多线程编程指南

原著：Apple Inc.

翻译：謝業蘭 【老狼】

联系：xyl.layne@gmail.com

鸣谢：有米移动广告平台

CocoaChina 社区

目录

简介

线程是单个应用中可以并发执行多个代码路径的多种技术之一。虽然更新的技术如操作对象（Operation objects）和 Grand Central Dispatch（GCD），提供一个更加现代化和高效率的基础设施来实现多核并发，但是 Mac OS X 和 iOS 也提供一套接口来创建和管理线程。

本文档介绍了 Mac OS X 上面的可用线程包，并且演示如何使用它们。本文档还描述了在你的应用中多线程代码同步的相关技术。

重要：*如果你正在创建一个新的应用程序，我们鼓励你研究 Mac OS X 上面实现并发的替代方法。如果还没有熟悉掌握需要实现一个多线程应用的设计技术的话，我们更鼓励你那样做。这些替代方法简化了大量原本你需要实现来执行并发路径的工作，并且提供了比传统线程更好的性能。获取更多相关技术的信息，你可以查阅Concurrency Programming Guide。*

本文档结构

本篇文档包含了以下章节和附录：

- “关于多线程编程”介绍了多线程的概念和它们在应用设计里面的角色。
- “线程管理”提供了关于 Mac OS X 上面线程技术的相关信息，并且教你如果使用它们。
- “Run Loops” 提供有关如何管理在辅助线程中的循环事件处理的信息。
- “同步(Synchronization)” 介绍同步问题和你可以用于阻止多线程破坏你的数据或者导致你程序崩溃的工具。
- “线程安全总结” 提供了 Mac OS X 和 iOS 上面固有的线程安全的高度总结和它们的主要框架

第一章　关于多线程编程

多年来，计算机的最大性能主要受限于它的中心微处理器的速度。然而由于个别处理器已经开始达到它的瓶颈限制，芯片制造商开始转向多核设计，让计算机具有了同时执行多个任务的能力。尽管 Mac OS X 利用了这些核心优势，在任何时候可以执行系统相关的任务，但自己的应用程序也可以通过多线程方法利用这些优势。

1.1　什么是多线程

多线程是一个比较轻量级的方法来实现单个应用程序内多个代码执行路径。在系统级别内，程序并排执行，系统分配到每个程序的执行时间是基于该程序的所需时间和其他程序的所需时间来决定的。然而在每个应程序的内部，存在一个或多个执行线程，它同时或在一个几乎同时发生的方式里执行不同的任务。系统本身管理这些执行的线程，调度它们在可用的内核上运行，并在需要让其他线程执行的时候抢先打断它们。

从技术角度来看，一个线程就是一个需要管理执行代码的内核级和应用级数据结构组合。内核级结构协助调度线程事件，并抢占式调度一个线程到可用的内核之上。应用级结构包括用于存储函数调用的调用堆栈和应用程序需要管理和操作线程属性和状态的结构。

在非并发的应用程序，只有一个执行线程。该线程开始和结束于你应用程序的 main 循环，一个个方法和函数的分支构成了你整个应用程序的所有行为。与此相反，支持并发的应用程序开始可以在需要额外的执行路径时候创建一个或多个线程。每个新的执行路径有它自己独立于应用程序 main 循环的定制开始循环。在应用程序中存在多个线程提供了两个非常重要的的潜在优势：

- 多个线程可以提高应用程序的感知响应。
- 多个线程可以提高应用程序在多核系统上的实时性能。

如果你的应用程序只有单独的线程，那么该独立程序需要完成所有的事情。它必须对事件作出响应，更新您的应用程序的窗口，并执行所有实现你应用程序行为需要的计算。拥有单独线程的主要问题是在同一时间里面它只能执行一个任务。那么当你的应用程序需要很长时间才能完成的时候会发生什么呢？当你的代码忙于计算你所

需要的值的时候，你的程序就会停止响应用户事件和更新它的窗口。如果这样的情况持续足够长的时间，用户就会误认为你的程序被挂起了，并试图强制退出。如果你把你的计算任务转移到一个独立的线程里面，那么你的应用程序主线程就可以自由并及时响应用户的交互。

当然多线程并不是解决程序性能问题的灵丹妙药。多线程带来好处同时也伴随着潜在问题。应用程序内拥有多个可执行路径，会给你的代码增加更多的复杂性。每个线程需要和其他线程协调其行为，以防止它破坏应用程序的状态信息。因为应用程序内的多个线程共享内存空间，它们访问相同的数据结构。如果两个线程试图同时处理相同的数据结构，一个线程有可能覆盖另外线程的改动导致破坏该数据结构。即使有适当的保护，你仍然要注意由于编译器的优化导致给你代码产生很微妙的（和不那么微妙）的 Bug。

1.2 线程术语

在讨论多线程和它支持的相关技术之前，我们有必要先了解一些基本的术语。如果你熟悉 Carbon 的多处理器服务 API 或者 UNIX 系统的话，你会发现本文档里面“任务(task)”被用于不同的定义。在 Mac OS 的早期版本，术语“任务(task)”是用来区分使用多处理器服务创建的线程和使用 Carbon 线程管理 API 创建的线程。在 UNIX 系统里面，术语“任务（task)”也在一段时间内被用于指代运行的进程。在实际应用中，多处理器服务任务是相当于抢占式的线程。

由于 Carbon 线程管理器和多处理器服务 API 是 Mac OS X 的传统技术，本文件采用下列术语：

- **线程(线程)**用于指代独立执行的代码段。
- **进程(process)**用于指代一个正在运行的可执行程序，它可以包含多个线程。
- **任务(task)**用于指代抽象的概念，表示需要执行工作。

1.3 多线程的替代方法

你自己创建多线程代码的一个问题就是它会给你的代码带来不确定性。多线程是一个相对较低的水平和复杂的方式来支持你的应用程序并发。如果你不完全理解你的设计选择的影响，你可能很容易遇到同步或定时问题，其范围可以从细微的行为变化

到严重到让你的应用程序崩溃并破坏用户数据。

你需要考虑的另一个因素是你是否真的需要多线程或并发。多线程解决了如何在同一个进程内并发的执行多路代码路径的问题。然而在很多情况下你是无法保证你所在做的工作是并发的。多线程引入带来大量的开销，包括内存消耗和 CPU 占用。你会发现这些开销对于你的工作而言实在太大，或者有其他方法会更容易实现。

表 1-1 列举了多线程的替代方法。该表包含了多线程的替代技术(比如操作对象和 GCD)和如何更高效的使用单个线程。

Table 1-1 Alternative technologies to threads

Technology	Description
Operation objects	Introduced in Mac OS X v10.5, an operation object is a wrapper for a task that would normally be executed on a secondary thread. This wrapper hides the thread management aspects of performing the task, leaving you free to focus on the task itself. You typically use these objects in conjunction with an operation queue object, which actually manages the execution of the operation objects on one more threads. For more information on how to use operation objects, see *Concurrency Programming Guide*.
Grand Central Dispatch (GCD)	Introduced in Mac OS x v10.6, Grand Central Dispatch is another alternative to threads that lets you focus on the tasks you need to perform rather than on thread management. With GCD, you define the task you want to perform and add it to a work queue, which handles the scheduling of your task on an appropriate thread. Work queues take into account the number of available cores and the current load to execute your tasks more efficiently than you could do yourself using threads. For information on how to use GCD and work queues, see *Concurrency Programming Guide*
Idle-time notifications	For tasks that are relatively short and very low priority, idle time notifications let you perform the task at a time when your application is not as busy. Cocoa provides support for idle-time notifications using the `NSNotificationQueue` object. To request an idle-time notification, post a notification to the default `NSNotificationQueue` object using the `NSPostWhenIdle` option. The queue delays the delivery of your notification object until the run loop becomes idle. For more information, see *Notification Programming Topics*.
Asynchronous functions	The system interfaces include many asynchronous functions that provide automatic concurrency for you. These APIs may use system daemons and processes or create custom threads to perform their task and return the results to you. (The actual implementation is irrelevant because it is separated from your code.) As you design your application, look for functions that offer asynchronous behavior and consider using them instead of using the equivalent synchronous function on a custom thread.
Timers	You can use timers on your application's main thread to perform periodic tasks that are too trivial to require a thread, but which still require servicing at regular intervals. For information on timers, see "Timer Sources."
Separate processes	Although more heavyweight than threads, creating a separate process might be useful in cases where the task is only tangentially related to your application. You might use a process if a task requires a significant amount of memory or must be executed using root privileges. For example, you might use a 64-bit server process to compute a large data set while your 32-bit application displays the results to the user.

注意:*当使用 fork 函数加载独立进程的时候，你必须总是在 fork 后面调用 exec 或者类似的函数。基于 Core Foundation、Cocao 或者 Core Data 框架（无论显式还是隐式关联）的应用程序随后*

调用 exec 函数或者类似的函数都会导出不确定的结果。

1.4 线程支持

如果你已经有代码使用了多线程，Mac OS X 和 iOS 提供几种技术来在你的应用程序里面创建多线程。此外，两个系统都提供了管理和同步你需要在这些线程里面处理的工作。以下几个部分描述了一些你在 Mac OS X 和 iOS 上面使用多线程的时候需要注意的关键技术。

1.4.1 线程包

虽然多线程的底层实现机制是 Mach 的线程，你很少（即使有）使用 Mach 级的线程。相反，你会经常使用到更多易用的 POSIX 的 API 或者它的衍生工具。Mach 的实现没有提供多线程的基本特征，但是包括抢占式的执行模型和调度线程的能力，所以它们是相互独立的。

列表 1-2 列举你可以在你的应用程序使用的线程技术。

Table 1-2 Thread technologies

Technology	Description
Cocoa threads	Cocoa implements threads using the `NSThread` class. Cocoa also provides methods on `NSObject` for spawning new threads and executing code on already-running threads. For more information, see "Using NSThread" and "Using NSObject to Spawn a Thread."
POSIX threads	POSIX threads provide a C-based interface for creating threads. If you are not writing a Cocoa application, this is the best choice for creating threads. The POSIX interface is relatively simple to use and offers ample flexibility for configuring your threads. For more information, see "Using POSIX Threads"
Multiprocessing Services	Multiprocessing Services is a legacy C-based interface used by applications transitioning from older versions of Mac OS. This technology is available in Mac OS X only and should be avoided for any new development. Instead, you should use the `NSThread` class or POSIX threads. If you need more information on this technology, see *Multiprocessing Services Programming Guide*.

在应用层上，其他平台一样所有线程的行为本质上是相同的。线程启动之后，线程就进入三个状态中的任何一个:运行(running)、就绪(ready)、阻塞(blocked)。如果一个线程当前没有运行，那么它不是处于阻塞，就是等待外部输入，或者已经准备就绪等待分配 CPU。线程持续在这三个状态之间切换，直到它最终退出或者进入中断状态。

当你创建一个新的线程，你必须指定该线程的入口点函数（或 Cocoa 线程时候为入口点方法）。该入口点函数由你想要在该线程上面执行的代码组成。但函数返回的

时候，或你显式的中断线程的时候，线程永久停止，且被系统回收。因为线程创建需要的内存和时间消耗都比较大，因此建议你的入口点函数做相当数量的工作，或建立一个运行循环允许进行经常性的工作。

为了获取更多关于线程支持的可用技术并且如何使用它们，请阅读“线程管理部分”。

1.4.2 Run Loops

注:为了便于记忆，文本后面部分翻译 Run Loops 的时候基本采用原义，而非翻译为“运行循环”。

一个 run loop 是用来在线程上管理事件异步到达的基础设施。一个 run loop 为线程监测一个或多个事件源。当事件到达的时候，系统唤醒线程并调度事件到 run loop,然后分配给指定程序。如果没有事件出现和准备处理，run loop 把线程置于休眠状态。

你创建线程的时候不需要使用一个 run loop，但是如果你这么做的话可以给用户带来更好的体验。Run Loops 可以让你使用最小的资源来创建长时间运行线程。因为 run loop 在没有任何事件处理的时候会把它的线程置于休眠状态，它消除了消耗 CPU 周期轮询，并防止处理器本身进入休眠状态并节省电源。

为了配置 run loop,你所需要做的是启动你的线程，获取 run loop 的对象引用，设置你的事件处理程序，并告诉 run loop 运行。Cocoa 和 Carbon 提供的基础设施会自动为你的主线程配置相应的 run loop。如果你打算创建长时间运行的辅助线程，那么你必须为你的线程配置相应的 run loop。

关于 run loops 的详细信息和如何使用它们的例子会在“Run Loops”部分介绍。

1.4.3 同步工具

线程编程的危害之一是在多个线程之间的资源争夺。如果多个线程在同一个时间试图使用或者修改同一个资源，就会出现问题。缓解该问题的方法之一是消除共享资源，并确保每个线程都有在它操作的资源上面的独特设置。因为保持完全独立的资源是不可行的，所以你可能必须使用锁，条件，原子操作和其他技术来同步资源的访问。

锁提供了一次只有一个线程可以执行代码的有效保护形式。最普遍的一种锁是互

斥排他锁，也就是我们通常所说的“mutex”。当一个线程试图获取一个当前已经被其他线程占据的互斥锁的时候，它就会被阻塞直到其他线程释放该互斥锁。系统的几个框架提供了对互斥锁的支持，虽然它们都是基于相同的底层技术。此外 Cocoa 提供了几个互斥锁的变种来支持不同的行为类型，比如递归。获取更多关于锁的种类的信息，请阅读“锁”部分内容。

除了锁，系统还提供了条件，确保在你的应用程序任务执行的适当顺序。一个条件作为一个看门人，阻塞给定的线程，直到它代表的条件变为真。当发生这种情况的时候，条件释放该线程并允许它继续执行。POSIX 级别和基础框架都直接提供了条件的支持。（如果你使用操作对象，你可以配置你的操作对象之间的依赖关系的顺序确定任务的执行顺序，这和条件提供的行为非常相似）。

尽管锁和条件在并发设计中使用非常普遍，原子操作也是另外一种保护和同步访问数据的方法。原子操作在以下情况的时候提供了替代锁的轻量级的方法，其中你可以执行标量数据类型的数学或逻辑运算。原子操作使用特殊的硬件设施来保证变量的改变在其他线程可以访问之前完成。

获取更多关于可用同步工具信息，请阅读“同步工具”部分。

1.4.4 线程间通信

虽然一个良好的设计最大限度地减少所需的通信量，但在某些时候，线程之间的通信显得十分必要。（线程的任务是为你的应用程序工作，但如果从来没有使用过这些工作的结果，那有什么好处呢？）线程可能需要处理新的工作要求，或向你应用程序的主线程报告其进度情况。在这些情况下，你需要一个方式来从其他线程获取信息。幸运的是，线程共享相同的进程空间，意味着你可以有大量的可选项来进行通信。

线程间通信有很多种方法，每种都有它的优点和缺点。“配置线程局部存储”列出了很多你可以在 Mac OS X 上面使用的通信机制。（异常的消息队列和 Cocoa 分布式对象，这些技术也可在 iOS 用来通信）。本表中的技术是按照复杂性的顺序列出。

Table 1-3 Communication mechanisms

Mechanism	Description
Direct messaging	Cocoa applications support the ability to perform selectors directly on other threads. This capability means that one thread can essentially execute a method on any other thread. Because they are executed in the context of the target thread, messages sent this way are automatically serialized on that thread. For information about input sources, see

	"Cocoa Perform Selector Sources."
Global variables, shared memory, and objects	Another simple way to communicate information between two threads is to use a global variable, shared object, or shared block of memory. Although shared variables are fast and simple, they are also more fragile than direct messaging. Shared variables must be carefully protected with locks or other synchronization mechanisms to ensure the correctness of your code. Failure to do so could lead to race conditions, corrupted data, or crashes.
Conditions	Conditions are a synchronization tool that you can use to control when a thread executes a particular portion of code. You can think of conditions as gate keepers, letting a thread run only when the stated condition is met. For information on how to use conditions, see "Using Conditions."
Run loop sources	A custom run loop source is one that you set up to receive application-specific messages on a thread. Because they are event driven, run loop sources put your thread to sleep automatically when there is nothing to do, which improves your thread's efficiency. For information about run loops and run loop sources, see "Run Loops."
Ports and sockets	Port-based communication is a more elaborate way to communication between two threads, but it is also a very reliable technique. More importantly, ports and sockets can be used to communicate with external entities, such as other processes and services. For efficiency, ports are implemented using run loop sources, so your thread sleeps when there is no data waiting on the port. For information about run loops and about port-based input sources, see "Run Loops."
Message queues	The legacy Multiprocessing Services defines a first-in, first-out (FIFO) queue abstraction for managing incoming and outgoing data. Although message queues are simple and convenient, they are not as efficient as some other communications techniques. For more information about how to use message queues, see *Multiprocessing Services Programming Guide*.
Cocoa distributed objects	Distributed objects is a Cocoa technology that provides a high-level implementation of port-based communications. Although it is possible to use this technology for inter-thread communication, doing so is highly discouraged because of the amount of overhead it incurs. Distributed objects is much more suitable for communicating with other processes, where the overhead of going between processes is already high. For more information, see*Distributed Objects Programming Topics*.

