iOS 多线程：『GCD』详尽总结

行走少年郎 关注

53 2016.09.03 19:47:39 字数 11,571 阅读 205,541

- 本文首发于我的个人博客：『不羁阁』
- 文章链接：传送门
- 本文更新时间： 2019-09-14 15:35:48

再次感谢大家对这篇文章的喜欢和支持。为了更好的让大家了解 iOS 多线程，以及 GCD 的相关知识，我第三次对这篇文章进行了梳理，修改了 **GCD 不同组合方式区别的相关总结，以及 队列、任务以及线程之间关系的形象理解。**

本文用来介绍 iOS 多线程中 GCD 的相关知识以及使用方法。这大概是史上最详细、清晰的关于 **GCD 的详细讲解 + 总结** 的文章了。通过本文，您将了解到：

1. GCD 简介

2. GCD 任务和队列

3. GCD 的使用步骤

4. GCD 的基本使用（六种组合不同区别，队列嵌套情况区别，相互关系形象理解）

5. GCD 线程间的通信

6. GCD 的其他方法（栅栏方法：dispatch_barrier_async、延时执行方法：dispatch_after、一次性代码（只执行一次）：dispatch_once、快速迭代方法：dispatch_apply、队列组：dispatch_group、信号量：dispatch_semaphore）

行走少年郎 总资产64 (约5

百度智能云 1212岁末感恩季 AI人脸离线SDK 仅售199元/个 注册即领 12120元感恩福袋 立即注册

1. GCD 简介

什么是 『**GCD**』？我们先来看看百度百科的解释简单了解下相关概念。

> 引自 百度百科
>
> **Grand Central Dispatch（GCD）** 是 Apple 开发的一个多核编程的较新的解决方法。它主要用于优化应用程序以支持多核处理器以及其他对称多处理系统。它是一个在线程池模式的基础上执行的并发任务。在 Mac OS X 10.6 雪豹中首次推出，也可在 iOS 4 及以上版本使用。

那为什么我们要使用 GCD 呢?

因为使用 GCD 有很多好处啊，具体如下：

- GCD 可用于多核的并行运算；
- GCD 会自动利用更多的 CPU 内核（比如双核、四核）；
- 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）；
- 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。

GCD 拥有以上这么多的好处，而且在多线程中处于举足轻重的地位。那么我们就很有必要系统地学习一下 GCD 的使用方法。

---

2. GCD 任务和队列

学习 GCD 之前，先来了解 GCD 中两个核心概念：**『任务』** 和 **『队列』**。

任务：就是执行操作的意思，换句话说就是你在线程中执行的那段代码。在 GCD 中是放在 block 中的。执行任务有两种方式：**『同步执行』** 和 **『异步执行』**。两者的主要区别是：**是否等待队列的任务执行结束，以及是否具备开启新线程的能力。**

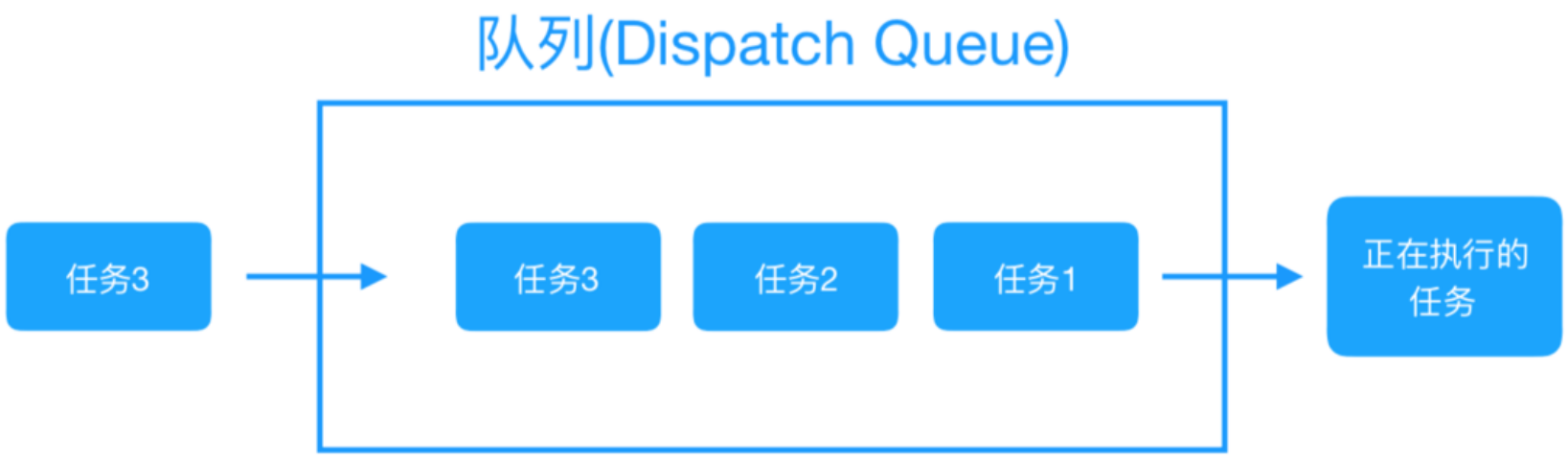
- **同步执行（sync）**：
  - 同步添加任务到指定的队列中，在添加的任务执行结束之前，会一直等待，直到队列里面的任务完成之后再继续执行。
  - 只能在当前线程中执行任务，不具备开启新线程的能力。
- **异步执行（async）**：
  - 异步添加任务到指定的队列中，它不会做任何等待，可以继续执行任务。
  - 可以在新的线程中执行任务，具备开启新线程的能力。

举个简单例子：你要打电话给小明和小白。
『同步执行』 就是：你打电话给小明的时候，不能同时打给小白。只有等到给小明打完了，才能打给小白（等待任务执行结束）。而且只能用当前的电话（不具备开启新线程的能力）。
『异步执行』 就是：你打电话给小明的时候，不用等着和小明通话结束（不用等待任务执行结

束），还能同时给小白打电话。而且除了当前电话，你还可以使用其他一个或多个电话（具备开启新线程的能力）。

> 注意：**异步执行（async）**虽然具有开启新线程的能力，但是并不一定开启新线程。这跟任务所指定的队列类型有关（下面会讲）。

队列（Dispatch Queue）：这里的队列指执行任务的等待队列，即用来存放任务的队列。队列是一种特殊的线性表，采用 FIFO（先进先出）的原则，即新任务总是被插入到队列的末尾，而读取任务的时候总是从队列的头部开始读取。每读取一个任务，则从队列中释放一个任务。队列的结构可参考下图：



