# Runloop

## RunLoop的实际应用

### 定时器
- NSTimer其实就是CFRunLoopTimerRef，之间是toll-free bridged
- Timer不是实时的，会有误差，可能回调的时间点，RunLoop正在执行一个长任务，那么该次回调就会被跳过

### GCD
- dispatch_async到mainQueue上会给主线程RunLoop发消息唤醒RunLoop，RunLoop从消息中取出block执行

### performSelector
- 需要所在线程有一个活跃的RunLoop
- 内部是创建一个Source0添加到当前线程的RunLoop中

### 系统默认注册 5个mode
- App默认mode，一般情况下主线程RunLoop在这个mode下运行 — kCFRunLoopDefaultMode
- 界面跟踪Mode，追踪ScrollView触摸滑动，保证界面滑动不受其他Mode影响 — UITrackingRunLoopMode
- App启动时进入的第一个Mode，启动完成就不用了 — UIInitializationRunLoopMode
- 接收系统事件的内部mode — GCEventReceiveRunLoopMode
- 占位Mode，用于注册Common mode — kCFRunLoopCommonModes

### AutoreleasePool
- App启动后，苹果在主线程RunLoop注册了两个Observer
- 第一个Observer监听Entry(即将进入RunLoop)，在通知回调中调用_objc_autoreleasePoolPush()创建自动释放池，优先级最高，保证自动释放池创建在所有回调之前
- 第二个Observer监听BeforeWaiting(即将进入休眠)和Exit(即将退出RunLoop)两个事件，在BeforeWaiting的回调中先释放旧的自动释放池，然后新建一个自动释放池(保证下一个RunLoop的自动释放池是新的)，在Exit的回调中释放自动释放池，这个观察者优先级最低，保证在所有回调执行完再释放自动释放池

### 事件响应
- 苹果注册了一个Source1(基于port)用来接收系统事件
- 当一个硬件事件(触摸/锁屏/摇晃等)发送后，首先IOKit生成一个IOHIDEvent事件由SpringBoard接收，然后通过mach port转发给需要的App进程，然后会唤醒RunLoop去触发Source1的回调
- Source1的回调中调用_UIApplicationHandleEventQueue()进行应用内部分发，该函数会将IOHIDEvent包装成UIEvent进行处理分发

### 手势识别
- 在_UIApplicationHandlerEventQueue()中识别到一个手势，首先调用Cancel停止当前touch相关的回调，然后标记手势为待处理
- 苹果注册了一个Observer监听BeforeWaiting(即将进入休眠)，在Observer回调中取出待处理的手势执行回调

### 界面更新
- 操作UI时，会将这些UIView/CALayer标记为待处理
  - 更改Frame
  - 更新UIView/CALayer的层级
  - 手动调用UIView/CALayer的setNeedsLayout/setNeedsDisplay
- 苹果注册了一个Observer监听BeforeWaiting(即将进入休眠)和Exit(即将退出RunLoop)，回调会去遍历所有被标记为待处理的UIView/CALayer执行实际的绘制和调整，并更新UI界面

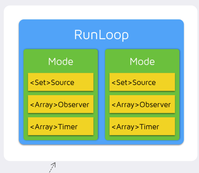
## RunLoop的底层实现
- 核心是基于mach port，其进入休眠时调用的是mach_msg()去接收消息，如果没有port消息过来，内核就会将线程置于等待的状态

## RunLoop的内部逻辑
- 通过指定modeName启动RunLoop，RunLoop启动后就会一直停留在循环中，直到超时或者手动被停止
- 1. 通知Observer，即将进入RunLoop
- RunLoop循环
  - 2. 通知Observer，即将处理Timer
  - 3. 通知Observer，即将处理Source0
  - 4.处理Source0
  - 5. 如果有Source1，跳到第9步
  - 6. 通知Observer， RunLoop即将休眠
  - 7. 休眠，等待唤醒
    - Source1(port）主动唤醒
    - Timer
    - 外部手动唤醒
  - 8. 通知Observer，RunLoop刚被唤醒
  - 9. 处理唤醒时收到的消息，然后跳到第二步
- 10. 通知Observer，即将退出RunLoop

## RunLoop的状态
- 即将进入RunLoop — kCFRunLoopEntry
- 即将处理Timers事件 — kCFRunLoopBeforeTimers
- 即将处理Sources事件 — kCFRunLoopBeforeSources
- 即将进入休眠 — kCFRunLoopBeforeWaiting
- 刚从休眠中唤醒，即将开始处理事件 — kCFRunLoopAfterWaiting
- 即将退出RunLoop — kCFRunLoopExit

## 概念
- RunLoop是Event Loop模型(保证一个线程能随时处理事件不退出)的一种实现
- Event Loop模型
  - 如何管理事件和消息
  - 没有消息时休眠避免资源浪费
  - 有新消息或新事件需要处理时立即唤醒
- RunLoop实际就是一个对象，用于管理事件和消息，并提供一个入口函数来执行Event Loop的逻辑，线程执行该函数后就会一直在函数内部循环，直到循环结束，函数返回
- iOS提供了两个Runloop对象
  - CFRunLoopRef — CoreFoundation框架中提供，纯C函数的API,线程安全
  - NSRunLoop — 对CFRunLoopRef的封装，提供面向对象的API，非线程安全

