示。在表示整个磁盘的IOMedia对象之上的是一个I/O Kit类，它表示磁盘上的分区表，在该示例中，即GUID分区表。每个分区都有一个IOMedia对象，表示分区的逻辑卷。

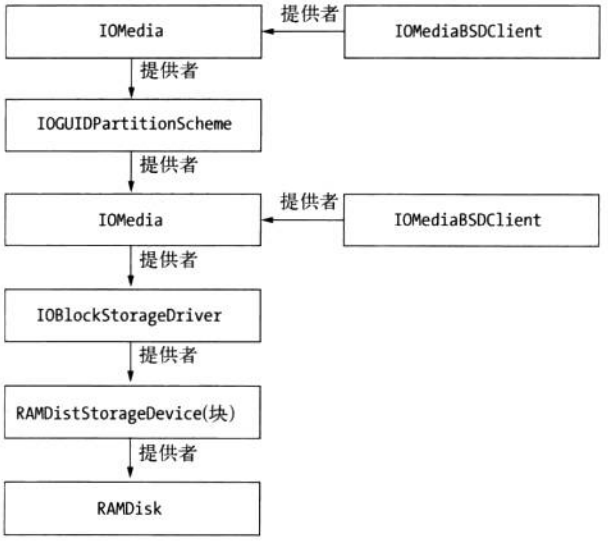



图14-7 使用包含单个分区的GUID分区表进行分区的设备驱动程序栈

实现RAM磁盘的驱动程序时，实现的所有方法都特定于设备的访问数据；无须提供任何方法来处理分区规则或文件系统，或使设备可供用户空间进程访问。所有这些功能都由I/O Kit的IOStorageFamily提供的类处理，特别是IOBlockStorageDevice、IOMedia和IOMediaBSDClient类。这种设计的好处是，功能为所有的存储设备共有，并且可以在共享的类中实现，而无须为每个存储设备编写相同的功能。

IOBlockStorageDevice和IOBlockStorageDriver类表示磁盘驱动器硬件，IOMedia类表示该驱动器中当前存在的磁盘。就我们的示例RAM磁盘驱动程序或USB闪存而言，存在存储设备时，总会有媒介存在，这两部分不可分割。但也并不总是如此，例如，CD驱动器会创建一个IOBlockStorageDevice（I/O Kit将为此创建一个对应的IOBlockStorageDriver），但如果CD驱动器中没有CD，驱动程序栈中就不会有IOMedia对象。

IOMedia类提供磁盘的逻辑表示。如果磁盘已经分区，一个存储设备将会有多个IOMedia对象，一个对应每个分区，另一个表示整个磁盘。就RAID而言，如果已创建了跨多个磁盘的单个卷，那么便会有一个IOMedia对象表示整个逻辑RAID卷。

内核中的每个IOMedia对象都有一个相应的IOMediaBSDClient对象，负责将逻辑磁盘提供给用户空间进程使用。在大多数情况下，进程无须与磁盘驱动程序直接交互，而只是使用装载卷上的文件系统读写卷上包含的文件。在一些情况下，用户空间进程可能需要直接读/写磁盘设备，或直接向磁盘驱动程序发送ioctl调用。

按照BSD惯例，块设备接口和字符设备接口均为每个磁盘及磁盘分区而创建。块设备执行缓冲的I/O，每次读写都通过缓冲区进行缓存。进程执行读操作时，包含受访数据的磁盘块将从磁

盘中读出，置于缓冲区缓存中，然后复制到用户进程提供的目标缓冲区中。如果进程要读取最近已访问的磁盘块，那么该数据很可能位于缓冲区缓存中，因此该读请求无须访问磁盘就能完成。

字符设备提供磁盘存储的直接访问，而不必通过缓冲区进行缓存。这意味着，每次读/写设备都会调用设备IOBlockStorageDevice接口的方法，直接读/写进程的缓冲区。因此，所有对字符设备的读写操作都必须在磁盘块边界上开始，而且传输的字节数必须是磁盘块长度的整数倍。

块设备接口和字符设备接口都由IOMediaBSDClient类创建。块设备接口可以通过/dev/diskN路径访问，而字符设备接口可以通过/dev/rdiskN路径进行访问，其中N为一个整数，确保设备的名称唯一。

构建并加载RAM磁盘驱动程序之后，系统中的磁盘设备列表可以运行终端命令diskutile list检查。对于RAM磁盘设备，使用该命令得到的输出示例，如代码清单14-7所示。disk1接口对应整个存储设备，disk1s1接口对应HFS+分区。

**代码清单14-7　对RAM磁盘设备使用diskutil list命令的输出结果**

```
/dev/disk1
   #:    TYPE                    NAME          SIZE          IDENTIFIER
   0:    GUID_partition_scheme                 *16.8 MB      disk1
   1:    Apple_HFS               VolumeName    16.7 MB       disk1s1
```

## 14.4　分区规则

一个磁盘可以拆分成许多个较小的逻辑单元，对用户来说，每个逻辑单元看起来就像一个单独的磁盘。即使硬盘只有一个分区，也会包含一个分区映射，列出一个或多个已经从该磁盘创建的分区。Mac OS X支持许多常见的分区规则，包括Mac OS X上默认的GUID分区规则、主引导记录（Windows上也很常见）以及苹果分区映射，分区映射在Mac转换成基于英特尔的芯片之前，是Mac OS的默认分区规则。

编写派生自IOPartitionScheme类的I/O Kit驱动程序，便可以向Mac OS X中添加新的分区规则支持。每当插入磁盘时，便会加载分区规则驱动程序，并扫描磁盘，查找驱动程序识别的分区表。如果找到该驱动程序支持的分区表，该驱动程序将为分区表中的每项创建一个IOMedia对象，并在驱动程序存储栈上为其加载这些IOMedia对象。分区驱动程序创建的IOMedia对象仅描述分区涵盖的磁盘段，而不是整个磁盘内容。

IOMedia对象可以描述整个磁盘，或磁盘上的单个分区，该分区包含具有物理连续性的块子集。当IOMedia对象实例化时，其构造函数中会包含一个属性，以确定IOMedia对象是否描述整个磁盘。尽管一个分区驱动程序会加载一个IOMedia对象，但它只加载描述整个磁盘内容的那个对象，因为通常分区表并不位于分区之中。

分区规则驱动程序成功扫描磁盘之后，最终会构建一个类似于图14-7所示的存储驱动程序栈，其中包含一个描述整个磁盘内容的IOMedia对象（该对象在驱动程序栈的IOBlockStorage-Device之上），以及每个分区的IOMedia对象（该对象在驱动程序栈的IOGUIDPartitionTableScheme之上）。

要注意，分区规则驱动程序只负责读取磁盘上存在的分区表，IOPartitionScheme类并不包含编写分区表的方法。分区表能通过提供一个可直接写入磁盘设备的用户空间实用程序进程创建。

### 14.4.1 实现一个示例分区规则

我们将在本节查看如何在I/O Kit中实现虚构分区映射驱动程序。首先，检查一下该驱动程序的属性列表，特别是其匹配字典，如代码清单14-8所示。

**代码清单14-8 分区规则驱动程序属性列表的示例匹配字典**

```
<key>IOKitPersonalities</key>
<dict>
        <key>SamplePartitionScheme</key>
        <dict>
                <key>CFBundleIdentifier</key>
                <string>com.osxkernel.SamplePartitionScheme</string>
                <key>IOClass</key>
                <string>com_osxkernel_driver_SamplePartitionScheme</string>
                <key>IOMatchCategory</key>
                <string>IOStorage</string>
                <key>IOProviderClass</key>
                <string>IOMedia</string>
                <key>IOPropertyMatch</key>
                <dict>
                        <key>Whole</key>
                        <true/>
                </dict>
        </dict>
</dict>
```

该匹配字典有三个重要的方面。

- 指定提供者类IOMedia，从而在每次插入新的磁盘（以及创建一个IOMedia对象，表示该磁盘）时，分区驱动程序都能够检查支持分区表的磁盘内容。
- 分区驱动程序只对表示整个磁盘的IOMedia对象感兴趣，因为分区表并不位于磁盘分区中。为了缩小匹配范围，仅匹配表示整个磁盘的IOMedia对象，匹配字典使用IOPropertyMatch键指定I/O Kit应该只加载满足以下对象的驱动程序，即Whole属性为布尔值true的IOMedia对象。这是IOMedia对象的一个标准属性，指定该对象是涵盖整个磁盘，还是磁盘的一个分区。
- 属性列表指定IOStorage的IOMatchCategory。该属性位于分区驱动程序的属性表中，对正确构造存储驱动程序栈非常重要。特别是，某些驱动程序将使用IOMatchCategory属性，确定它们是否位于驱动程序栈的顶层，或顶层的驱动程序是否也是IOStorage栈的一部分。

尽管分区规则驱动程序是存储驱动程序栈的一部分，并且仅在磁盘移除之后卸载，但该驱动程序本身仅在磁盘第一次插入时起作用，从磁盘读取分区表，并为其找到的每个分区实例化一个IOMedia对象。作为派生自标准I/O Kit类IOService的驱动程序，分区规则驱动程序通过init()、probe()和start()方法实现这些处理，并通过stop()和free()方法卸载。

probe()方法对IOPartitionScheme驱动程序具有特殊意义。一个磁盘添加到系统时，都会实例化一个分区规则驱动程序，由该驱动程序来确定磁盘是否包含支持的分区表，如果不包含，则加载更适合的IOPartitionScheme驱动程序。这通过标准的IOService方法probe()完成。通常，probe()方法的作用是检查硬件，确定该驱动程序是否支持该设备，如果支持，则返回一个整数值，表示该驱动程序对该硬件的适合程度。具有高检测得分的驱动程序将会被I/O Kit加载。

就IOPartitionScheme驱动程序而言，probe()方法的作用是充分读取磁盘，确定该磁盘上的分区表是否被该驱动程序所支持，如果支持，则继续读取分区表项。并不严格要求在probe()方法中读取整个分区表，但这么做在调用驱动程序的start()方法时，无须重新扫描分区表。苹果提供的分区规则驱动程序作为Mac OS X的一部分，更近了一步，而且为probe()方法中找到的每个分区实际实例化了一个IOMedia对象。

如果一个磁盘包含分区规则驱动程序识别的分区表，并且该分区驱动程序由I/O Kit选择作为最适合的驱动程序，那么会调用其start()方法。此时，分区规则驱动程序应该为每个分区项创建一个IOMedia对象，并将其装载到存储驱动程序栈中。

probe()和start()方法的示例实现，如代码清单14-9所示。该驱动程序基于分区规则驱动程序，为IOStorage族中Darwin源代码的一部分。

类似于Darwin中的分区驱动程序，代码清单14-9中的实现包含一个自定义方法scan()，用于检查磁盘，如果找到支持的分区表，则为每个分区实例化一个IOMedia对象，并通过OSSet对象将分区集返回给调用者。如果在扫描期间未找到IOMedia对象，则probe()方法执行失败，返回NULL结果值，而I/O Kit将继续为磁盘查找另一个分区规则驱动程序。如果找到支持的分区表，检测方法会将表示每个分区的IOMedia对象集保存到实例变量m_partitions中。

**代码清单14-9　分区规则驱动程序的probe()和start()方法实现**

```
#include <IOKit/storage/IOPartitionScheme.h>

//定义超类
#define super IOPartitionScheme

OSDefineMetaClassAndStructors(com_osxkernel_driver_PartitionScheme, IOPartitionScheme)

IOService* com_osxkernel_driver_PartitionScheme::probe(IOService* provider, SInt32* score)
{
        if (super::probe(provider, score) == NULL)
                return NULL;

        //扫描IOMedia，查找支持的分区表
        m_partitions = scan(score);

        //如果未找到分区表，则返回NULL
        return m_partitions ? this : NULL;
}

bool com_osxkernel_driver_PartitionScheme::start (IOService *provider)
{
```

```
        IOMedia*                partition;
        OSIterator*             partitionIterator;

        if (super::start(provider) == false)
                return false;

        //对检测时找到的和实例化的IOMedia对象创建一个迭代器
        partitionIterator = OSCollectionIterator::withCollection(m_partitions);
        if (partitionIterator == NULL)
                return false;

        //装载并注册每个IOMedia对象（表示找到的分区）
        while ((partition = (IOMedia*)partitionIterator->getNextObject()))
        {
                if (partition->attach(this))
                {
                        attachMediaObjectToDeviceTree(partition);
                        partition->registerService();
                }
        }
        partitionIterator->release();

        return true;
}
```

如果probe()方法返回成功，并且I/O Kit未找到较好的驱动程序，我们的分区驱动程序将会被添加到存储栈，其start()方法也将被调用。分区驱动程序start()方法的作用是，将它的每个IOMedia对象装载到存储驱动程序栈，这些IOMedia对象表示磁盘上的分区项。这通过IOService的attach()方法完成，该方法将IOMedia对象插入到I/O Registry的服务平面中，作为分区驱动程序（属提供者类）的一个子驱动程序。

除了将IOMedia对象插入到I/O Registry的服务平面，还需要将IOMedia对象插入到I/O Registry的设备平面中。这仅在分区可能会用作基于PowerPC的麦金塔电脑的启动卷时才需要。因为基于PowerPC的麦金塔电脑上的启动卷通过它在I/O Registry设备平面中的位置来确定，所以表示启动分区的IOMedia对象需要在设备平面中有一个选项。IOPartitionScheme超类提供了一个attachMediaObjectToDeviceTree()方法，将IOMedia对象插入到I/O Registry的设备平面中。

scan()方法是一个自定义方法，确定磁盘是否包含驱动程序所支持的分区规则，如果包含，则从该磁盘读取分区表项，并创建IOMedia对象集，表示每个分区。这需要分区驱动程序能够读取该磁盘，通过驱动程序的提供者对象执行。正如在驱动程序的匹配字典中指定的那样（参见代码清单14-8），分区驱动程序的提供者类是一个表示整个磁盘的IOMedia对象。scan()方法的实现示例，如代码清单14-10所示。

**代码清单14-10 检测磁盘上是否存在示例分区表，并为分区表实例化IOMedia对象的方法**

```
OSSet*  com_osxkernel_driver_PartitionScheme::scan(SInt32* score)
{
        IOBufferMemoryDescriptor*       buffer          = NULL;
        SamplePartitionTable*           sampleTable;
```

```
1       IOMedia*                    media          = getProvider();
        UInt64                      mediaBlockSize = media->getPreferredBlockSize();
        bool                        mediaIsOpen    = false;
        OSSet*                      partitions     = NULL;
        IOReturn                    status;

        //确定该媒介是否已经格式化
2       if (media->isFormatted() == false)
                goto bail;
        //分配一个扇区的缓冲区，保存从磁盘读取的数据
3       buffer = IOBufferMemoryDescriptor::withCapacity(mediaBlockSize, kIODirectionIn);
        if (buffer == NULL)
                goto bail;

        //分配一个集合，保存表示磁盘分区的媒介对象
4       partitions = OSSet::withCapacity(8);
        if (partitions == NULL)
                goto bail;

        //打开存储驱动程序栈（分区驱动程序是它的一部分），进行读访问
5       mediaIsOpen = open(this, 0, kIOStorageAccessReader);
        if (mediaIsOpen == false)
                goto bail;

        //读取磁盘的第一个扇区
6       status = media->read(this, 0, buffer);
        if (status != kIOReturnSuccess)
                goto bail;
        sampleTable = (SamplePartitionTable*)buffer->getBytesNoCopy();

        //确定第一个扇区是否包含我们识别的分区签名
7       if (strcmp(sampleTable->partitionIdentifier, kSamplePartitionIdentifier) != 0)
                goto bail;

        //扫描分区映射中有效的分区项
8       for (int index = 0; index < sampleTable->partitionCount; index++)
        {
9               if (isPartitionInvalid(&sampleTable->partitionEntries[index]))
                        continue;

                IOMedia*        newMedia;
10              newMedia = instantiateMediaObject(&sampleTable->partitionEntries[index],
                                                  1+index);
                if ( newMedia )
                {
                        partitions->setObject(newMedia);
                        newMedia->release();
                }
        }

        //释放临时资源
11      close(this);
        buffer->release();
```

```
        return partitions;

bail:
        //非成功返回，释放所有分配的对象
12      if ( mediaIsOpen )      close(this);
        if ( partitions )       partitions->release();
        if ( buffer )           buffer->release();

        return NULL;
}
```

下面列出的每个步骤都与代码清单14-10中有编号的行相对应，大致说明了各步执行的操作。

(1) 获取一个指向提供者类的指针，这是一个表示整个磁盘的IOMedia对象。所有的磁盘读取都通过该对象执行，包括读取磁盘的分区表。

(2) 检查分区表之前，检查磁盘介质的某些属性。如果磁盘的介质没有格式化，则放弃扫描。在此也对各种需求进行验证，如分区规则可能需要的最小磁盘块长度。

(3) 从磁盘读取的所有数据都写入IOMemoryDescriptor。因此分配一个IOBufferMemoryDescriptor，保存该方法从磁盘中读取的数据内容。由于该内存描述符将在从磁盘读取数据的操作中使用，所以其方向必须设置为kIODirectionIn。

(4) 分配一个OSSet容器，保存表示从磁盘上找到的每个分区对象IOMedia的集合。尽管OSSet集合的初始容量是8个对象，但OSSet在插入8个以上的对象时，会自动进行扩展。

(5) 打开存储驱动程序栈（分区驱动程序是它的一部分），进行读访问。由于分区驱动程序只从该磁盘读取分区表，而不修改磁盘内容，所以它只需要读访问。

(6) 从磁盘读取第一个扇区。这通常是分区表头的位置。该示例中使用的假定分区规则将其分区表头存储在第一个磁盘扇区中。read()方法的缓冲区参数指定读取数据的目标和读取的字节数。

(7) 分区驱动程序根据从磁盘中读出的数据来确定该磁盘是否包含其支持的分区规则。该代码面向此分区规则，我们的示例驱动程序使用的虚构分区规则通过写入磁盘第一个块的字符串常量来确定。因而，该驱动程序使用strcmp()函数，确定该字符串是否存在，如果无法找到，它会认为磁盘上存在另外一个分区规则，返回失败。

(8) 该代码迭代从磁盘读取的分区表中的每项。该代码面向此分区规则，该示例中使用的虚构分区规则将整个分区表存储在初始磁盘扇区中。

(9) 验证分区项。这可能涉及一些检查，如确保开始块和分区项的长度不超过磁盘容量。

(10) 实例化一个新的IOMedia对象，来表示分区项。

(11) 如果成功扫描分区表，则关闭该存储驱动程序栈（与前面的open()调用相呼应），并将IOMedia对象集返回给调用者。

(12) 如果发生错误，该代码将释放分配的资源。

第10步中调用的instantiateMediaObject()方法的实现，如代码清单14-11所示。这是由我们的分区规则驱动程序定义的一个自定义方法。

**代码清单14-11　实例化表示专用磁盘分区的IOMedia对象的方法**

```
IOMedia* com_osxkernel_driver_PartitionScheme::instantiateMediaObject
                                (SamplePartitionEntry* sampleEntry, int index)
{
        IOMedia*        media           = getProvider();
        UInt64          mediaBlockSize  = media->getPreferredBlockSize();
        IOMedia*        newMedia;

1       newMedia = new IOMedia;
        if ( newMedia )
        {
                UInt64          partitionBase, partitionSize;

2               partitionBase = OSSwapLittleToHostInt64(sampleEntry->blockStart) *
                                        mediaBlockSize;
                partitionSize = OSSwapLittleToHostInt64(sampleEntry->blockCount) *
                                        mediaBlockSize;

3               if ( newMedia->init(partitionBase, partitionSize, mediaBlockSize,
                     media->getAttributes(), false, media->isWritable()))
                {
4                       //为该分区设置一个名称
                        newMedia->setName(sampleEntry->name);

                        //为该分区设置一个位置值（分区号）
                        char location[12];
                        snprintf(location, sizeof(location), "%d", index);
                        newMedia->setLocation(location);

                        //为该分区设置Partition ID键
                        newMedia->setProperty(kIOMediaPartitionIDKey, index, 32);
                }
                else
                {
5                       newMedia->release();
                        newMedia = NULL;
                }
        }

6       return newMedia;
}
```

下面列出的每个步骤都与代码清单14-11中有编号的行相对应，大致说明了各步执行的操作。

(1) 使用C++操作符new分配一个IOMedia对象。

(2) 从分区表项读取分区的初始磁盘块号和分区长度。分区规则有一个标准端序，这可能与主机的本地字节顺序不同，该主机为运行驱动程序的主机，因此，重要的一点是，使用字节顺序宏（如OSSwapLittleToHostInt64()），确保能够正确读取数据。

(3) 初始化分配的IOMedia对象。IOMedia::init()方法的参数如下所示：

```
virtual bool init(UInt64                base,
                  UInt64                size,
```

```
UInt64                  preferredBlockSize,
IOMediaAttributeMask    attributes,
bool                    isWhole,
bool                    isWritable,
const char*             contentHint = 0,
OSDictionary*           properties  = 0);
```

参数base和size定义磁盘上的分区位置，单位为字节。另一个重要的参数是布尔参数isWhole，它被设置为false，表示该IOMedia对象代表一个分区，而不是整个磁盘。contentHint参数描述分区内容，如卷使用的文件系统。下一节会对contentHint属性进行描述。

(4) 在分区的IOMedia对象上设置该分区的各种属性。这包括分区名称以及位置和分区ID，这两项都源于该分区在分区表中的索引。

(5) 如果IOMedia对象无法成功初始化，则将其释放。

(6) 将初始化的IOMedia对象返回给调用者，如果该对象无法成功初始化，则为NULL。

最后，卸载该分区规则的驱动程序时，必须将其IOMedia对象从驱动程序栈移除，并将其释放。分区驱动程序被卸载的原因是，磁盘已经弹出，或者是磁盘已重新格式化，这种情况下，一个新分区表，甚至可能是一个不同的分区规则，已被写入该磁盘。

分区规则驱动程序stop()和free()方法的示例实现，如代码清单14-12所示。stop()方法将每个IOMedia对象从I/O Registry的设备平面移除，撤销分区驱动程序在其start()方法中执行的attachMediaObjectToDeviceTree()调用。在分区驱动程序卸载之前，会调用其free()方法，释放OSSet，其中保存着每个分区项的IOMedia对象集合。

**代码清单14-12 分区规则驱动程序stop()和free()方法的实现**

```
void com_osxkernel_driver_PartitionScheme::stop(IOService* provider)
{
        IOMedia*                partition;
        OSIterator*             partitionIterator;

        //释放前面装载到设备树上的媒介对象
        partitionIterator = OSCollectionIterator::withCollection(m_partitions);
        if (partitionIterator)
        {
                while ((partition = (IOMedia*)partitionIterator->getNextObject()))
                {
                        detachMediaObjectFromDeviceTree(partition);
                }

                partitionIterator->release();
        }

        super::stop(provider);
}

void com_osxkernel_driver_PartitionScheme::free (void)
{
        if (m_partitions != NULL)
```

```
            m_partitions->release();

        super::free();
}
```

### 14.4.2　媒介内容线索属性

我们在代码清单14-11中看到，IOMedia类的初始化方法有一个contentHint参数。尽管该参数并不需要IOMedia对象解释，但它在驱动程序存储栈的构造中起着非常重要的作用。contentHint参数是一个字符串值，描述磁盘上IOMedia对象所包含的内容。对于表示整个磁盘的IOMedia对象，内容线索可以标识磁盘包含的分区规则。对于表示单个分区的IOMedia对象，内容线索可以标识卷所使用的文件系统类型。内容线索的用途还可以自定义，例如，提供磁盘加密的驱动程序可以使用内容线索来描述磁盘上已经使用的加密规则。

内容线索不是用来向用户描述内容，而是提供系统上其他驱动程序可以使用的信息。传递给IOMedia类初始化方法的contentHint参数在IOMedia对象上设置为一个I/O Registry属性。这使存储栈中的其他驱动程序可以访问内容线索的值，但更重要的是，它提供一个可以指定的属性，匹配另一个驱动程序的匹配字典。

创建分区规则驱动程序时，会指定一个IOPropertyMatch项（参见代码清单14-8），限制该驱动程序匹配特定的IOMedia对象。就分区规则驱动程序而言，我们仅匹配表示整个磁盘的IOMedia对象。实现方法是通知I/O Kit，分区驱动程序应该仅匹配包含Whole属性且属性值为true的IOMedia对象。同样，驱动程序可以将IOPropertyMatch项添加到包含Content Hint键的匹配字典，并指定一个包含该驱动程序感兴趣的特殊内容类型的值。例如，这可以用来防止磁盘加密驱动程序加载未加密的IOMedia卷。

内容线索属性的另一个重要作用是，确定为IOMedia卷加载的正确文件系统驱动程序。Mac OS X将只在IOMedia对象的内容线索值标识支持的文件系统时，才加载文件系统驱动程序。

由于内容线索值需要在IOMedia对象初始化时指定，所以任何实例化IOMedia对象的驱动程序都需要知道该对象表示的磁盘或分区的内容。对于分区规则驱动程序，内容线索来自磁盘上存储的分区表。例如，苹果分区映射包含每个分区项的字符串值，直接用作内容线索值。GUID分区表包含每个分区的128位GUID，标识文件系统及该分区的内容。该GUID被转换为一个字符串表示，然后用作内容线索。这意味着，可能存在多个内容线索值来标识一个特定文件系统，因此，文件系统驱动程序必须匹配能够标识其文件系统的IOMedia内容线索的每个可能值。

## 14.5　媒介过滤器驱动程序

驱动程序栈的顶层可能包含一个或多个媒介过滤器驱动程序。媒介过滤器驱动程序，也称为过滤器规则驱动程序，匹配存储栈中现有的IOMedia对象，并创建一个新的IOMedia对象，表示过滤的媒介对象。所有对磁盘的读写请求都要通过过滤器规则驱动程序进行传递，过滤器驱动程序从而可以操作读取的块。在存储栈中，当数据在原始的IOMedia对象和它上面的过滤IOMedia对象

之间传输时，过滤器驱动程序甚至可以操作数据。

过滤器规则驱动程序可以实现各种功能。例如，过滤器驱动程序可以通过匹配表示磁盘上加密分区的IOMedia对象，并发布一个表示文件系统使用的非加密分区的IOMedia对象，来实现块级别的磁盘加密。过滤器规则驱动程序的另一种用途是，实现一个RAID驱动程序，匹配多个IOMedia对象，其中每个IOMedia对象表示RAID集中的一个磁盘，并创建一个IOMedia对象，表示逻辑卷。过滤器规则驱动程序与其控制的IOMedia对象和发布的IOMedia对象之间的关系，如图14-8所示。

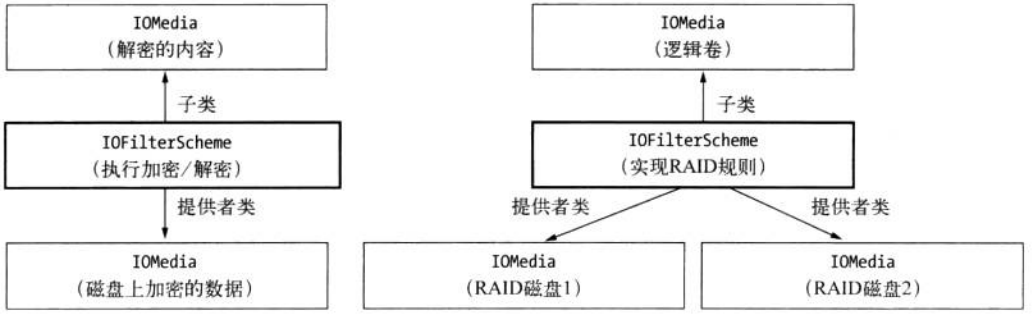



图14-8 过滤器规则驱动程序及其提供者类与为加密规则（左）和RAID驱动程序（右）创建的IOMedia对象之间的关系

在上一节中开发的分区规则驱动程序可以看做一种特殊类型的过滤器驱动程序。类似于过滤器驱动程序，分区规则驱动程序加载现有的IOMedia对象，创建一个或多个IOMedia对象，表示磁盘上的分区。但与分区规则不同，通用过滤器驱动程序可以有多个提供者类，如图14-8中的RAID驱动程序。另一个不同点是，与过滤器驱动程序不同，分区规则驱动程序通常并不处理通过它所创建的IOMedia对象而发出的每个读/写请求。

I/O Kit提供了一个IOFilterScheme类，是任何实现媒介过滤器规则的驱动程序的超类。过滤器规则驱动程序通常使用与其匹配的IOMedia对象的Content Hint属性值，来限制过滤器规则只加载它可以支持的IOMedia对象。例如，苹果RAID软驱动程序将RAID集中的每个磁盘格式化为GUID分区表，这样，每个磁盘的IOMedia对象都会包含一个GUID，作为其“内容线索”属性。苹果已经定义了一个GUID，表示该磁盘分区为RAID集的一部分，苹果RAID驱动程序将会与其匹配。当苹果RAID驱动程序创建表示逻辑卷的IOMedia子对象时，将会赋给该IOMedia对象一个内容线索，表示写入到整体RAID集中的文件系统。



**注意** 文件系统驱动程序将只加载驱动程序存储栈中顶层（叶）的IOMedia对象。这意味着，即使过滤器规则驱动程序匹配包含可读文件系统的IOMedia对象，并使用可读的文件系统创建另外一个IOMedia对象，也只有栈中过滤器规则驱动程序上面的对象能够装载到用户的桌面上。

### 14.5.1　加密过滤器规则示例

我们通过实现一个简单的块级别加密驱动程序，来看一下示例过滤器规则驱动程序的实现。从安全角度来看，该示例驱动程序并不执行复杂的工作，只是实现基本的XOR加密规则，但它确实能展示过滤器规则驱动程序的体系结构。

我们开发的过滤器规则将加密整个分区的内容，我们的示例需要将磁盘格式化为标准的GUID分区表。为了标识加密的分区，我们将定义一个新的GUID，描述分区内容，对此可以使用命令行工具uuidgen来实现。在该示例中，将使用GUID 8D7FD0BB-39A8-43C0-9432-F4E1A269F070，它由我们的示例驱动程序定义，用来描述包含HFS文件系统的加密磁盘分区。从现在开始，本章将使用术语Encrypted_HFS_GUID，而不是GUID全称。

本示例将使用标准的GUID分区表，这意味着，存储驱动程序栈中的分区规则驱动程序是标准的苹果GUID分区规则驱动程序。苹果驱动程序从磁盘上的分区头，将每个分区IOMedia对象的Content Hint属性设置为分区类型GUID。这意味着，我们的加密过滤器驱动程序希望加载的IOMedia对象都会有一个内容线索Encrypted_HFS_GUID。该加密过滤器驱动程序可以忽略所有其他的IOMedia对象，因为它知道这些对象并不表示加密分区。该要求可以通过在过滤器驱动程序的匹配字典中，添加一个IOPropertyMatch键来体现，如代码清单14-13所示。

**代码清单14-13　实现加密的示例过滤器规则驱动程序属性列表的匹配字典**

```
<key>IOKitPersonalities</key>
<dict>
        <key>SampleEncryptionFilter</key>
        <dict>
                <key>Content Mask</key>
                <string>Sample_Encrypted_Data</string>
                <key>CFBundleIdentifier</key>
                <string>com.osxkernel.SampleEncryptionFilter</string>
                <key>IOClass</key>
                <string>com_osxkernel_driver_SampleEncryptionFilter</string>
                <key>IOMatchCategory</key>
                <string>IOStorage</string>
                <key>IOProviderClass</key>
                <string>IOMedia</string>
                <key>IOPropertyMatch</key>
                <dict>
                        <key>Content Hint</key>
                        <string>8D7FD0BB-39A8-43C0-9432-F4E1A269F070</string>
                </dict>
        </dict>
</dict>
```