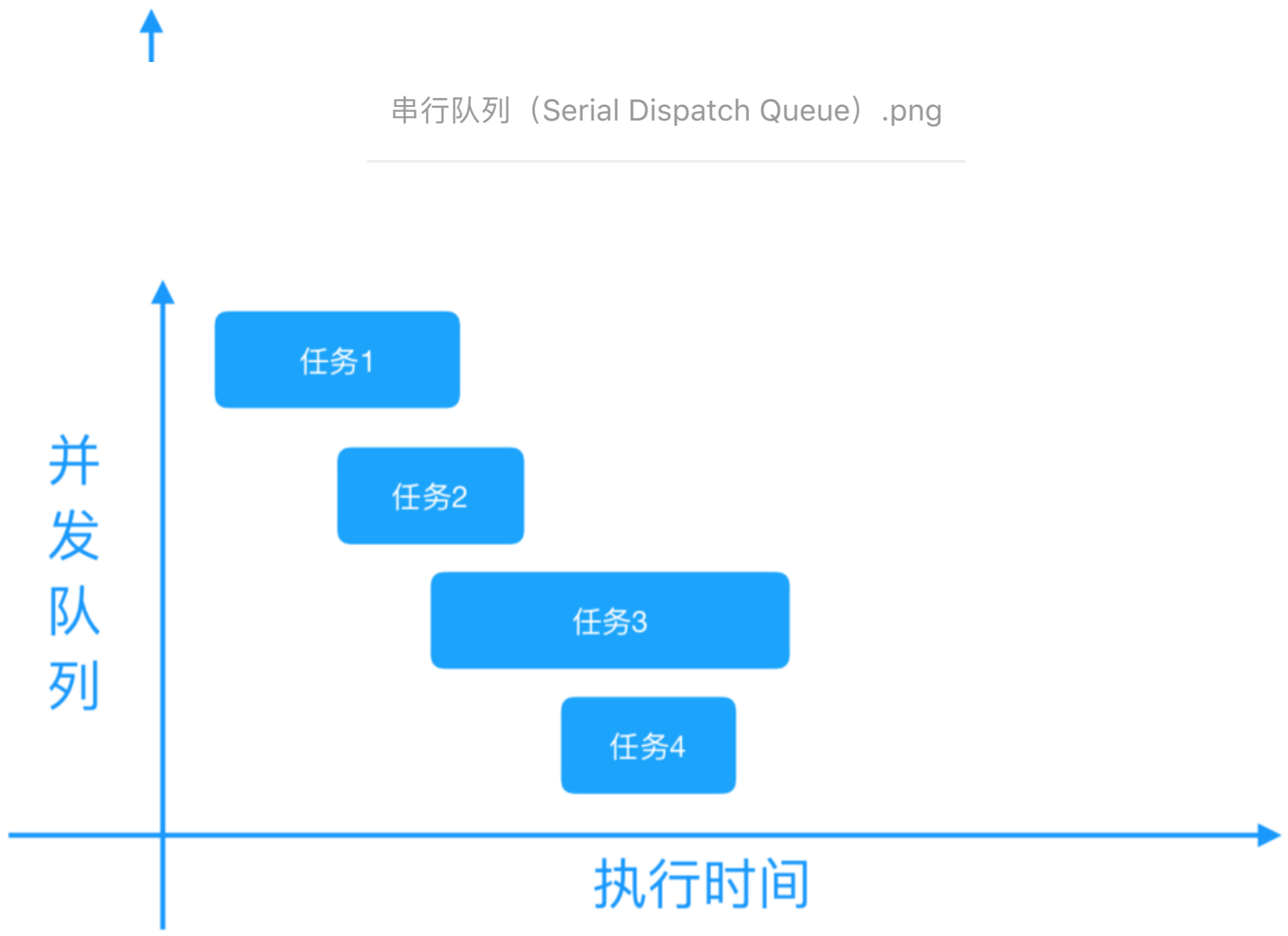
队列（Dispatch Queue）.png

在 GCD 中有两种队列：**『串行队列』** 和 **『并发队列』**。两者都符合 FIFO（先进先出）的原则。两者的主要区别是：**执行顺序不同，以及开启线程数不同。**

- **串行队列（Serial Dispatch Queue）**：
  - 每次只有一个任务被执行。让任务一个接着一个地执行。（只开启一个线程，一个任务执行完毕后，再执行下一个任务）
- **并发队列（Concurrent Dispatch Queue）**：
  - 可以让多个任务并发（同时）执行。（可以开启多个线程，并且同时执行任务）

> 注意：**并发队列** 的并发功能只有在异步（dispatch_async）方法下才有效。

两者具体区别如下两图所示：



串行队列（Serial Dispatch Queue）.png

并发队列（Concurrent Dispatch Queue）.png

3. GCD 的使用步骤

GCD 的使用步骤其实很简单，只有两步：

1. 创建一个队列（串行队列或并发队列）；
2. 将任务追加到任务的等待队列中，然后系统就会根据任务类型执行任务（同步执行或异步执行）。

下边来看看队列的创建方法 / 获取方法，以及任务的创建方法。

3.1 队列的创建方法 / 获取方法

- 可以使用 `dispatch_queue_create` 方法来创建队列。该方法需要传入两个参数：
  - 第一个参数表示队列的唯一标识符，用于 DEBUG，可为空。队列的名称推荐使用应用程序 ID 这种逆序全程域名。
  - 第二个参数用来识别是串行队列还是并发队列。`DISPATCH_QUEUE_SERIAL` 表示串行队列，`DISPATCH_QUEUE_CONCURRENT` 表示并发队列。

```
// 串行队列的创建方法
dispatch_queue_t queue = dispatch_queue_create("net.bujige.testQueue", DISPATCH_QUEUE_SERIAL);
// 并发队列的创建方法
dispatch_queue_t queue = dispatch_queue_create("net.bujige.testQueue", DISPATCH_QUEUE_CONCURRENT);
```

- 对于串行队列，GCD 默认提供了：**『主队列（Main Dispatch Queue）』**。
  - 所有放在主队列中的任务，都会放到主线程中执行。
  - 可使用 `dispatch_get_main_queue()` 方法获得主队列。

> 注意：**主队列其实并不特殊。** 主队列的实质上就是一个普通的串行队列，只是因为默认情况下，当前代码是放在主队列中的，然后主队列中的代码，有都会放到主线程中去执行，所以才造成了主队列特殊的现象。

```
// 主队列的获取方法
dispatch_queue_t queue = dispatch_get_main_queue();
```

- 对于并发队列，GCD 默认提供了 **『全局并发队列（Global Dispatch Queue）』**。
  - 可以使用 `dispatch_get_global_queue` 方法来获取全局并发队列。需要传入两个参数。第一个参数表示队列优先级，一般用 `DISPATCH_QUEUE_PRIORITY_DEFAULT`。第二个参数暂时没用，用 `0` 即可。

```
// 全局并发队列的获取方法
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
```

3.2 任务的创建方法

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`。

```
// 同步执行任务创建方法
dispatch_sync(queue, ^{
    // 这里放同步执行任务代码
});
// 异步执行任务创建方法
dispatch_async(queue, ^{
    // 这里放异步执行任务代码
});
```

虽然使用 GCD 只需两步，但是既然我们有两种队列（串行队列 / 并发队列），两种任务执行方式（同步执行 / 异步执行），那么我们就有了四种不同的组合方式。这四种不同的组合方式是：

> 1. 同步执行 + 并发队列
> 2. 异步执行 + 并发队列
> 3. 同步执行 + 串行队列
> 4. 异步执行 + 串行队列

实际上，刚才还说了两种默认队列：全局并发队列、主队列。全局并发队列可以作为普通并发队列来使用。但是当前代码默认放在主队列中，所以主队列很有必要专门来研究一下，所以我们就又多了两种组合方式。这样就有六种不同的组合方式了。

> 5. 同步执行 + 主队列
> 6. 异步执行 + 主队列

那么这几种不同组合方式各有什么区别呢?

这里我们先上结论，后面再来详细讲解。你可以直接查看 **3.3 任务和队列不同组合方式的区别** 中的表格结果，然后跳过 **4. GCD的基本使用** 继续往后看。

3.3 任务和队列不同组合方式的区别

我们先来考虑最基本的使用，也就是当前线程为 **『主线程』** 的环境下，**『不同队列』+『不同任务』** 简单组合使用的不同区别。暂时不考虑 **『队列中嵌套队列』** 的这种复杂情况。

『主线程』中，**『不同队列』+『不同任务』**简单组合的区别：

区别	并发队列	串行队列	主队列
同步（sync）	没有开启新线程，串行执行任务	没有开启新线程，串行执行任务	死锁卡住不执行
异步（async）	有开启新线程，并发执行任务	有开启新线程（1条），串行执行任务	没有开启新线程，串行执行任务

> 注意：从上边可看出： **『主线程』** 中调用 **『主队列』+『同步执行』** 会导致死锁问题。
> 这是因为 **主队列中追加的同步任务** 和 **主线程本身的任务** 两者之间相互等待，阻塞了 **『主队列』**，最终造成了主队列所在的线程（主线程）死锁问题。
> 而如果我们在 **『其他线程』** 调用 **『主队列』+『同步执行』**，则不会阻塞 **『主队列』**，自然也不会造成死锁问题。最终的结果是：**不会开启新线程，串行执行任务**。

3.4 队列嵌套情况下，不同组合方式区别

除了上边提到的『主线程』中调用『主队列』+『同步执行』会导致死锁问题。实际在使用『串行队列』的时候，也可能出现阻塞『串行队列』所在线程的情况发生，从而造成死锁问题。这种情况多见于同一个串行队列的嵌套使用。

比如下面代码这样：在『异步执行』+『串行队列』的任务中，又嵌套了『当前的串行队列』，然后进行『同步执行』。

```objc
dispatch_queue_t queue = dispatch_queue_create("test.queue", DISPATCH_QUEUE_SERIAL);
dispatch_async(queue, ^{    // 异步执行 + 串行队列
    dispatch_sync(queue, ^{  // 同步执行 + 当前串行队列
        // 追加任务 1
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"1---%@",[NSThread currentThread]);      // 打印当前线程
    });
});
```

