1. Runloop介绍
2. 作用
3. Runloop对象
4. Runloop和线程的关系

Runloop是基于线程来管理的，他们一一对应，共同存储在一个全局区的runLoopDic中，线程是key，Runloop是value。

1. Runloop的创建

我们绝对不会手动创建Runloop，只能获取不能创建。主线程所对应的Runloop在程序一启动进行创建主程序的时候系统就会自动帮我们创建，而子线程所对应的Runloop并不是在子线程创建的时候创建的，而是在第一次获取Runloop时创建，如果不获取这个Runloop，就永远不创建。

1. Runloop的获取

可以通过一个指定的线程从runLoopDic中获取它所对应的Runloop

1. Runloop的销毁

系统在创建Runloop的时候，会注册一个回调，确保线程在销毁的同时，也销毁掉漆对应的Runloop。

1. Runloop是基于线程的，并且与线程一起创建在一个全局的字典runLoopDic中，系统会帮我们创建好Runloop，之后我们通过这个字典来获取。当线程销毁时，系统也会自动帮我们销毁掉。
2. Runloop的相关类

CFRunLoopRef：NSRunLoop底层就是CFRunloopRef实现的，是基于C的

CFRunloopModeRef：是NSRunloop的运行模式，Runloop可以有多种运行模式，每次只会选用一种，每一个Mode都会有自己的特性

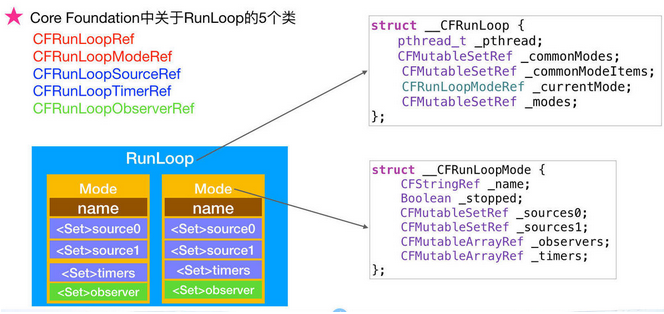
CFRunloopSourceRef：是用来响应事件的

CFRunLoopTimerRef：NSTimer的使用

CFRunLoopObserverRef：是用来监听RunLoop运行状态的一个类，UI刷新、自动释放池



1. 各个类之间的关系
2. 一个RunLoop包含若干个mode,每个mode又包含了多个Source/Timer/Oberver
3. 每次调用RunLoop主函数时，只能指定一个mode，这个mode也就是currentMode
4. 如果需要切换mode，只能先退出循环，再指定一个Mode进入。
5. 如果切换了Mode，但是Mode中不存在Source/Timer/Oberver，就会直接退出，不进入此循环



1. CFRunloopRef对象
2. NSRunloop对象是基于CFRunloopRef对象的，并且CFRunloopRef对象是基于C语言的，线程安全
3. 源码

struct \_\_CFRunLoop {

CFRuntimeBase \_base;

pthread\_mutex\_t \_lock;

\_\_CFPort \_wakeUpPort; //通过该函数CFRunLoopWakeUp内核向该端口发送消息可以唤醒runloop

Boolean \_unused;

volatile \_per\_run\_data \*\_perRunData;

pthread\_t \_pthread;//RunLoop对应的线程

uint32\_t \_winthread;

CFMutableSetRef \_commonModes;//存储的是字符串，记录所有标记为common的mode

CFMutableSetRef \_commonModeItems;//存储所有commonMode的item(source、timer、observer)

CFRunLoopModeRef \_currentMode;//当前运行的mode

CFMutableSetRef \_modes;//存储的是CFRunLoopModeRef

struct \_block\_item \*\_blocks\_head;//do blocks的时候用到

struct \_block\_item \*\_blocks\_tail;

CFAbsoluteTime \_runTime;

CFAbsoluteTime \_sleepTime;

CFTypeRef \_counterpart;

};

1. RunLoopModeRef
2. 源码

注意一点，运行模式是通过名称来识别的

typedef struct \_\_CFRunLoopMode \*CFRunLoopModeRef;

struct \_\_CFRunLoopMode {

CFRuntimeBase \_base;

pthread\_mutex\_t \_lock;

