

首页 > iOS开发

【iOS程序启动与运转】 - RunLoop个人小结

2015-08-21 09:34 编辑: suiling 分类: iOS开发 来源: 楚天舒的简书

44132

iOSRunLoop

招聘信息: iOS开发经理



作者: 楚天舒 授权本站转载。

学习iOS开发一般都是从UI开始的, 从只知道从IB拖控件, 到知道怎么在方法里写代码, 然后会显示什么样的视图, 产生什么样的事件, 等等。其实程序从启动开始, 一直都是按照苹果封装好的代码运行着, 暴露的一些属性和方法作为接口, 是让我们在给定的方法里写代码实现自定义功能, 做出各种各样的应用。这些方法的调用顺序最为关键, 熟悉了程序运转和方法调用的顺序, 才能更好地操控程序和代码, 尽量避免Xcode不报错又实现不了功能的BUG。从Xcode的线程函数调用栈可以看到一些方法调用顺序。


0 从程序启动开始到view显示:


start->(加载framework, 动态静态链接库, 启动图片, Info.plist, pch等)->main函数->UIApplicationMain函数:


```
1 - 初始化UIApplication单例对象
2 - 初始化AppDelegate对象, 并设为UIApplication对象的代理
3 - 检查Info.plist设置的xib文件是否有效, 如果有则解冻Nib文件并设置outlets, 创建显示key wir
4 - 建立一个主事件循环, 其中包含UIApplication的RunLoop来开始处理事件。
```


UIApplication:


热门资讯


- 


8行代码教你搞定导航控制器全屏滑动返回效果
点击量 10464
- 


不说鸡汤, 谈谈现实: 半路学编程能成大牛
点击量 8795
- 


UIWebView与JS的深度交互
点击量 8323
- 


超详细! iOS 并发编程之 Operation Queues
点击量 6931
- 

打造安全的App! iOS 安全系列之 HTTPS
点击量 6843
- 

为什么程序员的业余项目大多都死了?
点击量 6772
- 

这样好用的 ReactiveCocoa, 根本
点击量 6353
- 

为什么技术总是被轻视? 不服来辩!
点击量 6235
- 

一文让你彻底了解iOS 字体相关知识
点击量 5893
- 

深入理解Objective-C: 方法缓存
点击量 5119

综合评论

- 部分借鉴了《深入理解runloop》这篇内容, 正好我这两篇一起看的, 前面 speedvsfreedom 评论了 【iOS程序启动与运转】 - RunLoop个人小结
- 有图有步骤, 对新手非常有帮助。感谢作者。
幻想列车长 评论了 Xcode概览: 调试应用程序
- 打算研究下, 好像还挺不错的
canyu9512 评论了 这样好用的 ReactiveCocoa, 根本停不下来
- 回复 风之晕: 确实是, 新的东西如果

1. 通过window管理视图；
2. 发送RunLoop封装好的control消息给target；
3. 处理URL，应用图标警告，联网状态，状态栏，远程事件等。

AppDelegate：

管理UIApplication生命周期和应用的五种状态(notRunning/inactive/active/background/suspend)。

Key Window：

1. 显示view；
2. 管理rootViewController生命周期；
3. 发送UIApplication传来的事件消息给view。

rootViewController：

1. 管理view（view生命周期；view的数据源/代理；view与superView之间事件响应nextResponder的“备胎”）；
2. 界面跳转与传值；
3. 状态栏，屏幕旋转。

view：

1. 通过作为CALayer的代理，管理layer的渲染（顺序大概是先更新约束，再layout再display）和动画（默认layer的属性可动画，view默认禁止，在UIView的block分类方法里才打开动画）。layer是RGBA纹理，通过和mask位图（含alpha属性）关联将合成后的layer纹理填充在像素点内，GPU每1/60秒将计算出的纹理display在像素点中。
2. 布局子控件（屏幕旋转或者子视图布局变动时，view会重新布局）。
3. 事件响应：event和gesture。

