第1章 Cocoa与Mac OS X

自 2001 年发布以来，Mac OS X 的市场份额一直缓慢增长，8 年后几乎占到了桌面市场的 10%。造就这一成功的因素很多：坚实稳固、基于 UNIX 的系统基础架构，简洁的图形用户界面，以及对系统所有外观细节的关注。

还有一个备受第三方 Mac 开发者称赞的特性：Cocoa。这是一组干净、面向对象的 API，从 20 世纪 80 年代至今一直在不断改进。Cocoa 让 OS X 开发变得简单而有趣，那么到底什么是 Cocoa，它又是怎样适合现在的系统呢？

1.1 理解何时该用Cocoa

Cocoa 不是开发 OS X 程序的唯一方式，也不总是最佳选择。本节将着眼于 Cocoa 的替代品及其应该在何时使用。开发架构的选择并不总是非此即彼的。可以在一个程序里略有限制地混合 Cocoa、Carbon 及其他的框架。也可以混合使用不同的编程语言，程序中性能相关的部分用 C 或者 Objective-C，其他的部分用高级脚本语言。

1.1.1 Carbon

在 Mac OS X 之前，有 Classic MacOS（它在大部分时间都叫做 Apple System，只是最后几个版本才叫做 Mac OS）。它诞生于 1984 年，后来又经过了不断改进。该系统是从 Motorola 的 68000 系列架构移植到 Apple、IBM 和 Motorola 联合开发的 PowerPC 架构下的，包含有一个运行老代码用的模拟器。早期的基于 PowerPC 的机器要比基于当时最快的 Motorola 68040 的机器慢，因为很多操作系统代码和大多数程序都运行在模拟器中。

最初的版本中的图形用户界面固化在 ROM 中，叫做 Macintosh Toolbox。后来随着内存日渐丰富起来，它就被要载入到内存的新版本取代了[1]。

随着 Mac OS 的退出已成定局，Toolbox 的末日也随之而来。为了简化迁移过程，Apple 发布了一个整理后的 APICarbon。它运行在 Mac OS 8.1 及以后的版本上。

Carbon API 和老的 Toolbox 有很多重叠的部分。对有些程序，从 Toolbox 切换到 Carbon 只需重新编译即可，大部分程序也只需很小的修改。这个过程叫做碳化（carbonization）。

在 OS X 10.0 中 Carbon 和 Cocoa 都是一等公民。有些核心程序如 Finder 就是用 Carbon 写的。早期版本的 Carbon 有些限制，如无法访问服务（service）。混用 Carbon 和 Cocoa 也曾很困难，因为事件模型不一样。这些限制后来逐步清除掉了。似乎 Carbon 会在 OS X 中存在很久。

可就在 2007 年，改变发生了，Apple 公司宣布 Carbon 不会升级到 64 位。而此前 Apple 已经在 PowerPC G5 中发布了 64 位的芯片。在 PowerPC 架构下，大多数程序升级到 64 位意义都不大。指针变长了（会增加缓存使用），不过其他地方没什么不同。但切换到 Intel 芯片以后就有变化了。x86-64 架构相对 32 位版本的提升很大，包含更快的调用约定、更简单的内存模型、更多的寄存器，以及更好的浮点运算支持。

Carbon 虽然还在，但已被降级为 Mac OS X 的二等公民。OS X 10.6 的一大改进就是用基于 Cocoa 重写的 Finder 取代了原有的基于 Carbon 的 Finder。OS 10.6 搭载的 Xcode 3.2 也已经不再包含 Carbon 应用程序、框架等的项目模板，也没有多少新特性暴露给 Carbon 访问。这有效地把 Cocoa 变成了 OS X 唯一的“原生”应用程序框架。现在只在移植基于 Mac OS 9 或之前版本的程序时，才推荐使用 Carbon，不过已经没有多少值得移植而尚未移植的软件了。

Apple 的文档有时会把 OS X 上所有的 C API 都叫做“Carbon”，而不仅仅是

1 最初的Mac只有128 KB RAM，所以RAM是非常珍稀的资源。

Carbon 框架。Cocoa 这个术语也有类似的不一致的用法，有时候仅指 Foundation 和 Application Kit 框架，而有时候则是指 OS X 上任何用 Objective-C 写的库。OS X 的某些部分现在仍只能用 C API 来访问。它们没有消失；只有 Carbon 框架正在消失。