过滤器规则驱动程序加载时，可能需要检测其IOMedia提供者类，确定它是否包含过滤器驱动程序所支持的内容。如果包含，则创建一个或多个IOMedia子对象，表示过滤的卷。这些步骤类似于代码清单14-9中分区规则驱动程序probe()和start()方法的实现。但不同于分区规则驱动程序，我们的示例加密过滤器规则驱动程序并不需要probe()方法，因为该驱动程序的属性列表

已经进行了设置，能确保该驱动程序只加载Content Hint属性中包含Encrypted_HFS_GUID类型的IOMedia对象。因此，如果该驱动程序进行加载，可以断定其加载的是加密分区。

在本示例过滤器规则驱动程序的start()方法中，需要创建一个新的IOMedia对象，表示过滤的磁盘内容，通过该IOMedia对象，将未加密的数据提供给系统的其他部分（如文件系统）。类似于分区规则驱动程序，过滤器驱动程序需要正确设置它创建的IOMedia子对象的contentHint参数，这非常重要，因为通过该方式，系统能够确定哪一个文件系统（或另一个过滤器规则驱动程序）要加载该IOMedia卷。在本示例加密过滤器中，让它加密HFS文件系统，因此，过滤器规则发布的IOMedia子对象创建时，其contentHint值为Apple_HFS。

本示例加密过滤器规则start()方法的实现，如代码清单14-14所示。本示例过滤器规则并不提供init()或probe()方法的实现，因为超类提供的实现已经足够。

**代码清单14-14 提供加密的示例过滤器规则start()方法的实现**

```
#include <IOKit/storage/IOFilterScheme.h>

//定义超类
#define super IOFilterScheme

OSDefineMetaClassAndStructors(com_osxkernel_driver_SampleEncryptionFilter, IOFilterScheme)

bool com_osxkernel_driver_SampleEncryptionFilter::start (IOService *provider)
{
        if (super::start(provider) == false)
                return false;

        //保存我们提供者类的一个引用，并验证它是否是一个IOMedia对象
        m_encryptedMedia = OSDynamicCast(IOMedia, provider);
        if (m_encryptedMedia == NULL)
                return false;

        //创建一个子IOMedia对象，表示未加密的数据
        m_childMedia = instantiateMediaObject();
        if (m_childMedia == NULL)
                return false;

        //将未加密的IOMedia对象装载至存储驱动程序栈
        if (m_childMedia->attach(this) == false)
                return false;
        m_childMedia->registerService();

        return true;
}


IOMedia* com_osxkernel_driver_SampleEncryptionFilter::instantiateMediaObject ()
{
        IOMedia*        newMedia;

        //分配一个新的IOMedia对象
        newMedia = new IOMedia;
```

```
        if ( newMedia )
        {
                //初始化该子IOMedia对象
                //几乎它的所有参数都可以从提供者类获得
                if ( newMedia->init(0,                                  // base
                                m_encryptedMedia->getSize(),
                                m_encryptedMedia->getPreferredBlockSize(),
                                m_encryptedMedia->getAttributes(),
                                false,                                  // isWhole
                                m_encryptedMedia->isWritable(),
                                "Apple_HFS"))                           // contentHint
                {
                        //设置该媒介对象的位置值（分区号）
                        newMedia->setLocation("1");
                }
                else
                {
                        newMedia->release();
                        newMedia = NULL;
                }
        }

        return newMedia;
}
```

instantiateMediaObject()方法是SampleEncryptionFilter类的一个自定义方法，负责创建IOMedia子对象，表示未加密的磁盘内容。IOMedia子对象的许多属性可以直接从过滤器驱动程序的加密IOMedia提供者类获得。对于实现块级别加密的驱动程序，则无须修改IOMedia子对象的卷长度、磁盘块长度等属性。通常，无法防止过滤器规则驱动程序创建一个与其提供者类长度或块长度不同的IOMedia设备，RAID驱动程序的过滤器规则可能就需要这么做。

对于本示例加密驱动程序，需要截获未加密的IOMedia子对象上执行的所有读写操作。因为本过滤器规则位于未加密的IOMedia对象（我们创建的子对象）和加密的IOMedia对象（我们的提供者类）之间，所有在IOMedia子对象上执行的读写操作都会通过本过滤器驱动程序传递，因此，截获的这些操作仅涉及覆写read()和write()方法的超类实现。

就读操作而言，本加密过滤器驱动程序需要将读请求传递给加密的IOMedia对象，并解密返回数据。因读操作的异步执行而使这变得异常复杂，因此，过滤器驱动程序需要提供完成回调，以便在读操作完成时得到通知。这时，从加密的卷读回的数据已被解密，并调用原始的读完成回调（由启动该读操作的客户端提供）。该实现如代码清单14-15所示。

**代码清单14-15　提供加密的示例过滤器规则read()方法的实现**

```
void    com_osxkernel_driver_SampleEncryptionFilter::read (IOService* client,
                                UInt64 byteStart,
                                IOMemoryDescriptor* buffer, IOStorageAttributes* attributes,
                                IOStorageCompletion* completion)
{
        ReadCompletionParams*   context;
        IOStorageCompletion     newCompletion;
```

```
        //分配一个结构体，在读操作异步执行时，保存状态
1       context = (ReadCompletionParams*)IOMalloc(sizeof(ReadCompletionParams));
        if (context == NULL)
        {
                complete(completion, kIOReturnNoMemory);
                return;
        }

2       context->completion = *completion;
        context->buffer = buffer;
        context->buffer->retain();

        //设置一个回调函数，以便在加密数据已从磁盘读出时通知我们
3       newCompletion.target = this;
        newCompletion.action = readCompleted;
        newCompletion.parameter = context;

        //从磁盘读取加密数据
4       m_encryptedMedia->read(client, byteStart, buffer, attributes, &newCompletion);
}

void    com_osxkernel_driver_SampleEncryptionFilter::readCompleted (void* target,
                                                                    void* parameter,
                                                    IOReturn status, UInt64 actualByteCount)
{
        ReadCompletionParams*   context = (ReadCompletionParams*)parameter;

        //解密从磁盘读取的数据
5       if (status == kIOReturnSuccess)
                status = decryptBuffer(context->buffer, actualByteCount);

        //如果从磁盘读取或解密操作失败，则将actualByteCount设置为0
        if (status != kIOReturnSuccess)
                actualByteCount = 0;

        //调用原始调用者的完成函数

6       complete(&context->completion, status, actualByteCount);

7       context->buffer->release();
        IOFree(context, sizeof(ReadCompletionParams));
}
```

下面列出的每个步骤都与代码清单14-15中有编号的行相对应，大致说明了各步执行的操作。

(1) 由于读操作属异步执行，完成回调中读取操作所需的任何变量或状态都需要保存到一个临时结构体中。我们使用IOMalloc()函数在内存中分配一个结构体，保存需要传递给完成回调的内容。

(2) 初始化分配的上下文结构体。需要保存的一个参数是调用者提供的IOStorageCompletion结构，它包含调用者希望在读操作完成时通知的回调函数。还保存一个IOMemoryDescriptor引用，将磁盘数据读入其中。由于我们将在回调中引用该对象，所以要保留该对象，以防止它在回调触发之前释放。

(3) 设置一个IOStorageCompletion结构体，传递我们自己的回调函数，以在异步读操作完成时得到通知。

(4) 读取加密的IOMedia对象。

(5) 读操作完成时，调用我们指定的回调函数（readCompleted）。如果读操作成功完成，将对从加密的IOMedia对象读回的数据进行解密。

(6) 调用调用者提供的IOStorageCompletion回调，通知调用者缓冲区包含它请求的数据。

(7) 释放IOMemoryDescriptor的引用，并释放第一步中分配的结构体。

实现加密过滤器规则的写操作非常简单，因为在将结果数据写入到加密的IOMedia对象之前，可以执行加密。因而，即使异步执行写操作，也无须替换调用者提供的完成回调（不同于写操作）。

我们并不是就地加密数据，而是分配一个新的IOMemoryDescriptor，保存加密的数据。这样就无须修改调用者的缓冲区，这非常重要，因为写操作不应该改变源缓冲区的内容。即使异步执行写操作，存储栈中执行该操作的驱动程序在写期间，仍将保留IOMemoryDescriptor缓冲区。这可以在将写操作发送给加密的IOMedia对象之后，立即释放对该对象的引用。

加密过滤器规则write()方法的实现，如代码清单14-16所示。

**代码清单14-16　提供加密的示例过滤器规则write()方法的实现**

```
Void    com_osxkernel_driver_SampleEncryptionFilter::write (IOService* client,
                                UInt64 byteStart,
                                IOMemoryDescriptor* buffer, IOStorageAttributes* attributes,
                                IOStorageCompletion* completion)
{
        IOMemoryDescriptor*             newDesc;

        //分配一个缓冲区，保存加密数据，并执行加密
        newDesc = encryptBuffer(buffer);
        if (newDesc == NULL)
        {
                //目标缓冲区无法分配时，返回错误
                complete(completion, kIOReturnNoMemory);
                return;
        }

        //将加密数据写入到加密的IOMedia对象
        m_encryptedMedia->write(client, byteStart, newDesc, attributes, completion);

        //释放加密的IOMemoryDescriptor引用
        newDesc->release();
}
```

## 14.5.2　创建一个自定义GUID分区表

上一节中开发的加密过滤器规则，只加载Content Hint属性值为示例过滤器规则自定义值的IOMedia对象。为了测试该驱动程序，需要创建一个GUID分区表，包含自定义GUID类型的分区项。

这可以通过Mac OS X自带的各种命令行工具执行。为了演示目的，创建一个包含GUID分区表的卷，用来测试加密过滤器驱动程序。加密卷的存储设备由磁盘映像提供，这是一个类似于虚拟磁盘的普通文件，并包含一个可以装载为Mac OS X桌面上卷的文件系统（可能为一个分区规则）。磁盘映像提供了一个非常便捷的方式，来测试过滤器规则和分区规则驱动程序，它们很容易创建，无须格式化物理媒介。

本节提供了在存储栈中开发驱动程序时可以使用的一些命令行工具的演示示例。本节将创建一个磁盘映像，编写GUID分区表，其中包含我们指定类型的单一分区，并为我们的加密卷编写一个HFS文件系统。所有这些任务都通过命令行工具执行。

首先，打开Terminal应用程序。第一步是创建一个空磁盘映像，为我们的媒介提供存储空间，并作为磁盘使用。hdiutil命令行实用工具可以创建和操作磁盘映像文件。可以使用如下命令，创建一个25 Mib的空磁盘映像：

```
hdiutil create -megabytes 25 -layout NONE EncryptedImage.dmg
```

这会在当前工作目录中创建一个名为EncryptedImage.dmg的文件，由一个25 Mib的磁盘映像组成。-layout NONE选项表示我们并不在该磁盘映像上创建分区表。由于最终的磁盘映像并不包含分区映射和文件系统，所以无法装载。但可以使用以下命令，与该磁盘映像进行交互：

```
hdiutil attach -nomount EncryptedImage.dmg
```

了解存储设备如何在Mac OS X内核中实现，将非常有利于检查该命令在后台执行的操作。hdiutil attach命令将加载苹果提供的内核驱动程序，管理磁盘映像，这将派生自实现RAM磁盘的IOBlockStorageDevice超类。将构造一个存储驱动程序栈，其中包含一个表示整个磁盘镜像内容的IOMedia对象。该IOMedia对象有一个对应的IOMediaBSDClient对象，发布/dev目录中磁盘镜像的设备接口。这会创建一个块设备（如/dev/disk1）和一个字符设备（如/dev/rdisk1），通过这些设备可以访问该磁盘映像。装载磁盘映像时创建的块设备路径会输出到终端。

现在便可以通过其块设备接口读写磁盘映像了，因此下一步是在磁盘上创建一个GUID分区表。gpt命令是苹果在Mac OS X中提供的一个命令行工具，可以创建和操作磁盘的GUID分区表。可以使用以下命令，在空磁盘上创建一个GUID分区表。请确保将/dev/diskN路径替换为系统上装载的磁盘映像所对应的设备接口路径。

```
gpt create /dev/diskN
```

这会创建一个GUID分区表，但并不包含磁盘分区。我们希望在磁盘上创建一个分区，因此，接下来的命令便会将一个项插入到磁盘的GUID分区表中：

```
gpt add -t 8D7FD0BB-39A8-43C0-9432-F4E1A269F070 /dev/diskN
```

尽管gpt add命令可以指定分区长度和初始块偏移量，但如果未指定分区范围，则该实用程序默认会创建一个分区，从在磁盘上找到的第一个未使用的磁盘块范围开始。这对我们来说非常完美，因为这会创建一个填充整个磁盘的分区。-t选项用来指定所创建磁盘项的GUID类型。该创建的磁盘项为我们自定义的GUID类型，标识加密的HFS卷。通过将分区添加到磁盘，从而创建一个新的设备接口，表示该分区。分区块设备接口的路径会输出到终端，形式为/dev/diskNs1。

注意　BSD使用术语“分片”来表示磁盘分区。因此，disk2s1是指磁盘设备disk2的第一个分区（分片）。片数为IOMedia对象的位置值。在我们的分区规则驱动程序和加密过滤器规则中，我们会对创建的每个IOMedia对象调用setLocation()方法。提供的字符串用于生成设备接口的名称。

在内核层，在磁盘上创建分区表会导致加载苹果提供的IOGUIDPartitionScheme驱动程序。该驱动程序会响应我们添加的分区项，实例化一个IOMedia对象，表示该分区。该分区的IOMedia对象将有一个Content Hint属性，与我们赋给该分区的自定义GUID类型一样。这时，便可以加载加密过滤器驱动程序了。你会记得，我们的过滤器规则驱动程序会加载自定义内容线索GUID的IOMedia对象。如果创建的分区项没有指定的内容类型，那么我们的加密规则驱动程序将找不到可以匹配的IOMedia对象，也就无法测试我们的驱动程序。

加密过滤器规则驱动程序在加载时会创建一个新的IOMedia对象。进而会创建另外一个块设备，该块设备的接口名类似于diskNs1s1。该接口表示磁盘分区未加密的内容。在我们的示例加密过滤器驱动程序中，所创建IOMedia对象的Content Hint将被赋值为Apple_HFS，从而通知Mac OS X该媒介包含默认的HFS文件系统，并将未加密的卷加载为HFS文件系统。但在这个阶段，磁盘分区为空，并不包含任何文件系统。我们可以使用如下命令，在该卷上创建一个HFS文件系统：

```
newfs_hfs -v MyVolumeName /dev/diskNs1s1
```

-v选项允许指定卷名。在上面的示例中，我们将HFS卷命名为MyVolumeName。已经为该（未加密）卷编写了一个HFS文件系统，现在可以在Mac OS X桌面上装载该文件系统了。这可以通过以下命令实现：

```
hdiutil mountvol /dev/diskNs1s1
```

这会在桌面上显示一个新卷。由于我们的加密过滤器规则位于存储栈中，写入磁盘的任何文件在输入写入磁盘映像文件之前，都会由我们的异或加密修改。磁盘映像文件本身只是一个普通文件，因此，每个磁盘块的内容可以通过在任意一个Hex编辑器中打开.dmg文件进行检查。这使磁盘映像在调试、验证分区规则驱动程序或过滤器规则驱动程序操作是否正确方面，非常有用。

将分区表和文件系统写入到该磁盘之后，将来打开磁盘映像文件，就可以装载该磁盘映像。I/O Kit可以自动创建整个驱动程序存储栈，从该磁盘映像的传输驱动程序、GUID分区规则驱动程序、加密过滤器规则，到最后栈顶的HFS文件系统，整个创建过程都无需用户参与。

## 14.6　小结

- 包含文件系统的存储卷所提供的功能，通过一个包含多个驱动程序的栈实现，每个驱动程序可能由不同的厂商提供。存储驱动程序栈中每层的驱动程序负责执行特定功能。
- 栈的底层是传输驱动程序，直接与提供数据存储的硬件设备交互。

- 块存储驱动程序提供存储设备的一个抽象表示，将字节顺序组织到长度固定的块中，并提供其数据的随机访问。该块存储驱动程序位于驱动程序栈中的传输驱动程序上方。
- 分区驱动程序负责从磁盘读取分区表，并创建一个驱动程序对象，表示分区表中的每个逻辑卷。该分区驱动程序位于驱动程序栈中的块存储驱动程序上方。
- I/O Kit通过IOMedia驱动程序对象，表示逻辑卷。每个IOMedia对象都可以通过/dev目录中的接口，由用户空间进程访问。
- 通过I/O Kit，厂商可以将过滤器规则驱动程序插入到存储驱动程序栈中，截获所有对磁盘的读写请求。这可以用来实现RAID驱动程序，或对写入磁盘的数据进行加密。
- 文件系统驱动程序位于存储驱动程序栈的最顶层。尽管文件系统驱动程序通过I/O Kit存储驱动程序栈，间接进行读写操作，但它们实际上是Mac OS X的BSD层的一部分，而不是I/O Kit的一部分。

# 第15章 用户空间USB驱动程序

从用户角度来看，如果应用程序需要内核驱动程序，会大大降低用户的体验度。首先，安装驱动程序需要写入“扩展”字典，而这需要拥有管理权限。因此，用户需要运行安装程序，并输入管理账户密码，然后在使用该驱动程序之前可能还需要重启。而如果应用程序不需要内核驱动程序，那么安装过程就像从Mac App Store上下载应用程序一样简单。

在某些情况下，对于需要内核驱动程序的应用程序，通常无须开发者编写任何在内核中运行的代码。而该驱动程序执行的操作可以由应用程序完成。这种方案的优点是无须安装任何驱动程序，用户因此也不需要使用管理权限来安装应用程序。将驱动程序代码从内核移入应用程序的另一个优点是，代码中存在的错误在最坏的情况下，会使应用程序崩溃，但它们不会引起内核恐慌，使整个系统崩溃。

并不是所有的硬件设备都可以通过用户空间驱动程序控制，例如，包含内存映射地址的设备就需要内核驱动程序。类似地，生成中断的设备也需要内核驱动程序，因为只有内核代码才可以在主中断级别执行。这意味着所有的PCI和Thunderbolt设备都需要内核驱动程序的支持。而基于USB和FireWire的硬件设备是用户空间驱动程序的最佳候选。本章将编写一个仅存在于用户空间中的USB设备驱动程序，而无须编写内核驱动程序。

并不是所有的USB和FireWire设备都适合用户空间驱动程序。特别是需要在/dev目录中创建设备接口文件的驱动程序（如串行端口驱动程序或存储设备），需要内核驱动程序。另外，对于由多个应用程序同时使用或需要由系统使用的设备，如音频驱动程序，则需要为其编写内核驱动程序。幸好这些情况只是例外，而大多数需要自定义驱动程序的USB设备可以由用户空间驱动程序控制。

## 15.1 背景

本书前几章已经讲述了Mac OS X的体系结构，特别讲述了现代操作系统只支持硬件从内核直接访问，你可能很想知道这如何与本章内容保持一致。

在内部，用户空间驱动程序确实需要一个内核驱动程序，但这由相同的`IOUSBDevice`和`IOUSBInterface`对象提供，用来与USB设备交互，如果开发者已经确定为该硬件编写内核驱动程序。为了使这些对象能够用于用户空间应用程序，`IOUSBFamily`会为内核中创建的每个`IOUSBDevice`及`IOUSBInterface`实例发布一个用户客户端。

为这些类创建的用户客户端完全通用，而且其作用仅仅是将IOUSBDevice和IOUSBInterface类的方法向用户空间公开。例如，IOUSBDevice类的用户客户端包含获取/设置活动设备的配置、执行设备请求及迭代设备接口的方法。IOUSBInterface类的用户客户端包含读写指定端点数据的方法。

应用程序无须从用户客户端类直接调用这些方法，而是由I/O Kit框架提供高级API，来控制硬件。该API称为IOUSBLib。用户空间USB驱动程序的分层结构，如图15-1所示。开发者编写的自定义代码只存在于应用程序层，该层下方是苹果公司提供的一些常见库，是Mac OS X的一部分。

尽管无须理解IOUSBLib的实现方式，但了解相关知识会帮助你更好地理解应用程序从用户空间查找及与USB设备交互的各个步骤。

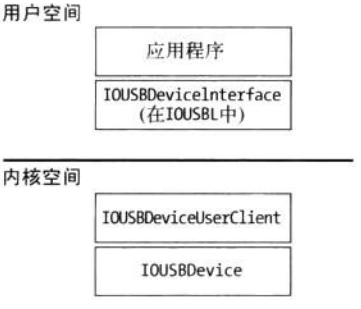



图15-1 用户空间驱动程序访问其USB硬件所通过的层

## 15.2 IOUSBLib 框架

用于编写USB设备用户空间驱动程序的库是IOUSBLib，它是I/O Kit框架的一部分（也就是说，用户空间应用程序与其本身定义的用户客户端通信时，也包括该框架）。

用户空间驱动程序需要执行的首要任务是，等待感兴趣的特定USB设备插入。由于USB设备能够随时在计算机上插拔，所以无法保证应用程序启动时，它感兴趣的设备会存在。因此，应用程序安装一个通知回调，等待它控制的USB设备插入，这是一个很好的做法。

从第5章看到应用程序会创建一个匹配字典，查找指定内核驱动程序的每个实例。实现用户空间USB驱动程序的应用程序也采用相同的方案，来确定它控制的设备。如第5章一样，我们首先创建一个字典，指定应用程序感兴趣的驱动程序对象的匹配类名。对于USB设备，实现方式如下所示：

```
matchingDictionary = IOServiceMatching(kIOUSBDeviceClassName); // "IOUSBDevice"
```

该匹配字典对大部分应用程序来说都极为通用，因为它会与连接到计算机上的所有USB设备进行匹配，包括键盘和鼠标。而应用程序通常只对特定的USB设备感兴趣，因此诸如此类的匹配字典并不合适。我们可以通过包含应用程序所支持USB设备的特定产品ID和厂商ID，来缩小匹配字典满足的设备列表。

为指定厂商ID和产品ID的USB设备创建匹配字典的示例函数，如代码清单15-1所示。

**代码清单15-1　创建一个USB匹配字典**

```
#include <IOKit/IOKitLib.h>
#include <IOKit/usb/IOUSBLib.h>
#include <CoreFoundation/CoreFoundation.h>

CFDictionaryRef         MyCreateUSBMatchingDictionary (SInt32 idVendor, SInt32 idProduct)
{
        CFMutableDictionaryRef  matchingDictionary = NULL;
        CFNumberRef             numberRef;

        //为IOUSBDevice创建一个匹配字典
        matchingDictionary = IOServiceMatching(kIOUSBDeviceClassName);
        if (matchingDictionary == NULL)
                goto bail;

        //将USB厂商ID添加到匹配字典
        numberRef = CFNumberCreate(kCFAllocatorDefault, kCFNumberSInt32Type, &idVendor);
        if (numberRef == NULL)
                goto bail;
        CFDictionaryAddValue(matchingDictionary, CFSTR(kUSBVendorID), numberRef);
        CFRelease(numberRef);

        //将USB产品ID添加到匹配字典
        numberRef = CFNumberCreate(kCFAllocatorDefault, kCFNumberSInt32Type, &idProduct);
        if (numberRef == NULL)
                goto bail;
        CFDictionaryAddValue(matchingDictionary, CFSTR(kUSBProductID), numberRef);
        CFRelease(numberRef);

        //成功—将该字典返回给调用者
        return matchingDictionary;

bail:
        //失败—释放资源并返回NULL
        if (matchingDictionary != NULL)
                CFRelease(matchingDictionary);

        return NULL;
}
```

**注意**　应用程序可以通过向基于内核的USB驱动程序（参见第8章）属性列表中添加一些键，来缩小匹配字典的范围。例如，应用程序可能包括idVendor、idProduct、bcdDevice、bDeviceSubClass或bDeviceProtocol等键。

对于复合USB设备，驱动程序会优先匹配特定接口，而不是整个USB设备。用户空间驱动程序可以通过为IOUSBInterface对象的特定实例创建一个匹配字典，来完成该工作。这是因为I/O Kit为内核中创建的IOUSBDevice类和IOUSBInterface类的每个实例都定义一个用户客户端，使两个类都可以用于用户进程。

创建匹配字典之后，应用程序可以使用该字典迭代所有与其规范匹配的内核对象，并在发现匹配的内核对象时，安装一个回调，接收通知。这在第5章已描述。迭代指定匹配字典所描述的所有内核设备的示例函数，如代码清单15-2所示。

**代码清单15-2 查找和迭代满足匹配字典的设备**

```
void    MyFindMatchingDevices (CFDictionaryRef matchingDictionary)
{
        io_iterator_t           iterator = 0;
        io_service_t            usbDeviceRef;
        kern_return_t           err;

        //查找所有匹配该字典的内核对象
        err = IOServiceGetMatchingServices(kIOMasterPortDefault, matchingDictionary,
                        &iterator);
        if (err == 0)
        {
                //迭代所有匹配的内核对象
                while ((usbDeviceRef = IOIteratorNext(iterator)) != 0)
                {
                        IOUSBDeviceInterface300**       usbDevice;

                        //为该设备实例创建一个驱动程序
                        usbDevice = MyStartDriver(usbDeviceRef);
                        IOObjectRelease(usbDeviceRef);
                }

                IOObjectRelease(iterator);
        }
}
```

如代码清单15-2所示，匹配字典的迭代器会返回多个io_service_t对象，每个对象表示一个内核对象。io_service_t对象是内核中IOUSBDevice或IOUSBInterface对象的用户空间表示。类似于I/O Kit类，IOUSBDevice和IOUSBInterface均采用C++类实现。这些类包含一个公共接口，通过它定义的接口，内核驱动程序与USB硬件设备交互。

IOUSBLib接口通过C++类实现，该C++类包装表示IOUSBDevice或IOUSBInterface的io_service_t对象。IOUSBDevice类的用户空间等价类是IOUSBDeviceInterface，而IOUSBInterface类的用户空间等价类是IOUSBInterfaceInterface。这些类的声明可以在头文件<IOKit/usb/IOUSBLib.h>中找到。代码清单15-3的代码示例演示了用户空间应用程序如何从io_service_t实例化IOUSBDe-viceInterface类。

**代码清单15-3 从io_service_t实例化IOUSBDeviceInterface对象**

```
IOUSBDeviceInterface300**       MyStartDriver (io_service_t usbDeviceRef)
{
        SInt32                          score;
        IOCFPlugInInterface**           plugin;
        IOUSBDeviceInterface300**       usbDevice = NULL;
        kern_return_t                   err;
```

```
        err = IOCreatePlugInInterfaceForService(usbDeviceRef, kIOUSBDeviceUserClientTypeID,
                                                kIOCFPlugInInterfaceID, &plugin, &score);
        if (err == 0)
        {
                err = (*plugin)->QueryInterface(plugin,
                        CFUUIDGetUUIDBytes(kIOUSBDeviceInterfaceID300),
                        (LPVOID)&usbDevice);
                IODestroyPlugInInterface(plugin);
        }

        return usbDevice;
}
```

如果应用程序已经创建了一个指定了IOUSBInterface的匹配字典，它接收的每个io_service_t值表示一个内核IOUSBInterface对象，而不是IOUSBDevice对象。在这种情况下，应用程序为表示内核对象而实例化的用户空间类是IOUSBInterfaceInterface。这只需要改变传递给代码清单15-3中调用的两个函数的参数。对IOCreatePlugInInterfaceForService的调用，如下所示。

```
err = IOCreatePlugInInterfaceForService(usbDeviceRef, kIOUSBInterfaceUserClientTypeID,
                                        kIOCFPlugInInterfaceID, &plugin, &score);
```

类似地，调用返回的plugin对象的QueryInterface会使用如下参数：

```
IOUSBInterfaceInterface300**            usbInterface = NULL;
err = (*plugin)->QueryInterface(plugin,
                        CFUUIDGetUUIDBytes(kIOUSBInterfaceInterfaceID300),
                        (LPVOID)&usbInterface);
```

不管应用程序是实例化IOUSBDeviceInterface还是IOUSBInterfaceInterface，代码结构均相同。在两种情况下，第一步都是调用IOCreatePlugInInterfaceForService()函数，返回一个IOCFPlugInInterface类型的对象。该对象作为实例化用户空间I/O Kit类的工厂，在该操作完成之后，便毫无用处。实际上，如代码清单15-3所示，该对象在IOUSBDeviceInterface对象创建之后，会立即调用IODestroyPlugInInterface()进行释放。

IOCFPlugInInterface类包含一个QueryInterface()方法，应用程序用该方法接收IOUSBDeviceInterface或IOUSBInterfaceInterface对象的指针。IOUSBLib提供了一种版本化类的方法。这使Mac OS X的未来版本可以扩展类，如IOUSBDeviceInterface，这样一来，维护面向旧版本类的应用程序的后向兼容性时，可以使之包含额外功能。

应用程序请求接口（如IOUSBDeviceInterface）时，还必须指定希望接收的类版本。类的版本是类名的一部分，例如，IOUSBDeviceInterface300接受包括IOUSBFamily 3.0.0版的IOUSBDeviceInterface版本。这包含在Mac OS X 10.5中。完整的类名及其版本，以及支持该类的操作系统的最低版本等信息都在IOUSBLib.h头文件中。

---

**提示**　通常，应该使用的IOUSBDeviceInterface和IOUSBInterfaceInterface类的版本将与你的应用程序支持的Mac OS X的最低版本紧密绑定。例如，Mac OS X 10.5及其之后版本的应用程序应该使用IOUSBDeviceInterface300和IOUSBInterfaceInterface300。

---

需要花费一定时间才能习惯用户客户端类是因为，IOUSBLib返回一个指向对象指针的指针。这意味着，在调用该对象的方法之前，持有该接口的变量需要一个额外的反引用。IOUSBLib类的另一个特性是，每个方法都需要引用该对象，使之作为第一个参数传递。例如，考虑一下由IOCFPlugInInterface类实现的QueryInterface()方法，虽然你希望使用下列代码行调用该方法：

```
plugin->QueryInterface(parameters); // INCORRECT
```

但因为plugin变量是IOCFPlugInInterface**类型，是一个指向指针的指针，所以该方法实际上必须使用以下结构调用：

```
(*plugin)->QueryInterface(plugin, parameters);
```

**注意** 如果熟悉微软的COM（Component Object Model，组件对象模型），你会马上认出QueryInterface()方法。所有的IOUSBLib类都基于COM编程模型，派生自基类IUnknown。这种设计对使用IOUSBLib应用程序的最大影响是，所有的IOUSBLib对象都是引用计数，通过调用AddRef()方法获得，并通过Release()方法释放。

## 15.3 处理异步操作

在随后对IOUSBLib类的讨论中我们将会看到，许多方法执行的操作都异步完成。所有这些异步方法都包含两个参数，一个是指向回调函数的指针，另一个是refcon参数，它可以使应用程序将任意上下文值传递给其回调。该回调函数的签名如下所示：

```
void    MyCallbackFunction (void* refcon, IOReturn result, void* arg0);
```

第一个参数refcon是应用程序的任意上下文参数。第二个参数报告操作的整体结果，kIOReturnSuccess值表示该操作成功完成。最后一个参数argo由IOUSBLib提供，取决于执行操作的类型。本章讲述异步方法时，还将描述IOUSBLib通过argo参数传递给回调函数的值。

内核驱动程序使用工作环，将其完成例程与其他驱动程序代码同步，与之类似，用户空间应用程序可以使用运行回路，将IOUSBLib的完成回调与其他代码同步。

首先，应用程序必须为执行异步操作的IOUSBLib类创建一个运行回路源。IOUSBLib类包含创建运行回路源或Mach端口以接收异步通知的方法，而大多数应用程序只需要处理运行回路源。

