解密 Runloop

按理说，这都 8102 年了，iOS 面试已经发展到手写 weak 关键字实现的今天，我原本不该再写 Runloop 这种土味题材的，还取了这么个「走近科学」栏目风格的标题。只是赶巧最近工作中又有涉及到，感觉自己有些新体会，趁着还热乎就写下来，希望能帮助一些读者在对 Runloop 的认识上，再深入浅出一些。

在开始之前，必须强调下，Runloop 是开源的，且关键代码其实非常少，建议大家直接通读一遍，这篇文章可以作为阅读的辅助材料，降低阅读难度。

Runloop 源码 Objective-C 版本：
https://opensource.apple.com/source/CF/CF-635.19/CFRunLoop.c.auto.html

Runloop 源码最新的 Swift 版本：https://github.com/apple/swift-corelibs-foundation/blob/master/CoreFoundation/RunLoop.subproj/CFRunLoop.c

预热

先介绍一个基础且重要的知识点，有助于对于 Runloop 的理解。

mach_msg

这是系统内核在某个 port 收发消息所使用的函数，理解这个函数对于理解 runloop 的运行机制非常重要。详细的说明可参考（http://web.mit.edu/darwin/src/modules/xnu/osfmk/man/mach_msg.html）

值得注意的是，收消息与发消息都是调用这个函数，只是参数不同，发送为 MACH_SEND_MSG，接收为 MACH_RCV_MSG。

可以简单的将 mach_msg 理解为多进程之间的一种通信机制，不同的进程可以使用同一个消息队列来交流数据，当使用 mach_msg 从消息队列里读取 msg 时，可以在参数中 timeout 值，在 timeout 之前如果没有读到 msg，当前线程会一直处于休眠状态。这也是 runloop 在没有任务可执行的时候，能够进入 sleep 状态的原因。

Runloop 流程解析

所谓 Runloop，简而言之，是 Apple 所设计的，一种在当前线程，持续调度各种任务的运行机制。说起来有些绕口，我们翻译成代码就非常直白了。

```
while (alive) {
  performTask() //执行任务
  callout_to_observer() //通知外部
  sleep() //休眠
}
```

每一次 loop 执行，主要做三件事：

- performTask()
- callout_to_observer()
- sleep()

再来依次看下上面三步的花式展开。

performTask

每一次 runloop 的运行都会执行若干个 task，执行 task 的方式有多种，有些方式可以被开发者使用，有些则只能被系统使用。逐一看下：

DoBlocks()

这种方式可以被开发者使用，使用方式很简单。可以先通过 CFRunLoopPerformBlock 将一个 block 插入目标队列，函数签名如下：

```
void CFRunLoopPerformBlock(CFRunLoopRef rl, CFTypeRef mode, void (^
```

详细使用方式可参考文档：https://developer.apple.com/documentation/corefoundation/1542985-cfrunloopperformblock?language=objc

可以看出该 block 插入队列的时候，是绑定到某个 runloop mode 的，runloop mode 的概念后面会详细解释，也是理解 runloop 运行机制的关键。

调用上面的 api 之后，runloop 在执行的时候，会通过如下 API 执行队列里所有的 block：

```
__CFRunLoopDoBlocks(rl, rlm);
```

很显然，执行的时候也是只执行和某个 mode 相关的所有 block。至于执行的时机点有多处，后面也会标注。

DoSources0()

Runloop 里有两种 source，source0 和 source1，虽然名称相似，二者运行机理并不相同。source0 有公开的 API 可供开发者调用，source1 却只能供系统使用，而且 source1 的实现原理是基于 mach_msg 函数，通过读取某个 port 上内核消息队列上的消息来决定执行的任务。

作为开发者要使用 source0 也很简单，先创建一个 CFRunLoopSourceContext，context 里需要传入被执行任务的函数指针作为参数，再将该 context 作为构造参数传入 CFRunLoopSourceCreate 创建一个 source，之后通过 CFRunLoopAddSource 将该 source 绑定的某个 runloop mode 即可。

详细文档可参考：
https://developer.apple.com/documentation/corefoundation/1542679-cfrunloopsourcecreate?language=objc

