多线程学习

> *当前使用的大多数操作系统都能同时运行几个程序(独立运行的程序又称之为进程)，对于同一个程序，它又可以分成若干个独立的执行流，我们称之为线程，线程提供了多任务处理的能力。用进程和线程的观点来研究软件是当今普遍采用的方法，进程和线程的概念的出现，对提高软件的并行性有着重要的意义。现在大部分的App都是多线程多任务处理，单线程的软件是不可想象的。因此掌握多线程多任务设计方法对每个程序员都是必需要掌握的。本节我们将会学习使用多种方式编写多线程应用程序。*
> *先来看生活中的一个例子，车站售票时如果仅仅有一个窗口售票，我们可以把售票系统看作为单线程，如果旅客过多（任务过多）时，就会造成售票过慢的问题，这时只需增加售票窗口同时售票，来加快售票速度。这就是多线程。你还能举出哪些多线程在生活中的例子呢?*

Apple给出的多线程编程定义：

> *Threads are a relatively lightweight way to implement multiple paths of execution inside of an application.翻译：线程是一个相对轻量级的，在应用程序内部，实现执行的多条路径的方式。*

章节一 多线程中的基本概念：

- 线程（Thread）：有时被称为轻量级进程(Lightweight Process，LWP），是程序执行流的最小单元。详情
- 进程（Process）：在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。详情
- 并行（concurrence）：一组程序按独立异步的速度执行，不等于时间上的重叠（同一个时刻发生)。要区别并发。
- 串行（serialization）：串行是与并行对应的概念，是指几个任务依次执行。
- 同步（synchronization）：在多线程编程中是指一个任务需要等待另一个任务反馈结果才能继续执行。是指两个任务之间的关系。
- 异步（asynchronization）：多线程中，一个任务不需等待其他任务反馈，也可以继续执行。与同步概念对应。
- 并发（concurrence）：在同一个时间段内，两个或多个程序执行，有时间上的重叠（宏观上是同时，只有多核心处理器才能做到微观上真正的同时）。

举例来说:

> *在田径赛场上，以上概念均可以体现出来，把每个运动员看成一行代码，运动员奔跑代表开始执行任务，跑完400米代表任务结束。*

1. **线程**就像一条一条跑道，不同的运动员在不同的跑道上，**可以达到互不干扰的目的**，这也是多线程的作用。
2. **进程**就像竞技场，每个竞技场都有自己的跑道，也就是说进程是线程的容器。在80年代以前，是没有线程概念的，而每次开启新的进程开销过大，就像一个竞技场只有一条跑道，新建竞技场需要花费的资源要远远大于多画一条跑道的成本。为了节约成本，就诞生了线程编程。
3. **并行**表示两个跑道之间的关系，运动员在两条跑道上奔跑是可以互不干扰的，任务的开始结束不会受到另个一任务的干扰。

4. **串行**就像接力赛，运动员之间需要交接接力棒。也就是说一个任务完成下一个任务才能开始。
5. **并发**是指并行的两个任务同时发生，就像运动员起跑，描述的是一个时间关系。但是起跑肯定有快有慢，有可能做不到完全同时，并发宏观是同时，围观有可能不是同时。
6. **同步**是描述两个事件交互时的概念，比如F1赛车中，由于速度过快导致车胎过热，磨损很快，需要频繁更换轮胎，把车手开车看做任务A，技师们更换轮胎看做任务B，当需要更换轮胎，A请求B任务，两个任务可以在不同的线程中，只有B任务寄予A反馈，A才能继续其他操作，B任务执行期间，A处于等待状态，也就是说会被阻塞。
7. **异步**是和同步概念对应的，A请求B任务时，A无需等待B回应可以继续其他操作。