插播控制器生命周期

runloop:

1. （要让马儿跑）通过do-while死循环让程序持续运行：接收用户输入，调度处理事件时间。
2. （要让马儿少吃草）通过mach_msg()让RunLoop没事时进入trap状态，节省CPU资源。

关于程序启动原理以及各个控件的资料，已经有太多资料介绍，平时我们也经常接触经常用到，但关于RunLoop的资料，官方文档总是太过简练，网上资源说法也不太统一，只能从CFRunLoopRef开源代码着手，试着学习总结下。（NSRunLoop是对CFRunLoopRef的面向对象封装，但是不是线程安全）。

1 Runloop

- 1、与线程和自动释放池相关：
- 2、CFRunLoopRef构造：数据结构；创建与退出；mode切换和item依赖；RunLoop启动

- CFRunLoopModeRef：数据结构（与CFRunLoopRef放一起了）；创建；类型；

modelItems：- CFRunLoopSourceRef：数据结构（source0/source1）；

- source0：

- source1：

jas_ 评论了 一种新的底栏交互方式尝试

赞楼主一个,写得好好!!!!

18970417572 评论了 七夕，献给屏幕前忙碌的你

好坏啊

barrylyl 评论了 小技巧：如何将NSString字符串放入剪贴板

是我的熊，不要抢

jackevin 评论了 七夕我们在一起，送你12只萌萌哒COCO熊！速抢！

"我们看到在类的定义里就有cache字段，没错，类的所有缓存都存在
ljysdfz 评论了 深入理解Objective-C：方法缓存

.....

z360623358 评论了 三年一个人使用虚幻引擎(UDK)开发的一个游戏心路

第一次翻译文章，谢谢大家。

熏修 评论了 iOS 9学习系列：UIStackView如何让你...

相关帖子

个人独立游戏上线《指点九宫》求下载

最近要上架一款自己的小游戏啦，叫《点金术士》

ios afnetworking 如何发送cookie给服务器进行身份验证？

妻给情夫生孩子，然而我竟成替死鬼

禅与 Objective-C 编程艺术 --- 中文译本

求助vs2015开发cocos2dx的win10通用应用无法启动

请问如何使UIScrollView内视图少滚动范围小的时候也可以自由拖动？

关于导航栏里的标题，求进

如何获取cell中textfield的值

- CFRunLoopTimerRef: 数据结构; 创建与生效; 相关类型 (GCD的timer与CADisplayLink)

- CFRunLoopObserverRef: 数据结构; 创建与添加; 监听的状态;

3、RunLoop内部逻辑: 关键在两个判断点 (是否睡觉, 是否退出)

- 代码实现:
- 函数作用栈显示:

4、RunLoop本质: mach port和mach_msg()。

5、如何处理事件:

- 界面刷新:
- 手势识别:
- GCD任务:
- timer: (与CADisplayLink)
- 网络请求:

6、应用:

- 滑动与图片刷新;
- 常驻子线程, 保持子线程一直处理事件
- 滑动与图片刷新;

RunLoop

1、与线程和自动释放池相关:

RunLoop的寄生于线程: 一个线程只能有唯一对应的RunLoop; 但这个根RunLoop里可以嵌套子RunLoop;

自动释放池寄生于RunLoop: 程序启动后, 主线程注册了两个Observer监听RunLoop的进出与睡觉。一个最高优先级OB监测Entry状态; 一个最低优先级OB监听BeforeWaiting状态和Exit状态。

线程(创建)-->RunLoop将进入-->最高优先级OB创建释放池-->RunLoop将睡-->最低优先级OB销毁旧池创建新池-->RunLoop将退出-->最低优先级OB销毁新池-->线程(销毁)

2、CFRunLoopRef构造:

数据结构:

```
1 // RunLoop数据结构
2 struct __CFRunLoopMode {
3     CFStringRef _name;           // Mode名字,
4     CFMutableSetRef _sources0;   // Set    CFMutableSetRef _sources1;   // Se
5 };
6 // mode数据结构
7 struct __CFRunLoop {
8     CFMutableSetRef _commonModes; // Set    CFMutableSetRef _commonModeItem
9     CFMutableSetRef _modes;       // Set    ...
10 };
```