绑定好之后，runloop 在执行的时候，会通过如下 API 执行所有的 source0:

```
__CFRunLoopDoSources0(rl, rlm, stopAfterHandle);
```

同理，每次执行的时候，也只会运行和当前 mode 相关的 source0。

DoSources1()

如上所述，source1 并不对开发者开放，系统会使用它来执行一些内部任务，比如渲染 UI。

公司内部有个厉害的工具，可以将某个线程一段时间内所执行的函数全部 dump 下来，上传到后台并以流程图的形式展示，很直观。得益于这个工具，我可以清楚的看到 DoBlocks，DoSources0， DoSources1 被使用时的 call stack，也就能知道系统是处于什么目的在使用上述三种任务调用机制，后面解释。

DoTimers()

这个比较简单，开发者使用 NSTimer 相关 API 即可注册被执行的任务，runloop 通过如下 API 执行相关任务：

```
__CFRunLoopDoTimers(rl, rlm, mach_absolute_time());
```

同理，每次执行的时候，也只会运行和当前 mode 相关的 timer。

DoMainQueue()

这个也再简单不过，开发者调用 GCD 的 API 将任务放入到 main queue 中，runloop 则通过如下 API 执行被调度的任务：

```
_dispatch_main_queue_callback_4CF(msg);
```

注意，这里就没有 rlm 参数了，也就是说 DoMainQueue 和 runloop mode 是不相关的。msg 是通过 mach_msg 函数从某个 port 上读出的 msg。

问题来了

综上所述，在 runloop 里一共有 5 种方式来执行任务，那么问题来了，苹果为什么要搞这么多花样，他们各自的使用场景是什么?

timer 和 mainqueue 无需多说，开发者大多熟悉其背后设计宗旨。至于 DoBlocks，DoSources0，和 DoSources1，我原先以为系统在使用时，他们各有分工，比如某些用来接收硬件事件，有些则负责渲染 Core Animation 任务，但实际观摩过一些主线程运行样本之后，我发现并无类似的 pattern。

比如我在 doSource0 里看到了这个 callstack：

```
...
__CFRunLoopDoSources0
...
[UIApplication sendEvent:]
...
```

显然是系统用 source0 任务来接收硬件事件。

又比如这个使用 mainqueue 的 callstack：

```
...
_dispatch_main_queue_callback_4CF
...
[UIView(Hierarchy) _makeSubtreePerformSelector:withObject:withObjec
...
```

系统在使用 doMainQueue 来执行 UIView 的布局任务。

再比如这个 callstack：

```
...
__CFRunLoopDoTimer
...
[UICollectionView _updateWithItems:tentativelyForReordering:animatc
...
```

这是系统在使用 doTimers 来 UI 绘制任务。

再看这个：

```
...
__CFRunLoopDoBlocks
...
CA::Context::commit_transaction(CA::Transaction*)
...
```

这是系统在使用 doBlocks 来提交 Core Animation 的绘制任务。

继续看这个：

```
...
__CFRunLoopDoSources0
...
CA::Transaction::commit()
...
```

这是系统在使用 doSource0 来提交 Core Animation 的绘制任务。

不知道大家看出什么 pattern 没，我没，唯一比较有规律的是硬件事件都是通过 doSource0 来传递的，总体感觉系统在使用的时候有点 free style。

callout_to_observer

这一分类主要是 runloop 用来通知外部 observer 用的，用来告知外部某个任务已被执行，或者是 runloop 当前处于什么状态。我们也来逐一看下：

DoObservers-Timer

故名思义，在 DoTimers 执行完毕之后，调用 DoObservers-Timer 来告知感兴趣的 observer，怎么注册 observer 呢？在介绍完各种 callback 机制之后，再统一说下。runloop 是通过如下函数来通知 observer：

```
__CFRunLoopDoObservers(rl, rlm, kCFRunLoopBeforeTimers);
```

DoObservers-Source0

同理，是在执行完 source0 之后，调用 DoObservers-Source0 来告知感兴趣的 observer，怎么注册后面统一介绍。runloop 通过如下函数来通知 observer：

```
__CFRunLoopDoObservers(rl, rlm, kCFRunLoopBeforeSources);
```

