

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](#)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 Run
        Loop。
        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 RunLo
        op。
        _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

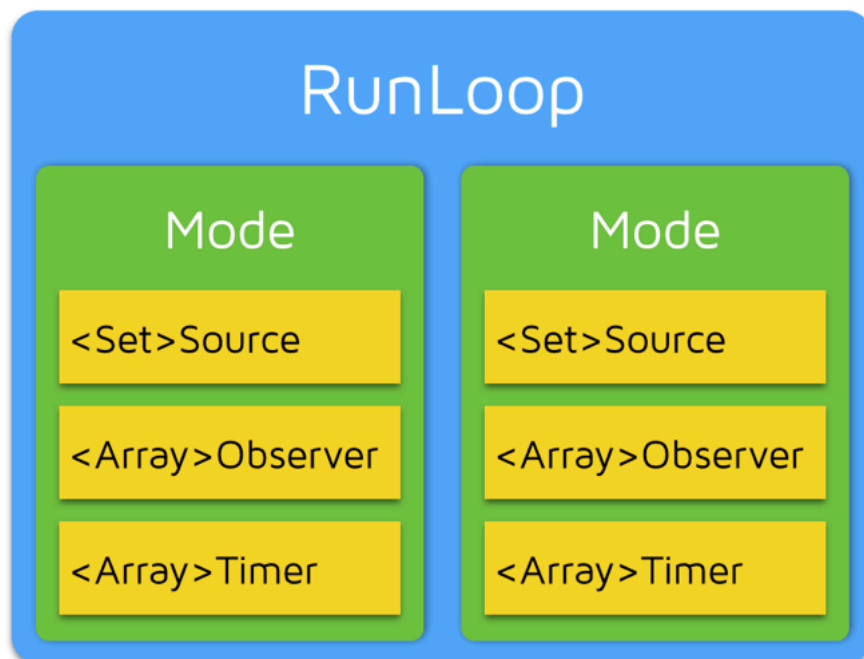
CFRunLoopModeRef

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 @"kCFRun
    LoopDefaultMode"
    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 Mod
    e
    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 都已经被标记为“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 “commonModelItems” 中。”commonModelItems” 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 r1, CFRunLoopSourceRef source, CFStringRef modeName);  
CFRunLoopAddObserver(CFRunLoopRef r1, CFRunLoopObserverRef observer, CFStringRef modeName);  
CFRunLoopAddTimer(CFRunLoopRef r1, CFRunLoopTimerRef timer, CFStringRef mode);  
CFRunLoopRemoveSource(CFRunLoopRef r1, CFRunLoopSourceRef source, CFStringRef modeName);  
CFRunLoopRemoveObserver(CFRunLoopRef r1, CFRunLoopObserverRef observer, CFStringRef modeName);  
CFRunLoopRemoveTimer(CFRunLoopRef r1, 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。

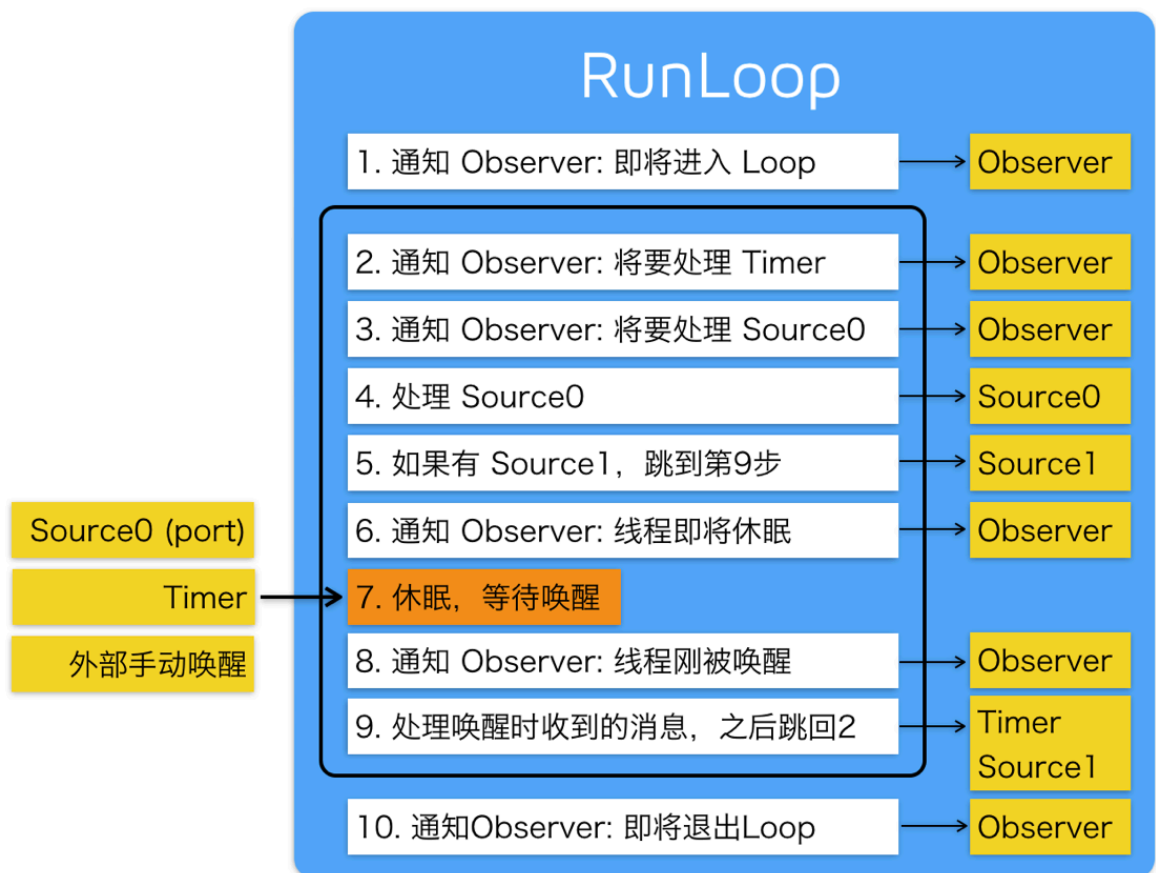
苹果公开提供的 Mode 有两个：kCFRunLoopDefaultMode

(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(), kCFRunLoopDe
faultMode, 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(), modeN
ame, seconds, returnAfterSourceHandled);
}

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

    /// 首先根据modeName找到对应mode
    CFRunLoopModeRef currentMode = __CFRunLoopFindMode(runlo
op, 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, kCFRunLoopE
ntry);

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

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

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

            /// 4. RunLoop 触发 Source0 (非port) 回调。
            sourceHandledThisLoop = __CFRunLoopDoSources0(ru
nloop, 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, kCF
RunLoopAfterWaiting);

    /// 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, ma
ch_absolute_time())
    }

    /// 9.2 如果有dispatch到main_queue的block，执行bl
    ock。
    else if (msg_is_dispatch) {
        __CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_Q
UEUE__(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 = __CFRunLoopMode
FindSourceForMachPort(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(r1, 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：

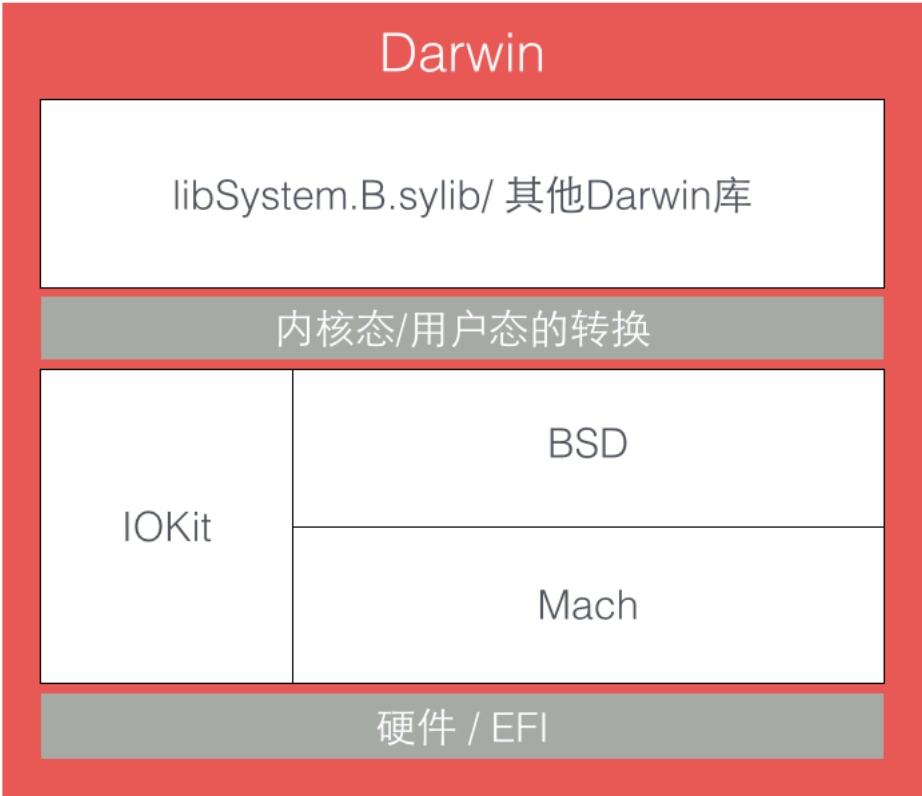
应用层包括用户能接触到的图形应用，例如 Spotlight、Aqua、SpringBoard 等。

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 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;

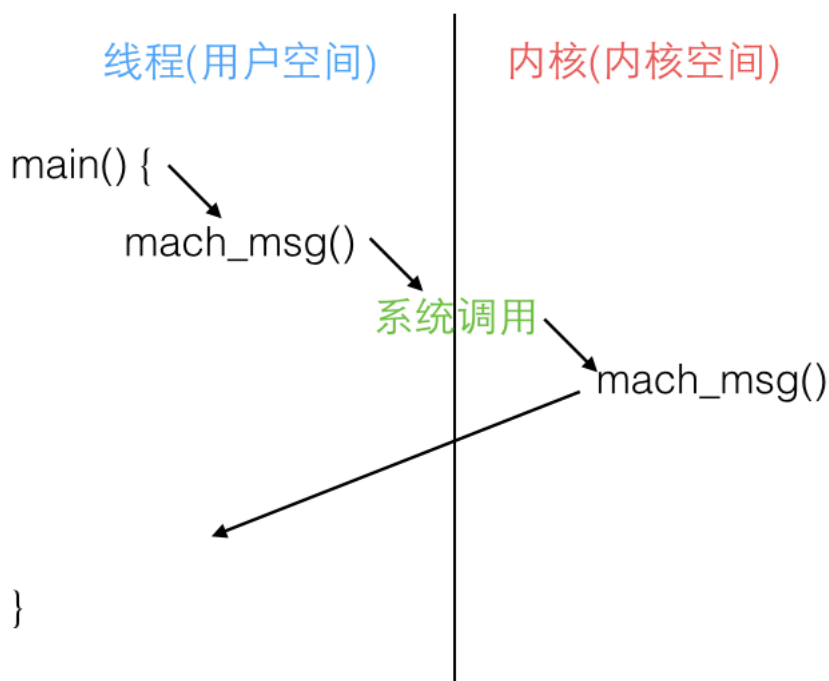
typedef struct {
    mach_msg_bits_t msgh_bits;
    mach_msg_size_t msgh_size;
    mach_port_t msgh_remote_port;
    mach_port_t msgh_local_port;
    mach_port_name_t msgh_voucher_port;
    mach_msg_id_t msgh_id;
} mach_msg_header_t;
```

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__CFMachPortPv1S1_}}
        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 = 0xa
0, // BeforeWaiting | Exit
    callout = _afterCACommitHandler}
CFRunLoopObserver {order = 2000000, activities = 0xa
0, // BeforeWaiting | Exit
    callout = _ZN2CA11Transaction17observer_callback
EP19__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 @ 962233
87169499),
    callout = _ZN2CAL14timer_callbackEP16__CFRunLoop
TimerPv (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 = FBSSerialQueueRunLoopSourceHan
dler}}
    },
    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. UIApplicationRunLoopMode: 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_autoreleas
    ePoolPush();
    __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNC
    TION__(kCFRunLoopEntry);
    do {

        /// 2. 通知 Observers: 即将触发 Timer 回调。
        __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_F
        UNCTION__(kCFRunLoopBeforeTimers);
        /// 3. 通知 Observers: 即将触发 Source (非基于port的,S
        ource0) 回调。
        __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_F
        UNCTION__(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_FUNC
        TION__(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_au
        toreleasePoolPop(); _objc_autoreleasePoolPush();
        __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_F
        UNCTION__(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_F
        UNCTION__(kCFRunLoopAfterWaiting);
    } while (true);
}
```

```

        /// 9. 如果是被Timer唤醒的, 回调Timer
        __CFRunLoopIsCallingOutToATimerCallbackFunction__(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
        __CFRunLoopIsServicingTheMainDispatchQueue__(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
        __CFRunLoopIsCallingOutToASource1PerformFunction__(source1);

    } while (...);

    /// 10. 通知Observers, 即将退出RunLoop
    /// 此处有Observer释放AutoreleasePool: _objc_autoreleasePoolPop();
    __CFRunLoopIsCallingOutToAnObserverCallbackFunction__(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。就比如等公交，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
```

