RunLoop学习

1.解决中文乱码的输出

注意:

1.为什么不需要.h文件?

因为通常写分类是添加新的方法,新的方法要写在.h文件中,而且使用里面的方法必须要导入.h文件

但是这个是重写原类的原有方法,因此不需要.h文件

结论:1 .h文件只是用于导入到另一个文件中相互使用 2.编译时,只会编译所有的.m文件,当发现你重写了方法,到时就会调用当前从写的方法,即使没有.h文件

代码如下:

#import <Foundation/Foundation.h>

@implementation NSDictionary (Log)

/\*\*

该方法就是该NSArray(NSExtendedArray)分类中的一个方法

当打印字典/数组时会调用字典/数组的这个方法,这个方法返回什么他就打印什么

-(NSString \*)descriptionWithLocale:(id)locale

但是Xcode8以后,不会调用上面那个方法了,但会调用下面这个方法

\*/

- (NSString \*)descriptionWithLocale:(id)locale indent:(NSUInteger)level

{

NSMutableString \*string = [NSMutableString string];

// \n换行符,\t首行缩进

// 开头有个{

[string appendString:@"{\n"];

// 遍历所有的键值对

[self enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL \*stop) {

//appendFormat:拼接内容,可以传参数

[string appendFormat:@"\t%@", key];

[string appendString:@" : "];

[string appendFormat:@"%@,\n", obj];

}];

// 结尾有个}

[string appendString:@"}"];

// 查找最后一个逗号

//NSBackwardsSearch反过来搜索,正常情况下是按顺序搜索,只会搜索到第一个","

NSRange range = [string rangeOfString:@"," options:NSBackwardsSearch];

//打印[]空数组时会报错

if (range.location != NSNotFound)

[string deleteCharactersInRange:range];

return string;

}

@end

@implementation NSArray (Log)

- (NSString \*)descriptionWithLocale:(id)locale indent:(NSUInteger)level

{

NSMutableString \*string = [NSMutableString string];

// 开头有个[

[string appendString:self.count ? @"[\n" : @"["];

// 遍历所有的元素

[self enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL \*stop) {

[string appendFormat:@"\t%@,\n", obj];

}];

// 结尾有个]

[string appendString:@"]"];

// 查找最后一个逗号

NSRange range = [string rangeOfString:@"," options:NSBackwardsSearch];

if (range.location != NSNotFound)

[string deleteCharactersInRange:range];

return string;

