注：*Run Loops 链接*

CJL

20200904

一、介绍

运行循环是与线程相关联的基本基础结构的一部分。运行循环是事件处理循环，可用于安排工作并协调收到的事件的接收。运行循环的目的是在有工作要做时让线程忙，而在没有工作时让线程进入睡眠状态。

运行循环管理不是完全自动的。您仍然必须设计线程的代码以在适当的时候启动运行循环并响应传入的事件。 Cocoa和Core Foundation均提供运行循环对象，以帮助您配置和管理线程的运行循环。您的应用程序不需要显式创建这些对象。每个线程（包括应用程序的主线程）都有一个关联的运行循环对象。但是，只有辅助线程需要显式地运行其运行循环。在应用程序启动过程中，应用程序框架会自动在主线程上设置并运行运行循环。

以下各节提供有关运行循环以及如何为应用程序配置循环的更多信息。有关运行循环对象的其他信息，请参见 *NSRunLoop Class Reference*和 *CFRunLoop Reference*。

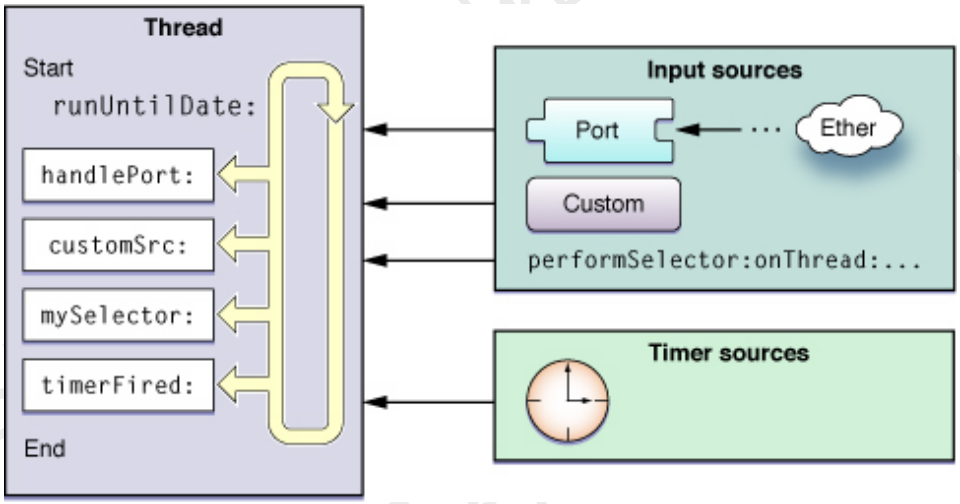
二、剖析运行循环

运行循环如其名一样。它是线程进入并用于运行事件处理程序以响应传入事件的循环。您的代码提供了用于实现运行循环实际循环部分的控制语句，换句话说，您的代码提供了驱动运行循环的while或for循环。在循环中，您可以使用运行循环对象来“运行”事件处理代码，以接收事件并调用已安装的处理程序。

运行循环从两种不同类型的源接收事件。输入源传递异步事件，通常是来自另一个线程或其他应用程序的消息。计时器源传递同步事件，这些事件在计划的时间或重复的间隔发生。两种类型的源都使用特定于应用程序的处理程序例程来处理事件到达时的事件。

图3-1显示了运行循环和各种源的概念结构。输入源将异步事件传递给相应的处理程序，并导致**runUntilDate:**方法（在线程的关联**NSRunLoop**对象上调用）退出。计时器源向其处理程序例程传递事件，但不会导致运行循环退出。

Figure 3-1 运行循环的结构及其源



除了处理输入源之外，运行循环还生成有关运行循环行为的通知。注册的runloop观察者可以接收这些通知，并使用它们在线程上进行其他处理。您可以使用Core Foundation在线程上安装runloop观察器。

以下章节提供了有关运行循环组件及其运行模式的更多信息。它们还描述了在处理事件期间的不同时间生成的通知。

2-1、运行循环模式

runloop模式是要监视的输入源和计时器的集合，以及要通知的运行循环观察者的集合。每次运行runloop时，都可以（显式或隐式）指定运行的特定“模式”。在运行循环的整个过程中，仅监视与该模式关联的源，并允许其传递事件。（类似地，只有与该模式相关联的观察者才被告知运行循环的进度。）与其他模式相关联的源将保留任何新事件，直到随后以适当的模式通过该循环。

在代码中，您可以通过名称标识模式。 Cocoa和Core Foundation都定义了默认模式和几种常用模式，以及用于在代码中指定这些模式的字符串。您可以通过简单地为模式名称指定自定义字符串来定义自定义模式。尽管您分配给自定义模式的名称是任意的，但这些模式的内容却不是。您必须确保将一个或多个输入源，计时器或运行循环观察器添加到您创建的任何模式中，以使其有用。

您可以使用模式从运行循环的特定遍历中过滤掉有害来源的事件。大多数时候，您将需要在系统定义的“默认”模式下运行runloop。但是，模式面板可以在“模式”模式下运行。在这种模式下，只有与模式面板相关的源才会将事件传递给线程。对于辅助线程，您可以使用自定义模式来防止低优先级源在时间紧迫的操作期间传递事件。

注：模式的区别取决于事件的来源，而不是事件的类型。例如，您不会使用模式来只匹配鼠标按下事件或键盘事件。您可以使用模式来监听不同的端口集，暂时挂起计时器，或者更改当前正在监视的源和运行循环观察器。

表3-1列出了Cocoa和Core Foundation定义的标准模式，以及何时使用该模式的说明。name列列出了用于在代码中指定模式的实际常量。

Table 3-1 预定义的运行循环模式

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. 默认模式是用于大多数操作的模式。大多数时候，您应该使用此模式来启动运行循环并配置输入源。
Connection	`NSConnectionReplyMode` (Cocoa)	Cocoa uses this mode in conjunction with `NSConnection` objects to monitor replies. You should rarely need to use this mode yourself. Cocoa将此模式与NSConnection对象结合使用来监视响应。你应该很少需要自己使用这种模式。
Modal	`NSModalPanelRunLoopMode` (Cocoa)	Cocoa uses this mode to identify events intended for modal panels. Cocoa使用此模式来识别用于模式面板的事件。
Event tracking	`NSEventTrackingRunLoopMode` (Cocoa)	Cocoa uses this mode to restrict incoming events during mouse-dragging loops and other sorts of user interface tracking loops. Cocoa使用此模式来限制鼠标拖动循环和其他类型的用户界面跟踪循环期间的传入事件。