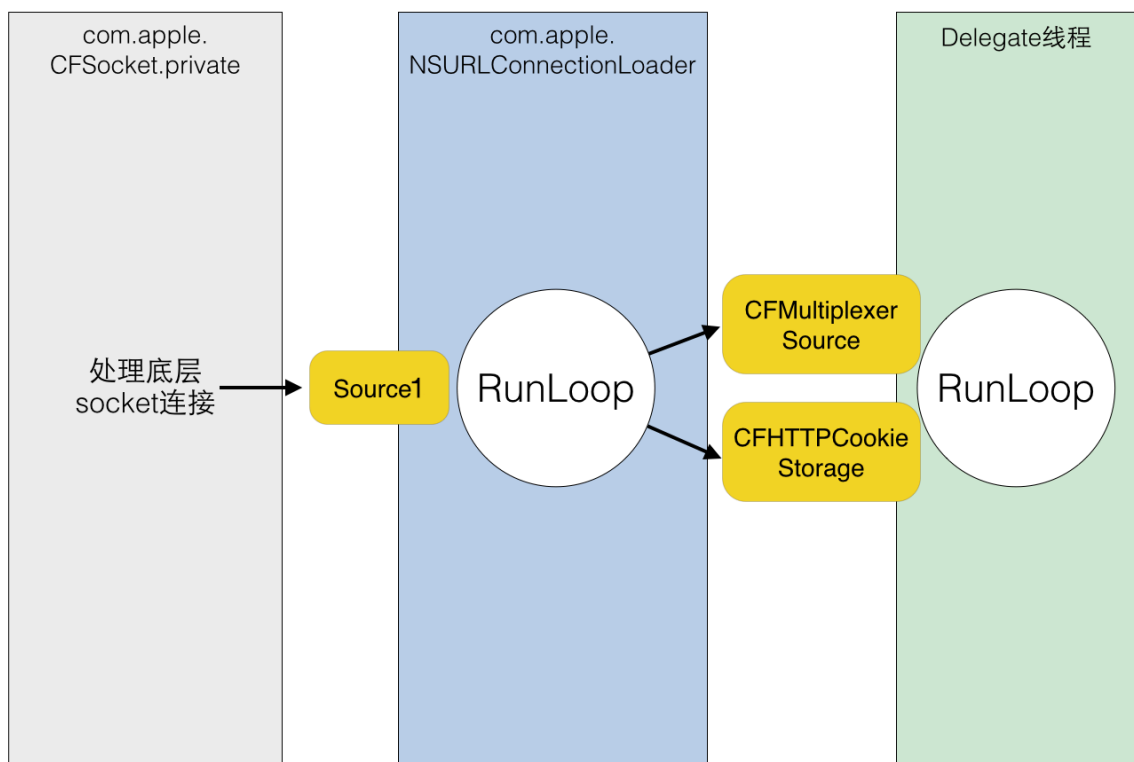
- CFSocket 是最底层的接口，只负责 socket 通信。

- 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 kCFSRunLoopBeforeWaiting 和 kCFSRunLoopExit 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, 排版看着还说得过去吧~ :oops:

157 评论






庞海礁    在 2015 年 5 月 26 日的 下午 8:14

回复

请问你是在百度工作的吗? 前段时间看了一个百度工作的介绍nsrunloop的视频, 不知道是不是你?



ibireme    在 2015 年 5 月 27 日的 上午 1:09

回复

不是, 我没在度厂待过。。

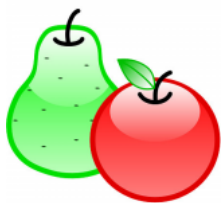


li    在 2016 年 12 月 22 日的 上午 10:35

回复

当调用 performSelector:onThread: 时, 实际上其会创建一个 Timer 加到对应的线程去 ;-);-);-) 这个说法 有点问题, 我发现 实际上是创建了一

↑source1



苹果梨 🇨🇳 🌐 🍏 在 2018 年 3 月 26 日 的 上午 11:24

回复

<https://developer.apple.com/documentation/objectivec/nsobject/1416176-performselector>

还是以官方文档为准比较好



我七岁就很帅 🇨🇳 🌐 🍏 在 2018 年 5 月 15 日 的 上午 9:32

官方文档说的是

performSelector:withObject:afterDelay:会在当前线程的RunLoop中创建并添加一个Timer。但是performSelector:onThread: 实际上是创建了一个Source0，是Source0，不是Source1，更不是Timer。



李重阳 🇨🇳 🌐 🍏 在 2019 年 3 月 15 日 的 上午 10:16

回复

是的 我打断点是 source1



李重阳 🇨🇳 🌐 🍏 在 2019 年 3 月 15 日 的 上午 10:33

是source0



ian 在 2015 年 5 月 31 日的下午 10:04

回复

@庞海礁 那个貌似是 我叫sunnyxx怎么了 的视频



淡水湖 在 2016 年 6 月 12 日的下午 6:11

回复

:twisted: 评论现在俩人在滴滴一起相亲相爱了



王千 在 2015 年 8 月 24 日的下午 7:28

回复

评论 ;-) 求视频,据说是两期线下活动的视频,搜不到啊,感谢1093534383@qq.com



sclcoder 在 2015 年 10 月 9 日的下午

回复

4:52

他的微博里有



钱嘘嘘 在 2015 年 12 月 15 日的上午 11:12

回复

http://v.youku.com/v_show/id_XODgxODkzODI0.html






张明明 在 2016 年 1 月 25 日的下午 10:29

回复

博主,这方面在工作中应用的多不,小白求推荐书籍






赤壁    在 2016 年 8 月 8 日的 下午 3:05

回复

百度介绍RUNLOOP 的 孙源 视频在优酷上有 那个视频我看了好几遍 记录的笔记也没有这篇文章详细 老实说 不过 如果对RUNLOOP 完全没有概念 是推荐看那个视频的






牧马    在 2017 年 6 月 18 日的 下午 7:21

回复

评论 :shock: 你说的是 sunny 孙源



John    在 2015 年 5 月 27 日的 下午 8:23

回复

好赞






daixunry    在 2015 年 5 月 28 日的 下午

回复

7:03




太棒了！收藏起来，可以经常回头看看



离青    在 2015 年 6 月 1 日的 上午 10:19

回复

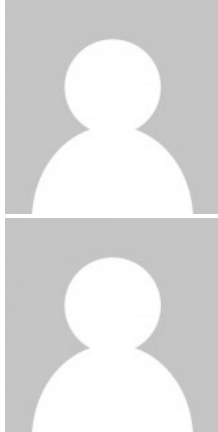
awwwwwwwwwwwwesome




icyblazek    在 2015 年 6 月 1 日的 下午 5:22

回复

整理得非常详细，深入，再一次深入了解

RunLoop






wangbq    在 2015 年 6 月 2 日的下午 2:27

回复

写的超棒! :oops: :oops: 有个小疑问, 文章中“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”, 如果我一直不获取 runloop的话, 这个线程就不能处理事件吗?






ibireme    在 2015 年 6 月 2 日的下午 9:17

回复

线程创建时有入口函数, 线程里能做的东西都是在那个函数里的。RunLoop也好你自己的逻辑也好, 都是在那个函数里完成的。你要是不用 RunLoop的话, 也可以自己实现一个类似的机制来处理你定义的事件。






wangbq    在 2015 年 6 月 8 日的下午 3:50

回复

理解了。还有一个问题哈, 就是UIButton点击事件打印堆栈看的话是从source0调出的, 文中说的是source1事件, 不知道哪个是正确的呢?



ibireme    在 2015 年 6 月 8 日的下午 9:14

回复

首先是由那个Source1 接收IOHIDEvent, 之后在回调
__IOHIDEventSystemClientQueueCallback() 内触发的 Source0, Source0 再触发的

`_UIApplicationHandleEventQueue()`。所以
`UIButton`事件看到是在 `Source0` 内的。你可以在
`__IOHIDEventSystemClientQueueCallback` 处下
一个 Symbolic Breakpoint 看一下。



ffl 🇯🇵 🌐 🍏 在 2015 年 7 月 17 日 的 下午 9:40

看孙源@sunnyxx的那个视频，他说UIEvent是属于source0，理由和上面这位同学是一样的，也是从过看log得出的结论。这样说来他说的是错的？



0xwangbo 🇨🇳 🌐 🍏 在 2018 年 12 月 13 日 的 下午 7:20

屏幕上就一个按钮，测试发现只要触摸屏幕，就会有 `__CFRunLoopDoSource1` 到 `__IOHIDEventSystemClientQueueCallback` 的流程，无论是否点击按钮，而点击按钮时，除了上述流程，会有一条新的调用从 `GSEventRunModal` 到 `CFRunLoopRunSpecific` 到 `__CFRunLoopDoSources0`，所以看起来按钮点击事件还是直接触发的 `source0`



0xwangbo 🇨🇳 🌐 🍏 在 2018 年 12 月 13 日 的 下午 7:33

(lldb) bt

* thread #6, name =

'com.apple.uikit.eventfetch-thread', stop

reason = breakpoint 2.1

* frame #0: 0x000000019d7793fc

IOKit__IOHIDEventSystemClientQueueCallback

frame #1: 0x000000019d452708

CoreFoundation__CFMachPortPerform + 192

frame #2: 0x000000019d47ad60

CoreFoundation__CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE1_PERFORM_FUNCTION__ + 60

frame #3: 0x000000019d47a468

CoreFoundation__CFRunLoopDoSource1 + 444

frame #4: 0x000000019d474ff8

CoreFoundation__CFRunLoopRun + 2088

frame #5: 0x000000019d4744b8

CoreFoundationCFRunLoopRunSpecific + 452

frame #6: 0x000000019de773e0 Foundation-[NSRunLoop(NSRunLoop)

runMode:beforeDate:] + 304

frame #7: 0x000000019de77284

Foundation-[NSRunLoop(NSRunLoop)runUntilDate:] + 152

frame #8: 0x00000001cb29b288 UIKitCore-[UIEventFetcher threadMain] + 140

frame #9: 0x000000019de75fc0

Foundation-[NSThread main] + 72

frame #10: 0x000000019dfb2c44

Foundation__NSThread__start__ + 1044
frame #11: 0x000000019d0ef974
libsystem_pthread.dylib_pthread_body +
132

frame #12: 0x000000019d0ef8d0
libsystem_pthread.dylib_pthread_start +
52

frame #13: 0x000000019d0f7ddc
libsystem_pthread.dylibthread_start + 4
(lldb) bt

* thread #6, name =
'com.apple.uikit.eventfetch-thread', stop
reason = breakpoint 5.1

* frame #0: 0x000000019d47a6a8

CoreFoundation__CFRunLoop_IS_CALLING_O
UT_TO_A_SOURCE0_PERFORM_FUNCTION__

frame #1: 0x000000019d47a640

CoreFoundation__CFRunLoopDoSource0 +
92

frame #2: 0x000000019d479ef8

CoreFoundation__CFRunLoopDoSources0 +
180

frame #3: 0x000000019d474bd8

CoreFoundation__CFRunLoopRun + 1032

frame #4: 0x000000019d4744b8

CoreFoundationCFRunLoopRunSpecific +
452

frame #5: 0x000000019de773e0

Foundation-[NSRunLoop(NSRunLoop)

```
runMode:beforeDate:] + 304
frame #6: 0x000000019de77284 Foundation-
[NSRunLoop(NSRunLoop) runUntilDate:]
+ 152
frame #7: 0x00000001cb29b288
UIKitCore-[UIEventFetcher threadMain] +
140
frame #8: 0x000000019de75fc0 Foundation-
[NSThread main] + 72
frame #9: 0x000000019dfb2c44
Foundation__NSThread__start__ + 1044
frame #10: 0x000000019d0ef974
libsystem_pthread.dylib_pthread_body +
132
frame #11: 0x000000019d0ef8d0
libsystem_pthread.dylib_pthread_start +
52
frame #12: 0x000000019d0f7ddc
libsystem_pthread.dylib`thread_start + 4
(lldb) bt
```