}

@end

**Runloop**

**1.**什么是**runloop**?

字面理解:运行循环/跑圈

**2.**基本作用

1>保持程序的持续运行

打开app它一直不死,就是因为runloop

2>处理App中的各种事件（比如触摸事件、定时器事件、Selector事件）

3>节省CPU资源，提高程序性能：该做事时做事，该休息时休息

能让我们的主线程,有事情就做事情,没事情就休息,比如点击事件:点击前主线程在休息,当点击事件处理完毕后,让主线程再次休息

…

**3.**如果没有**runloop**

int main(int argc, char \* argv[]) {

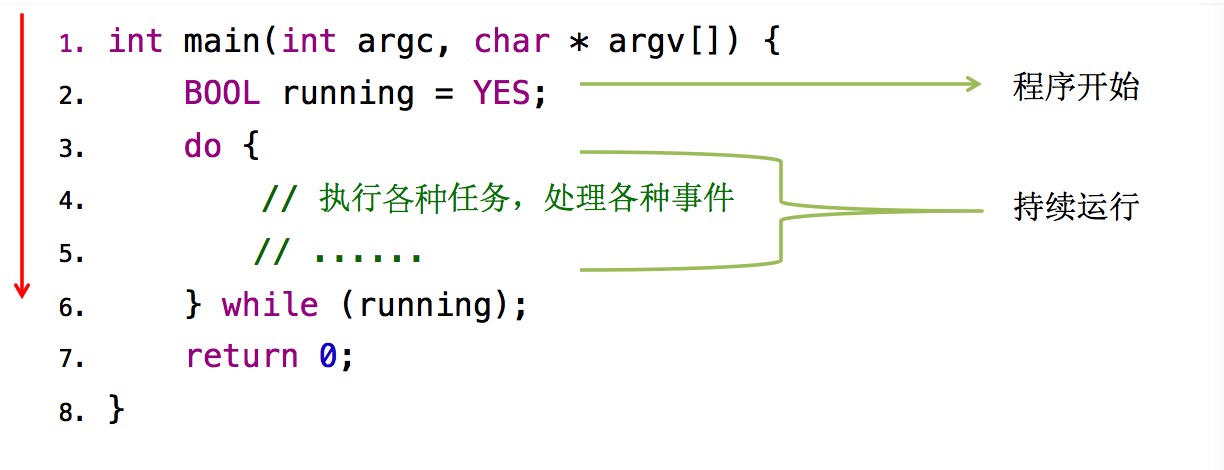
NSLog(@"execute main function");

return 0;

}

第3行后程序就结束了

如果有了runloop大致运行代码逻辑如下:



由于main函数里面( UIApplicationMain())启动了个RunLoop，所以程序并不会马上退出，保持持续运行状态

main函数中的RunLoop

int main(int argc, char \* argv[]) {

@autoreleasepool {

return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));

}

}

代码的UIApplicationMain函数内部就启动了一个RunLoop

所以UIApplicationMain函数一直没有返回，保持了程序的持续运行

这个默认启动的RunLoop是跟主线程相关联的

**4.Runloop**对象

iOS中有2套API来访问和使用RunLoop

Foundation(OC):NSRunLoop

Core Foundation(C):CFRunLoopRef

该框架有个特点所有的类都已CF开头,以Ref(reference:引用)结尾

NSRunLoop和CFRunLoopRef都代表着RunLoop对象

NSRunLoop是基于CFRunLoopRef的一层OC包装，所以要了解RunLoop内部结构，需要多研究CFRunLoopRef层面的API（Core Foundation层面）

**5.RunLoop**与线程

每条线程都有唯一的一个与之对应的RunLoop对象

主线程的RunLoop已经自动创建好了，子线程的RunLoop需要主动创建

RunLoop在第一次获取时创建，在线程结束时销毁

**6.**获得**RunLoop**对象

Foundation

[NSRunLoop currentRunLoop]; // 获得当前线程的RunLoop对象

[NSRunLoop mainRunLoop]; // 获得主线程的RunLoop对象

Core Foundation

CFRunLoopGetCurrent(); // 获得当前线程的RunLoop对象

CFRunLoopGetMain(); // 获得主线程的RunLoop对象

代码举例如下:

- (void)viewDidLoad {

[super viewDidLoad];

//当前线程就是主线程

NSLog(@"%p--%p",[NSRunLoop mainRunLoop],[NSRunLoop currentRunLoop]);

//创建一个线程

NSThread \*thread = [[NSThread alloc] initWithTarget:self selector:@selector(run) object:nil];

[thread start];

}

-(void)run{

/\*

[NSRunLoop currentRunLoop]:这就是在子线程中创建了一个runloop,不是通过alloc/init,创建是懒加载的

通过MJ分析runloop(C语言)源码:

创建runloop函数只有一个参数t:线程(这也说明一个runloop只有一个源码)

\_CFRunLoopGet0(pthread\_t t) {

1.如果t==kNilPthreadT(0的意思),那么t就等于主线程

\_\_CFRunLoops:本身是一个可变字典,用于存储所有的runloop,key值为线程,value为runloop

\_\_CFRunLoops :{

mainthread : mainloop,

thread1:loop1,

...

