

# 深入浅出 iOS 并发编程

故胤道长

[关注](#)

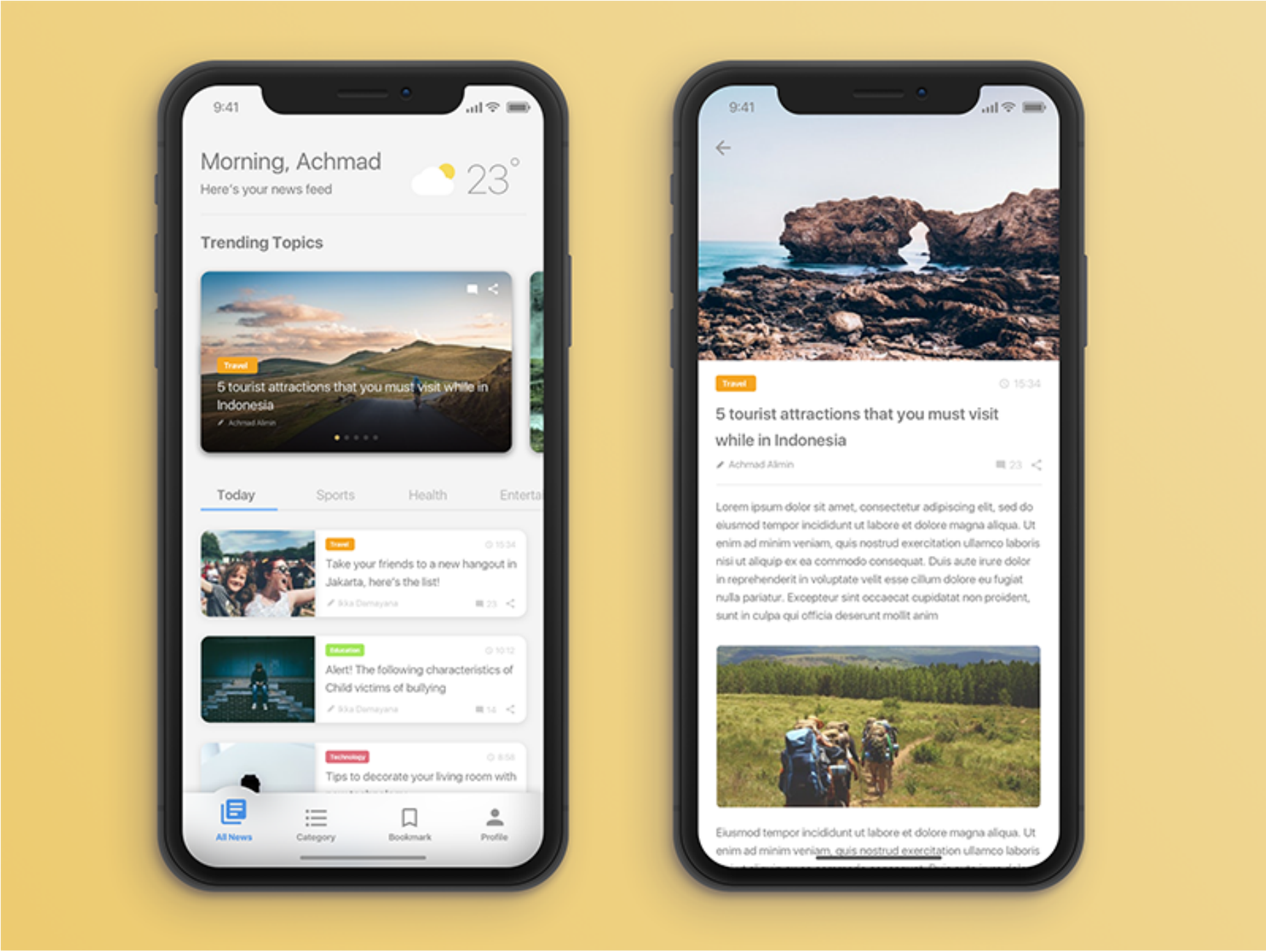
5 2018.09.04 05:53:00 字数 3,726 阅读 10,745

本文是我在上海 T 沙龙4月7日分享内容的文字版总结和拓展。相关视频和文档请见链接：[深入浅出 iOS 并发编程](#)