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. 这是一组常用模式的可配置组。将输入源与此模式相关联还将其与组中的每个模式相关联。对于Cocoa应用程序，此集合默认包括默认，模式和事件跟踪模式。最初，Core Foundation仅包括默认模式。您可以使用CFRunLoopAddCommonMode函数将自定义模式添加到集合中

2-2、输入源

输入源以异步方式向线程传递事件。事件的源取决于输入源的类型，通常是两种类型之一。基于端口的输入源监视应用程序的Mach端口。自定义输入源监视事件的自定义源。就运行循环而言，输入源是基于端口还是自定义的并不重要。系统通常实现两种类型的输入源，您可以按原样使用。这两个信号源之间的唯一区别是它们是如何发出信号的。基于端口的源由内核自动发出信号，自定义源必须从另一个线程手动发出信号。

创建输入源时，可以将其分配给运行循环的一种或多种模式。模式影响在任何给定时刻监控的输入源。大多数情况下，您可以在默认模式下运行runloop，但也可以指定自定义模式。如果输入源不在当前监视的模式下，它生成的任何事件都将被保留，直到运行循环以正确的模式运行。

以下各节描述了一些输入源。

2-3、基于端口的源

Cocoa和corefoundation为使用端口相关的对象和函数创建基于端口的输入源提供了内置支持。例如，在Cocoa中，根本不必直接创建输入源。您只需创建一个port对象并使用 `NSPort` 方法将该端口添加到运行循环中。port对象为您处理所需输入源的创建和配置。

在CoreFoundation中，必须手动创建端口及其运行循环源。在这两种情况下，都可以使用与端口不透明类型（(`CFMachPortRef`, `CFMessagePortRef`, 或 `CFSocketRef`)）关联的函数来创建适当的对象。

有关如何设置和配置自定义基于端口的源的示例，请参阅配置基于端口的输入源.。

2-4、自定义输入源

要创建自定义输入源，必须使用与Core Foundation中的 `CFRunLoopSourceRef`不透明类型关联的函数。您可以使用几个回调函数配置自定义输入源。CoreFoundation在不同的点调用这些函数来配置源代码，处理任何传入事件，并在源代码从运行循环中删除时将其销毁。

除了定义事件到达时自定义源的行为外，还必须定义事件传递机制。源代码的这一部分在单独的线程上运行，负责为输入源提供其数据，并在准备好处理数据时向其发出信号。事件传递机制由您决定，但不必过于复杂。

有关如何创建自定义输入源的示例，请参见 定义自定义输入源。有关自定义输入源的参考信息，请参见 *CFRunLoopSource Reference*

2-5、Cocoa执行选择器源

除了基于端口的源代码外，Cocoa还定义了一个自定义输入源，允许您在任何线程上执行选择器。与基于端口的源一样，perform selector请求在目标线程上序列化，从而减轻了在一个线程上运行多个方法时可能出现的许多同步问题。与基于端口的源不同，执行选择器源在执行其选择器后将自身从运行循环中移除。

注意：在OS X v10.5之前，执行选择器源主要用于向主线程发送消息，但在OS X v10.5及更高版本以及iOS中，您可以使用它们向任何线程发送消息。

在另一个线程上执行选择器时，目标线程必须具有活动的运行循环。对于您创建的线程，这意味着等待直到您的代码显式启动运行循环。但是，由于主线程启动其自己的运行循环，因此只要应用程序调用应用程序委托的**applicationdiffinishlaunching:**方法，就可以开始对该线程发出调用。运行循环每次通过循环处理所有排队的执行选择器调用，而不是在每次循环迭代期间处理一个

表3-2列出了NSObject上定义的方法，这些方法可用于在其他线程上执行选择器。由于这些方法是在**NSObject**上声明的，因此可以在任何有权访问Objective-C对象的线程中使用它们，包括POSIX线程。这些方法实际上并不创建新的线程来执行选择器。

Table 3-2 在其他线程上执行选择器

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.
	在应用程序的主线程的下一个运行循环周期中对该线程执行指定的选择器。这些方法提供了在执行选择器之前阻塞当前线程的选项。

`performSelector:onThread:withObject:waitUntilDone:`
 	
`performSelector:onThread:withObject:waitUntilDone:modes:`
	Performs the specified selector on any thread for which you have an `NSThread` object. These methods give you the option of blocking the current thread until the selector is performed.
	对具有 `NSThread` 对象的任何线程执行指定的选择器。这些方法提供了在执行选择器之前阻塞当前线程的选项。

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. 在下一个运行循环周期中以及可选的延迟时间之后，在当前线程上执行指定的选择器。因为它一直等到下一个运行循环周期执行选择器，所以这些方法提供了当前执行代码的最小自动延迟。多个排队的选择器按照排队的顺序依次执行。
cancelPreviousPerformRequestsWithTarget: cancelPreviousPerformRequestsWithTarget:selector:object:	Lets you cancel a message sent to the current thread using the performSelector:withObject:afterDelay: or performSelector:withObject:afterDelay:inModes: method. 使您可以使用performSelector:withObject:afterDelay:或performSelector:withObject:afterDelay:inModes:方法取消发送到当前线程的消息。

有关这些方法的详细信息，请参见*NSObject Class Reference*.

