iOS 多线程编程总结

1220 days ago

多线程之谜

很长时间以来，我个人（可能还有很多同学），对多线程编程都存在一些误解。一个很明显的表现是，很多人有这样的看法：

> 新开一个线程，能提高速度，避免阻塞主线程

毕竟多线程嘛，几个线程一起跑任务，速度快，还不阻塞主线程，简直完美。

在某些场合，我们还见过另一个“高深”的名词——“异步”。这东西好像和多线程挺类似，经过一番百度（阅读了很多质量层次不齐的文章）之后，很多人也没能真正搞懂何为“异步”。

于是，带着对“多线程”和“异步”的懵懂，很多人又开开心心踏上了多线程编程之旅，比如文章待会儿会提到的 GCD。

何为多线程

其实，如果不考虑其他任何因素和技术，多线程有百害而无一利，只能浪费时间，降低程序效率。

是的，我很清醒的写下这句话。

试想一下，一个任务由十个子任务组成。现在有两种方式完成这个任务： 1. 建十个线程，把每个子任务放在对应的线程中执行。执行完一个线程中的任

务就切换到另一个线程。
2. 把十个任务放在一个线程里，按顺序执行。

操作系统的基础知识告诉我们，线程，是执行程序最基本的单元，它有自己栈和寄存器。说得再具体一些，线程就是“一个 CPU 执行的一条无分叉的命令列”。

对于第一种方法，在十个线程之间来回切换，就意味着有十组栈和寄存器中的值需要不断地被备份、替换。 而对于对于第二种方法，只有一组寄存器和栈存在，显然效率完胜前者。

并发与并行

通过刚刚的分析我们看到，多线程本身会带来效率上的损失。准确来说，在处理并发任务时，多线程不仅不能提高效率，反而还会降低程序效率。

所谓的“并发”，英文翻译是 concurrent。要注意和“并行（parallelism）”的区别。

> 并发指的是一种现象，一种经常出现，无可避免的现象。它描述的是“多个任务同时发生，需要被处理”这一现象。它的侧重点在于“发生”。

比如有很多人排队等待检票，这一现象就可以理解为并发。

> 并行指的是一种技术，一个同时处理多个任务的技术。它描述了一种能够同时处理多个任务的能力，侧重点在于“运行”。

比如景点开放了多个检票窗口，同一时间内能服务多个游客。这种情况可以理解为并行。

并行的反义词就是串行，表示任务必须按顺序来，一个一个执行，前一个执行完了才能执行后一个。

我们经常挂在嘴边的“多线程”，正是采用了并行技术，从而提高了执行效率。因为有多个线程，所以计算机的多个 CPU 可以同时工作，同时处理不同线程内的指令。

小结

并发是一种现象，面对这一现象，我们首先创建多个线程，真正加快程序运行速度的，是并行技术。也就是让多个 CPU 同时工作。而多线程，是为了让多个 CPU 同时工作成为可能。

同步与异步

同步方法就是我们平时调用的哪些方法。因为任何有编程经验的人都知道，比如在第一行调用 `foo()` 方法，那么程序运行到第二行的时候，foo 方法肯定是执行完了。

所谓的异步，就是允许在执行某一个任务时，函数立刻返回，但是真正要执行的任务稍后完成。

比如我们在点击保存按钮之后，要先把数据写到磁盘，然后更新 UI。同步方法就是等到数据保存完再更新 UI，而异步则是立刻从保存数据的方法返回并向后执行代码，同时真正用来保存数据的指令将在稍后执行。

区别与联系

区别

假设现在有三个任务需要处理。假设单个 CPU 处理它们分别需要 3、1、1 秒。

并行与串行，其实讨论的是处理这三个任务的速度问题。如果三个 CPU 并行处理，那么一共只需要 3 秒。相比于串行处理，节约了两秒。

而同步 / 异步，其实描述的是任务之间先后顺序问题。假设需要三秒的那个是保存数据的任务，而另外两个是 UI 相关的任务。那么通过异步执行第一个任务，我们省去了三秒钟的卡顿时间。

联系

对于同步执行的三个任务来说，系统倾向于在同一个线程里执行它们。因为即使开了三个线程，也得等他们分别在各自的线程中完成。并不能减少总的处理时间，反而徒增了线程切换（这就是文章开头举的例子）

