**1. RunLoop 简介**

**1.1 什么是 RunLoop？**

* RunLoop 实际上是一个对象，这个对象在循环中用来处理程序运行过程中出现的各种事件（比如说触摸事件、UI刷新事件、定时器事件、Selector事件），从而保持程序的持续运行。
* RunLoop 在没有事件处理的时候，会使线程进入睡眠模式，从而节省 CPU 资源，提高程序性能。

**1.2 RunLoop 和线程**

RunLoop 和线程是息息相关的，我们知道线程的作用是用来执行特定的一个或多个任务，在默认情况下，线程执行完之后就会退出，就不能再执行任务了。这时我们就需要采用一种方式来让线程能够不断地处理任务，并不退出。所以，我们就有了 RunLoop。

* 一条线程对应一个RunLoop对象，每条线程都有唯一一个与之对应的 RunLoop 对象。
* RunLoop 并不保证线程安全。我们只能在当前线程内部操作当前线程的 RunLoop 对象，而不能在当前线程内部去操作其他线程的 RunLoop 对象方法。
* RunLoop 对象在第一次获取 RunLoop 时创建，销毁则是在线程结束的时候。
* 主线程的 RunLoop 对象系统自动帮助我们创建好了，而子线程的 RunLoop对象需要我们主动创建和维护。

主线程开启 RunLoop 的过程可以简单的理解为如下代码：

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

BOOL running = YES;

do {

// 执行各种任务，处理各种事件

// ......

} while (running); // 判断是否需要退出

return 0;

}

RunLoop 就是线程中的一个循环，RunLoop 会在循环中会不断检测，通过 Input sources（输入源）和 Timer sources（定时源）两种来源等待接受事件；然后对接受到的事件通知线程进行处理，并在没有事件的时候让线程进行休息。

**2. RunLoop 相关类**

下面我们来了解一下Core Foundation框架下关于 RunLoop 的 5 个类，只有弄懂这几个类的含义，我们才能深入了解 RunLoop 的运行机制。

1、CFRunLoopRef：代表 RunLoop 的对象  
2、CFRunLoopModeRef：代表 RunLoop 的运行模式  
3、CFRunLoopSourceRef：就是 RunLoop 模型图中提到的输入源 / 事件源  
4、CFRunLoopTimerRef：就是 RunLoop 模型图中提到的定时源  
5、CFRunLoopObserverRef：观察者，能够监听 RunLoop 的状态改变

这 5 个类的相互关系：  
一个RunLoop对象（CFRunLoopRef）中包含若干个运行模式（CFRunLoopModeRef）。而每一个运行模式下又包含若干个输入源（CFRunLoopSourceRef）、定时源（CFRunLoopTimerRef）、观察者（CFRunLoopObserverRef）。

* 每次 RunLoop 启动时，只能指定其中一个运行模式（CFRunLoopModeRef），这个运行模式（CFRunLoopModeRef）被称作当前运行模式（CurrentMode）。
* 如果需要切换运行模式（CFRunLoopModeRef），只能退出当前 Loop，再重新指定一个运行模式（CFRunLoopModeRef）进入。
* 这样做主要是为了分隔开不同组的输入源（CFRunLoopSourceRef）、定时源（CFRunLoopTimerRef）、观察者（CFRunLoopObserverRef），让其互不影响 。

**2.1 CFRunLoopRef 类**

CFRunLoopRef 是 Core Foundation 框架下 RunLoop 对象类。我们可通过以下方式来获取 RunLoop 对象：  
 1、Core Foundation

* CFRunLoopGetCurrent(); // 获得当前线程的 RunLoop 对象
* CFRunLoopGetMain(); // 获得主线程的 RunLoop 对象

当然，在Foundation 框架下获取 RunLoop 对象类的方法如下：  
 2、Foundation

* [NSRunLoop currentRunLoop]; // 获得当前线程的 RunLoop 对象
* [NSRunLoop mainRunLoop]; // 获得主线程的 RunLoop 对象

**2.2 CFRunLoopModeRef**

系统默认定义了多种运行模式（CFRunLoopModeRef），如下：

* kCFRunLoopDefaultMode：App的默认运行模式，通常主线程是在这个运行模式下运行
* UITrackingRunLoopMode：跟踪用户交互事件（用于 ScrollView 追踪触摸滑动，保证界面滑动时不受其他Mode影响）
* UIInitializationRunLoopMode：在刚启动App时第进入的第一个 Mode，启动完成后就不再使用
* GSEventReceiveRunLoopMode：接受系统内部事件，通常用不到
* kCFRunLoopCommonModes：伪模式，不是一种真正的运行模式

其中kCFRunLoopDefaultMode、UITrackingRunLoopMode、kCFRunLoopCommonModes是我们开发中需要用到的模式

**2.3 CFRunLoopSourceRef**

CFRunLoopSourceRef是事件源，它有两种分类方法。

1、第一种按照官方文档来分类（就像RunLoop模型图中那样）：

* Port-Based Sources（基于端口）
* Custom Input Sources（自定义）
* Cocoa Perform Selector Sources

2、第二种按照函数调用栈来分类：

* Source0 ：非基于Port
* Source1：基于Port，通过内核和其他线程通信，接收、分发系统事件

这两种分类方式其实没有区别，只不过第一种是通过官方理论来分类，第二种是在实际应用中通过调用函数来分类。

**2.4 CFRunLoopTimerRef**

CFRunLoopTimerRef是定时源，理解为基于时间的触发器，基本上就是NSTimer。

- (void)viewDidLoad {

[super viewDidLoad];