> 
> *对于赛车运动，时间是制胜的关键。在赛道上要争取1秒非常难，所以每一秒时间都很宝贵。如何更加迅速更换轮胎，成为每支车队的必修课题。正常情况下F1进维修区只换轮胎的话，时间为4秒。而红牛车队最快换胎速度仅需1.8秒。换胎速度快主要有以下几个因素：第一是车轮结构，F1在内的所有赛车，只要卸下一个中央螺栓就可以拆下车轮，而平常的车辆需要拆5-6个才行。第二是工具，民用车的4S店里正常都是用气泵，个别比较Out的还用手拧，F1的是气动，但是那个气泵的扭矩和转速都是专门定制的，根本无需换胎工在哪儿拧上之后再用手拧一下看看有没有上紧，一旦上了，扭矩必然是正好，也必须是正好。第三是人工，F1的换胎工都是经过特殊训练的，而且换一次轮胎，需要14个人，四个轮胎，每个轮胎三个人伺候，一个操作气泵拆装螺栓，一个卸轮胎一个装轮胎，另外两人一前一后操作撬杠。这里和我们常说的封装思想有很大的联系，每一个技师精通于自己擅长的工作，能够提高效率，并且善于管理。*

理解了以上7个概念，对于多线程编程来说是至关重要的。

章节二 多线程在iOS中的实现

多线程编程在iOS中具有非常重要的地位，乔布斯在2007年发布会上，用过这么一段有趣的话 “an ipod,a phone,a internet communicator，are you getting it?”，最终Apple用一个设备给出了答案而不是三个设备，那就是iPhone。毫无疑问iOS要有一个网络交互系统，访问网络难免会有延迟，多线程编程也就在iOS程序设计中占有了很大地位。学习多线程编程是一个漫长的过程，这里有两点需要提成的能力：1.**多线程工具** 2.**多线程思维**

我们将在本节学习**多线程工具**的使用，熟练使用需要几周或者几个月的时间，但是多线程的**编程思维**需要你用更长的时间去完善。

小节一 Run Loop

了解Run Loop需要从以下几点出发：

- Run Loop是干什么用的
- 为什么要用Run Loop
- Run Loop如何实现的

Run Loop是干什么用的

普通的代码执行：

```
int main(int argc, char * argv[]) {
    NSLog(@"hello world");
    return 0;
}
```

代码是以顺序结构执行的，两行代码后就会退出。显然我们日常使用的大部分应用程序不是以此方式运行的。

事件循环代码执行：

```
int main(int argc, char * argv[]) {

    while (AppIsRunning) {
        //睡眠，等待事件
        id whoWakesMe = sleepForWakeUp();
        //被事件唤醒后获取事件
        id event = getEvent(whoWakesMe);
        //处理事件
        handleEvent(event);
        //进入下次循环
    }
    return 0;
}
```

简单的说事件循环结构使应用程序能够持久运行，不断的获取事件。上述代码就是一个Run Loop工作的大概模式，但是真正的Run Loop要更加复杂。

为什么要用Run Loop

- 使程序一直运行，并且接收用户的输入
- 决定应用程序在何时应该处理哪些Event
- 调用解耦合（Message Queue）
- 节省CPU

Run Loops In Cocoa

相关类

- NSRunLoop，存在于Foundation框架中
- CFRunLoop，存在于Core Foundation框架中，NSRunLoop是CFRunLoop的上层封装，如同NSString和CFString的关系，真正处理数据的是Core Foundation中的类
- Run Loops 的实现用到了系统中的很多技术，如GCD，mach kernel，block，pthread...

依赖于Run Loop的类有很多，如：NSTimer，UIEvent，Autorelease，NSObject（NSDelayedPerforming，NSThreadPerformAddition），CADisplayLink，CATransition，CAAnimation，dispatch_get_main_queue()，NSURLConnection,AFNetworking...