> 执行上面的代码会导致 **串行队列中追加的任务** 和 **串行队列中原有的任务** 两者之间相互等待，阻塞了『串行队列』，最终造成了串行队列所在的线程（子线程）死锁问题。
>
> 主队列造成死锁也是基于这个原因，所以，这也进一步说明了主队列其实并不特殊。

关于 『队列中嵌套队列』这种复杂情况，这里也简单做一个总结。不过这里只考虑同一个队列的嵌套情况，关于多个队列的相互嵌套情况还请自行研究，或者等我最新的文章发布。

『不同队列』+『不同任务』 组合，以及 **『队列中嵌套队列』** 使用的区别：

区别	**『异步执行+并发队列』嵌套『同一个并发队列』**	**『同步执行+并发队列』嵌套『同一个并发队列』**	**『异步执行+串行队列』嵌套『同一个串行队列』**	**『同步执行+串行队列』嵌套『同一个串行队列』**
同步（sync）	没有开启新的线程，串行执行任务	没有开启新线程，串行执行任务	死锁卡住不执行	死锁卡住不执行
异步（async）	有开启新线程，并发执行任务	有开启新线程，并发执行任务	有开启新线程（1条），串行执行任务	有开启新线程（1条），串行执行任务

好了，关于『不同队列』+『不同任务』 组合不同区别总结就到这里。

3.5 关于不同队列和不同任务的形象理解

因为前一段时间看到了有朋友留言说对 `异步执行` 和 `并发队列` 中创建线程能力有所不理解，我觉得这个问题的确很容易造成困惑，所以很值得拿来专门分析一下。

> 他的问题：
>
> 在 异步 + 并发 中的解释：
> （异步执行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）
>
> 以及 同步 + 并发 中的解释：
> （虽然并发队列可以开启多个线程，并且同时执行多个任务。但是因为本身不能创建新线程，只有当前线程这一个线程（同步任务不具备开启新线程的能力）
>
> 这个地方看起来有点疑惑，你两个地方分别提到：异步执行开启新线程，并发队列也可以开启新线程，想请教下，你的意思是只有任务才拥有创建新线程的能力，而队列只有开启线程的能力，并不能创建线程？ 这二者是这样的关联吗?

关于这个问题，我想做一个很形象的类比，来帮助大家对 **队列**、**任务** 以及 **线程** 之间关系的理解。

> 假设现在有 5 个人要穿过一道门禁，这道门禁总共有 10 个入口，管理员可以决定同一时间打开几个入口，可以决定同一时间让一个人单独通过还是多个人一起通过。不过默认情况下，管理员只开启一个入口，且一个通道一次只能通过一个人。
>
> - 这个故事里，人好比是 **任务**，管理员好比是 **系统**，入口则代表 **线程**。
>
> - 5 个人表示有 5 个任务，10 个入口代表 10 条线程。
> - **串行队列** 好比是 5 个人排成一支长队。

    - **并发队列** 好比是 5 个人排成多支队伍，比如 2 队，或者 3 队。
    - **同步任务** 好比是管理员只开启了一个入口（当前线程）。
    - **异步任务** 好比是管理员同时开启了多个入口（当前线程 + 新开的线程）。
- **『异步执行 + 并发队列』** 可以理解为：现在管理员开启了多个入口（比如 3 个入口），5 个人排成了多支队伍（比如 3 支队伍），这样这 5 个人就可以 3 个人同时一起穿过门禁了。
- **『同步执行 + 并发队列』** 可以理解为：现在管理员只开启了 1 个入口，5 个人排成了多支队伍。虽然这 5 个人排成了多支队伍，但是只开了 1 个入口啊，这 5 个人虽然都想快点过去，但是 1 个入口一次只能过 1 个人，所以大家就只好一个接一个走过去了，表现的结果就是：顺次通过入口。
- 换成 GCD 里的语言就是说：

    - 『异步执行 + 并发队列』就是：系统开启了多个线程（主线程+其他子线程），任务可以多个同时运行。
    - 『同步执行 + 并发队列』就是：系统只默认开启了一个主线程，没有开启子线程，虽然任务处于并发队列中，但也只能一个接一个执行了。

下边我们来研究一下上边提到的六种简单组合方式的使用方法。

---

4. GCD 的基本使用

先来讲讲并发队列的两种执行方式。

4.1 同步执行 + 并发队列

- 在当前线程中执行任务，不会开启新线程，执行完一个任务，再执行下一个任务。

```
/**
 * 同步执行 + 并发队列
 * 特点：在当前线程中执行任务，不会开启新线程，执行完一个任务，再执行下一个任务。
 */
- (void)syncConcurrent {
    NSLog(@"currentThread---%@",[NSThread currentThread]);  // 打印当前线程
    NSLog(@"syncConcurrent---begin");

    dispatch_queue_t queue = dispatch_queue_create("net.bujige.testQueue", DISPATCH_QUEUE_CONCURR

    dispatch_sync(queue, ^{
        // 追加任务 1
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"1---%@",[NSThread currentThread]);      // 打印当前线程
    });

    dispatch_sync(queue, ^{
        // 追加任务 2
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"2---%@",[NSThread currentThread]);      // 打印当前线程
    });

    dispatch_sync(queue, ^{
        // 追加任务 3
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"3---%@",[NSThread currentThread]);      // 打印当前线程
    });

```

```
    NSLog(@"syncConcurrent---end");
}
```

输出结果：

2019-08-08 14:32:53.542816+0800 YSC-GCD-demo[16332:4171500] currentThread---<NSThread: 0x600002326940>{number = 1, name = main}
2019-08-08 14:32:53.542964+0800 YSC-GCD-demo[16332:4171500] syncConcurrent---begin
2019-08-08 14:32:55.544329+0800 YSC-GCD-demo[16332:4171500] 1---<NSThread: 0x600002326940>{number = 1, name = main}
2019-08-08 14:32:57.545779+0800 YSC-GCD-demo[16332:4171500] 2---<NSThread: 0x600002326940>{number = 1, name = main}
2019-08-08 14:32:59.547154+0800 YSC-GCD-demo[16332:4171500] 3---<NSThread: 0x600002326940>{number = 1, name = main}
2019-08-08 14:32:59.547365+0800 YSC-GCD-demo[16332:4171500] syncConcurrent---end

从 `同步执行 + 并发队列` 中可看到：

- 所有任务都是在当前线程（主线程）中执行的，没有开启新的线程（`同步执行` 不具备开启新线程的能力）。
- 所有任务都在打印的 `syncConcurrent---begin` 和 `syncConcurrent---end` 之间执行的（`同步任务` 需要等待队列的任务执行结束）。
- 任务按顺序执行的。按顺序执行的原因：虽然 `并发队列` 可以开启多个线程，并且同时执行多个任务。但是因为本身不能创建新线程，只有当前线程这一个线程（`同步任务` 不具备开启新线程的能力），所以也就不存在并发。而且当前线程只有等待当前队列中正在执行的任务执行完毕之后，才能继续接着执行下面的操作（`同步任务` 需要等待队列的任务执行结束）。所以任务只能一个接一个按顺序执行，不能同时被执行。

4.2 异步执行 + 并发队列

- 可以开启多个线程，任务交替（同时）执行。

```
/**
 * 异步执行 + 并发队列
 * 特点：可以开启多个线程，任务交替（同时）执行。
 */
- (void)asyncConcurrent {
    NSLog(@"currentThread---%@",[NSThread currentThread]);  // 打印当前线程
    NSLog(@"asyncConcurrent---begin");

    dispatch_queue_t queue = dispatch_queue_create("net.bujige.testQueue", DISPATCH_QUEUE_CONCURR

    dispatch_async(queue, ^{
        // 追加任务 1
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"1---%@",[NSThread currentThread]);      // 打印当前线程
    });

```

```
    dispatch_async(queue, ^{
        // 追加任务 2
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"2---%@",[NSThread currentThread]);      // 打印当前线程
    });

    dispatch_async(queue, ^{
        // 追加任务 3
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"3---%@",[NSThread currentThread]);      // 打印当前线程
    });

    NSLog(@"asyncConcurrent---end");
}
```

