IOS 多线程 RUNLOOP 机制

Run Loop

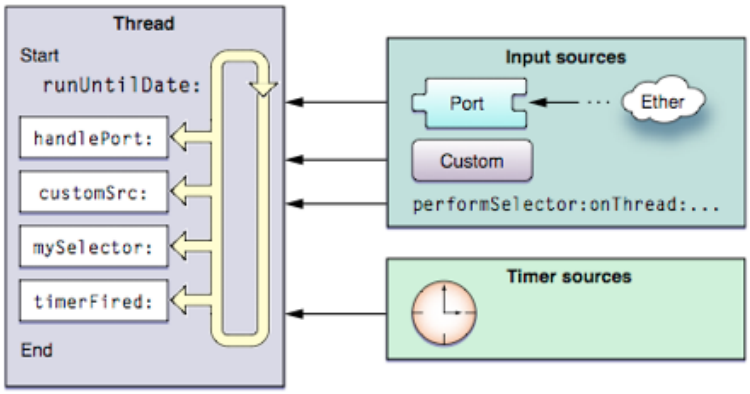
Run Loop 就是一个事件处理的循环，用来不停的调动工作以及处理输入事件。使用 Run Loop 的目的就是节省 CPU 效率，线程在有工作的时候忙于工作，而没工作的时候处于休眠状态。

一，Run Loop 剖析

Structure of a Run Loop and its sources

接下来看一下NSRunLoop具体的工作原理，首先是官方文档提供的说法，看图：

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



上图显示了线程的输入源

A，基于端口的输入源（Port Sources）

B，自定义输入源（Custom Sources）

C，Cocoa 执行 Selector 的源（"performSelector...方法" Sources）

D，定时源（Timer Sources ）

线程针对上面不同的输入源，有不同的处理机制

A，handlePort－－－处理基于端口的输入源

B，customSrc－－－处理用户自定义输入源

C，mySelector－－－处理 Selector 的源

D，timerFired－－－处理定时源

注：线程除了处理输入源，Run Loops 也会生成关于 Run Loop 行为的通知（notification）。Run Loop 观察者（Run-Loop Observers）可以收到这些通知，并在线程上面使用他们来作额为的处理

＝＝＝在新线程的 Run Loop 中注册 Observers：

－－－编写一个带有观测者的线程加载程序

```
- (void)observerRunLoop
{
    // 建立自动释放池
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    // 获得当前thread的Run loop
    NSRunLoop *myRunLoop = [NSRunLoop currentRunLoop];
    // 设置Run Loop observer的运行环境
    CFRunLoopObserverContext context = {0, self, NULL, NULL, NULL};

    // 创建Run loop observer对象
    // 第一个参数用于分配该observer对象的内存
    // 第二个参数用以设置该observer所要关注的的事件，详见回调函数myRunLoopObserver中注释
    // 第三个参数用于标识该observer是在第一次进入run loop时执行还是每次进入run loop处理时均执行
    // 第四个参数用于设置该observer的优先级
    // 第五个参数用于设置该observer的回调函数
    // 第六个参数用于设置该observer的运行环境
```

```
    CFRunLoopObserverRef observer =
CFRunLoopObserverCreate(kCFAllocatorDefault,
kCFRunLoopAllActivities, YES, 0,
&myRunLoopObserver, &context);
    if(observer)
    {
        // 将Cocoa的NSRunLoop类型转换程Core
Foundation的CFRunLoopRef类型
        CFRunLoopRef c  = [myRunLoop getCFRunLoop];
        // 将新建的observer加入到当前的thread的run loop
        CFRunLoopAddObserver(c, observer,
kCFRunLoopDefaultMode);
    }

    // Creates and returns a new NSTimer object and
schedules it on the current run loop in the default
mode
    [NSTimer scheduledTimerWithTimeInterval:0.1
target:self selector:@selector(doFireTimer:)
userInfo:nil repeats:YES];

    NSInteger loopCount = 10;
```

```
    do
    {
        // 启动当前thread的run loop直到所指定的时间到达，在run loop运行时，run loop会处理所有来自与该run loop联系的input sources的数据
        // 对于本例与当前run loop联系的input source只有Timer类型的source
        // 该Timer每隔0.1秒发送触发时间给run loop，run loop检测到该事件时会调用相应的处理方法（doFireTimer:）
        // 由于在run loop添加了observer，且设置observer对所有的run loop行为感兴趣
        // 当调用runUntilDate方法时，observer检测到run loop启动并进入循环，observer会调用其回调函数，第二个参数所传递的行为时kCFRunLoopEntry
        // observer检测到run loop的其他行为并调用回调函数的操作与上面的描述相类似
        [myRunLoop runUntilDate:[NSDate dateWithTimeIntervalSinceNow:1.0]];

        // 当run loop的运行时间到达时，会退出当前的run loop，observer同样会检测到run loop的退出行为，并调用其回调函数，第二个参数传递的行为是kCFRunLoopExit.
```

