

GCD常用和不常用API说明和Demo演示，让你轻松不费脑力的理解GCD的应用场景和操作姿势 <http://dongxuwei.com>

🔗 14 commits

🌿 1 branch

📦 0 packages

📦 0 releases

👤 2 contributors

Branch: master ▼

New pull request

Create new file

Upload files

Find file

Clone or download

👤 MrTung update	Latest commit caa8e4b on Jan 10, 2025	
📁 gcdtest.xcodeproj	demo优化	
📁 gcdtest	demo优化	
📄 README.md	update	12 months ago

📖 README.md

前言

各路大神对GCD的原理解析和使用方法网上到处都是,可以轻松搜索到。那为什么笔者还要自己动手写一篇所谓的"葵花宝典"呢？其实本篇文章主要是普及了一些基础知识概念。例如队列、线程、异步同步的区别,搞懂这些才是弄明白GCD的入门。笔者搜集了GCD的绝大部分api，包括不常用的、冷门的。这里没有高大上的实现原理，没有难懂的底层实现。以最浅显的简单的文字说明配上demo和代码实例最后结合运行log，让你轻松不费脑力的理解GCD的应用场景和操作姿势。



基础概念

关于GCD：

- (1)是基于c语言的底层api
- (2)用block定义任务，使用起来非常灵活便捷
- (3)GCD会自动利用更多的CPU内核（比如双核、四核）
- (4)GCD会自动管理线程的生命周期（创建线程、调度任务、销毁线程）
- (5)程序员只需要告诉GCD想要执行什么任务，不需要编写任何线程管理代码

关于进程：

正在进行中的程序被称为进程，负责程序运行的内存分配;每一个进程都有自己独立的虚拟内存空间；

关于线程：

线程是进程中一个独立的执行路径(控制单元);一个进程中至少包含一条线程，即主线程。

关于队列：


队列用来存放任务，一种符合 FIFO（先进先出）原则的数据结构，线程的创建和回收不需要程序员操作，由队列负责。

- 串行队列：队列中的任务只会顺序执行，一个任务执行完毕后，再执行下一个任务
- 并发队列：队列中的多个任务并发（同时）执行,而且无法确定任务的执行顺序
- 全局队列：是系统开发的，直接拿过来用就可以；与并行队列类似，但调试时，无法确认操作所在队列

主队列： 每一个应用程序对应唯一一个主队列，是gcd中自带的一种特殊的串行队列，直接获取即可。放在主队列中的任务，都会在主线程中执行。在多线程开发中，使用主队列更新UI。

关于操作：

dispatch_async 异步操作，在新的线程中执行任务，具备开启新线程的能力(不是百分百开启新线程，会取决于任务所在队列类型)，会立即返回，无法确定任务的执行顺序；
dispatch_sync 同步操作，在当前线程中执行任务，不具备开启新线程的能力，会依次顺序执行；

- 图例：



使用姿势

分为两步：

第一步： 创建一个队列；

第二步： 将任务放到队列中；

三个关键点：

第一点： 任务内容；

第二点： 队列类别；

第三点： 操作(追加)姿势；

队列和任务

1. 获取队列：

GCD中大体可以分为三种队列：

- 串行队列: `dispatch_queue_t queue = dispatch_queue_create("com.test.testQueue", DISPATCH_QUEUE_SERIAL);`
- 并发队列:
 - 一般并发队列 `dispatch_queue_t queue = dispatch_queue_create("com.test.testQueue", DISPATCH_QUEUE_CONCURRENT);`
 - 全局并发队列可以作为普通并发队列来使用 `dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);`
- 主队列: `dispatch_queue_t queue = dispatch_get_main_queue();`

2. 操作(追加)方式：

- 同步执行任务创建方法

```
dispatch_sync(queue, ^{
    NSLog(@"我是任务");
});
```

- 异步执行任务创建方法

```
dispatch_async(queue, ^{
    NSLog(@"我是任务");
});
```

```
});
```

3. 队列 + 任务：

3.1 队列 + 任务的六种组合

看到这里你不难发现,GCD 提供了同步执行任务的创建方法 `dispatch_sync` 和异步执行任务创建方法 `dispatch_async` ,配合三种队列形式（串行队列、并发队列、主队列), 那么就会存在六种不同的多线程使用方法，如下：

