作者 抱紧我的小鲤鱼 (/users/8c1cc9143ec6) 2016.11.16 11:48*
写了9388字，被49人关注，获得了60个喜欢
(/users/8c1cc9143ec6)

iOS多线程-从不会到熟练使用

字数5663 阅读1750 评论5 喜欢42

前言

多线程的东西好久没弄过了,项目里面用不到,慢慢都忘记了,最近不太忙,把这些东西重新拿出来弄一下,做下复习.这次先把api的东西捋一遍,下次再详说比较深层点的东西.

NSThead

先说几个点

> - NSThread 创建一个线程相对来说还是比较方便的
> - NSThread 管理多个线程比较困难,所以不太推荐使用
> - 苹果说了,现在推荐用GCD和NSOperation,也就是说其他的都不让(建议)用了
> - [NSTread currentThread] 跟踪任务所在线程,适用于NSTread,NSOperation,和GCD
> - 使用NSThread 的线程,不会自动添加autoreleasepool

iOS中隐式中创建线程的方法(NSObject的方法):

```
简/(NSObject)
//1.waitUntilDone：在主线程中运行方法，wait表示是否阻塞这个方法的调用，如果为yes则等待主线程中的运行方法结束。一般可
//2. modes:关于这个参数,暂时不做讲解,下次再说,这里先说一下,如果modes参数为kCFRunLoopCommonModes的话可以结局滑动过
//3.此方法不能够自动回收线程，如果并发数量多，会建立大量子线程。
- (void)performSelector:(SEL)aSelector onThread:(NSThread *)thr withObject:(nullable id)arg waitUnti
- (void)performSelector:(SEL)aSelector onThread:(NSThread *)thr withObject:(nullable id)arg waitUnti
- (void)performSelectorOnMainThread:(SEL)aSelector withObject:(nullable id)arg waitUntilDone:(BOOL)wa
- (void)performSelectorOnMainThread:(SEL)aSelector withObject:(nullable id)arg waitUntilDone:(BOOL)wa
- (void)performSelectorInBackground:(SEL)aSelector withObject:(nullable id)arg
//延迟执行这里,用到runloop的东西,不太懂runloop的同学,可以去学习一下,我下次也会说到
- (void)performSelector:(SEL)aSelector withObject:(nullable id)anArgument afterDelay:(NSTimeInterval)
- (void)performSelector:(SEL)aSelector withObject:(nullable id)anArgument afterDelay:(NSTimeInterval)
//取消线程队列中还没有执行的方法
+ (void)cancelPreviousPerformRequestsWithTarget:(id)aTarget;
+ (void)cancelPreviousPerformRequestsWithTarget:(id)aTarget selector:(SEL)aSelector object:(nullable
```

NSThread各个属性的意义

```
@property double threadPriority //线程优先级（double）0.0~1.0,默认0.5,优先级和执行顺序成正比
@property (nullable, copy) NSString *name  //线程名字
@property NSUInteger stackSize  //线程栈大小,默认主线程1m ,子线程512k,次属性可读写,但是写入大小必须为4k的倍数,最
@property (readonly) BOOL isMainThread // 是否是主线程
@property (readonly, getter=isExecuting) BOOL executing  //是否正在执行
@property (readonly, getter=isFinished) BOOL finished  //是否已经完成
@property (readonly, getter=isCancelled) BOOL cancelled  //是否已经取消
```

NSThead类方法,作用域:当前线程

```
+ (NSThread *)currentThread; //返回当前线程
+ (void)detachNewThreadSelector:(SEL)selector toTarget:(id)target withObject:(nullable id)argument; /
+ (void)sleepUntilDate:(NSDate *)date;//休眠到什么时候（具体日期）
+ (void)sleepForTimeInterval:(NSTimeInterval)ti; //休眠一段时间单位秒
+ (void)exit; //结束当前线程
+ (double)threadPriority; //返回当前线程优先级
+ (BOOL)setThreadPriority:(double)p; //设置当前线程优先级 0.0~1.0
+ (NSArray<NSNumber *> *)callStackReturnAddresses //返回当前线程访问的堆栈信息
+ (NSArray<NSString *> *)callStackSymbols //返回一堆十六进制的地址
+ (BOOL)isMainThread //返回当前线程是否是主线程
+ (NSThread *)mainThread //返回主线程
```