对于异步执行的三个任务来说，系统倾向于在三个新的线程里执行他们。因为这样可以最大程度的利用 CPU 性能，提升程序运行效率。

结论

于是我们可以得出结论，在需要同时处理 IO 和 UI 的情况下，真正起作用的是异步，而不是多线程。可以不用多线程（因为处理 UI 非常快），但不能不用异步（否则的话至少要等 IO 结束）。

注意到我把“倾向于”这三个加粗了，也就是说异步方法并不一定永远在新线程里面执行，反之亦然。在接下来关于 GCD 的部分会对此做出解释。

GCD 简介

GCD 以 block 为基本单位，一个 block 中的代码可以为一个任务。下文中提到任务，可以理解为执行某个 block

同时，GCD 中有两大最重要的概念，分别是“队列”和“执行方式”。

使用 block 的过程，概括来说就是把 block 放进合适的队列，并选择合适的执行方式去执行 block 的过程。

队列总的来说可以分为三种：

- 串行队列（先进入队列的任务先出队列，每次只执行一个任务）

- 并发队列（依然是“先入先出”，不过可以形成多个任务并发）
- 主队列（这是一个特殊的串行队列，而且队列中的任务一定会在主线程中执行）

两种基本的执行方式

- 同步执行
- 异步执行

关于同步异步、串行并行和线程的关系，下面通过一个表格来总结

| | 同步 | 异步 | | ---------- |:----------:| :----------:| | 主队列 | 在主线程中执行 | 在主线程中执行 | | 串行队列 | 在当前线程中执行 | 新建线程执行 | | 并发队列 | 在当前线程中执行 | 新建线程执行 |

可以看到，同步方法不一定在本线程，异步方法方法也不一定新开线程（考虑主队列）。

然而事实上，在本文一开始就揭开了“多线程”的神秘面纱，所以我们在编程时，更应该考虑的是：

> 同步 Or 异步

以及

> 串行 Or 并行

而非仅仅考虑是否新开线程。

当然，了解任务运行在那个线程中也是为了更加深入的理解整个程序的运行情况，尤其是接下来要讨论的死锁问题。

GCD 的死锁问题

在使用 GCD 的过程中，如果向当前串行队列中同步派发一个任务，就会导致死锁。

这句话有点绕，我们首先举个例子看看：

```
override func viewDidLoad() {
    super.viewDidLoad()
    let mainQueue = dispatch_get_main_queue()
    let block = { ()  in
        print(NSThread.currentThread())
    }
    dispatch_sync(mainQueue, block)
}
```

这段代码就会导致死锁，因为我们目前在主队列中，又将要同步地添加一个 `block` 到主队列 (串行) 中。

理论分析

我们知道 `dispatch_sync` 表示同步的执行任务，也就是说执行 `dispatch_sync` 后，当前队列会阻塞。而 `dispatch_sync` 中的 block 如果要在当前队列中执行，就得等待当前队列程执行完成。

在上面这个例子中，主队列在执行 `dispatch_sync` ，随后队列中新增一个任务 `block` 。因为主队列是同步队列，所以 `block` 要等 `dispatch_sync` 执行完才能执行，但是 `dispatch_sync` 是同步派发，要等 `block` 执行完才算是结束。在主队列中的两个任务互相等待，导致了死锁。

解决方案

其实在通常情况下我们不必要用 `dispatch_sync` ，因为 `dispatch_async` 能够更好的利用 CPU，提升程序运行速度。

只有当我们需要保证队列中的任务必须顺序执行时，才考虑使用 `dispatch_sync` 。在使用 `dispatch_sync` 的时候应该分析当前处于哪个队列，以及任务会提交到哪个队列。

GCD 任务组

了解完队列之后，很自然的会有一个想法：我们怎么知道所有任务都已经执行完了呢?

在单个串行队列中，这个不是问题，因为只要把回调 block 添加到队列末尾即可。

但是对于并行队列，以及多个串行、并行队列混合的情况，就需要使用 `dispatch_group` 了。

```
let group = dispatch_group_create()

dispatch_group_async(group, serialQueue, { () -> Void in
    for _ in 0..<2 {
        print("group-serial \(NSThread.currentThread())")
    }
})

dispatch_group_async(group, serialQueue, { () -> Void in
    for _ in 0..<3 {
        NSLog("group-02 - %@", NSThread.currentThread())
    }
})

dispatch_group_notify(group, serialQueue, { () -> Void in
    print("完成 - \(NSThread.currentThread())")
})
```