- 同步执行 & 并发队列：不新建线程，在当前线程中顺序执行
- 异步执行 & 并发队列：新建多个新线程，线程会复用，无序执行
- 同步执行 & 串行队列：在当前线程中顺序执行
- 异步执行 & 串行队列：新建一条新的线程，在该线程中顺序执行
- 异步执行 & 主队列：不新建线程，在主线程中顺序执行
- 同步执行 & 主队列（在主线程中会crash): 主线程中会产生死锁



3.2 各种组合的使用方法

同步执行 & 并发队列：

```
-(void)syncAndConcurrentqueue{

    NSLog(@"-----start-----当前线程---%@",[NSThread currentThread]);

    //下面提供两种并发队列的获取方式，其运行结果无差别，所以归为了一类，你可以自由选择
    dispatch_queue_t queue = dispatch_queue_create("com.test.testQueue", DISPATCH_QUEUE_CONCURRENT);

    // 全局并发队列
    //    dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    // 第一个任务
    dispatch_sync(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"-----执行第一个任务---当前线程%@",[NSThread currentThread]);
    });

    // 第二个任务
    dispatch_sync(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"-----执行第二个任务---当前线程%@",[NSThread currentThread]);
    });

    // 第三个任务
    dispatch_sync(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"-----执行第三个任务---当前线程%@",[NSThread currentThread]);
    });

    NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);
}
```

- 输出结果：



- 总结： 只会在当前线程中依次执行任务， 不会开启新线程， 执行完一个任务， 再执行下一个任务,按照1>2>3顺序执行,遵循FIFO原则。

异步执行 & 并发队列：

```
-(void)asyncAndConcurrentqueue{

    NSLog(@"-----start-----当前线程---%@",[NSThread currentThread]);

    //下面提供两种并发队列的获取方式， 其运行结果无差别， 所以归为了一类， 你可以自由选择

    //dispatch_queue_t queue = dispatch_queue_create("com.test.testQueue", DISPATCH_QUEUE_CONCURRENT);

    // 全局并发队列
    dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    // 第一个任务
    dispatch_async(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"-----执行第一个任务---当前线程%@",[NSThread currentThread]);
    });

    // 第二个任务
    dispatch_async(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"-----执行第二个任务---当前线程%@",[NSThread currentThread]);
    });

    // 第三个任务
    dispatch_async(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"-----执行第三个任务---当前线程%@",[NSThread currentThread]);
    });

    NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);

}
```

- 输出结果：



- 总结： 从log中可以发现,系统另外开启了3个线程， 并且任务是同时执行的,并不是按照1>2>3顺序执行。所以异步+并发队列具备开启新线程的能力,且并发队列可开启多个线程， 同时执行多个任务。

同步执行 & 串行队列：

```
-(void)syncAndSerialqueue{

    NSLog(@"-----start-----当前线程---%@",[NSThread currentThread]);
```

```
dispatch_queue_t queue = dispatch_queue_create("com.test.testQueue", DISPATCH_QUEUE_SERIAL);

// 第一个任务
dispatch_sync(queue, ^{

    //这里线程暂停2秒,模拟一般的任务的耗时操作
    [NSThread sleepForTimeInterval:2];

    NSLog(@"----执行第一个任务---当前线程%@",[NSThread currentThread]);
});

// 第二个任务
dispatch_sync(queue, ^{

    //这里线程暂停2秒,模拟一般的任务的耗时操作
    [NSThread sleepForTimeInterval:2];

    NSLog(@"----执行第二个任务---当前线程%@",[NSThread currentThread]);
});

// 第三个任务
dispatch_sync(queue, ^{

    //这里线程暂停2秒,模拟一般的任务的耗时操作
    [NSThread sleepForTimeInterval:2];

    NSLog(@"----执行第三个任务---当前线程%@",[NSThread currentThread]);
});

NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);

}
```

- 输出结果：



- 总结： 只会在当前线程中依次执行任务， 不会开启新线程， 执行完一个任务， 再执行下一个任务,按照1>2>3顺序执
循FIFO原则。

异步执行 & 串行队列：

```
-(void)asyncAndSerialqueue{
    NSLog(@"-----start-----当前线程---%@",[NSThread currentThread]);

    dispatch_queue_t queue = dispatch_queue_create("com.test.testQueue", DISPATCH_QUEUE_SERIAL);

    // 第一个任务
    dispatch_async(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"----执行第一个任务---当前线程%@",[NSThread currentThread]);
    });

    // 第二个任务
    dispatch_async(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];
```



```
        NSLog(@"----执行第二个任务---当前线程%@",[NSThread currentThread]);
    });


    // 第三个任务
    dispatch_async(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"----执行第三个任务---当前线程%@",[NSThread currentThread]);
    });

    NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);

}
```

