#### runloop

runloop概述

RunLoop 是什么、为什么存在：

Runloop是iOS底层三巨头之一，是线程的事件循环，让线程在有任务时执行、无任务时休眠，从而既保证线程常驻，又避免忙等。

runloop，顾名思义，是一个运行循环，也就是一个while循环，runloop保证了线程不退出以及负责事件分发。runloop和线程是一一对应的关系，每条线程的runloop默认不开启，线程的任务执行完毕自动退出，线程的runloop需要手动开启。主线程的runloop是APP启动时，appdelegate中的didfinish执行完毕后，系统api主动开启的。APP的首页的渲染任务就是主线程的runloop分发出去的。

runloop结构

runloop属于corefoundation框架，由C语言编写，runloop是c结构体对象，其成员变量有thread，modes，commonmodes，commonmodesitems，currentmode，blocks，thread标记了runloop对应的线程，modes表示有哪些模式，commonmodes表示哪些模式被标记为commond，commonmodeitems中的事件会同时被加载到被标记为commonmode的模式中，currentmode标记了当前runloop正在运行的模式，block是以链表的形式存储了需要执行的block，每个block也是和mode一一对应的。runloopmode同为结构体，成员变量主要有source0，source1，observers，timers。

runloop有多个模式，同一时刻runloop只能运行于一种模式，可以在不同的模式间切换，主线程的runloop默认有两种模式，defaultmode和uitrackingmode，程序员可以自定义新的模式，当程序员向runloop中添加事件的时候，比如timer，block等，需要指定具体的mode。当runloop处于某种模式的运行循环中，就会不断处理该mode对应的source0，source1，timers，observers，blocks。

runloop的运行循环伪代码如下：

notifyObservsers(Entry)

do{

notifyObservers(BeforeTimers)

notifyObservers(BeforeSources)

doBlocks()

if(hasSource0){

doSource0

doBlocks()

}

if(dispatchPort has msg){

handleDispatch()

goto AfterWait;

}

notifyObservers(BeforeWaiting)

livePort = mach\_msg(waitSet)

notifyObservers(AfterWaiting)

if(livePort == wakeUpPort){

just wake up, do nothing

}else if(livePort == dispatchPort){

handle dispatch()

}else if(livePort == TimePort){

do Timers()

}else if(livePort == OtherSource1){

do other source1

}else if(time out){

retal = timeout

}

After Wait:

do blocks()

if(shouldStop || timeout || mode is empty) break;

}while(true)

notifyObservers(Exit)

当runloop处于某种模式时，会不断处理该模式中的source0，source1，timers，observers，block等事件，当所有事件都处理完，会进入休眠状态，等待外部通过mach port的方式传递的source1的事件将其唤醒，并继续处理新的事件。

runloop是如何切换mode的？

runloop应用篇：应用篇分两部分：

1. 系统自带的runloop事件的分类；
2. 程序员基于runloop做的特有事件；

程序员常见的iOS系统的runloop的应用场景有哪些？这些场景都需要理解其原理

常见的source1：

常见的source0：

常见的blocks：

常见的timers：

1. GCD，使用GCDAsync给主队列添加任务时，都发生了什么？为什么在主线程添加的任务会延迟执行？

GCDAsync给主队列添加任务：

1. block入队：将任务block放入libdispatch维护的mainqueue；
2. 触发唤醒：
   * 1. 如果mainqueue是空闲的，libdispatch会向dispatchport写一条唤醒消息；
     2. 如果dispatchport已有消息，则不会写，避免重复唤醒；
3. runloop消费：runloop有两个时机执行mainqueue中待消费的block
   * 1. beforewaiting：休眠前检查dispatchport，有消息则立即执行；
     2. afterwaiting：被dispatchport唤醒并执行；