这是上述五种执行任务方式中，两种可以注册 observer 的，其他几个都不支持，mainQueue，source1，block 都不行。所以理论上，是没有办法准确测量各个任务执行的时长的。

DoObservers-Activity

这是 runloop 用来通知外部自己当前状态用的，当前 runloop 正执行到哪个 activity，那么一共有几种 activity 呢？看源码一清二楚：

```
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
};
```

啰嗦下，再一个个讲解：

kCFRunLoopEntry

每次 runloop 重新进入时的 activity，runloop 每一次进入一个 mode，就通知一次外部 kCFRunLoopEntry，之后会一直以该 mode 运行，知道当前 mode 被终止，进而切换到其他 mode，并再次通知 kCFRunLoopEntry。runloop mode 的切换也是个很有意思的话题，后面会提到。

kCFRunLoopBeforeTimers

这就是上面提到的 DoObservers-Timer，Apple 应该是为了代码的整洁，将 kCFRunLoopBeforeTimers 也归为了一种 activity，其含义上面已经介绍，不再赘述。

kCFRunLoopBeforeSources

同理，Apple 也将该 callout 归为了一种 runloop 的 activity。

kCFRunLoopBeforeWaiting

这个 activity 表示当前线程即将可能进入睡眠，如果能够从内核队列上读出 msg 则继续运行任务，如果当前队列上没多余消息，则进入睡眠状态。读取 msg 的函数为：

```
__CFRunLoopServiceMachPort(waitSet, &msg, sizeof(msg_buffer), poll
```

其本质是调用了开篇所说的 mach_msg 内核函数，注意 timeout 值，TIMEOUT_INFINITY 表示有可能无限进入睡眠状态。

kCFRunLoopAfterWaiting

这个 activity 是当前线程从睡眠状态中恢复过来，也就是说上面的 mach_msg 终于从队列里读出了 msg，可以继续执行任务了。这是每一次 runloop 从 idle 状态中恢复必调的一个 activity，如果你想设计一个工具检测 runloop 的执行周期，那么这个 activity 就可以作为周期的开始。

kCFRunLoopExit

exit 不必多言，切换 mode 的时候可能会调用到这个 activity，为什么说可能呢？这和 mode 切换的方式有关，后面会提及。

activity 的 回调并不是单单给开发者用的，事实上，系统也会通过注册相关 activity 的回调来完成一些任务，比如我看到过如下的 callstack：

```
...
__CFRunLoopDoObservers
...
[UIView(Hierarchy) addSubview:]
...
```

显然系统在观测到 runloop 进入某个 activity 之后，会进行一些 UIView 的布局工作。

再看这个：

```
...
__CFRunLoopDoObservers
...
[UIViewController __viewWillDisappear:]
...
```

这是系统在使用 DoObservers 传递 viewWillDisappear 回调。

以上即为 observer 的全部内容，一般开发者对 runloop 的 activity 感兴趣，多半是想分析主线程的业务代码执行情况，事实上，这些 activity 的回调不怎么可靠，也就是说有可能 runloop 哼哧运行来半天的代码，你一个 activity 的回调也收不到，或者收到了，但顺序也是完全出乎你的意料，后面会详细解释。

sleep

一言以蔽之，有任务就执行，没任务就 sleep。这部分逻辑就这么简单。

只是有个小细节需要注意，一般人印象里感觉 runloop 的每次 loop 总是按顺序执行上面的各种 performTask 和 callout_to_observer，执行完就 sleep，而实际上，这些任务的执行相互糅合在一起，还有 goto 的跳转逻辑，显得非常凌乱，而且 activity 的 callback 也可能不是按照 kCFRunLoopEntry->kCFRunLoopBeforeWaiting->kCFRunLoopAfterWaiting->kCFRunLoopExit 来的，后面我会画个流程图来解释下。

Runloop 的 loop 主函数为 __CFRunLoopRun，里面的这行调用会决定是否 sleep：

```
__CFRunLoopServiceMachPort(waitSet, &msg, sizeof(msg_buffer), poll
```

