

# iOS开发： 深入理解GCD 第二篇

## (dispatch\_group、dispatch\_barrier、基于线程安全的多读单写)

### 1. Dispatch Group

在追加到Dispatch Queue中的多个任务处理完毕之后想执行结束处理，这种需求会经常出现。如果只是使用一个Serial Dispatch Queue（串行队列）时，只要将想执行的处理全部追加到该串行队列中并在最后追加结束处理即可，但是在使用Concurrent Queue时，可能会同时使用多个Dispatch Queue时，源代码就会变得很复杂。

在这种情况下，就可以使用Dispatch Group。

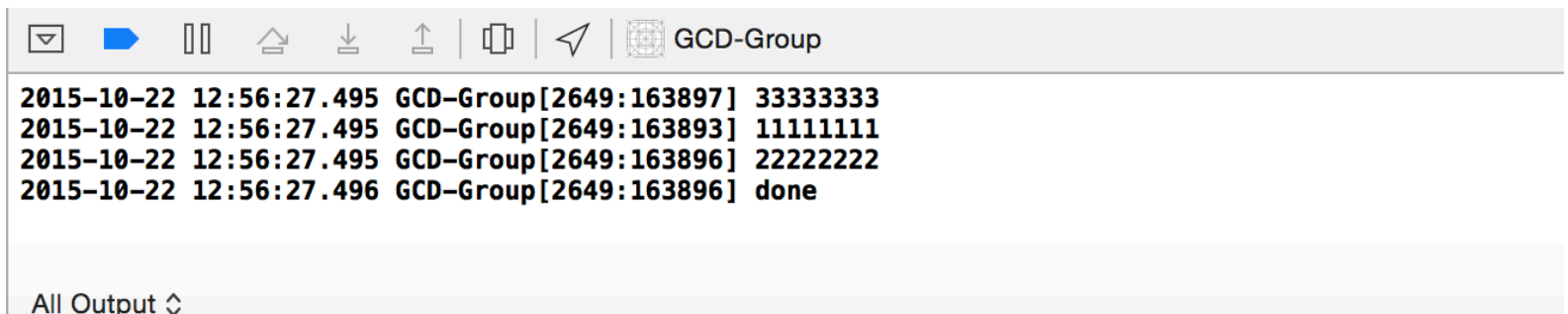
```
1
2  dispatch_group_t group = dispatch_group_create();
3      dispatch_queue_t queue = dispatch_queue_create("com.gcd-
group.www", DISPATCH_QUEUE_CONCURRENT);
4
5      dispatch_group_async(group, queue, ^{
6          for (int i = 0; i < 1000; i++) {
7              if (i == 999) {
8                  NSLog(@"11111111");
9              }
10         }
11     });
12
13     dispatch_group_async(group, queue, ^{
```

```

14         NSLog(@"22222222");
15     });
16
17     dispatch_group_async(group, queue, ^{
18         NSLog(@"33333333");
19     });
20
21     dispatch_group_notify(group, queue, ^{
22         NSLog(@"done");
23     });

```

控制台的输出：



```

2015-10-22 12:56:27.495 GCD-Group[2649:163897] 33333333
2015-10-22 12:56:27.495 GCD-Group[2649:163893] 11111111
2015-10-22 12:56:27.495 GCD-Group[2649:163896] 22222222
2015-10-22 12:56:27.496 GCD-Group[2649:163896] done

```

All Output ▾

因为向Concurrent Dispatch Queue追加处理，多个线程并行执行，所以追加处理的执行顺序不定。执行顺序会发生变化，但是此执行结果的done一定是最后输出的。

无论向什么样的Dispatch Queue中追加处理，使用Dispatch Group都可以监视这些处理执行的结果。一旦检测到所有处理执行结束，就可以将结束的处理追加到Dispatch Queue中，这就是使用Dispatch Group的原因。

