iOS Swift GCD 开发教程

本教程由 Sinking Soul 出品

本教程将带你详细了解 GCD 的概念和用法，通过文中的代码示例和附带的 Github 示例工程，可以进一步加深对这些概念的体会。附带的示例工程是一个完整可运行的 App 项目：DispatchQueueTest，项目地址点此处。

GCD 全称是 `Grand Central Dispatch`，翻译过来就是大规模中央调度。根据官方文档，它的作用是：“通过向系统管理的调度队列中提交任务，在多核硬件上同时执行代码。”。它提供了一套机制，让你可以充分利用硬件的多核性能，并且让你不用再调用哪些繁琐的底层线程 API，编写易于理解和修改的代码。

1. 队列和任务的概念

GCD 的核心就是为了解决如何让程序有序、高效的运行，由此衍生出队列等概念和一系列的方法。为了弄清楚这些概念，我们先来看看程序执行存在哪些问题需要解决。

理解任务

在 GCD 中把程序执行时做的事情都当成任务，一段代码、一个 API 调用、一个方法、函数、闭包等，都是任务，一个应用就是由很多任务组成的。任务的执行需要时间和相应的顺序，耗时有长短，顺序有先后，任务只有按照正确的时间和顺序进行编排，应用才能按照你的预期运行。我们举音乐播放的例子来看看关于任务有哪些需求。

1. 默认情况下，程序是按代码顺序执行的，但我们有时希望应用能同时做多件事情，比如同时下载歌词和音乐。这就有了第一个需求：让多个任务同时进行。
2. 对于下载这个任务，可以一次下载多首音乐，各下各的，不需要互相等待；然而当全部下载完了播放时，通常是一首接一首的播放，播放一首音乐这个任务是需要等待前面的播放任务完成了才能进行。这就有了第二个需求：有的任务需要等待它完成了才能进行下一个任务，有的任务不需要等待它完成。
3. 如果一首音乐还没下载，我们就点了播放键，我们看看需要做哪些事情：它需要把歌词、音乐分别下载了，等他们都下载完了，告诉应用你可以播放了，然后应用把歌词、音乐同时播放。那我们怎么知道歌词、音乐都下载完了呢？这就有了第三个需求：如果有个东西能把几个任务捆绑到一起就好了，当整个包都完成了再通知我。
4. 还是下载，如果我们勾选了一堆的音乐要下载，中间我想暂停一下，过一会再让它继续，这就要求这一系列的下载任务要可以暂停和继续。
5. 一般下载工具都可以设置同时最大下载数，这就要求有一个方法可以控制同时进行的任务数。
6. 很多播放器会有一个功能：播放 20 分钟后就停止，非常适合睡觉前用。这个时候需要有个任务，在 20 分钟后把音乐关了。延迟执行任务就是它需要的特性。

以上列举了 6 个经典的任务执行需要的特性，在 GCD 中分别提供了以下方法来支持它们：

1. 串行队列、并行队列
2. 同步任务、异步任务
3. 任务组、栅栏任务
4. 挂起、唤醒队列
5. 信号量
6. 延迟加入队列

下面我们先从队列开始分析。

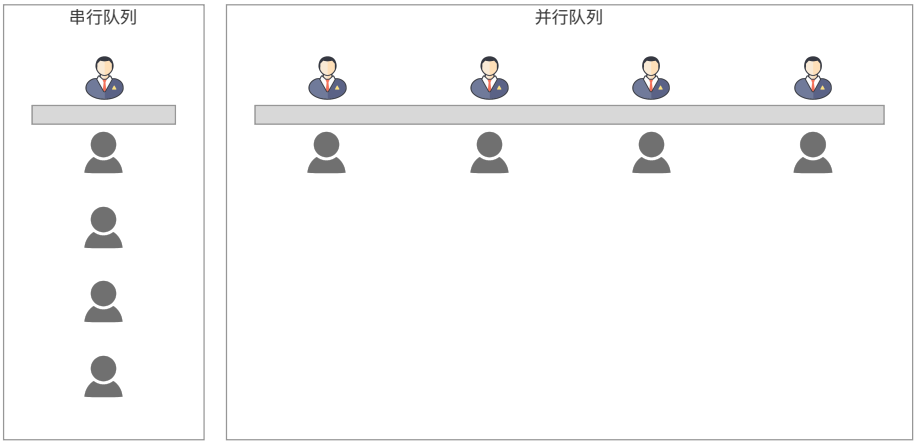
2. 创建队列

在系统底层，程序是运行在线程之中的，如果我们直接在线程层面进行操作，我们就需要告诉程序它应该运行在哪个线程、何时开始、何时结束等，这一列的操作都非常繁琐，而且很容易出错。为了简化线程的操作，GCD 封装了队列的概念。

可以把队列想象成办事窗口，有些类型窗口一次只能受理一个任务，通常只有一个办事员（线程），所有任务按进入的先后顺序来办理，而且不允许插队(阻塞线程)，这是串行队列。

有些类型窗口一次可以受理多个任务，多个任务可以同时办理，通常有多个办事员（线程），而且同一个任务在办理过程中允许被插队(阻塞线程)，这是并行队列。

在后面我们会详细讨论队列的特性。



创建队列非常的简单。

串行队列

系统为串行队列一般只分配一个线程（也有特例，下一章任务特性部分有解释），队列中如果有任务正在执行时，是不允许队列中的其他任务插队的（即暂停当前任务，转而执行其他任务），这个特性也可以理解为：串行队列中执行任务的线程不允许被**当前队列中的任务**阻塞（此时会死锁），但可以被别的队列任务阻塞。

创建时指定 `label` 便于调试，一般使用 `Bundle Identifier` 类似的命名方式：

```
let queue = DispatchQueue(label: "com.xxx.xxx.queueName")
```

并行队列

系统会为并行队列至少分配一个线程，线程允许被任何队列的任务阻塞。

```
let queue = DispatchQueue(label: "com.xxx.xxx.queueName", attributes: .concurrent)
```

其实在我们手动创建队列之前，系统已经帮我们创建好了 6 条队列，1 条系统主队列（串行），5 条全局并发队列（不同优先级），它们是我们创建的所有队列的最终目标队列（后面会解释），这 6 个队列负责所有队列的线程调度。

系统主队列

主队列是一个串行队列，它主要处理 UI 相关任务，也可以处理其他类型任务，但为了性能考虑，尽量让主队列执行 UI 相关或少量不耗时间和资源的操作。它通过类属性获取：

```
let mainQueue = DispatchQueue.main
```

系统全局并发队列

全局并发队列，存在 5 个不同的 QoS 级别，可以使用默认优先级，也可以单独指定：

```
let globalQueue = DispatchQueue.global() //  qos: .default
let globalQueue = DispatchQueue.global(qos: .background) // 后台运行级别
```

3. 添加队列任务

有些任务我们必须等待它的执行结果才能进行下一步，这种执行任务的方式称为同步，简称同步任务；有些任务只要把它放入队列就可以不管它了，可以继续执行其他任务，按这种方式执行的任务，称为异步任务。

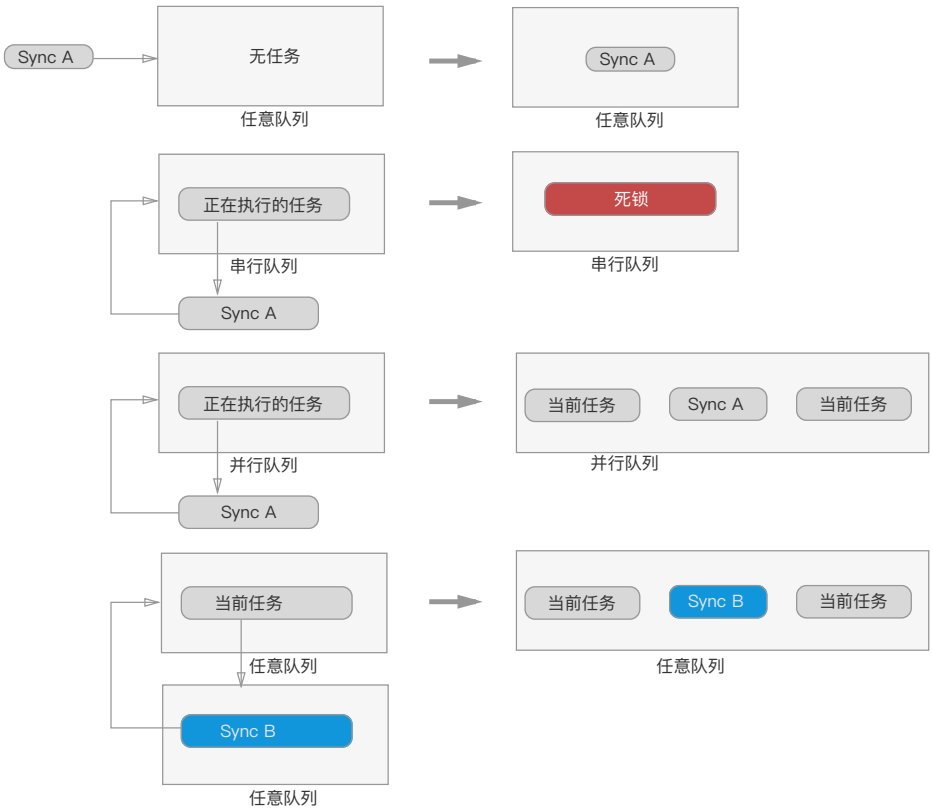
同步任务

特性：**任务一经提交就会阻塞当前线程（当前线程可以理解为下方代码示例中执行 `sync` 方法所在的线程 `thread0`），并请求队列立即安排其执行，执行任务的线程 `thread1` 默认等于 `thread0`，即同步任务直接在当前线程运行，任务完成后恢复线程原任务。**