> 输出结果：
>
> 2019-08-08 14:36:37.747966+0800 YSC-GCD-demo[17232:4187114] currentThread---<NSThread: 0x60000206d380>{number = 1, name = main}
>
> 2019-08-08 14:36:37.748150+0800 YSC-GCD-demo[17232:4187114] asyncConcurrent---begin
>
> 2019-08-08 14:36:37.748279+0800 YSC-GCD-demo[17232:4187114] asyncConcurrent---end
>
> 2019-08-08 14:36:39.752523+0800 YSC-GCD-demo[17232:4187204] 2---<NSThread: 0x600002010980>{number = 3, name = (null)}
>
> 2019-08-08 14:36:39.752527+0800 YSC-GCD-demo[17232:4187202] 3---<NSThread: 0x600002018480>{number = 5, name = (null)}
>
> 2019-08-08 14:36:39.752527+0800 YSC-GCD-demo[17232:4187203] 1---<NSThread: 0x600002023400>{number = 4, name = (null)}

在 `异步执行 + 并发队列` 中可以看出：

- 除了当前线程（主线程），系统又开启了 3 个线程，并且任务是交替/同时执行的。（`异步执行` 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）。
- 所有任务是在打印的 `syncConcurrent---begin` 和 `syncConcurrent---end` 之后才执行的。说明当前线程没有等待，而是直接开启了新线程，在新线程中执行任务（`异步执行` 不做等待，可以继续执行任务）。

接下来再来讲讲串行队列的两种执行方式。

4.3 同步执行 + 串行队列

- 不会开启新线程，在当前线程执行任务。任务是串行的，执行完一个任务，再执行下一个任务。

```
/**
 * 同步执行 + 串行队列
 * 特点：不会开启新线程，在当前线程执行任务。任务是串行的，执行完一个任务，再执行下一个任务。
 */
- (void)syncSerial {
    NSLog(@"currentThread---%@",[NSThread currentThread]);  // 打印当前线程
    NSLog(@"syncSerial---begin");
```

```
    dispatch_queue_t queue = dispatch_queue_create("net.bujige.testQueue", DISPATCH_QUEUE_SERIAL);

    dispatch_sync(queue, ^{
        // 追加任务 1
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"1---%@",[NSThread currentThread]);      // 打印当前线程
    });
    dispatch_sync(queue, ^{
        // 追加任务 2
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"2---%@",[NSThread currentThread]);      // 打印当前线程
    });
    dispatch_sync(queue, ^{
        // 追加任务 3
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"3---%@",[NSThread currentThread]);      // 打印当前线程
    });

    NSLog(@"syncSerial---end");
}
```

输出结果为：

2019-08-08 14:39:31.366815+0800 YSC-GCD-demo[17285:4197645] currentThread---<NSThread: 0x600001b5e940>{number = 1, name = main}
2019-08-08 14:39:31.366952+0800 YSC-GCD-demo[17285:4197645] syncSerial---begin
2019-08-08 14:39:33.368256+0800 YSC-GCD-demo[17285:4197645] 1---<NSThread: 0x600001b5e940>{number = 1, name = main}
2019-08-08 14:39:35.369661+0800 YSC-GCD-demo[17285:4197645] 2---<NSThread: 0x600001b5e940>{number = 1, name = main}
2019-08-08 14:39:37.370991+0800 YSC-GCD-demo[17285:4197645] 3---<NSThread: 0x600001b5e940>{number = 1, name = main}
2019-08-08 14:39:37.371192+0800 YSC-GCD-demo[17285:4197645] syncSerial---end

在 `同步执行 + 串行队列` 可以看到：

- 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（`同步执行` 不具备开启新线程的能力）。
- 所有任务都在打印的 `syncConcurrent---begin` 和 `syncConcurrent---end` 之间执行（`同步任务` 需要等待队列的任务执行结束）。
- 任务是按顺序执行的（`串行队列` 每次只有一个任务被执行，任务一个接一个按顺序执行）。

4.4 异步执行 + 串行队列

- 会开启新线程，但是因为任务是串行的，执行完一个任务，再执行下一个任务

```
/**
 * 异步执行 + 串行队列
 * 特点：会开启新线程，但是因为任务是串行的，执行完一个任务，再执行下一个任务。
 */
- (void)asyncSerial {
```

```
    NSLog(@"currentThread---%@",[NSThread currentThread]);  // 打印当前线程
    NSLog(@"asyncSerial---begin");

    dispatch_queue_t queue = dispatch_queue_create("net.bujige.testQueue", DISPATCH_QUEUE_SERIAL);

    dispatch_async(queue, ^{
        // 追加任务 1
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"1---%@",[NSThread currentThread]);      // 打印当前线程
    });
    dispatch_async(queue, ^{
        // 追加任务 2
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"2---%@",[NSThread currentThread]);      // 打印当前线程
    });
    dispatch_async(queue, ^{
        // 追加任务 3
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"3---%@",[NSThread currentThread]);      // 打印当前线程
    });

    NSLog(@"asyncSerial---end");
}
```

输出结果为：

```
2019-08-08 14:40:53.944502+0800 YSC-GCD-demo[17313:4203018]
currentThread---<NSThread: 0x6000015da940>{number = 1, name = main}
2019-08-08 14:40:53.944615+0800 YSC-GCD-demo[17313:4203018]
asyncSerial---begin
2019-08-08 14:40:53.944710+0800 YSC-GCD-demo[17313:4203018]
asyncSerial---end
2019-08-08 14:40:55.947709+0800 YSC-GCD-demo[17313:4203079] 1---
<NSThread: 0x6000015a0840>{number = 3, name = (null)}
2019-08-08 14:40:57.952453+0800 YSC-GCD-demo[17313:4203079] 2---
<NSThread: 0x6000015a0840>{number = 3, name = (null)}
2019-08-08 14:40:59.952943+0800 YSC-GCD-demo[17313:4203079] 3---
<NSThread: 0x6000015a0840>{number = 3, name = (null)}
```

在 `异步执行 + 串行队列` 可以看到：

- 开启了一条新线程（`异步执行` 具备开启新线程的能力，`串行队列` 只开启一个线程）。
- 所有任务是在打印的 `syncConcurrent---begin` 和 `syncConcurrent---end` 之后才开始执行的（`异步执行` 不会做任何等待，可以继续执行任务）。
- 任务是按顺序执行的（`串行队列` 每次只有一个任务被执行，任务一个接一个按顺序执行）。

下边讲讲刚才我们提到过的：**主队列**。

- 主队列：GCD 默认提供的 **串行队列**。
  - 默认情况下，平常所写代码是直接放在主队列中的。
  - 所有放在主队列中的任务，都会放到主线程中执行。
  - 可使用 `dispatch_get_main_queue()` 获得主队列。

我们再来看看主队列的两种组合方式。

4.5 同步执行 + 主队列

`同步执行 + 主队列` 在不同线程中调用结果也是不一样，在主线程中调用会发生死锁问题，而在其他线程中调用则不会。

4.5.1 在主线程中调用 『同步执行 + 主队列』

- 互相等待卡住不可行

```
/**
 * 同步执行 + 主队列
 * 特点(主线程调用)：互等卡主不执行。
 * 特点(其他线程调用)：不会开启新线程，执行完一个任务，再执行下一个任务。
 */
- (void)syncMain {

    NSLog(@"currentThread---%@",[NSThread currentThread]);  // 打印当前线程
    NSLog(@"syncMain---begin");

    dispatch_queue_t queue = dispatch_get_main_queue();

    dispatch_sync(queue, ^{
        // 追加任务 1
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"1---%@",[NSThread currentThread]);      // 打印当前线程
    });

    dispatch_sync(queue, ^{
        // 追加任务 2
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"2---%@",[NSThread currentThread]);      // 打印当前线程
    });

    dispatch_sync(queue, ^{
        // 追加任务 3
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"3---%@",[NSThread currentThread]);      // 打印当前线程
    });

    NSLog(@"syncMain---end");
}
```

> 输出结果
> 2019-08-08 14:43:58.062376+0800 YSC-GCD-demo[17371:4213562] currentThread---<NSThread: 0x6000026e2940>{number = 1, name = main}
> 2019-08-08 14:43:58.062518+0800 YSC-GCD-demo[17371:4213562] syncMain---begin
> (lldb)

在主线程中使用 `同步执行 + 主队列` 可以惊奇的发现：

- 追加到主线程的任务 1、任务 2、任务 3 都不再执行了，而且 `syncMain---end` 也没有打印，在 XCode 9 及以上版本上还会直接报崩溃。这是为什么呢?