1.1.2 Java

Java 是一种常常和跨平台联系在一起的语言。Apple 收购 NeXT 的时候，获得了很多 Java 的经验。那时候，NeXT 卖 WebObjects web 应用程序开发平台赚了很多钱。该产品本来是用 Objective-C 写的，第 5 版的时候移植到了 Java 上。其新版本现在是免费的，且包含在 Xcode 开发工具中[1]。

要在 OS X 上使用 Java，程序员有两个选择。可以安装纯 Java，完全不使用任何 Apple 专有的代码。显然这是从其他平台移植代码的一种方案。

历史上，OS X 曾是运行 Java 代码最好的平台。Apple 是最早在 VM 中实现多程序之间共享类文件的，他们花了很大的力气，在 OS X 的第一个发行版本中实现了对 Swing 主题功能的支持，为 Java 程序提供了和 Mac 程序一样的外观。

除了 Java GUI 库，Apple 还提供了对许多 Cocoa 对象的桥接，使得 Java 可以调用 Cocoa。这就是所谓的 Mocha（Java+Cocoa）。这有很多优点。相对而言，熟悉 Java 的程序员要更多，而且 Java 有一些 Objective-C 所没有的功能，例如垃圾回收。反之亦然。Java 在 1.3 以前不支持代理。[2]

不幸的是，几年后 Apple 对 Java 失去了兴趣。10.3 版本后，Apple 声明废弃了 Mocha，Cocoa 新引入的 API 不再用桥接暴露出来。作出这个决定很大程度上是因为开发者缺乏兴趣。Cocoa 是一个巨大的 API，而 Objective-C 是一个小语言。如果你想学习 Cocoa，那么学习 Objective-C 不会带来太多额外负担，而且可以得到 Cocoa 的原生语言本身的好处。

Java 对反射的支持比 Objective-C 限制多。Apple 花了很大力气去优化 Objective-C 运行时库，所以可以在程序中用内省驱动的机制来保持模型和视图对象的同步。在 Java 下做同样的事情会很难。

1 现在的Xcode不带WebObjects了，需要单独下载。——译者注

2 Apple内部曾经有一个计划是开发Objective-C的类Java语法版本，但最终放弃了。——译者注

1.1.3 Cocoa

OS X 刚被引入的时候，它是两个操作系统 Apple Mac OS 9 和 NeXT OPENSTEP 4 的结合。Carbon 来自 Mac OS，而 Cocoa 来自 NeXT。后来被叫做 Cocoa 的 API，原来叫做 NeXTSTEP API。

在 NeXTSTEP 中，大多数底层程序都用 C 语言来写，而图形界面套件用 Objective-C 来写。在买得起它的人中，这个平台非常流行。也许用这个框架开发的最有名的一个软件就是 WorldWideWeb，物理学家 Tim Berners-Lee 在 CERN 工作时写的一个简单程序。它是最早的 web 浏览器。

NeXT 的机器大部分都卖给了各个大学，有一些到了斯坦福，他们用这些机器开发了 HippoDraw，这是一个统计数据分析工具。后来其作者想把它移植到其他平台，不过他们没有选择重写，而是写了一个平台兼容库 libobjcX，在 X Window System 上实现了 NeXT 的图形界面。

这个系统的另外一个爱好者是 MIT 的黑客 Richard Stallman，他从 NeXT 系统得到了灵感，GNU 操作系统的很多部分的构思就来自于这个系统。

NeXTSTEP 框架非常专注于面向对象编程，它的图形界面构建工具是最早的快速开发工具（RAD）之一。1993 年前后，Sun 公司对这些工具表示了兴趣，并在随后与 NeXT 携手开发了一个改进的版本。

这次合作的产物是 OpenStep 规范。OpenStep 规范定义了两个库：Foundation Kit 和 Application Kit。第一个库包含了对任何程序都可能有用的核心对象，而第二个库则关注于带图形用户界面的程序。Display PostScript 规范也制定了出来，只是没几个程序员直接使用它。