首先我们要通过 `dispatch_group_create()` 方法生成一个组。

接下来，我们把 `dispatch_async` 方法换成 `dispatch_group_async` 。这个方法多了一个参数，第一个参数填刚刚创建的分组。

想问 `dispatch_sync` 对应的分组方法是什么的童鞋面壁思过三秒钟，思考一下 group 出现的目的和 `dispatch_sync` 的特点。

最后调用 `dispatch_group_notify` 方法。这个方法表示把第三个参数 block 传入第二个参数队列中去。而且可以保证第三个参数 block 执行时，group 中的所有任务已经全部完成。

dispatch_group

`dispatch_group_wait` 方法是一个很有用的方法，它的完整定义如下：

`dispatch_group_wait(group: dispatch_group_t, _ timeout: dispatch_time_t) -> Int`

第一个参数表示要等待的 group，第二个则表示等待时间。返回值表示经过指定的等待时间，属于这个 group 的任务是否已经全部执行完，如果是则返回 0，否则返回非 0。

第二个 `dispatch_time_t` 类型的参数还有两个特殊值： `DISPATCH_TIME_NOW` 和 `DISPATCH_TIME_FOREVER` 。

前者表示立刻检查属于这个 group 的任务是否已经完成，后者则表示一直等到属于这个 group 的任务全部完成。

dispatch_after 方法

通过 GCD 还可以进行简单的定时操作，比如在 1 秒后执行某个 block。代码如下：

```
let mainQueue = dispatch_get_main_queue()
let time = dispatch_time(DISPATCH_TIME_NOW, Int64(3) * Int64(NSEC_PER_SEC))
NSLog("%@",NSThread.currentThread())
dispatch_after(time, mainQueue, {() in NSLog("%@",NSThread.currentThread())})
```

`dispatch_after` 方法有三个参数。第一个表示时间，也就是从现在起往后三秒钟。第二三个参数分别表示要提交的任务和提交到哪个队列。

需要注意的是 `dispatch_after` 仅表示在指定时间后提交任务，而非执行任务。如果任务提交到主队列，它将在 main runloop 中执行，对于每隔 1/60 秒执行一次的 RunLoop，任务最多有可能在 3+1/60 秒后执行。

总结

经过上一章的学习，我们已经理解了多线程编程的基本概念，以及 GCD 的简单使用。在这一章中将会介绍和 `NSOperation` 和和 `NSOperationQueue` 。主要涉及这几个方面：

- `NSOperation` 和 `NSOperationQueue` 用法介绍
- `NSOperation` 的暂停、恢复和取消
- 通过 KVO 对 `NSOperation` 的状态进行检测
- 多个 `NSOperation` 的之间的依赖关系

NSOperation

从简单意义上来说， `NSOperation` 是对 GCD 中的 block 进行的封装，它也表示一个要被执行的任务。

与 GCD 中的 block 类似， `NSOperation` 对象有一个 `start()` 方法表示开始执行这个任务。

不仅如此， `NSOperation` 表示的任务还可以被取消。它还有三种状态 `isExecuted` 、 `isFinished` 和 `isCancelled` 以方便我们通过 KVC 对它的状态进行监听。

想要开始执行一个任务可以这么写：

```
let operation = NSBlockOperation { () -> Void in
    print(NSThread.currentThread())
}
operation.addExecutionBlock { () -> Void in
    print("execution block1 -- \(NSThread.currentThread())")
}
operation.start()
```

以上代码会得到这样的执行结果：

```
<NSThread: 0x7f89b1c070f0>{number = 1, name = main}
execution block1 -- <NSThread: 0x7f89b1e17030>{number = 2, name = (null)}
```

首先我们创建了一个 `NSBlockOperation` ，并且设置好它的 block，也就是将要执行的任务。这个任务会在主线程中执行。

用 `NSBlockOperation` 是因为 `NSOperation` 是一个基类，不应该直接生成 `NSOperation` 对象，而是应该用它的子类。 `NSBlockOperation` 是苹果预定义的子类，它可以用来封装一个或多个 block，后面会介绍如何自己创建 `NSOperation` 的子类。

同时，还可以调用 `addExecutionBlock` 方法追加几个任务，这些任务会并行执行（也就是说很有可能运行在别的线程里）。

最后，调用 `start` 方法让 `NSOperation` 方法运行起来。 `start` 是一个同步方法。

NSOperationQueue

刚刚我们知道，默认的 `NSOperation` 是同步执行的。简单的看一下 `NSOperation` 类的定义会发现它有一个只读属性 `asynchronous`

