



# 关于iOS多线程，你看我就够了



伯恩的遗产

关注

 19

2015.07.29 00:37:12

字数 7,439

阅读 249,325

在这篇文章中，我将为你整理一下 iOS 开发中几种多线程方案，以及其使用方法和注意事项。当然也会给出几种多线程的案例，在实际使用中感受它们的区别。还有一点需要说明的是，这篇文章将会使用 `Swift` 和 `Objective-c` 两种语言讲解，双语幼儿园。OK，let's begin!

## 概述

这篇文章中，我不会说多线程是什么、线程和进程的区别、多线程有什么用，当然我也不会说什么是串行、什么是并行等问题，这些我们应该都知道的。

在 iOS 中其实目前有 `4` 套多线程方案，他们分别是：

- Pthreads
- NSThread
- GCD
- NSOperation & NSOperationQueue

所以接下来，我会一一讲解这些方案的使用方法和一些案例。在将这些内容的时候，我也会顺带说一些多线程周边产品。比如：`线程同步`、`延时执行`、`单例模式` 等等。

## Pthreads

其实这个方案不用说的，只是拿来充个数，为了让大家了解一下就好了。百度百科里是这么说的：

POSIX线程（POSIX threads），简称Pthreads，是线程的POSIX标准。该标准定义了创建和操纵线程的一整套API。在类Unix操作系统（Unix、Linux、Mac OS X等）中，都使用Pthreads作为操作系统的线程。

简单地说，这是一套在很多操作系统上都通用的多线程API，所以移植性很强（然并卵），当然在 iOS 中也是可以的。不过这是基于 `c语言` 的框架，使用起来这酸爽！感受一下：

### OBJECTIVE-C

当然第一步要包含头文件

```
#import <pthread.h>
```

人工智能

硅谷讲师 |





伯恩的遗产

总资产20 (约

百度智能云

2020年度

爆款云服务器

消费满额送

立即预

然后创建线程，并执行任务

```
1 | - (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
2 |     pthread_t thread;
3 |     //创建一个线程并自动执行
4 |     pthread_create(&thread, NULL, start, NULL);
5 | }
6 |
7 | void *start(void *data) {
8 |     NSLog(@"%@", [NSThread currentThread]);
9 |
10 |     return NULL;
11 | }
```

打印输出：

```
2015-07-27 23:57:21.689 testThread[10616:2644653] <NSThread:
0x7fbb48d33690>{number = 2, name = (null)}
```

看代码就会发现他需要 **c语言函数**，这是比较蛋疼的，更蛋疼的是你需要手动处理线程的各个状态的转换即管理生命周期，比如，这段代码虽然创建了一个线程，但并没有销毁。

### SWIFT

很遗憾，在我目前的 **swift1.2** 中无法执行这套方法，原因是这个函数需要传入一个函数指针 **CFunctionPointer<T>** 类型，但是目前 swift 无法将方法转换成此类型。听说 **swift 2.0** 引入一个新特性 **@convention(c)**，可以完成 Swift 方法转换成 c 语言指针的。[在这里可以看到](#)

那么，**Pthreads** 方案的多线程我就介绍这么多，毕竟做 iOS 开发几乎不可能用到。但是如果你感兴趣的话，或者说想要自己实现一套多线程方案，从底层开始定制，那么可以去搜一下相关资料。

## NSThread

这套方案是经过苹果封装后的，并且完全面向对象的。所以你可以直接操控线程对象，非常直观和方便。但是，它的生命周期还是需要我们手动管理，所以这套方案也是偶尔用用，比如 **[NSThread currentThread]**，它可以获取当前线程类，你就可以知道当前线程的各种属性，用于调试十分方便。下面来看看它的一些用法。

### 创建并启动

- 先创建线程类，再启动

#### OBJECTIVE-C

```
1 | // 创建
2 | NSThread *thread = [[NSThread alloc] initWithTarget:self selector:@selector(run:) ob
3 |
4 | // 启动
5 | [thread start];
```

#### SWIFT

```
1 | //创建
2 | let thread = NSThread(target: self, selector: "run:", object: nil)
3 |
4 | //启动
5 | thread.start()
```

- 创建并自动启动

### OBJECTIVE-C

```
1 | [NSThread detachNewThreadSelector:@selector(run:) toTarget:self withObject:nil];
```

### SWIFT

```
1 | NSThread.detachNewThreadSelector("run:", toTarget: self, withObject: nil)
```

- 使用 NSObject 的方法创建并自动启动

### OBJECTIVE-C

```
1 | [self performSelectorInBackground:@selector(run:) withObject:nil];
```

### SWIFT

很遗憾 too! 苹果认为 `performSelector:` 不安全，所以在 Swift 去掉了这个方法。

Note: The performSelector: method and related selector-invoking methods are not imported in Swift because they are inherently unsafe.

## 其他方法

除了创建启动外，NSThread 还以很多方法，下面我列举一些常见的方法，当然我列举的并不完整，更多方法大家可以去类的定义里去看。

### OBJECTIVE-C

```
1 | //取消线程
2 | - (void)cancel;
3 |
4 | //启动线程
5 | - (void)start;
6 |
7 | //判断某个线程的状态的属性
8 | @property (readonly, getter=isExecuting) BOOL executing;
9 | @property (readonly, getter=isFinished) BOOL finished;
10 | @property (readonly, getter=isCancelled) BOOL cancelled;
11 |
12 | //设置和获取线程名字
13 | -(void)setName:(NSString *)n;
14 | -(NSString *)name;
15 |
16 | //获取当前线程信息
17 | + (NSThread *)currentThread;
18 |
19 | //获取主线程信息
```

```
20 + (NSThread *)mainThread;
21
22 //使当前线程暂停一段时间，或者暂停到某个时刻
23 + (void)sleepForTimeInterval:(NSTimeInterval)time;
24 + (void)sleepUntilDate:(NSDate *)date;
```