1.5 设计技巧

以下各节帮助你实现自己的线程提供了指导，以确保你代码的正确性。部分指南同时提供如何利用你的线程代码获得更好的性能。任何性能的技巧，你应该在你更改你代码之前、期间、之后总是收集相关的性能统计数据。

1.5.1 避免显式创建线程

手动编写线程创建代码是乏味的，而且容易出现错误，你应该尽可能避免这样做。Mac OS X 和 iOS 通过其他 API 接口提供了隐式的并发支持。你可以考虑使用异步 API，GCD 方式，或操作对象来实现并发，而不是自己创建一个线程。这些技术背后为你做了线程相关的工作，并保证是无误的。此外，比如 GCD 和操作对象技术被设计用来管

理线程，比通过自己的代码根据当前的负载调整活动线程的数量更高效。 关于更多 GCD 和操作对象的信息，你可以查阅“并发编程指南(Concurrency Programming Guid)”。

1.5.2 保持你的线程合理的忙

如果你准备人工创建和管理线程，记得多线程消耗系统宝贵的资源。你应该尽最大努力确保任何你分配到线程的任务是运行相当长时间和富有成效的。同时你不应该害怕中断那些消耗最大空闲时间的线程。线程使用一个平凡的内存量，它的一些有线，所以释放一个空闲线程，不仅有助于降低您的应用程序的内存占用，它也释放出更多的物理内存使用的其他系统进程。线程占用一定量的内存，其中一些是有线的，所以释放空闲线程不但帮助你减少了你应用程序的内存印记，而且还能释放出更多的物理内存给其他系统进程使用。

重要:*在你中断你的空闲线程开始之前，你必须总是记录你应用程序当前的性能基线测量。当你尝试修改后，采取额外的测量来确保你的修改实际上提高了性能，而不是对它操作损害。*

1.5.3 避免共享数据结构

避免造成线程相关资源冲突的最简单最容易的办法是给你应用程序的每个线程一份它需求的数据的副本。当最小化线程之间的通信和资源争夺时并行代码的效果最好。

创建多线程的应用是很困难的。即使你非常小心，并且在你的代码里面所有正确的地方锁住共享资源，你的代码依然可能语义不安全的。比如，当在一个特定的顺序里面修改共享数据结构的时候，你的代码有可能遇到问题。以原子方式修改你的代码，来弥补可能随后对多线程性能产生损耗的情况。把避免资源争夺放在首位通常可以得到简单的设计同样具有高性能的效果。

1.5.4 多线程和你的用户界面

如果你的应用程序具有一个图形用户界面，建议你在主线程里面接收和界面相关的事件和初始化更新你的界面。这种方法有助于避免与处理用户事件和窗口绘图相关的同步问题。一些框架，比如 Cocoa,通常需要这样操作，但是它的事件处理可以不

这样做，在主线程上保持这种行为的优势在于简化了管理你应用程序用户界面的逻辑。

有几个显著的例外，它有利于在其他线程执行图形操作。比如，QuickTime API 包含了一系列可以在辅助线程执行的操作，包括打开视频文件，渲染视频文件，压缩视频文件，和导入导出图像。类似的，在 Carbon 和 Cocoa 里面，你可以使用辅助线程来创建和处理图片和其他图片相关的计算。使用辅助线程来执行这些操作可以极大提高性能。如果你不确定一个操作是否和图像处理相关，那么你应该在主线程执行这些操作。

关于 QuickTime 线程安全的信息，查阅 Technical Note TN2125:“QuickTime 的线程安全编程”。关于 Cocoa 线程安全的更多信息，查阅“线程安全总结”。关于 Cocoa 绘画信息，查阅 Cocoa 绘画指南（Cocoa Drawing Guide）。

1.5.5 了解线程退出时的行为

进程一直运行直到所有非独立线程都已经退出为止。默认情况下，只有应用程序的主线程是以非独立的方式创建的，但是你也可以使用同样的方法来创建其他线程。当用户退出程序的时候，通常考虑适当的立即中断所有独立线程，因为通常独立线程所做的工作都是是可选的。如果你的应用程序使用后台线程来保存数据到硬盘或者做其他周期行的工作，那么你可能想把这些线程创建为非独立的来保证程序退出的时候不丢失数据。

以非独立的方式创建线程（又称作为可连接的）你需要做一些额外的工作。因为大部分上层线程封装技术默认情况下并没有提供创建可连接的线程，你必须使用 POSIX API 来创建你想要的线程。此外，你必须在你的主线程添加代码，来当它们最终退出的时候连接非独立的线程。更多有关创建可连接的线程信息，请查阅“设置线程的脱离状态”部分。

如果你正在编程 Cocoa 的程序，你也可以通过使用 applicationShouldTerminate: 的委托方法来延迟程序的中断直到一段时间后或者完成取消。当延迟中断的时候，你的程序需要等待直到任何周期线程已经完成它们的任务且调用了 replyToApplicationShouldTerminate:方法。关于更多这些方法的信息，请查阅 NSApplication Class Reference。

1.5.6 处理异常

当抛出一个异常时，异常的处理机制依赖于当前调用堆栈执行任何必要的清理。因为每个线程都有它自己的调用堆栈，所以每个线程都负责捕获它自己的异常。如果在辅助线程里面捕获一个抛出的异常失败，那么你的主线程也同样捕获该异常失败：它所属的进程就会中断。你无法捕获同一个进程里面其他线程抛出的异常。

如果你需要通知另一个线程（比如主线程）当前线程中的一个特殊情况，你应该捕捉异常，并简单地将消息发送到其他线程告知发生了什么事。根据你的模型和你正在尝试做的事情，引发异常的线程可以继续执行（如果可能的话），等待指示，或者干脆退出。

注意:*在 Cocoa 里面，一个 NSException 对象是一个自包含对象，一旦它被引发了，那么它可以从一个线程传递到另外一个线程。*

在一些情况下，异常处理可能是自动创建的。比如，Objective-C 中的 @synchronized 包含了一个隐式的异常处理。

1.5.7 干净地中断你的线程

线程自然退出的最好方式是让它达到其主入口结束点。虽然有不少函数可以用来立即中断线程，但是这些函数应仅用于作为最后的手段。在线程达到它自然结束点之前中断一个线程阻碍该线程清理完成它自己。如果线程已经分配了内存，打开了文件，或者获取了其他类型资源，你的代码可能没办法回收这些资源，结果造成内存泄漏或者其他潜在的问题。

关于更多正确退出线程的信息，请查阅“中断线程”部分。

1.5.8 线程安全的库

虽然应用程序开发人员控制应用程序是否执行多个线程，类库的开发者则无法这样控制。当开发类库时，你必须假设调用应用程序是多线程，或者多线程之间可以随时切换。因此你应该总是在你的临界区使用锁功能。

对类库开发者而言，只当应用程序是多线程的时候才创建锁是不明智的。如果你需要锁定你代码中的某些部分，早期应该创建锁对象给你的类库使用，更好是显式调用初始化类库。虽然你也可以使用静态库的初始化函数来创建这些锁，但是仅当没有

其他方式的才应该这样做。执行初始化函数需要延长加载你类库的时间，且可能对你程序性能造成不利影响。

注意：*永远记住在你的类库里面保持锁和释放锁的操作平衡。你应该总是记住锁定类库的数据结构，而不是依赖调用的代码提供线程安全环境。*

如果你真正开发 Cocoa 的类库，那么当你想在应用程序变成多线程的时候收到通知的话，你可以给 NSWillBecomeMultiThreadedNotification 注册一个观察者。不过你不应用依赖于这些收到的通知，因为它们可能在你的类库被调用之前已经被发出了。

第二章　线程管理

Mac OS X 和 iOS 里面的每个进程都是有一个或多个线程构成，每个线程都代表一个代码的执行路径。每个应用程序启动时候都是一个线程，它执行程序的 main 函数。应用程序可以生成额外的线程，其中每个线程执行一个特定功能的代码。

当应用程序生成一个新的线程的时候，该线程变成应用程序进程空间内的一个实体。每个线程都拥有它自己的执行堆栈，由内核调度独立的运行时间片。一个线程可以和其他线程或其他进程通信，执行 I/O 操作，甚至执行任何你想要它完成的任务。因为它们处于相同的进程空间，所以一个独立应用程序里面的所有线程共享相同的虚拟内存空间，并且具有和进程相同的访问权限。

本章提供了 Mac OS X 和 iOS 上面可用线程技术的预览，并给出了如何在你的应用程序里面使用它们的例子。

注意：*获取关于 Mac OS 上面线程架构，或者更多关于线程的背景资料。请参阅技术说明 TN2028 --“线程架构”。*

2.1　线程成本

多线程会占用你应用程序(和系统的)的内存使用和性能方面的资源。每个线程都需要分配一定的内核内存和应用程序内存空间的内存。管理你的线程和协调其调度所需的核心数据结构存储在使用 Wired Memory 的内核里面。你线程的堆栈空间和每个线程的数据都被存储在你应用程序的内存空间里面。这些数据结构里面的大部分都是当你首次创建线程或者进程的时候被创建和初始化的，它们所需的代价成本很高，因为需要和内核交互。

表 2-1 量化了在你应用程序创建一个新的用户级线程所需的大致成本。这些成本里面的部分是可配置的，比如为辅助线程分配堆栈空间的大小。创建一个线程所需的时间成本是粗略估计的，仅用于当互相比较的时候。线程创建时间很大程度依赖于处理器的负载，计算速度，和可用的系统和程序空间。

Table 2-1 Thread creation costs

Item	Approximate cost	Notes
Kernel data structures	Approximately 1 KB	This memory is used to store the thread data structures and attributes, much of which is allocated as wired memory and therefore cannot be

		paged to disk.
Stack space	512 KB (secondary threads) 8 MB (Mac OS X main thread) 1 MB (iOS main thread)	The minimum allowed stack size for secondary threads is 16 KB and the stack size must be a multiple of 4 KB. The space for this memory is set aside in your process space at thread creation time, but the actual pages associated with that memory are not created until they are needed.
Creation time	Approximately 90 microseconds	This value reflects the time between the initial call to create the thread and the time at which the thread's entry point routine began executing. The figures were determined by analyzing the mean and median values generated during thread creation on an Intel-based iMac with a 2 GHz Core Duo processor and 1 GB of RAM running Mac OS X v10.5.

注意:*因为底层内核的支持，操作对象(Operation objectis)可能创建线程更快。它们使用内核里面常驻线程池里面的线程来节省创建的时间，而不是每次都创建新的线程。关于更多使用操作对象(Operation objects)的信息，参阅并发编程指南(Concurrency Programming Guide)。*

当编写线程代码时另外一个需要考虑的成本是生产成本。设计一个线程应用程序有时会需要根本性改变你应用程序数据结构的组织方式。要做这些改变可能需要避免使用同步，因为本身设计不好的应用可能会造成巨大的性能损失。设计这些数据结构和在线程代码里面调试问题会增加开发一个线程应用所需的时间。然而避免这些消耗的话，可能在运行时候带来更大的问题，如果你的多线程花费太多的时间在锁的等待而没有做任何事情。

2.2 创建一个线程

创建低级别的线程相对简单。在所有情况下，你必须有一个函数或方法作为线程的主入口点，你必须使用一个可用的线程例程启动你的线程。以下几个部分介绍了比较常用线程创建的基本线程技术。线程创建使用了这些技术的继承属性的默认设置，由你所使用的技术来决定。关于更多如何配置你的线程的信息，参阅“线程属性配置”部分。

2.2.1 使用 NSThread

使用 NSThread 来创建线程有两个可以的方法:

- 使用 detachNewThreadSelector:toTarget:withObject:类方法来生成一个新的线程。
- 创建一个新的 NSThread 对象，并调用它的 start 方法。(仅在 iOS 和 Mac OS

X v10.5 及其之后才支持）

这两种创建线程的技术都在你的应用程序里面新建了一个脱离的线程。一个脱离的线程意味着当线程退出的时候线程的资源由系统自动回收。这也同样意味着之后不需要在其他线程里面显式的连接（join）。因为 detachNewThreadSelctor:toTarget:withObject:方法在 Mac OS X 的任何版本都支持，所以在 Cocoa 应用里面使用多线程的地方经常可以发现它。为了生成一个新的线程，你只要简单的提供你想要使用为线程主体入口的方法的名称(被指定为一个 selector)，和任何你想在启动时传递给线程的数据。下面的示例演示了这种方法的基本调用，来使用当前对象的自定义方法来生成一个线程。

```
[NSThread detachNewThreadSelector:@selector(myThreadMainMethod:) toTarget:self
withObject:nil];
```

在 Mac OS X v10.5 之前，你使用 NSThread 类来生成多线程。虽然你可以获取一个 NSThread 对象并访问线程的属性，但你只能在线程运行之后在其内部做到这些。在 Mac OS X v10.5 支持创建一个 NSThread 对象，而无需立即生成一个相应的新线程（这些在 iOS 里面同样可用）。新版支持使得在线程启动之前获取并设置线程的很多属性成为可能。这也让用线程对象来引用正在运行的线程成为可能。

在 Mac OS X v10.5 及其之后初始化一个 NSThread 对象的简单方法是使用 initWithTarget:selector:object:方法。该方法和 detachNewThreadSelector:toTarget:withObject:方法来初始化一个新的 NSThread 实例需要相同的额外开销。然而它并没有启动一个线程。为了启动一个线程，你可以显式调用先对象的 start 方法，如下面代码：

```
NSThread* myThread = [[NSThread alloc] initWithTarget:self
                                     selector:@selector(myThreadMainMethod:)
                                     object:nil];
[myThread start];  // Actually create the thread
```

注意：*使用 initWithTarget:selector:object:方法的替代办法是子类化 NSThread，并重写它的 main 方法。你可以使用你重写的该方法的版本来实现你线程的主体入口。更多信息，请参阅 NSThread Class Reference 里面子类化的提示。*

如果你拥有一个 NSThread 对象，它的线程当前真正运行，你可以给该线程发送消息的唯一方法是在你应用程序里面的任何对象使用

performSelector:onThread:withObject:waitUntilDone:方法。在 Mac OS X v10.5 支持在多线程上面执行 selectors（而不是在主线程里面），并且它是实现线程间通信的便捷方法。你使用该技术时所发送的消息会被其他线程作为 run-loop 主体的一部分直接执行（当然这些意味着目标线程必须在它的 run loop 里面运行，参阅“ Run Loops”）。当你使用该方法来实现线程通信的时候，你可能仍然需要一个同步操作，但是这比在线程间设置通信端口简单多了。

注意:*虽然在线程间的偶尔通信的时候使用该方法很好，但是你不能周期的或频繁的使用 performSelector:onThread:withObject:waitUntilDone:来实现线程间的通信。*

关于线程间通信的可选方法，参阅“设置线程的脱离状态”部分。

2.2.2 使用 POSIX 的多线程

Mac OS X 和 iOS 提供基于 C 语言支持的使用 POSIX 线程 API 来创建线程的方法。该技术实际上可以被任何类型的应用程序使用（包括 Cocoa 和 Cocoa Touch 的应用程序），并且如果你当前真为多平台开发应用的话，该技术可能更加方便。你使用来创建线程的 POSIX 例程被调用的时候，使用 pthread_create 刚好足够。

列表 2-1 显示了两个使用 POSIX 来创建线程的自定义函数。LaunchThread 函数创建了一个新的线程，该线程的例程由 PosixThreadMainRoutine 函数来实现。因为 POSIX 创建的线程默认情况是可连接的(joinable),下面的例子改变线程的属性来创建一个脱离的线程。把线程标记为脱离的，当它退出的时候让系统有机会立即回收该线程的资源。

Listing 2-1 Creating a thread in C

```
#include <assert.h>
#include <pthread.h>

void* PosixThreadMainRoutine(void* data)
{
    // Do some work here.

    return NULL;
}
```

```
void LaunchThread()
{
    // Create the thread using POSIX routines.
    pthread attr t  attr;
    pthread t      posixThreadID;
    int            returnVal;

    returnVal = pthread attr init(&attr);
    assert(!returnVal);
    returnVal = pthread attr setdetachstate(&attr, PTHREAD CREATE DETACHED);
    assert(!returnVal);

    int    threadError = pthread create(&posixThreadID, &attr, &PosixThreadMainRoutine,
NULL);

    returnVal = pthread attr destroy(&attr);
    assert(!returnVal);
    if (threadError != 0)
    {
        // Report an error.
    }
}
```

如果你把上面列表的代码添加到你任何一个源文件，并且调用 LaunchThread 函数，它将会在你的应用程序里面创建一个新的脱离线程。当然，新创建的线程使用该代码没有做任何有用的事情。线程将会加载并立即退出。为了让它更有兴趣，你需要添加代码到 PosixThreadMainRoutine 函数里面来做一些实际的工作。为了保证线程知道该干什么，你可以在创建的时候给线程传递一个数据的指针。把该指针作为 pthread_create 的最后一个参数。

为了在新建的线程里面和你应用程序的主线程通信，你需要建立一条和目标线程之间的稳定的通信路径。对于基于 C 语言的应用程序，有几种办法来实现线程间的通信，包括使用端口（ports），条件(conditions)和共享内存(shared memory)。对于

长期存在的线程，你应该几乎总是成立某种线程间的通信机制，让你的应用程序的主线程有办法来检查线程的状态或在应用程序退出时干净关闭它。

关于更多介绍 POSIX 线程函数的信息，参阅 pthread 的主页。

2.2.3 使用 NSObject 来生成一个线程

在 iOS 和 Mac OS X v10.5 及其之后，所有的对象都可能生成一个新的线程，并用它来执行它任意的方法。方法 performSelectorInBackground:withObject:新生成一个脱离的线程，使用指定的方法作为新线程的主体入口点。比如，如果你有一些对象（使用变量 myObj 来代表），并且这些对象拥有一个你想在后台运行的 doSomething 的方法，你可以使用如下的代码来生成一个新的线程:

```
[myObj performSelectorInBackground:@selector(doSomething) withObject:nil];
```

调用该方法的效果和你在当前对象里面使用 NSThread 的 detachNewThreadSelector:toTarget:withObject:传递 selectore，object 作为参数的方法一样。新的线程将会被立即生成并运行，它使用默认的设置。在 selectore 内部，你必须配置线程就像你在任何线程里面一样。比如，你可能需要设置一个自动释放池（如果你没有使用垃圾回收机制），在你要使用它的时候配置线程的 run loop。关于更是介绍如果配置线程的信息，参阅“配置线程属性”部分。

2.2.4 使用其他线程技术

尽管 POSIX 例程和 NSThread 类被推荐使用来创建低级线程，但是其他基于 C 语言的技术在 Mac OS X 上面同样可用。在这其中，唯一一个可以考虑使用的是多处理服务（Multiprocessing Services），它本身就是在 POSIX 线程上执行。多处理服务是专门为早期的 Mac OS 版本开发的，后来在 Mac OS X 里面的 Carbon 应用程序上面同样适用。如果你有代码真是有该技术，你可以继续使用它，尽管你应该把这些代码转化为 POSIX。该技术在 iOS 上面不可用。

关于更多如何使用多处理服务的信息，参阅*多处理服务编程指南（Multiprocessing Services Programming Guide）*。

2.2.5 在 Cocoa 程序上面使用 POSIX 线程

经管 NSThread 类是 Cocoa 应用程序里面创建多线程的主要接口，如果可以更方便的话你可以任意使用 POSIX 线程带替代。例如，如果你的代码里面已经使用了它，而你又不想改写它的话，这时你可能需要使用 POSIX 多线程。如果你真打算在 Cocoa 程序里面使用 POSIX 线程，你应该了解如果在 Cocoa 和线程间交互，并遵循以下部分的一些指南。

◆ Cocoa 框架的保护

对于多线程的应用程序，Cocoa 框架使用锁和其他同步方式来保证代码的正确执行。为了保护这些锁造成在单线程里面性能的损失，Cocoa 直到应用程序使用 NSThread 类生成它的第一个新的线程的时候才创建这些锁。如果你仅且使用 POSIX 例程来生成新的线程，Cocoa 不会收到关于你的应用程序当前变为多线程的通知。当这些刚好发生的时候，涉及 Cocoa 框架的操作哦可能会破坏甚至让你的应用程序崩溃。

为了让 Cocoa 知道你正打算使用多线程，你所需要做的是使用 NSThread 类生成一个线程，并让它立即退出。你线程的主体入口点不需要做任何事情。只需要使用 NSThread 来生成一个线程就足够保证 Cocoa 框架所需的锁到位。

如果你不确定 Cocoa 是否已经知道你的程序是多线程的，你可以使用 NSThread 的 isMultiThreaded 方法来检验一下。

◆ 混合 POSIX 和 Cocoa 的锁

在同一个应用程序里面混合使用 POSIX 和 Cocoa 的锁很安全。Cocoa 锁和条件对象基本上只是封装了 POSIX 的互斥体和条件。然而给定一个锁，你必须总是使用同样的接口来创建和操纵该锁。换言之，你不能使用 Cocoa 的 NSLock 对象来操纵一个你使用 pthread_mutex_init 函数生成的互斥体，反之亦然。

2.3 配置线程属性

创建线程之后，或者有时候是之前，你可能需要配置不同的线程环境。以下部分描述了一些你可以做的改变，和在什么时候你需要做这些改变。

2.3.1 配置线程的堆栈大小

对于每个你新创建的线程，系统会在你的进程空间里面分配一定的内存作为该线程的堆栈。该堆栈管理堆栈帧，也是任何线程局部变量声明的地方。给线程分配的内存大小在“线程成本”里面已经列举了。

如果你想要改变一个给定线程的堆栈大小，你必须在创建该线程之前做一些操作。所有的线程技术提供了一些办法来设置线程堆栈的大小。虽然可以使用 NSThread 来设置堆栈大小，但是它只能在 iOS 和 Mac OS X v10.5 及其之后才可用。表 2-2 列出了每种技术的对于不同的操作。

Table 2-2 Setting the stack size of a thread

Technology	Option
Cocoa	In iOS and Mac OS X v10.5 and later, allocate and initialize an `NSThread` object (do not use the`detachNewThreadSelector:toTarget:withObject:` method). Before calling the `start` method of the thread object, use the`setStackSize:` method to specify the new stack size.
POSIX	Create a new `pthread_attr_t` structure and use the `pthread_attr_setstacksize` function to change the default stack size. Pass the attributes to the `pthread_create` function when creating your thread.
Multiprocessing Services	Pass the appropriate stack size value to the `MPCreateTask` function when you create your thread.