其中主要内容包括：GCD与Operation的用法、并发编程中常见的问题、使用Operation进行流程化开发示范。

## 什么是并发编程

在大多数场景下，我们所写的代码是逐行顺序执行——在固定的时段内，程序只执行一个任务。而所谓并发编程，就是指在固定的时段内，程序执行多个任务。举个例子，当我们在微博 App 的首页滑动浏览时，微博也在从网络端预加载新的内容或者图片。并发编程可以充分利用硬件性能，合理分配软件资源，带来优秀的用户体验。在 iOS 开发中，我们主要依靠 GCD 和 Operation 来操作线程切换、异步操作，从而实现并发编程。



新闻类App首页经常需要同时处理 UI 显示、内容加载、缓存等多个任务

在 iOS 并发编程中，我们要知道这几个基本概念：

百度智能云

2021

云

低至

消费满

故胤道长

总资产295 (约

百度智能云

2021

云

低至

消费满

在于，并发不会同时执行多个任务，而是通过在任务间不断切换去完成所有工作。

- 同步（Sync）：会把当前的任务加入到队列中，除非该任务执行完成，线程才会返回继续运行，也就是说同步会阻塞线程。任务在执行和结束一定遵循先后顺序，即先执行的任务一定先结束。
- 异步（Async）：会把当前的任务加入到队列中，但它会立刻返回，无需等任务执行完成，也就是说异步不会阻塞线程。任务在执行和结束不遵循先后顺序。可能先执行的任务先结束，也可能后执行的任务先结束。

为了进一步说明说明串行/并发与同步/异步之间的关系，我们来看下面这段代码会打印出什么内容：

```
1 // serial, sync
2 serialQueue.sync {
3     print(1)
4 }
5 print(2)
6 serialQueue.sync {
7     print(3)
8 }
9 print(4)
10
11 // serial, async
12 serialQueue.async {
13     print(1)
14 }
15 print(2)
16 serialQueue.async {
17     print(3)
18 }
19 print(4)
20
21 // serial, sync in async
22 print(1)
23 serialQueue.async {
24     print(2)
25     serialQueue.sync {
26         print(3)
27     }
28     print(4)
29 }
30 print(5)
31
32
33 // serial, async in sync
34 print(1)
35 serialQueue.sync {
36     print(2)
37     serialQueue.async {
38         print(3)
39     }
40     print(4)
41 }
42 print(5)
```

首先，在串行队列上进行同步操作，所有任务将顺序发生，所以第一段的打印结果一定是1234；

其次，在串行队列上进行异步操作，此时任务完成的顺序并不保证。所以可能会打印出这几种结果：1234，2134，1243，2413，2143。注意1一定在3之前打印出来，因为前者在后者之前派发，串行队列一次只能执行一个任务，所以一旦派发完成就执行。同理2一定在4之前打印，2一定在3之前打印。

接着，对同一个串行队列中进行异步、同步嵌套。这里会构成死锁（具体原因参见下文），所以只会打印出 125 或者 152。

最后，在串行队列中进行同步、异步嵌套，不会构成死锁。这里会打印出 3 个结果：12345，12435，12453。这里1一定在最前，2 一定在 4 前，4 一定在 5 前。

现在我们把串行队列改为并发队列：

```
1 // concurrent, sync
2 concurrentQueue.sync {
3     print(1)
4 }
5 print(2)
6 concurrentQueue.sync {
7     print(3)
8 }
9 print(4)
10
11 // concurrent, async
12 concurrentQueue.async {
13     print(1)
14 }
15 print(2)
16 concurrentQueue.async {
17     print(3)
18 }
19 print(4)
20
21 // concurrent, sync in async
22 print(1)
23 concurrentQueue.async {
24     print(2)
25     concurrentQueue.sync {
26         print(3)
27     }
28     print(4)
29 }
30 print(5)
31
32
33 // concurrent, async in sync
34 print(1)
35 concurrentQueue.sync {
36     print(2)
37     concurrentQueue.async {
38         print(3)
39     }
40     print(4)
41 }
42 print(5)
```

首先，在并发队列上进行同步操作，所有任务将顺序执行、顺序完成，所以第一段的打印结果一定是 1234；

其次，在并发队列上进行异步操作，因为并行对列有多个线程 。所以这里只能保证 24 顺序执行，13 乱序，可能插在任意位置：2413 ，2431，2143，2341，2134，2314。