这是因为我们在主线程中执行 `syncMain` 方法，相当于把 `syncMain` 任务放到了主线程的队列中。而 `同步执行` 会等待当前队列中的任务执行完毕，才会接着执行。那么当我们把 `任务 1` 追加到主队列中，`任务 1` 就在等待主线程处理完 `syncMain` 任务。而 `syncMain` 任务需要等待 `任务 1` 执行完毕，才能接着执行。

那么，现在的情况就是 `syncMain` 任务和 `任务 1` 都在等对方执行完毕。这样大家互相等待，所以就卡住了，所以我们的任务执行不了，而且 `syncMain---end` 也没有打印。

要是如果不在主线程中调用，而在其他线程中调用会如何呢?

4.5.2 在其他线程中调用『同步执行 + 主队列』

- 不会开启新线程，执行完一个任务，再执行下一个任务

```
// 使用 NSThread 的 detachNewThreadSelector 方法会创建线程，并自动启动线程执行 selector 任务
[NSThread detachNewThreadSelector:@selector(syncMain) toTarget:self withObject:nil];
```

输出结果：

```
2019-08-08 14:51:38.137978+0800 YSC-GCD-demo[17482:4237818] currentThread---<NSThread: 0x600001dd6c00>{number = 3, name = (null)}
2019-08-08 14:51:38.138159+0800 YSC-GCD-demo[17482:4237818] syncMain---begin
2019-08-08 14:51:40.149065+0800 YSC-GCD-demo[17482:4237594] 1---<NSThread: 0x600001d8d380>{number = 1, name = main}
2019-08-08 14:51:42.151104+0800 YSC-GCD-demo[17482:4237594] 2---<NSThread: 0x600001d8d380>{number = 1, name = main}
2019-08-08 14:51:44.152583+0800 YSC-GCD-demo[17482:4237594] 3---<NSThread: 0x600001d8d380>{number = 1, name = main}
2019-08-08 14:51:44.152767+0800 YSC-GCD-demo[17482:4237818] syncMain---end
```

在其他线程中使用 `同步执行 + 主队列` 可看到：

- 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）。
- 所有任务都在打印的 `syncConcurrent---begin` 和 `syncConcurrent---end` 之间执行（`同步任务` 需要等待队列的任务执行结束）。
- 任务是按顺序执行的（主队列是 `串行队列`，每次只有一个任务被执行，任务一个接一个按顺序执行）。

为什么现在就不会卡住了呢?

因为 `syncMain 任务` 放到了其他线程里，而 `任务 1`、`任务 2`、`任务3` 都在追加到主队列中，这三个任务都会在主线程中执行。`syncMain 任务` 在其他线程中执行到追加 `任务 1` 到主队列中，因为主队列现在没有正在执行的任务，所以，会直接执行主队列的 `任务1`，等 `任务1` 执行完毕，再接着执行 `任务 2`、`任务 3`。所以这里不会卡住线程，也就不会造成死锁问题。

4.6 异步执行 + 主队列

- 只在主线程中执行任务，执行完一个任务，再执行下一个任务。

```
/**
 * 异步执行 + 主队列
 * 特点：只在主线程中执行任务，执行完一个任务，再执行下一个任务
 */
- (void)asyncMain {
    NSLog(@"currentThread---%@",[NSThread currentThread]);  // 打印当前线程
    NSLog(@"asyncMain---begin");

    dispatch_queue_t queue = dispatch_get_main_queue();

    dispatch_async(queue, ^{
        // 追加任务 1
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"1---%@",[NSThread currentThread]);      // 打印当前线程
    });

    dispatch_async(queue, ^{
        // 追加任务 2
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"2---%@",[NSThread currentThread]);      // 打印当前线程
    });

    dispatch_async(queue, ^{
        // 追加任务 3
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"3---%@",[NSThread currentThread]);      // 打印当前线程
    });

    NSLog(@"asyncMain---end");
}
```

> 输出结果：
> 2019-08-08 14:53:27.023091+0800 YSC-GCD-demo[17521:4243690] currentThread---<NSThread: 0x6000022a1380>{number = 1, name = main}
> 2019-08-08 14:53:27.023247+0800 YSC-GCD-demo[17521:4243690] asyncMain---begin
> 2019-08-08 14:53:27.023399+0800 YSC-GCD-demo[17521:4243690] asyncMain---end
> 2019-08-08 14:53:29.035565+0800 YSC-GCD-demo[17521:4243690] 1---<NSThread: 0x6000022a1380>{number = 1, name = main}
> 2019-08-08 14:53:31.036565+0800 YSC-GCD-demo[17521:4243690] 2---<NSThread: 0x6000022a1380>{number = 1, name = main}
> 2019-08-08 14:53:33.037092+0800 YSC-GCD-demo[17521:4243690] 3---<NSThread: 0x6000022a1380>{number = 1, name = main}

在 `异步执行 + 主队列` 可以看到：

- 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，但因为是主队列，所以所有任务都在主线程中）。
- 所有任务是在打印的 `syncConcurrent---begin` 和 `syncConcurrent---end` 之后才开始执行的（异步

执行不会做任何等待，可以继续执行任务）。
- 任务是按顺序执行的（因为主队列是 `串行队列`，每次只有一个任务被执行，任务一个接一个按顺序执行）。

弄懂了难理解、绕来绕去的『**不同队列**』+『**不同任务**』使用区别之后，我们来学习一个简单的东西：**5. GCD 线程间的通信**。

---

5. GCD 线程间的通信

在 iOS 开发过程中，我们一般在主线程里边进行 UI 刷新，例如：点击、滚动、拖拽等事件。我们通常把一些耗时的操作放在其他线程，比如说图片下载、文件上传等耗时操作。而当我们有时候在其他线程完成了耗时操作时，需要回到主线程，那么就用到了线程之间的通讯。

```
/**
 * 线程间通信
 */
- (void)communication {
    // 获取全局并发队列
    dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    // 获取主队列
    dispatch_queue_t mainQueue = dispatch_get_main_queue();

    dispatch_async(queue, ^{
        // 异步追加任务 1
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"1---%@",[NSThread currentThread]);      // 打印当前线程

        // 回到主线程
        dispatch_async(mainQueue, ^{
            // 追加在主线程中执行的任务
            [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
            NSLog(@"2---%@",[NSThread currentThread]);      // 打印当前线程
        });
    });
}
```

> 输出结果：
>
> 2019-08-08 14:56:22.973318+0800 YSC-GCD-demo[17573:4253201] 1---<NSThread: 0x60001846080>{number = 3, name = (null)}
>
> 2019-08-08 14:56:24.973902+0800 YSC-GCD-demo[17573:4253108] 2---<NSThread: 0x60000181e940>{number = 1, name = main}

- 可以看到在其他线程中先执行任务，执行完了之后回到主线程执行主线程的相应操作。

---

6. GCD 的其他方法

6.1 GCD 栅栏方法：dispatch_barrier_async

- 我们有时需要异步执行两组操作，而且第一组操作执行完之后，才能开始执行第二组操作。这样我们就需要一个相当于 `栅栏` 一样的一个方法将两组异步执行的操作组给分割起来，当然

这里的操作组里可以包含一个或多个任务。这就需要用到`dispatch_barrier_async`方法在两个操作组间形成栅栏。

`dispatch_barrier_async`方法会等待前边追加到并发队列中的任务全部执行完毕之后，再将指定的任务追加到该异步队列中。然后在`dispatch_barrier_async`方法追加的任务执行完毕之后，异步队列才恢复为一般动作，接着追加任务到该异步队列并开始执行。具体如下图所示：

dispatch_barrier_async.png

```
/**
 * 栅栏方法 dispatch_barrier_async
 */
- (void)barrier {
    dispatch_queue_t queue = dispatch_queue_create("net.bujige.testQueue", DISPATCH_QUEUE_CONCURR

    dispatch_async(queue, ^{
        // 追加任务 1
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"1---%@",[NSThread currentThread]);      // 打印当前线程
    });
    dispatch_async(queue, ^{
        // 追加任务 2
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"2---%@",[NSThread currentThread]);      // 打印当前线程
    });

    dispatch_barrier_async(queue, ^{
        // 追加任务 barrier
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"barrier---%@",[NSThread currentThread]);// 打印当前线程
    });

    dispatch_async(queue, ^{
        // 追加任务 3
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"3---%@",[NSThread currentThread]);      // 打印当前线程
    });
    dispatch_async(queue, ^{
        // 追加任务 4
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"4---%@",[NSThread currentThread]);      // 打印当前线程
    });
}
```

输出结果：

2019-08-08 14:59:02.540868+0800 YSC-GCD-demo[17648:4262933] 1---<NSThread: 0x600001ca4c40>{number = 3, name = (null)}