其内部无非是使用了我们开篇所提到的 mach_msg 函数。

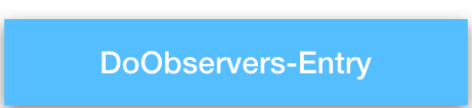
runloop sleep 的代码之处，作者还调皮的留了一句 hamlet 中的台词：

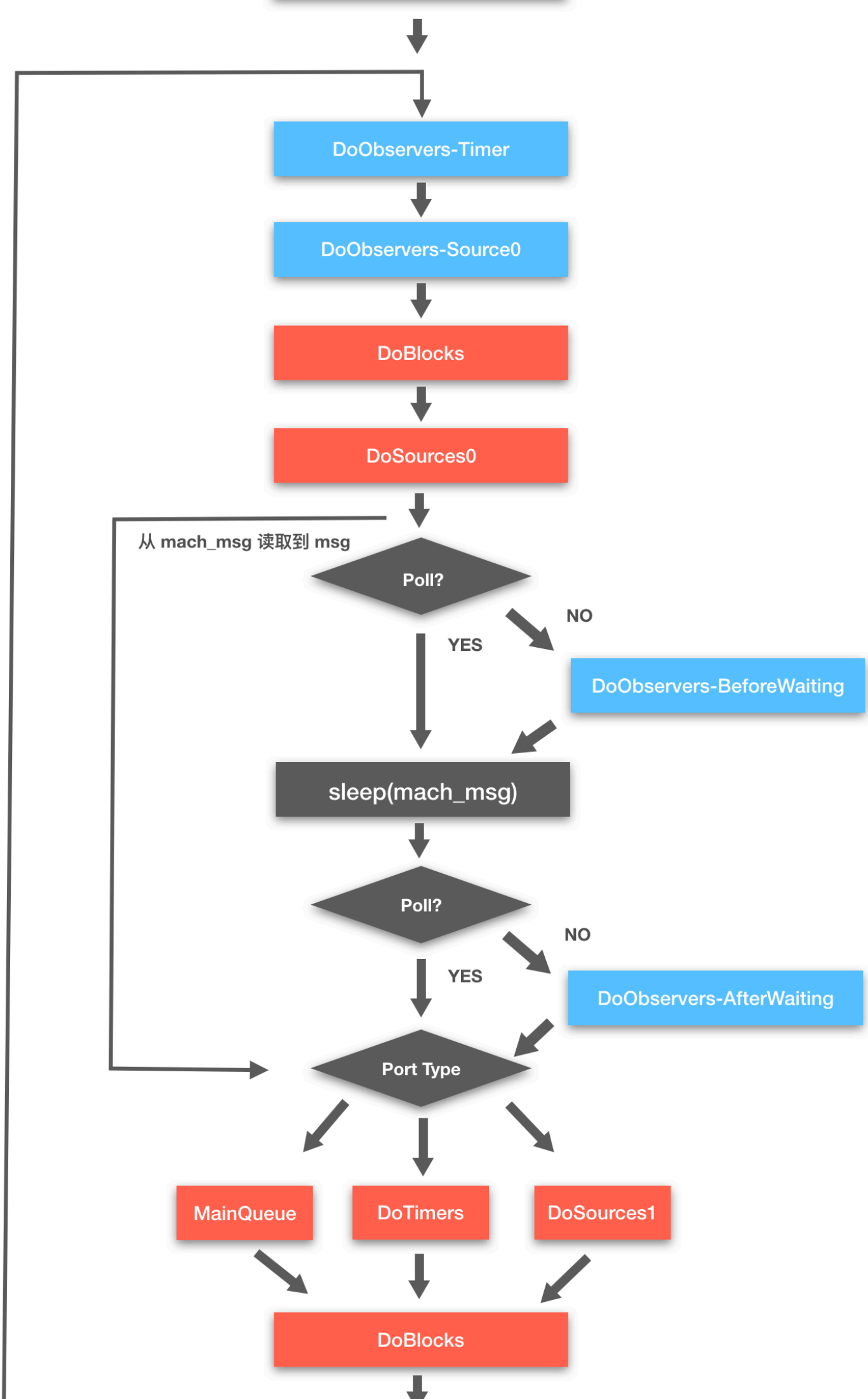
> *In that sleep of death what nightmares may come*

