深入解析 Mac OS X & iOS 操作系统

[美] Jonathan Levin 著

郑思遥 房佩慈 译

清华大学出版社

北 京

Jonathan Levin
Mac OS X and iOS Internals: To the Apple's Core
EISBN：978-1-118-05765-0

图书在版编目(CIP)数据

深入解析 Mac OS X & iOS 操作系统 / (美) 莱文(Levin, J.) 著；郑思遥，房佩慈 译.
—北京：清华大学出版社，2014

书名原文：Mac OS X and iOS Internals: To the Apple's Core

ISBN 978-7-302-34867-2

Ⅰ. ①深… Ⅱ. ①莱… ②郑… ③房… Ⅲ. ①操作系统 Ⅳ. ①TP316.84

中国版本图书馆 CIP 数据核字(2014)第 025702 号

责任编辑：王　军　刘伟琴
装帧设计：牛艳敏
责任校对：邱晓玉
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　印　　张：46.25　　字　　数：1241 千字
版　　次：2014 年 3 月第 1 版　　印　　次：2014 年 3 月第 1 次印刷
印　　数：1～4000
定　　价：108.00 元

产品编号：

译 者 序

根据网络分析公司Net Applications的数据，截至2013年10月，OS X的市场占有率已经达到7.73%，而且仍在稳步增长。OS X不仅占据了各种高端专业领域，在服务器领域中也能见到其身影。在移动平台上，iOS(包括手机和平板电脑)的市场占有率具有绝对优势，达到了55.39%。排名第二的是种类繁多的Android操作系统，占有率刚过30%。各种数据表明，苹果系的操作系统——OS X和iOS——正变得越来越重要。虽然市场占有率主要得益于苹果公司对生态圈建设所做的努力，但是其操作系统的核心技术也是重要的幕后功臣。我作为操作系统的爱好者及研究者，一直遗憾图书市场缺乏深入介绍苹果操作系统内核的书籍，而介绍其他操作系统的书籍，特别是Linux的书籍，则汗牛充栋。甚至连闭源的Windows都有不少经典好书。与苹果操作系统有关的著作主要是介绍各种BSD的书籍和学术论文，以及早期的一些来自于卡内基梅隆大学的与Mach内核相关的学术著作。当然还有一本不得不提的书籍是Amit Singh的经典著作*Mac OS X Internals: A Systems Approach*，该书同样也有些年头了，主要介绍的是PowerPC平台的OS X操作系统，当然也不会涉及iOS。而现在这本《深入解析Mac OS X & iOS操作系统》则很好地填补了这个遗憾。

本书不是一开始就讲解内核，而是从现象出发，首先从“超级用户”的角度来讲解苹果的内核提供的各种功能，以及有自己特色的地方。然后再进入内核，从Mach和BSD的角度分别讲解内核中各个子系统的实现原理。讲解内核的时候，基本上以各个子系统提供的API和数据结构为脉络，全面而深入地涵盖内核实现的各种细节。

本书不仅涉及开源XNU核心的内容，还涉及不少关于iOS的闭源XNU核心的内容，这也是本书的一大特色。由于iOS的核心是闭源的，所以本书多采用逆向工程的方法，对汇编代码进行分析，顺便介绍了各种逆向工程方法在越狱中的应用，使读者可以了解神秘的越狱过程。此外，书中还有各种和苹果操作系统开发或越狱相关的八卦趣闻，因此本书也是一本有趣的书。

本书不是操作系统的教材，因此没有介绍操作系统的基本原理。阅读本书要求读者具备操作系统工作原理方面的相关知识，例如操作系统的基本架构、硬件的引导过程、进程及其调度的基本原理、虚拟内存和内存管理的基本原理、文件I/O以及网络I/O的基本原理。读者最好也熟悉苹果的操作系统，了解Mac OS X和iOS的基本操作和特性。此外，为了能读懂书中的示例代码，还要求读者熟悉C语言编程，最好也了解一些汇编语言的知识。

翻译本书旨在为传播知识贡献一点绵薄之力，译文中如有不当之处，敬请广大读者批评指正。

郑思遥

2013年10月于北京

作 者 简 介

Jonathan Levin是一位经验丰富的技术培训师和咨询师，他的关注点是“三大系统”(Windows、Linux和Mac OS)以及它们的移动版本(Android和iOS)的内部工作原理。15年来，Jonathan坚持传播内核工程和修改技术的真知灼见，在DefCON以及其他技术会议上发表了很多技术演讲。他是Technologeeks.com公司的创始人和首席技术官(CTO)，这是由一些志趣相投的专家合伙创办的公司，致力于通过技术培训传播知识，通过咨询解决棘手的技术难题。他们的专业领域覆盖软件架构中的实时及其他关键部分、系统/内核级编程、调试、逆向工程以及性能优化。

技术编辑简介

Arie Haenel 是 NDS Ltd.(现属于 Cisco)的一位安全和底层专家。Haenel先生在整个数据安全和设备安全领域都有着丰富的经验。他拥有以色列耶路撒冷理工学院计算机科学系的科学工程学士学位，还拥有法国普瓦捷大学的 MBA 学位。他的兴趣包括学习犹太法典、柔道以及解谜语。他居住在以色列耶路撒冷。

Dwight Spivey 是好几本 Mac 相关著作的作者，包括 *OS X Mountain Lion Portable Genius* 和 *OS X Lion Portable Genius*。他还是 Konica Minolta 的产品经理，在那里他专门负责 Mac 操作系统、应用程序及硬件，还负责彩色和单色激光打印机。他在 Konica Minolta 教授 Mac 使用方面的课程，编写培训和技术支持材料，还是苹果开发者计划的成员。Dwight 和他漂亮的妻子 Cindy 及 4 个可爱的孩子 Victoria、Devyn、Emi 和 Reid 居住在阿拉巴马州的格尔夫海岸。他在越来越少的空闲时间会学习神学，画连环漫画，还会支持 Auburn Tigers 棒球队。

致　　谢

“你知道吗，Johnny”，我的朋友 Yoav 在上海的一个温暖的夏夜边吐着烟边对我说，“写本书吧!”

于是我就开始写这本书了。最初是 Yoav(Yobo) Chernitz 点燃了我写书的热情，这是一个改变，多年来我只读别人写的书。从那时起，在远东、中东还有美国东部(以及之间无数的航班上)，想法开始生根发芽，本书开始成型。我还不知道这本书会变得如此庞大，不知何时这本书开始有了自己的生命，让我付出很大的努力才能完成这本书。经历了无数次预料之外的复杂和延迟，真难以相信这本书完成了。我尝试覆盖这庞杂知识结构中最晦涩难懂的部分，描述这些知识，不留任何死角。下面应该由读者来评判我是否做到了。不过要知道，没有下面这些人的帮助我是不可能完成这本书的:

我的挚友 Arie Haenel——天生的黑客，绝不耍小聪明。总是给我最犀利的批判，绝对是技术评审的最佳人选。

Moshe Kravchik——作为本书的第一位读者提出了深刻的洞见和具有挑战性的问题，为后来的读者带来了一本可读性更好的书。

Yuval Navon——远在南半球的澳大利亚墨尔本，让我理解了好朋友是不会受到地域限制的。

最后，但绝对是同等重要的，我要感谢我亲爱的 Amy，她的耐心，她对我四处奔波的忍耐，以及无尽的理解支持我坚持到最后，她用无穷的智慧不断地提醒我，交稿的截止日期固然很重要，但对读者负责同样重要。

——Jonathan Levin

前　言

尽管 OS X已经诞生了十几年，但讨论OS X架构的书籍却少之又少，讨论 iOS 架构的书更是几乎没有。尽管关于 Objective-C、框架和 OS X 的 Cocoa API 的文档非常多，但是这些文档的讨论往往不够深入，缺少系统调用层次和实现细节。尽管也有一些关于内核的文档(大部分都是 Apple 公司提供的)，但也同样只关注驱动程序的构建(利用 I/O Kit)，只展示了一些优美的部分，而对于 XNU 的基础 Mach 核心却几乎没有任何涉及。XNU 是开放源代码的，但是尽管如此，却有着一百多万行的源代码(和注释)，有一些源代码甚至可以追溯到 1987 年，读起来绝对不是一件轻松愉快的事情。

而对于其他操作系统却并非如此。Linux也完全是一个开源的操作系统，但是从来不缺乏相关的书籍，O'Reilly就有很多非常棒的系列。Windows尽管是闭源的，但是Microsoft却提供了非常好的文档(其源码也在一些场合开放了)。本书对于XNU的意义，就好像Bovet和Cesati的*Understanding the Linux Kernel*对于Linux的意义，以及 Russinovich的*Windows Internals*对于Windows的意义。这两本书都是很棒的书，非常清晰地阐述了这些异常复杂的操作系统的架构。幸运的是，您正在读的这本书会用同样的方式讲解Apple的操作系统的内部工作原理。

其实之前有过一本关于Mac OS的书，这就是Amit Singh的优秀著作*MAC OS X Internals: A Systems Approach*，这是一本很棒的参考书，提供了大量有价值的信息。遗憾的是，该书针对的是PowerPC 架构，而且在Tiger之后(2006年左右)就没有任何更新了。从那时到现在，6年过去了。在这漫长的 6 年里，OS X 抛弃了 PowerPC 架构，完全移植到了 Intel 平台，而且已经经过了 4 个大版本的迭代。经历了 Leopard(美洲豹)、Snow Leopard(雪豹)、Lion(狮子)和最新的 Mountain Lion(山狮)，野生猫科动物的家族正在扩大，越来越多的新特性被加入操作系统中。不仅如此，OS X还经历了一次全新的移植。这一次移植的目标是 ARM 架构，改头换面成为了 iOS(根据某些统计资料，它是全世界领先的移动环境操作系统)。因此本书在前辈的基础上狗尾续貂，讨论 Apple 生态系统中新加入的猫科动物，还讨论了一些版本的 iOS。

Apple 的操作系统被认为是在不断地演进。本书最早是针对 iOS 5 和 Lion 编写的，但是这两个操作系统都在持续进化。在本书英文版即将付印的时候，iOS 的版本已经是 5.1.1 了，而且已经出现了 iOS 6即将发布的迹象。而 OS X版本依然在 Lion(10.7.4)，但是 Mountain Lion(10.8)已经推出开发者预览了，在本书英文版上架的时候应该已经发布正式版了。本书尽可能介绍最新的信息，覆盖所有的版本，并且持续地不断前进。

0.1 概述和阅读建议

这本书的规模很庞大。刚开始的时候，本来没有想把这本书设计得如此庞大细致，但是

随着我对 OS X的深入了解，我发现了更多的深奥难解的地方，而对于这些难题我找不到详细的解释或文档。因此我发现自己在编写这本书的过程中开始涉及越来越多的方面。一个操作系统是一个完整的生态系统，有着自己的地理形态(硬件)、大气(虚拟内存)、植被和动物(进程)。这本书尝试着尽可能有条理地记录这些内容，同时不牺牲细节的清晰性(或反之亦然)。这不仅仅是一个壮举。

0.1.1 架构一瞥

OS X和 iOS 有着复杂的架构，这个架构混杂了多项迥异的技术：NeXTSTEP 的 Cocoa 中遗留的 OS 9 对于 OS X 的 UI 和 API、BSD 的系统调用和内核层、源于 NeXTSTEP 的内核结构。尽管是一个融合体，但是各个组件之间的界线还是比较清晰的。图 0-1 给出了这个架构的鸟瞰图，并且标出了每一个组件对应本书的章节。

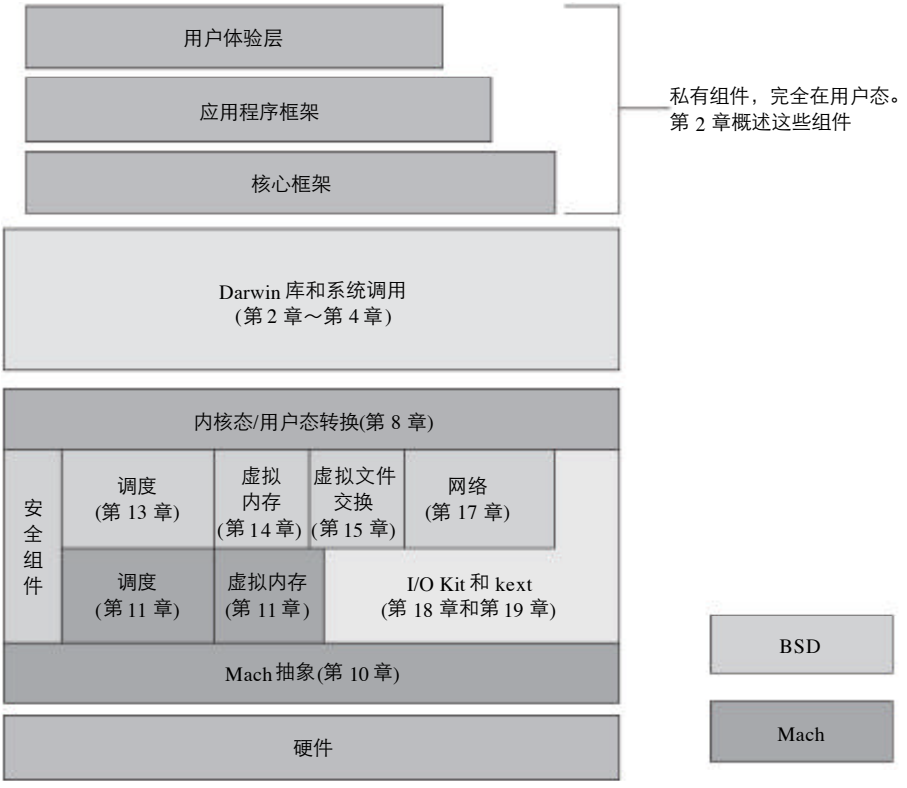

图 0-1 OS X 的架构以及每一个组件对应本书的章节

本书还另外包含了一些章节，介绍了和架构无关，但是非常重要的内容，例如调试(第 5 章)、固件(第 6 章)和用户态启动过程(第 7 章)、内核态启动过程(第 9 章)和内核模块(第 18 章)。最后还有两个附录，一个附录为 POSIX 系统调用和 Mach 陷阱提供了一个快速参考手册(这个附录在本书的支持网站上)，另一个附录对 Intel架构和 ARM 架构的汇编语言做了一个简单的介绍。

0.1.2　目标读者

有 4 类读者可能会对这本书的全部或部分内容感兴趣：

- 想更深入了解 OS X 工作原理的高级用户和系统管理员。Mac OS 的占有率正在稳步增长，争夺了多年来被 PC 霸占的市场份额。Mac 在企业环境中越来越流行，在学术界已经盖过了 PC 的风头。
- 不满足于 Objective-C 层面的用户态开发人员，这些人想要了解他们编写的程序是如何真正在系统层次执行的。
- 想要探寻内核态底层驱动程序设计、内核增强或者文件系统和网络层的内核态开发人员。
- 不满足于使用现有工具或补丁进行越狱的黑客和越狱者，这些人想要理解补丁的工作机理和修补的内容，以及如何进一步对系统进行调整使其能满足自己的需求。注意，在这个上下文中，这些目标读者指的是为了兴趣、快乐和挑战而深入了解内部工作原理的人，而不是那些为了任何违法邪恶目的的人。

0.1.3　选择自己的学习路径

尽管这本书可以从头读到尾，不过不要忘了这毕竟是一本技术书籍。书中的章节设计为可以单独阅读，既可以作为详细的解释也可以作为快速参考。既可以按顺序阅读所有章节，也可以随机阅读感兴趣的章节，略读甚至跳过某些章节，以后可以跳回来进行更深入的阅读。如果一章内引用了之前章节讨论的概念或函数，那么这些内容会清晰地标注出来。

您还可以根据自己所属的读者类型选择自己的阅读策略。例如，本书第 I 部分中的几章可以分解为图 0-2 所示的流程：

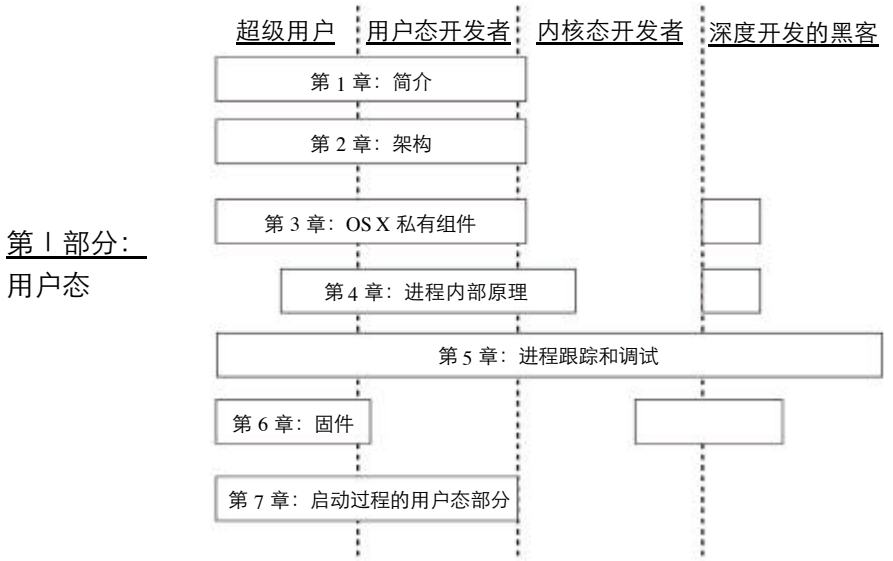

图 0-2　本书第 I 部分的阅读建议，这一部分关注于用户态的架构

在图 0-2 中，完整长度的条表示这一章的内容符合目标读者的兴趣，部分长度的条表示至少有部分符合目标读者的兴趣。当然，每一个读者的兴趣都不同。这也是为什么每一章开

头都要对本章讨论的内容做一个简单介绍的原因。同样地，只要看一下目录中每一章的小节标题就可以判断这一章是值得仔细阅读还是快速浏览即可。

本书的第 II 部分实际上可以自成一卷。这一部分关注于 XNU 内核的架构，因此比第 I 部分复杂得多。这是不可能避免的，内核本身就是一个更加复杂、实时且硬件受限的环境。这一部分包含更多的代码清单，甚至包含了少量用汇编实现的代码片段。本书第 II 部分的阅读建议如图 0-3 所示。

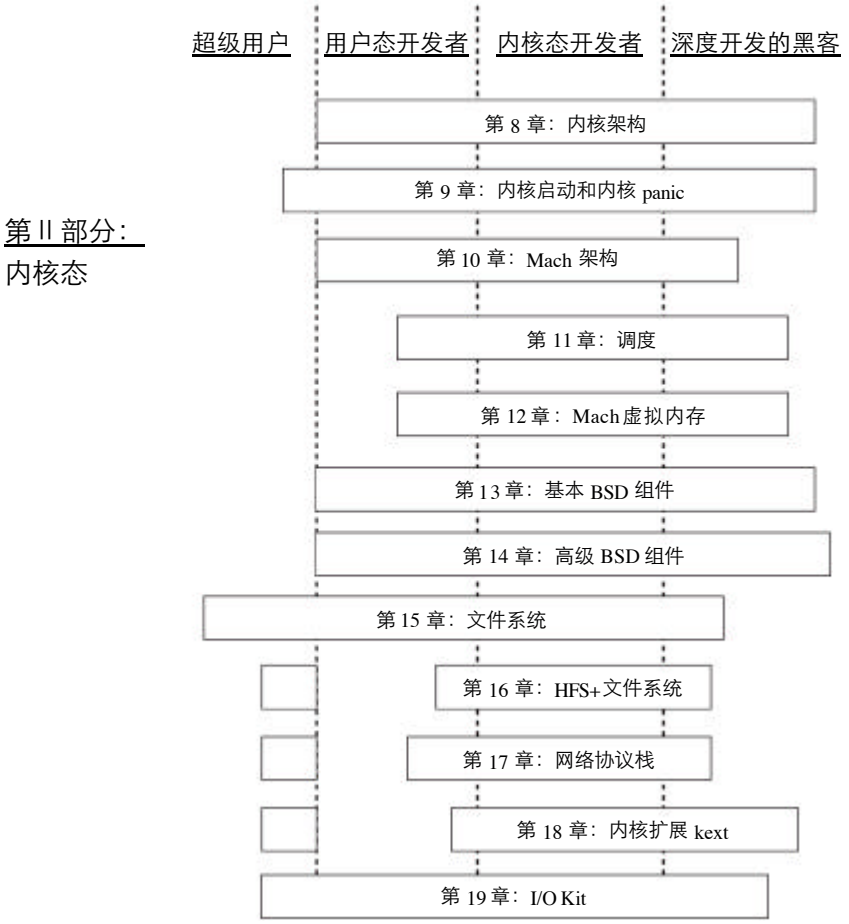

图 0-3　本书第 II 部分的阅读建议，这一部分关注于内核

0.2　实验

本书中很多章节都包含了“实验”，实验通常都需要运行一些 shell 命令，有时候需要运行一些示例程序。它们称为“实验”的原因是这些内容演示了操作系统中可能会变化的方面，这些方面可能会根据操作系统版本或配置发生变化。通常情况下，书中详细演示了这些实验的结果，但是鼓励读者在自己的系统上尝试这些实验，然后自己观察结果。和 UNIX 一样(Mac OS X 实现的就是一个 UNIX)，Mac OS X 也需要通过动手实践才能切身体会并且真正理解，