}

2.如果发现\_\_CFRunLoops为空值

2.1先创一个可变字典(就是\_\_CFRunLoops)

2.2根据主线程创建一个主runloop,即mainloop

2.3将mainloop存入可变字典,key值为主线程

3.根据传进来的线程到字典中拿到相应的runloop

4.没有创建一个newloop->将newloop存入可变字典->返回这个newloop

5.有就直接返回loop

}

\*/

NSLog(@"%p",[NSRunLoop currentRunLoop]);

}

**7.RunLoop**相关类

Core Foundation中关于RunLoop的5个类

CFRunLoopRef

CFRunLoopModeRef

CFRunLoopSourceRef

CFRunLoopTimerRef

CFRunLoopObserverRef

没有后4个runloop时不会跑圈的

分析下图:



runloop里面有很多mode对象->mode对象里面又有很多source/observer/timer对象->有了这些mode以及子内容runloop才能一直跑圈/休息/有事件触发(注意:runloop中的Mode中必须有source或者timer才能一直运行)

7.1CFRunLoopModeRef

CFRunLoopModeRef代表RunLoop的运行模式

一个 RunLoop 包含若干个 Mode，每个Mode又包含若干个Source/Timer/Observer

每次RunLoop启动时，只能指定其中一个 Mode，这个Mode被称作 CurrentMode

[NSRunLoop currentRunLoop] runMode: beforeDate:

如果需要切换Mode，只能退出Loop，再重新指定一个Mode进入

这样做主要是为了分隔开不同组的Source/Timer/Observer，让其互不影响

系统默认注册了5个Mode:(前两种最重要)

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

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

分析:当我们滚动图片时就会切换到这个模式,跟踪你的滚动,一旦停止滚动就会切换到Default模式

举例:当我们滚动的时候,其他事件没反应,比如定时器,当滚动的时候定时器没反应,因为它在Default模式中

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

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

kCFRunLoopCommonModes: 这是一个占位用的Mode，不是一种真正的Mode

7.2 CFRunLoopTimerRef

CFRunLoopTimerRef是基于时间的触发器,基本上说的就是NSTimer

它受runloop的Mode影响

GCD的定时器不受runLoop的Mode影响

GCD定时器实现不一样

代码举例:

- (void)viewDidLoad {

[super viewDidLoad];

//向sb中拖入一个textView

// [NSTimer scheduledTimerWithTimeInterval:1.0 target:self selector:@selector(test) userInfo:nil repeats:YES];

//上面一段代码做了很多事情,等价代码如下:

//timer添加到defaultMode里面去,mode再添加到runloop中去,runloop在启动时候再指定这个mode->拿出timer来用

NSTimer \*timer = [NSTimer timerWithTimeInterval:1.0 target:self selector:@selector(test) userInfo:nil repeats:YES];

//也就说只能在Defaultmode下好使

// [[NSRunLoop currentRunLoop] addTimer:timer forMode:NSDefaultRunLoopMode];

//情况一:没有滚动textView时,1秒打印一次,一旦滚动textView,定时器无用了,停止滚动,定时器有起作用了

//情况二:如果想要一滚动timer就有用,一停止滚动timer就失效,那么模式就用UITrackingRunLoopMode

// [[NSRunLoop currentRunLoop] addTimer:timer forMode:UITrackingRunLoopMode];

//情况三:滚动不影响定时器

//定时器只会跑在标记为CommonModes的模式下

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

//通过打印runloop,我们发现kCFRunLoopDefaultMode/UITrackingRunLoopMode这两种模式都被标记为了commonmodes标签了,因此就可以做到了

NSLog(@"%@",[NSRunLoop currentRunLoop]);

}

- (void)test {

NSLog(@"----");

}

7.3 CFRunLoopSourceRef(系统定的我们用不着)

CFRunLoopSourceRef是事件源（输入源）

我们的一些触发事件,都是由source触发的

1>以前的分法

Port-Based Sources

Custom Input Sources(处理接口调用)

Cocoa Perform Selector Sources(处理performselector调用)