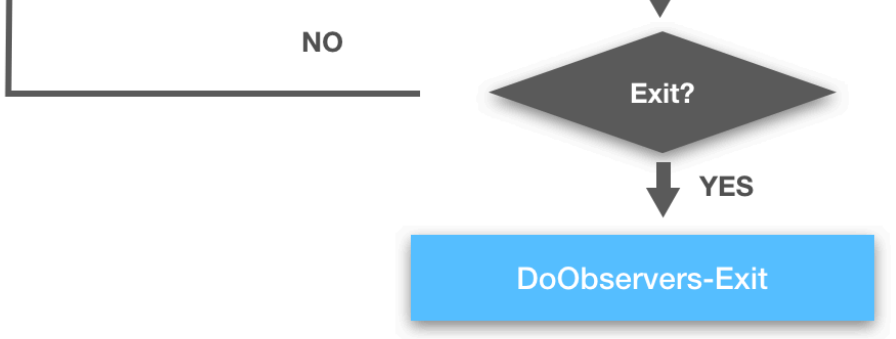
完整流程

至此，我们已将 runloop 中的关键代码分为了三类，并就这三类进行了展开，接下来我们看下完整的流程。

Apple 工程师提到 runloop 的实现可能会随着 iOS 版本而变化，我在对比 Objective C 和 Swift 版本代码之后，发现关键流程没多少区别，下面这张图是我阅读代码时顺手绘制的，希望能让读者对 runloop 的运行机制有更直观形象的认识：




DoObservers-Timer
DoObservers-Source0
DoBlocks
DoSources0
从 mach_msg 读取到 msg
Poll?
NO
YES
DoObservers-BeforeWaiting
sleep(mach_msg)
Poll?
NO
YES
DoObservers-AfterWaiting
Port Type
MainQueue
DoTimers
DoSources1
DoBlocks



我将 performTask 和 callout_to_observer 用不同的颜色加以了区分，从图中可以直观的看到 5 种 performTask 和 6 种 callout_to_observer 是在一次 loop 里如何分布的。

有些细节难以在图中体现，再单独拿出来解释下。

Poll?

每次 loop 如果处理了 source0 任务，那么 poll 值会为 true，直接的影响是不会 DoObservers-BeforeWaiting 和 DoObservers-AfterWaiting，也就是说 runloop 会直接进入睡眠，而且不会告知 BeforeWaiting 和 AfterWaiting 这两个 activity。所以你看，有些情况下，可能 runloop 经过了几个 loop，但你注册的 observer 却不会收到 callback。

两次 mach_msg

其实一次 loop 里有两次调用 mach_msg，有一次我没有标记出来，是发生在 DoSource0 之后，会主动去读取和 mainQueue 相关的 msg 队列，这不过这个 mach_msg 调用是不会进入睡眠的，因为 timeout 值传入的是 0，如果读到了消息，就直接 goto 到 DoMainQueue 的代码，这种设计应该是为了保障 dispatch 到 main queue 的代码总是有较高的机会得以运行。

Port Type

每次 runloop 被唤醒之后，会根据 port type 而决定到底执行哪一类任务，DoMainQueue，DoTimers，DoSource1 三者只会运行一个，剩下的会留到下一次 loop 里去执行。

Runloop Mode

接下来是关键里的重点，重点里的核心，关于 runloop mode 的理解。

开始之前，再回顾下 runloop 在一次 loop 里可能会做的事情，代码如下：

```
while (alive) {
  //执行任务
  DoBlocks();
  DoSources0();
  DoSources1();
  DoTimers();
  DoMainQueue();

  //通知外部
  DoObservers-Timer();
  DoObservers-Source0();
  DoObservers-Activity();

  //休眠
  sleep()
}
```

Runloop mode 的设计就是为了执行上述的逻辑服务，我反复提到过，大部分的任务和回调是和 mode 绑定的，那么我们来看下 mode 的数据结构是如何体现这部分功能的：

```
struct __CFRunLoopMode {
    ...
    CFStringRef _name;
    CFMutableSetRef _sources0;
    CFMutableSetRef _sources1;
    CFMutableArrayRef _observers;
    CFMutableArrayRef _timers;
    CFMutableDictionaryRef _portToV1SourceMap;
    CFIndex _observerMask;
        ...
};
```