2019-08-08 14:59:02.540868+0800 YSC-GCD-demo[17648:4262932] 2---<NSThread: 0x600001c84a00>{number = 4, name = (null)}

2019-08-08 14:59:04.542346+0800 YSC-GCD-demo[17648:4262933] barrier---<NSThread: 0x600001ca4c40>{number = 3, name = (null)}

2019-08-08 14:59:06.542772+0800 YSC-GCD-demo[17648:4262932] 4---<NSThread: 0x600001c84a00>{number = 4, name = (null)}

2019-08-08 14:59:06.542773+0800 YSC-GCD-demo[17648:4262933] 3---<NSThread: 0x600001ca4c40>{number = 3, name = (null)}

在 `dispatch_barrier_async` 执行结果中可以看出：

- 在执行完栅栏前面的操作之后，才执行栅栏操作，最后再执行栅栏后边的操作。

6.2 GCD 延时执行方法：dispatch_after

我们经常会遇到这样的需求：在指定时间（例如 3 秒）之后执行某个任务。可以用 GCD 的 `dispatch_after` 方法来实现。

需要注意的是：`dispatch_after` 方法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。严格来说，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` 方法是很有效的。

```
/**
 * 延时执行方法 dispatch_after
 */
- (void)after {
    NSLog(@"currentThread---%@",[NSThread currentThread]);  // 打印当前线程
    NSLog(@"asyncMain---begin");

    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(2.0 * NSEC_PER_SEC)), dispatch_get_
        // 2.0 秒后异步追加任务代码到主队列，并开始执行
        NSLog(@"after---%@",[NSThread currentThread]);  // 打印当前线程
    });
}
```

输出结果：

2019-08-08 15:01:33.569710+0800 YSC-GCD-demo[17702:4272430] currentThread---<NSThread: 0x600001ead340>{number = 1, name = main}

2019-08-08 15:01:33.569838+0800 YSC-GCD-demo[17702:4272430] asyncMain---begin

2019-08-08 15:01:35.570146+0800 YSC-GCD-demo[17702:4272430] after---<NSThread: 0x600001ead340>{number = 1, name = main}

可以看出：在打印 `asyncMain---begin` 之后大约 2.0 秒的时间，打印了 `after---<NSThread: 0x600001ead340>{number = 1, name = main}`

6.3 GCD 一次性代码（只执行一次）：dispatch_once

- 我们在创建单例、或者有整个程序运行过程中只执行一次的代码时，我们就用到了 GCD 的 `dispatch_once` 方法。使用 `dispatch_once` 方法能保证某段代码在程序运行过程中只被执行 1 次，并且即使在多线程的环境下，`dispatch_once` 也可以保证线程安全。

```
/**
 * 一次性代码（只执行一次）dispatch_once
 */
- (void)once {
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        // 只执行 1 次的代码（这里面默认是线程安全的）
    });
}
```

6.4 GCD 快速迭代方法：dispatch_apply

- 通常我们会用 for 循环遍历，但是 GCD 给我们提供了快速迭代的方法 `dispatch_apply`。`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。

如果是在串行队列中使用 `dispatch_apply`，那么就和 for 循环一样，按顺序同步执行。但是这样就体现不出快速迭代的意义了。

我们可以利用并发队列进行异步执行。比如说遍历 0~5 这 6 个数字，for 循环的做法是每次取出一个元素，逐个遍历。`dispatch_apply` 可以 在多个线程中同时（异步）遍历多个数字。

还有一点，无论是在串行队列，还是并发队列中，dispatch_apply 都会等待全部任务执行完毕，这点就像是同步操作，也像是队列组中的 `dispatch_group_wait` 方法。

```
/**
 * 快速迭代方法 dispatch_apply
 */
- (void)apply {
    dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    NSLog(@"apply---begin");
    dispatch_apply(6, queue, ^(size_t index) {
        NSLog(@"%zd---%@",index, [NSThread currentThread]);
    });
    NSLog(@"apply---end");
}
```

> 输出结果：
> 2019-08-08 15:05:04.715266+0800 YSC-GCD-demo[17771:4285619] apply---begin
> 2019-08-08 15:05:04.715492+0800 YSC-GCD-demo[17771:4285619] 0---<NSThread: 0x600003bd1380>{number = 1, name = main}
> 2019-08-08 15:05:04.715516+0800 YSC-GCD-demo[17771:4285722] 1---<NSThread: 0x600003b82340>{number = 3, name = (null)}

```
2019-08-08 15:05:04.715526+0800 YSC-GCD-demo[17771:4285720] 3---<NSThread: 0x600003ba4cc0>{number = 5, name = (null)}
2019-08-08 15:05:04.715564+0800 YSC-GCD-demo[17771:4285721] 2---<NSThread: 0x600003bb9a80>{number = 7, name = (null)}
2019-08-08 15:05:04.715555+0800 YSC-GCD-demo[17771:4285719] 4---<NSThread: 0x600003b98100>{number = 6, name = (null)}
2019-08-08 15:05:04.715578+0800 YSC-GCD-demo[17771:4285728] 5---<NSThread: 0x600003beb400>{number = 4, name = (null)}
2019-08-08 15:05:04.715677+0800 YSC-GCD-demo[17771:4285619] apply---end
```

因为是在并发队列中异步执行任务，所以各个任务的执行时间长短不定，最后结束顺序也不定。但是 `apply---end` 一定在最后执行。这是因为 `dispatch_apply` 方法会等待全部任务执行完毕。

6.5 GCD 队列组：dispatch_group

有时候我们会有这样的需求：分别异步执行2个耗时任务，然后当2个耗时任务都执行完毕后再回到主线程执行任务。这时候我们可以用到 GCD 的队列组。

- 调用队列组的 `dispatch_group_async` 先把任务放到队列中，然后将队列放入队列组中。或者使用队列组的 `dispatch_group_enter`、`dispatch_group_leave` 组合来实现 `dispatch_group_async`。
- 调用队列组的 `dispatch_group_notify` 回到指定线程执行任务。或者使用 `dispatch_group_wait` 回到当前线程继续向下执行（会阻塞当前线程）。

6.5.1 dispatch_group_notify

- 监听 group 中任务的完成状态，当所有的任务都执行完成后，追加任务到 group 中，并执行任务。

```
/**
 * 队列组 dispatch_group_notify
 */
- (void)groupNotify {
    NSLog(@"currentThread---%@",[NSThread currentThread]);  // 打印当前线程
    NSLog(@"group---begin");

    dispatch_group_t group =  dispatch_group_create();

    dispatch_group_async(group, dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        // 追加任务 1
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"1---%@",[NSThread currentThread]);      // 打印当前线程
    });

    dispatch_group_async(group, dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        // 追加任务 2
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"2---%@",[NSThread currentThread]);      // 打印当前线程
    });

    dispatch_group_notify(group, dispatch_get_main_queue(), ^{
        // 等前面的异步任务 1、任务 2 都执行完毕后，回到主线程执行下边任务
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
```

```
        NSLog(@"3---%@",[NSThread currentThread]);      // 打印当前线程
        NSLog(@"group---end");
    });
}
```

输出结果：

2019-08-08 15:07:21.601734+0800 YSC-GCD-demo[17813:4293874] currentThread---<NSThread: 0x600003aad380>{number = 1, name = main}

2019-08-08 15:07:21.601871+0800 YSC-GCD-demo[17813:4293874] group---begin

2019-08-08 15:07:23.604854+0800 YSC-GCD-demo[17813:4294048] 2---<NSThread: 0x600003add100>{number = 4, name = (null)}

2019-08-08 15:07:23.604852+0800 YSC-GCD-demo[17813:4294053] 1---<NSThread: 0x600003ace4c0>{number = 3, name = (null)}

2019-08-08 15:07:25.606067+0800 YSC-GCD-demo[17813:4293874] 3---<NSThread: 0x600003aad380>{number = 1, name = main}

2019-08-08 15:07:25.606255+0800 YSC-GCD-demo[17813:4293874] group---end

从 `dispatch_group_notify` 相关代码运行输出结果可以看出：

当所有任务都执行完成之后，才执行 `dispatch_group_notify` 相关 block 中的任务。

6.5.2 dispatch_group_wait

- 暂停当前线程（阻塞当前线程），等待指定的 group 中的任务执行完成后，才会往下继续执行。

```
/**
 * 队列组 dispatch_group_wait
 */
- (void)groupWait {
    NSLog(@"currentThread---%@",[NSThread currentThread]);  // 打印当前线程
    NSLog(@"group---begin");

    dispatch_group_t group =  dispatch_group_create();

    dispatch_group_async(group, dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        // 追加任务 1
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"1---%@",[NSThread currentThread]);      // 打印当前线程
    });

    dispatch_group_async(group, dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        // 追加任务 2
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"2---%@",[NSThread currentThread]);      // 打印当前线程
    });

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

    NSLog(@"group---end");

}
```