2.3.2 配置线程本地存储

每个线程都维护了一个键-值的字典，它可以在线程里面的任何地方被访问。你可以使用该字典来保存一些信息，这些信息在整个线程的执行过程中都保持不变。比如，你可以使用它来存储在你的整个线程过程中 Run loop 里面多次迭代的状态信息。

Cocoa 和 POSIX 以不同的方式保存线程的字典，所以你不能混淆并同时调用者两种技术。然而只要你在你的线程代码里面坚持使用了其中一种技术，最终的结果应该是一样的。在 Cocoa 里面，你使用 NSThread 的 threadDictionary 方法来检索一个 NSMutableDictionary 对象，你可以在它里面添加任何线程需要的键。在 POSIX 里面，你使用 pthread_setspecific 和 pthread_getspecific 函数来设置和访问你线程的键和值。

2.3.3 设置线程的脱离状态

大部分上层的线程技术都默认创建了**脱离线程**（Datached thread）。大部分情况下，**脱离线程**(Detached thread)更受欢迎，因为它们允许系统在线程完成的时候立即释放它的数据结构。**脱离线程**同时不需要显示的和你的应用程序交互。意味着线程检索的结果由你来决定。相比之下，系统不回收**可连接线程**（Joinable thread）的资源直到另一个线程明确加入该线程，这个过程可能会阻止线程执行加入。

你可以认为**可连接线程**类似于子线程。虽然你作为独立线程运行，但是**可连接线程**在它资源可以被系统回收之前必须被其他线程连接。**可连接线程**同时提供了一个显示的方式来把数据从一个正在退出的线程传递到其他线程。在它退出之前，**可连接线程**可以传递一个数据指针或者其他返回值给 pthread_exit 函数。其他线程可以通过 pthread_join 函数来拿到这些数据。

重要：*在应用程序退出时，脱离线程可以立即被中断，而可连接线程则不可以。每个可连接线程必须在进程被允许可以退出的时候被连接。所以当线程处于周期性工作而不允许被中断的时候，比如保存数据到硬盘，可连接线程是最佳选择。*

如果你想要创建可连接线程，唯一的办法是使用 POSIX 线程。POSIX 默认创建的线程是可连接的。为了把线程标记为脱离的或可连接的，使用 pthread_attr_setdetachstate 函数来修改正在创建的线程的属性。在线程启动后，你可以通过调用 pthread_detach 函数来把线程修改为可连接的。关于更多 POSIX 线程函数信息，参与 pthread 主页。关于更多如果连接一个线程，参阅 pthread_join 的主页。

2.3.4 设置线程的优先级

你创建的任何线程默认的优先级是和你本身线程相同。内核调度算法在决定该运行那个线程时，把线程的优先级作为考量因素，较高优先级的线程会比较低优先级的线程具有更多的运行机会。较高优先级不保证你的线程具体执行的时间，只是相比较低优先级的线程，它更有可能被调度器选择执行而已。

重要：*让你的线程处于默认优先级值是一个不错的选择。增加某些线程的优先级，同时有可能增加了某些较低优先级线程的饥饿程度。如果你的应用程序包含较高优先级和较低优先级线程，而且它们之间必须交互，那么较低优先级的饥饿状态有可能阻塞其他线程，并造成性能瓶颈。*

如果你想改变线程的优先级，Cocoa 和 POSIX 都提供了一种方法来实现。对于 Cocoa 线程而言，你可以使用 NSThread 的 setThreadPriority:类方法来设置当前运行线程的优先级。对于 POSIX 线程，你可以使用 pthread_setschedparam 函数来实现。关于更多信息，参与 NSThread Class Reference 或 pthread_setschedparam 主页。

2.4 编写你线程的主体入口点

对于大部分而言，Mac OS X 上面线程结构的主体入口点和其他平台基本一样。你需要初始化你的数据结构，做一些工作或可行的设置一个 run loop，并在线程代码被执行完后清理它。根据设计，当你写的主体入口点的时候有可能需要采取一些额外的步骤。

2.4.1 创建一个自动释放池（Autorelease Pool）

在 Objective - C 框架链接的应用程序，通常在它们的每一个线程必须创建至少一个自动释放池。如果应用程序使用管理模型，即应用程序处理的 retain 和 release 对象，那么自动释放池捕获任何从该线程 autorelease 的对象。

如果应用程序使用的垃圾回收机制，而不是管理的内存模型，那么创建一个自动释放池不是绝对必要的。在垃圾回收的应用程序里面，一个自动释放池是无害的，而且大部分情况是被忽略。允许通过个代码管理必须同时支持垃圾回收和内存管理模型。在这种情况下，内存管理模型必须支持自动释放池，当应用程序运行垃圾回收的时候，自动释放池只是被忽略而已。

如果你的应用程序使用内存管理模型，在你编写线程主体入口的时候第一件事情就是创建一个自动释放池。同样，在你的线程最后应该销毁该自动释放池。该池保证自动释放。虽然对象被调用，但是它们不被 release 直到线程退出。列表 2-2 显示了线程主体入口使用自动释放池的基本结构。

Listing 2-2 Defining your thread entry point routine

```
- (void)myThreadMainRoutine
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init]; // Top-level pool

    // Do thread work here.
```

```
    [pool release];  // Release the objects in the pool.
}
```

因为高级的自动释放池不会释放它的对象直到线程退出。长时运行的线程需求新建额外的自动释放池来更频繁的释放它的对象。比如，一个使用 run loop 的线程可能在每次运行完一次循环的时候创建并释放该自动释放池。更频繁的释放对象可以防止你的应用程序内存占用太大造成性能问题。虽然对于任何与性能相关的行为，你应该测量你代码的实际表现，并适当地调整使用自动释放池。

关于更多内存管理的信息和自动释放池，参阅“内存高级管理编程指南(Advanced Memory Management Programming Guide)”。

2.4.2 设置异常处理

如果你的应用程序捕获并处理异常，那么你的线程代码应该时刻准备捕获任何可能发生的异常。虽然最好的办法是在异常发生的地方捕获并处理它，但是如果在你的线程里面捕获一个抛出的异常失败的话有可能造成你的应用程序强退。在你线程的主体入口点安装一个 try/catch 模块，可以让你捕获任何未知的异常，并提供一个合适的响应。

当在 Xcode 构建你项目的时候，你可以使用 C++或者 Objective-C 的异常处理风格。 关于更多设置如何在 Objective-C 里面抛出和捕获异常的信息，参阅 Exception Programming Topics。

2.4.3 设置一个 Run Loop

当你想编写一个独立运行的线程时，你有两种选择。第一种选择是写代码作为一个长期的任务，很少甚至不中断，线程完成的时候退出。第二种选择是把你的线程放入一个循环里面，让它动态的处理到来的任务请求。第一种方法不需要在你的代码指定任何东西；你只需要启动的时候做你打算做的事情即可。然而第二种选择需要在你的线程里面添加一个 run loop。

Mac OS X 和 iOS 提供了在每个线程实现 run loop 内置支持。Cocoa、Carbon 和 UIKit 自动在你应用程序的主线程启动一个 run loop，但是如果你创建任何辅助线程，

你必须手工的设置一个 run loop 并启动它。

关于更多使用和配置 run loop 的信息，参阅“Run Loops”部分。

2.5 中断线程

退出一个线程推荐的方法是让它在它主体入口点正常退出。经管 Cocoa、POSIX 和 Multiprocessing Services 提供了直接杀死线程的例程，但是使用这些例程是强烈不鼓励的。杀死一个线程阻止了线程本身的清理工作。线程分配的内存可能造成泄露，并且其他线程当前使用的资源可能没有被正确清理干净，之后造成潜在的问题。

如果你的应用程序需要在一个操作中间中断一个线程，你应该设计你的线程响应取消或退出的消息。对于长时运行的操作，这意味着周期性停止工作来检查该消息是否到来。如果该消息的确到来并要求线程退出，那么线程就有机会来执行任何清理和退出工作；否则，它返回继续工作和处理下一个数据块。

响应取消消息的一个方法是使用 run loop 的输入源来接收这些消息。列表 2-3 显示了该结构的类似代码在你的线程的主体入口里面是怎么样的(该示例显示了主循环部分，不包括设立一个自动释放池或配置实际的工作步骤)。该示例在 run loop 上面安装了一个自定义的输入源，它可以从其他线程接收消息。关于更多设置输入源的信息，参阅“配置 Run Loop 源”。执行工作的总和的一部分后，线程运行的 run loop 来查看是否有消息抵达输入源。如果没有，run loop 立即退出，并且循环继续处理下一个数据块。因为该处理器并没有直接的访问 exitNow 局部变量，退出条件是通过线程的字典来传输的。

Listing 2-3 Checking for an exit condition during a long job

```
- (void)threadMainRoutine
{
    BOOL moreWorkToDo = YES;
    BOOL exitNow = NO;
    NSRunLoop* runLoop = [NSRunLoop currentRunLoop];

    // Add the exitNow BOOL to the thread dictionary.
    NSMutableDictionary* threadDict = [[NSThread currentThread] threadDictionary];
    [threadDict setValue:[NSNumber numberWithBool:exitNow]
```

```
forKey:@"ThreadShouldExitNow"];


    // Install an input source.
    [self myInstallCustomInputSource];


    while (moreWorkToDo && !exitNow)
    {
        // Do one chunk of a larger body of work here.
        // Change the value of the moreWorkToDo Boolean when done.


        // Run the run loop but timeout immediately if the input source isn't waiting to
fire.
        [runLoop runUntilDate:[NSDate date]];


        // Check to see if an input source handler changed the exitNow value.
        exitNow = [[threadDict valueForKey:@"ThreadShouldExitNow"] boolValue];
    }
}
```

第三章　Run Loops

Run loops 是线程相关的的基础框架的一部分。一个 run loop 就是一个事件处理的循环，用来不停的调度工作以及处理输入事件。使用 run loop 的目的是让你的线程在有工作的时候忙于工作，而没工作的时候处于休眠状态。

Run loop 的管理并不完全自动的。你仍然需要设计你的线程代码在合适的时候启动 run loop 并正确响应输入事件。Cocoa 和 Core Fundation 都提供了 run loop objects 来帮助配置和管理你线程的 run loop。你的应用程序不需要显式的创建这些对象(run loop objects)；每个线程，包括程序的主线程都有与之对应的 run loop object。只有辅助线程才需要显式的运行它的 run loop。在 Carbon 和 Cocoa 程序中，主线程会自动创建并运行它 run loop，作为一般应用程序启动过程的一部分。

以下各部分提供更多关于 run loops 以及如何为你的应用程序配置它们。关于 run loop object 的额外信息，参阅 NSRunLoop Class Reference 和 CFRunLoop Reference 文档。

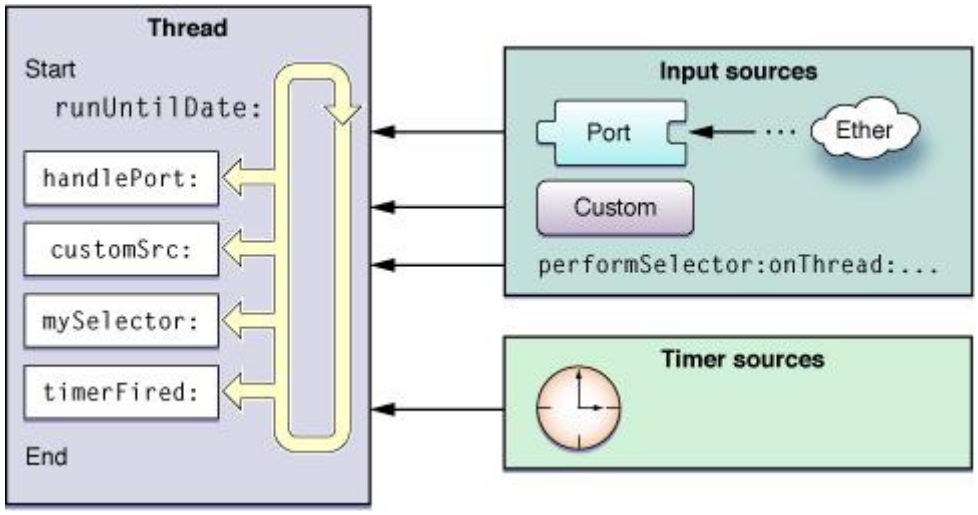
3.1　Run Loop 剖析

Run loop 本身听起来就和它的名字很像。它是一个循环，你的线程进入并使用它来运行响应输入事件的事件处理程序。你的代码要提供实现循环部分的控制语句，换言之就是要有 while 或 for 循环语句来驱动 run loop。在你的循环中，使用 run loop object 来运行事件处理代码，它响应接收到的事件并启动已经安装的处理程序。

Run loop 接收输入事件来自两种不同的来源：输入源（input source）和定时源（timer source）。**输入源**传递异步事件，通常消息来自于其他线程或程序。**定时源**则传递同步事件，发生在特定时间或者重复的时间间隔。两种源都使用程序的某一特定的处理例程来处理到达的事件。

图 3-1 显示了 run loop 的概念结构以及各种源。输入源传递异步消息给相应的处理例程，并调用 runUntilDate:方法来退出(在线程里面相关的 NSRunLoop 对象调用)。定时源则直接传递消息给处理例程，但并不会退出 run loop。

Figure 3-1 Structure of a run loop and its sources



除了处理输入源，run loops 也会生成关于 run loop 行为的通知(notifications)。注册的 run loop 观察者(run-loop Observers)可以收到这些通知，并在线程上面使用它们来做额外的处理。你可以使用 Core Foundation 在你的线程注册 run-loop 观察者。

下面部分介绍更多关于 run loop 的构成，以及其运行的模式。同时也提及在处理事件中不同时间生成的通知。

3.1.1 Run Loop 模式

Run loop 模式是所有要监视的输入源和定时源以及要通知的 run loop 注册观察者的集合。每次运行你的 run loop，你都要指定（无论显示还是隐式）其运行个模式。在 run loop 运行过程中，只有和模式相关的源才会被监视并允许他们传递事件消息。(类似的，只有和模式相关的观察者会通知 run loop 的进程)。和其他模式关联的源只有在 run loop 运行在其模式下才会运行，否则处于暂停状态。

通常在你的代码中，你可以通过指定名字来标识模式。Cocoa 和 Core foundation 定义了一个默认的和一些常用的模式，在你的代码中都是用字符串来标识这些模式。当然你也可以给模式名称指定一个字符串来自定义模式。虽然你可以给模式指定任意名字，但是模式的内容则不能是任意的。你必须添加一个或多个输入源，定时源或者 run loop 的观察者到你新建的模式中让他们有价值。

通过指定模式可以使得 run loop 在某一阶段过滤来源于源的事件。大多数时候，

run loop 都是运行在系统定义的默认模式上。但是模态面板（modal panel）可以运行在 “modal”模式下。在这种模式下，只有和模式面板相关的源才可以传递消息给线程。对于辅助线程，你可以使用自定义模式在一个时间周期操作上屏蔽优先级低的源传递消息。

注意：*模式区分基于事件的源而非事件的种类。例如，你不可以使用模式只选择处理鼠标按下或者键盘事件。你可以使用模式监听端口，暂停定时器或者改变其他源或者当前模式下处于监听状态 run loop 观察者。*

表 1-3 列出了 Cocoa 和 Core Foundation 定义的标准模式，并且介绍何时使用他们。名称那列列出了你用来在你代码中指定模式实际的常量。

Table 3-1 Predefined run loop modes

Mode	Name	Description
Default	NSDefaultRunLoopMode(Cocoa) kCFRunLoopDefaultMode (Core Foundation)	The default mode is the one used for most operations. Most of the time, you should use this mode to start your run loop and configure your input sources.
Connection	NSConnectionReplyMode(Cocoa)	Cocoa uses this mode in conjunction with NSConnection objects to monitor replies. You should rarely need to use this mode yourself.
Modal	NSModalPanelRunLoopMode(Cocoa)	Cocoa uses this mode to identify events intended for modal panels.
Event tracking	NSEventTrackingRunLoopMode(Cocoa)	Cocoa uses this mode to restrict incoming events during mouse-dragging loops and other sorts of user interface tracking loops.
Common modes	NSRunLoopCommonModes(Cocoa) kCFRunLoopCommonModes (Core Foundation)	This is a configurable group of commonly used modes. Associating an input source with this mode also associates it with each of the modes in the group. For Cocoa applications, this set includes the default, modal, and event tracking modes by default. Core Foundation includes just the default mode initially. You can add custom modes to the set using the CFRunLoopAddCommonMode function.

