简介

线程是在一个应用是实现执行多条 code path 的技术中一个。尽管 operation 对象和 Grand Central Dispatch 这类比较新的技术提供一个现代的、高效的架构来实现并发，OS X 和 iOS 同样也提供了创建和管理线程的接口。

这篇文档提供了关于 OS X 中线程整体的介绍，向你展示了怎么使用它们。这篇文档也展示了一些相关技术，提供了来支持多线程和同步多线程代码。

> **注意**: 如果你在开发一个新应用，我们鼓励你去看看 OS X 中实现并发的其他技术，尤其是当你对实现一个多线程应用所需要的设计技术不到熟悉的时候。这些其他简化了实现并发的困难，还提供了相对于传统的线程更高的性能。想要了解更多信息的话，可以参见 *Concurrency Programming Guide*

关于多线程编程

计算机的最高性能很大程度上被它的单核心所限制，这种状况持续了很多年。随着单个处理的速度开始达到它们的实际限度，芯片制造商开始转换到多核心设计，给了计算机同时执行多个任务的可能。尽管 OS X 总是在能的时候总是利用这些核心，但你的应用可以使用这些核心。

什么是线程?

线程是在应用中实现多条代码执行路径的相对轻的方式。在系统层面，程序是一个挨着一个的，系统根据各个程序的需要分发的运行时间。然而在每个程序内部，存在着一个或多个执行的线程，这些线程可以被用来同时执行不同的任务，或接近于同时。系统自己真正管理这些线程的执行，根据可用的核心数抢占式打断它们以允许其它线程执行。

从一个技术的角度来看，一个线程是一个 kernel 级别和用户级别的数据结构的组合，它们被用来管理线程的执行。kernel 级别的结构被用来协调分发事件给线程，和抢占式的在核心上调度线程。用户级别的数据结构存储调用栈的 call stack，和应用需要用来管理和操作线程属性和状态的数据结构。

在一个非并发的应用，只有一个执行线程。这个线程以应用的 `main` 函数开始和结束，一个接着一个分支到不同的函数或方法来完成整个应用的行为。对比下，一个支持并发的应用从一个线程开始，根据需要创建多个执行路径。每个新的执行路径有一个自定义的启动程序，跟应用的 main 函数相互独立。一个应用有多个线程有以下好：

- 多线程可以提高应用的视觉响应
- 多线程可以提高一个应用在多核系统上的性能

如果一个应用只有一个线程，那么这个线程必须做所有事。它必须响应事件、更新应用的窗口，进行实现应用行为的所有计算。在任一时刻只有一个线程的问题是它只能做一件事。所以如果当你的一个计算任务需要花费较长时间完成了？当你的应用忙于计算时，你的应用停止响应用户的事件和更新窗口了。如果这种情况持续时间足够长，用户可能会认为应用挂起了，尝试强制退出它。如果你把自定义的计算移到一个单独的线程，你的应用的主线程就可以更及时的响应用户的交互。

近来多核计算机越发普遍，线程给某些应用提供了提升性能的一种方式。线程可以在不同的处理核心上同时执行不同的任务，使得一个应用可以在给定的时间内做更多的工作。

当然，线程并不是解决应用性能的万灵药。随着线程提供的好处而来的时候一些潜在问题。有多些执行路径的应用会增加相当的复杂性。每个线程需要跟其他的线程协调它的行为，阻止扰乱应用的状态信息。因为单个应用的中线程是共享地址空间的，它们可以访问相同的数据结构。如果两个线程试着同时操作同样的数据结构，一个线程可能以某种扰乱的方式重写了另一个线程的改动。即使这时有合适的保护，你仍然还是的注意编译器的优化可能会引入微妙的 bugs。

线程的术语

在我们进一步讨论线程和支撑它们的技术前，有必要定一些基本的术语。

如果你熟悉 UNIX 系统，你可以会发现术语 "task" 跟这篇文档中的不大相同。在 UNIX 系统中，术语 "task" 用来代指一个执行的进程。

这篇文档采用了以下术语：

- *thread* 用来指代码执行的单一路径
- *process* 用来指 a running executable，由多条线程组成
- *task* 用来指需要做的工作的抽象概念

线程的替代品

创建线程的一个问题是给你的代码添加了不确定性。线程是一个实现并发相对底层且复杂的方式。如果你不确定你的设计选择意味着什么的话，你会很容易遇到同步或时序相关的问题，这些问题的严重性从轻微的行为变化到应用的 crash 和用户数据的错误修改。

另一个需要考虑的是我们到底需不需要多线程。线程解决了在进程中并发的执行多条代码路径的特定问题。然而有些情况是你所做工作的量并不值得使用并发。线程给进程引入了值得注意的负载，从内存和 CPU 时间的角度看。你也许会发现这样的负载对于要做的工作而言过重，或者其他更简单的方式实现。

下表列出了线程的一些替代方案。这份表格列出了线程技术的替代方案和一些其它专门用来提高单个线程效率的技术。

Technology	Description
Operation 对象	自 OS X v10.5 引入，一个 operation 对象是一个通常在其它线程上执行任务的封装。这层包装隐藏了执行任务所需要的线程管理，让你专注于需要做的任务。通常你会和 operation queue 一起使用这些对象，operation queue 会管理这些 operation 对象的在一个或多个线程上执行。更多的信息，参见 * Concurrency Programming Guide*
Grand Central Disaptch	自 OS X v10.6 引入，GCD 是线程的替换方案，让你专注于需要进行的任务，而不是线程管理。使用 GCD 你定义好需要执行的任务，提交任务到队列。队列负责在合适的线程上调度任务。工作队列会考虑可用核心数和当前系统负载，往往比你用线程自己做要高效很多。更多信息参见 *Concurrency Programming Guide*
Idle-time notifications	对于相对短且低优先级的任务，idle time notification 允许你在应用不忙的时候进行任务。Cocoa 通过 `NSNotificationQueue` 对象提供 idle-time notifications。要请求一个 idle-time notificaiton，使用默认的 `NSNotificationQueue` 对象发送 `NSPostWhenIdle` 选项的通知。这个队列会延迟投递 (delivery) 你的通知对象，直到 run loop 对象闲置的时候。更多信息参见 *Notification Programming Topics*
Asynchronous functions	系统 API 提供了自动实现并发的异步接口。这些 APIs 可能使用系统守候进程或创建自定义线程完成任务，并返回值给你。(具体的实现是不相关的，因为跟你的代码是分隔的) 在你设计你的应用的时候，寻找并考虑那些提供异步行为的接口，而不是使用同样功能的同步接口。
Timers	你可以在你的主线程上使用 timers 来进行定时的任务。
Separate processes	尽管进程比线程更重，在你的任务跟你的应用完全不相干的情况，创建独立的进程可能更有效。你也许会使用一个进程如果一个任务必须使用一定量的内存或必须使用根权限执行。

> **警告**: 当通过 `fork` 启动一个新进程的时候，你必须要接着调用一个 `exec` 或相似的函数。依赖于 Core Founadtion 的应用，Cocoa，或 Core Data 框架必须调用 `exec` 函数，否则这些框架可能不能正确的工作。

线程支持

如果你有线程的代码使用线程，OS X 和 iOS 提供了在应用中创建线程的一些技术。另外，两个系统都提供了管理和同步线程所需工作的支持。下面的部分描述了一些在 OS X 和 iOS 中是使用线程时你所需要知道的关键技术。

Threading Packages

尽管线程的底层实现机制是 Mach threads，你很少在 Mach 层面使用线程，而是使用更方便的 POSIX API 或它的衍生物。 Mach 的实现并没有提供所有线程的基本特性，然而包含抢占式执行模式和调度线程的能力。

下表列出了你可以使用的线程技术:

Technology	Description
Cocoa threads	Cocoa 通过 `NSThread` 实现线程。Cocoa 也在 `NSObject` 上提供了方法来生成线程和在已经跑着的线程上执行代码。
POSIX threads	POSIX 提供了基于 C 的接口来创建线程。如果你不是在写 Cocoa 应用，这是创建线程的最好选择。POSIX 接口相对而言使用简单并且提供了配置线程的充足灵活性。
Multiprocessing Services	Multiprocessing Services 是一个历史遗留的基于 C 的接口，被用于从老的 Mac OS 迁移过来的应用。这项技术只在 OS X 中可用，在新的开发中应该尽量避免。相反，你应该使用 `NSThread` 类或 POSIX 线程。

在应用的层面，所有的线程在不同的平台上行为是一致的。在启动一个线程后，线程跑在三个主要状态之一：running，ready，blocked。如果一个线程不在 running，它要么是被阻塞在等待输入要么是它准备好执行但还没有来得及被调度。线程在这些状态间来回的转换直到它最后退出，转为中止状态 (terminated)

当你创建一个线程的时候，你必须为线程指定一个入口函数 (或创建一个入口函数)。入口函数由你想在线程上执行的代码组成。当一个函数返回的时候，或当你显式的中止线程，线程会永远停止运行并被系统回收。因为从时间和内存的角度来看创建线程是相对昂贵的，因此推荐你的入口函数做一定量的工作，或设置一个 run loop 来执行重复发生的工作。

Run Loops

一个 run loop 是一个用来异步管理到达线程事件的架构的一部分。一个 run loop 的工作时为线程监听一个或多个事件源。事件到达的时候，系统唤醒线程，将时间分发给 run loop，run loop 再将事件分发到你指定的 handlers。如果没有事件出现和准备好被处理，run loop 会将线程睡眠掉。

对于你创建的线程你不是必须要使用 run loop，但是这么做的话给用户提供了更好的体验。Run loops 使得使用少量的资源创建长时间运行的线程变得可能。因为一个 run loop 在没有什么事情做的时候会将线程睡眠，消除了浪费 CPU cycle 轮询的需要，阻止了处理器自身进入睡眠，降低了能耗。

要配置一个 run loop，你需要做的是启动一个线程，获取 run loop 对象的引用，安装时间处理的 handler，然后告诉 run loop 跑起来。OS X 提供的基础架构为你自动的配置了主线程的 run loop 对象。如果你计划创建长时间运行的线程，你必须亲自配置这些线程的 run loop。

同步工具

多线程的一个危害是多条线程对于资源的竞争。如果你多个线程同时试着使用并修改资源，问题就可能产生。一种避免问题的方式是完全的消除共享资源，确保每个线程在它自己的资源上操作。当维护完全独立的资源变得不可能的时候，你可能必须使用锁、conditions、原子操作或其它技术来同步这些资源的访问。

锁为代码提供强力形式的保护，在一个时刻只有一个线程可被执行。最常用的锁类型时互斥锁，简称 `mutext` 。当一个线程试着去获得一个被另一个线程持有的锁时，它被阻塞直到锁被另一个线程所释放。好几个系统框架提供 mutex 锁的支持，尽管它们都是基于同样的底层技术。另外，Cocoa 提供了 mutex 的多种变种来支持不同类型的行为，如 recrusion。

除了锁外，系统提供了 conditions 的支持，conditions 保证了应用中任务的顺序。一个 condition 像是一个守门员，阻塞给定的线程直到它所代表的 condition 变为 true。当那种变化发生的时候，condition 释放线程，允许它继续执行。POSIX 和 Foundation 框架都提供了 conditions 的直接支持。(如果你使用 operation 对象的话，你可以配置 operation 对象间的依赖性来保证任务的执行顺序，这方面跟 conditions 提供的行为是比较像的)

尽管锁和 conditions 在并发的设计中很普遍，原子操作是另一种保护和同步访问数据的方式。原子操作在你进行原子数据类型的数学或逻辑运算的时候，提供了锁的替换方案。原子操作使用特殊的指令来保证对一个变量的修改在另一个线程可以访问它们之前完成。

线程间通讯

尽管一个好的设计可以最小化需要的通讯，但在某些点，线程间的通讯变得很有必要。(一个线程的工作是为应用做工作，如果它们工作的结果从不被使用的话，这样有什么好处了?) 线程可能需要处理新来的工作请求或向应用的主线程报告它的进度。在这些情况下，你需要一种方式在一个线程中获取另一个线程的信息。幸运的是，线程是共享进程的地址空间的，这意味着你有需要可供选择的选项。

有需要线程通讯的方式，每个都有优缺点。配置 Thread-Local Storage 是这些在 OS X 中你可以使用的通讯机制中最通用的。(除了 message queues 和 Cocoda distributed objects，这些技术也在 iOS 中可用) 表中的技术是以复杂度的增长来的。

Mechanism	Description
Direct message	Cocoa 应用支持直接在其它线程上执行 selector。本质上这个能力意味着一个线程可以在任何其它线程上执行一个方法。因为它们是在目标线程的上下文中执行，这种方式发送的消息自动在线程上串行起来。
Global variables, shared memory，and objects	另一个线程间通讯的简单方式是在线程间使用全局变量，共享对象，或一段内存。尽管共享变量是最快和简单，但相对于直接发送消息，它们更脆弱。共享的内存很小心的被锁或其他同步机制保护，以保证代码行为的正确性。没能这么做的话会导致 race conditions，corrupted data，或 crashes。
Conditions	Conditions 是一个同步工具，你可以用来控制什么时候一个线程执行一段代码。你可以认为 conditions 是一个守门员，只当描述的条件满足的时候，才让一个线程执行。
Run loop sources	一个 run loop source 是一个你设置来在应用指定的线程接受应用特定的消息。因为它们是事件驱动的，在没有事处理的时候，run loop source 会将线程置为睡眠状态。这样做提高了线程的效率。
Ports and sockets	基于 port 的通讯是一个线程间通讯相对豪华的方式，但也是一个非常可靠的技术。更重要的是，porst 和 sockets 可被用来和外部通讯，如其它的进程或服务。为了效率，ports 是用 run loop source 来实现的，所以当 port 上没有数据等待的时候，线程会睡眠。
Message queues	历史遗留的 Multiprocessing Services 定义了一个 first-in，first-out 队列，抽象来管理接受和发送数据。尽管消息队列简单方便，它们没有其他的通讯技术高效。更多的信息，参见 *Multiprocessing Services Programming Guide*
Cocoa distributed objects	Distributed objects 是一个 Cocoa 技术，提供了一个高级基于 port 通讯的实现。尽管使用这个技术实现线程间通讯是可能的，但这么做是很不鼓励的，因为随之而来的大量负载。Distributed objects 更适合于进程间的通讯，这种时候进程间的数据交流本身就负载很大。更多的信息，查看 *Distributed Objects Programming Topics *

Design Tips

下面的部分描述提供了一些引导，帮助你实现以正确的方式实现多线程。一些引导提供了关于使用线程代码实现更好的性能的指导。跟所有的性能指导一样，你总是应该收集相关的性能数据，在你改变的你代码之前、之中、之后。