OpenStep 规范在随后一个版本的 NextSTEP 上得到了实现，这个版本被改名叫 OPENSTEP（注意不同的大小写形式），以体现跟这个规范的关系。NeXT 开始出售基于 Windows NT 的实现，Sun 在 Solaris 里也包含了一个实现。与此同时 Sun 把兴趣转移到了 Java 上，大幅抛弃了 Solaris 里的 OpenStep。

这个规范发布后，libobjcX 被采纳为一个 GNU 项目，并改名叫 GNUstep。很多年前，GNUstep 就实现了和 OpenStep 规范的完全兼容。它现在谋求跟踪 Apple 在 Cocoa 中引入的更新。

NeXT 时代很少有人关注 GNUstep。大多数使用 OPENSTEP 的用户都很喜爱它，但是很少有人能买得起其标价 $499 的 i486 版本，或者是标价 $4999 的 NeXT 卖的最便宜的工作站。随着 Mac OS X 的发布，更多的开发者通过 Cocoa 接触到了 OpenStep，GNUstep 于是得到了一定的关注。

Cocoa 中仍可看到 NeXT 的残留。用 OpenStep 编写的程序大多数在 OS X 上直接就可以编译，虽然有几个不那么通用的类被去掉了。所有的 Cocoa 类的名字仍旧以“NS”开头，意指 NeXT-Sun（参与制定 OpenStep 规范的两家公司）。

非常偶然的情况下，你也许会交叉引用到更早版本就存在的类。它们使用 NX 前缀，意指 NeXT。Apple 保留了 Cocoa 核心类的 NS 前缀，其他代码则使用框架名相关的前缀。

1.1.4 UNIX API

从 10.5 版本起，OS X 成为了真正的 UNIX。也就是说它获得了 The Open Group 的认证，符合 Single UNIX Specification（SUS）。严格说来，这仅包含 Intel Mac 之上的 OS X。这个认证只适用于在特定硬件上的操作系统的特定版本，Apple 没有出钱给 PowerPC 硬件上的 OS X 进行认证。

这一认证结束了 The Open Group 和 Apple 之间的商标官司，此前 Apple 没有认证就把 OS X 叫做了 UNIX。Single UNIX Specification（SUS）是 Portable Operating System Interface（POSIX）标准的超集。POSIX 定义了一组任何操作系统都可以实现的接口，而 SUS 提供了一些扩展定义，只不过它们仅适用于类 UNIX 系统。

因为 OS X 已经被认证为 UNIX，所以它可以运行任何基于公开 UNIX 规范的程序。这提供了丰富的应用程序级的接口，虽然它们不包含任何图形支持。

Apple 提供一个可选的 X server X11.app，它实现了 X11 协议，这个协议是大多数其他 UNIX 系统跟显示设备沟通的接口。X11 的设计是网络透明的，即可以在一个系统上运行程序，在另一个上显示图形界面。因而可以把 OS X 当做远程 UNIX 程序的客户端，就像它们在本地运行一样使用。

如果你写了一个基于 X11 的程序，它可能不会被大多数 Mac 用户接受。X11 程序绝对是 Mac 桌面上的二等公民。它们和其他程序使用不同的菜单栏，使用 Dock 的方式也与众不同，而且只得到标准 OS X 剪贴板功能的有限支持。

X11 的好处是被很多其他系统支持，所以可以用做把 UNIX 软件移植到 OS X 的起点。可以先对软件略作修改、编译，在 OS X 的 X11 下运行，看它能否正常工作。然后可以给它写一个新的图形界面层。这对在设计上模型和视图就分得很清楚的软件非常容易，但是对其他的则很困难。

如果编写服务器程序，那么你可能希望使用标准 UNIX 接口。也可以选择 OS X 上的 Foundation 库，这一框架的开源实现可以帮助把程序移植到任何地方。

