RunLoop 相关面试整理

为什么 NSTimer 有时候不好使?

因为创建的 `NSTimer` 默认是被加入到了 `defaultMode`，所以当 `Runloop` 的 `Mode` 变化时，当前的 `NSTimer` 就不会工作了。

AFNetworking 中如何运用 Runloop?

`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)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;
}
```

```
- (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`

autoreleasePool 在何时被释放?

`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。

PerformSelector:afterDelay:这个方法在子线程

中是否起作用？为什么？怎么解决？

不起作用，子线程默认没有 `Runloop`，也就没有 `Timer`。
解决的办法是可以使用 `GCD` 来实现：`Dispatch_after`

RunLoop 的Mode

关于`Mode`首先要知道一个`RunLoop`对象中可能包含多个`Mode`，且每次调用 `RunLoop` 的主函数时，只能指定其中一个 `Mode(CurrentMode)`。切换 `Mode`，需要重新指定一个 `Mode` 。主要是为了分隔开不同的 `Source`、`Timer`、`Observer`，让它们之间互不影响。

当 `RunLoop`运行在 `Mode1` 上时，是无法接受处理 `Mode2` 或 `Mode3` 上的 `Source、Timer、Observer` 事件的

总共是有五种 `CFRunLoopMode`：

- `kCFRunLoopDefaultMode`：默认模式，主线程是在这个运行模式下运行
- `UITrackingRunLoopMode`：跟踪用户交互事件（用于 `ScrollView` 追踪触摸滑动，保证界面滑动时不受其他`Mode`影响）
- `UIInitializationRunLoopMod`：在刚启动 `App` 时第进入的第一个 `Mode`，启动完成后就不再使用
- `GSEventReceiveRunLoopMode`：接受系统内部事件，通常用不到
- `kCFRunLoopCommonModes`：伪模式，不是一种真正的运行模式，是同步 `Source/Timer/Observer`到多个 `Mode` 中的一种解决方案

RunLoop的实现机制

对于`RunLoop`而言最核心的事情就是保证线程在没有消息的时候休眠，在有消息时唤醒，以提高程序性能。`RunLoop` 这个机制是依靠系统内核来完成的（苹果操作系统核心组件`Darwin`中的`Mach`）。
`RunLoop` 通过`mach_msg()`函数接收、发送消息。它的本质是调用函数 `mach_msg_trap()`，相当于是一个系统调用，会触发内核状态切换。在用户态调用 `mach_msg_trap()`时会切换到内核态；内核态中内核实现的 `mach_msg()` 函数会完成实际的工作。

即基于 `port`的 `source1`，监听端口，端口有消息就会触发回调；而`source0`，要手动标记为待处理和手动唤醒 `RunLoop`

大致逻辑为：

1、通知观察者 `RunLoop` 即将启动。

2、通知观察者即将要处理 `Timer` 事件。

3、通知观察者即将要处理 `source0` 事件。

4、处理 `source0` 事件。

5、如果基于端口的源(`Source1`)准备好并处于等待状态，进入步骤9。

6、通知观察者线程即将进入休眠状态。

7、将线程置于休眠状态，由用户态切换到内核态，直到下面的任一事件发生才唤醒线程。

- 一个基于 `port` 的 `Source1` 的事件(图里应该是 `source0`)。
- 一个 `Timer` 到时间了。
- `RunLoop` 自身的超时时间到了。
- 被其他调用者手动唤醒。

8、通知观察者线程将被唤醒。

9、处理唤醒时收到的事件。

- 如果用户定义的定时器启动，处理定时器事件并重启 `RunLoop` 。进入步骤2。
- 如果输入源启动，传递相应的消息。
- 如果 `RunLoop` 被显示唤醒而且时间还没超时，重启 `RunLoop` 。进入步骤2

10、通知观察者 `RunLoop` 结束。

怎么创建一个常驻线程?