我理解偏了，应该说事件还是系统注册的那个 source1 接收，后续转给 source0 处理，博主说的没问题。






雷纯锋 🇨🇳 🌐 🍏 在 2015 年 6 月 3 日的下午 4:39

回复

文章写得非常好，赞一个。不过我有一点疑问想请教一下博主，就是 AutoreleasePool 小节中的

内容不知道博主是在哪里看到的呢？我在 CF 和 runtime 的源码中都没有找到哦。另外，如果可以的话，加一下我的 QQ 307213080 交个朋友哈。






ibireme    在 2015 年 6 月 3 日的下午 10:22

[回复](#)

那个是打印主线程RunLoop的状态，配合对象 dealloc处的断点看到的。Observer应该是 UIKit.framework添加的。






XiangqiTu    在 2015 年 6 月 3 日的下午

6:33

[回复](#)

写得好。






John    在 2015 年 6 月 5 日的上午 9:34

[回复](#)

你们现在开发中开始使用swift了没？roll:



只榆大叔    在 2015 年 6 月 8 日的上午 9:36

[回复](#)

nice

果然是深入理解呀。

不过对于初学者来说，有一个最重要的问题没有讲到：run loop为啥被设计成这样？

相信这个问题解释清楚了的话，看起来会更容易理解。



孙大龙 🇨🇳 🌐 🍏 在 2015 年 7 月 1 日的 下午 12:26

回复

```
while (!self.runLoopThreadDidFinishFlag) {  
    NSLog(@"Begin RunLoop");  
    [[NSRunLoop currentRunLoop]  
    runMode:NSDefaultRunLoopMode beforeDate:  
    [NSDate distantFuture]];  
    NSLog(@"End RunLoop");  
}
```

“ [[NSRunLoop currentRunLoop]
runMode:NSDefaultRunLoopMode beforeDate:
[NSDate distantFuture]];” 这句到底做什么？
是让主线程runloop进入休眠吗？不能理解为什么
加了这句，代码就不会往下执行了？



Veight Zhou 🇨🇳 🌐 🍏 在 2015 年 7 月 15 日的 下午

1:48

回复

『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。』



亮 🌐 🍏 在 2017 年 10 月 13 日的 下午 4:28

回复

运行runLoop 一次，阻塞当前线程以等待处理一次输入源。在处理了第一次到达的输入源或设定的beforeDate到时间后，runLoop 会 exit。



Oliver Hu 🇺🇸 🌐 🍏 在 2015 年 7 月 15 日的下午

3:15

回复

Saved my life. 感谢分享，如此详实的介绍的runloop的blog几乎就此一家了:)



ffl 🇯🇵 🌐 🍏 在 2015 年 7 月 16 日的下午 4:10

回复

查看source文件CFRunLoop.c，发现这儿有多处循环，一处是在方法void CFRunLoopRun(void)里面，然后一处是在static int32_t __CFRunLoopRun(CFRunLoopRef rl, CFRunLoopModeRef rlm, CFTimeInterval seconds, Boolean stopAfterHandle, CFRunLoopModeRef previousMode)里面，按照文章的解释，应该是在方法__CFRunLoopRun里循环的，那么外部这个循环是做什么用的？



ruikq 🇨🇳 🌐 🍏 在 2015 年 7 月 28 日的下午 2:00

回复

看懂一部分，有一个疑问，在SDWebImage源码中的DownloaderOperation中重写了NSOperation的start方法，当NSURLConnection start之后为什么要CFRunLoopRun()一下？这是让线程一直停留在runloop的那个循环里面么？如果是这个runloop的循环里面又包含哪些需要执行的东西

Kyle Sun 🇨🇳 🌐 🍏 在 2016 年 3 月 3 日的下午 3:07



我也正在看SDWebImage的源码，刚开始看这个地方有些看不懂，其实给你看一下

CFRunLoopRun()的源码就知道啥意思了：

```
void CFRunLoopRun(void) { /* DOES  
CALLOUT */  
int32_t result;  
do {  
result =  
CFRunLoopRunSpecific(CFRunLoopGetCurrent()  
(), kCFRunLoopDefaultMode, 1.0e10, false);  
CHECK_FOR_FORK();  
} while (kCFRunLoopRunStopped != result &&  
kCFRunLoopRunFinished != result);  
}
```

只要进入Run的循环之后只有Stop或者Finished的时候才会跳出循环，执行后面的代码，而是否Stop和Finish就是在NSURLConnection的delegate方法里面控制的。

回复



cli 🇺🇸 🌐 🌐 在 2015 年 8 月 4 日 的 上午 6:47

“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）。”

在哪里可以确认CADisplayLink的实现和NSTimer不一样？有源代码可以看吗？


liucien 🇨🇳 🌐 🍏 在 2015 年 8 月 17 日 的 下午 2:27



nice

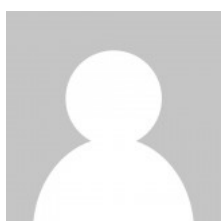
回复



任贵权    在 2015 年 8 月 20 日的 上午 10:17

回复

写的很好 但是还有一点我不是很疑惑
为什么我在主线程加上
`NSRunLoop.mainRunLoop().run()` 整个程序就卡死了






Ljson    在 2015 年 8 月 22 日的 下午 10:15

回复

过早的创建主运行循环并加到主线程当中，会卡主后面的操作，所以界面会卡主。我的猜想！





John咯    在 2016 年 3 月 3 日的 下午 4:19

回复

应该是主线程的RunLoop在程序刚启动时已经加锁启动进入循环了，而你之后再让主线程的RunLoop进入循环就会因为互斥锁而进入睡眠






iOS 小菜鸟    在 2016 年 5 月 26 日的 上午 11:45

回复

主线程进入循环了，这时候是在`do{}while ()`循环里，在这个循环里在调用`CFRunLoopRun ()`，应该是又进到一个循环里了。并且会一直停留在那个循环里。所以造成的现象就是主线程卡死了。
`runLoop`是可以嵌套的，因为他就是一个循环，循

环肯定是可以嵌套的。我个人的理解，不知道对不对啊。



罗德里格斯    在 2016 年 7 月 26 日的下午 6:18 [回复](#)

一个线程不是只有一个对应的RunLoop吗？怎么会有两个循环？这两个循环怎么理解？






nEar_Geo    在 2018 年 8 月 1 日的下午 6:05

Run loops can be run recursively. You can call `CFRunLoopRun` or `CFRunLoopRunInMode` from within any run loop callout and create nested run loop activations on the current thread's call stack.

这是CFRunLoop官方文档的一句话，它指出了一种创建嵌套的runloop的可能。但是这段话的前一段，文档明确指出了我们不能创建线程的runloop。

如果有人知道这一段所表达的意思，请回复我，我正在进行部分文档的翻译工作



yannmm    在 2019 年 3 月 28 日的下午 5:47 [回复](#)

如果暂停程序，堆栈会停留在
__CFRunLoopServiceMachPort，意味着 run

loop 正在监听 mach port 消息，这也是它休眠的原理。run 表示以 kCFRunLoopDefaultMode 运行 run loop，如果这个 mode 下的 source 和 timer 不做任何动作，那么 run loop 肯定会休眠。

尝试用 runMode:beforeDate: 替代 run，mode 填 UITrackingRunLoopMode，看看有什么不同？。

另外，run loop 支持嵌套调用，在 loop 中的某个任务执行环节重新调用 run 方法，会导致当前 run loop 运行环境被保存，和栈的原理很相似。



Klein_Mioke 🇨🇳 🌐 🍏 在 2015 年 9 月 7 日的 上午 11:58 [回复](#)

评论 :oops: 这篇超棒必须手动赞~~



huangzhong 🇨🇳 🌐 🍏 在 2015 年 9 月 9 日的 下午 4:21 [回复](#)

runloop 不是有几个mode嘛，如果以一种mode跑runloop，当想切换到另外一种mode的时候怎么切换呢？主线程怎么在几种mode间切换的呢？

G了个J 🇨🇳 🌐 🍏 在 2015 年 9 月 23 日的 上午 1:34 [回复](#)



必须要停止当前线程,再重新选择mode,再启动线程



huangzhong 🇨🇳 🌐 🍏 在 2015 年 10 月 8 日的下午 9:01

回复

那主线程怎么办呢



shayneyeorg 🇨🇳 🌐 🍏 在 2016 年 5 月 5 日的下午 4:14

回复

我觉得是不需要停止线程的，只是需要退出当前mode的Run Loop，再重新进入另一个mode的Run Loop。



Delpa 🇨🇳 🌐 🍏 在 2015 年 9 月 14 日的上午 10:36

回复

Runloop内部逻辑图，第7步的Source0(port), Source1(port)端口输入源



mario 🇨🇳 🌐 🍏 在 2015 年 10 月 14 日的下午 8:47

回复

:roll: 评论简直是太棒了，楼主我想给你生孩子！！！！






adobe 🇨🇳 🌐 🍏 在 2015 年 10 月 14 日的下午 9:57

回复

你好,博主,你用的是wordpress吧,主题很漂亮,可以



Nemo    在 2015 年 10 月 20 日 的 下午 2:45

回复

博主关于RunLoop应该是现在能搜到的中文资料里面最深入的，但是我在看关于RunLoop的时候，始终有几点没看明白，搜索了一些英文blog也没看到这几点的解释。

1.APP中 设备的触摸、网络请求到达、旋转等这些是属于source0的源还是基于port的source1的源呢，在stackflow上有一个回答

<http://stackoverflow.com/questions/22116698/does-uiapplication-sendevent-execute-in-a-nsrunloop>

里面提到，设备的触摸、旋转这些硬件的响应事件是属于source1 基于端口的源。如果是这样就和我第二个问题有发生矛盾的地方。

2.在RunLoop的一次loop循环中，每次循环开始首先处理source0的源，有source1就到第九步开始处理。

但是这里我有个疑问，一次loop是如何决定这次需要处理的timer和source源的，如果在处理source源或者timer的这次loop中不停地有新的timer和source源添加会怎么样？

在源码里面，每次loop循环和添加timer或者source源都会将RunLoop加锁，执行完之后再解锁。如果是这样，那每开始一次loop要处理的所有source源和timer都是固定的，新加入的source源和timer会等到下一次loop执行。

但是我新建了一个工程，在view的touchmoved方

法中每次执行的时候都加入了一个非常大的log字符串的循环，导致loop中的timer触发方法一直不被执行，只有停止触摸后，timer才正常打印。




所以这里就有两个矛盾的地方

1、如果触摸是source1的源，那按照loop循环执行顺序，第9步先执行timer，所以不应该存在timer不执行，那这说明触摸是source0的源？

2、如果一次loop要处理的source和timer是固定的，那不应该存在timer的方法不被执行的情况，所以timer这里不执行是否表明不停的新加source源的速度如果大于一次loop处理source源的速度，就会一直卡在处理source源的地方，导致timer方法不会被执行？

源码我能力有限，也只能看懂一点，结合文档、网上的一些文章和博主写的分析，始终也没能够想明白这中间的逻辑。



Delpa    在 2015 年 10 月 27 日 的 上午 11:22




回复

我在研究这块的时候也是遇到很多解释不了的，网上也很少资料，特别是RunLoop被唤醒后加锁这点，就有很多事件说不清。

触摸那个，我自己测试看到的情况是先Source1去接收事件，再执行Source0的回调去分发。

留个联系方式，我们交流一下吧



guinsoo    在 2015 年 11 月 15 日 的 下午 5:02

回复

因为如果timer之前执行了很长时间的任務，到

timer执行时还没执行完，则timer就会跳过。不会延迟执行。应该是你的touchmove持续触发。在同一个runloop循环中有太多的source1回调
__IOHIDEventSystemClientQueueCallback()触发的source0回调。所以导致期间的timer都被跳过了。



Xi Lin 🇨🇳 🌐 🍏 在 2015 年 12 月 13 日的 上午 1:43

回复

正如下面大家回复的，一次loop里肯定只会执行一类源，要么是source1要么是timer。而timer是设定好触发时机来执行的，文中定时器那段也讲了，如果执行时机正好被错过了那就是不会执行的。其实我不清楚你在touchmoved中做耗时的操作会使当前使用什么mode里，按说触摸事件没结束的话应该一直是在UITrackingRunLoopMode，那timer就更不会被触发了



song 🇨🇳 🌐 🌐 在 2015 年 11 月 13 日的 下午 4:55

回复

写的很棒！



飞流 🇺🇸 🌐 🌐 在 2015 年 11 月 16 日的 下午 8:39

回复

文章写得超赞，读了好多遍了，有一点不理解，RunLoop执行步骤中多次提到了执行加入到RunLoop的block，这个block指的是什么呀，方便解释一下嘛，谢谢



dibadalu 🇨🇳 🌐 🍏 在 2015 年 11 月 17 日的下午

5:48

[回复](#)

作为一个iOS开发初学者，我竟然看完了全文。虽然很多都不懂，但是看了最少也知道了runloop是个什么“东西”，以后要是遇到runloop的问题也知道怎么去寻找方案。 :smile:



ftxbird 🇨🇳 🌐 🍏 在 2015 年 11 月 18 日的下午 12:15

[回复](#)

评论 :smile: 看完一遍 受益匪浅!!!! 谢谢耀源兄



iOS小白菜 🇨🇳 🌐 🍏 在 2015 年 11 月 19 日的下午

11:12

[回复](#)

:cry: 看懂这些需要具备哪些知识呢，为啥我看着云里雾里的？ :???: :arrow:



不会反思的小花猫 🇨🇳 🌐 🍏 在 2015 年 11 月 27 日的下午 6:18

[回复](#)

博主，看了你的博文有以下个疑问，拜托给解答一下




1 ”__CFRunLoopDoBlocks(runloop, currentMode)“在runlooprun中一共执行了3次，其中前两次几乎并列写了两边,这是为什么？

2 __CFRunLoopDoBlocks 处理 加入runloop的代码块，跟 回调（例如

__CFRUNLOOP_IS_SERVICING_THE_MAIN_DIS

PATCH_QUEUE__) 有什么区别吗? 不都是处理 block 中代码吗? 跪求解答






Menzoda    在 2015 年 11 月 29 日的下午 10:02

回复




写的非常好, 干货, 赞.