3.1.2 输入源

输入源异步的发送消息给你的线程。事件来源取决于输入源的种类：**基于端口的输入源**和**自定义输入源**。基于端口的输入源监听程序相应的端口。自定义输入源则监听自定义的事件源。至于 run loop，它不关心输入源的是基于端口的输入源还是自定义的输入源。系统会实现两种输入源供你使用。两类输入源的区别在于如何显示：基于端口的输入源由内核自动发送，而自定义的则需要人工从其他线程发送。

当你创建输入源，你需要将其分配给 run loop 中的一个或多个模式。模式只会在特定事件影响监听的源。大多数情况下，run loop 运行在默认模式下，但是你也可以使其运行在自定义模式。若某一源在当前模式下不被监听，那么任何其生成的消息只在 run loop 运行在其关联的模式下才会被传递。

基于端口的输入源

Cocoa 和 Core Foundation 内置支持使用端口相关的对象和函数来创建的基于端口的源。例如，在 Cocoa 里面你从来不需要直接创建输入源。你只要简单的创建端口对象，并使用 NSPort 的方法把该端口添加到 run loop。端口对象会自己处理创建和配置输入源。

在 Core Foundation，你必须人工创建端口和它的 run loop 源.在两种情况下，你都可以使用端口相关的函数（CFMachPortRef，CFMessagePortRef，CFSocketRef）来创建合适的对象。

更多例子关于如何设置和配置一个自定义端口源，参阅“配置一个基于端口的输入源”部分。

自定义输入源

为了创建自定义输入源，必须使用 Core Foundation 里面的 CFRunLoopSourceRef 类型相关的函数来创建。你可以使用回调函数来配置自定义输入源。Core Fundation 会在配置源的不同地方调用回调函数，处理输入事件，在源从 run loop 移除的时候清理它。

除了定义在事件到达时自定义输入源的行为，你也必须定义消息传递机制。源的这部分运行在单独的线程里面，并负责在数据等待处理的时候传递数据给源并通知它处理数据。消息传递机制的定义取决于你，但最好不要过于复杂。

关于创建自定义输入源的例子，参阅“定义一个自定义输入源”。关于自定义输入源的信息，参阅 CFRunLoopSource Reference。

Cocoa 执行 Selector 的源

除了基于端口的源，Cocoa 定义了自定义输入源，允许你在任何线程执行

selector。和基于端口的源一样，执行 selector 请求会在目标线程上序列化，减缓许多在线程上允许多个方法容易引起的同步问题。不像基于端口的源，一个 selector 执行完后会自动从 run loop 里面移除。

注意:在 Mac OS X v10.5 之前，执行 selector 多半可能是给主线程发送消息，但是在 Mac OS X v10.5 及其之后和在 iOS 里面，你可以使用它们给任何线程发送消息。

当在其他线程上面执行 selector 时，目标线程须有一个活动的 run loop。对于你创建的线程，这意味着线程在你显式的启动 run loop 之前处于等待状态。由于主线程自己启动它的 run loop，那么在程序通过委托调用 applicationDidFinishlaunching:的时候你会遇到线程调用的问题。因为 Run loop 通过每次循环来处理所有队列的 selector 的调用，而不是通过 loop 的迭代来处理 selector。

表 3-2 列出了 NSObject 中可在其它线程执行的 selector。由于这些方法时定义在 NSObject 中，你可以在任何可以访问 Objective-C 对象的线程里面使用它们，包括 POSIX 的所有线程。这些方法实际上并没有创建新的线程执行 selector。

Table 3-2 Performing selectors on other threads

Methods	Description
performSelectorOnMainThread:withObject:waitUntilDone: performSelectorOnMainThread:withObject:waitUntilDone:modes:	Performs the specified selector on the application's main thread during that thread's next run loop cycle. These methods give you the option of blocking the current thread until the selector is performed.
performSelector:onThread:withObject:waitUntilDone: performSelector:onThread:withObject:waitUntilDone:modes:	Performs the specified selector on any thread for which you have an NSThreadobject. These methods give you the option of blocking the current thread until the selector is performed.
performSelector:withObject:afterDelay: performSelector:withObject:afterDelay:inModes:	Performs the specified selector on the current thread during the next run loop cycle and after an optional delay period. Because it waits until the next run loop cycle to perform the selector, these methods provide an automatic mini delay from the currently executing code. Multiple queued selectors are performed one after another in the order they were queued.
cancelPreviousPerformRequestsWithTarget: cancelPreviousPerformRequestsWithTarget:selector:object:	Lets you cancel a message sent to the current thread using theperformSelector:withObject:afterDelay: orperformSelector:withObject:afterDelay:inModes: method.

关于更多介绍这些方法的信息，参阅 NSObject Class Reference。

定时源

定时源在预设的时间点同步方式传递消息。定时器是线程通知自己做某事的一种方法。例如，搜索控件可以使用定时器，当用户连续输入的时间超过一定时间时，就开始一次搜索。这样使用延迟时间，就可以让用户在搜索前有足够的时间来输入想要搜索的关键字。

经管定时器可以产生基于时间的通知，但它并不是实时机制。和输入源一样，定时器也和你的 run loop 的特定模式相关。如果定时器所在的模式当前未被 run loop 监视，那么定时器将不会开始直到 run loop 运行在相应的模式下。类似的，如果定时器在 run loop 处理某一事件期间开始，定时器会一直等待直到下次 run loop 开始相应的处理程序。如果 run loop 不再运行，那定时器也将永远不启动。

你可以配置定时器工作仅一次还是重复工作。重复工作定时器会基于安排好的时间而非实际时间调度它自己运行。举个例子，如果定时器被设定在某一特定时间开始并 5 秒重复一次，那么定时器会在那个特定时间后 5 秒启动，即使在那个特定的触发时间延迟了。如果定时器被延迟以至于它错过了一个或多个触发时间，那么定时器会在下一个最近的触发事件启动，而后面会按照触发间隔正常执行。

关于更多配置定时源的信息，参阅“配置定时源”部分。关于引用信息，查看 NSTimer Class Reference 或 CFRunLoopTimer Reference。

Run Loop 观察者

源是合适的同步或异步事件发生时触发，而 run loop 观察者则是在 run loop 本身运行的特定时候触发。你可以使用 run loop 观察者来为处理某一特定事件或是进入休眠的线程做准备。你可以将 run loop 观察者和以下事件关联：

- Run loop 入口
- Run loop 何时处理一个定时器
- Run loop 何时处理一个输入源
- Run loop 何时进入睡眠状态
- Run loop 何时被唤醒，但在唤醒之前要处理的事件

- Run loop 终止

你可以给 run loop 观察者添加到 Cocoa 和 Carbon 程序里面，但是如果你要定义观察者并把它添加到 run loop 的话，那就只能使用 Core Fundation 了。为了创建一个 run loop 观察者，你可以创建一个 CFRunLoopObserverRef 类型的实例。它会追踪你自定义的回调函数以及其它你感兴趣的活动。

和定时器类似，run loop 观察者可以只用一次或循环使用。若只用一次，那么在它启动后，会把它自己从 run loop 里面移除，而循环的观察者则不会。你在创建 run loop 观察者的时候需要指定它是运行一次还是多次。

关于如何创建一个 run loop 观察者的实例，参阅“配置 run loop”部分。关于更多的相关信息，参阅 CFRunLoopObserver Reference。

Run Loop 的事件队列

每次运行 run loop，你线程的 run loop 对会自动处理之前未处理的消息，并通知相关的观察者。具体的顺序如下：

1. 通知观察者 run loop 已经启动
2. 通知观察者任何即将要开始的定时器
3. 通知观察者任何即将启动的非基于端口的源
4. 启动任何准备好的非基于端口的源
5. 如果基于端口的源准备好并处于等待状态，立即启动；并进入步骤 9。
6. 通知观察者线程进入休眠
7. 将线程置于休眠直到任一下面的事件发生：
   - 某一事件到达基于端口的源
   - 定时器启动
   - Run loop 设置的时间已经超时
   - run loop 被显式唤醒
8. 通知观察者线程将被唤醒。
9. 处理未处理的事件
   - 如果用户定义的定时器启动，处理定时器事件并重启 run loop。进入步骤 2
   - 如果输入源启动，传递相应的消息

■ 如果 run loop 被显式唤醒而且时间还没超时，重启 run loop。进入步骤 2

10. 通知观察者 run loop 结束。

因为定时器和输入源的观察者是在相应的事件发生之前传递消息，所以通知的时间和实际事件发生的时间之间可能存在误差。如果需要精确时间控制，你可以使用休眠和唤醒通知来帮助你校对实际发生事件的时间。

因为当你运行 run loop 时定时器和其它周期性事件经常需要被传递，撤销 run loop 也会终止消息传递。典型的例子就是鼠标路径追踪。因为你的代码直接获取到消息而不是经由程序传递，因此活跃的定时器不会开始直到鼠标追踪结束并将控制权交给程序。

Run loop 可以由 run loop 对象显式唤醒。其它消息也可以唤醒 run loop。例如，添加新的非基于端口的源会唤醒 run loop 从而可以立即处理输入源而不需要等待其他事件发生后再处理。

3.2 何时使用 Run Loop

仅当在为你的程序创建辅助线程的时候，你才需要显式运行一个 run loop。Run loop 是程序主线程基础设施的关键部分。所以，Cocoa 和 Carbon 程序提供了代码运行主程序的循环并自动启动 run loop。IOS 程序中 UIApplication 的 run 方法（或 Mac OS X 中的 NSApplication）作为程序启动步骤的一部分，它在程序正常启动的时候就会启动程序的主循环。类似的，RunApplicationEventLoop 函数为 Carbon 程序启动主循环。如果你使用 xcode 提供的模板创建你的程序，那你永远不需要自己去显式的调用这些例程。

对于辅助线程，你需要判断一个 run loop 是否是必须的。如果是必须的，那么你要自己配置并启动它。你不需要在任何情况下都去启动一个线程的 run loop。比如，你使用线程来处理一个预先定义的长时间运行的任务时，你应该避免启动 run loop。Run loop 在你要和线程有更多的交互时才需要，比如以下情况：

- 使用端口或自定义输入源来和其他线程通信
- 使用线程的定时器
- Cocoa 中使用任何 performSelector...的方法
- 使线程周期性工作

如果你决定在程序中使用 run loop，那么它的配置和启动都很简单。和所有线程编程一样，你需要计划好在辅助线程退出线程的情形。让线程自然退出往往比强制关闭它更好。关于更多介绍如何配置和退出一个 run loop,参阅”使用 Run Loop 对象”的介绍。

3.3 使用 Run Loop 对象

Run loop 对象提供了添加输入源，定时器和 run loop 的观察者以及启动 run loop 的接口。每个线程都有唯一的与之关联的 run loop 对象。在 Cocoa 中，该对象是 NSRunLoop 类的一个实例；而在 Carbon 或 BSD 程序中则是一个指向 CFRunLoopRef 类型的指针。

3.3.1 获得 Run Loop 对象

为了获得当前线程的 run loop，你可以采用以下任一方式：

- 在 Cocoa 程序中，使用 NSRunLoop 的 currentRunLoop 类方法来检索一个 NSRunLoop 对象。
- 使用 CFRunLoopGetCurrent 函数。

虽然它们并不是完全相同的类型，但是你可以在需要的时候从 NSRunLoop 对象中获取 CFRunLoopRef 类型。NSRunLoop 类定义了一个 getCFRunLoop 方法，该方法返回一个可以传递给 Core Foundation 例程的 CFRunLoopRef 类型。因为两者都指向同一个 run loop，你可以在需要的时候混合使用 NSRunLoop 对象和 CFRunLoopRef 不透明类型。

3.3.2 配置 Run Loop

在你在辅助线程运行 run loop 之前，你必须至少添加一输入源或定时器给它。如果 run loop 没有任何源需要监视的话，它会在你启动之际立马退出。关于如何添加源到 run loop 里面的例子，参阅”配置 Run Loop 源”。

除了安装源，你也可以添加 run loop 观察者来监视 run loop 的不同执行阶段情况。为了给 run loop 添加一个观察者，你可以创建 CFRunLoopObserverRef 不透明类型，并使用 CFRunLoopAddObserver 将它添加到你的 run loop。Run loop 观察者必须

由 Core foundation 函数创建，即使是 Cocoa 程序。

列表 3-1 显示了附加一个 run loop 的观察者到它的 run loop 的线程主体例程。该例子的主要目的是显示如何创建一个 run loop 观察者，所以该代码只是简单的设置一个观察者来监视 run loop 的所有活动。基础处理程序(没有显示)只是简单的打印出 run loop 活动处理定时器请求的日志信息。

Listing 3-1 Creating a run loop observer

```
- (void)threadMain
{
    // The application uses garbage collection, so no autorelease pool is needed.
    NSRunLoop* myRunLoop = [NSRunLoop currentRunLoop];

    // Create a run loop observer and attach it to the run loop.
    CFRunLoopObserverContext  context = {0, self, NULL, NULL, NULL};
    CFRunLoopObserverRef    observer = CFRunLoopObserverCreate(kCFAllocatorDefault,
           kCFRunLoopAllActivities, YES, 0, &myRunLoopObserver, &context);

    if (observer)
    {
        CFRunLoopRef    cfLoop = [myRunLoop getCFRunLoop];
        CFRunLoopAddObserver(cfLoop, observer, kCFRunLoopDefaultMode);
    }

    // Create and schedule the timer.
    [NSTimer scheduledTimerWithTimeInterval:0.1 target:self
              selector:@selector(doFireTimer:) userInfo:nil repeats:YES];

    NSInteger    loopCount = 10;
    do
    {
        // Run the run loop 10 times to let the timer fire.
        [myRunLoop runUntilDate:[NSDate dateWithTimeIntervalSinceNow:1]];
        loopCount--;
```

```
    }
    while (loopCount);
}
```

当当前长时间运行的线程配置 run loop 的时候，最好添加至少一个输入源到 run loop 以接收消息。虽然你可以使用附属的定时器来进入 run loop，但是一旦定时器触发后，它通常就变为无效了，这会导致 run loop 退出。虽然附加一个循环的定时器可以让 run loop 运行一个相对较长的周期，但是这也会导致周期性的唤醒线程，这实际上是轮询（polling）的另一种形式而已。与之相反，输入源会一直等待某事件发生，在事情导致前它让线程处于休眠状态。

3.3.3 启动 Run Loop

启动 run loop 只对程序的辅助线程有意义。一个 run loop 通常必须包含一个输入源或定时器来监听事件。如果一个都没有，run loop 启动后立即退出。

有几种方式可以启动 run loop，包括以下这些：

- 无条件的
- 设置超时时间
- 特定的模式

无条件的进入 run loop 是最简单的方法，但也最不推荐使用的。因为这样会使你的线程处在一个永久的循环中，这会让你对 run loop 本身的控制很少。你可以添加或删除输入源和定时器，但是退出 run loop 的唯一方法是杀死它。没有任何办法可以让这 run loop 运行在自定义模式下。

替代无条件进入 run loop 更好的办法是用预设超时时间来运行 run loop，这样 run loop 运作直到某一事件到达或者规定的时间已经到期。如果是事件到达，消息会被传递给相应的处理程序来处理，然后 run loop 退出。你可以重新启动 run loop 来等待下一事件。如果是规定时间到期了，你只需简单的重启 run loop 或使用此段时间来做任何的其他工作。

除了超时机制，你也可以使用特定的模式来运行你的 run loop。模式和超时不是互斥的，他们可以在启动 run loop 的时候同时使用。模式限制了可以传递事件给 run loop 的输入源的类型，这在”Run Loop 模式”部分介绍。

列表 3-2 描述了线程的主要例程的架构。本示例的关键是说明了 run loop 的基本结构。本质上讲你添加自己的输入源或定时器到 run loop 里面，然后重复的调用一个程序来启动 run loop。每次 run loop 返回的时候，你需要检查是否有使线程退出的条件成立。示例中使用了 Core Foundation 的 run loop 例程，以便可以检查返回结果从而确定 run loop 为何退出。若是在 Cocoa 程序，你也可以使用 NSRunLoop 的方法运行 run loop，无需检查返回值。（关于使用 NSRunLoop 返回运行 run loop 的例子，查看列表 3-12）

Listing 3-2 Running a run loop

```
- (void)skeletonThreadMain
{
    // Set up an autorelease pool here if not using garbage collection.
    BOOL done = NO;

    // Add your sources or timers to the run loop and do any other setup.

    do
    {
        // Start the run loop but return after each source is handled.
        SInt32    result = CFRunLoopRunInMode(kCFRunLoopDefaultMode, 10, YES);

        // If a source explicitly stopped the run loop, or if there are no
        // sources or timers, go ahead and exit.
        if ((result == kCFRunLoopRunStopped) || (result == kCFRunLoopRunFinished))
            done = YES;

        // Check for any other exit conditions here and set the
        // done variable as needed.
    }
    while (!done);

    // Clean up code here. Be sure to release any allocated autorelease pools.
}
```

可以递归的运行 run loop。换句话说你可以使用 CFRunLoopRun，CFRunLoopRunInMode 或者任一 NSRunLoop 的方法在输入源或定时器的处理程序里面启动 run loop。这样做的话，你可以使用任何模式启动嵌套的 run loop，包括被外层 run loop 使用的模式。

3.3.4 退出 Run Loop

有两种方法可以让 run loop 处理事件之前退出:

- 给 run loop 设置超时时间
- 通知 run loop 停止

如果可以配置的话，推荐使用第一种方法。指定一个超时时间可以使 run loop 退出前完成所有正常操作，包括发送消息给 run loop 观察者。

使用 CFRunLoopStop 来显式的停止 run loop 和使用超时时间产生的结果相似。Run loop 把所有剩余的通知发送出去再退出。与设置超时的不同的是你可以在无条件启动的 run loop 里面使用该技术。

尽管移除 run loop 的输入源和定时器也可能导致 run loop 退出，但这并不是可靠的退出 run loop 的方法。一些系统例程会添加输入源到 run loop 里面来处理所需事件。因为你的代码未必会考虑到这些输入源，这样可能导致你无法没从系统例程中移除它们，从而导致退出 run loop。

3.3.5 线程安全和 Run Loop 对象

线程是否安全取决于你使用那些 API 来操纵你的 run loop。Core Foundation 中的函数通常是线程安全的，可以被任意线程调用。但是如果你修改了 run loop 的配置然后需要执行某些操作，任何时候你最好还是在 run loop 所属的线程执行这些操作。

至于 Cocoa 的 NSRunLoop 类则不像 Core Foundation 具有与生俱来的线程安全性。如果你想使用 NSRunLoop 类来修改你的 run loop，你应用在 run loop 所属的线程里面完成这些操作。给属于不同线程的 run loop 添加输入源和定时器有可能导致你的代码崩溃或产生不可预知的行为。

3.4 配置 Run loop 的源

以下部分列举了在 Cocoa 和 Core Foundation 里面如何设置不同类型的输入源的例子。

3.4.1 定义自定义输入源

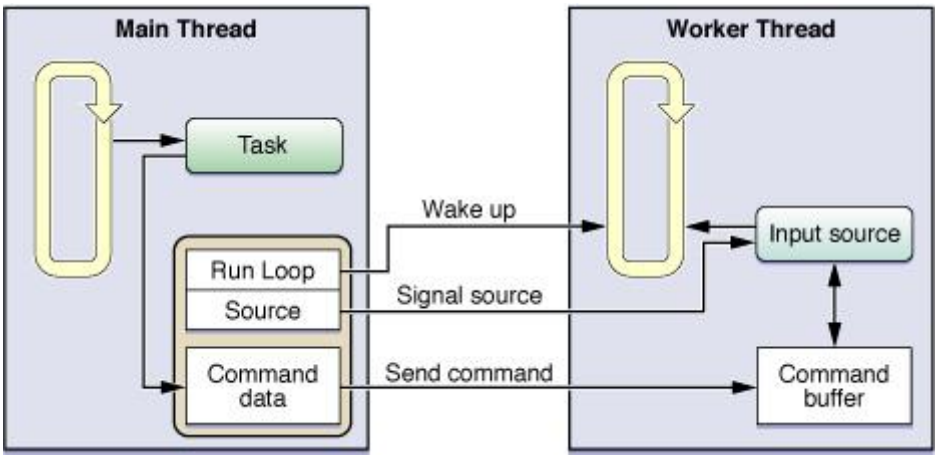
创建自定义的输入源包括定义以下内容：

- 输入源要处理的信息。
- 使感兴趣的客户端（可理解为其他线程）知道如何和输入源交互的调度例程。
- 处理其他任何客户端（可理解为其他线程）发送请求的例程。
- 使输入源失效的取消例程。