调用堆栈

什么是调用堆栈，调用堆栈简单地讲就是程序从运行开始到当前的函数（方法）之间，所有调用过的函数（方法）的前后顺序。

iOS打印调用堆栈可以使用以下方法：

```
NSLog(@"%@",[NSThread callStackSymbols]);
```

打印结果如下：

```
2015-09-11 01:55:26.717 05 RunLoopTest[2459:869295] (
    0   05 RunLoopTest                      0x000000010ed38b2c -[ViewController viewDidLoad] +
76
    1   UIKit                               0x000000010fafa1d0 -[UIViewController
loadViewIfRequired] + 738
    2   UIKit                               0x000000010fafa3ce -[UIViewController view] + 27
    3   UIKit                               0x000000010fa15289 -[UIWindow
addRootViewControllerViewIfPossible] + 58
    4   UIKit                               0x000000010fa1564f -[UIWindow _setHidden:forced:]
+ 247
    5   UIKit                               0x000000010fa21de1 -[UIWindow makeKeyAndVisible] +
42
    6   UIKit                               0x000000010f9c5417 -[UIApplication
_callInitializationDelegatesForMainScene:transitionContext:] + 2732
    7   UIKit                               0x000000010f9c819e -[UIApplication
_runWithMainScene:transitionContext:completion:] + 1349
    8   UIKit                               0x000000010f9c7095 -[UIApplication
workspaceDidEndTransaction:] + 179
    9   FrontBoardServices                  0x0000000111218a5e5 __31-[FBSSerialQueue
performAsync:]_block_invoke_2 + 21
    10  CoreFoundation                      0x000000010f50441c
__CFRUNLOOP_IS_CALLING_OUT_TO_A_BLOCK__ + 12
    11  CoreFoundation                      0x000000010f4fa165 __CFRunLoopDoBlocks + 341
    12  CoreFoundation                      0x000000010f4f9f25 __CFRunLoopRun + 2389
    13  CoreFoundation                      0x000000010f4f9366 CFRunLoopRunSpecific + 470
    14  UIKit                               0x000000010f9c6b02 -[UIApplication _run] + 413
    15  UIKit                               0x000000010f9c98c0 UIApplicationMain + 1282
    16  05 RunLoopTest                      0x000000010ed38e7f main + 111
    17  libdyld.dylib                       0x0000000111b74145 start + 1
)
```

序号从17到0来看是顺序调用，从17开始解释

- 17 `start` 函数，函数所在的位置是 `libdyld.dylib`，所有程序开始的位置
- 16 `main` 函数，就是我们在 `main.m` 文件中的 `main` 函数
- 15 `UIApplicationMain` 函数就是在 `main` 函数中我们调用的函数，所在位置是 `UIKit` 框架
- 14 `UIApplication` 调用了 `run` 方法
- **13-10 都是在Core Foundation框架中的，这就是Run Loop的主要工作函数了，也是下面要讲解的主要内容**
- 中间省略...
- 1 可以看到 `loadViewIfRequired` 方法,如果在ViewController中重写该方法，会发现在loadView时会调用该方法

  ```
  -(void)loadViewIfRequired{
      //loadViewIfRequired 是loadView中调用的一个方法，作用如之前的所讲，如果_view是空指针会加载一个UIView给_view赋值
  }
  ```

对比下面的调用堆栈，你会发现多出了几点：

```
2015-09-11 22:20:51.611 05 RunLoopTest[2653:979182] (
    0   05 RunLoopTest                      0x0000000109254b5d -[ViewController tap:] + 61
    1   UIKit                               0x0000000109ee6d62 -[UIApplication
sendAction:to:from:forEvent:] + 75
    2   UIKit                               0x0000000109ff850a -[UIControl
_sendActionsForEvents:withEvent:] + 467
    3   UIKit                               0x0000000109ff78d9 -[UIControl
touchesEnded:withEvent:] + 522
    4   UIKit                               0x0000000109f33958 -[UIWindow
_sendTouchesForEvent:] + 735
    5   UIKit                               0x0000000109f34282 -[UIWindow sendEvent:] + 682
    6   UIKit                               0x0000000109efa541 -[UIApplication sendEvent:] +
246
    7   UIKit                               0x0000000109f07cdc
_UIApplicationHandleEventFromQueueEvent + 18265
    8   UIKit                               0x0000000109ee259c _UIApplicationHandleEventQueue
+ 2066
    9   CoreFoundation                      0x0000000109a20431
__CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION__ + 17
    10  CoreFoundation                      0x0000000109a162fd __CFRunLoopDoSources0 + 269
    11  CoreFoundation                      0x0000000109a15934 __CFRunLoopRun + 868
    12  CoreFoundation                      0x0000000109a15366 CFRunLoopRunSpecific + 470
    13  GraphicsServices                    0x000000010d012a3e GSEventRunModal + 161
    14  UIKit                               0x0000000109ee58c0 UIApplicationMain + 1282
    15  05 RunLoopTest                      0x0000000109254e7f main + 111
    16  libdyld.dylib                       0x000000010c090145 start + 1
)
```