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

    // 释放自动释放池
    [pool release];
}
-(void)doFireTimer:(NSTimer *)timer
{
    NSLog(@"timer=%@",timer);
}
//===observer的回调函数:
void myRunLoopObserver(CFRunLoopObserverRef
observer, CFRunLoopActivity activity, void *info)
{
    switch(activity)
    {
            // The entrance of run loop, before
entering the event processing loop.
            // This activity occurs once for each call
to CFRunLoopRun / CFRunLoopRunInMode
        case kCFRunLoopEntry:
            NSLog(@"run loop entry");
```

```
            break;
            // Inside the event processing loop before
any timers are processed
        case kCFRunLoopBeforeTimers:
            NSLog(@"run loop before timers");
            break;
            // Inside the event processing loop before
any sources are processed
        case kCFRunLoopBeforeSources:
            NSLog(@"run loop before sources");
            break;
            // Inside the event processing loop before
the run loop sleeps, waiting for a source or timer
to fire
            // This activity does not occur if
CFRunLoopRunInMode is called with a timeout of o
seconds
            // It also does not occur in a particular
iteration of the event processing loop if a version
0 source fires
        case kCFRunLoopBeforeWaiting:
            NSLog(@"run loop before waiting");
```

```
            break;
            // Inside the event processing loop after
the run loop wakes up, but before processing the event
that woke it up
            // This activity occurs only if the run
loop did in fact go to sleep during the current loop
        case kCFRunLoopAfterWaiting:
            NSLog(@"run loop after waiting");
            break;
            // The exit of the run loop, after exiting
the event processing loop
            // This activity occurs once for each call
to CFRunLoopRun and CFRunLoopRunInMode
        case kCFRunLoopExit:
            NSLog(@"run loop exit");
            break;
            /*
             A combination of all the preceding stages
             case kCFRunLoopAllActivities:
             break;
             */
        default:
```