NSThead实例方法

```
简(instancetype)init //总共五个实例方法中没有给NSThread 加 selector 的方法，那这个方法是干什么用的呢？目
- (instancetype)initWithTarget:(id)target selector:(SEL)selector object:(nullable id)argument //通过s
- (void)main //主方法，用于子类继承重写
- (void)start //开始线程
- (void)cancel //取消线程
```

一些注意点:

- 在子类重写父类的方法中，start 方法先于main方法执行；
- 线程中的自动释放池：

  @autoreleasepool{}自动释放池

  主线程中是有自动释放池的，使用gcd 和nsoperation 也会自动添加自动释放池，但是nsthread 和nsobject 不会，如果在后台线程中创建了autorelease的对象，需要使用自动释放池，否则会出现内存泄露

  当自动释放池销毁时，对池中的所有对象发送release 消息，清空自动释放池

  当所有autorelease 的对象，在出了作用域之后，会自动添加到（最近一次创建的自动释放池中）自动释放池中

NSOperation

基本属性:

```
@property (nullable, copy) NSString *name  //该操作的名称
@property (readonly, getter=isCancelled) BOOL cancelled; // 该操作是否已经取消
@property (readonly, getter=isExecuting) BOOL executing;  //该操作是否正在执行
@property (readonly, getter=isFinished) BOOL finished;  //该操作是否已经完成
@property (readonly, getter=isConcurrent) BOOL concurrent;   //该操作是否是并行操作
@property (readonly, getter=isAsynchronous) BOOL asynchronous   //该操作是否是异步的
@property (readonly, copy) NSArray<NSOperation *> *dependencies;  //和该操作有关的依赖关系
@property NSOperationQueuePriority queuePriority;  //该操作的优先级
@property (nullable, copy) void (^completionBlock)(void)   //该操作的完成回调block
@property double threadPriority  // 该操作的优先级
```

实例方法:

```
- (void)start; //开始方法，被加入队列或手动开始的时候会被调用
- (void)main; // 队列的主方法，start 后执行，子类应重写此方法相比于start方法
- (void)cancel; // 取消操作
- (void)addDependency:(NSOperation *)op;  //添加依赖，依赖只是设置先后执行的顺序关系，可以跨队列依赖，不可以循环
- (void)removeDependency:(NSOperation *)op;  //移除依赖关系
- (void)waitUntilFinished  //
```

子类NSInvocationOperation

属性:

```
@property (readonly, retain) NSInvocation *invocation; //此队列的NSInvocation参数，注意此参数只读
@property (nullable, readonly, retain) id result; //
```

实例方法:

```
- (nullable instancetype)initWithTarget:(id)target selector:(SEL)sel object:(nullable id)arg;//通过@s
- (instancetype)initWithInvocation:(NSInvocation *)inv //通过NSInvocation初始化
```

一些注意点:

> - 调用方式,可以调用start方法,也可以添加到队列调用操作但是:调用start和添加到队列不可以同时使用

```
NSInvocationOperation *op = [[NSInvocationOperation alloc] initWithTarget:self
                                                                 selector:@selector(test1)
                                                                   object:@"Invocation"];
```

添加到队列方式启动

```
NSOperationQueue *queue = nil;
[queue addOperation:op];
```

调用start方式启动

```
[op start];
```

> - 调用start 开启任务，会在 当前线程 开启任务，如果几个操作一起调用start 则在 当前线程串行 完成任务
> - 把操作放到队列queue中开启任务会在 其他线程 去执行任务

子类NSInvocationOperation

属性:

```
@property (readonly, copy) NSArray<void (^)(void)> *executionBlocks; //此操作的所有blcok
```

类方法:

```
+ (instancetype)blockOperationWithBlock:(void (^)(void))block;// 通过block初始化操作，该block没有参数没有
```

实例方法:

```
- (void)addExecutionBlock:(void (^)(void))block;//添加block，该block没有参数没有返回值
```

一些注意点:

- 关于循环引用:
  (前者)我们平时用的适合基本类和队列和操作的关系为 self -(持有)->queue -(持有)->block -(持有)->self 这种关系不会造成循环引用
  (后者)self-(持有)->block -(持有)->self 这种会造成循环引用