接着，对同一个并发队列中进行异步、同步嵌套。这里不会构成死锁，因为同步操作只会阻塞一个线程，而并发队列对应多个线程。这里会打印出 4 个结果：12345，12534，12354，15234。注意同步操作保证了 3 一定会在 4 之前打印出来。

最后，在并发队列中进行同步、异步嵌套，不会构成死锁。而且由于是并发队列，所以在运行异步操作时也同时会运行其他操作。这里会打印出 3 个结果：12345，12435，12453。这里



同步操作保证了 2 和 4 一定分别在 3 和 5 之前打印出来。

在实际开发中，我们还需要知道主线程的特性、GCD 和 Operation 的 API、如发现并调试并发编程中的技巧。

## GCD vs. Operation

在 iOS 开发中，我们一般用 GCD 和 Operation 来处理并发编程问题。我们先来看看 GCD 的基本用法：

```
1 // serial queue
2 let serialQueue = DispatchQueue(label: "serial")
3
4 // global queue, gcd defined concurrent queue
5 let globalQueue = DispatchQueue.global(qos: .default)
6
7 // custom concurrent queue
8 let concurrentQueue = DispatchQueue(label: "concurrent", attributes: .concurrent)
```

其中，全局队列的优先级由 QoS (Quality of Service)决定。如果不指定优先级，就是默认（default）优先级。另外还有 background，utility，user-Initiated，unspecified，user-Interactive。下面按照优先级顺序从低到高来排列：

- **Background**：用来处理特别耗时的后台操作，例如同步、数据持久化。
- **Utility**：用来处理需要一点时间而又不需要立刻返回结果的操作。特别适用于网络加载、计算、输入输出等。
- **Default**：默认优先级。一般来说开发者应该指定优先级。属于特殊情况。
- **User-Initiated**：用来处理用户触发的、需要立刻返回结果的操作。比如打开用户点击的文件、加载图片等。
- **User-Interactive**：用来处理用户交互的操作。一般用于主线程，如果不及时响应就可能阻塞主线程的操作。
- **Unspecified**：未确定优先级，由系统根据不同环境推断。比如使用过时的 API 不支持优先级，此时就可以设定为未确定优先级。属于特殊情况。

在日常开发中，GCD 的常见应用有处理后台任务、延时、单例（Objective-C）、线程组等操作，这里不作赘述。下面我们来看看 Operation 的基本操作：

```
1 // serial queue
2 let serialQueue = OperationQueue()
3 serialQueue.maxConcurrentOperationCount = 1
4
5 // concurrent queue
6 let concurrentQueue = OperationQueue()
```

Operation 作为 NSObject 的子类，一般被用于单独的任务。我们将其继承重写之后加入到 OperationQueue 中去运行。iOS 亦提供 BlockOperation 这个子类去方便地执行多个代码片段。相比于 GCD，Operation 最主要的特点在于其拥有暂停、继续、终止等多个可控状态，从而可以更加灵活得适应并发编程的场景。

基于 Operation 和 GCD API 的特点，我们可以得出以下结论：GCD 适用于处理并行开发中的简单小任务，总体写法轻便快捷；Operation 适合于封装模块化的任务，支持多任务之间相互依赖的场景。两者之间的区别同 UIAnimation 和 CALayor Animation 差别异曲同工——由此可见苹

果在设计 API 时一以贯之的思路：提供一个简单快捷的 API 满足80%的场景，在提供一套更全面的 API 应对剩下20%更复杂的场景。

## 并发编程中常见问题

在并发编程中，一般会面对这样的三个问题：竞态条件、优先倒置、死锁问题。针对 iOS 开发，它们的具体定义为：

- **竞态条件（Race Condition）**。指两个或两个以上线程对共享的数据进行读写操作时，最终的数据结果不确定的情况。例如以下代码：

```
1  var num = 0
2  DispatchQueue.global().async {
3      for _ in 1...10000 {
4          num += 1
5      }
6  }
7
8  for _ in 1...10000 {
9      num += 1
10 }
```

最后的计算结果 num 很有可能小于 20000，因为其操作为非原子操作。在上述两个线程对num进行读写时其值会随着进程执行顺序的不同而产生不同结果。

竞态条件一般发生在多个线程对同一个资源进行读写时。解决方法有两个，第一是串行队列加同步操作，无论读写，指定时间只能优先做当前唯一操作，这样就保证了读写的安全。其缺点是速度慢，尤其在大量读写操作发生时，每次只能做单个读或写操作的效率实在太低。另一个方法是，用并发队列和 barrier flag，这样保证此时所有并发队列只进行当前唯一的写操作（类似将并发队列暂时转为串行队列），而无视其他操作。