为了阅读方便，我略去了一些不太相关的细节。很容易看出，执行任务和通知外部所需要的信息全都定义在了 mode 的数据结构里，基本上都是一个 array 来持有相关引用，比如当前 loop 需要 DoTimers() 的时候，只需要将 _timers 遍历并 invoke 即可：

```
static Boolean __CFRunLoopDoTimers(CFRunLoopRef rl, CFRunLoopModeRe
    for (CFIndex idx = 0, cnt = rlm->_timers ? CFArrayGetCount(rlm-
                ...
    }
    return timerHandled;
}
```

_observerMask 包含所有 observer 感兴趣的 activity，每次 observer 通过如下 API 创建一个新的 activity callback 并注册的时候, mask 也会随之更新：

```
CFRunLoopObserverRef CFRunLoopObserverCreateWithHandler(CFAllocator
```

而 mainQueue 任务的执行和 mode 无关，所以 mode 的结构定义里并无 mainQueue 相关的信息。

其他都比较直白，无须多言。

mode 的种类

关于 mode 的种类以及其背后设计思想，没有太多的文档可以参考，但这部分信息却至关重要。

简单来说 mode 分为两类，common mode 和 private mode。

比如我们所熟知的 kCFRunLoopDefaultMode 和 UITrackingRunLoopMode 是属于 common mode，kCFRunLoopDefaultMode 是默认模式下 runloop 使用的 mode，scrollView 滑动的时候切换到 UITrackingRunLoopMode。

除此之外，系统还定义了其他一些 private mode，比如 UIInitializationRunLoopMode 和 GSEventReceiveRunLoopMode。如果你在 app 启动的时候通过 CFRunLoopCopyAllModes 打印出所有的 runloop mode，就可以看到这两个 mode 了。我们简单探讨下 GSEventReceiveRunLoopMode 的使用场景。

GSEventReceiveRunLoopMode 以 GS 开头，是属于 GraphicsServices 这个并不公开的 framework，GSEvent 封装了系统传递给 app 的重要事件，比如音量按钮事件，屏幕点击事件等，而我们所熟知的 UIEvent 也不过是 GSEvent 的封装。我曾一度怀疑 apple 会使用 GSEventReceiveRunLoopMode 来传递各类系统事件，可惜的是，我在线上代码里设置里一段捕捉逻辑，上报所有未知的 runloop mode，却并没有捕获到 GSEventReceiveRunLoopMode 的使用场景。之后出于好奇，使用了一次召唤神龙的机会，给 Apple 工程师提了个 TSL，接我单的小哥只是隐晦的承认了 GSEventReceiveRunLoopMode 的存在，并表示这事不能说太细，Apple 的确会在一些场景下基于需要使用一些 private mode，事实上，开发者自己也可以创建 private mode 来实现一些功能，比如这个 post 里的例子：https://forums.developer.apple.com/message/187122#187122。除此之外，我并没有得到其他什么有用的信息，有点想退货。

这篇文档列举了一些公开的 mode：http://iphonedevwiki.net/index.php/CFRunLoop。

我设置的捕捉代码也捕获到了另一些有意思的 mode，比如这个 _kCFStreamBlockingOpenMode，google 一下，这是 CFStream 里用来调度网络任务所使用的 private mode，源码 https://opensource.apple.com/source/CF/CF-476.19/CFStream.c.auto.html。

问题来了

runloop mode 分为 common 和 private 对我们日常生活有哪些影响呢？影响很大。

当我们对 runloop 的 activity 感兴趣，并通过如下 API 注册 observer 的时候

```
CF_EXPORT void CFRunLoopAddObserver(CFRunLoopRef rl, CFRunLoopObser
```

大多数时候我们都会传入 kCFRunLoopCommonModes 作为参数，这也就意味着你的 observer 只会在 common mode 被运行的时候 call back，如果当前 loop 是以 private mode 运行的，那么你的 observer 将对 runloop 当前的 activity 浑然不觉。如果你的代码强依赖于 runloop activity 的监测，这显然会成为一个关键缺陷。private mode 使用的场景之多可能超过你的想象。

简而言之，每次 loop 只会以一种 mode 运行，以该 mode 运行的时候，就只执行和该 mode 相关的任务，只通知该 mode 注册过的 observer。

runloop mode 是如何切换的呢?