韦振宁    在 2016 年 6 月 24 日的下午 5:51
同问

回复



大凌    在 2019 年 3 月 4 日的下午 4:56

回复

我的理解:

1. 在 RunLoop 中处理完 source0 和 source1 后才执行自己添加到 RunLoop 的 Block。
2. dispatch_async(dispatch_get_main_queue(), block) & __CFRunLoopPerformBlock 的区别是: 1>. 调用 GCD 后是可以在 common mode 中主动唤醒主线程 RunLoop 并在 __CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE__ 回调 block。而用 __CFRunLoopPerformBlock 添加到 runloop 的 block 中是不会主动唤醒主线程 RunLoop, 需要 CFRunLoopWakeUp 来唤醒, 才会在 __CFRunLoopDoBlock 中依次执行。2>. 在主线程的 RunLoop 中

dispatch_async(dispatch_get_main_queue(), block)是串行队列，会先执行完主线程的代码后才会执行block。而__CFRunLoopDoBlock则是在根据顺序执行。



Loe_Lin 🇨🇳 🌐 🍏 在 2015 年 12 月 5 日 的 下午 4:26

回复

膜拜一下 ;-)



brownfeng 🇨🇳 🌐 🍏 在 2015 年 12 月 7 日 的 下午

4:53

回复

评论 :oops: :oops: :oops: :oops: 太牛了!!!



John Wong 🌐 🍏 在 2015 年 12 月 10 日 的 上午

11:24

回复

RunLoop 内部的逻辑图第7步，唤醒的条件port-based input source，应该是source1吧？



xuyafei 🌐 🍏 在 2016 年 3 月 10 日 的 上午 8:35

回复

7. Put the thread to sleep until one of the following events occurs:




- An event arrives for a port-based input source.
- A timer fires.
- The timeout value set for the run loop

expires.

·The run loop is explicitly woken up.

“An event arrives for a port-based input source.” 我也觉得应该是source1吧？楼上@Delpan也有提。





Xi Lin    在 2015 年 12 月 13 日 的 上午 1:36

回复

好文手动点赞！

P.S.关于网络请求那的配图里CFSocket应该是调用Source0吧？






1   在 2015 年 12 月 14 日 的 下午 6:04

回复

timer不是通过GCD实现的，是通过MK_TIMER实现的





ibireme    在 2015 年 12 月 14 日 的 下午 9:58

回复

感谢指正，我又看了下 timer 部分的源码，GCD 和 mk_timer 都用到

了：<https://github.com/apple/swift-corelibs-foundation/blob/master/CoreFoundation%2FRunLoop.subproj%2FCFRunLoop.c#L1948-L1980>




1   在 2015 年 12 月 15 日 的 上午 10:58

回复



嗯，源代码里面是GCD和mk timer都有，实际上使用的还是mk timer，它是根据tolerate time参数去决定使用哪一种，我在实际例子中发现timer还是通过mac port调出的。应该是mk timer





ibireme    在 2015 年 12 月 15 日 的 下午 10:11

回复




回到家又仔细调了一下这块儿代码，我把RunLoop 相关 struct 挪到真机上跑起来调试，发现并没有用到USE_DISPATCH_SOURCE_FOR_TIMERS。那NSTimer 确实是由 mk_timer 驱动而不是 GCD 驱动的。感谢提示，我会更新一下文章。



1   在 2015 年 12 月 16 日 的 上午 10:16




runloop源代码可调试?怎么做的



ibireme    在 2015 年 12 月 16 日 的 下午 6:13

没有调试源码，只是把 struct 结构放到代码中，调试时就能直接看到对象的内部数据了。





亮子123    在 2017 年 10 月 16 日 的 下午 9:05

CFRunLoop.c 这段宏定义还不能说明问题吗?

———

```
#if DEPLOYMENT_TARGET_MACOSX
#define
USE_DISPATCH_SOURCE_FOR_TIMERS 1
#define USE_MK_TIMER_TOO 1
#else
#define
USE_DISPATCH_SOURCE_FOR_TIMERS 0
#define USE_MK_TIMER_TOO 1
#endif
```



9527   在 2015 年 12 月 15 日的 下午 9:40

回复

你好，我最近也看了RunLoop的源码，对
__CFRunLoopRun里关于timer的回调有点疑惑：

1.想问一下CFRunLoopTimer是不是由 GCD 的
dispatch_source_set_timer 实现并进行回调通知
CFRunLoop该执行timer回调了
(__CFRunLoopDoTimer)。

2.<https://github.com/apple/swift-corelibs-foundation/blob/master/CoreFoundation/RunLoop.subproj/CFRunLoop.c>。对do while 里面的那个关于睡眠的片段不是很理解

```
#if DEPLOYMENT_TARGET_MACOSX ||
DEPLOYMENT_TARGET_EMBEDDED ||
DEPLOYMENT_TARGET_EMBEDDED_MINI
#if USE_DISPATCH_SOURCE_FOR_TIMERS
do {
if (kCFUseCollectableAllocator) {
// objc_clear_stack(0);
//
```

```

memset(msg_buffer, 0, sizeof(msg_buffer));
}

msg = (mach_msg_header_t *)msg_buffer;

__CFRunLoopServiceMachPort(waitSet, &msg,
sizeof(msg_buffer), &livePort, poll ? 0 :
TIMEOUT_INFINITY, &voucherState,
&voucherCopy);

if (modeQueuePort != MACH_PORT_NULL &&
livePort == modeQueuePort) {
// Drain the internal queue. If one of the
callout blocks sets the timerFired flag, break
out and service the timer.
while
(_dispatch_runloop_root_queue_perform_4CF(r
lm->_queue));
if (rlm->_timerFired) {
// Leave livePort as the queue port, and
service timers below
rlm->_timerFired = false;
break;
} else {
if (msg && msg != (mach_msg_header_t
*)msg_buffer) free(msg);
}
} else {
// Go ahead and leave the inner loop.
break;
}

```

```
} while (1);
```

```
#else 。。。
```

想请教一下这一段是什么意思。所到底，就是还没有弄清楚 Runloop是如何实现timer的 :cry:
:cry:



ibireme 🇨🇳 🌐 🍏 在 2015 年 12 月 15 日 的 下午

10:12

回复

详情见这上面一条回复的讨论。



Va No 🇨🇳 🌐 🍏 在 2016 年 1 月 11 日 的 上午 9:41

回复

这是我见过,关于RunLoop写得最清楚,最准确(基于源码)的一篇博文! 期待博主的更多分享. 上 Donate二维码吧,我小支持下.....



weiyang 🇺🇸 🌐 🍏 在 2016 年 2 月 11 日 的 上午 11:24

回复

失效文档:

[http://www.fenestrated.net/mirrors/Apple%20Technotes%20\(As%20of%202002\)/tn/tn2028.html](http://www.fenestrated.net/mirrors/Apple%20Technotes%20(As%20of%202002)/tn/tn2028.html)

POSIX threads (pthreads) are layered on top of Mach threads.

Cocoa threads (NSThreads) are layered directly on top of pthreads.



Tiger 🇨🇳 🌐 🍏 在 2016 年 2 月 22 日的 下午 4:27

[回复](#)

文中说道:

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()。

但按钮的事件处理是停在

__CFRunLoopDoSources0, 是Sources0

而不是Source1

请问,问题出在哪里?



Tiger 🇨🇳 🌐 🍏 在 2016 年 2 月 22 日的 下午 5:06

[回复](#)

看到评论里有解答的,谢谢



小新 🇨🇳 🌐 🍏 在 2016 年 2 月 23 日的 上午 11:24

[回复](#)

当我突然发现这篇文章竟然是YYKit的作者写的时候, 我非常的震惊, 这人是有多牛X. 我最近在看YYKit(虽然目前还看不懂), 也有在看runTime和runLoop. 这篇文章简直太好了, 至少要看10遍, 收获非常大, 十分感激



孙大龙 🇨🇳 🌐 🍏 在 2016 年 2 月 26 日的 上午 10:58

[回复](#)

问博主:

```
While (!self.runLoopThreadDidFinishFlag) {
```

```
NSLog(@"Begin RunLoop");  
[[NSRunLoop currentRunLoop]  
runMode:NSDefaultRunLoopMode beforeDate:  
[NSDate distantFuture]];  
NSLog(@"End RunLoop");  
}
```

“ [[NSRunLoop currentRunLoop]
runMode:NSDefaultRunLoopMode beforeDate:
[NSDate distantFuture]];” 这句到底做什么？
是让主线程runloop进入休眠吗？ 不能理解为什么
加了这句，代码就不会往下执行了？



Chen略 🇨🇳 🌐 🍏 在 2016 年 3 月 3 日 的 下午 3:25

回复

在CF115.16中的__CFRunLoopRun函数中发现了以下代码：

```
#if USE_DISPATCH_SOURCE_FOR_TIMERS  
mach_port_name_t modeQueuePort =  
MACH_PORT_NULL;  
if (rlm->_queue) {  
modeQueuePort =  
_dispatch_runloop_root_queue_get_port_4CF(rlm->_queue);  
if (!modeQueuePort) {  
CRASH("Unable to get port for run loop mode  
queue (%d)", -1);  
}  
}  
#endif
```

看名字好像是可以使用Dispatch Source来实现
Timer



wyf 🇨🇳 🌐 🍏 在 2016 年 3 月 16 日 的 下午 10:25

[回复](#)

结合着源码讲解挺不错的。

<https://github.com/wuyunfeng/LightWeightRunLoop> 这个是我用BSD pipe 和 内核队列实现的RunLoop，同时实现了你分析的timer、urlconnection 等，还有很多有意思得事情，感兴趣的话可以看看



jackman 🇨🇳 🌐 🍏 在 2016 年 3 月 17 日 的 上午 11:27

[回复](#)

写得着实详尽，看了两遍了，依然有不懂的地方，准备再看第三遍 :shock:



Zsam 🇨🇳 🌐 🍏 在 2016 年 3 月 18 日 的 下午 1:16

[回复](#)

找到的最好地讲RunLoop的文章,我想问下博主,源码里有CHECK_FOR_FORK(),这是什么意思呢?是定义在哪个文件的呢?



leo 🇨🇳 🌐 🍏 在 2016 年 3 月 20 日 的 下午 5:16

[回复](#)

您好 在文章中您提到如果需要切换 Mode，只能退出 Loop，再重新指定一个 Mode 进入,请问调用什么样的API来退出Loop



wyf 🇨🇳 🌐 🍏 在 2016 年 3 月 21 日的 下午 8:45

回复

切换mode不需要退出RunLoop 哦，
<https://github.com/wuyunfeng/LightWeightRunLoop>



林栖谷隐 🇨🇳 🌐 📱 在 2016 年 4 月 6 日的 上午

回复

12:34

大神你好，多谢分享，看了收获很多，但一直有一个问题困扰我，也没有找到答案，希望帮忙解答一下，runloop的内部已经有一个死循环去处理loop，来不停获取消息并处理，为什么我们在实现自定义子线程runloop的时候还需要一个死循环去驱动runloop呢？