避免显式的创建线程

手动写创建线程的代码很无聊，而且很容犯错，可能的情况下你应该尽量避免。 OS X 和 iOS 通过 API 提供了隐式的并发支持。与其自己创建线程，你应该考虑使用异步的 APIs，GCD，operations 对象来完成你的工作。这些技术在幕后为你做好线程相关的工作，并保证正确的做好这些。另外，像 GCD 和 operation 这些技术通过基于系统的当前的负载调整线程数的设计使得比你自己的代码更高效的管理线程。更多关于 GCD 和 operation 对象更多的信息，可以参见 *Concurrency Programming Guide*

保持你的线程合理的忙

如果你决定手动创建和管理线程，记住线程消耗宝贵的系统资源。你应该尽最大努力保证任何你赋给线程的工作是长时间执行的并有产出的。同时，你不应该害怕中止那些闲置的线程。线程使用非琐碎量的内存，有部分是很紧张的类型，所以释放一个闲置线程不仅减少了应用内存足迹，它同样给系统其他的进程释放了更多的物理内存。

> **注意**: 在你开始中止线程之前，你总是应该记录你应用的目前性能的一些基本数据。在你尝试你的改变的时候，采取额外的步骤来验证这些改变是否真的提高了性能。

避免共享数据结构

避免线程相关的资源冲突问题最简单容易的方法是给与每个线程一份自己所需数据的拷贝。并行的代码越少的通讯和资源竞争工作得越好。

创建一个多线程的应用是困难的。即使用你很小心，总是在代码的结合点锁住共享的数据结构，你的代码可能仍然是不安全的。例如，如果你的代码期望共享的数据结构以指定的顺序修改。将你的代码改成基于事物的模型来解决这个问题的话，可能最终会抵消掉使用多线程所带来的性能提升。最开始消除资源竞争常常会导致一个更简单的设计，更好的性能。

线程和你的 UI

如果你的你敢用有一个图形界面，推荐从你的主线程接收用户相关的事件，和发起界面更新。这种尝试避免了与处理用户事件和画窗口内容相关的同步问题。一些框架，像 Cocoa，要求这样的行为，但即使对于没有要求的框架，在主线程上保持这种行为，有简化管理用户界面代码逻辑的好处。

有些在非主线程执行图形操作反而更好的例外。例如，你可以使用非主线程创建和处理图片，或其他图片相关的计算。使用非主线程来做这些操作可以极大地提升性能。如果你关于一个特别的图形操作不确定的话，计划在主线程上执行它。

关于 drawing in Cocoa，参见 `Cocoa Drawing Guide` 。

Be Aware of Thread Behaviors at Quit Time

一个进程在所有的 non-detached 线程退出之后退出。默认，只有应用的主线程被创建成 non-detached，但是你也可以创建这样的线程。当用户退出一个应用的时候，立即中止所有的 detached 线程被认为是合适的行为，因为 detached 线程所做的工作被认为是可选的。如果你的应用的使用后台线程来保存数据到硬盘或做些关键的工作，你也许想要创建这些线程为 non-detached，来阻止应用退出的时候数据的丢失。

创建 non-detached (also known as joinable) 线程需要你额外的工作。因为大部分高级的线程技术默认不创建 joinable 的线程，你也许需要使用 POSXI API 来创建你的线程。另外，你必须添加代码到你的主线程来 join 那些 non-detached 线程当它们结束运行的时候。

如果你在写一个 Cocoa 应用，你可以使用 `applicationShouldTerminate:` delegate 方法来延迟你应用的结束运行直到一个稍后的时间或直接取消退出。当你延迟退出的时候，你的应用需要等待任何关键线程已经完成它们的工作，然后调用 `replyToApplicationShouldTerminate:` 方法。

处理异常

当异常被跑出时异常处理机制依赖于当前的调用栈来进行任何必须的清理工作。因为每个线程有它自己的调用栈，每个线程负责捕获它自己的异常。非主线程上没有捕获的异常和主线程上没有捕获是一样的：进程被终止。你不能抛出一个未捕获的异常到一个其他的线程处理。

如果你需要通知其它的线程 (如主线程) 当前线程的异常情况，你应该捕获这个异常然后简单的发送一个消息到相应的线程标识发生了什么。依赖于你的模型和你在做什么，捕获异常的线程可以继续执行，等待指令，或简单的退出。

> **注意**: 在 Cocoa中，一个 `NSException` 对象是一个自包含的对象，一旦被捕获后可以在线程间传递。

有些情况下，一个异常 handler 也许会自动的给你创建。例如 Objective-C `@synchronized` 命令就包含一个隐式的异常 handler。

干净的终止你的线程

终止线程最好的方式是让它自然的到达 main entry point 函数的结束。尽管有函数可以立即终止线程，这些函数应该被当做最后的求助方式。在线程到达它自然的终点前终止它阻止了它清理资源。如果线程分配了内存，打开了文件，或获得了其它类型的资源，你的代码也许不能回收这些资源，导致内存泄露或其它潜在问题。

Libraries 中的线程安全

尽管一个应用开发者拥有是否让一个应用具有多线程的控制权，library 的开发者没有。当开发 library 时，你必须假设调用的应用是多线程的，或可能在某个时刻切到多线程。结果是，你总是应该使用锁保护代码的 critical sections。

对于 library 开发者而言，仅当应用切换到多线程的时候创建锁并不明智。如果在某些点你需要锁住你的代码的话，在使用你的 library 的早期创建这些锁对象，一个比较可取的方式是显式的调用函数来初始化相应的 library。尽管你可以可以使用静态初始化函数来创建这些锁，当且仅当没有其它方式的时候这么做。一个初始化函数的执行增加了加载 library 所需的时间，可能相反的影响了性能。

> **注意**: 记住在你的 library 中你需要平衡调用 lock 和 unlock 一个 mutex 锁。你也应该记住锁住 library 数据结构而不是依赖调用方提供一个线程安全的环境。

如果你再开发一个 Cocoa library的话，你可以注册成 `NSWillBecomeMultiThreadedNotification` 的观察者，如果想要当应用编程多线程的时候被通知。但是你不应该依赖于收到这些通知，因为通知可能在你的 library 代码被调用之前已经被发送了。

线程管理

OS X 和 iOS的每个进程都是由一个或多个线程组成，每个线程代表了应用代码的一条单一执行路径。每个应用都是以一个线程开始，这个线程执行应用的 `main` 函数。应用可以新生另外的线程，这些新生的线程可以执行其他的函数代码。

当一个线程新生线程的时候，新生的线程成为应用中的独立实体。每个线程有它自己的执行栈，被 kernel 单独的调度执行。一个线程可以和其它的线程和进程通讯，进行 I/O 操作，做任何你需要做的事。因为它们在同样的进程空间，应用中的所有线程共享同样的虚拟内存地址空间，和进程有一样的访问权限。

这章提供 OS X 和 iOS 中可用的线程技术的概览，通过实例代码展示了应用中怎么使用这些技术。

> **注意**: 想看看 Mac OS 多线程架构的历史，线程的背景信息的话，查看 Technical Note TN2028 "Threading Architectures"

线程消耗

从内存使用和性能的角度线程对于你的程序是有实际的消耗的。每个线程需要分配内核地址空间和应用程序地址空间的内存。需要用来管理你的线程和协调调度的数据结构消耗着内核空间中 wired memory。线程的栈空间和 per-thread 数据保存在程序的地址空间。这些结构大部分在创建线程的时候被创建。一个进程因为需要和内核打交道，相对昂贵。

下表具体量化了创建一个用户级别的线程相关联的消耗。一些消耗是可配置的，分配给非主线程的栈空间大小。创建线程的时间只是一个粗略估计，只应该用于相对对比。线程的创建时间很大程度上依赖于核心负载，计算机的速度，系统和程序可用的内存 。

Item	Approximae cost	Notes
Kernel data structures	1KB	这些内存用于存储线程数据结构和属性。大部分以 wired memory 形式分配，因此不能被 paged 到硬盘。
Stack space	512 KB (非主线程) 8MB (OS X 主线程) 1MB (iOS 主线程)	分配给非主线程最小的栈大小是 16KB，并且栈的大小必须是 4KB 的整数倍。这些内存是用户进程空间分配的，但真正与这些内存的 pages 是在需要的时候分配的。
Creation time	大概 90 ms	这个时间反应了调用创建线程到线程的 entry point 被执行之间的这段时间。这个数据依赖外部环境的。

> **注意**： 因为底层内核支持，operation 对象经常可以更快的创建线程。与其每次从头创建线程，operations 对象使用已经存在于内核的线程空间来节省分配时间。需要了解更多关于使用 operations 对象，可以参见 *Concurrency Programming Guide*

编写线程代码的时候另一个需要考虑的消耗是生产投入。设计一个多线程应用有时可能需要从根本上改变你组织应用中数据结构的方式。为了避免使用同步机制，因为这些同步机制本身可能造成极大的性能损耗在很糟的设计下，这些改变往往就显得很有必要。设计这些数据结构，在多线程环境下调试代码，可能会增加开发一个多线程应用的时间。如果你的线程花了太多时间等待锁或啥也不做的话，避免这些消耗可能在运行时创建更大的问题。

创建线程

创建底层的线程相对简单。在所有的情况，你必须有一个函数或方法作为线程的主入口函数，你必须使用任何可用线程函数启动线程。接下来的部分展示常用线程技术的基本创建线程过程。使用这些技术创建的线程继承了默认的属性，这些属性由你使用的技术决定。

使用 NSThread

使用 `NSThread` 类有两种方式创建一个线程：

- 使用 `detachNewThreadSelector:toTarget:withObject:` 类方法生成一个线程
- 创建一个新的 `NSThread` 对象，调用它的 `start` 方法。

上述技术在应用中创建一个 detached 线程。一个 detached 线程意味着线程的资源会被自动的被系统回收当线程退出后。也意味着你的代码不需要显式的 join 那个线程。

因为 `detachNewThreadSelector:toTarget:withObject:` 方法在所有的 OS X 版本中都支持，它经常在使用线程的 Cocoa 应用中遇到。要 detach 一个新建线程，你只需简单的提供你使用来作为线程入口函数的方法，实现方法的对象，和线程启动时需要传递给它方法的参数。下面的例子展示使用这个函数的调用生成一个线程：

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

在 OS X v10.5 之前，你主要使用 `NSThread` 来创建线程。尽管你可以创建 `NSThread` 对象并访问一些线程属性，你只能从线程自身里这么做。在 OS X v10.5，添加了对创建 `NSThread` 对象不用立即生成相应的线程的支持 (iOS 中同样支持)。这种支持使得在启动线程之前获取和设置线程的属性变得可能。同样使得使用指向稍后运行的线程的县城对象变得可能。

自 OS X v10.5 起，初始化一个 `NSThread` 对象的简单方式是使用 `initWithTarget:selector:object:` 方法。这个方法需要的信息和 `detachNewThreadSelector:toTarget:withObject:` 完全相同，并使用这些信息创建一个 `NSThread` 对象。然而，要启动这个线程，你显式调用线程的 `start` 方法，如以下代码：

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

> **注意**: 跟使用 `initWithTarget:selector:object:` 方法类似的方案是继承 `NSThread` 类，override 子类的 `main` 方法。你可以使用覆盖的 `main` 方法来作为线程的入口函数。

如果你有一个 `NSThread` 对象，它的线程正在跑着，你可以调用应用任何对象的 `performSelector:onThread:withObject:waitUntilDone:` 方法给这个线程发送消息。支持在线程上执行 selectors 是 OS X v10.5 中引入的，是线程间通讯很方便的一种方式。使用这种技术发送的消息在其他的线程上作为正常的 run-loop 处理过程的一部分。(当然，这以为目标线程必须运行着它自己的 run loop)。使用这种方式通讯的时候你可能仍然需要一些同步方式，但是线程间设置通讯端口很容易。

> **注意**: 尽管线程间偶尔通讯很好，你不应该使用 `performSelector:onThread:withObject:waitUntilDone:` 方法进行高时效或频繁的线程间通讯。

使用 POSIX 线程

OS X 和 iOS 提供了使用基于 C 的 POSIX 线程 API 来创建线程。这项技术可以被使用再任何类型的应用中 (包含 Cocoa 和 Cocoa Touch 应用)，如果你编写适合于多平台应用的话更适合。使用来创建线程的 POSIX 函数很直白， `pthread_create` 。

下列中展示两个使用 POSIX 函数创建线程的自定义函数。 `LaunchThread` 函数创建一个新线程，这个新线程的主入口是 `PosixThreadMainRoutine` 函数。因为 POSIX 默认创建 joinable 的线程，这个例子修改线程的属性，以使线程为 detached。使线程是 detached 给了系统机会回收线程的资源，当线程退出的时候。

```
#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.
  }
}
```

如果你添加上例中的代码到你的一个源文件中，调用 `LaunchThread` 函数，它将会创建一个 detached 线程。当然，使用这段代码新建的线程不会做任何有效的事情。线程会立即运行并退出。为了使事情更有趣点儿，你需要添加代码到 `PosixThreadMainRoutine` 函数来做些有效的工作。为了保证一个线程知道做什么工作，你可以在创建线程的时候传递一个指向某些数据的指针。你将这个指针作为 `pthread_create` 函数的最后一个参数。

为了从你新建的线程传递回信息到应用的主线程，你需要建立目标线程间的通讯路径。对于基于 C 的应用，有好多方式来进行线程间的通讯，包括使用 ports，conditions，或共享内存。对于长时间运行的线程，你总是应该设置某种形式的线程间通讯机制，以给应用的主线程一种方式来检查线程的状态或当应用退出的时候干净的关闭线程。

使用 NSObject 来生成线程

在 iOS 和 OS X v10.5 之后，所有的对象都有生成线程的能力，并使用线程执行一个它们的方法。 `performSelectorInBackground:withObject:` 方法创建一个 detached 线程，并使用指定的方法作为新生线程的入口函数。例如，如果你有一些对象，并且那个对象有一个方法叫 `doSomething` ，你想在后台线程调用这个方法，你可以使用以下代码来做这些：

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

调用这个方法的效果和你使用当前对象，指定的 selector 和参数调用 `NSThread` 的这个 `detachNewThreadSelector:toTarget:withObject:` 方法效果是一样的。新生的线程使用默认的配置，立即开始执行。在 selector 内部，你必须像配置其他线程一样配置。例如，你需要设置一个 autoreleasing pool (如果你没有使用 garbage collection的话)，配置线程的 run loop 如果你计划使用它的话。