- 输出结果：
- 总结:开启了一条新线程，异步执行具备开启新线程的能力,因为是串行队列所以只开启一个线程，在该线程中执行完务，再执行下一个任务,按照1>2>3顺序执行，遵循FIFO原则。

异步执行 & 主队列：

```
-(void)asyncAndMainqueue{
    NSLog(@"-----start-----当前线程---%@",[NSThread currentThread]);

    //获取主队列
    dispatch_queue_t queue = dispatch_get_main_queue();

    // 第一个任务
    dispatch_async(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"----执行第一个任务---当前线程%@",[NSThread currentThread]);
    });

    // 第二个任务
    dispatch_async(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"----执行第二个任务---当前线程%@",[NSThread currentThread]);
    });


    // 第三个任务
    dispatch_async(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"----执行第三个任务---当前线程%@",[NSThread currentThread]);
    });

    NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);

}
```

- 输出结果：

- 总结：所有任务都是在当前线程（主线程）中执行的，并没有开启新的线程（虽然异步执行具备开启线程的能力，但主线程是主队列，所以所有任务都在主线程中），在主线程中执行完一个任务，再执行下一个任务,按照1>2>3顺序执行，遵循顺序规则。

同步执行 & 主队列（在主线程中会crash）：

```
-(void)syncAndMainqueue{
    NSLog(@"-----start-----当前线程---%@",[NSThread currentThread]);

    //获取主队列
    dispatch_queue_t queue = dispatch_get_main_queue();

    // 第一个任务
    dispatch_sync(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"----执行第一个任务---当前线程%@",[NSThread currentThread]);
    });

    // 第二个任务
    dispatch_sync(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"----执行第二个任务---当前线程%@",[NSThread currentThread]);
    });

    // 第三个任务
    dispatch_sync(queue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"----执行第三个任务---当前线程%@",[NSThread currentThread]);
    });

    NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);
}
```

//下面的例子类似：在同一个同步串行队列中，再使用该队列同步执行任务也是会发生死锁。

```
-(void)syncAndMainqueue1{

    dispatch_queue_t queue1 = dispatch_queue_create("com.test.testQueue", DISPATCH_QUEUE_SERIAL);

    dispatch_sync(queue1, ^{

        [NSThread sleepForTimeInterval:2];

        NSLog(@"----11111-----当前线程%@",[NSThread currentThread]);//到这里就死锁了

        dispatch_sync(queue1, ^{


            [NSThread sleepForTimeInterval:2];

            NSLog(@"----22222---当前线程%@",[NSThread currentThread]);
        });

        NSLog(@"----333333-----当前线程%@",[NSThread currentThread]);

    });

    NSLog(@"----44444-----当前线程%@",[NSThread currentThread]);
}
```

- }
- 输出结果：

 - 总结:直接crash。这是因为发生了死锁，在gcd中，禁止在主队列(串行队列)中再以同步操作执行主队列任务。同理一个同步串行队列中，再使用该队列同步执行任务也是会发生死锁。

同步执行 & 主队列（在其它线程中）：

```

-(void)othersyncAndMainqueue{

    NSLog(@"-----start-----当前线程---%@",[NSThread currentThread]);

    dispatch_queue_t queue = dispatch_queue_create("com.test.testQueue", DISPATCH_QUEUE_SERIAL);

    // 第一个任务
    dispatch_async(queue, ^{

        NSLog(@"----执行任务---%@",[NSThread currentThread]);

        //获取主队列
        dispatch_queue_t queue = dispatch_get_main_queue();


        // 第一个任务
        dispatch_sync(queue, ^{

            //这里线程暂停2秒,模拟一般的任务的耗时操作
            [NSThread sleepForTimeInterval:2];

            NSLog(@"----执行第一个任务---当前线程%@",[NSThread currentThread]);
        });
    });
    NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);

}