任务提交方式如下：

```
// current thread – thread0
queue.sync {
    // current thread – thread1 == thread0
    // do something
}
```

我们分别根据下图中的 4 种情况举 4 个例子，来说明同步任务的特性。



1. 队列中如果没有任务在执行，那么提交同步任务后，将立即执行该任务，并阻塞线程 `thread0`，任务完成后再恢复线程 `thread0` 中被阻塞的任务。
2. 如果串行队列中有任务在执行，如果该任务又向该队列提交了一个同步任务，将会立即发生死锁。
3. 如果并行队列中有任务在执行，如果该任务又向该队列提交了一个同步任务，那么当前线程会转而执行新的同步任务，结束后再回到原任务。
4. 如果队列中有任务在执行，如果该任务向另一个队列提交了一个同步任务，那么当前线程会转而执行新的同步任务，结束后再回到原任务。

看例子前先介绍两个辅助方法：

1.打印当前线程，使用 `Thread.current` 属性：

```
/// 打印当前线程
func printCurrentThread(with des: String, _ terminator: String = "") {
    print("\(des) at thread: \(Thread.current), this is \(Thread.isMainThread ? "" : "not ")main thread\(terminator)")
}
```

2.测试任务是否在指定队列中，通过给队列设置一个标识，使用 `DispatchQueue.getSpecific` 方法来获取这个标识，如果能获取到，说明任务在该队列中：

```
/// 队列类型
enum DispatchTaskType: String {
    case serial
    case concurrent
    case main
    case global
}
```

```
// 定义队列
let serialQueue = DispatchQueue(label: "com.sinkingsoul.DispatchQueueTest.serialQueue")
let concurrentQueue = DispatchQueue(
    label: "com.sinkingsoul.DispatchQueueTest.concurrentQueue",
    attributes: .concurrent)
let mainQueue = DispatchQueue.main
let globalQueue = DispatchQueue.global()

// 定义队列 key
let serialQueueKey = DispatchSpecificKey<String>()
let concurrentQueueKey = DispatchSpecificKey<String>()
let mainQueueKey = DispatchSpecificKey<String>()
let globalQueueKey = DispatchSpecificKey<String>()

// 初始化队列 key
init() {
    serialQueue.setSpecific(key: serialQueueKey, value: DispatchTaskType.serial.rawValue)
    concurrentQueue.setSpecific(key: concurrentQueueKey, value:
DispatchTaskType.concurrent.rawValue)
    mainQueue.setSpecific(key: mainQueueKey, value: DispatchTaskType.main.rawValue)
    globalQueue.setSpecific(key: globalQueueKey, value: DispatchTaskType.global.rawValue)
}

/// 测试任务是否在指定队列中
func testIsTaskInQueue(_ queueType: DispatchTaskType, key: DispatchSpecificKey<String>) {
    let value = DispatchQueue.getSpecific(key: key)
    let opnValue: String? = queueType.rawValue
    print("Is task in \(queueType.rawValue) queue: \(value == opnValue)")
}
```

下面我们看看这 4 个例子：

代码示例

> 本章对应的代码见示例工程中 `QueueTestListTableViewController+createQueueWithTask.swift`，`CreateQueueWithTask.swift`。

示例 3.1：串行队列中新增同步任务

```
/// 串行队列中新增同步任务
func testSyncTaskInSerialQueue() {
    self.printCurrentThread(with: "start test")
    serialQueue.sync {
        print("\nserialQueue sync task--->")
        self.printCurrentThread(with: "serialQueue sync task")
        self.testIsTaskInQueue(.serial, key: serialQueueKey)
        print("--->serialQueue sync task\n")
    }
    self.printCurrentThread(with: "end test")
}
```

执行结果，任务是在主线程中执行的，结束后又回到了主线程，可以理解为这个同步任务把主线程阻塞了，让自己优先插队执行:

```
start test at thread: <NSThread: 0x1c4260900>{number = 1, name = main}, this is main thread

serialQueue sync task--->
serialQueue sync task at thread: <NSThread: 0x1c4260900>{number = 1, name = main}, this is main thread
Is task in serial queue: true
--->serialQueue sync task

end test at thread: <NSThread: 0x1c4260900>{number = 1, name = main}, this is main thread
```

示例 3.2 串行队列任务中嵌套本队列的同步任务

```
/// 串行队列任务中嵌套本队列的同步任务
func testSyncTaskNestedInSameSerialQueue() {
    printCurrentThread(with: "start test")
    serialQueue.async {
        print("\nserialQueue async task--->")
        self.printCurrentThread(with: "serialQueue async task")
        self.testIsTaskInQueue(.serial, key: self.serialQueueKey)

        self.serialQueue.sync {
            print("\nserialQueue sync task--->")
            self.printCurrentThread(with: "serialQueue sync task")
            self.testIsTaskInQueue(.serial, key: self.serialQueueKey)
            print("--->serialQueue sync task\n")
        } // Thread 9: EXC_BREAKPOINT (code=1, subcode=0x101613ba4)

        print("--->serialQueue async task\n")
    }
    printCurrentThread(with: "end test")
}
```

执行结果，执行到嵌套任务时程序就崩溃了，这是死锁导致的。其中有个有意思的现象，这里串行队列的第一个任务运行在非主线程上，在异步任务部分会解释。这里死锁是由两个因素导致：串行队列、同步任务，回顾一下串行队列的特性就好解释了：串行队列中执行任务的线程不允许被**当前队列中的任务**阻塞。下个例子我们试试：并行队列 + 同步任务，看看会不会导致死锁。

```
start test at thread: <NSThread: 0x1c006db80>{number = 1, name = main}, this is main thread

serialQueue async task--->
end test at thread: <NSThread: 0x1c006db80>{number = 1, name = main}, this is main thread
serialQueue async task at thread: <NSThread: 0x1c4466340>{number = 3, name = (null)}, this is not main thread
Is task in serial queue: true

(lldb)
```

示例 3.3 并行队列任务中嵌套本队列的同步任务

```
/// 并行队列任务中嵌套本队列的同步任务
func testSyncTaskNestedInSameConcurrentQueue() {
    printCurrentThread(with: "start test")
    concurrentQueue.async {
        print("\nconcurrentQueue async task--->")
        self.printCurrentThread(with: "concurrentQueue async task")
        self.testIsTaskInQueue(.concurrent, key: self.concurrentQueueKey)

        self.concurrentQueue.sync {
            print("\nconcurrentQueue sync task--->")
            self.printCurrentThread(with: "concurrentQueue sync task")
            self.testIsTaskInQueue(.concurrent, key: self.concurrentQueueKey)
            print("--->concurrentQueue sync task\n")
        }

        print("--->concurrentQueue async task\n")
    }
    printCurrentThread(with: "end test")
}
```

执行结果，嵌套的同步任务执行的非常顺利，而且印证了同步任务的另一个特性：同步任务直接在当前线程运行。

```
start test at thread: <NSThread: 0x1c4263a40>{number = 1, name = main}, this is main thread

concurrentQueue async task--->
end test at thread: <NSThread: 0x1c4263a40>{number = 1, name = main}, this is main thread
concurrentQueue async task at thread: <NSThread: 0x1c426cd80>{number = 3, name = (null)}, this is not main thread
Is task in concurrent queue: true

concurrentQueue sync task--->
concurrentQueue sync task at thread: <NSThread: 0x1c426cd80>{number = 3, name = (null)}, this is not main thread
Is task in concurrent queue: true
```

```
---->concurrentQueue sync task

---->concurrentQueue async task
```