1.1.5 其他选择

虽然这些是图形界面开发的主要工具，但也还是有一些其他的选择。跨平台工具箱，例如 wxWidgets 和 Qt，相对来说在 *NIX 和 Windows 上比较流行。但是由于 OS X 的人机界面指南（HIG）太过强大，因而它们在这个平台上没那么普遍。这些，还有开发工具让你可以很简单的跟随他们，大多数 Cocoa 和 Carbon 程序看起来，特别是他们的行为是符合用户预期的。这方面有无数的例子，如对话框里按钮的布置、菜单的顺序，甚至是用来在文本框中跳转的快捷键。

使用跨平台工具箱开发，往往是只得本地程序之形，而不得其神。很多时候，这是最差的组合，因为那些视觉线索会让用户误以为这个程序的行为会符合他们的预期。最糟糕的一个例子就是 Trolltech 公司的 Qt 框架，它为程序提供了一个 Aqua 风格的外观，但在很多版本中，他们把在文本框内跳前跳后的快捷键弄错了。这对用户干扰很大，会让他们厌恶你的程序，即使他们不知道是为什么。

所以，在 OS X 上使用跨平台 GUI Toolkit 开发的成功软件要比在其他系统上少得多。但这并不是说跨平台开发就不好。很多程序的实现都可能在 GUI 和其他代码间加上一个干净的抽象层，这样在不同的平台就可以接入不同的用户界面。VLC 多媒体播放器就是一个很好的例子，它在 Windows、*NIX 和 Mac OS X 上都很流行，有很多用户界面，包括一个 Cocoa 的，甚至还包括一个用于远程控制的小 web 服务器。

如果可以把程序用这种方式拆分，那么就可能做出一个可移植的程序，在所有平台的外观和感觉都很优雅。如果你在别的平台使用 GNUstep，那么可以在核心逻辑中使用 Cocoa Foundation 库，用 GNUstep 或者为你想支持的每个平台编写本地用户界面。或者，也可以用你更熟悉的框架开发你的核心逻辑，然后在上面编写 Cocoa 的用户界面。

当然不是所有程序都需要图形界面。OS X 包含了很丰富的开发工具来构建命令行系统。为类 UNIX 系统设计的纯 C 语言、Python、Ruby 及 shell 脚本，经常无须任何修改就可以在 OS X 上正常运行。

1.2 理解Cocoa在Mac OS X中的角色

Cocoa 是 Mac OS X 开发中很大很重要的一部分，但不是唯一的部分。理解它和整个宏伟蓝图如何配合是很重要的。图 1.1 给出了 OS X 主要组件如何结合的简单概览。作为一个开发者，你可以选择忽略 Cocoa 之下各层的细节，但是理解它们有助于写出更好的程序。

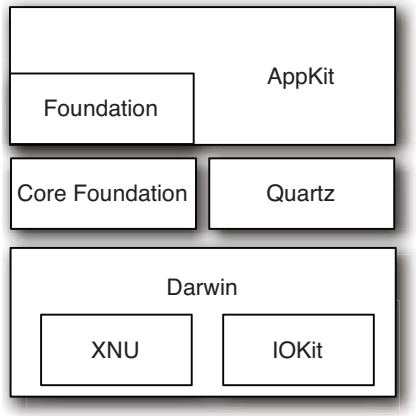

图 1.1：OS X 主要组件概览

1.2.1 Cocoa

Cocoa 是开发者使用的最高层 API。它提供了两个不同层次的框架：Foundation 和 AppKit。Foundation 提供了标准数据类型，如字符串及集合容器类。它还包含了很多底层功能，如文件和网络的接口。

Foundation 框架功能最强的一个部分是分布式对象（Distributed Objects）框架。这就给 Objective-C 提供了代理机制，以及给 Foundation 对象提供了序列化的能力，这样不同进程或者不同机器上的不同对象就可以像它们都在本地一样交互。

Foundation 最重要的部分就是内存管理的代码。OpenStep 之前，NeXT 代码使用 C 语言风格的分配和释放机制，对象都有宿主，要显式地由宿主释放。

跟踪所有权关系是 bug 的主要来源。在 OpenStep 下，对象维持一个内部的引用计数，让这个问题变得简单了一点。引用计数并不完美，而 Cocoa 最新的版本引入了一个跟踪式的垃圾回收器。这不能完全解决内存管理问题——甚至一些 Apple 自己的程序都还有内存泄漏——但是它可以让你轻松很多。