由于你自己创建输入源来处理自定义消息，实际配置选是灵活配置的。调度例程，处理例程和取消例程都是你创建自定义输入源时最关键的例程。然而输入源其他的大部分行为都发生在这些例程的外部。比如，由你决定数据传输到输入源的机制，还有输入源和其他线程的通信机制也是由你决定。

图 3-2 显示了一个自定义输入源的配置的例子。在该例中，程序的主线程维护了输入源的引用，输入源所需的自定义命令缓冲区和输入源所在的 run loop。当主线程有任务需要分发给工作线程时，主线程会给命令缓冲区发送命令和必须的信息来通知工作线程开始执行任务。（因为主线程和输入源所在工作线程都可以访问命令缓冲区，因此这些访问必须是同步的）一旦命令传送出去，主线程会通知输入源并且唤醒工作线程的 run loop。而一收到唤醒命令，run loop 会调用输入源的处理程序，由它来执行命令缓冲区中相应的命令。

Figure 3-2 Operating a custom input source



以下部分解释下上图的实现自定义输入源关键部分和你需要实现的关键代码。

定义输入源

定义自定义的输入源需要使用 Core Foundation 的例程来配置你的 run loop 源并把它添加到 run loop。尽管这些基本的处理例程是基于 C 的函数，但并不排除你可以对这些函数进行封装，并使用 Objective-C 或 Objective-C++来实现你代码的主体。

图 3-2 中的输入源使用了 Objective-C 的对象辅助 run loop 来管理命令缓冲区。列表 3-3 给出了该对象的定义。RunLoopSource 对象管理着命令缓冲区并以此来接收其他线程的消息。例子同样给出了 RunLoopContext 对象的定义，它是一个用于传递 RunLoopSource 对象和 run loop 引用给程序主线程的一个容器。

Listing 3-3 The custom input source object definition

```
@interface RunLoopSource : NSObject
{
    CFRunLoopSourceRef runLoopSource;
    NSMutableArray* commands;
}

- (id)init;
- (void)addToCurrentRunLoop;
- (void)invalidate;

// Handler method
- (void)sourceFired;

// Client interface for registering commands to process
- (void)addCommand:(NSInteger)command withData:(id)data;
- (void)fireAllCommandsOnRunLoop:(CFRunLoopRef)runloop;

@end

// These are the CFRunLoopSourceRef callback functions.
```

```
void RunLoopSourceScheduleRoutine (void *info, CFRunLoopRef rl, CFStringRef mode);
void RunLoopSourcePerformRoutine (void *info);
void RunLoopSourceCancelRoutine (void *info, CFRunLoopRef rl, CFStringRef mode);

// RunLoopContext is a container object used during registration of the input source.
@interface RunLoopContext : NSObject
{
    CFRunLoopRef       runLoop;
    RunLoopSource*       source;
}
@property (readonly) CFRunLoopRef runLoop;
@property (readonly) RunLoopSource* source;

- (id)initWithSource:(RunLoopSource*)src andLoop:(CFRunLoopRef)loop;
@end
```

尽管使用 Objective-C 代码来管理输入源的自定义数据，但是将输入源附加到 run loop 却需要使用基于 C 的回调函数。当你正在把你的 run loop 源附加到 run loop 的时候，使用列表 3-4 中的第一个函数（RunLoopSourceScheduleRoutine）。因为这个输入源只有一个客户端（即主线程），它使用调度函数发送注册信息给应用程序的委托（delegate）。当委托需要和输入源通信时，它会使用 RunLoopContext 对象来完成。

Listing 3-4 Scheduling a run loop source

```
void RunLoopSourceScheduleRoutine (void *info, CFRunLoopRef rl, CFStringRef mode)
{
    RunLoopSource* obj = (RunLoopSource*)info;
    AppDelegate*   del = [AppDelegate sharedAppDelegate];
    RunLoopContext* theContext = [[RunLoopContext alloc] initWithSource:obj andLoop:rl];

    [del performSelectorOnMainThread:@selector(registerSource:)
                          withObject:theContext waitUntilDone:NO];
}
```

一个最重要的回调例程就在输入源被告知时用来处理自定义数据的那个例程。列表 3-5 显示了如何调用这个和 RunLoopSource 对象相关回调例程。这里只是简单的让 RunLoopSource 执行 sourceFired 方法，然后继续处理在命令缓存区出现的命令。

Listing 3-5 Performing work in the input source

```
void RunLoopSourcePerformRoutine (void *info)
{
    RunLoopSource*  obj = (RunLoopSource*)info;
    [obj sourceFired];
}
```

如果你使用 CFRunLoopSourceInvalidate 函数把输入源从 run loop 里面移除的话，系统会调用你输入源的取消例程。你可以使用该例程来通知其他客户端该输入源已经失效，客户端应该释放输入源的引用。列表 3-6 显示了由已注册的 RunLoopSource 对取消例程的调用。这个函数将另一个 RunLoopContext 对象发送给应用的委托，当这次是要通知委托释放 run loop 源的引用。

Listing 3-6 Invalidating an input source

```
void RunLoopSourceCancelRoutine (void *info, CFRunLoopRef rl, CFStringRef mode)
{
    RunLoopSource* obj = (RunLoopSource*)info;
    AppDelegate* del = [AppDelegate sharedAppDelegate];
    RunLoopContext* theContext = [[RunLoopContext alloc] initWithSource:obj andLoop:rl];

    [del performSelectorOnMainThread:@selector(removeSource:)
                          withObject:theContext waitUntilDone:YES];
}
```

注意:*应用委托的 registerSource:和 removeSource:方法将在”协调客输入源的客户端”部分介绍。*

安装输入源到 Run Loop

列表 3-7 显示了 RunLoopSource 的 init 和 addToCurrentRunLoop 的方法。Init 方法创建 CFRunLoopSourceRef 的不透明类型，该类型必须被附加到 run loop 里面。它把 RunLoopSource 对象做为上下文引用参数，以便回调例程持有该对象的一个引用

指针。输入源的安装只在工作线程调用 addToCurrentRunLoop 方法才发生，此时 RunLoopSourceScheduledRoutine 被调用。一旦输入源被添加到 run loop，线程就运行 run loop 并等待事件。

Listing 3-7 Installing the run loop source

```
- (id)init
{
    CFRunLoopSourceContext    context = {0, self, NULL, NULL, NULL, NULL, NULL,
                                    &RunLoopSourceScheduleRoutine,
                                    RunLoopSourceCancelRoutine,
                                    RunLoopSourcePerformRoutine};

    runLoopSource = CFRunLoopSourceCreate(NULL, 0, &context);
    commands = [[NSMutableArray alloc] init];

    return self;
}

- (void)addToCurrentRunLoop
{
    CFRunLoopRef runLoop = CFRunLoopGetCurrent();
    CFRunLoopAddSource(runLoop, runLoopSource, kCFRunLoopDefaultMode);
}
```

协调输入源的客户端

为了让添加的输入源有用，你需要维护它并从其他线程给它发送信号。输入源的主要工作就是将与输入源相关的线程置于休眠状态直到有事件发生。这就意味着程序中的要有其他线程知道该输入源信息并有办法与之通信。

通知客户端关于你输入源信息的方法之一就是当你的输入源开始安装到你的 run loop 上面后发送注册请求。你把输入源注册到任意数量的客户端，或者通过由代理将输入源注册到感兴趣的客户端那。列表 3-8 显示了应用委托定义的注册方法以及它在 RunLoopSource 对象的调度函数被调用时如何运行。该方法接收 RunLoopSource

提供的 RunLoopContext 对象，然后将其添加到它自己的源列表里面。另外，还显示了输入源从 run loop 移除时候的使用来取消注册例程。

Listing 3-8 Registering and removing an input source with the application delegate

```
- (void)registerSource:(RunLoopContext*)sourceInfo;
{
    [sourcesToPing addObject:sourceInfo];
}

- (void)removeSource:(RunLoopContext*)sourceInfo
{
    id    objToRemove = nil;

    for (RunLoopContext* context in sourcesToPing)
    {
        if ([context isEqual:sourceInfo])
        {
            objToRemove = context;
            break;
        }
    }

    if (objToRemove)
        [sourcesToPing removeObject:objToRemove];
}
```

注意：*该回调函数调用了列表 3-4 和列表 3-6 中描述的方法。*

通知输入源

在客户端发送数据到输入源后，它必须发信号通知源并且唤醒它的 run loop。发送信号给源可以让 run loop 知道该源已经做好处理消息的准备。而且因为信号发送时线程可能处于休眠状态，你必须总是显式的唤醒 run loop。如果不这样做的话会导致延迟处理输入源。

列表 3-9 显示了 RunLoopSource 对象的 fireCommandsOnRunLoop 方法。当客户端

准备好处理加入缓冲区的命令后会调用此方法。

Listing 3-9 Waking up the run loop

```
- (void)fireCommandsOnRunLoop:(CFRunLoopRef)runloop
{
    CFRunLoopSourceSignal(runLoopSource);
    CFRunLoopWakeUp(runloop);
}
```

注意：*你不应该试图通过自定义输入源处理一个 SIGHUP 或其他进程级别类型的信号。Core Foundation 唤醒 run loop 的函数不是信号安全的，不能在你的应用信号处理例程（signal handler routines）里面使用。关于更多信号处理例程，参阅 sigaction 主页。*

3.4.2 配置定时源

为了创建一个定时源，你所需要做只是创建一个定时器对象并把它调度到你的 run loop。Cocoa 程序中使用 NSTimer 类来创建一个新的定时器对象，而 Core Foundation 中使用 CFRunLoopTimerRef 不透明类型。本质上，NSTimer 类是 Core Foundation 的简单扩展，它提供了便利的特征，例如能使用相同的方法创建和调配定时器。

Cocoa 中可以使用以下 NSTimer 类方法来创建并调配一个定时器：

- scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:
- scheduledTimerWithTimeInterval:invocation:repeats:

上述方法创建了定时器并以默认模式把它们添加到当前线程的 run loop。你可以手工的创建 NSTimer 对象，并通过 NSRunLoop 的 addTimer:forMode:把它添加到 run loop。两种方法都做了相同的事，区别在于你对定时器配置的控制权。例如，如果你手工创建定时器并把它添加到 run loop，你可以选择要添加的模式而不使用默认模式。列表 3-10 显示了如何使用这这两种方法创建定时器。第一个定时器在初始化后 1 秒开始运行，此后每隔 0.1 秒运行。第二个定时器则在初始化后 0.2 秒开始运行，此后每隔 0.2 秒运行。

Listing 3-10 Creating and scheduling timers using NSTimer

```
NSRunLoop* myRunLoop = [NSRunLoop currentRunLoop];
```

```
// Create and schedule the first timer.
NSDate* futureDate = [NSDate dateWithTimeIntervalSinceNow:1.0];
NSTimer* myTimer = [[NSTimer alloc] initWithFireDate:futureDate
                    interval:0.1
                    target:self
                    selector:@selector(myDoFireTimer1:)
                    userInfo:nil
                    repeats:YES];
[myRunLoop addTimer:myTimer forMode:NSDefaultRunLoopMode];

// Create and schedule the second timer.
[NSTimer scheduledTimerWithTimeInterval:0.2
                    target:self
                    selector:@selector(myDoFireTimer2:)
                    userInfo:nil
                    repeats:YES];
```

列表 3-11 显示了使用 Core Foundation 函数来配置定时器的代码。尽管这个例子中并没有把任何用户定义的信息作为上下文结构，但是你可以使用这个上下文结构传递任何你想传递的信息给定时器。关于该上下文结构的内容的详细信息，参阅 CFRunLoopTimer Reference。

Listing 3-11 Creating and scheduling a timer using Core Foundation

```
CFRunLoopRef runLoop = CFRunLoopGetCurrent();
CFRunLoopTimerContext context = {0, NULL, NULL, NULL, NULL};
CFRunLoopTimerRef timer = CFRunLoopTimerCreate(kCFAllocatorDefault, 0.1, 0.3, 0, 0,
                                    &myCFTimerCallback, &context);

CFRunLoopAddTimer(runLoop, timer, kCFRunLoopCommonModes);
```

3.4.3 配置基于端口的输入源

Cocoa 和 Core Foundation 都提供了基于端口的对象用于线程或进程间的通信。以下部分显示如何使用几种不同类型的端口对象建立端口通信。

配置 NSMachPort 对象

为了和 NSMachPort 对象建立稳定的本地连接，你需要创建端口对象并将之加入相应的线程的 run loop。当运行辅助线程的时候，你传递端口对象到线程的主体入口点。辅助线程可以使用相同的端口对象将消息返回给原线程。

a） 实现主线程的代码

列表 3-12 显示了加载辅助线程的主线程代码。因为 Cocoa 框架执行许多配置端口和 run loop 相关的步骤，所以 lauchThread 方法比相应的 Core Foundation 版本（列表 3-17）要明显简短。然而两种方法的本质几乎是一样的，唯一的区别就是在 Cocoa 中直接发送 NSPort 对象，而不是发送本地端口名称。

Listing 3-12 Main thread launch method

```
- (void)launchThread
{
    NSPort* myPort = [NSMachPort port];
    if (myPort)
    {
        // This class handles incoming port messages.
        [myPort setDelegate:self];

        // Install the port as an input source on the current run loop.
        [[NSRunLoop currentRunLoop] addPort:myPort forMode:NSDefaultRunLoopMode];

        // Detach the thread. Let the worker release the port.
        [NSThread detachNewThreadSelector:@selector(LaunchThreadWithPort:)
              toTarget:[MyWorkerClass class] withObject:myPort];
    }
}
```

为了在你的线程间建立双向的通信，你需要让你的工作线程在签到的消息中发送自己的本地端口到主线程。主线程接收到签到消息后就可以知道辅助线程运行正常，并且提供了发送消息给辅助线程的方法。

列表 3-13 显示了主要线程的 handlePortMessage：方法。当由数据到达线程的本地端口时，该方法被调用。当签到消息到达时，此方法可以直接从辅助线程里面检索端口并保存下来以备后续使用。

Listing 3-13 Handling Mach port messages

```
#define kCheckinMessage 100


// Handle responses from the worker thread.
- (void)handlePortMessage:(NSPortMessage *)portMessage
{
    unsigned int message = [portMessage msgid];
    NSPort* distantPort = nil;


    if (message == kCheckinMessage)
    {
        // Get the worker thread's communications port.
        distantPort = [portMessage sendPort];


        // Retain and save the worker port for later use.
        [self storeDistantPort:distantPort];
    }
    else
    {
        // Handle other messages.
    }
}
```

b） 辅助线程的实现代码

对于辅助工作线程，你必须配置线程使用特定的端口以发送消息返回给主要线程。

列表 3-14 显示了如何设置工作线程的代码。创建了线程的自动释放池后，紧接着创建工作对象驱动线程运行。工作对象的 sendCheckinMessage：方法（如列表 3-15 所示）创建了工作线程的本地端口并发送签到消息回主线程。

Listing 3-14 Launching the worker thread using Mach ports

```
+(void)LaunchThreadWithPort:(id)inData
{
    NSAutoreleasePool*  pool = [[NSAutoreleasePool alloc] init];
```

```
    // Set up the connection between this thread and the main thread.
    NSPort* distantPort = (NSPort*)inData;

    MyWorkerClass*  workerObj = [[self alloc] init];
    [workerObj sendCheckinMessage:distantPort];
    [distantPort release];

    // Let the run loop process things.
    do
    {
        [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode
                           beforeDate:[NSDate distantFuture]];
    }
    while (![workerObj shouldExit]);

    [workerObj release];
    [pool release];
}
```

当使用 NSMachPort 时候，本地和远程线程可以使用相同的端口对象在线程间进行单边通信。换句话说，一个线程创建的本地端口对象成为另一个线程的远程端口对象。

列表 3-15 显示了辅助线程的签到例程，该方法为之后的通信设置自己的本地端口，然后发送签到消息给主线程。它使用 LaunchThreadWithPort：方法中收到的端口对象做为目标消息。

Listing 3-15 Sending the check-in message using Mach ports

```
// Worker thread check-in method
- (void)sendCheckinMessage:(NSPort*)outPort
{
    // Retain and save the remote port for future use.
    [self setRemotePort:outPort];
```

```
    // Create and configure the worker thread port.
    NSPort* myPort = [NSMachPort port];
    [myPort setDelegate:self];
    [[NSRunLoop currentRunLoop] addPort:myPort forMode:NSDefaultRunLoopMode];

    // Create the check-in message.
    NSPortMessage* messageObj = [[NSPortMessage alloc] initWithSendPort:outPort
                                    receivePort:myPort components:nil];

    if (messageObj)
    {
        // Finish configuring the message and send it immediately.
        [messageObj setMsgId:setMsgid:kCheckinMessage];
        [messageObj sendBeforeDate:[NSDate date]];
    }
}
```

配置 NSMessagePort 对象

为了和 NSMeaasgePort 的建立稳定的本地连接，你不能简单的在线程间传递端口对象。远程消息端口必须通过名字来获得。在 Cocoa 中这需要你给本地端口指定一个名字，并将名字传递到远程线程以便远程线程可以获得合适的端口对象用于通信。列表 3-16 显示端口创建，注册到你想要使用消息端口的进程。

Listing 3-16 Registering a message port

```
NSPort* localPort = [[NSMessagePort alloc] init];

// Configure the object and add it to the current run loop.
[localPort setDelegate:self];
[[NSRunLoop currentRunLoop] addPort:localPort forMode:NSDefaultRunLoopMode];

// Register the port using a specific name. The name must be unique.
NSString* localPortName = [NSString stringWithFormat:@"MyPortName"];
[[NSMessagePortNameServer sharedInstance] registerPort:localPort
```

```
                name:localPortName];
```

在 Core Foundation 中配置基于端口的源

这部分介绍了在 Core Foundation 中如何在程序主线程和工作线程间建立双通道通信。

列表 3-17 显示了程序主线程加载工作线程的代码。第一步是设置 CFMessagePortRef 不透明类型来监听工作线程的消息。工作线程需要端口的名称来建立连接，以便使字符串传递给工作线程的主入口函数。在当前的用户上下文中端口名必须是唯一的，否则可能在运行时造成冲突。

Listing 3-17 Attaching a Core Foundation message port to a new thread

```
#define kThreadStackSize       (8 *4096)


OSStatus MySpawnThread()
{
    // Create a local port for receiving responses.
    CFStringRef myPortName;
    CFMessagePortRef myPort;
    CFRunLoopSourceRef rlSource;
    CFMessagePortContext context = {0, NULL, NULL, NULL, NULL};
    Boolean shouldFreeInfo;


    // Create a string with the port name.
    myPortName = CFStringCreateWithFormat(NULL, NULL, CFSTR("com.myapp.MainThread"));


    // Create the port.
    myPort = CFMessagePortCreateLocal(NULL,
              myPortName,
              &MainThreadResponseHandler,
              &context,
              &shouldFreeInfo);


    if (myPort != NULL)
```

```
    {
        // The port was successfully created.
        // Now create a run loop source for it.
        rlSource = CFMessagePortCreateRunLoopSource(NULL, myPort, 0);

        if (rlSource)
        {
            // Add the source to the current run loop.
            CFRunLoopAddSource(CFRunLoopGetCurrent(), rlSource, kCFRunLoopDefaultMode);

            // Once installed, these can be freed.
            CFRelease(myPort);
            CFRelease(rlSource);
        }
    }

    // Create the thread and continue processing.
    MPTaskID       taskID;
    return(MPCreateTask(&ServerThreadEntryPoint,
                  (void*)myPortName,
                  kThreadStackSize,
                  NULL,
                  NULL,
                  NULL,
                  0,
                  &taskID));
}
```