```

- 输出结果：

- 总结：所有任务都是在主线程（非当前线程）中执行的，没有开启新的线程（所有放在主队列中的任务，都会放到主线程中执行）。在主线程中执行完一个任务，再执行下一个任务,按照1>2>3顺序执行，遵循FIFO原则。

GCD常用API及其使用方法

1. Dispatch Queue：

```

//各种队列的获取方法
-(void)getQueue{

    //主队列的获取方法:主队列是串行队列，主队列中的任务都将在主线程中执行
    dispatch_queue_t mainqueue = dispatch_get_main_queue();

    //串行队列的创建方法:第一个参数表示队列的唯一标识,第二个参数用来识别是串行队列还是并发队列（若为NULL时，默认是DISPATCH_QUEUE_SERIAL）
    dispatch_queue_t seriaQueue = dispatch_queue_create("com.test.testQueue", DISPATCH_QUEUE_SERIAL);

    //并发队列的创建方法:第一个参数表示队列的唯一标识,第二个参数用来识别是串行队列还是并发队列（若为NULL时，默认是DISPATCH_QUEUE_SERIAL）
    dispatch_queue_t concurrentQueue = dispatch_queue_create("com.test.testQueue", DISPATCH_QUEUE_CONCURRENT);

    //全局并发队列的获取方法:第一个参数表示队列优先级,我们选择默认的好了,第二个参数flags作为保留字段备用,一般都直接填0
}

```



```
dispatch_queue_t globalQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
}
```

2. dispatch_queue_create :

```
//自定义队列的创建方法
-(void)queue_create{
    //串行队列的创建方法:第一个参数表示队列的唯一标识,第二个参数用来识别是串行队列还是并发队列（若为NULL时，默认是DISPATCH_QUEUE_SERIAL）
    dispatch_queue_t serialQueue = dispatch_queue_create("com.test.testQueue", DISPATCH_QUEUE_SERIAL);

    //并发队列的创建方法:第一个参数表示队列的唯一标识,第二个参数用来识别是串行队列还是并发队列（若为NULL时，默认是DISPATCH_QUEUE_SERIAL）
    dispatch_queue_t concurrentQueue = dispatch_queue_create("com.test.testQueue", DISPATCH_QUEUE_CONCURRENT);
}
```

3. dispatch_set_target_queue :

- dispatch_set_target_queue 可以更改 Dispatch Queue 的执行优先级 dispatch_queue_create 函数生成的 Dispatch Queue 不管是 Serial Dispatch Queue 还是 Concurrent Dispatch Queue ,执行的优先级都与默认优先级的 Global Dispatch queue 相同,如果需要变更生成的 Dispatch Queue 的执行优先级则需要使用 dispatch_set_target_queue 函数。

```
//使用dispatch_set_target_queue更改Dispatch Queue的执行优先级
-(void)testTargetQueue1{

    NSLog(@"-----start-----当前线程---%@",[NSThread currentThread]);

    //串行队列的创建方法:第一个参数表示队列的唯一标识,第二个参数用来识别是串行队列还是并发队列（若为NULL时，默认是DISPATCH_QUEUE_SERIAL）
    dispatch_queue_t serialQueue = dispatch_queue_create("com.test.testQueue", NULL);

    //指定一个任务
    dispatch_async(serialQueue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"-----执行第一个任务---当前线程%@",[NSThread currentThread]);
    });

    //全局并发队列的获取方法:第一个参数表示队列优先级,我们选择默认的好了,第二个参数flags作为保留字段备用,一般都直接填0
    dispatch_queue_t globalQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    //指定一个任务
    dispatch_async(globalQueue, ^{

        //这里线程暂停2秒,模拟一般的任务的耗时操作
        [NSThread sleepForTimeInterval:2];

        NSLog(@"-----执行第二个任务---当前线程%@",[NSThread currentThread]);
    });

    //第一个参数为要设置优先级的queue,第二个参数是参照物，即将第一个queue的优先级和第二个queue的优先级设置一样。
    //第一个参数如果是系统提供的【主队列】或【全局队列】 ,则不知道会出现什么情况，因此最好不要设置第一参数为系统提供的队列
    dispatch_set_target_queue(serialQueue,globalQueue);

    NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);

}
```