- **优先倒置（Priority Inverstion）**。指低优先级的任务会因为各种原因先于高优先级任务执行。例如以下代码：

```
1  var highPriorityQueue = DispatchQueue.global(qos: .userInitiated)
2  var lowPriorityQueue = DispatchQueue.global(qos: .utility)
3
4  let semaphore = DispatchSemaphore(value: 1)
5
6  lowPriorityQueue.async {
7      semaphore.wait()
8      for i in 0...10 {
9          print(i)
10     }
11     semaphore.signal()
12 }
13
14 highPriorityQueue.async {
15     semaphore.wait()
16     for i in 11...20 {
17         print(i)
18     }
19     semaphore.signal()
20 }
```

上述代码如果没有 semaphore，高优先权的 highPriorityQueue 会优先执行，所以程序会优先打印完 11 到 20。而加了 semaphore 之后，低优先权的 lowPriorityQueue 会先挂起

semaphore，高优先权的highPriorityQueue 就只有等 semaphore 被释放才能再执行打印。

也就是说，低优先权的线程可以锁上某种高优先权线程需要的资源，从而优于迫使高优先权的线程等待低优先权的线程，这就叫做优先倒置。其对应的解决方法是，对同一个资源不同队列的操作，我们应该用同一个QoS指定其优先级。

- **死锁问题（Dead Lock）**。指两个或两个以上的线程，它们之间互相等待彼此停止执行，以获得某种资源，但是没有一方会提前退出的情况。iOS 中有个经典的例子就是两个 Operation 互相依赖：

```
1 | let operationA = Operation()
2 | let operationB = Operation()
3 |
4 | operationA.addDependency(operationB)
5 | operationB.addDependency(operationA)
```

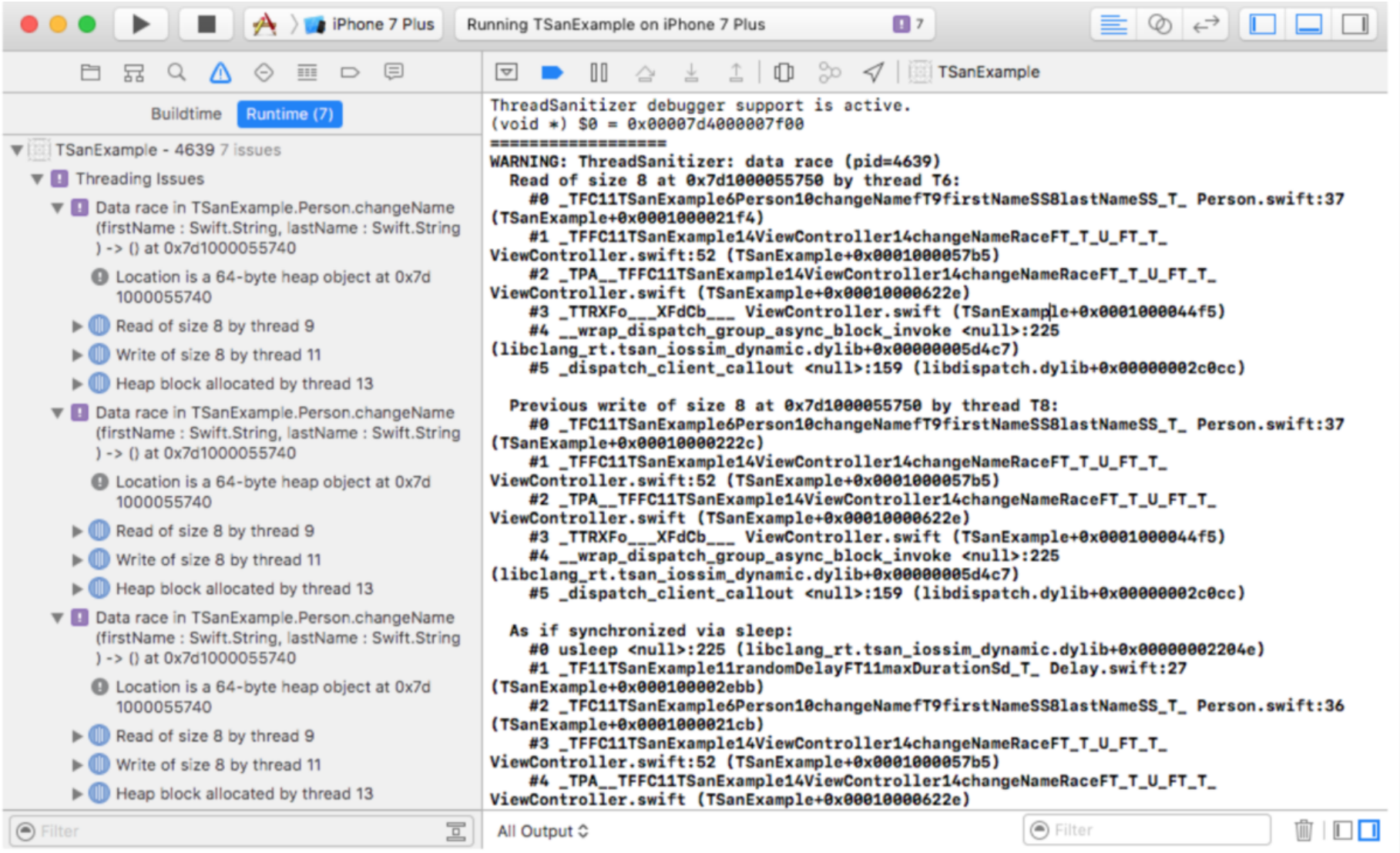
还有一种经典的情况，就是在对同一个串行队列中进行异步、同步嵌套：

```
1 | serialQueue.async {
2 |     serialQueue.sync {
3 |     }
4 | }
```