## SWIFT

Swift的方法名字和OC的方法名都一样，我就不浪费空间列举出来了。

其实，NSThread 用起来也挺简单的，因为它就那几种方法。同时，我们也只有在一些非常简单的场景才会用 NSThread, 毕竟它还不够智能，不能优雅地处理多线程中的其他高级概念。所以接下来要说的内容才是重点。

## GCD

**Grand Central Dispatch**，听名字就霸气。它是苹果为多核的并行运算提出的解决方案，所以会自动合理地利用更多的CPU内核（比如双核、四核），最重要的是它会自动管理线程的生命周期（创建线程、调度任务、销毁线程），完全不需要我们管理，我们只需要告诉干什么就行。同时它使用的也是 **c语言**，不过由于使用了 Block（Swift里叫做闭包），使得使用起来更加方便，而且灵活。所以基本上大家都使用 **GCD** 这套方案，老少咸宜，实在是居家旅行、杀人灭口，必备良药。不好意思，有点中二，咱们继续。

## 任务和队列

在 **GCD** 中，加入了两个非常重要的概念：**任务** 和 **队列**。

- 任务：即操作，你想要干什么，说白了就是一段代码，在 GCD 中就是一个 Block，所以添加任务十分方便。任务有两种执行方式：**同步执行** 和 **异步执行**，他们之间的区别是 **是否会创建新的线程**。

**同步执行**：~~只要是同步执行的任务，都会在当前线程执行，不会另开线程。~~

**异步执行**：~~只要是异步执行的任务，都会另开线程，在别的线程执行。~~

更新：

这里说的并不准确，**同步 (sync)** 和 **异步 (async)** 的主要区别在于会不会阻塞当前线程，直到 **Block** 中的任务执行完毕！

如果是 **同步 (sync)** 操作，它会阻塞当前线程并等待 **Block** 中的任务执行完毕，然后当前线程才会继续往下运行。

如果是 **异步 (async)** 操作，当前线程会直接往下执行，它不会阻塞当前线程。

- 队列：用于存放任务。一共有两种队列，**串行队列** 和 **并行队列**。

**串行队列** 中的任务会根据队列的定义 FIFO 的执行，一个接一个的先进先出的进行执行。

更新：放到串行队列的任务，GCD 会 **FIFO (先进先出)** 地取出来一个，执行一个，然后取下一个，这样一个一个的执行。

并行队列 中的任务 根据同步或异步有不同的执行方式。

更新：放到并行队列的任务，GCD 也会 **FIFO** 的取出来，但不同的是，它取出来一个就会放到别的线程，然后再取出来一个又放到另一个的线程。这样由于取的动作很快，忽略不计，看起来，所有的任务都是一起执行的。不过需要注意，GCD 会根据系统资源控制并行的数量，所以如果任务很多，它并不会让所有任务同时执行。

虽然很绕，但请看下表：

	同步执行	异步执行
串行队列	当前线程，一个一个执行	其他线程，一个一个执行
并行队列	当前线程，一个一个执行	开很多线程，一起执行

## 创建队列

- 主队列：这是一个特殊的 **串行队列**。什么是主队列，大家都知道吧，它用于刷新 UI，任何需要刷新 UI 的工作都要在主队列执行，所以一般耗时的任务都要放到别的线程执行。

```
1 //OBJECTIVE-C
2 dispatch_queue_t queue = ispatch_get_main_queue();
3
4 //SWIFT
5 let queue = ispatch_get_main_queue()
```

- 自己创建的队列：凡是自己创建的队列都是 **串行队列**。其中第一个参数是标识符，用于 DEBUG 的时候标识唯一的队列，可以为空。大家可以看xcode的文档查看参数意义。

更新：自己可以创建 **串行队列**，也可以创建 **并行队列**。看下面的代码（代码已更新），它有两个参数，第一个上面已经说了，第二个才是最重要的。第二个参数用来表示创建的队列是串行的还是并行的，传入 **DISPATCH\_QUEUE\_SERIAL** 或 **NULL** 表示创建串行队列。传入 **DISPATCH\_QUEUE\_CONCURRENT** 表示创建并行队列。

```
1 //OBJECTIVE-C
2 //串行队列
3 dispatch_queue_t queue = dispatch_queue_create("tk.bourne.testQueue", NULL);
4 dispatch_queue_t queue = dispatch_queue_create("tk.bourne.testQueue", DISPATCH_QUEUE_SERIAL);
5 //并行队列
6 dispatch_queue_t queue = dispatch_queue_create("tk.bourne.testQueue", DISPATCH_QUEUE_CONCURRENT);
7
8 //SWIFT
9 //串行队列
10 let queue = dispatch_queue_create("tk.bourne.testQueue", nil);
11 let queue = dispatch_queue_create("tk.bourne.testQueue", DISPATCH_QUEUE_SERIAL)
12 //并行队列
13 let queue = dispatch_queue_create("tk.bourne.testQueue", DISPATCH_QUEUE_CONCURRENT)
```

- 全局并行队列：这应该是唯一一个并行队列，只要是并行任务一般都加入到这个队列。这是系统提供的一个并发队列。



```
1 //OBJECTIVE-C
2 dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
3
4 //SWIFT
5 let queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)
```

## 创建任务

- 同步任务： ~~不会~~另开线程 改： 会阻塞当前线程 (SYNC)

### OBJECTIVE-C

```
1 dispatch_sync(<#queue#>, ^{
2     //code here
3     NSLog(@"%@", [NSThread currentThread]);
4 });
```

### SWIFT

```
1 dispatch_sync(<#queue#>, { () -> Void in
2     //code here
3     println(NSThread.currentThread())
4 })
```