//dispatch_set_target_queue除了能用来设置队列的优先级之外，还能够创建队列的层次体系，当我们想让不同队列中的任务同步的时候，我们可以创建一个串行队列，然后将这些队列的target指向新创建的队列即可。

```
- (void)testTargetQueue2 {

    NSLog(@"-----start-----当前线程---%@",[NSThread currentThread]);

    dispatch_queue_t targetQueue = dispatch_queue_create("com.test.target_queue", DISPATCH_QUEUE_SERIAL);

    dispatch_queue_t queue1 = dispatch_queue_create("com.test.queue1", DISPATCH_QUEUE_SERIAL);

    dispatch_queue_t queue2 = dispatch_queue_create("com.test.queue2", DISPATCH_QUEUE_CONCURRENT);

    dispatch_set_target_queue(queue1, targetQueue);

    dispatch_set_target_queue(queue2, targetQueue);

    //指定一个异步任务
    dispatch_async(queue1, ^{
        NSLog(@"-----执行第一个任务---当前线程%@",[NSThread currentThread]);
        [NSThread sleepForTimeInterval:2];
    });

    //指定一个异步任务
    dispatch_async(queue2, ^{
        NSLog(@"-----执行第二个任务---当前线程%@",[NSThread currentThread]);
        [NSThread sleepForTimeInterval:2];
    });

    //指定一个异步任务
    dispatch_async(queue2, ^{
        NSLog(@"-----执行第三个任务---当前线程%@",[NSThread currentThread]);
        [NSThread sleepForTimeInterval:2];
    });

    NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);

}
```

4. dispatch_after:

//延时执行,需要注意的是：dispatch_after函数并不是在指定时间之后才开始执行处理，而是在指定时间之后将任务追加到主队列中。来说，这个时间并不是绝对准确的，但想要大致延迟执行任务，dispatch_after函数是很有效的。

```
-(void)dispatch_after{
    NSLog(@"-----start-----当前线程---%@",[NSThread currentThread]);

    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(2.0 * NSEC_PER_SEC)), dispatch_get_main_queue())
        // 2秒后异步追加任务代码到主队列等待执行
        NSLog(@"-----执行第一个任务---当前线程%@",[NSThread currentThread]);
    });

    NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);

}
```

5. dispatch_once:

//只执行一次,通常在创建单例时使用，多线程环境下也能保证线程安全

```
-(void)dispatch_once_1{

    NSLog(@"-----start-----当前线程---%@",[NSThread currentThread]);

    static dispatch_once_t onceToken;
```

```
dispatch_once(&onceToken, ^{
    NSLog(@"----只执行一次的任务---当前线程%@",[NSThread currentThread]);
});

NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);

}
```

6. dispatch_apply:

```
//快速遍历方法，可以替代for循环的函数。dispatch_apply按照指定的次数将指定的任务追加到指定的队列中，并等待全部队列执行结束
//会创建新的线程，并发执行
-(void)dispatch_apply{

    NSLog(@"----start-----当前线程---%@",[NSThread currentThread]);

    dispatch_queue_t globalQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    dispatch_apply(100, globalQueue, ^(size_t index) {
        NSLog(@"执行第%d次的任务---%@",index, [NSThread currentThread]);
    });

    NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);

}
```