2-6、定时器源

计时器源在将来的预设时间将事件同步传递到线程。计时器是线程通知自身执行某项操作的一种方式。例如，搜索字段可以使用计时器在用户连续按键之间经过一定时间后启动自动搜索。使用这个延迟时间，用户就有机会在开始搜索之前键入尽可能多的所需搜索字符串。

尽管计时器生成基于时间的通知，但它不是一种实时机制。与输入源一样，计时器与运行循环的特定模式相关联。如果计时器未处于运行循环当前监视的模式，则在以计时器支持的模式之一运行运行循环之前，它不会触发。类似地，如果在运行循环正在执行处理程序例程的过程中触发计时器，则计时器将等到下一次通过运行循环调用其处理程序例程。如果运行循环根本没有运行，计时器就不会触发。

您可以将计时器配置为仅生成一次事件或重复生成事件。重复计时器根据预定的触发时间而不是实际触发时间自动重新调度自己。例如，如果一个计时器被安排在某个特定时间触发，并且此后每隔5秒触发一次，那么即使实际触发时间被延迟，计划的触发时间也将始终落在原来的5秒时间间隔上。如果触发时间延迟太久，以致错过了一个或多个预定的触发时间，则对于错过的时间段，计时器只触发一次。在为错过的时间段触发后，计时器将重新安排为下一个预定的触发时间

有关配置计时器源的更多信息，请参见配置计时器源。有关参考信息，请参见*NSTimer Class Reference* 或 *CFRunLoopTimer Reference*。

2-7、运行循环观察者

与在发生适当的异步或同步事件时触发的源不同。运行循环观察器在运行循环本身执行期间的特殊位置触发。您可以使用运行循环观察器来准备线程以处理给定事件，或者在线程进入睡眠状态之前准备线程。您可以将运行循环观察者与运行循环中的以下事件相关联：

- 运行循环的入口。
- 当运行循环将要处理计时器时。
- 当运行循环将要处理输入源时。
- 当运行循环即将进入睡眠状态时。
- 当运行循环已唤醒，但需要在处理该事件之前将其唤醒。
- 从运行循环中退出。

您可以使用Core Foundation将运行循环观察器添加到应用程序。要创建运行循环观察器，请创建 `CFRunLoopObserverRef`透明类型的新实例。此类型跟踪自定义回调函数及其感兴趣的活动。

与计时器类似，运行循环观察器可以使用一次或重复使用。一次触发的观察者在触发后将自己从运行循环中删除，而重复的观察者仍保持连接。您可以指定观察者在创建时是运行一次还是重复运行。

有关如何创建运行循环观察器的示例，请参阅 配置运行循环。有关参考信息，请参阅 *CFRunLoopObserver Reference*。

2-8、事件的运行循环序列

每次运行它时，线程的运行循环都会处理未决事件，并为所有附加的观察者生成通知。它的执行顺序非常具体，如下所示：

1. 通知观察者已进入运行循环。
2. 通知观察者准备就绪的计时器即将触发。.
3. 通知观察者任何不基于端口的输入源都将被触发。
4. 触发所有准备触发的非基于端口的输入源。
5. 如果基于端口的输入源已准备好并等待启动，请立即处理事件。转到步骤9。
6. 通知观察者线程即将休眠。.
7. 使线程进入睡眠状态，直到发生以下事件之一：
    - 基于端口的输入源的事件到达.
    - 计时器触发.

- 运行循环设置的超时值已过期.
- 运行循环被显式唤醒

8. 通知观察者线程刚刚醒来。

9. 处理挂起的事件。.
   - 如果触发了用户定义的计时器，请处理计时器事件并重新启动循环。转到步骤2。.
   - 如果触发了输入源，则传递事件
   - 如果运行循环已显式唤醒但尚未超时，请重新启动循环。转到步骤2。.

10. 通知观察者运行循环已退出。.

由于计时器和输入源的观察者通知是在这些事件实际发生之前传递的，因此通知时间和实际事件时间之间可能会有差距。如果这些事件之间的时间很关键，则可以使用sleep和asleep-from-sleep通知来帮助您关联实际事件之间的时间。

因为计时器和其他周期性事件是在运行runloop时传递的，因此规避该循环会中断这些事件的传递。这种行为的典型示例是，每当您通过输入循环并重复从应用程序请求事件来实现鼠标跟踪例程时，都会发生这种行为。因为您的代码直接捕获事件，而不是让应用程序正常分配事件，所以在您的滑鼠追踪程式退出并将控制权传回应用程式之前，活动计时器将无法启动。

可以使用运行循环对象显式地唤醒运行循环。其他事件也可能导致运行循环被唤醒。例如，添加另一个非基于端口的输入源会唤醒运行循环，以便可以立即处理该输入源，而不是等待其他事件发生。

三、 什么时候使用运行循环?

唯一需要显式运行runloop的时间是为应用程序创建辅助线程时。应用程序主线程的运行循环是基础架构的重要组成部分。因此，应用程序框架提供运行主应用程序循环的代码，并自动启动该循环。iOS中 `UIApplication`（或osx中的 `NSApplication`）的run方法作为正常启动序列的一部分启动应用程序的主循环。如果您使用Xcode模板项目来创建应用程序，则永远不必显式调用这些例程。

对于辅助线程，您需要确定是否需要运行循环，如果需要，请自行配置并启动它。您不需要在所有情况下都启动线程的运行循环。例如，如果使用线程执行一些长时间运行的预定任务，则可以避免启动运行循环。运行循环用于需要与线程进行更多交互的情况。例如，如果您打算执行以下任一操作，则需要启动运行循环：

- 使用端口或自定义输入源与其他线程通信.
- 在线程上使用定时器。.
- 在Cocoa应用程序中使用任何**performSelector…**方法.
- 保持线程执行周期性任务.

如果确实选择使用运行循环，则配置和设置非常简单。与所有线程编程一样，您应该有一个计划，在适当的情况下退出辅助线程。最好通过让线程退出干净地结束线程，而不是强制终止线程。使用runloop对象中介绍了有关如何配置和退出运行循环的信息。

四、使用运行循环对象

运行循环对象提供主接口，用于向运行循环添加输入源、计时器和运行循环观察器，然后运行它。每个线程都有一个与之关联的运行循环对象。在Cocoa中，此对象是 `NSRunLoop`类的实例。在低级应用程序中，它是指向 `CFRunLoopRef`不透明类型的指针

4-1、获取运行循环对象

要获取当前线程的运行循环，请使用以下方法之一：

- 在Cocoa应用程序中，使用 NSRunLoop的 currentRunLoop类方法检索**NSRunLoop**对象。.