- 下面分别解释两者区别,以下就称之为前者和后者:前者为什么不会造成循环引用? 因为self持有queue ,然后queue 持有block ,因为queue 执行每个block都是一个线程,线程是一条直线,这个线程执行完了这个block也就随之消失了,所以这时候block所引用的self也会随之消失.当queue里的任务执行完了,那持有关系就会变成 self 持有queue而已,后面就没有了.然而后者.self持有block ,block持有self,任务执行完后block还在,那么它引用的self也还在,这时候就造成了循环引用,但是如果给前者的线程都加上runloop(也就是把线程的线性改为环性，也就是不让线程执行完任务就死掉。主线程系统默认创建runloop，子线程默认不创建)那么前者也会造成循环引用.
  (1).单纯在操作中使用(也就是操作和self之前没有引用关系)self不会造成循环引用
  (2).如果self对象持有操作对象的引用,同时操作对象中又直接访问了self时会造成循环引用.
  (3).只有self直接强引用block 才会出现循环引用.
  (4).block 的管理以及线程的创建和销毁是由队列负责的,直接再block中使用self没有关系.
  (5).weakSelf 也不是可以在任何block中都能使用,详情去讯息runloop.

NSOperationQueue

属性:

```
@property (readonly, copy) NSArray<__kindof NSOperation *> *operations; //当前队列的所有操作
@property (readonly) NSUInteger operationCount;// 当前队列的所有操作的数量
@property NSInteger maxConcurrentOperationCount; //设置最大并发量
@property (getter=isSuspended) BOOL suspended; //挂起任务和开始任务，挂起的队列不会影响已执行中和执行完的任务，
@property (nullable, copy) NSString *name ;//队列的名字(不只是操作,队列也可以有名字)
@property (nullable, assign /* actually retain */) dispatch_queue_t underlyingQueue //
```

实例方法:

```
- (void)addOperation:(NSOperation *)op; //添加操作
- (void)addOperations:(NSArray<NSOperation *> *)ops waitUntilFinished:(BOOL)wait;//批量添加操作
- (void)addOperationWithBlock:(void (^)(void))block;//添加操作块
- (void)cancelAllOperations; //取消所有操作，已经执行的不会被取消,这时候会设置所有子操作对象的isCanceled 为yes,
- (void)waitUntilAllOperationsAreFinished; //暂时不知道这个方法的作用
```

类方法

```
+ (nullable NSOperationQueue *)currentQueue;//返回当前线程所在的队列
+ (NSOperationQueue *)mainQueue;//返回主队列
```

一些注意点:

- 所有添加到非主队列执行的操作都将在非主线程执行
- 自定义NSOperation:

  原理:重写start 或 main 方法,main方法由系统自动调用(注意 main方法内一定要用 @autoreleasepool)

  自定义操作 通过start 开始会在当前线程执行操作，通过添加到队列的线程会在其他线程执行

  父类的 setCompletionBlock属性设置以后会被系统自动调用,但是调用的回调block 不会是主线程也不会是当前线程而是其他线程(所以如果要用系统的回掉方法一定要手动添加在主线程回调的方法)这个block不可以传递参数

  关于自定义操作的取消,队列可以取消全部操作,操作也可以取消自己,但是只可以取消还没有开始的操作,如果操作已经开始了就不能取消了,因为取消的原理就是判断操作的属性 isCanceled ,操作在执行时会判断这个值,没有开始的操作判断操作被取消了就不会执行了,但是对于已经执行了的操作只能手动代码改,一般情况下会在自定义操作的main方法里和合适地方(一般都是回调方法前面)判断isCanceled的值,即使是操作已经执行完了,在这时候也可以回调失败方法或者是直接不回调造成正在执行的操作也被取消了的假象.

关于NSOperation的子类在NSOperationQueue中执行完无法释放的问题

简感谢简友_JB (http://www.jianshu.com/users/61457fc80af9)提出的问题,原文为: 你的NSOperation dealloc了么？并发数真的生效了么？ (http://www.jianshu.com/p/f81421e8459a)

问题是这样的: NSOperation的子类创建出来并且执行完任务以后没有被释放.然后问题的具体描述和解决办法上文已经写的很清楚啦,我就不重复一遍了.
然后说一下我对这个问题的产生的理解:

- 看完上面还是不太理解的同学可以在AFNetworking (https://github.com/AFNetworking/AFNetworking)或者SDWebImage (https://github.com/rs/SDWebImage)里面搜 `willChangeValueForKey` 看下大神是怎么用的.
- NSOperation基于GCD,所以它的很多特性基本都是GCD的特性,比如控制最大并发量,应该就是封装的信号量(因为没有源码所以不能百分之百确定它就是用的信号量的原理),所以它应该和GCD的特性是一样的,在使用的时候考虑GCD的原理,可以适当避免一些错误.