而通过道听途说是不够的。

在有些情况下，试验中有一些部分被留作练习让读者自己完成。尽管本书的支持网站上会有解答(练习的完整可用版本)，但是鼓励读者尽可能地自己独立完成这些部分。仔细地阅读本书，只要有少量的常识，您应该可以收获所有应该收获的内容。

0.3 工具

这本书还应用了一些工具，这些工具都是作者为了支撑这本书而开发的。这些工具都继承了 UNIX 的优良传统，都是命令行工具，而且输出结果很容易被 grep(1)处理，因此这些工具不仅适合手工使用，还适合在脚本中调用。

0.3.1 filemon

第 3 章介绍了一个名为“filemon”的工具，这个工具可以实时显示 OS X 和 iOS 上文件系统的活动。这个工具的命名是为了向 Russinovich 的同名工具致敬。这个简单的工具基于 OS X 和 iOS 5 上的 FSEvents 设备，能够跟踪文件系统相关的事件，例如文件的创建和删除。

0.3.2 psx

第 4 章介绍了一个名为 psx 的工具，这是一个类似 ps 的增强版工具，可以显示 OS X 中进程和线程的相关信息。这个工具在第 4 章中特别重要，该章讲解了进程的内部结构，并且演示了一个没有文档的系统调用 proc_info。如果查看的是自己创建的线程，这个工具不需要有特别的权限，但是查看其他进程则需要 root 权限。这个工具可以从本书的支持网站免费下载，带有完整的源代码。

0.3.3 jtool

对于大部分和二进制文件相关的功能，可以使用 OS X 自带的 otool(1)，这个工具能够很全面地分析数据区域(data section)，而且由于 ARM 架构具有两种模式的汇编，所以显示 ARM 二进制文件的时候会让人感到混淆。jtool 的目标是增强 otool，不仅解决了 otool 的这些缺陷，还提供了静态二进制分析的新特性。这个工具对第 4 章特别有用，该章详细分析了 Mach-O 文件格式，而且由于这个工具有很多有用的特性，例如寻找文件中的引用以及一些有限的反汇编能力，所以在后面几章也挺有用。这个工具可以从本书的支持网站免费下载，但是是闭源的。

0.3.4 dEFI

这个简单的程序可以导出 Intel Mac 上的固件(EFI)变量，还可以显示注册的 EFI 提供商。这个工具演示了基本的 EFI 编程技术——如何与引导服务和运行时服务进行接口。这个工具可以免费下载，带有完整的源代码。第 6 章介绍这个工具。

0.3.5 joker

第 8 章介绍的 joker 工具是一个简单的工具，这个工具可以用来和内核交互(特别是在 iOS 上)。这个工具可以查找并显示 iOS 和 OS X内核的系统调用和 Mach 陷阱表，显示 sysctl 结构，并且在二进制文件中查找特定的模式。这个工具非常适合于逆向工程和黑客一类的活动，因为 Apple 已经不再导出陷阱和系统调用符号了。

0.3.6 corerupt

第 11 章讨论 Mach 虚拟内存管理器的底层 API。为了演示这些 API 是多么强大(和危险)，本书提供了 corerupt 工具。通过这个工具可以将任何进程的虚拟内存映射以核心转储兼容的格式导出到一个文件，类似于 Windows 的 Create Dump File 功能，也很像 *Mac OS X Internals: A Systems Approach* 一书中使用的 gcore 工具。corerupt 在其前身的基础上有了增强，提供了对 ARM 的支持，还支持对虚拟内存映射进行入侵式的操作，例如修改内存页面。

0.3.7 HFSleuth

HFSleuth 是本书中的一个重要工具，这是一个用于查看 OS X 原生的文件系统 HFS+支撑结构的多合一命令行工具。之所以开发这个工具，是因为确实没有什么其他方式能够展示这个非常复杂的文件系统的内部工作原理。Singh 的书 *Mac OS X Internals: A Systems Approach* 中也包含了一个类似的工具，称为 hfsdebug，这个工具的功能少一些，而且只支持 PowerPC，后来被一个商业工具 fileXRay 所替代。

为了在真实的文件系统上使用 HFSleuth，必须能够读这个文件系统。一种方法就是引导这个文件系统。HFSleuth 的功能几乎都是只读的，所以可以保证这个工具非常安全。但是访问文件系统所在的底层块设备(有时候是字符设备)的访问权限通常是 rw-r-----，意味着设备不能被其他用户读访问。如果您通常情况下不信任 root，并且坚持要使用较低的权限(这是一个明智的决定！)，那么一个同等有效的替代方案是通过 chmod(1)修改 HFS+分区所在设备的权限，使得自己的用户有权限读(通常使用 o+r 参数)。一些高级功能(例如修复和 HFS+/HFSX 转换)要求写访问权限。HFSleuth 可以从本书的支持网站免费下载，而且会一直免费。不过和其前任一样，这个工具也不是开源的。

0.3.8 lsock

netstat -o 是一个大家都非常需要的功能，能够显示进程对系统中 socket 的所有权，但是这个功能在 OS X 中就没有了。在 OS X 中，这个功能要通过 lsof(1)实现，但是后者需要剔除其他打开的文件将 socket 滤出来，所以比较麻烦。另一个缺失的功能是显示正在创建的 socket 连接，就像 Windows 的 TCPMon 提供的功能一样。第 17 章介绍的这个工具使用了一个没有正式文档的内核控制协议 com.apple.network.statistics，这个协议可以获得 socket 创建时候的实时通知。这个工具特别容易集成到脚本中，因此可以很方便地用作连接事件处理程序。

0.3.9 jkextstat

本书中使用的最后一个工具是 jkextstat，这是一个 kextstat(8)兼容的工具，能够列出内核

扩展。和最初的版本不一样，这个工具支持详细输出(verbose)模式，而且可以用于 iOS。因此，这个工具对于 iOS 内核的探讨来说价值重大，而这件事在这本书出现之前是很困难的，因为 iOS 的二进制 kextstat 使用了不再被支持的 API。这个工具在最初的版本上有了改进，允许更详细的输出信息，能关注于特定的内核扩展，而且还支持输出到 XML 格式。

这里提到的所有工具都可以免费获得，而且会一直免费下去，不论您有没有购买(或复制)这本书。这是因为这些工具都很有用，而且填补了很多高级功能的空白，这些高级功能原本要么没有，要么隐藏在 Apple 自己的工具中。

0.4 本书使用的约定

为了使本书更容易阅读，避免经常反复强调示例代码和程序的具体背景，本书使用了一些约定，通过这些约定可以巧妙地提醒您给定代码清单的背景。

0.4.1 出场演员表

本书中的演示和代码清单自然是在各种不同版本的 Apple 电脑和 i-设备上产生和测试的。根据系统管理员给设备命名的习惯，每一台主机都有自己的“个性”和名字。为了避免重复说明每一个演示是在哪一台设备和操作系统上运行的，书中保留了 shell 的命令行提示符，通过其中的主机名可以找到这个演示所在平台对应的 OS X 或 iOS 版本(详见表 0-1)。

表 0-1　本书演示程序所用的主机名和版本信息

主 机 名	设 备 类 型	操作系统版本	使 用 目 的
Ergo	MacBook Air, 2010	Snow Leopard, 10.6.8	OS X 的一般性特性演示。在 Snow Leopard 及更新的版本上测试
iPhonoclast	iPhone 4S	iOS 5.1.1	iOS 5 及更新的版本在 A5 处理器(ARM 的多核处理器)上的特性测试
Minion	Mac Mini, 2010	Lion, 10.7.4	Lion 相关的特性演示
Simulacrum	VMWare 镜像	Mountain Lion, 10.8.0 DP3	Mountain Lion(开发者预览版)相关的特性演示
Padishah	iPad 2	iOS 4.3.3	iOS 4 及更新的版本的特性
Podicum	iPod Touch, 4G	iOS 5.0.1	iOS 5 相关的特性，在 A4 或 A5 处理器上

此外，带有 root@的命令行提示符表示只能通过 root 用户运行的命令。通过这个标志很容易区分哪些示例能运行在哪些系统上，以及需要什么权限。

0.4.2 代码节选和示例

本书包含了大量代码示例，分为以下两类：

- 示例程序：主要出现在第 I 部分。这些程序通常用于演示在用户态有效的简单概念和原理，或者演示一些特殊的 API 和库。这些示例程序都是由作者本人编写的，而且注释良好，您可以随意尝试这些程序，可以按照自己希望的方式修改，或者也可以不去碰这些程序。如果想要偷懒的话，还可以从本书的支持网站上下载这些程序的源代码和二进制程序。
- Darwin 的代码节选：主要出现在第 II 部分。这些代码几乎都是 XNU 代码的完整片段，截取自最新开源版本 1699.26.8(对应于 Lion 10.7.4)。所有代码都是开源的，但是要遵循 Apple 的 Public Source License。本书提供代码节选的目的是展示 XNU 架构中的相关部分。由于自然语言容易产生一些歧义，而代码既不用依赖上下文又准确(遗憾的是，有时候可读性不是那么好)，因此最准确的解释往往来自于对代码的阅读。当书中引用代码的时候，有可能指的是/usr/include 目录下的头文件(通过标准的 C 语言< >记号表示，例如<mach/mach-o.h>)。有时候，也有可能指的是来自于 XNU 或相关软件包的 Darwin 的源代码。在这种情况下会使用相对路径(例如 osfmk/kern/spl.c，指的是相对于 XNU 内核源代码解压的路径)。相关软件包总是会标注出来，不过书中第部 II 分引用的代码几乎都是 XNU 内核的代码。

XNU 和 Darwin 组件的代码都有很好的文档，不过本书尝试更进一步，在必要的时候在代码中以注释的形式添加额外的解释。为了区分，这种不属于原始代码的注释都是用 C++风格的注释清晰地标注出来，而不是 Darwin 中使用的 C 风格的注释，例如下面这个代码清单示例：

代码清单 0-1：示例代码清单

```
/* This is a Darwin comment, as it appears in the original source */

// This is an annotation provided by the author, elaborating or explaining
// something which the documentation may or may not leave wanting

// Where the source code is long and tedious, or just obvious, some parts may
// be omitted, and this is denoted by a comment marking ellipsis (...), i.e:

// ...

    important parts of a listing or output may be shown in bold
```

本书区分“输出清单”和“代码清单”。代码清单是直接从程序源代码文件或系统配置文件中摘抄出来的。而输出清单则是用户命令运行捕获的文本，用于演示在 OS X、iOS 或二者上运行的结果。本书旨在比较和对比两个系统，因此常常会见到同样的命令序列在两个系统上的执行结果。在输出清单中，可以看到用户命令用粗体表示，我们鼓励读者仿照书中在自己的系统上实验这些命令。

通常情况下，书中提供代码清单的目的是为了阐述清楚问题，而不是让人感到更迷惑。自然语言带有一定的二义性，但是代码则只有一种解释方式(即使有时候这样的方式并不完全清晰明了)。只要有可能，书中会用清晰的描述辅以详尽的图示，期望能够帮助您快速理解代码。对 C 语言(有时候要求一点汇编语言)当然能够帮助阅读书中的代码示例，但是并不是必要的。代码中的注释——特别是额外的注解——可以帮助您理解代码中的要点。书中使用更多的是框图和流程图，把函数当做黑盒子。读者可以自己选择是停留在大致了解的层次，还是深入研究实现中具体的变量和函数。不过值得注意的是，代码本身非常复杂，是很多人和很多编码风格的产物，在整个 XNU 中可以看到各种风格迥异的代码。

在 iOS 方面，XNU 仍然保持闭源。iOS 版本使用的 XNU 版本实际上比公开发布的版本领先很多版本。因而，书中也无法展示相应的示例源代码，但有时候会给出一些反汇编的代码(主要来自于 iOS 5.x)。这里使用的汇编是 ARM 汇编，代码中有一些帮助解释内部工作原理的注释(这些注释全部是由本书作者提供的)。对于和汇编相关的知识，可以参考本书的附录快速了解。

0.4.3　排版约定

本书约定命令后面括号中的数字表示这条命令所在 man 手册的节数(如果有的话)。例如：ls(1)表示一条用户命令，write(2)表示一个系统调用，printf(3)表示一个库函数调用，ipfw(8)表示一条系统管理命令。本书中描述的大部分命令和系统调用都在 man 手册中有很好的文档，本书也不打算替代精美手册的地位(因此，有问题请首先参阅手册)。然而，文档偶尔会遗漏一些内容——甚至根本没有听过文档记载——这时本书就会给出更多的信息。

0.5　支持网站

OS X和iOS都在高速地演进，而且会一直持续下去。我尽可能跟上演进的步伐，并且为本书更新维护了一个支持网站，地址是http://newosxbook.com。我的公司(http://technologeeks.com)也在LinkedIn上维护了OS X和iOS内核开发者小组(与那些Windows和Android的小组并列)，它包含一个问答论坛，这个论坛有希望成为一个热门的OS X和iOS相关问题的讨论场所。

本书支持网站的内容包括：

- 一个列出了各种 POSIX 和 Mach 系统调用的附录。
- 本书中所有实验中包含的示例程序——让有热情但是懒得敲代码的读者尝试。这些程序不仅提供了源代码的形式，还提供了二进制文件(给那些甚至懒得编译或不想使用 Xcode 的读者)。
- 本书中介绍的(也就是在这个前言中讨论的)工具，可以免费下载使用 OS X 平台和 iOS 平台的二进制文件，有些还提供了源代码。
- 其他网络资源的更新引用和链接。
- 随着时间的推移，会有一些关于新特性和改进的更新文章。
- 勘误表——人都会犯错误——尤其是 iOS 中的错误，因为大部分和 iOS 有关的细节都是通过反汇编挖掘出来的，这些内容中可能有一些不准确的地方或版本的差异需要修正。

本书是一段不可思议的旅程，您将带着(玩小猫的时候用的)护目镜，慢慢地揭开支持用户态应用程序的真实脉络。我真切地希望读者能像我一样得到启示，了解的不仅是关于OS X和 iOS 的思想，还有整个操作系统架构和软件设计的基本思想。

翻开这本书，开始这段奇妙的旅程吧！

目　录

第 I 部分

高级用户指南

第 1 章

达尔文主义：OS X 的进化史

Mac OS 从诞生之日起到现在发生了天翻地覆的变化。它从一小撮狂热人群的小众操作系统，慢慢地进入主流市场。随着近些年 Macbook、Macbook Pro 和 Macbook Air 前所未有地无处不在，Mac OS 显示出了爆炸式增长，打败了日渐衰弱的 PC，重新获得市场份额。根据一些统计数据，Mac OS 在移动平台上的衍生品 iOS 是市场份额最大的移动操作系统，在市场上和 Linux 的衍生品 Android 正面交锋。

然而这种增长并非是一夜之间突然发生的。事实上，这是一个非常漫长且痛苦的过程，从这个过程中可以看到 Mac OS 差点消亡，之后却重生为“OS X”。仅仅用“重生”这个词无法表达 Mac OS 的蜕变是多么彻底，Mac OS 经历了彻底的再造，整个架构推翻了重来。即使是这样，Mac OS 在重整旗鼓之前还面临了一个重大难题——那就是苹果向 Intel 架构的转变，完全抛弃了在 PowerPC 架构上的漫长历史。

在本书英文版面世之前，Mac OS 发布了最新最伟大的版本 OS X 10.7，也称为 Lion，同时还发布了当时最新的 iOS 操作系统 5.x 版。为了理解这些操作系统的特性以及两者之间的关系，我们首先要退一步理解这个架构统一是如何发生的。

我们不可能列出完整的特性列表，而只是一个高层次的概览。大家都知道苹果每次发布新版本都会添加数百项新特性，大部分新特性都与 GUI 和应用程序支持框架相关。而本书的重点则关注设计与工程特性。有关 Mac OS 版本相关的最新专著可以参阅 Amit Singh 在这个方面的文章[1]，还可以查阅 Ars Technica 全面的评论报道[2]。此外，维基百科还维护了一个非常完整的更新列表[3]。

1.1 前达尔文时代：Mac OS Classic

Mac OS Classic 是 Mac OS 在 OS X 之前的时代的名称。在那个时代，这个操作系统乏善可陈。的确，这个操作系统的创新确实是全 GUI 的系统(早期版本甚至没有像今天的“Terminal”这样的应用)。但是，这个操作系统的内存管理却很糟糕，而且多任务系统是协作式的(cooperative)——以今天的标准看，可以说是非常原始的。在协作式的多任务系统中，进程需要资源放弃自己的 CPU 时间片，如果进程都行为良好，这种方式就可以很好地工作。然而如果有的进程拒绝协作，即使只有一个

这样的进程，整个系统也都会停止运行。尽管如此，Mac OS Classic 还是给今天的 Mac OS(或称为 OS X)奠定了基础。这些基础主要包括“Finder”的 GUI，以及第一代 HFS 文件系统对“fork”的支持。时至今日，它们依然在影响着 OS X。

1.2　浪子回头：NeXTSTEP

尽管 Mac OS 面对强大的 PC 正在经受巨大的痛苦，其创始人史蒂夫 • 乔布斯离开了苹果公司(有一些说法是被驱赶出苹果公司的)，忙于一家新的完全不同的公司。这家公司是 NeXT，NeXT 公司生产专用硬件：NeXT 计算机和 NeXTstation，这些机器运行着专有的操作系统 NeXTSTEP。

NeXTSTEP 号称具有一些在当时前卫的特性：

- NeXTSTEP 采用的是 Mach 微内核系统，这个内核是卡内基梅隆大学(CMU)开发的一个鲜有人知的内核。尽管微内核的概念本身是一个创新，但是到现在也很少有采用微内核实现的操作系统。
- 使用的开发语言是 Objective-C，这是 C 语言的一个超集。Objective-C 和 C++不同，它是一个重度面向对象的语言。
- 面向对象的思想贯穿于整个操作系统。操作系统提供了很多框架(framework)和工具包(kit)，开发者可以使用丰富的对象库进行快速 GUI 开发，这些对象都基于 NSObject。
- 设备驱动开发环境也是一个面向对象的框架，称为 DriverKit。驱动程序可以是其他驱动程序的子类，继承其他驱动程序的功能，并且扩展其他驱动程序的功能。
- 应用程序和库以自包含的 bundle 形式发布。bundle 由一个固定的目录结构组成，用于封装一个软件包，其中带有自己所需要的依赖和相关的文件，因此软件的安装和删除就像移动一个目录一样简单。
- 系统中重度使用 PostScript，还包含一个称为“display PostScript”的变体，能够以 PostScript 的形式显示图像。因而对打印的支持也是一一对应的，而不像其他操作系统那样需要将要打印的文档转换为打印机能识别的格式。

NeXTSTEP 偏离了作为更优秀操作系统应该走的道路(还记不记得 OS/2)，现在已经绝迹了，只剩下一个 GNUStep 移植。然而，NeXTSTEP 的遗产仍然存活到今天。在 1997 年冬季的一天，苹果公司——只有一个没有任何发展前途的操作系统——最终收购了 NeXT 公司，将 NeXT 的知识产权带入了苹果，同时回到苹果的还有乔布斯。剩下的，按照人们说的，就成了历史了。

1.3　走进新时代：OS X 操作系统

作为收购 NeXT 公司的结果，苹果公司获得了 NeXTSTEP 架构中的 Mach 和 Objective-C 等设计。尽管 NeXTSTEP 本身已经不再发展了，但是其中的组件在 OS X 中获得了新生。事实上，可以将 OS X 看成是 Mac OS Classic 和 NeXTSTEP 的融合，更准确地说，应该是后者慢慢地吸收了前者。这个转变并不是瞬间发生的，Mac OS 经历过一个名为 Rhapsody 的临时操作系统，这个系统从未公布于众。然而，就是 Rhapsody 这个系统最终演化成了 Mac OS X 的第一个版本，而这个操作系统的内核也就是我们今天所熟知的 Darwin(中文名称：达尔文)。

在所有操作系统里面，Mac OS X 在设计上和实现上与 NeXTSTEP 最接近，甚至超过了苹果公司自己的 OS 9。OS X 的核心组件——Cocoa、Mach、IOKit、Xcode 的 Interface Builder 以及很多其他组件——都直接来自于 NeXTSTEP。这两个极端的小众的操作系统——一个有着伟大的 GUI 但是设计糟糕，一个设计很棒但是 GUI 非常平淡——融合的结果就是一个比两者加起来都要流行得多的全新操作系统。