示例 3.4：串行队列中嵌套其他队列的同步任务

```
/// 串行队列中嵌套其他队列的同步任务
func testSyncTaskNestedInOtherSerialQueue() {
    // 创新另一个串行队列
    let serialQueue2 = DispatchQueue(
        label: "com.sinkingsoul.DispatchQueueTest.serialQueue2")
    let serialQueueKey2 = DispatchSpecificKey<String>()
    serialQueue2.setSpecific(key: serialQueueKey2, value: "serial2")

    self.printCurrentThread(with: "start test")
    serialQueue.sync {
        print("\nserialQueue sync task--->")
        self.printCurrentThread(with: "nserialQueue sync task")
        self.testIsTaskInQueue(.serial, key: self.serialQueueKey)

        serialQueue2.sync {
            print("\nserialQueue2 sync task--->")
            self.printCurrentThread(with: "serialQueue2 sync task")
            self.testIsTaskInQueue(.serial, key: self.serialQueueKey)

            let value = DispatchQueue.getSpecific(key: serialQueueKey2)
            let opnValue: String? = "serial2"
            print("Is task in serialQueue2: \(value == opnValue)")
            print("--->serialQueue2 sync task\n")
        }

        print("--->serialQueue sync task\n")
    }
}
```

执行结果，串行队列嵌套的同步任务执行成功了，和前面的例子不一样啊。是的，因为这里嵌套的是另一个队列的任务，虽然它们都运行在同一个线程上，一个串行队列可以对另一个串行队列视而不见。不同队列复用线程这是系统级的队列作出的优化，但是在同一个串行队列内部，任务一定都是按顺序执行的，这是自定义队列的最本质作用。

```
start test at thread: <NSThread: 0x1c4263a40>{number = 1, name = main}, this is main thread

serialQueue sync task--->
nserialQueue sync task at thread: <NSThread: 0x1c4263a40>{number = 1, name = main}, this is main thread
Is task in serial queue: true

serialQueue2 sync task--->
serialQueue2 sync task at thread: <NSThread: 0x1c4263a40>{number = 1, name = main}, this is main thread
Is task in serial queue: false
```

```
Is task in serialQueue2: true
--->serialQueue2 sync task

--->serialQueue sync task
```

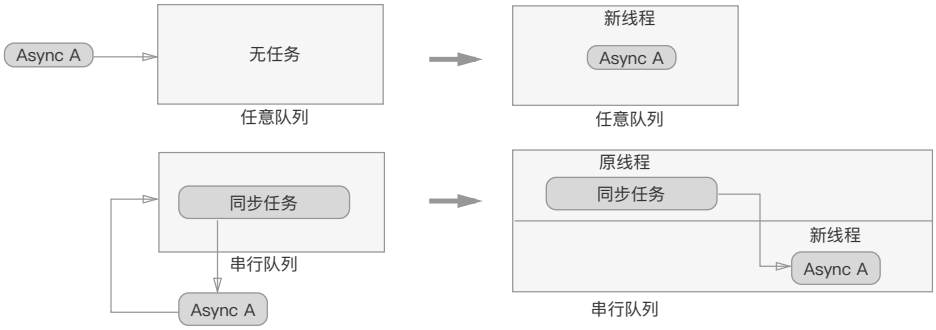
异步任务

特性：**任务提交后不会阻塞当前线程，会由队列安排另一个线程执行。**

任务提交方式如下：

```
// current thread - thread0
queue.async {
    // current thread - thread1 != thread0
    // do something
}
```

我们分别根据下图举 3 个例子，来说明异步任务的特性。



下面我们看看这 3 个例子：

代码示例

示例3.5：并行队列中新增异步任务

```
/// 并行队列中新增异步任务
func testAsyncTaskInConcurrentQueue() {
    printCurrentThread(with: "start test")
    concurrentQueue.async {
        print("\nconcurrentQueue async task--->")
        self.printCurrentThread(with: "concurrentQueue async task")
        self.testIsTaskInQueue(.concurrent, key: self.concurrentQueueKey)
        print("--->concurrentQueue async task\n")
    }
    printCurrentThread(with: "end test")
}
```

执行结果，执行异步任务时新开了一个线程。

```
start test at thread: <NSThread: 0x1c0078080>{number = 1, name = main}, this is main thread
end test at thread: <NSThread: 0x1c0078080>{number = 1, name = main}, this is main thread

concurrentQueue async task--->
concurrentQueue async task at thread: <NSThread: 0x1c04799c0>{number = 3, name = (null)}, this is not main thread
Is task in concurrent queue: true
--->concurrentQueue async task
```

示例3.6：串行队列中新增异步任务

```
/// 串行队列中新增异步任务
func testAsyncTaskInSerialQueue() {
    printCurrentThread(with: "start test")
    serialQueue.async {
        print("\nserialQueue async task--->")
        self.printCurrentThread(with: "serialQueue async task")
        self.testIsTaskInQueue(.serial, key: self.serialQueueKey)
        print("--->serialQueue async task\n")
    }
    printCurrentThread(with: "end test")
}
```

执行结果，同样新开了一个线程。

```
start test at thread: <NSThread: 0x1c0078080>{number = 1, name = main}, this is main thread
end test at thread: <NSThread: 0x1c0078080>{number = 1, name = main}, this is main thread

serialQueue async task--->
serialQueue async task at thread: <NSThread: 0x1c4473740>{number = 4, name = (null)}, this is not main thread
Is task in serial queue: true
--->serialQueue async task
```

示例3.7：串行队列任务中嵌套本队列的异步任务

```
/// 串行队列任务中嵌套本队列的异步任务
func testAsyncTaskNestedInSameSerialQueue() {
    printCurrentThread(with: "start test")
    serialQueue.sync {
        print("\nserialQueue sync task--->")
        self.printCurrentThread(with: "serialQueue sync task")
        self.testIsTaskInQueue(.serial, key: self.serialQueueKey)

        self.serialQueue.async {
            print("\nserialQueue async task--->")
            self.printCurrentThread(with: "serialQueue async task")
            self.testIsTaskInQueue(.serial, key: self.serialQueueKey)
```

```
            print("--->serialQueue async task\n")
        }

        print("--->serialQueue sync task\n")
    }
    printCurrentThread(with: "end test")
}
```

执行结果，这个例子再一次刷新了对串行队列的认识：**串行队列并不是只能运行一个线程**。第一层的同步任务运行在主线程上，第二层的异步任务运行在其他线程上，但它们在时间片上是分开的。这里再严格定义一下：**串行队列同一时间只会运行一个线程，只有碰到异步任务时，才会使用不同于当前的线程，但都是按时间顺序执行，只有前一个任务完成了，才会执行下一个任务。**

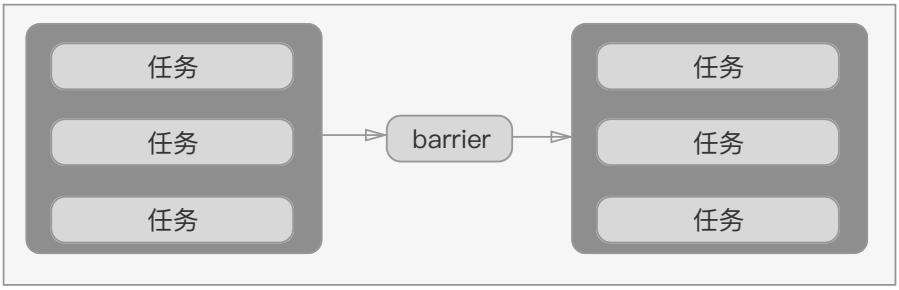
```
start test at thread: <NSThread: 0x1c0078080>{number = 1, name = main}, this is main thread

serialQueue sync task--->
serialQueue sync task at thread: <NSThread: 0x1c0078080>{number = 1, name = main}, this is main thread
Is task in serial queue: true
--->serialQueue sync task

serialQueue async task--->
end test at thread: <NSThread: 0x1c0078080>{number = 1, name = main}, this is main thread
serialQueue async task at thread: <NSThread: 0x1c4473a40>{number = 5, name = (null)}, this is not main thread
Is task in serial queue: true
--->serialQueue async task
```