重要的是要注意IOUSBLib类提供前缀是Create和Get的方法（如CreateDeviceAsyncEventSource和GetDeviceAsyncEventSource）。而Get方法仅返回前面通过Create方法实例化的对象。创建、安装运行回路源，以接收IOUSBDeviceInterface类异步通知的示例函数，如以下代码片段所示。

```
IOReturn   InstallRunLoopSourceForUSBDevice (IOUSBDeviceInterface300** usbDevice)
{
    CFRunLoopSourceRef          runLoopSource;
    IOReturn                    error;

    error = (*usbDevice)->CreateDeviceAsyncEventSource(usbDevice, &runLoopSource);
```

```
    if (error == kIOReturnSuccess)
        CFRunLoopAddSource(CFRunLoopGetCurrent(), runLoopSource, kCFRunLoopDefaultMode);

    return error;
}
```

应用程序可以自由选择在其希望的特定线程运行回路上安装运行回路源，包括应用程序主线程的运行回路。对象所有权规则与所有的核心基础函数遵循相同的惯例：如果应用程序获得一个来自Create方法的对象，则它拥有该对象，并负责释放它。如果应用程序获得一个来自Get方法的对象，则它并不拥有该对象的引用，因此，如果该应用程序希望持有该对象，那么它首先必须显式持有该对象。

## 15.4　IOUSBDeviceInterface 类

IOUSBDeviceInterface类是一个用户空间类，提供的功能与内核驱动程序使用的IOUSBDevice类相同。值得注意的是，尽管用户客户端类提供的功能与其内核对应类相似，但它通过一种不同的方法集实现其功能。因此，并不能将处理IOUSBDevice类的内核USB驱动程序简单地引入到用户空间USB驱动程序。

获得用户空间IOUSBDeviceInterface的引用之后，应用程序需要做的第一个工作是配置USB硬件。应用程序执行的步骤非常接近第8章中讲述的内核驱动程序的操作。首先，应用程序需要获得USB硬件的独占访问，防止硬件配置被另一个驱动程序改变，另一个驱动程序可能是另一个用户空间驱动程序，也可能是内核驱动程序。这通过调用IOUSBDeviceInterface方法USBDeviceOpen()来实现，如下所示：

```
error = (*usbDevice)->USBDeviceOpen(usbDevice);
```

如果该方法返回的错误代码是kIOReturnSuccess，则表示该应用程序已经获得独占访问，从而配置硬件。如果另一个应用程序或驱动程序已经获得该硬件的独占访问，那么调用USBDeviceOpen()便会失败，错误代码是kIOReturnExclusiveAccess，应用程序此时应该放弃所有对该设备的进一步访问，并向用户报告错误。

当应用程序已经使用完该设备，则应该调用IOUSBDeviceInterface方法USBDeviceClose()，释放对硬件的独占访问，如下所示：

```
error = (*usbDevice)->USBDeviceClose(usbDevice);
```

以下是IOUSBDeviceInterface类提供的方法汇总，这些方法可以访问USB设备描述符中包含的信息。本章介绍的是IOUSBDeviceInterface300类，因此，下面一些方法在IOUSBDeviceInterface类的早期版本中并不存在。

- GetDeviceClass、GetDeviceSubClass和GetDeviceProtocol：分别从USB设备描述符返回设备类（bDeviceClass）、子类（bDeviceSubClass）及协议（bDeviceProtocol）。同时，这三个值基于USB规范中定义的值，定义设备功能。
- GetDeviceVendor：返回设备的USB厂商ID（idVendor）。

- GetDeviceProduct：返回设备的USB产品ID（idProduct）。
- GetDeviceReleaseNumber：返回设备的版本号。
- USBGetManufactureStringIndex：返回设备厂商名称（iManufacturer）字符串的索引。为了从设备读取实际的字符串，应用程序必须发送标准设备请求“获取描述符”，来读取设备字符串表中的一项。
- USBGetProductStringIndex：返回设备产品名称（iProduct）字符串的索引。
- USBGetSerialNumberStringIndex：返回设备序列号（iSerialNumber）字符串的索引。
- GetNumberOfConfigurations：返回设备在当前速度下支持的配置编号（bNumConfigurations）。

应用程序使用下列方法，读取与USB设备当前状态相关的动态属性。

- GetDeviceAddress：返回USB设备的地址，这对其连接的总线具有唯一性。
- GetDeviceSpeed：返回设备的速度。可能的值有：kUSBDeviceSpeedLow、kUSBDeviceSpeedFull或kUSBDeviceSpeedHigh。
- GetLocationID：返回一个32位值，基于设备连接的USB集线器和端口，该值唯一标识系统上的USB设备。计算机重启时，位置ID并不会发生变化，但当设备连接到另一个集线器或端口时，会发生改变。因此，如果USB设备提供一个序列号字符串，跟踪设备重启或断开更为可取。
- GetBusFrameNumber：返回设备连接的USB总线的当前帧号。该函数还返回系统主机时间，对应内核驱动程序处理该请求的时间。系统时间可能位于返回的USB帧中的任意位置。
- GetBusFrameNumberWithTime：返回设备连接的USB总线的当前帧号，还返回对应帧开始的系统主机时间。该方法在IOUSBDeviceInterface类的后续版本中引入，替代了GetBusFrameNumber()方法。
- GetBusMicroFrameNumber：返回设备连接的USB总线的当前微帧号。该函数还返回系统主机时间，对应内核驱动程序处理该请求的时间（该方法与GetBusFrameNumber()非常相似）。

应用程序采用下列方法，重置USB设备。

- ResetDevice：重置USB设备，使其恢复至未设置的状态。
- USBDeviceReEnumerate：根据布尔参数值，指示设备连接的集线器重置设备所连接到的端口。这就是说断开设备与USB端口的连接，并重新连接。
- USBDeviceSuspend：不管该方法的名称如何，根据布尔参数值，它或者暂停，或者恢复USB设备所连接到的端口。如果该方法暂停设备，则放弃任何未完成的传输。
- USBDeviceAbortPipeZero：放弃控制端点上未完成的传输。
- IOUSBDeviceInterface类提供了下列方法，使应用程序可以将控制请求发送至端点0上的设备。
- DeviceRequest是一个异步方法，在设备请求完成之后返回。该设备请求与内核USB驱动程序采用相同的IOUSBDevRequest结构来描述。
- DeviceRequestTO是一个同步方法，有两个超时值，单位是微秒。该方法在设备请求完成，或指定的超时时间已经过去之后返回（无论哪一种情况先发生）。调用者提供了两个超时

值。设备在处理请求时，会停止发送或接收数据，因此一个超时值指定从最后的数据传输开始到放弃控制请求之前所等待的最长时间。另一个超时值指定控制请求从开始到结束的最长完成时间。如果发生任一指定的超时情况，便会放弃控制请求。

- DeviceRequestAsync是DeviceRequest的一个同步等价方法。该方法有一个回调函数，该请求完成之后，I/O Kit用来通知调用者。重要的是要注意，该方法的返回值kIOReturnSuccess并不表示该请求成功完成，而是表示操作成功开始。该操作的实际结果通过回调函数返回。传递给完成回调arg0参数的值保存从设备读取或写入设备的字节数。

代码清单15-4是使用DeviceRequest方法从USB字符串表读取厂商字符串的示例函数。该函数首先调用USBGetManufactureStringIndex获取厂商字符串的索引。接下来，设置设备请求结构体。bmRequestType字段指定该请求是数据读取（kUSBIn），是一个标准请求（而不是设备类定义或特定厂商的请求），该请求的接收者是设备（而不是接口或端点）。根据USB规范，该字符串表仅被看做是另一个描述符表，所以发送给该设备来读取字符串的控制请求是kUSBRqGetDescriptor请求。wValue字段表示我们正在读取字符串描述符，该字段还包含读取的字符串索引。

**代码清单15-4　IOUSBDeviceInterface类执行设备请求的演示函数**

```
#include <IOKit/usb/IOUSBLib.h>
#include <IOKit/usb/USBSpec.h>
#include <CoreFoundation/CoreFoundation.h>

IOReturn                PrintDeviceManufacturer (IOUSBDeviceInterface300** usbDevice)
{
        UInt8                   stringIndex;
        IOUSBDevRequest         devRequest;
        UInt8                   buffer[256];
        CFStringRef             manufString;
        IOReturn                error;

        //获取字符串表中的厂商索引
        error = (*usbDevice)->USBGetManufacturerStringIndex(usbDevice, &stringIndex);
        if (error != kIOReturnSuccess)
                return error;

        //执行设备请求，读取字符串描述符
        devRequest.bmRequestType = USBmakebmRequestType(kUSBIn, kUSBStandard, kUSBDevice);
        devRequest.bRequest = kUSBRqGetDescriptor;
        devRequest.wValue = (kUSBStringDesc << 8) | stringIndex;
        devRequest.wIndex = 0x409;              //语言设置，指定为US英语
        devRequest.wLength = sizeof(buffer);
        devRequest.pData = &buffer[0];
        bzero(&buffer[0], sizeof(buffer));
        //
        error = (*usbDevice)->DeviceRequest(usbDevice, &devRequest);
        if (error != kIOReturnSuccess)
                return error;

        //创建返回数据的CFString表示
        int             strLength;
```

```
        strLength = buffer[0] - 2;                  //第一个字节是长度（单位是字节）
        manufString = CFStringCreateWithBytes(kCFAllocatorDefault, &buffer[2], strLength,
                                              kCFStringEncodingUTF16LE, false);
        //输出厂商字符串
        CFShow(manufString);
        CFRelease(manufString);

        return error;
}
```

最后，如果成功从设备读取字符串数据，该函数会从返回的数据创建一个CFStringRef，采用UTF-16小端序编码。

应用程序使用下列方法，检查和设置设备配置，以及迭代设备接口。这些方法通常由应用程序调用来初始化首次检测到的USB设备。

- GetConfigurationDescriptionPtr：返回一个指向指定配置描述符的指针，请注意，尽管调用者接收IOUSBConfigurationDescriptorPtr结构体指针，但该缓冲区属于IOUSBDeviceInterface对象，并且不由该调用者释放。
- GetConfiguration：返回设备的活动配置编号。请注意，这并不是配置的索引，而是活动配置描述符的bConfigurationValue值。
- SetConfiguration：设置设备的活动配置。通过从所需的配置描述符传递bConfiguration-Value来指定该配置。
- CreateInterfaceIterator：创建设备接口的迭代器。与其内核等价方法类似，调用者提供一个IOUSBFindInterfaceRequest结构体，来指定返回接口必须匹配的属性。

看了IOUSBDeviceInterface类提供的功能之后，现在开始考虑一下应用程序初始化装载到系统上的新USB设备时，所执行的步骤。初始化函数示例，如代码清单15-5所示。

**代码清单15-5 初始化时配置USB设备的示例函数**

```
IOReturn        MyConfigureDevice (IOUSBDeviceInterface300** usbDevice)
{
        UInt8                               numConfigurations;
        IOUSBConfigurationDescriptorPtr     configDesc;
        IOUSBFindInterfaceRequest           interfaceRequest;
        io_iterator_t                       interfaceIterator;
        io_service_t                        usbInterfaceRef;
        IOReturn                            error;

        //获取设备的配置计数
        error = (*usbDevice)->GetNumberOfConfigurations(usbDevice, &numConfigurations);
        if (error != kIOReturnSuccess)
                return error;
        //确保设备至少有一个配置
        if (numConfigurations == 0)
                return kIOReturnError;

        //读取第一个配置的描述符
        error = (*usbDevice)->GetConfigurationDescriptorPtr(usbDevice, 0, &configDesc);
```

```
        if (error != kIOReturnSuccess)
                return error;

        //将第一个配置设置为活动配置
        error = (*usbDevice)->SetConfiguration(usbDevice, configDesc->bConfigurationValue);
        if (error != kIOReturnSuccess)
                return error;

        //创建一个迭代器，迭代活动配置中的所有接口
        interfaceRequest.bInterfaceClass = kIOUSBFindInterfaceDontCare;
        interfaceRequest.bInterfaceSubClass = kIOUSBFindInterfaceDontCare;
        interfaceRequest.bInterfaceProtocol = kIOUSBFindInterfaceDontCare;
        interfaceRequest.bAlternateSetting = kIOUSBFindInterfaceDontCare;

        error = (*usbDevice)->CreateInterfaceIterator(usbDevice, &interfaceRequest,
                        &interfaceIterator);
        if (error != kIOReturnSuccess)
                return error;

        //迭代所有的接口
        while ((usbInterfaceRef = IOIteratorNext(interfaceIterator)) != 0)
        {
                MySetupInterface(usbInterfaceRef);
                IOObjectRelease(usbInterfaceRef);
        }
        IOObjectRelease(interfaceIterator);

        return kIOReturnSuccess;
}
```

代码清单15-5中的代码首先将设备的活动配置设置为一个已知配置，在该示例中是设备的第一个配置。这一步是必需的，因为在应用程序启动之前，设备可能已经由另一个应用程序使用，所以此时设备可能处于未知状态。

接下来，应用程序迭代活动配置中的所有接口。如果我们对特定的接口感兴趣，则可以通过指定所需的类、子类、协议或接口的备用设置，来缩小迭代器返回的接口列表。获得一个USB接口对象尤为重要，因为应用程序访问设备端点（而不是控制端点）的唯一方式是通过IOUSBInterfaceInterface类。

## 15.5　IOUSBInterfaceInterface 类

IOUSBInterfaceInterface类是一个用户空间类，提供的功能与内核驱动程序使用的IOUSBInterface类相同。应用程序可以通过迭代设备接口，获取IOUSBInterfaceInterface类的引用，如代码清单15-5所示，或者通过创建指定服务名称kIOUSBInterfaceClassName的匹配字典，直接获取IOUSBInterfaceInterface对象。

USB接口包含一个或多个端点，可以使数据写入设备，或从设备读出。用户空间USB驱动程序可以使用的端点类型没有限制，所有的端点类型（包括批量、同步和中断端点）都可用于用户

空间驱动程序。用户空间驱动程序可以达到与内核驱动程序相类似的数据带宽，这意味着，即使需要大量数据传输的应用程序也可以在用户空间中编写。

无论应用程序是使用IOUSBDeviceInterface迭代USB接口，还是使用匹配字典创建一个迭代器，直接获得一个USB接口，该应用程序都会接收一个io_service_t，提供内核中底层IOUSBInterface对象的用户空间表示。代码清单15-6演示了如何通过io_service_t对象创建一个IOUSBInterface-Interface对象。

**代码清单15-6 从io_service_t实例化一个IOUSBInterfaceInterface对象**

```
IOUSBInterfaceInterface300**     MyCreateInterfaceClass (io_service_t usbInterfaceRef)
{
        SInt32                          score;
        IOCFPlugInInterface**           plugin;
        IOUSBInterfaceInterface300**    usbInterface = NULL;
        kern_return_t                   err;

        err = IOCreatePlugInInterfaceForService(usbInterfaceRef,
                    kIOUSBInterfaceUserClientTypeID,
                    kIOCFPlugInInterfaceID, &plugin, &score);
        if (err == 0)
        {
                err = (*plugin)->QueryInterface(plugin,
                        CFUUIDGetUUIDBytes(kIOUSBInterfaceInterfaceID300),
                        (LPVOID)&usbInterface);
                IODestroyPlugInInterface(plugin);
        }

        return usbInterface;
}
```

类似于USB设备类，应用程序在向/从USB接口上的端点传输数据之前，必须独占访问USB接口对象。这通过调用IOUSBInterfaceInterface方法USBInterfaceOpen()实现，如下所示：

```
error = (*usbInterface)->USBInterfaceOpen(usbInterface);
```

应用程序用完接口时，应该调用IOUSBInterfaceInterface方法USBInterfaceClose()，释放独占访问，如下所示：

```
error = (*usbInterface)->USBInterfaceClose(usbInterface);
```

IOUSBInterfaceInterface类提供的方法分为两类：一类是获得/设置USB设备或接口的属性，一类与向/从接口上的端点传输数据相关。接下来将总结IOUSBInterfaceInterface类提供的各个方法。本章介绍的是IOUSBInterfaceInterface300类，因此，下面的一些方法在IOUSBInterface-Interface类的早期版本中并不存在。



## 15.5.1 属性方法

IOUSBInterfaceInterface类包含获取/设置与接口和USB设备相关属性的方法。尽管一些方法可能看起来与前面讲述的IOUSBDeviceInterface类提供的功能重复，但这是有意为之，因为它支

持与USB接口类相匹配而与USB设备类不匹配的应用程序访问公用设备功能。

下列方法虽然位于IOUSBInterfaceInterface类中，但却与USB设备相关。

- GetDeviceVendor：返回USB设备的USB厂商ID。
- GetDeviceProduct：返回USB设备的USB产品ID。
- GetDeviceReleaseNumber：返回USB设备的设备版本号。
- GetLocationID：返回一个32位值，根据设备连接到USB集线器和端口，唯一标识系统上的USB设备。
- GetDevice：返回一个对应内核IOUSBDevice对象的io_service_t。应用程序可以从该对象实例化一个表示USB设备的IOUSBDeviceInterface。
- GetBusFrameNumber：返回设备连接的USB集线器的当前帧号。
- GetBusFrameNumberWithTime：返回设备所连接到的USB集线器的当前帧号，还返回对应帧开始的系统主机时间。
- GetBusMicroFrameNumber：返回设备连接的USB总线的当前微帧号。
- GetFrameListTime：执行的功能与IOUSBDeviceInterface方法GetDeviceSpeed相类似，尽管返回的设备速度为它当前速度下每个USB帧的微秒数。全速设备将返回kUSBFullSpeedMicrosecondsInFrame（1000微秒），而高速设备将返回kUSBHighSpeedMicrosecondsInFrame（125微秒）。

下列方法用于获取/设置接口的属性。

- GetInterfaceClass、GetInterfaceSubClass和GetInterfaceProtocol：从USB接口描述符返回接口类（bInterfaceClass）、子类（bInterfaceSubClass）和协议（bInterfaceProtocol）。同时，这三个值基于USB规范中定义的值，定义接口函数。
- GetConfigurationValue：标识包含该接口的设备配置。返回值为活动配置描述符的bConfigurationValue。
- GetInterfaceNumber：返回活动配置中该接口从0开始的索引（bInterfaceNumber）。
- GetAlternateSetting：返回该接口的活动备用设置（bAlternateSetting）。
- SetAlternateInterface：设置该接口的活动备用设置。通过传递所需接口描述符的bAlternateSetting值，指定备用设置。
- USBInterfaceGetStringIndex：返回接口描述符的字符串索引，取自接口描述符的iInterface字段。

### 15.5.2　端点数据传输方法

一个USB接口能包含一个或多个端点，数据可以从该端点进行读取或写入。不同于内核USB驱动程序，IOUSBLib并不包含表示端点管道的用户空间对象，而所有的数据传输都要通过IOUSBInterfaceInterface类进行。应用程序可以使用下列方法，确定接口上的端点。

- GetNumEndpoints：返回接口提供的端点数。

- GetEndpointProperties：返回指定端点的类型（bDescriptorType）、最大分组长度（wMaxPacketSize）和轮询时间间隔（bInterval）。该端点由3个值指定：所在的备用接口设置、端点编号和端点传输方向。
- GetPipeProperties：利用该方法应用程序可以指定端点从0开始的索引，包括GetNumEndpoints的返回值，而GetEndpointPorperties需要调用者知道其感兴趣的端点编号及方向。索引0的端点对应默认控制端点。该方法返回的信息几乎包含端点描述符的所有数据，其中包括端点编号、类型、最大分组长度和轮询时间间隔（若为中断或同步端点）。
- GetPipeStatus：可以用来确定指定管道是否停顿。该方法在管道停顿时，返回kIOUSBPipeStalled，否则返回kIOReturnSuccess。如果调用者指定了一个无效的管道索引，则会返回kIOUSBUnknownPipeErr。
- AbortPipe：放弃指定管道索引的未完成传输。任何放弃的操作都会完成，其结果是kIOReturnAborted。
- ResetPipe和ClearPipeStall：尽管这两个方法相同，但优先使用ClearPipeStall。每个方法都可以由应用程序调用来重置停顿的端点。清除端点暂停位，并重置数据触发位。
- ClearPipeStallBothEnds：等同于ClearPipeStall，但区别是除了清除暂停位和重置主机端的数据触发，还清除设备上的暂停位，并重置设备的数据触发。主机和设备都被重置，这能够确保数据向/从该端点进行下一次传输时，不会丢失。

为了向/从端点传输数据，应用程序必须确定它想要访问的端点所对应的管道索引。尽管端点地址是一个常量，但由于它取自设备提供的描述符，所以管道索引由I/O Kit分配。因此，确定某端点管道索引的唯一方式是，枚举USB接口中包含的每个管道。示例实现如代码清单15-7所示。

**代码清单15-7 查找批量输出端点的管道引用函数**

```
IOReturn        MyFindBulkOutEndpoint (IOUSBInterfaceInterface300** usbInterface,
                                       UInt8* pipeRef)
{
        UInt8           numEndpoints;
        UInt8           i;
        IOReturn        error;

        //确定该接口中的端点数
        error = (*usbInterface)->GetNumEndpoints(usbInterface, &numEndpoints);
        if (error != kIOReturnSuccess)
                return error;

        //迭代该接口中所有的端点（跳过端点0，即控制端点）
        for (i = 1; i <= numEndpoints; i++)
        {
                UInt8           direction, number, transferType;
                UInt16          maxPacketSize;
                UInt8           interval;

                error = (*usbInterface)->GetPipeProperties(usbInterface, i, &direction,
                                                           &number,
```

```
                                                &transferType, &maxPacketSize, &interval);
            if (error != kIOReturnSuccess)
                    continue;

            //如果找到一个批量输出端点，则将其管道索引返回给调用者
            if ((transferType == kUSBBulk) && (direction == kUSBOut))
            {
                    *pipeRef = i;
                    return kIOReturnSuccess;
            }
        }

        return kIOReturnNotFound;
    }
```

确定了管道索引（被IOUSBLib中的方法称为管道引用）之后，应用程序可以通过该管道读/写数据。应用程序使用哪一个方法来管理管道，取决于管道所在端点的类型。根据管道连接的是控制端点、批量端点、中断端点，还是同步端点，IOUSBInterfaceInterface类使用的方法也不同。

IOUSBInterfaceInterface类提供了一些方法，向/从端点传输数据。根据调用的方法不同，操作可能同步完成，也可能异步完成。其他的方法允许调用者提供一个超时值，指定该请求被放弃之前可以使用最长时间完成操作。最后，还存在一些异步完成的方法，允许调用者指定一个超时值。

IOUSBLib的惯例是方法同步完成，除非该方法的名称中包含Async。除了传递给其对应同步方法的参数之外，异步方法还包含两个额外的参数：一个回调函数，操作完成时运行；一个任意类型的指针，可以将上下文信息传递给回调函数。

调用者指定超时的方法是在方法名称中包含字母TO。超时值由传递给该方法的两个额外参数表示。该设备在处理请求时，会停止发送或接收数据，因此，超时值是指从最后的数据传输开始，到操作放弃之前所等待的最长时间。第二个超时值指定完成操作从开始到结束所需的最长时间。满足任意一个超时条件的情况发生时，操作都会被放弃。

IOUSBInterfaceInterface类提供了下列方法来执行控制传输。这些方法与IOUSBDeviceInterface类提供的方法类似，不同的是每个方法都有一个额外参数——管道引用。如果传递给管道引用0值，那么该控制请求将发送给默认控制端点，即端点0。

- ControlRequest：同步执行控制请求。该方法使用IOUSBDevRequest结构体来描述控制请求，并将其发送给指定的控制端点。
- ControlRequestAsync：异步执行控制请求。该方法有一个IOUSBDevRequest结构体，并将其发送给指定的控制端点。请求完成时，执行提供给该函数的回调函数。传递给完成回调的argO参数值指定从设备读取或写入的字节数。
- ControlRequestTO：同步执行控制请求，允许调用者指定超时参数。
- ControlRequestAsyncTO：异步执行控制请求，关于请求可以执行的时间有一个最大限制。

设备可能会使用批量端点来执行数据传输。第8章介绍过，批量端点可以将大批量数据读/写至设备，具有高吞吐量和可保证的数据发送，但存在变量延迟（这取决于其他设备此时是否试图

通过USB总线传输数据）。IOUSBInterfaceInterface类提供了下列方法，使应用程序可以通过到批量端点的管道读写缓冲区。下列方法还可用于通过中断端点进行数据传输。

- ReadPipe：同步执行从USB设备上的批量或中断端点到应用程序缓冲区的数据传输。应用程序缓冲区由其地址及缓冲区长度（单位为字节）描述。缓冲区长度还是一个输出参数，如果该方法成功完成，那么读入缓冲区的字节数将通过该size参数，报告给调用者。
- WritePipe：同步执行从应用程序提供的缓冲区到USB设备上的批量或中断端点的数据传输。应用程序的缓冲区由其地址及缓冲区的长度（单位为字节）描述。
- ReadPipeAsync：异步执行从USB设备上批量或中断端点到应用程序所提供缓冲区的数据传输。传输完成时，将操作结果及从设备读取的字节数传递给回调函数（通过argo参数传递）。
- WritePipeAsync：异步执行从应用程序所提供缓冲区到USB设备上批量或中断端点的数据传输。传输完成时，将操作结果及写入到设备的字节数传递给提供的回调函数（通过argo参数传递）。
- ReadPipeTO：同步执行从批量或中断端点到应用程序缓冲区的数据传输，带有一个为该操作指定的超时值。
- WritePipeTO：同步执行从应用程序缓冲区到批量或中断端点的数据传输，带有一个为该操作指定的超时值。
- ReadPipeAsyncTO：是一个ReadPipeTO方法的异步等价方法。传输完成或超时时，调用应用程序提供的回调函数。
- WritePipeAsyncTO：是一个WritePipeTO方法的异步等价方法。传输完成或超时时，调用应用程序提供的回调函数。

使用异步方法ReadPipeAsyncTO从批量端点读取数据的函数示例，如代码清单15-8所示。

**代码清单15-8 使用ReadPipeAsyncTO方法的演示函数**

```
IOReturn        MyAsyncBulkRead (IOUSBInterfaceInterface300** usbInterface, UInt8 pipeRef)
{
        void*           dataBlock;
        const UInt32    noDataTimeout = 50;             // 50 ms
        const UInt32    completionTimeout = 500;        // 500 ms
        void*           refcon;
        IOReturn        error;

        //分配一个缓冲区，保存读取的数据
        dataBlock = malloc(kMyTransferSize);

        //执行异步读取，并指定超时值
        //通过refcon参数，将dataBlock传递给回调函数
        refcon = dataBlock;
        error = (*usbInterface)->ReadPipeAsyncTO(usbInterface, pipeRef, dataBlock,
                                                 kMyTransferSize,
                                                 noDataTimeout, completionTimeout,
                                                 ReadCompletedCallback, refcon);
```

```
        //如果该方法返回一个错误，则回调将不被调用
        if (error != kIOReturnSuccess)
                free(dataBlock);

        return error;
}

void    ReadCompletedCallback (void* refcon, IOReturn result, void* arg0)
{
        void*           dataBlock = refcon;
        UInt32          byteCount = (UInt32)arg0;

        //如果读取成功完成，则处理从设备读取的数据
        if (result == kIOReturnSuccess)
        {
                ProcessReadData(dataBlock, byteCount);
        }

        //释放MyAsyncBulkRead中分配的缓冲区
        free(dataBlock);
}
```

另一个USB端点类型是同步端点。第8章已经介绍过，同步端点用于设备传输数据流，发送时延迟必须最小化，如音频或视频数据流。IOUSBLib完全支持到用户空间应用程序的同步数据传输。

同步管道可以保证USB总线上的带宽。设备报告其带宽需求，如果USB主机能满足它的需求，那么该设备便可以访问USB总线。全速设备可以通过每个帧上的同步管道传输数据（每毫秒一次），而高速或超速设备可以在每个微帧上传输数据（每125微秒一次）。

类似于内核实现，应用程序的同步数据传输通过填充IOUSBIsocFrame结构体数组进行设置，IOUSBIsocFrame结构体描述应用程序通过每个微帧（或帧，用于全速设备）上的同步管道所希望读/写的字节数。IOUSBIsocFrame结构体的定义如下所示：

```
typedef struct IOUSBIsocFrame {
   IOReturn     frStatus;     //返回时，帧的传输结果
   UInt16       frReqCount;   //该帧上请求的读/写字节数
   UInt16       frActCount;   //返回时，该帧上实际的读/写字节数
} IOUSBIsocFrame;
```

通过同步管道进行的读/写通常是指数十毫秒的传输，每当未完成的请求完成时，应用程序便会发送一个新的同步传输请求。IOUSBInterfaceInterface类提供了下列方法，通过同步管道来执行传输。

- ReadIsochPipeAsync：从同步管道执行异步数据传输。该数据传输会传递一个缓冲区地址，以保存从设备读取的数据，还传递一个IOUSBIsocFrame成员数组，描述每个帧可以接受的最大数据数。该方法有一个frameStart参数，用来确定数据传输开始的USB帧数，USB帧通常位于当前帧之后。当前的USB帧可以调用GetBusFrameNumber方法来确定。传输完成之后，应用程序提供的回调函数便会调用。传递给回调函数的arg0值是IOUSBIsoc-Frame数组的地址。应用程序可以查看该数组，确定每个帧上实际读取的字节数。

- WriteIsochPipeAsync：执行到同步管道的异步数据传输。调用者提供一个缓冲区的地址，该缓冲区包含写入到设备的数据，还提供一个IOUSBIsocFrame成员数组，用来描述每个USB帧上从该缓冲区传输的字节数。该方法有一个frameStart参数，用来确定数据传输开始的USB帧数， USB帧通常位于当前帧之后。当前的USB帧可以调用GetBusFrameNumber方法确定。传输完成之后，应用程序提供的回调函数便会调用。传递给回调函数的arg0值是IOUSBIsocFrame数组的地址。
- GetBandwidthAvailable：返回USB总线上的可用带宽，这是设备可以分配的最大带宽。该带宽是能够分配给每个帧（用于全速设备）或微帧（用于高速设备）同步管道的最大字节数。
- SetPipePolicy：应用程序可以修改同步管道或中断管道的带宽分配。对于同步管道，帧（用于全速设备）或微帧（用于高速设备）上可以传输的最大容许分组长度具有可变性。但该分组长度不可以大于设备端点描述符中的最大分组长度。对于中断管道，该方法可以设置最大容许分组长度和轮询时间间隔。但分组长度和轮询时间间隔不可以超过端点描述符中请求的值。如果设备端点描述符中请求的初始带宽无法满足，那么USB主机便无法通过同步管道传输数据，该方法可以为管道指定一个降低带宽的分配，使其可用。

执行同步读操作的函数示例，如代码清单15-9所示。请注意，在发送请求之前，该函数会初始化一个IOUSBIsocFrame成员数组，用来描述它在每个帧（或微帧，用于高速设备）上可以接受的最大字节数。请求完成之后，设备可以在每个帧上处理接收到的数据，它可能会少于请求的最大字节数。

请求开始的帧号通过startFrame参数指定，示例函数中，则通过将一个固定延迟添加到当前帧来指定。一般来说，应用程序通常会将同步请求的初始帧号设置为上一个同步请求的最后帧号。通过这种方式，驱动程序便可以从设备连续进行读取，而不会存在缺口。

