GCD

Queue

GCD有三种queue.

- main queue: 主线程队列。是一个串行队列。一般用来更新UI。
- global queue: 全局队列，是一个并行队列。使用方法相信大家都知道。

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORIT
Y_DEFAULT, 0), ^{
        [self dosomething];
    });
```

上面的意思就是开启一个异步线程，在全局队列中执行。

- custom queue:自定义队列。有两种自定义队列。

```
 dispatch_queue_t serial_queue = dispatch_queue_create("com.rev
iewcode.www", DISPATCH_QUEUE_SERIAL);
 dispatch_queue_t concurrent_queue = dispatch_queue_create("com
.zangqilong.www", DISPATCH_QUEUE_CONCURRENT);
```

`serial_queue`即是我自定义的一个串行队列。上面提到的主线程队列也是一个串行队列。

`concurrent_queue`是我自定义的一个并行队列。上面提到的global queue就是一个并行队列。

现在我们来讨论2个问题。
1. `dispatch_async`里使用串行队列和并行队列的效果。

2. `dispatch_sync`里使用串行队列和并行队列的效果。

串行队列和并行队列的创建如下。

```
 serial_queue = dispatch_queue_create("com.reviewcode.www", DIS
PATCH_QUEUE_SERIAL);
 concurrent_queue = dispatch_queue_create("com.zangqilong.www",
 DISPATCH_QUEUE_CONCURRENT);
```

讨论的问题

在dispatch_async使用串行队列

看代码。

```
- (void)testSerialQueueWithAsync
{
    for (int index = 0; index < 10; index++) {
        dispatch_async(serial_queue, ^{
            NSLog(@"index = %d", index);
            NSLog(@"current thread is %@", [NSThread currentThr
ead]);
        });

    }

    NSLog(@"Running on main Thread");
}
```

然后在`viewDidLoad()`方法里打印。打印结果如下。

```
index = 0
Running on main Thread
current thread is <NSThread: 0x7fad5be0f020>{number = 2, name =
 (null)}
index = 1
current thread is <NSThread: 0x7fad5be0f020>{number = 2, name =
 (null)}
index = 2
current thread is <NSThread: 0x7fad5be0f020>{number = 2, name =
 (null)}
index = 3
current thread is <NSThread: 0x7fad5be0f020>{number = 2, name =
 (null)}
index = 4
current thread is <NSThread: 0x7fad5be0f020>{number = 2, name =
```

```
 (null)}
index = 5
current thread is <NSThread: 0x7fad5be0f020>{number = 2, name =
 (null)}
index = 6
current thread is <NSThread: 0x7fad5be0f020>{number = 2, name =
 (null)}
index = 7
current thread is <NSThread: 0x7fad5be0f020>{number = 2, name =
 (null)}
index = 8
current thread is <NSThread: 0x7fad5be0f020>{number = 2, name =
 (null)}
index = 9
current thread is <NSThread: 0x7fad5be0f020>{number = 2, name =
 (null)}
```

打印结果的几个特征。

1. 在dispatch_async使用的所有Thread均为同一个Thread。因为他们的指针地址完全相同。

2. 输出结果是按顺序输出，符合我们对串行队列的期待。即FIFO。先进先出原则。
3. `Running on main Thread` 这句话并没有在最后执行，而是会出现在随机的位置，这也符合我们对dispatch_async的期待，因为他会开辟一个新的线程执行，不会阻塞主线程。
ok，让我们测试下一个。

在dispatch_async中使用并行队列

看代码。

```
- (void)testConcurrentQueueWithAsync {
    for (int index =0; index<10; index++) {
        dispatch_async(concurrent_queue, ^{
            NSLog(@"index = %d", index);
            NSLog(@"current thread is %@", [NSThread currentThr
ead]);
        });