- GSEventRunModal 在GraphicsServices框架中，GraphicsServices就在我们称之为Core Services的那一层，用来确定用户的点击事件
- 不同于

  ```
    10  CoreFoundation                      0x000000010f50441c
  __CFRUNLOOP_IS_CALLING_OUT_TO_A_BLOCK__ + 12
    11  CoreFoundation                      0x000000010f4fa165 __CFRunLoopDoBlocks + 341
    12  CoreFoundation                      0x000000010f4f9f25 __CFRunLoopRun + 2389
    13  CoreFoundation                      0x000000010f4f9366 CFRunLoopRunSpecific + 470
  ```

  这里变成了

  ```
    9   CoreFoundation                      0x0000000109a20431
  __CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION__ + 17
    10  CoreFoundation                      0x0000000109a162fd __CFRunLoopDoSources0 + 269
    11  CoreFoundation                      0x0000000109a15934 __CFRunLoopRun + 868
    12  CoreFoundation                      0x0000000109a15366 CFRunLoopRunSpecific + 470
  ```

  不同点就是`__CFRunLoopDoBlocks`变成了`__CFRunLoopDoSources0`，`__CFRUNLOOP_IS_CALLING_OUT_TO_A_BLOCK__`变成了`__CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION__`

那么为什么会多处一个GraphicsServices呢，因为我们这次是在一个button的点击事件中打印的调用堆栈。

RunLoop Callouts

存在RunLoop的线程中，几乎所有的函数（方法）都是从以下6个函数之一开始调用的：

- CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION

```
CFRunloop is calling out to an observer callback function
用于向外部报告 RunLoop 当前状态的更改，框架中很多机制都由 RunLoopObserver 触发，如 CAAnimation
```

- CFRUNLOOP_IS_CALLING_OUT_TO_A_BLOCK

```
CFRunloop is calling out to a block
消息通知、非延迟的perform、dispatch调用、block回调、KVO
```

- CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE

```
CFRunloop is servicing the main desipatch queue
```

CFRUNLOOP_IS_CALLING_OUT_TO_A_TIMER_CALLBACK_FUNCTION

```
CFRunloop is calling out to a timer callback function
延迟的perform，延迟dispatch调用
```

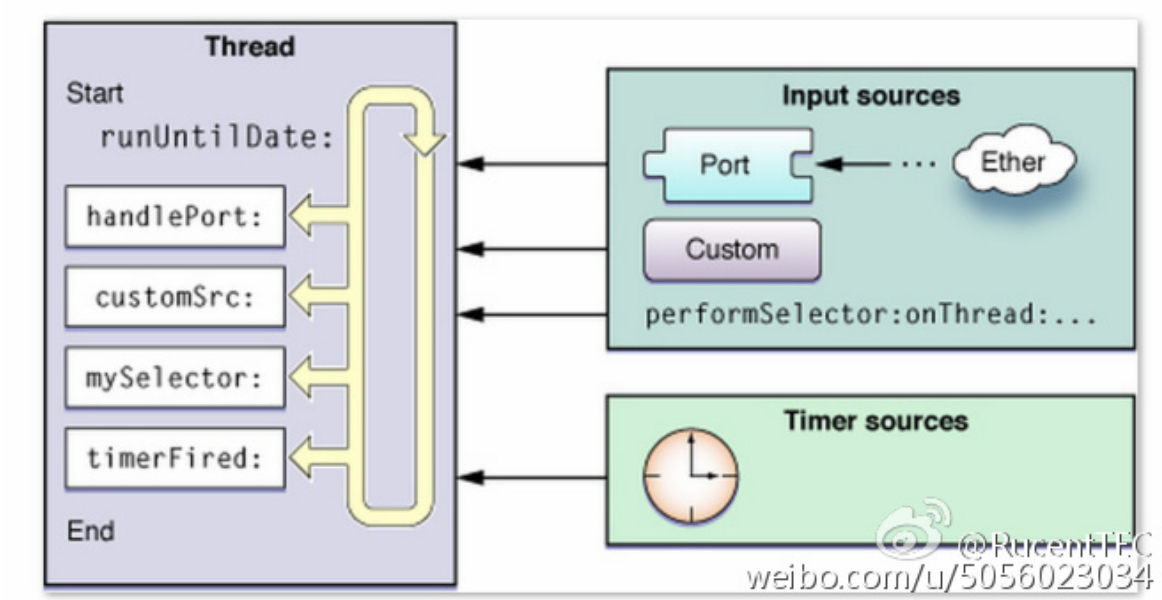
- CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION

```
CFRunloop is calling out to a source 0 perform function
处理App内部事件、App自己负责管理（触发），如UIEvent、CFSocket。普通函数调用，系统调用
```

*CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE1_PERFORM_FUNCTION

```
CFRunloop is calling out to a source 1 perform function
由RunLoop和内核管理，Mach port驱动，如CFMachPort、CFMessagePort
```

如图所示：



支持接收处理输入源（Input Source）事件，包括：

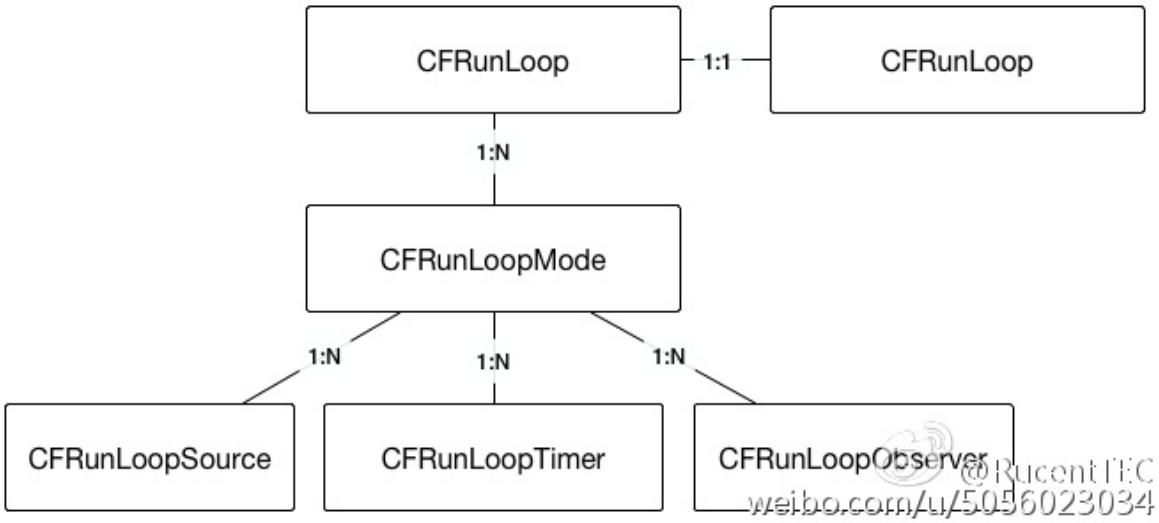
- 系统的Mach Port事件，是一种通讯事件自定义输入事件
- 支持接受处理定时源（Timer）事件

在启动RunLoop之前，必须添加监听的**输入源事件或者定时源事件**，否则调用[runloop run]会直接返回，而不会进入循环让线程长驻。 如果没有添加任何输入源事件或Timer事件，线程会一直在无限循环空转中，会一直占用CPU时间片，没有实现资源的合理分配。没有while循环且没有添加任何输入源或Timer的线程，线程会直接完成，被系统回收。