端口建立而且线程启动后，主线程在等待线程签到时可以继续执行。当签到消息到达后，主线程使用 MainThreadResponseHandler 来分发消息，如列表 3-18 所示。这个函数提取工作线程的端口名，并创建用于未来通信的管道。

Listing 3-18 Receiving the checkin message

```
#define kCheckinMessage 100
```

```
// Main thread port message handler

CFDataRef MainThreadResponseHandler(CFMessagePortRef local,

                 SInt32 msgid,

                 CFDataRef data,

                 void* info)

{

   if (msgid == kCheckinMessage)

   {

      CFMessagePortRef messagePort;

      CFStringRef threadPortName;

      CFIndex bufferLength = CFDataGetLength(data);

      UInt8* buffer = CFAllocatorAllocate(NULL, bufferLength, 0);



      CFDataGetBytes(data, CFRangeMake(0, bufferLength), buffer);

      threadPortName = CFStringCreateWithBytes (NULL, buffer, bufferLength,
kCFStringEncodingASCII, FALSE);



      // You must obtain a remote message port by name.

      messagePort = CFMessagePortCreateRemote(NULL, (CFStringRef)threadPortName);



      if (messagePort)

      {

         // Retain and save the thread's comm port for future reference.

         AddPortToListOfActiveThreads(messagePort);



         // Since the port is retained by the previous function, release

         // it here.

         CFRelease(messagePort);

      }



      // Clean up.

      CFRelease(threadPortName);
```

```
        CFAllocatorDeallocate(NULL, buffer);
    }
    else
    {
        // Process other messages.
    }

    return NULL;
}
```

主线程配置好后，剩下的唯一事情是让新创建的工作线程创建自己的端口然后签到。列表 3-19 显示了工作线程的入口函数。函数获取了主线程的端口名并使用它来创建和主线程的远程连接。然后这个函数创建自己的本地端口号，安装到线程的 run loop，最后连同本地端口名称一起发回主线程签到。

Listing 3-19 Setting up the thread structures

```
OSStatus ServerThreadEntryPoint(void* param)
{
    // Create the remote port to the main thread.
    CFMessagePortRef mainThreadPort;
    CFStringRef portName = (CFStringRef)param;

    mainThreadPort = CFMessagePortCreateRemote(NULL, portName);

    // Free the string that was passed in param.
    CFRelease(portName);

    // Create a port for the worker thread.
    CFStringRef myPortName = CFStringCreateWithFormat(NULL, NULL,
CFSTR("com.MyApp.Thread-%d"), MPCurrentTaskID());

    // Store the port in this thread's context info for later reference.
    CFMessagePortContext context = {0, mainThreadPort, NULL, NULL, NULL};
    Boolean shouldFreeInfo;
    Boolean shouldAbort = TRUE;
```

```
CFMessagePortRef myPort = CFMessagePortCreateLocal(NULL,
          myPortName,
          &ProcessClientRequest,
          &context,
          &shouldFreeInfo);

if (shouldFreeInfo)
{
   // Couldn't create a local port, so kill the thread.
   MPExit(0);
}

CFRunLoopSourceRef rlSource = CFMessagePortCreateRunLoopSource(NULL, myPort, 0);
if (!rlSource)
{
   // Couldn't create a local port, so kill the thread.
   MPExit(0);
}

// Add the source to the current run loop.
CFRunLoopAddSource(CFRunLoopGetCurrent(), rlSource, kCFRunLoopDefaultMode);

// Once installed, these can be freed.
CFRelease(myPort);
CFRelease(rlSource);

// Package up the port name and send the check-in message.
CFDataRef returnData = nil;
CFDataRef outData;
CFIndex stringLength = CFStringGetLength(myPortName);
UInt8* buffer = CFAllocatorAllocate(NULL, stringLength, 0);
```

```
    CFStringGetBytes(myPortName,
                CFRangeMake(0,stringLength),
                kCFStringEncodingASCII,
                0,
                FALSE,
                buffer,
                stringLength,
                NULL);

    outData = CFDataCreate(NULL, buffer, stringLength);

    CFMessagePortSendRequest(mainThreadPort, kCheckinMessage, outData, 0.1, 0.0, NULL,
NULL);

    // Clean up thread data structures.
    CFRelease(outData);
    CFAllocatorDeallocate(NULL, buffer);

    // Enter the run loop.
    CFRunLoopRun();
}
```

一旦线程进入了它的 run loop，所有发送到线程端口的事件都会由 ProcessClientRequest 函数处理。函数的具体实现依赖于线程的工作方式，这里就不举例了。

第四章　线程同步

应用程序里面多个线程的存在引发了多个执行线程安全访问资源的潜在问题。两个线程同时修改同一资源有可能以意想不到的方式互相干扰。比如，一个线程可能覆盖其他线程改动的地方，或让应用程序进入一个未知的潜在无效状态。如果你幸运的话，受损的资源可能会导致明显的性能问题或崩溃，这样比较容易跟踪并修复它。然而如果你不走运，资源受损可能导致微妙的错误，这些错误不会立即显现出来，而是很久之后才出现，或者导致其他可能需要一个底层的编码来显著修复的错误。

但涉及到线程安全时，一个好的设计是最好的保护。避免共享资源，并尽量减少线程间的相互作用，这样可以让它们减少互相的干扰。但是一个完全无干扰的设计是不可能的。在线程必须交互的情况下，你需要使用同步工具，来确保当它们交互的时候是安全的。

Mac OS X 和 iOS 提供了你可以使用的多个同步工具，从提供互斥访问你程序的有序的事件的工具等。以下个部分介绍了这些工具和如何在代码中使用他们来影响安全的访问程序的资源。

4.1　同步工具

为了防止不同线程意外修改数据，你可以设计你的程序没有同步问题，或你也可以使用同步工具。尽管完全避免出现同步问题相对更好一点，但是几乎总是无法实现。以下个部分介绍了你可以使用的同步工具的基本类别。

4.1.1 原子操作

原子操作是同步的一个简单的形式，它处理简单的数据类型。原子操作的优势是它们不妨碍竞争的线程。对于简单的操作，比如递增一个计数器，原子操作比使用锁具有更高的性能优势。

Mac OS X 和 iOS 包含了许多在 32 位和 64 位执行基本的数学和逻辑运算的操作。这些操作都使用了原子版本来操作比较和交换，测试和设置，测试和清理等。查看支持原子操作的列表，参阅/user/include/libkern/OSAtomic.h 头文件和参见 atomic 主页。

4.1.2 内存屏障和 Volatile 变量

为了达到最佳性能，编译器通常会对汇编基本的指令进行重新排序来尽可能保持处理器的指令流水线。作为优化的一部分，编译器有可能对访问主内存的指令，如果它认为这有可能产生不正确的数据时，将会对指令进行重新排序。不幸的是，靠编译器检测到所有可能内存依赖的操作几乎总是不太可能的。如果看似独立的变量实际上是相互影响，那么编译器优化有可能把这些变量更新位错误的顺序，导致潜在不不正确结果。

内存屏障（memory barrier）是一个使用来确保内存操作按照正确的顺序工作的非阻塞的同步工具。内存屏障的作用就像一个栅栏，迫使处理器来完成位于障碍前面的任何加载和存储操作，才允许它执行位于屏障之后的加载和存储操作。内存屏障同样使用来确保一个线程（但对另外一个线程可见）的内存操作总是按照预定的顺序完成。如果在这些地方缺少内存屏障有可能让其他线程看到看似不可能的结果（比如，内存屏障的维基百科条目）。为了使用一个内存屏障，你只要在你代码里面需要的地方简单的调用 OSMemoryBarrier 函数。

Volatile 变量适用于独立变量的另一个内存限制类型。编译器优化代码通过加载这些变量的值进入寄存器。对于本地变量，这通常不会有什么问题。但是如果一个变量对另外一个线程可见，那么这种优化可能会阻止其他线程发现变量的任何变化。在变量之前加上关键字 volatile 可以强制编译器每次使用变量的时候都从内存里面加载。如果一个变量的值随时可能给编译器无法检测的外部源更改，那么你可以把该变量声明为 volatile 变量。

因为内存屏障和 volatile 变量降低了编译器可执行的优化，因此你应该谨慎使用它们，只在有需要的地方时候，以确保正确性。关于更多使用内存屏障的信息，参阅 OSMemoryBarrier 主页。

4.1.3 锁

锁是最常用的同步工具。你可以是使用锁来保护临界区(critical section)，这些代码段在同一个时间只能允许被一个线程访问。比如，一个临界区可能会操作一个特定的数据结构，或使用了每次只能一个客户端访问的资源。

表 4-1 列出了程序最常使用的锁。Mac OS X 和 iOS 提供了这些锁里面大部分类型

的实现，但是并不是全部实现。对于不支持的锁类型，说明列解析了为什么这些锁不能直接在平台上面实现的原因。

Table 4-1 Lock types

Lock	Description
Mutex [互斥锁]	A mutually exclusive (or **mutex**) lock acts as a protective barrier around a resource. A mutex is a type of semaphore that grants access to only one thread at a time. If a mutex is in use and another thread tries to acquire it, that thread blocks until the mutex is released by its original holder. If multiple threads compete for the same mutex, only one at a time is allowed access to it.
Recursive lock [递归锁]	A recursive lock is a variant on the mutex lock. A recursive lock allows a single thread to acquire the lock multiple times before releasing it. Other threads remain blocked until the owner of the lock releases the lock the same number of times it acquired it. Recursive locks are used during recursive iterations primarily but may also be used in cases where multiple methods each need to acquire the lock separately.
Read-write lock [读写锁]	A read-write lock is also referred to as a shared-exclusive lock. This type of lock is typically used in larger-scale operations and can significantly improve performance if the protected data structure is read frequently and modified only occasionally. During normal operation, multiple readers can access the data structure simultaneously. When a thread wants to write to the structure, though, it blocks until all readers release the lock, at which point it acquires the lock and can update the structure. While a writing thread is waiting for the lock, new reader threads block until the writing thread is finished. The system supports read-write locks using POSIX threads only. For more information on how to use these locks, see the `pthread` man page.
Distributed lock [分布锁]	A distributed lock provides mutually exclusive access at the process level. Unlike a true mutex, a distributed lock does not block a process or prevent it from running. It simply reports when the lock is busy and lets the process decide how to proceed.
Spin lock [自旋锁]	A spin lock polls its lock condition repeatedly until that condition becomes true. Spin locks are most often used on multiprocessor systems where the expected wait time for a lock is small. In these situations, it is often more efficient to poll than to block the thread, which involves a context switch and the updating of thread data structures. The system does not provide any implementations of spin locks because of their polling nature, but you can easily implement them in specific situations. For information on implementing spin locks in the kernel, see *Kernel Programming Guide*.
Double-checked lock [双重检查锁]	A double-checked lock is an attempt to reduce the overhead of taking a lock by testing the locking criteria prior to taking the lock. Because double-checked locks are potentially unsafe, the system does not provide explicit support for them and their use is discouraged.[注意系统不显式支持该锁类型]

注意：*大部分锁类型都合并了内存屏障来确保在进入临界区之前它前面的加载和存储指令都已经完成。*

关于如何使用锁的信息，参阅”使用锁”部分。

4.1.4 条件

条件是**信号量**的另外一个形式，它允许在条件为真的时候线程间互相发送信号。条件通常被使用来说明资源可用性，或用来确保任务以特定的顺序执行。当一个线程测试一个条件时，它会被阻塞直到条件为真。它会一直阻塞直到其他线程显式的修改

信号量的状态。条件和互斥锁(mutex lock)的区别在于多个线程被允许同时访问一个条件。条件更多是允许不同线程根据一些指定的标准通过的守门人。

一个方式是你使用条件来管理挂起事件的池。事件队列可能使用条件变量来给等待线程发送信号，此时它们在事件队列中的时候。如果一个事件到达时，队列将给条件发送合适信号。如果一个线程已经处于等待，它会被唤醒，届时它将会取出事件并处理它。如果两个事件到达队列的时间大致相同，队列将会发送两次信号唤醒两个线程。

系统通过几个不同的技术来支持条件。然而正确实现条件需要仔细编写代码，因此你应该在你自己代码中使用条件之前查看”使用条件”部分的例子。

4.1.5 执行 Selector 例程

Cocoa 程序包含了一个在一个线程以同步的方式传递消息的方便方法。NSObject 类声明方法来在应用的一个活动线程上面执行 selector 的方法。这些方法允许你的线程以异步的方式来传递消息，以确保它们在同一个线程上面执行是同步的。比如，你可以通过执行 selector 消息来把一个从你分布计算的结果传递给你的应用的主线程或其他目标线程。每个执行 selector 的请求都会被放入一个目标线程的 run loop 的队列里面，然后请求会按照它们到达的顺序被目标线程有序的处理。

关于执行 selector 例程的总结和更多关于如何使用它们的信息，参阅 Cocoa 执行 Selector 源。

4.2 同步的成本和性能

同步帮助确保你代码的正确性，但同时将会牺牲部分性能。甚至在无争议的情况下，同步工具的使用将在后面介绍。锁和原子操作通常包含了内存屏障和内核级别同步的使用来确保代码正确被保护。如果，发生锁的争夺，你的线程有可能进入阻塞，在体验上会产生更大的迟延。

表 4-2 列出了在无争议情况下使用互斥锁和原子操作的近似的相关成本。这些测试的平均值是使用了上千的样本分析出的结果。随着线程创建时间的推移，互斥采集时间（即使在无争议情况下）可能相差也很大，这依赖于进程的加载，计算机的处理速度和系统和程序现有可用的内存。

Table 4-2 Mutex and atomic operation costs

Item	Approximate cost	Notes
Mutex acquisition time	Approximately 0.2 microseconds [0.2 微秒]	This is the lock acquisition time in an uncontested case. If the lock is held by another thread, the acquisition time can be much greater. The figures were determined by analyzing the mean and median values generated during mutex acquisition on an Intel-based iMac with a 2 GHz Core Duo processor and 1 GB of RAM running Mac OS X v10.5.
Atomic compare-and-swap	Approximately 0.05 microseconds [0.05 微秒]	This is the compare-and-swap time in an uncontested case. The figures were determined by analyzing the mean and median values for the operation and were generated on an Intel-based iMac with a 2 GHz Core Duo processor and 1 GB of RAM running Mac OS X v10.5.

当设计你的并发任务时，正确性是最重要的因素，但是也要考虑性能因素。代码在多个线程下面正确执行，但比相同代码在当线程执行慢，这是难以改善的。如果你是改造已有的单线程应用，你应该始终给关键任务的性能设置测量基线。当增加额外线程后，对相同的任务你应该采取新的测量方法并比较多线程和单线程情况下的性能状况。在改变代码之后，线程并没有提高性能，你应该需要重新考虑具体的实现或同时使用线程。

关于性能的信息和收集指标的工具，参阅 Performance Overview。关于锁原子成本的特定信息，参阅”线程成本”部分。

4.3 线程安全和信号量

当涉及到多线程应用程序时，没有什么比处理信号量更令人恐惧和困惑的了。信号量是底层 BSD 机制，它可以用来传递信息给进程或以某种方式操纵它。一些应用程序使用信号量来检测特定事件，比如子进程的消亡。系统使用信号量来终止失控进程，和作为其他类型的通信消息。

使用信号量的问题并不是你要做什么，而是当你程序是多线程的时候它们的行为。在当线程应用程序里面，所有的信号量处理都在主线程进行。在多线程应用程序里面，信号量被传递到恰好运行的线程，而不依赖于特定的硬件错误（比如非法指令）。如果多个线程同时运行，信号量被传递到任何一个系统挑选的线程。换而言之，信号量可以传递给你应用的任何线程。

在你应用程序里面实现信号量处理的第一条规则是避免假设任一线程处理信号量。如果一个指定的线程想要处理给定的信号，你需要通过某些方法来通知该线程信

号何时到达。你不能只是假设该线程的一个信号处理例程的安装会导致信号被传递到同一线程里面。

关于更多信号量的信息和信号量处理例程的安装信息，参见 signal 和 sigaction 主页。

4.4 线程安全设计的技巧

同步工具是让你代码安全的有用方法，但是它们并非灵丹妙药。使用太多锁和其他同步的类型原语和非多线程相比明显会降低你应用的线程性能。在性能和安全之间寻找平衡是一门需要经验的艺术。以下各部分提供帮助你为你应用选择合适的同步级别的技巧。

4.4.1 完全避免同步

对于你新的项目，甚至已有项目，设计你的代码和数据结构来避免使用同步是一个很好的解决办法。虽然锁和其他类型同步工具很有用，但是它们会影响任何应用的性能。而且如果整体设计导致特定资源的高竞争，你的线程可能需要等待更长时间。

实现并发最好的方法是减少你并发任务之间的交互和相互依赖。如果每个任务在它自己的数据集上面操作，那它不需要使用锁来保护这些数据。甚至如果两个任务共享一个普通数据集，你可以查看分区方法，它们设置或提供拷贝每一项任务的方法。当然，拷贝数据集本身也需要成本，所以在你做出决定前，你需要权衡这些成本和使用同步工具造成的成本那个更可以接受。

4.4.2 了解同步的限制

同步工具只有当它们被用在应用程序中的所有线程是一致时才是有效的。如果你创建了互斥锁来限制特定资源的访问，你所有线程都必须在试图操纵资源前获得同一互斥锁。如果不这样做导致破坏一个互斥锁提供的保护，这是编程的错误。

4.4.3 注意对代码正确性的威胁

当你使用锁和内存屏障时，你应该总是小心的把它们放在你代码正确的地方。即使有条件的锁（似乎很好放置）也可能会让你产生一个虚假的安全感。以下一系列例

子试图通过指出看似无害的代码的漏洞来举例说明该问题。其基本前提是你有一个可变的数组，它包含一组不可变的对象集。假设你想要调用数组中第一个对象的方法。你可能会做类似下面那样的代码：

```
NSLock* arrayLock = GetArrayLock();
NSMutableArray* myArray = GetSharedArray();
id anObject;

[arrayLock lock];
anObject = [myArray objectAtIndex:0];
[arrayLock unlock];

[anObject doSomething];
```

因为数组是可变的，所有数组周围的锁防止其他线程修改该数组直到你获得了想要的对象。而且因为对象限制它们本身是不可更改的，所以在调用对象的 doSomething 方法周围不需要锁。

但是上面显式的例子有一个问题。如果当你释放该锁，而在你有机会执行 doSomething 方法前其他线程到来并从数组中删除所有对象，那会发生什么呢？对于没有使用垃圾回收的应用程序，你代码用户的对象可能已经释放了，让 anObject 对象指向一个非法的内存地址。了修正该问题，你可能决定简单的重新安排你的代码，让它在调用 doSomething 之后才释放锁，如下所示：

```
NSLock* arrayLock = GetArrayLock();
NSMutableArray* myArray = GetSharedArray();
id anObject;

[arrayLock lock];
anObject = [myArray objectAtIndex:0];
[anObject doSomething];
[arrayLock unlock];
```