2>现在的分法

Source0：非基于Port的

Source1：基于Port的

7.4 CFRunLoopObserverRef

CFRunLoopObserverRef是观察者，能够监听RunLoop的状态改变

可以监听的时间点有以下几个:

/\* Run Loop Observer Activities \*/

typedef CF\_OPTIONS(CFOptionFlags, CFRunLoopActivity) {

kCFRunLoopEntry = (1UL << 0),//即将进入loop,左移动0->2的0次放= 1

kCFRunLoopBeforeTimers = (1UL << 1),//即将处理Timer …2

kCFRunLoopBeforeSources = (1UL << 2),//即将处理source...4

kCFRunLoopBeforeWaiting = (1UL << 5),//即将进入休眠 …32

kCFRunLoopAfterWaiting = (1UL << 6),//刚刚从休眠中唤醒...64

kCFRunLoopExit = (1UL << 7),//即将推出loop 128

kCFRunLoopAllActivities = 0x0FFFFFFFU

};

代码举例:

// 创建observer

CFRunLoopObserverRef observer = CFRunLoopObserverCreateWithHandler(CFAllocatorGetDefault(), kCFRunLoopAllActivities, YES, 0, ^(CFRunLoopObserverRef observer, CFRunLoopActivity activity) {

NSLog(@"----监听到RunLoop状态发生改变---%zd", activity);

});

//添加观察者,监听runloop的状态

CFRunLoopAddObserver(CFRunLoopGetCurrent(), observer, kCFRunLoopDefaultMode);

// 释放Observer

CFRelease(observer);

/\*

//需要自己管理内存

CF的内存管理（Core Foundation）

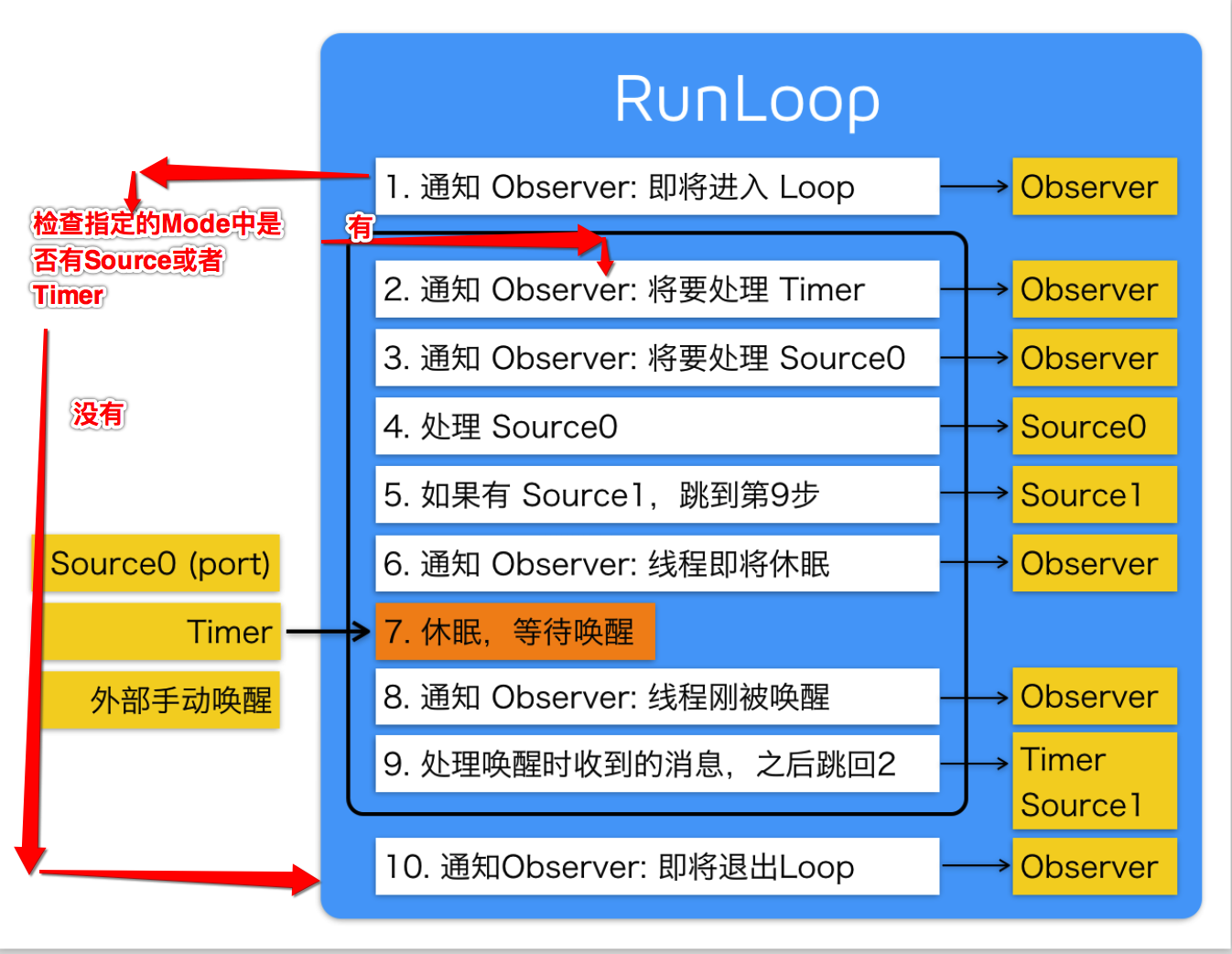
1.凡是带有Create、Copy、Retain等字眼的函数，创建出来的对象，都需要在最后做一次release