    }
```

```
    NSLog(@"Running on main Thread");
}
```

打印结果如下。

```
index = 1
index = 5
Running on main Thread
index = 2
index = 6
index = 7
index = 3
index = 0
index = 8
index = 4
index = 9
current thread is <NSThread: 0x7fded2a17e20>{number = 2, name =
 (null)}
current thread is <NSThread: 0x7fded1508e40>{number = 3, name =
 (null)}
current thread is <NSThread: 0x7fded290d6e0>{number = 5, name =
 (null)}
current thread is <NSThread: 0x7fded2a17e60>{number = 4, name =
 (null)}
current thread is <NSThread: 0x7fded1510470>{number = 8, name =
 (null)}
current thread is <NSThread: 0x7fded17050f0>{number = 6, name =
 (null)}
current thread is <NSThread: 0x7fded1704b30>{number = 10, name
= (null)}
current thread is <NSThread: 0x7fded2806fe0>{number = 7, name =
 (null)}
current thread is <NSThread: 0x7fded2900e10>{number = 9, name =
 (null)}
current thread is <NSThread: 0x7fded2b01060>{number = 11, name
= (null)}
```

打印结果的特征如下：

1. 输出的结果是乱序的，说明我们的输出语句是并发的，由多个线程共同执行的。

2. `Running on main Thread` 这句话依然没有被阻塞，直接输出了。

3. 每次打印语句的Thread均不相同。

仔细比对两次打印结果的异同点。提出问题。

1. 串行队列如何保证在异步线程中遵守先进先出原则（从Demo里看，也就是顺序打印我们的结果）？

很简单，它会保证每次dispatch_async开辟线程执行串行队列中的任务时，总是使用同一个异步线程。这也是为什么我们的第一次打印结果中，NSThread总是同一个。

2. 在dispatch_async中放入并行队列并执行的时候，为什么执行顺序总是乱序的?

因为在并行对列中，每执行一次任务的时候，dispatch_async总会为我们开辟一个新的线程（当然，开辟线程的总量是有限制的，你可以试试循环一万次并打印Thread信息）来执行任务，所以不同线程开始结束的时间都不一样，导致了结果是乱序的。

在dispatch_sync中使用串行队列

```
- (void)testSerialQueueWithSync
{
    for (int index =0; index<10; index++) {
        dispatch_sync(serial_queue, ^{
            NSLog(@"index = %d", index);
            NSLog(@"current thread is %@", [NSThread currentThr
ead]);
        });

    }

    NSLog(@"Running on main Thread");

}
```

打印结果如下。

```
index = 0
```

```
current thread is <NSThread: 0x7ff84b507ac0>{number = 1, name =
 main}
index = 1
current thread is <NSThread: 0x7ff84b507ac0>{number = 1, name =
 main}
index = 2
current thread is <NSThread: 0x7ff84b507ac0>{number = 1, name =
 main}
index = 3
current thread is <NSThread: 0x7ff84b507ac0>{number = 1, name =
 main}
index = 4
current thread is <NSThread: 0x7ff84b507ac0>{number = 1, name =
 main}
index = 5
current thread is <NSThread: 0x7ff84b507ac0>{number = 1, name =
 main}
index = 6
current thread is <NSThread: 0x7ff84b507ac0>{number = 1, name =
 main}
index = 7
current thread is <NSThread: 0x7ff84b507ac0>{number = 1, name =
 main}
index = 8
current thread is <NSThread: 0x7ff84b507ac0>{number = 1, name =
 main}
index = 9
current thread is <NSThread: 0x7ff84b507ac0>{number = 1, name =
 main}
Running on main Thread
```

1. dispatch_sync并没有开辟一个新的线程，直接在当前线程中执行代码（即main线程）。所以会阻塞当前线程。
2. Running on main Thread在最后输出。

也就是说，当使用dispatch_sync执行串行队列的任务时，不会开辟新的线程，会直接使用当前线程执行队列中的任务。

在dispatch_sync中使用并行队列

代码如下。

```
- (void)testConcurrentQueueWithSync {
```

```
    for (int index = 0; index<10; index++) {

        dispatch_sync(concurrent_queue, ^{
            NSLog(@"index = %d", index);
            NSLog(@"current thread is %@", [NSThread currentThr
ead]);