通过把 doSomething 的调用移到锁的内部，你的代码可以保证该方法被调用的时候该对象还是有效的。不幸的是，如果 doSomething 方法需要耗费很长的时间，这有可能导致你的代码保持拥有该锁很长时间，这会产生一个性能瓶颈。

该代码的问题不是关键区域定义不清，而是实际问题是不可理解的。真正的问题是由其他线程引发的内存管理的问题。因为它可以被其他线程释放，最好的解决办法是在释放锁之前 retain anObject。该解决方案涉及对象被释放，并没有引发一个强制的性能损失。

```
NSLock* arrayLock = GetArrayLock();
NSMutableArray* myArray = GetSharedArray();
id anObject;

[arrayLock lock];
anObject = [myArray objectAtIndex:0];
[anObject retain];
[arrayLock unlock];

[anObject doSomething];
[anObject release];
```

尽管前面的例子非常简单，它们说明了非常重要的一点。当它涉及到正确性时，你需要考虑不仅仅是问题的表面。内存管理和其他影响你设计的因子都有可能因为出现多个线程而受到影响，所以你必须考虑从上到下考虑这些问题。此外，你应该在涉及安全的时候假设编译器总是出现最坏的情况。这种意识和警惕性，可以帮你避免潜在的问题，并确保你的代码运行正确。

关于更多介绍如何让你应用程序安全的额外例子，参阅 Technical Note TN2059:”Using Collection Classes Safely in Multithreaded Application”。

4.4.4 当心死锁（Deadlocks）和活锁(Livelocks)

任何时候线程试图同时获得多于一个锁，都有可能引发潜在的死锁。当两个不同的线程分别保持一个锁（而该锁是另外一个线程需要的）又试图获得另外线程保持的锁时就会发生死锁。结果是每个线程都会进入持久性阻塞状态，因为它永远不可能获得另外那个锁。

一个活锁和死锁类似，当两个线程竞争同一个资源的时候就可能发生活锁。在发生活锁的情况里，一个线程放弃它的第一个锁并试图获得第二个锁。一旦它获得第二

个锁，它返回并试图再次获得一个锁。线程就会被锁起来，因为它花费所有的时间来释放一个锁，并试图获取其他锁，而不做实际的工作。

避免死锁和活锁的最好方法是同一个时间只拥有一个锁。如果你必须在同一时间获取多于一个锁，你应该确保其他线程没有做类似的事情。

4.4.5 正确使用 Volatile 变量

如果你已经使用了一个互斥锁来保护一个代码段，不要自动假设你需要使用关键词 volatile 来保护该代码段的重要的变量。一个互斥锁包含了内存屏障来确保加载和存储操作是按照正确顺序的。在一个临界区添加关键字 volatile 到变量上面会强制每次访问该变量的时候都要从内存里面从加载。这两种同步技巧的组合使用在一些特定区域是必须的，但是同样会导致显著的性能损失。如果单独使用互斥锁已经可以保护变量，那么忽略关键字 volatile。

为了避免使用互斥锁而不使用 volatile 变量同样很重要。通常情况下，互斥锁和其他同步机制是比 volatile 变量更好的方式来保护数据结构的完整性。关键字 volatile 只是确保从内存加载变量而不是使用寄存器里面的变量。它不保证你代码访问变量是正确的。

4.5 使用原子操作

非阻塞同步的方式是用来执行某些类型的操作而避免扩展使用锁。尽管锁是同步两个线程的很好方式，获取一个锁是一个很昂贵的操作，即使在无竞争的状态下。相比，许多原子操作花费很少的时间来完成操作也可以达到和锁一样的效果。

原子操作可以让你在 32 位或 64 位的处理器上面执行简单的数学和逻辑的运算操作。这些操作依赖于特定的硬件设施（和可选的内存屏障）来保证给定的操作在影响内存再次访问的时候已经完成。在多线程情况下，你应该总是使用原子操作，它和内存屏障组合使用来保证多个线程间正确的同步内存。

表 4-3 列出了可用的原子运算和本地操作和相应的函数名。这些函数声明在 /usr/include/libkern/OSAtomic.h 头文件里面，在那里你也可以找到完整的语法。这些函数的 64-位版本只能在 64 位的进程里面使用。

Table 4-3 Atomic math and logic operations

Operation	Function name	Description

Add	OSAtomicAdd32 OSAtomicAdd32Barrier OSAtomicAdd64 OSAtomicAdd64Barrier	Adds two integer values together and stores the result in one of the specified variables.
Increment	OSAtomicIncrement32 OSAtomicIncrement32Barrier OSAtomicIncrement64 OSAtomicIncrement64Barrier	Increments the specified integer value by 1.
Decrement	OSAtomicDecrement32 OSAtomicDecrement32Barrier OSAtomicDecrement64 OSAtomicDecrement64Barrier	Decrements the specified integer value by 1.
Logical OR	OSAtomicOr32 OSAtomicOr32Barrier	Performs a logical OR between the specified 32-bit value and a 32-bit mask.
Logical AND	OSAtomicAnd32 OSAtomicAnd32Barrier	Performs a logical AND between the specified 32-bit value and a 32-bit mask.
Logical XOR	OSAtomicXor32 OSAtomicXor32Barrier	Performs a logical XOR between the specified 32-bit value and a 32-bit mask.
Compare and swap	OSAtomicCompareAndSwap32 OSAtomicCompareAndSwap32Barrier OSAtomicCompareAndSwap64 OSAtomicCompareAndSwap64Barrier OSAtomicCompareAndSwapPtr OSAtomicCompareAndSwapPtrBarrier OSAtomicCompareAndSwapInt OSAtomicCompareAndSwapIntBarrier OSAtomicCompareAndSwapLong OSAtomicCompareAndSwapLongBarrier	Compares a variable against the specified old value. If the two values are equal, this function assigns the specified new value to the variable; otherwise, it does nothing. The comparison and assignment are done as one atomic operation and the function returns a Boolean value indicating whether the swap actually occurred.
Test and set	OSAtomicTestAndSet OSAtomicTestAndSetBarrier	Tests a bit in the specified variable, sets that bit to 1, and returns the value of the old bit as a Boolean value. Bits are tested according to the formula `(0x80 >> (n & 7))` of byte `((char*)address + (n >> 3))` where `n` is the bit number and `address` is a pointer to the variable. This formula effectively breaks up the variable into 8-bit sized chunks and orders the bits in each chunk in reverse. For example, to test the lowest-order bit (bit 0) of a 32-bit integer, you would actually specify 7 for the bit number; similarly, to test the highest order bit (bit 32), you would specify 24 for the bit number.
Test and clear	OSAtomicTestAndClear OSAtomicTestAndClearBarrier	Tests a bit in the specified variable, sets that bit to 0, and returns the value of the old bit as a Boolean value. Bits are tested according to the formula `(0x80 >> (n & 7))` of byte `((char*)address + (n >> 3))` where `n` is the bit number and `address` is a pointer to the variable. This formula effectively breaks up the variable into 8-bit sized chunks and orders the bits in each chunk in reverse. For example, to test the lowest-order bit (bit 0) of a 32-bit integer, you would actually specify 7 for the bit number; similarly, to test the highest order bit (bit 32), you would specify 24 for the bit number.

大部分原子函数的行为是相对简单的并应该是你想要的。然而列表 4-1 显式了测试-设置和比较-交换操作的原子行为，它们相对复杂一点。OSAtomicTestAndSet 第一次调用展示了如何对一个整形值进行位运算操作，而它的结果和你预期的有差异。最后两次调用 OSAtomicCompareAndSwap32 显式它的行为。所有情况下，这些函数都是无竞争的下调用的，此时没有其他线程试图操作这些值。

Listing 4-1 Performing atomic operations

```
int32_t  theValue = 0;
OSAtomicTestAndSet(0, &theValue);
// theValue is now 128.

theValue = 0;
OSAtomicTestAndSet(7, &theValue);
// theValue is now 1.

theValue = 0;
OSAtomicTestAndSet(15, &theValue)
// theValue is now 256.

OSAtomicCompareAndSwap32(256, 512, &theValue);
// theValue is now 512.

OSAtomicCompareAndSwap32(256, 1024, &theValue);
// theValue is still 512.
```

关于原子操作的更多信息，参见 atomic 的主页和 /usr/include/libkern/OSAtomic.h 头文件。

4.6 使用锁

锁是线程编程同步工具的基础。锁可以让你很容易保护代码中一大块区域以便你可以确保代码的正确性。Mac OS X 和 iOS 都位所有类型的应用程序提供了互斥锁，而 Foundation 框架定义一些特殊情况下互斥锁的额外变种。以下个部分显式了如何使用这些锁的类型。

4.6.1 使用 POSIX 互斥锁

POSIX 互斥锁在很多程序里面很容易使用。为了新建一个互斥锁，你声明并初始化一个 pthread_mutex_t 的结构。为了锁住和解锁一个互斥锁，你可以使用 pthread_mutex_lock 和 pthread_mutex_unlock 函数。列表 4-2 显式了要初始化并使用一个 POSIX 线程的互斥锁的基础代码。当你用完一个锁之后，只要简单的调用 pthread_mutex_destroy 来释放该锁的数据结构。

Listing 4-2 Using a mutex lock

```
pthread mutex t mutex;

void MyInitFunction()

{

    pthread mutex init(&mutex, NULL);

}



void MyLockingFunction()

{

    pthread mutex lock(&mutex);

    // Do work.

    pthread mutex unlock(&mutex);

}
```

注意：*上面的代码只是简单的显式了使用一个 POSIX 线程互斥锁的步骤。你自己的代码应该检查这些函数返回的错误码，并适当的处理它们。*

4.6.2 使用 NSLock 类

在 Cocoa 程序中 NSLock 中实现了一个简单的互斥锁。所有锁（包括 NSLock）的接口实际上都是通过 NSLocking 协议定义的，它定义了 lock 和 unlock 方法。你使用这些方法来获取和释放该锁。

除了标准的锁行为，NSLock 类还增加了 tryLock 和 lockBeforeDate:方法。方法 tryLock 试图获取一个锁，但是如果锁不可用的时候，它不会阻塞线程。相反，它只是返回 NO。而 lockBeforeDate:方法试图获取一个锁，但是如果锁没有在规定的时间内被获得，它会让线程从阻塞状态变为非阻塞状态（或者返回 NO）。

下面的例子显式了你可以是 NSLock 对象来协助更新一个可视化显式，它的数据结构被多个线程计算。如果线程没有立即获的锁，它只是简单的继续计算直到它可以获得锁再更新显式。

```
BOOL moreToDo = YES;
NSLock *theLock = [[NSLock alloc] init];
...
while (moreToDo) {
    /* Do another increment of calculation */
    /* until there's no more to do. */
    if ([theLock tryLock]) {
        /* Update display used by all threads. */
        [theLock unlock];
    }
}
```

4.6.3 使用@synchronized 指令

@synchronized 指令是在 Objective-C 代码中创建一个互斥锁非常方便的方法。@synchronized 指令做和其他互斥锁一样的工作（它防止不同的线程在同一时间获取同一个锁）。然而在这种情况下，你不需要直接创建一个互斥锁或锁对象。相反，你只需要简单的使用 Objective-C 对象作为锁的令牌，如下面例子所示：

```
- (void)myMethod:(id)anObj
{
    @synchronized(anObj)
    {
        // Everything between the braces is protected by the @synchronized directive.
    }
}
```

创建给@synchronized 指令的对象是一个用来区别保护块的唯一标示符。如果你在两个不同的线程里面执行上述方法，每次在一个线程传递了一个不同的对象给 anObj 参数，那么每次都将会拥有它的锁，并持续处理，中间不被其他线程阻塞。然而，如果你传递的是同一个对象，那么多个线程中的一个线程会首先获得该锁，而其

他线程将会被阻塞直到第一个线程完成它的临界区。

作为一种预防措施，@synchronized 块隐式的添加一个异常处理例程来保护代码。该处理例程会在异常抛出的时候自动的释放互斥锁。这意味着为了使用@synchronized 指令，你必须在你的代码中启用异常处理。了如果你不想让隐式的异常处理例程带来额外的开销，你应该考虑使用锁的类。

关于更多@synchronized 指令的信息，参阅 The Objective-C Programming Language。

4.6.4 使用其他 Cocoa 锁

以下个部分描述了使用 Cocoa 其他类型的锁。

使用 NSRecursiveLock 对象

NSRecursiveLock 类定义的锁可以在同一线程多次获得，而不会造成死锁。一个递归锁会跟踪它被多少次成功获得了。每次成功的获得该锁都必须平衡调用锁住和解锁的操作。只有所有的锁住和解锁操作都平衡的时候，锁才真正被释放给其他线程获得。

正如它名字所言，这种类型的锁通常被用在一个递归函数里面来防止递归造成阻塞线程。你可以类似的在非递归的情况下使用他来调用函数，这些函数的语义要求它们使用锁。以下是一个简单递归函数，它在递归中获取锁。如果你不在该代码里使用 NSRecursiveLock 对象，当函数被再次调用的时候线程将会出现死锁。

```
NSRecursiveLock *theLock = [[NSRecursiveLock alloc] init];


void MyRecursiveFunction(int value)
{
    [theLock lock];
    if (value != 0)
    {
        --value;
        MyRecursiveFunction(value);
    }
```

```
    [theLock unlock];
}

MyRecursiveFunction(5);
```

注意：*因为一个递归锁不会被释放直到所有锁的调用平衡使用了解锁操作，所以你必须仔细权衡是否决定使用锁对性能的潜在影响。长时间持有一个锁将会导致其他线程阻塞直到递归完成。如果你可以重写你的代码来消除递归或消除使用一个递归锁，你可能会获得更好的性能。*

使用 NSConditionLock 对象

NSConditionLock 对象定义了一个互斥锁，可以使用特定值来锁住和解锁。不要把该类型的锁和条件（参见“条件”部分）混淆了。它的行为和条件有点类似，但是它们的实现非常不同。

通常，当多线程需要以特定的顺序来执行任务的时候，你可以使用一个 NSConditionLock 对象，比如当一个线程生产数据，而另外一个线程消费数据。生产者执行时，消费者使用由你程序指定的条件来获取锁（条件本身是一个你定义的整形值）。当生产者完成时，它会解锁该锁并设置锁的条件为合适的整形值来唤醒消费者线程，之后消费线程继续处理数据。

NSConditionLock 的锁住和解锁方法可以任意组合使用。比如，你可以使用 unlockWithCondition:和 lock 消息，或使用 lockWhenCondition:和 unlock 消息。当然，后面的组合可以解锁一个锁但是可能没有释放任何等待某特定条件值的线程。

下面的例子显示了生产者-消费者问题如何使用条件锁来处理。想象一个应用程序包含一个数据的队列。一个生产者线程把数据添加到队列，而消费者线程从队列中取出数据。生产者不需要等待特定的条件，但是它必须等待锁可用以便它可以安全的把数据添加到队列。

```
id condLock = [[NSConditionLock alloc] initWithCondition:NO_DATA];

while(true)
{
    [condLock lock];
    /* Add data to the queue. */
```

```
    [condLock unlockWithCondition:HAS_DATA];
}
```

因为初始化条件锁的值为 NO_DATA，生产者线程在初始化的时候可以毫无问题的获取该锁。它会添加队列数据，并把条件设置为 HAS_DATA。在随后的迭代中，生产者线程可以把到达的数据添加到队列，无论队列是否为空或依然有数据。唯一让它进入阻塞的情况是当一个消费者线程充队列取出数据的时候。

因为消费者线程必须要有数据来处理，它会使用一个特定的条件来等待队列。当生产者把数据放入队列时，消费者线程被唤醒并获取它的锁。它可以从队列中取出数据，并更新队列的状态。下列代码显示了消费者线程处理循环的基本结构。

```
while (true)
{
    [condLock lockWhenCondition:HAS_DATA];
    /* Remove data from the queue. */
    [condLock unlockWithCondition:(isEmpty ? NO_DATA : HAS_DATA)];

    // Process the data locally.
}
```

使用 NSDistributedLock 对象

NSDistributedLock 类可以被多台主机上的多个应用程序使用来限制对某些共享资源的访问，比如一个文件。锁本身是一个高效的互斥锁，它使用文件系统项目来实现，比如一个文件或目录。对于一个可用的 NSDistributedLock 对象，锁必须由所有使用它的程序写入。这通常意味着把它放在文件系统，该文件系统可以被所有运行在计算机上面的应用程序访问。

不像其他类型的锁，NSDistributedLock 并没有实现 NSLocking 协议，所有它没有 lock 方法。一个 lock 方法将会阻塞线程的执行，并要求系统以预定的速度轮询锁。以其在你的代码中实现这种约束，NSDistributedLock 提供了一个 tryLock 方法，并让你决定是否轮询。

因为它使用文件系统来实现，一个 NSDistributedLock 对象不会被释放除非它的拥有者显式的释放它。如果你的程序在用户一个分布锁的时候崩溃了，其他客户端简

无法访问该受保护的资源。在这种情况下，你可以使用 breadLock 方法来打破现存的锁以便你可以获取它。但是通常应该避免打破锁，除非你确定拥有进程已经死亡并不可能再释放该锁。

和其他类型的锁一样，当你使用 NSDistributedLock 对象时，你可以通过调用 unlock 方法来释放它。

4.7 使用条件

条件是一个特殊类型的锁，你可以使用它来同步操作必须处理的顺序。它们和互斥锁有微妙的不同。一个线程等待条件会一直处于阻塞状态直到条件获得其他线程显式发出的信号。

由于微妙之处包含在操作系统实现上，条件锁被允许返回伪成功，即使实际上它们并没有被你的代码告知。为了避免这些伪信号操作的问题，你应该总是在你的条件锁里面使用一个断言。该断言是一个更好的方法来确定是否安全让你的线程处理。条件简单的让你的线程保持休眠直到断言被发送信号的线程设置了。

以下部分介绍了如何在你的代码中使用条件。

4.7.1 使用 NSCondition 类

NSCondition 类提供了和 POSIX 条件相同的语义，但是它把锁和条件数据结构封装在一个单一对象里面。结果是一个你可以像互斥锁那样使用的对象，然后等待特定条件。

列表 4-3 显示了一个代码片段，它展示了为等待一个 NSCondition 对象的事件序列。cocaoCondition 变量包含了一个 NSCondition 对象，而 timeToDoWork 变量是一个整形，它在其他线程里面发送条件信号时立即递增。

Listing 4-3 Using a Cocoa condition

```
[cocoaCondition lock];
while (timeToDoWork <= 0)
    [cocoaCondition wait];

timeToDoWork--;
```

```
// Do real work here.


[cocoaCondition unlock];
```

列表 4-4 显示了用于给 Cocoa 条件发送信号的代码，并递增他断言变量。你应该在给它发送信号前锁住条件。

Listing 4-4 Signaling a Cocoa condition

```
                                [cocoaCondition lock];

                                    timeToDoWork++;

                                [cocoaCondition signal];

                                [cocoaCondition unlock];
```