因为串行队列一次只能执行一个任务，所以首先它会把异步 block 中的任务派发执行，当进入到 block 中时，同步操作意味着阻塞当前队列 。而此时外部 block 正在等待内部 block 操作完成，而内部block 又阻塞其操作完成，即内部 block 在等待外部 block 操作完成。所以串行队列自己等待自己释放资源，构成死锁。

对于死锁问题的解决方法是，注意Operation的依赖添加，以及谨慎使用同步操作。其实聪明的读者应该已经发现，在主线程使用同步操作是一定会构成死锁的，所以我个人建议在串行队列中不要使用同步操作。

尽管我们已经知道了并发编程中的问题，以及其对应方法。但是日常开发中，我们怎样及时发现这些问题呢？其实 Xcode 提供了一个非常便利的工具 —— Thread Sanitizer (TSan)。在 Schemes中勾选之后，TSan就会将所有的并发问题在 Runtime 中显示出来，如下图：



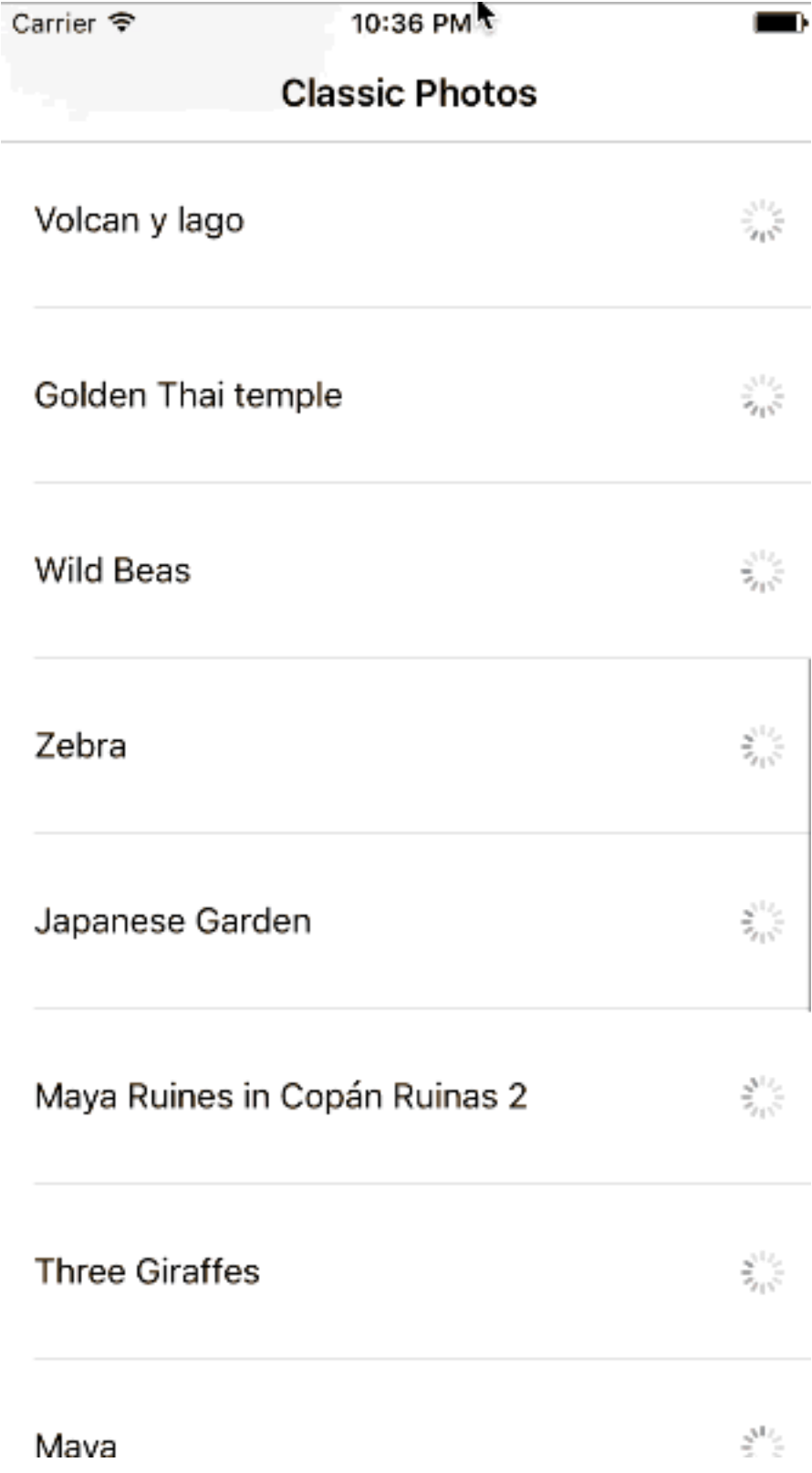


这里我们有7个线程问题，TSan清晰地告诉了我们这是读写问题，展开之后会告诉我们具体触发代码，十分方便。16年的WWDC上，苹果也郑重向大家宣告，如果有并发问题，请记得用TSan。