**代码清单15-9 用户空间驱动程序同步数据传输的示例函数**

```
const int      framesPerRequest = 512;
const int     bytesPerFrame = 1024;

IOReturn MyScheduleIsocRead (IOUSBInterfaceInterface300** usbInterface, UInt8 pipeRef,
                             void* destinationBuffer)
{
    UInt64              startFrame;
    AbsoluteTime        timeNow;
    IOUSBIsocFrame*     frameList;
    void*               refcon;
    IOReturn            error;

    //读取当前帧号
    //（另外，我们可以在上一个请求之后，发送该isoc请求）
    error = (*usbInterface)->GetBusFrameNumber(usbInterface, &startFrame, &timeNow);
    if (error != kIOReturnSuccess)
        return error;
    //为请求开始时所在的帧添加一个偏移量
    //这样该帧可以恰好在当前帧的前面开始
```

```
    startFrame += 8;

    //分配一个IOUSBIsocFrame成员数组
    frameList = (IOUSBIsocFrame*)malloc(framesPerRequest * sizeof(IOUSBIsocFrame));
    //设置同步请求的每个帧上所读取的字节数
    for (int i = 0; i < framesPerRequest; i++)
        frameList[i].frReqCount = bytesPerFrame;

    //执行同步请求
    //通过refcon参数，将destinationBuffer传递给回调函数
    refcon = destinationBuffer;
    error = (*usbInterface)->ReadIsochPipeAsync(usbInterface, pipeRef, destinationBuffer,
                                startFrame,
                                framesPerRequest, frameList, IsocReadCompletedCallback,
                                refcon);
    //如果该方法返回一个错误，那么回调将不会调用
    if (error != kIOReturnSuccess)
         free(frameList);

    return error;
}

void IsocReadCompletedCallback (void* refcon, IOReturn result, void* arg0)
{
    uint8_t*           destinationBuffer = (uint8_t*)refcon;
    IOUSBIsocFrame*    frameList = (IOUSBIsocFrame*)arg0;

    //如果读取成功完成，则处理从设备读取的数据
    if (result == kIOReturnSuccess)
    {
        for (int i = 0; i < framesPerRequest; i++)
        {
            if (frameList[i].frStatus == kIOReturnSuccess)
            {
              UInt16          bytesRead;

              //处理该帧上读取的数据
              bytesRead = frameList[i].frActCount;
              ProcessReadData(destinationBuffer, bytesRead);
            }

            //计算下一帧数据的起始地址
            destinationBuffer += frameList[i].frReqCount;
        }
    }

    //释放MyScheduleIsocRead中分配的帧列表
    free(frameList);
}
```

### 15.5.3　低延迟同步传输

同步端点通常用于传输多媒体数据，如连续流向/来自USB设备的音频或视频。处理这种数

据的应用程序通常希望以最小延迟进行传输，例如，应用程序可能希望在USB设备接收到音频数据之后，立即开始处理从麦克风读取的数据。而前面描述的同步方法存在一个问题，即应用程序在能开始处理读取的数据之前，必须一直等到完成回调函数得到通知。如果应用程序发送一个包含10毫秒USB帧的同步请求，这意味着从该请求开始，一直到应用程序开始处理从设备读取的数据，至少会有10毫秒的延迟。

这一问题的解决方案是，IOUSBInterfaceInterface类包含低延迟同步传输方法，使设备在接收到数据之后，应用程序可以立即对其进行访问，即使整个同步传输还没有完成。为此，在同步传输进行时，I/O Kit会更新帧列表的值，frStatus和frActCount的值在传输期间便被写入，并用于应用程序。应用程序并不是等待完成回调触发，而是周期性地检查其帧列表的值，确定何时处理新数据。这可以通过用户空间应用程序中的实时线程等实现。

应用程序通过指定两次更新之间的毫秒数，来指定操作进行时，I/O Kit更新帧列表的频率（0~8的一个值）。例如，更新频率1是指应用程序的帧列表在传输期间，每毫秒由I/O Kit更新一次。值0意味着在传输结束之前，I/O Kit不会更新帧列表，尽管更新之后的帧列表在传输结束之后，应用程序接收完成回调之前，会立即用于应用程序。

为了使用低延迟的同步传输，用于帧列表数组和保存传输的源/目标数据的缓冲区必须使用IOUSBInterfaceInterface类提供的方法进行分配。这样，I/O Kit才能确保该缓冲区可用于内核（在传输时更新该缓冲区）及应用程序（在传输时读取结果）。

请注意，低延迟方法并不是使用IOUSBIsocFrame结构体来描述USB帧的传输，而是使用IOUSBLowLatencyIsocFrame结构体。该结构体包含一个额外字段frTimeStamp，用来保存I/O Kit结构体值的更新时间。

下列方法可用于低延迟的同步传输。

- LowLatencyCreateBuffer：分配一个缓冲区，用来保存低延迟同步写的源数据、低延迟同步读的目标数据，或描述同步传输的IOUSBLowLatencyIsocFrame成员数组。应用程序对该缓冲区的操作意图通过下列枚举值指定：kUSBLowLatencyWriteBuffer、kUSBLowLatencyReadBuffer或kUSBLowLatencyFrameListBuffer。
- LowLatencyDestroyBuffer：释放前面由LowLatencyCreateBuffer方法分配的缓冲区。
- LowLatencyReadIsochPipeAsync：从USB设备上的同步端点执行读操作。该方法对ReadIsochPipeAsync方法的参数进行扩展，使之包括updateFrequency参数，该参数用来指定I/O Kit在传输期间更新帧列表的频率（指定为毫秒数）。IOUSBLowLatencyIsocFrame数组和目标缓冲区都必须通过LowLatencyCreateBuffer方法分配。
- LowLatencyWriteIsochPipeAsync：写入到USB设备上的同步端点。该方法对WriteIsochPipeAsync方法的参数进行扩展，使之包括updateFrequency参数，该参数用来指定I/O Kit在传输期间更新帧列表的频率（指定为毫秒数）。IOUSBLowLatencyIsocFrame数组和源缓冲区都必须通过LowLatencyCreateBuffer方法分配。

## 15.6　小结

- 对于某些类型的设备（如USB设备），I/O Kit可以放弃内核驱动程序，完全在用户空间中实现其驱动程序。对最终用户来说，这提供了更好的体验。
- 并不是所有的USB设备都适合用户空间驱动程序，系统本身使用的设备（如全系统范围的音频设备或USB存储设备），必须在内核中实现。
- I/O Kit提供了一个用户空间库IOUSBLib，用于应用程序与USB设备的交互。IOUSBLib提供了IOUSBDevice和IOUSBInterface内核类的用户空间等价类，分别是IOUSBDeviceInterface和IOUSBInterfaceInterface。
- 应用程序可以通过创建匹配字典，并安装一个通知回调（参见第5章），来查看它所支持USB设备的插入和移除。通知应用程序它感兴趣的USB设备或USB接口已经连接到计算机上，之后应用程序可以实例化IOUSBDeviceInterface或IOUSBInterfaceInterface对象，以访问USB硬件。
- IOUSBLib对用户空间驱动程序的USB设备功能提供全面支持。在用户空间中，所有的端点类型都得到支持，包括控制、批量、中断和同步传输。
- 应用程序可以使用IOUSBLib执行同步或异步操作。使用回调通知应用程序异步请求已经完成。

# 第 16 章 调 试 16

调试是开发流程的一部分，也是内核级驱动程序或扩展持续维护的一部分。因此，具有高效的内核调试技能及知识是内核工程师工作中必不可少的重要部分。尽管在开发、质量保证流程中都特别用心，但也难以避免会出现bug。部分原因是即使在发布以后，驱动程序也可能会在没有经过充分测试的硬件/软件组合上运行。例如，驱动程序可能会运行在一个比当初测试时快或慢的CPU上，因而会产生计时器问题。

许多人都将内核调试看做是一种黑色艺术，而且有很好的理由。应用程序崩溃时，可以被转储到调试器，然后它通常可以精确定位到引起问题的具体代码行。而在内核中，事情却并非如此简单。尽管可以使用GNU Debugger进行调试，但这需要进行一些设置，并且通常需要两台计算机。另外，内核中的崩溃通常发生在与你的扩展完全不相关的部分，很难证明是否是驱动程序的问题。如果你足够幸运，在崩溃之后，也许还可以抽取信息或装载一个调试器，但如果内存被破坏，那么数据结构或回调栈中的值就不那么可靠了。

如果你正在为硬件设备编写驱动程序，事情可能会更复杂，因为设备发生故障时（用原型设备时更为常见），也会破坏内存，或者造成计算机锁定或崩溃。这时很难判断到底是硬件问题还是软件问题。

本章将介绍调试常见问题的各种技术和策略。

## 16.1 常见的问题类型

内核为什么会崩溃，为什么会发生其他问题，存在很多原因。但这通常是由一些常见错误引起的，了解了你所处理的问题所属的类别之后，再检查代码问题就会非常容易。下面简要看一下内核开发时，会遇到的一些问题。

- 竞态条件：一般类别的bug，用来描述彼此执行冲突的多个线程，最终结果取决于哪一个线程首先到达。竞态条件十分常见，它通常由多线程环境中的设计或锁较差造成。它们有时很难再现，在发现之前可能已经潜伏了很长时间。例如，当一系列特定事件以特定顺序发生时，可能就会出现问题，因为这取决于用户输入。
- 死锁：当锁的实现较差，线程等待一个从不会发生的事件或锁时，便会发生死锁，很可能是因为另一线程在用完之后未能成功释放该锁。死锁问题发生之后，也会影响到系统

中需要该锁的新线程。从用户角度来看，它们的应用程序好像被挂起，并且可能无法强制退出，因为它已受阻，要等待内核中的事件。

- 锁竞争：这是一个性能问题，当许多线程需要同一个锁，并过久等待该锁而不执行有用的操作时，便会造成锁竞争。锁竞争通常由于设计较差造成，可以通过采用正确的锁方案来避免。一般原则是锁定对数据的访问，而不是代码。受保护的大块代码可能看起来比共享数据的细粒度锁更简单，但这会降低性能，更容易发生死锁。
- 访问无效的内存：这是造成内核恐慌最常见的原因。与用户空间程序被放弃不同，CPU引起无效的内存异常时，内核便会恐慌。如果启用一个调试器，内核将会转储到该调试器，而不是在Mac OS X上显示死机灰屏或重启（iOS下的行为）。缓冲区溢出有时也会引起无效的内存异常，除非该缓冲区之后恰好紧跟着有效的内存，而这种情况下，可能会发生内存中断。
- 内存和资源泄漏：例如，驱动程序卸载，资源（如对象和缓冲区）处理不正确时，便会发生该问题。还可能是因为扩展在每次接收到请求时都分配一些内存，但在请求完成之后未能成功释放内存。内核并不具备垃圾收集功能，因此，泄漏会随着时间的推移而越积越多，最终引起内核恐慌。
- 非法的指令/操作数：如果CPU检测到指令无效或指令参数无效，便会发生该异常。发生的原因有：内存破坏，或者编写较差的驱动程序试图使用CPU上不存在的特性，例如，使用SSE3指令设置不支持该指令的机器。也可以将其归结为内存破坏的结果。
- 主中断上下文中的阻塞：这会导致内核恐慌，因为在主中断上下文中不可以阻塞。阻塞需要一个调度线程，因为阻塞通过将线程自愿置入睡眠状态实现。所以，在这种情况下，线程状态将会被保存，并且当调度器确定再次运行该线程时才恢复。而主中断处理程序无法恢复，它必须一直运行，直至结束，不可以中断。许多内核API在某些情况下可能会阻塞。例如，系统的内存很低时，内存分配便会阻塞，这将导致一些内存换出到磁盘，为该请求释放内存。因此，某些函数在主中断上下文中不可以使用，如IOMalloc()或IOLog()。
- 自愿恐慌：当内核自愿决定崩溃时发生，因为内核已经确定某些操作将出现严重错误，或者发生的意外情况无法修复。例如，无法阻塞的内存分配失败时，便会出现这种情况。驱动程序可以调用IOPanic()使内核发生恐慌，该函数是panic()函数的一个包装器。

当然，还存在许多其他问题，但大多数问题都是上述典型问题的其他表现形式。

## 16.2　内核恐慌

恐慌是内核处理异常情况的主要防御机制，如前面的问题列表。最安全的操作流程并不是试图保持错误状态继续执行，而是立即终止系统执行，以避免损坏文件系统。当Mac OS X系统恐慌时，用户很可能会看到一个多语言的恐慌屏幕，除非内核已经配置了调试选项，指示用户重新启动计算机，如图16-1所示。

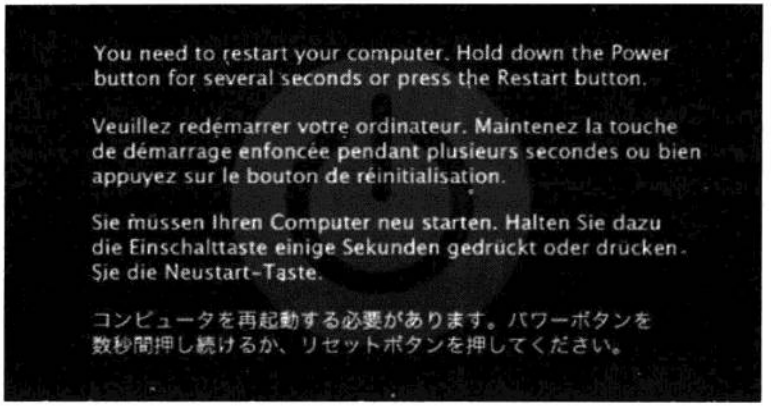



图16-1 Mac OS X恐慌屏幕，相当于Windows的死机蓝屏

在后台，系统会保存一个内核恐慌日志，其中包含内核恐慌所在处理器（核心）的栈跟踪。内核恐慌日志会临时写入到系统的NVRAM（non-volatile random-access memory，非易失性RAM）中，因为系统发生内核恐慌之后，访问文件系统通常并不安全。毕竟，内核恐慌的目的不是给用户带来麻烦，而是在该部分失去控制之前将其关闭，保护文件系统不受损坏。

文件系统再次启动之后，内核恐慌日志会复制到/Library/Logs/DiagnosticReports/目录中。系统还会弹出一个崩溃报告对话框窗口，使用户可以将该问题报告给苹果公司。

## 16.3 调试机制

调试技术有多种，具体使用哪一种调试技术，取决于当前问题的性质。这里讲述一些可以帮助调试内核问题的技术。表16-1综述了本章将讲述的机制。

表16-1 内核调试机制综述

| 名 称 | 描 述 |
|---|---|
| IOLog()、kprintf()、printf() | 这些函数执行系统日志（kernel.log）的基本跟踪。还可以通过FireWire连接或串行端口（如果可用），将kprintf()定向至远程系统 |
| 通过以太网或FireWire使用GDB远程调试 | 这可以将GDB装载到崩溃的或暂停的远程系统上。远程调试的协议被称为KDP（Kernel Debug Protocol，内核调试协议），内置于内核中，但必须手工启用。FireWireKDP实现还可以捕获核心转储（系统的内存镜像） |
| 使用GDB实时调试运行的内核 | 这在实时系统中，可以使GDB装载到其运行所在的内核上 |
| KDB | 这是内核中的一个调试器，默认情况下并不内置。KDB只能通过传统的串行端口工作，并且只存在于Xserve服务器（以及OS X的虚拟机实例）上 |
| 通过以太网进行远程核心转储 | kdumpd服务器可以配置成自动从网络中崩溃的Mac上下载核心转储 |

上面讲述的大部分调试技术默认情况下都包含在内，但在内核中并不一定默认启用，因为它们可能会与运行的内核或硬件设备进行交互，或者造成安全风险，因为它可能会获取目标内存的敏感内容。

通过设置内核启动参数，可以启用调试机制或控制内核的调试行为。启动参数可以通过以下两种方式进行设置：使用nvram命令，或将其添加至/Library/Preferences/SystemConfiguration/com.apple.Boot.plist的启动参数键中。可用的启动参数有很多，但我们最关注的是debug参数，它用来控制调试器和系统调试行为。该参数是一个整数值，包含的标志如表16-2所示。

**表16-2 调试启动参数的标志**

| 名　称 | 值 | 描　述 |
|---|---|---|
| DB_HALT | 0x01 | 内核在系统启动时暂停，等待装载调试器 |
| DB_PRT | 0x02 | 将printf()输出发送至控制台 |
| DB_NMI | 0x04 | 在内核恐慌、按下电源按钮或Command-Option-Control-Shift-Escape时，系统进入GDB |
| DB_KPRT | 0x08 | 将kprintf()输出发送至串行端口（如果可用）或FireWire远程日志 |
| DB_KDB | 0x10 | 将KDB设置为默认调试器（需要构建一个自定义内核） |
| DB_SLOG | 0x20 | 将额外的诊断信息输出到系统日志 |
| DB_ARP | 0x40 | 允许调试器发送ARP请求，通过路由器进行调试，而无须配置固定的ARP项 |
| DB_KDP_BP_DIS | 0x80 | 允许GDB的旧版本装载到新系统上 |
| DB_LOG_PI_SCRN | 0x100 | 禁用图16-1所示的图形内核恐慌对话框 |
| DB_KDP_GETC_ENA | 0x200 | 在内核恐慌之后，提示c = continue、r = reboot和k = enter KDB |
| DB_KERN_DUMP_ON_PANIC | 0x400 | 触发内核恐慌的核心转储 |
| DB_KERN_DUMP_ON_NMI | 0x800 | 启用NMI事件的核心转 |
| DB_DBG_POST_CORE | 0x1000 | NMI核心转储之后，它在调试器中等待 |
| DBG_PANICLOG_DUMP | 0x2000 | 如果设置，则传输内核恐慌日志，而不是核心转储 |
| DBPG_REBOOT_POST_CORE | 0x4000 | 在核心/内核恐慌日志转储之后，初始化重启 |

这些值作为位掩码，可以通过操作或运算，将多个值合并到一起。例如，启用NMI，禁用图形内核恐慌屏，并进入内核恐慌调试器。我们可以将这些值合并到一起，如下所示：

```
0x04 | 0x100 = 0x104
```

该值可以在com.apple.Boot.plist中设置，或使用nvram命令：

```
$ sudo nvram boot-args="original_contents debug=0x104"
```

如果禁用或移除调试选项，可以执行如下命令：

```
$ sudo nvram boot-args="original_contents"
```

如果你的启动参数设置已经存在，该命令将会对其进行覆写，因此，一定要先查询boot-args参数，防止对其覆写。

### 16.3.1 启动时修复崩溃

你的扩展可能安装在/System/Library/Extensions目录中，在系统启动时会自动加载。如果在系统启动时，该扩展总是崩溃，那么系统可以采用下列方式进行修复。

- 在你听到启动音之后，按住shift键，以安全模式启动，出现苹果徽标时，松开按键。这只能确保基本的内核扩展（KEXT）被加载，你也可以移除KEXT，手工启动系统。
- 使用FireWire或Thunderbolt线，将该系统装载到另一台计算机上，并在启动时按住T键，在目标磁盘模式中启动。然后可以从系统磁盘移除相关扩展。
- 启动进入另一个分区（如果该分区可用）。
- 如果引起问题的KEXT是硬件驱动程序，那么将其从系统上移除很可能会妨碍该驱动程序加载。
- 如果需要重置启动参数，请执行NVRAM重置。

如果无法确定引起崩溃的位置，则可以按下Command-V，在Verbose模式下启动系统。

**提示** 基于英特尔的Mac所支持的启动组合键详细信息，请参见http://support.apple.com/kb/ht1533。

### 16.3.2 采用IOLog()追踪

我们已经在本书多处看到了追踪操作。追踪主要是将IOLog(…)语句置于代码中，输出变量，测试条件块是否被触发。IOLog()最终输出到kernel.log文件中。内核中存储IOLog()消息的缓冲区非常有限，因此，如果你非常快速地写入大量长信息，该缓冲区将会回绕，在syslog守护进程将数据写入日志之前，可能会覆写数据。当输出与预期不符时，可能会引起混淆，导致不正确的判定。

驱动程序中的许多函数每秒可能会被调用数百甚至上千次，因此，每次调用都输出消息可能会适得其反。尽管记录守护进程可以合并相同的消息，但如果这些消息之间的差别比较细微，这就会变得相当分散。守护进程也无法防止缓冲区回绕，以及覆写某些消息。

可以通过重新编译内核，或使用更简单的方式，将内核标志msgbuf=n（*n*为缓冲区长度，单位为字节）添加到内核的启动配置属性列表com.apple.Boot.plist中，以此来改变IOLog()写入的系统日志缓冲区长度。该文件位于/Library/Preferences/SystemConfiguration/目录中。

即使较大的缓冲区能够降低数据出错的可能性，但当从驱动程序“热路径”中的函数输出时，也无法防止日志中涌入消息。为了避免该问题，可以使用该技术的变体，如代码清单16-1所示。

**代码清单16-1 限制IOLog输出**

```
static uint32_t        conditionCount = 0;
com_osxkernel_MyDriver::driverMethod()
{
    ...
    if (someCondition)
    {
```

```
        conditionCount++;
        if (conditionCount % 1000 == 0)
            IOLog("condition has occurred: %u times\n", conditionCount);
    }
}
```

这会记录一个条件发生的次数，但每1000次只输出一个消息。如果你希望调试主中断过滤器，便可使用这一方案的变体。但IOLog()调用需要从主中断过滤器例程移至可安全调用的地方，如二级中断处理程序或一个自定义IOUserClient方法。

如果你的驱动程序有多个实例，你可能希望输出一些额外信息，以便判断哪一个实例正在执行什么操作。如果你处于IOService上下文中，则可以输出this指针，来唯一标识每个实例。

你是否应该在驱动程序代码中放置IOLog()语句？是和不是。答案可能会不同，但对驱动程序来说，向系统日志中写入无用的输出显然非常糟糕，这会隐藏某些可能对系统其他部分来说非常重要的消息。但输出一些关于异常条件的消息却可以接受。如果你喜欢在代码中放置IOLog()语句，那么可以使用一个能够由用户空间客户端开启或关闭的条件变量，来防止它们输出。这样做的问题是会稍微增加可执行长度， CPU要额外执行条件调试语句。另一个方案是使用预处理器指令，如#ifdef DEBUG … #endif，该语句会被编译，但不会执行。当然，还有一种做法是将大部分调试语句完全注释掉，使代码更易于阅读。如果用户报告一个问题，后两种方案的缺点是驱动程序发生问题后，无法启用调试，也不要求用户安装驱动程序的调试版本。采用这三个方案的组合当然也可以，但通常建议不要在代码中随意设置调试记录，而只放在必要的地方，这会对你有所帮助，甚至能解决你未曾预料到的问题。

虽然使用IOLog()追踪这一方案看起来非常原始，但它在查找bug时通常十分高效。当然，该方案在系统实际并未崩溃，而你可以通过系统日志观察操作行为时最有效。但如果系统崩溃，syslog守护进程则无法将最新的日志缓冲区内容写入到日志文件。因此，该输出在下一次重新启动时，可能会丢失。解决该问题的方法有许多，如通过FireWire远程追踪，这将在本章后面讲述。

### 16.3.3　输出栈跟踪

IOLog()函数能够用于许多地方，但在某些情况下，仅知道函数被调用还不够。还需要知道调用函数的调用栈，因为它可能从多个代码路径调用。OSReportWithBackTrace()函数便可以输出调用栈，如代码清单16-2所示。

**代码清单16-2　使用OSReportWithBackTrace()转储调用栈**

```
void testFunc3() {
    IOLog("address of testFunc3: %p\n", &testFunc3);
    OSReportWithBacktrace("OSReportWithBacktrace() called from testFunc3()");
}
void testFunc2() {
    IOLog("address of testFunc2: %p\n", &testFunc2);
    testFunc3();
}
void testFunc1() {
```

```
    IOLog("address of testFunc1: %p\n", &testFunc1);
    testFunc2();
}
bool com_osxkernel_MyDebugDriver::start(IOService * provider) {
    testFunc1();
    ...
}
```

代码清单16-2中的代码，将生成如下结果：

```
address of testFunc1: 0x9ac280
address of testFunc2: 0x9ac250
address of testFunc3: 0x9ac220
OSReportWithBacktrace() called from testFunc3()
Backtrace 0x9ac26f 0x9ac29f 0x9ac2ee 0x543f60 0x542137 0x5426e9 0x5443d5
    Kernel Extensions in backtrace (with dependencies):
        com.osxkernel.MyDebugDriver(1)@0x9ab000->0x9acfff
```

你可能已经发现，输出的测试函数地址都非常相似，但却与回溯中输出的地址不同。IOLog()输出的地址是与每个函数的开头相对，但在回溯中，你看到的是每个测试函数调用下一个测试函数的地址。OSReportWithBackTrace()函数还输出回溯中每个KEXT的起始地址和结束地址。我们可以看到，KEXT在内存地址0x9ab000处被加载，你可能注意到，testFunc1()、testFunc2()和testFunc3()都处于MyDebugDriver的地址范围中。使用该信息，将基地址减去函数地址，还可以计算出函数在其可执行映像中的偏移量，例如，对于testFunc1()：0x9ac280 - 0x9ab000 = 4736字节。然后，可以对GDB调试器使用disassemble <offset>命令。

### 16.3.4 通过FireWire远程跟踪

通过与另一个系统的FireWire连接，可以重定向krprintf()函数的输出。该方法比使用IOLog()更健壮，因为在崩溃时，日志输出会保存在远程系统上。该方案的另一个优点是在初始启动中便可以使用，这对调试执行系统启动处理的驱动程序非常有用，如存储和显示驱动程序。这还可以调试关机和睡眠事件。你的驱动程序无须任何特殊的支持或修改，便可以通过FireWire输出日志信息。

通过FireWire配置远程记录的操作已经包含在Mac OS X 10.5及之后的版本中，而无须安装额外的KEXT。你需要两台Mac来对此进行设置。这两个系统无须运行相同的Mac OS X版本。

在目标机上，对你希望发送日志输出的机器，执行如下设置：

```
$ sudo nvram boot-args="debug=0x8"
```

该启动选项可以重定向kprintf()函数，从而将输出映像到FireWire接口及系统日志。重启系统，使该选项生效。接下来在两个系统之间连接一个FireWire线。与IOLog()不同，kprintf()函数是同步的，这意味着在它返回时，便已经通过FireWire传输了消息。kprintf()在完成之前，会一直禁用中断，过多使用该函数会影响与定时器相关的事件。因为中断被禁用，该函数的参数所引用的内存恰好被换出时，便会引起崩溃。

**警告**　建议在调试时，目标机和调试机都断开所有其他的FireWire设备。

在接收调试输出的机器上，运行fwkpfv命令，这是FireWire日志查看器实用程序。如果目标机连接正确，并且FireWire线正确连接，那么几秒之后你便会接收到调试输出。以下示例是目标机启动时捕获的信息片段：

```
Welcome to FireWireKPrintf. (viewer v2.6)
AppleFWOHCI_KPF: version 4.7.1 - init
u>626665 AppleFWOHCI_KPF: Time format-> Microseconds = 'u>clock_uptime_micro'
u>1141021 AppleUSBHub::setPowerState(0x4fbe200, 0 -> 4) took 301 ms
u>2065689 [Bluetooth::CSRHIDTransition] DeviceRequest error: e00002ed
u>2 109928 AppleUSBHub::powerStateWillChangeTo(0x4f13200, AppleUSBHub, 4 -> 3) took 100 ms
u>2117294 AppleUSBHub::powerStateWillChangeTo(0x4f70e00, AppleUSBHub, 4 -> 3) took 100 ms
u>2130057 AppleUSBHub::powerStateWillChangeTo(0x4f70a00, AppleUSBHub, 4 -> 3) took 100 ms
…
…
…
u>20236388 Adding domain PPP (family 34)
u>24073549 kPEDisableScreen -1
u>24158956 kPEDisableScreen 1
u>25568081 initialize_screen: b=4645B000, w=00000690, h=0000041A, r=00002000, d=00000001
u>25568131 kPEEnableScreen 1
u>304129579 IOSCSIPeripheralDeviceType00::setPowerState(0x4fe3500, 3 -> 4) async took 351 ms
u>1070371211 IOSCSIPeripheralDeviceType00::setPowerState(0x4fe3500, 3 -> 2) async took 1153
ms
```

如果你希望通过FireWire记录KEXT，则必须使用kprintf()函数，而不是IOLog()，因为IOLog()函数在内部并不使用kprintf，而是调用printf()，而它只能访问到内核日志。解决该问题的一个策略是创建你自己的IOLog()和kprintf()包装器函数，前者用于发布版，后者用于调试版。

使用kprintf()进行记录的示例，如代码清单16-3所示。

**代码清单16-3　使用kprintf()，通过FireWire记录远程机器的消息**

```
#include <kern/debug.h> // Declares kprintf()
bool com_osxkernel_MyDebugDriver::start(IOService * provider)
{
        kprintf("%s::start - Hello FireWire Listeners\n", getName());
        return true;
}
```

运行该代码，在远程系统上会生成如下结果：

```
u>1071578492 com_osxkernel_MyDebugDriver::start - Hello FireWire Listeners
```

如果内核崩溃，还可以通过FireWire日志查看器获取内核恐慌日志。此外，还可以使用FireWire，将GNU调试器远程装载到内核，这将在本章后面讲述。

### 16.3.5 远程内核核心转储

Mac OS X提供了一种通过网络将核心转储从崩溃（或挂起）的系统传输到远程机器的机制。核心转储是系统内存固定内容的二进制映像。通过捕获核心转储，可以获取系统崩溃时所处的准确状态，而且可以通过GDB使用该映像，获取系统中所有线程的栈追踪，检查内存内容及内核数据结构。

方便的是，启用核心转储所需的所有操作在Mac OS X中已经存在，只需要稍微配置一下即可。在转储服务器上，对于从崩溃机器上接收核心转储的机器，需要激活kdumpd守护进程，如下所示：

```
# sudo vi /System/Library/LaunchDaemons/com.apple.kdumpd.plist
```

将Disabled键由true改为false。如果你愿意，还可以配置转储所在的目录，默认目录是/PanicDumps。kdumpd守护进程的启动，如下所示：

```
# sudo launchctl load /System/Library/LaunchDaemons/com.apple.kdumpd.plist
# sudo launchctl start com.apple.kdumpd
```

该服务器在1069端口上使用UDP，因此，你应该确保目标机器和服务器之间没有防火墙。目标机器和服务器无须运行相同的Mac OS X版本。

---

**注意** 如果要使用kdumpd，目标机器（崩溃的系统）需要连接到以太网络。不可以使用AirPort设备，因为KDB的驱动程序（处理转储传输）只处理以太网设备。加载时，可以使用IORegisteryExplorer检查该内核调试驱动程序的装载位置，并搜索IOKernelDebugger驱动程序。

---

kdumpd守护进程可以接收多个机器的核心转储，并根据机器的IP地址，对每个转储进行归档。如果你的公司开发内核中运行的软件，可以将所有的Mac配置为崩溃发生时自动将转储发送至中央服务器。这在质量保证测试人员在测试过程中遇到问题时，会节省许多时间，因为工程可以立即开始调试转储的映像。

---

**警告** 需要注意的是，只可以在信赖的网络中使用kdumpd，因为崩溃系统的内容在网络上传输时并未加密，其中可能会包含一些敏感信息，如密码和私钥。

---

配置目标机非常简单，可以设置/Library/Preferences/SystemConfiguration/com.apple.Boot.plist中的内核启动参数，或使用nvram命令实现，如下所示：

```
# sudo nvram boot-args="debug=0xd44 _panicd_ip=192.168.1.1"
```

当内核恐慌或NMI（non-maskable interrupt，非屏蔽中断）事件触发时，前面所说的命令会指示内核转储核心。在系统完全挂起，并且看起来没有反应但实际上又没有发生恐慌时，使用