OS X和 Darwin

有时候人们会混淆 OS X 和 Darwin 这两个名词的定义以及两者之间的关系。下面来澄清一下：

OS X 是整个操作系统的一个集体名称。根据下一章的讨论，这个操作系统由很多组件构成，Darwin 就是其中的一个组件。

Darwin 是操作系统的类 UNIX 核心，本身由内核(kernel)、XNU(“X is Not UNIX” 的缩写，这个缩写类似于 GNU 的递归式缩写)和运行时组成。Darwin 是开源的(iOS 中的 Darwin 是在 ARM 上的移植，这个 Darwin 则是不开源的，详见稍后的讨论)，而 OS X 中的其他部分，即苹果公司提供的各种框架，不是开源的。

OS X 的版本和 Darwin 的版本之间有一个简单的关系。除了 OS X 10.0 对应 Darwin 1.3.x 之外，其他的版本都服从以下简单的公式：

```
If (OSX.version == 10.x.y)
  Darwin.version = (4+x).y
```

因此，例如 Mountain Lion 的 10.8.0 版本对应 Darwin 12.0。Snow Leopard 的 10.6.8 版本对应 Darwin 10.8。尽管看上去有点混乱，但至少是一致的。

1.4　迄今为止的所有 OS X 版本

Mac OS X自诞生以来经历了很多版本。从一个创新但不成熟的操作系统开始，已经发展到现在成了一个功能丰富的平台 Lion(译者注：翻译本书时已经是 Mountain Lion)。下面各小节对每一个版本的主要特性进行了概述，特别是那些和架构或内核模式变化相关的特性。

1.4.1　10.0——Cheetah，初出茅庐

Mac OS X 10.0，代号为Cheetah(猎豹)，是OS X平台的第一个公共发布版。经过了一年的公共beta版Kodiak之后，苹果于2001年3月发布了10.0。这标志着Mac OS已经脱离了旧式的Mac OS，开始整合NeXT/Openstep的特性以及稍后要讨论的分层架构。这是对Mac OS 9彻头彻尾的重写，它和Mac OS 9共有的部分很少，可能只保留了Carbon接口，这是为了维持和OS 9 API的兼容性。10.0有5个子版本(10.0到10.0.4)，每个子版本之间都有较小的修改。核心操作系统软件包称为Darwin，版本号是1.3.1。XNU的版本号是123。

1.4.2　10.1——Puma，更强大

尽管 OS 10.0 绝对是一个创新的系统，但还是被认为不成熟不稳定，更不要提速度慢了。尽管号称支持抢占式多任务和内存保护，但是和其他所有竞争操作系统一样，还是给人们留下了太多期望。大概半年之后，苹果发布了 Mac OS X 10.1，代号为 Puma(美洲狮)，这个版本解决了稳定性和

性能的问题，还添加了更多改善用户体验的特性。随着 Puma 的发布，苹果也正式放弃 Mac OS 9，集中开发新一代 OS X。Puma 经历了 6 个子版本(10.1 到 10.1.5)，在 10.1.1 中，核心操作系统 Darwin 的版本号从 1.4.1 修改为 5.1，之后，Darwin 版本号遵循主版本号比 OS X小版本号大 4，并且小版本号和 OS X 子版本号一致的风格。XNU 版本是 201。

1.4.3　10.2——Jaguar，渐入佳境

一年之后，Mac OS X 10.2 发布了，代号为 Jaguar(黑豹)，这是一个更为成熟的操作系统，增强了大量的用户体验特性，还引入了“Quartz Extreme”框架用于更快速的图形显示。另一项新特性是苹果的 Bonjour协议(后来称为 Rendezvous)，这是ZeroConf的一种形式，是一种类似于 uPNP(Universal Plug and Play)的协议，允许苹果的设备可以在同一个局域网内互相发现(本书后面会讨论这个话题)。Darwin 升级为 6.0。OS X 10.2 经历了 9 个子版本(10.2 至 10.2.8，分别对应 Darwin 6.0 到 6.8)。XNU 版本为 344。

1.4.4　10.3——Panther 和 Safari

又过了一年，2003 年苹果发布了 Mac OS X 10.3，代号为 Panther(黑豹)，这个版本增强了更多的用户体验特性，例如 Exposé。苹果还开发了自己的网络浏览器 Safari，替换了 Internet Explorer for Mac，因为苹果和微软已经渐行渐远。

Panther 中另一个值得注意的功能增强是 FileVault，这个特性实现了透明的磁盘加密。Mac OS X 10.3 坚持了一年半，总共有 10 个子版本(10.3 到 10.3.9)，Darwin 也升级到了 7.x 版(7.0 到 7.9)。XNU 的版本为 517。

1.4.5　10.4——Tiger，转投 Intel 的怀抱

Mac OS 的下一个版本升级在 2004 年 5 月宣布，但是直到官方正式发布 Mac OS X 10.4(代号 Tiger，老虎)还花费了将近一年的时间。和往常一样，这个版本带有大量新的 GUI 特性，例如 spotlight 和 dashboard 小物件。更重要的是，这个版本还有重大的架构变化，其中最重要的是从 10.4.4 开始入侵 Intel x86 处理器的地盘，在那之前，Mac OS 都要求在 PowerPC 架构上运行。10.4.4 是第一个引入“通用二进制(universal binary)”概念的操作系统，通用二进制代码既可以在 PPC 架构上执行也可以在 x86 架构上执行。这个内核还有一个重大改进，就是支持 64 位的指针。

在这个版本中，其他和开发者相关的重要特性包括 4 个重要的框架：Core Data、Image、Video 和 Audio。Core Data 处理数据操作(撤消/重做/保存)。Core Image 和 Core Video 通过充分利用 GPU 的特性加速图形显示，Core Audio 将音频完全内建在操作系统中——允许 Mac 实现“文本到语音(TTS)”引擎、Voice Over 功能和神奇的“say”命令(“有一台能和您对话的电脑岂不妙哉？”)。

Tiger 的生命期延续了两年多，经历了 12 个子版本——10.4.0(对应 Darwin 8.0)到 10.4.11(对应 Darwin 8.11)。XNU 的版本号是 792。

1.4.6　10.5——Leopard 和 UNIX

Leopard 从 2006 年 6 月宣布到 2007 年 10 月发布也经历了一年多时间。Leopard 号称有数百项新特性。从开发者关心的角度看，主要更新有：

- Core Animation，界面动画的任务转交给了这个框架
- Objective-C 2.0

- OpenGL 2.1
- 增强了脚本语言，还增加了新的语言，包括 Python 和 Ruby
- DTrace(从 Solaris 10 移植而来)及对应的 GUI 和 Instrument
- FSEvents，实现了类似 Linux 的 inotify 的功能(文件系统/目录相关的通知)
- Leopard 开始完全遵循 UNIX/POSIX 规范

Leopard 经历了 10.5 到 10.5.8 版；对应 Darwin 9.0 到 9.8。XNU 版本号跳到了 1228。

1.4.7　10.6——Snow Leopard

Snow Leopard 的变化较少，主要都在底层的变化。现在这似乎已经成了一个传统，Snow Leopard 从 2008 年 6 月宣布到 2009 年 8 月发布经历了一年多的时间。从用户体验的角度看，变化很少，主要变化是所有的自带应用程序都移植到了 64 位。然而从开发者的角度看，则有重要的变化，其中包括：

- 完整的 64 位支持：既包括用户态的库，也包括内核空间(K64)。
- 文件系统层次的压缩：几乎感觉不到这个特性的存在，因为大部分命令和 API 所报告的都是文件的真实大小。而实际上，大部分文件——特别是操作系统相关的文件——都被透明地压缩以节省磁盘空间。
- Grand Central Dispatch：通过一个核心 API 实现对多核编程的支持。
- OpenCL：实现了将计算负载转移到 GPU，充分利用图形适配器用于非图形任务日益增长的计算能力。苹果公司最早开发了这个标准，因而依然对这个名字具有商标权。开发工作已经移交给 Khronos 小组(www.khronos.org)，这是由一些行业巨头(包括 AMD、Intel、NVidia 等很多公司)组成的合作组织。这个组织还领导了 OpenGL(用于图形)和 OpenSL(用于音频)。

Snow Leopard 完成了从 10.4.4 就开始的从 PPC 到 x86/x64 架构的移植。这个版本再也不支持 PowerPC 了，因此用于支持这个架构的通用二进制再也没有必要了，二进制文件的大小缩小了，还节省了很多磁盘空间。在实际中，大部分二进制程序还同时包含支持 32 位和 64 位 Intel 平台的代码。

Snow Leopard 的最新版本是 2011 年 7 月发布的 10.6.8(对应 Darwin 10.8.0)。XNU 版本是 1504。

1.4.8　10.7——Lion

Lion 是本书在写作时最新的 OS X 版本(更准确地说，应该是最新的公共版本，因为本书付印时 Mountain Lion 已经发布了开发者预览版)。这是一个比较高端的系统，要求运行在 Intel Core 2 Duo 及更新的处理器上(不过现在已经可以成功地虚拟化了)。

Lion 提供了很多新特性，不过很多都是用户态的特性。有一些新特性受到了 iOS(OS X 在 i 系列设备上的移植，稍后讨论)的严重影响。这些特性包括：

- iCloud：苹果全新的云存储服务，和 Lion 紧密结合，允许应用程序通过 Objective-C 运行时和 NSDocument 将文档直接保存在云中。
- 更强的安全性：采用最初用于 iOS 的应用程序沙盒运行(Sandboxing)和权限分离模型。
- 内建应用程序的更新：例如 Finder、Mail 和 Preview 都有更新，还有一些从 iOS 移植过来的应用程序，主要有 FaceTime 和类似 iOS 的 LaunchPad。
- 很多新的框架特性：从覆盖层滚动条和其他 GUI 改进，到 Voice Over 和类似 iOS 的自动文本纠错，到语言学和词性标注帮助实现基于自然语言处理的应用程序。

- Core Storage：支持逻辑卷，可以用于新的分区特性。一个特别有用的特性是支持将文件系统扩展在多个分区上。
- FileVault 2：用于加密文件系统，已经可以深入根宗卷(root volume)层次，标志着苹果进入了全盘加密(Full Disk Encryption，FDE)的世界。这项特性依赖于 Core Storage 在逻辑卷层次的加密功能。这个加密采用的是 XTS 模式的 AES-128 加密，这种模式对硬盘加密是特别优化的(Core Storage 和 FileVault 都在本书第 15 章详细讨论)。
- Air Drop：对苹果已经非常优秀的节点发现能力(Bonjour 的功劳)进行了扩展，允许主机间通过 WiFi 快速共享文件。
- 64 位模式：在更多的 Mac 机器上默认开启。尽管 Snow Leopard 已经使用了 64 位的内核，但是在非 Pro 版的 Macbook 上依然引导的是 32 位的内核。

在本书写作时，Lion 的最新版本是 10.7.3，XNU 版本是 1699.24.23。随着 Mountain Lion(目标版本 10.8)的正式公布，Lion 看上去会非常短命。

1.4.9　10.8——Mountain Lion

在 2012 年 2 月，就在本书要送印的前几天，苹果宣布了代号为 Mountain Lion 的 OS X 10.8，惊动了全世界。这是非常不同寻常的，因为苹果的操作系统生命期通常都有一年以上，特别是对 Lion(狮子)这样的大猫来说，很多人认为狮子会终结猫科动物。本书尽所有可能涵盖和 Mountain Lion 相关的最新内容，但是这个操作系统向公众发布时远晚于本书出版的时间，大概要在 2012 年的夏天(译者注：Mountain Lion 是 2012 年 7 月份发布的)。

Mountain Lion 的目标是要更紧密地将 iOS 和 OS X 集合在一起，就像本书猜测的一样(详见 1.6 节)。10.8 继续沿着 Lion 设置的这个趋势发展下去，将更多的 iOS 特性引入了 OS X，印证了宣传标语“Inspired by iPad, reimagined for Mac”。苹果公司广告中的大部分特性都是用户态的特性。不过有意思的是，内核似乎也进行了大版本更新，因为 XNU 的版本号比之前高了很多——2050。内核中一个值得注意的新特性是地址空间随机化，这个特性使得 OS X 能够更好地防备 Rootkit 以及内核相关的攻击。内核也只支持 64 位，抛弃了对 32 位 API 的支持。Darwin 12(以及 XNU)的源代码要等到 Mountain Lion 正式发布之后才会公开。

使用 uname(1)命令

在这本书中会展示很多 UNIX 命令和 OS X 特有的命令。uname(1)就是第一个这样的命令，这条命令显示 UNIX 系统名称。运行 uname 可以得到有关架构的详细信息以及 Darwin 的版本信息。这条命令有一些开关，-a 可以显示所有参数。下面在输出清单 1-1A 到 1-1B 中展示的代码片段演示了在两个不同的 OS X 系统上 uname 命令的输出：

输出清单 1-1A：通过 uname(1)查看 Darwin 在 32 位系统 Snow Leopard 10.6.8 上的版本信息

```
morpheus@ergo (~) uname -a
Darwin Ergo 10.8.0 Darwin Kernel Version 10.8.0: Tue Jun 7 16:33:36 PDT 2011; root:
xnu-1504.15.3~1/RELEASE_I386 i386
```

输出清单 1-1B：通过 uname(1)查看 Darwin 在 64 位系统 Lion 10.7.3 上的版本信息

```
morpheus@Minion (~) uname -a
```

```
Darwin Minion.local 11.3.0 Darwin Kernel Version 11.3.0: Thu Jan 12 18:47:41 PST 2012;
root:xnu-1699.24.23~1/RELEASE_X86_64 x86_64
```

如果在 Mountain Lion(开发者预览版)上运行 uname(1)，可以看到更新的版本：

输出清单 1-1C：通过 uname(1)查看 Darwin 在 64 位系统 Mountain Lion 10.8 (DP3)上的版本信息

```
morpheus@Simulacrum (~) uname -a
Darwin Simulacrum.local 12.0.0 Darwin Kernel Version 12.0.0: Sun Apr 8 21:22:58 PDT 2012;
root:xnu-2050.3.19~1
```

在非苹果的硬件上运行 OS X

根据苹果的规定，在苹果 Mac 产品线之外的任何硬件上运行 OS X 都是违反 EULA(最终用户许可协议)的。苹果公司对 Mac 克隆品发起了一场圣战，起诉了像 Psystar 这种尝试对 OS X的非苹果移植进行商业化的公司，而且还打赢了官司。然而，这并没有阻止很多狂热者将 OS X移植到普通 PC 机，以及最近在虚拟化环境下运行的尝试。

OpenDarwin/PureDarwin 项目利用开源的 Darwin 环境，将 Darwin 制作成一个可以完整引导并且安装的 ISO 镜像。之后 OSX86 项目在此基础上继续发扬光大，努力将 OS X 完整移植到 PC、笔记本、甚至上网本(这种行为通常称为“Hackintosh”，在中文中通常称为“黑苹果”)。有了可引导的 ISO 镜像，就可以绕过 OS X 安装程序的保护，从而可以在非苹果的硬件上安装系统。黑客(善良意义的)们通过一个基于苹果 Boot-132 的引导加载器(Chameleon，苹果在 Tiger v10.4.8 使用的临时引导加载器)仿真 EFI 环境(在 Mac 硬件上默认的环境，但是在 PC 上比较少见)。最早时，还需要对内核打一些补丁——不过这是可行的，因为 XNU 一直都开源。

随着虚拟化技术的崛起以及像 VMWare 这样优秀产品的出现，用户可以简单地下载预装好完整功能 OS X系统的 VM 镜像。最早可用的镜像是 Leopard 的后期版本的镜像，而且不容易得到，但是现在最新的 Lion 甚至 Mountain Lion 都可以从一些网站下载了。

尽管这种行为也是违反 EULA 的，但是苹果对于这种非商业版移植的态度似乎不是非常强硬。Lion 添加了一些需要因特网连接才能安装的特性(即“通过苹果验证产品”)，但是这一招并不能打压黑苹果的气焰。毕竟，人们在自己家里私下做的事情是自己的事情。

1.5　iOS——走向移动平台的 OS X

Windows 有 Windows Mobile，Linux有 Android，而 OS X 也有对应的移动平台分支——即万众瞩目的 iOS。iOS 最初称为 iPhone OS(在 2010 年中之前)，苹果(在和 Cisco 一个短暂的商标争端之后)将这个操作系统更名为 iOS，表示一统 i 系列设备的操作系统：iPhone、iPod、iPad 和 Apple TV。

和 OS X一样，iOS 也有版本的历史，本书写作时当前版本是 iOS 5.1。尽管所有的版本都有代号名称，但是这些名称都是苹果内部使用的，因而通常只有越狱社区才熟知这些名称。

1.5.1　1.x——Heavenly，第一代 iPhone

这个版本从 iPhone 在 2007 年中出现到 2008 年中。版本号从 1.0 到 1.02，然后从 1.1 到 1.1.5。最初唯一支持这个版本的设备是 iPhone，后来 iPod Touch 也支持了。最早的版本代号为“Alpine”(这也是 i 系列设备的默认 root 密码)，但是最后发布的版本代号为“Heavenly”。

从越狱者的角度来看，这个版本真的是heavenly(像天堂一样美好)。完整的调试符号，未加密，还容易反汇编。事实上，在后面的很多版本中，很多越狱者都依赖于从这个版本中提取出的符号和函数调用关系图。

1.5.2　2.x——App Store、3G 和企业级的特性

iPhone OS 2.0(代号为 BigBear)随着 iPhone 3G 一起发布，这软硬件两者都立即成为了热点。这个操作系统宣布了很多新的特性使得 iPhone 能够更加符合企业级的需求，例如 VPN 支持和对 Microsoft Exchange 的支持。这个操作系统还支持大量其他语言，标志着 iPhone 正式走向全球。

更重要的是，在这个版本中苹果推出了 App Store，这个平台后来成为世界上最大的软件分发平台。由于这个平台采用的分成模型，App Store 为苹果带来了更多的收益(由于这个平台太成功了，所以苹果后来在 Snow Leopard 中也推出了 Mac App Store，但不是那么成功)。

2.x 经历了 2.0～2.02 版本、2.1 版本(代号 SugarBowl)、2.2～2.2.1 版本(代号 Timberline)，直到 2009 年初才开始发布 3.x 版本。2.0.0 中的 XNU 版本是 1228.6.76，对应 Darwin 9.3.1。

1.5.3　3.x——告别第一代，迎来 iPad

iOS 的 3.x 版本带来了万众期待的剪切/粘贴功能，支持更小众的语言，支持 spotlight 搜索，还对内建的应用程序有很多增强。从更技术的层面说，这是第一个允许 tethering(个人热点)的 iOS，也是第一个可以插入 Nike+接收器的 iOS。这表明 i 系列设备不仅能自己作为客户端，本身还能作为其他附加设备的主机。

3.0 版(代号为 KirkWood)很快被 3.1 版(代号为 NorthStar)取代，3.1 版最高版本号为 3.1.3，这也是“第一代”设备能支持的最新操作系统。3.2 版(代号为 WildCat)在 2010 年 4 月发布，这是特别为称为 iPad 的平板电脑所发布的(这一代 iPad 后来被世人嘲笑)。当这个系统被 Comex(Star 2.0)通过基于 Web 的方式越狱后，被打上补丁升级为 3.2.2，这也是 3.x 系列的最后一个版本。3.1.2 中的 Darwin 版本是 10.0.0d3，XNU 版本是 1357.5.30。

1.5.4　4.x——iPhone 4、Apple TV 和 iPad 2

自 2010 年 6 下旬发布 iOS 4.0 和 iPhone 4，iOS 4.x 版本引入了很多新特性和应用，例如 FaceTime 和语音控制。iOS 4.x 版本是第一个支持真正多任务的版本，不过越狱的 3.x 版本也提供了不成熟的多任务 hack。

iOS 4 是生存期最长的 iOS 版本，经历了 4.0～4.0.2(代号为 Apex)、4.1(代号为 Baker 或 Mohave，这是 iOS 的第一个 Apple TV 版)以及 4.2～4.2.10(代号为 Jasper)。版本 4.3(代号为 Durango)开始支持 iPad 2 和 iPad 2 搭载的全新双核 A5 芯片。这个版本的 iOS 的另一个新特性是 Address Space Layout Randomization(地址空间布局随机化，即 ASLR，本书后面会详细讨论)，这是一项用户感觉不到的特性，但是苹果希望能够让黑客感到无法攻破。尽管苹果如此希望，但是 4.3.3 版本的 ASLR 还是被“Saffron”破解了，越狱者 Comex发布了天才的“Start 3.0”越狱工具，结束了 iPad 2“无法破解”的历史。苹果很快发布了 4.3.4 版本修复了这个漏洞(本书后面也会详细讨论)。苹果公司发现，阻止未来越狱的唯一方法就是追随越狱者本身——所以苹果公司把 Comex 招安了(译者注：不过后来，2012 年 10 月，Comex 离开了苹果公司)。4.3.x 的最后一个版本是 4.3.5，这个版本修复了另一个小安全漏洞。

iOS 4.3.3 对应的 Darwin 版本是 11.0.0，和 Lion 一致。然而 XNU 内核的版本是 1735.46.10，远远领先于 Lion。

1.5.5　5.x——iPhone 4S 和更新的硬件

在本书写作时，iOS 已经进入了第 5 个阶段：代号为 Telluride 的 5.0.0 和 5.0.1，以及代号为 Hoodoo 的 5.1，这两个代号名称都来自于滑雪度假胜地。iOS 5.0 和 iPhone 4S 同时发布，推出了苹果基于自然语言的语音控制服务 Siri(这项服务当时只支持这款手机)。iOS 5还带有很多新特性，例如众望所归的通知功能、NewsStand(电子出版的 App Store)以及一些 iOS 用户从来都不知道自己需要的特性，例如 Twitter 集成。另一项重大改进是 iCloud(同时在 Lion 中支持)。