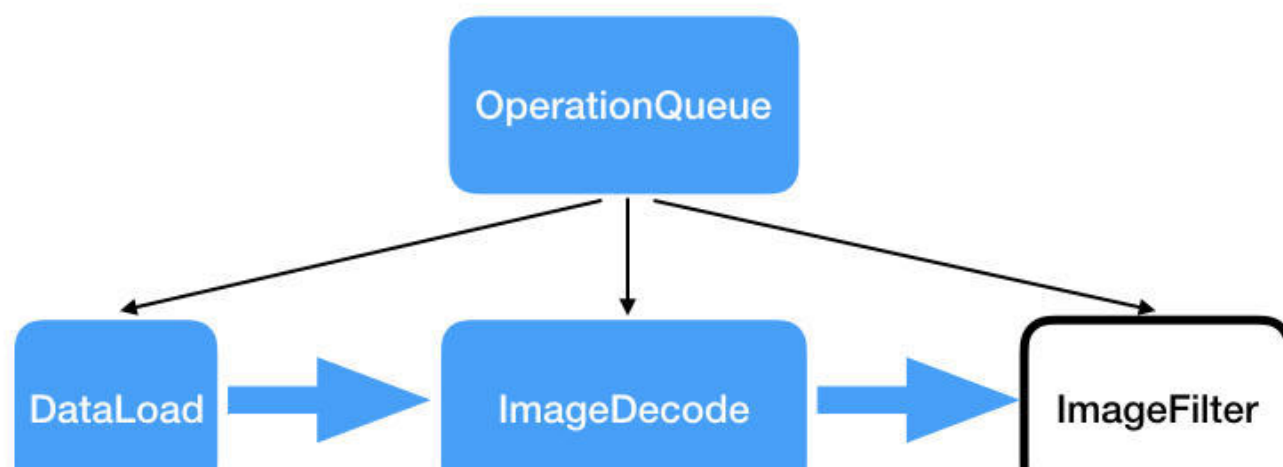
## Operation 流程化开发

上文中提到 Operation 特别适合模块化工作，也支持多任务的互相依赖。这里我们就来看一个具体的开发案例吧：

实现一个相册 App，其首页是个滑动列表（Table View）。列表每行展示加上了滤镜的图片。具体实现如下图：



仔细分析一下相关的操作，实际上就是三步：先加载数据，然后解码成图片，最后再给图片加上滤镜。所以用 Operation 实现起来如下图：



对于加载数据，我们可以定义如下的 Operation 子类来进行操作：

```
1 class DataLoadOperation: Operation {
2
3     fileprivate let url: URL
4     fileprivate var loadedData: Data?
5     fileprivate let completion: ((Data?) -> ())?
6
7     init(url: URL, completion: ((Data?) -> ())? = nil) {
8         ...
9     }
10
11     override func main() {
12         if isCancelled { return }
13         ImageService.loadData(at: url) { data in
14             if isCancelled { return }
15             loadedData = data
16             completion?(data)
17         }
18     }
19 }
```

这里我们要注意，DataLoadOperation中的三个变量皆为私有。这是因为其实后续图片解码操作并不关心数据是如何操作的，它只关心是否能提供解码图片的数据，所以我们可以用 Protocol 来提供这个借口即可：

```
1 // 此协议定义应和 ImageDecodeOperation 放在同一文件
2 protocol ImageDecodeOperationDataProvider {
3     var encodedData: Data? { get }
4 }
5
6 // 次扩展应和 DataLoadOperation 放在同一文件
7 extension DataLoadOperation: ImageDecodeOperationDataProvider {
8     var encodedData: Data? { return loadedData }
9 }
```

接着再来看看解码图片的 Operation 如何实现：

```
1 class ImageDecodeOperation: Operation {
2
3     fileprivate let inputData: Data?
4     fileprivate var outputImage: UIImage?
5     fileprivate let completion: ((UIImage?) -> ())?
6
7     init(data: Data?, completion: ((UIImage?) -> ())? = nil) {
8         ...
9     }
10
11     override func main() {
12         let encodedData: Data?
13         if isCancelled { return }
14         if let inputData = inputData {
```



```

15     encodedData = inputData
16 } else {
17     let dataProvider = dependencies
18         .filter { $0 is ImageDecodeOperationDataProvider }
