

# 深入理解RunLoop 由 ibireme | 2015-05-18 | iOS, 技术

RunLoop 是 iOS 和 OSX 开发中非常基础的一个概念，这篇文章将从 CFRunLoop 的源码入手，介绍 RunLoop 的概念以及底层实现原理。之后会介绍一下在 iOS 中，苹果是如何利用 RunLoop 实现自动释放池、延迟回调、触摸事件、屏幕刷新等功能的。

Index

[RunLoop 的概念](#)

[RunLoop 与线程的关系](#)

[RunLoop 对外的接口](#)

[RunLoop 的 Mode](#)

[RunLoop 的内部逻辑](#)

[RunLoop 的底层实现](#)

[苹果用 RunLoop 实现的功能](#)

[AutoreleasePool](#)

[事件响应](#)

[手势识别](#)

[界面更新](#)

[定时器](#)

[PerformSelector](#)

[关于GCD](#)

[关于网络请求](#)

[RunLoop 的实际应用举例](#)

[AFNetworking](#)

[AsyncDisplayKit](#)

## RunLoop 的概念

一般来讲，一个线程一次只能执行一个任务，执行完成后线程就会退出。如果我们需要一个机制，让线程能随时处理事件但并不退出，通常的代码逻辑是这样的：

```
function loop() {  
    initialize();  
    do {  
        var message = get_next_message();  
        process_message(message);  
    } while (message != quit);  
}
```

这种模型通常被称作 [Event Loop](#)。Event Loop 在很多系统和框架里都有实现，比如 Node.js 的事件处理，比如 Windows 程序的消息循环，再比如 OSX/iOS 里的 RunLoop。实现这种模型的关键点在于：如何管理事件/消息，如何让线程在没有处理消息时休眠以避免资源占用、在有消息到来时立刻被唤醒。

所以，RunLoop 实际上就是一个对象，这个对象管理了其需要处理的事件和消息，并提供了一个入口函数来执行上面 Event Loop 的逻辑。线程执行了这个函数后，就会一直处于这个函数内部“接受消息->等待->处理”的循环中，直到这个循环结束（比如传入 quit 的消息），函数返回。

OSX/iOS 系统中，提供了两个这样的对象：NSRunLoop 和 CFRunLoopRef。

CFRunLoopRef 是在 CoreFoundation 框架内的，它提供了纯 C 函数的 API，所有这些 API 都是线程安全的。

NSRunLoop 是基于 CFRunLoopRef 的封装，提供了面向对象的 API，但是这些 API 不是线程安全的。

CFRunLoopRef 的代码是[开源](http://opensource.apple.com/tarballs/CF/)的，你可以在这里 <http://opensource.apple.com/tarballs/CF/> 下载到整个 CoreFoundation 的源码来查看。

(Update: Swift 开源后，苹果又维护了一个跨平台的 CoreFoundation 版本：<https://github.com/apple/swift-corelibs-foundation/>，这个版本的源码可能和现有 iOS 系统中的实现略不一样，但更容易编译，而且已经适配了 Linux/Windows。)

## RunLoop 与线程的关系

首先，iOS 开发中能遇到两个线程对象：pthread\_t 和 NSThread。过去苹果有份[文档](#)标明了 NSThread 只是 pthread\_t 的封装，但那份文档已经失效了，现在它们也有可能都是直接包装自最底层的 mach thread。苹果并没有提供这两个对象相互转换的接口，但不管怎么样，可以肯定的是 pthread\_t 和 NSThread 是一一对应的。比如，你可以通过 pthread\_main\_thread\_np() 或 [NSThread mainThread] 来获取主线程；也可以通过 pthread\_self() 或 [NSThread currentThread] 来获取当前线程。CFRunLoop 是基于 pthread 来管理的。

苹果不允许直接创建 RunLoop，它只提供了两个自动获取的函数：CFRunLoopGetMain() 和 CFRunLoopGetCurrent()。这两个函数内部的逻辑大概是下面这样：

```

/// 全局的Dictionary, key 是 pthread_t, value 是 CFRunLoopRef
static CFMutableDictionaryRef loopsDic;
/// 访问 loopsDic 时的锁
static CFSpinLock_t loopsLock;

/// 获取一个 pthread 对应的 RunLoop。
CFRunLoopRef _CFRunLoopGet(pthread_t thread) {
    OSSpinLockLock(&loopsLock);

    if (!loopsDic) {
        // 第一次进入时, 初始化全局Dic, 并先为主线程创建一个 RunLoop。
        loopsDic = CFDictionaryCreateMutable();
        CFRunLoopRef mainLoop = _CFRunLoopCreate();
        CFDictionarySetValue(loopsDic, pthread_main_thread_np(), mainLoop);
    }

    /// 直接从 Dictionary 里获取。
    CFRunLoopRef loop = CFDictionaryGetValue(loopsDic, thread));

    if (!loop) {
        /// 取不到时, 创建一个
        loop = _CFRunLoopCreate();
        CFDictionarySetValue(loopsDic, thread, loop);
        /// 注册一个回调, 当线程销毁时, 顺便也销毁其对应的 RunLoop。
        _CFSetTSD(..., thread, loop, __CFFinalizeRunLoop);
    }

    OSSpinLockUnlock(&loopsLock);
    return loop;
}

CFRunLoopRef CFRunLoopGetMain() {
    return _CFRunLoopGet(pthread_main_thread_np());
}

CFRunLoopRef CFRunLoopGetCurrent() {
    return _CFRunLoopGet(pthread_self());
}

```

从上面的代码可以看出, 线程和 RunLoop 之间是一一对应的, 其关系是保存在一个全局的 Dictionary 里。线程刚创建时并没有 RunLoop, 如果你不主动获取, 那它一直都不会有。RunLoop 的创建是发生在第一次获取时, RunLoop 的销毁是发生在线程结束时。你只能在一个线程的内部获取其 RunLoop (主线程除外)。