- 异步任务： 会另开线程 改： 不会阻塞当前线程 (ASYNC)

### OBJECTIVE-C

```
1 dispatch_async(<#queue#>, ^{
2     //code here
3     NSLog(@"%@", [NSThread currentThread]);
4 });
```

### SWIFT

```
1 dispatch_async(<#queue#>, { () -> Void in
2     //code here
3     println(NSThread.currentThread())
4 })
```

### 更新：

为了更好的理解同步和异步，和各种队列的使用，下面看两个示例：

### 示例一：

以下代码在主线程调用，结果是什么？

```
1 NSLog("之前 - %@", NSThread.currentThread())
2 dispatch_sync(dispatch_get_main_queue(), { () -> Void in
3     NSLog("sync - %@", NSThread.currentThread())
4 })
5 NSLog("之后 - %@", NSThread.currentThread())
```

答案：

只会打印第一句： `之前 - <NSThread: 0x7fb3a9e16470>{number = 1, name = main}`，然后主线程就卡死了，你可以在界面上放一个按钮，你就会发现点不了了。

解释：

同步任务会阻塞当前线程，然后把 Block 中的任务放到指定的队列中执行，只有等到 Block 中的任务完成后才会让当前线程继续往下运行。

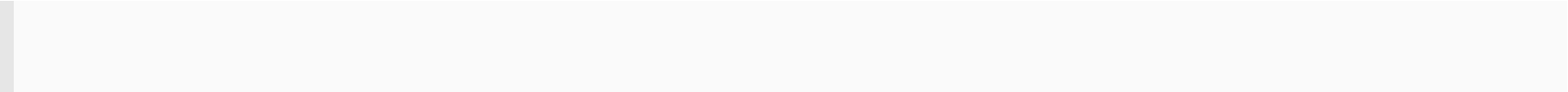
那么这里的步骤就是：打印完第一句后，`dispatch_sync` 立即阻塞当前的主线程，然后把 Block 中的任务放到 `main_queue` 中，可是 `main_queue` 中的任务会被取出来放到主线程中执行，但主线程这个时候已经被阻塞了，所以 Block 中的任务就不能完成，它不完成，`dispatch_sync` 就会一直阻塞主线程，这就是死锁现象。导致主线程一直卡死。

示例二：

以下代码会产生什么结果？

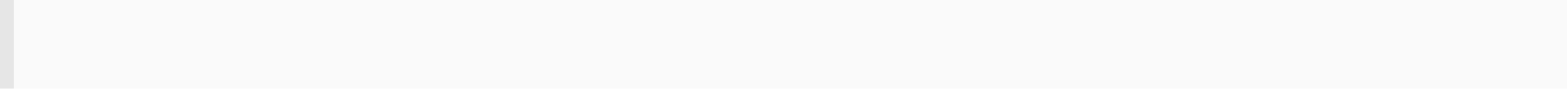
```
1 | let queue = dispatch_queue_create("myQueue", DISPATCH_QUEUE_SERIAL)
2 |
```

NSLog("之前 - %@", NSThread.currentThread())



```
1 | dispatch_async(queue, { () -> Void in
2 |     NSLog("sync之前 - %@", NSThread.currentThread())
3 |     dispatch_sync(queue, { () -> Void in
4 |         NSLog("sync - %@", NSThread.currentThread())
5 |     })
6 |     NSLog("sync之后 - %@", NSThread.currentThread())
```

}}



NSLog("之后 - %@", NSThread.currentThread())

```
1 | **答案： **
2 | 2015-07-30 02:06:51.058 test[33329:8793087] 之前 - <NSThread: 0x7fe32050dbb0>{number = 1,
3 | 2015-07-30 02:06:51.059 test[33329:8793356] sync之前 - <NSThread: 0x7fe32062e9f0>{number
4 | 2015-07-30 02:06:51.059 test[33329:8793087] 之后 - <NSThread: 0x7fe32050dbb0>{number = 1,
5 | 很明显 `sync - %@` 和 `sync之后 - %@` 没有被打印出来！这是为什么呢？我们再分析一下：
6 |
7 | >**分析： **
8 | 我们按执行顺序一步步来哦：
9 | 1. 使用 `DISPATCH_QUEUE_SERIAL` 这个参数，创建了一个 **串行队列**。
10 | 2. 打印出 `之前 - %@` 这句。
11 | 3. `dispatch_async` 异步执行，所以当前线程不会被阻塞，于是有了两条线程，一条当前线程继续往下打印出 `之
12 | 4. 注意，高潮来了。现在的情况和上一个例子一样了。`dispatch_sync` 同步执行，于是它所在的线程会被阻塞，一
13 |
14 |
15 | ### 队列组
16 |
17 | 队列组可以将很多队列添加到一个组里，这样做的好处是，当这个组里所有的任务都执行完了，队列组会通过一个方法通
18 |
19 | ##### OBJECTIVE-C
```

```
20
21 ```` objective-c
22 //1.创建队列组
23 dispatch_group_t group = dispatch_group_create();
24 //2.创建队列
25 dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
26
27 //3.多次使用队列组的方法执行任务，只有异步方法
28 //3.1.执行3次循环
29 dispatch_group_async(group, queue, ^{
30     for (NSInteger i = 0; i < 3; i++) {
31         NSLog(@"group-01 - %@", [NSThread currentThread]);
32     }
33 });
34
35 //3.2.主队列执行8次循环
36 dispatch_group_async(group, dispatch_get_main_queue(), ^{
37     for (NSInteger i = 0; i < 8; i++) {
38         NSLog(@"group-02 - %@", [NSThread currentThread]);
39     }
40 });
41
42 //3.3.执行5次循环
43 dispatch_group_async(group, queue, ^{
44     for (NSInteger i = 0; i < 5; i++) {
45         NSLog(@"group-03 - %@", [NSThread currentThread]);
46     }
47 });
48
49 //4.都完成后会自动通知
50 dispatch_group_notify(group, dispatch_get_main_queue(), ^{
51     NSLog(@"完成 - %@", [NSThread currentThread]);
52 });
```