- 使用 CFRunLoopGetCurrent函数.

尽管它们不是免费的桥接类型，但是您可以在需要时从**NSRunLoop**对象获取**CFRunLoopRef**不透明类型。 **NSRunLoop**类定义了一个 getCFRunLoop 方法，该方法返回可以传递给Core Foundation例程的**CFRunLoopRef**类型。由于两个对象都引用同一个运行循环，因此您可以根据需要混合对**NSRunLoop**对象和**CFRunLoopRef**不透明类型的调用。

4-2、配置运行循环

在辅助线程上运行runloop之前，必须向其添加至少一个输入源或计时器。如果运行循环没有任何要监视的源，则当您尝试运行它时，它将立即退出。有关如何将源添加到运行循环的示例，请参见 配置运行循环源。

除了安装源代码之外，还可以安装运行循环观察器，并使用它们来检测运行循环的不同执行阶段。要安装运行循环观察器，需要创建一个 CFRunLoopObserverRef不透明类型，并使用 CFRunLoopAddObserver函数将其添加到运行循环中。运行循环观察者必须使用Core Foundation创建，即使对于Cocoa应用程序也是如此。

清单3-1显示了一个线程的主例程，该线程将一个运行循环观察器附加到它的运行循环上。该示例的目的是向您展示如何创建运行循环观察器，因此代码只需设置一个运行循环观察器来监视所有运行循环活动。基本处理程序例程（未显示）只是在处理计时器请求时记录运行循环活动。

Listing 3-1 创建运行循环观察者

```
- (void)threadMain
{
    // The application uses garbage collection, so no autorelease pool is needed.
      //应用程序使用垃圾回收，因此不需要自动释放池
    NSRunLoop* myRunLoop = [NSRunLoop currentRunLoop];

    // Create a run loop observer and attach it to the run loop.
      //创建一个运行循环观察器并将其附加到运行循环
    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.创建并安排计时器。
    [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.运行运行循环10次，让计时器
启动。
        [myRunLoop runUntilDate:[NSDate dateWithTimeIntervalSinceNow:1]];
        loopCount--;
    }
    while (loopCount);
}
```

在为长生存期线程配置运行循环时，最好至少添加一个输入源来接收消息。虽然您可以在只附加计时器的情况下进入运行循环，但一旦计时器触发，它通常会失效，这将导致运行循环退出。附加一个重复计时器可以使运行循环保持较长时间运行，但是会涉及定期触发计时器以唤醒线程，这实际上是轮询的另一种形式。相比之下，输入源等待事件发生，让线程一直处于休眠状态。

4-3、启动运行循环

只有应用程序中的辅助线程才需要启动运行循环。运行循环必须至少有一个要监视的输入源或计时器。如果未连接，则运行循环将立即退出。

有几种启动运行循环的方法，包括以下几种：

- 无条件
- 设定时间限制
- 在特定模式下

无条件地进入运行循环是最简单的选择，但也是最不可取的。无条件地运行运行循环会将线程放入一个永久循环中，这使您几乎无法控制运行循环本身。您可以添加和删除输入源和计时器，但停止运行循环的唯一方法是终止它。也无法在自定义模式下运行runloop。

与其无条件地运行runloop，不如使用超时值运行runloop。当您使用超时值时，运行循环将运行直到事件到达或指定的时间到期为止。如果事件到达，则将该事件分派到处理程序进行处理，然后退出运行循环。然后，您的代码可以重新启动运行循环以处理下一个事件。如果分配的时间到期了，您可以简单地重新启动运行循环或使用该时间进行任何必要的内务处理。

除了超时值之外，您还可以使用特定模式运行runloop。模式和超时值不是互斥的，并且在启动运行循环时都可以使用。模式限制了将事件传递到运行循环的源的类型， Run Loop Modes中对此进行了详细描述。

清单3-2显示了一个线程主入口例程的框架版本。本质上，您将输入源和计时器添加到运行循环中，然后反复调用其中一个例程来启动运行循环。每次运行循环例程返回时，您都要检查是否出现了可能需要退出线程的条件。该示例使用Core Foundation运行循环例程，以便可以检查返回结果并确定为什么退出运行循环。如果您使用的是Cocoa，并且不需要检查返回值，则也可以使用NSRunLoop类的方法以类似的方式运行运行循环。 （有关调用NSRunLoop类的方法的运行循环的示例，请参见清单3-14。）