在 Foundation 之上是 Application Kit，或简称 AppKit。它包含了图形界面相关的代码和一大堆其他的东西。它和 Foundation 框架结合紧密，有很多例子都是类声明在 Foundation 中，在 AppKit 中被扩展或包装。

其中一个例子是 runloop。NSRunLoop 定义在 Foundation 中，用于处理简单的循环，重复调用对象的方法。在 AppKit 中，它被扩展加入了程序对象，来处理每次循环中的事件分发。

Cocoa 和 iPhone

2007 年，Apple 发布了 iPhone 和 iPod Touch 这两种运行裁剪版 OS X 的手持设备。它们使用 Cocoa 的 Foundation 部分，但不使用 AppKit，而是使用了 UIKit，一个为小型设备设计的框架。UIKit 基于很多和 AppKit 类似的概念，很多 AppKit 的类在 UIKit 里都有对应的类。

跟桌面版 OSX 不同的是，iPhone OS X 无须支持历史遗留的程序。这是一个全新的平台，Apple 利用这个机会取消了很多遗留功能。可以把 UIKit 想象成简洁版的 AppKit。AppKit 的新扩展和 UIKit 重合的部分比 AppKit 中老的部分还要多。因为 UIKit 和 AppKit 有很大的重合子集，熟悉其中一个的开发者可以轻松地适应另外一个。在两者间移植代码也很容易。

OS X 的 iPhone 版本没有包含很多更高级的桌面框架，还有一些新的功能如垃圾回收，因为它在 iPhone 这种运算能力较弱的设备上无法正常运行。可以期待桌面版和 iPhone 版的 OS X 将来会融合。虽然他们不太可能使用完全一样的 API，但是会有一个更大的可以在两边都正常工作的通用子集。

另一个例子是 NSAttributedString 类。它在 Foundation 中定义为一个简单的类，保存字符串内的一个范围和属性之间的映射关系到一个字典中。在 Foundation 中，属性是任意的键—值对。AppKit 扩展了它，定义了特定键的意义，比如键“font”指文字范围的字体。

很多其他的框架现在也包含在了略嫌混乱的 Cocoa 旗下。有些涉及系统整合，如 Address Book 框架。它提供了保存跟人相关的任意信息的支持，允许即时消息、E-mail 及其他的通信工具共享同一个存储。还有，如 Core Data，让程序的数据管理更简单。

术语 Cocoa 有两个意思。Cocoa 框架本身仅仅是 Foundation 和 AppKit 框架的简单包装。不过通常人们提到“Cocoa”指的是“OS X 系统包含的 Objective-C 框架”。这就已经很多了，但还有更多的第三方的框架可供开发者使用。

1.2.2 Quartz

NeXTSTEP 使用 Display PostScript（DPS）来显示内容。这使桌面出版业对这个平台很感兴趣，因为输出到屏幕和打印机的东西是一样的。

PostScript 是非常复杂的语言。它是图灵完备的，也就是可以实现任何算法。这对打印很方便，因为可以把复杂的程序输给打印机生成复杂的文档。但对屏幕

绘制就没那么便利。如果你把一个死循环的 PostScript 程序发送给打印机，只需关掉打印机重新提交即可。如果你对显示服务器做同样的事情就麻烦了。好的 Display PostScript 实现可以杀死运行了太久的 widget，但是这个“太久”不好定义。

大多数用户用不到 DPS 的大部分能力。它们使用贝赛尔曲线和简单的绘图命令，但是用 C 语言和 Objective-C 来编写控制结构。这就失去了类 DPS 环境的主要优势。Sun 推出过一款类似的基于 PostScript 的窗口系统叫做 NeWS，在图形 UNIX 的早期，它是 X Windowing System 的竞争对手。NeWS 相对 X 的主要优势是图形对象用 PostScript 编写，可以通过网络发给显示器。当你在 X GUI 上按下一个按钮，它就发送一个“鼠标点击”事件到远程机器，然后远程机器发回一个 X 命令处理图形界面的更新。而对于 NeWS，则是 PostScript UI 处理事件，在本地显示出按下的按钮，然后发送一个“按钮被按下”的事件给远程机器。这样就可以提供延迟更低的远程显示。然而，NeXT 从未推出过基于 DPS 的远程显示，所以没发挥这种能力。