Andrew 🇨🇳 🌐 🍏 在 2016 年 10 月 14 日的 上午 11:17

回复

看上面的简化代码能看得出来，run 和 runInMode 都是调用带有循环的那个函数。run 的时候timeout参数是一个很小的数。在我们自己写线程的时候，如果调用的是[[NSRunLoop currentRunLoop] run]，那就需要在外边加个大循环，因为run是跑完一个loop，runloop内部的循环就退出了。但如果是调用的 runUntilDate 那就不用大循环

XVXVXXX 🇨🇳 🌐 🍏 在 2016 年 4 月 7 日的 下午

回复

7:17



评论 :shock: :shock: 写的真好，感谢分享，学习了



zzzzzzzz 🇨🇳 🌐 🍏 在 2016 年 11 月 20 日的 上午 11:20

回复

竟然在这里看到教练了



王杰 🇨🇳 🌐 🍏 在 2016 年 4 月 13 日的 上午 10:34

回复

博主，为什么我用dispatch_async来标记
tableView刷新完成 有时候不成功
[tableView reloadData];
dispatch_async(dispatch_get_main_queue(), ^{
NSLog(@"刷新完成");
});



Charvel 🌐 🍏 在 2016 年 6 月 5 日的 下午 11:22

回复




我的理解是 tableView reload data 仅仅只是标记
要去刷新 tableView。

然后 dispatch 那里，你又给主线程的 Runloop
发送了任务 NSLog 。

接下来，主线程将会做刷新 tableView 的操作，
同时，你的 NSLog 任务也在中间穿插执行了。

因此，NSLog 的执行，与 tableView reload 的顺序
并没有什么依赖，不一定谁先执行。






天天    在 2016 年 4 月 20 日的下午 12:49

回复

求助大神, 我一直有一些问题想不通:




1. 我用一个控制台程序, 在main函数中测试
[[NSRunLoop currentRunLoop] run];NSLog(@"
is running");, 发现没有打印, 我的理解是runloop
一直没有结束, 所以执行不到后面的语句. 可是就
是因为这个结果我产生了新的疑问.
2. 结合1中的理解, 在我的思维中, 肯定是先有主队
列和主线程, 然后才可以将运行runloop的任务加
入到主队列中的, 进而由主队列进行分发执行, 可
是问题来了: 例如在一个iOS程序中, 假设所有的任
务都在主线程中执行. 某一个时刻runloop在休眠
期,然后用户的点击触发了一个事件, 接着runloop
被唤醒, 开始响应用户的点击. 因为假设只有一个
主线程,事件的响应会被加入到主队列, 此时
runloop也正在主线程中执行, 并且不会结束, 主线
程中的任务是同步执行的. 可是runloop没有结束
的情况下, 主线程却还是可以处理接受主队列的任
务去用户的点击事件, 让我很疑惑, 是不是自己哪
里理解错了, 求大神指点.



Naituw    在 2016 年 5 月 30 日的下午 6:02

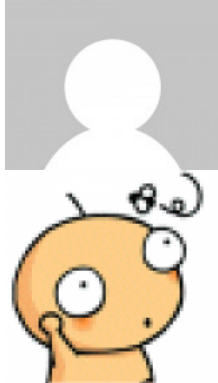
回复

主线程在休眠状态, 点击事件在其他线程唤醒主
线程

xX    在 2016 年 4 月 25 日的下午 2:15

回复

看后 受益匪浅。。



Doon 🇺🇸 🌐 🍏 在 2016 年 6 月 2 日的 上午 5:14

[回复](#)

难得见到写的这么深入的中文文章，非常不错。
iOS深入以后其实都是操作系统层级的东西。



FrogTan 🌐 🍏 在 2016 年 6 月 7 日的 下午 3:53

[回复](#)

写的漂亮



漫漫代码路 🇨🇳 🌐 🍏 在 2016 年 6 月 13 日的 下午

2:45

[回复](#)

:oops: :oops: :oops: 评论



taylor 🇨🇳 🌐 🍏 在 2016 年 6 月 14 日的 下午 2:48

[回复](#)

目前来说国内发表的runloop最深刻的文章，赞一个。 :oops:

纠正一个小不严谨，

/// 如果mode里没有source/timer/observer，直接返回。

这里看了官方文档与源代码，应该是如果没有input source 与 timer就会返回。

就算有observer也是会返回的。

源代码：

```
__CFRunLoopModelsEmpty
{
```

```
if (NULL != rlm->_sources0 && 0 _sources0))  
return false;  
if (NULL != rlm->_sources1 && 0 _sources1))  
return false;  
if (NULL != rlm->_timers && 0 _timers)) return  
false;  
}
```



dreampiggy 🇨🇳 🌐 🍏 在 2016 年 6 月 16 日的 下午
5:32 [回复](#)

非常感谢，写得非常相近，而且对整个RunLoop简介，伪代码实现，Darwin介绍，以及很多开源库的应用都介绍的很详细，真是可以当作教科书般的内容。对我这种菜鸟简直是太受用了。
:smile:



mark_stray 🇨🇳 🌐 🍏 在 2016 年 6 月 24 日的 下午
5:12 [回复](#)

评论 博主 膜拜啊！太牛掰，评论去的人也很牛掰！牛掰到 看了你的这篇文章之后我才知道我是新手？学习中 留个脚印



AntiMoron 🇨🇳 🌐 🍏 在 2016 年 7 月 14 日的 上午
11:32 [回复](#)

博主、我再问一个问题，怀疑是和run loop有关。
/**

* 添加快慢效果

*/

```
private func animateView(view: UIView, i: Int) {
    var option =
        UIViewAnimationOptions.CurveEaseInOut
    var duration = 0.5
    switch i {
    case 0:
        option = .CurveEaseIn
        break
    case 1:
        duration = 0.3
        option = .CurveLinear
        break
    case 2:
        duration = 0.3
        option = .CurveLinear
        break
    case 3:
        option = .CurveEaseOut
        break
    default:
        break
    }
    UIView.animateWithDuration(duration, delay: 0,
        options: option, animations: {
            view.transform =
                CGAffineTransformRotate(view.transform, -
                    CGFloat(M_PI) * 0.5)
        }) { (completed: Bool) in
```

```
self.animateView(view, i: (i + 1) % 4)
}
}
/**
 * 对loading做无限旋转动画
 */
private func animateView(view: UIView) {
    animateView(view, i: 0)
}
```

这样的代码，去做动画。当被运动的UIView变多的（10个，应该还好）时候。在模拟器里却越运行越卡。看他的内存使用没有增加，目测不是无限递归导致（毕竟是尾递归）。

想要分析性能问题出在哪里但是不知道从何入手。能支两招不？



宝强 🇨🇳 🌐 🍏 在 2016 年 7 月 30 日的 下午 9:29

回复

看懂这篇文章（或者说深入学习iOS，什么调用栈分析之类的）都需要哪些基础知识。麻烦列举一下，好为后面深入学习做准备。



kudocc 🇨🇳 🌐 🍏 在 2016 年 8 月 1 日的 下午 4:13

回复




刚刚在测试的时候看函数的调用栈发现手势识别 UITapGestureRecognizer和UIButton的事件都是在

source0(__CFRUNLOOP_IS_CALLING_OUT_TO

_A_SOURCE0_PERFORM_FUNCTION_)这个函数调用的。

我本人也觉得应该是系统通过message port来把事件传递过来的，而source0貌似只在进程内部来使用，有点困惑啊。

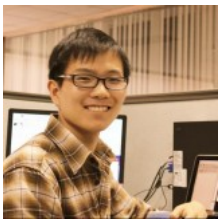


九二    在 2016 年 8 月 6 日的下午 3:52

[回复](#)

> 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。

这里应该是Timer依旧会被调用吧，不是同步了吗？



AiOSDeveloper    在 2016 年 8 月 16 日的下午 5:12

[回复](#)

Hi, 线程执行入口函数后，循环逻辑是不是“接受消息->处理->等待”，而不是“接受消息->等待->处理”？不好意思，可能没完全理解



iCodeWoods    在 2016 年 8 月 20 日的上午 10:38

[回复](#)

疑问：

每当 RunLoop 的内容发生变化时，RunLoop 都会自动将 _commonModelItems 里的 Source/Observer/Timer 同步到具有

“Common” 标记的所有Mode里。

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。




Timer不是在DefaultMode里吗？

DefaultMode不是”Common”的吗？

RunLoop不是会自动将_commonModelItems里的 Source/Observer/Timer同步到具有”Common”标记的所有Mode里吗？

那为什么Timer不会回调咧？难道此时 Timer 不在_commonModelItems里吗？



Reader    在 2016 年 8 月 23 日的 下午 4:54

回复

这个地方我也有疑问，“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 都已经被标记为”Common”属性。”为啥这两个“已经被标记为”Common”属性。”这里

UITrackingRunLoopMode 应该没有吧，否则 Timer 在 UITrackingRunLoopMode 下应该执行啊？期待大神解答



Reader 🇨🇳 🌐 🍏 在 2016 年 8 月 23 日的 下午 5:28

[回复](#)

明白了，新添加的 Timer 是在 DefaultMode 里，但不在 _commonModelItems 里，所以此时切换到 TrackingMode 时，虽然 _commonModelItems 里的状态都被同步了到各自 Mode，但新添加的 Timer 因为不在 _commonModelItems 中，所以不会回调



iCodeWoods 🇨🇳 🌐 🍏 在 2016 年 8 月 25 日的 下午 5:46

[回复](#)

你确定Timer不在_commonModelItems里吗？那 _commonModelItems放的都是哪些东西呢？



Gocy 🇨🇳 🌐 🍏 在 2016 年 9 月 3 日的 下午 5:41

[回复](#)

_commonModelItems放的是schedule在 NSRunLoopCommonModes里的事件源呀，如果你是schedule在DefaultMode中，自然就不在这里，Common这个概念对Mode有用，对Event也有用，只有Common Event+Common Mode才等于多种Mode下都可以跑






Gocy 🇨🇳 🌐 🍏 在 2016 年 9 月 3 日的 下午 5:34

[回复](#)

评论 :| 博主你好，关于你写在定时器部分的有关超时丢弃的部分，我写了一个test，我重写了一个View的drawRect，啥也不干就sleep 3秒，然

后在主线程中调用[view setNeedsDisplay], 然后立刻在mainRunLoop中schedule一个timeInterval < 3 的timer, 这个timer最后的表现是延迟执行, 和文中不符0 0, 求问可能原因? 另外对于一个Repeat的Timer, 如何判断其前几次该触发时的超时, 是被缓存延后调用, 还是直接被丢弃呢?






菜鸟一游    在 2016 年 9 月 5 日 的 上午 9:46

[回复](#)

评论 :smile: vSync和runloop之间的关系好像没有细讲...正常transaction的commit不是会在runloop exit和beforeWaiting吗, 那这和vSync似乎并没有关系, 而且有可能是矛盾的...这两部分是怎么联系起来呢?



向    在 2016 年 9 月 27 日 的 下午 5:59

[回复](#)

“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。”请问如何得知内部创建了一个Timer?



cookieLib    在 2016 年 11 月 3 日 的 下午

11:20

[回复](#)

有个问题, 为什么启动app的时候, 或者点击button的时候, 会调用多次beforeTimers,

beforsource。。。呢?