        });
    }



    NSLog(@"Running on main Thread");
}
```

打印结果如下。

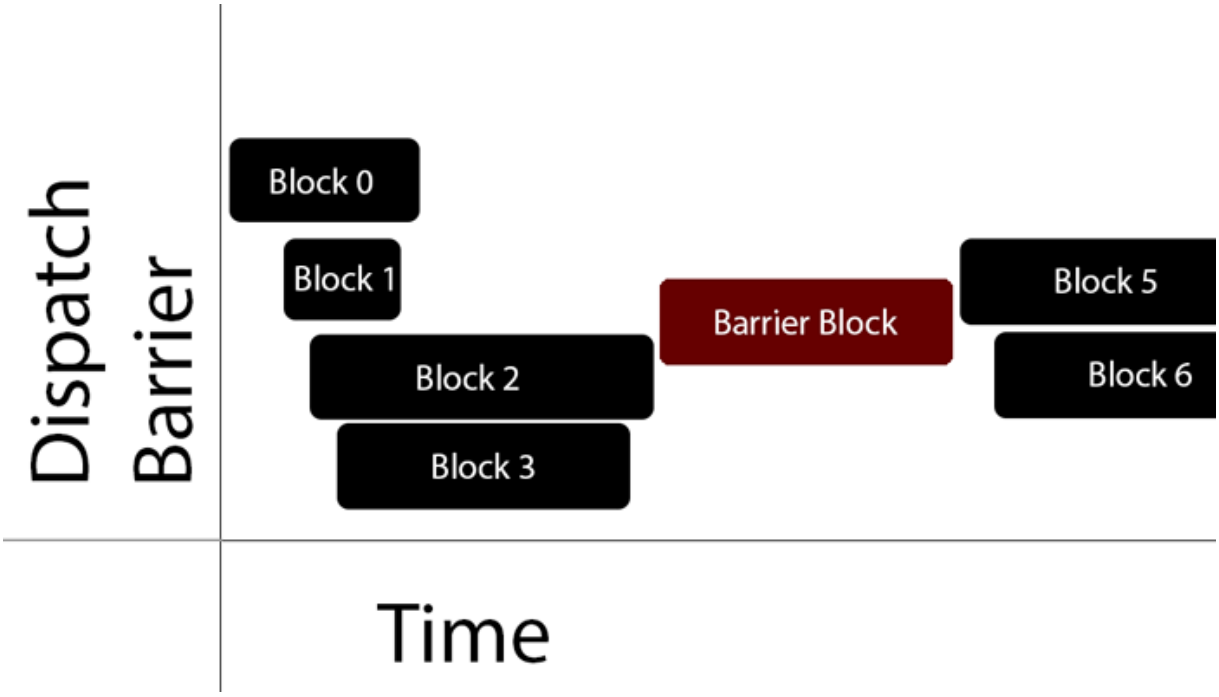
```
index = 0
current thread is <NSThread: 0x7fc1c0e048e0>{number = 1, name =
 main}
index = 1
current thread is <NSThread: 0x7fc1c0e048e0>{number = 1, name =
 main}
index = 2
current thread is <NSThread: 0x7fc1c0e048e0>{number = 1, name =
 main}
index = 3
current thread is <NSThread: 0x7fc1c0e048e0>{number = 1, name =
 main}
index = 4
current thread is <NSThread: 0x7fc1c0e048e0>{number = 1, name =
 main}
index = 5
current thread is <NSThread: 0x7fc1c0e048e0>{number = 1, name =
 main}
index = 6
current thread is <NSThread: 0x7fc1c0e048e0>{number = 1, name =
 main}
index = 7
current thread is <NSThread: 0x7fc1c0e048e0>{number = 1, name =
 main}
index = 8
current thread is <NSThread: 0x7fc1c0e048e0>{number = 1, name =
 main}
index = 9
```

```
current thread is <NSThread: 0x7fc1c0e048e0>{number = 1, name =
 main}
Running on main Thread
```

结果很奇怪，和在串行队列执行的效果一模一样。按我们的思考，并行队列里执行任务不应该是多个线程同时跑么？其实是由于dispatch_sync并不会开辟新的线程执行任务，所以导致了执行并行队列任务的线程总会是一个线程，自然，结果是一样的。

dispatch_barrier

讲完了GCD最基本的用法。我们来看看一个不太常用的GCD。`dispatch_barrier`。
这个barrier我感觉使用霸道总裁来形容比较合适。这里借用raywenderlich上介绍barrier的一张图。



看的有点懵逼?

不要紧。我来解释一下。
首先，在一个并行队列中，有多个线程在执行多个任务，在这个并行队列中，有一个dispatch_barrier任务。这样会有一个什么效果呢?
就是，所有在这个dispatch_barrier之后的任务总会等待barrier之前的所有任务结

束之后，才会执行。那么细心的同学可能会发现这里有个问题，既然所有在barrier之后的任务都会等待在barrier之前的任务结束之后执行，那么barrier本身执行是否会阻塞当前线程?
所以，dispatch_barrier也是有两个方法的。`dispatch_barrier_sync`和`dispatch_barrier_async`.

dispatch_barrier_sync

还是看代码理解的更快一点。代码如下。