下面试一个使用Dispatch Group异步下载两张图片，然后合并成一张图片的medo（注意，我们总是应该在主线程中更新UI）：

```

1

```

```
2
3
4
5
6 #import "ViewController.h"
7
8 @interface ViewController ()
9
10 @property (nonatomic, strong) UIImage *imageOne;
11 @property (nonatomic, strong) UIImage *imageTwo;
12 @property (nonatomic, weak) UILabel *textLabel;
13
14 @end
15
16 @implementation ViewController
17
18 - (void)viewDidLoad {
19     [super viewDidLoad];
20
21     [self operation1];
22 }
23
24 - (void)operation1
25 {
26     UILabel *textLabel = [[UILabel alloc] initWithFrame:CGRectMake(
27 450, 0, 0)];
28
29     textLabel.text = @"正在下载图片";
30
31     [textLabel sizeToFit];
32
33     [self.view addSubview:textLabel];
34
35     self.textLabel = textLabel;
36
37     [self group];
38
39     NSLog(@"在下载图片的时候，主线程貌似还可以干点什么");
40 }
41
42 - (void)group
```

```
27 {
28     UIImageView *imageView = [[UIImageView alloc] init];
29     [self.view addSubview:imageView];
30
31     dispatch_group_t group = dispatch_group_create();
32     dispatch_queue_t queue = dispatch_queue_create("cn.gcd-group",
33     DISPATCH_QUEUE_CONCURRENT);
34
35     dispatch_group_async(group, queue, ^{
36         NSLog(@"正在下载第一张图片");
37
38         NSData *data = [NSData dataWithContentsOfURL:[NSURL
39         URLWithString:@"http://images2015.cnblogs.com/blog/471463/201509,
40         20150912213125372-589808688.png"]];
41
42         NSLog(@"第一张图片下载完毕");
43
44         self.imageOne = [UIImage imageWithData:data];
45     });
46
47     dispatch_group_async(group, queue, ^{
48         NSLog(@"正在下载第二张图片");
49
50         NSData *data = [NSData dataWithContentsOfURL:[NSURL
51         URLWithString:@"http://images2015.cnblogs.com/blog/471463/201509,
52         20150912212457684-585830854.png"]];
53
54         NSLog(@"第二张图片下载完毕");
55
56         self.imageTwo = [UIImage imageWithData:data];
57     });
58
59     dispatch_group_notify(group, queue, ^{
60         UIGraphicsBeginImageContext(CGSizeMake(300, 400));
61
62         [self.imageOne drawInRect:CGRectMake(0, 0, 150, 400)];
```

```

52         [self.imageTwo drawInRect:CGRectMake(150, 0, 150, 400)];
53
54         UIImage *newImage = UIGraphicsGetImageFromCurrentImageContext();
55         UIGraphicsEndImageContext();
56
57         dispatch_async(dispatch_get_main_queue(), ^{
58             UIImageView *imageView = [[UIImageView alloc]
59 initWithImage:newImage];
60             [self.view addSubview:imageView];
61             self.textLabel.text = @"图片合并完毕";
62         });
63     });
64 }
65 @end
66
67

```

## 2. dispatch\_barrier\_async

在访问数据库或者文件的时候，我们可以使用Serial Dispatch Queue可避免数据竞争问题，代码如下所示：

先看看，如果我们在平常编码中，如果要保证某个属性可以线程安全的读写，如何写的：

```

1
2
3
4
5     #import <Foundation/Foundation.h>
6
7     @interface ZYPerson : NSObject
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