由于民众对 5.0 的电池寿命抱怨很多，因此苹果立即发布了 5.0.1 版，不过抱怨还是没有消失。2012 年 3 月，版本 5.1 随着 iPad 3 发布了。

本书付印时，iPhone 4S 是最新最棒的型号，而且 iPad 3 也刚刚宣布，号称采用了带有 4 核图形处理器的 A5X 芯片。如果苹果仍然重复着自己的模式，那么下一个硬件很可能是大家都非常期待的 iPhone 5。苹果的更新周期到目前为止都是先更新 iPad，再更新 iPhone，最后更新 iPod。不过从 iOS 的角度看，这么做的影响并不重大，因为设备升级关注的一直都是采用更好的硬件，而软件特性添加得却不多。

Darwin的版本仍然是 11.0.0，但是XNU仍然远远领先于Lion：iOS 5.1 中的XNU版本为 1878.11.8。

1.5.6　iOS 和 OS X 对比

从本质上看，iOS 实际上就是 Mac OS X，但是两者之间还是有一些显著的区别：

- iOS 内核和二进制文件编译的目标架构是基于 ARM 的架构，而不是 Intel i386 和 x86_64。尽管目标处理器可能不同(A4、A5 和 A5X 等)，但都是采用 ARM 的设计。相比 Intel，ARM 的主要优势在于电源管理，因此 ARM 的处理器设计对于 iOS 这样的移动操作系统(及其强大对手 Android)来说都非常重要。
- iOS 的内核源码依然闭源——尽管苹果公司承诺 OS X内核 XNU 要一直开源，但是苹果公司的这个承诺显然在回避其移动版本。有时候，一些 iOS 修改会在公共开放的源代码中泄露(这些代码可以通过__arm__和 ARM_ARCH 的#ifdef 条件编译变量看出来)，不过这些泄露代码的数量随着新版本内核的发布越来越少。
- iOS 内核的编译稍有不同，关注的是嵌入式特性和一些新的 API。有一些新的 API 最终会进入 OS X，但是其他的不会。
- iOS的系统GUI是SpringBoard，这是大家熟知的触屏应用加载器；而OS X中的GUI的Aqua，是鼠标驱动的，而且特别为窗口系统所设计。由于SpringBoard如此流行，因此在Lion中以LaunchPad的形式移植到了OS X中。
- iOS 的内存管理要紧凑得多，因为在移动设备上没有几乎无穷的交换空间可以使用。因此，开发者需要适应更严酷的内存限制以及编程模型的变化。
- 系统的限制更严(或称为 jailed)，因此应用程序不允许访问底层 UNIX API(即 Darwin)，也没有 root 访问权限，而且只能访问自己的目录里的数据。只有苹果的应用才能有访问整个系统的权力。App Store 的应用被严格受限，而且必须通过苹果的审查。

最后一点区别是最重要的：苹果公司竭尽全力保证 iOS 作为一个移动平台操作系统的封闭性。实际上，这种做法将操作系统限制为只允许开发者访问苹果公司认为是“安全”或“推荐”的功能，而不允许开发者访问整个硬件的功能——而事实上苹果硬件的能力已经能媲美不错的桌面计算机了。但是这些限制都是人为添加的，也就是说，在其核心部分，iOS 几乎可以完成 OS X 能完成的一切任务。既然已经有了一个很优秀的操作系统，而且可以方便地进行移植，所以没有必要重新编写一个操作系统。此外，OS X 已经经历过一次从 PPC 到 x86 的移植，因此可以推断，移植到 ARM 也不会太困难。

不管您是否拥有 i 系列的设备，您肯定听过很多关于“越狱(jailbreaking)”过程的流行词汇，越狱过程是破解苹果设置的限制的过程。我们不要涉及这个过程的法律问题(有人说苹果雇员更像是律师而不是程序员)，只要知道所有的 i 系列设备(从第一代 iPhone 到 iPhone 4S)都可以被越狱(译者注：现在运行 iOS 6.1 的 iPhone 5 也能被越狱了)。苹果公司似乎在和越狱者玩猫捉老鼠的游戏，每个版本都比前一个版本的越狱难度更高，但是黑客们总是能找到苹果没有注意到的漏洞，让苹果感到非常懊恼。

本书中的大部分例子应用于 iOS 时都要求使用一台越狱的设备。如果没有越狱设备的话，还可以有另外一种选择：获得一份 iOS 软件更新——这个更新文件通常都被加密了，以防止人们窥探其中的奥秘——但是可以从一些 iPhone 专题的 Wiki 网站上找到一些泄露的解密密钥而轻松解密。通过将 iOS 镜像解密可以查看其文件系统，并且查看其中所有的文件，但是不能自己运行任何进程。因此，使用越狱的设备有更多的好处。(从苹果的角度看)越狱的坏处和(从微软的角度看)开源对健康的坏处是一样的。苹果发布了知识库文章 HT3743[4]告诉大家对 iOS 进行“非授权的修改”会产生多么可怕的后果。本书不会教您如何越狱，但是有很多网站分享了这方面的信息。

如果您要越狱您的设备，那么这个过程会安装一个名为 Cydia 的替换软件包，通过这个软件包可以安装未经苹果验证的第三方应用。尽管有很多这样的应用，但是本书中例子使用的应用只有这些：

- OpenSSH：允许通过 SSH 协议从任何客户端远程连接至设备，连接的客户端可以是 OS X、Linux(ssh 是一个原生的命令行应用)和 Windows(有很多可用的 SSH 客户端，例如 PuTTY)。
- Core Utilities：包装了 UNIX 的/bin 目录下能找到的一切基础工具。
- Adv-cmds 和 top：高级命令，例如查看进程的 ps 命令。

通过 SSH 连接到设备，首先可以尝试运行一下标准的 UNIX uname 命令，也就是之前在 OS X 中运行过的命令。例如，在运行 iOS 4.3.3 的 iPad 2 上运行这个命令可以看到类似以下的输出：

输出清单 1-2A：uname(1)命令在运行 iOS 4 的 iPad 2 上的输出

```
root@Padishah (/) # uname -a
Darwin Padishah 11.0.0 Darwin Kernel Version 11.0.0: Wed Mar 30 18:52:42 PDT 2011;
root:xnu-1735.46~10/RELEASE_ARM_S5L8940X iPad2,3 arm K95AP Darwin
```

在运行 iOS 5 的 iPod 上，可以看到以下输出：

输出清单 1-2B：uname(1)命令在运行 iOS 5.0 的第 4 代 iPod 上的输出

```
root@Podicum (/) # uname -a
Darwin Podicum 11.0.0 Darwin Kernel Version 11.0.0: Thu Sep 15 23:34:16 PDT 2011;
root:xnu-1878.4.43~2/RELEASE_ARM_S5L8930X iPod4,1 arm N81AP Darwin
```

因此，从内核的角度看，(几乎)就是同一个内核，但是目标架构是ARM(S5L8940X 是 iPad 2 上的处理器型号，也就是常说的 A5 处理器，而 S5L8930X 则是 A4 处理器的型号。新 iPad(译者注：也就是国内常说的牛排，第三代 iPad)硬件型号报告为 iPad3.1，处理器 A5X 的型号为 S5L8945X)。

表 1-1 列出了部分 OS X和 iOS 对应的 XNU 版本。从表中可以看出，在 4.2.1 之前，iOS 基本上和同时期的 OS X 使用的是同版本的 XNU。因此可以比较容易地对编译好的内核实施反向工程(而且还存在着大量的调试符号)。然而从 iOS 4.3 起，iOS 开始在内核增强方面发力，将 OS X 甩在后面。Mountain Lion似乎反过来又将iOS甩在后面了，但是很可能iOS 6出来时这个状况又会改变(译者注：作者没错，iOS 6.0 的 XNU 版本为 2107.2.33)。

表 1-1　各个版本 OS X 和 iOS 对应的内核版本以及发布日期

操 作 系 统	发 布 日 期	内 核 版 本
Puma (10.1.x)	2001 年 9 月	201.*.*
Jaguar (10.2.x)	2002 年 8 月	344.*.*
Panther (10.3.x)	2003 年 10 月	517.*.*
Tiger (10.4.x)	2005 年 4 月	792.*.*
iOS 1.1	2007 年 6 月	933.0.0.78
Leopard (10.5.4)	2007 年 10 月	1228.5.20
iOS 2.0.0	2008 年 7 月	1228.6.76
iOS 3.1.2	2009 年 6 月	1357.5.30
Snow Leopard (10.6.8)	2009 年 8 月	1504.15.3
iOS 4.2.1	2010 年 11 月	1504.58.28
iOS 4.3.1	2011 年 3 月	1735.46
Lion (10.7.0)	2011 年 8 月	1699.22.73
iOS 5	2011 年 10 月	1878.4.43
Lion (10.7.3)	2012 年 2 月	1699.24.23
iOS 5.1	2012 年 3 月	1878.11.8
Mountain Lion (DP1)	2012 年 3 月	2050.1.12

1.6　OS X 的未来

在本书写作时，Mac OS X 最新公共版本是 OS X 10.7 Lion，而此时 OS X 10.8 Mountain Lion 也蓄势待发。鉴于 OS X 10.8 的小版本号已经达到了 8，而且猫科动物已经用完了，所以很可能这是最后一款“OS X”操作系统了(当然，这只是一个猜测而已)(译者注：作者关于“猫科动物”猜对了，但是“最后一款 OS X”猜错了。10.8 之后还有 10.9，最新的 OS X 10.9 在 2013 年 9 月正式发布，这一代的 10.9 的代号为 Mavericks，这是加州一个冲浪圣地的地名)。

OS X 已经经历了十年的发展并成为一个成熟优秀的操作系统。不过从架构的角度看，变化并不算太大。除了到 Intel 架构的巨大转移和向 64 位平台的转移，内核本身在过去几个版本中的变化相对较小。那么我们会期待 OS XI 带来什么变化呢?

- 根除 Mach：内核中的 Mach API(本书详细地讲解了这套 API)是 NeXTSTEP 时代的遗毒。这些 API 大部分都是隐藏的，而大部分应用程序使用的是更为流行的 BSD API。尽管如此，Mach API 对整个系统来说都是至关重要的，如果突然移除了这些 API，几乎所有的应用程序都无法运行。尽管如此，Mach 不仅不方便，而且速度也比较慢。从本书后面的讲解可以看出，虽然基于微内核的消息传递架构可能很优雅，但是效率很难赶得上同时期的宏内核(monolithic kernel)(事实上，XNU 更趋近于宏内核架构而不是微内核架构，详见第 8 章的讨论)。如果将 Mach 全部移除并且将内核巩固为完整的 BSD，那么收益会是巨大的，不过这很有可能需要巨大的工作量。
- ELF 格式的二进制:Mac OS 无法完全融合进 UN*X世界的另一个困难在于坚持使用 Mach-O 二进制格式。尽管几乎所有其他的 UN*X 都支持 ELF 格式，但是 OS X 不支持，OS X 的整个二进制架构都基于遗留产物 Mach-O。如果 Mach 被移除了，那么 Mach-O 也没有存在的理由了，从而铺平了到 ELF 的转换之路。如果支持了 ELF 格式，再加上 OS X 已经提供的 POSIX 兼容性，那么 OS X将能提供源码和二进制的兼容性，允许 Solaris、BSD 和 Linux 的应用程序能够不经修改直接迁移到 OS X 上。
- ZFS：Mac OS 的原生文件系统 HFS+已经受到了大量的批评。HFS+是对 HFS 的补丁升级，而 HFS 是 OS 8和 OS 9中使用的文件系统。ZFS 能提供大量 HFS+无法提供的特性。尽管 Core Storage 向逻辑卷和多分区卷的实现迈进了一大步，但是还有很漫长的道路需要走。
- 和 iOS 的合并：目前的开发模式是，先在 OS X 中尝试一些新的特性，然后有时候会将一些特性移植到 iOS 上去，反之亦然。例如，目前 Lion 中的两个主要特性 Launchpad 和手势都源于 iOS。这两个系统在很多方面都非常类似，但是支持的框架和特性却仍有不同。Lion 引入了一些来自于 iOS 的 UI 概念，而 iOS 5.0 则引入了一些移植自 OS X 的框架。随着移动平台变得越来越强大，两个系统最终融合为一体也不是不可能的，从而铺平 iOS 应用在 OS X 上执行的道路。其实苹果以前实现过一个硬件架构翻译机制——Rosetta，用于在 Intel 上模拟 PPC 架构。

1.7　本章小结

多年来，Mac OS 已经取得了长足的发展。Mac OS 从操作系统中不起眼的小辈——只有一小撮死忠用户——发展为一个主流、现代且健壮的操作系统，获得了越来越高的市场占有率。而其移动平台上的衍生品 iOS 也是如今使用人数最多的操作系统之一。

接下来的章节将详细讨论 OS X内部结构：首先讨论基本架构，然后深入讨论进程、线程、调试和性能剖析。

参考文献

[1] Amit Singh's Technical History of Apple's Operating Systems: http://osxbook.com/book/bonus/chapter1/pdf/macosxinternals-singh-1.pdf

[2] ARS Technica: http://arstechnica.com

[3] Wikipedia's Mac OS X entry: http://en.wikipedia.org/wiki/Mac_OS_X

[4] "Unauthorized modification of iOS has been a major source of instability, disruption of services, and other issues": http://support.apple.com/kb/HT3743

第 2 章

合众为一：OS X 和 iOS 的架构

OS X和 iOS 是根据简单的架构原则和基础模块构建的。这一章讲解了这些基础模块，然后以自底向上的方式进一步关注系统中用户态的组件。内核态的组件会在本书的第 II 部分中更详细地讨论。

我们会比较和对比 iOS 和 OS X 的架构。可以看出，iOS 实际上是完整 OS X 精简之后的版本，和 OS X 有两大主要区别：首先 iOS 的架构是基于 ARM(而不是 Intel x86 或 x86_64)；第二，为了满足移动设备的局限性或特性需求，有一些组件被简化了或干脆被移除了。诸如 GPS、动作感应和触摸等概念最早是出现在 iOS 上的，而且在本书写作时这些概念也只能用于移动设备，但是现在这些概念也在逐步地整合进主流的 OS X Lion 中。

2.1 OS X 架构概述

和前辈 OS 9 相比，OS X 算是一个技术奇迹。整个操作系统从内至外全部重新设计了，而且改头换面成为了目前最具创新性的操作系统。不论是图形用户界面(GUI)还是底层的编程 API 接口，OS X 的很多特性都仍然算是创新的，而且很多特性正在快速地向 Windows 和 Linux 移植(甚至可以说是被抄袭)。

苹果的官方 OS X和 iOS 文档展示了一种非常优雅分层的方法，当然这个层次结构有一点过于简化：

- 用户体验层：包括 Aqua、Dashboard、Spotlight 和辅助功能(accessibility)等。在 iOS 中，用户体验层包括 SpringBoard，同时还支持 Spotlight。
- 应用框架层：包括 Cocoa、Carbon 和 Java。而在 iOS 中只有 Cocoa(严格地说应该是 Cocoa 的衍生品 Cocoa Touch)。
- 核心框架：有时候称为图形和媒体层。包括核心框架、Open GL 和 QuickTime。
- Darwin：操作系统核心——包括内核和 UNIX shell 环境。

在这些层次中，Darwin 是完全开源的，是整个系统的基础，并提供了底层 API。而上面那些层次则是闭源的，属于苹果私有的知识产权。

图 2-1 展示了这些层次的高层次架构。这幅图和苹果官方给出的图之间的差别在于这幅图以阶梯的形式画出了各个层次。这种阶梯式表示的意思是应用程序可以以直接和底层接口的方式编写，甚至可以完全存在于更低的层次中。例如，命令行应用程序虽然没有“用户体验层”的交互，但是可以和应用框架层或核心框架层交互。

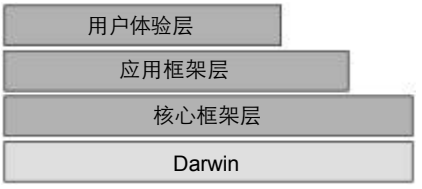

图 2-1　OS X 和 iOS 架构框图

简化到这个抽象程度时，两个系统的架构都可用这幅图表示。但是深入其中，还是可以发现差别的。例如，两个系统的用户体验层是不一样的：OS X 使用的是 Aqua，而 iOS 使用的是 SpringBoard。框架大体上是相似的，但是 iOS 包含一些 OS X 不包含的框架，OS X 也包含一些 iOS 不包含的框架。

尽管图 2-1 很好看也很整洁，但是太简化了，无法满足我们的需求。每一层都可以进一步分解为多个组件。本书关注的主要内容是 Darwin，而 Darwin 本身并不是一个单层的组件，而是一个层次化的架构，如图 2-2 所示。

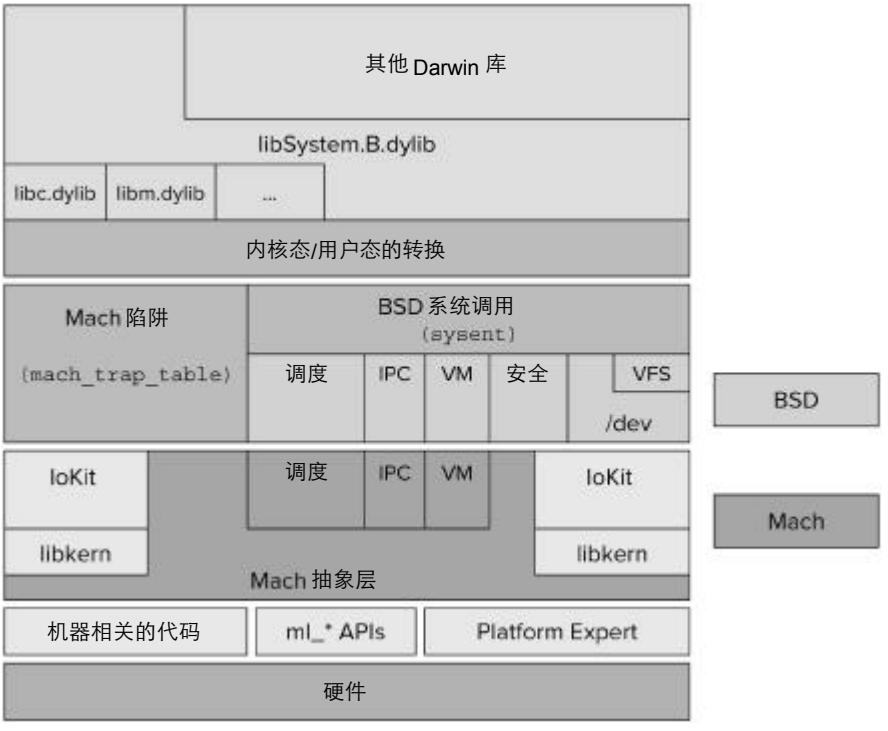

图 2-2　Darwin 架构

图 2-2 更接近Darwin(特别是内核XNU)的真实架构，不过也有一定的简化。这幅图揭示了一个恼人的事实：XNU实际上是由两种技术混合在一起的：Mach和BSD，此外还添加了一些其他的组件，主要是IOKit。意料之中的是，苹果提供的简图和文档都没有达到这个细节层次。事实上，苹果几乎不愿意承认Mach的存在。

关于这些简化的好处在于，忽略这些内容也是有好处的。大部分用户态的应用程序，特别是通过 Objective-C 编写的应用程序只需要使用到框架的接口——主要是 Cocoa 框架，这是推荐使用的应用框架；有时候还会使用一些核心框架的接口。因此大部分 OS X和 iOS 开发者实际上都忽略了这些更低层次、Darwin 的存在，更不用说内核了。尽管如此，用户态的每一个层次都可以被应用程序访问。在内核中，有一些组件可以被设备驱动的开发者访问。我们在接下来的几节将深入学习细节内容。具体地说，是 Darwin 的 shell 环境。本书的第 II 部分将深入内核的细节。