GCD

先列一下GCD中: 队列，同步任务，异步任务，线程 之间的关系,验证代码就不贴这里了,有兴趣的同学可以挨个验证一下:

简

- 串行队列(DISPATCH_QUEUE_SERIAL)的同步任务:(dispatch_sync)会在当前线程上依次执行串行队列各个任务，并不会创建新的线程。
- 串行队列(DISPATCH_QUEUE_SERIAL)的异步任务(dispatch_async)会在不是主线程的另外一个线程(注意是 一个 线程)上依次执行串行队列中的各个任务。

  这个是很常用的一种方式,比如:从网络上上现在一个图片然后处理曝光，滤镜。。等。
- 并行队列(DISPATCH_QUEUE_CONCURRENT)的同步任务(dispatch_sync)会在当前线程上依次执行并行队列中的各个任务，并不会创建新的线程.

  这点和串行队列的同步任务效果相同.
- 并行队列(DISPATCH_QUEUE_CONCURRENT)的异步任务(dispatch_async)会给 当前线程之外 的每一个任务都开启一个线程（因为队列中的所有任务都是异步的）分别执行每个任务.每个任务都是从头开始的.哪个任务先结束由任务本身决定,但是系统都会给每一个任务一个单独的线程去执行。
- 全局队列(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0))的同步任务(dispatch_sync)会在主线程上依次执行全局队列中的各个任务,并不会创建新的线程.
- 全局队列(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0))的异步任务(dispatch_async)会给当前线程之外的每一个任务都开启一个线程(因为队列中的所有任务都是异步的)分别执行每个任务.
- 主队列(dispatch_get_main_queue())的同步任务(dispatch_sync)会阻塞线程
- 主队列(dispatch_get_main_queue())的异步任务.(dispatch_async)会在主线程依次执行各个任务.

总结:

- 串行队列,并行队列同步任务会在当前线程依次执行各个任务
- 主队列的同步任务会阻塞线程.
- 全局队列的同步任务会在主线程依次执行各个任务.
- 串行队列的异步任务会在当前线程外的另外一个线程依次执行任务.并行队列和全局队列的异步任务会给每一个任务分配一个线程去执行,因为执行效果相同所以猜测全局队列是并行队列.
- 主队列的同步任务会阻塞线程,主队列的异步任务会在主线程上依次执行各个任务.
- 串行队列的异步任务既可以保证效率(它会开另外一个线程)又可以实现并发.

全局队列,主队列,并行队列,串行队列区别和联系:

简

- 全局队列为系统创建,不需要自己创建,拿过来就用,用着方便
- 并行队列和全局队列执行效果相同
- 全局队列没有名称，调试时，无法确认准确队列
- 每一个应用程序都只有一个主线程
- iOS开发中所有UI更新工作都必须在主线程上执行,因为线程安全需要消耗性能,苹果为了保留性能优势所以放弃了线程安全 ,所以改UI都要在主线程上.
- 异步任务任务在主线程上是保持队形的，也就是异步是在主线程一个线程上执行的主线程不可以执行同步任务，因为同步任务是要等到上一个线程结束后才会执行,但是主线程一直不会停止当前的任务，所以它不会去执行一个同步任务
- 无论什么队列和什么任务,线程的创建和回收不需要程序员参与,线程的创建和回收工作是由队列负责的

dispatch_barrier_async栅栏函数

- `dispatch_barrier_async` 必须使用自定义队列,否则执行效果和全局队列一致,也就是说barrier任务会和其他任务一样被当做一个普通任务去执行而不是等其他所有之前的并行任务执行完后才最后执行.
- 经过大量测试不管是串行队列或者是并行队列的异步任务(为什么不是同步任务呢？因为同步任务都会按任务的顺序执行这样就不会有等所有线程执行完以后再执行barrier里的任务这一说了)dispatch_barrier_async会在最后一个任务执行的线程执行本任务.
- `dispatch_barrier_sync` 会在主线程中执行任务。

```
  NSLog(@">>>>>111:%@",[NSThread mainThread]);
dispatch_queue_t concurrentQueue = dispatch_queue_create("my.concurrent.queue", DISPATCH_QUEUE_C
dispatch_async(concurrentQueue, ^(){
    NSLog(@"dispatch-1:%@",[NSThread currentThread]);
});
dispatch_async(concurrentQueue, ^(){
    NSLog(@"dispatch-2:%@",[NSThread currentThread]);
});
dispatch_barrier_async(concurrentQueue, ^(){
    NSLog(@"dispatch-barrier:%@",[NSThread currentThread]);
});
dispatch_async(concurrentQueue, ^(){
    NSLog(@"dispatch-3:%@",[NSThread currentThread]);
});
dispatch_async(concurrentQueue, ^(){
    NSLog(@"dispatch-4:%@",[NSThread currentThread]);
});
```