```
6  @property (nonatomic, copy) NSString *name;
7
8  #import "ZYPerson.h"
9
10 static NSString *_name;
11
12 @implementation ZYPerson
13
14 - (void)setName:(NSString *)name
15 {
16     @synchronized(self) {
17         _name = [name copy];
18     }
19 }
20
21 - (NSString *)name
22 {
23     @synchronized(self) {
24         return _name;
25     }
26 }
27
28 @end
```

这是我在刚学iOS开发，刚涉及并发中的数据竞争时，书本上提到的一种解决方案。如果有多个线程要执行同一份代码，那么有时候可能会出现问題，这种情况下，通常要使用锁来实现某种同步机制。iOS提供了一种加锁的方式，就是采用内置的synchronization block，也就是上面代码所写的。

这种写法会根据给定的对象，自动创建一个锁，并等待块中的代码执

行完毕。执行到这段代码结尾处，锁也就释放了。在上面的例子中，同步行为所针对的对象是self。这么写通常没错，但是@synchronized(self)会大大降低代码效率，甚至很多时候，还可以被人感觉到效率明显下降了，因为共用同一个锁的那些同步块，都必须按顺序执行。若在self对象上频繁加锁，那么程序可能就要等另一段与此无关的代码执行完毕，才可以继续执行当前代码，这样做是很没必要的。

@synchronized(self)会大大降低代码效率，因为所有的同步块（@synchronized(self)）都会彼此抢夺同一个锁。要是有多属性这么写，每个属性的同步块（@synchronized(self)）都要等其他所有的同步块执行完毕之后才能执行，这并不是我们想要的结果，我们只想要每个属性各自独立的同步。

还有，不得不说，按上面这么做，虽然可以在一定程度上提供“线程安全”，但却无法保证访问该对象时是绝对线程安全的。事实上，上面的写法，就是atomic，也就是原子性属性xcode自动生成的代码，这种方法，在访问属性时，必定可以从中得到有效值，然而如果在一个线程上多次调用getter方法，每次得到的结果却未必相同，在两次读操作之间，其他线程可能会写入新的属性值。

其实使用GCD可以简单高效的代替同步块或者锁对象，可以使用，串行同步队列，将读操作以及写操作都安排在同一队列里，即可保证数据同步，代码如下：

|   |  |
|---|--|
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 | <code>#import &lt;Foundation/Foundation.h&gt;</code>     |
| 6 | <code>@interface ZYPerson : NSObject</code>              |
| 7 | <code>@property (nonatomic, copy) NSString *name;</code> |
| 8 | <code>@end</code>  |

```
9  #import "ZYPerson.h"

10 @interface ZYPerson ()

11 @end

12 static NSString *_name;

13 static dispatch_queue_t _queue;

14 @implementation ZYPerson

15 - (instancetype)init

16 {

17     if (self = [super init]) {

18         _queue = dispatch_queue_create("com.person.syncQueue",
19 DISPATCH_QUEUE_SERIAL);

19     }

20     return self;

21 }

22 - (void)setName:(NSString *)name

23 {

24     dispatch_sync(_queue, ^{

25         _name = [name copy];

26     });

27 }

28 - (NSString *)name

29 {

30     __block NSString *tempName;

31     dispatch_sync(_queue, ^{

32         tempName = _name;

33     });

34     return tempName;

35 }
```



|    |      |
|----|------|
| 34 | @end |
| 35 |      |
| 36 |      |
| 37 |      |
| 38 |      |
| 39 |      |

这样写的思路是：把写操作与读操作都安排在同一个同步串行队列里面执行，这样的话，所有针对属性的访问操作就都同步了。这种方法的确已经足够好了，但还不是最优的，它只可以实现单读、单写。整体来看，我们最终要解决的问题是，在写的过程中不能被读，以免数据不对，但是读与读之间并没有任何的冲突！

多个getter方法（也就是读取）是可以并发执行的，而getter(读)与setter（写）方法是不能并发执行的，利用这个特点，还能写出更快的代码来，这次注意，不用串行队列，而改用并行队列：

|    |  |
|----|--|
| 1  |  |
| 2  |  |
| 3  |  |
| 4  | <code>#import &lt;Foundation/Foundation.h&gt;</code>     |
| 5  | <code>@interface ZYPerson : NSObject</code>              |
| 6  | <code>@property (nonatomic, copy) NSString *name;</code> |
| 7  | <code>@end</code>  |
| 8  | <code>#import "ZYPerson.h"</code>                        |
| 9  | <code>@interface ZYPerson ()</code>                      |
| 10 | <code>@end</code>  |
| 11 | <code>static NSString *_name;</code>                     |
| 12 | <code>static dispatch_queue_t _concurrentQueue;</code>   |