这意味着如果想要异步执行，就需要自定义 `NSOperation` 的子类。或者使用 `NSOperationQueue`

`NSOperationQueue` 类似于 GCD 中的队列。我们知道 GCD 中的队列有三种：主队列、串行队列和并行队列。 `NSOperationQueue` 更简单，只有两种：主队列和非主队列。

我们自己生成的 `NSOperationQueue` 对象都是非主队列，主队列可以用 `NSOperationQueue.mainQueue` 取得。

`NSOperationQueue` 的主队列是串行队列，而且其中所有 `NSOperation` 都会在主线程中执行。

对于非主队列来说，一旦一个 `NSOperation` 被放入其中，那这个 `NSOperation` 一定是并发执行的。因为 `NSOperationQueue` 会为每一个 `NSOperation` 创建线程并调用它的 `start` 方法。

`NSOperationQueue` 有一个属性叫 `maxConcurrentOperationCount` ，它表示最多支持多少个 `NSOperation` 并发执行。如果 `maxConcurrentOperationCount` 被设为 1，就以为这个队列是串行队列。

因此， `NSOperationQueue` 和 GCD 中的队列有这样的对应关系：

| | NSOperationQueue | GCD | | ---------- |:----------:| :----------:| ---| | 主队列 | NSOperationQueue.mainQueue | dispatch_get_main_queue()| | 串行队列 | 自建队列 maxConcurrentOperationCount 为 1 | dispatch_queue_create("", DISPATCH_QUEUE_SERIAL) | | 并发队列 | 自建队列 maxConcurrentOperationCount 大于 1 | dispatch_queue_create("", DISPATCH_QUEUE_CONCURRENT) |

回到开头的问题，如何利用 `NSOperationQueue` 实现异步操作呢，代码如下：

```
let operationQueue = NSOperationQueue()
let operation = NSBlockOperation ()
operation.addExecutionBlock { () -> Void in
    print("exec block1 -- \(NSThread.currentThread())")
}
operation.addExecutionBlock { () -> Void in
    print("exec block2 -- \(NSThread.currentThread())")
}
operation.addExecutionBlock { () -> Void in
    print("exec block3 -- \(NSThread.currentThread())")
}
operationQueue.addOperation(operation)
print("操作结束")
```

得到运行结果如下：

```
操作结束
exec block1 -- <NSThread: 0x125672f10>{number = 2, name = (null)}
exec block2 -- <NSThread: 0x12556ba40>{number = 3, name = (null)}
exec block3 -- <NSThread: 0x125672f10>{number = 2, name = (null)}
```

使用 `NSOperationQueue` 来执行任务与之前的区别在于，首先创建一个非主队列。然后用 `addOperation` 方法替换之前的 `start` 方法。刚刚已经说过，`NSOperationQueue` 会为每一个 `NSOperation` 建立线程并调用他们的 `start` 方法。

观察一下运行结果，所有的 `NSOperation` 都没有在主线程执行，从而成功的实现了异步、并行处理。

NSOperation 新特性

在学习 `NSOperation` 的时候，我们总是用 GCD 的概念去解释。但是 `NSOperation` 作为对 GCD 更高层次的封装，它有着一些 GCD 无法实现（或者至少说很难实现）的特性。由于 `NSOperation` 和 `NSOperationQueue` 良好的封装，这些新特性的使用都非常简单。

取消任务

如果我们有两次网络请求，第二次请求会用到第一次的数据。如果此时网络情况不好，第一次请求超时了，那么第二次请求也没有必要发送了。当然，用户也有可能人为地取消某个 `NSOperation` 。

当某个 `NSOperation` 被取消时，我们应该尽可能的清除 `NSOperation` 内部的数据并且把 `cancelled` 和 `finished` 设为 `true` ，把 `executing` 设为 `false` 。

```
//取消某个NSOperation
operation1.cancel()

//取消某个NSOperationQueue剩余的NSOperation
queue.cencelAllOperations()
```

设置依赖

依然考虑刚刚所说的两次网络请求的例子。因为第二次请求会用到第一次的数据，所以我们要保证发出第二次请求的时候第一个请求已经执行完。但是

我们同时还希望利用到 `NSOperationQueue` 的并发特性（因为可能不止这两个任务）。

这时候我们可以设置 `NSOperation` 之间的依赖关系。语法非常简洁：

```
operation2.addDependency(operation1)
```

需要注意的是 `NSOperation` 之间的相互依赖会导致死锁

NSOperationQueue 的暂停与恢复

这个更加简单，只要修改 `suspended` 属性即可

```
queue.suspended = true //暂停queue中所有operation
queue.suspended = false //恢复queue中所有operation
```