微博



CocoaChina

已关注

【设计一款成功的社交游戏不可缺少的因素】过去, 我非常沉迷于《Nintendo Power》的游戏评级系统。当然了, 不管整体游戏是出色还是糟糕, 但是决定游戏是金子还是垃圾的分类也是决定它是否具有乐趣的关键。以下便是来自这本已经不复存在的杂志的例子: <http://t.cn/RLkhaVn>



8月21日 19:16 转发(3) | 评论

【iOS程序启动与运转- RunLoop个人小结】学习iOS开发一般都是从UI开始的, 从只知道从IB拖控件, 到知道怎么在方法里写代码, 然后会显示什么样的视图, 产生什么样的事件, 等

Coding —— 代码托管, 云端协作



免费私有库, 运行空间, 项目/质量管理 WebIDE, CodeInsight 让开发更简单!

C++开发教程免费在线试听

炒股软件: 免费下载

创建与退出：mode切换和item依赖

a 主线程的runloop自动创建，子线程的runloop默认不创建（在子线程中调用NSRunLoop *runloop = [NSRunLoop currentRunLoop];

获取RunLoop对象的时候，就会创建RunLoop）；

b runloop退出的条件：app退出；线程关闭；设置最大时间到期；modeItem为空；

c 同一时间一个runloop只能在一个mode，切换mode只能退出runloop，再重进指定mode（隔离modeItems使之互不干扰）；

d 一个item可以加到不同mode；一个mode被标记到commonModes里（这样runloop不用切换mode）。

启动RunLoop：

```
1 // 用DefaultMode启动
2 void CFRunLoopRun(void) {
3     CFRunLoopRunSpecific(CFRunLoopGetCurrent(), kCFRunLoopDefaultMode, 1.0e10, f
4 }
5 // 用指定的Mode启动，允许设置RunLoop最大时间（假无限循环），执行完毕是否退出
6 int CFRunLoopRunInMode(CFStringRef modeName, CFTimeInterval seconds, Boolean sto
7     return CFRunLoopRunSpecific(CFRunLoopGetCurrent(), modeName, seconds, return
8 }
```

CFRunLoopModeRef：

数据结构（见上）；

创建添加：runloop自动创建对应的mode；mode只能添加不能删除

```
1 // 添加mode
2 CFRunLoopAddCommonMode(CFRunLoopRef runloop, CFStringRef modeName);
```

类型：

1. kCFRunLoopDefaultMode: 默认 mode，通常主线程在这个 Mode 下运行。
2. UITrackingRunLoopMode: 追踪mode，保证ScrollView滑动顺畅不受其他 mode 影响。
3. UIInitializationRunLoopMode: 启动程序后的过渡mode，启动完成后就不再使用。
4. GSEventReceiveRunLoopMode: Graphic相关事件的mode，通常用不到。
5. kCFRunLoopCommonModes: 占位mode，作为标记DefaultMode和CommonMode用。

• modelItems：

```
1 // 添加移除item的函数（参数：添加/移除哪个item到哪个runloop的哪个mode下）
2 CFRunLoopAddSource(CFRunLoopRef rl, CFRunLoopSourceRef source, CFStringRef modeN
3 CFRunLoopAddObserver(CFRunLoopRef rl, CFRunLoopObserverRef observer, CFStringRef
4 CFRunLoopAddTimer(CFRunLoopRef rl, CFRunLoopTimerRef timer, CFStringRef mode);
5 CFRunLoopRemoveSource(CFRunLoopRef rl, CFRunLoopSourceRef source, CFStringRef mo
6 CFRunLoopRemoveObserver(CFRunLoopRef rl, CFRunLoopObserverRef observer, CFString
7 CFRunLoopRemoveTimer(CFRunLoopRef rl, CFRunLoopTimerRef timer, CFStringRef mode)
```