2.2　用户体验层

在 OS X 中，用户界面由用户体验层提供。OS X 引以为自豪的就是其提供的创新的特性，而且是有充分理由值得骄傲的。自 Cheetah 以来发布的漂亮界面一直在演化中，而且成为了其他系统仿制的目标，影响了其他基于 GUI 的操作系统，例如 Vista 和 Windows 7。

苹果将以下组件列为用户体验层中的组件：

- Aqua
- Quick Look
- Spotlight
- Accessibility 选项

尽管 iOS 的架构在用户体验层之下基本上是一致的，但是在用户体验层却有很大的区别。SpringBoard(大家熟悉的触屏驱动的 UI)完全负责所有的用户界面的任务(以及很多其他任务)。第 6 章会更详细地讨论 SpringBoard。

2.2.1　Aqua

Aqua 是大家熟知的 OS X GUI。Aqua 有很多特性，例如大家熟知的半透明窗口和图形特效，但是这些特性都不是本书关心的内容。本书关心的内容是如何具体支持这些特性。

系统的第一个用户态进程 launchd(第 6 章详细讨论)负责启动 GUI。支持 GUI 工作的主进程是 WindowServer(窗口服务器)。苹果故意没有提供这个进程的文档，而且这个程序是 Core Graphics 框架的一部分，而 Core Graphics 框架深深隐藏在另一个框架 Application Services 中。因此，这个程序的完整路径是/System/Library/Frameworks/ApplicationServices.framework/Frameworks/CoreGraphics.framework/Resources/WindowServer。

窗口服务器启动时传入了-daemon 开关参数。窗口服务器的代码实际上不完成任何实际的工作——所有的工作都是 CoreGraphics 框架中的 CGXServer(Core Graphics X Server)函数完成的。CGXServer 会检查自己是否以后台服务程序的方式和/或以主控制台 getty 的方式运行。然后在后台 fork 出自己的子进程。当子进程准备好之后，LoginWindow 进程(同样是由 launchd 启动的)启动交互式登录过程。

系统也可以引导进入控制台模式，就像以前 UNIX 时代一样。loginwindow 的设置在/etc/ttys 文件中，这个文件中 console 部分配置如下：

```
root@Ergo (/)# cat /etc/ttys | grep console
#console "/usr/libexec/getty std.57600" vt100 on secure
console "/System/Library/CoreServices/loginwindow.app/Contents/MacOS/
loginwindow" vt100 on secure onoption="/usr/libexec/getty std.9600"
```

将第一个 console 行注释掉，系统就可以以单用户模式引导。另外还有一种方法：在 System Settings 中选择 Users & Groups，然后进入 Login Options，将 Display login window as 项设置为 Name and password，这样的话在登录界面中，用户名部分填写“>console”，密码留空，就可以进入系统控制台了。如果想要回到 GUI，只要按 CTRL+D(或者在登录 shell 中运行 exit 命令)就可以恢复 WindowServer。还可以试试输入“>sleep”和“>reboot”。

2.2.2 QuickLook

QuickLook是Leopard(10.5)引入的一项新特性，允许在Finder中快速预览多种不同类型的文件。不需要双击鼠标来打开文件，只需要按下空格键就可以通过QuickLook快速预览文件内容。QuickLook采用的是可扩展的架构，使得大部分工作都由插件完成。这些插件是后缀为.qlgenerator 的 bundle，只要将这些bundle 文件拖放到 QuickLook 目录(系统范围的 QuickLook 目录为/System/Library/QuickLook，针对用户个人的QuickLook目录为~/Library/QuickLook)中即可完成插件的安装。

bundle 是 OS X中使用的一项基础软件部署架构，本章后面会对此详述。现在只要知道一个 bundle 是一个遵循固定结构的目录层次即可。

实际的插件是一个特殊编译的程序——但不是一个独立的可执行程序。插件程序没有传统的 main()入口点，而是实现了 QuickLookGeneratorPluginFactory 入口点。另外还有一个配置文件负责将插件和对应的文件类型关联起来。文件类型通过苹果的 UTI(Uniform Type Identifier)表示，这实际上就是逆 DNS 表示法(reverse DNS notation)。

为什么使用逆 DNS 表示法

使用逆 DNS 表示法作为软件包的标识符是有合理原因的。具体原因为：

- 因特网 DNS 格式用于形成主机名称全局唯一的层次名称空间，实际上组成一个树状结构，根节点为空域名(.)，顶级域名包括.com、.net 和.org 等。
- 在软件中使用类似名称空间的思想最早源于Java。为了避免名称空间的冲突，Sun(现为Oracle)注意到可以使用DNS名称(尽管是倒过来的)提供一种非常类似于文件系统的层次结构。
- 在阅读本书的过程中您会发现，苹果在 OS X 中广泛使用了逆 DNS 表示法。

quicklookd(8) 是系统的“QuickLook 服务器”，是通过 /System/Library/LaunchAgents/com.apple.quicklook.plist 文件在登录时启动的。这个后台服务程序本身在 QuickLook 框架中，而且不带 GUI。qlmanage(1)命令的作用是维护插件，并且控制后台服务程序。这条命令的使用如输出清单 2-1 所示：

输出清单 2-1：演示 qlmanage(1)的使用

```
morpheus@Ergo (/) % qlmanage –m
  living for 4019s (5 requests handled - 0 generated thumbnails) -
  instant off: yes - arch: X86_64 - user id: 501
memory used: 1 MB (1132720 bytes)
last burst: during 0.010s - 1 requests - 0.000s idle
plugins:
  org.openxmlformats.wordprocessingml.document ->
/System/Library/QuickLook/Office.qlgenerator (26.0)
com.apple.iwork.keynote.sffkey -> /Library/QuickLook/iWork.qlgenerator
(11)
..
  org.openxmlformats.spreadsheetml.template ->
/System/Library/QuickLook/Office.qlgenerator (26.0)
  com.microsoft.word.stationery -> /System/Library/QuickLook/Office.qlgenerator (26.0)
  com.vmware.vm-package -> /Library/QuickLook/VMware Fusion
  QuickLook.qlgenerator (282344)
  com.microsoft.powerpoint.pot -> /System/Library/QuickLook/Office.qlgenerator (26.0)
```

2.2.3 Spotlight

Spotlight 是苹果在 Tiger(10.4)中引入的一项快速搜索技术。在 Leopard 中，Spotlight 已经无缝地整合进了 Finder。从 iOS 3.0 开始，Spotlight 被移植到了 iOS。在 OS X 中，用户单击系统菜单栏右侧的放大镜图标就可以使用 Spotlight。在 iOS 中，用户用手指滑向主屏幕画面左侧就可以打开类似的窗口。

Spotlight背后的核心力量是一个索引服务器mds，mds在MetaData框架中，而这个框架是系统核心服务的一部分(路径为/System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Support/mds)。mds是一个没有GUI的后台服务程序。每当有任何文件操作(创建、修改和删除)发生时，内核都会通知这个后台服务程序。这个通知机制称为fsevents，本章后面会对此详述。

当mds收到通知时，mds会通过工作进程(mdworker)将各种元数据信息导入数据库。mdworker进程可以加载一个具体的Spotlight Importer(Spotlight导入器)从文件中提取元数据信息。系统提供的导入器位于/System/Library/Spotlight 目录，用户提供的导入器位于/Library/Spotlight目录。和QuickLook类似，这些导入器都是实现了固定API的插件(在Xcode中选择MetaData Importer项目模板时可以创建出API框架)。

在命令行可以通过以下命令访问 Spotlight：

- mdutil：管理元数据数据库
- mdfind：发出 spotlight 查询
- mdimport：配置和测试 spotlight 插件
- mdls：列出文件的元数据属性
- mdcheckschema：验证元数据的布局(译者注：这个工具在 10.8 中移除了)

- mddiagnose：Lion 引入的新功能，这个工具能对 spotlight 子系统(mds 和 mdworker)以及系统上的附加数据进行完整的诊断。

另外一个文档稀少的特性是通过在一些路径创建特定的文件控制 Spotlight(即 mds)的行为。例如，在一个目录中创建一个.metadata_never_index 隐藏文件可以防止这个目录被索引(最初是为可移动媒体设计的)。

2.3 Darwin——UNIX 核心

OS X 中的 Darwin 是一个完全成熟的 UNIX 实现。苹果公司一点也没想隐藏这个事实，反倒是引以为自豪。苹果专门维护了一份文档[2]，重点阐述 Darwin 中和 UNIX 相关的特性。Leopard(10.5)是第一个通过了 UNIX 认证的 OS X版本。然而，UNIX 界面对于大多数用户来说是隐藏的：GUI 环境巧妙地隐藏了底层的 UNIX 目录结构。由于本书关注的内容是操作系统内部的工作原理，所以大部分讨论和示例都是在 UNIX 命令行下进行的。

2.3.1 Shell

命令行的访问是一件很简单的事情：Terminal 应用程序会打开一个带有 UNIX shell 的终端模拟器。默认使用的 shell 是/bin/bash，即 GNU 的“Bourne Again”shell，不过 OS X 还提供了其他 shell 可供选择：

- /bin/sh(Bourne shell)：最基本的 UNIX shell，由 Stephen Bourne 创建。在 1977 年之前被认为是标准 shell。有一定的局限性。
- /bin/bash(Bourne Again shell)：默认 shell。和基本的 Bourne shell 反向兼容，但是先进得多。在很多操作系统(例如 Linux 和 Solaris)上是现代的标准 shell。
- /bin/csh(C-shell)：一个基本 shell 的替代品，采用类似 C 的语法。
- /bin/tcsh(TC-shell)：类似 C-shell，但是有着更为强大的别名、自动完成和命令行编辑特性。
- /bin/ksh(Korn shell)：另一个标准 shell，由 David Korn 在 20 世纪 80 年代创建。适合高效的脚本编写，但是对命令行环境却不是很友好。
- /bin/zsh(Z-shell)：慢慢成为标准，官方网站为 http://www.zsh.org 。对 Bourne shell 和 Bourne Again shell 完全兼容，有着更为强大的特性。

OS X(和 iOS)中的命令行工具也可以通过 telnet 或 SSH 远程访问。这两种远程访问方式默认都是禁止的，而且严重不推荐 telnet 方式的访问，因为这种方式本质上不安全，而且是不加密的。而 SSH 顺势成为了替代品(以及之前的 Berkeley“R 系列工具”，例如 rcp/rlogin/rsh)。

在 OS X 上可以方便地启用 telnet 和 SSH，只要编辑/System/Library/LaunchDaemons 目录中对应的 plist 文件即可(telnet.plist 和 ssh.plist)，将 Disabled 键的值设置为 false 即可(或者干脆删掉这个键)。不过要编辑这个文件，首先要获得 root 权限——例如 sudo bash(或其他 shell)。

在 iOS 上，SSH 也是默认禁用的，但是在越狱的系统上，在越狱的过程中就安装并启用了 SSH。可以交互式登录的两个用户是 root 和 mobile。默认 root 密码为 alpine，这也是第一个版本 iOS 的代号名称。

2.3.2　文件系统

Mac OS X 使用了 Hierarchical File System Plus(简称 HFS+)文件系统。名称中的“Plus”表示 HFS+是原来在 OS X 之前的系统中使用的老 Hierarchical File System 文件系统的继任。

HFS+有 4 个不同的变种：

- 大小写敏感/不敏感：HFS+总是会保存大小写的区别，但是可以对大小写敏感或不敏感。当设置为大小写敏感时，HFS+指的就是 HFSX。HFSX 大概是在 Panther 引入的，尽管没有在 OS X 中使用，但是是 iOS 上默认的文件系统。
- 可选的日志功能：HFS+还可以选择打开日志功能，这种情况下通常称为 JHFS(或 JHFSX)。通过使用日志，文件系统可以在强行卸载(例如电力中断)的情况下更加健壮，因为日志文件系统通过一个日志记录文件系统事务完成的过程。如果文件系统在挂载时发现日志中包含了事务，那么既可以重放事务(即完成事务)，也可以抛弃事务。尽管有可能会丢失数据，但是文件系统更有可能维护一致性的状态。

在 OS X的大小写不敏感的文件系统中，文件创建时可以选择任何大小写组合的文件名，而且显示时会完全按照创建时采用的方式显示，但是访问时却可以使用任何大小写组合。因此，两个文件绝对不可以采用同一个不考虑大小写完全相同的名字。然而，不小心打开 caps lock 却不会影响文件系统的操作。不信的话可以试试 LS /ETC/PASSWD 命令。

在 iOS 中，默认使用的是大小写敏感的 HFSX。在这个文件系统中，不仅会保留大小写，而且多个文件可以具有除了大小写之外其他字符都相同的文件名。很自然，大小写敏感也意味着输入错误会导致完全不同的命令或引用完全不同的文件，通常情况下都会产生错误。

HFS 文件系统具有独特的特性，例如扩展属性和透明压缩，第 15 章会深入讨论这些特性。然而从编程的角度看，HFS+和 HFSX 的接口与其他文件系统都是一样的，因为内核提供的 API 实际上是通过一个公共的文件系统适配层提供的，这个适配层名为虚拟文件交换(Virtual File system Switch，VFS)。VFS 是内核中使用的所有文件系统的统一接口，既适用于 UNIX 的文件系统也适用于外部文件系统。同样，HFS+和 HFSX 都提供了用户“默认”会使用的功能，这些功能是 UNIX 文件系统共有的体验——权限、硬连接和软连接、文件所有权和类型，这些体验和其他 UNIX 都是一样的。

2.4　UNIX 的系统目录

OS X 是一个符合 UNIX 标准的系统，因此也有那些标准 UNIX 具有的目录结构：

- /bin：UNIX 中的二进制程序。这是常用 UNIX 命令(例如 ls、rm、mv 和 df 等)所在的地方。
- /sbin：系统程序。这些二进制程序用于系统管理，例如文件系统管理和网络配置等。
- /usr：User 目录。这并不是说这个目录是给用户用的，而更像是 Windows 中的 Program Files 目录，第三方的软件可以安装在这里。
- /usr：目录中包含的 bin、sbin 和 lib。/usr/lib 用于存放共享的目标文件(类似于 Windows 中存放 DLL 文件的\windows\system32 目录)。这个目录还包含一个 include/子目录，所有标准的 C 头文件都存放在此。

- /etc：其他文件。这个目录包含了大部分系统配置文件，例如密码文件(/etc/passwd)。在 OS X 中，这个目录实际上是指向/private/etc 的符号链接。
- /dev：BSD 设备文件。这些特殊文件表示了系统中存在的设备(字符设备和块设备)。
- /tmp：临时目录。这是系统中唯一所有人都可写的目录(权限为 rwxrwxrwx)。在 OS X中，这个目录实际上是指向/private/tmp 的符号链接。
- /var：各种杂项文件。这个目录中保存了日志文件、邮件存储、打印队列和其他数据。在 OS X 中，这个目录实际上是指向/private/var 的符号链接。

UNIX 目录在 Finder 中是看不见的。通过 BSD 的系统调用 chflags(2)可以设置一个特殊的文件属性“hidden”，这个属性控制文件是否从 GUI 视图中隐藏。ls 有一个非标准的选项-O 可以显示文件属性，如输出清单 2-2 所示。第 14 章讨论其他的特殊文件属性，例如压缩属性。

输出清单 2-2：通过 ls 的非标准的“-O”选项显示文件属性

```
morpheus@Ergo (/) % ls –lO /
drwxrwxr-x+ 39 root admin  -        1326 Dec  5 02:42 Applications
drwxrwxr-x@ 17 root admin  -         578 Nov  5 23:40 Developer
drwxrwxr-t+ 55 root admin  -        1870 Dec 29 17:23 Library
drwxr-xr-x@  2 root wheel  hidden     68 Apr 28 2010 Network
drwxr-xr-x   4 root wheel  -         136 Nov 11 09:52 System
drwxr-xr-x   6 root admin  -         204 Nov 14 21:07 Users
drwxrwxrwt@  3 root admin  hidden    102 Feb  6 11:17 Volumes
drwxr-xr-x@ 39 root wheel  hidden   1326 Nov 11 09:50 bin
drwxrwxr-t@  3 root admin  hidden    102 Jan 21 02:40 cores
dr-xr-xr-x   3 root wheel  hidden   4077 Feb  6 11:17 dev
...
```

2.4.1 OS X 特有的目录

OS X 在 UNIX 目录树中添加了自己特有的目录。在系统根目录下，这些目录包括：

- /Applications：系统中所有应用程序的默认目录。
- /Developer：如果安装了Xcode，那么这是所有开发者工具的默认安装位置(译者注：现在Xcode 是以应用的形式通过Mac App Store 安装的，所以所有开发工具都在/Applications/Xcode.app这个bundle内了)。
- /Library：系统应用的数据文件、帮助和文档等数据都放在这个目录下。
- /Network：用于邻居节点发现和访问的虚拟目录。
- /System：系统文件目录。其中只包含一个Library子目录，这个子目录几乎包含了系统中的所有重要组件，例如框架(/System/Library/Frameworks)、内核模块(/System/Library/Extensions)和字体等。
- /Users：所有用户的主目录所在的目录。每一个用户在这里都会创建一个自己的目录。
- /Volumes：可移动媒体和网络文件系统的挂载点所在的目录。
- /Cores：如果启用了核心转储(core dump)，那么这个目录保存核心转储文件。如果 ulimit(1)命令允许创建核心转储，那么当进程崩溃时会创建核心转储文件，其中包含进程的核心虚拟内存镜像。第 4 章会详细讨论核心转储。

2.4.2　iOS 文件系统的区别

iOS 的文件系统和 OS X 非常类似，但是有以下区别：

- 文件系统(HFSX)是大小写敏感的(而 OS X 的 HFS+能保留大小写但是不敏感)。此外，文件系统是部分加密的。
- 内核已经以 kernelcache 的形式将内核扩展打包在内核中(kernelcache 在 /System/Library/Caches/com.apple.kernelcaches 中)。和OS X的内核缓存不同(OS X的内核缓存为压缩镜像)，iOS的内核缓存是加密的Img3 文件。详见第 5 章的讨论。

内核缓存在第 18 章详细讨论，现在只要将内核镜像想象为预配置的内核即可。

- /Applications 可能是指向/var/stash/Applications 的符号链接。这是越狱系统的一个特性，而不是标准 iOS 的特性。
- 没有/Users 目录，只有一个/User 目录，而这个目录实际上是指向/var/mobile 的符号链接。
- 没有/Volumes 目录(而且没有必要有这个目录，也没有必要进行磁盘仲裁，因为在 iOS 中无法在指定系统中添加更多的存储设备)。
- /Developer 目录只有在 iOS 设备被 Xcode 选中为“Use for development”时才会出现。在这种情况下，iOS SDK 中的 DeveloperDiskImage.dmg 镜像会在设备上挂载。

2.5　bundle

bundle 是 OS X 中的一个重要概念，是 NeXTSTEP 的遗产，而且随着移动应用的普及，已经成为了事实上的标准。bundle 的概念不仅是应用程序的根基，也是框架、插件、小物件(widget)、甚至内核扩展的根基，因为这些组件都被打包在 bundle 中。因此我们现在最好停下来先看一下 bundle，之后再讨论一种特殊的应用程序：框架。

在 Mac OS 中，“bundle”这个词实际上描述的是两种不同的术语：第一种是本节中讨论的目录结构(有时候也称为“包(package)”)；第二种是共享库目标的一种文件目标格式，共享库由进程显式地加载(和普通的库不同，普通的库是隐式加载的)。这个词有时候也表示一个插件。

苹果对 bundle 的定义是：“一种标准化的层次结构，保存了可执行代码以及代码所需要的资源”[1]。尽管具体的 bundle 类型可能会不同，而且 bundle 中的内容会不同，但是所有的 bundle 都有同样的基本目录结构，而且每一个 bundle 都带有相同的目录。例如，一个 OS X 应用程序的 bundle 内部结构如代码清单 2-1 所示。

代码清单 2-1：应用程序的 bundle 结构

```
Contents/
    CodeResources/
    Info.plist          包的主 manifest 文件
    MacOS/              包中的二进制文件内容
    PkgInfo             包的 8 字节标识符
    Resources/          .nib 文件(用于 GUI)和.lproj 文件
    Version.plist       包版本信息
    CodeSignature/
    CodeResources
```

Cocoa提供了一种简单的编程方法用于访问和加载bundle，通过NSBundle对象和CoreFoundation提供的CFBundle系列API可以访问和加载bundle。

2.6　应用程序和 app

OS X 对应用程序的处理方式是 NeXTSTEP 的另一项遗产。应用程序整洁地包装在 bundle 中。应用程序的 bundle 包含运行这个应用程序所需要的大部分文件：主二进制文件、私有的库、图标、UI 元素以及图形元素。而用户可以对这一切毫不知情，因为 bundle 在 Finder 中只是显示为一个图标。因此，在 Mac OS 上安装应用程序的体验非常简单——只需要将应用程序图标拖到 Applications 目录中即可。要窥探应用程序的内部文件结构，只需要在图标上单击鼠标右键即可。