CFRunLoop的实现

因为NSRunLoop本身是由CFRunLoop实现的，所以我们只讲解CFRunLoop的实现。

了解CFRunLoop要从结构入手， 如图所示：



Run Loop Mode：

理解Run Loop Mode就是流水线上支持生产的产品类型，流水线在一个时刻只能在一种模式下运行，生产某一类型

的产品。消息事件就是订单。Mode定义在不同的框架之中

1. RunLoop在同一时间内，必须且只能设定一种模式
2. 更换Mode时，需要先停止当前Loop，然后重启Loop
3. Mode机制是iOS App滑动非常流畅的关键所在

CocoaTouch定义了三种常用Mode，（Cocoa中要多一些） 1. Default：NSDefaultRunLoopMode（Foundation框架），默认模式，在Run Loop没有指定Mode的时候，默认就跑在Default Mode下 2. Tracking，UITrackingRunLoop 追踪运动事件模式，如动画，拖动等等 3. Common mode：NSRunLoopCommonModes（Foundation框架），是一个模式集合，当绑定一个事件源到这个模式集合的时候就相当于绑定到了集合内的每一个模式 4. 可以自定义更多的Mode，比如UIKit框架中定义了一个私有的UIInitializationRunLoopMode，用于启动时使用

验证Mode的唯一性，我们可以通过NSTimer来测试以下：

```
//默认情况下会使用Default：NSDefaultRunLoopMode，此时ScrollView如果滚动，timer所在的loop会首先切换模式到UITrackingRunLoop模式下，那么以前的timer将会被暂停，直到切换回NSDefaultRunLoopMode才会继续运行
    NSTimer *timer = [NSTimer scheduledTimerWithTimeInterval:.2 target:self selector:@selector(timeMethod) userInfo:nil repeats:YES];

//添加如下代码，把timer的模式设置为    NSRunLoopCommonModes，此时ScrollView滑动会一并加入到NSRunLoopCommonModes模式中，所以timer不会被影响
//    [[NSRunLoop currentRunLoop] addTimer:timer forMode:NSRunLoopCommonModes];
```

CFRunLoopTimer：

看到Timer很直接想到NSTimer，没错，NSTimer就是用CFRunLoopTimer封装的，同样的还有CADisplayLink、还有我们常用的延时调用方法`performSelectorAfterDelay`

CFRunLoopSource：

针对的是输入源方式的Loop，Source是RunLoop的输入员抽象类（protocol），默认的实现有两种： 1. source0: 处理App内部事件、App自己负责管理（触发），如UIEvent、CFSocket。对应的结构体:CFRunLoopSourceContext 2. source1: 由RunLoop和内核管理，Mach port驱动，如CFMachPort、CFMessagePort。对应的结构体:CFRunLoopSourceContext1

CFRunLoopObserver：

用于向外部报告 RunLoop 当前状态的更改。CAAnimation中的CATransition，当Push到新的界面中，并非立马调用，而是收集完成足够的信息后，再更改RunLoop的状态，更改状态后会报告更改的动作，然后动画开始。状态有以下几种：

```
    /* Run Loop Observer Activities */
    typedef CF_OPTIONS(CFOptionFlags, CFRunLoopActivity) {
        kCFRunLoopEntry = (1UL << 0),
        kCFRunLoopBeforeTimers = (1UL << 1),
        kCFRunLoopBeforeSources = (1UL << 2),
        kCFRunLoopBeforeWaiting = (1UL << 5),
        kCFRunLoopAfterWaiting = (1UL << 6),
        kCFRunLoopExit = (1UL << 7),
        kCFRunLoopAllActivities = 0x0FFFFFFFU
    };

*--AutoReleasePool与RunLoopObserver--*

在面试中经常会被问到AutoReleasePool什么时候释放内部AutoRelease对象，其实跟这里的RunLoopObserver有关，当RunLoop从活跃状态变为等待前，AutoReleasePool会释放掉上一次等待结束后到当前之间的AutoRelease对象。
```