CFStringRef \_name;//mode名称，运行模式是通过名称来识别的

Boolean \_stopped;//mode是否被终止

char \_padding[3];

//整个结构体最核心的部分

---------------------------------------------------------------------------------

CFMutableSetRef \_sources0;//sources0

CFMutableSetRef \_sources1;//sources1

CFMutableArrayRef \_observers;//观察者

CFMutableArrayRef \_timers;//定时器

---------------------------------------------------------------------------------

CFMutableDictionaryRef \_portToV1SourceMap; //字典 key是mach\_port\_t，value是CFRunLoopSourceRef

\_\_CFPortSet \_portSet;//保存所有需要监听的port，比如\_wakeUpPort，\_timerPort都保存在这个数组中

CFIndex \_observerMask;

#if USE\_DISPATCH\_SOURCE\_FOR\_TIMERS

dispatch\_source\_t \_timerSource;

dispatch\_queue\_t \_queue;

Boolean \_timerFired;

Boolean \_dispatchTimerArmed;

#endif

#if USE\_MK\_TIMER\_TOO

mach\_port\_t \_timerPort;

Boolean \_mkTimerArmed;

#endif

#if DEPLOYMENT\_TARGET\_WINDOWS

DWORD \_msgQMask;

void (\*\_msgPump)(void);

#endif

uint64\_t \_timerSoftDeadline; /\* TSR \*/

uint64\_t \_timerHardDeadline; /\* TSR \*/

};

1. 系统默认注册的五个Mode

1. kCFRunLoopDefaultMode：App的默认Mode，通常主线程是在这个Mode下运行

2. UITrackingRunLoopMode：界面跟踪Mode，用于ScrollView追踪触摸滑动，保证界面滑动时不受其他 Mode 影响

3. UIInitializationRunLoopMode: 在刚启动 App 时第进入的第一个 Mode，启动完成后就不再使用，会切换到kCFRunLoopDefaultMode

4. GSEventReceiveRunLoopMode: 接受系统事件的内部 Mode，通常用不到

5. kCFRunLoopCommonModes: 这是一个占位用的Mode，作为标记kCFRunLoopDefaultMode和UITrackingRunLoopMode用，并不是一种真正的Mode ，也可以看做是公共的Mode

1. CommonModes

在RunLoop对象中有\_commonModes和\_commonModeItems两个字段

1. NSRunloopCommonModes表示为通用的Mode，也就是任意的一个Mode，里面的ModeItem是公共的。
2. 一个Mode可以将自己标记为Common属性，通过将Mode的名称添加到\_commonModes中来标记
3. 在\_commonModeItmes中添加的是Source/Timer/Observer
4. 当RunLoop的内容发生变化时，会自动的将\_commonModeItems的数据全部同步到\_commonModes中的每个Mode。
5. 可以使用NSRunLoopCommonModes来操作Common Items，或标记一个Mode为Common

[[NSRunLoop currentRunLoop] addTimer:timer forMode:NSRunLoopCommonModes];

1. 管理Mode的接口

//标记Mode为common

CFRunLoopAddCommonMode(CFRunLoopRef runloop, CFStringRef modeName);

//设置某一个Mode给一个RunLoop（一个RunLoop的当前Mode只能有一个）

CFRunLoopRunInMode(CFStringRef modeName, ...);

1. 对于一个Runloop，其内部的Mode只能增加不能删除，因为很明显我们不能自动创建和删除Mode本身，只能添加
2. 管理Mode Items的接口
3. RunLoop需要处理的消息属于Mode Item，包括Timer以及Source
4. RunLoop可以被别人监听，被监听的对象就是Observer，也属于Mode Item
5. 所有的Mode Item都可以加入到Mode中，一个Mode可以包含多个Mode Item，一个Item也可以被同时加入到多个Mode中，但同一个Item被加入到多个Mode会被覆盖掉。
6. 如果一个Mode中没有Mode Item，RunLoop直接退出，不进行循环
7. 当给一个Runloop传入一个Mode时，如果不存在，系统会帮我们创建对应的CFRunloopModeRef。
8. 接口：