在 OS X 中，应用程序通常放在/Applications 目录中。每一个应用程序都在自己的名为 AppName.app 的目录中(AppName 表示这个应用程序的名字)。每一个应用程序都严格遵守一个固定的格式，之后会详细讨论这个格式，其中各种资源根据类别分组在不同的子目录中。

在 iOS 中，app 的结构则没有这么整洁——尽管应用仍然包含在自己的目录中，但是没有那么严格地遵循 bundle 格式。事实上，app 的目录结构可以很混乱，app 的所有文件都可以丢在根目录中，不过有时候国际化(internationalization，i18n)所要求的文件会放在子目录中(例如 xxx.lproj 目录，其中 xxx 表示语言，或者表示 ISO 语言代码)。

此外，iOS会区分苹果自己提供的默认应用程序(放在/Applications目录下，或者在较老越狱版本的 iOS 中的/var/stash/Applications 目录下)和通过 App Store 购买的应用程序(放在/var/mobile/Applications目录下)。通过App Store购买的app安装在一个表示 128 位GUID的目录下，这个GUID可以按照字节数分解为更易于管理的结构：4-2-2-2-6(例如A8CB4133-414E-4AF6-06DA-210490939163，其中每一个十六进制的数码表示 4 个位)。

在以 GUID 命名的目录中，可以找到和往常一样的 .app 目录结构，除此之外还有其他一些额外的目录。

表 2-1 列出了这个特殊的目录结构，这个目录结构是一定要满足的，因为 iOS App 在运行时会被 chroot(2)到自己的应用目录——即名字为 GUID 的目录——因此不能逃脱这个目录访问文件系统的其他部分。这样可以保证第三方的应用会受到限制，甚至不能看到并排安装的其他应用——既能保证用户的隐私，也能保证苹果对操作系统的绝对掌控(当然，越狱能改变这一切)。在这种体系下，应用程序将自己 GUID 名字的目录当做根目录，因此当需要临时目录时，/tmp 指向的是 GUID/tmp。

表 2-1　iOS app 的默认目录结构

iOS app 目录下的元素	用　途
Documents	应用程序保存的数据文件(例如游戏的最高分、文档和记录等)
iTunesArtwork	app 的高分辨率图标，通常是一个 JPG 图像文件
iTunesMetaData.plist	app 的属性列表文件，是二进制格式的 plist 文件(之后会详细讨论 plist)
Library/	杂项 app 文件。这个目录下包括 Caches、Cookies 和 Preferences 等，有时候还有 WebKit(用于内建了浏览功能的 app)
tmp	保存临时文件的目录

从 App Store(或其他地方)下载应用时，应用以.ipa 文件的形式打包——这种文件实际上就是一个.zip 文件(可以通过 unzip(1)打开)，这个文件在 Payload/目录下压缩了应用程序的目录内容。如果您手边没有一台越狱的设备，可以 unzip -t 一个.ipa 文件，了解应用程序的结构。.ipa 文件保存在本地计算机的~/Music/iTunes/iTunes Media/Mobile Applications/目录下。

2.6.1　Info.plist

Info.plist文件位于应用程序(以及其他大部分bundle)的Contents/子目录下，这个文件保存了bundle的元数据信息。这个文件是必备的，操作系统通过这个文件判定依赖关系和其他属性。

属性列表(或 plist)的格式在其自己的手册页 plist(5)中已经有很好的文档了。有 3 种保存属性列表的格式：

- XML：这种格式的列表是人类可读的格式，可以很容易地通过文件头部的 XML 签名和文档类型定义(DTD)识别出来。属性列表中的所有元素都包含在一个<plist>元素内，这个元素定义了一个数组或一个字典(<dict>，即键/值的关联数组)。这是 OS X 上属性列表的常用格式。
- 二进制格式：也称为 BPlists，可以通过文件头部的魔数 bplist 识别。这种文件是编译后的 plist 文件，因此人类可读性很差，但是对操作系统却是高度优化的，因为使用这种格式的 plist不需要复杂的 XML 解析和处理。此外，BPlists 的串行化也很简单直接，因为数据可以直接通过 memcpy 写入，而不需要转换至 ASCII。BPlists 是 OS X 10.2 引入的，在 iOS 上比在 OS X 上更为常用。
- JSON：使用 JavaScript Object Notation，键/值以容易阅读的格式保存，而且也很容易解析。这个格式的使用不如 XML 格式和二进制格式常见。

这三种格式都是原生支持的。事实上，Objective-C 的运行时让开发者可以完全忽略具体的格式。在 Cocoa 中，可以通过内建的字典或数组对象直接实例化一个 plist，而不需要指定文件的格式：

```
NSDictionary *dictionary = [NSDictionary dictionaryWithContentsOfURL:plistURL];
NSArray *array = [NSArray arrayWithContentsOfURL:plistURL];
```

自然地，人类会更喜欢 XML 格式。OS X和 iOS 都带有一个控制台程序 plutil(1)，这个程序可以转换不同的表示方式。输出清单 2-3 展示了 plutil(1)进行转换的使用实例：

输出清单 2-3：将一个 app 的 Info.plist 转换为更适合人类可读的格式并显示出来

```
morpheus@ergo (~) $ cd ~/Music/iTunes/iTunes\ Media/Mobile\ Applications/

# 注意，.ipa 文件只是一个 zip 文件

morpheus@ergo(Mob..) $ file someApp.ipa
someApp.ipa: Zip archive data, at least v1.0 to extract

# 通过 unzip -j 取消文件的子目录结构，将所有文件解压至同一个目录，不带有目录结构

morpheus@ergo (Mob..) $ unzip -j someApp.ipa Payload/someApp.app/Info.plist
Archive: someApp.ipa
 inflating: Info.plist

# 得到的文件是一个二进制的 plist：

morpheus@ergo (Mob..) $ file Info.plist
Payload/someApp.app/Info.plist: Apple binary property list

# 这个文件可以通过 plutil 转换

morpheus@ergo (Mob..) $ plutil -convert xml1 - -o - < Info.plist > converted.Info.plist

# 然后将转换后的文件显示出来：

morpheus@ergo (Mob..) $ more converted.Info.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd"> <plist version="1.0">
<dict>
     <key>BuildMachineOSBuild</key>
     <string>10K549</string>
     <key>CFBundleDevelopmentRegion</key>
     <string>English</string>
     <key>CFBundleDisplayName</key>
... (为简洁起见将输出截断)...
```

一个标准的 Info.plist 文件包含以下条目：

- CFBundleDevelopmentRegion：如果找不到用户指定的语言，则表示默认的语言。
- CFBundleDisplayName：显示给用户的 bundle 名称。
- CFBundleDocumentTypes：这个 bundle 关联的文档类型。这是一个字典，字典中的值指定了这个 bundle 能够处理的文件扩展名。这个字典还指定了对于关联文档显示的图标。
- CFBundleExecutable：这个 bundle 中实际的可执行文件(二进制文件或库文件)。可执行文件位于 Contents/MacOS 目录。
- CFBundleIconFile：在 Finder 视图中显示的图标文件。
- CFBundleIdentifier：逆 DNS 表示法的标识符。
- CFBundleName：bundle 的名称(限制在 16 个字符之内)。
- CFBundlePackageType：表示一个 4 字母的代码，例如 APPL = Application，FRMW = Framework，BNDL = Bundle。

- CFBundleSignature：bundle 的 4 字母短名。
- CFBundleURLTypes：这个 bundle 关联的 URL。这是一个字典，值指定了这个 bundle 处理的 URL scheme 以及处理方式。

上述列表中所有的键都是以 CF 为前缀的，这些键是由 Core Foundation 框架定义并处理的。Cocoa 应用程序还包含 NS 开头的键，定义了应用程序是否允许脚本操作、Java 需求(如果有的话)，以及和系统偏好(System Preference)设置应用程序面板的整合能力。大部分 NS 开头的键都只能用于 OS X 而不能用于 iOS。

2.6.2 Resources 目录

Resources 目录包含应用程序要求使用的所有文件。这是使用 bundle 格式的一大好处。和其他操作系统不一样，其他操作系统要求资源文件编译进可执行文件，而 bundle 允许资源文件依然保持独立。这样不仅能使可执行文件小得多，而且还允许选择性地升级或增加新的资源，而不需要重新编译。

资源文件的具体内容和应用程序的相关性很高，而且几乎可以是任何类型的文件。不过往往会发现一些反复出现的文件类型。下面讨论这些文件类型。

2.6.3 NIB 文件

.nib 文件是二进制的 plist 文件，其中保存了应用程序中 GUI 组件的位置信息和设置信息。这些文件是通过 Xcode 的 Interface Builder 创建的。Interface Builder 编辑.xib 文件的文本版本，然后再将这些文件打包成二进制格式(打包之后就不可编辑了)。.nib 后缀最早来源于 NEXT Interface Builder，这是 Xcode Interface Builder 的前身。这个文件是一个属性列表，在 OS X 和 iOS 上都保存为二进制的形式。

plutil(1)命令可以部分地将.nib 反编译回 XML 表示形式，不过相比起最初的.xib 文件(以下代码中展示)，反编译的结果会丢失信息。这种设计无疑是有意的，因为.nib 文件本来就不应该被编辑。如果可以随意编辑的话，那么应用程序的用户界面就可以在外部随意修改了。

.XIB 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<archive type="com.apple.InterfaceBuilder3.CocoaTouch.XIB" version="7.10">
  <data>
     <int key="IBDocument.SystemTarget">1056</int>
     <string key="IBDocument.SystemVersion">10J869</string>
     <string key="IBDocument.InterfaceBuilderVersion">1306</string>
     <string key="IBDocument.AppKitVersion">1038.35</string>
     <string key="IBDocument.HIToolboxVersion">461.00</string>
     <object class="NSMutableDictionary" key=
     "IBDocument.PluginVersions">
     ...
        <string key="NS.key.0">com.apple.InterfaceBuilder
         .IBCocoaTouchPlugin</string>
        <string key="NS.object.0">301</string>
     </object>
     <object class="NSArray" key="IBDocument
         .IntegratedClassDependencies">
```

```
        <bool key="EncodedWithXMLCoder">YES</bool>
        <string>IBUIButton</string>
        <string>IBUIImageView</string>
        <string>IBUIView</string>
        <string>IBUILabel</string>
        <string>IBProxyObject</string>
    </object>
```

.NIB 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
                    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>$archiver</key>
  <string>NSKeyedArchiver</string>
  <key>$objects</key>
  <array>
     <string>$null</string>
     <dict>
        <key>$class</key>
        <dict>
           <key>CF$UID</key>
           <integer>135</integer>
        </dict>
        <key>NS.objects</key>
        <array>
           <dict>
              <key>CF$UID</key>
              <integer>2</integer>
           </dict>
```

2.6.4　通过.lproj 文件实现国际化

bundle 在设计上就支持国际化。每一种语言都对应一个子目录。语言目录的后缀为.lproj。有一些语言使用其英文名(例如 English、Dutch 和 French 等), 另一些语言使用国家和语言代码(例如 zh_CN 表示大陆中文，zh_TW 表示台湾繁体字)。在表示语言的目录中包含针对指定语言本地化的字符串文件、.nib 文件和多媒体文件。

2.6.5　图标文件(.icns)

一个应用程序通常包含一个或多个表示视觉形象的图标。应用程序图标会用在 Finder、Dock 和应用程序相关的系统消息中(例如 Force Quit)。

图标文件通常都封装在一个文件 appname.icns 中，带有多种分辨率，从 32×32 一直到 512×512。

2.6.6　CodeResources

应用程序中包含的最后一个重要文件是CodeResources，这实际上是一个指向_CodeSignature/CodeResources 的符号链接。这个文件是一个属性列表，包含bundle 中所有其他文件的列表。这个属性列表只有一项files，这是一个字典，其中键是文件名，值通常是Base64 格式的散列值。如果键表示的文件是可选的，那么值本身也是一个字典，这个字典有一个hash 键和一个optional 键(这个键的值

当然是布尔值true)。

CodeResources 文件可以用于判断一个应用程序是否完好无损，能够防止不小心修改或损坏资源文件。

1. 应用程序的默认设置

和其他常见的操作系统不同，OS X和 iOS 都没有在系统中为应用程序设置维护一个注册表。因此，应用程序必须通过其他机制存储用户首选项和各种默认设置。

苹果提供的这个机制称为 defaults，同样，这也是 NeXTSTEP 的遗产。背后的思想很简单：每一个应用程序都会得到一个属于自己的名称空间，应用程序可以在这个名称空间中根据需要随意添加、修改和删除设置。这个名称空间称为应用程序的域。此外，还有一个所有应用程序共用的全局域(NSGlobalDomain)。

应用程序的默认设置通常保存在属性列表中。苹果建议使用逆 DNS 命名法的约定来保存 plist，而且通常使用二进制格式。这些配置文件保存在用户自己的目录~/Library/Preferences 下。此外，应用程序还可以在/Library/Preferences 目录下保存整个系统范围的(即所有用户都可以使用的)首选项信息。NSGlobalDomain 保存在一个隐藏文件.GlobalPreferences.plist中，这个文件也保存在上述的两个目录中。

系统管理员或超级用户可以通过 defaults(1)命令操纵默认设置信息——这种方式通常比直接编辑 plist文件要方便。这条命令还接受一个-host开关选项，允许对同一个应用程序在不同的主机上设置不同的默认设置。

要注意的是，defaults 机制只负责处理保存和访问设置信息的存储管理。应用程序使用这种机制的方式完全取决于应用程序本身。有一些应用程序(例如 VMWare Fusion)甚至没有遵循 plist 的要求和命名约定。

应用程序很少有自包含的。任何开发者都明白一个道理，应用程序不能重新发明轮子，因此必须依赖于操作系统提供的功能和 API。在 UNIX 中，这种机制称为共享库。苹果在这个基础上建立了“框架”这个独特的概念。

2. 加载默认应用程序

和大部分 GUI 操作系统一样，OS X维护了文件类型和注册了相应文件类型的应用程序之间的关联。通过这种机制，当用户双击一个文件时，操作系统可以启动(或者用苹果的说法是“加载(launch)”)默认应用程序；或者右键单击文件选择 Open With 选项打开一个包含所有注册了这个文件类型的应用程序的子目录。这种机制在终端中也有用，在终端中通过 open(1)命令可以启动关联了文件类型的默认应用程序。

Windows 用户可能对注册表很熟悉，这种文件类型关联就是通过注册表实现的(具体说是 HKEY_CLASSES_ROOT 子键)。OS X通过一个名为 LaunchServices 的框架实现了这个功能。这个框架(注意这个框架和 launchd(1)没有任何关系，后者是 OS X 的引导进程)是 Core Services 框架(本章后面会对此详述)的一部分。

LaunchServices 框架包含一个名为 lsregister 的二进制程序，这个程序可以导出(也可以重置)LaunchServices 的数据库，如代码清单 2-2 所示。

代码清单 2-2：通过 lsregister 查看文件类型注册信息

```
morpheus@Ergo (~)$ cd /System/Library/Frameworks/CoreServices.Framework
morpheus@Ergo (../Core..work)$ cd Frameworks/LaunchServices.framework/Support
morpheus@Ergo (../Support)$ ./lsregister -dump
Checking data integrity......done.
Status: Database is seeded.
Status: Preferences are loaded.
--------------------------------------------------------------------------
... // 为简洁起见，这里省略了一些行
bundle id:      1760
  path:         /System/Library/CoreServices/Archive Utility.app
  name:         Archive Utility
  category:
  identifier:   com.apple.archiveutility (0x8000bd0c)
  version:      58
  mod date:     5/5/2011 2:16:50
  reg date:     5/19/2011 10:04:01
  type code:    'APPL'
  creator code: '????'
  sys version: 0
  flags:        apple-internal display-name relative-icon-path wildcard
  item flags:   container package application extension-hidden
                                                    native-app i386
x86_64
  icon:         Contents/Resources/bah.icns
  executable:   Contents/MacOS/Archive Utility
  inode:        37623
  exec inode:   37629
  container id: 32
  library:
  library items:
   ------------------------------------------------------------
   claim id:      8484
      name:
      rank:       Default
      roles:      Viewer
      flags:      apple-internal wildcard
      icon:
      bindings:  '****', 'fold'
   ------------------------------------------------------------
   claim id:      8512
      name:       PAX archive
      rank:       Default
      roles:      Viewer
      flags:      apple-default apple-internal relative-icon-path
      icon:       Contents/Resources/bah-pax.icns
      bindings:  public.cpio-archive, .pax
   ------------------------------------------------------------
   claim id:      8848
      name:       bzip2 compressed archive
      rank:       Default
      roles:      Viewer
      flags:      apple-default apple-internal relative-icon-path
      icon:       Contents/Resources/bah-bzip2.icns
```

```
        bindings:  .bzip2
    ...
    // 为简洁起见省略了更多的行
```

如果 Open With 子菜单中的项目太多(通常是因为安装了很多应用程序)，那么解决这个问题的一个常用方法就是通过以下命令重建数据库：lsregister -kill -r -domain local -domain system -domain user。

2.7　框架

在 OS X的世界中，另一个重要的组件是框架(framework)。框架就是 bundle，包含一个或多个共享库以及相关的支持文件。

框架很像库(事实上二进制格式都是一样的)，但是框架是苹果系统特有的，所以不可移植。框架也不算是 Darwin 的一部分，因为 Darwin 的组件都是开源的，而大部分框架都是闭源的。这是因为框架主要负责提供独特的外观和体验，并且提供了苹果操作系统特有的其他高级特性，苹果显然不愿意这些特性被移植到其他平台。苹果的系统中仍然提供了“传统”的库(而且事实上框架都是基于这些库实现的)。框架提供了完整的运行时接口，而且能够用于隐藏底层的系统和库 API(特别是通过 Objective-C)。

2.7.1　框架 bundle 格式

和应用程序(以及 OS X 上大部分其他文件)一样，框架实际上是 bundle。因此，框架具有固定的目录结构：

`CodeResources/`	指向 `Code Signature/CodeResources plist` 文件的符号链接
`Headers/`	指向这个框架提供的`.h` 文件所在目录的符号链接
`Resources/`	指向这个框架所需要的`.nib` 文件(用于 `GUI`)、`.lproj` 文件和其他文件所在目录的符号链接
`Versions/`	在这个子目录下实现版本控制
`A/`	字母名称的目录表示这个框架的版本
`Current/`	指向这个框架首选版本的符号链接
`framework-name`	指向这个框架首选版本的二进制文件的符号链接

可以看出，框架bundle和应用程序bundle的格式稍有不同。主要的区别在于内建的版本化机制：框架可以包含多个版本的代码，不同版本并排放在不同的子目录中，例如Versions/A、Versions/B，以此类推。首选版本可以通过创建名为Current的符号链接轻松切换。框架文件本身都连接到选中的版本文件。这种方法效仿了UN*X符号链接库的模型，但是把同样的机制扩展到了头文件。此外，尽管大部分框架只有一个版本(通常使用A，但有时候会使用B或C)，这种架构还允许前向和后向兼容。

OS X 和 iOS 的 GCC 支持-framework 选项，通过这个选项可以包含任何框架，既包括苹果自带的也包括第三方框架。使用这个选项可以提示编译器在哪里找到头文件(类似-I 选项)以及提示编译器在哪里能找到库文件(类似-l 选项，但不完全一样)。

1. 查找框架

框架保存在文件系统中的多个位置：

? /System/Library/Frameworks 包含苹果提供的框架，iOS 和 OS X 中都是如此。

? /Network/Library/Frameworks 可能会用于(罕见)网络上安装的公共框架。

? /Library/Frameworks 保存第三方框架(不出预料，在 iOS 上这个目录为空)。

? ~/Library/Frameworks 保存用户提供的框架(如果有的话)。

此外，应用程序可能包含自己的框架。苹果的 GarageBand、iDVD 和 iPhoto 都是典型例子，这些软件都在 Contents/Frameworks 目录下保存了应用程序专用的框架。

框架搜索路径可以进一步通过用户定义的环境变量修改，这些变量的顺序如下：

? DYLD_FRAMEWORK_PATH

? DYLD_LIBRARY_PATH

? DYLD_FALLBACK_FRAMEWORK_PATH

? DYLD_FALLBACK_LIBRARY_PATH

苹果提供了大量框架——在 Snow Leopard 上有 90 多个框架，在 Lion 中则超过了 100 个框架。而私有框架的数目则更为庞大，这些私有框架是公共框架内部使用的，或者是苹果的应用程序直接使用的。私有框架放在/System/Library/PrivateFrameworks 目录下。私有框架的结构和公共框架的结构除了头文件之外完全一样，头文件是有意不包含在内的。

2. 顶层框架

Carbon 和 Cocoa 是 OS X 中两个最为重要的框架：

Carbon

Carbon 是 OS 9 遗留编程接口的名称。Carbon 已经被宣布废弃，不过还有很多应用程序依然依赖 Carbon，甚至包含一些苹果自己的应用程序。尽管 Carbon 中有很多接口是为了 OS 9 的兼容性而特别设置的，但是 Carbon 中也添加了很多新的特性，而且还看不出消失的迹象。