## SWIFT

```
1 //1.创建队列组
2 let group = dispatch_group_create()
3 //2.创建队列
4 let queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)
5
6 //3.多次使用队列组的方法执行任务，只有异步方法
7 //3.1.执行3次循环
8 dispatch_group_async(group, queue) { () -> Void in
9     for _ in 0..<3 {
10         NSLog("group-01 - %@", NSThread.currentThread())
11     }
12 }
13
14 //3.2.主队列执行8次循环
15 dispatch_group_async(group, dispatch_get_main_queue()) { () -> Void in
16     for _ in 0..<8 {
17         NSLog("group-02 - %@", NSThread.currentThread())
18     }
19 }
20
21 //3.3.执行5次循环
22 dispatch_group_async(group, queue) { () -> Void in
23     for _ in 0..<5 {
24         NSLog("group-03 - %@", NSThread.currentThread())
25     }
26 }
27
28 //4.都完成后会自动通知
29 dispatch_group_notify(group, dispatch_get_main_queue()) { () -> Void in
30     NSLog("完成 - %@", NSThread.currentThread())
31 }
```

## 打印结果



2015-07-28 03:40:34.277 test[12540:3319271] group-03 - <NSThread: 0x7f9772536f00>{number = 3, name = (null)}

2015-07-28 03:40:34.277 test[12540:3319146] group-02 - <NSThread: 0x7f977240ba60>{number = 1, name = main}

2015-07-28 03:40:34.277 test[12540:3319146] group-02 - <NSThread: 0x7f977240ba60>{number = 1, name = main}

2015-07-28 03:40:34.277 test[12540:3319271] group-03 - <NSThread: 0x7f9772536f00>{number = 3, name = (null)}

2015-07-28 03:40:34.278 test[12540:3319146] group-02 - <NSThread: 0x7f977240ba60>{number = 1, name = main}

2015-07-28 03:40:34.278 test[12540:3319271] group-03 - <NSThread: 0x7f9772536f00>{number = 3, name = (null)}

2015-07-28 03:40:34.278 test[12540:3319271] group-03 - <NSThread: 0x7f9772536f00>{number = 3, name = (null)}

2015-07-28 03:40:34.278 test[12540:3319146] group-02 - <NSThread: 0x7f977240ba60>{number = 1, name = main}

2015-07-28 03:40:34.277 test[12540:3319273] group-01 - <NSThread: 0x7f977272e8d0>{number = 2, name = (null)}

2015-07-28 03:40:34.278 test[12540:3319271] group-03 - <NSThread: 0x7f9772536f00>{number = 3, name = (null)}

2015-07-28 03:40:34.278 test[12540:3319146] group-02 - <NSThread: 0x7f977240ba60>{number = 1, name = main}

2015-07-28 03:40:34.278 test[12540:3319273] group-01 - <NSThread: 0x7f977272e8d0>{number = 2, name = (null)}

2015-07-28 03:40:34.278 test[12540:3319146] group-02 - <NSThread: 0x7f977240ba60>{number = 1, name = main}

2015-07-28 03:40:34.278 test[12540:3319273] group-01 - <NSThread: 0x7f977272e8d0>{number = 2, name = (null)}

2015-07-28 03:40:34.279 test[12540:3319146] group-02 - <NSThread: 0x7f977240ba60>{number = 1, name = main}

2015-07-28 03:40:34.279 test[12540:3319146] group-02 - <NSThread: 0x7f977240ba60>{number = 1, name = main}

```
2015-07-28 03:40:34.279 test[12540:3319146] 完成 - <NSThread:
0x7f977240ba60>{number = 1, name = main}
```

这些就是 GCD 的基本功能，但是它的能力远不止这些，等讲完 NSOperation 后，我们再来看看它的一些其他方面用途。而且，只要你想象力够丰富，你可以组合出更好的用法。

更新：关于GCD，还有两个需要说的：

- func dispatch\_barrier\_async(\_ queue: dispatch\_queue\_t, \_ block: dispatch\_block\_t)：

这个方法重点是你传入的 **queue**，当你传入的 **queue** 是通过 DISPATCH\_QUEUE\_CONCURRENT 参数自己创建的 **queue** 时，这个方法会阻塞这个 **queue**（注意是阻塞 **queue**，而不是阻塞当前线程），一直等到这个 **queue** 中排在它前面的任务都执行完成后才会开始执行自己，自己执行完毕后，再会取消阻塞，使这个 **queue** 中排在它后面的任务继续执行。如果你传入的是其他的 **queue**, 那么它就和 dispatch\_async 一样了。

- func dispatch\_barrier\_sync(\_ queue: dispatch\_queue\_t, \_ block: dispatch\_block\_t)：

这个方法的使用和上一个一样，传入 自定义的并发队列（DISPATCH\_QUEUE\_CONCURRENT），它和上一个方法一样的阻塞 **queue**，不同的是这个方法还会 阻塞当前线程。如果你传入的是其他的 **queue**, 那么它就和 dispatch\_sync 一样了。

## NSOperation和NSOperationQueue

NSOperation 是苹果公司对 GCD 的封装，完全面向对象，所以使用起来更好理解。大家可以看到 NSOperation 和 NSOperationQueue 分别对应 GCD 的 任务和 队列。操作步骤也很好理解：