```
            break;
    }
}
```

1，Run Loop 模式－－－是所有要监测的输入源和定时源以及要通知的 run loop 注册观察者的集合。在 run loop 运行过程中，只有和模式相关的源才会被监测并允许他们传递事件消息。相反，没有被添加的输入源将会被过滤。

可以自定自己的 Run Loop 模式，但是每个模式必须有一个或者多个输入源，定时源或者 run loop 的观察者，否则，run loop 直接退出，Run Loop 模式将没有意义。

另，模式区分基于事件的源而非事件的种类。例如，不能使用模式只选择处理鼠标按下或者键盘事件。可以使用模式监听端口，而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 观测着。

表 1－1 列出了 cocoa 和 Core Foundation 预先定义的模式。

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.
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.

2，Run Loop 的输入源

A，基于端口的输入源

通过内置的端口相关的对象和函数，创建配置基于端口的输入源。相关的端口函数－－－CFMachPort/CFMessagePortRef/CFSocketRf

B，自定义输入源

自定义输入源使用 CFRunLoopSourceRef 对象创建，它需要自定义自己的行为和消息传递机制

C，Cocoa 执行 Selector 的源

和基于端口的源一样，执行 Selector 的请求会在目标线程上序列化，减缓在线程上允许许多各方法容易引起的同步问题。两者区别：一个 Selector 执行完成后会自动从 Run Loop 上移除。

Table： 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 NSThread object. 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.

D，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。

E，Run Loop 观察者

源是合适的同步／异步事件发生时触发。观察者则是在Run Loop本身运行的特定时候触发。

观察者触发的相关事件（参考上面红色程序里面的函数：myRunLoopObserves(...)）

1）Run Loop 入口

2）Run Loop 何时处理一个定时器

3）Run Loop 何时处理一个输入源

4）Run Loop 何时进入休眠状态

5）Run Loop 何时被唤醒，但在唤醒之前要处理的事件

6）Run Loop 终止

注： 1,观察者通过 CFRunLoopObserverRef 对象创建的

2,观察者会在相应事件发生之前传递消息,所以通知的时间和事件实际发生的时间之间肯定有误差

F，Run Loop 的事件队列－－包括观察者的事件队列

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输入源

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了并处于等待状态，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下面任一事件发生

A）某一事件到达基于端口的源

B）定时器启动

C）Run Loop 设置的时间已经超时

D）Run Loop 被显式唤醒

8）通知观察者线程被唤醒

9）处理未处理的事件

A）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

B）如果输入源启动，传递相应消息

C）如果 Run Loop 被显式唤醒而且时间还没有超时，重启 Run Loop，进入步骤 2

10）通知观察者 Run Loop 结束

二，何时使用 Run Loop

对于辅助线程，在需要和线程有更多交互时，才使用 Run Loop。

比如：1）使用端口或者自定义输入源来和其他线程通讯

2）使用线程定时器

3)Cocoa中使用任何performSelector...的方法(参考Table：Performing selectors on other threads)

4）使线程周期性工作

三，如何使用 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，使用 Run Loop 包活获取——配置——启动——退出四个过程

1，获取 Run Loop 的对象

A，通过 NSRunLoop 获取

```
// 获得当前 thread 的 Run loop
NSRunLoop *myRunLoop = [NSRunLoop currentRunLoop];
// 将 Cocoa 的 NSRunLoop 类型转换程 Core Foundation 的 CFRunLoopRef 类型
CFRunLoopRef c = [myRunLoop getCFRunLoop];
```

B，使用 CFRunLoopGetCurrent()函数

2，配置 Run Loop

所谓配置 Run Loop 主要是给 Run Loop 添加输入源,定时器或者添加观察者,即设置 Run Loop 模式。上面函数- (void)observerRunLoop 就是配置了一个带有观察者，添加了一个定时器的 Run Loop 线程。相关对象———CFRunLoopObserverRef 对象和 CFRunLoopAddObserver 函数

3，启动 Run Loop

一个 Run Loop 通常必须包含一个输入源或者定时器来监听事件，如果一个都没有，Run Loop 启动后立即退出。

启动 Run Loop 的方式

1）无条件的———最简单的启动方法，但是退出 Run Loop 的唯一方式就是杀死它。

2）设置超时时间———预设超时时间来运行 Run Loop。Run Loop 运行直到某一事件到达或者规定的时间已经到期。

A，如果是事件到达，消息被传递给相应的处理程序来处理，然后 Run Loop 退出。可以循环重启 Run Loop 来等待下一事件。

B,如果是规定的时间到期了,可以使用此段时间来做任何的其他工作,然后 Run Loop 退出,或者直接循环重启 Run Loop。

3）特定模式

使用特定模式运行 Run Loop

```
//=====Running a run loop: skeleton
- (void)skeletonThreadMain
{
    BOOL done = NO;
    // Set up a autorelease pool here if not using
garbage collection.
    //.........
    // Add Sources/Timers to the run loop  and do any
other setup
    //.........