在 Cocoa 应用中使用 POSIX 线程

尽管 `NSThread` 类是 Cocoa 应用中创建线程的主要接口，你仍然可以自由的使用 POSIX 线程，当然如果你觉得这么做更方便的话。例如，你也许会使用 POSIX 线程如果现有代码正在使用它们且你不想重写它们的话。如果你计划在一个 Cocoa 应用中使用 POSIX 线程的话，你仍然应该意识到 Cocoa 和线程间的交互和遵守下面部分的引导。

保护 Cocoa 框架

对于多线程的应用，Cocoa 框架使用锁和其它形式的内部同步机制来保证它们行为的正确。为了防止锁在当线程的环境下降低性能，Cocoa 直到你的应用使用 `NSThread` 类创建一个线程时才创建这些锁或使用同步机制。如果你只使用 POSIX 线程 APIs 的话，Cocoa 不会收到它用来判别应用是否现在是多线程的通知。当这些发生的时候，涉及到 Cocoa 框架的操作可能会不稳定或导致应用奔溃。

为了让 Cocoa 知道你想要使用多线程，所有你需要做的是使用 `NSThread` 类创建一个线程，然后让这个线程退出。你的线程入口函数不需要做任何事。仅仅是使用 `NSThread` 创建一个线程就足以保证 Cocoa 框架需要的锁在需要的地方出现。

如果你不确定 Cocoa 是否认为你的应用是多线程的，你可以使用 `NSThread` 的 `isMultiThreaded` 方法。

混合使用 POSIX 和 Cocoa 锁

在同一个应用中混合的使用 POSIX 和 Cocoa 的锁是安全的。Cocoa 锁和 conditions 对象本质上是 POSIX mutexes 和 condistions 的包装。对一个给定的锁，然而你必须使用统一的借口创建和操作这个锁。换句话说，你不能使用一个 Cocoa `NSLock` 对象来操作一个你使用 `pthread_mutex_init` 创建的 mutex ，相反也是一样。

配置线程的属性

在你创建一个线程之后，有些是之前，你可能想要配置线程环境的不同部分。下面的部分描述了一些你可以做的改变，还有什么时候做这些改变。

配置线程的栈大小

对于你创建的每个新线程，系统在你的用户进程空间分配指定量的内存作为线程的调用栈。调用栈管理着栈帧，同时也是线程局部变成存储的位置。分配给线程的内存量大小你可以查看 **线程消耗** 章节。

如果你想要修改给定线程的栈大小，你必须在你创建线程前做这些。所有的线程技术都提供了某些方式来修改栈大小，但是使用 `NSThread` 来修改站大小只在 OS X 和 iOS 中可用。

Technology	Option
Cocoa	在 OS X 和 iOS 中，创建一个 `NSTread` 对象 (不要用 `detachNewThreadSelector: toTarget:withObject:` 方法)，在调用 `start` 方法之前，调用 `setStackSize:` 方法来指定栈大小
POSIX	创建一个 `pthread_attr_t` 结构，使用 `pthread_attr_setstacksize` 函数来改变默认的栈大小。传递这个结构给 `pthread_create` 函数创建线程的时候
Muilprocessing Services	传递合适的栈大小给 `MPCreateTask` 函数当你创建线程的时候

配置 Thread-Local Storage

每个线程维维护了一个 key-value 的字典，可被线程的任何地方访问。你可以使用字典来存储你想要在整个线程执行过程保持的信息。例如，你可以使用它来存储你线程的 run loop 对象迭代的次数。

Cocoa 和 POSIX 以不同的方式存储这个字典，所以不能混合使用不同的技术。只要你在你的线程中始终使用一项技术，最终结果应该相似。在 Cocoa 中，你使用 `NSThread` 的 `threadDictionary` 方法来获取一个 `NSMutableDictionary` 对象，你可以用来添加线程需要的任何信息。在 POSIX 中，你可以使用 `pthread_setspecific` 和 `pthread_getspecific` 函数来设置和获取线程的 key 和 value。

设置线程的 Detached 状态

大部分高级别的技术默认创建 detached 状态的线程。大部分情况下，detached 线程是更好的，因为它们允许系统在线程退出的时候立即释放线程的数据结构。Detached 线程也不需要程序中显式的交互。这意味着是否要获取线程的结果是由你自己决定的。对比下，系统不会回收 joinable 线程的资源，直到你另外的线程显式的 join 那个线程，一个进程也许会阻塞进行 join 操作的线程。

你可以认为 joinable 线程是子线程。尽管它们是相互独立运行的线程，一个 joinable 的线程必须被其它的线程 join，系统才能回收它的资源。Joinable 的线程也提供了一个显式方式来传送数据给另一个线程。在它退出前，一个 joinable 的线程可以传送一个数据指针，或其它返回值给 `pthread_exit` 函数。一个线程可以获取这个只通过调用 `pthread_join` 函数。

> **重要**: 在应用退出的时候，deached 线程可以立即终止，但是 joinable 线程不行。每个 joinable 线程必须在进程被允许退出前被 join。Joinable 线程因此在一个线程做很关键的工作，如保存数据到硬盘时更受欢迎。

如果你确实想要创建 joinable 的线程，这么做的唯一方式是使用 POSIX 线程。POSIX 默认创建 joinable 线程。要将线程标记为 detached 或 joinable的话，通过 `pthread_attr_setdetachstate` 函数修改线程的属性，在创建线程前。在线程开始前，你可以通过调用 `pthread_detach` 将一个 joinable 的线程改为 detached 的线程。想了解更多的关于 POSIX 线程的函数的话，查看 `pthread` man page。要查看更多关于怎么 join 一个线程的话，查看 `pthread_join` man page。

设置线程的优先级

任何你创建的新线程有一个默认的优先级。内核的调度算法在决定执行哪个线程时会考虑线程的优先级。对于高优先级的线程会有更大的可能执行，相对于低优先级的线程。高优先级并不保证你的线程一定量的执行时间，只是高优先级的线程相对于低优先级的线程更可能被调度而已。

> **重要**: 通常让线程处于默认的优先级是不错的注意。增加线程的优先级同样增加了低优先级线程的饥饿可能。如果你的应用包含必须要相互交互的高优先级和低优先级的线程，那么低优先级的线程可能会阻塞其他线程，创建性能瓶颈。

如果你确实想修改线程优先级，Cocoa 和 POSIX 都提供了这么做的方式。对于 Cocoa 线程，你可以使用 `NSThread` 的类方法 `setThreadPriority:` 来设置当前线程的优先级。对于 POSIX 线程，你可以使用 `pthread_setschedparam` 函数来这么做。更多信息可查看 *NSThread Class Reference* 或 `pthread_setschedparam` man page。

编写线程的入口函数

线程的入口函数在 OS X 和其它系统上大部分是一样的。你初始化你的数据结构，做些工作或可选的设置一个 run loop，当工作完成的时候清理资源。根据你的设计，当你编写你的入口函数时可能有额外的步骤。

创建一个 autorealeasing pool

链接 Objective-C 框架的应用通常必须创建至少一个 autoreleasing pool 在每个线程。如果一个应用使用 managed model —— 应用负责对象的 retain 和 release —— autoreleasing pool 会缓存住任何被 autoreleased 对象。

如果一个应用使用垃圾回收而不是 managed memory model，严格意义上创建 autorelease pool 并不需要。在垃圾回收应用中 autoreleasing pool 的存在并不有害，它们大部分被忽略。一个代码模块同时支持垃圾回收和 managed memory model 的情况是允许的。在这种情况下，一个 autorelease pool 必须存在来支持 managed memory model 代码，并在垃圾回收机制开启的时候自动被忽略。

如果你的应用使用了 managed memory model，创建一个 autorelease pool 应该是线程入口函数应该干的第一件事。同样，销毁这个 autoreleasing pool 应该是你在线程最后做的一件事。这个 pool 保证了 autoreleased 对象被捕获，尽管直到线程退出才释放它们。下面的代码展示了线程入口函数的基本结构。

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

  // Do thread work here.

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

因为顶层的 autoreleasing pool 在线程退出的时候释放它的对象，长时间运行的线程应该创建额外的 autorelease pool 来更频繁的释放对象。例如，一个使用 run loop 的线程可能创建和释放一个 autorelease pool 每次通过 run loop。频繁的时候对象阻止应用的内存足迹增长得太大，增长得太大会引起性能问题。跟其他任何性能相关的行为一样，你应该测量你的代码性能和合适的调节使用 autorelease pools。

关于内存管理和 autorelease pool 的更多信息，请参见 *Advanced Memory Management Programming Guide*

设置一个 exception handler

如果你的应用捕获并处理异常，你的线程代码应该准备好捕获任何可能发生的异常。尽管最好是在异常发生的地方处理异常，但如未能在线程中捕获抛出的线程使得应用奔溃。在线程的入口函数安装一个 try/catch 允许你捕获任何未知的异常，并提供合适的响应。

在你写你的应用时你可以使用 C++ 或 Objective-C 的异常处理风格。关于在 Objective-C 中怎样抛出或捕获异常，可参见 `Exception Programming Topics` 。

设置一个 run loop

当你编写需要在单独线程上执行的代码时，你有两个选择。第一个选择是给这个线程写相应的代码，作为一个长时间很少或没有打断的任务，任务结束的时候线程退出。第二个选择是让你的线程处于一个循环，让它处理动态过来的请求。第一个选择不需要你的代码额外的设置。你只需要启动你需要做的任务就可以。第二个选项涉及到设置好线程的 run loop。

OS X 和 iOS 提供了在每个线程设置 run loops 的内置支持。应用框架自动在主线程的 run loop 开始。如果你创建非主线程，你必须配置相应的 run loop，并手动的启动它。

终止一个线程

终止一个线程的推荐方式是让它自然的到达入口函数的终点。尽管 Cocoa，POSIX 和 Multiprocessing Services 提供了直接杀死线程的 APIs，但强烈不建议使用这些 API。杀死一个线程阻止了线程清理资源。线程分配的内存可能泄露，或其它线程正在使用的资源可能还没来得及被合适的清理，稍后会产生潜在问题。

如果你确实需要在一个操作的中间终止线程的执行，你应该在一开始的时候设计你的线程响应 cancel 或 exit 消息。对于长时间运行的操作，这也许意味着间隔的停止工作，检测是否有收到消息。如果一个消息确实到达，要求线程退出，线程将会有机会来进行需要的清理工作并优雅的退出；没有的话，它简单的回到之前的工作，并处理下段数据。

一种响应 cancel 消息的方法是跑一个 run loop input source 来接受这样的消息。下例代码展示了这样线程的入口函数的基本结构 (例子只展示了 main loop 部分，并不包含设置 autorelease pool 的步骤，和配置实际工作的部分)。样例安装了一个自定义 input source 到 run loop 上，假设可以从另外的线程发送消息。关于怎么设置 input source，看后文。在进行了一部分工作之后，线程简单的抛下 run loop 查看 input source 是否有新消息到达。如果没有的话，run loop 立即退出，loop 接着处理下一段数据。因为 handler 没有本地变量 `exitNow` 的访问权限。退出的条件是通过 thread dictionary 来通知的。

```
- (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:@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];
    }
}
```

Run Loops

Run Loops 是与线程相关的基础架构的一部分。一个 *run loop* 是一个事件处理循环，你使用来调度工作和协调收到的事件。run loop 的目的是有工作做的时候保持线程忙碌，没有事做的时候让线程睡眠。

Run loop 的管理并不是全自动的。你仍然必须设计你的线程在合适的时候启动 run loop 并响应即将到来的事件。Cocoa 和 Core Foundation 都提供了 *run loop* 对象来帮助配置和管理线程的 run loop。你的应用不需要显式的创建这些对象；每个线程，包括应用的主线程，有一个相关联的 run loop 对象。只是非主线程需要显式的跑它们的 run loop。App 框架会自动设置并跑起主线程上的 run loop，作为应用启动的一部分。

下面的部分提供了更多关于 run loops 的信息，怎么为你的应用配置它们。需要查看更多的信息，可以参见 *NSRunLoop Class Reference* 和 *CFRunLoop Reference*.

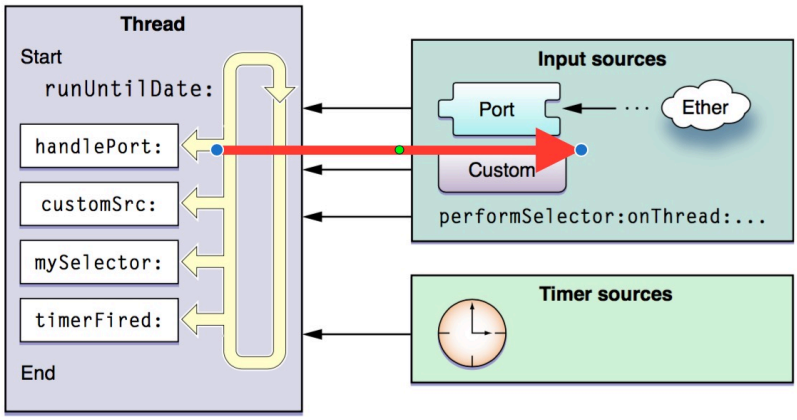
Run Loops 的剖析

run loop 非常的人如其名。它是一个线程进入并调用事件 handler 来响应事件的循环。你的代码提供用来实现 run loop 的 loop 部分的控制语句 —— 换句话就是，你的代码提供了 `while` 或 `for` 循环来驱动 run loop。在你的循环内，你使用一个 run loop 对象来跑收到事件的事件处理代码，并调用安装的 handlers。

一个 run loop 从两种不同类型的 source 接受事件。*input source* 传递异步事件，通常消息来自其它线程或不同应用。*timer source* 传递同步事件，发生在规划的时间点或重复的间隔。两种类型的 source 使用应用特定的 handler 函数来处理这些到来的事件。

下图展示了一个 run loop 和各种 sources 的概念图。Input source 传递异步事件到相应的 handler，导致 `runUntilDate` (在线程关联的 NSRunLoop 对象上执行) 方法退出。Timer sources 传递事件到它们的 handler 函数但不导致 run loop 退出。

Structure of a run loop and its sources



除了处理输入的源外，run loop 会产生关于 run loop 行为的通知。这些通知的观察者可以收到这些通知，使用它们在线程上进行额外的处理。你可以使用 Core Foundation 来安装线程上的 run loop observer。

下面的部分提供了 run loop 组件的更多信息，它们工作的 modes。也描述了在处理事件的不同阶段产生的通知。

Run Loop Modes

一个 *run loop mode* 是一个被监听的 input source 和 timer 的集合和需要被通知的 run loop observers 的集合。每次你跑你的 run loop 的时候，你指定 (显式或隐式的) 特定的 mode 来跑。在 run loop 的这次跑的过程中，只有那些与指定 mode 关联的被监听的 source 允许传递它们的事件 (同样，只有那些与该 mode 关联的 observers 被通知 run loop 的进程)。与其他 mode 相关联的 sources 会维持住任何新事件，直到 run loop 以指定的 mode 通过 loop 的时候才处理。