- 将要执行的任务封装到一个 NSOperation 对象中。
- 将此任务添加到一个 NSOperationQueue 对象中。

然后系统就会自动在执行任务。至于同步还是异步、串行还是并行请继续往下看：

### 添加任务

值得说明的是，NSOperation 只是一个抽象类，所以不能封装任务。但它有 2 个子类用于封装任务。分别是： NSInvocationOperation 和 NSBlockOperation。创建一个 Operation 后，需要调用 start 方法来启动任务，它会 默认在当前队列同步执行。当然你也可以在中途取消一个任务，只需要调用其 cancel 方法即可。

- NSInvocationOperation：需要传入一个方法名。

#### OBJECTIVE-C

```
2     NSInvocationOperation *operation = [[NSInvocationOperation alloc] initWithTarget:self];
3
4     //2.开始执行
5     [operation start];
```

## SWIFT

在 Swift 构建的和谐社会里，是容不下 `NSInvocationOperation` 这种不是类型安全的败类的。

苹果如是说。[这里有相关解释](#)

- NSBlockOperation

## OBJECTIVE-C

```
1     //1.创建NSBlockOperation对象
2     NSBlockOperation *operation = [NSBlockOperation blockOperationWithBlock:^(
3         NSLog(@"%@", [NSThread currentThread]);
4     )];
5
6     //2.开始任务
7     [operation start];
```

## SWIFT

```
1     //1.创建NSBlockOperation对象
2     let operation = NSBlockOperation { () -> Void in
3         println(NSThread.currentThread())
4     }
5
6     //2.开始任务
7     operation.start()
```

之前说过这样的任务，默认会在当前线程执行。但是 `NSBlockOperation` 还有一个方法：`addExecutionBlock:`，通过这个方法可以给 Operation 添加多个执行 Block。这样 Operation 中的任务 会并发执行，它会在主线程和其它的多个线程 执行这些任务，注意下面的打印结果：

## OBJECTIVE-C

```
1     //1.创建NSBlockOperation对象
2     NSBlockOperation *operation = [NSBlockOperation blockOperationWithBlock:^(
3         NSLog(@"%@", [NSThread currentThread]);
4     )];
5
6     //添加多个Block
7     for (NSInteger i = 0; i < 5; i++) {
8         [operation addExecutionBlock:^(
9             NSLog(@"第%d次: %@", i, [NSThread currentThread]);
10        )];
11    }
12
13    //2.开始任务
14    [operation start];
```

## SWIFT

```
1     //1.创建NSBlockOperation对象
2     let operation = NSBlockOperation { () -> Void in
3         NSLog(@"%@", NSThread.currentThread())
```

```
4     }
5
6     //2.添加多个Block
7     for i in 0..<5 {
8         operation.addExecutionBlock { () -> Void in
9             NSLog("第%d次 - %@", i, NSThread.currentThread())
10        }
11    }
12
13    //2.开始任务
14    operation.start()
```

## 打印输出

```
2015-07-28 17:50:16.585 test[17527:4095467] 第2次 - <NSThread:
0x7ff5c9701910>{number = 1, name = main}

2015-07-28 17:50:16.585 test[17527:4095666] 第1次 - <NSThread:
0x7ff5c972caf0>{number = 4, name = (null)}

2015-07-28 17:50:16.585 test[17527:4095665] <NSThread: 0x7ff5c961b610>
{number = 3, name = (null)}

2015-07-28 17:50:16.585 test[17527:4095662] 第0次 - <NSThread:
0x7ff5c948d310>{number = 2, name = (null)}

2015-07-28 17:50:16.586 test[17527:4095666] 第3次 - <NSThread:
0x7ff5c972caf0>{number = 4, name = (null)}

2015-07-28 17:50:16.586 test[17527:4095467] 第4次 - <NSThread:
0x7ff5c9701910>{number = 1, name = main}
```

**NOTE:** `addExecutionBlock` 方法必须在 `start()` 方法之前执行，否则就会报错：

```
**** -[NSBlockOperation addExecutionBlock:]: blocks cannot be added after the
operation has started executing or finished'
```

**NOTE:** 大家可能发现了一个问题，为什么我在 Swift 里打印输出使用 `NSLog()` 而不是 `println()` 呢？原因是使用 `print()` / `println()` 输出的话，它会简单地使用 流（stream）的概念，学过 C++ 的都知道。它会把需要输出的每个字符一个一个的输出到控制台。普通使用并没有问题，可是当多线程同步输出的时候问题就来了，由于很多 `println()` 同时打印，就会导致控制台上的字符混乱的堆在一起，而 `NSLog()` 就没有这个问题。到底是什么样子的呢？你可以把上面 `NSLog()` 改为 `println()`，然后一试便知。 [更多 NSLog\(\) 与 println\(\) 的区别看这里](#)

- 自定义Operation

除了上面的两种 Operation 以外，我们还可以自定义 Operation。自定义 Operation 需要继承 `NSOperation` 类，并实现其 `main()` 方法，因为在调用 `start()` 方法的时候，内部会调用 `main()` 方法完成相关逻辑。所以如果以上的两个类无法满足你的欲望的时候，你就需要自定义了。你想要实现什么功能都可以写在里面。除此之外，你还需要实现 `cancel()` 在内的各种方法。所以这个功能提供给高级玩家，我在这里就不说了，等我需要用到时在研究它，到时候可能会再做更新。

## 创建队列

看过上面的内容就知道，我们可以调用一个 `NSOperation` 对象的 `start()` 方法来启动这个任务，但是这样做他们默认是 **同步执行** 的。就算是 `addExecutionBlock` 方法，也会在 **当前线程和其他线程** 中执行，也就是说还是会占用当前线程。这是就要用到队列 `NSOperationQueue` 了。而且，按类型来说的话一共有两种类型：主队列、其他队列。只要添加到队列，会自动调用任务的 `start()` 方法