设计 OS X 的时候，Apple 曾经考查过是否可以用 X11 来代替 DPS，因为那时（现在也是）它是类 UNIX 系统的标准。但是他们发现它缺少很多自己所需要的功能，如对字体反锯齿的良好支持、色彩校正，以及混合（compositing）。DPS 也没有全部支持这些，所以 Apple 写了一个新的系统。

既然没人在用 DPS 的流程控制，所以先把它去掉。Adobe 当时已经创建了一个没有流程控制的类 PostScript 语言：Portable Document Format（PDF）。新的窗口系统采用了 PDF 的绘图模型，从一开始就整合了混合支持，有时它被叫做 Display PDF。早期的窗口系统——X11、DPS、NeWS、以及 Windows GDI——都创建于内存非常昂贵的年代。缓存显示器上每一个单独窗口可能很容易就耗费掉数十兆字节的内存。原来的 NeXT 工作站独立的帧缓存就有大约 2MB（1120×832，16 位色），窗口覆盖所需的缓存可以轻松地填满彩色版本包含的 12MB 内存。OS X 推出的时候，目标机器至少有 64MB 内存，16MB 视频内存，以及差不多的显示器尺寸。所以缓存一切就变得很有吸引力了，因为开销不大，而且可以降低重绘和避免画面撕裂而消耗的 CPU 使用。

OS X 最初的版本在有大的图形更新时相对缓慢，因为所有的图像混合是由 CPU 执行的。后面的版本改用 GPU 来进行混合操作，就快很多了。这个更新的版本叫做 Quartz Exteme。

10.4 版时，Apple 试图把更多的任务交给 GPU。略显讽刺的是，在 OS X 上最耗费资源的图形操作是渲染文本。Quartz GL，以前叫 Quartz 2D Extreme，把每个字形渲染成反锯齿的轮廓作为 texture，然后把它们混合在一起输出，以此提高渲染速度。这比把每个字形画成一个贝赛尔曲线要快得多。刚推出的时候 Quartz

2D Extreme 比它的前任可靠性要差一点，所以默认没有打开。最终用户可以使用 Quartz Debug 把它打开，这个工具包含在开发者工具包里。

常有人说 Quartz 使用了 OpenGL。不是这样的。Quartz 可能使用 3D 硬件，但是它确实是用自己的接口。OpenGL 和 Quartz 都是构建在 3D 硬件驱动程序之上的。Quartz 窗口可能包含 OpenGL 内容，允许 OpenGL 在 OS X 上面使用。窗口变换和混合操作都是直接由 Quartz 和窗口服务器实现的。

Quartz 的可编程接口叫做 Core Graphics。它提供了 PDF 模型下的底层绘制功能。可以使用 Core Graphics 来绘制反锯齿贝赛尔曲线、填充形状和混合位图。这些功能也被 AppKit API 包装。iPhone 下虽然没有 AppKit 但也有 Core Graphics 的实现，所以用这些 API 绘图的代码在这两个系统都可以使用。

1.2.3 Core Foundation

Quartz 提供了底层的显示功能，而 Core Foundation 提供了底层的数据操作功能。Cocoa 和 Carbon 都使用 Core Foundation 做很多事情，包括字符串和集合的表示。

在调试器里面查看 Cocoa 对象时，会发现真正的对象有一个 CF 前缀，而不是 NS 前缀（或者是在 NS 前缀后面有一个 CF 前缀）。这是因为 Core Foundation 用 C 实现了简单版本的 Objective-C 对象模型，一些常见对象是这么实现的。这样就可以在 Carbon 中将其简单地用做 C 语言类型，而在 Cocoa 中用做 Objective-C 对象。在创建一个 Objective-C 字符串对象时就会看到，在代码中是 NSString 类的实例，但是在调试器中是 NSCFString。C 程序员可以用宏创建它们，将其看做 CFString 封装类型的实例。