在你的代码中，你通过名字来标识 modes。Cocoa 和 Core Foundation 定义了一个默认 mode 和几个通用 modes，通过字符串在代码中指定这些 modes。你可以自定义 mode，只需要给自定义的 mode 指定一个自定义字符串就可以。尽管你赋给自定义 mode 的字符串是随意的，但是 mode 的内容却不是。你必须添加一个或多个 input sources，或 run-loop observers 到任何你创建的 mode 上，这样它们才是有用的。

在 run loop 的一个 pass through 中你可以使用 modes 从你不想要的 sources 中过滤出事件。大部分时候，你会想要你的 run loop 对象跑在系统定义的默认模式。然而一个 modal panel 也许跑在 "modal" mode。在这个 mode 时，只有那些和 modal panel 相关的 sources 会给线程传送事件。对于非主线程，你也需要使用自定义的 modes 来阻止低优先级的 sources 在关键操作时传送事件。

> **注意**: modes 的区别在于事件的 source，不在于事件的类型。例如，你不会使用 mode 来单独匹配 mouse-down 事件或键盘事件。你可以使用 modes 来监听不同的 ports，暂停 timers，或改变目前被监听的 sources 和 run loop observers

下表列出了 Cocoa 和 Core Foundation 定义的标准 modes，和什么时候使用这些 mode 的说明。name 列列出了你在代码中真实指定的常量。

Mode	Name	Description
Default	NSDefaultRunLoopMode (Cocoa) kCFRunLoopDefaultMode (Core Foundation)	大部分操作使用这个 mode。大部分时候你使用这个 mode 来启动你的 run loop 和 配置你的 input sources。
Connection	NSConnectionReplyMode (Cocoa)	Cocoa 使用这个 mode 和 `NSConnection` 对象来监听返回。你基本不会自己用到这个 mode。
Modal	NSModalPanelRunLoopMode	Cocoa 使用这个 mode 来标识专用于 modal panels 的事件。
Event tracking	NSEventTrackingRunLoopMode (Cocoa)	Cocoa 使用在 mouse-dragging loops 或其它用户交互的 tracking loop 中使用这个 mode 来限制收到的事件。
Common Modes	NSRunLoopCommonModes (Cocoa) kCFRunLoopCommonModes (Core Foundation)	这是一个可配置的常用 mode group。与这个 mode 相关联的 input source 也会和这个 group 中的 modes 相关联。对于 Cocoa 应用而言，这个集合默认包括 default，modal，event tracking 这些 modes。Core Foundation 初始的时候包含 default 的mode。你可以使用 `CFRunLoopAddCommonMode` 函数添加自定义 mode 到这个集合。

Input Sources

Input sources 传递异步事件到线程。事件的来源依赖于你的 input source 的类型，input source 的类型通常是两类之一。基于 Port 的 input source 监听应用的 Mach ports。自定义 input sources 监听自定义事件来源。你的 run loop 不需要关心 input source 是自定义的还是基于 port 的。系统通常实现了两种类型的 input sources，你可以直接拿来用。这两种类型 input sources 的唯一区别是它们怎么被发送信号了。基于 Port-based input source 自动被内核发送信号，而自定义的 input source 必须手动的从另一个线程发送信号给它。

当你创建一个 input source 的时候，你赋给它一个或多个你的 run loop 的 modes。Modes 会影响在某刻哪些 input sources 被监听。大部分时候，你会将你的 run loop 跑在 default mode，但是你也可以指定自定义 modes。如果你一个 input source 目前没有在被监听的 mode 中，任何它产生的事件会被维持住直到 run loop 跑在正确的 mode 时。

基于 Port 的 Sources

Cocoa 和 Core Foundation 提供了使用 port 相关的函数和对象创建 port-based input sources 的内建支持。例如，在 Cocoa 中，你从不需要直接创建一个 input source。你简单的创建一个 port 对象，使用 `NSPort` 的方法添加这个 port 到 run loop。这个 port 对象会为你负责创建和配置所需的 input source。

在 Core Foundation 中，你必须手动创建 port 对象和它的 input source。两种情况你都需要使用 port opaque type 相关的函数 (`CFMachPortRef` , `CFMessagePortRef` , `CFSocketRef`) 来创建合适的对象。

自定义 Input Sources

创建一个自定义 input source，你必须使用 Core Foundation 中 `CFRunLoopSourceRef` opaque type 相关的函数。你通过使用几个回调函数来创建自定义 input source。Core Foundation 在不同点调用这些毁掉函数配置 input source，处理到达的事件，释放 input source 当 input source 从 run loop 移除的时候。

除了定义事件到来时自定义 input source 的行为外，你也必须定义事件传递的机制。Input source 的这部分跑在单独的线程上，负责给 input source 提供数据，当数据可用的时候发送信号给 input source 来告知它。事件交付机制是由你决定的，但不要太过复杂。

Cocoa Perform Selector Sources

除了基于 port 的 source 外，Cocoa 定义了一个自定义 input source，你可以是用来在任一线程执行一个 selector。跟 port-based input source 相似，执行 selector 的请求在目标线程上是串行的，减去了多个 method 在一个线程上执行时可能发生的同步问题 (请看原文，这里不确定翻译得是否准确，因为一个线程上执行多个方法是没有问题的啊)。跟 port-based source 不同的是，一个 perform selector source 会在它执行了它的 selector 之后从 run loop 中移除自己。

> **注意**: 在 OSX v10.5 之前，perform selector source 大部分用在给主线程发送消息，但是在 OS X v10.5 之后和 iOS 中，你可以使用它们给任何线程发送消息。

当在另一个线程执行 selector 时， 目标线程必须有一个 active 的run loop。对于你创建的线程，这意味着等到你显式的启动 run loop。因为主线程会启动自己的 run loop，因此，在应用调用了application delegate 的 `applicationDidFinishLaunching:` 方法后， 你就可以给主线程发送调用了。run loop 在一次 loop 迭代中处理掉所有排好队的 perform selector calls，而不是每次 loop 迭代只处理一个。

下表中列出了 `NSObject` 上可以用来在其它线程上 perform selector 的方法。因为这些方法在 `NSObject` 上申明，你可以在任何可以访问 Objectvie-C 对象的地方使用这些方法，包括 POSIX 线程。这些方法并不创建线程类执行 selector。

Methods	Description
`performSelectorOnMainThread: withObject: waitUntilDone:` `performSelectorOnMainThread: withObject:waitUntilDone:modes:`	在应用的主线程上的下次 run loop 迭代中执行指定的 selector。这些方法允许你选择是否阻塞当前线程直到 selector 被执行。
`performSelector: onThread:withObject: waitUntilDone:` `performSelector: onThread:withObject:waitUntilDone:modes:`	在 `NSThread` 对象指定的线程上执行指定的 selector。这些方法允许你选择是否阻塞当前线程直到 selector 执行完。
`performSelector: withObject: afterDelay:` `performSelector: withObject: afterDelay:inModes:`	在当前线程的下次 run loop 迭代中执行指定的 selector，可以指定可选的 delay 时间。因为它会等到下次 run loop 迭代，这些方法天然的有一个 mini delay 相对于目前执行的代码。多个排好队的 selectors 以它们排队的顺序执行。
`cancelPreviousPerformRequestsWithTarget:` `cancelPreviousPerformRequestsWithTarget:selector:object:`	让你取消使用 `performSelector: withObject: afterDelay:` 或 `performSelector: withObject: afterDelay:inModes:` 发送给一个线程的消息。

Timer Sources

Timer source 在未来的某个时刻发送同步事件到线程。Timers 是一种线程通知自己做些什么的一种方式。例如，一个搜索框使用一个 timer 来发起自动的搜索，一旦用户输入超过一段时间。这样的延迟给用户足够多的时间输入足够多的字符。

尽管它产生基于时间的通知，一个 timer 并不是一个实时的机制。跟 input source 一样，timers 是跟特定的 run loop modes 相关联的。如果一个 timer 不在 run loop 目前监听的 modes 中时，它会等待 run loop 跑在 timer 关联的 modes 时才 fire。同样，如果你一个 timer fire 了，但是 run loop 正在执行一个 handler 函数的过程中，timer 会等到 run loop 的下次迭代，再调用它的 handler 函数。如果 run loop 不运行了，timer 永远都不会 fire。

你更可以配置 timer 单次或重复的产生事件。一个重复的 timer 自动的基于调度的 fire time 来重新规划它的 fire 时间，而不是真正的 fire time。例如，一个 timer 规划在某个时刻和这个时刻之后的每 5 秒 fire，那么规划的 firing time 总会在原来的 5 秒间隔，即使真正的 firing time 被延迟了。如果 firing time 被延迟得超过了好几个规划的 firing times 的话，timer 对于错过的时间段只会 fire 一次。在为错过的时间 fire 之后，timer is rescheduled for the next scheduled firing time.

Run Loop Observer

相对于当异步或同步事件发生的时候 fire 的 sources，run loop observers 是 run loop 自身执行的特殊位置 fire。你可能使用 run loop observers 来准备好线程处理一个给定的事件，或准备好线程进入睡眠。你可以关联 run loop observers 到 run loop 的以下事件：

- 进入 run loop
- Run loop 准备开始处理一个 timer
- Run loop 准备开始处理一个 input source
- Run loop 准备进入睡眠状态
- 当 Run loop 被唤醒时，但在处理唤醒它的事件前
- 从 run loop 退出

你可以使用 Core Foundation 添加 run loop observers。要创建一个 run loop observer，你可以创建一个 `CFRunLoopObserverRef` opaque type 的实例。这个类型持有你自定义的回调函数和你敢兴趣的行为的记录。

和 timers 相似，run-loop observers 可以被一次或重复的使用。一个一次的 observer 会在 fire 后从 run loop 移除，然而一个重复的 observer 会持续附着在 run loop 上。在创建它的时候你指定一个 observer 是否跑一次还是重复的跑。

Run Loop 事件的顺序

每次你跑 run loop，线程的 run loop 处理待处理的事件，给任 observers 产生通知。这些的顺序如下：

1. 通知 observers run loop 已经进入
2. 通知 observers 任何 timers 准备 fire
3. 通知 observers 任何不是基于 port 的 input source 准备 fire 了
4. Fire 任何非基于 port 准备好 fire 的 input sources
5. 如果一个基于 port 的 input source 准备好并等待 fire 的话，立即处理事件，跳到步骤 9

6. 通知 observers 线程将要睡眠
7. 将线程置为睡眠状态，直到以下事件发生：
    - 一个给 port-based input source 的事件到达
    - 一个 timer fires
    - 给 run loop 设置的 timeout 值超时了
    - run loop 被显式的唤醒
8. 通知 observers 线程刚被唤醒
9. 处理待处理的事件
    - 如果用户定义的 timer fired，处理 timer 事件并重启 loop。跳到步骤 2
    - 如果一个 input source fired，处理 (deliver 原文) 事件
    - 如果 run loop 显示的被唤醒但是还没有 time out，重启 loop。跳到步骤2
10. 通知 observers run loop 已经退出

因为 timer 和 input sources 的 observer 通知在事件发生之前被传递，可能在通知的时间和事件发生的时间之间有时间间隙。如果在这些事件之间的计时很重要的话，你可以使用 sleep 和 awake-from-sleep 通知来帮助纠正真实时间的计时。

因为 timers 和其他间隔事件是当你跑 run loop 的时候传递，绕过 run loop 会干扰这些事件的传递。这种行为发生的典型例子是你实现一个鼠标追踪的程序时，你的代码在进入 loop 中后，不断的重复的从应用中请求事件。因为你的应用直接拿取事件，而不是让应用正常的分发这些事件，active timers 将不能正常的 fire 直到你追踪鼠标的代码退出，并将控制返回给应用。

一个 run loop 可以显式的唤醒 run loop 对象。其它时间可能引起 run loop 被唤醒。例如，添加另一个不基于 port 的 input source 会唤醒 run loop，从而 input source 可以立即被处理，而不是等待直到其它的事件到达。

什么时候使用一个 Run Loop

唯一需要显式跑一个 run loop 的时候是你为你的应用创建非主线程的时候。应用主线程的 run loop 架构中的关键部分。结果是，app 框架提供了自动跑起主线程 run loop 的代码。 `UIApplication` 的 `run` 方法启动了应用主线程的 run loop，作为应用启动的一部分。如果你使用 Xcode 模板项目来创建你的应用，你应该永远都不会显式调用这些函数。

对于非主线程，你需要决定一个 run loop 是否必要，如果是的，配置并启动它。你并不是总是需要启动线程的 run loop。例如，如果你使用一个线程执行一些长时间的具体任务，你可能会避免启动一个 run loop。Run loop 适用于你想要跟线程有更多交互的场景。例如，你需要启动一个 run loop如果你计划做以下事之一：

- 使用 ports 或自定义 input sources 来与其它线程通讯
- 在线程上使用 timers
- 在 Cocoa 应用中使用任何 `performSelector...` 方法
- 维持住线程来进行间隔的任务

如果你确定使用 run loop，配置和启动是很直接的。跟所有的多线程编程一样，你应该计划在合适的时机退出线程。干净的结束线程总是比暴力的终止线程好。

使用 Run Loop 对象

一个 run loop 对象提供了添加 input sources，timers，和 run-loop observers 到你的 run loop，然后跑起它的接口。每个线程有一个 run loop 对象与之关联。在 Cocoa 中，这个对象是 `NSRunLoop` 类的实例。在应用的底层，它是一个指向 `CFRunLoopRef` opaque type 的实例。

获取一个 Run Loop 对象

要获取当前线程的 run loop，你使用以下方法之一：

- 在一个 Cocoa 应用中，使用 `NSRunLoop` 的 `currentRunLoop` 类方法获取 `NSRunLoop` 对象。
- 使用 `CFRunLoopGetCurrent` 函数

尽管它们不是 toll-free bridged 类型，你可以从一个 `NSRunLoop` 对象中获得一个 `CFRunLoopRef` opaque type。 `NSRunLoop` 类定义了一个 `getCFRunLoop` 方法返回一个 `CFRunLoopRef` 类型，你可以使用来传递给 Core Foundation 函数。这两个对象指向相同的对象，你可以根据需要混合的使用 `NSRunLoop` 对象和 `CFRunLoopRef` opage type.