执行时调用\_\_CFRUNLOOP\_IS\_SERVICING\_THE\_MAIN\_DISPATCH\_QUEUE\_\_， 一次性清空所有block

为什么在主线程使用GCDasync添加的任务，会延迟执行？

因为GCD Async所在的代码块是一个runloop任务，而它所添加的任务，在当前任 务执行完毕后才会执行。

1. CADisplayLink，这种定时器的工作原理，信号是如何发出去的
   1. CADisplayLink第一次 add 时，CoreAnimation 为当前 mode 注册Source1监听VSync信号，CADisplayLink对象由coreanimation框架维护。
   2. IOKit的VSync 信号写入消息，触发 Source1，其回调为CA 的内部函数，遍历 CADisplayLink 列表，调用用户回调。
   3. CADisplayLink 不是 RunLoop 的 Timer，而是 CA 持有的对象，RunLoop 只是用来接收 VSync 信号和唤醒调度。
   4. CADisplayLink对象和source1的回调函数由coreanimation框架维护，而vsync信号由IOKit发出，runloop则居中起协调作用。
2. NSTimer的底层原理，和GCDTimer的区别，为什么注册在 defaultmode 中的定时器在scrollview滚动的时 失效
   1. NSTimer的底层原理：
      1. Addtimer tomode 函数被调用，添加定时器至该 mode 的 timers，根据 timers 中最早触发的定时器的时间间隔，执行系统调用mk\_timer\_arm()设置内核定时器，同时在该 mode 绑定的 timerPort 监听 source1事件（timeport 和监听 timerport 的 source1都是懒创建的）；
      2. 内核定时器到时后，通过 timerport 端口唤醒 runloop，触发 source1事件，source1的回调函数会遍历该 mode 中的定时器，触发应当响应的定时器的用户回调，如果该定时器是 repeat，则重复上述过程。
   2. GCDTimer是直接去内核注册、监听定时器，并不通过runloop来触发。GCDTimer比NSTimer要准，因为NSTimer触发的时候依赖于runloop，但是如果runloop中刚好有耗时任务，则时间上就会延迟。
   3. 当scrollview滚动的时候，runloop切换为UITrackingmode，而UITrackingmode和defaultmode的 timeport不一致，当内核定时器到时后给DefaultmodeTimePort 发消息，但UITrackingmode下并不会去读取该端口的消息，UITrackingmode的定时器 source1无法被触发，故无法触发用户设置的定时器回调。当滚动结束重回defaultmode，取 timeport 中的消息，如果该消息的定时器时间已过期，则runloop会判定该定时器过期并舍弃该定时器回调。
3. perform selector onthread waituntildone
   1. 将任务封装为blocks加入目标线程runloop的commonMode中
   2. 同步调用CFRunloopWakeup函数，唤醒runloop，runloop在合适时机执行blocks
   3. waituntildone是阻塞创造任务的线程
4. perform selector afterdelay