> 输出结果：
> 2019-08-08 15:09:12.441729+0800 YSC-GCD-demo[17844:4299926] currentThread---<NSThread: 0x6000013e2940>{number = 1, name = main}
> 2019-08-08 15:09:12.441870+0800 YSC-GCD-demo[17844:4299926] group---begin
> 2019-08-08 15:09:14.445790+0800 YSC-GCD-demo[17844:4300046] 2---<NSThread: 0x60001389780>{number = 4, name = (null)}
> 2019-08-08 15:09:14.445760+0800 YSC-GCD-demo[17844:4300043] 1---<NSThread: 0x60001381880>{number = 3, name = (null)}
> 2019-08-08 15:09:14.446039+0800 YSC-GCD-demo[17844:4299926] group---end

从 `dispatch_group_wait` 相关代码运行输出结果可以看出：当所有任务执行完成之后，才执行 `dispatch_group_wait` 之后的操作。但是，使用 `dispatch_group_wait` 会阻塞当前线程。

6.5.3 dispatch_group_enter、dispatch_group_leave

- `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` 中的任务。

```
/**
 * 队列组 dispatch_group_enter、dispatch_group_leave
 */
- (void)groupEnterAndLeave {
    NSLog(@"currentThread---%@",[NSThread currentThread]);  // 打印当前线程
    NSLog(@"group---begin");

    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, ^{
        // 追加任务 1
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"1---%@",[NSThread currentThread]);      // 打印当前线程

        dispatch_group_leave(group);
    });

    dispatch_group_enter(group);
    dispatch_async(queue, ^{
        // 追加任务 2
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"2---%@",[NSThread currentThread]);      // 打印当前线程

        dispatch_group_leave(group);
    });

    dispatch_group_notify(group, dispatch_get_main_queue(), ^{
        // 等前面的异步操作都执行完毕后，回到主线程.
```

```
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"3---%@",[NSThread currentThread]);      // 打印当前线程

        NSLog(@"group---end");
    });
}
```

输出结果：

2019-08-08 15:13:17.983283+0800 YSC-GCD-demo[17924:4314716] currentThread---<NSThread: 0x600001ee5380>{number = 1, name = main}

2019-08-08 15:13:17.983429+0800 YSC-GCD-demo[17924:4314716] group---begin

2019-08-08 15:13:19.988898+0800 YSC-GCD-demo[17924:4314816] 2---<NSThread: 0x600001e9ca00>{number = 3, name = (null)}

2019-08-08 15:13:19.988888+0800 YSC-GCD-demo[17924:4314808] 1---<NSThread: 0x600001e94100>{number = 4, name = (null)}

2019-08-08 15:13:21.990450+0800 YSC-GCD-demo[17924:4314716] 3---<NSThread: 0x600001ee5380>{number = 1, name = main}

2019-08-08 15:13:21.990711+0800 YSC-GCD-demo[17924:4314716] group---end

从 `dispatch_group_enter、dispatch_group_leave` 相关代码运行结果中可以看出：当所有任务执行完成之后，才执行 `dispatch_group_notify` 中的任务。这里的 `dispatch_group_enter` 、 `dispatch_group_leave` 组合，其实等同于 `dispatch_group_async` 。

6.6 GCD 信号量：dispatch_semaphore

GCD 中的信号量是指 **Dispatch Semaphore**，是持有计数的信号。类似于过高速路收费站的栏杆。可以通过时，打开栏杆，不可以通过时，关闭栏杆。在 **Dispatch Semaphore** 中，使用计数来完成这个功能，计数小于 0 时等待，不可通过。计数为 0 或大于 0 时，计数减 1 且不等待，可通过。

Dispatch Semaphore 提供了三个方法：

- `dispatch_semaphore_create`：创建一个 Semaphore 并初始化信号的总量
- `dispatch_semaphore_signal`：发送一个信号，让信号总量加 1
- `dispatch_semaphore_wait`：可以使总信号量减 1，信号总量小于 0 时就会一直等待（阻塞所在线程），否则就可以正常执行。

> 注意：信号量的使用前提是：想清楚你需要处理哪个线程等待（阻塞），又要哪个线程继续执行，然后使用信号量。

Dispatch Semaphore 在实际开发中主要用于：

- 保持线程同步，将异步执行任务转换为同步执行任务
- 保证线程安全，为线程加锁

6.6.1 Dispatch Semaphore 线程同步

我们在开发中，会遇到这样的需求：异步执行耗时任务，并使用异步执行的结果进行一些额外的操作。换句话说，相当于，将将异步执行任务转换为同步执行任务。比如说：AFNetworking 中 AFURLSessionManager.m 里面的 `tasksForKeyPath:` 方法。通过引入信号量的方式，等待异步执行任务结果，获取到 tasks，然后再返回该 tasks。

```
- (NSArray *)tasksForKeyPath:(NSString *)keyPath {
    __block NSArray *tasks = nil;
    dispatch_semaphore_t semaphore = dispatch_semaphore_create(0);
    [self.session getTasksWithCompletionHandler:^(NSArray *dataTasks, NSArray *uploadTasks, NSArr
        if ([keyPath isEqualToString:NSStringFromSelector(@selector(dataTasks))]) {
            tasks = dataTasks;
        } else if ([keyPath isEqualToString:NSStringFromSelector(@selector(uploadTasks))]) {
            tasks = uploadTasks;
        } else if ([keyPath isEqualToString:NSStringFromSelector(@selector(downloadTasks))]) {
            tasks = downloadTasks;
        } else if ([keyPath isEqualToString:NSStringFromSelector(@selector(tasks))]) {
            tasks = [@[dataTasks, uploadTasks, downloadTasks] valueForKeyPath:@"@unionOfArrays.se
        }

        dispatch_semaphore_signal(semaphore);
    }];

    dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);

    return tasks;
}
```

下面，我们来利用 Dispatch Semaphore 实现线程同步，将异步执行任务转换为同步执行任务。

```
/**
 * semaphore 线程同步
 */
- (void)semaphoreSync {

    NSLog(@"currentThread---%@",[NSThread currentThread]);  // 打印当前线程
    NSLog(@"semaphore---begin");

    dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_semaphore_t semaphore = dispatch_semaphore_create(0);

    __block int number = 0;
    dispatch_async(queue, ^{
        // 追加任务 1
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"1---%@",[NSThread currentThread]);      // 打印当前线程

        number = 100;

        dispatch_semaphore_signal(semaphore);
    });

    dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
    NSLog(@"semaphore---end,number = %zd",number);
}
```

> 输出结果：
>
> 2019-08-08 15:16:56.781543+0800 YSC-GCD-demo[17988:4325744] currentThread---<NSThread: 0x60000298d380>{number = 1, name = main}
>
> 2019-08-08 15:16:56.781698+0800 YSC-GCD-demo[17988:4325744]

```
semaphore---begin
2019-08-08 15:16:58.785232+0800 YSC-GCD-demo[17988:4325867] 1---
<NSThread: 0x6000029eba80>{number = 3, name = (null)}
2019-08-08 15:16:58.785432+0800 YSC-GCD-demo[17988:4325744]
semaphore---end,number = 100
```

从 Dispatch Semaphore 实现线程同步的代码可以看到：

- `semaphore---end` 是在执行完 `number = 100;` 之后才打印的。而且输出结果 number 为 100。这是因为 `异步执行` 不会做任何等待，可以继续执行任务。

  执行顺如下：

  1. semaphore 初始创建时计数为 0。
  2. `异步执行` 将 `任务 1` 追加到队列之后，不做等待，接着执行 `dispatch_semaphore_wait` 方法，semaphore 减 1，此时 `semaphore == -1`，当前线程进入等待状态。
  3. 然后，异步任务 1 开始执行。任务 1 执行到 `dispatch_semaphore_signal` 之后，总信号量加 1，此时 `semaphore == 0`，正在被阻塞的线程（主线程）恢复继续执行。
  4. 最后打印 `semaphore---end,number = 100`。