- 主队列

细心的同学就会发现，每套多线程方案都会有一个主线程（当然啦，说的是iOS中，像 pthread 这种多系统的方案并没有，因为 **UI线程** 理论需要每种操作系统自己定制）。这是一个特殊的线程，必须串行。所以添加到主队列的任务都会一个接一个地排着队在主线程处理。

```
1 //OBJECTIVE-C
2 NSOperationQueue *queue = [NSOperationQueue mainQueue];
3
4 //SWIFT
5 let queue = NSOperationQueue.mainQueue()
```

- 其他队列

因为主队列比较特殊，所以会单独有一个类方法来获得主队列。那么通过初始化产生的队列就是其他队列了，因为只有这两种队列，除了主队列，其他队列就不需要名字了。

注意：其他队列的任务会在其他线程并行执行。

### OBJECTIVE-C

```
1 //1.创建一个其他队列
2 NSOperationQueue *queue = [[NSOperationQueue alloc] init];
3
4 //2.创建NSBlockOperation对象
5 NSBlockOperation *operation = [NSBlockOperation blockOperationWithBlock:^(
6     NSLog(@"%@", [NSThread currentThread]);
7 }];
8
9 //3.添加多个Block
10 for (NSInteger i = 0; i < 5; i++) {
11     [operation addExecutionBlock:^(
12         NSLog(@"第%d次: %@", i, [NSThread currentThread]);
13     }];
14 }
```



```
15 |
16 | //4. 队列添加任务
17 | [queue addOperation:operation];
```

SWIFT

```
1 | //1. 创建其他队列
2 | let queue = NSOperationQueue()
3 |
4 | //2. 创建NSBlockOperation对象
5 | let operation = NSBlockOperation { () -> Void in
6 |     NSLog("%@", NSThread.currentThread())
7 | }
8 |
9 | //3. 添加多个Block
10 | for i in 0..<5 {
11 |     operation.addExecutionBlock { () -> Void in
12 |         NSLog("第%d次 - %@", i, NSThread.currentThread())
13 |     }
14 | }
15 |
16 | //4. 队列添加任务
17 | queue.addOperation(operation)
```

打印输出

```
2015-07-28 20:26:28.463 test[18622:4443534] <NSThread: 0x7fd022c3ac10>
{number = 5, name = (null)}

2015-07-28 20:26:28.463 test[18622:4443536] 第2次 - <NSThread:
0x7fd022e36d50>{number = 2, name = (null)}

2015-07-28 20:26:28.463 test[18622:4443535] 第0次 - <NSThread:
0x7fd022f237f0>{number = 4, name = (null)}

2015-07-28 20:26:28.463 test[18622:4443533] 第1次 - <NSThread:
0x7fd022d372b0>{number = 3, name = (null)}

2015-07-28 20:26:28.463 test[18622:4443534] 第3次 - <NSThread:
0x7fd022c3ac10>{number = 5, name = (null)}

2015-07-28 20:26:28.463 test[18622:4443536] 第4次 - <NSThread:
0x7fd022e36d50>{number = 2, name = (null)}
```

OK, 这时应该发问了，大家将 `NSOperationQueue` 与 `GCD的队列` 相比较就会发现，这里没有串行队列，那如果我想要10个任务在其他线程串行的执行怎么办？

这就是苹果封装的妙处，你不用管串行、并行、同步、异步这些名词。`NSOperationQueue` 有一个参数 `maxConcurrentOperationCount` 最大并发数，用来设置最多可以让多少个任务同时执行。当你把它设置为 `1` 的时候，他不就是串行了嘛！

`NSOperationQueue` 还有一个添加任务的方法，`- (void)addOperationWithBlock:(void (^)(void))block;`，这是不是和 GCD 差不多？这样就可以添加一个任务到队列中了，十分方便。

**NSOperation** 有一个非常实用的功能，那就是添加依赖。比如有 3 个任务：A: 从服务器上下载一张图片，B: 给这张图片加个水印，C: 把图片返回给服务器。这时就可以用到依赖了：

### OBJECTIVE-C

```
1 //1.任务一： 下载图片
2 NSBlockOperation *operation1 = [NSBlockOperation blockOperationWithBlock:^(
3     NSLog(@"下载图片 - %@", [NSThread currentThread]);
4     [NSThread sleepForTimeInterval:1.0];
5 }];
6
7 //2.任务二： 打水印
8 NSBlockOperation *operation2 = [NSBlockOperation blockOperationWithBlock:^(
9     NSLog(@"打水印 - %@", [NSThread currentThread]);
10    [NSThread sleepForTimeInterval:1.0];
11 }];
12
13 //3.任务三： 上传图片
14 NSBlockOperation *operation3 = [NSBlockOperation blockOperationWithBlock:^(
15     NSLog(@"上传图片 - %@", [NSThread currentThread]);
16     [NSThread sleepForTimeInterval:1.0];
17 }];
18
19 //4.设置依赖
20 [operation2 addDependency:operation1];          //任务二依赖任务一
21 [operation3 addDependency:operation2];          //任务三依赖任务二
22
23 //5.创建队列并加入任务
24 NSOperationQueue *queue = [[NSOperationQueue alloc] init];
25 [queue addOperations:@[operation3, operation2, operation1] waitUntilFinished:NO];
```

### SWIFT

```
1 //1.任务一： 下载图片
2 let operation1 = NSBlockOperation { () -> Void in
3     NSLog("下载图片 - %@", NSThread.currentThread())
4     NSThread.sleepForTimeInterval(1.0)
5 }
6
7 //2.任务二： 打水印
8 let operation2 = NSBlockOperation { () -> Void in
9     NSLog("打水印 - %@", NSThread.currentThread())
10    NSThread.sleepForTimeInterval(1.0)
11 }
12
13 //3.任务三： 上传图片
14 let operation3 = NSBlockOperation { () -> Void in
15     NSLog("上传图片 - %@", NSThread.currentThread())
16     NSThread.sleepForTimeInterval(1.0)
17 }
18
19 //4.设置依赖
20 operation2.addDependency(operation1)          //任务二依赖任务一
21 operation3.addDependency(operation2)          //任务三依赖任务二
22
23 //5.创建队列并加入任务
24 let queue = NSOperationQueue()
25 queue.addOperations([operation3, operation2, operation1], waitUntilFinished: false)
```