无损耗桥接（toll-free bridge）就是用来描述这种情况的。意思是可以把一个 Cocoa 或者 Core Foundation 对象传递给需要另外一种形式的参数的函数或方法，而不会带来任何的开销。实际上实现的方式是一些深层的“魔术”。Core Foundation 对象的类指针都设置为一个小于 0xFFFF 的值，消息分发函数把这些值当做特殊情况处理。

每个 Objective-C 对象都是一个结构体，第一个元素是指向类的指针。这个指针用于帮助消息发送函数寻找要调用的正确方法。Core Foundation 类型把该指针设置为不合法的值，允许其方法由消息传递函数调用，或者直接被调用。和 Objective-C 方法不同，Core Foundation 方法取对象作为参数，而不是选择器（方法名）。这意味着某些 Objective-C 的高级特性，如消息转发，Core Foundation 类型无法使用。

Carbon 程序是直接使用 Core Foundation 函数的，Cocoa 开发者可以使用等价方法。用户空间中很多程序，如 Launchd，都用到了 Core Foundation。所有可以在属性列表中被序列化的数据类型都由 Core Foundation 实现，允许纯 C 程序使用 Cocoa 程序创建的属性列表。这在 OS X 中比较常见，系统后台程序使用 Core Foundation，但处理其配置信息的图形界面程序使用 Cocoa。

Core Foundation 文档中经常可以看到，术语“类”和 Core Foundation 类型联系在一起。和在 Objective-C 中大体相同。最大的区别在于 Core Foundation 的类是纯粹的抽象概念，在运行期间不存在。Objective-C 类可以通过获取其指针进行查看，而 Core Foundation 类由一个整数标识，其所有行为都是固定编码的。

CFLite

除了 Core Foundation 的完整版本，Apple 还发布过它的一个开源子集，叫做 CFLite。里面没有实现对无损耗桥接的支持，因为这是 Apple Objective-C 运行时的功能；然而，它实现了所有的 C API。也就是说，仅使用 Core Foundation 库的程序可以相对容易地移植到其他平台。不过要注意，GNUstep 没有使用 CFLite，所以它不能帮助移植混合 Core Foundation 和 Cocoa 调用的代码，除非使用这两个库的代码是完全分开的。

1.2.4 Darwin

OS X 的核心是 Darwin 操作系统。原始的 NeXTSTEP 系统基于 CMU 的 Mach 操作系统，与单一 BSD 服务器一起提供了很多通常与内核相关的服务。Mach 是一个微内核，一个尽可能做得越少越好的简单内核。使用微内核有两种方式。一般的设计是把内核分成不同的组件，叫做服务器（server），每个服务器提供一部分操作系统的功能。另外一种方案是运行一个单一的服务器来做所有的事情。OS X 选择了第二个方案。基于效率的原因，这个单一的 BSD 服务器运行在内核地址空间，这避免了很多早期基于 Mach 的系统的性能问题，代价是丧失了 Mach 给传统类 UNIX 系统带来的在稳定性方面的好处。

Apple 收购 NeXT 以后，开始用 NetBSD 的代码来更新核心操作系统，后来改用 FreeBSD 的代码。开始用户空间是 NeXTSTEP、FreeBSD 和 GNU 几大平台的工具大杂烩。很多非常老的 NeXTSTEP 代码现在已经被去掉了。init 系统和一些相关系统如 cron 及 inetd 都已经被整合到了 launchd 中，来自 NeXTSTEP 的 NetInfo 目录服务也被 LDAP 及一个基于属性列表的更简单的本地目录系统取代了。

现在的用户空间非常类似于 FreeBSD 系统，只不过带有些 Apple 的改进。有些改进让 Linux 用户感觉更自在，例如，使用 GNU 的 bash 而不是 C shell 作为默认的 shell。此外，还包含有其他的一些 GNU 工具，替换了 FreeBSD 下对应的工具。

Darwin 和其他开源类 UNIX 系统最大的不同在于它并不使用 GNU 二进制格式工具。大多数系统使用 ELF（Executable and Linking Format）二进制格式，而 Darwin 使用 Mach-O 格式。从能力上看它们几乎是一样的。为了支持这种格式，Darwin 使用了自己的链接器和载入器，以及自己的用来查看二进制格式的工具。