Listing 3-2 运行runloop

```
- (void)skeletonThreadMain
{
    // Set up an autorelease pool here if not using garbage collection.
      // 如果不使用垃圾回收，请在此处设置自动释放池。
    BOOL done = NO;

    // Add your sources or timers to the run loop and do any other setup.
      // 将源或计时器添加到运行循环中，并执行任何其他设置。

    do
    {
        // Start the run loop but return after each source is handled.
        // 启动运行循环，但在处理每个源之后返回。
        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.
            //如果源显式停止了运行循环，或者如果没有源或计时器，则继续并退出。
        if ((result == kCFRunLoopRunStopped) || (result == kCFRunLoopRunFinished))
            done = YES;

        // Check for any other exit conditions here and set the
        // done variable as needed.
            //检查此处是否有其他退出条件，并根据需要设置done变量。
    }
    while (!done);

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

可以递归地运行runloop。换句话说，可以调用 CFRunLoopRun, CFRunLoopRunInMode,或任何**NSRunLoop**方法，以便从输入源或计时器的处理程序例程中启动运行循环。这样做时，您可以使用任何要运行嵌套运行循环的模式，包括外部运行循环使用的模式

4-4、退出运行循环

在处理事件之前，有两种方法可以使运行循环退出：

- 配置运行循环以使用超时值运行.
- 告诉runloop停止.

如果可以管理的话，使用超时值当然是首选。指定一个超时值可以让运行循环在退出之前完成其所有的正常处理，包括向运行循环观察者发送通知。

使用 CFRunLoopStop函数显式停止运行循环会产生类似于超时的结果。运行循环将发出所有剩余的运行循环通知，然后退出。区别在于您可以在无条件启动的运行循环中使用此技术

尽管删除运行循环的输入源和计时器也可能导致运行循环退出，但这不是停止运行循环的可靠方法。一些系统例程将输入源添加到运行循环中以处理所需的事件。因为您的代码可能不知道这些输入源，所以它将无法删除它们，这将阻止运行循环退出

4-5、线程安全和运行循环对象

线程安全性取决于您使用哪个API来操纵运行循环。 Core Foundation中的函数通常是线程安全的，可以从任何线程中调用。但是，如果执行的操作更改了运行循环的配置，那么只要可能，最好从拥有运行循环的线程进行更改

Cocoa NSRunLoop类本质上不像其Core Foundation对应类那样具有线程安全性。如果使用NSRunLoop类来修改运行循环，则应仅从拥有该运行循环的同一线程进行修改。将输入源或计时器添加到属于不同线程的运行循环中可能会导致代码崩溃或行为异常。

五、配置运行循环源

下面的部分展示了如何在Cocoa和corefoundation中设置不同类型的输入源的示例

5-1、定义自定义输入源

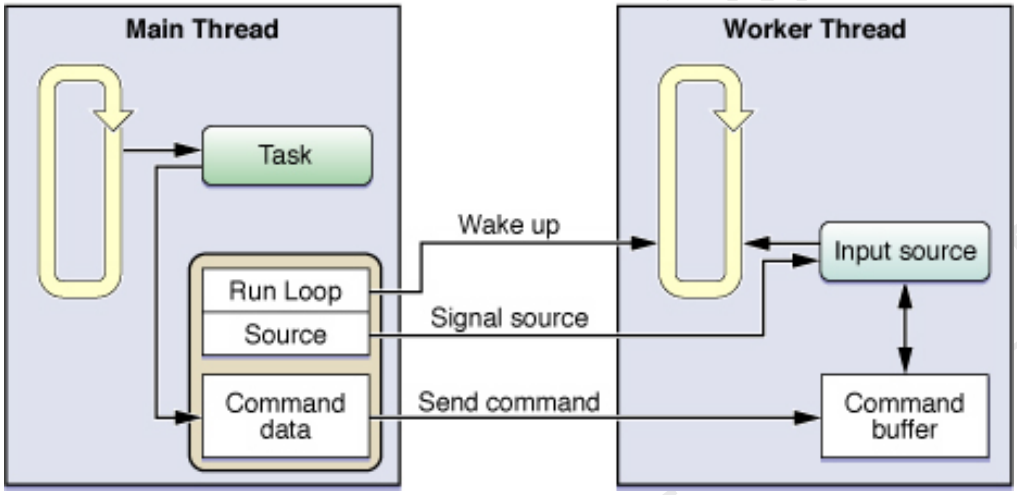
创建自定义输入源需要定义以下内容：

- 希望输入源处理的信息.
- 调度程序，让感兴趣的客户知道如何联系您的输入源.
- 处理程序，用于执行任何客户端发送的请求。.
- 取消例程，用于使您的输入源无效.

因为您创建了一个自定义输入源来处理自定义信息，所以实际配置的设计是灵活的。调度程序、处理程序和取消例程是自定义输入源几乎总是需要的关键例程。然而，其余的输入源行为大多发生在这些处理程序例程之外。例如，由您定义将数据传递到输入源以及将输入源的存在与其他线程通信的机制。

图3-2显示了自定义输入源的示例配置。在此示例中，应用程序的主线程维护对输入源，该输入源的自定义命令缓冲区以及安装该输入源的运行循环的引用。当主线程有一个要移交给工作线程的任务时，它将一个命令连同工作线程启动该任务所需的所有信息一起发布到命令缓冲区。（由于主线程和工作线程的输入源都可以访问命令缓冲区，所以该访问必须同步。）一旦发布命令，主线程将向输入源发出信号并唤醒工作线程的运行循环。收到唤醒命令后，运行循环将调用输入源的处理程序，该处理程序将处理在命令缓冲区中找到的命令。

Figure 3-2 操作自定义输入源



以下部分将解释上图中自定义输入源的实现，并显示需要实现的关键代码。

定义输入源

定义自定义输入源需要使用Core Foundation例程来配置您的运行循环源并将其附加到运行循环。尽管基本处理程序是基于C的函数，但这并不妨碍您编写这些函数的包装程序并使用Objective-C或C ++来实现代码主体。

图 Figure 3-2 中引入的输入源使用一个Objective-C对象来管理命令缓冲区并与运行循环协调。清单3-3显示了此对象的定义。**RunLoopSource**对象管理命令缓冲区，并使用该缓冲区从其他线程接收消息。此清单还显示了**RunLoopContext**对象的定义，该对象实际上只是一个容器对象，用于传递**RunLoopSource**对象和对应用程序主线程的运行循环引用

Listing 3-3 自定义输入对象定义

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

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

// Handler method 处理程序方法
- (void)sourceFired;

// Client interface for registering commands to process
//客户端接口，用于注册要处理的命令
- (void)addCommand:(NSInteger)command withData:(id)data;
- (void)fireAllCommandsOnRunLoop:(CFRunLoopRef)runloop;

@end

// These are the CFRunLoopSourceRef callback functions.
//这些是CFRunLoopSourceRef回调函数。
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.
//RunLoopContext是在注册输入源时使用的容器对象。
@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代码管理输入源的自定义数据，但是将输入源附加到运行循环需要基于C的回调函数。当您将运行循环源实际附加到运行循环时，将调用这些函数中的第一个，如清单3-4所示。因为此输入源只有一个客户端（主线程），所以它使用调度程序函数发送一条消息，向该线程上的应用程序委托注册自己。当委托想要与输入源进行通信时，它将使用**RunLoopContext**对象中的信息进行通信。

Listing 3-4 调度运行循环源

```
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];
}
```

最重要的回调例程之一是在输入源被信号通知时用于处理自定义数据的例程。清单3-5显示了与**RunLoopSource**对象关联的perform回调例程。此函数只是将完成工作的请求转发到**sourceFired**方法，该方法然后处理命令缓冲区中存在的所有命令。

Listing 3-5 在输入源中执行工作

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

如果使用 `CFRunLoopSourceInvalidate`函数从运行循环中删除输入源，系统将调用输入源的取消例程。您可以使用此例程通知客户端您的输入源不再有效，并且应该删除对它的任何引用。清单3-6显示了注册到RunLoopSource对象的取消回调例程。此函数将另一个RunLoopContext对象发送给应用程序委托，但这次请求委托删除对运行循环源的引用。

Listing 3-6 使输入源无效