CFRunLoop的挂起和唤醒

程序空闲时暂停程序，你将会看到以下调试结果：

解释：

- 序号0 当前Loop正处于waiting状态，等待事件介入。
- 序号2 获取MachPort

挂起和唤醒RunLoop的步骤如下：

- 指定唤醒mach_port端口
- 调用mach_msg监听唤醒端口，被唤醒前，系统内核将这个线程挂起，停留在mach_msg_trap状态，RunLoop被挂起
- 在其他线程（可以是不同进程中的）向内核发送指定端口号的msg时，trap状态被唤醒，RunLoop继续工作

CFRunLoop的执行顺序

模拟代码：

```
bool timeout = false;
bool stop = false;
SetUpRunLoopTimourTimer(&timeout);//GCD timer
do {


//        CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION    =
__CFRunLoopObservers

    __CFRunLoopObservers(kCFRunLoopBeforeTimers); //调用__CFRunLoopObservers函数，通知更改状态
    __CFRunLoopObservers(kCFRunLoopBeforeSources);

//        CFRUNLOOP_IS_CALLING_OUT_TO_A_BLOCK    =    __CFRunLoopDoBlocks
    __CFRunLoopDoBlocks();
    __CFRunLoopDoSource0();
    __CheckIfExistMsgsInMainDispatchQueue(); // GCD
    __CFRunLoopObservers(kCFRunLoopBeforeWaiting);

    CFPort wakUpPort = __CFServiceRunLoopMachPort(); //设置Port   .   mach_msg_trap...休眠
中..mach_msg收到消息

//        收到消息，更改状态
    __CFRunLoopObservers(kCFRunLoopAfterWaiting);
//        甄别消息类型，向外callout
    if (wakUpPort == _runLoopTimerMachPort) {

        CFRUNLOOP_IS_CALLING_OUT_TO_A_TIMER_CALLBACK_FUNCTION();

    }else if(wakUpPort == _runLoopDispatchQueueMachPort){

        __CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE();

    }else{
//            CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE1_PERFORM_FUNCTION    =
__CFRunLoopDoSource1
        __CFRunLoopDoSource1();
    }



} while (!timeout&&!stop);
```

Run Loop的使用

- 如何创建一个常驻服务线程

  AFNetworking的网络线程实现：

```
+ (void)networkRequestThreadEntryPoint:(id)__unused object {
    @autoreleasepool {
        [[NSThread currentThread] setName:@"AFNetworking"];

        NSRunLoop *runLoop = [NSRunLoop currentRunLoop];
        [runLoop addPort:[NSMachPort port] forMode:NSDefaultRunLoopMode];
        [runLoop run];
    }
}

+ (NSThread *)networkRequestThread {
    static NSThread *_networkRequestThread = nil;
    static dispatch_once_t oncePredicate;
    dispatch_once(&oncePredicate, ^{
        _networkRequestThread = [[NSThread alloc] initWithTarget:self
selector:@selector(networkRequestThreadEntryPoint:) object:nil];
        [_networkRequestThread start];
    });

    return _networkRequestThread;
}
```

- 解决tableView延迟加载图片

  tableView的cell中如果有ImageView，那么在滚动时就更新图片时有可能造成卡顿现象的，所以我们可以在tableView滚动停止后设置图片内容：

```
//    CustomCell.m
UIImage *image= ...;
[self.headerImageView performSelector:@selector(setImage:) withObject:image
afterDelay:0 inModes:@[NSDefaultRunLoopMode]];
```

- App崩溃后立即重启

```
CFRunLoopRef runLoop = CFRunLoopGetCurrent();
NSArray *allModes = CFBridgingRelease(CFRunLoopCopyAllModes(runLoop));
while (true) {
    for (NSString *mode in allModes) {

        CFRunLoopRunInMode((CFStringRef)mode, 1/MAXFLOAT, false);
    }
}
```

小节总结： 理解RunLoop内容，需要有更加丰富的多线程编程思想，但是理解NSRunLoop的基本概念会对我们学习下面的多线程编程工具类打下坚实的基础。