配置 Run Loop

你在非主线程跑起一个 run loop 之前，你必须添加至少一个 input source 或 timer 到 run loop 上。如果一个 run loop 没有监听任何 sources，当你试着跑它的时候它会立马退出。怎么添加 input source 到 run loop， 请看后文。

除了安装 sources 外，你也可以安装 run loop observers，使用它们来检测 run loop 的不同执行阶段。要安装一个 run loop observer，你创建一个 `CFRunLoopObserverRef` opaque type，使用 `CFRunLoopAddObserver` 添加到 run loop 上。Run loop observers 必须要通过 Core Foundation 来创建，即使是 Cocoa 应用。

下例展示了线程的主函数，它添加一个 run loop observer 到它的 run loop。这个例子的目的是给你展示怎么创建一个 run loop observer 来监听 run loop 的行为。这些基本的 handler 函数只是简单的将 run loop 的行为打下 log。

```
- (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, &myRu
nLoopObserver, &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);
}
```

当给一个常驻的线程配置 run loop 的时候，至少添加一个 input source 来接受消息会更好。尽管只有一个 timer 附着在 run loop 上时，你可以可以进入 run loop，但是一旦 timers fires，通常它变为 invalidated，这会导致 run loop 退出。附着一个重复的 timer 可以保证你的 run loop 执行很长的一段时间，但会涉及到间隔性的 firing timer 来唤醒线程，本质上是另一种轮询的形式而已。对比下一个 input source 等待事件发生，保持线程 sleep 直到事件发生。

启动 Run Loop

启动一个 run loop 只对应用中的非主线程是必须的。一个 run loop 必须有一个 input source 或 timer 来监听。如果没有一个附着，run loop 会立即退出。

有几种方式启动 run loop，如下：

- 无条件的
- 设置一个 time out
- 进入一个指定的 mode

unconditionally 进入 run loop 是最简单的选项，但也是你最不想用的。无条件的跑你的 run loop 会使线程进入永久的循环，给你很少对 run loop 自身的控制。你可以添加和移除 input sources 和 timers，但是停止 run loop 的唯一方式是杀掉它。同样也没有方式将 run loop 跑再自定义的 mode。

与其 unconditionally 的跑一个 run loop，设置一个 timeout 来跑 run loop 会更好。当你使用一个 timeout 值时，run loop 在事件到来或被分配的时间过期之前一直跑。如果事件到来，事件被分发给 handler 处理，然后 run loop 退出。你的代码可能重新启动 run loop 来处理下一个事件。如果被分配的时间过期了，你可以简单的重启 run loop 或是用这个时间值做些必须的记录。

除了设置 timeout 外，你也可以在指定的 mode 来跑 run loop。 Modes 和 timeout 值并不相互排斥，可以同时使用来跑一个 run loop。Modes 限定了传递事件给 run loop 的 sources 类型。

下例展示了一个线程入口函数的基本结构。例子的关键部分展示 run loop 的基本结构。本质上，你添加 input sources 和 timers 到 run loop，然后重复调用一个 run loop 的函数之一来启动 run loop。每次 run loop 函数返回的时候，你检查条件是否满足退出线程。例子使用 Core Foundation run loop 函数以便检查返回值，决定 run loop 为什么退出。如果你用 Cocoa 且不需要检查返回值的话你也可以使用 `NSRunLoop` 的类似的方法。

```
- (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.
}
```

可以嵌套的 run 一个 run loop。也就是说，你可以从 input source 或 timer 的 handler 函数中调用 `CFRunLoopRun`，`CFRunLoopRunInMode`，或任何 `NSRunLoop` 的跑 run loop 的方法。当你这么做的时候，你可以使用任何 mode 来跑嵌套的 run loop，包括被外围 run loop 使用的 mode。

退出 Run Loop

在 run loop 处理一个事件前，有两种方式来是一个 run loop 退出:

- 使用一个 timeout 值来配置 run loop
- 告诉 run loop 停止

使用一个 timeout 的值当然更受偏好，如果你可以管理 run loop 的话。指定一个 timeout 的值使得 run loop 完成它所有的正常处理流程，包括在退出前传递通知到 run loop observers。

使用 `CFRunLoopStop` 函数显式停止 run loop 产生与 timeout 值相似的结果。run loop 发送出任何剩余的 run-loop notifications 然后退出。不同点是你可以使用这个在那些你 unconditionally 启动的 run loop 上。

尽管移除一个 run loop 的 input sources 和 timers 同样可能使得一个 run loop 退出，但这不是一个可靠的方式停止一个 run loop。一些系统函数添加 input source 到 run loop 来处理所需的事件。因为你的代码可能不知道这些 input sources，这样将不能移除它们，这也就阻止了 run loop 退出。

线程安全和 Run Loop 对象

线程安全依赖于你使用哪个 API 来操作你的 run loop。在 Core Foundation 中的函数通常是线程安全的，可以从任何线程中调用。然而如果你进行改变 run loop 配置的操作，在那些拥有 run loop 的线程上仍然是更推从的。

Cocoa `NSRunLoop` 类本身不是线程安全的。如果你使用 `NSNunLoop` 类来修改你的 run loop，你应该在拥有 run loop 的线程上做。添加 input source 或 timer 到一个属于另一个线程的 run loop 可能导致你的应用奔溃或有异常行为。

配置 Run Loop Sources

下面的部分介绍了在 Cocoa 和 Core Foundation 中怎么配置不同类型的 input source。

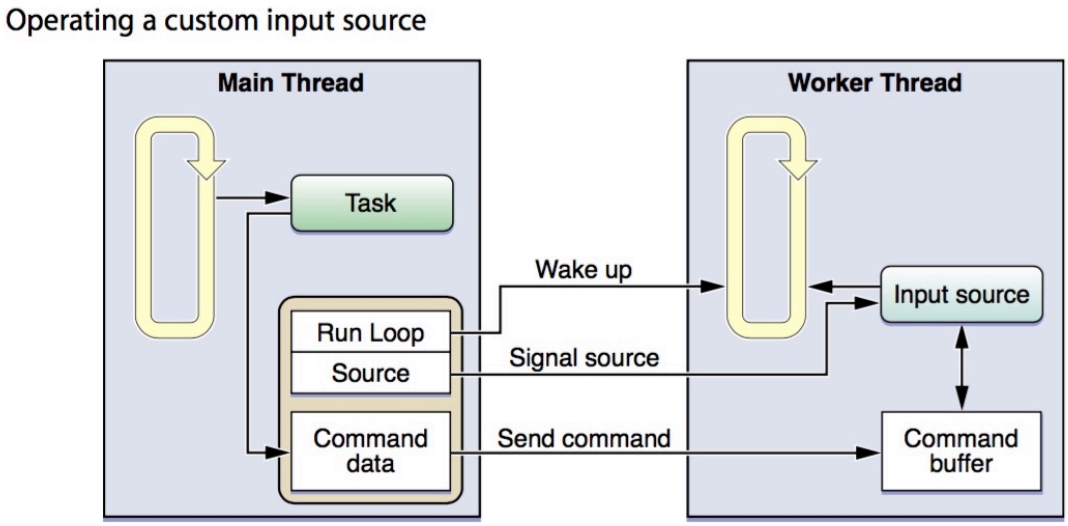
定义一个自定义 Input Source

创建一个自定义 input source 涉及定义以下：

- 你想要你的 input source 处理的信息
- 一个 scheduler routine 让感兴趣的客户知道怎样联系你的 input source
- 一个处理用户发送的 perform request 的 handler
- 一个 cancellation 函数 invalidate 你的 input source

因为你创建一个自定义 input source 来处理自定义信息，真正的配置被设计得很灵活。scheduler，handler，cancellation handler 函数你创建自定义 input source 必须的关键函数。然后 input source 的部分其它行为发生这些函数之外。例如由你定义传送数据到 input source 的机制，和通知其它线程你的 input source 的存在。

下例展示了一个自定义 input source 的样例配置。在这个例子中，应用的主线程保持了这个 input source 的引用，给 input source 自定义的 command buffer，和 input source 被安装在哪个 run loop。当主线程有任务交付给 work 线程时，它发送一个 command 到 command buffer，command 包含 worker 线程开始工作所需的信息。(因为主线程和 worker 线程的 input source 都有 command buffer 的访问权限，它们的访问必须同步) 一旦 command 被发送，主线程给 input source 发送信号，唤醒 worker 线程的 run loop。一旦收到唤醒的命令，run loop 为 input source 调用 handler，处理 command buffer 中的 command。



下面的部分解释了上图种的自定义 input source 的实现，展示了你需要实现的核心代码。

定义 Input Source

定义一个自定义 input source 需要使用 Core Foundation 函数来配置你的 run loop source，并把它附着到 run loop。尽管基本的 handlers 是基于 C 的函数，但这并不阻止你使用 Objective-C 或 C++ 实现这些函数的 wrappers。

上图中的 input source 使用 Objective-C 对象来管理一个 command buffer，和协调 run loop。下面的代码展示了这个对象的定义。 `RunLoopSource` 对象管理一个 command buffer 和使用 buffer 来从其它线程接受消息。代码也展示了 `RunLoopContext` 对象的定义，这个对象只是一个 container 对象，使用来传递 `RunLoopSource` 对象和一个 run loop 引用到应用的主线程。

```
@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 Objective-C 代码管理 input source 的自定义数据，但将 input source 附着到一个 run loop 需要基于 C 的函数回调。当你添加 run loop source 到 run loop 时，这些函数中的第一个被调用。因为这个 input source 只有一个 client (主线程)，它使用 scheduler 函数使用主线程上的 application delegate 发送一个消息来注册自己。当 application delegate 想要与 input source 通讯的时候，它使用 `RunLoopContext` 对象中的信息来达到母的。

```
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];
}
```

最重要的回调函数之一是当你的 input source 被发送信号，用来处理自定义数据的那个。下例展示了与 `RunLoopSource` 对象关联的进行回调函数。这个函数简单的转发做相关工作的请求到 `sourceFired` 方法，然后处理任何在 command buffer 中的 command。

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

如果你有需要从 run loop 移除你的 input source 的话，使用 `CFRunLoopSourceInvalidate` 函数，系统调用你的 input source 的 cancellation 函数。你可以使用这个函数通知 input source 的使用者你的 input source 不再是 valid，并应该移除任何 input source 的引用。下例中展示 cancellation 回调函数。这个函数给 application delegate 发送另一个 `RunLoopContext` 对象，但是这次要求 delegate 移除这个 run loop source 的引用

```
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];
}
```

安装 Input Source 到 Run Loop

下例展示了 `RunLoopSource` 的 `init` 和 `addToCurrentRunLoop` 方法。 `init` 方法创建真实被添加到 run loop 的 `CFRunLoopSourceRef` opaque type。init 方法中它将自身传递给 contextual information，以便回调函数有指向它的指针。 input source 的安装直到 worker 线程调用 `addToCurrentRunLoop` 方法，在这时 `RunLoopSourceScheduleRoutine` 回调函数被调用。一旦 input source 被添加到 run loop，跑起 run loop 的线程进入等待。

```
- (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);
}
```

Input Source 的 Clients 间的协调

为了你的 input source 是有用的，你需要从另外的线程操作它并给它发送信号。input source 的整个作用是使他关联的线程睡眠到有事可做。这样的话，让应用中其它的线程知道你的 input source 并有一种与之通讯的方式下显得很有必要。

一种通知 input source 用户的方式是当 input source 一开始添加到 run loop 的时候，发出注册信息。你可以注册任意多 clients，或者简单的到中心代理那里注册下，由代理向 input source 的 clients 提供它。下例中展示了由 application delegate 定义的注册方法。由 `RunLoopSource` 对象的 scheduler 函数调用。这个方法收到一个由 `RunLoopSource` 提供的 `RunLoopContext` 对象。下例也展示了当 input source 从 run loop 移除时用来取消注册的方法。

```
- (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];
    }
}
```

给 Input Source 发送信号

在给 input source 传送数据后，一个 client 需要给 input source 发送信号，唤醒它的 run loop。给 input source 发送信号使得 run loop 知道 input source 准备好被处理。因为信号发生的时候线程是睡眠的，你总是应该显式唤醒 run loop。如果没这么做的话会导致处理 input source 的延迟。

下例中展示了 `RunLoopSource` 对象的 `fireCommandsOnRunLoop` 。当 clients 已经准备好 input source 来处理它们添加到 command buffer 的 commands 的时候它们调用这个方法。

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

> **注意**: 你永远不应该通过给一个自定义的 input source 发送信号来处理 `SIGHUP` 或其它进程级别的信号。Core Foundation 唤醒 run loop 的函数并不是 signal 安全的，不应该在应用的 signal handler 函数中使用。关于 signal handler 函数更多的信息，可以参见 *sigaction* man page。

配置 Timer Sources

创建一个 timer source，所有你需要做的是创建一个 timer 对象，然后在 run loop 上 schedule 它。在 Cocoa 中，使用 NSTimer 类来创建新的 timer 对象，在 Core Foundation 中，你使用 `CFRunLoopTimerRef` opaque type。 `NSTimer` 的内部只是一个 Core Foundation 的简单扩展，提供了一些易用的特性，像使用一个方法创建和 schedule 一个 timer。

在 Cocoa 中，你可以同时创建和 schedule 使用以下任一方法：

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

这些方法创建一个 timer 并添加它们到当前线程的 run loop，与 default mode 关联。你也可以手动 schedule 一个 timer 如果你想通过创建你的 `NSTimer` 对象，然后使用 `NSRunLoop` 的 `addTimer:forMode:` 方法。两种技术基本上做的是同样的事情，但是给了你不同层面配置 timer 的选择。例如，如果你创建一个 timer，并手动的添加到 run loop，你可以使用一个非 default 的一个 mode。下例展示怎么使用两种技术创建 timer。第一个 timer 延迟 1s 后每隔 0.1s fire。第二个 timer 初始延迟 0.2s 后每隔 0.2s fire。

```
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(myDoFireTim
er1:) 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];
```

接下来看看怎么使用 Core Foundation 来配置一个 timer。尽管这个例子并没有在 context structur 中传递任何用户定义的信息，你可以使用这个结构来传递你的 timer 所需的任何自定义数据。更多的信息可以参见 *CFRunLoopTimer* reference.