    // The cycle of run loop
    do {
        // start  the run loop
        //after each source handle

        SInt32
result=CFRunLoopRunInMode(kCFRunLoopDefaultMode,1
0,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 (YES);

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

注：可以递归运行 Run Loop，即可以使用 CFRunLoopRun，CFRunLoopRunInMode 或者任一 NSRunLoop 的方法在输入源或者定时器的处理程序里面启动 Run Loop

4，退出 Run Loop

有两种方法可以让 Run Loop 在处理事件之前退出

A，给 Run Loop 设置超时时间

B，通知 Run Loop 停止－－－使用 CFRunLoopStopped 函数可以显式停止 run loop

5，线程安全和 Run Loop 对象

NSRunLoop 线程不安全

CFRunLoop 线程安全

对 Run Loop 对象的修改尽可能在所有线程内部完成这些操作

四，配置 Run Loop 源－－－配置源的过程就是源的创建调用过程

配置过程分为以下几个阶段－－－定义/创建（一个源）－－－安装（将输入源安装到所在 Run Loop 中）－－－注册（将输入源注册到客户端，协调输入源的客户端）－－－调用（通知输入源，开始工作）

4-1，定义自定义输入源

创建自定义输入源需要定义以下内容

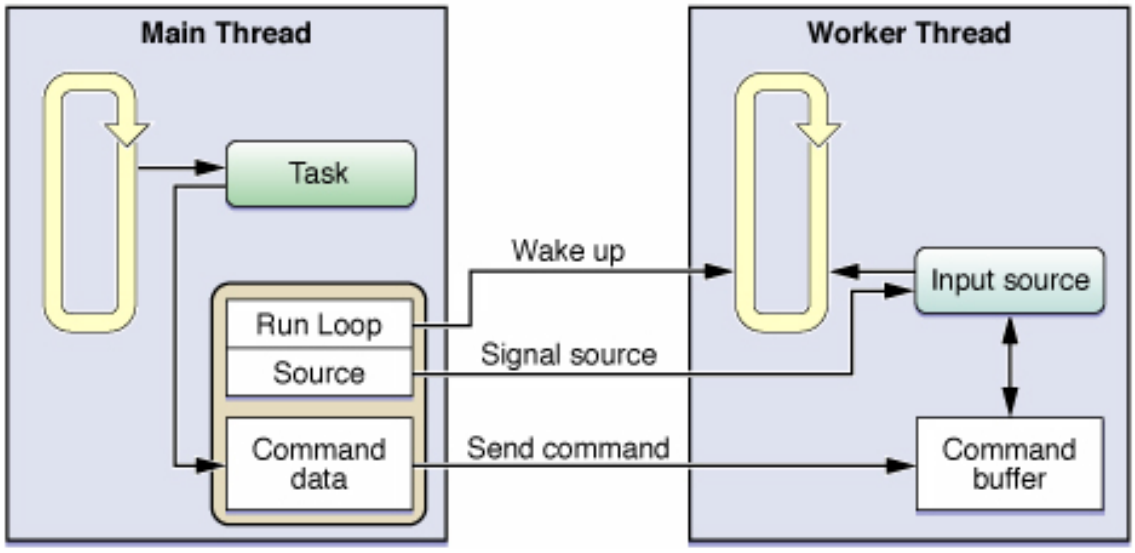
1）输入源要处理的信息

2）使感兴趣的客户端知道如何和输入源交互的调度例程

3）处理其他任何客户发送请求的例程

4）使输入源失效的取消例程

Figure 3-2 Operating a custom input source



上图的处理流程：主线程（Main Thread）发起任务（Task）给工作线程（Worker Thread），主线程会给命令缓冲区（send command－－>Command Buffer），通知输入源（signal source-->Input Source），并唤醒工作线程（Wake Up-->Worker Thread）。工作线程收到唤醒命令，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。

注：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（Command Buffer），因此这些访问必须使同步的

1）定义输入源（The custom input source object definition）

下面代码中，定义了 RunLoopSource 对象，它管理命令缓冲区，并以此来接收其他线程的消息。RunLoopContext 对象是一个用来传递 RunLoopSource 对象（RunLoopSource* source）和“run loop 引用”（CFRunLoopRef runLoop）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)sourecFired;
// Client interface for registering commands to
process
- (void)addCommand:(NSInteger)command
withData:(id)data;
```