## RunLoop 对外的接口

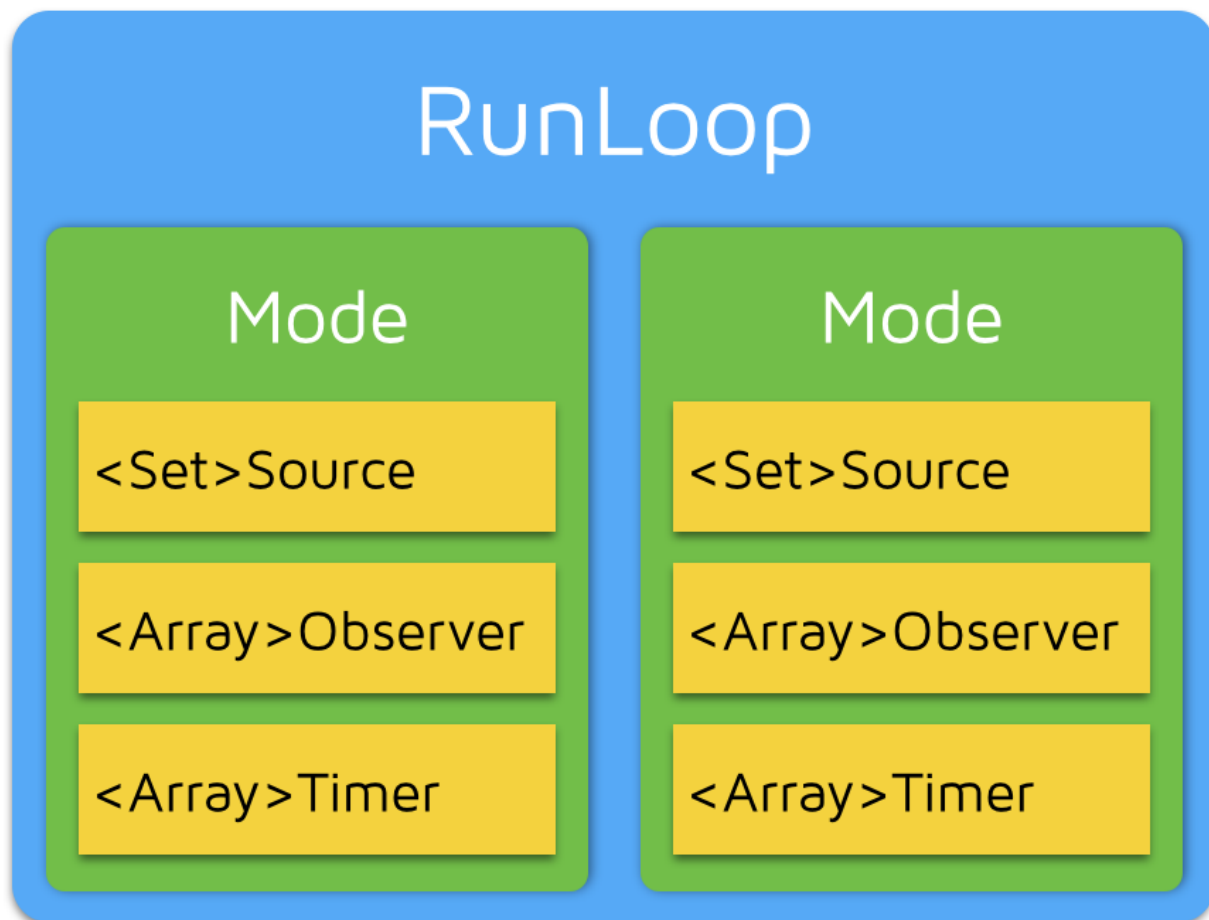
在 CoreFoundation 里面关于 RunLoop 有5个类:

```

CFRunLoopRef
CFRunLoopModeRef
CFRunLoopSourceRef
CFRunLoopTimerRef
CFRunLoopObserverRef

```

其中 `CFRunLoopModeRef` 类并没有对外暴露，只是通过 `CFRunLoopRef` 的接口进行了封装。他们的关系如下：



一个 `RunLoop` 包含若干个 `Mode`，每个 `Mode` 又包含若干个 `Source/Timer/Observer`。每次调用 `RunLoop` 的主函数时，只能指定其中一个 `Mode`，这个 `Mode` 被称作 `CurrentMode`。如果需要切换 `Mode`，只能退出 `Loop`，再重新指定一个 `Mode` 进入。这样做主要是为了分隔开不同组的 `Source/Timer/Observer`，让其互不影响。

**`CFRunLoopSourceRef`** 是事件产生的地方。`Source` 有两个版本：`Source0` 和 `Source1`。

- `Source0` 只包含了一个回调（函数指针），它并不能主动触发事件。使用时，你需要先调用 `CFRunLoopSourceSignal(source)`，将这个 `Source` 标记为待处理，然后手动调用 `CFRunLoopWakeUp(runloop)` 来唤醒 `RunLoop`，让其处理这个事件。
- `Source1` 包含了一个 `mach_port` 和一个回调（函数指针），被用于通过内核和其他线程相互发送消息。这种 `Source` 能主动唤醒 `RunLoop` 的线程，其原理在下面会讲到。

**`CFRunLoopTimerRef`** 是基于时间的触发器，它和 `NSTimer` 是 toll-free bridged 的，可以混用。其包含一个时间长度和一个回调（函数指针）。当其加入到 `RunLoop` 时，`RunLoop` 会注册对应的时间点，当时间点到时，`RunLoop` 会被唤醒以执行那个回调。

**CFRunLoopObserverRef** 是观察者，每个 Observer 都包含了一个回调（函数指针），当 RunLoop 的状态发生变化时，观察者就能通过回调接受到这个变化。可以观测的时间点有以下几个：

```
typedef CF_OPTIONS(CFOptionFlags, CFRunLoopActivity) {
    kCFRunLoopEntry           = (1UL << 0), // 即将进入Loop
    kCFRunLoopBeforeTimers    = (1UL << 1), // 即将处理 Timer
    kCFRunLoopBeforeSources    = (1UL << 2), // 即将处理 Source
    kCFRunLoopBeforeWaiting    = (1UL << 5), // 即将进入休眠
    kCFRunLoopAfterWaiting     = (1UL << 6), // 刚从休眠中唤醒
    kCFRunLoopExit             = (1UL << 7), // 即将退出Loop
};
```

上面的 Source/Timer/Observer 被统称为 **mode item**，一个 item 可以被同时加入多个 mode。但一个 item 被重复加入同一个 mode 时是不会有效果的。如果一个 mode 中一个 item 都没有，则 RunLoop 会直接退出，不进入循环。

## RunLoop 的 Mode

CFRunLoopMode 和 CFRunLoop 的结构大致如下：

```
struct __CFRunLoopMode {
    CFStringRef _name;           // Mode Name, 例如 @"kCFRunLoopDefaultMode"
    CFMutableSetRef _sources0;    // Set
    CFMutableSetRef _sources1;    // Set
    CFMutableArrayRef _observers; // Array
    CFMutableArrayRef _timers;    // Array
    ...
};

struct __CFRunLoop {
    CFMutableSetRef _commonModes; // Set
    CFMutableSetRef _commonModeItems; // Set<Source/Observer/Timer>
    CFRunLoopModeRef _currentMode; // Current Runloop Mode
    CFMutableSetRef _modes;        // Set
    ...
};
```

这里有个概念叫“CommonModes”：一个 Mode 可以将自己标记为“Common”属性（通过将其 ModeName 添加到 RunLoop 的“commonModes”中）。每当 RunLoop 的内容发生变化时，RunLoop 都会自动将 \_commonModeItems 里的 Source/Observer/Timer 同步到具有“Common”标记的所有 Mode 里。

应用场景举例：主线程的 RunLoop 里有两个预置的 Mode：kCFRunLoopDefaultMode 和 UITrackingRunLoopMode。这两个 Mode 都被标记为“Common”属性。DefaultMode 是 App 平时所处的状态，TrackingRunLoopMode 是追踪 ScrollView 滑动时的状态。当你创建一个 Timer 并加到 DefaultMode 时，Timer 会得到重复回调，但此时滑动一个 TableView 时，RunLoop 会将 mode 切换为 TrackingRunLoopMode，这时 Timer 就不会被回调，并且也不会影响到滑动操作。

有时你需要一个 Timer，在两个 Mode 中都能得到回调，一种办法就是将这个 Timer 分别加入这两个 Mode。还有一种方式，就是将 Timer 加入到顶层的 RunLoop 的 “commonModeItems” 中。“commonModeItems” 被 RunLoop 自动更新到所有具有 “Common” 属性的 Mode 里去。

CFRunLoop 对外暴露的管理 Mode 接口只有下面 2 个：

```
CFRunLoopAddCommonMode(CFRunLoopRef runloop, CFStringRef modeName);  
CFRunLoopRunInMode(CFStringRef modeName, ...);
```