```
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);
```

配置一个基于 Port 的 Input Source

Cocoa 和 Core Foundation 都提供了基于 port 的对象来支持线程或进程间的通讯。下面的部分将给你展示怎么使用不同类型的 port 设置 port 通讯。

配置一个 `NSMachPort` 对象

为了使用 `NSMachPort` 对象建立一个本地连接，你创建一个 port 对象，添加它到线程的 run loop 上，当你启动你的非主线程的时候，传递同样的对象给你线程的入口函数。副线程使用同样的对象给你的主线程发送消息。

实现主线程的代码

下面的代码展示了主线程起一个非主线程作为 worker 线程。因为 Cocoa 框架进行许多配置 port 和 run loop 的中间步骤， `launchThread` 方法要比使用 Core Foundation 实现同样的功能要短很多。Core Foundation 的实现传递 local port 的名字给 worker 线程，而 Cocoa 传递一个 `NSPort` 对象。

```
- (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:my
Port];
    }
}
```

为了设置成线程间双向的通讯，你也许想要 worker 线程在 check-in 的消息中把它自己的 local port 告诉主线程。收到 check-in 消息让主线程知道在启动 worker 线程的过程中一切正常，同时也给了一种方式发送进一步的消息给 worker 线程。

下面的代码展示了主线程的 `handlePortMessage:` 方法。这个方法当有数据到达线程自己的 local port 时被调用。当一个 check-in 消息到达的时候，方法直接从 port message 中获取 worker 线程的 port，然后本地保存。

```
#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.
    }
}
```

实现非主线程代码

对于非主线程，你必须配置它使用指定的 port 来传回信息给主线程。

下面展示了设置 worker 线程的代码。给线程创建了 autorelease pool 后，接着创建了一个 worker 对象来驱动线程的执行。worker 对象的 `sendCheckinMessage:` 方法给 worker 线程创建一个 local port 和发送一个 check-in 消息回主线程。

```
+(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];
    [workerObj sendCheckinMessage:distantPort];

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

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

当使用 NSMachPort 时，local 和 remote 线程可以使用同一个 port 对象来进行线程间的单向通讯。也就是说，一个线程创建的 local port 成了其它线程的 remote port。

下面的代码展示了 worker 线程的 check-in 函数。这个方法设置好用于未来通讯的 local port，然后发送一个 check-in 的消息回主线程。发送使用 `LaunchThreadWithPort:` 函数收到的 port 对象作为消息的目标。

```
// 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]];
    }
}
```

配置一个 NSMessagePort 对象

要使用一个 `NSMessagePort` 对象创建一个本地连接，你不能简单的在线程间传递 port 对象。Remote message ports 必须通过名字来获得。在 Cocoa 中为了可能通过名字获得 message port，需要使用指定的名称注册你的 local port，然后传递给想获取合适的 port 对象进行的通讯的 remote 线程。下面的代码展示了在你想要使用 message ports 时创建和注册 port 的过程。

```
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 中配置一个基于 Port 的 Input Source

这部分展示了怎么使用 Core Foundation 在应用的主线程和一个 worker 线程间设置一个双向的通讯通道。

下面的代码被应用的主线程调用来启动一个 worker 线程。代码做的第一件事是设置一个 `CFMessagePortRef` opaque type 来监听 worker 线程发送过来的消息。worker 线程需要 port 的名字来建立连接，所以名字字符串被传递给 worker 线程的入口函数。Port names 应该在当前用户的上下中是唯一的，不然你可能会遇到冲突。

首先添加 Core Foundation Message Port 到一个线程

```
#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);
}
```

port 被安装和线程启动后，主线程在等待 check-in 消息时可以继续正常的执行。当 check-in 消息到达的时候，它被分发给主线程的 `MainThreadResponseHandler` 函数。函数代码如下，它取出给 work 线程的 port 名字，创建一个 conduit 以便将来通讯。

```
#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，剩下的唯一需要做的是给新建的线程创建它自己的 port 和 check in。下面的代码展示了 worker 线程的入口函数。函数取出主线程的 port 名称，是用来创建一个创建一个 remote 连接回主线程。然后然后给自己创建一个 local port，安装 port 到线程的 run loop 上，发送 check-in 包含 local port 的消息回主线程。

```
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, &shouldFreeIn
fo);

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

    CFRunLoopSourceRef rlSource = CFMessagePortCreateRunLoopSource(NULL, myPort,
    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();
}
```

同步

应用中多线程的出现引起了潜在的与多线程访问共享资源相关的问题。两个线程同时修改相同的资源可能以你不想的方式相互干扰。例如，一个线程也许覆盖了另一个线程的修改，或将应用置于一种未知或可能的无效状态。如果你运气好，毁坏 (corrupted) 的资源也许引起明显的性能问题或应用奔溃，这些问

题比较好解决。如果你运气不好，这种错误可能引起微妙的错误，这些微妙的错误直到很晚的时候才显现出来，或这些错误需要大量的对你的代码检查。

当设计到线程安全的时候，你能有的最好的保护是一个好的设计。避免共享资源和最小化线程间的交互使得这些线程有更小的可能相互干扰。一个完全没有交互的设计并不总是可行的。在你的线程必须交互的场景，你需要使用同步工具来保证安全性。

OS X 和 iOS 提供一些同步工具供你使用，这些工具从提供互斥访问到正确的给应用中的事件排序。下面的部分描述了这些工具和你怎么在代码中使用它们来影响对程序资源的访问。

同步工具

为了阻止不同的线程意外的修改数据，你要么设计你的应用没有同步问题，要么你使用同步工具。尽管完全避免同步问题是更受欢迎的，但是并不总是可行。下面的部分描述了你可以使用的同步工具分类。

原子操作

原子操作工作在简单数据类型上的简单同步形态。原子操作的好处是它们不阻塞竞争的线程。对于简单的操作，如递增一个计数变量，使用它们可以获得更好的性能相对于使用锁。

OS X 和 iOS 包含大量的操作来进行基本的数学和逻辑操作在 32 位和 64 位的值上。在这些操作中有 compare-and-swap、test-and-set, and test-and-clear 操作的原子版。要查看所有支持的原子操作的话，可以看 `/usr/include/libkern/OSAtomic.h` 头文件或 `atomic` man page。

Memory Barriers 和 Volatile Variables

为了取得最佳的性能，编译器经常重排汇编级别的指令来使得给处理器的指定管道尽量是满的。作为这种优化的一部分，编译器也许重排方位主内存的指令当它认为这么做不会产生不正确的数据。不行的是，对于编译器并不是总是可能检测到所以依赖于内存的操作。如果看上去相互独立的变量却实际相互影响，compiler 优化可能以一种错误的方式更新这些变量，从而产生潜在不正确的结果。

一个 memory barrier 是一种 nonblocking 同步工具，被用来保证内存访问操作以正确的顺序发生。一个 memory barrier 像一个栅栏样行为，强行让处理器完成任何在它之前的加载和保存操作，在它允许进行任何在它之后的加载和存储操作。Memory barriers 通常被用来保证一个线程 (对另一个线程可见) 的内存操作总是以期望的顺序发生。这种情况下缺少一个 memory barrier 的话允许会使一个线程看到看起来不可能存在的结果。(要查看例子的话，可以查看 Wikipedia 条目 memory barriers) 要使用一个 memory barrier，你简单的在你代码合适的点调用 `OSMemoryBarrier` 函数。

Volatile 变量会应用另一种形式的内存限制到单个变量上。编译器通常通过加载变量值到寄存器来优化代码。对于局部变量，这通常不是一个问题。如果一个变量对于另一个线程可见，这样的优化可能阻止其它的线程注意到任何对这个变量的修改。使用 `volatile` 关键字到一个变量强制编译器每次使用变量的时候从内存加载。你也许声明一个变量为 `volatile` 如果它的值可以被编译器不能检测到的外在源所修改。

因为 memory barries 和 volatile 变量都是减少编译器的优化，你应该很少使用它们，只在需要保证正确性的地方。关于使用 memory barriers 的更多信息，可以参见 `OSMemoryBarrier` man page。

锁

锁是最普遍使用的同步工具之一。你可以使用锁来保护你代码的 `critical section`，这段代码在任一时刻只有一个线程被允许访问。例如，一个 critical section 也许操作一个特殊的数据结构或使用某些在每一刻只允许一个 client 使用的资源。通过使用锁包围这段代码，你阻止其它线程做可能影响你代码正确性的修改。

下表列出了程序员常用的锁。OS X 和 iOS 提供了这些锁类型的大部分实现。对于不支持的锁类型，描述列解释了为什这些锁没有在这些平台直接实现的原因。

Lock	Description
Mutex	一个 mutually exclusive (or mutex，互斥) lock 行为像一个资源周边的栅栏。一个互斥锁是一种任一时刻只允许一个线程访问的 semaphore。如果一个互斥锁正在被用，另一个线程想要获得它，这个线程会阻塞直到互斥锁被原来的拥有者释放。如果多个线程竞争同一个互斥锁，任一时刻只有一个线程被允许访问它。
Recursive lock	一个 recursive lock 是互斥锁的一种变种。一个 recursive lock 允许一个线程在释放锁之前多次获得这个锁。其它的线程保持阻塞直到锁的拥有者释放所的次数跟获得锁的次数一样。Recursive lock 主要用在递归迭代，但也可以使用在多个方法每个都需要单独获得锁。
Read-write lock	一个 read-write lock 也被称作一个 shared-exclusive lock。这种类型的锁通常用在大规模操作并可以显著提升性能，如果被保护的数据结构被频繁的读取，并偶尔被写入。在正常的操作过程中，多个 readers 可以同时访问数据结构。当一个线程想要写入数据到数据结构，它会阻塞到所有的读线程释放锁，在这刻写线程获得锁，并可以更新数据结构。当一个写线程在等待锁时，任何新的读线程阻塞直到写线程完成写操作。系统只支持 read-write locks 使用 POSIX 线程。想要查看更多关于怎么使用这些锁，查看 *pthread* man page.
Distributed lock	一个 distributed lock 在进程的级别提供互斥访问。不像一个真实的 mutex，一个 distributed lock 不会阻塞一个进程或阻止它运行。当锁忙时它只是简单的报告下，让进程自行决定怎么处理。
Spin lock	一个 spin lock 会轮询它的 lock condition 直到 lock condition 可用。Spin locks 最长用在多核系统并且等待锁的期望时间不长。在这些场景，轮询比阻塞线程更高效，阻塞线程涉及到上下文更换和线程数据结构的更新。系统没有提供任何 spin locks 的实现因为它们轮询的性质，但是在特定的场景你可以简单的实现它们。关于实现内核中的 spin locks 更多信息，查看 *Kernel Programming Guide *。
Double-checked lock	一个 double-checked lock 是一种通过在获得锁之前测试 locking criteria 来减少获得锁的负载。因为 double-checked locks 是潜在不安全的，系统没有提供对它们的显式支持，它们的使用也是不鼓励的。

Conditions

一个 condition 是另一种形式的 semaphore，允许线程在特定条件满足的时候相互间发送信号。Conditions 通常用来标一个资源的可用性或保证任务按照指定的顺序执行。当一个线程测试一个 condition 时，除非 condition 已经是 true，它才不会阻塞。它会保持阻塞直到其他线程改变并给 condition 发送信号。一个 condition 和 一个 metux 的区别是多个线程可能被允许同时访问同一个 condition。conditon 更多像是一个守门员，依赖于指定的方案让不同的线程通过大门。

一种你使用 condition 的方式是管理一些待处理的事件。这个事件队列使用一个 condition 变量来给线程发送信号代表队列中有事件。如果一个事件到达，队列会合适的给 condition 发送信号。如果一个线程已经在等待，它将被唤醒，从队列中取出事件并处理。如果两个事件几乎同时到达队列，这个队列会给 condition 发两次信号，唤醒两个线程。

系统在几个不同的技术中提供了对 condition 的支持。conditon 的正确实现需要谨慎的编码。

Perform Selector Routines

Cocoa 应用哟一个方便的方式同步的给一个线程发送消息。`NSObject` 类申明了可以在应用的 active 线程执行一个 selector 的方法。这些方法让你的线程异步的传递消息，并保证这些消息同步的被目标线程执行。例如，你也许会使用 perform selecotr messages 来传递结果到应用的主线程，或一个指定协作线程。每个 request to perform a selector 在目标线程的 run loop 上排队，然后 requests按它们被收到的顺序执行。

同步消耗和性能

同步帮助保证代码的正确性，但是这么做会损耗性能。同步工具的使用引入了延迟，即使未竞争的场景。锁和原子操作通常涉及到使用 memory barriers 和内核级别的同步来保证被正确的保护。如果有对所的竞争，你的线程可能 阻塞，经历更大的延迟。

下表列出了 mutexes 和 原子操作在未竞争的情况下大概的消耗。这些测量代表了几千次取样的平均值。但是跟线程创建时间一样，mutex acquisition 时间在不同的处理器负载，计算机速度，系统可用资源和程序内存下会有很大的差别。

Item	Approximate cost	Notes
Mutex acquisition time	大概 0.2 ms	这是在未竞争条件下的 lock acquisition time。如果锁被另外的线程持有的话，acquisition time 会更大。这个数据会随着测量的手段和测试的中值。
Atomic compare-and-swap	大概 0.05 ms	同上

当你设计你的并发任务时，正确性永远是最重要的因素，但是你也应该考虑性能因素。正确的并发代码性能不如同样的代码单线程执行的话，很难说是一种

增进。

如果你在重构你现有的饿单线程应用，你应该对这些关键任务做些基准测量。一旦添加了多线程，你应该给这些任务重新测量性能数据，然后对比多线程和单线程不场景的数据。如果在你调节代码之后，多线程没有提升性能，你也许想要重新考虑你的实现或是否要使用线程。

线程安全和信号

对于多线程应用，没有什么比处理信号的问题更让人恐惧或困惑。Signal 是 BSD 传递信息给进程或操作它的底层机制。一些程序使用信号来检测一些事件，如子进程的死亡。系统使用 signals 来终止失控的进程和传递其他类型的信息。

Signal 的问题不是它们做了什么，而是当你的应用有多个线程时它们的行为。在一个单线程应用中，所有的信号 handler 在主线程上执行。对一个多线程应用中，不是跟特定硬件错误绑定的 signals 被传递给任意正在运行的线程。如果多个线程正在运行的话，signal 被传给任意系统选择的线程。换句话说，signals 可以被传递给应用的任一线程。

实现 signal handler 的第一条规则是避免对对用 handler 的线程有任何假设。如果一个特定的线程想处理 signals，你需要找到一种方式通知线程，当信号发生的时候。你不能假设从那个线程安装信号处理器的话就会导致 signal 被传到相同的线程。

关于信号和安装信号处理器的更多信息，可以查看 `signal` 和 `sigaction` man page.