\* 比如CFRunLoopObserverCreate

2.release函数：CFRelease(对象);

\*/

**8.RunLoop**的逻辑流程图



**9.RunLoop**的应用

1>performSelector的使用

//延迟3秒设置图片并且值只在NSDefaultRunLoopMode模式下

[self.imageView performSelector:@selector(setImage:) withObject:[UIImage imageNamed:@"123"] afterDelay:3.0 inModes:@[NSDefaultRunLoopMode]

2>常驻线程

#import "ViewController.h"

#import "ZHThread.h"

@interface ViewController ()

@property (nonatomic,strong) ZHThread \*thread;

@end

@implementation ViewController

/\*

常驻线程:让一个线程永远不死

通过继承NSThread并且从写delloc监听线程,可以看到,开启线程后执行完任务立马死掉

注意:不能简单搞个属性强引用,因为线程一旦执行完毕,即使不释放掉,当前的线程也处于消亡状态,从新开启([self.thread start])会崩溃

当然不通过start调用,通过performselector调用虽然不会崩溃,但是不会调用run2,因为线程处于消亡状态

//解决办法:线程中添加runloop

好处:可以随时让拿到这个线程让他做一些事情.没有做事情之前,这个runloop处于休眠状态,一旦调用该线程处理事件,就回唤醒runloop去执行该线程的事件

使用:搞一个常驻线程监控联网状态

原理:

1.创建一个子线程并开启线程

2.在开启方法(run)中添加一个runloop,并开启runloop

3.runloop会一直跑圈不会执行完毕,阻止了开启方法的执行完毕即卡住子线程不让他执行完毕

4.一旦子线程中有事件,runloop会被唤醒,让改事件在该线程中执行,一旦执行完毕,继续跑圈卡住该线程

如果用一个死循环代替runloop是不可以的,因为死循环一直在处理这个事件,不会停下来先去执行触发事件

\*/

- (void)viewDidLoad {

[super viewDidLoad];

self.thread = [[ZHThread alloc] initWithTarget:self selector:@selector(run) object:nil];

//执行线程

[self.thread start];