王飞    在 2016 年 11 月 18 日的 下午 8:06

回复

男神你好：你博客里这样写：“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 _commonModelItems 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 都已经被标记为“Common”属性”

我的疑问是_commonModelItems里面存放的是 commonModes的items吗？

如果是，为什么主线程的两个“common”标记的 mode没有同步呢？

如果不是，那存储的是什么呢？



凯旋 孙   在 2017 年 4 月 1 日的 下午 2:43

回复

DefaultMode有自己的items，TrackingMode也有自己的items，这两个都属于CommonModes，你给CommonModes添加的items就会同时添加给DefaultMode和TrackingMode，所以无论当前RunLoop是处于DefaultMode还是

TrackingMode, 就都会执行CommonMode的 items。实际上CommonModes并不是一个 mode, 他只是一个集合。



Cass 🇨🇳 🌐 🍏 在 2016 年 12 月 3 日的 上午 11:12

回复

“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。”

ibireme大大,这里添加一个Observer item 也可以让runloop中持续跑下去吗? 怎么我在一些文章中和代码试验中发现不会跑下去, 例如这里的代码:

<http://www.jianshu.com/users/c5db87501fe1/timeline>。



谭真 🇨🇳 🌐 🍏 在 2016 年 12 月 15 日的 上午 11:42

回复

来看第三遍~ :| 发现一个错别字: 我们在深入看一下 Darwin 这个核心的架构, 是再~ !:



丹丹 🇺🇸 🌐 🍏 在 2016 年 12 月 22 日的 下午 3:47

回复

学习受教了, 赞赞赞, 收藏了。



Kenshin 🇨🇳 🌐 🍏 在 2017 年 2 月 20 日 的 下午 6:47

[回复](#)

CADisplayLink和NSTimer应该都是
CFRunLoopTimer吧，从RunLoop看不到创建了
Source。



kevin 🇨🇳 🌐 🍏 在 2017 年 2 月 26 日 的 下午 2:56

[回复](#)

楼主RunLoop内部逻辑示例图第7步唤醒runloop的
应该是source1(port)



kevin 🇨🇳 🌐 🍏 在 2017 年 2 月 26 日 的 下午 2:57

[回复](#)

楼主RunLoop内部逻辑示例图第7步唤醒runloop的
应该是source1(port)



余泽锋 🇨🇳 🌐 🍏 在 2017 年 3 月 14 日 的 上午 9:57

[回复](#)

醍醐灌顶啊，大神能抽时间讲点线程的知识么？



君 🇨🇳 🌐 🍏 在 2017 年 3 月 20 日 的 下午 2:02

[回复](#)

有个问题，如果被source1唤醒需要第9步处理
source1。那么外部收不到处理这个source1的通
知？




Will 🇨🇳 🌐 📱 在 2017 年 3 月 30 日 的 上午 12:03

[回复](#)



在看af中创建线程的时为了让该线程的runloop不退出而创建了一个nsmachport对象，这里不能是一个timer source或者非port输入源吗？或者是添加一个observer啊？






陈伟鑫    在 2017 年 4 月 13 日 的 下午 2:45

回复

楼主，你好，我这边有一个疑问，看到你上面说的“如果需要切换 Mode，只能退出 Loop，再重新指定一个 Mode 进入”，这里为什么要退出 Loop？我测试tableView滑动打印堆栈，发现他里面是通过pushRunLoopMode将一个UITrackingRunLoopMode push进去，停止滑动的时候通过popRunLoopMode 返回让当前Mode切换回kCFRunLoopDefaultMode。希望可以解答一下。感谢。





刘元明    在 2017 年 5 月 6 日 的 上午 11:26

回复

大神，pthread_main_np()被写成了pthread_main_thread_np()



小莊   在 2017 年 6 月 9 日 的 下午 12:24

回复

ibireme 您好。关于RunLoop想请教您一些问题，望解惑。__CFRunLoopDoBlocks(rl, rlm) 这个方法是在执行被加入的block，应该怎么理解？Before waiting之前和After Waiting之后都有一次__CFRunLoopDoBlocks执行，After Waiting

这个在执行什么block? WeMobileDev公众号分享的“iOS 事件处理机制与图像渲染过程”一文中有关于标记UITableView reload 完成的一段代码, 这个reload的动作的执行时机怎么会在两个block之间呢?

```
dispatch_async(dispatch_get_main_queue(), ^{
    _isReloadDone = NO;
    [tableView reload];
    dispatch_async(dispatch_get_main_queue(), ^{
        _isReloadDone = YES;
    });
});
```



Kylin Roc 🇨🇳 🌐 🍏 在 2017 年 6 月 25 日的下午 9:54

回复

__CFRunLoopDoBlocks 函数执行的是被
CFRunLoopPerformBlock 函数加入的 blocks



luca 🇨🇳 🌐 🍏 在 2017 年 6 月 11 日的上午 11:41

回复

在CFRunLoop对外暴露管理Mode的接口这里

CFRunLoopRunInMode这个方法, 返回的是
CFRunLoopRunResult。

实际上没有CFStringRef这个参数类型, 并不需要
传入modeName。根据文档介绍: Runs the
current thread's CFRunLoop object in a
particular mode.

需要传入的是CFRunLoopMode, CFTimeInterval和Boolean这三个类型。



蠢蛋 🇨🇳 🌐 🍏 在 2017 年 6 月 11 日 的 下午 12:32

回复

mode

The run loop mode to run. mode can be any arbitrary CFString. You do not need to explicitly create a run loop mode, although a run loop mode needs to contain at least one source or timer to run.



Kylin Roc 🇨🇳 🌐 🍏 在 2017 年 6 月 25 日 的 下午

9:56

回复

```
typedef CFStringRef CFRunLoopMode  
CF_EXTENSIBLE_STRING_ENUM;
```



王 🇨🇳 🌐 🍏 在 2018 年 2 月 7 日 的 下午 6:04

回复

runloop内部逻辑的配图，左边的标注是source0，按照官方文档，应该是基于port的事件，那就应该是source1呀，是不是？



paul 🇨🇳 🌐 🍏 在 2018 年 3 月 9 日 的 上午 10:48

回复

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()。

这里是不是写错了，应该是source0吧？



paul 🇨🇳 🌐 🍏 在 2018 年 3 月 9 日的 上午 10:51

回复

附断点堆栈：

```
frame #22: 0x00000001887ec350 UIKit-
[UICollectionView
touchesEnded:withEvent:] + 540
frame #23: 0x00000001887592a4
UIKitforwardTouchMethod + 336
frame #24: 0x00000001887a793c UIKit-
[UIResponder
touchesEnded:withEvent:] + 60
frame #25: 0x00000001887592a4
UIKitforwardTouchMethod + 336
frame #26: 0x00000001887a793c UIKit-
[UIResponder
touchesEnded:withEvent:] + 60
frame #27: 0x00000001885a4294
UIKit_UIGestureRecognizerUpdate + 8988
frame #28: 0x00000001885e4820 UIKit-
[UIWindow _sendGesturesForEvent:] +
1132
frame #29: 0x00000001885e3e1c UIKit-
[UIWindow sendEvent:] + 764
frame #30: 0x0000000101ef51c0 LIKE-
[HiidoSDKCht sendEvent1:]
(self=0x00000001257e28a0, _cmd=,
```

```
event=) at HiidoSDKCht.m:121 [opt]
* frame #31: 0x00000001885b44cc
UIKit-[UIApplication sendEvent:] + 248
frame #32: 0x00000001885b2794
UIKit_UIApplicationHandleEventQueue +
5528
frame #33: 0x0000000183878efc
CoreFoundation__CFRunLoop_IS_CALLING
_OUT_TO_A_SOURCE0_PERFORM_FUNCTION
__ + 24
frame #34: 0x0000000183878990
CoreFoundation__CFRunLoopDoSources0 +
540
frame #35: 0x0000000183876690
CoreFoundation__CFRunLoopRun + 724
frame #36: 0x00000001837a5680
CoreFoundation`CFRunLoopRunSpecific + 384
```



谢晨 🇨🇳 🌐 🍏 在 2018 年 3 月 24 日的 上午 12:47

回复

我设置了

__IOHIDEventSystemClientQueueCallback 符号断点, 打印当前线程, {number = 2, name = com.apple.uikit.eventfetch-thread} 好像并不是主线程, 这里该如何理解?

jjdd 🇨🇳 🌐 🍏 在 2018 年 4 月 10 日的 下午 4:09

回复

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。

好像是source0不是timer



Jeff 🇨🇳 🌐 🍏 在 2018 年 5 月 2 日的 上午 11:19

[回复](#)

__Source0DidDispatchPortLastTime 这个是不是错了？应该是
__Source1DidDispatchPortLastTime



JayWang 🇨🇳 🌐 🍏 在 2018 年 7 月 20 日的 下午 6:05

[回复](#)

膜拜下大神，看第二遍了，还是有很多地方看不懂，当行不够。继续看吧，希望看第三遍的时候，可以豁然开朗。



cr02y 🇺🇸 🌐 🍏 在 2018 年 10 月 23 日的 下午 10:34

[回复](#)

文章中说到在beforewaiting和afterwaiting之间的这段时间线程会休眠，但是我在子线程b中用时间间隔为1ms的定时器去检查子线程a的isExecuting时，发现这个值一直是yes，是我的测试代码写得有问题吗？

另外，我在测试中发现runloop的循环周期时间长度是不固定的。这是不是由于任务量少的时候，

runloop循环执行变快。反之，变慢（甚至有可能等于一个上限）。这个上限具体是多少呢？



Mister_Leo    在 2018 年 10 月 25 日的上午 11:26

回复

内部处理逻辑写翻译的有问题啊，原文是这样的

The Run Loop Sequence of Events

Each time you run it, your thread's run loop processes pending events and generates notifications for any attached observers. The order in which it does this is very specific and is as follows:

Notify observers that the run loop has been entered.

Notify observers that any ready timers are about to fire.

Notify observers that any input sources that are not port based are about to fire.

Fire any non-port-based input sources that are ready to fire.

If a port-based input source is ready and waiting to fire, process the event immediately.

Go to step 9.

Notify observers that the thread is about to sleep.

Put the thread to sleep until one of the following events occurs:

An event arrives for a port-based input

source.

A timer fires.

The timeout value set for the run loop expires.

The run loop is explicitly woken up.

Notify observers that the thread just woke up.

Process the pending event.

If a user-defined timer fired, process the timer event and restart the loop. Go to step 2.

If an input source fired, deliver the event.

If the run loop was explicitly woken up but has not yet timed out, restart the loop. Go to step 2.

Notify observers that the run loop has exited.



kysonzhu 🇨🇳 🌐 🍏 在 2018 年 11 月 22 日的下午 11:43

回复

麻烦问一下，如何调试cf代码



Mattttt 🇨🇳 🌐 🍏 在 2019 年 3 月 2 日的下午 6:25

回复

ibireme 我很崇拜你啊



Mattttt 🇨🇳 🌐 🍏 在 2019 年 3 月 26 日的下午 5:46

回复

就目前来看，其实应该是这样的：

事件的响应的根源：事件的来源是 runloop 注册了一个基于 mach port 的 source1。当发生一个

硬件事件时，会生成 IOHIDEvent 对象并注册一个 source0

当 Runloop 被唤醒后处理 Source0，此时回调

__handleHIDEventFetcherDrain() 再转调

__handleEventQueueInternal() 到

__dispatchPreprocessedEventFromEventQueue

，来处理并包装成 UIEvent 进行处理和分发。

对于手势识别：当上面的事件进行到 UIWindow

时，window 识别了一个手势，于是向

UIGestureRecognizer 分派该 UIEvent，

UIGestureRecognizer 将找到合适的手势对象并发送 Action。

而在之后的一系列相关的 Event 都将被侦测并转而调用 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。






@许还真 🌈 🍏 在 2019 年 5 月 7 日的 下午 11:15

回复

挺喜欢作者这种风格，从底层（甚至到驱动层，描述掉帧那篇文章，vsync,hsync同步信号）到上

层（应用层）讲了一遍，自底向上，非常棒！学习！






johnXia    在 2019 年 7 月 13 日的 下午 8:14

[回复](#)

runloop可以阻塞住代码运行，有没有同学玩用过.....我出现个问题，弹出的alert点击不能取消






hhb    在 2019 年 7 月 19 日的 上午 10:29

[回复](#)

IOHIDEvent HID的含义能说说吗？



yumo zhu    在 2020 年 2 月 2 日的 下午

[回复](#)

11:08

首先表示感谢，看了两晚上，再艰难地阅读了下源码后，终于大概了解了个七七八八。

另外我这有个疑问，观察

kCFRunLoopBeforeTimers有实际的应用场景吗？因为在我的理解中，接收到这个通知貌似除了意味着新的一个循环开始执行了，不能代表其他任何事，甚至都不一定存在timer，只是因为不停有source事件插入于是不停地在循环，于是就会进行多次kCFRunLoopBeforeTimers的回调，那这种回调有什么意义？



引用/广播

1. [WWDC 15看点汇总 – iOS移动开发周报 – 剑客|关注科技互联网](#)  [W](#)  – [...] 《深入理解RunLoop》：iOS 开发中对RunLoop 和 Thread 的概念的理解和使用往往是区分开发者层次的重要部分。这篇文章中从基础开始详细介绍了 Runloop 的种种，很值得学习。 [...]
2. [\[转载\]《招聘一个靠谱的iOS》面试题参考答案（下） – See You Again](#)  [W](#)  – [...] 《深入理解RunLoop》 [...]
3. [招聘一个靠谱的iOS程序员面试题答案部分【下】](#) |  [W](#)  – [...] 《深入理解RunLoop》 [...]
4. [深度解析iOS应用程序的生命周期 | jwzhangjie](#)  [W](#)  – [...] 深入理解RunLoop [...]
5. [iOS 保持界面流畅的技巧 | Garan no dou](#)  [W](#)  – [...] Runloop 还不太了解，可以看一下我之前的文章 深入理解RunLoop，里面对 ASDK [...]
6. [iOS 保持界面流畅的技巧 | 青岛诺动信息科技有限公司](#)  [W](#)  – [...] Runloop 还不太了解，可以看一下我之前的文章深入理解RunLoop，里面对 ASDK [...]
7. [深入理解RunLoop](#)  [W](#)  – [...] 的概念RunLoop 与线程的关系RunLoop 对外的接口RunLoop 的 ModeRunLoop 的内部逻辑RunLoop 的底层实现苹果用 [...]
8. [如何让iOS 保持界面流畅？这些技巧你知道吗](#)  [W](#)  – [...] Runloop 还不太了解，可以看一下我之前的文章 深入理解RunLoop，里面对 ASDK [...]
9. [深入理解RunLoop\(转\) | Hello World.](#)  [W](#)  – [...] 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 [...]
10. [iOS 保持界面流畅的技巧（转） | Hello World.](#)  [W](#)  – [...] Runloop 还不太了解，可以看一下我之前的文章 深入理解RunLoop，里面对 ASDK [...]
11. [芒果iOS开发之高级面试题二-IT大道](#)  [W](#)  – [...] 《深入理解RunLoop》 [...]