```
-
(void)fireAllCommandsOnRunLoop:(CFRunLoopRef)runL
oop;
@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
```

===当将输入源附加到 run loop 时，调用这个协调调度例程，将源注册到客户端（可以理解为其他线程）

```
// Listing 3-4 Scheduling a run loop source
//当 source 添加进 runloop 的时候，调用此回调方法
<== CFRunLoopAddSource(runLoop, source, mode);
//当source添加进runloop的时候，调用此回调方法 <==
CFRunLoopAddSource(runLoop, source, mode);
void RunLoopSourceScheduleRoutine (void *info,
CFRunLoopRef rl, CFStringRef mode)
{
    RunLoopSource* obj = (RunLoopSource*)info;
    AppDelegate* del = (AppDelegate *)[UIApplication
sharedApplication].delegate;
    RunLoopContext* theContext = [[RunLoopContext
alloc] initWithSource:obj andLoop:rl];
    [del
performSelectorOnMainThread:@selector(registerSou
rce:) withObject:theContext
                 waitUntilDone:NO];
}
```

===在输入源被告知（signal source）时，调用这个处理例程，这儿只是简单的调用了 [obj sourceFired]方法

```
//  Listing 3-5 Performing work in the input source
//当sourcer接收到消息的时候，调用此回调方法//当sourcer接收到消息的时候，调用此回调方法
(CFRunLoopSourceSignal(source);CFRunLoopWakeUp(runLoop);
void RunLoopSourcePerformRoutine (void *info)
{
    RunLoopSource* obj = (RunLoopSource*)info;
    [obj sourceFired];
}
```

===如果使用 CFRunLoopSourceInvalidate/CFRunLoopRemoveSource 函数把输入源从 run loop 里面移除的话，系统会调用这个取消例程，并且把输入源从注册的客户端（可以理解为其他线程）里面移除

```
//  Listing 3-6 Invalidating an input source ==
//当 source 从 runloop 里删除的时候，调用此回调方法
<== CFRunLoopRemoveSource(runLoop, source, mode);
void RunLoopSourceCancelRoutine (void *info,
CFRunLoopRef rl, CFStringRef mode)
{
    RunLoopSource* obj = (RunLoopSource*)info;
    AppDelegate* del = (AppDelegate *)[UIApplication
sharedApplication].delegate;
    RunLoopContext* theContext = [[RunLoopContext
alloc] initWithSource:obj andLoop:rl];
```

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

2）安装输入源到 Run Loop－－－分两步首先初始化一个输入源，然后将这个输入源添加到当前 Run Loop 里面

```
// List 3-7 Installing the run loop source
- (id)init
{
    /*
     // Setup the context.
     context.version = 0;
     context.info = self;
     context.retain = NULL;
     context.release = NULL;
     context.copyDescription = CFCopyDescription;
     context.equal = CFEqual;
     context.hash = CFHash;
     context.schedule =
RunLoopSourceScheduleRoutine;
     context.cancel = RunLoopSourceCancelRoutine;
```

```
     context.perform = RunLoopSourcePerformRoutine;
     */
    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();
    //Add the new CFRunLoopSourceRef to the indicated
runloop, 并且回调RunLoopSourceScheduleRoutine函数
    CFRunLoopAddSource(runLoop, runLoopSource,
kCFRunLoopDefaultMode);
}
```

3）协调输入源的客户端（将输入源注册到客户端）

输入源的主要工作就是将与输入源相关联的线程置于休眠状态，直到有事件发生。要达到这个目的，首先客户端（其他线程）知道有该输入源信息，并且有办法与之通信。
通知客户端关于这个输入源信息的方法之一，就是当该输入源开始安装到你的 run loop 上面后发送注册请求给相关的客户端，该输入源可以注册到任意数量的客户端
也可以通过由代理将输入源注册到感兴趣的客户端