打印结果

```
>>>>>1:<NSThread: 0x7fc023c08bd0>{number = 1, name = main}
>>>>>111:<NSThread: 0x7fc023c08bd0>{number = 1, name = main}
dispatch-1:<NSThread: 0x7fc023f23780>{number = 2, name = (null)}
dispatch-2:<NSThread: 0x7fc025b00000>{number = 3, name = (null)}
dispatch-barrier:<NSThread: 0x7fc025b00000>{number = 3, name = (null)}
dispatch-4:<NSThread: 0x7fc023f23780>{number = 2, name = (null)}
dispatch-3:<NSThread: 0x7fc025b00000>{number = 3, name = (null)}
```

dispatch_group_t队列组

> - 队列组只能异步执行，没有同步执行的函数.
> - dispatch_group_wait 阻塞组任务一定时间:参数是一个dispatch_group_t和dispatch_time_t。dispatch_group_t是需要阻塞的组,dispatch_time_t是需要阻塞的时间.
> - 线程组的用法和原理:可以给一个任务组设置一个阻塞(超时)时间,如果给一个任务组的前两个任务设置一个超时时间那么如果这前两个任务提前执行结束了,也就是在设定的阻塞时间内完成了任务,那么这个阻塞就会放行下面的任务去执行任务,如果超过了超时时间那么这个阻塞也会放行,让下面的任务可以执行.阻塞时间并不能杀死任务,也就是说如果你设定了阻塞(超时)时间,阻塞时间前的任务在阻塞放行前并没有执行完成,然后任务就被放行了,系统继续往下执行,但是这时候阻塞时间前的任务也会执行完成如果你不手动杀死它的话.

队列组可以手动管理也可以交给系统管理.

> 系统管理队列组:

```
dispatch_group_async(group, queue, ^{

});
```

> 手动管理队列组:dispatch_group_enter和dispatch_group_leave(这两个必须成对使用)

```
dispatch_group_enter(group);
    dispatch_async(queue, ^{

        dispatch_group_leave(group);
    });
```

以上的两段代码是相同的.

dispatch_group_notify队列组监听

在并行队列里面可以同时并发执行多个任务,但是有时我们可能需要等到全部任务都执行完成以后才执行另外一个任务,这时候就需要用到GCD的这一个特性.
先看代码:

```
    dispatch_queue_t queue = dispatch_queue_create("com.test", DISPATCH_QUEUE_CONCURRENT);
    dispatch_group_t group = dispatch_group_create();
    dispatch_group_async(group, queue, ^{
        NSLog(@">>>>>>111:%@",[NSThread currentThread]);
        [NSThread sleepForTimeInterval:1];
    });
    dispatch_group_async(group, queue, ^{
        NSLog(@">>>>>>222:%@",[NSThread currentThread]);
        [NSThread sleepForTimeInterval:2];
    });
    dispatch_group_async(group, queue, ^{
        NSLog(@">>>>>>333:%@",[NSThread currentThread]);
        [NSThread sleepForTimeInterval:5];
    });

    dispatch_group_notify(group, queue, ^{
        NSLog(@">>>>>>444:%@",[NSThread currentThread]);
    });
```

打印日志:

```
>>>>>>111:<NSThread: 0x7fd830414c80>{number = 2, name = (null)}
>>>>>>222:<NSThread: 0x7fd830405780>{number = 4, name = (null)}
>>>>>>333:<NSThread: 0x7fd830586c40>{number = 3, name = (null)}
>>>>>>444:<NSThread: 0x7fd830586c40>{number = 3, name = (null)}
```

- 可以看到最后一条打印是等到最后一个任务执行完成以后才执行最后监听方法
- 并行队列和全局队列：每一个任务会分配一个线程，所有任务并行执行，最后一个任务执行完毕后会调用监测方法dispatch_group_notify，监测方法在最后一个任务执行的线程中执行
- 串行队列：所有任务都在主线程外的另外一个线程执行，并且任务有序执行，最后一个任务执行完毕后会调用监测方法dispatch_group_notify监测方法在最后一个任务执行的线程中执行
- 主队列：所有任务都在主线程执行，并且任务有序执行，最后一个任务执行完毕后会调用监测方法dispatch_group_notify监测方法在最后一个任务执行的线程中执行（也就是主线程）

dispatch_group_notify总结:此方法用于监听多并发队列,非并发无法监听,也没什么意义.

dispatch_semaphore_t信号量 和 dispatch_group_wait队列组等待
信号量是用于控制并发数量的，所以只用在全局队列和并行队列中.