12. [iOS 保持界面流畅的技巧\(最全最详尽的了\)-IT大道](#)  [W](#)  - [...] 深入理解RunLoop, 里面对 ASDK 也有所提及。 [...]
13. [深入理解RunLoop-IT大道](#)  [W](#)  - [...] 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 [...]
14. [iOS 保持界面流畅的技巧-IT大道](#)  [W](#)  - [...] Runloop 还不太了解, 可以看一下我之前的文章 深入理解RunLoop, 里面对 ASDK [...]
15. [互联网上iOS学习系列博文 \(收集\) -IT大道](#)  [W](#)  - [...] 深入理解RunLoop | Garan no dou [...]
16. [RunLoop深度探究 \(三\) -IT大道](#)  [W](#)  - [...] <http://blog.ibireme.com/2015/05/18/runloop/#more-41710> [...]
17. [RunLoop深度探究 \(二\) -IT大道](#)  [W](#)  - [...] <http://blog.ibireme.com/2015/05/18/runloop/#more-41710> [...]
18. [从代码中认识RunLoop-IT大道](#)  [W](#)  - [...] 现在网上关于RunLoop的资料真是太多了, 而且大同小异, 如果只是看一遍不在代码里面实现一下的话, 也只能了解点皮毛, 当然这样动笔写一些, 更能加深印象。这次学习笔记参考自: 链接1链接2链接3
本文代码都可以下载demo调试或者自己编写测试 [...]
19. [iOS知识树, 知识目录 \(包括对象、Block、消息转发、GCD、运行时、runloop、动画、Push、KVO、tableview, UIViewController、提交AppStore\) - 移动开发 - 阿里欧歌](#)  [W](#)  - [...] | -> CFRunLoop **【ibireme出品 点击】**
| [...]
20. [Toll-free bridging - IT大道](#)  [W](#)  - [...] <http://blog.ibireme.com/2015/05/18/runloop/> [...]
21. [iOS知识树, 知识目录 \(包括对象、Block、消息转发、GCD、运行时、runloop、动画、Push、KVO、tableview, UIViewController、提交AppStore\) - IT大道](#)  [W](#)  - [...]

...



22. [深入理解RunLoop – IT大道](#)  [W](#)  – [...] [阅读原文](#) [...]


23. [Alex博客笔记 | 移动开发技能点收集整理\(长期更新2016.3.14\)](#)  [W](#)  – [...] [深入理解RunLoop](#) [...]



24. [让界面更加流畅的几个技巧 – IT大道](#)  [W](#)  – [...]

RunLoop 还不太了解，可以看一下我之前的文章[深入理解RunLoop](#)，里面对 ASDK [...]


25. [从NSTimer的失效性谈起（一）：关于NSTimer和NSRunLoop – 移动开发 – 阿里欧歌](#)  [W](#)  – [...] 关于NSRunLoop的进一步探究，可以参考：[Run Loops](#)，[NSRunLoop Internals](#)，[CFRunLoop.c](#)，[深入理解RunLoop](#)。 [...]

26. [从NSTimer的失效性谈起（一）：关于NSTimer和NSRunLoop – IT大道](#)  [W](#)  – [...] 关于NSRunLoop的进一步探究，可以参考：[Run Loops](#)，[NSRunLoop Internals](#)，[CFRunLoop.c](#)，[深入理解RunLoop](#)。 [...]

27. [深入理解RunLoop | W](#)  – [...] 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 [...]

28. [61.iOS 保持界面流畅的技巧 – IT大道](#)  [W](#)  – [...] [深入理解RunLoop](#)，里面对 ASDK 也有所提及。 [...]

29. [初次接触 RunLoop – IT大道](#)  [W](#)  – [...] [深入理解RunLoop](#) [...]

30. [iOS Timer 盘点 | 神刀安全网](#)  [W](#)  – [...] <http://blog.ibireme.com/2015/05/18/runloop/> [...]

31. [让 BAT 的 Offer 不再难拿 — developerWorks-同行快讯网](#)  – [...] [RunLoop](#) [...]

32. [problems part.2 – Startor](#)  [W](#)  – [...] 《[深入理解RunLoop](#)》 [...]

33. [iOS 保持界面流畅的技巧 – 编程语言 – 阿里欧歌](#)  [W](#)  –

[...] Runloop 还不太了解，可以看一下我之前的文章深入理解RunLoop，里面对 ASDK [...]

34. [\[iOS\] RunLoop 学习 | 技术学习小组](#) 🇨🇳 W ? – [...] ibireme 的博客 [...]

35. [让 BAT 的 Offer 不再难拿 – kyoucr博客](#) 🇺🇸 W ? – [...] RunLoop [...]

36. [三月份实习校招后，也该为 BAT 秋招囤一些干货了 | 老豆比IT站](#) 🇨🇳 W ? – [...] Runloop 基本概念，猜想一下内部是怎么实现的 – 参考链接 [...]

37. [iOS中autorelease的那些事儿 | 大众阅读](#) 🇨🇳 W ? – [...] 深入理解RunLoop [...]

38. [小笨狼漫谈多线程：NSThread – 恰同学少年-不称](#) 🇨🇳 W ? – [...] runloop相关知识可以阅读ibireme的深入理解RunLoop [...]

39. [iOS 事件处理机制与图像渲染过程 – 恰同学少年-不称](#) 🇨🇳 W ? – [...] 深入理解runloop
(<http://blog.ibireme.com/2015/05/18/runloop/>) [...]

40. [如何让iOS 保持界面流畅？这些技巧你知道吗 – 恰同学少年-不称](#) 🇨🇳 W ? – [...] Runloop 还不太了解，可以看一下我之前的文章 深入理解RunLoop，里面对 ASDK [...]

41. [《招聘一个靠谱的iOS》面试题参考答案（下） – 恰同学少年-不称](#) 🇨🇳 W ? – [...] 《深入理解RunLoop》 [...]

42. [iOS应用程序的生命周期 – 恰同学少年-不称](#) 🇨🇳 W ? – [...] Programming Guide for iOSDeveloping iOS 7 App for iPhone and iPad深入理解RunLoopObjective-C Autorelease Pool [...]

43. [从实践谈iOS生命周期-IT技术](#) 🇨🇳 W ? – [...] 深入理解RunLoop [...]

44. [多线程：NSThread-IT技术](#) 🇨🇳 W ? – [...] runloop相关知识可以阅读ibireme的 深入理解RunLoop [...]



45. [小笨狼漫谈多线程：NSThread-IT技术](#) 🇨🇳 W ? – [...]

runloop相关知识可以阅读ibireme的 深入理解RunLoop [...]



46. [其实 iOS 多线程，都在这里了-IT技术](#)  [W](#)  - [...]

RunLoop 来处理消息，RunLoop 相关的知识可以去看 @ibireme 大神的文章《深入理解 RunLoop》。主线程负责处理 App 的 UI [...]

47. [初识 Run Loop-IT技术](#)  [W](#)  - [...] 深入理解RunLoop | Garan no dou [...]


48. [初识 Run Loop | 神刀安全网](#)  [W](#)  - [...] 深入理解 RunLoop | Garan no dou [...]

49. [试答卓同学的 iOS 面试题 | 神刀安全网](#)  [W](#)  - [...] 深挖请去看此文 深入理解RunLoop [...]



50. [阅读文章记录 - Take a Seat](#)  [W](#)  - [...] 深入理解 RunLoop [...]

51. [再看CVE-2016-1757-浅析mach message的使用 | 神刀安全网](#)  [W](#)  - [...] 图片转自

(<http://blog.ibireme.com/2015/05/18/runloop/>)。 [...]

52. [再看CVE-2016-1757-浅析mach message的使用 - 莹莹之色](#)  [W](#)  - [...] 图片转自



(<http://blog.ibireme.com/2015/05/18/runloop/>)。 [...]

53. [深入研究 Runloop 与线程保活 | 神刀安全网](#)  [W](#)  - [...] 深入理解 RunLoop [...]

54. [iOS网址大全 - ShawnDu的博客](#)  [W](#)  - [...] 深入理解 runloop [...]

55. [iOS网址大全 | ShawnDu博客](#)  [W](#)  - [...] 深入理解 runloop [...]

56. [iOS 网址大全 开发资源集合 - Shawn Du's Blog](#)  [W](#)  - [...] 深入理解runloop [...]

57. [iOS 保持界面流畅的技巧 | Codeba](#)  [W](#)  - [...] Runloop 还不太了解，可以看一下我之前的文章深入理解RunLoop，里面对 ASDK [...]

58. [iOS 保持界面流畅的技巧 | 梦幻泡影--孔德志个人博客](#) 

W ? - [...] Runloop 还不太了解，可以看一下我之前的文章深入理解RunLoop，里面对 ASDK [...]

59. [关于 iOS 异步开发的知识总结 - \(Objective-C & Swift 双语\) - 莹莹之色](#) 🇺🇸 W ? - [...] Runloop 来处理消息，

RunLoop 相关的知识可以去看 @ibireme 大神的文章《深入理解 RunLoop》是我看到最好的介绍 Runloop 相关知识的文章。主线程负责处理 App 的 UI [...]

60. [爱代码 > iOS开发 - RunLoop的基本概念与运用](#) 🇨🇳 W ? - [...] 【<http://blog.ibireme.com/2015/05/18/runloop/>】 [...]

61. [iOS开发 | Codeba](#) 🇯🇵 W ? - [...]

【<http://blog.ibireme.com/2015/05/18/runloop/>】 [...]

62. [从源码看runLoop | 恰同学少年-不称](#) 🇨🇳 W ? - [...] 深入理解RunLoop [...]

63. [iOS多线程-彻底学会多线程之『RunLoop』 | 神刀安全网](#)

🇯🇵 W ? - [...] 先来看一张表示这5个类的关系图（来源：

<http://blog.ibireme.com/2015/05/18/runloop/>）。 [...]

64. [使用 ASDK 性能调优 - 提升 iOS 界面的渲染性能 | 恰同学少年-不称](#) 🇨🇳 W ? - [...] 深入理解 RunLoop [...]

65. [RunLoop 总结：RunLoop的应用场景（二） - 范东](#) 🇨🇳 W ? - [...] 深入理解RunLoop(不要看到右边滚动条很长，其实文章占篇幅2/5左右，下面有很多的评论，可见这篇文章的火热) [...]

66. [RunLoop 总结：RunLoop的应用场景（二） | 恰同学少年-不称](#) 🇨🇳 W ? - [...] 深入理解RunLoop(不要看到右边滚动条很长，其实文章占篇幅2/5左右，下面有很多的评论，可见这篇文章的火热) [...]

67. [从源码看runLoop - 项目经验积累与分享](#) 🇨🇳 W ? - [...] 深入理解RunLoop [...]

68. [iOS多线程-彻底学会多线程之『RunLoop』 - 项目经验积累与分享](#) 🇨🇳 W ? - [...] 先来看一张表示这5个类的关系图（来源：<http://blog.ibireme.com/2015/05/18/runloop/>）。 [...]

69. [iOS 开发 - 深入理解 NSTimer 为什么要配合 NSRunLoop](#)

进行使用 — 项目经验积累与分享 🇨🇳 W ? — [...] RunLoop的基本概念与例子分析##深入理解RunLoop##5.另外附上本片博客所使用的 demo 的 github 地址 [...]