}

-(void)touchesBegan:(NSSet<UITouch \*> \*)touches withEvent:(UIEvent \*)event{

[super touchesBegan:touches withEvent:event];

//随时拿到这个子线程执行事件

[self performSelector:@selector(run2) onThread:self.thread withObject:nil waitUntilDone:nil];

}

-(void)run{

NSLog(@"=====run====");

//run:默认为NSDefaultRunLoopMode模式,时间是永远,下面三行代码等价

//往里面添加source(mode中啥都没有,runloop没用)

[[NSRunLoop currentRunLoop] addPort:[NSPort port] forMode:NSDefaultRunLoopMode];

//启动runloop

[[NSRunLoop currentRunLoop] run];

// [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode beforeDate:[NSDate distantFuture]];

// [[NSRunLoop currentRunLoop] runUntilDate:[NSDate distantFuture]];

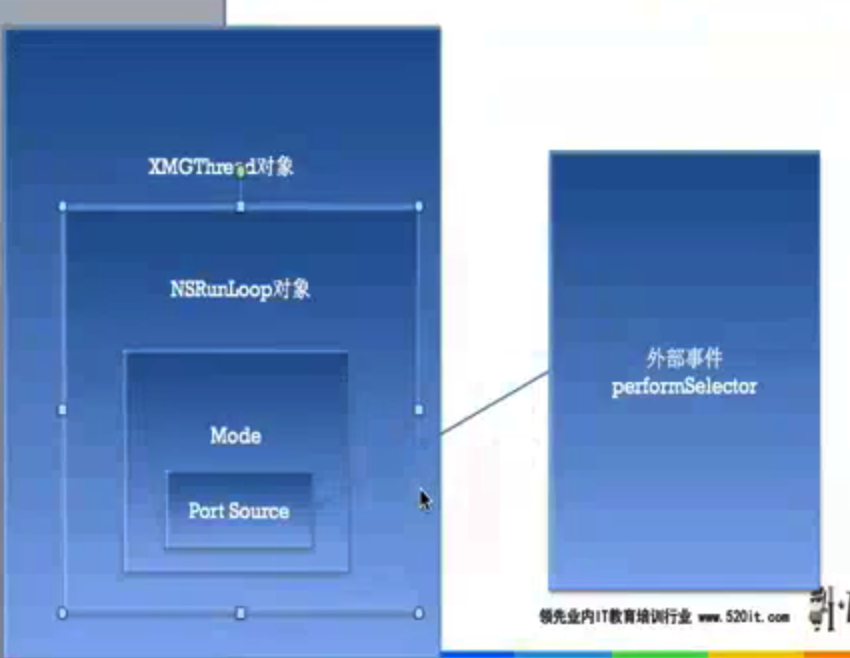
NSLog(@"永远不会调用======");

}

-(void)run2{

NSLog(@"=====run2====");

}



3>在子线程搞个定时器一直调用

//创建子线程初始化调用这个方法(将上面viewdidload方法中run改为runx即可)

-(void)runx{

@autoreleasepool {

//创建定时器

NSTimer \*timer = [NSTimer timerWithTimeInterval:2.0 target:self selector:@selector(run2) userInfo:nil repeats:YES];

//将定时器添加到runloop

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

[[NSRunLoop currentRunLoop] run];

//或者

// [NSTimer scheduledTimerWithTimeInterval:2.0 target:self selector:@selector(run2) userInfo:nil repeats:YES];

//放在主线程中就不会用到这句,因为主线程一直在run,不用手动run

// [[NSRunLoop currentRunLoop] run];

}

}

4>自动释放池

将所有对象都放到这个池子中去,到池子释放的时候给池子中的每一个对象release

自动释放池什么时候释放?(看源码得到)

runloop休眠之前会释放一次,因为休眠可能会很长,如果不释放,会堆积很多

释放后会同时创建一个新的释放池,处理新一轮事件

因此,在创建runloop的代码外层最好包装一层自动释放池

比如上面的代码

**10.GCD**的定时器

GCD的定时器不受runLoop的Mode影响(滚动界面时,定时器不受影响)