Mode 暴露的管理 mode item 的接口有下面几个：

```
CFRunLoopAddSource(CFRunLoopRef rl, CFRunLoopSourceRef source, CFStringRef  
modeName);  
CFRunLoopAddObserver(CFRunLoopRef rl, CFRunLoopObserverRef observer, CFStringRef  
modeName);  
CFRunLoopAddTimer(CFRunLoopRef rl, CFRunLoopTimerRef timer, CFStringRef mode);  
CFRunLoopRemoveSource(CFRunLoopRef rl, CFRunLoopSourceRef source, CFStringRef  
modeName);  
CFRunLoopRemoveObserver(CFRunLoopRef rl, CFRunLoopObserverRef observer,  
CFStringRef modeName);  
CFRunLoopRemoveTimer(CFRunLoopRef rl, CFRunLoopTimerRef timer, CFStringRef  
mode);
```

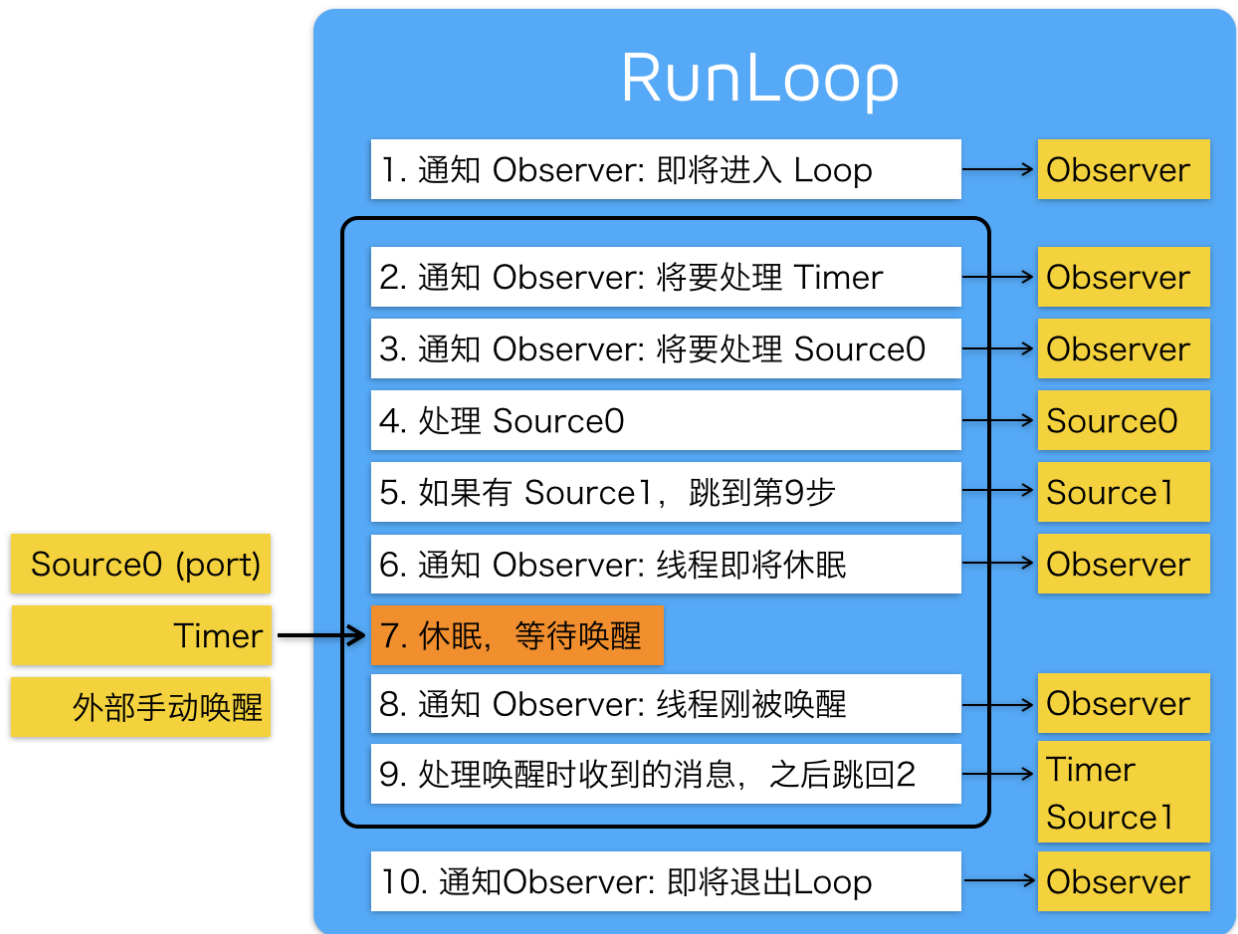
你只能通过 mode name 来操作内部的 mode，当你传入一个新的 mode name 但 RunLoop 内部没有对应 mode 时，RunLoop 会自动帮你创建对应的 CFRunLoopModeRef。对于一个 RunLoop 来说，其内部的 mode 只能增加不能删除。

苹果公开提供的 Mode 有两个：kCFRunLoopDefaultMode (NSDefaultRunLoopMode) 和 UITrackingRunLoopMode，你可以用这两个 Mode Name 来操作其对应的 Mode。

同时苹果还提供了一个操作 Common 标记的字符串：kCFRunLoopCommonModes (NSRunLoopCommonModes)，你可以用这个字符串来操作 Common Items，或标记一个 Mode 为 “Common”。使用时注意区分这个字符串和其他 mode name。

## RunLoop 的内部逻辑

根据苹果在[文档](#)里的说明，RunLoop 内部的逻辑大致如下：



其内部代码整理如下（太长了不想看可以直接跳过去，后面会有说明）：

```

/// 用DefaultMode启动
void CFRunLoopRun(void) {
    CFRunLoopRunSpecific(CFRunLoopGetCurrent(), kCFRunLoopDefaultMode, 1.0e10,
false);
}

/// 用指定的Mode启动, 允许设置RunLoop超时时间
int CFRunLoopRunInMode(CFStringRef modeName, CFTimeInterval seconds, Boolean
stopAfterHandle) {
    return CFRunLoopRunSpecific(CFRunLoopGetCurrent(), modeName, seconds,
returnAfterSourceHandled);
}

/// RunLoop的实现
int CFRunLoopRunSpecific(runloop, modeName, seconds, stopAfterHandle) {

    /// 首先根据modeName找到对应mode
    CFRunLoopModeRef currentMode = __CFRunLoopFindMode(runloop, modeName,
false);
    /// 如果mode里没有source/timer/observer, 直接返回。
    if (__CFRunLoopModeIsEmpty(currentMode)) return;

    /// 1. 通知 Observers: RunLoop 即将进入 loop。
    __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopEntry);

    /// 内部函数, 进入loop
    __CFRunLoopRun(runloop, currentMode, seconds, returnAfterSourceHandled) {

        Boolean sourceHandledThisLoop = NO;
        int retVal = 0;
        do {

            /// 2. 通知 Observers: RunLoop 即将触发 Timer 回调。
            __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopBeforeTimers);
            /// 3. 通知 Observers: RunLoop 即将触发 Source0 (非port) 回调。
            __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopBeforeSources);
            /// 执行被加入的block
            __CFRunLoopDoBlocks(runloop, currentMode);

            /// 4. RunLoop 触发 Source0 (非port) 回调。
            sourceHandledThisLoop = __CFRunLoopDoSources0(runloop, currentMode,
stopAfterHandle);
            /// 执行被加入的block
            __CFRunLoopDoBlocks(runloop, currentMode);

            /// 5. 如果有 Source1 (基于port) 处于 ready 状态, 直接处理这个 Source1 然后跳转去
            处理消息。
            if (__Source0DidDispatchPortLastTime) {
                Boolean hasMsg = __CFRunLoopServiceMachPort(dispatchPort, &msg)
                if (hasMsg) goto handle_msg;
            }

            /// 通知 Observers: RunLoop 的线程即将进入休眠(sleep)。
            if (!sourceHandledThisLoop) {
                __CFRunLoopDoObservers(runloop, currentMode,
kCFRunLoopBeforeWaiting);
            }

            /// 7. 调用 mach_msg 等待接受 mach_port 的消息。线程将进入休眠, 直到被下面某一个
            事件唤醒。
            /// • 一个基于 port 的Source 的事件。
            /// • 一个 Timer 到时间了
            /// • RunLoop 自身的超时时间到了
            /// • 被其他什么调用者手动唤醒