Tips for Thread-Safe Designs

同步工具是保证线程安全的有效工具，但是它们不是万灵药。使用过多的话，锁或其他类型的同步原语可能会降低多线程的性能，甚至比它的单线程版本还低。找到安全和性能的平衡点是一种经验艺术。下面的部分提供一些 tips 来帮助你选择合适的同步的级别。

Avoid Synchronization Altogether

对于你工作的新项目，即使是现存的项目，设计你的代码和数据结构来避免同步的需求是可能最好的方案。禁锢锁和其他同步工具很有用，它们会影响应用的性能。如果总的设计会导致指定资源的高竞争，你的线程可能会等待得更久。

实现并发的最好方式是减少并发任务的交互和内在依赖。如果每个任务在它自己的数据集上操作，它就不需要使用锁来保护数据。即使在两个任务确实共享一个数据集的时候，你也可以找到找到方式将数据集分区，或给每个任务提供它自己的 copy。当然，拷贝数据集有它自己的消耗，所以你需要权衡这些消耗和同步的消耗。

Understand the Limits of Synchronization

同步工具只在应用中被所有的线程一致的使用的时候才有效。如果你为一个特定资源创建了一个 mutex，你所有的线程在尝试操作这个资源的时候必须获取这个 mutex。没能这么做的话 mutext 提供的保护不会起作用，并且这是一个变成错误。

Be Aware of Threats to Code Correctness

当使用锁或 memory barriers 时，你总是应该仔细考虑它们在你代码中的位置。尽管锁看起来被安置得很好，但是实际上可能只是给你了一种安全的假象。下面的一系列例子尝试通过指出开起来没什么问题的代码中的缺陷来说明这个问题。基本前提是你有一个包含可变对象的可变数组。假设你想要在数组中对象的一个方法，你也许会使用下面的代码：

```
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，包围数据的锁阻止了其他线程修改数组直到你获得指定的对象。因为你获取的对象本身是不可变的，因此一个围绕 `doSomething` 调用的锁是不需要的。

但上面的例子扔有一个问题。如果你释放了锁，另一个线程进入并删除了数组中所有的对象在调用 `doSomething` 之前，会发生什么了？在一个没有垃圾回收的应用中，你获取的对象可能被释放，使得 `anObject` 指向非法地址。要修复这个问题，你可以简单重新安排锁的位置，在调用 `doSomething` 之后释放锁。

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

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

通过将 `doSomething` 调用移到锁内，你的代码保证了这个对象在方法调用的过程中是有效的。但不幸的是，如果这儿方法运行的时间过程的话，你的锁会长时间被持有，可能导致一个性能瓶颈。

这段代码的问题不是 critical region 被定义得很差，而是真实的问题没有被理解。真实的问题是多线程的出现导致的内存管理的问题。因为它可能被其它线程释放，一个更好的方案是在锁内 retain 获取的对象。这种方案解决了对象被释放的问题，并且这样做不会导致性能问题。

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

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

尽管前面的代码本质很简单，但它们很好的说明了一个重要的点。当讨论到正确性的时候，你必须考虑的不仅仅是显而易见的问题。内存管理和你设计的其它部分都可能被多线程的出现而被影响，所以你必须提前考虑这些问题。另外，当考虑到安全你总是应该假设编译器做了最坏的事。这种清楚认知和警觉应该可以帮助你避免潜在的问题，并保证代码行为正确性。

Watch Out for Deadlocks and Livelocks

任何时刻一个线程同时试着去获取多于一个的锁时，就会有死锁的可能发生。一个死锁发生在两个不同的线程拥有一个彼此想要的锁，并且彼此都试着去获取彼此的锁。结果就是两个线程被永久的阻塞，因为它们谁也拿不到彼此的锁。

一个 livelock 跟死锁相似，发生在两个线程为相同的资源竞争。在一个 livelock 的情况下，一个线程放弃了第一个锁，然后试着去获取第二把锁。一旦它获得了第二把锁，它重新试着获取第一把锁。因为它总是在释放一把锁和尝试获取另一把锁而不是做任何有效的工作，所以它某种程度上也是被锁住了。

避免死锁和 livelock 情况的最好方式是，每次只获取一把锁。如果你必须获取多把锁的话，你应该确保其他线程没有尝试做类似的事。

Use Volatile Variables Correctly

如果你已经使用一个 mutex 来保护一段代码，不要自动假设你需要使用 `volatile` 关键字来保护代码段中的变量。一个 mutex 包含一个 memory barrier 来保证加载和存储操作的正确顺序。添加 `volatile` 关键字到 critical section 的变量会强制变量每次访问的时候都从内存加载。这两种同步机制的组合可能在特定的情况下是必要的，但是也导致了严重的性能损耗。如果 mutex 足够保护变量的话，就不需要使用 `volatile` 关键字。

同样不要尝试使用 `volatile` 变量来避免使用 mutex。一般，mutex 和其它同步机制是一个更好的方式来保护你的数据结构。 `volatile` 关键字只是保证变量总是从内存加载，而不是从寄存器中。它并不保证变量正确的被你的代码访问。

Using Atomic Operations

非阻塞机制是一种执行某些类型的操作并避免锁的昂贵消耗的方式。尽管锁是同步线程的有效方式，获取一个锁是一个相对昂贵的操作，即使在非竞争的条件下。对比之下，许多原子操作只需要琐碎的时间来完成，并可以像锁一样有效。

原子操作让你在 32 位或 64 未进行简单的数学和逻辑操作。这些操作依赖于特殊的硬件指令 (和一个可选的 memory barrier) 来保证给定的操作在原子操作完成之后被影响的内存才可被再次访问。在多线程的情况下，你总是应该使用包含一个 memory barrier 的原子操作来保证内存被线程正确的同步访问。

下表中列出了可用的原子数学和逻辑操作和相应的函数名。这些函数被申明在 `/usr/include/libkern/OSAtomic.h` 头文件中，你可以在这个头文件中找到完整的语法。64-bit 的版本只适合于 64 位的处理器。

Operation	Funcation name	Description
Add	OSAtomicAdd32 OSAtomicAdd32Barrier OSAtomicAdd64 OSAtomicAdd64Barrier	将两个整数相加，并保存结果到指定的变量
Increment	OSAtomicIncrement32 OSAtomicIncrement32Barrier OSAtomicIncrement64 OSAtomicIncrement64Barrier	将指定的整数加 1
Decrement	OSAtomicDecrement32 OSAtomicDecrement32Barrier OSAtomicDecrement64 OSAtomicDecrement64Barrier	将指定的整数减 1
Logical OR	OSAtomicOr32 OSAtomicOr32Barrier	将指定的 32 bit 值与一个 32 bit mask 做 OR 操作
Logical AND	OSAtomicAnd32 OSAtomicAnd32Barrier	将指定的 32 bit 值与一个 32 bit mask 做 AND 操作
Logical XOR	OSAtomicXor32 OSAtomicXor32Barrier	将指定的 32 bit 值与一个 32 bit mask 做 XOR 操作
Compare and swap	OSAtomicCompareAndSwap32 OSAtomicCompareAndSwap32Barrier OSAtomicCompareAndSwap64 OSAtomicCompareAndSwap64Barrier OSAtomicCompareAndSwapPtr OSAtomicCompareAndSwapPtrBarrier OSAtomicCompareAndSwapInt OSAtomicCompareAndSwapIntBarrier OSAtomicCompareAndSwapLong OSAtomicCompareAndSwapLongBarrier	将指定的值与一个 old value 作比较，如果相等，函数将 new value 赋给指定变量；不是的话，啥也不做。比较和赋值是作为一个原子操作完成的，返回返回一个 boolean 值表示 swap 是否发生
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.

原子操作的行为大部分都相对直接，并和你期望的相关。下例中，展示了原子操作 test-and-set 和 compare-and-swap 操作，这些相对发杂。最开始的三个 `OSAtomicTestAndSet` 函数调用证明了位操作方程，它们的结果跟你所期望的不大相同。最后的两个调用展示了 `OSAtomicCompareAndSwap32` 的行为。在所有的情况下，这些函数是在未竞争的条件下。

```
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.
```

Using Locks

锁是多线程编程的基本同步工具。锁使你很容易的保护一部分代码，以保证这部分代码行为的正确性。OS X 和 iOS 为所有的应用类型提供了基本 mutex locks，Foundation 框架定义了 mutex 一些额外变种来支持特殊情况。下面的部分将给你展示怎么使用这些类型的锁。

Using a POSIX Mutex Lock

POSIX mutex lock 对于任何应用都是很容易使用的。要创建一个 mutex lock，你声明并初始化一个 `pthread_mutex_t` 结构。要加锁和解锁这个 mutex lock，你使用 `pthread_mutex_lock` 和 `pthread_mutex_unlock` 函数。下例展示了初始化和使用一个 POSIX 线程 mutex lock 的必须基本代码。当你使用完锁后，调用 `pthread_mutex_destroy` 来释放锁结构。

```
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**: 上面的代码比较简单，主要是为了说明 POSIX thread mutex 函数的用法。你自己的代码应该检查返回的 error code，并相应的处理它们。

Using the NSLock Class

一个 `NSLock` 对象为 Cocoa 应用实现了一个基本的 mutex。对于所有锁 (包括 `NSLock`) 的接口实际上是在 `NSLocking` protocal 中定义的，定义了 `lock` 和 `unlock` 方法。你使用这些方法来获取和释放任何 mutex。

除了标准的锁定行为外， `NSLock` 类添加了 `tryLock` 和 `lockBeforeDate:` 等方法。 `tryLock` 方法尝试获取锁但不阻塞当前线程如果锁不可用的话；相反，这个函数简单的返回 `NO` . `lockBeforeDate:` 方法尝试获取锁，但是会唤起线程如果指定的锁在指定的时间点还没有获得的话。

下面的代码展示了怎么使用一个 `NSLock` 对象来协调更新视图，它的数据由多个线程来计算。如果线程不能立即获得锁，它只是简单的继续计算，直到它可以或得这个锁并更新显示。

```
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];
    }
}
```

Using the @synchronized Directive

`@synchronized` 关键字是在 Objective-C 代码中匆忙匆忙创建 mutex lock 的方便方式。 `@synchronized` 命令做任何其它 mutex lock 所做的——组织不同的线程同时获得同一个锁。然而在这种情况下，你不需要创建 mutex 或直接锁住对象。相反，你可以简单的使用任何 Objective-C 对象作为 lock token，正如下面的代码所示：

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

传给 `@synchronized` 命令的对象是一个 unique identifier，用来区别被保护的代码段。如果你在不同的线程执行上面的代码，并传递不同的对象给 `anObj` 参数，每个线程将会获取它自己的锁，并继续执行，而不会被阻塞。如果你在两个线程中传递相同的对象的话，其中一个线程会获先获得这个锁，另一个线程就会阻塞到第一个线程释放这个锁为止。

作为一个预防措施， `@synchronized` 代码段隐式的添加了一个 exception handler 来保护代码。这个 handler 会在发生异常事件的时候释放这个 mutex。这意味着为了使用 `@synchronized` 命令，你必须在你的代码中开启 Objective-C 的异常处理。如果你不想要隐式异常处理带来的额外负载的话，你应该考虑使用 lock 类。

关于 `@synchronized` 的更多信息，可以参见 *The Objective-C Programming Language*.

Using Other Cocoa Locks

下面的部分描述了使用其它不同类型 Cocoa lock 的方法。

Using an NSRecursiveLock Object

`NSRecursiveLock` 类定义了一个可被同一个线程多次获得的锁，同时不会引起这个线程死锁。一个 recursive lock 会记录它被多少次成功的获得。每次成功的获得这个锁必须与锁的 unlock 调用平衡。只有当这个锁的 lock 和 unlock 调用平衡的时候，这个锁才会真正被释放，这样其它的线程才可以获得它。

正如它的名字所说，这种类型的锁通常用在递归函数中来阻止递归阻塞了当前线程。你可以在非递归的情况下类似的使用它来调用那些语义上也需要它们也持有锁的函数。(就是函数调用其它的函数，不同的函数都需要自己去持有锁) 下面是一个简单的递归函数，递归的过程中需要获取锁。如果你对于以下代码不使用递归锁的话，函数在再次调用的时候会造成死锁。

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

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

MyRecursiveFunction(5);
```

> **注意**: 因为一个递归锁直到所有的 lock 调用被 unlock 调用平衡掉才会释放，你应该仔细权衡到底使用一个 performance lock 还是一个 potential performances implication. 持有任何锁超过一定时间会导致其它的线程阻塞到递归完成。如果你重构代码消除递归或消除使用锁的需要，你也许可以获

> 得更好的性能。

Using an NSConditionLock Object

一个 `NSConditionLock` 对象定义了一个可以被 lock 和 unlock 指定次数的 mutex lock。你不应该与一个 condition 混淆。虽说行为与 conditions 相似，但是实现完全不同。

通常，当线程需要按指定的顺序执行任务的时候，你使用一个 `NSConditionLock` 对象，当一个线程生产内容，其他线程消耗。当生产者执行的时候，消耗者使用程序中特定的 condition 来获取锁。(condition 本身只是一个你定义的整数值) 当生产者结束时，它解锁并设置 lock condition 到适合的整数值来唤醒消耗者线程，然后消耗者线程处理数据。

`NSConditionLock` 对象的 lock 和 unlock 方法可以以任意组合使用。例如，你可以将 `lock` 和 `unlockWithCondition:` 配对使用，或 `lockWhenCondition:` 与 `unlock` 配对使用。当然，但是后面的组合 unlock 了锁后可能并不释放任何等待指定 condition value 的线程。

下面的例子显示了一个 producer-consumer 问题怎么使用 condition locks 来解决。想象一个应用一个数据队列。一个 producer 线程添加数据到队列，consumer 线程从队列抽取数据。producer 不需要等待指定的 condition，但是它必须等待锁可用以便安全写入数据到队列。

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

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

因为代码中锁的初始 condition 是 `NO_DATA`，producer 线程初始时应该没有问题获得这个锁。它填充队列并设置 condition 为 `HAS_DATA`。在接下来的迭代中，producer 线程添加新数据，不管队列是否有数据或是空的。它阻塞的为一时候是当一个 consumer 线程正在从队列中取数据。

因为 consumer 线程有数据要处理，它使用指定的 condition 等待队列。当 producer 添加数据到队列，consumer 线程被唤醒并获得锁。然后它可以从队列获取数据，并更新队列状态。下面的例子展示了 consumer 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.
}
```

Using an NSDistributedLock Object

`NSDistributedLock` 类可以被多台主机上的多个应用使用来限制相同资源的访问，如一个文件。这个锁的本质是一个 mutex lock，它使用一个文件系统 item 实现，例如一个文件或目录。对于一个 `NSDistributedLock` 对象来说它要有用的话，lock 必须可悲使用它的多个应用可写。这通常意味着把它放在一个任何跑应用都可以访问的文件系统上。

跟其它类型的锁不像， `NSDistributedLock` 并没有实现 `NSLocking` protocol，因此没有 `lock` 方法。一个 `lock` 方法会阻塞线程的执行，需要系统以设定的频率来轮询这个锁。为了对你的代码造成性能损耗， `NSDistributedLock` 提供了 `tryLock` 方法，让你决定是否轮询。

因为它是使用文件系统实现的，一个 `NSDistributedLock` 对象除非在它的拥有者显式释放它不会释放。如果你拥有一个 distributed lock 的应用奔溃了，其它的用户就访问不了共享的资源了。在这种情况，你可以使用 `breakLock` 方法来打断现有的锁以便你可以获得锁。breaking lock 应该竭力避免，除非你很确定拥有者线程死了并且不能释放锁。

和其它类型的锁一样，当你结束使用一个 `NSDistributedLock` 对象时，你应该 unlock 它。

Using Conditions

Condition 是一个特殊的锁类型，你可以用来同步任务进行的顺序。它跟 mutex 有微妙的区别。一个等待 condition 的线程会保持阻塞，直到这个 condition 被另一个线程显式的发送信号。

由于操作系统实现的细微区别，condition lock 被允许返回虚假的成功，即使它们并未被你的代码真实的发送信号。为了避免这些虚假的信号引起的问题，你总是应该将一个 predicate 与你的 condition lock 一起使用。Predicate 是一个更具体的方式来决定你的线程进一步处理是否安全。Condition 只是保证你的线程保持唤醒，直到这个 predicate 可以被 signaling 线程设置。

Using the NSCondition Class

`NSCondition` 类提供了与 POSIX conditions 同样的语义，但同时将所有的锁和 condition 数据结构包装到一个对象。结果是你可以像 mutex 一样加锁，然后等待一个 condition。

下面的代码片段展示了等待一个 `NSCondition` 对象的事件顺序。`cocoaCondition` 变量包含一个 `NSCondition` 对象，`timeToDoWork` 变量是一个被另一个线程递增的整数，递增操作就发生 signaling condition 之前。

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

timeToDoWork--;

// Do real work here.

