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。

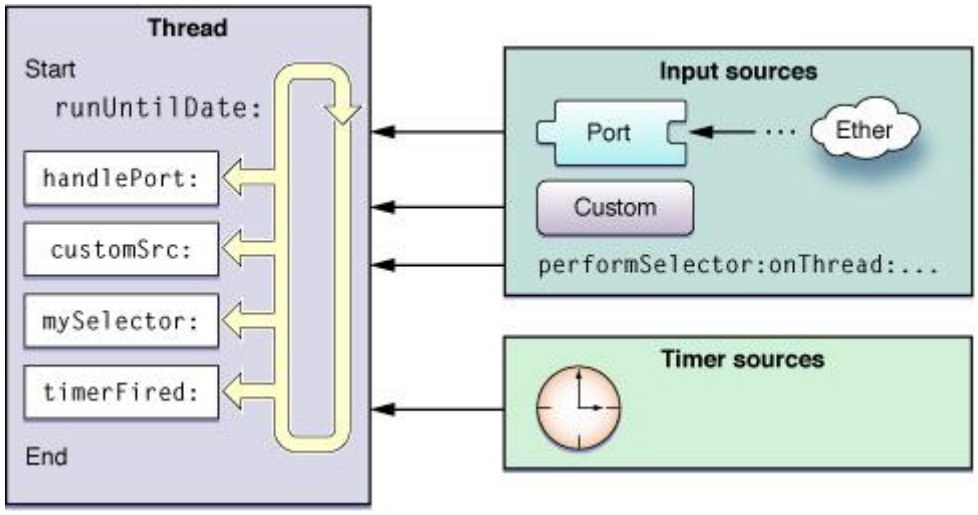
3.1 Run Loop 剖析

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

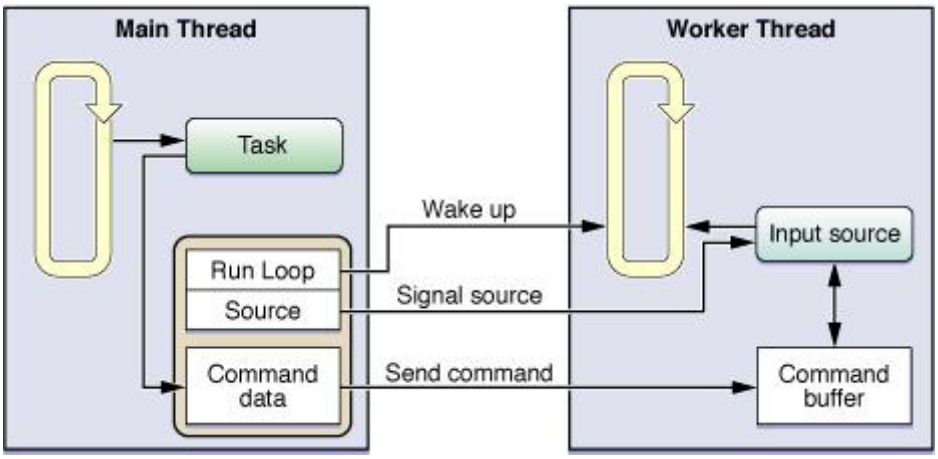
创建自定义的输入源包括定义以下内容：

- 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。