如果熟悉其他的类 UNIX 系统，你可能会使用 ldd 来查看共享库。Darwin 上并不存在该程序；它的功能被归入了 otool 程序，这个程序提供了很多查看 Mach-O 格式二进制对象的选项。

跟二进制格式工具一样，OS X 提供了自己的 C++ 标准库和一个基于 FreeBSD 的 C 库。

从 10.5 版本开始，OS X 被 The Open Group 认证为符合 Single UNIX Specification 2003。于是就有了把它叫做 UNIX 的权利，Apple 在早前出于商业考虑已经这么做了。注意这个认证是区分平台和版本的。严格说来，Intel 芯片上的 OS X 是 UNIX，PowerPC 上的则不是。

Apple 使用多种许可把 Darwin 核心代码开源。来自 Apple 的部分使用 Apple Public Source License（APSL），其他的部分则基于其原作者选择的许可证。

把 Darwin 当做一个独立操作系统来使用是可能的，不过这么做并不流行。直到 10.5 版本（Darwin 9），内核的性能都显著低于其他的类 UNIX 系统，而且还有一些限制。在大多数其他 UNIX 系统中，声音的生成是通过写入 /dev/dsp 或者类似路径实现的。OS X 下，用户空间无法从底层访问声音硬件，一切声音都要通过 Core Audio，但它不是开源的。也就是说 Darwin 很难支持声音，除非你准备直接在驱动程序之上编写程序来替换这一接口。3D 图形方面也有类似的情况。

顺便说下，注意，Darwin 的版本号来自于 NeXTSTEP 和 OPENSTEP。OS X 的第一个版本是 Darwin 5，紧跟 OPENSTEP 4。

1.2.5 XNU

Darwin 的核心是 XNU 内核。这是一个单服务器的 Mach 微内核，有一个主要来自于 FreeBSD 的 BSD 服务器。XNU 内核中 Mach 和 BSD 的区别主要是学术上的，因为它们运行在同一个进程里。XNU 中的 Mach 和 Windows 中的 HAL 角色很类似。它是很小的一层平台相关代码，可以很容易地移植到新的架构上去。

它负责管理内存和进程相关的活动，而所有的高层接口都是由 BSD 层处理的。另外，BSD 层负责为 Mach 的功能提供 POSIX 接口。

Mach 微内核负责处理线程、进程及进程内通信。其他的东西都由 BSD 层实现。有些东西在两者之间共享。虽然可以直接和 Mach 线程交互，但是大多数时候开发者会选择使用 POSIX 函数。BSD 子系统负责为由 Mach 任务实现的进程维护 UNIX 进程结构，允许类 UNIX 的系统调用来操作内核。

OPENSTEP 的设备驱动程序使用 DriverKit 编写，这是一个 Objective-C 的框架。值得注意的是，对在 25MHz 的 Motorola 68040 上运行的平台来说，Objective-C 已经足够快，可以用来写驱动程序。而在 OS X 上，DriverKit 被 IOKit 所取代了。这个框架的设计没有太大的改变，但是用 Embedded C++ 实现的，这是 C++ 的一个非常小的子集，适合嵌入式开发。

1.3 概览

OS X 系统是分层构建的，每层都构建在另外一层之上。它的核心是 XNU 内核，提供了一个功能完整的 UNIX03 规范系统。Darwin 用户空间代码环绕着这个内核，其上运行着 Quartz 显示系统。API 也是用类似方式构建的，Core Foundation 使用 C 标准库提供所有程序使用的一组抽象数据类型。然后他们被 Cocoa Foundation 框架包装、扩展提供了友好的 Objective-C API，然后进一步由 Application Kit 扩展。

这些层都没有替代其下面的层。在同一个程序里，可以调用 C 标准库、Core Foundation、Foundation 和 AppKit。比如在关注速度的地方使用底层功能，在需要灵活性时使用抽象接口。这个平台主要的一个好处就是，在速度和方便的权衡上有很细粒度的选择。