这里我们总结一下队列和任务的特性：

- 串行队列同一时间只会使用同一线程、运行同一任务，并严格按照任务顺序执行。
- 并行队列同一时间可以使用多个线程、运行多个任务，执行顺序不分先后。
- 同步任务会阻塞当前线程，并在当前线程执行。
- 异步任务不会阻塞当前线程，并在与当前线程不同的线程执行。
- 如何避免死锁：不要在串行或主队列中嵌套执行同步任务。

下面介绍两个特殊的任务类型：栅栏任务、迭代任务。

栅栏任务



并行队列

栅栏任务的主要特性是可以对队列中的任务进行阻隔，执行栅栏任务时，它会先等待队列中已有的任务全部执行完成，然后它再执行，在它之后加入的任务也必须等栅栏任务执行完后才能执行。

这个特性更适合并行队列，而且对栅栏任务使用同步或异步方法效果都相同。

- 创建方式，先创建 `WorkItem`，标记为：`barrier`，再添加至队列中：

```
let queue = DispatchQueue(label: "queueName", attributes: .concurrent)
let task = DispatchWorkItem(flags: .barrier) {
    // do something
}
queue.async(execute: task)
queue.sync(execute: task) // 与 async 效果一样
```

下面看看栅栏任务的例子：

代码示例

示例3.8：并行队列中执行栅栏任务

```
/// 栅栏任务
func barrierTask() {
    let queue = concurrentQueue
    let barrierTask = DispatchWorkItem(flags: .barrier) {
        print("\nbarrierTask--->")
        self.printCurrentThread(with: "barrierTask")
        print("--->barrierTask\n")
    }

    printCurrentThread(with: "start test")

    queue.async {
        print("\nasync task1--->")
        self.printCurrentThread(with: "async task1")
        print("--->async task1\n")
    }
    queue.async {
        print("\nasync task2--->")
        self.printCurrentThread(with: "async task2")
        print("--->async task2\n")
```

```
    }
    queue.async {
        print("\nasync task3--->")
        self.printCurrentThread(with: "async task3")
        print("--->async task3\n")
    }

    queue.async(execute: barrierTask) // 栅栏任务

    queue.async {
        print("\nasync task4--->")
        self.printCurrentThread(with: "async task4")
        print("--->async task4\n")
    }
    queue.async {
        print("\nasync task5--->")
        self.printCurrentThread(with: "async task5")
        print("--->async task5\n")
    }
    queue.async {
        print("\nasync task6--->")
        self.printCurrentThread(with: "async task6")
        print("--->async task6\n")
    }
    printCurrentThread(with: "end test")
}
```

执行结果，任务 1、2、3 都在栅栏任务前同时执行，任务 4、5、6 都在栅栏任务后同时执行：

```
start test at thread: <NSThread: 0x1c407e7c0>{number = 1, name = main}, this is main thread
end test at thread: <NSThread: 0x1c407e7c0>{number = 1, name = main}, this is main thread

async task1--->

async task2--->
async task2 at thread: <NSThread: 0x1c066c480>{number = 4, name = (null)}, this is not main thread
--->async task2

async task3--->
async task3 at thread: <NSThread: 0x1c066c480>{number = 4, name = (null)}, this is not main thread
--->async task3

async task1 at thread: <NSThread: 0x1c066c600>{number = 3, name = (null)}, this is not main thread
--->async task1

barrierTask--->
barrierTask at thread: <NSThread: 0x1c066c600>{number = 3, name = (null)}, this is not main thread
--->barrierTask

async task5--->
async task5 at thread: <NSThread: 0x1c066c480>{number = 4, name = (null)}, this is not main thread
--->async task5
```

```
async task6--->
async task6 at thread: <NSThread: 0x1c066c480>{number = 4, name = (null)}, this is not main thread
--->async task6

async task4--->
async task4 at thread: <NSThread: 0x1c066c600>{number = 3, name = (null)}, this is not main thread
--->async task4
```

迭代任务

并行队列利用多个线程执行任务，可以提高程序执行的效率。而迭代任务可以更高效地利用多核性能，它可以利用 CPU 当前所有可用线程进行计算（任务小也可能只用一个线程）。如果一个任务可以分解为多个相似但独立的子任务，那么迭代任务是提高性能最适合的选择。

使用 `concurrentPerform` 方法执行迭代任务，迭代任务的后续任务需要等待它执行完成才会继续。本方法类似于 Objc 中的 `dispatch_apply` 方法，创建方式如下：

```
DispatchQueue.concurrentPerform(iterations: 10) {(index) -> Void in // 10 为迭代次数，可修改。
    // do something
}
```

迭代任务可以单独执行，也可以放在指定的队列中：

```
let queue = DispatchQueue.global() // 全局并发队列
queue.async {
    DispatchQueue.concurrentPerform(iterations: 100) {(index) -> Void in
        // do something
    }
    //可以转至主线程执行其他任务
    DispatchQueue.main.async {
        // do something
    }
}
```

下面看看迭代任务的例子：

代码示例

示例3.9：迭代任务

本示例查找 1–10000 之间能被 13 整除的整数，我们直接使用 10000 次迭代对每个数进行判断，符合的通过异步方法写入到结果数组中：

```
/// 迭代任务
func concurrentPerformTask() {
    printCurrentThread(with: "start test")

    /// 判断一个数是否能被另一个数整除
    func isDividedExactlyBy(_ divisor: Int, with number: Int) -> Bool {
```

```
        return number % divisor == 0
    }

    let array = Array(1...100)
    var result: [Int] = []

    globalQueue.async {
        //通过concurrentPerform，循环变量数组
        print("concurrentPerform task start--->")
        DispatchQueue.concurrentPerform(iterations: 100) {(index) -> Void in
            if isDividedExactlyBy(13, with: array[index]) {
                self.printCurrentThread(with: "find a match: \(array[index])")
                self.mainQueue.async {
                    result.append(array[index])
                }
            }
        }
        print("--->concurrentPerform task over")
        //执行完毕，主线程更新结果。
        DispatchQueue.main.sync {
            print("back to main thread")
            print("result: find \(result.count) number - \(result)")
        }
    }
    printCurrentThread(with: "end test")
}
```

iPhone 7 Plus 执行结果，使用了 2 个线程，iPhone 8 的 CPU 有 6 个核心，据说可以同时开启，手头有的可以试一下：

```
start test at thread: <NSThread: 0x1c4076900>{number = 1, name = main}, this is main thread
end test at thread: <NSThread: 0x1c4076900>{number = 1, name = main}, this is main thread
concurrentPerform task start--->
find a match: 13 at thread: <NSThread: 0x1c0469bc0>{number = 3, name = (null)}, this is not main thread
find a match: 39 at thread: <NSThread: 0x1c0469bc0>{number = 3, name = (null)}, this is not main thread
find a match: 52 at thread: <NSThread: 0x1c0469bc0>{number = 3, name = (null)}, this is not main thread
find a match: 65 at thread: <NSThread: 0x1c0469bc0>{number = 3, name = (null)}, this is not main thread
find a match: 26 at thread: <NSThread: 0x1c0469cc0>{number = 4, name = (null)}, this is not main thread
find a match: 91 at thread: <NSThread: 0x1c0469cc0>{number = 4, name = (null)}, this is not main thread
find a match: 78 at thread: <NSThread: 0x1c0469bc0>{number = 3, name = (null)}, this is not main thread
--->concurrentPerform task over
back to main thread
result: find 7 number - [13, 39, 52, 65, 26, 91, 78]
```

Mac 上使用 Xcode 模拟器执行结果，使用了 4 个线程：

```
start test at thread: <NSThread: 0x604000070c40>{number = 1, name = main}, this is main thread
concurrentPerform task start--->
end test at thread: <NSThread: 0x604000070c40>{number = 1, name = main}, this is main thread
find a match: 26 at thread: <NSThread: 0x60400047b800>{number = 3, name = (null)}, this is not main
thread
find a match: 13 at thread: <NSThread: 0x60000046ec80>{number = 4, name = (null)}, this is not main
```

```
thread
find a match: 65 at thread: <NSThread: 0x60400047b800>{number = 3, name = (null)}, this is not main thread
find a match: 91 at thread: <NSThread: 0x60400047b800>{number = 3, name = (null)}, this is not main thread
find a match: 78 at thread: <NSThread: 0x60000046ec80>{number = 4, name = (null)}, this is not main thread
find a match: 39 at thread: <NSThread: 0x60000046ed80>{number = 5, name = (null)}, this is not main thread
find a match: 52 at thread: <NSThread: 0x604000475140>{number = 6, name = (null)}, this is not main thread
--->concurrentPerform task over
back to main thread
result: find 7 number - [26, 13, 65, 91, 78, 39, 52]
```