CFRunLoopAddSource(CFRunLoopRef rl, CFRunLoopSourceRef source, CFStringRef modeName);

CFRunLoopAddObserver(CFRunLoopRef rl, CFRunLoopObserverRef observer, CFStringRef modeName);

CFRunLoopAddTimer(CFRunLoopRef rl, CFRunLoopTimerRef timer, CFStringRef mode);

CFRunLoopRemoveSource(CFRunLoopRef rl, CFRunLoopSourceRef source, CFStringRef modeName);

CFRunLoopRemoveObserver(CFRunLoopRef rl, CFRunLoopObserverRef observer, CFStringRef modeName);

CFRunLoopRemoveTimer(CFRunLoopRef rl, CFRunLoopTimerRef timer, CFStringRef mode);

1. Mode的切换



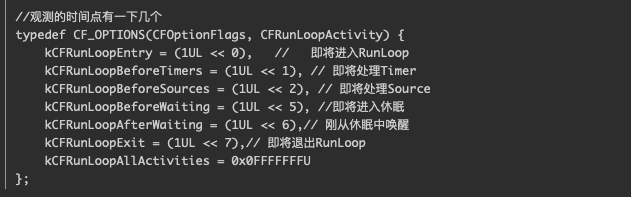
1. CFRunloopSourceRef
2. 有两个版本，Source0只包含了一个回调（函数指针），并不能主动触发事件。使用时，需要先调用CFRunLoopSourceSignal(source)，将这个Source标记为待处理，然后手动调用CFRunLoopWakeUp（runLoop），让其处理这个事件
3. Source1包含了一个mach\_port和一个回调（函数指针），被用于通过内核和其他线程相互发送消息。这种Source能主动唤醒RunLoop的线程。
4. 源代码



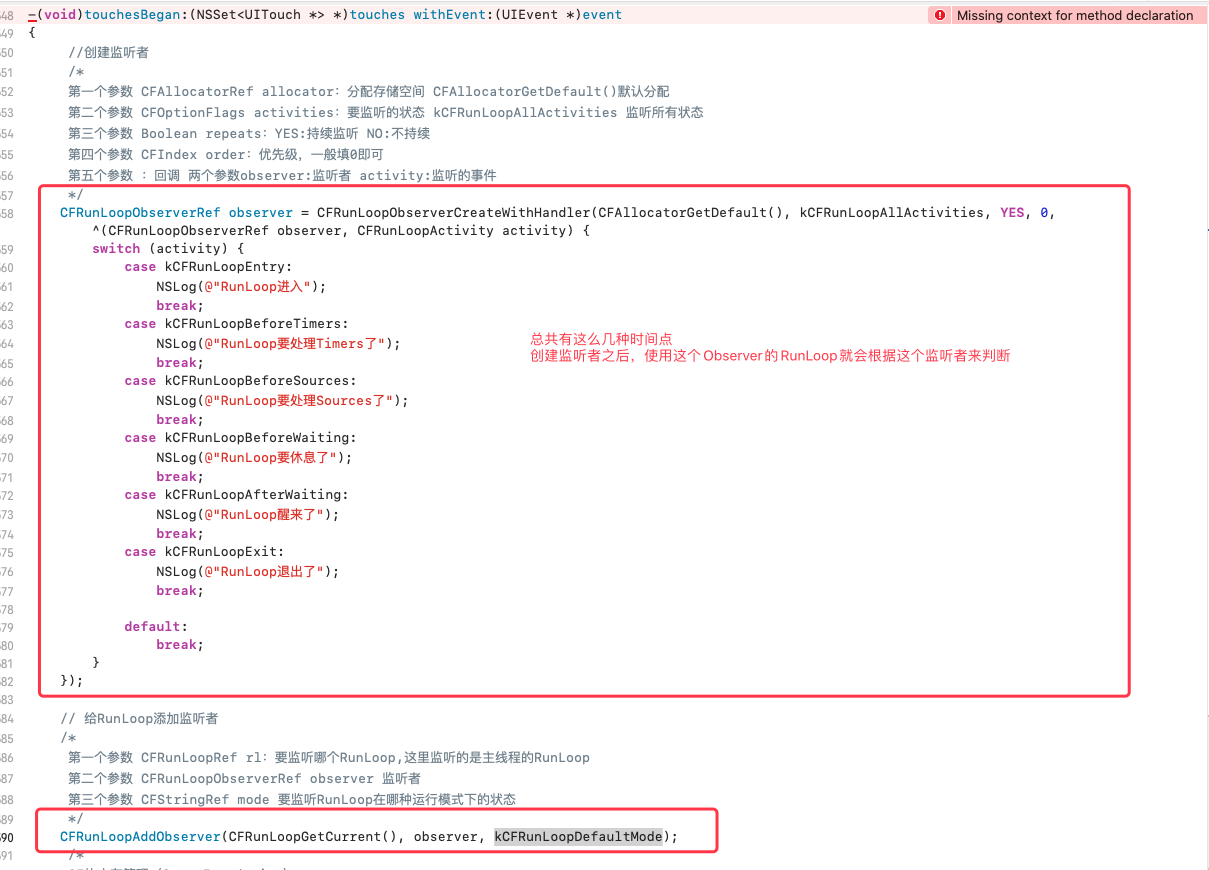
1. CFRunloopTimerRef
2. CFRunloopTimerRef是基于时间的触发器，包含了一个时间长度和一个回调（函数指针）
3. 当加入到RunLoop时，会注册相应的时间点，当时间点到时，RunLoop会主动被唤醒和执行回调操作
4. 源代码