- 发信号dispatch_semaphore_signal(dispatch_semaphore_t)

  一个参数，要发信号的信号量对象（dispatch_semaphore_t）这个函数会使传入的信号量dsema的值加1，所以会用在线程完成后的最后面，告诉系统当前任务已经完了，你可以把这个线程腾出来给其他任务用了.

  返回值（long）当返回值为0时表示当前并没有线程等待其处理的信号量，其处理的信号量的值加1即可。当返回值不为0时，表示其当前有（一个或多个）线程等待其处理的信号量，并且该函数唤醒了一个等待的线程（当线程有优先级时，唤醒优先级最高的线程；否则随机唤醒）

- 创建信号量（dispatch_semaphore_t）

  dispatch_semaphore_create(5).

  创建信号量需要一个long型参数,这个数字(必须大于等于0)标记你设置的最大线程并发量,也就是说在并发队列里你最多允许同时创建执行5个线程去执行任务.其他任务需要在正在执行的任务完成后腾出新的线程再去执行.当然这还取决于你设置的等待时间的时长:如果设置的等待时间为永远等待(dispatch_time_t: `DISPATCH_TIME_FOREVER`)那么它会按前面说的去执行.但是如果你设置的超时时间小于你执行任务需要的时间的话.系统会创建新的线程去执行你在等待的任务.尽管这样创建的线程会大于你设置的最大并发量.

- 设置信号量并发数的工作原理:

  这个函数会使传入的信号量dsema的值减1.这个函数的作用是这样的,如果dsema信号量的值大于0.该函数所处线程就继续执行下面的语句.并且将信号量的值减1.如果desema的值为0，那么这个函数就阻塞当前线程等待timeout(注意timeout的类型为dispatch_time_t，不能直接传入整形或float型数),如果等待的期间desema的值被dispatch_semaphore_signal函数加1了,且该函数(即dispatch_semaphore_wait)所处线程获得了信号量，那么就继续向下执行并将信号量减1.如果等待期间没有获取到信号量或者信号量的值一直为0,那么等到timeout时,其所处线程自动执行其后语句.

信号量等待: `dispatch_semaphore_wait(dispatch_semaphore_t, dispatch_time_t)`

简设置等待需要两个参数:

(a)需要等待的信号量(dispatch_semaphore_t);

(b)等待时间(dispatch_time_t)这个时间表示你可以为执行任务所等待的时间,一般都设置为永远等待(DISPATCH_TIME_FOREVER)这样的话如果前面的任务没有执行完就一直等待到有任务执行完然后腾出来线程以后去执行任务.另外还可以设置一个时间(dispatch_time_t)比如说是1秒，这就意味这一秒以后如果前面的任务执行完了那么就按正常走,否则的话就新建一个另外的线程去执行等的着急的任务.

返回值(long)

如果是0就是等待成功(也就是没有超时)否则为失败(也就是超时了)

关于dispatch_semaphore_t:

在设置timeout时,比较有用的两个宏: `DISPATCH_TIME_NOW` 和 `DISPATCH_TIME_FOREVER` . `DISPATCH_TIME_NOW` 表示当前. `DISPATCH_TIME_FOREVER` 表示遥远的未来.一般可以直接设置timeout为这两个宏其中的一个,或者自己创建一个dispatch_time_t类型的变量.

创建dispatch_time_t类型的变量有两种方法,dispatch_time和dispatch_walltime.利用创建dispatch_time创建dispatch_time_t类型变量的时候一般也会用到这两个变量.

dispatch_time的声明如下:

dispatch_time_t dispatch_time(dispatch_time_t when, int64_t delta)； 其参数when需传入一个dispatch_time_t类型的变量,和一个delta值.表示when加delta时间就是timeout的时间.例如:dispatch_time_t t = dispatch_time(DISPATCH_TIME_NOW,1*1000*1000*1000);表示当前时间向后延时一秒为timeout的时间.