19         .first as? ImageDecodeOperationDataProvider
20     encodedData = dataProvider?.encodedData
21 }
22
23 guard let data = encodedData else { return }
24
25 if isCancelled { return }
26 if let decodedData = Decoder.decodeData(data) {
27     outputImage = UIImage(data: decodedData)
28 }
29
30 if isCancelled { return }
31 completion?(outputImage)
32 }
33 }
34
35 extension ImageDecodeOperation: ImageFilterDataProvider {
36     var image: UIImage? { return outputImage }
37 }

```

最后我们再来看 ImageFilterOperation 及其子类如何实现。这里由于直接输出 Image，所以就无需用：

```

1  protocol ImageFilterDataProvider {
2      var image: UIImage? { get }
3  }
4
5  class ImageFilterOperation: Operation {
6      fileprivate let filterInput: UIImage?
7      fileprivate var filterOutput: UIImage?
8      fileprivate let completion: ((UIImage?) -> ())?
9
10     init(image: UIImage?, completion:
11         ((UIImage?) -> ())? = nil) {
12         ...
13     }
14
15     var filterInput: UIImage? {
16         var image: UIImage?
17         if let inputImage = _filterInput {
18             image = inputImage
19         } else if let dataProvider = dependencies
20             .filter({ $0 is ImageFilterDataProvider })
21             .first as? ImageFilterDataProvider {
22             image = dataProvider.image
23         }
24         return image
25     }
26 }
27
28 // LarkFilter 和 ReyesFilter 的实现也类似
29 class MoonFilterOperation : ImageFilterOperation {
30     override func main() {
31         if isCancelled { return }
32         guard let filterInput = filterInput else { return }
33
34         if isCancelled { return }
35         filterOutput = filterInput.applyMoonEffect()
36         if isCancelled { return }
37         completion(imageFiltered)
38     }
39 }