NSOperation 的优先级

GCD 中，任务（block）是没有优先级的，而队列具有优先级。和 GCD 相反，我们一般考虑 `NSOperation` 的优先级

`NSOperation` 有一个 `NSOperationQueuePriority` 枚举类型的属性 `queuePriority`

```
public enum NSOperationQueuePriority : Int {
    case VeryLow
    case Low
    case Normal
    case High
    case VeryHigh
}
```

需要注意的是， `NSOperationQueue` 也不能完全保证优先级高的任务一定先执行。

NSOperation 和 GCD 如何选择

其实经过这两篇文章的分析，我们大概对 `NSOperation` 和 `GCD` 都有了比较详细的了解，同时在亲自运用这两者的过程中有了自己的理解。

GCD 以 block 为单位，代码简洁。同时 GCD 中的队列、组、信号量、source、barriers 都是组成并行编程的基本原语。对于一次性的计算，或是仅仅为了加快现有方法的运行速度，选择轻量化的 GCD 就更加方便。

而 `NSOperation` 可以用来规划一组任务之间的依赖关系，设置它们的优先级，任务能被取消。队列可以暂停、恢复。 `NSOperation` 还可以被子类化。这些都是 GCD 所不具备的。

所以我们要记住的是：

> `NSOperation` 和 GCD 并不是互斥的，有效地结合两者可以开发出更棒的应用

总结

到目前为止，我们已经理解了多线程编程的基本概念，GCD 的简单使用，用 `NSOperation` 实现 GCD 的功能，以及 `NSOperation` 的高级特性。在最后一章 iOS 多线程编程总结（下）中会介绍 GCD 的底层特性，如 barrier、suspend、resume、semaphore 等。

主要内容

到目前为止，我们已经了解了 GCD 和 NSOperation 在多线程编程中的使用。NSOperation 是对 GCD 更高层次的封装，提供了任务的取消、暂停、恢复功能。但 GCD 因为更加接近底层，所以也有自己的优势。本章将会由浅入深，讨论以下几个部分：

- `dispatch_suspend` 和 `dispatch_resume`
- `dispathc_once`

- `dispatch_barrier_async`
- `dispatch_semaphore`

dispatchsuspend 和 dispatchresume

我们知道 `NSOperationQueue` 有暂停 (suspend) 和恢复(resume)。其实 GCD 中的队列也有类似的功能。用法也非常简单：

```
dispatch_suspend(queue) //暂停某个队列
dispatch_resume(queue)  //恢复某个队列
```

这些函数不会影响到队列中已经执行的任务，队列暂停后，已经添加到队列中但还没有执行的任务不会执行，直到队列被恢复。

dispathc_once

首先我们来看一下最简单的 * `dispathc_once` 函数，这在单例模式中被广泛使用。

- `dispathc_once` 函数可以确保某个 block 在应用程序执行的过程中只被处理一次，而且它是线程安全的。所以单例模式可以很简单的实现，以 OC 中 Manager 类为例

```
+ (Manager *)sharedInstance {
    static Manager *sharedManagerInstance = nil;
    static dispatch_once_t once;

    dispatch_once($once, ^{
        sharedManagerInstance = [[Manager alloc] init];
    });

    return sharedManagerInstance;
}
```

这段代码中我们创建一个值为 nil 的 `sharedManagerInstance` 静态对象，然后把它的初始化代码放到 `dispatch_once` 中完成。

这样，只有第一次调用 `sharedInstance` 方法时才会进行对象的初始化，以后每次只是返回 `sharedManagerInstance` 而已。

dispatchbarrierasync

我们知道数据在写入时，不能在其他线程读取或写入。但是多个线程同时读取数据是没有问题的。所以我们可以把读取任务放入并行队列，把写入任务放入串行队列，并且保证写入任务执行过程中没有读取任务可以执行。

这样的需求比较常见，GCD 提供了一个非常简单的解决办法——`dispatch_barrier_async`

假设我们有四个读取任务，在第二三个任务之间有一个写入任务，代码大概是这样：

```
let queue = dispatch_queue_create("com.gcd.kt", DISPATCH_QUEUE_CONCURRENT)

dispatch_async(queue, block1_for_reading)
dispatch_async(queue, block2_for_reading)

/*
    这里插入写入任务，比如：
    dispatch_async(queue, block_for_writing)
*/

dispatch_async(queue, block3_for_reading)
dispatch_async(queue, block4_for_reading)
```