代码示例:

```
dispatch_group_t group = dispatch_group_create();
dispatch_semaphore_t semaphore = dispatch_semaphore_create(3);

dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
for (int i = 0; i < 10; i++){

    //dispatch_time_t wait = dispatch_time(DISPATCH_TIME_NOW, (int64_t)(3 * NSEC_PER_SEC));
    dispatch_time_t wait = DISPATCH_TIME_FOREVER;
    dispatch_semaphore_wait(semaphore, wait);
    dispatch_group_async(group, queue, ^{
        NSLog(@"----------------: %i :%@",i,[NSThread currentThread]);
        [NSThread sleepForTimeInterval:1];
        dispatch_semaphore_signal(semaphore);
    });

}
dispatch_group_notify(group, queue, ^{
    NSLog(@"xxxxxxxxxxxxx");
});
```

打印日志:

```
多线程总结[86647:6419764] ----------------: 0 :<NSThread: 0x7fc011d1ac90>{number = 2, name = (n
多线程总结[86647:6419757] ----------------: 1 :<NSThread: 0x7fc011f0ac80>{number = 3, name = (null)}
多线程总结[86647:6419768] ----------------: 2 :<NSThread: 0x7fc011c0f880>{number = 4, name = (null)}
多线程总结[86647:6419764] ----------------: 5 :<NSThread: 0x7fc011d1ac90>{number = 2, name = (null)}
多线程总结[86647:6419768] ----------------: 3 :<NSThread: 0x7fc011c0f880>{number = 4, name = (null)}
多线程总结[86647:6419757] ----------------: 4 :<NSThread: 0x7fc011f0ac80>{number = 3, name = (null)}
多线程总结[86647:6419757] ----------------: 7 :<NSThread: 0x7fc011f0ac80>{number = 3, name = (null)}
多线程总结[86647:6419764] ----------------: 6 :<NSThread: 0x7fc011d1ac90>{number = 2, name = (null)}
多线程总结[86647:6419768] ----------------: 8 :<NSThread: 0x7fc011c0f880>{number = 4, name = (null)}
多线程总结[86647:6419768] ----------------: 9 :<NSThread: 0x7fc011c0f880>{number = 4, name = (null)}
多线程总结[86647:6419768] xxxxxxxxxxxxxx
```

代码分析:

> 首先这里面创建了一个组,这个组放在这里只是告诉你可以这么用,它不会影响信号量的功能.
> 这里创建了一个并发为3的信号量然而for循环是10个任务,那么理论上10个任务会创建十个线程去执行，但是你信号量为3所以只能创建3个线程去执行前面10个任务，然后等待前3个任务执行完成了腾出来新的线程再去执行等待执行的.所以下面的线程 number 最大是4,当然这前提是你设置的超时时间大于任务执行完的时间.如果设置超时时间是0.1秒的话，任务等超时了还是会开启另外的线程去执行任务，这样就达不到控制并发量的要求了.

关于信号量举个小栗子帮助大家理解下:

> 一般可以用停车来比喻：
> 停车场剩余4个车位，那么即使同时来了四辆车也能停的下。如果此时来了五辆车，那么就有一辆需要等待。信号量的值就相当于剩余车位的数目，dispatch_semaphore_wait函数就相当于来了一辆车，dispatch_semaphore_signal就相当于走了一辆车。停车位的剩余数目在初始化的时候就已经指明了（dispatch_semaphore_create（long value）），调用一次dispatch_semaphore_signal，剩余的车位就增加一个；调用一次dispatch_semaphore_wait剩余车位就减少一个；当剩余车位为0时，再来车（即调用dispatch_semaphore_wait）就只能等待。有可能同时有几辆车等待一个停车位。有些车主没有耐心，给自己设定了一段等待时间，这段时间内等不到停车位就走了，如果等到了就开进去停车。而有些车主就像把车停在这，所以就一直等下去。

NSOperation与GCD的对比

- 将任务（block）添加到队列(串行/并发/主队列)，并且指定任务执行的函数(同步/异步)
- GCD是底层的C语言构成的API
- iOS 4.0 推出的，针对多核处理器的并发技术
- 在队列中执行的是由 block 构成的任务，这是一个轻量级的数据结构
- 要停止已经加入 queue 的 block 需要写复杂的代码
- 需要通过 Barrier 或者同步任务设置任务之间的依赖关系
- 只能设置队列的优先级
- 高级功能：
  - 一次性 once
  - 延迟操作 after
  - 调度组