4. 队列详细属性

下面介绍一下在创建队列时，可以设置的一些更丰富的属性。创建队列的完整方法如下：

```
convenience init(label: String, qos: DispatchQoS = default, attributes: DispatchQueue.Attributes = default, autoreleaseFrequency: DispatchQueue.AutoreleaseFrequency = default, target: DispatchQueue? = default)
```

QoS

队列在执行上是有优先级的，更高的优先级可以享受更多的计算资源，从高到低包含以下几个等级：

- userInteractive
- userInitiated
- default
- utility
- background

Attributes

包含两个属性：

- concurrent：标识队列为并行队列
- initiallyInactive：标识运行队列中的任务需要动手触发（未添加此标识时，向队列中添加任务会自动运行），触发时通过 `queue.activate()` 方法。

AutoreleaseFrequency

这个属性表示 `autorelease pool` 的自动释放频率， `autorelease pool` 管理着任务对象的内存周期。

包含三个属性：

- inherit：继承目标队列的该属性
- workItem：跟随每个任务的执行周期进行自动创建和释放
- never：不会自动创建 `autorelease pool`，需要手动管理。

一般任务采用 `.workItem` 属性就够了，特殊任务如在任务内部大量重复创建对象的操作可选择 `.never` 属性手动创建 `autorelease pool`。

Target

这个属性设置的是一个队列的目标队列，即实际将该队列的任务放入指定队列中运行。目标队列最终约束了队列优先级等属性。

在程序中手动创建的队列，其实最后都指向系统自带的 `主队列` 或 `全局并发队列`。

你也许会问，为什么不直接将任务添加至系统队列中，而是自定义队列，因为这样的好处是可以将任务进行分组管理。如单独阻塞队列中的任务，而不是阻塞系统队列中的全部任务。如果阻塞了目标队列，所有指向它的原队列也将被阻塞。

在 Swift 3 及之后，对目标队列的设置进行了约束，只有两种情况可以显式地设置目标队列（原因参考）：

- 初始化方法中，指定目标队列。
- 初始化方法中，`attributes` 设定为 `initiallyInactive`，然后在队列执行 `activate()` 之前可以指定目标队列。

在其他地方都不能再改变目标队列。

关于目标队列的详细阐述，可以参考这篇文章：GCD Target Queues。

5. 延迟加入队列

有时候你并不需要立即将任务加入队列中运行，而是需要等待一段时间后再进入队列中，这时候可以使用 `asyncAfter` 方法。

例如，我们封装一个方法指定延迟加入队列的时间：

```
class AsyncAfter {
    /// 延迟执行闭包
    static func dispatch_later(_ time: TimeInterval, block: @escaping ()->()) {
        let t = DispatchTime.now() + time
        DispatchQueue.main.asyncAfter(deadline: t, execute: block)
    }
}

AsyncAfter.dispatch_later(2) {
    print("打个电话 at: \(Date())") // 将在 2 秒后执行
}
```

这里要注意延迟的时间是加入队列的时间，而不是开始执行任务的时间。

下面我们构造一个复杂一点的例子，我们封装一个方法，可以延迟执行任务，在计时结束前还可以取消任务或者将原任务替换为一个新任务。主要的思路是，将延迟后实际执行的任务代码进行替换，替换为空闭包则相当于取消了任务，或者替换为你想执行的其他任务：

```
class AsyncAfter {

    typealias ExchangableTask = (_ newDelayTime: TimeInterval?,
        _ anotherTask:@escaping (() -> ())
        ) -> Void

    /// 延迟执行一个任务，并支持在实际执行前替换为新的任务，并设定新的延迟时间。
    ///
    /// - Parameters:
    ///   - time: 延迟时间
    ///   - yourTask: 要执行的任务
    /// - Returns: 可替换原任务的闭包
    static func delay(_ time: TimeInterval, yourTask: @escaping ()->()) -> ExchangableTask {
        var exchangingTask: (() -> ())? // 备用替代任务
        var newDelayTime: TimeInterval? // 新的延迟时间

        let finalClosure = { () -> Void in
            if exchangingTask == nil {
                DispatchQueue.main.async(execute: yourTask)
            } else {
                if newDelayTime == nil {
                    DispatchQueue.main.async {
                        print("任务已更改，现在是：\(Date())")
                        exchangingTask!()
                    }
                }
                print("原任务取消了，现在是：\(Date())")
            }
        }

        dispatch_later(time) { finalClosure() }

        let exchangableTask: ExchangableTask =
        { delayTime, anotherTask in
            exchangingTask = anotherTask
            newDelayTime = delayTime

            if delayTime != nil {
                self.dispatch_later(delayTime!) {
                    anotherTask()
                    print("任务已更改，现在是：\(Date())")
                }
            }
        }
```

```
        return exchangableTask
    }
}
```

简单说明一下：

`delay` 方法接收两个参数，并返回一个闭包：

- TimeInterval：延迟时间
- @escaping () -> (): 要延迟执行的任务
- 返回：可替换原任务的闭包，我们去了一个别名：`ExchangableTask`

`ExchangableTask` 类型定义的闭包，接收一个新的延迟时间，和一个新的任务。

如果不执行返回的闭包，则在 `delay` 方法内部，通过 `dispatch_later` 方法会继续执行原任务。

如果执行了返回的 `ExchangableTask` 闭包，则会选择执行新的任务。

代码示例

> 本章对应的代码见示例工程中 `QueueTestListTableViewController+AsyncAfter.swift`，`AsyncAfter.swift`.

示例 5.1：延迟执行任务，在计时结束前取消。

```
extension QueueTestListTableViewController {
    /// 延迟任务，在执行前临时取消任务。
    @IBAction func ayncAfterCancelButtonTapped(_ sender: Any) {
        print("现在是：\(Date())")
        let task = AsyncAfter.delay(2) {
            print("打个电话 at: \(Date())")
        }

        // 立即取消任务
        task(0) {}
    }
}
```

根据我们封装的方法，只要提供一个空的闭包 `{}` 来替换原任务即相当于取消任务，同时还可以指定取消的时间，`task(0) {}` 表示立即取消，`task(nil) {}` 表示按原计划时间取消。

执行结果，可以看到任务立即就被替换了，但延迟 2 秒的任务还在，只是变成了一个空任务：

```
现在是：2018-03-14 01:38:20 +0000
任务已更改，现在是：2018-03-14 01:38:20 +0000
原任务取消了，现在是：2018-03-14 01:38:22 +0000
```

示例 5.2：延迟执行任务，在执行前临时替换为新的任务。

```
extension QueueTestListTableViewController {
    @IBAction func ayncAfterNewTaskButtonTapped(_ sender: Any) {
        print("现在是：\(Date())")
        let task = AsyncAfter.delay(2) {
            print("打个电话 at: \(Date())")
        }

        // 3 秒后改为执行一个新任务
        task(3) {
            print("吃了个披萨，现在是：\(Date())")
        }
    }
}
```

执行结果，可以看到 3 秒后执行了新的任务:

```
现在是：2018-03-14 03:14:08 +0000
原任务取消了，现在是：2018-03-14 03:14:10 +0000
吃了个披萨，现在是：2018-03-14 03:14:11 +0000
任务已更改，现在是：2018-03-14 03:14:11 +0000
```

6. 挂起和唤醒队列

GCD 提供了一套机制，可以挂起队列中尚未执行的任务，已经在执行的任务会继续执行完，后续还可以手动再唤醒队列。

这两个方法是属于 `DispatchObject` 对象的方法，而这个对象是 `DispatchQueue`、`DispatchGroup`、`DispatchSource`、`DispatchIO`、`DispatchSemaphore` 这几个类的父类，但这两个方法只有 `DispatchQueue`、`DispatchSource` 支持，调用时需注意。

挂起使用 `suspend()`，唤醒使用 `resume()`。对于队列，这两个方法调用时需配对，因为可以多次挂起，调用唤醒的次数应等于挂起的次数才能生效，唤醒的次数更多则会报错，所以使用时最好设置一个计数器，或者封装一个挂起、唤醒的方法，在方法内部进行检查。