1. CFRunloopObserverRef
2. 这是用来监听RunLoop本身的，每个Observer都包含一个回调（函数指针），当RunLoop的状态发生变化时，监听者就能通过回调接收到这个变化。
3. 观察时间点

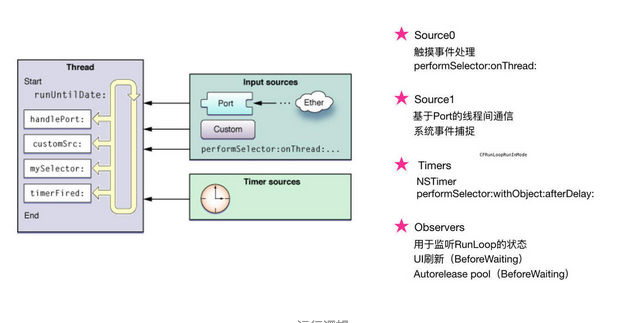


1. 具体的执行



1. RunLoop的实现逻辑
2. 输入事件类型
3. 图示

可以看到有有两种不同的来源，输入源（Input source)和定时源(Timer source)，输入源又分为Port和Custom



1. 来源按同步异步分类（同步异步的区别）

\*输入源是异步事件，通常消息来自于其他线程或程序，按照是否来源于内核（Port）还可以分类

\*Port -Based Sources，基于内核（Port）的事件，一般是由内核自动发出信号，我们不用管

\*Custom Input Sources，非内核（port）事件，一般是用户手动创建的Source，需要从其他线程手动发送信号

\*Cocoa Perform Selector Sources，performSelector系列方法，也是一种事件源。

\*输入源是同步事件

\*Timer Sources，基于时间的通知

\*包含两部分，NSTimer和performSelector:withObject:afterDelay:

1. 来源按对象分类

\*Source1

\*对应于上面说的基于内核（Port）的事件，通过内核来和其他线程进行通信

\*接收、分发系统事件。大部分屏幕交互事件有Source1接收，之后分发下去，最后由 Source0来处理

\*所以包括1）基于Port的线程间通信2）系统事件的捕获

\*Source0

\*非Port事件，与界面交互的触摸事件的最终处理

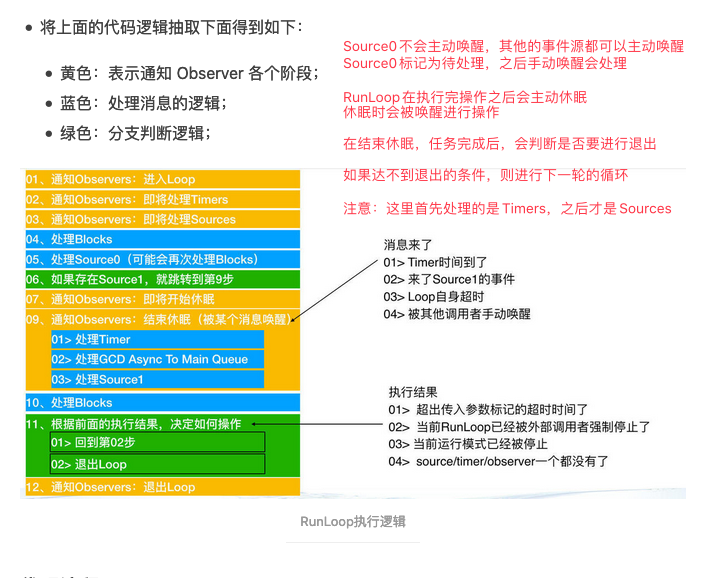
\*performSelector处理（没有delay）

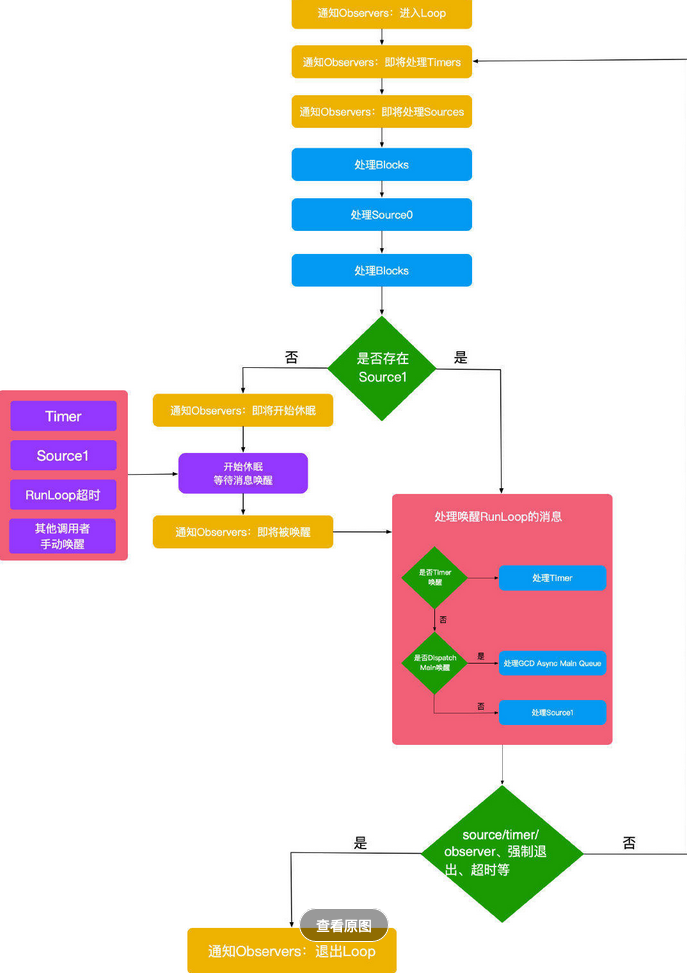
\*Timers

\*基于时间的处理

\*包含两部分，NSTimer和performSelector:withObject:afterDelay:

1. 源代码解析（后面打开源码再一步步分析）
2. 图解

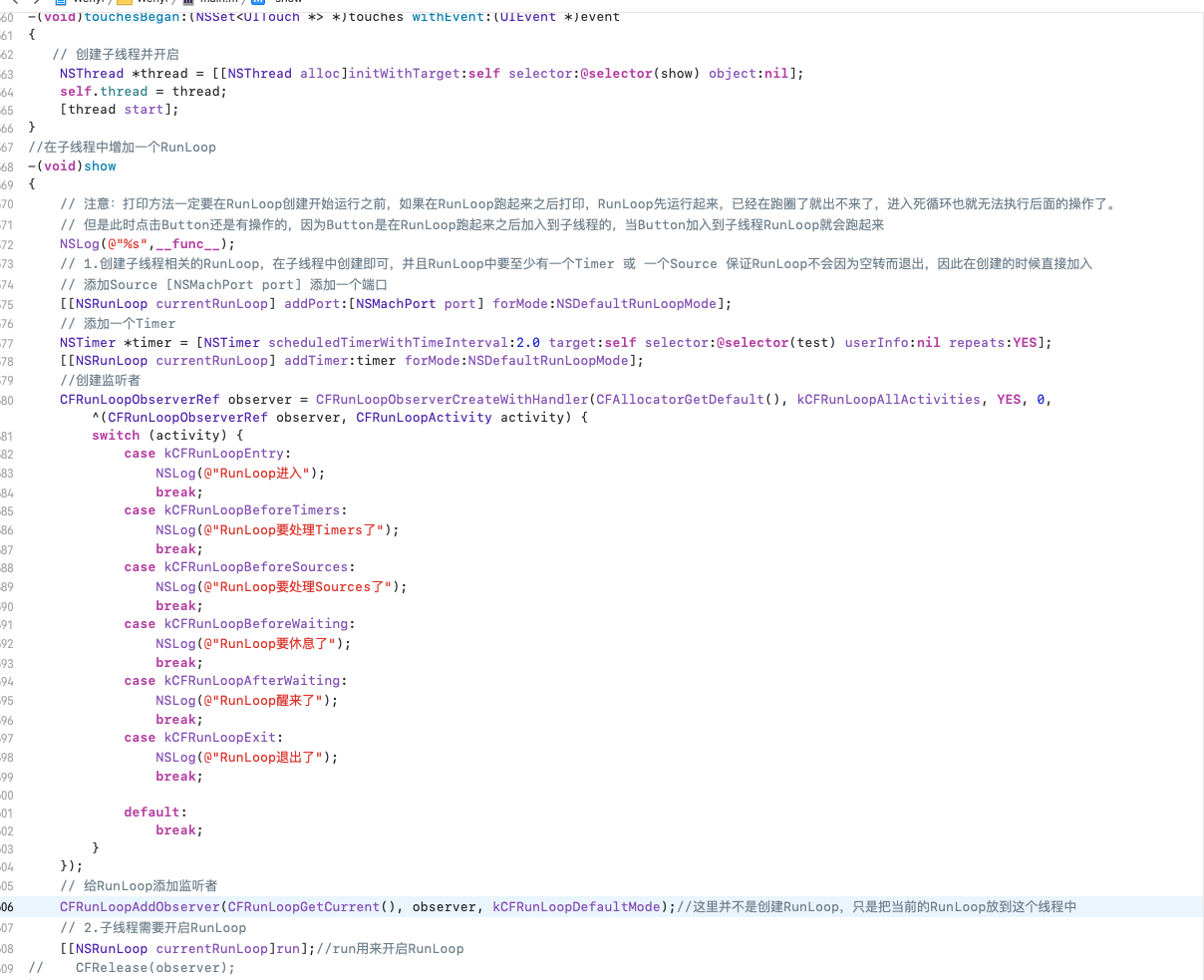




1. RunLoop的应用
2. 常驻线程

我们知道RunLoop可以让程序持续运行并响应事件，这个是针对它所属的线程而言的，因此我们想要给这个线程不停止，一直持续运行，就可以给它加一个RunLoop。

1. 代码



1. NSTimer的应用
2. 定时器的使用
3. 滑动时失效

滑动失效是因为我们把定时器加到了默认Mode了，不是**UITrackingRunLoopMode，所以在滑动时不会计时**

NSRunLoopCommonModes在任意的Mode下都可以使用。

[[NSRunLoop currentRunLoop] addTimer:timer forMode:NSRunLoopCommonModes];

1. 不准时

NSTimer是不准时的，因为一个NSTimer注册到RunLoop后，RunLoop会为其部分时间点注册好事件。但是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有一个属性叫Tolerance（宽容度），表示了当时间到后，容许有多少宽容度

1. 不延后执行

如果在执行一个很长的任务时，中间的一个时间点已经被错过了，那么这个回调函数也会被错过，不会延后执行。

1. 造成界面卡顿的一个原因

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，中间的一帧被跳过去了，就会造成界面卡顿

1. AutoreleasePool