70. runLoop的终极大杀器 — 项目经验积累与分享 🇨🇳 W ? — [...] 参考: 一个iOS菜菜的白话文记录YY大神百度孙源的runLoop视频 [...]

71. 获取任意线程调用栈的那些事 — 项目经验积累与分享 🇨🇳 W ? — [...] 深入理解RunLoop [...]

72. iOS 常见知识点 (二) : RunLoop — 项目经验积累与分享 🇨🇳 W ? — [...] ibireme 的 深入理解RunLoop [...]

73. 深入研究 Runloop 与线程保活 — 项目经验积累与分享 🇨🇳 W ? — [...] 深入理解 RunLoop [...]

74. iOS 精品博客资源 (持续完善) — 项目经验积累与分享 🇨🇳 W ? — [...] 1、深入理解RunLoop [...]

75. [进阶] 关于 iOS 多线程, 都在这里了 — 项目经验积累与分享 🇨🇳 W ? — [...] Runloop 来处理消息, Runloop 相关的知识可以去看 @ibireme 大神的文章《深入理解 RunLoop》是我看到最好的介绍 Runloop 相关知识的文章。主线程负责处理 App 的 UI [...]

76. iOS RunLoop进阶 — 项目经验积累与分享 🇨🇳 W ? — [...] 1.http://blog.ibireme.com/2015/05/18/runloop/2.http://iphonedevwiki.net/index.php/IOHIDFamily3.http://www.dreamingwish.com/article/ios-multithread-program-runloop-the.html [...]

77. iOS RunLoop 学习笔记 — 项目经验积累与分享 🇨🇳 W ? — [...] RunLoop 时所作的一些笔记, 主要来源于 YYKit 作者的这篇文章, 感谢 ibireme [...]

78. iOS里的内存泄露 — 项目经验积累与分享 🇨🇳 W ? — [...] 我们能看到断点1和断点2 runloop还是在执行的, 断点3表示runloop一个迭代已经结束了, 即将进入睡眠。这里如果对runloop不了解的话可以看ibireme的这篇深入理解RunLoop。我

截取里面的一小段，大家可以看一下。 [...]

79. [初次接触RunLoop — 项目经验积累与分享](#) 🇨🇳 W ? – [...]

深入理解RunLoop [...]

80. [从代码中认识RunLoop — 项目经验积累与分享](#) 🇨🇳 W ? – [...]

现在网上关于RunLoop的资料真是太多了，而且大同小异，如果只是看一遍不在代码里面实现一下的话，也只能了解点皮毛，当然这样动笔写一些，更能加深印象。这次学习笔记参考自：链接1链接2链接3本文代码都可以下载demo调试或者自己编写测试 [...]

81. [小笨狼漫谈多线程：NSThread — 项目经验积累与分享](#) 🇨🇳 W ? – [...]

RunLoop相关知识可以阅读ibireme的深入理解RunLoop [...]

82. [关于OC运行时的干货集合 — 项目经验积累与分享](#) 🇨🇳 W ? – [...]

深入理解RunLoop [...]

83. [iOS应用程序的生命周期 — 项目经验积累与分享](#) 🇨🇳 W ? – [...]

Programming Guide for iOSDeveloping iOS 7 App for iPhone and iPad深入理解RunLoopObjective-C Autorelease Pool [...]

84. [看 CFRunLoop源码深入理解 RunLoop — Shawn Du's Blog](#) 🇨🇳 W ? – [...]

深入理解RunLoop [...]

85. [RunLoop 总结：RunLoop的应用场景（二） | BkCoding](#) 🇨🇳 W ? – [...]

深入理解RunLoop(不要看到右边滚动条很长，其实文章占篇幅2/5左右，下面有很多的评论，可见这篇文章的火热) [...]

86. [使用 ASDK 性能调优 — 提升 iOS 界面的渲染性能 | BkCoding](#) 🇨🇳 W ? – [...]

深入理解 RunLoop [...]

87. [从源码看RunLoop | BkCoding](#) 🇨🇳 W ? – [...]

深入理解RunLoop [...]

88. [深入理解RunLoop — Chen's Blog](#) W ? – [...]

深入理解RunLoop [...]


89. [RunLoop学习笔记-IT文库](#) 🇨🇳 W ? – [...]


深入理解RunLoop [...]

RunLoop [...]

90. [基于runloop的线程保活、销毁与通信-IT文库](#)  W ? - [...]


关于RunLoop，还有一些更深层次拓展性的内容，包括与gcd的协同关系、AutoreleasePool的释放时机、GCDTimer和NSTimer区别等等，本来是想写一写的，奈何篇幅已经很大了，实在写不动了，推荐感兴趣的可以看看YY大神这篇：深入理解RunLoop [...]


91. [RunLoop总结：RunLoop的应用场景（三）滚动视图流畅性优化-IT文库](#)  W ? - [...] 深入理解RunLoop(不要看到右边滚动条很长，其实文章占篇幅2/5左右，下面有很多的评论，可见这篇文章的火热) [...]

92. [获取任意线程调用栈的那些事-IT文库](#)  W ? - [...] 深入理解RunLoop [...]


93. [iOS应用程序的生命周期-IT文库](#)  W ? - [...]

Programming Guide for iOSDeveloping iOS 7 App for iPhone and iPad深入理解RunLoopObjective-C Autorelease Pool [...]


94. [深入研究 Runloop 与线程保活-IT文库](#)  W ? - [...] 深入理解 RunLoop [...]

95. [RunLoop 总结：RunLoop的应用场景（二）让Timer正常运转-IT文库](#)  W ? - [...] 深入理解RunLoop(不要看到右边滚动条很长，其实文章占篇幅2/5左右，下面有很多的评论，可见这篇文章的火热) [...]

96. [逃不出的圈子 -- RunLoop - 莹莹之色](#)  W ? - [...] 深入理解RunLoop [...]

97. [看 CFRRunLoop源码深入理解 RunLoop-IT文库](#)  W ? - [...] 深入理解RunLoop [...]


98. [深入理解RunLoop-IT文库](#)  W ? - [...] 深入理解RunLoop [...]


99. [iOS 保持界面流畅的技巧-IT文库](#)  W ? - [...] Runloop 还不太了解，可以看一下我之前的文章 深入理解RunLoop，里面


对 ASDK [...]

100. [iOS 网址大全 开发资源集合-IT文库](#)  [W ?](#) - [...] 深入理解RunLoop [...]

101. [深入理解RunLoop-IT文库](#)  [W ?](#) - [...] 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 [...]

102. [逃不出的圈子 — RunLoop-IT文库](#)  [W ?](#) - [...] 深入理解RunLoop [...]

103. [iOS 常见知识点（二）：RunLoop | 神刀安全网](#)  [W ?](#) -
[...] ibireme 的 深入理解RunLoop [...]

104. [深入研究 Runloop 与线程保活 | 秀品折](#)  [W ?](#) - [...] 深入理解 RunLoop [...]

105. [iOS NSTimer 为什么要配合 NSRunLoop 进行使用 | 秀品折](#) [W ?](#) - [...] 4.如果对 runLoop 的一些更详细的概念和底层实现有兴趣的，在这里我推荐两篇博客：##iOS开发-- RunLoop 的基本概念与例子分析##深入理解RunLoop## [...]

106. [iOS 保持界面流畅的技巧 — 木穆的博客](#) [W ?](#) - [...] Runloop 还不太了解，可以看一下我之前的文章 深入理解 RunLoop，里面对 ASDK [...]

107. [RunLoop的终极大杀器 | 秀品折](#) [W ?](#) - [...] 参考: 一个iOS菜单的白话文记录YY大神百度孙源的RunLoop视频 [...]

108. [iOS里的内存泄露 | 秀品折](#) [W ?](#) - [...] 我们能看到断点1和断点2 runloop还是在执行的，断点3表示runloop一个迭代已经结束了，即将进入睡眠。这里如果对runloop不了解的话可以看 ibireme的这篇深入理解RunLoop。我截取里面的一小段，大家可以看一下。 [...]

109. [各个线程 Autorelease 对象的内存管理 | 秀品折](#) [W ?](#) - [...] 针对第一个问题，比较容易理解，可以看一下： ibireme 的 深入理解RunLoop，主线程默认为我们开启 Runloop，RunLoop 会自动帮我们创建Autoreleasepool，并进行Push、Pop 等操作来进行内存管理 [...]

110. [基于runloop的线程保活、销毁与通信 | 神刀安全网](#) 🇨🇳 W ?

– [...] 关于runloop，还有一些更深层次拓展性的内容，包括与gcd的协同关系、AutoreleasePool的释放时机、GCDTimer和NSTimer区别等等，本来是想写一写的，奈何篇幅已经很大了，实在写不动了，推荐感兴趣的可以看看YY大神这篇：深入理解RunLoop [...]

111. [iOS多线程 | 秀品折](#) W ? – [...] 先来看一张表示这5个类的关系图（来源：

<http://blog.ibireme.com/2015/05/18/runloop/>） 。 [...]

112. [从VVeboTableViewDemo到YYAsyncLayer（二）-IT文库](#) 🇨🇳 W ? – [...] RunLoop：深入理解RunLoopiOS线下分享

《RunLoop》iOS RunLoop 编程手册（译）runloop原理 [...]

113. [老司机出品——源码解析之RunLoop详解-IT文库](#) 🇨🇳 W ? – [...] ——引自深入理解RunLoop [...]

114. [深入理解RunLoop，看我一篇就够了 | 零动科技](#) 🇨🇳 W ? – [...] 实在太多了，如果大家对runloop想有更深层次的了解，大家可以看看YYKit的作者写的这篇文章：深入理解RunLoop 其中介绍了与GCD的协同关系、AutoreleasePool的释放时机、GCDTimer与NSTimer还有CADisplayLink的区别等等，并举出AFNetworking和AsyncDisplayKit的实际应用，对提升很有帮助。最后的最后，奉上一个runloop的小demo，通过监听runloop的状态，优化图片加载，地址是：RunLoopDemo [...]

115. [SDWebImage 源码阅读笔记-IT文库](#) 🇨🇳 W ? – [...]

<http://blog.ibireme.com/2015/05/18/runloop/> [...]

116. [深入理解RunLoop-IT文库](#) 🇨🇳 W ? – [...] 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 [...]

117. [深入理解RunLoop-IT文库](#) 🇨🇳 W ? – [...] 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 [...]

118. [iOS-autorelease与autoreleasepool-IT文库](#)  W ? - [...] 参考资料官方文档黑幕背后的Autorelease深入理解
RunLoopObjective-C Autorelease Pool 的实现原理 [...]
119. [iOS-RunLoop浅析-IT文库](#)  W ? - [...] 参考资料:CF框架源码RunLoop 原理和核心机制CoreFoundation深入理解
RunLoop滑动卡顿优化官方文档 [...]
120. [iOS-RunLoop浅析-IT文库](#)  W ? - [...] 参考资料:CF框架源码RunLoop 原理和核心机制CoreFoundation深入理解
RunLoop滑动卡顿优化官方文档 [...]
121. [Re:从零开始的RunLoop实践01-IT文库](#)  W ? - [...] <http://blog.ibireme.com/2015/05/18/runloop/> 此篇为RunLoop写的相当好的一篇 强烈建议大家学习 也希望我早日能写出这种技术博文 [...]
122. [Re:从零开始的RunLoop实践02-使用ports 或custom input sources 和其他线程通信-IT文库](#)  W ? - [...] <http://www.jianshu.com/p/4d5b6fc33519><http://blog.ibireme.com/2015/05/18/runloop/> [...]
123. [关于RunLoop的原理探究及基本使用-IT文库](#)  W ? - [...] 参考文章: 官方文档经典RunLoop技术文: 深入理解
RunLoopRunLoop源码分析 (RunLoop几个重要的函数源码阅读): Runloop源码分析Mach相关: Mach消息发送机制进程间通信 (OSX/iOS)其他: runloop嵌套: NSRunLoop原理详解——不再有盲点 Cocoa深入学习:NSOperationQueue、NSRunLoop和线程安全源码解析之RunLoop详解ios开发——RunLoop 与GCD、Autorelease Pool之间的关系 [...]
124. [丁香园iOS电话面试问题总结 | 恰同学少年-不称](#)  W ? - [...] 超级棒的文章地址 [...]

关于

伽蓝之堂——
一只魔法师的工坊

相关链接

[Github](#)

[Weibo](#)

[Twitter](#)

[LinkedIn](#)

[DeviantART](#)

功能

[登录](#)

[文章RSS](#)

[评论RSS](#)

[WordPress.org](#)