```
13 @implementation ZYPerson
14 - (instancetype)init
15 {
16     if (self = [super init]) {
17         _concurrentQueue =
dispatch_queue_create("com.person.syncQueue",
DISPATCH_QUEUE_CONCURRENT);
18     }
19     return self;
20 }
21 - (void)setName:(NSString *)name
22 {
23     dispatch_barrier_async(_concurrentQueue, ^{
24         _name = [name copy];
25     });
26 }
27 - (NSString *)name
28 {
29     __block NSString *tempName;
30     dispatch_sync(_concurrentQueue, ^{
31         tempName = _name;
32     });
33     return tempName;
34 }
35 @end
36
37
```

这样优化，测试一下性能，可以发现这种做法肯定比使用串行队列要快。

在这个代码中，我用了点新的东西，`dispatch_barrier_async`，可以翻译成栅栏（barrier），它可以往队列里面发送任务（块，也就是block），这个任务有栅栏（barrier）的作用。

在队列中，barrier块必须单独执行，不能与其他block并行。这只对并发队列有意义，并发队列如果发现接下来要执行的block是个barrier block，那么就一定要等到当前所有并发的block都执行完毕，才会单独执行这个barrier block代码块，等到这个barrier block执行完毕，再继续正常处理其他并发block。在上面的代码中，setter方法中使用了barrier block以后，对象的读取操作依然是可以并发执行的，但是写入操作就必须单独执行了。

### 3. 附录（也算是追加）

也许会对并行、串行、异步、同步有所理解不深。

并行：就是队列里面的任务（代码块，block）不是一个个执行，而是并发执行，也就是可以同时执行的意思

串行：队列里面的任务一个接着一个执行，要等前一个任务结束，下一个任务才可以执行

异步：具有新开线程的能力

同步：不具有新开线程的能力，只能在当前线程执行任务

那么，如果他们相互串起来，会怎么样呢？

并行+异步：就是真正的并发，新开有有多个线程处理任务，任务并发执行（不按顺序执行）

串行+异步：新开一个线程，任务一个接一个执行，上一个任务处理完毕，下一个任务才可以被执行

并行+同步：不新开线程，任务一个接一个执行

串行+同步：不新开线程，任务一个接一个执行

如此，并行+异步，串行+异步倒是很好辨别，他们有各自的特点，那么并行+同步，串行+同步，貌似效果是一样的，好像用哪种都一

样??!!

不，这其中区别很大（不得不说一句，我所看多线程书籍，基本没讲到这点的，不知道是不是他们都认为这是基础，没必要说明）。

这其中的区别，我觉得代码体现更直观，直接上代码吧：

这一份是并行+同步，我在外面并发添加代码块：

```
1 dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
2 ^{
3     dispatch_sync(dispatch_get_main_queue(), ^{
4         NSLog(@"11  %@", [NSThread currentThread]);
5     });
6
7     dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIOR
8 0), ^{
9         dispatch_sync(dispatch_get_main_queue(), ^{
10             NSLog(@"22  %@", [NSThread currentThread]);
11         });
12
13     dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIOR
14 0), ^{
15         dispatch_sync(dispatch_get_main_queue(), ^{
16             NSLog(@"33  %@", [NSThread currentThread]);
17         });
18
19     dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIOR
20 0), ^{
21         dispatch_sync(dispatch_get_main_queue(), ^{
22             NSLog(@"44  %@", [NSThread currentThread]);
```

```

22         });
23     });
24
25     dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
26 0), ^{
27         dispatch_sync(dispatch_get_main_queue(), ^{
28             NSLog(@"55  %@", [NSThread currentThread]);
29         });
30     });

```

看看打印：

```

2015-10-22 17:21:37.298 线程安全的多读单写[4920:344719] 11 <NSThread: 0x7f9daa401690>{number = 1, name = main}
2015-10-22 17:21:37.300 线程安全的多读单写[4920:344719] 22 <NSThread: 0x7f9daa401690>{number = 1, name = main}
2015-10-22 17:21:37.301 线程安全的多读单写[4920:344719] 33 <NSThread: 0x7f9daa401690>{number = 1, name = main}
2015-10-22 17:21:37.301 线程安全的多读单写[4920:344719] 44 <NSThread: 0x7f9daa401690>{number = 1, name = main}
2015-10-22 17:21:37.301 线程安全的多读单写[4920:344719] 55 <NSThread: 0x7f9daa401690>{number = 1, name = main}

```

上面这个代码可能描述的并不直观，毕竟我采用的是主队列，新增代码：