```
- (void)testBarrierSyncWithConCurrentQueue {
    for (int index = 0; index<10; index++) {
        dispatch_async(concurrent_queue, ^{
            NSLog(@"index = %d", index);
        });

    }

    for (int j = 0; j<10000; j++) {
        dispatch_barrier_sync(concurrent_queue, ^{
            if (j == 10000-1) {
                NSLog(@"barrier Finished");
                NSLog(@"current thread is %@", [NSThread curren
tThread]);
            }
        });
    }

    NSLog(@"Running on Main Thread");

    for (int index =10; index<20; index++) {
        dispatch_async(concurrent_queue, ^{
            NSLog(@"index = %d", index);
        });
    }

}
```

那么运行之后我们的输出结果如下。

```
 index = 2
 index = 7
 index = 0
```

```
index = 1
index = 3
index = 4
index = 5
index = 8
index = 6
index = 9
barrier Finished
current thread is <NSThread: x7fa81bd08050>{number = 1, name =
main}
Running on Main Thread
index = 10
index = 11
index = 12
index = 13
index = 14
index = 15
index = 16
index = 17
index = 18
index = 19
```

ok，总结一下。

1. `dispatch_barrier_sync` 确实是会在队列中充当一个栅栏的作用，凡是在他之后进入队列的任务，总会在 `dispatch_barrier_sync` 之前的所有任务执行完毕之后才执行。

2. 见名知意， `dispatch_barrier_sync` 是会在主线程执行队列中的任务的，所以， `Running on Main Thread` 这句话会被阻塞，从而在barrier之后执行。

dispatch_barrier_async

那么我们再看看 `dispatch_barrier_async` 执行的效果。

代码如下。

```
- (void)testBarrierAsyncWithConCurrentQueue {
    for (int index = 0; index<10; index++) {
        dispatch_async(concurrent_queue, ^{
            MyLog(@"index = %d", index);
```

```
        });

    }

    for (int j = 0; j<10000000; j++) {
        dispatch_barrier_async(concurrent_queue, ^{
            if (j == 10000000-1) {
                MyLog(@"barrier Finished");
                MyLog(@"current thread is %@", [NSThread curren
tThread]);
            }
        });
    }

    MyLog(@"Running on Main Thread");

    for (int index =10; index<20; index++) {
        dispatch_async(concurrent_queue, ^{
            MyLog(@"index = %d", index);
        });
    }

}
```

其实结果不难猜到。

```
index = 1
index = 0
index = 4
index = 2
index = 3
index = 8
index = 9
index = 7
index = 5
index = 6
Running on Main Thread
barrier Finished
current thread is <NSThread: x7f9b0b6023c0>{number = 2, name =
(null)}
index = 11
index = 13
index = 14
index = 10
index = 15
index = 17
```

```
index = 16
index = 12
index = 18
index = 19
```

`dispatch_barrier_async`会开辟一条新的线程执行其中的任务，所以不会阻塞当前线程。其他的功能和`dispatch_barrier_sync`相同。

几个小问题

1. 为什么我们只举了barrier和并行队列的例子，而没有举barrier和串行队列的例子?

   因为，barrier和串行队列配合是完全没有意义的。barrier的目的是什么？目的是为了在某种情况下，同一个队列中一些并发任务必须在另一些并发任务之后执行，所以需要一个类似于拦截的功能，迫使后执行的任务必须等待。那么，串行队列中的所有任务本身就是按照顺序执行的，那么又有什么必要使用拦截的功能呢?

2. 在global queue中使用barrier没有意义，为什么?

   barrier实现的基本条件是，要写在同一队列中。举个例子，你现在创建了两个并行队列，你在其中一个队列中插入了一个barrier任务，那么你不可能期待他可以在第二个队列中生效，对吧。同样的，每一次使用global queue，系统分配给你的可能是不同的并行队列，你在其中插入一个barrier任务，又有什么意义呢?

该做点什么了

思考一个问题，NSMutableDictionary是否是线程安全的?

做个测试好了。