NMI事件非常有用。这种情况下，可以使用计算机上的电源按钮，触发NMI事件，启动核心转储。此时，计算机将会停止，并且没有进程运行。如果机器有反应，而且你执行了这一操作，那么在核心转储传输之后，机器将会恢复，就像什么都没有发生过一样。

_panicd_ip参数指定运行kdumpd机器的IP地址。如果你打算设置一个一直运行的内核恐慌服务器，最好使用静态IP地址。服务器不可以使用主机名或DNS名，因为不能进行名称转换。

如果按下电源按钮，目标机器的屏幕上将会显示如下输出：

```
Entering system dump routine
Attempting connection to panic server configured at IP 192.168.1.1, port 1069
Resolved 192.168.1.1's (or proxy's) link level address
Transmitting packets to link level address: 00:16:cb:a6:73:8b
Kernel map size is 4546437120
Sending write request for core-xnu-1699.22.73-192.168.1.2-2fe8a6d9
Protocol features: 0x1
Kernel map has 1389 entries
Generated Mach-O header size was 100224
```

目标机将向屏幕上写点（.），直至转储完成。如果是NMI事件触发转储，系统将会恢复；如果是内核恐慌触发转储，系统将会等待装载远程调试器。

### 16.3.6　KDB

内核支持一个内核中的调试器，即KDB。KDB只能通过串行端口进行调试。内核中默认情况下并不包括KDB，因此，为了使用KBD，需要编译并安装一个自定义内核。KDB包含一些应用程序，用于非常底层的调试，在这些地方，FireWire和以太网都均不可用。KDB需要机器上存在一个用于调试的本地串行端口，而它只在目前停产的Xserve中存在（尽管这种串行端口在虚拟机下使用Mac OS X时可用）。综合考虑，我们推荐使用GDB。这里只讲述KDB，以免与GDB混淆。

### 16.3.7　通过以太网或FireWire使用GDB远程调试

内核可以通过以太网（IP/UDP）或FireWire连接来使用GDB。同样，这需要两台运行Mac OS X的计算机。虽然GDB通过Xcode得到支持，但不可以使用Xcode调试内核，必须使用命令行接口。但GDB属于Xcode的一部分。调试的机器上无须安装Xcode，它只在远程系统（客户端）上才需要安装。

**注意**　严格来讲，可以使用另一个运行GDB的操作系统作为主机来调试Mac OS X目标系统。尽管并没有文档对此进行说明，但一些文档暗示它具有可行性，但却非常繁琐。

不同于KDB，内核中默认内置了调试支持。但调试功能默认禁用，在目标机器上添加适当的启动参数，便可以轻松启用，例如：

```
$ sudo nvram boot-args="debug=0x144 -v".
```

-v（详细）标志并不是非常需要。在启动时，它会禁用带有苹果徽标的灰屏，而显示一个文本控制台，常见于UNIX和Linux系统上，它会在系统启动时显示日志消息。设置了启动参数之后，系统需要重启，更改才会生效。

### 1. 配置主机

设置目标机器非常简单。但主机需要进行一些准备。开始调试之前，需要下载目标系统使用的内核版本所对应的正确Kernel Debug Kit。苹果好像并不会及时为每个构建发布一个版本，因此，目标系统必须降级或升级，以匹配Kernel Debug Kit的发布版本。如果使用的版本不正确，GDB将无法解析正确的符号和数据结构，结果可能就会出错，并引起严重混淆，例如，调用了不应该调用的函数。

---

**提示** Kernel Debug Kit并不是Xcode的一部分，可以从http://developer.apple.com/hardwaredrivers/download/kerneldebugkits.html上下载。该网页包含的是旧版本，而新版本发布在苹果开发者网站的Downloads➤Developer Tools中。该网页仅限于Mac开发者计划会员使用（缴纳年费才能成为会员）。

---

Kernel Debug Kit包含以下内容：

- 内核的调试版本（mach_kernel）；
- I/O Kit族的调试版本及选择的KEXT调试版本；
- 符号文件；
- 各种脚本；
- GDB宏。

---

**警告** 不要将系统上的文件替换为Kernel Debug Kit的文件，这些文件无须安装在目标机器或主机系统中。实际上，无须安装工具包中的任何文件，通过装载的映像，便可以直接访问它们。

---

可以将Mac OS X系统的默认内核（mach_kernel）替换为Kernel Debug Kit中的调试版本。这会帮助你获得更精确的结果和栈追踪，因为优化已被禁用。

如果你有主机系统上安装的XNU内核的源代码，则可以将其与调试器链接，在调试器中便可以看到源代码，而不是汇编代码，尽管在你仅调试自有扩展时，不会经常用到它（在这种情况下，可以只为自己的扩展链接源代码）。

### 2. 装载到远程目标

如果你已经在目标机器上，使用nvram命令设置了适当的启动参数，并且主机上已经准备好了Kernel Debug Kit，那么现在可以按下系统的电源按钮，在目标系统上触发NMI事件。然后，屏幕左上角会显示如下文本：

```
Debugger called: <Button SCI>
ethernet MAC address: 00:16:cb:a6:74:8b
ip address: 192.168.1.1

Waiting for remote debugger connection.
```

执行如下步骤，启动GDB，并装载到远程目标：

```
$ gdb -arch i386 /Volumes/KernelDebugKit/mach_kernel
```

如果调试主机运行在与目标系统不同的体系结构上，则可以指定arch参数。例如，该示例中，我们在64位内核系统上调试一个运行32位内核的目标机器。还可以指定x86_64进行相反的操作。

```
(gdb) source /Volumes/KernelDebugKit/kgmacros
```

前面的命令行将加载指定的GDB宏，帮助你检查目标系统内核的状态。例如，可以转储一个运行任务或线程的列表。输入help kgm，获得可用宏的完整列表。

使用如下命令，装载到目标机器：

```
(gdb) target remote-kdp
(gdb) attach 192.168.1.1
Connected.
(gdb)
```

前面的命令将执行到远程目标机器的装载，从而可以开始调试会话。然后，就能发送命令，例如，发送bt以获取栈追踪，看起来如下所示：

```
#0  Debugger (message=0xba97e4 "Button SCI") at /SourceCache/xnu/xnu-
1504.15.3/osfmk/i386/AT386/model_dep.c:867
#1  0x00ba8de3 in ?? ()
#2  0x00556636 in IOFilterInterruptEventSource::normalInterruptOccurred (this=0x4eab980) at
/SourceCache/xnu/xnu-1504.15.3/iokit/Kernel/IOFilterInterruptEventSource.cpp:140
#3  0x00b73a50 in ?? ()
#4  0x00b72ccd in ?? ()
#5  0x00b85fc5 in ?? ()
#6  0x00b89621 in ?? ()
#7  0x0056ac20 in IOSharedInterruptController::handleInterrupt (this=0x4e9cd80,
nub=0x4e9cd80) at /SourceCache/xnu/xnu-1504.15.3/iokit/Kernel/IOInterruptController.cpp:727
#8  0x00bcf5bb in ?? ()
#9  0x00b66213 in ?? ()
#10 0x00b71911 in ?? ()
#11 0x00580d96 in PE_incoming_interrupt (interrupt=73) at /SourceCache/xnu/xnu-
1504.15.3/pexpert/i386/pe_interrupt.c:65
#12 0x002ab432 in interrupt (state=0x4e9dd20) at /SourceCache/xnu/xnu-
1504.15.3/osfmk/i386/trap.c:511
#13 0x002a1c2e in lo_allintrs () at cpu_data.h:397
#14 0x00225bba in processor_idle (thread=0x10bff0, processor=0x4cf0dac) at
/SourceCache/xnu/xnu-1504.15.3/osfmk/kern/sched_prim.c:2982
#15 0x0022698c in thread_select (thread=0x5e407a8, processor=<value temporarily unavailable,
due to optimizations>) at /SourceCache/xnu/xnu-1504.15.3/osfmk/kern/sched_prim.c:1327
#16 0x002275b0 in thread_block_reason (continuation=0, parameter=0x0, reason=<value
temporarily unavailable, due to optimizations>) at /SourceCache/xnu/xnu-
1504.15.3/osfmk/kern/sched_prim.c:1856
```

```
#17 0x00227654 in thread_block (continuation=0) at /SourceCache/xnu/xnu-
1504.15.3/osfmk/kern/sched_prim.c:1875
#18 0x464debbc in ?? ()
```

该命令将显示内核栈，这是CPU执行NMI事件时的函数调用顺序。在该示例中，我们可以看到，系统暂停之前的最后操作是响应NMI中断。如果想查看其他CPU（核心）正在执行的操作，可以发送showcurrentstacks命令，输出系统中每个CPU（核心）的栈追踪。

现在可以设置断点或检查内核的状态。发送continue命令，恢复内核。本章后面将会查看GDB的详细用法。

### 3. 使用FireWire调试

除了以太网，还可以通过FireWire采用FireWireKDP机制，来使用内核调试协议。fwkdp工具可以用来帮助设置适当的调试参数，也可以手动设置。FireWireKDP还可以通过FireWire进行记录，并将核心转储传输到远程系统。

执行如下操作，在目标系统上配置FireWireKDP：

```
$ sudo fwkdp --setargs
FireWire KDP Tool (v1.3)
Boot-args helper mode:
*** Would you like to enable kernel core dumps? y|[n] > y
Setting boot-args with 'sudo nvram boot-args="debug=0xd46 kdp_match_name=FireWire
_panicd_ip=1.2.3.4"'
Setting boot-args... done.
Restart for the nvram changes to take effect.
```

**提示** fwkdp的帮助页面（man page）包含更多信息。

在运行调试器的主机系统上，还需要运行fwkdp。请确保在代理模式中启动fwkdp。该操作完成之后，可以采用与以太网相同的方式，使用GDB进行调试。装载到远程目标机器的处理几乎相同，唯一的关键不同点是要装载到localhost，而不是目标机器的IP地址。目标系统重启之后，像之前一样，按下电源按钮，生成一个NMI事件，便可以进入调试器。目标机器屏幕上的输出结果如下所示：

```
Debugger Called: <Button SCI>
Entering system dump routine
Attempting connection to panic server configured at 1.2.3.4 port 1069
AppleFWOHCI_KDP: Darwin Kernel Version 10.8.0: Tue Jun 7 16:33:36 PDT 2011; root:xnu-
1504.15.3~1/RELEASE_i386
AppleFWOHCI_KDP: v4.7.3 configured as KDP sender/receiver.
Recevied a debugger packet, transferring control to the debugger
Transmitting packets to link level address: 00:1c:df:f7:e0:72
Kernel map size is 1187131392
Sending write request for core-xnu-1504.15.3-0.0.0.0-ff28cfb5
Kernel map has 850 entries
Generated Mach-O header size was 79932
Transmitting kernel state, please wait: ……
Total number of packets transmitted: 502848
Waiting for remote debugger connection.
Connected to remote debugger.
```

如果一切顺利，运行在主机上的fwkdp代理会将核心文件下载到其工作目录中。前面示例中的核心转储被命名为core-xnu-1504.15.3-0.0.0.0-ff28cfb5。下载该核心之后，远程系统将等待装载调试器。

### 16.3.8　实时调试运行的内核

Mac OS X（自10.5版本开始）鲜为人知但却非常强大的特性是，能够将调试器装载到一个运行的系统上。实时调试需要启用/dev/kmem字符设备文件支持，使用户空间进程可以读写内核的内存地址空间。使用如下命令，启用/dev/kmem支持：

```
$ sudo nvram boot-args="kmem=1"
```

---

**注意**　前面的命令会清除现有的启动参数，因此，需要添加该参数以及其他任何你想要的参数，例如，启用远程调试或FireWire记录。

---

重启之后，可以通过检查dev/kmem文件是否存在，来测试它是否工作。装载到实时内核的处理与装载到远程目标机器类似，步骤如下所示：

```
$ sudo gdb /Volumes/KernelDebugKit/mach_kernel
(gdb) target darwin-kernel
(gdb) source /Volumes/KernelDebugKit/kgmacros
Loading Kernel GDB Macros package. Type "help kgm" for more info.
(gdb) attach
Connected.
```

此时，可以检查内核状态，例如，使用showcurrentthreads命令。

在许多情况下，实时调试都非常有用，例如，使用驱动程序的应用程序执行用户客户端方法时在内核中挂起的情形。你可以连接到内核，找出该事件的位置。还可以检查驱动程序的内存及其数据结构。实时调试仅在系统可操作时使用，但无法调试崩溃或死锁的系统。

### 16.3.9　使用虚拟机调试

如果没有第二个可用的机器，则可以使用虚拟机执行内核调试。VMWare Fusion或Parallels desktop等软件能够虚拟化运行Mac OS X的另一个副本。将所需的启动参数放入/Library/Preferences/SystemConfiguration/com.apple.Boot.plist中，便可以在虚拟机上启用内核调试特性。在虚拟机上，无法调试硬件驱动程序，因为在虚拟机上，无法直接使用PCI或Thunderbolt设备。但可以将USB设备分配给虚拟机实例。Mac OS X Lion可以作为虚拟实例运行，但只有Mac OS X的服务器版才能虚拟化。

### 16.3.10　在内核中使用GDB调试

GDB可以使用多种方式调试载入的内核或KEXT，如：

- 通过以太网远程调试；
- 通过FireWire远程调试；
- 在捕获的核心转储文件上调试；
- 在实时系统上实时调试；
- 使用虚拟机调试。

另外，内核可以采用下列方式，进入调试器：

- 因为内核恐慌；
- 手工触发NMI事件；
- 因为设置了DB_HALT选项，在启动时暂停系统，进入调试器；
- 调用PE_enter_debugger()函数，采用代码编程方式

采用编程方式进入调试器只可用于远程调试设置；实时调试在启动时无法使用，当系统崩溃或使用NMI事件暂停时，也无法采用编程方式，因为这会停止系统，直至装载一个远程调试器，因而gdb实例也会运行调试会话。

### 1. 内核GDB宏

Kernel Debug Kit预装了大量有用的GDB宏函数，极大地简化了检查内核的任务。这些宏可以帮助解释常见的内核数据结构，如任务和线程描述符，以及与内存相关的数据结构，如VM映射。另外，它还提供了访问PCI配置空间、I/O空间及未转换物理内存的函数。部分可用的宏，如表16-3所示。

表16-3 有用的内核GDB宏

| 名 称 | 描 述 |
|---|---|
| showalltasks | 显示所有系统任务的一个列表 |
| showallthreads | 显示所有系统线程的一个列表 |
| showallstacks | 输出每个线程的栈追踪（在有大量数据的情况下使用） |
| showallkmods | 显示所有加载的KEXT及其在内存中所加载地址的一个列表 |
| showallclasses | 显示所有已知的类、类长度及实例计数 |
| showregistry | 将I/O Registry信息显示在屏幕上（类似于ioreg） |
| paniclog | 显示内核恐慌日志 |
| systemlog | 显示内核日志 |
| showcurrentstacks | 显示每个处理器上执行的任务/线程以及栈追踪 |
| hexdump | 从内存地址转储HEX/ASCII |
| pci_cfg_(read\|write)(8\|16\|32) | 可以检查PCI设备的状态 |

### 2. 为KEXT创建符号信息

在GDB中调试你的KEXT之前，需要生成KEXT的符号信息。因为KEXT为动态加载，所以无法知道它在内存中的位置。尽管KEXT中包括符号信息，但函数和数据的地址都与KEXT二进制

相关。在KEXT加载之后，KEXT中函数的绝对地址是kernel_load_address + relative_address。

幸好，Kernel Debug Kit为我们提供了一小段脚本，帮助生成最终的符号表，包含内核地址空间中的绝对地址。该脚本为createsymbolfiles，用法如下所示：

```
$ /Volumes/KernelDebugKit/createsymbolfiles -a i386 -s ./ MyDebugDriver.kext
MyDebugDriver.kext appears to be loadable (not including linkage for on-disk libraries).

Enter the hexadecimal load addresses for these extensions
(press Return to skip symbol generation for an extension):

com.osxkernel.MyDebugDriver: 0x9b9000
```

加载地址可以在多个地方找到，例如使用kextstat，如下所示：

```
$ kextstat
Index Refs Address    Size       Wired      Name (Version) <Linked Against>
  126    0 0x9b9000   0x3000     0x2000     com.osxkernel.MyDebugDriver (1) <5 4 3>
```

在前面的示例中，KEXT并没有其他依赖，但在现实情况中，一个KEXT可能依赖于一个或多个其他的KEXT，如I/O Kit族，在这种情况下，还会提示你输入它们的地址。如果你正在从一个远程系统调试一个崩溃转储，或直接调试一个崩溃的系统，扩展的加载地址和依赖将会在内核恐慌日志中找到（如果你的扩展崩溃的话）。请注意，KEXT可能在每次加载时都被赋予不同的地址，因此，每次都要重新生成符号信息。

上面给出了足够的基本符号信息，但仅限于函数的符号名称，而无法提供给我们源代码行的信息。可以通过配置扩展的调试信息格式来获取该信息，如图16-2所示。

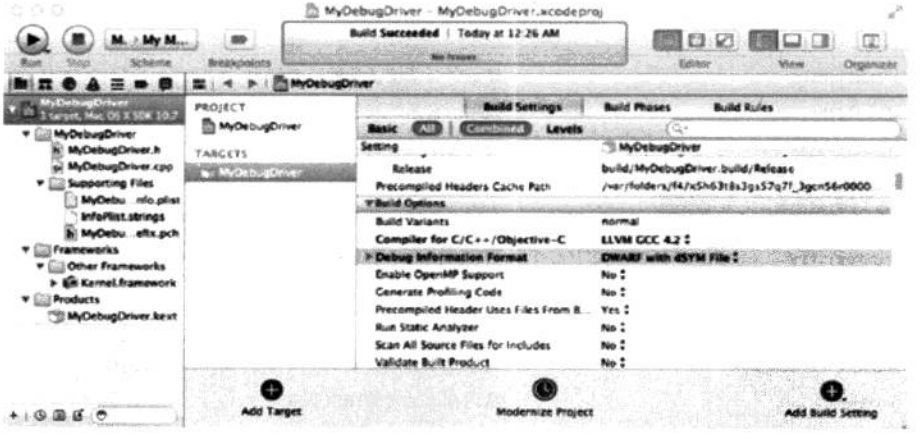



图16-2　使用Xcode设置调试信息格式

带有dSYM的DWARF为你创建了一个单独的文件，其中包含一个完整的符号集，以及将代码位置映射回源代码行所需的信息。构建调试版时，这些信息通常会嵌入到执行文件中，而你可以在一个外部文件中为驱动程序的发布版生成这些信息。一种很好的做法是归档该调试信息，供将

来使用。如果应该由用户报告崩溃，这将使调试变得更容易。在我们的示例中，dSYM文件（实际上为包）被命名为MyDebugDriver.kext.dSYM。在运行createsymbolfiles脚本之前，应该将dSYM包与KEXT放在一起。

### 3. 使用GDB调试KEXT

以下各节将分析一个由虚构驱动程序MyDebugDriver引起的崩溃。MyDebugDriver头文件的定义如代码清单16-4所示。

**代码清单16-4 MyDebugDriver头文件**

```
class com_osxkernel_MyDebugDriver : public IOService
{
    OSDeclareDefaultStructors(com_osxkernel_MyDebugDriver);
public:
    virtual bool init(OSDictionary* dict);
    virtual bool start(IOService* provider);
    virtual void stop(IOService* provider);

    void testFunc1(UInt32 arg1, UInt32 arg2, UInt32 arg3, UInt32 arg4);
    void testFunc2(UInt32 arg1, UInt32 arg2, UInt32 arg3, UInt32 arg4);
    void testFunc3(UInt32 arg1, UInt32 arg2, UInt32 arg3, UInt32 arg4);

    static void  timerFired(OSObject* owner, IOTimerEventSource* sender);
private:

    IOTimerEventSource*     fTimer;
    int                     fVariable1;
    int                     fVariable2;
};
```

该驱动程序启动一个定时器，从该定时器调用testFunc1()，该函数进而调用testFunc2()，testFunc2()调用testFunc3()，testFunc3()因空指针解引用而引起内核恐慌。每个函数接受4个无符号整数（随机选择），并且从第一个函数传递到最后一个函数时不发生改变。传递的参数值为：65261、48879、0和5380。

我们使用FireWire核心转储机制，从崩溃系统成功生成了一个核心转储，此前，使用正确的装载地址，为驱动程序构建了符号信息。我们现在开始亲自动手，将其载入GDB。查找驱动程序中的bug需要执行的步骤，如下所示。

- 使用正确的加载地址创建符号文件（参见最后一节）。
- 启动GDB，加载核心转储。
- 从内核添加符号信息。
- 加载Kernel Debug Kit的GDB宏。
- 将MyDebugDriver的KEXT二进制文件载入GDB。
- 加载MyDebugDriver的符号信息（如果GDB在KEXT的位置找不到符号信息的话）。
- 将MyDebugDriver源代码的位置告诉GDB（可选）。

以下是MyDebugDriver的完整调试会话：

```
$ gdb -c core-xnu-1504.15.3-0.0.0.0-fb3a74d3
GNU gdb 6.3.50-20050815 (Apple version gdb-1704) (Thu Jun 23 10:48:29 UTC 2011)
…
This GDB was configured as "x86_64-apple-darwin".
#0  0x002b1e3e in ?? ()
(gdb) add-symbol-file /Volumes/KernelDebugKit/mach_kernel
add symbol table from file "/Volumes/KernelDebugKit/mach_kernel"? (y or n) y
Reading symbols from /Volumes/KernelDebugKit/mach_kernel...Reading symbols from
/Volumes/KernelDebugKit/mach_kernel.dSYM/Contents/Resources/DWARF/mach_kernel...done.
done.
(gdb) source /Volumes/KernelDebugKit/kgmacros
Loading Kernel GDB Macros package.  Type "help kgm" for more info.
```

前面的命令加载核心转储文件，从Kernel Debug Kit为内核添加符号信息，最后加载GDB宏，该加载必须在内核符号信息正确初始化之后进行。

现在可以发送backtrace命令，查看系统崩溃的位置，如下所示：

```
(gdb) backtrace
#0  0x002b1e3e in Debugger (message=0x5dd7fc "panic")
#1  0x0021b837 in panic (str=0x59e3d0 "Kernel trap at 0x%08x, type %d=%s, registers:\nCR0:
0x%08x, CR2: 0x%08x, CR3: 0x%08x, CR4: 0x%08x\nEAX: 0x%08x, EBX: 0x%08x, ECX: 0x%08x, EDX:
0x%08x\nCR2: 0x%08x, EBP: 0x%08x, ESI: 0x%08x, EDI: 0x%08x\nE"...) at /SourceCache/xnu/xnu-
1504.15.3/osfmk/kern/debug.c:303
#2  0x002abf6a in panic_trap [inlined] () at :1052
#3  0x002abf6a in kernel_trap (state=0x46ee3e10) at /SourceCache/xnu/xnu-
1504.15.3/osfmk/i386/trap.c:1001
#4  0x002a1a78 in trap_from_kernel () at cpu_data.h:397
#5  0x009ba0b7 in last_kernel_symbol ()
#6  0x009ba356 in last_kernel_symbol ()
#7  0x009ba3b8 in last_kernel_symbol ()
#8  0x009ba45b in last_kernel_symbol ()
#9  0x005571d5 in IOTimerEventSource::timeoutAndRelease (self=0x2a17b0, c=0x5022071) at
/SourceCache/xnu/xnu-1504.15.3/iokit/Kernel/IOTimerEventSource.cpp:122
#10 0x00230235 in thread_call_thread (group=0x863ea0) at /SourceCache/xnu/xnu-
1504.15.3/osfmk/kern/thread_call.c:848
```

因为我们尚未加载自己的KEXT，所以符号#5~#8显然无效，调试器无法将该函数的地址解析为符号名。我们对此进行修订，将MyDebugDriver KEXT及其符号信息载入GDB，如下所示：

```
(gdb) add-kext MyDebugDriver.kext
Reading symbols from com.osxkernel.MyDebugDriver.sym...Reading symbols from
MyDebugDriver.kext.dSYM/Contents/Resources/DWARF/MyDebugDriver...done.
(gdb) directory MyDebugDriver/
Source directories searched: MyDebugDriver:$cdir:$cwd
```

现在已经加载了驱动程序及其符号信息，并通知GDB查找MyDebugDriver源代码的位置。内核本身的源代码位置硬连接到调试内核映像。如果你希望在GDB中显示源代码，则需要为你下载XNU源代码的目录/SourceCache/创建一个符号链接。

我们再次对加载的KEXT及符号执行backtrace命令，如下所示：

```
(gdb) backtrace
…
```

```
#5  0x009ba0b7 in com_osxkernel_MyDebugDriver::testFunc3 (this=0x4e80f80, arg1=65261,
arg2=48879, arg3=0, arg4=5380) at MyDebugDriver.cpp:14
#6  0x009ba356 in com_osxkernel_MyDebugDriver::testFunc2 (this=0x4e80f80, arg1=65261,
arg2=48879, arg3=0, arg4=5380) at MyDebugDriver.cpp:21
#7  0x009ba3b8 in com_osxkernel_MyDebugDriver::testFunc1 (this=0x4e80f80, arg1=65261,
arg2=48879, arg3=0, arg4=5380) at MyDebugDriver.cpp:27
#8  0x009ba45b in com_osxkernel_MyDebugDriver::timerFired (owner=0x4e80f80, sender=0xb63adc0)
at MyDebugDriver.cpp:64
...
```

这非常易读。现在已经确定了具体的调用栈，并且可以看到驱动程序中相关方法所在的文件及行号。还可以看到传递给该方法的参数，以及我们早些时候所选的对应值。我们跳到崩溃发生的第五个栈帧，进一步检查该崩溃，如下所示：

```
(gdb) frame 5
#5  0x009ba0b7 in com_osxkernel_MyDebugDriver::testFunc3 (this=0x4e80f80, arg1=65261,
arg2=48879, arg3=0, arg4=5380) at MyDebugDriver.cpp:14
14                      thisWillNotWork->fVariable1 = arg3;
Current language:  auto; currently c++
(gdb) print thisWillNotWork
$1 = (com_osxkernel_MyDebugDriver *) 0x0
```

我想已经发现了问题所在！我们正试图将一个值分配给成员变量fVariable1，而该对象未初始化。我们还可以列出testFunc3()的源代码，如下所示：

```
(gdb) list com_osxkernel_MyDebugDriver::testFunc3,15
9       void com_osxkernel_MyDebugDriver::testFunc3(UInt32 arg1, UInt32 arg2, UInt32 arg3, UInt32 arg4)
10      {
11          if (arg3 == 0)
12          {
13              com_osxkernel_MyDebugDriver *thisWillNotWork = NULL;
14              thisWillNotWork->fVariable1 = arg3;
15          }
```

嗯，那将永远不会被执行！已经找到了bug，它看起来仅在传递的第三个参数设置为0时才执行。

如果你只有符号信息，而没有将地址映射到指定源代码位置所需的调试信息，那么可以在GDB中使用disassemble命令，根据该方法的地址，显示其反汇编代码。我们来看一下testFunc3()的反汇编代码，如下所示：

```
(gdb) disassemble 0x009ba0b7
...
0x009ba09f <testFunc3Emmmm+35>:     mov    %eax,-0x1c(%ebp)
0x009ba0a2 <testFunc3Emmmm+38>:     mov    -0x18(%ebp),%eax
0x009ba0a5 <testFunc3Emmmm+41>:     cmp    $0x0,%eax
0x009ba0a8 <testFunc3Emmmm+44>:     jne    0x9ba0ba <testFunc3Emmmm+62>
0x009ba0aa <testFunc3Emmmm+46>:     movl   $0x0,-0x20(%ebp)
0x009ba0b1 <testFunc3Emmmm+53>:     mov    -0x18(%ebp),%eax
0x009ba0b4 <testFunc3Emmmm+56>:     mov    -0x20(%ebp),%ecx
0x009ba0b7 <testFunc3Emmmm+59>:     mov    %eax,0x54(%ecx)
0x009ba0ba <testFunc3Emmmm+62>:     add    $0x18,%esp
...
```

如果你不熟悉汇编，这看起来很烦人，但通过比较反汇编代码和原始源代码，可以推断出许

多信息。例如，cmp指令将eax寄存器的值与常量$0x0进行比较，由此我们可以准确猜出，这对应第11行的if语句。

尽管我们已经找到了问题所在，但要思考一下，为什么给第三个参数传递了0值。也许我们的驱动程序使用了一个内部状态来计算传递给testFunc3()的值。这种情况下，要进一步检查驱动程序在崩溃时刻的状态。因为testFunc3()是com_osxkernel_MyDebugDriver的成员方法，所以我们知道指向该类实例的指针总是作为this指针自动传递给成员函数。可以从前面的栈追踪，解引用this指针地址，如下所示：

```
(gdb) print *(com_osxkernel_MyDebugDriver*)0x4e80f80
$9 = {
  <IOService> = {
    <IORegistryEntry> = {
      <OSObject> = {
        <OSMetaClassBase> = {
          _vptr$OSMetaClassBase = 0x9baa00
        },
        members of OSObject:
        retainCount = 65537
      },
      members of IORegistryEntry:
      reserved = 0x4e71a40,
      fRegistryTable = 0xb693800,
      fPropertyTable = 0x503a400
    },
    members of IOService:
    reserved = 0x0,
    __provider = 0x4d17f00,
……
  },
  members of com_osxkernel_MyDebugDriver:
  fTimer = 0xb63adc0,
  fVariable1 = 2,
  fVariable2 = 4,
  static gMetaClass = {
    <OSMetaClass> = {
      <OSMetaClassBase> = {
        _vptr$OSMetaClassBase = 0x9ba980
      },
      members of OSMetaClass:
      reserved = 0xb63ba00,
      superClassLink = 0x85fac8,
      className = 0x4e5b5a0,
      classSize = 92,
      instanceCount = 1
    }, <No data fields>},
  static metaClass = 0x9ba000,
  static superClass = 0x9ba020
```

现在检查驱动程序实例的内部状态，可以看到成员变量fVariable1和fVariable2的值。此外还可以看到我们的类从元类信息退出的实例个数，并确定该驱动程序的引用计数。

### 4. 理解内核恐慌日志

崩溃发生之后，可以在/Library/Logs/DiagnosticReports/目录中找到内核恐慌日志，或通过从核心转储或与崩溃目标机器的远程GDB会话来提取。作为内核开发者，客户可能希望你能够分析他们的计算机所发送的内核恐慌日志，而你对这些计算机几乎不能进行实际访问。另外，客户可能并不愿意协助你获取核心转储。因此，能从日志中尽可能地理解和提取信息至关重要。我们来查看一下内核恐慌日志，以及能够从中提取的信息。上一节讨论的MyDebugDriver崩溃的内核恐慌日志，如代码清单16-5所示。