```



```

__CFRunLoopServiceMachPort(waitSet, &msg, sizeof(msg_buffer), &livePort) {
    mach_msg(msg, MACH_RCV_MSG, port); // thread wait for receive msg
}

/// 8. 通知 Observers: RunLoop 的线程刚刚被唤醒了。
__CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopAfterWaiting);

/// 收到消息，处理消息。
handle_msg:

/// 9.1 如果一个 Timer 到时间了，触发这个Timer的回调。
if (msg_is_timer) {
    __CFRunLoopDoTimers(runloop, currentMode, mach_absolute_time())
}

/// 9.2 如果有dispatch到main_queue的block，执行block。
else if (msg_is_dispatch) {
    __CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE__(msg);
}

/// 9.3 如果一个 Source1 (基于port) 发出事件了，处理这个事件
else {
    CFRunLoopSourceRef source1 =
__CFRunLoopModeFindSourceForMachPort(runloop, currentMode, livePort);
    sourceHandledThisLoop = __CFRunLoopDoSource1(runloop, currentMode,
source1, msg);
    if (sourceHandledThisLoop) {
        mach_msg(reply, MACH_SEND_MSG, reply);
    }
}

/// 执行加入到Loop的block
__CFRunLoopDoBlocks(runloop, currentMode);

if (sourceHandledThisLoop && stopAfterHandle) {
    /// 进入loop时参数说处理完事件就返回。
    retVal = kCFRunLoopRunHandledSource;
} else if (timeout) {
    /// 超出传入参数标记的超时时间了
    retVal = kCFRunLoopRunTimedOut;
} else if (__CFRunLoopIsStopped(runloop)) {
    /// 被外部调用者强制停止了
    retVal = kCFRunLoopRunStopped;
} else if (__CFRunLoopModeIsEmpty(runloop, currentMode)) {
    /// source/timer/observer一个都没有了
    retVal = kCFRunLoopRunFinished;
}

/// 如果没超时，mode里没空，loop也没被停止，那继续loop。
} while (retVal == 0);

/// 10. 通知 Observers: RunLoop 即将退出。
__CFRunLoopDoObservers(rl, currentMode, kCFRunLoopExit);
}

```

可以看到，实际上 RunLoop 就是这样一个函数，其内部是一个 do-while 循环。当你调用 CFRunLoopRun() 时，线程就会一直停留在这个循环里；直到超时或被手动停止，该函数才会返回。

# RunLoop 的底层实现

从上面代码可以看到，RunLoop 的核心是基于 mach port 的，其进入休眠时调用的函数是 mach\_msg()。为了解释这个逻辑，下面稍微介绍一下 OSX/iOS 的系统架构。



苹果官方将整个系统大致划分为上述4个层次：

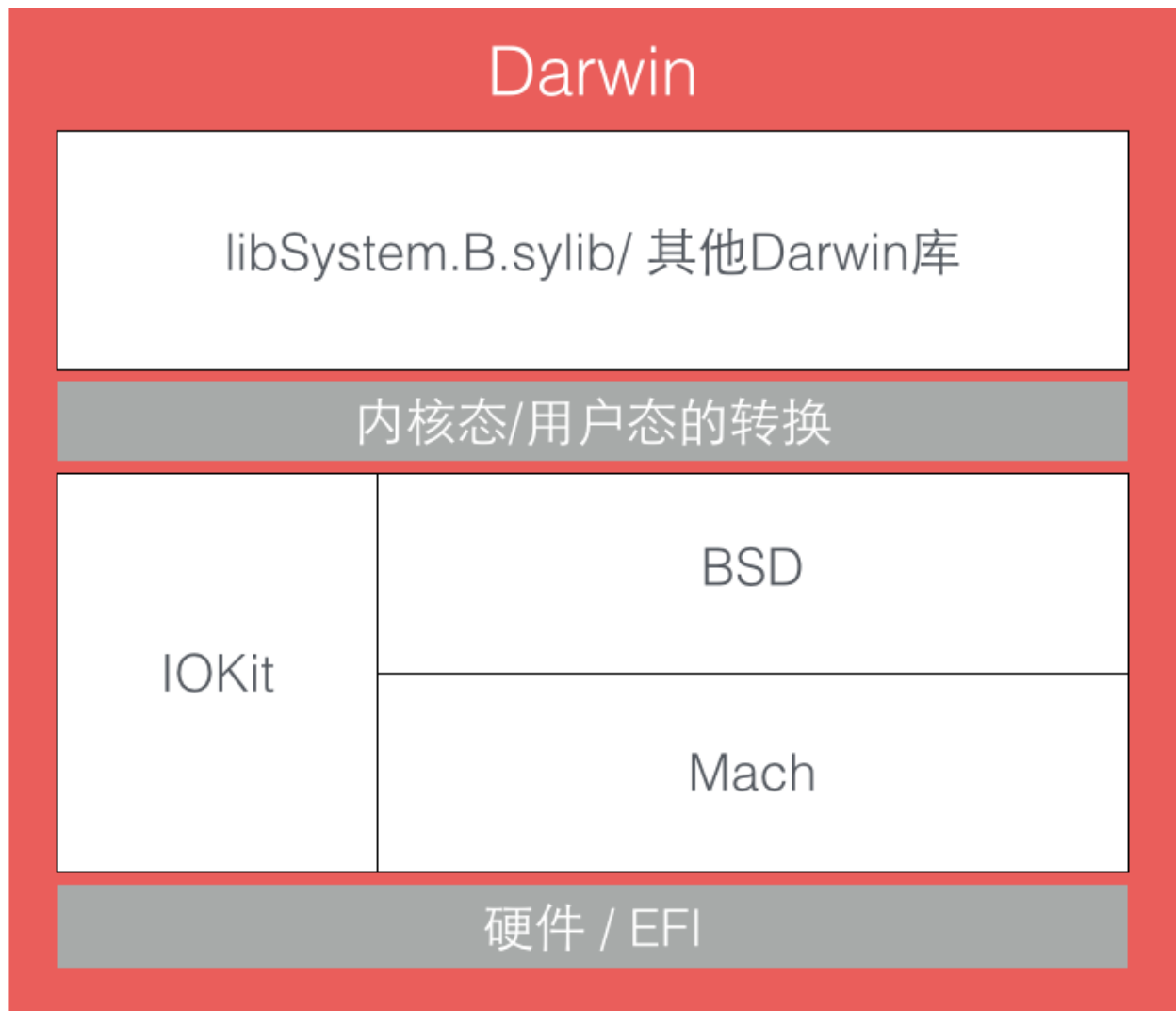
应用层包括用户能接触到的图形应用，例如 Spotlight、Aqua、SpringBoard 等。