```
- (void)testMutableDictionaryThreadSafe {
    dispatch_semaphore_t sema = dispatch_semaphore_create(0);
    dispatch_async(concurrent_queue, ^{
        for (int index = 0; index<1000 ; index++) {
            dict[@(index)] = @(index);
```

```
        }
        dispatch_semaphore_signal(sema);
    });

    dispatch_async(concurrent_queue, ^{
        for (int index = 0; index<1000 ; index++) {
            dict[@(index)] = @(0);
        }
        dispatch_semaphore_signal(sema);
    });
    dispatch_semaphore_wait(sema, DISPATCH_TIME_FOREVER);
    dispatch_semaphore_wait(sema, DISPATCH_TIME_FOREVER);

    NSLog(@"dict is %@", dict);

}
```

运行结果是，直接崩了。。
因为`NSMutableDictionary`不是线程安全的，任意一个线程在往字典里写入数据的时候是不允许有其他线程访问的，不管是读或者写都不可以。
所以，现在的任务就是，我们需要使用gcd来实现一个线程安全的NSMutableDictionary。

第一个方案，serial queue + dispatch_async

首先，串行队列能保证每一个读或者写操作都是按顺序执行的，并且会在同一个线程执行任务，dispatch_async又能保证读写操作均能在异步线程执行，所以不会卡当前线程。所以表面上看是没问题的。
关键的问题是，太慢了,因为你每次只会有一个线程读或者写。如果同时有一百个读的请求，那么你的数据必须要按照顺序，一个一个的读出来。所以这个方案行不通。

第二个方案，concurrent queue + dispatch_async

用这个方案意味着，我们可以多线程同时读取字典里的数据。但是我们得确保一个条件。我们读取字典数据的时候，必须保证没有别的线程在写。
所以，确定读取线程安全的条件变成了，如何迫使写的这个操作在同一时刻，只有一个线程在写，并且，没有其他线程读或者写。那么，当然可以用dispatch_barrier_async来实现我们的需求了。
为什么dispatch_barrier_async可以实现我们的需求?
想一下dispatch_barrier_async的几个特性。

同一个队列中，只要插入了一个barrier，那么在他之后的所有任务都必须等他完成了才可以继续。所以，只要我们保证所有写操作都在barrier里完成，那么，我们不可能在一个concurrent queue里同时有多个线程在往字典里写数据。因为假如有多个写的操作，每一个写操作总会等待前一个写操作完成之后才执行。

所以代码如下。

```
typedef void (^ThreadSafeDictionaryBlock)(ThreadSafeDictionary
*dict, NSString *key, id object);

@interface ThreadSafeDictionary ()
{
    dispatch_queue_t concurrentQueue;
}
@end

@implementation ThreadSafeDictionary

- (id)init
{
    if (self = [super init]) {
        concurrentQueue = dispatch_queue_create("www.reviewcode
.cn", DISPATCH_QUEUE_CONCURRENT);
    }

    return self;
}

- (void)objectForKey:(id)aKey block:(ThreadSafeDictionaryBlock)
block
{
    id key = [aKey copy];
    __weak __typeof__(self) weakSelf = self;
    dispatch_async(concurrentQueue, ^{
        ThreadSafeDictionary *strongSelf = weakSelf;
        if (!strongSelf)
            return;
        id object = [self objectForKey:key];
        block(self, key, object);
    });
}

- (void)setObject:(id)object forKey:(NSString *)key block:(Thre
adSafeDictionaryBlock)block
{
```

```
    if (!key || !object)
        return;