**代码清单16-5 MyDebugDriver崩溃的内核恐慌日志**

```
panic(cpu 1 caller 0xffffff80002c268d): Kernel trap at 0xffffff7f81345570, type 14=page fault,
registers:
CR0: 0x0000000080010033, CR2: 0x0000000000000090, CR3: 0x0000000000100000, CR4:
0x0000000000000660
RAX: 0x0000000000000000, RBX: 0x0000000000000000, RCX: 0x0000000000000000, RDX:
0x000000000000beef
RSP: 0xffffff808e973e80, RBP: 0xffffff808e973ea0, RSI: 0x000000000000feed, RDI:
0xffffff801ab61100
R8:  0x0000000000001504, R9:  0x000000000000beef, R10: 0x0000000000000000, R11:
0x0000000000001504
R12: 0xffffff7f8134591a, R13: 0xffffff800c81d200, R14: 0xffffff800c81d200, R15:
0xffffff800b735880
RFL: 0x0000000000010246, RIP: 0xffffff7f81345570, CS:  0x0000000000000008, SS:
0x0000000000000010
CR2: 0x0000000000000090, Error code: 0x0000000000000002, Faulting CPU: 0x1

Backtrace (CPU 1), Frame : Return Address
0xffffff808e973b40 : 0xffffff8000220702
0xffffff808e973bc0 : 0xffffff80002c268d
0xffffff808e973d60 : 0xffffff80002d7a3d
0xffffff808e973d80 : 0xffffff7f81345570
0xffffff808e973ea0 : 0xffffff7f813458b6
0xffffff808e973ed0 : 0xffffff7f81345914
0xffffff808e973f00 : 0xffffff7f813459f4
0xffffff808e973f40 : 0xffffff800063bc61
0xffffff808e973f70 : 0xffffff800023dafc
0xffffff808e973fb0 : 0xffffff8000820057
      Kernel Extensions in backtrace:
         com.osxkernel.MyDebugDriver(1.0)[FF6F45C8-68F8-3150-9C43-
99A2F19B3FB1]@0xffffff7f81345000->0xffffff7f81348fff

BSD process name corresponding to current thread: kernel_task
Boot args: debug=0xd44 _panicd_ip=192.168.1.1 panicd_ip=192.168.1.1

Mac OS version:
11A511

Kernel version:
Darwin Kernel Version 11.0.0: Sat Jun 18 12:56:35 PDT 2011; root:xnu-
1699.22.73~1/RELEASE_X86_64
Kernel UUID: 24CC17EB-30B0-3F6C-907F-1A9B2057AF78
```

```
System model name: MacBook5,1 (Mac-F42D89C8)

System uptime in nanoseconds: 200305435891999
last loaded kext at 200285007562702: com.osxkernel.MyDebugDriver  1 (addr 0xffffff7f81345000,
size 16384)
last unloaded kext at 187374587106276: com.apple.driver.AppleUSBCDC     4.1.15 (addr
0xffffff7f8133d000, size 12288)
loaded kexts:
com.osxkernel.MyDebugDriver  1
com.apple.driver.AppleUSBDisplays   302.1.2
com.apple.driver.AppleIntelProfile  83
com.apple.filesystems.afpfs 9.8
```

代码清单16-5中的内核恐慌日志在一个与之前系统不同的系统上生成。该系统运行新版本的Mac OS X Lion，并且只运行64位内核。该内核恐慌日志包含下列内容。

- 发生的内核恐慌/问题类型及其所在的CPU（核心）号。
- CPU状态的转储（寄存器值）。
- 崩溃时CPU执行作业的回溯。
- 崩溃涉及的内核扩展及其依赖（上面示例中并不存在）。
- 引起崩溃的进程（任务）名称。
- 内核编译版本及版本号。
- 系统模型。
- 关于最近所加载/卸载KEXT的信息。
- 所加载KEXT的完整列表。

你可能首先注意到该内核恐慌由缺页异常引起，这为随后在代码中的查找提供了入手点。查看引起该问题的任务通常也非常有用。在该示例中，崩溃发生时，驱动程序正在内核上下文（kernel_task）中执行，而且并不代表用户空间线程执行。

首先查看一个回溯，确定驱动程序确实存在问题。前面一个很好的机会是将我们的驱动程序列为回溯的一部分。你还会注意到驱动程序后面的两个地址：0xffffff7f81345000->0xffffff7f81348fff。这是我们KEXT的指令及数据载入内核地址空间的加载地址。为了确定栈上的哪些函数属于我们的驱动程序，可以只查找栈上该范围内的地址。它可以识别4个地址：0xffffff7f81345570、0xffffff7f813458b6、0xffffff7f81345914和0xffffff7f813459f4。

从前面了解到，这些值很可能对应testFunc3()、testFunc2()、testFunc1()和timerFired()。

---

**提示**　在前面的回溯中，第一列地址是栈帧入口地址，而第二列地址是返回地址，是前面的函数调用完成时的返回位置。在该示例中，我们关注的是返回地址。你可能已经发现栈帧地址包含增加地址，都位于一页中。如果该值看起来为随机产生，并且超过该位置，那么很可能是栈帧已经被损坏，该回溯已没有用，因为这些信息并不可信。

---

假如我们不知道这些地址对应驱动程序的哪些函数，可以使用一个简单的小技巧。通过从加载地址减去一个函数的地址，确定该函数在驱动程序可执行映像中的偏移量，如下所示：

```
0xffffff7f81345570 - 0xffffff7f81345000 = 1392字节
```

现在知道，该函数距离KEXT起始处有1392字节，如果我们有驱动程序的可用执行映像（该崩溃所涉及的准确版本和构建版本），就可以执行下列操作：

```
$ gdb MyDebugDriver.kext/Contents/MacOS/MyDebugDriver
GNU gdb 6.3.50-20050815 (Apple version gdb-1704) (Thu Jun 23 10:48:29 UTC 2011)
This GDB was configured as "x86_64-apple-darwin"...
(gdb) disassemble 1392
Dump of assembler code for function _ZN27com_osxkernel_MyDebugDriver9testFunc3Ejjjj:
0x0000000000000540 <testFunc3Ejjjj+0>:       push   %rbp
0x0000000000000541 <testFunc3Ejjjj+1>:       mov    %rsp,%rbp
0x0000000000000544 <testFunc3Ejjjj+4>:       sub    $0x20,%rsp
0x0000000000000548 <testFunc3Ejjjj+8>:       mov    %rdi,-0x8(%rbp)
0x000000000000054c <testFunc3Ejjjj+12>:      mov    %esi,-0xc(%rbp)
0x000000000000054f <testFunc3Ejjjj+15>:      mov    %edx,-0x10(%rbp)
0x0000000000000552 <testFunc3Ejjjj+18>:      mov    %ecx,-0x14(%rbp)
0x0000000000000555 <testFunc3Ejjjj+21>:      mov    %r8d,-0x18(%rbp)
0x0000000000000559 <testFunc3Ejjjj+25>:      mov    -0x14(%rbp),%eax
0x000000000000055c <testFunc3Ejjjj+28>:      cmp    $0x0,%eax
0x000000000000055f <testFunc3Ejjjj+31>:      jne    0x576
<_ZN27com_osxkernel_MyDebugDriver9testFunc3Ejjjj+54>
…
End of assembler dump.
(gdb) info line *1392
Line 14 of "MyDebugDriver.cpp" starts at address 0x569
<_ZN27com_osxkernel_MyDebugDriver9testFunc3Ejjjj+41> and ends at 0x576
<_ZN27com_osxkernel_MyDebugDriver9testFunc3Ejjjj+54>.
```

我们找到了崩溃位置！对CPU寄存器的详细描述超出了本书范围，但只要知道它们包含大量有用的信息就已足够。下一节讨论如何使用寄存器信息获取函数参数。该内核恐慌日志中的处理器运行在64位模式中。x86_64比i386系统包含更多的可用寄存器，局部变量通常在通用寄存器而不是栈中传递。

**提示** 技术说明2063讨论了调试、理解内核恐慌的细节，还包括调试PowerPC系统的内核恐慌日志：http://developer.apple.com/library/mac/#technotes/tn2063/_index.html。

### 5. x86-64调用规约

调用规约是函数传递参数的一种方案。调用规约依赖编程语言、操作系统、体系结构和编译器。理解使用的调用规约可以帮助我们解码崩溃发生时的寄存器状态。例如，在运行64位可执行代码或内核的Mac OS X上，使用System V AMD64 ABI规约（请注意，Windows使用不同的调用规约，因此寄存器的使用也不同）。在64位任务的Mac OS X上，函数调用参数的寄存器分配，如表16-4所示。

表16-4　x86_64上函数参数的寄存器使用

| 参　数 | 寄 存 器 |
| --- | --- |
| 第一个参数 | RDI |
| 第二个参数 | RSI |
| 第三个参数 | RDX |
| 第四个参数 | RCX |
| 第五个参数 | R8 |
| 第六个参数 | R9 |
| 向量/浮点参数 0 ~ 7 | XMM0~XMM7 |

如果一个函数带有6个以上的参数，那么其余的参数将在栈上传递。检查C++代码时需要考虑的是，非静态C++成员方法总是将this指针作为第一个参数传递，因此，该方法的第一个实际参数是在RDI中传递，而this指针将被放在寄存器RSI中。

看一下代码清单16-5所示的内核恐慌日志中的寄存器，通过检查寄存器的状态，计算出哪些参数传递给了函数。

- RDI：0xffffff801ab61100（this指针）。
- RSI：0x000000000000feed（十进制=65261）。
- RDX：0x000000000000beef（十进制=48879）。
- RCX：0x0000000000000000（十进制=0）。
- R8：0x0000000000001504（十进制=5380）。

可以看到，代码清单16-5的寄存器内容精确匹配了传递给testFunc3()的四个参数：65261、48879、0和5380。我们还可以看到，第一个参数看起来像一个指针，很可能是this指针，表示MyDebugDriver的当前实例。

假如testFunc3()是一个更长、更复杂的方法，并且崩溃发生在该函数的更深处，寄存器崩溃的位置很可能已经被重新使用和覆写。在这种情况下，可能无法恢复参数的原始值。

### 16.3.11　使用Activity Monitor诊断挂起进程

Mac OS X的Activity Monitor可以帮助诊断内核问题，如图16-3所示。

Activity Monitor显示的任务都可以采样（kernel_task除外），生成采样时该任务所有线程的调用图。这对调试非常有用，可以看到一个进程的线程是否正在通过系统调用（如IOConnectCallMethod()）调入你的驱动程序。如果进程挂起，强制其退出将会启动Crash Reporter，它会给你更详细的日志，包括线程的内核栈（如果该线程运行在内核中）。采样进程函数还可以帮助你确定性能问题，找出应用程序最耗时的地方。

图16-3 Activity Monitor的进程输出示例

## 16.3.12 查找内存和资源泄漏

防止内存和资源泄漏对运行时动态加载和卸载的扩展来说特别重要。一个有用的检测内存的工具是ioclasscount实用程序，它会显示每个已知类（载入内核的）的实例计数。典型的输出看起来如下所示：

```
CHUDUtils = 1
com_apple_AppleFSCompression_AppleFSCompressionTypeZlib = 1
com_apple_BootCache = 1
com_apple_driver_AudioIPCDevice = 1
com_apple_driver_AudioIPCEngine = 1
com_belkin_f2cd0007_adapter = 0
com_osxkernel_MyDebugDriver = 5
com_vmware_kext_KeyboardState = 1
com_vmware_kext_UsbDevice = 2
com_vmware_kext_UsbPortArbiter = 1
.....
```

这显示我们的驱动程序已经被引用5次（retain()）。该驱动程序的free()函数在引用计数变为0之前，将不会被调用，即使其控制的硬件设备被移除。这将防止内核扩展被完全卸载。引用计数通常在用户空间应用程序打开该驱动程序，或另一个驱动程序或该驱动程序使用的辅助支持类对其调用retain()时增加。如果调用release()的次数与retain()的次数不相等，将会导致泄漏。无须改变内核启动参数，便可以使用ioclasscount实用程序，系统上默认并未安装该实用程序，它作为Xcode发布版的一部分安装。该实用程序可以被复制到没有Xcode的系统上，用于调试。已经被卸载且其所有引用都已经被释放（引用计数=0）的驱动程序，将由kextd守护进程卸载。尽管引用计数已经减少至0，但完全卸载KEXT可能会花费一分钟。

如果你能够使用GDB实时调试内核，那么可以使用showallclasses或showregistry等宏，如下所示：

```
(gdb) showallclasses
…
1    x    84 bytes com_vmware_kext_VmmonService
2    x    80 bytes com_vmware_kext_UsbDevice
1    x   104 bytes com_vmware_kext_UsbPortArbiter
1    x   136 bytes com_vmware_kext_UsbPortArbiterUserClient
1    x   132 bytes com_vmware_kext_KeyboardState
1    x    92 bytes com_osxkernel_MyDebugDriver
....
(gdb) showregistry
...
      +-o com_osxkernel_MyDebugDriver  <object 0x0703e680, id 0x1000007a9, vtable 0xcbc980,
!registered, !matched, active, busy 0, retain count 5>
...
```

**提示**　zprint和showioalloc宏可以进一步辅助跟踪内存使用。

为了进一步帮助调试引用计数bug，可以覆写OSObject::taggedRetain(const void *tag)和OSObject::taggedRelease(const void *)。例如，输出消息或调用栈的回溯，帮助确定泄漏的位置。

## 16.4　小结

- 内核中很可能会发生一些常见问题，如死锁和无效的内存访问，进而引起内核恐慌。
- 内核恐慌是对内核无法恢复的异常或错误情形的一种防御机制。这一般会禁用系统，防止破坏文件系统或其他的文件存储。
- Mac OS X提供了大量有用的专用调试机制，从简单的追踪和记录机制到内置的远程内核调试。
- Mac OS X可以使用kdumpd进行配置，当远程系统崩溃（或手工触发）时，接受其核心转储。核心转储包含活动/固定的内存，可以被载入GDB。
- 内核可以通过FireWire和以太网从远程系统进行调试。该机制为系统自带，但默认情况下并未被激活。可以通过设置适当的NVRAM参数和标志启用远程调试。
- 苹果通常为Mac OS X的每个发布版提供一个Kernel Debug Kit。该工具包包含脚本、内核的调试版及I/O Kit族KEXT。调试工具包还包含简化GDB内核调试的宏。这些宏可以获取调用栈信息，检查内核的关键数据结构。内核与GDB运行在同一个机器上时，还可以使用GDB进行实时调试。
- 为了调试你自己的KEXT，必须生成该KEXT的调试符号。因为KEXT在内核中动态加载，所以需要生成KEXT的正确符号地址。Kernel Debug Kit提供了createsymbolfiles来实现该处理。
- 内核恐慌日志包含许多有用的信息，可以用来回溯和查找引起崩溃的位置。
- ioclasscount工具跟踪I/O Kit中类的实例计数，并且可以检测内存泄漏或其他问题。

# 第 17 章 高级内核编程

本章主要涵盖一些高级内核开发者比较感兴趣的主题。我们将讨论内核中SSE（Streaming SIMD Extension，流式SIMD扩展）及浮点的使用方式（SIMD是Single Input Multiple Data的缩写）。我们还将查看多功能设备驱动程序的处理策略，并讨论I/O Kit族的实现。本章将讲述内核控制KPI，它用来与KEXT（如NKE）进行内核空间通信，而不是使用I/O Kit。我们还将讲述如何从内核处理及操作进程，如获取进程的PID，向进程发送通知。某些驱动程序可能需要从文件系统加载其他资源，如固件映像。本章将讨论如何使用`OSKextRequestResource()`函数加载这些资源。本章结尾还将讨论驱动程序如何使用通知将消息发送给用户空间守护进程。

## 17.1 内核中的 SSE 和浮点

SSE是继MMX之后的一种特殊指令集，存在于大多数现代英特尔和AMD处理器上，这些处理器支持对数组（矢量）而不是单个值（标量）执行`add`、`multiply`及`shift`等指令。这能够大大提高许多计算任务的执行速度，特别是在数字信号处理、音频、图像和视频领域。通常，内核中并不执行大量计算，但在某些领域却不可避免，例如，就音频驱动程序而言，它需要转换多个用户空间应用程序的音频采样，并将这些采样混合成一个缓冲区输出。RAID-5和RAID-6算法的软件实现也需要在内核中进行，并且可以使用SSE进行优化。

通常，SSE和浮点在内核中并不常用。某些操作系统（如Linux）需要显式保存浮点/SSE状态，并在使用之后将其恢复，否则，线程的浮点状态可能会被覆盖。但在Mac OS X中，内核可以自由使用浮点和SSE，而无须手动保存和恢复寄存器的状态。通常，当一个线程完成其时间片或被另一个线程抢占时，线程停止执行时CPU寄存器的状态会被保存至内存，并在线程继续执行时恢复。为了优化性能，内核只存储通用寄存器，而不是浮点和SSE寄存器，因为它们并不常用。实际上，许多程序根本不使用浮点和SSE。另一个线程要使用该寄存器时，CPU会发送一个异常/中断，以保存之前的内容，并清除该寄存器，供新线程使用。当原始线程恢复执行时，浮点和SSE寄存器将恢复至之前的状态。

SSE的用法有两种。

- 通过使用汇编或内联汇编，直接使用CPU指令。
- 使用GCC提供的内在函数，GCC提供了大部分指令的C函数包装器，便于使用。对于SSE2，

这些包装器位于头文件emmintrin.h中，该头文件并不是内核框架的一部分。但你可以复制该文件，使之包括在你的工程中。可以这么做，因为这些函数均为内联函数，不依赖任何外部库。

SSE指令集有许多版本。目前最新的主要版本是SSE4。SSE4出现于2007年，一些旧系统可能并不支持SSE4。试图在不支持SSE4的CPU上执行SSE4指令，将会导致内核恐慌。为了防止该问题，在执行SSE指令之前，需要提供CPU功能的运行时检测，或采用较早的版本，如SSE2，它比所有基于英特尔的Mac出现得都要早。

## 17.2　多功能驱动程序

USB和PCI设备可能是包含多个独立设备的复合设备，其中每个设备的驱动程序可以单独处理一个功能。其他设备可能包含一个由多个驱动程序处理的逻辑设备。我们来看一下带有HDMI输出端口的现代显卡，并以它为例。HDMI能够传送音频和视频，因此，提供支持设备使用Core Audio的音频驱动程序，将会非常理想。该设备是一个显卡，因此，该硬件并不具有大部分音频硬件普遍拥有的DMA引擎。相反，音频数据与视频帧一起，以固定的垂直场消隐间隔发送。这种设计意味着音频和视频部分紧密相连，为了操作，二者之间需要一个共享状态。由于没有明显的界限，该驱动程序的结构如图17-1所示。

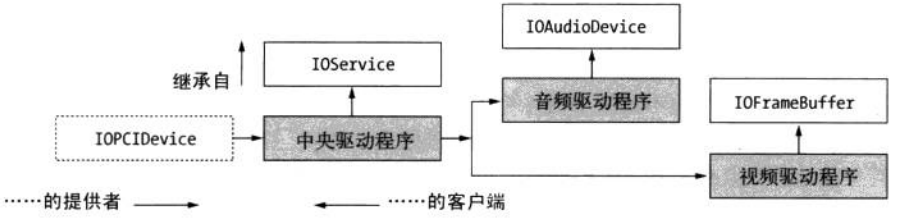



图17-1　多功能驱动程序

图17-1中的设计使用了一个中央驱动程序，来协调硬件并管理提供者。中央驱动程序匹配硬件提供者。该中央驱动程序随后创建一个基于IOAudioFamily的音频驱动程序，以及一个基于IOGraphicsFamily的视频/图形驱动程序。中央驱动程序与附属驱动程序之间的关系管理方式有两种。

- 中央、音频和视频驱动程序位于独立的KEXT中。中央驱动程序匹配硬件资源，而音频和视频驱动程序将匹配中央驱动程序，并将其作为提供者。
- 所有的驱动程序都位于同一个KEXT中。中央驱动程序需要手动管理从属驱动程序的生命周期。

## 17.3　编写 I/O Kit 族

现在为止，我们已经查看了各种驱动程序的实现方式，其中大部分都与一个或多个族交互。

某些情况下，你可能希望实现一个族，而不是驱动程序，例如，你需要支持一种新的总线技术，或拥有一个硬件设备族，其中这些硬件设备都依赖于同一个基础设施或通用服务。族区别于驱动程序的特性有：

- 族通常包括多个IOService类，这些类捆绑在提供相关服务的KEXT中；
- 族可以看做用户空间共享库的内核模拟；
- 驱动程序依赖于族，而不是族依赖于驱动程序；
- 族并不直接加载，驱动程序已经表示依赖时，该族才被加载；
- 族并没有匹配字典，不参与被动或主动匹配；
- 与任何其他的KEXT一样，族安装在/System/Library/Extensions目录中。

并没有特殊的API或方案可以用来编写族。族的实现方式与任何其他驱动程序一样。苹果公司建议关注基础良好的面向对象设计，并允许其进行自然演变，而不是特意着手创建一个族。驱动程序可以在其属性列表文件中使用OSBundleLibraries键来表示对族（或任何其他的KEXT）的依赖。在所有的依赖被解析并加载之前，驱动程序将无法加载并链接到内核。kextd守护进程负责执行该处理。驱动程序需要检查时，kextd守护进程将检查驱动程序Info.plist文件中的依赖。如果Info.plist指定错误或未能列出某些依赖，将会导致链接错误，那么该驱动程序将不会加载。为了依赖一个族，驱动程序必须列出该族的程序包ID，例如com.apple.iokit.IOAudioFamily，以及版本号，例如1.7.9fc8。

I/O Kit必须保证依赖族在依赖于它的驱动程序之前进行加载，否则驱动程序中的符号将无法解析。

由于苹果公司的许多I/O Kit族都属开源，所以可以对这些族进行修改，以替换原始版本。但我们并不建议这么做，因为KEXT族很可能会被苹果公司后续软件的更新所覆盖，这样，被修改的KEXT可能就会丢失功能。在某些情况下，通过插入额外的跟踪来修改一个族可能会帮助你进行调试。

相对于直接修改一个族来说，对族进行扩展是更好的选择。扩展非常容易，因为I/O Kit中的大部分类都将其方法声明为虚函数，即使该类本身为非抽象类。你可能希望这样做的原因有很多。例如，如果你需要支持一种新型的USB控制器，而IOUSBFamily并不支持它，那么可以创建自定义的IOUSBController子类，表示这个新控制器。扩展的类可以编译进需要该类的驱动程序的同一个KEXT中，或你自己的库/族KEXT中。IONetworkingFamily及其他族都专门用于此种形式的扩展。

## 17.4 内核控制 KPI

内核控制接口<sys/kern_control.h>是一个KPI，允许KEXT与用户空间进程进行双向通信。内核控制系统位于内核的BSD部分，因而是采用C而不是C++编写（I/O Kit使用C++）。

该KPI允许用户空间程序控制和配置KEXT。例如，假定你已经实现了一个自定义防火墙NKE（Network Kernel Extension，网络内核扩展）。然后可以使用该内核控制API告诉你的防火墙应该阻塞哪些地址或端口，以及获取日志和统计信息。

该KPI在内核空间和用户空间中的使用相对来说都比较简单。实际上，对于用户空间来说，并没有专门的API用来使用内核控制机制，因为该机制通过普通套接字进行访问。getsockopt()或setsockopt()系统调用函数可以用来发送用户空间的控制请求。内核控制系统类似于ioctl()系统调用，但又与之不同，内核控制系统更适合在内核/用户空间边界传输大量数据。通过使用用户空间的send()和recv()系统调用函数发送和接收数据。在内核中，数据传输使用先前讨论过的mbuf数据结构进行处理。

为了使用内核控制接口，首先必须注册一个新的接口，确保用户空间客户端可以找到它，并且可以与它连接。这通过声明和填充一个包含各种事件回调的C结构体，以及一个标识名称实现。C语言不是面向对象的语言，因此，“对象”通常由包含数据和函数指针的结构体表示。该注册结构如代码清单17-1所示。

**代码清单17-1　<sys/kern_control.h>中的kern_ctl_reg结构**

```
struct kern_ctl_reg
{
    /*控制信息*/
    char                    ctl_name[MAX_KCTL_NAME];
    u_int32_t               ctl_id;
    u_int32_t               ctl_unit;
    /*控制设置*/
    u_int32_t               ctl_flags;
    u_int32_t               ctl_sendsize;
    u_int32_t               ctl_recvsize;
    /*调度函数*/
    ctl_connect_func        ctl_connect;
    ctl_disconnect_func     ctl_disconnect;
    ctl_send_func           ctl_send;
    ctl_setopt_func         ctl_setopt;
    ctl_getopt_func         ctl_getopt;
};
```

我们详细看一下该数据结构的各的字段。

- ctl_name应该设置为KEXT的程序包ID。
- ctl_id字段用于传统的寻址，因为一个KEXT可能同时注册多个内核控制。该字段可以由苹果公司的开发者技术支持（Developer Technical Support，DTS）进行动态注册或分配。
- ctl_unit字段只用于DTS分配的ID。ctl_flags只有两个标志。第一个是CTL_FLAG_PRIVILEGED，设置之后，用户空间程序必须拥有root权限，才能连接到内核控制。第二个标志是CTL_FLAG_REG_ID_UNIT，应该在使用DTS分配的ID时指定。
- ctl_sendsize和ctl_recvsize字段可以调整使用send()和recv()发送数据的发送及接收缓冲区的长度。

其余字段是函数指针，在相应事件发生时调用。

- ctl_connect和ctl_disconnect回调在用户空间客户端连接或断开时调用。

- ctl_setopt和ctl_getopt回调在客户端使用setsockopt()或getsockopt()函数时调用。这两个回调通常用来获取或设置配置参数。下一个回调是ctl_send，它可能有些令人困惑，因为它并不是用来发送数据，而是从发送客户端接收数据。实际发送数据使用的是ctl_enqueuedata()函数。

### 17.4.1 内核控制注册

我们来看一个使用内核控制接口的例子（HelloKernControl）。该示例将实现一个最基本的内核控制，一个获取操作和一个设置操作。获取操作返回内核中存储的一个字符串。设置操作重写该字符串，以便随后的获取操作返回这个新字符串。以下是一个填写好的内核控制注册结构体的示例。

```
static struct kern_ctl_reg g_kern_ctl_reg = {
    "com.osxkernel.HelloKernControl",
    0,
    0,
    CTL_FLAG_PRIVILEGED,
    0,
    0,
    hello_ctl_connect,
    hello_ctl_disconnect,
    NULL,
    hello_ctl_set,
    hello_ctl_get
};
```

我们使用一个动态分配的ID，指定我们的内核控制只可以由有权限的客户端（root）访问。我们定义了4个回调，但将ctl_send回调设为NULL，因为该示例并不支持该回调。以下是注册及注销内核控制的代码：

```
static boolean_t g_filter_registered = FALSE;
static kern_ctl_ref g_ctl_ref;

kern_return_t HelloKernControl_start (kmod_info_t* ki, void* d)

{
    strncpy(g_string_buf, DEFAULT_STRING, strlen(DEFAULT_STRING));

    /*注册控制*/
    int ret = ctl_register(&g_kern_ctl_reg, &g_ctl_ref);

    if (ret == KERN_SUCCESS)
    {
        g_filter_registered = TRUE;
        return KERN_SUCCESS;
    }
    return KERN_FAILURE;
}
```

```
kern_return_t HelloKernControl_stop (kmod_info_t* ki, void* d)
{
    if (g_clients_connected != 0)
        return KERN_FAILURE;

    if (g_filter_registered)
        ctl_deregister(g_ctl_ref);

    return KERN_SUCCESS;
}
```

我们在KEXT的start()函数中注册该接口，并在stop()函数中注销该接口，它在KEXT卸载之前调用。因为内核控制通常与用户空间共享一些数据，所以需要定义一个共享的头文件，来存储内核和用户空间都使用的公共声明。HelloKernControl的共享头文件，如下所示：

```
#ifndef HelloKernControl_HelloKernControl_h
#define HelloKernControl_HelloKernControl_h

#define BUNDLE_ID "com.osxkernel.HelloKernControl"

#define HELLO_CONTROL_GET_STRING  1
#define HELLO_CONTROL_SET_STRING  2

#define DEFAULT_STRING            "Hello World"
#define MAX_STRING_LEN            256

#endif
```

## 17.4.2　客户端连接

连接和断开回调的实现如下所示：

```
static int hello_ctl_connect(kern_ctl_ref ctl_ref, struct sockaddr_ctl *sac, void** unitinfo)
{
    printf("process with pid=%d connected\n", proc_selfpid());
    return 0;
}
static errno_t hello_ctl_disconnect(kern_ctl_ref ctl_ref, u_int32_t unit, void* unitinfo)
{
    printf("process with pid=%d disconnected\n", proc_selfpid());
    return 0;
}
```

在前面的示例中，hello_ctl_connect()函数输出所打开内核控制的客户端PID。维护每个客户端的数据结构通常都很必要。该数据结构应该分配给unitinfo参数，例如：*unitinfo = myStructure;。该结构现在可以在其他回调中获取。如果你在客户端连接时分配内存，那就应该在断开回调时释放内存。如果你希望拒绝一个客户端，例如，因为在某一时刻只可以连接一个客户端，或已经连接了最大数量的客户端，那么可以仅返回一个错误码，如EBUSY或EPERM。

### 17.4.3 获取和设置选项

客户端成功连接之后，便可以将获取/设置选项请求发送给内核控制。控制获取函数的实现，如下所示：

```
static int hello_ctl_get(kern_ctl_ref ctl_ref, u_int32_t unit, void *unitinfo, int opt,
                         void *data, size_t *len)
{
    int ret = 0;
    switch (opt) {
        case HELLO_CONTROL_GET_STRING:
            *len = min(MAX_STRING_LEN, *len);
            strncpy(data, g_string_buf, *len);
            break;
        default:
            ret = ENOTSUP;
            break;
    }
    return ret;
}
```

opt参数来自客户端，指定客户端对哪一个选项感兴趣。常见的一种方案是创建一个共享头文件，其中包含KEXT和用户空间程序之间共享选项的定义。

**警告** 要注意共享的数据结构，因为KEXT和用户空间程序可能对该数据结构填充不同的内容。这会引起bug、损坏，或者更糟糕的情况。

上面的示例仅处理一个选项。该选项由HELLO_CONTROL_GET_STRING定义，返回所有客户端之间共享的全局变量g_string_buf中的字符串。如果在连接回调时已经分配了私有数据，那么可以通过转换unitinfo参数的数据类型来获取私有数据。

为了将字符串返回给客户端，需要将其复制到data参数指定的内存地址中。len参数是一个输入/输出参数，包含data缓冲区的长度。显然，你必须确保不会写出边界。如果你写入该缓冲区，那么应该修改len，以反映实际写入了多少字节。

设置选项函数的实现非常相似：

```
static int hello_ctl_set(kern_ctl_ref ctl_ref, u_int32_t unit, void* unitinfo, int opt,
                         void* data, size_t len)
{
    int ret = 0;
    switch (opt) {
        case HELLO_CONTROL_SET_STRING:
            strncpy(g_string_buf, (char*)data, min(MAX_STRING_LEN, len));
            printf("HELLP_CONTROL_SET_STRING: new string set to: \"%s\"\n", g_string_buf);
            break;
       default:
            ret = ENOTSUP;
            break;
```

```
    }
    return ret;
}
```

类似于控制获取选项函数，我们将接收到一个带有用户空间数据的缓冲区，以及该缓冲区的长度，分别是data和len参数。在该函数返回之后，数据便会失效，因此，必须复制保存你想要的任何数据。

### 17.4.4　从用户空间访问内核控制

代码清单17-2示例演示了如何连接上一节所描述的内核控制接口。

**代码清单17-2　连接内核控制接口的用户空间工具**

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>

#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/kern_control.h>
#include <sys/sys_domain.h>

#include "HelloKernControl.h"

int main(int argc, char* const*argv)
{
    struct ctl_info ctl_info;
    struct sockaddr_ctl sc;
    char str[MAX_STRING_LEN];

    int sock = socket(PF_SYSTEM, SOCK_DGRAM, SYSPROTO_CONTROL);
       if (sock < 0)
       return -1;

    bzero(&ctl_info, sizeof(struct ctl_info));
    strcpy(ctl_info.ctl_name, "com.osxkernel.HelloKernControl");

    if (ioctl(sock, CTLIOCGINFO, &ctl_info) == -1)
       return -1;

    bzero(&sc, sizeof(struct sockaddr_ctl));
    sc.sc_len = sizeof(struct sockaddr_ctl);
    sc.sc_family = AF_SYSTEM;
    sc.ss_sysaddr = SYSPROTO_CONTROL;
    sc.sc_id = ctl_info.ctl_id;
    sc.sc_unit = 0;

    if (connect(sock, (struct sockaddr *)&sc, sizeof(struct sockaddr_ctl)))
        return -1;
```