```
//= = = 下面显示了应用委托（AppDelegate）定义的注册方法及从应用委托（AppDelegate）中移除的方法
- (void)registerSource:(RunLoopContext *)sourceInfo
{
    NSMutableArray *sourceToPing;
    [sourceToPing addObject:sourceInfo];
}

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

    NSMutableArray *sourceToPing;

    for(RunLoopContext *context in sourceToPing)
    {
        if([context isEqual:sourceInfo])
        {
            objToRemove = context;
```

```
            break;
        }
    }

    if(objToRemove)
    {
        [sourceToPing removeObject:objToRemove];
    }
}
```

注：上面两个函数分别在 RunLoopSourceScheduleRoutine / RunLoopSourceCancelRoutine 函数中被调用
4)通知输入源－－－客户端(其他线程)发数据到输入源，分两步首先发信号给输入源(signal source)，然后唤醒输入源的 run loop
＝＝＝下面显示了客户端发送数据到输入源的方法，在本例中这个方法被放在 RunLoopSource 对象里面

```
// Listing 3-9 Waking up the run loop

-
(void)fireAllCommandsOnRunLoop:(CFRunLoopRef)runL
oop
{
    //当手动调用此方法的时候，将会触发
RunLoopSourceContext的performCallback
    CFRunLoopSourceSignal(runLoopSource);
    CFRunLoopWakeUp(runLoop);
}
```

注：1，输入源就是一类事件（命令）处理机制。他是线程间的事件（命令）异步通讯机制，所以不能试图通过这个机制实现进程间的通讯。

2，因为 CFRunLoopWakeUp 函数不是信号安全的，所以对 run loop 的唤醒，不能在应用信号处理例程（RunLoopSourcePerformRoutine）里面使用。

4-2，配置定时源

配置源的过程其实是源在相关 run loop 的使用过程，包括定义（创建），安装（添加到相关 run loop），注册（注册到其他需要交互的线程），以及使用四个过程。但是因为定时源在预设的时间点同步方式传递消息，是线程通知自己做某事情的一种方法，所以配置定时源，主要工作是创建，和安装。其他两个过程 run loop 自己就可以完成。

创建定时源有两种方式

＝＝＝一种是使用 NSTimer 对象创建，一种是使用 CFRunLoopTimerRef 对象创建，及将创建的事例安装到 Run Loop 中

```
// Listing 3-10 Creating and scheduling timers using NSTimer
-(void)createAndScheduleTimerToRunLoopUsingNSTimer
{
    // get the current run loop
    NSRunLoop *myRunLoop = [NSRunLoop currentRunLoop];

    // create and schedule the first timer  === 可以配置到不同的run loop模式
    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 === 只
能在run loop的NSDefaultRunLoopMode下有效
    [NSTimer scheduledTimerWithTimeInterval:0.1
target:self selector:@selector(myDoFireTimer2:)
userInfo:nil repeats:YES];
}

// Listing 3-11 Creating and scheduling a timer using
Core Foundation
-
(void)createAndScheduleTimerToRunLoopUsingCFRunLo
opTimerRef
{
    // get the current run loop
    CFRunLoopRef runLoop = CFRunLoopGetCurrent();
```

```
    CFRunLoopTimerContext context = {0, NULL, NULL,
NULL, NULL};
    CFRunLoopTimerRef timer =
CFRunLoopTimerCreate(kCFAllocatorDefault, 0.1, 0.3,
0, 0, &myCFTimerCallBack, &context);

    // add the CFRunLoopTimerRef to run loop
kCFRunLoopCommonModes mode
    CFRunLoopAddTimer(runLoop, timer,
kCFRunLoopCommonModes);
}
```

4-2，配置基于端口的输入源

iOS 系统提供了基于端口的输入源对象，用以线程或进程间通讯。

1）配置 NSMachPort 对象－－－本地线程间通信，通过传递端口对象变量进行端口间通讯

基本机制：A 线程（父线程）创建 NSMachPort 对象，并加入 A 线程的 run loop。当创建 B 线程（辅助线程）时，将创建的 NSMachPort 对象传递到主体入口点，B 线程（辅助线程）就可以使用相同的端口对象将消息传回 A 线程（父线程）。

A，实现 A 线程（父线程）的代码

===在下面代码中，创建一个 NSPort 对象，并把这个端口对象安装（add）到线程的 Run Loop 里面，同时创建一个新的线程，并把 NSPort 对象作为参数传入到新线程的主体入口点