// 定义一个定时器，约定两秒之后调用self的run方法

NSTimer \*timer = [NSTimer timerWithTimeInterval:2.0 target:self selector:@selector(run) userInfo:nil repeats:YES];

// 将定时器添加到当前RunLoop的NSDefaultRunLoopMode下

[[NSRunLoop currentRunLoop] addTimer:timer forMode:NSDefaultRunLoopMode];

}

- (void)run

{

NSLog(@"---run");

}

等价于：

[NSTimer scheduledTimerWithTimeInterval:2.0 target:self selector:@selector(run) userInfo:nil repeats:YES];

**2.5 CFRunLoopObserverRef**

CFRunLoopObserverRef是观察者，用来监听RunLoop的状态改变  
CFRunLoopObserverRef可以监听的状态改变有以下几种：

typedef CF\_OPTIONS(CFOptionFlags, CFRunLoopActivity) {

kCFRunLoopEntry = (1UL << 0), // 即将进入Loop：1

kCFRunLoopBeforeTimers = (1UL << 1), // 即将处理Timer：2

kCFRunLoopBeforeSources = (1UL << 2), // 即将处理Source：4

kCFRunLoopBeforeWaiting = (1UL << 5), // 即将进入休眠：32

kCFRunLoopAfterWaiting = (1UL << 6), // 即将从休眠中唤醒：64

kCFRunLoopExit = (1UL << 7), // 即将从Loop中退出：128

kCFRunLoopAllActivities = 0x0FFFFFFFU // 监听全部状态改变

};

下边我们通过代码来监听下RunLoop中的状态改变。

- (void)viewDidLoad {

[super viewDidLoad];

// 创建观察者

CFRunLoopObserverRef observer = CFRunLoopObserverCreateWithHandler(CFAllocatorGetDefault(), kCFRunLoopAllActivities, YES, 0, ^(CFRunLoopObserverRef observer, CFRunLoopActivity activity) {

NSLog(@"监听到RunLoop发生改变---%zd",activity);

});

// 添加观察者到当前RunLoop中

CFRunLoopAddObserver(CFRunLoopGetCurrent(), observer, kCFRunLoopDefaultMode);

// 释放observer，最后添加完需要释放掉

CFRelease(observer);

}

**3. RunLoop原理**

RunLoop运行逻辑图.png

在每次运行开启RunLoop的时候，所在线程的RunLoop会自动处理之前未处理的事件，并且通知相关的观察者。

具体的顺序如下：

1、通知观察者RunLoop已经启动  
2、通知观察者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状态  
7、将线程置于休眠知道任一下面的事件发生：  
   1.某一事件到达基于端口的源  
   2.定时器启动  
   3.RunLoop设置的时间已经超时  
   4.RunLoop被显示唤醒  
8、通知观察者线程将被唤醒  
9、处理未处理的事件  
   1.如果用户定义的定时器启动，处理定时器事件并重启RunLoop。进入步骤2  
   2.如果输入源启动，传递相应的消息  
   3.如果RunLoop被显示唤醒而且时间还没超时，重启RunLoop。进入步骤2  
10、通知观察者RunLoop结束。

**4. RunLoop实战应用**

1、 NSTimer的使用  
2、 ImageView推迟显示  
有时候，我们会遇到这种情况：  
当界面中含有UITableView，而且每个UITableViewCell里边都有图片。这时候当我们滚动UITableView的时候，如果有一堆的图片需要显示，那么可能会出现卡顿的现象。

* 1. 监听UIScrollView的滚动  
     因为UITableView继承自UIScrollView，所以我们可以通过监听UIScrollView的滚动，实现UIScrollView相关delegate即可。
  2. 利用PerformSelector设置当前线程的RunLoop的运行模式

3 、后台常驻线程（很常用）  
我们在开发应用程序的过程中，如果后台操作特别频繁，经常会在子线程做一些耗时操作（下载文件、后台播放音乐等），我们最好能让这条线程永远常驻内存。

- (void)viewDidLoad {

[super viewDidLoad];

// 创建线程，并调用run1方法执行任务

self.thread = [[NSThread alloc] initWithTarget:self selector:@selector(run1) object:nil];

// 开启线程

[self.thread start];

}

- (void) run1

{

// 这里写任务

NSLog(@"----run1-----");

// 添加下边两句代码，就可以开启RunLoop，之后self.thread就变成了常驻线程，可随时添加任务，并交于RunLoop处理

[[NSRunLoop currentRunLoop] addPort:[NSPort port] forMode:NSDefaultRunLoopMode];

[[NSRunLoop currentRunLoop] run];

// 测试是否开启了RunLoop，如果开启RunLoop，则来不了这里，因为RunLoop开启了循环。

NSLog(@"未开启RunLoop");

}

运行之后发现打印了----run1-----，而未开启RunLoop则未打印。  
这时，我们就开启了一条常驻线程，下边我们来试着添加其他任务，除了之前创建的时候调用了run1方法，我们另外在点击的时候调用run2方法。

那么，我们在touchesBegan中调用PerformSelector，从而实现在点击屏幕的时候调用run2方法。

- (void)touchesBegan:(NSSet<UITouch \*> \*)touches withEvent:(UIEvent \*)event

{

// 利用performSelector，在self.thread的线程中调用run2方法执行任务

[self performSelector:@selector(run2) onThread:self.thread withObject:nil waitUntilDone:NO];

}

- (void) run2

{

NSLog(@"----run2------");

}