### 打印结果

2015-07-28 21:24:28.622 test[19392:4637517] 下载图片 - <NSThread:  
0x7fc10ad4d970>{number = 2, name = (null)}

```
2015-07-28 21:24:29.622 test[19392:4637515] 打水印 - <NSThread:
0x7fc10af20ef0>{number = 3, name = (null)}
```

```
2015-07-28 21:24:30.627 test[19392:4637515] 上传图片 - <NSThread:
0x7fc10af20ef0>{number = 3, name = (null)}
```

- 注意：不能添加相互依赖，会死锁，比如 A依赖B，B依赖A。
- 可以使用 `removeDependency` 来解除依赖关系。
- 可以在不同的队列之间依赖，反正就是这个依赖是添加到任务身上的，和队列没关系。

## 其他方法

以上就是一些主要方法, 下面还有一些常用方法需要大家注意：

- NSOperation

```
BOOL executing; //判断任务是否正在执行

BOOL finished; //判断任务是否完成

void (^completionBlock)(void); //用来设置完成后需要执行的操作

- (void)cancel; //取消任务

- (void)waitUntilFinished; //阻塞当前线程直到此任务执行完毕
```

- NSOperationQueue

```
NSUInteger operationCount; //获取队列的任务数

- (void)cancelAllOperations; //取消队列中所有的任务

- (void)waitUntilAllOperationsAreFinished; //阻塞当前线程直到此队列中的所有任务执行完毕

[queue setSuspended:YES]; // 暂停queue

[queue setSuspended:NO]; // 继续queue
```

好啦，到这里差不多就讲完了。当然，我讲的并不完整，可能有一些知识我并没有讲到，但作为常用方法，这些已经足够了。不过我在这里只是告诉你了一些方法的功能，只是怎么把他们用到合适的地方，就需要多多实践了。下面我会说一些关于多线程的案例，是大家更加什么地了解。

## 其他用法

在这部分，我会说一些和多线程知识相关的案例，可能有些很简单，大家早都知道的，不过因为这篇文章讲的是多线程嘛，所以应该尽可能的全面嘛。还有就是，我会尽可能的使用多种方法实现，让大家看看其中的区别。

## 线程同步

所谓线程同步就是为了防止多个线程抢夺同一个资源造成的数据安全问题，所采取的一种措施。当然也有很多实现方法，请往下看：

- **互斥锁**：给需要同步的代码块加一个互斥锁，就可以保证每次只有一个线程访问此代码块。

### OBJECTIVE-C

```
1 | @synchronized(self) {
2 |     //需要执行的代码块
3 | }
```

### SWIFT

```
1 | objc_sync_enter(self)
2 | //需要执行的代码块
3 | objc_sync_exit(self)
```

- **同步执行**：我们可以使用多线程的知识，把多个线程都要执行此段代码添加到同一个串行队列，这样就实现了线程同步的概念。当然这里可以使用 `GCD` 和 `NSOperation` 两种方案，我都写出来。

### OBJECTIVE-C

```
1 |
2 | //GCD
3 | //需要一个全局变量queue，要让所有线程的这个操作都加到一个queue中
4 | dispatch_sync(queue, ^{
5 |     NSInteger ticket = lastTicket;
6 |     [NSThread sleepForTimeInterval:0.1];
7 |     NSLog(@"%ld - %@",ticket, [NSThread currentThread]);
8 |     ticket -= 1;
9 |     lastTicket = ticket;
10 | });
11 |
12 |
13 | //NSOperation & NSOperationQueue
14 | //重点：1. 全局的 NSOperationQueue，所有的操作添加到同一个queue中
15 | //      2. 设置 queue 的 maxConcurrentOperationCount 为 1
16 | //      3. 如果后续操作需要Block中的结果，就需要调用每个操作的waitUntilFinished，阻塞当前线程，一直
17 |
18 | NSBlockOperation *operation = [NSBlockOperation blockOperationWithBlock:^(
19 |     NSInteger ticket = lastTicket;
20 |     [NSThread sleepForTimeInterval:1];
21 |     NSLog(@"%ld - %@",ticket, [NSThread currentThread]);
22 |     ticket -= 1;
23 |     lastTicket = ticket;
24 | )];
25 |
26 | [queue addOperation:operation];
27 |
28 | [operation waitUntilFinished];
29 |
30 | //后续要做的事
```

SWIFT

这里的 swift 代码，我就不写了，因为每句都一样，只是语法不同而已，照着 OC 的代码就能写出 Swift 的。这篇文章已经老长老长了，我就不浪费篇幅了，又不是高中写作文。

延迟执行

所谓延迟执行就是延时一段时间再执行某段代码。下面说一些常用方法。

- perform

OBJECTIVE-C

```
1 | // 3秒后自动调用self的run:方法，并且传递参数: @"abc"
2 | [self performSelector:@selector(run:) withObject:@"abc" afterDelay:3];
```

SWIFT

```
1 | 之前就已经说过，Swift 里去掉了这个方法。
```

- GCD

可以使用 GCD 中的 `dispatch_after` 方法，OC 和 Swift 都可以使用，这里只写 OC 的，Swift 的是一样的。

OBJECTIVE-C

```
1 | // 创建队列
2 | dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
3 | // 设置延时，单位秒
4 | double delay = 3;
5 |
6 | dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(delay * NSEC_PER_SEC)), queue, ^{
7 |     // 3秒后需要执行的任务
8 | });
```