```
// Listing 3-12 Main thread launch sub-thread method
- (void)lanchThread
{
    NSPort *myPort = [NSPort port];
```

```
    if(myPort)
    {
        // This class handles incoming port message
        [myPort setDelegate:self];

        // install the port as an input source
        [[NSRunLoop currentRunLoop] addPort:myPort
forMode:NSDefaultRunLoopMode];

        // Detach the thread.Let the worker release
the port--加载一个子线程
        [NSThread
detachNewThreadSelector:@selector(LaunchThreadWit
hPort:) toTarget:[MyWorkerThread Class]
withObject:myPort];
    }
}
```

===NSPort是通过代理模式传送消息，下面这段代码是NSPortDelegate的代理方法，用以从B线程回传消息

```
（(NSPortMessage *)message）
#pragma mark
#define kCheckinMessage     100
```

```
#pragma mark NSPortDelegate Method
// This is the NSPort Delegate Method
// It handle responses from the worker thread(B线程)
- (void)handlePortMessage:(NSPortMessage *)message
{
    uint32_t mssgId = [message msgid];
    NSPort *distantPort = nil;

    if(message == kCheckinMessage)
    {
        // get the worker thread's communication port
        distantPort = [message sendPort];

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

```
    }
}
```

B，B线程（辅助线程）的实现代码

＝＝＝下面是辅助线程的类方法，里面创建了辅助线程实例，并把传入的NSPort保存下来。同时，通过NSMachPort发送一个Check-In Message（NSPortMessage类型，在这个Message类型中设置了SendPort/ReceivePort，链接这两个端口）给A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;

    MyWorkerThread *workObj = [[self alloc] init];
    [workObj sendCheckinMessage:distantPort];
    [distantPort release];
```

```
    // Let the run loop process things
    do
    {

    }while (![workerObj shouldExit]);

    [workObj release];
    [pool release];
}

// Listing 3-15 Sending the check-in message using
Mach ports
- (void)sendCheckinMessage:(NSPort *)outPort
{
    // Retain and save the remote port for future use
    [self setRemotePort:outPort]; // set NSPort
*remotePort;

    // 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];
         }
         }
```

===注：最终，B 线程 NSMachPort）端口传递消息给 A 线程的（NSPort）端口（handlePortMessage 代理方法）。

2）配置 NSMessagePort 对象

===只能在一个设备内程序间通信，不能在不同设备间通信。将端口名称注册到 NSMessagePortNameServer 里面，其他线程通过这个端口名称从 NSMessagePortNameServer 来获取这个端口对象。

```
// Listing 3-9 Waking up the run loop
```

```
-
(void)fireAllCommandsOnRunLoop:(CFRunLoopRef)runL
oop
{
    //当手动调用此方法的时候，将会触发
RunLoopSourceContext的performCallback
    CFRunLoopSourceSignal(runLoopSource);
    CFRunLoopWakeUp(runLoop);

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

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

    // register the port using the specific name, and
The name is unique
    NSString *localPortName = [NSString
stringWithFormat:@"MyPortName"];
```