1、为当前线程开启一个 `RunLoop` （第一次调用 `[NSRunLoop currentRunLoop]` 方法时实际是会先去创建一个 `RunLoop` ）

2、向当前 `RunLoop` 中添加一个 `Port/Source` 等维持 `RunLoop` 的事件循环（如果 `RunLoop` 的 `mode` 中一个 `item` 都没有， `RunLoop` 会退出）

3、启动该 `RunLoop`

RunLoop 的数据结构

`NSRunLoop(Foundation)`是`CFRunLoop(CoreFoundation)`的封装，提供了面向对象的API
`RunLoop`相关的主要涉及五个类:
`CFRunLoop`：`RunLoop`对象
`CFRunLoopMode`：运行模式
`CFRunLoopSource`：输入源/事件源
`CFRunLoopTimer`：定时源
`CFRunLoopObserver`：观察者

1、CFRunLoop

由`pthread`(线程对象，说明`RunLoop`和线程是一一对应的)、currentMode(当前所处的运行模式)、`modes`(多个运行模式的集合)、`commonModes`(模式名称字符串集合)、`commonModelItems`(`Observer,Timer,Source`集合)构成

2、CFRunLoopMode

由`name、source0、source1、observers、timers`构成

3、CFRunLoopSource

分为`source0`和`source1`两种

source0:
即非基于port的，也就是用户触发的事件。需要手动唤醒线程，将当前线程从内核态切换到用户态

source1:
基于port的，包含一个 mach_port 和一个回调，可监听系统端口和通过内核和其他线程发送的消息，能主动唤醒RunLoop，接收分发系统事件。具备唤醒线程的能力

4、CFRunLoopTimer

基于时间的触发器，基本上说的就是`NSTimer`。在预设的时间点唤醒`RunLoop`执行回调。因为它是基于`RunLoop`的，因此它不是实时的（就是`NSTimer`是不准确的。 因为`RunLoop`只负责分发源的消息。如果线程当前正在处理繁重的任务，就有可能导致`Timer`本次延时，或者少执行一次）。

5、CFRunLoopObserver
监听以下时间点: CFRunLoopActivity
kCFRunLoopEntry : RunLoop准备启动

kCFRunLoopBeforeTimers： RunLoop将要处理一些Timer相关事件

kCFRunLoopBeforeSources： RunLoop将要处理一些Source事件

kCFRunLoopBeforeWaiting： RunLoop将要进行休眠状态,即将由用户态切换到内核态

kCFRunLoopAfterWaiting： RunLoop被唤醒，即从内核态切换到用户态后

kCFRunLoopExit： RunLoop退出

kCFRunLoopAllActivities： 监听所有状态

6、各数据结构之间的联系

线程和`RunLoop`一一对应，`RunLoop`和`Mode`是一对多的，`Mode`和`source`、`timer`、`observer`也是一对多的

解释一下 NSTimer

`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`

解释一下事件响应的过程？

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。这个过程的详细情况可以参考这里。

`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。

解释一下手势识别的过程？

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。

利用 runloop 解释一下页面的渲染的过程？

当我们调用 `[UIView setNeedsDisplay]` 时，这时会调用当前 `View.layer` 的 `[view.layer setNeedsDisplay]`方法。

这等于给当前的 `layer` 打上了一个脏标记，而此时并没有直接进行绘制工作。而是会到当前的 `Runloop` 即将休眠，也就是 `beforeWaiting` 时才会进行绘制工作。

紧接着会调用 `[CALayer display]`，进入到真正绘制的工作。`CALayer` 层会判断自己的 `delegate` 有没有实现异步绘制的代理方法 `displayer:`，这个代理方法是异步绘制的入口，如果没有实现这个方法，那么会继续进行系统绘制的流程，然后绘制结束。

`CALayer` 内部会创建一个 `Backing Store`，用来获取图形上下文。接下来会判断这个 `layer` 是否有 `delegate`。

如果有的话，会调用 `[layer.delegate drawLayer:inContext:]`，并且会返回给我们 `[UIView DrawRect:]` 的回调，让我们在系统绘制的基础之上再做一些事情。

如果没有 `delegate`，那么会调用 `[CALayer drawInContext:]`。
以上两个分支，最终 `CALayer` 都会将位图提交到 `Backing Store`，最后提交给 `GPU`。
至此绘制的过程结束。

什么是异步绘制？

异步绘制，就是可以在子线程把需要绘制的图形，提前在子线程处理好。将准备好的图像数据直接返给主线程使用，这样可以降低主线程的压力。

异步绘制的过程

要通过系统的 `[view.delegate displayLayer:]` 这个入口来实现异步绘制。

代理负责生成对应的 `Bitmap`

设置该 `Bitmap` 为 `layer.contents` 属性的值