7. dispatch_group:

```
//队列组:当我们遇到需要异步下载3张图片，都下载完之后再拼接成一个整图的时候，就需要用到gcd队列组。
-(void)dispatch_group{

    NSLog(@"----start-----当前线程---%@",[NSThread currentThread]);

    dispatch_group_t group =  dispatch_group_create();

    dispatch_group_async(group, dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        // 第一个任务
        [NSThread sleepForTimeInterval:2];

        NSLog(@"----执行第一个任务---当前线程%@",[NSThread currentThread]);

    });

    dispatch_group_async(group, dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        // 第二个任务
        [NSThread sleepForTimeInterval:2];

        NSLog(@"----执行第二个任务---当前线程%@",[NSThread currentThread]);
    });

    dispatch_group_async(group, dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        // 第三个任务
        [NSThread sleepForTimeInterval:2];

        NSLog(@"----执行第三个任务---当前线程%@",[NSThread currentThread]);
    });

    dispatch_group_notify(group, dispatch_get_main_queue(), ^{
        [NSThread sleepForTimeInterval:2];
    });

}
```

```

        NSLog(@"----执行最后的汇总任务---当前线程%@",[NSThread currentThread]);
    });

//若想执行完上面的任务再走下面这行代码可以加上下面这句代码

// 等待上面的任务全部完成后，往下继续执行（会阻塞当前线程）
//    dispatch_group_wait(group, DISPATCH_TIME_FOREVER);

    NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);
}

//dispatch_group_enter 标志着一个任务追加到 group，执行一次，相当于 group 中未执行完毕任务数+1
//dispatch_group_leave 标志着一个任务离开了 group，执行一次，相当于 group 中未执行完毕任务数-1。
//当 group 中未执行完毕任务数为0的时候，才会使dispatch_group_wait解除阻塞，以及执行追加到dispatch_group_notify中的
-(void)dispatch_group_1{

    NSLog(@"----start-----当前线程---%@",[NSThread currentThread]);

    dispatch_group_t group = dispatch_group_create();

    dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    dispatch_group_enter(group);
    dispatch_async(queue, ^{
        // 第一个任务
        [NSThread sleepForTimeInterval:2];

        NSLog(@"----执行第一个任务---当前线程%@",[NSThread currentThread]);

        dispatch_group_leave(group);
    });

    dispatch_group_enter(group);
    dispatch_async(queue, ^{
        // 第二个任务
        [NSThread sleepForTimeInterval:2];

        NSLog(@"----执行第二个任务---当前线程%@",[NSThread currentThread]);

        dispatch_group_leave(group);
    });

    dispatch_group_enter(group);
    dispatch_async(queue, ^{
        // 第三个任务
        [NSThread sleepForTimeInterval:2];

        NSLog(@"----执行第三个任务---当前线程%@",[NSThread currentThread]);

        dispatch_group_leave(group);
    });

    dispatch_group_notify(group, dispatch_get_main_queue(), ^{

        [NSThread sleepForTimeInterval:2];
        NSLog(@"----执行最后的汇总任务---当前线程%@",[NSThread currentThread]);
    });

    NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);

}

```

8. dispatch_semaphore:

```

//信号量
//总结:信号量设置的是2，在当前场景下，同一时间内执行的线程就不会超过2，先执行2个线程，等执行完一个，下一个会开始执行。
-(void)dispatch_semaphore{

    NSLog(@"----start-----当前线程---%@",[NSThread currentThread]);

    dispatch_semaphore_t semaphore = dispatch_semaphore_create(2);

    dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    //任务1
    dispatch_async(queue, ^{

        dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);

        NSLog(@"----开始执行第一个任务---当前线程%@",[NSThread currentThread]);

        [NSThread sleepForTimeInterval:2];

        NSLog(@"----结束执行第一个任务---当前线程%@",[NSThread currentThread]);

        dispatch_semaphore_signal(semaphore);
    });

    //任务2
    dispatch_async(queue, ^{

        dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);

        NSLog(@"----开始执行第二个任务---当前线程%@",[NSThread currentThread]);

        [NSThread sleepForTimeInterval:1];

        NSLog(@"----结束执行第二个任务---当前线程%@",[NSThread currentThread]);

        dispatch_semaphore_signal(semaphore);
    });

    //任务3
    dispatch_async(queue, ^{

        dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);

        NSLog(@"----开始执行第三个任务---当前线程%@",[NSThread currentThread]);

        [NSThread sleepForTimeInterval:2];

        NSLog(@"----结束执行第三个任务---当前线程%@",[NSThread currentThread]);

        dispatch_semaphore_signal(semaphore);
    });

    NSLog(@"----end-----当前线程---%@",[NSThread currentThread]);

}

```

9. Dispatch I/O:

```

//以下为苹果中使用Dispatch I/O和Dispatch Data的例子
pipe_q = dispatch_queue_create("PipeQ",NULL);
pipe_channel = dispatch_io_create(DISPATCH_IO_STREAM,fd,pipe_q,^(int err){
    close(fd);