    NSString *akey = [key copy];
    __weak ThreadSafeDictionary *weakSelf = self;

    dispatch_barrier_async(concurrentQueue, ^{
        __strong typeof(weakSelf)strongSelf = weakSelf;;
        if (!strongSelf)
            return;
        [self setObject:object forKey:akey];

        if (block) {
            block(strongSelf, akey, object);
        }
    });
}
@end
```

这里可能会有人存疑，外部用`__weak`关键在声明weakSelf这个没什么问题，防止retain cycle。但里面为什么又用`__strong`声明了一个strongSelf?

这是因为，你享受了weak的好处，同时也要承担风险，由于不持有这个weakSelf，所以你无法保证在代码运行过程中，self不会被释放。

那可能又有人问了，你在外部用weak声明的原因是防止retain cycle，结果又在里面声明了一个strong，那不相当于做无用功了么。但问题是，在block内部声明的strongSelf是局部变量，他的生命周期只是在block内而已，当block执行完它自动就被销毁了。所以不会造成retain cycle。

还有一个问题，如果在第一句`__strong typeof(weakSelf)strongSelf = weakSelf;`的时候weakSelf已经被销毁了怎么办?

你没看我下面有个`if(! strongSelf)`么？不就是应对这种情况的。

dispatch_semaphore

信号量。
信号量的用法相当简单。
一共有三个方法。

```
dispatch_semaphore_create => 创建一个信号量
```

```
dispatch_semaphore_signal => 发送一个信号
dispatch_semaphore_wait => 等待信号
```

dispatch_semaphore的使用场景是处理并发控制。
如果感觉不是很明白的话，想想`NSOperationQueue`的一个属性，`maxConcurrentOperationCount`.这个属性的意思就是设定NSOperationQueue里的NSOperation同时运行的最大数量。

我们的信号量也可以实现同样的功能。
首先，创建一个信号量。
`dispatch_semaphore_t semaphore = dispatch_semaphore_create(0);`
创建方法里会传入一个long型的参数，这个东西你可以想象是一个库存。有了库存才可以出货。
`dispatch_semaphore_wait`，就是每运行一次，会先清一个库存，如果库存为0，那么根据传入的等待时间，决定等待增加库存的时间，如果设置为DISPATCHTIMEFOREVER,那么意思就是永久等待增加库存，否则就永远不往下面走。
`dispatch_semaphore_signal`，就是每运行一次，增加一个库存。
那么下面的代码运行起来会是怎样的结果呢?

```
dispatch_semaphore_t semaphore = dispatch_semaphore_create(0);
    dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
    NSLog(@"等待semaphore");
```

结果就是，`等待semaphore`这句话永远不会输出。原因有两个。
1. 你初始化信号量的时候，并没有库存，也就是你传入的值是0.
2. 你传入等待增加库存的时间是`DISPATCH_TIME_FOREVER`，也就是说，除非有地方运行了`dispatch_semaphore_signal`增加了库存，否则我永远等待下去。
基于上述的两个原因，导致了程序不往下走了。

所以，其实这样的特性可以帮助我们完成一个功能，测试异步网络请求是否成功。
比如，在以前写Unit Test测试网络请求的时候，我们常常这么写。

```
- (void)downloadImageURLWithString:(NSString *)URLString
{
    // 1
    dispatch_semaphore_t semaphore = dispatch_semaphore_create(
0);
```

```
    NSURL *url = [NSURL URLWithString:URLString];
    __unused Photo *photo = [[Photo alloc]
                             initwithURL:url
                             withCompletionBlock:^(UIImage *ima
ge, NSError *error) {
                                 if (error) {
                                     XCTFail(@"%@ failed. %@",
URLString, error);
                                 }

                                 // 2
                                 dispatch_semaphore_signal(sema
phore);
                             }];

    // 3
    dispatch_time_t timeoutTime = dispatch_time(DISPATCH_TIME_N
OW, 5);
    if (dispatch_semaphore_wait(semaphore, timeoutTime)) {
        XCTFail(@"%@ timed out", URLString);
    }
}
```

解释一下。
1. 我们创建了一个信号量，并且传入的参数为0，可以想象现在是0库存.
2. 开启了一个异步线程下载网络数据，在回调的block中判断error是否为nil。如果为nil，直接使用`XCTFail`，报错。如果成功了。那么增加一个库存，告诉信号量，好了，现在有库存了。你可以继续运货了。
3. 设置一个超时时间，5秒钟，这个是干嘛的呢？这个是告诉信号量，我等待库存增加的时间只有5秒钟，如果超过了五秒钟，我就报错。怎么判断是否超过5秒钟呢？`dispatch_semaphore_wait(semaphore, timeoutTime)`，如果超过了5秒钟，那么这个方法会返回一个非0的长整型。

这样，我们就可以给一个网络接口写单元测试了。

上文我们提到过，信号量的主要功能还是控制并发量，可以实现类似于NSOperationQueue的功能。那么我现在尝试实现一个。