```

最后我们用 OperationQueue 将这些 Operation 拼接在一起：

```
1 let operationQueue = OperationQueue()
2 let dataLoadOperation = DataLoadOperation(url: url)
3 let imageDecodeOperation = imageDecodeOperation(data: nil)
4 let moonFilterOperation = MoonFilterOperation(image: nil, completion: completion)
5 let operations = [dataLoadOperation, imageDecodeOperation, moonFilterOperation]
6
7 // Add dependencies
8 imageDecodeOperation.addDependency(dataLoadOperation)
9 moonFilterOperation.addDependency(imageDecodeOperation)
10
11 operationQueue.addOperations(operations, waitUntilFinished: false)
```

大功告成。从上面我们可以发现，每个操作模块都可以用 Operation 进行自定义和封装。模块的对应逻辑非常清楚，代码复用率和灵活度也非常之高。如果要继续改进，我们还可以实现一个 AsyncOperation 的类，然后让 DataLoadOperation 继承该类，这样数据加载由同步变为异步，其效率会大大提高。

## 总结

iOS 开发中，并发编程主要用于提升 App 的运行性能，保证App实时响应用户的操作。主线程一般用于负责 UI 相关操作，如绘制图层、布局、交互相应。很多 UIKit 相关的控件如果不在主线程操作，会产生未知效果。Xcode 中的 Main Thread Checker 可以将相关问题检测出来并报错。

其他线程例如后天线程一般用来处理比较耗时的工作。网络请求、数据解析、复杂计算、图片的编码解码管理等都属于耗时的工作，应该放在其他线程处理。iOS 提供了两套灵活丰富的 API：GCD 和 Operation。GCD的优点在于简单快捷，Operation 胜在功能丰富、适合模块化操作。我们享受其便利的同时，也应该及时发现和处理并发编程中的三大问题。

[< 上一篇](#)

[查看连载目录](#)

[下一篇 >](#)



142人点赞 >





"我是道长，欢迎点赞"

赞赏支持

还没有人赞赏，支持一下



**故胤道长** 

卡内基梅隆大学硕士毕业，常年居住于美国的 iOS 开发者。 现 Q...

总资产295 (约28.32元)


共写了8.3W字

获得3,896个赞

共7,353个粉丝

关注

文章来自以下连载



**技术分享**

4.9W字 201,546阅读 332人关注

关注连载



种牙的利弊



輿情监测平台



新出大型网游



个人写真 昌平



一颗全瓷牙费用

广告

## 被以下专题收入，发现更多相似内容

-  iOS开发攻城...
-  iOS 开发
-  iOS技术博客...
-  技术收藏
-  iOS
-  移动开发技术前沿
-  好东西
- 展开更多 

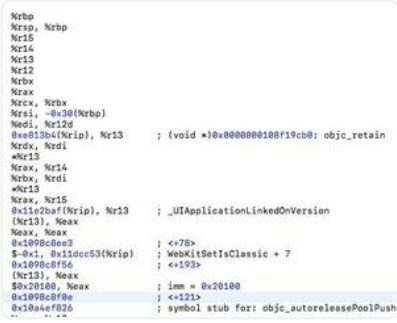
## 推荐阅读

更多精彩内容 

### iOS多线程编程

iOS多线程编程 基本知识 1. 进程（process）进程是指在系统中正在运行的一个应用程序，就是一段程序的执...

 陵无山 阅读 297 评论 0 赞 4



### 超详细！iOS 并发编程之 Operation Queues

原文链接：<http://www.cocoachina.com/ios/20150807/12911.html> 现如...

 Kevin追梦先生 阅读 494 评论 0 赞 3

### iOS 面试宝典 没有比这更全的了（持续更新）

1.ios高性能编程 (1).内层 最小的内层平均值和峰值(2).耗电量 高效的算法和数据结构(3).初始化时...

 欧辰\_OSR 阅读 12,455 评论 7 赞 164

### iOS并发编程--GCD、操作队列、线程

现在iOS的多线程方案主要有以下几种： GCD（Grand Central Dispatch）： 使用dispat...

 寒光冷剑 阅读 268 评论 0 赞 1

### Java面试宝典Beta5.0

pdf下载地址：Java面试宝典 第一章内容介绍 20 第二章JavaSE基础 21 一、Java面向对象 21 ...

 王震阳 阅读 76,663 评论 25 赞 508