这个问题涉及到 runloop 的 mode 到底是如何使用的，显然我们无法得知系统是如何使用的，就如同那些 Apple 讳莫如深的 private mode。好在我们还是可以从代码得出分析。

每次如果要切换 mode，为了保证多线程安全，必会先通过如下代码 lock：

```
__CFRunLoopLock(rl);
__CFRunLoopModeLock(rlm);
```

切换完之后再 unlock。

而整个runloop 关键流程函数里，主要有三处 unlock 的调用。

一处是在 sleep 之前，runloop 可能一觉醒来，发现 mode 已经物是人非。

另一处是在 doMainQueue 之前，执行完 GCD main queue 中的任务后，mode 也能会发生变化。

最后一处是在 CFRunLoopRunSpecific 函数，也就是 runloop exit 之后。

所以我们可以得出结论，runloop 有两种切换 mode 的方式，一是在 loop 的中途切换，二是按顺序在当前 mode 结束之后切换。

如果你也对 mode 的使用比较感兴趣，真相都在下面这三个可供开发者使用的函数里：

```
CF_EXPORT CFRunLoopMode CFRunLoopCopyCurrentMode(CFRunLoopRef rl);
CF_EXPORT CFArrayRef CFRunLoopCopyAllModes(CFRunLoopRef rl);
CF_EXPORT void CFRunLoopAddCommonMode(CFRunLoopRef rl, CFRunLoopMod
```

经典 NSTimer 问题解析

介绍完 runloop mode，我们再来看下经典的 NSTimer 调度问题。

我们知道 NSTimer 默认只会调度到 kCFRunLoopDefaultMode 这个模式下，当 scrollView 滑动到时候，runloop 会进入 UITrackingRunLoopMode，那么在 doTimer 的时候自然就不会触发 NSTimer 的任务了，解决办法是将 NSTimer 也加入到 UITrackingRunLoopMode，或者由于 UITrackingRunLoopMode 也是一种 common mode，我们干脆把 NSTimer 加入到 kCFRunLoopCommonModes 里，那么所有所有被标记为 common mode 就都支持 NSTimer 的调用了。

这样是否就支持了所有场景呢？介绍完 runloop 的 private mode 之后，答案显然易见了。即使将 NSTimer 加入了 common mode，当 runloop 在使用任何 private mode （比如上面提到的 _kCFStreamBlockingOpenMode）时，你所设置的 NSTimer 任务还是会被冷落，被延后执行。

那怎么解决呢？没有一劳永逸的解法，毕竟系统和开发者都可以随意创建并使用 private mode，好在主线程 Runloop 觉大部分时候都是以 kCFRunLoopDefaultMode 和 UITrackingRunLoopMode 这两种 mode 运行，而且如果你的任务是对时间非常敏感的，相信你也不会使用 NSTimer 了。

Runloop 的运用

对 runloop 的运用也可以大致分为两类，一是开发者通过 runloop 执行自己的任务，比如 mainQueue，timer 等。另一类就是通过 runloop 观测分析主线程的运行状态。这两类运用的大部分有用信息都在 CoreFoundation/CFRunloop.h 的头文件里。

执行任务

执行任务除了常用的 doMainQueue 和 doTimers 之外，在开源代码里还比较容易看到 doBlocks 和 doSource0 的使用场景。

当然，你还可以建立自己的 runloop mode，设置为 common 或者 private，并将自己想要调度的任务如下 API 放入该 mode 运行：

```
CF_EXPORT CFRunLoopRunResult CFRunLoopRunInMode(CFRunLoopMode mode,
```

观测状态

对于 runloop 状态的观测，无非就是观测 runloop 当前处于那个 activity，或者是处于那个 mode。

总结

本文老调重弹，重炒了 runloop 这碗冷饭，希望对大家有所帮助。

上一篇
iOS App 后台任务的坑 **(/blog/ios-background-task/)**