而对于 `DispatchSource` 则有所不同，它必须先调用 `resume()` 才能接收消息，所以此时唤醒的数量等于挂起的数量加一。

下面通过例子看看实现：

```
/// 挂起、唤醒测试类
class SuspendAndResum {
    let createQueueWithTask = CreateQueueWithTask()
    var concurrentQueue: DispatchQueue {
        return createQueueWithTask.concurrentQueue
    }
    var suspendCount = 0 // 队列挂起的次数
```

```swift
// MARK: ----------队列方法------------

/// 挂起测试
func suspendQueue() {
    createQueueWithTask.printCurrentThread(with: "start test")
    concurrentQueue.async {
        self.createQueueWithTask.printCurrentThread(with: "concurrentQueue async task1")
    }
    concurrentQueue.async {
        self.createQueueWithTask.printCurrentThread(with: "concurrentQueue async task2")
    }

    // 通过栅栏挂起任务
    let barrierTask = DispatchWorkItem(flags: .barrier) {
        self.safeSuspend(self.concurrentQueue)
    }
    concurrentQueue.async(execute: barrierTask)

    concurrentQueue.async {
        self.createQueueWithTask.printCurrentThread(with: "concurrentQueue async task3")
    }

    concurrentQueue.async {
        self.createQueueWithTask.printCurrentThread(with: "concurrentQueue async task4")
    }
    concurrentQueue.async {
        self.createQueueWithTask.printCurrentThread(with: "concurrentQueue async task5")
    }
    createQueueWithTask.printCurrentThread(with: "end test")
}

/// 唤醒测试
func resumeQueue() {
    self.safeResume(self.concurrentQueue)
}

/// 安全的挂失操作
func safeSuspend(_ queue: DispatchQueue) {
    suspendCount += 1
    queue.suspend()
    print("任务挂起了")
}

/// 安全的唤醒操作
func safeResume(_ queue: DispatchQueue) {
    if suspendCount == 1 {
        queue.resume()
        suspendCount = 0
        print("任务唤醒了")
```

```
        } else if suspendCount < 1 {
            print("唤醒的次数过多")
        } else {
            queue.resume()
            suspendCount -= 1
            print("唤醒的次数不够，还需要 \(suspendCount) 次唤醒。")
        }
    }

}
```

通过按钮调用测试：

```
let suspendAndResum = SuspendAndResum()

extension QueueTestListTableViewController {
    // 挂起
    @IBAction func suspendButtonTapped(_ sender: Any) {
        suspendAndResum.suspendQueue()
    }

    // 唤醒
    @IBAction func resumeButtonTapped(_ sender: Any) {
        suspendAndResum.resumeQueue()
    }
}
```

挂起的执行结果：

```
start test at thread: <NSThread: 0x17d357d0>{number = 1, name = main}, this is main thread
end test at thread: <NSThread: 0x17d357d0>{number = 1, name = main}, this is main thread
concurrentQueue async task1 at thread: <NSThread: 0x17d5c560>{number = 3, name = (null)}, this is not main thread
concurrentQueue async task2 at thread: <NSThread: 0x17d5c560>{number = 3, name = (null)}, this is not main thread
任务挂起了
```

唤醒的执行结果：

```
任务唤醒了
concurrentQueue async task4 at thread: <NSThread: 0x17d5c560>{number = 3, name = (null)}, this is not main thread
concurrentQueue async task5 at thread: <NSThread: 0x17d5c560>{number = 3, name = (null)}, this is not main thread
concurrentQueue async task3 at thread: <NSThread: 0x17eae370>{number = 4, name = (null)}, this is not main thread
```

如果再按一次唤醒按钮，则会提示：

```
唤醒的次数过多
```

7. 任务组

任务组相当于一系列任务的松散集合，它可以来自相同或不同队列，扮演着组织者的角色。它可以通知外部队列，组内的任务是否都已完成。或者阻塞当前的线程，直到组内的任务都完成。所有适合组队执行的任务都可以使用任务组，且任务组更适合集合异步任务（如果都是同步任务，直接使用串行队列即可）。

创建任务组

创建的方式相当简单，无需任何参数：

```
let queueGroup = DispatchGroup()
```

将任务加入到任务组中

有两种方式加入任务组：

- 添加任务时指定任务组

```
let queue = DispatchQueue.global()
queue.async(group: queueGroup) {
    print("喝一杯牛奶")
}
```

- 使用 `Group.enter()` 、 `Group.leave()` 配对方法，标识任务加入任务组。

```
queueGroup.enter()
queue.async {
    print("吃一个苹果")
    queueGroup.leave()
}
```

两种加入方式在对任务处理的特性上是没有区别的，只是便利之处不同。如果任务所在的队列是自己创建或引用的系统队列，那么直接使用第一种方式直接加入即可。如果任务是由系统或第三方的 API 创建的，由于无法获取到对应的队列，只能使用第二种方式将任务加入组内，例如将 `URLSession` 的 `addDataTask` 方法加入任务组中:

```
extension URLSession {
    func addDataTask(to group: DispatchGroup,
                     with request: URLRequest,
                     completionHandler: @escaping (Data?, URLResponse?, Error?) -> Void)
        -> URLSessionDataTask {
            group.enter() // 进入任务组
            return dataTask(with: request) { (data, response, error) in
                completionHandler(data, response, error)
                group.leave() // 离开任务组
            }
    }
}
```

任务组通知

等待任务组中的任务全部完成后，可以统一对外发送通知，有两种方式：

- `group.notify` 方法，它可以在所有任务完成后通知指定队列并执行一个指定任务，这个通知的操作是异步的（意味着通知后续的代码不需要等待任务，可以继续执行）：

```
let group = DispatchGroup()

let queueBook = DispatchQueue(label: "book")
queueBook.async(group: group) {
    // do something 1
}
let queueVideo = DispatchQueue(label: "video")
queueVideo.async(group: group) {
    // do something 2
}

group.notify(queue: DispatchQueue.main) {
    print("all task done")
}

print("do something else.")

// 执行结果
// do something else.
// do something 1(任务 1、2 完成顺序不固定)
// do something 2
// all task done
```

- `group.wait` 方法，它会在所有任务完成后再执行当前线程中后续的代码，因此这个操作是起到阻塞的作用：

```
let group = DispatchGroup()

let queueBook = DispatchQueue(label: "book")
queueBook.async(group: group) {
    // do something 1
}
let queueVideo = DispatchQueue(label: "video")
queueVideo.async(group: group) {
    // do something 2
}

group.wait()

print("do something else.")

// 执行结果
// do something 1(任务 1、2 完成顺序不固定)
// do something 2
```

```
// do something else.
```

`wait` 方法中还可以指定具体的时间，它表示将等待不超过这个时间，如果任务组在指定时间之内完成则立即恢复当前线程，否则将等到时间结束时再恢复当前线程。

- 方式1，使用 `DispatchTime`，它表示一个时间间隔，精确到纳秒（1/1000,000,000 秒）：

```
let waitTime = DispatchTime.now() + 2.0 // 表示从当前时间开始后 2 秒，数字字面量也可以改为使用 TimeInterval 类型变量
group.wait(timeout: waitTime)
```

- 方式2，使用 `DispatchWallTime`，它表示当前的绝对时间戳，精确到微秒（1/1000,000 秒），通常使用字面量即可设置延时时间，也可以使用 `timespec` 结构体来设置一个精确的时间戳，具体参见附录章节的《时间相关的结构体说明 – DispatchWallTime》：

```
// 使用字面量设置
var wallTime = DispatchWallTime.now() + 2.0 // 表示从当前时间开始后 2 秒，数字字面量也可以改为使用 TimeInterval 类型变量
```

代码示例

> 本章对应的代码见示例工程中 `QueueTestListTableViewController+DispatchGroup.swift`，`DispatchGroup.swift`.