A-- CFRunLoopSourceRef：事件来源

按照官方文档CFRunLoopSourceRef为3类，但数据结构只有两类（？？？）

Port-Based Sources:与内核端口相关

Custom Input Sources:与自定义source相关

Cocoa Perform Selector Sources:与PerformSEL方法相关)

数据结构 (source0/source1) ;

```
1 // source0 (manual): order(优先级), callout(回调函数)
2 CFRunLoopSource {order =..., {callout =... }}
3 // source1 (mach port): order(优先级), port:(端口), callout(回调函数)
4 CFRunLoopSource {order = ..., {port = ..., callout =...}}
```

source0: event事件, 只含有回调, 需要标记待处理 (signal), 然后手动将runloop唤醒 (wakeup) ;

source1 : 包含一个 mach_port 和一个回调, 被用于通过内核和其他线程发送的消息, 能主动唤醒runloop。

B-- CFRunLoopTimerRef: 系统内“定时闹钟”

NSTimer和performSEL方法实际上是对CFRunLoopTimerRef的封装; runloop启动时设置的最大超时时间实际上是GCD的dispatch_source_t类型。

数据结构:

```
1 // Timer: interval:(闹钟间隔), tolerance:(延期时间容忍度), callout(回调函数)
2 CFRunLoopTimer {firing =..., interval = ...,tolerance = ...,next fire date = ...}
```

创建与生效;

```
1 //NSTimer:
2 // 创建一个定时器 (需要手动加到runloop的模式中)
3 + (NSTimer *)timerWithTimeInterval:(NSTimeInterval)ti invocation:(NSInvocation)
4 // 默认已经添加到主线程的runloop的DefaultMode中
5 + (NSTimer *)scheduledTimerWithTimeInterval:(NSTimeInterval)ti invocation:(NSIn
6 // performSEL方法
7 // 内部会创建一个Timer到当前线程的runloop中 (如果当前线程没runloop则方法无效; performSelec
8 - (void)performSelector:(SEL)aSelector withObject:(id)anArgument afterDelay:(NST
```

相关类型 (GCD的timer与CADisplayLink)

GCD的timer:

dispatch_source_t 类型, 可以精确的参数, 不用以来runloop和mode, 性能消耗更小。

```
1 dispatch_source_set_timer(dispatch_source_t source, // 定时器对象
2                           dispatch_time_t start, // 定时器开始执行的时间
3                           uint64_t interval, // 定时器的间隔时间
4                           uint64_t leeway // 定时器的精度
5                           );
```

CADisplayLink :

Timer的tolerance表示最大延期时间, 如果因为阻塞错过了这个时间精度, 这个时间点的回调也会跳过去, 不会延后执行。

CADisplayLink 是一个和屏幕刷新率一致的定时器, 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, 只是没有tolerance容忍时间), 造成界面卡顿的感觉。

C--CFRunLoopObserverRef: 监听runloop状态, 接收回调信息 (常见于自动释放池创建销毁)

数据结构:

```

1 // Observer: order (优先级), ativity (监听状态), callout (回调函数)
2 CFRunLoopObserver {order = ..., activities = ..., callout = ...}

```

创建与添加;

```

1 // 第一个参数用于分配该observer对象的内存空间
2 // 第二个参数用以设置该observer监听什么状态
3 // 第三个参数用于标识该observer是在第一次进入run loop时执行还是每次进入run loop处理时均执行
4 // 第四个参数用于设置该observer的优先级,一般为0
5 // 第五个参数用于设置该observer的回调函数
6 // 第六个参数observer的运行状态
7 CFRunLoopObserverCreateWithHandler(CFAllocatorGetDefault(), kCFRunLoopAllActivit
8 // 执行代码
9 }