```
//从runloop中删除输入源
void RunLoopSourceCancelRoutine (void *info, CFRunLoopRef rl,
CFStringRef mode)
{
      //获取输入源
    RunLoopSource* obj = (RunLoopSource*)info;
    AppDelegate* del = [AppDelegate sharedAppDelegate];
    RunLoopContext* theContext = [[RunLoopContext alloc]
initWithSource:obj andLoop:rl];

      //删除输入源
    [del performSelectorOnMainThread:@selector(removeSource:)
                                withObject:theContext
waitUntilDone:YES];
}
```

注意：应用程序委托的registerSource:和removeSource:方法的代码显示在 Coordinating with Clients of the Input Source 中

在运行循环上安装输入源

清单3-7显示了**RunLoopSource**类的**init**和**addToCurrentRunLoop**方法。 init方法创建 `CFRunLoopSourceRef`不透明类型，该类型必须实际附加到运行循环。它会将RunLoopSource对象本身作为上下文信息传递，以便回调例程具有指向该对象的指针。在工作线程调用**addToCurrentRunLoop**方法之前，不会安装输入源，此时将调用Run**LoopSourceScheduleRoutine**回调函数。将输入源添加到运行循环后，线程可以运行其运行循环以等待它。

Listing 3-7 Installing the run loop source

```
//初始化
- (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;
}

//添加到当前的runloop
- (void)addToCurrentRunLoop
{
    CFRunLoopRef runLoop = CFRunLoopGetCurrent();
    CFRunLoopAddSource(runLoop, runLoopSource, kCFRunLoopDefaultMode);
}
```

与输入源的客户端协调

为了使您的输入源有用，您需要操纵它并从另一个线程发出信号。输入源的全部意义是将其关联的线程置于休眠状态，直到有事情要做。这一事实要求应用程序中的其他线程了解输入源并有方法与之通信。

通知客户端输入源的一种方法是在输入源首次安装到其运行循环中时发送注册请求。您可以向任意多个客户注册您的输入源，或者您可以简单地向某个中央机构注册，然后将您的输入源出售给感兴趣的客户。清单3-8显示了由应用程序委托定义的注册方法，并在**RunLoopSource**对象的调度程序函数被调用时调用。此方法接收**RunLoopSource**对象提供的**RunLoopContext**对象并将其添加到其源列表中。此清单还显示了当输入源从其运行循环中移除时用于注销该输入源的例程

Listing 3-8 使用应用程序委托注册和删除输入源

```
//注册输入源
- (void)registerSource:(RunLoopContext*)sourceInfo;
{
    [sourcesToPing addObject:sourceInfo];
}

//移除输入源
- (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：清单3-4和清单3-6中显示了调用前面清单中方法的回调函数

向输入源发送信号

在将数据传递给输入源之后，客户端必须向该源发出信号并唤醒其运行循环。发信号给源让运行循环知道源已经准备好被处理。因为当信号出现时线程可能处于休眠状态，所以应该始终显式地唤醒运行循环。否则，可能会导致处理输入源的延迟。

清单3-9显示了RunLoopSource对象的fireCommandsOnRunLoop方法。当客户机准备好让源代码处理它们添加到缓冲区中的命令时，它们会调用此方法。

Listing 3-9 唤醒runloop

```
// 输入源处理缓存区中的命令
- (void)fireCommandsOnRunLoop:(CFRunLoopRef)runloop
{
      //客户端向输入源发送信号
    CFRunLoopSourceSignal(runLoopSource);
      // 唤醒runloop
    CFRunLoopWakeUp(runloop);
}
```

注意：切勿尝试通过传递自定义输入源来处理SIGHUP或其他类型的进程级信号。用于唤醒运行循环的Core Foundation函数不是信号安全的，不应在应用程序的信号处理程序例程中使用。有关信号处理程序例程的更多信息，请参见 `sigaction` 手册页。

5-2、配置计时器源

要创建计时器源，您要做的就是创建一个计时器对象并将其安排在运行循环中。在Cocoa中，使用 `NSTimer`类创建新的计时器对象，在Core Foundation中，使用 `CFRunLoopTimerRef`不透明类型。在内部，NSTimer类只是Core Foundation的扩展，它提供了一些便利功能，例如使用同一方法创建和安排计时器的功能。

在Cocoa中，可以使用以下任一类方法一次创建和调度计时器：

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

这些方法会创建计时器，并以默认模式（`NSDefaultRunLoopMode`）将其添加到当前线程的运行循环中。如果需要，还可以手动调度计时器，方法是创建 `NSTimer`对象，然后使用 `NSRunLoop`的 `addTimer:forMode:` 方法将其添加到运行循环中。两种技术基本上都具有相同的作用，但是可以为您提供对计时器配置的不同级别的控制。例如，如果您创建计时器并将其手动添加到运行循环中，则可以使用默认模式以外的其他模式来执行此操作。清单3-10显示了如何使用这两种技术创建计时器。第一个计时器的初始延迟为1秒，但此后每隔0.1秒有规律地触发一次。第二个计时器在最初的0.2秒延迟后开始触发，然后在此之后每0.2秒触发一次。

Listing 3-10 使用NSTimer创建和安排计时器

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

// Create and schedule the first timer. 手动调度定时器
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. 默认模式创建定时器
[NSTimer scheduledTimerWithTimeInterval:0.2
                        target:self
                        selector:@selector(myDoFireTimer2:)
                        userInfo:nil
                        repeats:YES];
```

清单3-11显示了使用Core Foundation函数配置计时器所需的代码。尽管此示例未在上下文结构中传递任何用户定义的信息，但您可以使用此结构传递计时器所需的任何自定义数据。有关此结构的内容的更多信息，请参见 *CFRunLoopTimer Reference*中的描述。

Listing 3-11 使用Core Foundation创建和调度计时器