4.7.2 使用 POSIX 条件

POSIX 线程条件锁要求同时使用条件数据结构和一个互斥锁。经管两个锁结构是分开的，互斥锁在运行的时候和条件结构紧密联系在一起。多线程等待某一信号应该总是一起使用相同的互斥锁和条件结构。修改该成双结构将会导致错误。

列表 4-5 显示了基本初始化过程，条件和断言的使用。在初始化之后，条件和互斥锁，使用 ready_to_go 变量作为断言等待线程进入一个 while 循环。仅当断言被设置并且随后的条件信号等待线程被唤醒和开始工作。

Listing 4-5 Using a POSIX condition

```
pthread mutex t mutex;

pthread cond t condition;

Boolean    ready to go = true;


void MyCondInitFunction()

{

    pthread mutex init(&mutex);

    pthread cond init(&condition, NULL);

}


void MyWaitOnConditionFunction()

{
```

```
    // Lock the mutex.
    pthread mutex lock(&mutex);

    // If the predicate is already set, then the while loop is bypassed;
    // otherwise, the thread sleeps until the predicate is set.
    while(ready to go == false)
    {
        pthread cond wait(&condition, &mutex);
    }

    // Do work. (The mutex should stay locked.)

    // Reset the predicate and release the mutex.
    ready to go = false;
    pthread mutex unlock(&mutex);
}
```

信号线程负责设置断言和发送信号给条件锁。列表 4-6 显示了实现该行为的代码。在该例子中，条件被互斥锁内被发送信号来防止等待条件的线程间发生竞争条件。

Listing 4-6 Signaling a condition lock

```
void SignalThreadUsingCondition()
{
    // At this point, there should be work for the other thread to do.
    pthread mutex lock(&mutex);
    ready to go = true;

    // Signal the other thread to begin work.
    pthread cond signal(&condition);

    pthread mutex unlock(&mutex);
}
```

注意：*上述代码是显示使用 POSIX 线程条件函数的简单例子。你自己的代码应该检测这些函数返回错误码并恰当的处理它们。*

附录 A：线程安全总结

本附录描述了 Mac OS X 和 iOS 上面一些关键的高级线程安全的框架。本附录的信息有可能会发生改变。

Cocoa

在 Cocoa 上面使用多线程的指南包括以下这些：

- 不可改变的对象一般是线程安全的。一旦你创建了它们，你可以把这些对象在线程间安全的传递。另一方面，可变对象通常不是线程安全的。为了在多线程应用里面使用可变对象，应用必须适当的同步。关于更多信息，参阅”可变和不可变对比”。
- 许多对象在多线程里面不安全的使用被视为是”线程不安全的”。只要同一时间只有一个线程，那么许多这些对象可以被多个线程使用。这种被称为专门限制应用程序的主线程的对象通常被这样调用。
- 应用的主线程负责处理事件。尽管 Application Kit 在其他线程被包含在事件路径里面时还会继续工作，但操作可能会被打乱顺序。
- 如果你想使用一个线程来绘画一个视图，把所有绘画的代码放在 NSView 的 lockFocusIfCanDraw 和 unlockFocus 方法中间。

为了在 Cocoa 里面使用 POSIX 线程，你必须首先把 Cocoa 变为多线程模式。关于更多信息，参阅“在 Cocoa 应用里面使用 POSIX 线程”部分。

基础框架（Fondation Framework）的线程安全

有一种误解，认为基础框架(Foundation framework)是线程安全的，而 Application Kit 是非线程安全的。不幸的是，这是一个总的概括，从而造成一点误导。每个框架都包含了线程安全部分和非线程安全部分。以下部分介绍 Foundation framework 里面的线程安全部分。

线程安全的类和函数

下面这些类和函数通常被认为是线程安全的。你可以在多个线程里面使用它们的同一个实例，而无需获取一个锁。

- NSArray
- NSAssertionHandler
- NSAttributedString
- NSCalendarDate
- NSCharacterSet
- NSConditionLock
- NSConnection
- NSData
- NSDate
- NSDecimal functions
- NSDecimalNumber
- NSDecimalNumberHandler
- NSDeserializer
- NSDictionary
- NSDistantObject
- NSDistributedLock
- NSDistributedNotificationCenter
- NSException
- NSFileManager (in Mac OS X v10.5 and later)
- NSHost
- NSLock
- NSLog/NSLogv
- NSMethodSignature
- NSNotification
- NSNotificationCenter
- NSNumber
- NSObject

- NSPortCoder
- NSPortMessage
- NSPortNameServer
- NSProtocolChecker
- NSProxy
- NSRecursiveLock
- NSSet
- NSString
- NSThread
- NSTimer
- NSTimeZone
- NSUserDefaults
- NSValue
- 还有对象的 allocation 和 retain 函数
- Zone 和内存函数

非线程安全类

以下这些类和函数通常被认为是非线程安全的。在大部分情况下，你可以在任何线程里面使用这些类，只要你在同一个时间只在一个线程里面使用它们。参考这些类对于的额外详细信息的文档。

- NSArchiver
- NSAutoreleasePool
- NSBundle
- NSCalendar
- NSCoder
- NSCountedSet
- NSDateFormatter

- NSEnumerator
- NSFileHandle
- NSFormatter
- NSHashTable functions
- NSInvocation
- NSJavaSetup functions
- NSMapTable functions
- NSMutableArray
- NSMutableAttributedString
- NSMutableCharacterSet
- NSMutableData
- NSMutableDictionary
- NSMutableSet
- NSMutableString
- NSNotificationQueue
- NSNumberFormatter
- NSPipe
- NSPort
- NSProcessInfo
- NSRunLoop
- NSScanner
- NSSerializer
- NSTask
- NSUnarchiver
- NSUndoManager
- User name and home directory functions

注意，尽管 NSSerializer,NSArchiver,NSCoder 和 NSEnumerator 对象本身是线程安全的，但是它们被放置这这里是因为当它们封装的对象被使用的时候，更改这些对象数据是不安全的。比如，在归档情况下，修改被归档的对象是不安全的。对于一个枚举，任何线程修改枚举的集合都是不安全的。

只能用于主线程的类

以下的类必须只能在应用的主线程类使用。

- NSAppleScript

可变 vs 不可变

不可变对象通常是线程安全的。一旦你创建了它们，你可以把它们安全的在线程间传递。当前，在使用不可变对象时，你还应该记得正确使用引用计数。如果不适当的释放了一个你没有引用的对象，你在随后有可能造成一个异常。

可变对象通常是非线程安全的。为了在多线程应用里面使用可变对象，应用应该使用锁来同步访问它们（关于更多信息，参见“原子操作”部分）。通常情况下，集合类（比如，NSMutableArray,NSMutableDictionary）是考虑多变时是非线程安全的。这意味着，如果一个或多个线程同时改变一个数组，将会发生问题。你应该在线程读取和写入它们的地方使用锁包围着。

即使一个方法要求返回一个不可变对象，你不应该简单的假设返回的对象就是不可变的。依赖于方法的实现，返回的对象有可能是可变的或着不可变的。比如，一个返回类型是 NSString 的方法有可能实际上由于它的实现返回了一个 NSMutableString。如果你想要确保对象是不可变的，你应该使用不可变的拷贝。

可重入性

可重入性是可以让同一对象或者不同对象上一个操作“调用”其他操作成为可能。保持和释放对象就是一个有可能被忽视的”调用”的例子。

以下列表列出了 Foundation framework 的部分显式的可重入对象。所有其他类可能是或可能不是可重入的，或者它们将来有可能是可重入的。对于可重入性的一个完整的分析是不可能完成的，而且该列表将会是无穷尽的。

- Distributed Objects
- NSConditionLock
- NSDistributedLock
- NSLock
- NSLog/NSLogv
- NSNotificationCenter
- NSRecursiveLock
- NSRunLoop
- NSUserDefaults

类的初始化

Objective-C 的运行时系统在类收到其他任何消息之前给它发送一个 initialize 消息。这可以让类有机会在它被使用前设置它的运行时环境。在一个多线程应用里面，运行时保证仅有一个线程(该线程恰好发送第一条消息给类)执行 initialized 方法，第二个线程阻塞直到第一个线程的 initialize 方法执行完成。在此期间，第一个线程可以继续调用其他类上的方法。该 initialize 方法不应该依赖于第二个线程对这个类的调用。如果不是这样的话，两个线程将会造成死锁。

自动释放池(Autorelease Pools)

每个线程都维护它自己的 NSAutoreleasePool 的栈对象。Cocoa 希望在每个当前线程的栈里面有一个可用的自动释放池。如果一个自动释放池不可用，对象将不会给释放，从而造成内存泄露。对于 Application Kit 的主线程通常它会自动创建并消耗一个自动释放池，但是辅助线程（和其他只有 Foundationd 的程序）在使用 Cocoa 前必须自己手工创建。如果你的线程是长时间运行的，那么有可能潜在产生很多自动释放的对象，你应该周期性的销毁它们并创建自动释放池(就像 Application Kit 对主线程那样)。否则，自动释放对象将会积累并造成内存大量占用。如果你的脱离线程没有使用 Cocoa，你不需要创建一个自动释放池。

Run Loops

每个线程都有一个或多个 run loop。然而每个 run loop 和每个线程都有它自己的输入模式来决定 run loop 运行的释放监听那些输入源。输入模式定义在一个 run loop 上面，不会影响定义在其他 run loop 的输入模式，即使它们的名字相同。

如果你的线程是基于 Application Kti 的话，主线程的 run loop 会自动运行，但是辅助线程(和只有 Foundation 的应用)必须自己启动它们的 run loop。如果一个脱离线程没有进入 run loop，那么线程在完成它们的方法执行后会立即退出。

尽管外表显式可能是线程安全的，但是 NSRunLoop 类是非线程安全的。你只能在拥有它们的线程里面调用它实例的方法。

Application Kit 框架的线程安全

以下部分介绍了 Application Kit 框架的线程安全。

非线程安全类

以下这些类和函数通常是非线程安全的。大部分情况下，你可以在任何线程使用这些类，只要你在同一时间只有一个线程使用它们。查看这些类的文档来获得更多的详细信息。

- NSGraphicsContext。多信息，参见“NSGraphicsContext 限制”。
- NSImage.更多信息，参见“NSImage 限制”。
- NSResponder。
- NSWindow 和所有它的子类。更多信息，参见“Window 限制

只能用于主线程的类

以下的类必须只能在应用的主线程使用。

- NSCell 和所有它的子类。
- NSView 和所有它的子类。更多信息，参见“NSView 限制”。

Window 限制

你可以在辅助线程创建一个 window。Application Kit 确保和 window 相关的数

据结构在主线程释放来避免产生条件。在同时包含大量 windows 的应用中，window 对象有可能会发生泄漏。

你也可以在辅助线程创建 modal window。在主线程运行 modal loop 时，Application Kit 阻塞辅助线程的调用。

事件处理例程限制

应用的主线程负责处理事件。主线程阻塞在 NSApplication 的 run 方法，通常该方法被包含在 main 函数里面。在 Application Kit 继续工作时，如果其他线程被包含在事件路径，那么操作有可能打乱顺序。比如，如果两个不同的线程负责关键事件，那么关键事件有可能不是按照顺序到达。通过让主线程来处理事件，事件可以被分配到辅助线程由它们处理。

你可以在辅助线程里面使用 NSApplication 的 postEvent:atStart 方法传递一个事件给主线程的事件队列。然而，顺序不能保证和用户输入的事件顺序相同。应用的主线程仍然辅助处理事件队列的事件。

绘画限制

Application Kit 在使用它的绘画函数和类时通常是线程安全的，包括 NSBezierPath 和 NSString 类。关于使用这些类的详细信息，在以下各部分介绍。关于绘画的额外信息和线程可以查看 Cocoa Drawing Guide。

a) NSView 限制

NSView 通常是线程安全的，包含几个异常。你应该仅在应用的主线程里面执行对 NSView 的创建、销毁、调整大小、移动和其他操作。在其他辅助线程里面只要你把绘画的代码放在 lockFocusIfCanDraw 和 unlockFocus 方法之间也是线程安全的。

如果应用的辅助线程想要告知主线程重绘视图，一定不能在辅助线程直接调用 display,setNeedsDisplay:,setNeedsDisplayInRect:，或 setViewsNeedDisplay:方法。相反，你应该给给主线程发生一个消息让它调用这些方法，或者使用 performSelectorOnMainThread:withObject:waitUntilDone:方法。

系统视图的图形状态(gstates)是基于每个线程不同的。使用图形状态可以在单线程的应用里面获得更好的绘画性能，但是现在已经不是这样了。不正确使用图形状

态可能导致主线程的绘画代码更低效。

b） NSGraphicsContext 限制

NSGraphicsContext 类代表了绘画上下文，它由底层绘画系统提供。每个 NSGraphicsContext 实例都拥有它独立的绘画状态：坐标系统、裁剪、当前字体等。该类的实例在主线程自动创建自己的 NSWindow 实例。如果你在任何辅助线程执行绘画操作，需要特定为该线程创建一个新的 NSGraphicsContext 实例。

如果你在任何辅助线程执行绘画，你必须手工的刷新绘画调用。Cocoa 不会自动更新辅助线程绘画的内容，所以你当你完成绘画后需要调用 NSGraphicsContext 的 flusGrahics 方法。如果你的应用程序只在主线程绘画，你不需要刷新绘画调用。

c） NSImage 限制

线程可以创建 NSImage 对象，把它绘画到图片缓冲区，还可以把它传递给主线程来绘画。底层的图片缓存被所有线程共享。关于图片和如何缓存的更多信息，参阅 Ccocoa Drawing Guide。

Core Data 框架

Core Data 框架通常支持多线程，尽管需要注意一些使用注意事项。关于这些注意事项的更多信息,参阅 Core Data Programing Guide 的“Multi-Threading with Core Data”部分。

Core Foundation（核心框架）

Core Foundation 是足够线程安全的，如果你的程序注意一下的话，应该不会遇到任何线程竞争的问题。通常情况下是线程安全的，比如当你查询（query）、引用(retain)、释放(release)和传递(pass)不可变对象时。甚至在多个线程查询中央共享对象也是线程安全的。

像 Cocoa 那样，当涉及对象或它们内容突变时，Core Foundation 是非线程安全的。比如，正如你所期望的，无论修改一个可变数据或可变数组对象，还是修改一个可变数组里面的对象都是非线程安全的。其中一个原因是性能，这是在这种情况下的关键。此外，在该级别上实现完全线程安全是几乎不可能的。例如，你不能排除从集合中引用（retain）一个对象产生的无法确定的结果。该集合本身在被调用来引用

(retain)它所包含的对象之前有可能已经被释放了。

这些情况下，当你的对象被多个线程访问或修改，你的代码应该在相应的地方使用锁来保护它们不要被同时访问。例如，枚举 Core Foundation 数组对象的代码，在枚举块代码周围应该使用合适的锁来保护它免遭其他线程修改。

术语表

- **应用（application）**

  一个显示一个图形用户界面给用户的特定样式程序。

- **条件(condition)**

  一个用来同步资源访问的结构。线程等待某一**条件**来决定是否被允许继续运行，直到其他线程显式的给该条件发送信号。

- **临界区(critical section)**

  同一时间只能不被一个线程执行的代码。

- **输入源(input source)**

  一个线程的异步事件源。输入源可以是基于端口的或手工触发，并且必须被附加到某一个线程的 run loop 上面。

- **可连接的线程(join thread)**

  退出时资源不会被立即回收的线程。可连接的线程在资源被回收之前必须被显式脱离或由其他线程连接。可连接线程提供了一个返回值给连接它的线程。

- **主线程(main thread)**

  当创建进程时一起创建的特定类型的线程。当程序的主线程退出，则程序即退出。

- **互斥锁(mutex)**

  提供共享资源互斥访问的锁。一个互斥锁同一时间只能被一个线程拥有。试图获取一个已经被其他线程拥有的互斥锁，会把当前线程置于休眠状态知道该锁被其他线程释放并让当前线程获得。

- **操作对象(operation object)**

  NSOperation 类的实例。操作对象封装了和某一任务相关的代码和数据到一个执行单元里面。

- **操作队列(operation queue)**

  NSOperationQueue 类的实例。操作队列管理操作对象的执行。

- **进程(process)**

  应用或程序的运行时实例。一个进程拥有独立于分配给其他程序的的内存空

间和系统资源（包括端口权限）。进程总是包含至少一个线程（即主线程）和任意数量的额外线程。

- **程序(program)**

  可以用来执行某些任务的代码和资源的组合。程序不需要一个图形用户界面，尽管图形应用也被称为程序。

- **递归锁(recursive lock)**

  可以被同一线程多次锁住的锁。

- **Run loop（运行循环）**

  一个事件处理循环，在此期间事件被接收并分配给合适的处理例程。

- **Run loop 模式(run loop mode)**

  与某一特定名称相关的输入源、定时源和 run loop 观察者的集合。当运行在某一特定“模式”下，一个 run loop 监视和该模式相关的源和观察者。

- **Run loop 对象(run loop object)**

  NSRunLoop 类或 CFRunLoopRef 不透明类型的实例。这些对象提供线程里面实现事件处理循环的接口。

- **Run loop 观察者(run loop observer)**

  在 run loop 运行的不同阶段时接收通知的对象。

- **信号量(semaphore)**

  一个受保护的变量，它限制共享资源的访问。互斥锁(mutexes)和条件(conditions)都是不同类型的信号量。

- **任务(task)**

  要执行的工作数量。尽管一些技术(最显著的是 Carbon 多进程服务—Carbon Multiprocessing Services)使用该术语的意义有时不同，但是最通用的用法是表明需要执行的工作数量的抽象概念。

- **线程(thread)**

  进程里面的一个执行过程流。每个线程都有它自己的栈空间，但除此之外同一进程的其他线程共享内存。

- **定时源(timer source)**

  为线程同步事件的源。定时器产生预定时间将要执行的一次或重复事件。

结束语

多线程编程在开发应用的时候非常有帮助。比如你可以在后台加载图片，等图片加载完成后再在主线程更新等，或者在后台处理一些需要占用 CPU 很长时间的事件（比如请求服务器，加载数据等）。要体会多线程编程的好处，还得多实战，结合使用多种多线程技术。特别要注意 Run Loop 的使用，很多开发者在编写多线程应用的时候很少关注过 Run Loop。如果你仔细阅读并掌握 Run Loop 的细节，将会帮助你写出更优美的代码。同步是多线程编程的老生常谈，估计大学时候大家都基本熟悉了同步的重要性。

最后，本文在翻译过程中发现很多地方直译成中文比较晦涩，所以采用了意译的方式，这不可避免的造成有一些地方可能和原文有一定的出入，所以如果你阅读的时候发现有任何的错误都可以给我发邮件:xyl.layne@gmail.com。

最后可以关注我微博大家一起沟通交流学习。

微博地址: http://weibo.com/u/1826448972

推荐资源

- **核心动画编程指南【Core Animation Programming Guide】**

下载地址:
http://www.cocoachina.com/bbs/read.php?tid=84461

- **Blocks 编程要点【Blocks Programming Topics】**

下载地址:
http://www.cocoachina.com/bbs/read.php?tid=87593

- **Instruments 用户指南【Instruments User Guide】**

下载地址:【近期推出，敬请关注微博动态】