## RunLoop与线程的关系
- RunLoop与线程一一对应，两者的关系存储在全局的字典中
- RunLoop不允许直接创建
  - CFRunLoopGetMain() — [NSRunLoop mainRunLoop]
  - CFRunLoopGetCurrent() — [NSRunLoop currentRunLoop]
  - RunLoop的创建发生在第一次获取的时候
- 主线程 — RunLoop默认开启
- 其他线程
  - RunLoop需要手动开启
  - 只能在线程内部获取其对应的RunLoop




## RunLoop对外接口
- 关系图
- CFRunLoopRef
  - 包含多个mode
    - 包含多个Source
    - 包含多个Observer
    - 包含多个Timer
  - 启动RunLoop需要指定mode，一次只能指定一个mode，切换mode需要退出RunLoop重新进入
- CFRunLoopModeRef — 没有对外暴露，封装在CFRunLoopRef中
- CFRunLoopSourceRef
  - 事件产生的地方
  - source0
    - 非基于port的事件，主要是App内部事件(点击，触摸等)
    - 只包含一个回调(函数指针)，不能主动触发
      - CFRunLoopSourceSignal(source)标记为待处理
      - 手动调用CFRunLoopWakeUp(runloop)唤醒RunLoop处理事件
  - source1
    - 基于port的事件，通过内核和其他线程相互发消息，接收和分发系统事件
    - 包含mach_port和一个回调(函数指针)，会主动唤醒RunLoop处理事件
- CFRunLoopTimeRef
  - 基于时间的触发器
  - 包含一个时间长度和回调(函数指针)，加入RunLoop时，RunLoop会注册其对应时间点，时间点到时，RunLoop会被唤醒执行回调
- CFRunLoopObserverRef
  - 观察者
  - 包含一个回调(函数指针),当RunLoop状态变化时，观察者通过回调接收通知

## RunLoop的Mode

### 类型
- Default — NSDefaultRunLoopMode(kCFRunLoopDefaultMode) — 最常用的mode，运行RunLoop和配置Source的首选
- Event tracking
  - NSEventTrackingRunLoopMode(Cocoa) — 追踪鼠标拖拽或者其他界面跟踪之类的事件切换到这个mode
  - UITrackingRunLoopMode(Cocoa Touch) — 追踪ScrollView的滑动等用户交互轨迹
- Common modes — NSRunLoopCommonModes（kCFRunLoopCommonModes） — 可配置的通用模式集合
- Modal — NSModalPanelRunLoopMode — modal panel使 它接收与之相关的Source事件
- Connection — NSConnectionReplyMode — 结合NSConnection使用， 用于监听回复(很少用到，已经废弃)

### CommonModes的概念
- 一个Mode可以将其标记为“Common”属性(通过其ModeName添加到RunLoop的CommonModes中)
- 然后将ModeItem(Source/Timer/Observer)标记成“Common”，这样的ModeItem就会自动更新到具有“Common”属性的Mode中去
- 应用场景
  - 主线程RunLoop预置 ·两个Mode:kCFRunLoopDefaultMode 和 UITrackingRunLoopMode，两者都已经被标记为“Common”属性，默认情况下App处在defaultMode，如果创建一个Timer添加到defaultMode中，Timer会重复回调，当滑动界面，RunLoop会切换到UITrackingRunLoopMode，此时Timer就不会被回调了
  - 解决方案
    - 手动将timer添加到需要回调的多个mode中去
    - 直接将timer添加到NSRunLoopCommonModes中，这样timer会同步给每个具有“Common”属性的mode

### API
- 管理Mode
  - CFRunLoopAddCommonMode(runloop, modeName) — 将一个mode注册到RunLoop的CommonModes中
  - CFRunloopRunInMode(modeName, ...) — 指定modeName启动RunLoop
- 管理ModeItem(Source/Timer/Observer)
  - CFRunLoopAddSource(CFRunLoopRef rl, CFRunLoopSourceRef source, CFStringRef modeName);
  - CFRunLoopAddObserver(CFRunLoopRef rl, CFRunLoopObserverRef observer, CFStringRef modeName);
  - CFRunLoopAddTimer(CFRunLoopRef rl, CFRunLoopTimerRef timer, CFStringRef modeName);
  - CFRunLoopRemoveSource(CFRunLoopRef rl, CFRunLoopSourceRef source, CFStringRef modeName);
  - CFRunLoopRemoveObserver(CFRunLoopRef rl, CFRunLoopObserverRef observer, CFStringRef modeName);
  - CFRunLoopRemoveTimer(CFRunLoopRef rl, CFRunLoopTimerRef timer, CFStringRef modeName);