Cocoa

Cocoa 是首选的应用程序编程环境。Cocoa 是 NeXTSTEP 环境在现代的化身，从 Cocoa 中的很多基类前缀就可以看出——NS 表示 NeXTSTEP/Sun。Cocoa 编程的首选语言是 Objective-C，不过 Cocoa 也可以通过 Java 和 AppleScript 访问。

如果查看一下 Cocoa 和 Carbon 框架，可以发现这两个框架都很小，几乎是微小——在 Snow Leopard 上差不多就 40KB 大小。对于这样一种大型的 API 来说，这个框架小得有点不正常。考虑到 Cocoa 是一个支持 3 种架构(包括废弃的 PPC)的“胖”二进制，这就更令人吃惊了。背后的秘密在于这些框架是基于其他框架构建的，实际上只是其他框架的包装而已——将其依赖的框架的符号重新导出为自己的符号。

Cocoa 框架只包含了其他 3 个框架：AppKit、CoreData 和 Foundation，从 Headers/cocoa.h 头文件可以直接看出来。使用苹果的术语，封装了其他框架的框架通常称为保护伞(umbrella)框架。这个术语既可以用于仅仅是#import了其他框架的框架，例如Cocoa这样的框架；也可以用于嵌套了其他框架的框架，如Application Services框架和Core Services框架这样的框架。从代码可以看出来：

```
/*
  Cocoa.h
  Cocoa Framework
  Copyright (c) 2000-2004, Apple Computer, Inc.
  All rights reserved.
```

为了方便构建，所有 Cocoa 应用程序都应该包含这个文件。应该优先选择包含这个文件而不是包含独立的头文件，因为这个头文件会使用一个预编译好的版本。

如果是不带用户界面或不依赖于AppKit的工具应用，应该只包含<Foundation/Foundation.h>头文件。

```
*/

#import <Foundation/Foundation.h>
#import <AppKit/AppKit.h>
#import <CoreData/CoreData.h>
```

2.7.2　OS X 和 iOS 公共框架列表

表 2-2 列出了 OS X和 iOS 中的框架，并且列出了开始支持这些框架的操作系统版本。版本号信息取自于苹果的官方文档[3,4]，在官方文档中也能找到类似的表格(而且还有可能更新)。这些框架具有很多重合度，有很多框架从 OS X 移植到了 iOS，还有一些框架从 iOS 移植回 OS X(例如 CoreMedia)。在 Mountain Lion 中从 iOS 移植的框架更多，例如 Game Center 和 Twitter 都是从 iOS 移植的框架。此外，还有一些 OS X 的框架在 iOS 中以私有框架的形式存在。

表 2-2　Mac OS X 和 iOS 中的公共框架

框　架	OS X 版本	iOS 版本	作　用
AGL	10.0	--	OpenGL 的 Carbon 接口
Accounts	10.8	5.0	用户账号数据库——用于支持单点登录
Accelerate	10.3	4.0	加速的向量操作
AddressBook	10.2	2.0	地址簿相关的功能
AddressBookUI	--	2.0	显示联络人信息的 UI(iOS)
AppKit	10.0	--	Cocoa 的主要库之一(由 Cocoa.Framework 依赖)，本身是其他库的保护伞。还包含 XPC(在 iOS 中为私有框架)
AppKitScripting	10.0	--	被 Appkit 替代
AppleScriptKit	10.0	--	AppleScript 的插件
AppleScriptObjC	10.0	--	AppleScript 的基于 Objective-C 的插件
AppleShareClientCore	10.0	--	AFP 客户端实现
AppleTalk	10.0	--	AFP 协议的核心实现

(续表)

框　架	OS X 版本	iOS 版本	作　用
ApplicationServices	10.0	--	CoreGraphics、CoreText、ColorSync 和其他一些框架的保护伞框架(仅头文件)，还包含 SpeechSynthesis(这是作者的最爱)
AudioToolBox	10.0	2.0	音频录制/处理以及其他相关功能
AssetsLibrary	--	4.0	照片和视频
AudioUnit	10.0	2.0	音频单元(插件)和编解码器
AudioVideoBridging	10.8	--	AirPlay
AVFoundation	10.7	2.2	对于音频/视频媒体的 Objective-C 支持。最近才移植到 Lion
Automator	10.4	--	Automator 插件支持
CalendarStore	10.5	--	iCal 支持
Carbon	10.0	--	OS 9 的遗产 API Carbon 的保护伞框架(仅头文件)
Cocoa	10.0	--	Cocoa API 的保护伞框架(仅头文件)，包含 AppKit、CoreData 和 Foundation
Collaboration	10.5	--	CBIdentity*系列 API
CoreAudio	10.0	2.0	音频抽象层
CoreAudioKit	10.4	--	音频的 Objective-C 接口
CoreBlueTooth	--	5.0	蓝牙相关的 API
CoreData	10.4	3.0	数据模型，包含 NSEntityMappings 等
CoreFoundation	10.0	2.0	从字面上可以理解，向其他所有框架提供基础数据结构(各种 CF 开头的类)的核心框架
CoreLocation	10.6	2.0	GPS 相关的服务
CoreMedia	10.7	4.0	音频和视频相关的底层例程
CoreMediaIO	10.7	--	CoreMedia 的抽象层
CoreMIDI	10.0	--	MIDI 客户端接口
CoreMIDIServer	10.0	--	MIDI 驱动程序接口
CoreMotion	--	4.0	加速度计和陀螺仪相关的接口
CoreServices	10.0	--	AppleEvents、Bonjour、Sockets、Spotlight、FSEvents 以及很多其他服务框架的保护伞框架(嵌套的子框架)
CoreTelephony	--	4.0	电话相关的数据
CoreText	10.5	3.2	文本和字体相关的框架。在 OS X 中这是 ApplicationServices 框架中的子框架
CoreVideo	10.5	4.0	被其他库使用的视频格式支持
CoreWifi	10.8	P	在 iOS 中，这个框架称为“MobileWiFi”，而且是私有的框架

(续表)

框　架	OS X 版本	iOS 版本	作　用
CoreWLAN	10.6	--	无线 LAN(WiFi)相关的框架
DVComponentGlue	10.0	--	数字视频录制器/相机
DVDPlayback	10.3	--	DVD 播放
DirectoryService	10.0	--	LDAP 访问
DiscRecording	10.2	--	光盘烧录相关的库
DiscRecordingUI	10.2	--	光盘烧录相关的库和用户界面
DiskArbitration	10.4	--	系统卷管理器 DiskArbitrationD 的接口
DrawSprocket	10.0	--	Sprocket 组件
EventKit	10.8	4.0	Calendar 支持
EventKitUI	--	4.0	Calendar 用户界面
ExceptionHandling	10.0	--	Cocoa 异常处理
ExternalAccessory	--	3.0	处理插在 iPad/iPod/iPhone 上的硬件配件的库
FWAUserLib	10.2	--	火线音频库
ForceFeedback	10.2	--	处理力反馈功能的设备(游戏杆、游戏板等)
Foundation	10.0	2.0	底层数据结构支持
GameKit	10.8	3.0	游戏的点对点连接
GLKit	10.8	5.0	OpenGLES 辅助库
GLUT	10.0	--	OpenGL Utility 框架
GSS	10.7	5.0	Generic Security Services API(RFC2078)，带有一些苹果私有的扩展
iAd	--	4.0	苹果的移动广告分发系统
ICADevices	10.3	--	扫描仪/照相机相关的库(类似于 TWAIN)
IMCore	10.6	--	由 InstantMessaging 内部使用
ImageCaptureCore	10.6	P	替代了之前的 ImageCapture
ImageIO	--	4.0	图像格式的读写
IMServicePlugin	10.7	--	iChat 服务提供者
InputMethodKit	10.5	--	替换的输入方法
InstallerPlugins	10.4	--	系统安装器插件
InstantMessage	10.4	M	即时消息和 iChat
IOBluetooth	10.2	--	OS X 的蓝牙支持
IOBluetoothUI	10.2	--	OS X 的蓝牙支持
IOKit	10.0	2.0	设备驱动程序的用户态组件

(续表)

框　架	OS X 版本	iOS 版本	作　用
IOSurface	10.6	P	在应用程序之间共享图像
JavaEmbedding	10.0～10.7	--	在 Carbon 中嵌入 Java。在 Lion 和更新的版本中不再支持
JavaFrameEmbedding	10.5	--	在 Cocoa 中嵌入 Java
JavaScriptCore	10.5	5.0	Safari 和其他 WebKit 程序使用的 JavaScript 解释器
JavaVM	10.0	--	Java 运行时库的苹果移植
Kerberos	10.0	--	提供 Kerberos 支持(Active Directory 整合和一些 UNIX 域要求这项支持)
Kernel	10.0	--	内核扩展所需要的框架
LDAP	10.0	P	最早的 LDAP 支持，被 OpenDirectory 所替代
LatentSemanticMapping	10.5	--	隐含语义映射
MapKit	--	4.0	嵌入地图和地理信息编码数据
MediaPlayer	--	2.0	iPod 播放器界面和电影播放界面
MediaToolbox	10.8	P	
Message	10.0	P	电子邮件消息的支持
MessageUI	--	3.0	发送消息和 Mail.app(ComposeView 和其他相关的视图)的 UI 资源
MobileCoreServices	--	3.0	精简版的 CoreServices
Newsstandkit	--	5.0	支持 iOS 5.0 引入的 “Newsstand”
NetFS	10.6	--	网络文件系统(AFP、NFS)
OSAKit	10.4	--	Cocoa 中整合 OSA 脚本的支持
OpenAL	10.4	2.0	跨平台音频库
OpenCL	10.6	P	GPU/并行编程框架
OpenDirectory	10.6	--	Open Directory(LDAP)的 Objective-C 绑定
OpenGL	10.0	--	OpenGL——3D 图形支持。在受支持的芯片组上链接至 OpenCL
OpenGLES	--	2.0	嵌入式 OpenGL——在 iOS 上替换 OpenGL
PCSC	10.0	--	智能卡的支持
PreferencePanes	10.0	--	System Preference 面板支持。System Preference 中的面板实际上都是保存在/System/Library/PreferencePanes 目录下的 bundle
PubSub	10.5	--	RSS/Atom 支持
Python	10.3	--	Python 脚本语言
QTKit	10.4	--	QuickTime 支持
Quartz	10.4	--	保护伞框架，包括 PDF 支持、ImageKit、QuartzComposer、QuartzFilter 和 QuickLookUI。负责系统中大部分 2D 图形的绘制

(续表)

框　架	OS X 版本	iOS 版本	作　用
QuartzCore	10.4	2.0	Quartz 和 Core 相关的框架之间的接口
QuickLook	10.5	4.0	文件预览和缩略图生成
QuickTime	10.0	--	Quicktime 嵌入
Ruby	10.5	--	流行的脚本语言 Ruby
RubyCocoa	10.5	--	Ruby 语言的 Cocoa 绑定
SceneKit	10.8	--	3D 渲染。在 Lion 中是私有框架，但是在 Mountain Lion 中成为了公共框架
ScreenSaver	10.0	--	屏幕保护相关的 API
Scripting	10.0	--	最早的脚本框架，现已废弃
ScriptingBridge	10.5	--	Objective-C 语言的脚本适配器
Security	10.0	3.0	证书、密钥和安全随机数等的支持
SecurityFoundation	10.0	--	SF*系列认证
SecurityInterface	10.3	--	证书、认证和钥匙链相关的用户界面的 SF*系列头文件
ServerNotification	10.6	--	Notficiation 支持
ServiceManagement	10.6	--	launchd 的接口
StoreKit	10.7	3.0	应用内购的支持
SyncServices	10.4	--	和.Mac 同步日历
System	10.0	2.0	其他框架内部使用的框架
SystemConfiguration	10.0、10.3	2.0	包含 SCNetwork 和 SCDynamicStore
TWAIN	10.2	--	扫描仪支持
Twitter	10.8	5.0	Twitter 支持(在 iOS 5 中)
Tcl	10.3	--	TCL 解释器
Tk	10.4	--	Tk Toolkits
UIKit	--	2.0	Cocoa Touch——AppKit 在 iOS 上的替代
VideoDecodeAcceleration	10.6.3	--	通过 GPU 进行 H.264 加速(TN2267)
VideoToolkit	10.8	P	替代 QuickTime 图像压缩管理器，提供视频格式支持
WebKit	10.2	P	HTML 渲染(Safari 的核心)
XgridFoundation	10.4～10.7	--	Xgrid 集群(在 Mountain Lion 中被移除)
vecLib	10.0	--	向量计算(Accelerate 的子框架)

练习：演示框架的强大

OS X 的框架真是技术的奇迹。不论从什么角度看，框架的精巧设计和可重用性都是超群的。图形相关的框架中有很多出色的例子，但是 SpeechSynthesis.Framework 框架的例子是真正有用而且同样令人印象深刻的。

利用这个框架，可以快速简单地将 Text-to-Speech 特性嵌入到程序中，充分利用苹果已经实现(大部分由苹果实现，而且由苹果完善)的复杂逻辑。/System/Library/Speech 目录下面包含 Synthesizers(合成器)目录(Mountain Lion 中包含两个：MacinTalk 和 MultiLingual)，其中包含的都是 Mach-O 格式的二进制 bundle。这些 bundle 可以像库一样加载至几乎任何进程中。此外，还有一些预编程好的声音(在 Voices/子目录中)和 Recognizers(识别器，用于 Speech-to-Text)。声音将音高和其他语音参数编码在一种自主产权的二进制格式中。在苹果开发者文档“The Speech Synthesis API”中对此有详细描述。苹果还提供了一个很酷的工具用于自定义语音：“Repeat After Me”(这是 Xcode 的一部分，位于/Developer/Applications/Utilities/ Speech/Repeat After Me)。

然而一般的开发者却不需要关心这些。通过SpeechSynthesis.Framework框架(当然也有其他方法)可以访问语音合成器，这个框架本身位于ApplicationServices 框架(对于Carbon)或AppKit(对于Cocoa)。通过使用这个框架，一个C语言或Objective-C语言的应用程序可以利用区区数行代码实现Text-to-Speech的功能(使用系统中提供的多种声音中的一种声音)，下面的例子就演示了如何实现这一点。这个例子展示了一个利用OS X提供的Text-to-Speech功能的简单示例(这个例子比较简单粗暴)。

为了避免涉及凌乱的 Objective-C 语法，代码清单 2-3 列出的示例代码使用的是 C 语言，因此使用的是 ApplicationServices 框架而不是 AppKit 框架。

代码清单 2-3：演示如何简单地实现 say(1)实用工具的(部分)功能

```
#include <ApplicationServices/ApplicationServices.h>

// OS X say(1)命令的简单粗暴实现
// 编译时传入-framework ApplicationServices 参数

void main (int argc, char **argv) {

  OSErr rc;
   SpeechChannel channel;
   VoiceSpec vs;
   int voice;
   char *text = "What do you want me to say?";

   if (!argv[1]) { voice = 1; } else { voice = atoi(argv[1]); }

   if (argc == 3) { text = argv[2]; }

   // GetIndVoice 获得某个索引(正数)定义的声音
   rc= GetIndVoice(voice, // SInt16      index,
                  &vs); // VoiceSpec * voice

   // NewSpeechChannel 函数使得选定的声音可用
   rc = NewSpeechChannel(&vs,// VoiceSpec * voice, /* 可以为 NULL */
```

```
                        &channel);


    // 调用 SpeakText 函数说出文本
    rc = SpeakText(channel,     // SpeechChannel chan,
                text,         // const void *  textBuf,
              strlen(text));  //unsigned long  textBytes

    if (rc) { fprintf (stderr,"Unable to speak!\n"); exit(1);}

    // 由于语音播放是异步的，所以要等待播放结束。
    // Objective-C 可以使用更为优雅的回调函数

    while (SpeechBusy()) sleep(1);
    exit(0);
}
```

这个语音框架也可以通过其他方法访问。有一些其他语言也有这个框架的绑定，例如 Python 和 Ruby；对于非程序员，还可以使用命令行工具 say(1)(也就是上面这个例子模仿的程序)，此外还可以使用苹果优秀的脚本语言 Applescript(通过 osascript(1)访问)。您可以自己尝试一下，玩一玩命令行程序(这也有可能会闹笑话，或者说是有创意的用法，参见图 2-3 中的漫画)。

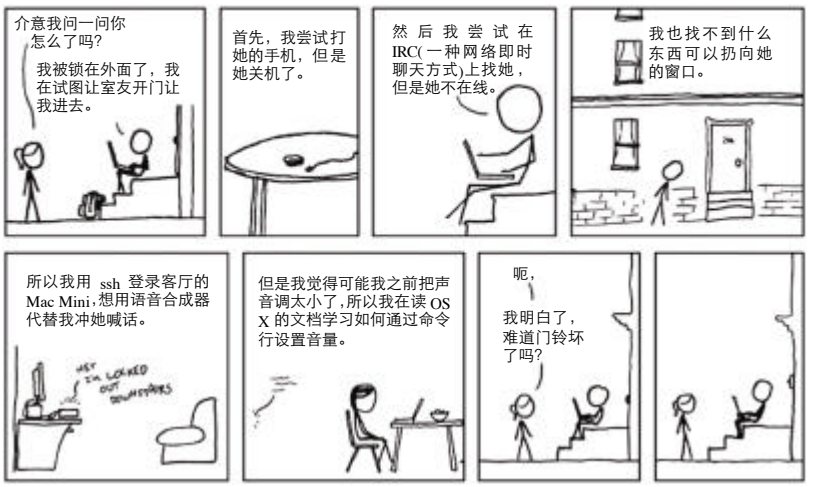

图 2-3　OS X 语音特性的其他有创意的用法。摘自有意思的网站 http://XKCD.com/530

(漫画中的人物所需要的命令是 osascript -e“set Volume 10”)

根据前文所述，应用程序可以完全依赖于框架，事实上很多 OS X和 iOS 应用就是这么干的。然而框架本身也依赖于操作系统提供的库，下面就要讨论库。

2.8　库

框架可以说就是一种特殊形式的库。实际上，框架中的二进制文件就是库，通过 file(1)命令可以验证这一点。苹果仍然强调这两个名词之间的区别，因为与库相比，框架更倾向于是 OS X(和 iOS)特有的，而库则是所有 UNIX 系统共有的。

OS X 和 iOS 将“传统”的库保存在/usr/lib 目录中(不存在/lib 目录)。库文件使用.dylib 后缀，而不是其他 UNIX 上常用的 ELF 格式的.so(共享目标)文件。除了后缀名的不同之外(当然二进制格式也不同，.dylib 不兼容.so 文件)，两者在概念上都是一样的。在 OS X 中可以找到其他 UNIX 中也有的库，只不过都是.dylib 格式的。

如果浏览一下iOS的文件系统——既可以在一个运行的越狱系统上浏览，也可以通过iOS软件更新镜像(.ipsw文件)来查看，您会发现很多库文件(框架文件也是)都不见了！这是因为采用了库缓存的优化机制(也许是一种混淆机制)，下一章会详细讨论库缓存机制。因此，在iPhone SDK中更容易看到这些库文件，在/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS#.#.sdk/目录中可以看到这些文件。

核心 C 库 libc 被吸收到苹果自己的 libSystem.B.dylib 中了。这个库还包含了原本由数学库(libm)和 PThreads(libpthread)以及其他一些库提供的功能，这些库都是指向 libSystem 的符号链接，从输出清单 2-4 中可以看出这一点：

输出清单 2-4：/usr/lib 中的库实际上都是由 libSystem.dylib 实现的

```
morpheus@Minion (/)$ ls -l /usr/lib | grep ^l | grep libSystem.dylib
lrwxr-xr-x1rootwheel 17Sep 26 02:08 libSystem.dylib -> libSystem.B.dylib
lrwxr-xr-x1 root wheel 15Sep 26 02:08 libc.dylib -> libSystem.dylib
lrwxr-xr-x1 root wheel 15Sep 26 02:08 libdbm.dylib -> libSystem.dylib
lrwxr-xr-x1 root wheel 15Sep 26 02:08 libdl.dylib -> libSystem.dylib
lrwxr-xr-x1 root wheel 15Sep 26 02:08 libinfo.dylib -> libSystem.dylib
lrwxr-xr-x1 root wheel 15Sep 26 02:08 libm.dylib -> libSystem.dylib
lrwxr-xr-x1 root wheel 15Sep 26 02:08 libpoll.dylib -> libSystem.dylib
lrwxr-xr-x1 root wheel 15Sep 26 02:08 libproc.dylib -> libSystem.dylib
lrwxr-xr-x1rootwheel 15Sep 26 02:08 libpthread.dylib -> libSystem.dylib
lrwxr-xr-x1root wheel 15Sep 26 02:08 librpcsvc.dylib -> libSystem.dylib
```