是nstimer的语法糖，默认向defaultmode中添加事件

1. 触摸事件以及其他硬件事件
   1. 触摸事件（包括键盘等）用户态框架UIKit、其他硬件（陀螺仪、加速计、耳机控制）用户态框架CoreMotion
      1. IOKit通过machmsg将触摸事件发给runloop，触发source1回调，该source1把事件交给UIKit/CoreMotion
      2. UIKit/CoreMotion把事件包装为UIEvent投递到source0，执行source0的时候，触发UIKit/CoreMotion的链条
2. 网络请求和runloop
   1. 网络请求的发起是在某个block、source0等中使用socket系统调用发起的，socket发起请求后会返回一个fd，CFNetwork会维护一个哈希表，以键值对的形式存储fd和请求对象；
   2. 网络请求的响应是source1转source0；
   3. BSDSocket->machport->CFNetwork，source1的回调中，CFNetwork根据消息体中的fd标识找到请求对象，封装为source0投递给runloop处理用户回调。
   4. 如果用户回调使用GCDAsyncMainQueue包裹，则还会用到dispatchport，在mainqueue处理真正的用户回调，所以用户回调经历了两个source1和block的封装
3. 什么样的blocks会被添加到runloop的block链表中，在doblocks的时候执行：
   1. CFRunLoopPerformBlock / NSRunLoop performBlock
   2. performSelector:onThread：内部实现会把 selector 封装成 block，投递到目标线程 RunLoop 的 block 链表。
4. source0和source1的区别是什么？哪些常见的事件是source0，哪些是source1？哪些source1的回 调会创建source0？
   1. source1是内核态machmsg触发，其主要用于和内核通信；source0是用户态代码触发，主要用于固定通道的事件的分发；
   2. 常见的source0事件，硬件事件（触摸、加速、陀螺仪、蓝牙、耳机、屏幕刷新、网络）source1转source0
   3. 常见的source1有wakeupport，dispatchport，timerport，硬件事件（触摸、加速、陀螺仪、蓝牙、耳机、屏幕刷新、网络）等
5. 主线程的runloop，系统都提前注册好了哪些observers？
   1. AutoreleasePool 管理
      1. entry时autoreleasePoolPush，beforewaiting时autoreleasePoolPop，afterwaiting时autoreleasePoolPush，exit时autoreleasePoolPop
   2. UIKit 绘制/布局相关
      1. BeforeWaiting，具体做了什么：
         1. 更新约束：收集“需要更新约束”的视图，调用 updateConstraintsIfNeeded，驱动 Auto Layout 约束求解。
         2. 布局：对“需要布局”的视图调用 layoutIfNeeded，自顶向下递归 layoutSubviews，计算 frame。
         3. 显示（绘制）：对“需要显示”的视图/层调用 displayIfNeeded（setNeedsDisplay 标记），最终可能触发 drawRect: 或 CALayerDelegate 的 displayLayer:。
   3. CA (Core Animation) 渲染
      1. BeforeWaiting和Exit，具体做了什么
         1. 收尾/合并隐式动画与属性变更：把对 CALayer 的修改与隐式动画一起装入当前 CATransaction。
         2. 计算动画起止值，生成动画时间线（需要的话）。
         3. 提交事务：CA::Transaction::commit() 将 Layer 树的更新以及动画指令 编码并通过 IPC 发送给系统的渲染服务（iOS 上是 CoreAnimation/BackBoard 的合成器）。
         4. 需要注意的是Core Animation和renderserver的通信是单向的，viewdidappear的调用，是Core Animation根据动画截止时间设置的类似于定时器的内部操作。
         5. 与 VSync 协同：
            1. CADisplayLink 是通过 Source1（VSync 的 mach port） 驱动的；
            2. 在一帧到来前，CA 会确保把这帧需要的事务提交好，以便渲染服务按时合成；
            3. 若你本轮没有显式触发 display link，CA 的 BeforeWaiting 提交也能保证“改动不会滞后到下一轮”。
6. 线程间通信，包括performselector onthread和GCD，线程间通信，是如何唤醒主线程runloop的？
   1. performselector onthread是通过wakeupPort+block的形式
   2. GCD是dispatchport source1的形式

程序员常见的自定义的runloop的应用场景有哪些？

1. 监测屏幕卡顿
2. 子线程保活，为什么runloop启动后，必须加入事件，否则就会退出？

nstimer定时器

\_\_CFRUNLOOP\_IS\_CALLING\_OUT\_TO\_A\_TIMER\_CALLBACK\_FUNCTION\_\_

runloop block \_\_CFRUNLOOP\_IS\_CALLING\_OUT\_TO\_A\_BLOCK\_\_

\_\_CFRunLoopDoBlocks

gcdblock

\_\_CFRUNLOOP\_IS\_SERVICING\_THE\_MAIN\_DISPATCH\_QUEUE\_\_

按钮点击事件：

#15 0x0000000180429214 in \_\_CFRUNLOOP\_IS\_CALLING\_OUT\_TO\_A\_SOURCE0\_PERFORM\_FUNCTION\_\_ ()

#16 0x000000018042915c in \_\_CFRunLoopDoSource0 ()

#17 0x00000001804288e4 in \_\_CFRunLoopDoSources0 ()

总结 + 提炼价值：

RunLoop 不仅是 iOS 事件驱动的核心，也是一把钥匙。理解它能解释各种‘奇怪现象’（Timer 停止、GCD 延迟、UI 卡顿），同时也能指导我们做一些工程化实践（卡顿监控、子线程保活、任务调度）。所以这是一个把系统理解和工程实践打通的知识点。