GCD定时器实现不一样

GCD比NSTimer准确

代码举例:

#import "ViewController.h"

@interface ViewController ()

/\*\* 定时器(这里不用带\*，因为dispatch\_source\_t就是个类，内部已经包含了\*) \*/

@property (nonatomic, strong) dispatch\_source\_t timer;

@end

@implementation ViewController

int count = 0;

- (void)touchesBegan:(NSSet \*)touches withEvent:(UIEvent \*)event

{

// 获得队列

// dispatch\_queue\_t queue = dispatch\_get\_global\_queue(0, 0);

dispatch\_queue\_t queue = dispatch\_get\_main\_queue();

// 创建一个定时器(dispatch\_source\_t本质还是个OC对象)

self.timer = dispatch\_source\_create(DISPATCH\_SOURCE\_TYPE\_TIMER, 0, 0, queue);

// 设置定时器的各种属性（几时开始任务，每隔多长时间执行一次）

// GCD的时间参数，一般是纳秒（1秒 == 10的9次方纳秒）

// 何时开始执行第一个任务

// dispatch\_time(DISPATCH\_TIME\_NOW, 3.0 \* NSEC\_PER\_SEC) 比当前时间晚3秒

//当前时间直接写:DISPATCH\_TIME\_NOW

dispatch\_time\_t start = dispatch\_time(DISPATCH\_TIME\_NOW, (int64\_t)(1.0 \* NSEC\_PER\_SEC));

uint64\_t interval = (uint64\_t)(1.0 \* NSEC\_PER\_SEC);

dispatch\_source\_set\_timer(self.timer, start, interval, 0);

// 设置回调

dispatch\_source\_set\_event\_handler(self.timer, ^{

NSLog(@"------------%@", [NSThread currentThread]);

count++;

if (count == 4) {

// 取消定时器

dispatch\_cancel(self.timer);

self.timer = nil;

}

});

// 启动定时器

dispatch\_resume(self.timer);

}

@end

**RunLoop**面试题

1.什么是RunLoop?

1>从字面意思看:运行循环/跑圈

2>其实它内部就是do-while循环,在这个循环内部不断地处理各种任务(比如:source/Timer/Observer)

3>一个线程对应一个runloop,主线程的runloop默认已经启动,子线程的runloop需要自己手动启动(调用run方法)

4>RunLoop只能选择一个Mode启动,如果当Mode中没有任何source(source0/source1),Timer,那么久直接退出RunLoop

2.自动释放池什么时候释放?

通过Observer监听RunLoop的状态,一旦监听到RunLoop即将进入休眠等待状态,就释放自动释放池(KCFRunLoopBeforeWaiting)

3.在开发中如何使用RunLoop?什么应用场景?

1>开启一个常驻线程(让一个子线程不进入消亡状态,等待其他的线程发来消息,处理其他事件)

在子线程中开启一个定时器

在子线程中进行一些长期监控

2>可以控制定时器在特定模式下运行

3>可以让某些事件(行为/任务)在特定模式下执行

4>可以添加Observer监听runloop的状态,比如监听点击事件的处理(在所有点击事件之前做一些事情)

**IOS**程序启动与运转**:**

从Xcode的线程函数调用栈(注意看看)可以看到一些方法调用顺序。

(主线程开启)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：{

- 初始化UIApplication单例对象

  - 初始化AppDelegate对象，并设为UIApplication对象的代理

  - 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 window、rootViewController、与rootViewController关联的根view（没有关联则看rootViewController同名的xib），否则launch之后由程序员手动加载。

  - 建立一个主事件循环，其中包含UIApplication的Runloop来开始处理事件。

}

**UIApplication**：

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guesture。

**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。