然而 libSystem 本身又依赖于几个内部使用的库，这些内部的库都存放在/usr/lib/system 目录中。libSystem 导入这些库，然后将这些库的公共符号重新导出，就好像这些符号都是 libSystem 库自己的一样。在 Snow Leopard 中，这种库的数目很少。在 Lion 和 iOS 5中，这种库的数目则比较大。如输出清单 2-5 所示，其中演示了通过 Xcode 的 otool(1)工具显示库的依赖关系。注意，由于 libSystem 库是被缓存的(因此在 iOS 的文件系统中也看不到)，所以最好在 iPhone SDK 中库的目录中运行这条命令。

输出清单 2-5：通过 otool(1)查看 iOS 5 中 libSystem 库的依赖关系

```
morpheus@ergo (.../Developer/SDKs/iPhoneOS5.0.sdk/usr/lib)$ otool -L libSystem.B.dylib
libSystem.B.dylib (architecture armv7):
  /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 161.0.0)
  /usr/lib/system/libcache.dylib (compatibility version 1.0.0, current version 49.0.0)
  /usr/lib/system/libcommonCrypto.dylib (compatibility version 1.0.0, current version 40142.0.0)
  /usr/lib/system/libcompiler_rt.dylib (compatibility version 1.0.0, current version 16.0.0)
  /usr/lib/system/libcopyfile.dylib (compatibility version 1.0.0, current version 87.0.0)
  /usr/lib/system/libdispatch.dylib (compatibility version 1.0.0, current version 192.1.0)
  /usr/lib/system/libdnsinfo.dylib (compatibility version 1.0.0, current version 423.0.0)
```

```
/usr/lib/system/libdyld.dylib (compatibility version 1.0.0, current version 199.3.0)
/usr/lib/system/libkeymgr.dylib (compatibility version 1.0.0, current version 25.0.0)
/usr/lib/system/liblaunch.dylib (compatibility version 1.0.0, current version 406.4.0)
/usr/lib/system/libmacho.dylib (compatibility version 1.0.0, current version 806.2.0)
/usr/lib/system/libnotify.dylib (compatibility version 1.0.0, current version 87.0.0)
/usr/lib/system/libremovefile.dylib (compatibility version 1.0.0, current version 22.0.0)
/usr/lib/system/libsystem_blocks.dylib (compatibility version 1.0.0, current version 54.0.0)
/usr/lib/system/libsystem_c.dylib (compatibility version 1.0.0, current version 770.4.0)
/usr/lib/system/libsystem_dnssd.dylib (compatibility version 1.0.0, current version 1.0.0)
/usr/lib/system/libsystem_info.dylib (compatibility version 1.0.0, current version 1.0.0)
/usr/lib/system/libsystem_kernel.dylib (compatibility version 1.0.0, current version 1878.4.20)
/usr/lib/system/libsystem_network.dylib (compatibility version 1.0.0, current version 1.0.0)
/usr/lib/system/libsystem_sandbox.dylib (compatibility version 1.0.0, current version 1.0.0)
/usr/lib/system/libunwind.dylib (compatibility version 1.0.0, current version 34.0.0)
/usr/lib/system/libxpc.dylib (compatibility version 1.0.0, current version 89.5.0)
```

OS X中的加载器 dyld(1)也称为 Mach-O 加载器。下一章会详细讨论这个加载器，从用户模式的角度讨论进程加载和执行的内部原理。

OS X还自带了很多其他开源的库，这些库都包含在Darwin中(iOS中同样也包含了这些库)。这些库包括OpenSSL、OpenSSH、libZ、libXSLT以及其他很多库，既可以在苹果自己的开源网站上下载这些库，也可以从SourceForge和其他软件仓库网站上下载，还可以下载已编译的文件。有趣的是，这不是第一次(也不是最后一次)iOS越狱的突破口就来源于这些开源的库(也许是libTiff、FreeType或其他的库)。

2.9　其他应用程序类型

目前讨论的这些应用程序和 App bundle 并不是唯一支持的应用类型。OS X(iOS 也能在一定程度上)还支持其他一些类型的应用程序。

2.9.1　Java(仅限于 OS X)

OS X包含一个和Java 1.6完整兼容的Java虚拟机。和其他操作系统一样，Java应用程序是以.class文件的形式提供的。.class 文件格式并不是 OS X 原生支持的格式，因此还是需要 java(1)命令行应用程序来执行 Java 程序，就和其他操作系统一样。不过这个 JVM 实现是由苹果维护的。java 命令行工具(java、javac 和其他相关工具)都是公共框架 JavaVM.framework 中的一部分。还有其他两个框架 JavaEmbedding.framework 和 JavaFrameEmbedding.framework 用于将 Java 程序连接和嵌入 Objective-C 程序。

Java 虚拟机进程的实际加载工作是由私有框架 JavaLaunching.framework 和 JavaApplicationLauncher.framework 完成的。目前 iOS 还不支持 Java。

2.9.2　Widget

Dashboard widget(小物件，简单称为 Widget)是 HTML/JavaScript 编写的迷你页面，这些页面在 dashboard 中展现出来。由于这些迷你应用很容易编程实现(基本上就和网页编程差不多)，所以越来越流行。

Widget 保存在/Library/Widgets 目录中，是带有.wdgt 扩展名的 bundle。这种 bundle 的结构比较

松散，包含以下元素：

- 一个 HTML 文件(widgetname.html)，这个文件表示了 Widget 的 UI。Widget 的 UI 就像普通 HTML 一样用标记的方式描述，通常带有两个 元素——一个表示 widget 的正面，一个表示 widget 的背面。
- 一个 JavaScript(JS)文件(widgetname.js)，这是 Widget 的“引擎”，提供了交互能力。
- 一个层叠样式表(CSS)文件(widgetname.css)，提供了样式、字体等的定义。
- 语言目录，和其他 bundle 一样，包含本地化的字符串。
- 任何所需要的图像和其他文件，通常保存在 Images/子目录中。
- 任何所需要的二进制插件，这些插件实现 widget 所需要但是无法完全通过 JavaScript 实现的功能。这种文件是可选的(例如 Calculator.wdgt 就没有这种文件)，如果有的话，则包含其他 bundle，其中包含一个二进制的插件文件(文件类型是 Mach-O 二进制子类型“bundle”)。这些文件可以被加载在 Dashboard 中，提供需要打破浏览器环境限制的复杂功能，例如本地文件的访问。

2.9.3　BSD/Mach 原生程序

尽管 iOS 和 OS X首选的开发语言是 Objective-C，但是依然可以使用 C/C++语言编写原生应用程序，还可以选择不使用框架，而是直接调用系统库和 BSD/Mach 的底层接口。这样可以使得基于 UNIX 的代码库较方便地移植，例如 PHP、Apache、SSH 以及很多其他开源的产品。此外，还有一些创新产品(例如“MacPorts”和“fink”)更进一步，模仿 Linux 的 RPM/APT/DEB 等(包管理器)模型将这些源代码编译打包，提供快速二进制安装的体验。

由于 OS X 兼容 POSIX，所以应用程序的移植很方便，只要是依赖标准的系统调用以及之前讨论的库的应用程序就可以方便地移植。在 iOS 中也是如此，开发者们几乎把所有可移植的软件都移植了，通过 Cydia 可以访问这些程序。然而，还有另外一组 API 子集——Mach Trap(Mach 陷阱)，这套 API 仍然是 OS X(和 GNUStep)特有的，并且和 BSD 共存。接下来会从用户的角度解释这些 API。

2.10　系统调用

作为所有操作系统都遵守的准则，用户程序不允许直接访问系统资源。用户程序可以操作通用寄存器，执行一些简单的计算，但是如果需要执行任何重要的功能，例如打开文件或套接字、甚至是发送一条简单的消息，都必须使用系统调用。系统调用指的是由内核导出的预定义函数的入口点，在用户态需要链接/usr/lib/libSystem.B.dylib 才能访问这些系统调用。OS X 的系统调用的特殊之处在于实际上导出了两套独特的调用接口——一个是 Mach 调用，另一个是 POSIX 调用。

2.10.1　POSIX

OS X 从 Leopard(10.5)开始是一个经过认证的 UNIX 实现。这意味着 OS X 能够完全兼容 Portable Operating System Interface(可移植操作系统接口，也就是常说的 POSIX)。POSIX 是一套标准的 API，具体定义了以下内容：

- 系统调用原型：所有的 POSIX 系统调用(不论底层实现如何)都有相同的原型，也就是说具有相同的参数和返回值。例如，所有的 POSIX 系统上 open(2)的原型如下所示：

```
int open(const char *path, int oflag, ...);
```

path 表示要打开的文件的文件名，oflag 表示<fcntl.h>文件中定义的标志的按位 OR 组合(例如 O_RDONLY、O_RDWR 和 O_EXCL 等标志)。

这样可以保证只要是 POSIX 兼容的代码就可以在任何 POSIX 兼容的操作系统上在源代码的层次移植。在 OS X上的代码可以移植到 Linux、FreeBSD，甚至可以移植到 Solaris，只要代码仅仅依赖于 POSIX 调用和 C/C++标准库。

? 系统调用编号：除了固定的原型之外，POSIX 还完整定义了系统调用的编号。这在一定程度上允许了二进制层次的可移植性——也就是说，POSIX 兼容的二进制代码可以在底层架构相同(例如，Solaris 可以运行 Linux 的二进制程序，这两个系统都兼容 ELF 格式的二进制)的 POSIX 系统之间移植。不过 OS X 不支持这种移植，因为 OS X 的目标文件格式 Mach-O 和 ELF 不兼容。此外，OS X 的系统调用编号和标准编号不同。

POSIX 兼容性是由 XNU 中 BSD 层提供的。这个系统调用的原型在<unistd.h>头文件中。我们在第 8 章将讨论这些系统调用的实现。

2.10.2　Mach 系统调用

之前提到过，OS X是在 Mach 内核的基础上构建的，而 Mach 是 NeXTSTEP 的遗产。BSD 层是对 Mach 内核的包装，但是 Mach 系统调用仍然可以在用户态访问。事实上，如果没有 Mach 的系统调用，像 top 这样的常见命令都无法工作。

在 32 位系统上，Mach 系统调用的编号都为负数。这个聪明的技巧使得 POSIX 和 Mach 系统调用可以共存。由于 POSIX 只定义了非负的系统调用，所以负数空间还没有使用，因此 Mach 就使用了负数。

在 64 位系统上，Mach 系统调用是正数，但是以 0x2000000 开头，而 POSIX 调用编号以 0x1000000 开头，所以两者可以明显区分开。

http://newosxbook.com 网站上的在线附录列出了各种 POSIX 调用和 Mach 系统调用。第 8 章将详细讨论用户态到内核态的转换，第 9 章和第 13 章将从内核的角度讨论系统调用和陷阱。

实验：显示 Mach 和 BSD 系统调用

系统调用并不是直接调用的，而是通过 libSystem.B.dylib 中的浅层封装进行的。通过 OS X 上默认的 Mach-O 处理工具和反汇编器 otool(1)可以对任何二进制文件进行反汇编(利用-tV 参数选项)，因此我们可以查看 libSystem 内部的秘密。据此，我们可以了解 OS X 中系统调用接口是如何利用 Mach 和 BSD 调用工作的。

在 32 位系统上，Mach 系统调用类似于：

```
Morpheus@Ergo (/) % otool –arch i386 –tV /usr/lib/libSystem.B.dylib | more
/usr/lib/libSystem.B.dylib:
(__TEXT,__text) section
_mach_reply_port:
000010c0     movl    $0xfffffffe6,%eax  ; 将系统调用编号加载至 EAX 寄存器
000010c5      calll   __sysenter_trap
000010ca       ret
000010cb      nop                       ; 填充至 32 位边界
_thread_self_trap:
```

```
000010cc      movl    $0xfffffffe5,%eax  ; 将系统调用编号加载至 EAX 寄存器
000010d1       calll   __sysenter_trap
000010d6        ret
000010d7       nop                       ; 填充至 32 位边界
__sysenter_trap:
000013d8        popl %edx
000013d9        movl %esp,%ecx
000013db       sysenter                   ; 执行 sysenter 指令
000013dd        nopl (%eax)
```

系统调用编号被加载至 EAX 寄存器。注意这个编号是 0xFFFFxxxx。由于这个编号是按照有符号整数处理的，所以 Mach API 调用编号实际上是负数。下面看一下 BSD 系统调用：

```
Ergo (/) % otool –arch i386 –tV /usr/lib/libSystem.B.dylib –p _chown | more
/usr/lib/libSystem.B.dylib:
(__TEXT,__text) section
_chown:
0005d350       movl    $0x000c0010,%eax   ; 加载系统调用编号
0005d355       calll   0x00000dd8         ; 跳转至__sysenter_trap
0005d35a       jae     0x0005d36a         ; 如果返回码 >= 0，则跳到 ret 指令
0005d35c        calll   0x0005d361
0005d361        popl    %edx
0005d362        movl    0x0014c587(%edx),%edx
0005d368        jmp     *%edx
0005d36a        ret
0005d87c       calll   0x0005d881         ; 如果出错...
0005d881        popl
0005d882        movl
0005d888        jmp
0005d88a        ret
```

同样的例子在 64 位硬件架构上可以看出略有不同的实现：

```
Ergo (/) % otool –arch x86_64 –tV /usr/lib/libSystem.B.dylib | more
/usr/lib/libSystem.B.dylib:
(__TEXT,__text) section
_mach_reply_port:
00000000000012a0        movq  %rcx,%r10
00000000000012a3       movl  $0x0100001a,%eax   ; 加载系统调用编号 0x1a，标志 0x01
00000000000012a8      syscall                   ; 直接调用 syscall
00000000000012aa        ret
00000000000012ab        nop
```

对于 POSIX(BSD)系统调用：

```
Ergo (/) % otool –arch x86_64 –tV /usr/lib/libSystem.B.dylib –p _chown | more
/usr/lib/libSystem.B.dylib:
(__TEXT,__text) section
___chown:
0000000000042f20      movl  $0x02000010,%eax ; 加载系统调用编号 0x10，标志 0x02
0000000000042f25      movq  %rcx,%r10
0000000000042f28      syscall              ; 直接调用 syscall
0000000000042f2a      jae   0x00042f31     ; 如果返回码 >= 0，跳转到 ret 指令
0000000000042f2c      jmp   cerror         ; 否则跳转到 cerror
                                           ; (即返回-1，设置 errno)
0000000000042f31      ret
```

如果接着在 ARM 架构上(即 iOS 平台上)运行这个例子，可以看到类似的流程：将系统调用编号加载至 r12 寄存器(内部程序寄存器)，然后通过 svc 执行系统调用(有时候反汇编器会将这条指令解码为 swi，即软件中断)。在下面的例子中(用的是 GDB，不过也可以使用 otool(1))，反汇编了 BSD 的 chown(2)系统调用和 Mach 的 mach_reply_port 系统调用。注意后者是通过 mvn(Move Negative，加载负数)指令加载的。根据 ARM 的约定，返回代码保存在 r0 中。

```
(gdb) disass chown
0x30d2ad54 <chown>:      mov  r12, #16             ; 0x10
0x30d2ad58 <chown+4>:    svc  0x00000080
0x32f9c758 <chown+8>:    bcc  0x32f9c770 <chown+32>; 返回值 >= 0 时跳转到退出
0x32f9c75c <chown+12>:  ldr  r12, [pc, #4]        ; 0x32f9c768 <chown+24>
0x32f9c760 <chown+16>:  ldr  r12, [pc, r12]
0x32f9c764 <chown+20>:  b    0x32f9c76c <chown+28>
0x32f9c768 <chown+24>: bleq 0x321e2a50          ; 跳转到设置 errno 的代码
0x32f9c76c <chown+28>:  bx   r12
0x32f9c770 <chown+32>:  bx   lr
(gdb) disass mach_reply_port
Dump of assembler code for function mach_reply_port:
0x32f99bbc <mach_reply_port+0>:  mvn   r12, #25    ; 0x19
0x32f99bc0 <mach_reply_port+4>:  svc   0x00000080
0x32f99bc4 <mach_reply_port+8>:  bx    lr
```

2.11　XNU 概述

内核 XNU 是 Darwin 的核心，也是整个 OS X 的核心。XNU(据说是一个无限递归的缩写：XNU's Not UNIX)本身由以下几个组件构成：

- Mach 微内核
- BSD 层
- libKern
- I/O Kit

此外，内核是模块化的，允许根据需要动态加载插件形式的内核扩展(Kernel Extension，KExt)。

本书的重头戏(即整个第 II 部分)是深入讲解 XNU。不过在这里先简单概述一下 XNU。

2.11.1　Mach

XNU 的核心、“原子核”，如果您愿意的话，可以认为就是 Mach。Mach 最初是一个在卡内基梅隆大学(CMU)开发的研究型操作系统，致力于研制一个用于操作系统的轻量级且高效的平台。这个项目的研究成果就是 Mach 微内核(microkernel)，这个微内核仅能处理操作系统最基本的职责：

- 进程和线程抽象
- 虚拟内存管理
- 任务调度
- 进程间通信和消息传递机制

Mach 本身的 API 非常有限，而且本身也不是设计为一个具有全套功能的操作系统。苹果不鼓励使用 Mach 的 API，不过可以看出来，这些 API 非常基础，如果没有这些 API 的话，其他工作都

无法实施。任何额外的功能，例如文件和设备访问，都必须在此基础上实现，而这些额外的功能都是 BSD 层实现的。

2.11.2　BSD 层

BSD 层建立在 Mach 之上，也是 XNU 中一个不可分割的部分。这一层是一个很可靠且更现代的 API，提供了之前提到的 POSIX 兼容性。BSD 层提供了更高层次的抽象，其中包括：

- UNIX 进程模型
- POSIX 线程模型(Pthread)及其相关的同步原语
- UNIX 用户和组
- 网络协议栈(BSD Socket API)
- 文件系统访问
- 设备访问(通过/dev 目录访问)

XNU 中的 BSD 实现很大程度上和 FreeBSD 的实现兼容，但是也有一些非常显著的变化。本书在讲解了 Mach 之后，再开始讲解 BSD，关注于 BSD 核心的实现，提供了有关虚拟文件系统交换(VFS)和网络协议栈的细节内容，这些内容都有专门的章节描述。

2.11.3　libkern

大部分内核都是完全使用 C 语言和底层汇编编写的。而 XNU 则有不同。设备驱动程序——称为 I/O Kit 驱动程序，可以用 C++语言编写，下一小节会讨论 I/O Kit。为了支持 C++运行时并提供所需要的基类，XNU 包含 libkern 库，这是一个内建的、自包含的 C++库。尽管没有直接向用户态导出 API，但是 libkern 是一个基础，如果没有这个基础的话，很多高级功能都无法实现。

2.11.4　I/O Kit

苹果对 XNU 最重要的修改是引入了 I/O Kit 设备驱动程序框架。这是一个在内核中的完整的、自包含的执行环境，让开发者可以快速创建优雅稳定的设备驱动程序。能实现这一点的原因是 I/O Kit 形成了一个受限的 C++环境(通过 libkern)，其中带有语言提供的最重要功能——继承和重载。

那么，编写 I/O Kit 驱动程序这件事情就被极大地简化了，只需要找到一个已有的驱动程序作为超类，并且在运行时中继承其所有功能。这样可以避免样板化代码的复制，这种代码复制很可能会引入稳定性的 bug，此外还能使得代码规模非常小——在内存紧张的内核空间中总是好事情。在驱动程序中添加新的方法或重载/隐藏已有的方法可以进行功能的修改。

使用 C++环境的另一个好处是驱动程序可以工作在一个面向对象的环境中。因此 OS X 的驱动程序和其他操作系统上的设备驱动程序大有不同，其他操作系统上的设备驱动程序只能使用 C 语言编写，而且就算是最简单的功能也需要大量的代码才能实现。I/O Kit 在 XNU 中组成了一个几乎自包含的系统，带有一个由很多驱动程序组成的丰富环境。单是 I/O Kit 的内容就可以用一本书来描述(事实上最近就有这么一本书)，不过本书的整个第 18 章都在讲解 I/O Kit 的架构。

2.12　本章小结

这一章介绍了 OS X 和 iOS 的架构。尽管这两个操作系统是为不同的平台而设计的，但是两者

实际上非常类似，而且随着新版本的发布，两个操作系统之间的差异越来越小。

这一章提供了一个详细的概述，但是仍然在比较高的层次，尽可能少地接触到代码示例。下一章开始更加深入地讨论 OS X 特有的 API，其中带有大量可以实验的代码示例。

参考文献

[1] Apple Developer — Bundle Programming Guide

[2] “OS X for UNIX Users”(Lion version): http://images.apple.com/macosx/docs/OSX_for_UNIX_Users_TB_July2011.pdf

[3] Apple Developer—OS X Technology Overview: (details all the frameworks): http://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/OSX_Technology_Overview/SystemFrameworks/SystemFrameworks.html

[4] Details frameworks for iOS: http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSFrameworks/iPhoneOSFrameworks.html