应用框架层即开发人员接触到的 Cocoa 等框架。

核心框架层包括各种核心框架、OpenGL 等内容。

Darwin 即操作系统的核心，包括系统内核、驱动、Shell 等内容，这一层是开源的，其所有源码都可以在 [opensource.apple.com](https://opensource.apple.com) 里找到。

我们在深入看一下 Darwin 这个核心的架构：



其中，在硬件层上面的三个组成部分：Mach、BSD、IOKit (还包括一些上面没标注的内容)，共同组成了 XNU 内核。

XNU 内核的内环被称作 Mach，其作为一个微内核，仅提供了诸如处理器调度、IPC (进程间通信)等非常少量的基础服务。

BSD 层可以看作围绕 Mach 层的一个外环，其提供了诸如进程管理、文件系统和网络等功能。

IOKit 层是为设备驱动提供了一个面向对象(C++)的一个框架。

Mach 本身提供的 API 非常有限，而且苹果也不鼓励使用 Mach 的 API，但是这些API非常基础，如果没有这些API的话，其他任何工作都无法实施。在 Mach 中，所有的东西都是通过自己的对象实现的，进程、线程和虚拟内存都被称为“对象”。和其他架构不同，Mach 的对象间不能直接调用，只能通过消息传递的方式实现对象间的通信。“消息”是 Mach 中最基础的概念，消息在两个端口 (port) 之间传递，这就是 Mach 的 IPC (进程间通信) 的核心。

Mach 的消息定义是在 <mach/message.h> 头文件的，很简单：

```
typedef struct {
    mach_msg_header_t header;
    mach_msg_body_t body;
} mach_msg_base_t;

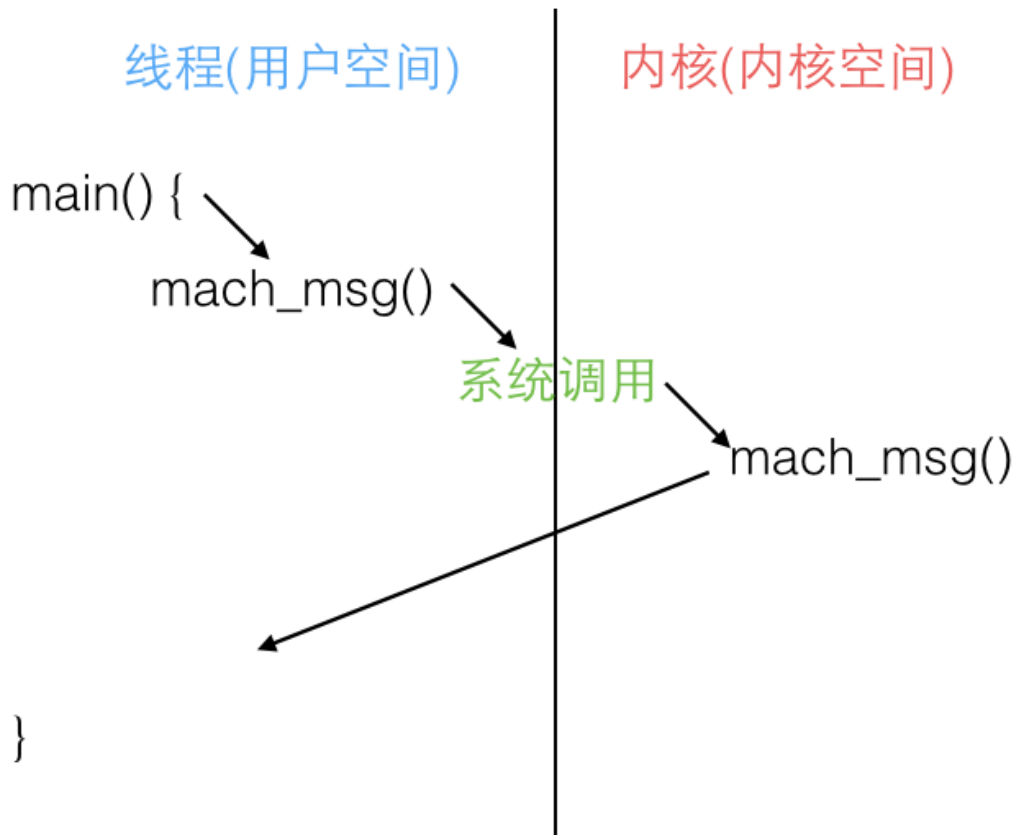
typedef struct {
    mach_msg_bits_t msg_bits;
    mach_msg_size_t msg_size;
    mach_port_t msg_remote_port;
    mach_port_t msg_local_port;
    mach_port_name_t msg_voucher_port;
    mach_msg_id_t msg_id;
} mach_msg_header_t;
```

一条 Mach 消息实际上就是一个二进制数据包 (BLOB)，其头部定义了当前端口 `local_port` 和目标端口 `remote_port`，

发送和接受消息是通过同一个 API 进行的，其 `option` 标记了消息传递的方向：

```
mach_msg_return_t mach_msg(
    mach_msg_header_t *msg,
    mach_msg_option_t option,
    mach_msg_size_t send_size,
    mach_msg_size_t rcv_size,
    mach_port_name_t rcv_name,
    mach_msg_timeout_t timeout,
    mach_port_name_t notify);
```

为了实现消息的发送和接收，`mach_msg()` 函数实际上是调用了一个 Mach 陷阱 (trap)，即函数 `mach_msg_trap()`，陷阱这个概念在 Mach 中等同于系统调用。当你在用户态调用 `mach_msg_trap()` 时会触发陷阱机制，切换到内核态；内核态中内核实现的 `mach_msg()` 函数会完成实际的工作，如下图：



这些概念可以参考维基百科：[System\\_call](#)、[Trap\\_\(computing\)](#)。

RunLoop 的核心就是一个 `mach_msg()` (见上面代码的第7步)，RunLoop 调用这个函数去接收消息，如果没有别人发送 port 消息过来，内核会将线程置于等待状态。例如你在模拟器里跑起一个 iOS 的 App，然后在 App 静止时点击暂停，你会看到主线程调用栈是停留在 `mach_msg_trap()` 这个地方。