示例 7.1：创建任务组，并以常规方式添加任务。

示例中我们通过一个按钮触发，创建一个任务组、通过常规方式添加任务、任务完成时通知主线程。

```
extension QueueTestListTableViewController {
    @IBAction func creatTaskGroupButtonTapped(_ sender: Any) {
        let groupTest = DispatchGroupTest()
        let group = groupTest.creatAGroup()
        let queue = DispatchQueue.global()

        groupTest.addTaskNormally(to: group, in: queue)
        groupTest.notifyMainQueue(from: group)
    }
}

/// 任务组测试类，验证任务组相关的特性。
class DispatchGroupTest {
    /// 创建一个新任务组
    func creatAGroup() -> DispatchGroup{
        return DispatchGroup()
    }
```

```
    /// 通知主线程任务组中的任务都完成
    func notifyMainQueue(from group: DispatchGroup) {
        group.notify(queue: DispatchQueue.main) {
            print("任务组通知：任务都完成了。\n")
        }
    }

    /// 创建常规的异步任务，并加入任务组中。
    func addTaskNormally(to group: DispatchGroup, in queue: DispatchQueue) {
        queue.async(group: group) {
            print("任务：喝一杯牛奶\n")
        }

        queue.async(group: group) {
            print("任务：吃一个苹果\n")
        }
    }
}
```

执行结果：

```
任务：吃一个苹果


任务：喝一杯牛奶


任务组通知：任务都完成了。
```

示例 7.2：添加系统任务至任务组

我们通过封装系统 SDK 中的 `URLSession` 的 `dataTask` API，将系统任务加入至任务组中，使用 `Group.enter()`、`Group.leave()` 配对方法进行标识。

本示例中，我们将通过封装后的 API 尝试从豆瓣同时下载两本书的标签集，当下载任务完成后返回一个打印任务的闭包，在主线程收到任务组全部完成的通知后，执行该打印闭包。

```
extension QueueTestListTableViewController {
    @IBAction func addSystemTaskToGroupButtonTapped(_ sender: Any) {
        let groupTest = DispatchGroupTest()
        let group = groupTest.creatAGroup()

        let book1ID = "5416832" // https://book.douban.com/subject/5416832/
        let book2ID = "1046265" // https://book.douban.com/subject/1046265/

        // 根据书籍 ID 下载一本豆瓣书籍的标签集，并返回一个打印前 5 个标签的任务闭包。
        let printBookTagBlock1 = groupTest.getBookTag(book1ID, in: group)
        let printBookTagBlock2 = groupTest.getBookTag(book2ID, in: group)
```

```
        // 下载任务完成后，通知主线程完成打印任务。
        groupTest.notifyMainQueue(from: group) {
            printBookTagBlock1("辛亥：摇晃的中国")
            printBookTagBlock2("挪威的森林")
        }
    }
}

class DispatchGroupTest {
    /// 根据书籍 ID 下载一本豆瓣书籍的标签集，并返回一个打印前 5 个标签的任务闭包。此任务将加入指定的任务组中执行。
    func getBookTag(_ bookID: String, in taskGroup: DispatchGroup) -> (String)->() {
        let url = "https://api.douban.com/v2/book/\(bookID)/tags"
        var printBookTagBlock: (_ bookName: String)->() = {_ in print("还未收到返回的书籍信息") }

        // 创建网络信息获取成功后的任务
        let completion = {(data: Data?, response: URLResponse?, error: Error?) in
            printBookTagBlock = { bookName in
                if error != nil{
                    print(error.debugDescription)
                } else {
                    guard let data = data else { return }
                    print("书籍 《\(bookName)》的标签信息如下：")
                    BookTags.printBookPreviousFiveTags(data)
                }
            }
        }

        print("任务：下载书籍 \(bookID) 的信息 \(Date())")
        // 获取网络信息
        httpGet(url: url, in: taskGroup, completion: completion)

        let returnBlock: (String)->() = { bookName in
            printBookTagBlock(bookName)
        }
        return returnBlock
    }
}

/// 执行 http get 方法，并加入指定的任务组。
func httpGet(url: String,
             getString: String? = nil,
             session: URLSession = URLSession.shared,
             in taskGroup: DispatchGroup,
             completion: @escaping (Data?, URLResponse?, Error?) -> Void)
{
    let httpMethod = "GET"
    let urlStruct = URL(string: url) //创建URL对象
```

```
    var request = URLRequest(url: urlStruct!) //创建请求对象
    var dataTask: URLSessionTask

    request.httpMethod = httpMethod
    request.httpBody = getString?.data(using: .utf8)

    dataTask = session.addDataTask(to: taskGroup,
                                   with: request,
                                   completionHandler: completion)
    dataTask.resume() // 启动任务
}

extension URLSession {
    /// 将数据获取的任务加入任务组中
    func addDataTask(to group: DispatchGroup,
                     with request: URLRequest,
                     completionHandler: @escaping (Data?, URLResponse?, Error?) -> Void)
        -> URLSessionDataTask {
            group.enter()
            return dataTask(with: request) { (data, response, error) in
                print("下载结束：\(Date())")
                completionHandler(data, response, error)
                group.leave()
            }
    }
}
```

执行结果，可以看到两本书几乎是同时开始下载的，全部下载结束后再进行打印：

```
任务：下载书籍 5416832 的信息 2018-03-13 03:21:29 +0000
任务：下载书籍 1046265 的信息 2018-03-13 03:21:29 +0000
下载结束：2018-03-13 03:21:30 +0000
下载结束：2018-03-13 03:21:30 +0000
任务组通知：任务都完成了。

书籍 《辛亥：摇晃的中国》的标签信息如下：
历史
张鸣
辛亥革命
民国
中國近代史

书籍 《挪威的森林》的标签信息如下：
村上春树
挪威的森林
小说
日本文学
日本
```

示例 7.3：添加系统及自定义任务至任务组

本示例中，我们将先从豆瓣下载一本书的标签集，并设置一个很短的等待时间，等待过后开启打印任务。然后再加入一个自定义队列的任务，以及里一个书籍下载任务，当这两个任务都完成后，再打印第二本书籍标签信息。

```
extension QueueTestListTableViewController {
    @IBAction func addSystemTaskToGroupButtonTapped(_ sender: Any) {
        let groupTest = DispatchGroupTest()
        let group = groupTest.creatAGroup()
        let queue = DispatchQueue.global()

        let book1ID = "5416832" // https://book.douban.com/subject/5416832/
        let book2ID = "1046265" // https://book.douban.com/subject/1046265/

        // 根据书籍 ID 下载一本豆瓣书籍的标签集，并返回一个打印前 5 个标签的任务闭包。
        let printBookTagBlock1 = groupTest.getBookTag(book1ID, in: group)
        groupTest.wait(group: group, after: 0.01) // 等待前面的任务执行不超过 0.01 秒
        printBookTagBlock1("辛亥：摇晃的中国") // 等待后进行打印

        // 创建常规的异步任务，并加入任务组中。
        groupTest.addTaskNormally(to: group, in: queue)
        // 再次进行下载任务
        let printBookTagBlock2 = groupTest.getBookTag(book2ID, in: group)

        // 全部任务完成后，通知主线程完成打印任务。
        groupTest.notifyMainQueue(from: group) {
            printBookTagBlock2("挪威的森林")
        }
    }
}
```

执行结果，可以看到由于等待时间太短，第一本书还未下载完就开始打印了，因此只打印了空信息。而第二本书等待正常下载完再打印的：

```
任务：下载书籍 5416832 的信息 2018-03-13 03:42:21 +0000
还未收到返回的书籍信息
任务：喝一杯牛奶

任务：下载书籍 1046265 的信息 2018-03-13 03:42:21 +0000
任务：吃一个苹果

下载结束：2018-03-13 03:42:22 +0000
下载结束：2018-03-13 03:42:22 +0000
任务组通知：任务都完成了。

书籍 《挪威的森林》的标签信息如下：
村上春树
挪威的森林
小说
日本文学
日本
```

8. DispatchSource

GCD 中提供了一个 `DispatchSource` 类，它可以帮你监听系统底层一些对象的活动，例如这些对象： `Mach port` 、`Unix descriptor` 、`Unix signal` 、`VFS node` ，并允许你在这些活动发生时，向队列提交一个任务以进行异步处理。

这些可监听的对象都有具体的类型，你可以使用 `DispatchSource` 的类方法来构建这些类型，这里就不一一列举了。下面以文件监听为例说明 `DispatchSource` 的用法。

例子中监听了一个指定目录下文件的写入事件，创建监听主要有几个步骤：

- 通过 `makeFileSystemObjectSource` 方法创建 source
- 通过 `setEventHandler` 设定事件处理程序， `setCancelHandler` 设定取消监听的处理。
- 执行 `resume()` 方法开始接收事件

```
class DispatchSourceTest {
    var filePath: String
    var counter = 0
    let queue = DispatchQueue.global()

    init() {
        filePath = "\(NSTemporaryDirectory())"
        startObserve {
            print("File was changed")
        }
    }

    func startObserve(closure: @escaping () -> Void) {
        let fileURL = URL(fileURLWithPath: filePath)
        let monitoredDirectoryFileDescriptor = open(fileURL.path, O_EVTONLY)

        let source = DispatchSource.makeFileSystemObjectSource(
            fileDescriptor: monitoredDirectoryFileDescriptor,
            eventMask: .write, queue: queue)
        source.setEventHandler(handler: closure)
        source.setCancelHandler {
            close(monitoredDirectoryFileDescriptor)
        }
        source.resume()
    }

    func changeFile() {
        DispatchSourceTest.createFile(name: "DispatchSourceTest.md", filePath:
NSTemporaryDirectory())
        counter += 1
        let text = "\(counter)"
        try! text.write(toFile: "\(filePath)/DispatchSourceTest.md", atomically: true, encoding:
String.Encoding.utf8)
```

```
        print("file writed.")
    }

    static func createFile(name: String, filePath: String){
        let manager = FileManager.default
        let fileBaseUrl = URL(fileURLWithPath: filePath)
        let file = fileBaseUrl.appendingPathComponent(name)
        print("文件: \(file)")

        // 写入 "hello world"
        let exist = manager.fileExists(atPath: file.path)
        if !exist {
            let data = Data(base64Encoded:"aGVsbG8gd29ybGQ="
,options:.ignoreUnknownCharacters)
            let createSuccess = manager.createFile(atPath: file.path,contents:data,attributes:nil)
            print("文件创建结果: \(createSuccess)")
        }
    }
}
```