```
@implementation CustomOperationQueue

- (id)initWithConcurrentCount:(int)count
```

```
{
    self = [super init];
    if (self)
    {
        if (count < 1) count = 5;
        semaphore = dispatch_semaphore_create(count);
        queue = Dispatch_queue_create("com.zangqilong.www", DIS
PATCH_QUEUE_CONCURRENT);
    }
    return self;
}

- (id)init
{
    return [self initWithConcurrentCount:5];
}

- (void)addTask:(CallBackBlock)block
{
    dispatch_async(queue, ^{
        dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVE
R);
        dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE
_PRIORITY_DEFAULT, 0), 0), ^{
            block();
            dispatch_semaphore_signal(semaphore);
        });
    });
}

@end
```

我们把注意力放在`- (void)addTask:(CallBackBlock)block`这个方法里。
1. 我们创建了一个初始库存是5的一个信号量
2. 在addTask方法里，由于我们的初始库存是5，所以第一次添加了一个任务之后，`dispatch_semaphore_wait`会直接放行，并减少一个库存，所以现在库存是4，然后，每当我们完成一个任务，也就意味着，我们可以把库存还回去了。所以就会调用`dispatch_semaphore_signal`去增加一个库存。
3. 那么，如果我们的每一个任务耗时都相当长，所以我们是一直消耗库存但是没有还回库存，所以当添加到第6个任务的时候，这个时候由于库存已经为0，所以wait方法会一直等待，不执行第六个任务，直到有前面的任务完成，库存大于0，这时候才会执行第六个任务。

这样，我们就完成了并发量的控制。

dispatch_group

直接看代码

```
dispatch_group_t serviceGroup = dispatch_group_create();

// 开始第一个请求
// 进入组
dispatch_group_enter(serviceGroup);
[self.configService startWithCompletion:^(ConfigResponse *resul
ts, NSError* error){
    configError = error;
    // 离开组
    dispatch_group_leave(serviceGroup);
}];

// 开始第二个请求
// 先进入组
dispatch_group_enter(serviceGroup);
[self.preferenceService startWithCompletion:^(PreferenceRespons
e *results, NSError* error){
    // 离开组
    preferenceError = error;
    dispatch_group_leave(serviceGroup);
}];

// 当小组里的任务都清空以后，通知主线程完成了所有任务
dispatch_group_notify(serviceGroup,dispatch_get_main_queue(),^{
    // Assess any errors
    NSError *overallError = nil;
    if (configError || preferenceError)
    {
        // Either make a new error or assign one of them to the
 overall error
        overallError = configError ?: preferenceError;
    }
    // Now call the final completion block
    completion(overallError);
});
```

这个`dispatch_group`最常见的功能就是，一个页面有多个异步网络请求，如何监测所有任务都完成了。

这个方法相信大家都会用。
那么需要注意的是。这里的网络请求回调只有一个block，所以无论请求出错还是成功了，只需要写一次`dispatch_group_leave`,但是如果你的回调block有两个，分成功的回调和错误的回调，那么你必须在两个block里都写`disptach_group_leave`,因为你不知道你的请求到底会走成功的block还是失败的block，如果少写了，一旦某个请求失败了，那么你的notify方法就永远也不会执行了。

dispatch_source

dispatch source是一种用于处理事件的数据类型，这些被处理的事件为操作系统中的底层级别。Grand Central Dispatch（GCD）支持如下的dispatch sources类型：

1. Timer dispatch sources：定时器类型，能够产生周期性的通知事件；
2. Signal dispatch sources：信号类型，当UNIX信号到底时，能够通知应用程序；
3. Descriptor sources：文件描述符类型，处理UNIX的文件或socket描述符，如：
    - 数据可读
    - 数据可写
    - 文件被删除、修改或移动
    - 文件的元信息被修改
4. Process dispatch sources：进程类型，能够通知一些与进程相关的事件类型，如：
    - 当进程退出
    - 当进程调用了fork或exec
    - 当一个信号传递给了进程
5. Mach port dispatch sources：端口匹配类型，能够通知一些端口事件的类型；
6. Custom dispatch sources：自定义类型，可以自定义一些事件类型。