```
//获取当前的runloop
CFRunLoopRef runLoop = CFRunLoopGetCurrent();

//CFRunLoopTimerContext 用于配置CFRunLoopTimer的行为
CFRunLoopTimerContext context = {0, NULL, NULL, NULL, NULL};

//使用Core Foundation 创建定时器
CFRunLoopTimerRef timer = CFRunLoopTimerCreate(kCFAllocatorDefault, 0.1,
0.3, 0, 0,
                                        &myCFTimerCallback, &context);

//将定时器加入到当前runloop
CFRunLoopAddTimer(runLoop, timer, kCFRunLoopCommonModes);
```

5-3、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，用于在线程之间或进程之间进行通信。以下各节说明如何使用几种不同类型的端口来设置端口通信。

配置NSMachPort对象

要与 NSMachPort对象建立本地连接，需要创建端口对象并将其添加到主线程的运行循环中。启动辅助线程时，将同一对象传递给线程的入口点函数。辅助线程可以使用同一对象将消息发送回您的主线程。

实现主线程代码

清单3-12显示了启动辅助工作线程的主线程代码。因为Cocoa框架执行许多配置端口和运行循环的中间步骤，所以**launchThread**方法明显短于其Core Foundation等效方法（(Listing 3-17）；但是，两者的行为几乎相同。一个区别是，此方法不是将本地端口的名称发送到工作线程，而是直接发送NSPort对象。

Listing 3-12 主线程启动方法

```
//启动线程
- (void)launchThread
{
      //创建 端口对象
    NSPort* myPort = [NSMachPort port];
    if (myPort)
    {
        // This class handles incoming port messages.
           //端口对象处理传入的端口消息。
        [myPort setDelegate:self];

        // Install the port as an input source on the current run loop.
           //将端口添加为当前runloop的输入源。
        [[NSRunLoop currentRunLoop] addPort:myPort
forMode:NSDefaultRunLoopMode];

        // Detach the thread. Let the worker release the port.
           //分离线程。并手动释放端口。
        [NSThread
detachNewThreadSelector:@selector(LaunchThreadWithPort:)
                toTarget:[MyWorkerClass class] withObject:myPort];
    }
}
```

为了在线程之间建立双向通信通道，您可能希望工作线程在签入消息中将其自己的本地端口发送到主线程。接收到签入消息可以使您的主线程知道在启动第二个线程时一切进展顺利，还为您提供了一种向该线程发送更多消息的方法。

清单3-13显示了主线程的 handlePortMessage: 方法。当数据到达线程自己的本地端口时调用此方法。当签入消息到达时，该方法直接从端口消息中检索辅助线程的端口，并将其保存以供以后使用。

Listing 3-13 处理Mach端口消息

```
#define kCheckinMessage 100

// Handle responses from the worker thread.
// 处理来自工作线程的响应。
- (void)handlePortMessage:(NSPortMessage *)portMessage
{
    unsigned int message = [portMessage msgid];
    NSPort* distantPort = nil;

    if (message == kCheckinMessage)
    {
        // Get the worker thread's communications port.
          // 获取工作线程的通信端口
        distantPort = [portMessage sendPort];

        // Retain and save the worker port for later use.
          // 保留并保存工作端口以供以后使用。
        [self storeDistantPort:distantPort];
    }
    else
    {
        // Handle other messages.处理其他消息
    }
}
```

实现辅助线程代码

对于辅助工作线程，必须配置线程并使用指定的端口将信息传递回主线程

清单3-14显示了设置工作线程的代码。在为线程创建自动释放池之后，该方法创建一个辅助对象来驱动线程的执行。worker对象的**sendCheckinMessage:**方法（如 Listing 3-15所示）为工作线程创建一个本地端口，并将签入消息发送回主线程。

Listing 3-14 使用Mach端口启动工作线程

```
// 使用Mach端口启动工作线程
+(void)LaunchThreadWithPort:(id)inData
{
     //创建自动释放池
    NSAutoreleasePool*  pool = [[NSAutoreleasePool alloc] init];

    // Set up the connection between this thread and the main
thread.
     //设置此线程与主线程之间的连接。
    NSPort* distantPort = (NSPort*)inData;

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

    // Let the run loop process things.
     // 让运行循环处理事情。
    do
    {
        [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode
                                 beforeDate:[NSDate distantFuture]];
    }
    while (![workerObj shouldExit]);

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

使用**NSMachPort**时，本地线程和远程线程可以将同一端口对象用于线程之间的单向通信。换句话说，一个线程创建的本地端口对象成为另一线程的远程端口对象。

清单3-15显示了辅助线程的签入例程。此方法设置自己的本地端口以用于将来的通信，然后将签入消息发送回主线程。该方法使用在LaunchThreadWithPort：方法中接收的端口对象作为消息的目标。

Listing 3-15 使用Mach端口发送签入消息

```
// Worker thread check-in method
// 工作线程签入方法
- (void)sendCheckinMessage:(NSPort*)outPort
{
    // Retain and save the remote port for future use.
     //保留并保存远程端口以备将来使用
    [self setRemotePort:outPort];

    // Create and configure the worker thread port.
     // 创建并配置工作线程端口。
    NSPort* myPort = [NSMachPort port];
    [myPort setDelegate:self];
    [[NSRunLoop currentRunLoop] addPort:myPort
forMode:NSDefaultRunLoopMode];

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

    if (messageObj)
    {
        // Finish configuring the message and send it
immediately.
          // 完成消息配置并立即发送。
        [messageObj setMsgId:setMsgid:kCheckinMessage];
        [messageObj sendBeforeDate:[NSDate date]];
```

配置NSMessagePort对象

要与 NSMessagePort对象建立本地连接，不能简单地在线程之间传递端口对象。远程消息端口必须按名称获取。在Cocoa中实现这一点需要用一个特定的名称注册本地端口，然后将该名称传递给远程线程，以便它可以获得适当的端口对象进行通信。清单3-16显示了在需要使用消息端口的情况下创建和注册端口的过程。