- NSOperation
  - 核心概念：把操作(异步)添加到队列(全局的并发队列)
  - OC 框架，更加面向对象，是对 GCD 的封装
  - iOS 2.0 推出的，苹果推出 GCD 之后，对 NSOperation 的底层全部重写
  - Operation作为一个对象，为我们提供了更多的选择
  - 可以随时取消已经设定要准备执行的任务，已经执行的除外
  - 可以跨队列设置操作的依赖关系
  - 可以设置队列中每一个操作的优先级
  - 高级功能：
    - 最大操作并发数(GCD不好做)
    - 继续/暂停/全部取消
    - 跨队列设置操作的依赖关系

iOS的常用多线程使用方式基本都在这里了.但是这里没有说线程安全,原理,线程锁,runloop等和线程相关的东西的.先这样,下次再详细说说更深点的东西.
上面的内容是很久以前笔记记下的东西,现在没重新做验证,如有错误,望请指正,多谢多谢.

如果觉得我的文章对您有用，请随意打赏。您的支持将鼓励我继续创作！

¥ 打赏支持

♡ 喜欢 4

分享到微博 分享到微信
更多分享

5条评论 （ 按时间正序 · 按时间倒序 · 按喜欢排序 ）

添加新评论

_JD (/users/61457fc80af9)
2楼 · 2016.11.17 09:03 (/p/ac11fe7ef78c/comments/5729753#comment-5729753)
(/users/61457fc80af9)

楼主好，前两天研究了一下nsoperation关于并发数以及使用的，胡乱写了下，感觉你写的比较全，你看一下有没有什么值得补充到你文章里的。 http://www.jianshu.com/p/f81421e8459a (http://www.jianshu.com/p/f81421e8459a)

♡ 喜欢(0)　回复

> 抱紧我的小鲤鱼 (/users/8c1cc9143ec6)： @_JD (/users/61457fc80af9) 好的,我有时间尽快看一下,如有补充我就补充一下,并加上文章来源,多谢多谢~
> 2016.11.18 10:08 (/p/ac11fe7ef78c/comments/5743009#comment-5743009)　回复
>
> 添加新回复

Coder_xd (/users/eaf33d493a1d)
3楼 · 2016.11.19 12:31 (/p/ac11fe7ef78c/comments/5774546#comment-5774546)
(/users/eaf33d493a1d)

楼主好,看了你写的关于线程的知识点,感觉总结的太好了,感谢分享!

♡ 喜欢(0)　回复

Xml_Sw (/users/7e4ad4b37883)
4楼 · 2016.11.21 19:37 (/p/ac11fe7ef78c/comments/5835147#comment-5835147)
(/users/7e4ad4b37883)

楼主，初学多线程，不知道哪种情况下用多线程，关于什么样的情况下，怎样使用，能不能出一篇供我们初学者学习的文章，拜托……

♡ 喜欢(0)　回复

> 抱紧我的小鲤鱼 (/users/8c1cc9143ec6)： @Xml_Sw (/users/7e4ad4b37883) 一般情况来说，系统在做需要耗时任务或者和ui无关的任务时会用。但是具体情况还需要据情形而定，现在已经写的很简单了，只是没有写demo，我有时间补充下demo吧

2016.11.22 09:41 (/p/ac11fe7ef78c/comments/5848029#comment-5848029)

回复

添加新回复

加载更多 (/notes/6980119/comments?max_id=5848029&order=asc&page=2)

写下你的评论...

发 表 ⌘+Return 发表

被以下专题收入，发现更多相似内容：

程序员 (/collection/NEt52a)

如果你是程序员，或者有一颗喜欢写程序的心，喜欢分享技术干货、项目经验、程序员日常囧事等等，欢迎投稿《程序员》专题。 专题主编：小...

28594篇文章 (/collection/NEt52a) · 240625人关注

正在关注 (/collections/16/unsubscribe)

(/collection/NEt52a)

首页投稿 (/collection/bDHhpK)

玩转简书的第一步，从这个专题开始。 想上首页热门榜么？好内容想被更多人看到么？来投稿吧！如果被拒也不要灰心哦～入选文章会进一个队...

119331篇文章 (/collection/bDHhpK) · 142447人关注

添加关注 (/collections/47/subscribe)

(/collection/bDHhpK)

iOS Developer (/collection/3233d1a249ca)

分享 iOS 开发的知识，解决大家遇到的问题，讨论iOS开发的前沿，欢迎大家投稿。

15926篇文章 (/collection/3233d1a249ca) · 28702人关注

正在关注 (/collections/1276/unsubscribe)

(/collection/3233d1a249ca)