```

1
2
3
4     dispatch_queue_t queue = dispatch_queue_create("com.hao123.www",
5     DISPATCH_QUEUE_SERIAL);
6
7     dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
8 0), ^{
9         dispatch_sync(queue, ^{
10             NSLog(@"11  %@", [NSThread currentThread]);
11         });
12     });
13
14     dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
15 0), ^{
16         dispatch_sync(queue, ^{
17             NSLog(@"22  %@", [NSThread currentThread]);
18         });
19     });

```

```

13     });
14 });
15 dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
16     0), ^{
17     dispatch_sync(queue, ^{
18         NSLog(@"33  %@",[NSThread currentThread]);
19     });
20 });
21 dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
22     0), ^{
23     dispatch_sync(queue, ^{
24         NSLog(@"44  %@",[NSThread currentThread]);
25     });
26 });
27 dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
28     0), ^{
29     dispatch_sync(queue, ^{
30         NSLog(@"55  %@",[NSThread currentThread]);
31     });
32 });

```

可以发现，结论是一样的，里面的任务依次执行：

```

2016-05-24 11:20:41.757 线程安全的多读单写[2831:56893] 11 <NSThread: 0x7ff96b52fd70>{number = 2, name = (null)}
2016-05-24 11:20:41.759 线程安全的多读单写[2831:56899] 22 <NSThread: 0x7ff96b6232e0>{number = 3, name = (null)}
2016-05-24 11:20:41.759 线程安全的多读单写[2831:56908] 33 <NSThread: 0x7ff96b5024e0>{number = 4, name = (null)}
2016-05-24 11:20:41.759 线程安全的多读单写[2831:56909] 44 <NSThread: 0x7ff96b705380>{number = 5, name = (null)}
2016-05-24 11:20:41.760 线程安全的多读单写[2831:56910] 55 <NSThread: 0x7ff96b615fd0>{number = 6, name = (null)}

```

在一个线程里面，任务一个接一个的执行。

再看看，并行+同步，并发调用：

```
1
2
3 dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
4 0), ^{
5     dispatch_sync(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
6 0), ^{
7         NSLog(@"11  %@",[NSThread currentThread]);
8     });
9 });
10 dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
11 0), ^{
12     dispatch_sync(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
13 0), ^{
14         NSLog(@"22  %@",[NSThread currentThread]);
15     });
16 });
17 dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
18 0), ^{
19     dispatch_sync(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
20 0), ^{
21         NSLog(@"33  %@",[NSThread currentThread]);
22     });
23 });
24 dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
25 0), ^{
26     dispatch_sync(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
27 0), ^{
28         NSLog(@"44  %@",[NSThread currentThread]);
29     });
30 });
```

```

23 dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
    0), ^{
24     dispatch_sync(dispatch_get_global_queue(DISPATCH_QUEUE_PRIOR
    0), ^{
25
26         NSLog(@"55  %@", [NSThread currentThread]);
27     });
28 });
29

```

打印：

```

2015-10-22 17:24:06.169 线程安全的多读单写[4952:346971] 22 <NSThread: 0x7fd31a60d170>{number = 4, name = (null)}
2015-10-22 17:24:06.169 线程安全的多读单写[4952:346973] 11 <NSThread: 0x7fd31a60ccd0>{number = 2, name = (null)}
2015-10-22 17:24:06.169 线程安全的多读单写[4952:346980] 44 <NSThread: 0x7fd31a714270>{number = 5, name = (null)}
2015-10-22 17:24:06.169 线程安全的多读单写[4952:346972] 33 <NSThread: 0x7fd31a507960>{number = 3, name = (null)}
2015-10-22 17:24:06.169 线程安全的多读单写[4952:346981] 55 <NSThread: 0x7fd31a504cc0>{number = 6, name = (null)}

```

新开了多个线程，任务并发执行。

至此，难道可以说，我之前的所说的

并行+同步：不新开线程，任务一个接一个执行

串行+同步：不新开线程，任务一个接一个执行

是错的？不，并没有错，只是这种说法，是将任务添加到这种模式的内部。

朋友们，虽然这个世界日益浮躁起来，只要能够为了当时纯粹的梦想和感动坚持努力下去，不管其它人怎么样，我们也能够保持自己的本色走下去。