如果代码这样写，由于这几个 block 是并发执行，就有可能在前两个 block 中读取到已经修改了的数据。如果是有多写入任务，那问题更严重，可能会有数据竞争。

如果使用 `dispatch_barrier_async` 函数，代码就可以这么写：

```
dispatch_async(queue, block1_for_reading)
dispatch_async(queue, block2_for_reading)

dispatch_barrier_async(queue, block_for_writing)
```

```
dispatch_async(queue, block3_for_reading)
dispatch_async(queue, block4_for_reading)
```

`dispatch_barrier_async` 会把并行队列的运行周期分为这三个过程：

- 首先等目前追加到并行队列中所有任务都执行完成
- 开始执行 `dispatch_barrier_async` 中的任务，这时候即使向并行队列提交任务，也不会执行
- `dispatch_barrier_async` 中的任务执行完成后，并行队列恢复正常。

总的来说，`dispatch_barrier_async` 起到了“承上启下”的作用。它保证此前的任务都先于自己执行，此后的任务也迟于自己执行。正如 barrier 的含义一样，它起到了一个栅栏、或是分水岭的作用。

这样一来，使用并行队列和 `dispatc_barrier_async` 方法，就可以高效的进行数据和文件读写了。

dispatch_semaphore

首先介绍一下信号量 (semaphore) 的概念。信号量是持有计数的信号，不过这么解释等于没解释。我们举个生活中的例子来看看。

假设有一个房子，它对应进程的概念，房子里的人就对应着线程。一个进程可以包括多个线程。这个房子 (进程) 有很多资源，比如花园、客厅等，是所有人 (线程) 共享的。

但是有些地方，比如卧室，最多只有两个人能进去睡觉。怎么办呢，在卧室门口挂上两把钥匙。进去的人 (线程) 拿着钥匙进去，没有钥匙就不能进去，出来的时候把钥匙放回门口。

这时候，门口的钥匙数量就称为信号量 (Semaphore)。很明显，信号量为 0 时需要等待，信号量不为零时，减去 1 而且不等待。

在 GCD 中，创建信号量的语法如下：

```
var semaphore = dispatch_semaphore_create(2)
```

这句代码通过 `dispatch_semaphore_create` 方法创建一个信号量并设置初始值为 2。然后就可以调用 `dispatch_semaphore_wait` 方法了。

```
dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER)
```

`dispatch_semaphore_wait` 方法表示一直等待直到信号量的值大于等于一，当这个方法执行后，会把第一个信号量参数的值减 1。

第二个参数是一个 `dispatch_time_t` 类型的时间，它表示这个方法最大的等待时间。这在第一章中已经讲过，比如 `DISPATCH_TIME_FOREVER` 表示永久等待。

返回值也和 `dispatch_group_wait` 方法一样，返回 0 表示在规定的等待时间内第一个参数信号量的值已经大于等于 1，否则表示已超过规定等待时间，但信号量的值还是 0。

`dispatch_semaphore_wait` 方法返回 0，因为此时的信号量的值大于等于一，任务获得了可以执行的权限。这时候我们就可以安全的执行需要进行排他控制的任务了。

任务结束时还需要调用 `dispatch_semaphore_signal()` 方法，将信号量的值加 1。这类似于之前所说的，从卧室出来要把锁放回门上，否则后来的人就无法进入了。

我们来看一个完整的例子：

```
var semaphore = dispatch_semaphore_create(1)
let queue = dispatch_queue_create("com.gcd.kt", DISPATCH_QUEUE_CONCURRENT)
var array: [Int] = []

for i in 1...100000 {
    dispatch_async(queue, { () -> Void in
        /*
            某个线程执行到这里，如果信号量值为1，那么wait方法返回1，开始执行接下来的操作。
            与此同时，因为信号量变为0，其它执行到这里的线程都必须等待
        */
        dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER)
```

```
            /*
                执行了wait方法后，信号量的值变成了0。可以进行接下来的操作。
                这时候其它线程都得等待wait方法返回。
                可以对array修改的线程在任意时刻都只有一个，可以安全的修改array
            */
            array.append(i)

            /*
                排他操作执行结束，记得要调用signal方法，把信号量的值加1。
                这样，如果有别的线程在等待wait函数返回，就由最先等待的线程执行。
            */
            dispatch_semaphore_signal(semaphore)
        })
    }
```

如果你想知道不用信号量会出什么问题，可以看我的另一篇文章 Swift 数组 append 方法研究

全文完