```



```
});

*out_fd = fdpair[i];

dispatch_io_set_low_water(pipe_channel,SIZE_MAX);

dispatch_io_read(pipe_channel,0,SIZE_MAX,pipe_q, ^(bool done,dispatch_data_t pipe_data,int err){
    if(err == 0)
    {
        size_t len = dispatch_data_get_size(pipe_data);
        if(len > 0)
        {
            const char *bytes = NULL;
            char *encoded;

            dispatch_data_t md = dispatch_data_create_map(pipe_data,(const void **)&bytes,&len);
            asl_set((aslmsg)merged_msg,ASL_KEY_AUX_DATA,encoded);
            free(encoded);
            _asl_send_message(NULL,merged_msg,-1,NULL);
            asl_msg_release(merged_msg);
            dispatch_release(md);
        }
    }

    if(done)
    {
        dispatch_semaphore_signal(sem);
        dispatch_release(pipe_channel);
        dispatch_release(pipe_q);
    }
});
```

10. dispatch_barrier_async:

```
//隔断方法：当前面的写入操作全部完成之后，再执行后面的读取任务。当然也可以用Dispatch Group和dispatch_set_target_queue
是比较而言，dispatch_barrier_async会更加顺滑
-(void)dispatch_barrier_async{

    NSLog(@"-----start-----当前线程---%@",[NSThread currentThread]);

    dispatch_queue_t queue = dispatch_queue_create("com.test.testQueue", DISPATCH_QUEUE_CONCURRENT);

    dispatch_async(queue, ^{
        // 第一个写入任务
        [NSThread sleepForTimeInterval:3];

        NSLog(@"-----执行第一个写入任务---当前线程%@",[NSThread currentThread]);

    });
    dispatch_async(queue, ^{
        // 第二个写入任务
        [NSThread sleepForTimeInterval:1];

        NSLog(@"-----执行第二个任务---当前线程%@",[NSThread currentThread]);

    });

    dispatch_barrier_async(queue, ^{
        // 等待处理
        [NSThread sleepForTimeInterval:2];

        NSLog(@"-----等待前面的任务完成---当前线程%@",[NSThread currentThread]);

    });
}
```

```
dispatch_async(queue, ^{
    // 第一个读取任务
    [NSThread sleepForTimeInterval:2];

    NSLog(@"----执行第一个读取任务---当前线程%@",[NSThread currentThread]);

});
dispatch_async(queue, ^{
    // 第二个读取任务
    [NSThread sleepForTimeInterval:2];

    NSLog(@"----执行第二个读取任务---当前线程%@",[NSThread currentThread]);

});

NSLog(@"-----end-----当前线程---%@",[NSThread currentThread]);

}
```

11. dispatch_suspend & dispatch_resume:

```
//场景：当追加大量处理到Dispatch Queue时，在追加处理的过程中，有时希望不执行已追加的处理。例如演算结果被Block截获时，
//理会对这个演算结果造成影响。在这种情况下，只要挂起Dispatch Queue即可。当可以执行时再恢复。
//总结:dispatch_suspend, dispatch_resume提供了“挂起、恢复”队列的功能，简单来说，就是可以暂停、恢复队列上的任务。但是
//的“挂起”，并不能保证可以立即停止队列上正在运行的任务，也就是如果挂起之前已经有队列中的任务在进行中，那么该任务依然会被执
//毕
-(void)dispatch_suspend{

    NSLog(@"-----start-----当前线程---%@",[NSThread currentThread]);

    dispatch_queue_t queue = dispatch_queue_create("com.test.testQueue", DISPATCH_QUEUE_SERIAL);

    dispatch_async(queue, ^{
        // 执行第一个任务
        [NSThread sleepForTimeInterval:5];

        NSLog(@"----执行第一个任务---当前线程%@",[NSThread currentThread]);

    });

    dispatch_async(queue, ^{
        // 执行第二个任务
        [NSThread sleepForTimeInterval:5];

        NSLog(@"----执行第二个任务---当前线程%@",[NSThread currentThread]);

    });

    dispatch_async(queue, ^{
        // 执行第三个任务
        [NSThread sleepForTimeInterval:5];

        NSLog(@"----执行第三个任务---当前线程%@",[NSThread currentThread]);

    });

    //此时发现意外情况，挂起队列
    NSLog(@"suspend");
    dispatch_suspend(queue);

    //挂起10秒之后，恢复正常
    [NSThread sleepForTimeInterval:10];

    //恢复队列
```

```
NSLog(@"resume");
dispatch_resume(queue);

NSLog(@"----end-----当前线程---%@",[NSThread currentThread]);

}
```

参考资料：

- Objective-C 高级编程 iOS 与 OS X 多线程和内存管理