这样就实现了线程同步，将异步执行任务转换为同步执行任务。

6.6.2 Dispatch Semaphore 线程安全和线程同步（为线程加锁）

线程安全：如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。

若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作（更改变量），一般都需要考虑线程同步，否则的话就可能影响线程安全。

线程同步：可理解为线程 A 和 线程 B 一块配合，A 执行到一定程度时要依靠线程 B 的某个结果，于是停下来，示意 B 运行；B 依言执行，再将结果给 A；A 再继续操作。

举个简单例子就是：两个人在一起聊天。两个人不能同时说话，避免听不清(操作冲突)。等一个人说完(一个线程结束操作)，另一个再说(另一个线程再开始操作)。

下面，我们模拟火车票售卖的方式，实现 NSThread 线程安全和解决线程同步问题。

> 场景：总共有 50 张火车票，有两个售卖火车票的窗口，一个是北京火车票售卖窗口，另一个是上海火车票售卖窗口。两个窗口同时售卖火车票，卖完为止。

6.6.2.1 非线程安全（不使用 semaphore）

先来看看不考虑线程安全的代码：

```
/**
```

```
 * 非线程安全：不使用 semaphore
 * 初始化火车票数量、卖票窗口（非线程安全）、并开始卖票
 */
- (void)initTicketStatusNotSave {
    NSLog(@"currentThread---%@",[NSThread currentThread]);  // 打印当前线程
    NSLog(@"semaphore---begin");

    self.ticketSurplusCount = 50;

    // queue1 代表北京火车票售卖窗口
    dispatch_queue_t queue1 = dispatch_queue_create("net.bujige.testQueue1", DISPATCH_QUEUE_SERIA
    // queue2 代表上海火车票售卖窗口
    dispatch_queue_t queue2 = dispatch_queue_create("net.bujige.testQueue2", DISPATCH_QUEUE_SERIA

    __weak typeof(self) weakSelf = self;
    dispatch_async(queue1, ^{
        [weakSelf saleTicketNotSafe];
    });

    dispatch_async(queue2, ^{
        [weakSelf saleTicketNotSafe];
    });
}

/**
 * 售卖火车票（非线程安全）
 */
- (void)saleTicketNotSafe {
    while (1) {

        if (self.ticketSurplusCount > 0) {  // 如果还有票，继续售卖
            self.ticketSurplusCount--;
            NSLog(@"%@", [NSString stringWithFormat:@"剩余票数：%d 窗口：%@", self.ticketSurplusCo
            [NSThread sleepForTimeInterval:0.2];
        } else { // 如果已卖完，关闭售票窗口
            NSLog(@"所有火车票均已售完");
            break;
        }

    }
}

```

输出结果（部分）：

2019-08-08 15:21:39.772655+0800 YSC-GCD-demo[18071:4340555]
currentThread---<NSThread: 0x6000015a2f40>{number = 1, name = main}
2019-08-08 15:21:39.772790+0800 YSC-GCD-demo[18071:4340555]
semaphore---begin
2019-08-08 15:21:39.773101+0800 YSC-GCD-demo[18071:4340604] 剩余票数：
48 窗口：<NSThread: 0x6000015cc600>{number = 4, name = (null)}
2019-08-08 15:21:39.773115+0800 YSC-GCD-demo[18071:4340605] 剩余票数：
49 窗口：<NSThread: 0x6000015f8600>{number = 3, name = (null)}
2019-08-08 15:21:39.975041+0800 YSC-GCD-demo[18071:4340605] 剩余票数：
47 窗口：<NSThread: 0x6000015f8600>{number = 3, name = (null)}
2019-08-08 15:21:39.975037+0800 YSC-GCD-demo[18071:4340604] 剩余票数：
47 窗口：<NSThread: 0x6000015cc600>{number = 4, name = (null)}
2019-08-08 15:21:40.176567+0800 YSC-GCD-demo[18071:4340604] 剩余票数：
46 窗口：<NSThread: 0x6000015cc600>{number = 4, name = (null)}
...

可以看到在不考虑线程安全，不使用 semaphore 的情况下，得到票数是错乱的，这样显然不符合我们的需求，所以我们需要考虑线程安全问题。

6.6.2.2 线程安全（使用 semaphore 加锁）

考虑线程安全的代码：

```
/**
 * 线程安全：使用 semaphore 加锁
 * 初始化火车票数量、卖票窗口（线程安全）、并开始卖票
 */
- (void)initTicketStatusSave {
    NSLog(@"currentThread---%@",[NSThread currentThread]);  // 打印当前线程
    NSLog(@"semaphore---begin");

    semaphoreLock = dispatch_semaphore_create(1);

    self.ticketSurplusCount = 50;

    // queue1 代表北京火车票售卖窗口
    dispatch_queue_t queue1 = dispatch_queue_create("net.bujige.testQueue1", DISPATCH_QUEUE_SERIA
    // queue2 代表上海火车票售卖窗口
    dispatch_queue_t queue2 = dispatch_queue_create("net.bujige.testQueue2", DISPATCH_QUEUE_SERIA

    __weak typeof(self) weakSelf = self;
    dispatch_async(queue1, ^{
        [weakSelf saleTicketSafe];
    });

    dispatch_async(queue2, ^{
        [weakSelf saleTicketSafe];
    });
}

/**
 * 售卖火车票（线程安全）
 */
- (void)saleTicketSafe {
    while (1) {
        // 相当于加锁
        dispatch_semaphore_wait(semaphoreLock, DISPATCH_TIME_FOREVER);

        if (self.ticketSurplusCount > 0) {  // 如果还有票，继续售卖
            self.ticketSurplusCount--;
            NSLog(@"%@", [NSString stringWithFormat:@"剩余票数：%d 窗口：%@", self.ticketSurplusCo
            [NSThread sleepForTimeInterval:0.2];
        } else { // 如果已卖完，关闭售票窗口
            NSLog(@"所有火车票均已售完");

            // 相当于解锁
            dispatch_semaphore_signal(semaphoreLock);
            break;
        }

        // 相当于解锁
        dispatch_semaphore_signal(semaphoreLock);
    }
}
```

> 输出结果为：
>
> 2019-08-08 15:23:58.819891+0800 YSC-GCD-demo[18116:4348091]
>
> currentThread---<NSThread: 0x600000681380>{number = 1, name = main}
>
> 2019-08-08 15:23:58.820041+0800 YSC-GCD-demo[18116:4348091]

```
semaphore---begin
2019-08-08 15:23:58.820305+0800 YSC-GCD-demo[18116:4348159] 剩余票数：
49 窗口：<NSThread: 0x6000006ede80>{number = 3, name = (null)}
2019-08-08 15:23:59.022165+0800 YSC-GCD-demo[18116:4348157] 剩余票数：
48 窗口：<NSThread: 0x6000006e4b40>{number = 4, name = (null)}
2019-08-08 15:23:59.225299+0800 YSC-GCD-demo[18116:4348159] 剩余票数：
47 窗口：<NSThread: 0x6000006ede80>{number = 3, name = (null)}
...
2019-08-08 15:24:08.355977+0800 YSC-GCD-demo[18116:4348157] 剩余票数：
2 窗口：<NSThread: 0x6000006e4b40>{number = 4, name = (null)}
2019-08-08 15:24:08.559201+0800 YSC-GCD-demo[18116:4348159] 剩余票数：
1 窗口：<NSThread: 0x6000006ede80>{number = 3, name = (null)}
2019-08-08 15:24:08.759630+0800 YSC-GCD-demo[18116:4348157] 剩余票数：
0 窗口：<NSThread: 0x6000006e4b40>{number = 4, name = (null)}
2019-08-08 15:24:08.965100+0800 YSC-GCD-demo[18116:4348159] 所有火车票
均已售完
2019-08-08 15:24:08.965440+0800 YSC-GCD-demo[18116:4348157] 所有火车票
均已售完
```

可以看出，在考虑了线程安全的情况下，使用 `dispatch_semaphore` 机制之后，得到的票数是正确的，没有出现混乱的情况。我们也就解决了多个线程同步的问题。

参考资料：

- 书籍：『Objective-C 高级编程 iOS 与 OS X 多线程和内存管理』
- 博文：iOS GCD 之 dispatch_semaphore（信号量）

---

iOS 多线程详尽总结系列文章：

- iOS多线程：『pthread、NSThread』详尽总结
- iOS多线程：『GCD』详尽总结
- iOS多线程：『NSOperation』详尽总结
- iOS多线程：『RunLoop』详尽总结