- NSTimer

NSTimer 是iOS中的一个计时器类，除了延迟执行还有很多用法，不过这里直说延迟执行的用法。同样只写 OC 版的，Swift 也是相同的。

OBJECTIVE-C

```
1 | [NSTimer scheduledTimerWithTimeInterval:3.0 target:self selector:@selector(run:) userI
```

单例模式

至于什么是单例模式，我也不多说，我只说说一般怎么实现。在 Objective-C 中，实现单例的方法已经很具体了，虽然有别的方法，但是一般都是用一个标准的方法了，下面来看看。

OBJECTIVE-C



```

1 | @interface Tool : NSObject <NSCopying>
2 |
3 | + (instancetype)sharedTool;
4 |
5 | @end
6 |
7 | @implementation Tool
8 |
9 | static id _instance;
10 |
11 | + (instancetype)sharedTool {
12 |     static dispatch_once_t onceToken;
13 |     dispatch_once(&onceToken, ^{
14 |         _instance = [[Tool alloc] init];
15 |     });
16 |
17 |     return _instance;
18 | }
19 |
20 | @end

```

这里之所以将单例模式，是因为其中用到了 GCD 的 `dispatch_once` 方法。下面看 Swift 中的单例模式，在Swift中单例模式非常简单！想知道怎么从 OC 那么复杂的方法变成下面的写法的，[请看这里](#)

## SWIFT

```

1 | class Tool: NSObject {
2 |     static let sharedTool = Tool()
3 |
4 |     // 私有化构造方法，阻止其他对象使用这个类的默认的'()'构造方法
5 |     private override init() {}
6 | }

```

## 从其他线程回到主线程的方法

我们都知道在其他线程操作完成后必须到主线程更新UI。所以，介绍完所有的多线程方案后，我们来看看有哪些方法可以回到主线程。

- NSThread

```

1 | //Objective-C
2 | [self performSelectorOnMainThread:@selector(run) withObject:nil waitUntilDone:NO];
3 |
4 | //Swift
5 | //swift 取消了 performSelector 方法。

```

- GCD

```

1 | //Objective-C
2 | dispatch_async(dispatch_get_main_queue(), ^{
3 |
4 | });
5 |
6 | //Swift
7 | dispatch_async(dispatch_get_main_queue(), { () -> Void in
8 |
9 | })

```

- NSOperationQueue

```
1 //Objective-C
2 [[NSOperationQueue mainQueue] addOperationWithBlock:^(
3
4 }];
5
6 //Swift
7 NSOperationQueue.mainQueue().addOperationWithBlock { () -> Void in
8
9 }
```

## 总结

好的吧，总算写完了，纯手敲6k多字，感动死我了。花了两天，时间跨度有点大，所以可能有些地方上段不接下段或者有的地方不完整，如果你看着比较费力或者有什么地方有问题，都可以在评论区告诉我，我会及时修改的。当然啦，多线程的东西也不止这些，题目也就只是个题目，不要当真。想要了解更多的东西，还得自己去网上挖掘相关资料。多看看官方文档。实在是编不下去了，大家好好看~。对了，看我写的这么卖力，不打赏的话得点个喜欢也是极好的。

更新：第一次放出来的时候，有很多地方有错误，很感谢有朋友提出来了。如果你看到有错误的地方，一定记得指出来，这样对大家都有帮助。还有一点对初学者来说，遇到不懂的方法，最好的办法就是查看官方文档，那里是最准确的，就算有几个单词不认识，查一下就好了，不会影响对整体的理解。

我看到有网站转载了我的文章，但转载的可能存在问题，而我只能在简书上更新，所以如果要看 完整版本 还是到简书来看吧：[这里是地址](#)。

2235人点赞 >

About iOS

"您觉得这篇文章对你有帮助吗？看我这么用心，鼓励一下呗~"

赞赏支持

共19人赞赏

伯恩的遗产

我叫翁亚伟（@翁呀伟呀） iOS Developer，目前还是一名学生 此...  
总资产20 (约1.99元) 共写了3.3W字 获得3,460个赞 共3,496个粉丝

关注

种牙的副作用

舆情监测平台

怎样使鼻翼缩小

黑马it培训

智能井盖

被以下专题收入，发现更多相似内容

程序员

首页投稿（暂停...

iOS

iOS

iOS开发技巧


iOS开发

ios

展开更多 ▾

## 关于iOS多线程，你看我就够了（已更新）

作者：@翁呀伟呀授权本站转载。在这篇文章中，我将为你整理一下 iOS 开发中几种多线程方案，以及其使用方法和注意事...

 \_\_Lex 阅读 502 评论 2 赞 12

## 关于iOS多线程，你看我就够了(读)

原文地址：关于iOS多线程，你看我就够了感谢博主的劳动，下面是对于原文的一些解读和测试 #import void...

 十顿十 阅读 254 评论 0 赞 0

## iOS多线程简析 --2016.03

在这篇文章中，我将为你整理一下 iOS 开发中几种多线程方案，以及其使用方法和注意事项。当然也会给出几种多线程的案...

 张战威ican 阅读 160 评论 0 赞 0

## IOS 多线程知识学习

学习多线程，转载两篇大神的帖子，留着以后回顾！第一篇：关于iOS多线程，你看我就够了 第二篇：GCD 使用经验与技巧...

 John\_LS 阅读 283 评论 0 赞 3

## 假如我身患绝症，请让我有尊严的死去！

作者|格桑小巫 不知道是因为自己年岁越长，见识过的生离死别，累计得越来越多的缘故，还是绝症发病率真的越来越高，感觉...

 格桑小巫 阅读 5,841 评论 1 赞 6