```

监听的状态;

```

1 typedef CF_OPTIONS(CFOptionFlags, CFRunLoopActivity) {
2     kCFRunLoopEntry           = (1UL << 0), // 即将进入Loop
3     kCFRunLoopBeforeTimers    = (1UL << 1), // 即将处理 Timer
4     kCFRunLoopBeforeSources    = (1UL << 2), // 即将处理 Source
5     kCFRunLoopBeforeWaiting    = (1UL << 5), // 即将进入休眠
6     kCFRunLoopAfterWaiting     = (1UL << 6), // 刚从休眠中唤醒
7     kCFRunLoopExit            = (1UL << 7), // 即将退出Loop
8 };

```

3、RunLoop内部逻辑: 关键在两个判断点 (是否睡觉, 是否退出)

代码实现:

```

1 // RunLoop的实现
2 int CFRunLoopRunSpecific(runloop, modeName, seconds, stopAfterHandle) {
3     // 0.1 根据modeName找到对应mode
4     CFRunLoopModeRef currentMode = __CFRunLoopFindMode(runloop, modeName, false
5     // 0.2 如果mode里没有source/timer/observer, 直接返回。
6     if (__CFRunLoopModeIsEmpty(currentMode)) return;
7     // 1.1 通知 Observers: RunLoop 即将进入 loop。--- (0B会创建释放池)
8     __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopEntry);
9     // 1.2 内部函数, 进入loop
10    __CFRunLoopRun(runloop, currentMode, seconds, returnAfterSourceHandled) {
11        Boolean sourceHandledThisLoop = NO;
12        int retVal = 0;
13        do {
14            // 2.1 通知 Observers: RunLoop 即将触发 Timer 回调。
15            __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopBeforeTimers
16            // 2.2 通知 Observers: RunLoop 即将触发 Source0 (非port) 回调。
17            __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopBeforeSource
18            // 执行被加入的block
19            __CFRunLoopDoBlocks(runloop, currentMode);
20            // 2.3 RunLoop 触发 Source0 (非port) 回调。
21            sourceHandledThisLoop = __CFRunLoopDoSources0(runloop, currentMode,
22            // 执行被加入的block
23            __CFRunLoopDoBlocks(runloop, currentMode);
24            // 2.4 如果有 Source1 (基于port) 处于 ready 状态, 直接处理这个 Source1 然
25            if (__Source0DidDispatchPortLastTime) {
26                Boolean hasMsg = __CFRunLoopServiceMachPort(dispatchPort, &msg)
27                if (hasMsg) goto handle_msg;
28            }
29            // 3.1 如果没有待处理消息, 通知 Observers: RunLoop 的线程即将进入休眠(slee
30            if (!sourceHandledThisLoop) {
31                __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopBeforeWa
32            }
33            // 3.2. 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
34            // - 一个基于 port 的Source1 的事件。
35            // - 一个 Timer 到时间了
36            // - RunLoop 启动时设置的最大超时时间到了
37            // - 被手动唤醒
38            __CFRunLoopServiceMachPort(waitSet, &msg, sizeof(msg_buffer), &live
39            mach_msg(msg, MACH_RCV_MSG, port); // thread wait for receive m
40            }
41            // 3.3. 被唤醒, 通知 Observers: RunLoop 的线程刚刚被唤醒了。
42            __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopAfterWaiting
43            // 4.0 处理消息。
44            handle_msg:
45            // 4.1 如果消息是Timer类型, 触发这个Timer的回调。
46            if (msg_is_timer) {
47                __CFRunLoopDoTimers(runloop, currentMode, mach_absolute_time())
48            }
49            // 4.2 如果消息是dispatch到main_queue的block, 执行block。
50            else if (msg_is_dispatch) {
51                __CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE__(msg);