```
    /*从内核获取现有的字符串*/
    unsigned int size = MAX_STRING_LEN;
    if (getsockopt(sock, SYSPROTO_CONTROL, HELLO_CONTROL_GET_STRING, &str, &size) == -1)
        return -1;

    printf("kernel string is: %s\n", str);

    /*设置一个新的字符串*/
    strcpy(str, "Hello Kernel, here's your new string, enjoy!");
    if (setsockopt(sock, SYSPROTO_CONTROL, HELLO_CONTROL_SET_STRING,
                   str, (socklen_t)strlen(str)) == -1)
        return -1;

    close(sock);

    return 0;
}
```

执行代码清单17-2中的程序，应该会看到如下结果：

```
$ sudo kextload HelloKernControl.kext

$ sudo ./hello_tool
kernel string is: Hello World
$ sudo ./hello_tool
kernel string is: Hello Kernel, here's your new string, enjoy!
```

## 17.5 内核中的进程处理

内核的BSD部分提供了一个KPI，以获取系统中活动进程的信息。请注意，BSD中使用进程这一术语，而不是任务，任务这个概念用于内核的Mach部分，尽管它们实际上指同一个东西。

**内核私有KPI**

如果查看内核头文件，可能会看到预处理器指令KERNEL_PRIVATE。在这些部分定义的函数或其他符号并不能用于第三方内核扩展，即使它包含了正确的头文件，但如果使用这些函数或符号，仍会因无法解析而导致加载KEXT失败。苹果公司自己的KEXT通过向com.apple.kpi.private添加一个依赖，能够访问这些符号。如果你在自己的KEXT中为该KPI添加一个依赖，会加载失败，因为只有苹果公司的KEXT才可以使用。

在第13章的AppWall示例中，已经看到了如何获取进程信息，我们使用proc_selfname()函数获取当前运行进程的进程名称。如果该函数在内核拥有的线程中调用，则将返回内核进程名称kernel_task。

如果需要知道当前运行进程的PID，而不是名称，那么可以调用proc_selfpid()。如果你已

经知道了进程的PID，还可以使用proc_name(int pid, char *buf, int size);函数获取该进程的名称。进程KPI中的函数概述，如表17-1所示。

表17-1　进程操作函数（完整的列表请参见sys/proc.h）

| 函　数 | 描　述 |
| --- | --- |
| int proc_selfpid(void) | 返回当前进程的PID |
| int proc_selfppid(void) | 返回当前进程父进程的PID |
| void proc_signal(int pid, int signum) | 向指定PID的进程发送通知（例如，SIGTERM、SIGKILL） |
| int proc_issignal(int pid,sigset_t mask) | 检查mask指定的任意通知对指定PID的进程是否挂起 |
| int proc_isinferior(int pid1, int pid2) | 如果pid1从属于pid2，则返回1 |
| void proc_name(int pid, char *buf, int size) | 将进程名称复制到buf。如果该名称比size短，则将被截断 |
| void proc_selfname(char *buf, int size) | 与上一个函数相同，但用于当前进程 |
| proc_t proc_find(int pid) | 获取指定PID进程的进程句柄。这会添加一个该进程的引用，该引用必须使用proc_rele()释放 |
| proc_t proc_self(void) | 返回当前进程的进程句柄 |
| int proc_rele(proc_t p) | 释放进程句柄p |
| int proc_pid(proc_t p) | 返回进程p的PID |
| int proc_ppid(proc_t p) | 返回p父进程的PID |
| int proc_is64bit(proc_t p) | 如果进程运行在64位地址空间，则返回1 |
| int proc_exiting(proc_t p) | 如果进程退出，则返回1 |
| int proc_suser(proc_t p) | 如果进程正在使用超级用户权限运行，则返回1 |

## 17.6　加载资源

I/O Kit不提供驱动程序用来访问文件系统的任何类或函数。这是一种有意为之的设计，并不是I/O Kit设计的疏忽。理论上，驱动程序无须访问磁盘上的文件。驱动程序的作用是响应操作系统的请求，管理它拥有的硬件设备，而不是启动自身的请求。但实际上，因为种种原因，驱动程序可能需要访问文件系统的数据。最常见的一种情形是读取资源数据，如驱动程序硬件的固件数据。

尽管I/O Kit无法访问通用文件系统，但它提供了一种方法，使驱动程序可以访问驱动程序包中Resources目录中的文件。I/O Kit的资源加载API定义在头文件<libkern/OSKextLib.h>中。该API为异步的，驱动程序请求它希望加载的资源，并提供一个回调函数，I/O Kit用来通知驱动程序数据何时可用。

前面提到，I/O Kit并不提供通用文件系统访问，但除此之外，I/O Kit本身也不能访问文件系统。为了为驱动程序加载资源文件，I/O Kit依赖用户空间辅助进程，该辅助进程代表I/O Kit来读取请求的文件，并将文件内容传回给I/O Kit。I/O Kit随后通知请求的驱动程序。

由于I/O Kit依赖于用户空间辅助进程加载资源，所以在辅助进程启动之前，将无法启动进程

中加载的资源。但在大多数情况下，这并不会引起驱动程序问题，因为I/O Kit将对请求进行排队，直至辅助进程接收内核的请求。

驱动程序可以调用OSKextRequestResource()函数，请求I/O Kit所加载驱动程序资源目录下的文件。该函数的定义如下所示：

```
OSReturn OSKextRequestResource(const char* kextIdentifier,
                               const char* resourceName,
                               OSKextRequestResourceCallback  callback,
                               void* context,
                               OSKextRequestTag* requestTagOut);
```

第一个参数kextIdentifier，指定包含加载资源的驱动程序的包标识符，这几乎总是驱动程序Info.plist文件的CFBundleIdentifier键所指定的值。第二个参数resourceName是从驱动程序包加载资源文件名。接下来的两个参数是回调函数和相关的上下文参数，资源已被加载时，将该上下文参数传递给该回调函数。最后一个参数requestTagOut会及时返回给调用者，用来跟踪资源加载操作。

如果成功调用OSKextRequestResource，请求完成时，驱动程序将通过其指定的回调函数得到通知。完成回调的签名如下所示：

```
typedef void (*OSKextRequestResourceCallback)(OSKextRequestTag requestTag,
                                               OSReturn result,
                                               const void* resourceData,
                                               uint32_t resourceDataLength,
                                               void* context);
```

提供给该回调的第一个参数requestTag确定该完成回调所涉及的资源。第二个参数result通知调用者操作是否成功完成。如果result的值为kIOReturnSuccess，则已经成功从磁盘读出资源数据，接下来的两个参数resourceData和resourceDataLength包含所请求资源文件的内容。resourceData缓冲区仅在该回调中有效，因此，如果驱动程序希望在该回调之外引用资源数据，则必须生成该数据的一个副本。最后一个参数context包含传递给OSKextRequestResource函数的上下文参数值。

剩下的步骤是将资源文件添加到驱动程序包。通过OSKextRequestResource函数加载的任意资源必须位于驱动程序包的Resources子目录中。大多数情况下，这可以通过将文件添加到驱动程序的Xcode工程中实现。对于除源代码之外的文件类型，构建工程时，Xcode会默认将该文件复制到程序包的资源目录中。

## 17.7 KEXT 资源之外的内容

上一节讨论的资源加载函数为特定目的而设计。利用I/O Kit函数，驱动程序可以只读访问其Resources目录中的文件内容。但在许多情况下，驱动程序还需要访问其包之外的文件，以及写入到磁盘的文件。例如，提供持久化设置的驱动程序需要某些方法从磁盘上的文件读取这些设置，它还需要将这些设置写入磁盘。

尽管I/O Kit并不包含为驱动程序提供这种功能的函数，但其资源加载函数的实现为我们提供了将这种功能添加到我们驱动程序的线索。正如I/O Kit依赖用户空间守护进程来代替内核加载资源文件一样，驱动程序可以通过提供它拥有的用户空间守护进程来代替它处理请求，读写磁盘文件的持久化设置。这种设计（下一节讲述）开创了一种更通用的方案，除驱动程序首选设置之外该，方案还能进行其他拓展。

## 17.8　内核驱动程序通知

解决从内核驱动程序访问任意文件系统项这一问题的一般方案是实现一个用户空间守护进程，使其作为代表内核驱动程序的辅助进程。该进程处理驱动程序的请求，执行指定操作，然后将操作结果返回给驱动程序。该方案除了可以执行读写文件等请求之外，还可以执行内核中无法进行的操作，如向用户显示对话框。内核中一个用户交互的好例子就是标准USB族，它在设备连接时会显示警告对话框，提示需要的电源已超过USB总线所能发送的电源。

---

**注意**　IOUSBFamily目前使用一个弃用的接口（称为内核－用户通知中心）来显示警报消息，如低电源警告。I/O Kit提供了一个API，使驱动程序可以通过标准系统守护进程显示对话框。但该API现在已经弃用，驱动程序目前必须提供它自己的守护进程。

---

驱动程序希望执行但在内核中无法进行的任何操作，如写入文件、显示错误消息或启动应用程序，都可以通过用户空间守护进程执行。实际上，驱动程序代码分为两部分：内核驱动程序和用户空间守护进程。这种设计使用的技术与第5章所讨论的内容相同。但用户并不启动用户空间进程，该进程是一个后台守护进程，由系统自动启动。

用户空间守护进程和驱动程序互相配合，执行某些操作。用户空间进程大部分时间都处于空闲状态，仅在接收到内核驱动程序的请求时才活动。用户空间守护进程需要响应的通知有3种：

- 载入新的内核驱动程序；
- 卸载内核驱动程序；
- 驱动程序请求执行一个操作。

由于守护进程在系统启动时启动，所以它很可能会在对应的内核驱动程序启动之前启动。因此，该进程应该安装一个回调，接收它的内核驱动程序启动或停止时的通知。大多数情况下，守护进程与驱动程序实例具有一对多的关系，一个守护进程处理其驱动程序当前所有加载实例的请求。

守护进程通过安装驱动程序的匹配字典来监视内核驱动程序实例的加载和卸载，如第5章所述。进程也执行同样的步骤实现该操作，无论是后台守护进程，还是带有用户接口的应用程序。守护进程通过驱动程序的IOUserClient类发送请求，使用第5章介绍的I/O Kit框架函数与其内核驱动程序通信。

第5章已经介绍过用户空间进程如何将请求和数据发送给内核驱动程序。这非常重要，因为

这是守护进程用来将操作结果发送回内核驱动程序的一种解决方案。现在，我们来讲述进程如何接收内核通知，如代表驱动程序执行操作的请求，例如向用户显示一个错误消息。

内核驱动程序通过Mach端口与用户空间进程通信。下列步骤将设置一个通信信道，内核驱动程序用它来向用户空间进程发送通知。

(1) 用户空间守护进程确定它的驱动程序的一个实例，调用IOServiceOpen()，打开与驱动程序用户客户端的连接，如第5章所述。

(2) 该守护进程创建一个Mach端口，接收内核驱动程序的通知。这通过CFMachPortCreate()函数实现。该函数有许多参数，其中包括一个回调函数，用来发送通知。

(3) 该守护进程为Mach端口创建一个运行回路源，并将该源安装到它的一个线程的运行回路中。随后，当Mach端口上接收到通知时，该守护进程的回调便在运行回路线程上运行。

(4) 该守护进程使用IOConnectSetNotificationPort()函数将Mach端口传递给内核驱动程序。相应地，驱动程序的用户客户端接收其registerNotificationPort()方法的调用。

(5) 在内核中，用户客户端实现虚拟方法registerNotificationPort()。该客户端接收用户空间守护进程创建的Mach端口，并将该值保存在一个实例变量中。

(6) 当驱动程序希望通知用户空间守护进程一个事件时，便会调用mach_msg_send_from_kernel()函数，并提供其希望传递给用户空间守护进程的任何数据。

(7) 相应地，守护进程的回调函数会被调用。该回调函数会接收内核驱动程序传递的数据，并处理内核请求。如果操作结果需要传回给内核，那么用户空间守护进程便可以通过调用驱动程序用户客户端定义的某些方法来实现，如第5章所述。

本节的剩余部分会介绍一个从内核驱动程序向用户空间进程发送通知的例子。首先，需要定义一个结构体，来描述内核驱动程序发向用户空间守护进程的数据。该结构体必须的第一个成员必须是mach_msg_header_t结构体，因为这描述了用户空间守护进程中接收数据的目标Mach端口。在mach_msg_header_t字段之后，该结构体还包含许多字段，使数据可以和通知一起发送给用户空间守护进程。该结构体定义必须对用户空间守护进程和内核驱动程序都可用，因此，它应该放在两个工程都能包括的头文件中。以下是该结构体的一个示例定义，它允许两个整数参数从内核传递到用户守护进程中：

```
typedef struct {
        mach_msg_header_t       messageHeader;
        uint32_t                customParameter1;
        uint32_t                customParameter2;
} MyNotificationMessage;
```

接收内核通知的用户空间回调函数示例，如下所示。msg参数包含整个MyNotificationMessage结构体，其中包括添加的两个任意整数。消息头之后的这些额外字段用来描述驱动程序希望用户空间守护进程执行的操作，以及传递该操作所需的其他参数。

```
void    MyDriverRequestCallback (CFMachPortRef port, void *msg, CFIndex size, void *info)
{
        MyNotificationMessage* notify = (MyNotificationMessage*)msg;
```

```
        printf("Param 1 is %x, param 2 is %x\n", notify->customParameter1,
               notify->customParameter2);
}
```

代码清单17-3所示的代码片段演示了用户空间守护进程安装Mach端口时必须执行的操作步骤，该端口是它接收内核驱动程序通知的端口。第一步是分配Mach端口及相应的运行回路源，并将该Mach端口安装到运行回路中。下一步将Mach端口提供给内核驱动程序。每当该驱动程序希望向用户空间进程发送通知时，就会通过提供的Mach端口发送该请求。

**代码清单17-3　安装一个回调，接收内核驱动程序通知的用户空间代码**

```
CFMachPortContext       portContext;
CFMachPortRef           notificationPort = NULL;
CFRunLoopSourceRef      runLoopSource = NULL;
kern_return_t           kr;

//创建Mach端口时设置所需的CFMachPortContext结构体
portContext.version = 0;
portContext.info = (void*)context; //提供给回调的任意指针
portContext.retain = NULL;
portContext.release = NULL;
portContext.copyDescription = NULL;

//创建Mach端口
notificationPort = CFMachPortCreate(kCFAllocatorDefault, MyDriverRequestCallback,
&portContext,  NULL);
if (notificationPort)
{
     //为Mach端口创建一个运行回路源
     runLoopSource = CFMachPortCreateRunLoopSource(kCFAllocatorDefault, notificationPort, 0);
     //在与当前线程对应的运行回路上安装运行回路源
     CFRunLoopAddSource(CFRunLoopGetCurrent(), runLoopSource, kCFRunLoopDefaultMode);
}
//将通知传递给驱动程序
kr = IOConnectSetNotificationPort(driverConnection, 0,
                                  CFMachPortGetPort(notificationPort), 0);
```

用户空间函数IOConnectSetNotificationPort()中会调用registerNotificationPort()方法的驱动程序用户客户端。这是一个虚拟方法，定义在IOUserClient基类中，但需要由每个子类实现。在下面的示例实现中，registerNotificationPort()方法取得Mach端口的一个副本，该端口与用户空间进程的通知端口相对应，这样，将来每当驱动程序希望通知用户空间进程时都可以使用该端口。

```
IOReturn       com_osxkernel_driver_IOKitTestUserClient::
       registerNotificationPort (mach_port_t port, UInt32 type, io_user_reference_t refCon)
{
     m_notificationPort = port;
     return kIOReturnSuccess;
}
```

设置了进程的通知端口之后，内核驱动程序便能在必要时通知用户空间守护进程。这一般通

过IOUserClient子类执行，因为该通知端口专门用于特定的用户客户端。自定义用户客户端方法示例，如代码清单17-4所示，可以调用该方法，将两个任意整数传递给用户空间进程。

**代码清单17-4 从驱动程序向用户空间进程发送通知的自定义方法**

```
IOReturn       com_osxkernel_driver_IOKitTestUserClient::
       mySendNotification (uint32_t parameter1, uint32_t parameter2)
{
    MyNotificationMessage  notification;
    IOReturn               result;

    if (m_notificationPort == MACH_PORT_NULL)
        return kIOReturnError;

    //设置标准的mach_msg_header_t字段
    notification.messageHeader.msgh_bits = MACH_MSGH_BITS(MACH_MSG_TYPE_COPY_SEND, 0);
    notification.messageHeader.msgh_size = sizeof(MyNotificationMessage);
    notification.messageHeader.msgh_remote_port = m_notificationPort;
    notification.messageHeader.msgh_local_port = MACH_PORT_NULL;
    notification.messageHeader.msgh_reserved = 0;
    notification.messageHeader.msgh_id = 0;

    notification.customParameter1 = parameter1;
    notification.customParameter2 = parameter2;

    //将请求发送给用户空间
    result = mach_msg_send_from_kernel(
                &notification.messageHeader, sizeof(MyNotificationMessage));

    return result;
}
```

## 17.9 小结

本章讲述了如下内容。

- 内核中浮点及SSE的使用方法。介绍了OS X与其他操作系统的不同之处：不需要任何特殊处理来支持这些操作。
- 编写多功能驱动程序的策略。
- 内核控制KPI是一个BSD KPI，用于内核扩展和用户空间之间的通信。通常与NKE联合使用，但在I/O Kit中很少使用。
- 讲述了内核进程处理KPI。该KPI包括发送通知、获取进程名称及PID的函数。
- 内核扩展从其程序包加载外部资源及处理驱动程序偏好的方式。
- 内核扩展发送消息及通知用户空间进程的方式。

# 第18章 部署

到目前为止，我们已经讲述了各背景信息，查看了某些类型的驱动程序及内核扩展的实际实现。本章主要讲述如何将开发成果交付给最终用户。苹果公司以能提供用户友好型软件（和硬件）解决方案著称，该公司及其用户都希望从第三方厂商得到同等的用户体验。复杂的安装流程将会让用户相当泄气，这样在任何平台上都很容易失去竞争力。部署内核扩展等软件乍看起来可能非常容易，但实际上有很多问题需要考虑，例如怎样兼容大量不同的硬件及操作系统版本。许多大型商业或政府用户往往非常抵触升级，因此，你可能既要支持最前沿的操作系统，又要支持传统的操作系统，而这些操作系统都有不同的特性，需要专门处理。除了外部因素，软件分布方式也可能非常复杂。通常，很少会只发布KEXT本身，通常需要捆绑其他一些软件。例如，计算机显卡可能捆绑系统首选项面板、用于访问设备特性的框架、升级固件的应用程序，以及一些展示显卡功能（如游戏）的应用程序。此外，用户可能要对软件发布版进行升级或降级，这些情况都需要处理。

虽然这看起来（实际上也的确）让人怯步，但希望总是有的。苹果公司会像往常一样，提供工具来简化该处理。部署所用的高级软件安装工具是PackageMaker。PackageMaker采用易于使用的图形化用户界面来创建安装向导。PackageMaker还具有命令行实用工具特性，可以用来把程序包整合到较大的构建系统中。

## 18.1 安装和加载内核扩展

KEXT包通常安装在系统目录/System/Library/Extensions中。可以将KEXT放置在该目录外，但你需要使用下一节讲述的方法自己手工加载该KEXT。KEXT仍然需要拥有正确的权限设置。它需要属于root用户所有，从属于wheel群组，其权限掩码是0755，以便由KEXT守护进程（kextd）加载。

对基于I/O Kit的驱动程序，当KEXT的提供者在I/O Registry中注册时，该KEXT通常由KEXT守护进程自动加载（假定该KEXT在其Info.plist文件中定义了正确的个性信息）。与硬件提供者无关的KEXT使用IOResources块作为提供者，在系统启动时自动加载自身。对基于非I/O Kit的KEXT（如NKE或虚拟网络驱动程序）来说，这不能实现，因为IOResources无法用于I/O Kit之外的KEXT。

对于非I/O Kit的KEXT，可以创建一个Launch Daemon。Launch Daemon和Agent是苹果公司

提供的许多传统UNIX服务（包括init.d、cron及inetd）的替代物。当用户基于其安全权限登入系统时，Agent便会运行。另一方面，守护进程属于系统级，通常以root权限运行。可以在计算机启动时，使用Launch Daemon执行一个Shell脚本，加载KEXT。为了创建启动代理，简单定义一个plist文件，如代码清单18-1所示。

**代码清单18-1 Launch Daemon属性列表文件**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
        "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.osxkernel.launchd.HelloWorld</string>
    <key>ProgramArguments</key>
    <array>
        <string>/Library/Application\ Support/HelloWorld/loadkext.sh</string>
        <string>load</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
</dict>
</plist>
```

该文件应该安装到/System/Library/LaunchDaemon或~/Library/LaunchDaemon目录中，即com.osxkernel.launchd.HelloWorld.plist。现在，Launch Daemon在启动时便会触发loadkext.sh脚本。该脚本本身的实现，如代码清单18-2所示。

**代码清单18-2 加载KEXT的UNIX Shell脚本**

```
#!/bin/sh
COMMAND=$1

THEKEXT=/System/Library/Extensions/HelloWorld.kext

if [ -f "$THEKEXT" ]
then
    echo "KEXT does not exist"
    exit 1
fi
if [ "$COMMAND" = "load" ]
then
        kextload $THEKEXT
elif [ "$COMMAND" = "unload" ]
then
        kextunload $THEKEXT
fi
```

在某些情况下，可能需要根据要求来加载KEXT，而不是在系统启动时加载。例如，就VPN（Virtual Private Network，虚拟专用网络）应用程序而言，它可能预装了一个KEXT，用于处理定制网络级的加密方案或者安装虚拟VPN接口。该KEXT仅在应用程序活动时需要。使其保持活动

状态会浪费内存资源，而在启动时加载它也会影响启动时间。另外，由于该KEXT与网络栈进行交互，所以它可能会阻碍并影响系统的网络性能。这种情况下，应用程序可能希望动态加载和卸载该KEXT。这可以通过使用类似上面的脚本来实现。应用程序需要使用root权限运行才能加载和卸载该KEXT。但让整个应用程序都setuid root可不是什么好主意，因为可能会造成严重的安全问题。一个替代解决方案是使用AuthorizationExecuteWithPrivileges() API临时提高执行程序的权限。在此期间，会提示用户输入计算机的系统密码。为了使KEXT可以由非root用户加载，可以修改KEXT的plist文件，包含如下代码：

```
<key>OSBundleAllowUserLoad</key>
    <true/>
```

虽然这会让没有权限的进程或用户加载该KEXT，但之后卸载KEXT时仍然需要root权限。因而，前一种技术是首选，也更安全，因为用户必须明确同意该操作。

**App Store上的内核扩展**

苹果公司目前不允许应用程序在Mac App Store或iOS App Store上安装KEXT（如果不违反许可证协议，将无法构建KEXT）。如果你的应用程序依赖于某个内核扩展，那么需要在App Store外发布该应用程序。Mac App Store应用程序不可以请求root权限，否则需要安装并加载KEXT。即使某些应用程序可能要依赖于特定的硬件设备，也可以提交这些应用程序，但在这种情况下，需要通过一个认证的用户空间API来与它进行沟通（具体细节请参见第15章）。

## 18.2　加载首选项和设置

许多驱动程序可能需要某些用户为设备单独配置的首选项或设置。例如，音频设备可能有一些设置来控制输出音量，而且需要在系统重启时仍然发挥作用。驱动程序无法访问文件系统（这被认为是一个糟糕的设计），因而，它无法从文件读取首选项信息，但I/O Kit驱动程序可以访问其plist文件中存储的信息。plist文件为只读，由KEXT实例化的所有驱动程序实例共享。由于一个驱动程序可能会创建多个实例，而每个实例需要不同的设置，所以plist文件对此并不适用。

尽管你可以采用自己的方案，将所有设备设置都放在一个plist文件中，但这并不是一个很好的设计。另外，应用程序很难修改plist文件，因为这通常需要root权限。Mac OS X并不提供处理驱动程序及KEXT首选项或设置的标准化机制或API。因此，如果需要该功能，必须实现一个用户空间应用程序来处理该操作。可以安装一个通知等待设备接入，接入后使用IOConnectCall*()函数调用驱动程序的用户客户端，从一个文件恢复设置。至于如何安装驱动程序或设备通知，请参见第5章。

如果你正在实现一个不可移除设备的驱动程序，可以使用Launch Daemon或启动项，在系统启动或开始时运行用户空间设置助手。为此，必须为驱动程序注册一个通知，确保该助手在设备

接入之前不会运行。该进程能在设备接入之后退出。如果你有一个可移除的设备，如USB、FireWire或Thunderbolt设备，那么就要实现一个永久性的守护进程（Launch Daemon），来监听设备插入，并在设备插入之后恢复设置。如果该设置专门用于独占使用驱动程序或设备应用程序，可以从该应用程序本身来管理该设置，并在应用程序退出之后，选择将设置恢复至之前的状态。该辅助程序或守护进程在与其交互的用户客户端或内核控制许可时，可以采用普通用户权限运行。

如果你的驱动程序处理同一个设备的多个实例，那么指出哪些设置属于哪个设备对你来说可能是一种挑战，因为其顺序在重启前后可能并不一致，USB设备的插入顺序也可能不同，或者插入的端口不同。为了解决该问题，可以使用设备的唯一串口号或另一种标识符。一个网络接口有一个唯一的MAC地址，而USB设备通常也会有一个串口号或类似的东西。Thunderbolt设备（理论上）肯定会有一个全局唯一标识符（UID）。

## 18.3 内核扩展的版本管理

如果用户空间应用程序能够直接访问内核扩展，你可能希望提供一个版本控制系统，防止旧的用户空间应用程序访问新的内核扩展，或者反过来。对于音频驱动程序或其他使用系统提供的IOUserClient的驱动程序来说，这没有必要，但在没有版本控制的情况下，KEXT会为应用程序提供一个API，来保持兼容。如果KEXT已经升级，旧的应用程序可能会中断，甚至崩溃。处理该问题并没有标准的方式，具体的解决方案取决于KEXT特性及访问该KEXT的驱动程序。一个策略是在共享的头文件中包含版本号：

```
//
// SharedHeaderFile.h
//
#ifndef Shared_Header_H_
#define Shared_Header_H_

const int KernelUserClientAPIVersion = 1001;

#endif
```

由于KEXT和用户应用程序都会将KernelUserClientAPIVersion版本号编译到其二进制映像中，所以用户应用程序可以确定KEXT的版本号是否与其版本号相匹配。每次改变接口，例如添加、移除或修改IOUserClient方法时，版本号必须随之更新。如果你的KEXT向第三方开发者提供一个可用的API，最好的解决方案是提供一个内部版本管理API，而不是让开发者直接访问IOUserClient。这样，内核和用户空间之间的接口发生改变时，就无须中断现有的应用程序，甚至可以将同一个API提供给完全不同的驱动程序或KEXT。

## 18.4 测试和品质保证

测试内核扩展是一项非常有挑战性的工作。现代计算机系统和设备通常都非常复杂，即使你的组件完全独立于系统的其他部分，但由于它是整个运转系统中的一个组件，也不可避免地会发

生意外。正确的测试至关重要，最好由未直接参与开发的人员进行，直接参与开发的人员往往会对产品运行有先入为主的观念，而无法发现最终用户会碰到的问题。测试内核扩展很大程度上依赖于扩展的性质及类型，因此，不可能提供详尽的测试列表。以下是一些一般性的建议。

- 测试驱动程序的睡眠和唤醒。可以使用SleepX工具（Xcode的一部分），很方便地对此进行负载测试。应用程序可以在每次被计算机唤醒时，执行一个脚本或外部程序。可以调整睡眠时间间隔，以及你希望测试的周期数。至关重要的是你的驱动程序能够正确处理睡眠，特别是对移动设备及笔记本电脑来说，因为让驱动程序保持活动状态就意味着CPU必须处于活动状态，这会耗费电量，并且在笔记本电脑合上盖子时，甚至会产生过热问题。
- 使用kextload和kextunload测试驱动程序反复处理加载和卸载的情况。可以使用ioclasscount及ioalloccount，测试引用泄漏或内存分配泄漏。
- 通过执行常见的用户任务，测试使用驱动程序的应用程序。例如，对于音频设备，使用尽可能多的设置及格式组合，进行音频回放和采集。
- 在所有支持的平台和操作系统上测试，如Macbook Pro、Mac Mini、iMac、Mac Pro或iPod、iPhone和iPad。对于Mac，务必要测试32位及64位版本，以及所有支持的操作系统版本，例如Snow Leopard、Lion等。
- 对于热插拔设备：
  - 测试驱动程序及设备在反复插拔之后仍可以继续正确工作。可能至少需要反复执行100次以上，才能确信这方面不会出现问题。
  - 测试驱动程序处理系统睡眠。驱动程序还应该处理系统睡眠时移除的设备。
- 测试KEXT的32位及64位版本。如果你支持每种Lion操作系统，那么应该提供通用的二进制代码，支持32位和64位两种系统。
- 测试安装包，确保所有的文件都能正确安装——采用正确的权限，位于正确的位置。

无论是测试时发现错误，还是用户报告错误，保留每次所发布KEXT的预归档符号信息调试版本是一个很好的做法。这样，就可以迅速重现该问题，并进行调试。下面这种情况对大多数人来说可能很容易理解，但确实会经常发生：某人交付一个软件组件的发布版，如一个KEXT，但却没有将其版本号加1。每次修改、发布/交付给某一外部组织（甚至是内部QA组）时，都应该有一个唯一的版本号，否则你很快将无法跟踪，让相关的每个人都感到不知所措。

## 18.5　打包 KEXT 和软件

PackageMaker在打包及发布包含多个组件的软件时是首选工具。简单的应用程序所需的每个组件都是自包含的，并且嵌在应用程序包中，而复杂软件的发布版（如包含KEXT的发布版）可能需要将组件安装到文件系统的多个位置。例如，一个驱动程序需要将自身置于/System/Library/Extensions目录中，它可能还包含辅助程序以加载首选项或升级固件，这就需要安装Launch Agent或Daemon。PackageMaker可以完成诸如此类乃至更多的任务。PackageMaker的用户界面如图18-1所示。

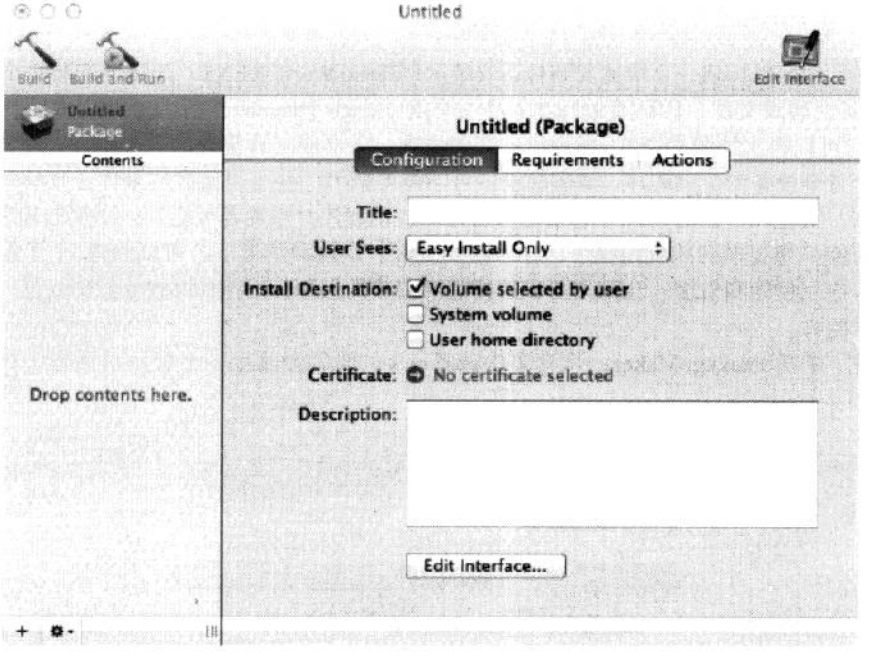



图18-1 PackageMaker用户界面

PackageMaker输出一个扩展名为.pkg的二进制压缩包文件，可以使用Installer.app程序或installer UNIX命令安装，执行类似下面的命令：

```
sudo installer -pkg HelloWorld.pkg -target /
```