关于具体的如何利用 mach port 发送信息，可以看看 [NSHipster 这一篇文章](#)，或者[这里的中文翻译](#)。

关于Mach的历史可以看看这篇很有趣的文章：[Mac OS X 背后的故事 \(三\) Mach 之父 Avie Tevanian](#)。

## 苹果用 RunLoop 实现的功能

首先我们可以看一下 App 启动后 RunLoop 的状态：

```

CFRunLoop {
    current mode = kCFRunLoopDefaultMode
    common modes = {
        UITrackingRunLoopMode
        kCFRunLoopDefaultMode
    }

    common mode items = {

        // source0 (manual)
        CFRunLoopSource {order = -1, {
            callout = _UIApplicationHandleEventQueue}}
        CFRunLoopSource {order = -1, {
            callout = PurpleEventSignalCallback }}
        CFRunLoopSource {order = 0, {
            callout = FBSSerialQueueRunLoopSourceHandler}}

        // source1 (mach port)
        CFRunLoopSource {order = 0, {port = 17923}}
        CFRunLoopSource {order = 0, {port = 12039}}
        CFRunLoopSource {order = 0, {port = 16647}}
        CFRunLoopSource {order = -1, {
            callout = PurpleEventCallback}}
        CFRunLoopSource {order = 0, {port = 2407,
            callout = _ZL20notify_port_callbackP12__CFMachPortPvIS1_}}
        CFRunLoopSource {order = 0, {port = 1c03,
            callout = __IOHIDEventSystemClientAvailabilityCallback}}
        CFRunLoopSource {order = 0, {port = 1b03,
            callout = __IOHIDEventSystemClientQueueCallback}}
        CFRunLoopSource {order = 1, {port = 1903,
            callout = __IOMIGMachPortPortCallback}}

        // Ovserver
        CFRunLoopObserver {order = -2147483647, activities = 0x1, // Entry
            callout = _wrapRunLoopWithAutoreleasePoolHandler}
        CFRunLoopObserver {order = 0, activities = 0x20, // BeforeWaiting
            callout = _UIGestureRecognizerUpdateObserver}
        CFRunLoopObserver {order = 1999000, activities = 0xa0, // BeforeWaiting |
Exit
            callout = _afterCACCommitHandler}
        CFRunLoopObserver {order = 2000000, activities = 0xa0, // BeforeWaiting |
Exit
            callout =
            _ZN2CA11Transaction17observer_callbackEP19__CFRunLoopObservermPv}
        CFRunLoopObserver {order = 2147483647, activities = 0xa0, // BeforeWaiting |
Exit
            callout = _wrapRunLoopWithAutoreleasePoolHandler}

        // Timer
        CFRunLoopTimer {firing = No, interval = 3.1536e+09, tolerance = 0,
            next fire date = 453098071 (-4421.76019 @ 96223387169499),
            callout = _ZN2CAL14timer_callbackEP16__CFRunLoopTimerPv
(QuartzCore.framework)}}
    },

    modes = {
        CFRunLoopMode {
            sources0 = { /* same as 'common mode items' */ },
            sources1 = { /* same as 'common mode items' */ },
            observers = { /* same as 'common mode items' */ },
            timers = { /* same as 'common mode items' */ },

```

```

    },

    CFRunLoopMode {
        sources0 = { /* same as 'common mode items' */ },
        sources1 = { /* same as 'common mode items' */ },
        observers = { /* same as 'common mode items' */ },
        timers = { /* same as 'common mode items' */ },
    },

    CFRunLoopMode {
        sources0 = {
            CFRunLoopSource {order = 0, {
                callout = FBSSerialQueueRunLoopSourceHandler}}
        },
        sources1 = (null),
        observers = {
            CFRunLoopObserver >{activities = 0xa0, order = 2000000,
                callout =
_ZN2CA11Transaction17observer_callbackEP19__CFRunLoopObservermPv}
            }},
        timers = (null),
    },

    CFRunLoopMode {
        sources0 = {
            CFRunLoopSource {order = -1, {
                callout = PurpleEventSignalCallback}}
        },
        sources1 = {
            CFRunLoopSource {order = -1, {
                callout = PurpleEventCallback}}
        },
        observers = (null),
        timers = (null),
    },

    CFRunLoopMode {
        sources0 = (null),
        sources1 = (null),
        observers = (null),
        timers = (null),
    }
}
}

```

可以看到，系统默认注册了5个Mode：

1. kCFRunLoopDefaultMode: App的默认 Mode，通常主线程是在这个 Mode 下运行的。
2. UITrackingRunLoopMode: 界面跟踪 Mode，用于 ScrollView 追踪触摸滑动，保证界面滑动时不受其他 Mode 影响。
3. 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。
4. GSEventReceiveRunLoopMode: 接受系统事件的内部 Mode，通常用不到。
5. kCFRunLoopCommonModes: 这是一个占位的 Mode，没有实际作用。

你可以在[这里](#)看到更多的苹果内部的 Mode，但那些 Mode 在开发中就很难遇到了。

当 RunLoop 进行回调时，一般都是通过一个很长的函数调用出去 (call out), 当你在你的代码中下断点调试时，通常能在调用栈上看到这些函数。下面是这几个函数的整理版本，如果你在调用栈中看到这些长函数名，在这里查找一下就能定位到具体的调用地点了：

```
{
    /// 1. 通知Observers, 即将进入RunLoop
    /// 此处有Observer会创建AutoreleasePool: _objc_autoreleasePoolPush();
    __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__(kCFRunLoopEntry);
    do {

        /// 2. 通知 Observers: 即将触发 Timer 回调。
        __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__(kCFRunLoopBeforeTimer);
        /// 3. 通知 Observers: 即将触发 Source (非基于port的,Source0) 回调。
        __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__(kCFRunLoopBeforeSources);
        __CFRUNLOOP_IS_CALLING_OUT_TO_A_BLOCK__(block);

        /// 4. 触发 Source0 (非基于port的) 回调。
        __CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION__(source0);
        __CFRUNLOOP_IS_CALLING_OUT_TO_A_BLOCK__(block);

        /// 6. 通知Observers, 即将进入休眠
        /// 此处有Observer释放并新建AutoreleasePool: _objc_autoreleasePoolPop(); _objc_autoreleasePoolPush();
        __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__(kCFRunLoopBeforeWaiting);

        /// 7. sleep to wait msg.
        mach_msg() -> mach_msg_trap();

        /// 8. 通知Observers, 线程被唤醒
        __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__(kCFRunLoopAfterWaiting);

        /// 9. 如果是被Timer唤醒的, 回调Timer
        __CFRUNLOOP_IS_CALLING_OUT_TO_A_TIMER_CALLBACK_FUNCTION__(timer);

        /// 9. 如果是被dispatch唤醒的, 执行所有调用 dispatch_async 等方法放入main queue 的 block
        __CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE__(dispatched_block);

        /// 9. 如果如果RunLoop是被 Source1 (基于port的) 的事件唤醒了, 处理这个事件
        __CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE1_PERFORM_FUNCTION__(source1);

    } while (...);

    /// 10. 通知Observers, 即将退出RunLoop
    /// 此处有Observer释放AutoreleasePool: _objc_autoreleasePoolPop();
    __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__(kCFRunLoopExit);
}
```

## AutoreleasePool

App启动后，苹果在主线程 RunLoop 里注册了两个 Observer，其回调都是 `_wrapRunLoopWithAutoreleasePoolHandler()`。

第一个 Observer 监视的事件是 Entry(即将进入Loop)，其回调内会调用 `_objc_autoreleasePoolPush()` 创建自动释放池。其 order 是-2147483647，优先级最高，保证创建释放池发生在其他所有回调之前。



第二个 Observer 监视了两个事件：BeforeWaiting(准备进入休眠) 时调用 `_objc_autoreleasePoolPop()` 和 `_objc_autoreleasePoolPush()` 释放旧的池并创建新池；Exit(即将退出Loop) 时调用 `_objc_autoreleasePoolPop()` 来释放自动释放池。这个 Observer 的 order 是 2147483647，优先级最低，保证其释放池子发生在其他所有回调之后。