[cocoaCondition unlock];
```

下面的代码显式用来给 Cocoa condition 发送信号很递增 predicate 变量的代码。你总是应该在 condition 发送信号之前给它加锁。

```
[cocoaCondition lock];
timeToDoWork++;
[cocoaCondition signal];
[cocoaCondition unlock];
```

Using POSIX Conditions

POSIX 线程 condition 锁需要同时使用一个 condition 数据结构和一个 mutex。尽管两个锁结构是分开的，在运行时 mutex lock 是密切的与这个 condition 结构绑定的。等待一个信号的线程总是应该使用同样的 mutex lock 和 condition 数据结构。改变这对结构会导致错误。

下面的示例显示了一个 condition 和 predicate 的基本初始化和使用。在初始化好 condition 和 mutex lock。等待线程使用 `ready_to_go` 变量作为它的 predicate 进入一个 while 循环。只有当 predicate 被设置并且 condition 后序被发送信号，等待线程才会被唤醒并开始执行它的工作。

```
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);
}
```

signaling 线程负责设置 predicate 和发送信号给 condition lock。下例展示实现这种行为的代码。在这个例子中，condition 在加锁状态下被发送信号来阻止多个线程等待 condition 时 race condition 发生。

```
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**: 上面的示例只是简单说明 POSIX 线程 condition 函数的使用。你自己的代码中应该检查这些函数返回的 error code，并合理的处理它们。

Appendix A: Thread Safety Summary

这个附录描述 iOS 和 OS X 中一些关键框架的高级别线程安全。这个附录中的内容将来会改变。

Cocoa

从多个线程使用 Cocoa 的指导如下：

- Immutable 对象一般是线程安全的。一旦你创建它们，你可以安全的在多个线程间传递它们。另一方面可变对象通常不是线程安全的。要在多线程应用中使用 mutable 变量，应用应该合理的同步它们的访问。
- 许多被视为 `thread-unsafe` 的对象只是从多个线程中使用才不安全。许多这类对象可在任意线程使用只要在任意时刻没有多个线程同时使用它。对于只能在应用主线程中使用的对象不属于此类。
- 应用的主线程负责处理事件。尽管如果其他线程被涉入到 event path 是 Application Kit 会继续工作，但 operation 会不安顺序发生。
- 如果你想要使用一个线程绘制到一个视图，将所有的绘制代码包围在 `lockFocusIfCanDraw` 和 `unlockFocus` 方法间。
- 在 Cocoa 中使用 POSIX 线程，你必须先将 Cocoa 置为多线程代码。

Foundation Framework Thread Safety

有个误解说 Foundation 框架是线程安全的，而 Application Kit Framework 不是的。不幸运的是，这是一个过于粗略的概述，有一定程度的误导。每个框架都有线程安全的部分和线程不安全的部分。下面的部分描述了 Foundation 框架的线程安全。

Thread-Safe Classes and Functions

下面的雷和方法通常被认为是线程安全的。你可以在多个线程间使用同一个同一个对象而不需要首先获得一个锁。

```
NSArray
NSAssertionHandler
NSAttributedString
 NSCalendarDate
NSCharacterSet
NSConditionLock
NSConnection
NSData
NSDate
NSDecimal functions
NSDecimalNumber
NSDecimal
NumberHandler
NSDeserializer
NSDictionary
NSDistantObject
NSDistributedLock
NSDistributedNotificationCenter
NSException
NSFileManager (in 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
NSXMLParser
Object allocation and retain count functions
Zone and memory functions
```

Thread-Unsafe Classes

下面的函数和类通常被认为是线程不安全的。在大部分时候，你可以使用这些在任意线程使用这些类，只要在任一时刻只有一个线程使用就可以。

```
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` `NSEnumerator` 对象它们自身是线程安全的，它们被列在这里的原因是在它们被使用的过程中改变它们所包装的数据对象是不安全的。例如，一个 anchiver 的例子，改变正在被 archived 的 object graph 是不安全的。对于一个 enumerator 而言，任意线程修改这个被枚举的集合都是不安全的。

Main Thread Only Classes

下面的类只能在应用的主线程上使用

```
NSAppleScript
```

Mutable Versus Immutable

Immutable 对象通常是线程安全的；一旦你创建了它们，你可以安全在线程间传递这些对象。当然，使用 immutable 对象时，你仍然需要正确的使用 reference count。如果不合适的 release 一个不是你 retain 的对象的话，你可能稍后会引起异常。

Mutable 对象通常是线程不安全的。在多线程应用中使用它们，应用必须使用锁来同步它们的访问。通常，mutable 的容器类不是线程安全的。也就是说，如果一个或多个线程在改变同一个数组的话，会产生问题。你必须使用锁来包围读和写发生的地方来保证线程安全。

即使一个方法声明返回一个 immutable 对象，你绝不应该简单的假设返回的对象是 immutable 的。返回的对象可能是 immutable 或 mutable 的，这依赖于方法的实现。例如，一个方法返回 `NSString` 类型，但它可能返回 `NSMutableString` 。如果你想要保证你所持有的对象是 immutable 的，你应该 copy 出一份 immutable 的对象。

Reentrancy

Reentrancy is only possible where operations “call out” to other operations in the same object or on different objects. Retaining and releasing objects is one such “call out” that is sometimes overlooked.

下面的列表列出了 Foundation 框架部分直观可重入的类。其他的类可能或不能重入，或者将来被改得可以重入。一个完成的可重入性分析一直没有进行，所以这个列表可能不够充分：

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

Class Initialization

Objective-C 运行时系统会发送一个 `initialize` 消息给每个类对象，在这个类收到任何消息之前。这给了这个类一个在它被使用前设置运行环境的机会。在多线程的应用，运行时会保证只有一个线程——就是碰巧给给发送第一个消息的线程——执行这个 `initialize` 方法。如果第二个线程正在 `initialize` 方法执行的过程中尝试给这个类发送消息的话，这个线程会阻塞到 `initialize` 方法完成执行。同时，第一个线程可以继续调用这个类上的其他方法。这个 `initialize` 方法不应该依赖第二个线程调用这个类的方法，不然的话，两个线程就死锁了。

因为到 OS X v10.1.x 的版本，一个线程可以发送消息到一个类，在另一个线程执行完类的 `initialize` 方法。线程可以访问没有被完全初始化的的一些值，这样可能导致应用 crash。如果你遇到这个问题，你需要引入锁来阻止对这些值的访问，直到它们被完全初始化好之后或在应用变成多线程之前强制初始化它们。

Autorelease Pools

每个线程维护它自己的 `NSAutoreleasePool` 对象。Cocoa 期望总是有一个 autorelease pool 在当前线程的栈上。如果没有一个 pool 的话，对象不会被释放，然后你就泄露内存了。在基于 Application Kit 的应用的主线程上一个 `NSAutoreleasePool` 对象会被自动的创建，但是副线程必须创建它们自己的在使用 Cocoa 之前。如果你的线程是一个常驻内存的线程，可能产生大量 autoreleased 对象，你应该定时销毁和创建 autorelease pools。否则，autoreleased 对象会被累积，你应用的内存足迹会增长。如果 detached 线程没有使用 Cocoa，你不需要创建一个 autorelease pool。

Run Loops

每个线程有唯一一个 run loop。每个 run loop，即每个线程，有它自己的一系列 input mode 来决定 run loop 跑的时候监听哪些 input source。定义在一个 run loop 上的 input modes 不影响另一个 run loop 的 input modes，即使它们有相同的名字。

主线程的 run loop 会自动跑起来如果你的应用是基于 Application Kit，但副线程必须它们自己跑起它们的 run loop。如果一个 detached 线程没有进入 run loop，线程在 detached method 完成执行时推出。

不管它一些外部特征，`NSRunLoop` 类不是线程安全的。你应该只在拥有 run loop 的线程上的调用这个实例的方法。

Application Kit Framework Thread Safety

下面的部分将 Application Kit 框架的线程安全。

Thread-Unsafe Classes

下面的类和函数通常被认为是线程不安全的。大部分情况下，你可以从任意线程使用这些类，只要在任一时刻只有一个线程。查看 class 文档来获取额外的消息。

- `NSGraphicsContext` . 查看下文
- `NSImage` . 查看下文
- `NSResponder`
- `NSWindow` 和它的后裔。查看下文

Main Thread Only Classes

下面的类必须从应用的主线程上访问：

- `NSCell` 和它所有的子类
- `NSView` 和它所有的子类

Window Restrictions

你可以在一个父线程上创建一个 window。Application Kit 保证与 window 关联的数据结构在主线程上被释放以避免 race condition。在一个并发处理多个 window 的应用中，window 对象有可能会泄露。

你可以在一个副线程上创建一个 modal window。Application Kit 会阻塞调用的副线程直到主线程跑在 modal loop。

Event Handling Restrictions

应用的主线程负责处理事件。主线程被阻塞在 `NSApplication` 的 `run` 方法里面，通常调用应用的 `main` 方法。Application Kit 继续工作的时候，如果其它线程参与到 key events，接受到的 keys 可能是错序的。通过让主线程处理事件，你可以获得更一致的用户体验。一旦受到事件，事件可以被分发都副线程来进一步处理。

你可以调用 `NSApplication` 的 `postEvent:atStart:` 方法来发送一个事件到主线程的事件队列上。相对于用户输入事件顺序不是保证的。应用的主线程仍然负责处理事件队列中的事件。

Drawing Restrictions

当使用 Application Kit 的图形函数和类时，Application Kit 通常是线程安全的，包括 `NSBezierPath` 和 `NSString` 类。使用具体类的详情将在下面描述。关于绘制和线程的更多信息可以参见 *Cocoa Drawing Guide*.

NSView Restrictions

`NSView` 类通常不是线程安全的。你应该只在应用的主线程上创建，销毁，resize，move，和在 `NSView` 上进行其它操作。只要你用 `lockFocusIfCanDraw` 和 `unlockFocus` 围绕你的绘制调用，你在副线程上绘制就是安全的。

如果应用的一个副线程想要使得视图的一部分在主线程上被重绘，它必须不要使用 `display`， `setNeedsDisplay:`， `setNeedsDisplayInRect:` 或 `setViewsNeedDisplay:` 方法，相反，它应该发送一个消息到主线程，或使用 `performSelectorOnMainThread:withObject:waitUntilDone:` 来调用这些方法。

视图系统的 graphics states 是 per-thread。使用 graphics states 以前是在单线程应用中获得更好绘制性能的一种方式，但现在不是了。不正确的使用 graphics states 实际上可能导致绘制代码比在主线程上绘制更低效。

NSGraphicsContext Restrictions

`NSGraphicsContext` 类代表了底层图形系统提供的绘制上下文。每个 `NSGraphicsContext` 实例持有它自己独立的 graphics state: coordinate system，clipping，current font 等等。在主线程上这个类的实例自动的为每个 `NSWindow` 实例创建。如果你从副线程做任何绘制操作，一个新的 `NSGraphicsContext` 实例会专门为这个线程创建。

如果你从副线程做任何绘制，你必须手动 flush 你的绘制调用。 Cocoa 并不自动使用副线程绘制的内容更新视图，所以当你结束绘制的时候，你需要手动调用 `flushGraphics` 方法。如果你的应用只从主线程绘制内容，你不需要 flush 你的绘制。

NSImage Restrictions

一个线程创建一个 `NSImage` 对象，绘制到 image buffer 中，传递给主线程绘制。潜在的 image cache 所有线程共享。要了解关于 images 和 cache 的工作原理，可以看看 *Cocoa Drawing Guide*.

Core Data Framework

Core Data 框架通常支持线程，尽管有些使用注意事项。更多关于这些注意事项的信息，可以参见 *Core Data Programming Guide*

Core Foundation

如果你认真的编码，Core Foundation 是足够线程安全的，你应该不会遇到任何与线程竞争相关的问题。通常情况下它是线程安全的，例如当你查询，retain，release，传递 immutable objects。即使是那些共享的中心对象被多个线程查询，它们也是线程安全的

像 Cocoa，Core Foundation 在 mutable 的对象或内容时不是线程安全的。例如，修改一个 mutable 数据或 mutable array 不是线程安全的，跟你想的一样，修改一个 immutable 中的一个对象也不是。这种情况的一种原因是性能，而性能又是这些情况时很重要的。更何况，在这个层面很难保证绝对的线程安全。你不能完全控制，例如，retaining 一个从容器获得的对象的中间行为。容器自身可能在调用 retain 之前已经释放，导致对象被释放。

在那些 Core Foundation 对象被从多个线程访问并且是 mutated。你的代码应该通过在访问点使用锁来保护并发访问。例如，枚举迭代访问一个 Core Foundation 数组应该使用合适的锁调用来包围枚举 block，来阻止其他人改变这个数组。