```
    // there is only NSMessagePortNameServer in the
mac os x system
    //[[NSMessagePortNameServer sharedInstance]
registerPort:localPort name:localPortName];
}
```

3）在 Core Fundation 中配置基于端口的源

CFMessagePortRef－－－实现本地设备内程序间通讯。CFMessagePort objects provide a communications channel that can transmit arbitrary data between multiple threads or processes on the local machine.

You create a local message port with CFMessagePortCreateLocal and make it available to other processes by giving it a name, either when you create it or later with CFMessagePortSetName. Other processes then connect to it using CFMessagePortCreateRemote, specifying the name of the port.

To listen for messages, you need to create a run loop source with CFMessagePortCreateRunLoopSource and add it to a run loop with CFRunLoopAddSource.

Important: If you want to tear down the connection, you must invalidate the port (using CFMessagePortInvalidate) before releasing the runloop source and the message port object.

Your message port's callback function will be called when a message arrives. To send data, you store the data in a CFData object and call CFMessagePortSendRequest. You can optionally have the function wait for a reply and return the reply in another CFData object.

Message ports only support communication on the local machine. For network communication, you have to use a CFSocket object.

Functions by Task Creating a CFMessagePort Object

* CFMessagePortCreateLocal－－－监听线程创建监听端口，Returns a local CFMessagePort object. * CFMessagePortCreateRemote－－－工作线程用来获取监听线程的端口对象，Returns a CFMessagePort object connected to a remote port.

Configuring a CFMessagePort Object

* CFMessagePortCreateRunLoopSource * CFMessagePortSetInvalidationCallBack * CFMessagePortSetName

Using a Message Port

* CFMessagePortInvalidate * CFMessagePortSendRequest－－－工作线程给监听端口发送请求

Examining a Message Port

* CFMessagePortGetContext * CFMessagePortGetInvalidationCallBack * CFMessagePortGetName * CFMessagePortIsRemote * CFMessagePortIsValid

Getting the CFMessagePort Type ID

* CFMessagePortGetTypeID

===主线程代码，创建 CFMessagePortRef 本地端口作为监听端口，并注册到当前 run loop。MainThreadResponseHandler 函数是消息回调函数，当有消息从工作线程传递过来时将调用这个函数

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

#define kThreadStackSize (8 *4096)


void MySpawnThread( void )
{
    // 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 for 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
        rlSource =
CFMessagePortCreateRunLoopSource(NULL, myPort, 0);
        if(rlSource != NULL)
        {
            // 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
    NSThread *thread = [[NSThread alloc]
initWithTarget:[MyThreadClass Class]
selector:@selector(ServerThreadEntryPoint:)
object:@"com.myapp.MainThread"];
    [thread setStackSize:kThreadStackSize];
    [thread start];
}

// Listing 3-18 Receiving the checkin me
#define kCheckinMessage 100

CFDataRef
MainThreadResponseHandler(CFMessagePortRef local,
SInt32 msgid, CFDataRef data, void*info)
{
    if(msgid == kCheckinMessage)
    {
        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);
#if 0
        CFMessagePortRef messagePort;

        // You must obtain a remote message port by
name for future reference
        messagePort =
CFMessagePortCreateRemote(NULL, threadPortName);
        if(messagePort)
        {
            // Retain and save the thread's com port
for future reference

AddPortToListOfActiveThreads(messagePort);
```

```
            // Since the port is retained by the
previous function, release it here
            CFRelease(messagePort);
        }
#endif
        // clean up
        CFRelease(threadPortName);
        CFAllocatorDeallocate(NULL, buffer);
    }
    else
    {
        // handle other messages
    }
    return NULL;
}
```

===工作线程代码，获取主线程端口，通过CFMessagePortSendRequest传递消息到主线程端口

```
// Listing 3-19 Setting up the thread structures
+ (void)ServerThreadEntryPoint:(NSString *)name
{
    // create the remote port to main thread
    CFMessagePortRef mainThreadPort;
```

```
    CFStringRef portName = (CFStringRef)name;
    mainThreadPort = CFMessagePortCreateRemote(NULL,
portName);

#if 1
    // Create a port for the worker thread.
    CFStringRef myPortName =
CFStringCreateWithFormat(NULL, NULL,
CFSTR("com.MyApp.Thread-%d"),CFRunLoopGetTypeID()
);
    //  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)
```

```
    {
        CFRunLoopSourceRef 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);

            // 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,
&returnData);

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

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

    }
#endif
}
```