在主线程执行的代码，通常是写在诸如事件回调、Timer回调内的。这些回调会被 RunLoop 创建好的 AutoreleasePool 环绕着，所以不会出现内存泄漏，开发者也不必显示创建 Pool 了。

## 事件响应

苹果注册了一个 Source1 (基于 mach port 的) 用来接收系统事件，其回调函数为 `__IOHIDEventSystemClientQueueCallback()`。

当一个硬件事件(触摸/锁屏/摇晃等)发生后，首先由 IOKit.framework 生成一个 IOHIDEvent 事件并由 SpringBoard 接收。这个过程的具体情况可以参考[这里](#)。SpringBoard 只接收按键(锁屏/静音等)，触摸，加速，接近传感器等几种 Event，随后用 mach port 转发给需要的App进程。随后苹果注册的那个 Source1 就会触发回调，并调用 `_UIApplicationHandleEventQueue()` 进行应用内部的分发。

`_UIApplicationHandleEventQueue()` 会把 IOHIDEvent 处理并包装成 UIEvent 进行处理或分发，其中包括识别 UIGesture/处理屏幕旋转/发送给 UIWindow 等。通常事件比如 UIButton 点击、touchesBegin/Move/End/Cancel 事件都是在这个回调中完成的。

## 手势识别

当上面的 `_UIApplicationHandleEventQueue()` 识别了一个手势时，其首先会调用 Cancel 将当前的 touchesBegin/Move/End 系列回调打断。随后系统将对应的 UIGestureRecognizer 标记为待处理。

苹果注册了一个 Observer 监测 BeforeWaiting (Loop即将进入休眠) 事件，这个Observer的回调函数是 `_UIGestureRecognizerUpdateObserver()`，其内部会获取所有刚被标记为待处理的 GestureRecognizer，并执行GestureRecognizer的回调。

当有 UIGestureRecognizer 的变化(创建/销毁/状态改变)时，这个回调都会进行相应处理。

## 界面更新

当在操作 UI 时，比如改变了 Frame、更新了 UIView/CALayer 的层次时，或者手动调用了 UIView/CALayer 的 `setNeedsLayout/setNeedsDisplay`方法后，这个 UIView/CALayer 就被标记为待处理，并被提交到一个全局的容器去。

苹果注册了一个 Observer 监听 BeforeWaiting(即将进入休眠) 和 Exit (即将退出Loop) 事件，回调去执行一个很长的函数：

`_ZN2CA11Transaction17observer_callbackEP19__CFRunLoopObservermPv()`。这个函数里会遍历所有待处理的 UIView/CALayer 以执行实际的绘制和调整，并更新 UI 界面。

这个函数内部的调用栈大概是这样的：

```
_ZN2CA11Transaction17observer_callbackEP19__CFRunLoopObservermPv()  
QuartzCore:CA::Transaction::observer_callback:  
    CA::Transaction::commit();  
    CA::Context::commit_transaction();  
    CA::Layer::layout_and_display_if_needed();  
    CA::Layer::layout_if_needed();  
        [CALayer layoutSublayers];  
        [UIView layoutSubviews];  
    CA::Layer::display_if_needed();  
        [CALayer display];  
        [UIView drawRect];
```

## 定时器

NSTimer 其实就是 CFRunLoopTimerRef，他们之间是 toll-free bridged 的。一个 NSTimer 注册到 RunLoop 后，RunLoop 会为其重复的时间点注册好事件。例如 10:00, 10:10, 10:20 这几个时间点。RunLoop 为了节省资源，并不会在非常准确的时间点回调这个 Timer。Timer 有个属性叫做 Tolerance (宽容度)，标示了当时间点到后，容许有多少最大误差。

如果某个时间点被错过了，例如执行了一个很长的任务，则那个时间点的回调也会跳过去，不会延后执行。就好比等公交，如果 10:10 时我忙着玩手机错过了那个点的公交，那我只能等 10:20 这一趟了。

CADisplayLink 是一个和屏幕刷新率一致的定时器（但实际实现原理更复杂，和 NSTimer 并不一样，其内部实际是操作了一个 Source）。如果在两次屏幕刷新之间执行了一个长任务，那其中就会有一帧被跳过去（和 NSTimer 相似），造成界面卡顿的感觉。在快速滑动 TableView 时，即使一帧的卡顿也会让用户有所察觉。Facebook 开源的 AsyncDisplayLink 就是为了解决界面卡顿的问题，其内部也用到了 RunLoop，这个稍后我会再单独写一页博客来分析。

## PerformSelector

当调用 NSObject 的 performSelector:afterDelay: 后，实际上其内部会创建一个 Timer 并添加到当前线程的 RunLoop 中。所以如果当前线程没有 RunLoop，则这个方法会失效。

当调用 performSelector:onThread: 时，实际上其会创建一个 Timer 加到对应的线程去，同样的，如果对应线程没有 RunLoop 该方法也会失效。

## 关于GCD

实际上 RunLoop 底层也会用到 GCD 的东西，比如 RunLoop 是用 dispatch\_source\_t 实现的 Timer（评论中有人提醒，NSTimer 是用了 XNU 内核的 mk\_timer，我也仔细调试了一下，发现 NSTimer 确实是由 mk\_timer 驱动，而非 GCD 驱动的）。但同时 GCD 提供的某些接口也用到了 RunLoop，例如 dispatch\_async()。

当调用 dispatch\_async(dispatch\_get\_main\_queue(), block) 时，libDispatch 会向主线程的 RunLoop 发送消息，RunLoop 会被唤醒，并从消息中取得这个 block，并在回调 \_\_CFRUNLOOP\_IS\_SERVICING\_THE\_MAIN\_DISPATCH\_QUEUE\_\_() 里执行这个 block。但这个逻辑仅限于 dispatch 到主线程，dispatch 到其他线程仍然是由 libDispatch 处理的。

## 关于网络请求

iOS 中，关于网络请求的接口自下至上有如下几层：

```
CFSocket
CFNetwork    ->ASIHttpRequest
NSURLConnection ->AFNetworking
NSURLSession  ->AFNetworking2, Alamofire
```