```



```

52     }
53     // 4.3 如果消息是Source1类型, 处理这个事件
54     else {
55         CFRunLoopSourceRef source1 = __CFRunLoopModeFindSourceForMachPo
56         sourceHandledThisLoop = __CFRunLoopDoSource1(runloop, currentMo
57         if (sourceHandledThisLoop) {
58             mach_msg(reply, MACH_SEND_MSG, reply);
59         }
60     }
61     // 执行加入到Loop的block
62     CFRunLoopDoBlocks(runloop, currentMode);
63     // 5.1 如果处理事件完毕, 启动RunLoop时设置参数为一次性执行, 设置while参数退出:
64     if (sourceHandledThisLoop && stopAfterHandle) {
65         retVal = kCFRunLoopRunHandledSource;
66     // 5.2 如果启动RunLoop时设置的最大运转时间到期, 设置while参数退出RunLoop
67     } else if (timeout) {
68         retVal = kCFRunLoopRunTimedOut;
69     // 5.3 如果启动RunLoop被外部调用强制停止, 设置while参数退出RunLoop
70     } else if (__CFRunLoopIsStopped(runloop)) {
71         retVal = kCFRunLoopRunStopped;
72     // 5.4 如果启动RunLoop的modeItems为空, 设置while参数退出RunLoop
73     } else if (__CFRunLoopModeIsEmpty(runloop, currentMode)) {
74         retVal = kCFRunLoopRunFinished;
75     }
76     // 5.5 如果没超时, mode里没空, loop也没被停止, 那继续loop, 回到第2步循环。
77     } while (retVal == 0);
78 }
79 // 6. 如果第6步判断后loop退出, 通知 Observers: RunLoop 退出。--- (OB会销毁新释放池)
80 __CFRunLoopDoObservers(rl, currentMode, kCFRunLoopExit);
81 }

```

• 函数作用栈显示:

```

1 {
2     // 1.1 通知Observers, 即将进入RunLoop
3     // 此处有Observer会创建AutoreleasePool: _objc_autoreleasePoolPush();
4     _CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__ (kCFRunLoopEnt
5     do {
6         // 2.1 通知 Observers: 即将触发 Timer 回调。
7         _CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__ (kCFRunLoopo
8         // 2.2 通知 Observers: 即将触发 Source (非基于port的, Source0) 回调。
9         _CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__ (kCFRunLoopo
10        // 执行Block
11        _CFRUNLOOP_IS_CALLING_OUT_TO_A_BLOCK__ (block);
12        // 2.3 触发 Source0 (非基于port的) 回调。
13        _CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION__ (source0);
14        // 执行Block
15        _CFRUNLOOP_IS_CALLING_OUT_TO_A_BLOCK__ (block);
16        // 3.1 通知Observers, 即将进入休眠
17        // 此处有Observer释放并新建AutoreleasePool: _objc_autoreleasePoolPop(); _o
18        _CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__ (kCFRunLoopo
19        // 3.2 sleep to wait msg.
20        mach_msg() -> mach_msg_trap();
21        // 3.3 通知Observers, 线程被唤醒
22        _CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__ (kCFRunLoopo
23        // 4.1 如果是被Timer唤醒的, 回调Timer
24        _CFRUNLOOP_IS_CALLING_OUT_TO_A_TIMER_CALLBACK_FUNCTION__ (timer);
25        // 4.2 如果是被dispatch唤醒的, 执行所有调用 dispatch_async 等方法放入main que
26        _CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE__ (dispatched block);
27        // 4.3 如果RunLoop是被Source1 (基于port的) 的事件唤醒了, 处理这个事件
28        _CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE1_PERFORM_FUNCTION__ (source1);
29        // 5. 退出判断函数调用栈无显示
30    } while (...);
31    // 6. 通知Observers, 即将退出RunLoop
32    // 此处有Observer释放AutoreleasePool: _objc_autoreleasePoolPop();
33    _CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__ (kCFRunLoopExi
34 }

```

一步一步写具体的实现逻辑过于繁琐不便理解, 按RunLoop状态大致分为:

- 1- Entry: 通知OB (创建pool);
- 2- 执行阶段: 按顺序通知OB并执行timer, source0; 若有source1执行source1;
- 3- 休眠阶段: 利用mach_msg判断进入休眠, 通知OB (pool的销毁重建); 被消息唤醒通知OB;
- 4- 执行阶段: 按消息类型处理事件;