Listing 3-16 注册消息端口

```
//创建本地端口
NSPort* localPort = [[NSMessagePort alloc] init];

// Configure the object and add it to the current run loop.
// 配置对象并将其添加到当前运行循环
[localPort setDelegate:self];
[[NSRunLoop currentRunLoop] addPort:localPort
forMode:NSDefaultRunLoopMode];

// Register the port using a specific name. The name must be
unique.
//使用特定名称注册本地端口，名称必须唯一。
NSString* localPortName = [NSString
stringWithFormat:@“MyPortName"];
[[NSMessagePortNameServer sharedInstance] registerPort:localPort
                      name:localPortName];
```

在Core Foundation中配置基于端口的输入源

本节介绍如何使用CoreFoundation在应用程序的主线程和工作线程之间设置双向通信通道。

清单3-17显示了应用程序的主线程调用以启动工作线程的代码。代码所做的第一件事是设置一个 CFMessagePortRef 不透明类型来侦听来自工作线程的消息。工作线程需要连接端口的名称，以便将字符串值传递到工作线程的入口点函数。端口名在当前用户上下文中通常应该是唯一的；否则，您可能会遇到冲突。

Listing 3-17 将Core Foundation消息端口附加到新线程

```
#define kThreadStackSize        (8 *4096)

OSStatus MySpawnThread()
{
    // Create a local port for receiving responses.创建用于接收响应的本地端口
    CFStringRef myPortName;
    CFMessagePortRef myPort;
    CFRunLoopSourceRef rlSource;
    CFMessagePortContext context = {0, NULL, NULL, NULL, NULL};
    Boolean shouldFreeInfo;

    // Create a string with the port name.用端口名创建一个字符串
    myPortName = CFStringCreateWithFormat(NULL, NULL,
CFSTR("com.myapp.MainThread"));

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

    if (myPort != NULL)
    {
        // The port was successfully created.已成功创建端口。
       // Now create a run loop source for it.现在为它创建一个运行循环源。
        rlSource = CFMessagePortCreateRunLoopSource(NULL, myPort, 0);

        if (rlSource)
        {
            // Add the source to the current run loop.
            // 将run loop source添加到当前runloop
            CFRunLoopAddSource(CFRunLoopGetCurrent(), rlSource,
kCFRunLoopDefaultMode);

            // Once installed, these can be freed.安装后，即可释放它们。
            CFRelease(myPort);
            CFRelease(rlSource);
        }
    }

    // Create the thread and continue processing.创建线程并继续处理。
    MPTaskID        taskID;
    return(MPCreateTask(&ServerThreadEntryPoint,
                    (void*)myPortName,
                    kThreadStackSize,
                    NULL,
                    NULL,
                    NULL,
                    0,
                    &taskID));
}
```

安装了端口并启动了线程后，主线程可以在等待线程签入时继续其常规执行。当check-in消息到达时，它被调度到主线程的**MainThreadResponseHandler**函数，如清单3-18所示。此函数用于提取工作线程的端口名，并为将来的通信创建管道。

Listing 3-18 接收签入消息

```
#define kCheckinMessage 100

// Main thread port message handler 主线程端口消息处理程序
CFDataRef MainThreadResponseHandler(CFMessagePortRef local, SInt32 msgid, CFDataRef data,
void* info)
{
    if (msgid == kCheckinMessage) //如果是check-in消息
    {
        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.必须通过远程端口名获取消息
        messagePort = CFMessagePortCreateRemote(NULL, (CFStringRef)threadPortName);

        if (messagePort)
        {
            // Retain and save the thread's comm port for future reference.
             //保留并保存线程的comm端口以备将来参考
            AddPortToListOfActiveThreads(messagePort);

            // Since the port is retained by the previous function, release
            // it here.由于该端口由先前的功能保留，因此在此处释放
            CFRelease(messagePort);
        }

        // Clean up. 释放
        CFRelease(threadPortName);
        CFAllocatorDeallocate(NULL, buffer);
    }
    else
    {
        // Process other messages. 处理其他消息
    }

    return NULL;
}
```

配置了主线程后，剩下的唯一任务就是让新创建的工作线程创建自己的端口并签入。清单3-19显示了工作线程的入口点函数。该函数提取主线程的端口名，并使用它来创建回主线程的远程连接。然后，该函数为自己创建一个本地端口，在线程的运行循环中安装该端口，并向主线程发送一个包含本地端口名的签入消息。

Listing 3-19 Setting up the thread structures 设置线程结构

```
//工作线程的入口点函数
OSStatus ServerThreadEntryPoint(void* param)
{
    // Create the remote port to the main thread.
      //创建到主线程的远程端口。
    CFMessagePortRef mainThreadPort;
    CFStringRef portName = (CFStringRef)param;

    mainThreadPort = CFMessagePortCreateRemote(NULL, portName);

    // Free the string that was passed in param.释放在param中传递的字符串
    CFRelease(portName);

    // Create a port for the worker thread.为工作线程创建端口。
    CFStringRef myPortName = CFStringCreateWithFormat(NULL, NULL,
CFSTR("com.MyApp.Thread-%d"), MPCurrentTaskID());

    // Store the port in this thread's context info for later reference.
      // 将端口存储在此线程的上下文信息中，以供以后参考。
    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.
            //无法创建本地端口，所以终止线程。
        MPExit(0);
    }

    CFRunLoopSourceRef rlSource = CFMessagePortCreateRunLoopSource(NULL, myPort,
0);
```

```
   if (!rlSource)
    {
        // Couldn't create a local port, so kill the thread.
            //无法创建本地端口，所以终止线程。
        MPExit(0);
    }

      // Add the source to the current run loop.将源添加到当前运行循环。
    CFRunLoopAddSource(CFRunLoopGetCurrent(), rlSource,
kCFRunLoopDefaultMode);

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

    // Package up the port name and send the check-in message.
      //打包端口名并发送签入消息。
    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.清理线程数据结构。
    CFRelease(outData);
    CFAllocatorDeallocate(NULL, buffer);

    // Enter the run loop.进入运行循环
    CFRunLoopRun();
}
```

一旦进入运行循环，所有发送到线程端口的事件都由ProcessClientRequest函数处理。该函数的实现取决于线程所做的工作的类型，这里不显示。