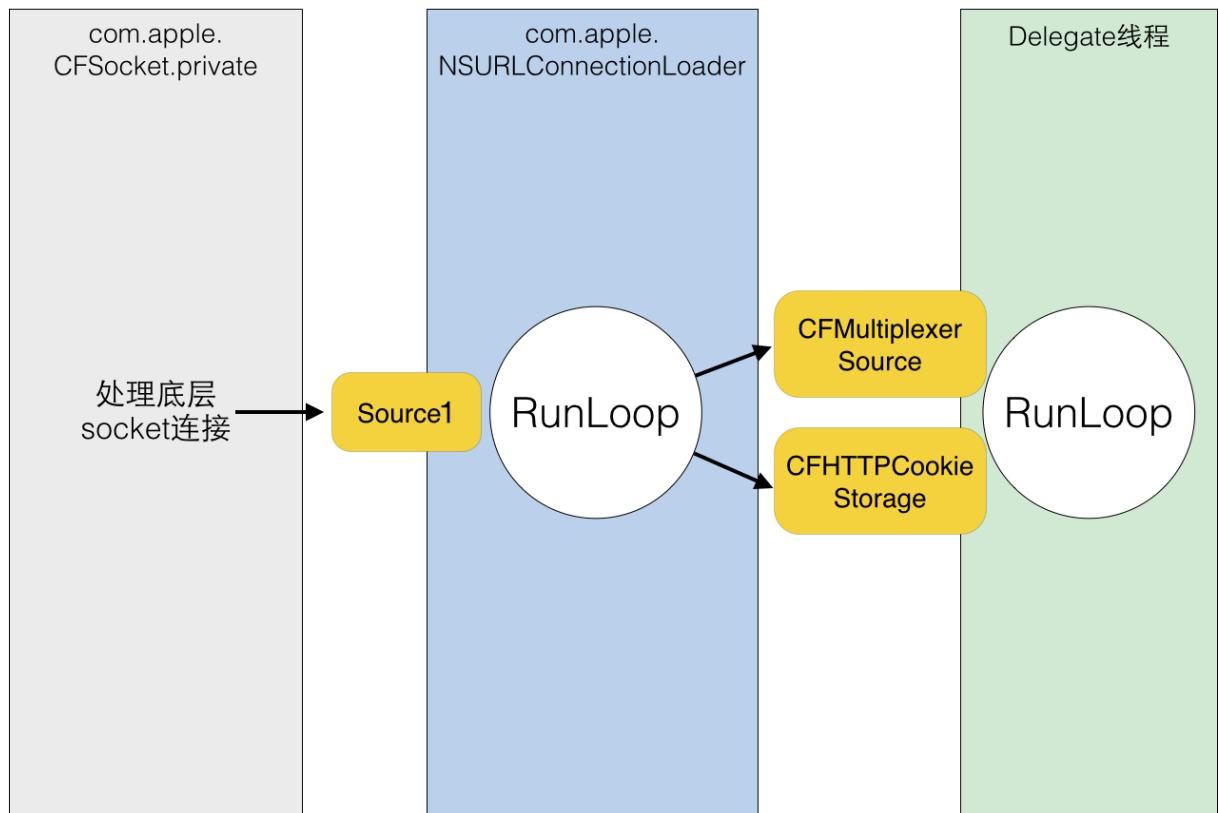
- CFSocket 是最底层的接口，只负责 socket 通信。
- CFNetwork 是基于 CFSocket 等接口的上层封装，ASIHttpRequest 工作于这一层。
- NSURLConnection 是基于 CFNetwork 的更高层的封装，提供面向对象的接口，AFNetworking 工作于这一层。
- NSURLSession 是 iOS7 中新增的接口，表面上是和 NSURLConnection 并列的，但底层仍然用到了 NSURLConnection 的部分功能（比如 com.apple.NSURLConnectionLoader 线程），AFNetworking2 和 Alamofire 工作于这一层。

下面主要介绍下 NSURLConnection 的工作过程。

通常使用 NSURLConnection 时，你会传入一个 Delegate，当调用了 [connection start] 后，这个 Delegate 就会不停收到事件回调。实际上，start 这个函数的内部会获取 CurrentRunLoop，然后在其中的 DefaultMode 添加了4个 Source0（即需要手动触发的Source）。CFMultiplexerSource 是负责各种 Delegate 回调的，CFHTTPCookieStorage 是处理各种 Cookie 的。

当开始网络传输时，我们可以看到 NSURLConnection 创建了两个新线程：

com.apple.NSURLConnectionLoader 和 com.apple.CFSocket.private。其中 CFSocket 线程是处理底层 socket 连接的。NSURLConnectionLoader 这个线程内部会使用 RunLoop 来接收底层 socket 的事件，并通过之前添加的 Source0 通知到上层的 Delegate。



NSURLConnectionLoader 中的 RunLoop 通过一些基于 mach port 的 Source 接收来自底层 CFSocket 的通知。当收到通知后，其会在合适的时机向 CFMultiplexerSource 等 Source0 发送通知，同时唤醒 Delegate 线程的 RunLoop 来让其处理这些通知。CFMultiplexerSource 会在 Delegate 线程的 RunLoop 对 Delegate 执行实际的回调。

## RunLoop 的实际应用举例

### AFNetworking

[AFURLConnectionOperation](#) 这个类是基于 NSURLConnection 构建的，其希望能在后台线程接收 Delegate 回调。为此 AFNetworking 单独创建了一个线程，并在这个线程中启动了一个 RunLoop：

```

+ (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;
}

```

RunLoop 启动前内部必须要有至少一个 Timer/Observer/Source，所以 AFNetworking 在 [runLoop run] 之前先创建了一个新的 NSMachPort 添加进去了。通常情况下，调用者需要持有这个 NSMachPort (mach\_port) 并在外部线程通过这个 port 发送消息到 loop 内；但此处添加 port 只是为了让 RunLoop 不至于退出，并没有用于实际的发送消息。

```

- (void)start {
    [self.lock lock];
    if ([self isCancelled]) {
        [self performSelector:@selector(cancelConnection) onThread:[self class]
        networkRequestThread] withObject:nil waitUntilDone:NO modes:[self.runLoopModes
        allObjects]];
    } else if ([self isReady]) {
        self.state = AFOperationExecutingState;
        [self performSelector:@selector(operationDidStart) onThread:[self class]
        networkRequestThread] withObject:nil waitUntilDone:NO modes:[self.runLoopModes
        allObjects]];
    }
    [self.lock unlock];
}

```

当需要这个后台线程执行任务时，AFNetworking 通过调用 [NSObject performSelector:onThread:...] 将这个任务扔到了后台线程的 RunLoop 中。

## AsyncDisplayKit

[AsyncDisplayKit](#) 是 Facebook 推出的用于保持界面流畅性的框架，其原理大致如下：

UI 线程中一旦出现繁重的任务就会导致界面卡顿，这类任务通常分为3类：排版，绘制，UI对象操作。

排版通常包括计算视图大小、计算文本高度、重新计算子式图的排版等操作。

绘制一般有文本绘制（例如 CoreText）、图片绘制（例如预先解压）、元素绘制（Quartz）等操作。

UI对象操作通常包括 UIView/CALayer 等 UI 对象的创建、设置属性和销毁。

其中前两类操作可以通过各种方法扔到后台线程执行，而最后一类操作只能在主线程完成，并且有时后面的操作需要依赖前面操作的结果（例如TextView创建时可能需要提前计算出文本的大小）。ASDK 所做的，就是尽量将能放入后台的任务放入后台，不能的则尽量推迟（例如视图的创建、属性的调整）。

为此，ASDK 创建了一个名为 ASDisplayNode 的对象，并在内部封装了 UIView/CALayer，它具有和 UIView/CALayer 相似的属性，例如 frame、backgroundColor等。所有这些属性都可以在后台线程更改，开发者可以只通过 Node 来操作其内部的 UIView/CALayer，这样就可以将排版和绘制放入了后台线程。但是无论怎么操作，这些属性总需要在某个时刻同步到主线程的 UIView/CALayer 去。

ASDK 仿照 QuartzCore/UIKit 框架的模式，实现了一套类似的界面更新的机制：即在主线程的 RunLoop 中添加一个 Observer，监听了 kCFRunLoopBeforeWaiting 和 kCFRunLoopExit 事件，在收到回调时，遍历所有之前放入队列的待处理的任务，然后一一执行。

具体的代码可以看这里：[\\_ASAsyncTransactionGroup](#)。

## 最后

好长时间没写博客了喵～前几天给博客搬了个家，从越来越慢的 AWS 迁到了 Linode，然后很认真的换了一套新的博客主题，排版看着还说过得去吧～ 😊