5- 判断退出条件：如果符合退出条件（一次性执行，超时，强制停止，modelItem为空）则退出，否则回到第2阶段；

6- Exit：通知OB（销毁pool）。

4、RunLoop本质：mach port和mach_msg()。

Mach是XNU的内核，进程、线程和虚拟内存等对象通过端口发消息进行通信，RunLoop通过mach_msg()函数发送消息，如果没有port 消息，内核会将线程置于等待状态 mach_msg_trap()。如果有消息，判断消息类型处理事件，并通过modelItem的callback回调(处理事件的具体执行是在DoBlock里还是在回调里目前我还不太明白???)。

RunLoop有两个关键判断点，一个是通过msg决定RunLoop是否等待，一个是通过判断退出条件来决定RunLoop是否循环。

5、如何处理事件：

- 界面刷新：

当UI改变（Frame变化、UIView/CALayer 的继承结构变化等）时，或手动调用了 UIView/CALayer 的 setNeedsLayout/setNeedsDisplay方法后，这个 UIView/CALayer 就被标记为待处理。

苹果注册了一个用来监听BeforeWaiting和Exit的Observer，在它的回调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。

- 事件响应：

当一个硬件事件(触摸/锁屏/摇晃/加速等)发生后，首先由 IOKit.framework 生成一个 IOHIDEvent 事件并由 SpringBoard 接收，随后由mach port 转发给需要的App进程。

苹果注册了一个 Source1 (基于 mach port 的) 来接收系统事件，通过回调函数触发Source0（所以UIEvent实际上是基于Source0的），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。

- 手势识别：

如果上一步的 _UIApplicationHandleEventQueue() 识别到是一个gesture手势，会调用Cancel方法将当前的 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，其回调函数为 _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。

- GCD任务：

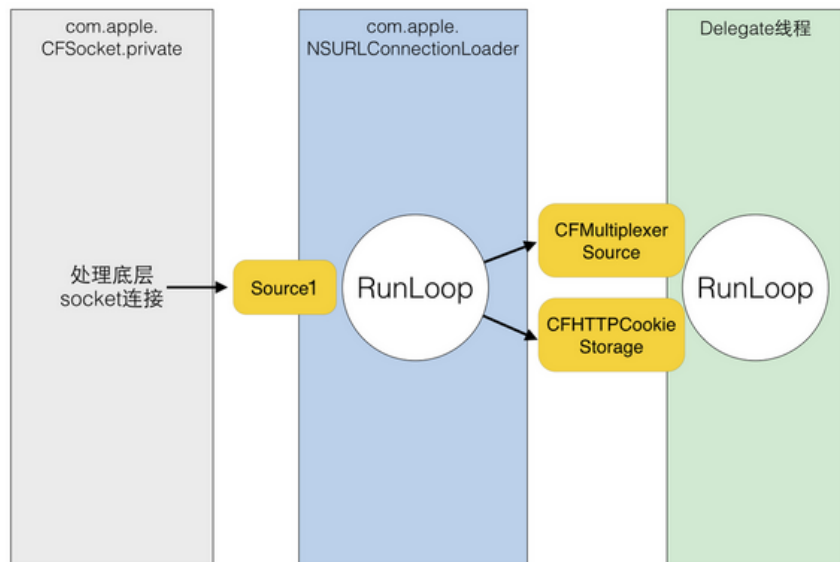
当调用 dispatch_async(dispatch_get_main_queue(), block) 时，libDispatch 会向主线程的 RunLoop 发送消息，RunLoop会被唤醒，并从消息中取得这个 block，并在回调里执行这个 block。RunLoop只处理主线程的block，dispatch 到其他线程仍然是由 libDispatch 处理的。

- timer：（见上modelItem部分）

- 网络请求:

关于网络请求的接口:最底层是CFSocket层, 然后是CFNetwork将其封装, 然后是NSURLConnection对CFNetwork进行面向对象的封装, NSURLSession 是 iOS7 中新增的接口, 也用到NSURLConnection的loader线程。所以还是以NSURLConnection为例。

当开始网络传输时, 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。



6、应用:

- 滑动与图片刷新;

当tableView的cell上有需要从网络获取的图片的时候, 滚动tableView, 异步线程会去加载图片, 加载完成后主线程就会设置cell的图片, 但是会造成卡顿。可以让设置图片的任务在CFRunLoopDefaultMode下进行, 当滚动tableView的时候, RunLoop是在 UITrackingRunLoopMode 下进行, 不去设置图片, 而是当停止的时候, 再去设置图片。

```
1 - (void)viewDidLoad {
2     [super viewDidLoad];
3     // 只在NSDefaultRunLoopMode下执行(刷新图片)
4     [self.myImageView performSelector:@selector(setImage:) withObject:[UIImage ima
5 }

```

- 常驻子线程, 保持子线程一直处理事件

为了保证线程长期运转, 可以在子线程中加入RunLoop, 并且给RunLoop设置item, 防止RunLoop自动退出。

```
1 + (void)networkRequestThreadEntryPoint:(id)__unused object {
2     @autoreleasepool {
3         [[NSThread currentThread] setName:@"AFNetworking"];
4         NSRunLoop *runLoop = [NSRunLoop currentRunLoop];
5         [runLoop addPort:[NSMachPort port] forMode:NSDefaultRunLoopMode];
6         [runLoop run];
7     }
8 }
9 + (NSThread *)networkRequestThread {
10     static NSThread *networkRequestThread = nil;
11     static dispatch_once_t oncePredicate;
12     dispatch_once(&oncePredicate, ^{
13         networkRequestThread = [NSThread alloc] initWithTarget:self selector:
14         [_networkRequestThread start];
15     });

```

```
16         return _networkRequestThread;
17     }
18     - (void)start {
19         [self.lock lock];
20         if ([self isCancelled]) {
21             [self performSelector:@selector(cancelConnection) onThread:[self class
22         ] else if ([self isReady]) {
23             self.state = AFOperationExecutingState;
24             [self performSelector:@selector(operationDidStart) onThread:[self clas
25         ]
26         [self.lock unlock];
27     }
```



微信扫一扫
订阅每日移动开发及APP推广热点资讯
公众号：CocoaChina

我要投稿

收藏文章

分享到:

上一篇：iOS 9学习系列：UIStackView如何让你的开发更简单

下一篇：iOS 9学习系列：如何使用ATS提高应用的安全性

相关资讯	
听故事搞懂多线程开发-- 屎壳郎老板和它的收费公厕	iOS 9学习系列：如何使用ATS提高应用的安全性
iOS 9学习系列：UIStackView如何让你的开发更简单	iOS 项目的目录结构能看出你的开发经验
iOS 9学习系列：Search API	iOS-CoreLocation：无论你在哪里，我都要找到你！
HTTP Live Streaming直播(iOS直播)技术分析与实现	理解Bitcode：一种中间代码
深入理解Objective-C：方法缓存	一分钟搭建个人详情界面



坚持1个月 看美剧不用字幕

每天45分钟 30天见证你的英语奇迹

立即行动 >



文章评论 (3)

- 

pengyoubieku 2015-08-22 19:47:09

http://www.cocoachina.com/ios/20150601/11970.html,还不如看原文

支持(0) 回复(0)
- 

speedvsfreedom 2015-08-21 14:47:15

部分借鉴了《深入理解runloop》这篇内容，正好我这张两篇一起看的，前面内容，顺的还可以

支持(0) 回复(1)
- 

2015-08-22 18:10:54

回复 speedvsfreedom: 发下你所指的《深入理解runloop》链接

支持(0) 回复
- 

leonajkl 2015-08-21 13:19:21

不错!!

支持(0) 回复(0)

对这篇文章有什么感想，写一下吧.....

发表评论