使用Installer.app进行安装将会生成一个基于GUI的向导，指导用户执行安装流程。如果程序包中有多个可选的子组件，用户将能够选择或撤销选择。用户还可以设置Volume selected by user选项，改变程序包安装的目标卷，前提是程序包明确允许这么做。

PackageMaker并不仅限于将软件组件置于文件系统中，它还可以检查系统需求及先决条件，其中包括是否支持该操作系统版本，或检查其依赖的某些其他软件组件是否存在。PackageMaker还可以在软件安装之前及之后，触发每个组件的自定义UNIX脚本。对于硬件驱动程序，PackageMaker还能在用户可以继续之前，验证你的设备是否存在。这项功能对FireWire、PCI及USB都起作用。

**提示** 打包KEXT进行发布时，请确保Xcode针对你能够支持的体系结构进行构建。通过对KEXT的二进制文件使用file命令，可以检查它支持哪些体系结构。此外，还需要确保发布的不是KEXT的调试版本。

### 18.5.1　构建Hello World内核扩展安装包

现在，你可以构建一个快速安装包，安装示例Hello World KEXT。还需要安装一个Launch Daemon属性列表文件（代码清单18-1）及一个由Launch Daemon执行的Shell脚本（代码清单18-2），它可以再次加载KEXT。在设计安装包之前，需要确定其包含的各个组件。例如，如果发布一个音频设备的驱动程序，该驱动程序可能需要捆绑一些可以回放或采集音频的软件以及一个SDK，其他的开发者可以用它来编写自己的应用程序。开发者可能只想要驱动程序，而最终用户可能只想要捆绑的应用程序，并不需要SDK。在这种情形下，可以创建3个子组件：驱动程序、应用程序和SDK，让用户可以选择想要安装的组件。对于HelloWorld安装包，只需要添加一个组件。

首先，打开PackageMaker，从菜单中选择new。然后会显示一个提示对话框，如图18-2所示。

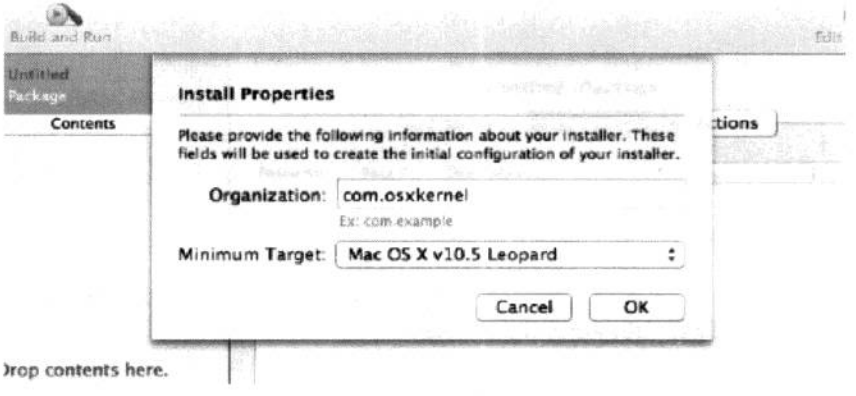



图18-2　安装属性对话框

Organization字段是你组织的反向DNS名称。该值用来标识安装包，名字必须唯一。通过Minimum Target下拉列表可以选择希望兼容的旧操作系统版本。安装包将无法安装在比选定版本更早的系统上。选择之后，可以给安装包设置一个标题。

### 18.5.2　将内容添加至安装包

现在可以着手将文件添加到安装包中。从主菜单选择“Project ➤ Add Contents...”，或者简单地将文件拖入PackageMaker的左边面板中完成添加。包含HelloWorld发布版所需文件的内容面板如图18-3所示。

我们的HelloWorld安装包只包含3个文件（如图18-3中左边所示）：KEXT及两个辅助文件（可以使KEXT在启动时加载）。在将一些文件拖入或添加到安装包时，PackageMaker会自动创建一个选择项（choice），在该示例中，我们将它命名为Driver。选择项表示子组件，用户可以逐个选择。

但如果主面板中的User Sees字段设置为Easy Install Only，那么即使存在多个组件，也不会提示用户选择。Installation Destination选项设置安装包将要安装到的位置。此时，我们不会检查任何选项，因为我们的所有文件都将安装到绝对路径中。如果选择Volume selected by user，那么安装向导将询问用户希望安装到哪一个卷（硬盘）。为方便用户，建议设置该选项，特别是对于较大的软件安装包，用户可能希望将其存储在另一个更大的数据驱动器上。

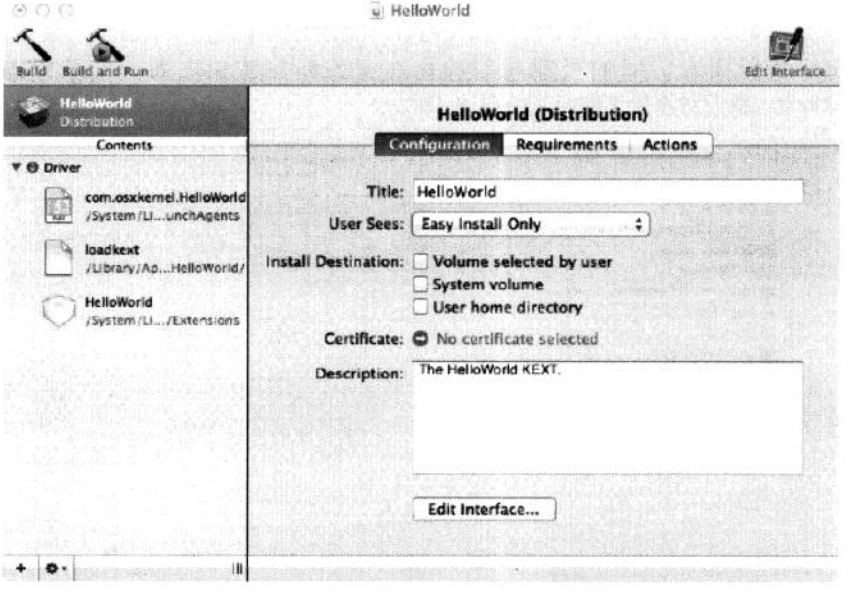



图18-3 HelloWorld发布版

## 18.5.3 配置安装包

点击图18-3配置面板中的Edit Interface选项，将会弹出在使用`Installer.app`打开安装包时显示的对话框。PackageMaker允许交互编辑文本，并设置背景图片。还可以添加本地化支持，设置其他语言。

主面板上有三个标签页，如图18-3所示。Requirements标签页可以配置对整个安装包的全局需求。还可以为各个选择项设置需求，即使系统并不满足其他组件的需求，用户也可以安装某些组件。需求可以在需求编辑器中指定，需求编辑器有许多可配置的预定义检查。以下是部分可用测试：

- 目标机上可用的兆字节；
- 最低限度的CPU频率（HZ）；
- 可用内存（字节）；

- 系统操作系统版本/目标机操作系统版本；
- 文件是否存在于系统上；
- FireWire、USB、PCI设备是否存在；
- 脚本执行结果。

最后一个选项Result of Script在预定义的测试都不合适时特别有用。可以编写一个Shell脚本，执行你自己的测试，使安装程序基于脚本的返回码执行操作。还可以指定自定义的错误消息，在需求测试失败时显示。Action选项卡允许配置能自动执行的操作，预安装或后安装。例如，就HelloWorld安装包而言，我们可能希望确保在安装新安装包之前，卸载以前安装的HelloWorld.kext。我们对该操作的定义如图18-4所示。

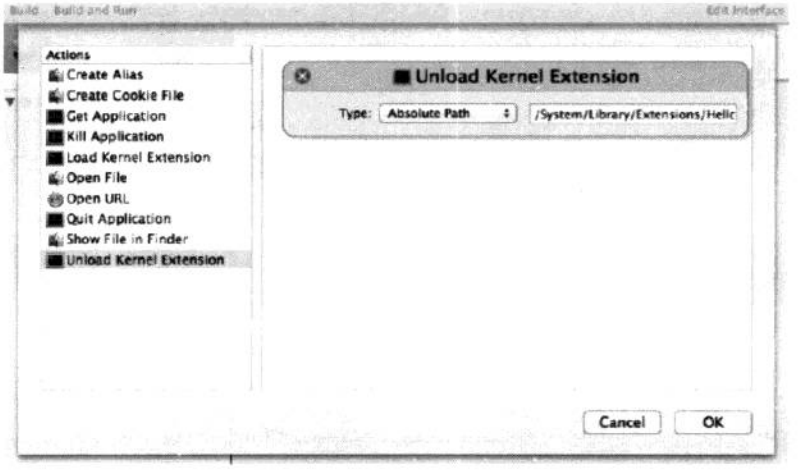



图18-4　操作编辑器

在某些情况下，假定你有一个USB设备驱动程序，可能想要确保所有的应用程序在安装新驱动程序之前都被关掉。打开的应用程序可能会妨碍驱动程序正常卸载。你可能已经发现，并没有运行程序或脚本的选项，但这可以通过选中一个文件并选择Scripts选项卡，基于每个文件实现。

目前，我们使用术语“安装包”来表示整个工程，但在PackageMaker术语中，实际上指包含几个较小安装包的发布版。较小的安装包由添加到发布版中的各个文件或对象组成，如KEXT、应用程序包或PDF文件。每个对象都包含与其相关的元数据及版本信息。

图18-5是添加到发布版的安装包的主配置面板。Install字段指定源位置，而Destination字段指定该对象在文件系统中的安装位置。选中Allow custom location选项表示用户可以指定其他位置。源位置和目标位置都可以是相对或绝对路径。大多数情况下，推荐使用前者。在上面的示例中（图18-5），我们使用源文件的相对位置，而对目标字段使用文件系统中的绝对路径，因为该文件必须放置到该目录，无法基于用户设置进行安装。KEXT也是这样，需要置于/System/Library/Extensions中。相对路径是相对于PackageMaker工程的位置。

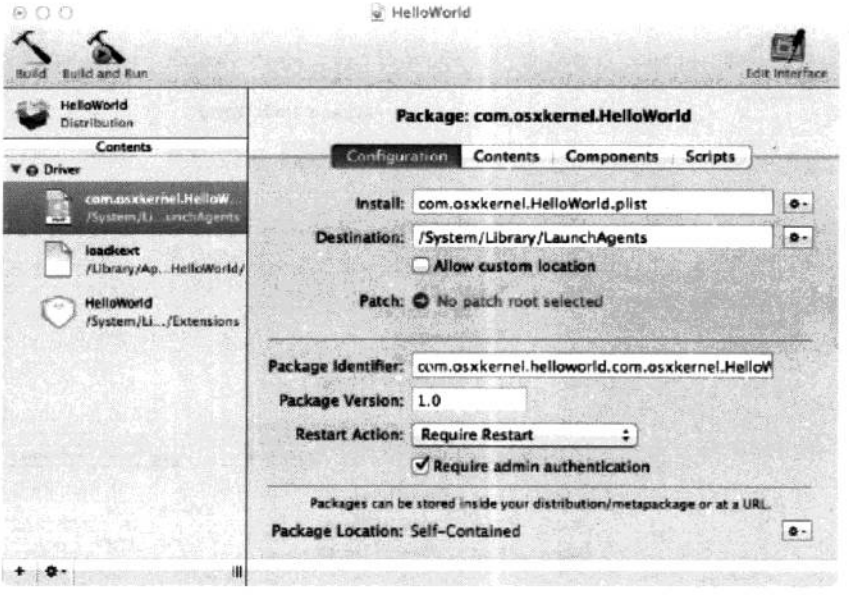



图18-5 各个安装包的配置面板

Patch字段可以指定安装对象的较早版本，以便安装程序可以修复现有的文件，而不是安装全新的文件。

在配置面板中，还可以指定一个版本号，单独标识每个安装包。Restart Action可以提示用户在安装完成之后重启计算机。如果有必须启动的Launch Agent或Daemon，或者替换了系统运行时难以重新加载的驱动程序（例如存储或图形驱动程序），可能就需要这么做。还可以要求用户退出或关闭系统。Require admin authentication复选框将提示用户在安装之前输入管理员密码。如果在用户根目录之外安装文件，通常必须指定这一项，因为安装程序默认情况下没有管理员权限。大部分KEXT——特别是驱动程序——必须安装在/System下，普通用户通常无法写入该目录。

最后，Package Location字段可以指定一个其他位置，安装包对象文件从该位置进行安装。例如，你可以指定一个HTTP URL。

安装包的Contents选项卡如图18-6所示。

Contents选项卡可以检查安装包的各个文件。通常，安装包包含一个程序包或单个文件。一个程序包可以包含其他的程序包或其中的任意文件。如果要排除一些文件，可以指定正则表达式来实现。例如源代码版本控制系统的元数据目录，如Subversion，可以使用“/\.svn$”表达式排除。还可以在各个文件安装到文件系统之后，配置其文件权限。Apply Recommendations按钮将基于文件类型及安装目标来推测正确的权限。

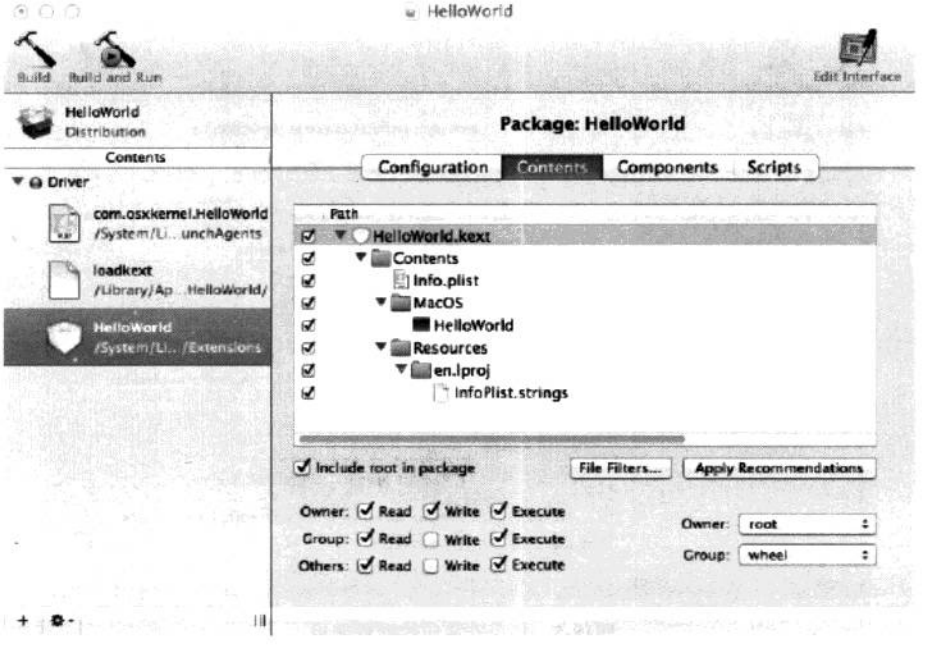



图18-6　安装包配置面板的Contents选项卡

**警告**　内核扩展对其权限非常挑剔，必须由root用户和wheel组拥有。另外，拥有者必须有读、写和执行权限，而组应该只有读和执行权限。这与UNIX权限掩码0755相对应。元数据文件（如Info.plist文件）无须拥有可执行的比特位集。

Scripts选项卡可以定义安装包的预安装和后安装脚本。这些脚本可以执行Actions编辑器无法定义的自定义安装步骤。可以使用预安装脚本在守护进程或应用程序被替换为新版本之前将其关闭。还可以用它来清除新版本不再需要的文件。该脚本通常采用Bash或其他脚本语言编写。如果你使用脚本移除旧的文件，请慎用`rm`命令，因为你的安装包可能采用管理权限运行，指定文件名错误可能会导致误删文件或目录。比如这行代码：`rm -rf /System/Library/$MYKEXT`。如果`$MYKEXT`变量为空，则该命令将删除/System/Library目录。然后，用户的投诉就会接踵而来。

### 18.5.4　构建安装包

添加并配置发布版中包括的所有部分之后，可以按下工具栏上的Build或Build and Run进行编译。后者将在构建完成之后，使用`Installer.app`打开安装包。在构建阶段，安装包会进行验证和检查，如果存在错误或警告，可以修改并重新构建安装包。最终结果看起来类似于图18-7。

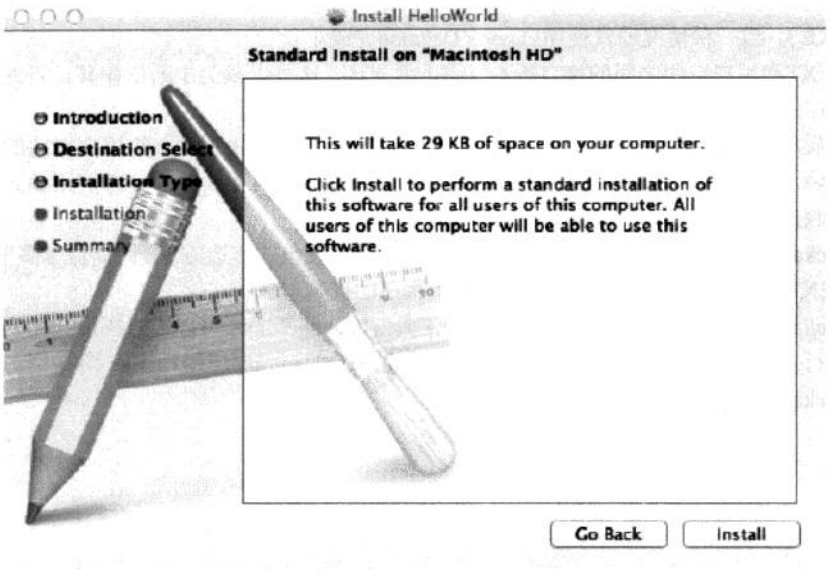



图18-7 安装包配置面板的Contents选项卡

生成的安装包会被写入到文件名是.pkg的文件中。安装包被压缩且不是一个程序包，因此其内容无法在Finder中查看。保存PackageMaker工程时，该工程将保存为一个名为.pmdoc的程序包。该工程程序包包含定义该工程的XML文件。构建产品时，可以在文本编辑器中编辑这些XML文件，或使用脚本自动替换或更新其内容。如果你希望从命令行构建安装包，可以执行如下命令：

```
$ /Developer/Applications/Utilities/PackageMaker.app/Contents/MacOS/PackageMaker --doc
HelloWorld.pmdoc --version 2.0 --title 'HelloWorld'
```

前面的命令将构建版本号为2.0的.pkg。

### 18.5.5 卸载安装包

很遗憾，并没有类似的机制可以自动卸载PackageMaker创建的安装包。为了手动卸载安装包，你需要确定由安装包安装的文件并将其删除。还必须停止并移除由安装包安装的Launch Agent或Daemon。为用户提供一个脚本或程序来执行卸载不失为一个良策。但PackageMaker可以处理升级。

## 18.6 小结

本章讲述了如下内容。

- 系统如何自动加载I/O Kit内核扩展，以及如何手动加载那些无法自动加载的KEXT（如NKE）。后一种情况通常使用Launch Daemon完成。
- KEXT和用户空间之间的接口版本管理非常重要，这可以避免中断依赖其运行的其他应用程序。一种解决策略是在共享的头文件中包含一个版本号。
- 品质保证和测试是开发周期中非常重要的一部分，是将软件发布给用户之前的最后一道屏障。至关重要的是要在所有支持的平台上对所有部门及软件的各种使用模式进行恰当地测试。
- PackageMaker软件是Mac OS X上发布复杂软件安装包的首选方式，例如包含多个组件（如KEXT）、辅助守护进程及最终用户应用程序的安装包。
- PackageMaker是使用Xcode发布的一种工具。PackageMaker可以通过图形化用户界面或命令行使用。PackageMaker创建的安装包通过`Installer.app`应用程序安装。
- PackageMaker并不提供安装包卸载机制。

# 索　引

## C


采样框架, 216
采样深度, 216
操作系统原理, 1
　调度, 8, 9, 17, 18, 19, 20, 22, 27, 31, 77, 78, 103, 109, 162, 163, 164, 166, 167, 229, 332, 360
　服务, 1, 2, 3, 5, 10, 11, 12, 14, 15, 17, 18, 21, 27, 28, 30, 31, 32, 39, 41, 42, 45, 57, 72, 83, 87, 102, 111, 114, 115, 138, 162, 171, 178, 218, 221, 238, 259, 263, 275, 276, 282, 295, 320, 333, 339, 340, 344, 359, 373
　进程地址空间, 4, 5, 6, 7, 8, 23, 76
　进程管理, 3, 17, 27
　内核空间代码, 2
　内核扩展, 1, 2, 5, 12, 14, 17–40, 43, 47, 54, 84, 87, 276, 288, 289, 335, 352, 355, 365, 371, 372, 374, 375, 378, 382, 384
　启动序列, 3
　声卡, 3, 10, 43
　网络接口, 28, 154, 239, 242, 249, 253, 255–259, 262, 269, 270, 274, 275, 276
　虚拟内存, 6, 7, 8, 18, 22, 24, 25, 26, 31, 76, 84, 86, 87, 89, 90, 93, 97, 98, 100, 170, 171
　硬件和驱动程序, 9
　硬件配置, 2, 3, 316
　用户空间代码, 2, 41, 81, 83, 99, 109
长模式, 87
存储系统, 277
　传输层驱动程序, 278
　分层栈, 277
　分区规则, 291–295, 297–303, 307, 308
　驱动程序栈, 138, 194, 277, 278, 279, 282, 284, 285, 288–297, 299, 300, 302, 303, 307, 308, 309
　外部USB闪存驱动器与内部硬盘, 277
传统模式, 87
串行端口驱动程序, 58, 192–198, 200, 201, 202, 203, 204, 205, 207, 209, 210, 211, 212, 214, 310
　内核串行驱动程序栈, 194
　通信设备类, 195


## D


电源管理, 51, 177, 178, 179, 181–185, 187, 188, 189, 191, 222
　I/O Registry电源平面, 177
　睡眠模式, 177, 191, 221
调试, 21, 33, 37, 39, 51, 66, 104, 124, 132, 162, 167, 192, 242, 244, 251, 258, 259, 260, 308, 331–334, 336–349, 353–356, 359, 376, 377
　非法的指令/操作数, 332
　竞态条件, 104–107, 111, 331
　内存和资源泄漏, 332, 355
　内核恐慌, 5, 6, 32, 54, 91, 100, 101, 182, 187, 251, 310, 332, 333, 334, 338, 339, 340, 345, 346, 347, 351–354, 356, 358
　死锁, 91, 108, 109, 110, 115, 163, 204, 331, 332, 344, 356
　锁竞争, 108, 109, 332
　远程内核核心转储, 339
　栈跟踪, 333, 336
　主中断上下文, 147, 163, 164, 165, 167, 332


## E


二级中断处理程序, 108, 114, 118, 163–168, 336

## G

高级内核编程, 357
I/O Kit族, 341, 346, 356, 357, 358, 359
内核控制KPI, 242, 357, 359, 371
驱动程序通知, 80, 163, 368, 370
高级图形端口, 151
工作环, 113–119, 147, 163, 164, 165, 168, 183, 199, 315
IOCommandGate, 115, 116, 199–203, 208, 209, 210, 221
回调函数, 58, 61–64, 66, 76, 77, 78, 80, 81, 83, 114–117, 221, 224, 282, 305, 306, 315, 318, 324–329, 366, 367, 369
硬件中断或定时器, 113
中断处理程序, 91, 108, 109, 113, 114, 115, 118, 119, 164, 167, 168, 234, 272, 274, 332
过滤器规则驱动程序, 300–304, 308, 309

## H

活动链表, 88

## I

I/O Kit框架, 30, 32, 33, 41, 58, 60, 62, 63, 65, 68, 73, 74, 80, 81, 83, 167, 179, 192, 193, 311, 368
Info.plist文件, 35, 37, 43, 48, 49, 51, 55, 69, 131, 136, 153, 288, 359, 367, 372, 382
内核框架, 30, 31, 358
样板文件代码, 42
用户空间应用程序, 21, 27, 41, 42, 56, 57, 58, 59, 62, 63, 65–70, 72, 75, 80, 109, 110, 112, 115, 118, 192, 193, 194, 196, 200, 211, 212, 214, 221, 238, 243, 310, 311, 313, 315, 326, 329, 355, 357, 374, 375
I/O内存管理单元, 85, 170
I/O内存区, 158, 160, 176
帧缓冲区, 159, 160

## J

基地址寄存器, 150
接口过滤器, 242, 243, 249, 255–258, 263, 269, 276
输出过滤器函数, 253, 258

## K

开放主机控制器接口, 123
可扩展主机控制器接口, 123

## L

连接和断开回调, 362
流式SIMD扩展, 357

## M

Mac OS X和iOS, 6, 12, 14, 17, 25, 28, 31, 33, 61, 88, 128, 217, 236, 238, 239
编程API, 2, 14, 15, 16
英特尔x86平台, 16
脉冲编码调制, 215
媒介过滤器驱动程序, 300
块级别的磁盘加密, 301
内容线索属性, 300

## N

内存管理, 5, 22, 27, 84
内核内存分配, 88
映射内存, 22, 95–99, 160, 161, 162, 171, 174
内存管理单元, 23, 24, 85
内存映射I/O, 158, 161
内核地址空间管理, 23
内核控制KPI, 242, 357, 359, 371
内核扩展, 1, 2, 5, 12, 14, 17, 18, 29, 30, 32–40, 43, 47, 54, 84, 87, 276, 288, 289, 335, 352, 355, 365, 371, 372, 374, 375, 378, 382, 384
内核私有, 365

## P

PackageMaker

FireWire、PCI及USB, 377
安装包配置, 382, 383
二进制压缩包文件, 377
软件发布版, 372
苹果AHCI存储驱动程序栈, 279

## Q

驱动程序, 57
I/O Kit框架, 30, 32, 33, 41, 58, 60, 62, 63, 65, 68, 73, 74, 80, 81, 83, 167, 179, 192, 193, 311, 368
串行端口驱动程序, 58, 192, 193, 194, 195, 196, 197, 198, 200–205, 207, 209–212, 214, 310
匹配字典, 43–46, 48–51, 54, 55, 58–63, 66, 130, 131, 135, 136, 152, 176, 195, 197, 212, 288, 289, 293, 295, 300, 302, 311–314, 320, 321, 330, 359, 368
用户空间应用程序, 21, 27, 41, 42, 56–59, 62, 63, 65, 66–70, 72, 75, 80, 109, 110, 112, 115, 118, 192, 193, 194, 196, 200, 211, 212, 214, 221, 238, 243, 310, 311, 313, 315, 326, 329, 355, 357, 374, 375

## R

热插拔设备, 376

## S

设备地址转化表, 85
数据链路接口层, 259

## T

Thunderbolt, 123, 148, 149, 151, 153, 154, 157, 161, 162, 175, 176, 221, 262, 264, 271, 310, 335, 344, 375
套接字过滤器, 242–246, 248–251, 255, 256, 259, 276
AppWall, 244–250, 365
辅助协议, 239, 243
装载和卸载, 18, 246
通用主机控制器接口, 123
同步和线程, 102
异步事件, 113, 116
原语, 30, 97, 102, 110, 113, 183
原子操作, 104, 105, 106, 107
中断服务例程, 102, 162
仲裁, 3, 102
统一缓冲区缓存, 27, 29, 88

## U

USB, 1, 3, 10, 11, 30, 41–46, 49, 51, 54, 58–63, 66, 93, 120–133, 136–141, 143, 144, 146–151, 155, 159, 177, 178, 192–195, 198, 199, 209, 214, 217, 221, 261, 264, 277, 278, 280, 291, 310–313, 316–323, 325–330, 344, 358, 359, 368, 375, 380
大容量存储器设备驱动程序, 133
端点描述符, 122, 127, 132, 145, 323, 327
接口描述符, 127, 128, 129, 131, 146, 322
控制端点, 126, 143, 317, 320, 323, 324
控制请求, 6, 41, 45, 63, 65, 68, 72–77, 80–83, 112, 118, 126, 138, 143, 221, 264, 317, 318, 324, 360
流管道, 126
配置描述符, 127, 129, 130, 319, 322
批量端点, 126, 144–147, 324, 325
设备类, 30, 43, 127, 128, 130, 150, 179, 195, 277, 316, 318, 321, 322
设备描述符, 126, 127, 129, 130, 316
设备请求, 141, 142, 143, 145, 311, 317, 318
设备移除, 63, 64, 138, 161, 162, 176
同步端点, 126, 140, 145, 146, 147, 323, 324, 326, 328, 329
拓扑, 120, 121, 147, 149
消息管道, 126
中断传输, 122, 145
中断端点, 126, 140, 144–147, 320, 324, 325
主机控制器, 44, 120–124, 126, 128, 129, 144, 145, 147, 195

## W

外围组件互连, 148
配置空间寄存器, 149, 150, 151, 156, 157, 161
网络, 1, 5, 17–21, 27, 28, 31, 33, 71, 87, 91, 122, 124, 141, 147, 149, 150, 162, 163, 169, 215, 219, 238, 239, 240, 242, 244, 246, 248, 249, 250, 254, 255, 257, 258,

259–266, 268–272, 274, 275, 276, 333, 339, 372, 373, 375
内核网络体系结构, 238
套接字API, 238, 239, 242, 243, 245
网络地址转换, 28, 255
网络内核扩展, 1, 28, 239, 242, 359
接口KPI, 242, 257, 258
接口过滤器, 242, 243, 249, 255, 256, 257, 258, 263, 269, 276
内核控制接口, 242, 359, 360, 361, 364
套接字过滤器, 242–246, 248–251, 255, 256, 259, 276
协议通道, 242
唯一标识符, 34, 36, 375
位深度, 216
物理地址扩展, 22, 85

## X

消息信号中断, 162, 165
写时复制优化, 25
虚拟内存地址, 8, 11, 86, 100, 101
虚拟文件系统层, 31, 277
虚拟专用网络, 373

## Y

音频驱动程序, 10, 57, 95, 215, 216, 218, 219, 221, 223, 225, 229, 310, 357, 358, 375
I/O Kit音频支持, 218
I/O引擎初始化, 226
数字音频和音频设备, 215
硬件初始化, 221, 225, 226
硬件抽象层, 217
用户空间USB驱动程序, 310, 311, 316, 320
扩展目录, 274
异步操作, 68, 80, 81, 82, 315, 330

## Z

自旋锁, 107, 108, 109, 119, 163, 260
多处理器系统, 108
死锁, 91, 108, 109, 110, 115, 163, 204, 331, 332, 344, 356
中断处理程序, 91, 108, 109, 113, 114, 115, 118, 119, 164, 167, 168, 234, 272, 274, 332
直接内存访问, 148, 168
传出DMA, 169
分散/聚集列表, 169, 170, 172, 173, 174, 176, 271
换出操作, 170
中断服务例程, 102, 162
主中断过滤方法, 166
增强型主机控制器接口, 123