在 iOS 中这套 `DispatchSource` API 并不常用（`DispatchSourceTimer` 可能用的多点），而且仅上面的文件监听例子经常接收不到事件，在 Mac 中情况可能好点。对于需要经常和底层打交道的人来说，这里面还有很多坑需要去填。`DispatchSource` 的更多例子还可以 参考这里。

9. DispatchIO

`DispatchIO` 对象提供一个操作文件描述符的通道。简单讲你可以利用多线程异步高效地读写文件。

发起读写操作一般步骤如下：

- 创建 `DispatchIO` 对象，或者说创建一个通道，并设置结束处理闭包。
- 调用 `read` / `write` 方法
- 调用 `close` 方法关闭通道
- 在 `close` 方法后系统将自动调用结束处理闭包

下面介绍下各方法的使用。

初始化方法

一般使用两种方式初始化：文件描述符，或者文件路径。

文件描述符方式

文件描述符使用 `open` 方法创建：`open(_ path: UnsafePointer<CChar>, _ oflag: Int32, _ mode: mode_t) -> Int32`，第一个参数是 `UnsafePointer<Int8>` 类型的路径，`oflag` 、`mode` 指文件的操作权限，一个是系统 API 级的，一个是文件系统级的，可选项如下：

oflag：

Flag	备注	功能
O_RDONLY	以只读方式打开文件	此三种读写类型只能有一种
O_WRONLY	以只写方式打开文件	此三种读写类型只能有一种
O_RDWR	以读和写的方式打开文件	此三种读写类型只能有一种
O_CREAT	打开文件，如果文件不存在则创建文件	创建文件时会使用Mode参数与Umask配合设置文件权限
O_EXCL	如果已经置O_CREAT且文件存在，则强制open()失败	可以用来检测多个进程之间创建文件的原子操作
O_TRUNC	将文件的长度截为0	无论打开方式是RD,WR,RDWR，只要打开就会把文件清空
O_APPEND	强制write()从文件尾开始不care当前文件偏移量所处位置，只会在文件末尾开始添加	如果不使用的话，只会在文件偏移量处开始覆盖原有内容写文件

mode：以 | 分隔开的分别是User/Group/Other三个组对应的权限掩码。

```
S_IRWXU | S_IRWXG | S_IRWXO：可读&可写&可执行
S_IRUSR | S_IRGRP | S_IROTH：可读
S_IWUSR | S_IWGR  | S_IWOTH：可写
S_IXUSR | S_IXGRP | S_IXOTH：可执行
```

创建的通道有两种类型：

- 连续数据流：DispatchIO.StreamType.stream，这个方式是对文件从头到尾完整操作的。
- 随机片段数据：DispatchIO.StreamType.random，这个方式是在文件的任意一个位置（偏移量）开始操作的。

```
let filePath: NSString = "test.zip"
// 创建一个可读写的文件描述符
let fileDescriptor = open(filePath.utf8String!, (O_RDWR | O_CREAT | O_APPEND), (S_IRWXU | S_IRWXG))
let queue = DispatchQueue(label: "com.sinkingsoul.DispatchQueueTest.serialQueue")
let cleanupHandler: (Int32) -> Void = { errorNumber in
}
let io = DispatchIO(type: .stream, fileDescriptor: fileDescriptor, queue: queue, cleanupHandler: cleanupHandler)
```

文件路径方式

```
let io = DispatchIO(type: .stream, path: filePath.utf8String!, oflag: (O_RDWR | O_CREAT | O_APPEND), mode: (S_IRWXU | S_IRWXG), queue: queue, cleanupHandler: cleanupHandler)
```

数据块大小阀值

`DispatchIO` 支持多线程操作的原因之一就是它将文件拆分为数据块进行并行操作，你可以设置数据块大小的上下限，系统会采取合适的大小，使用这两个方法即可：`setLimit(highWater: Int)`、`setLimit(lowWater: Int)`，单位是 `byte`。

```
io.setLimit(highWater: 1024*1024)
```

数据块如果设置小一点（如 1M），则可以节省 App 的内存，如果内存足够则可以大一点换取更快速度。在进行读写操作时，有一个性能问题需要注意，如果同时读写的话一般分两个通道，且读到一个数据块就立即写到另一个数据块中，那么写通道的数据块上限不要小于读通道的，否则会造成内存大量积压无法及时释放。

读操作

方法示例：

```
ioRead.read(offset: 0, length: Int.max, queue: ioReadQueue) { doneReading, data, error in
        if (error > 0) {
            print("读取发生错误了，错误码：\(error)")
            return
        }
    if (data != nil) {
            // 使用数据
        }
    if (doneReading) {
            ioRead.close()
        }
}
```

`offset` 指定读取的偏移量，如果通道是 `stream` 类型，值不起作用，写为 0 即可，将从文件开头读起；如果是 `random` 类型，则指相对于创建通道时文件的起始位置的偏移量。

`length` 指定读取的长度，如果是读取文件全部内容，设置 `Int.max` 即可，否则设置一个小于文件大小的值（单位是 `byte`）。

每读取到一个数据块都会调用你设置的处理闭包，系统会提供三个入参给你：结束标志、本次读取到的数据块、错误码：

- 在所有数据读取完成后，会额外再调用一个闭包，通过结束标志告诉你操作结束了，此时 data 大小是 0，错误码也是 0。
- 如果读取中间发生了错误，则会停止读取，结束标志会被设置为 `true`，并返回相应的错误码，错误码表参考稍后的【关闭通道】小节：

写操作

方法示例：

```
ioWrite.write(offset: 0, data: data!, queue: ioWriteQueue) { doneWriting, data, error in
    if (error > 0) {
        print("写入发生错误了，错误码：\(error)")
        return
    }
    if doneWriting {
        //...
        ioWrite.close()
    }
}
```

写操作与读操作的唯一区别是：每当写完一个数据块时，回调闭包返回的 data 是剩余的全部数据。同时注意如果是 `stream` 类型，将接着文件的末尾写数据。

关闭通道

当读写正常完成，或者你需要中途结束操作时，需要调用 `close` 方法，这个方法带一个 `DispatchIO.CloseFlags` 类型参数，如果不指定将默认值为 `DispatchIO.CloseFlags.stop` 。

这个方法传入 `stop` 标志时将会停止所有未完成的读写操作，影响范围是所有 `I/O channel`，其他 `DispatchIO` 对象进行中的读写操作将会收到一个 `ECANCELED` 错误码，`rawValue` 值是 89，这个错误码是 `POSIXError` 结构的一个属性，而 `POSIXError` 又是 `NSError` 中预定义的一个错误域。

因此如果要在不同 `DispatchIO` 对象中并行读取操作互不影响， `close` 方法标志可以设置一个空值：`DispatchIO.CloseFlags()` 。如果设置了 `stop` 标志，则要做好不同 IO 之间的隔离，通过任务组的 `enter`、`leave`、`wait` 方法可以做到较好的隔离。

```
ioWrite.close() // 停止标志
ioWrite.close(flags: DispatchIO.CloseFlags()) // 空标志
```

`POSIXError` 码表：

```
EPERM  = 1  // 无
ENOENT  = 2  // No such file or directory.
ESRCH  = 3  // No such process.
EINTR  = 4  // Interrupted system call.
EIO  = 5  // Input/output error.
ENXIO  = 6  // Device not configured.
E2BIG  = 7  // Argument list too long.
ENOEXEC  = 8  // Exec format error.
EBADF  = 9  // Bad file descriptor.
ECHILD  = 10  // No child processes.
EDEADLK  = 11  // Resource deadlock avoided.
ENOMEM  = 12  // Cannot allocate memory.
EACCES  = 13  // Permission denied.
EFAULT  = 14  // Bad address.
ENOTBLK  = 15  // Block device required.
EBUSY  = 16  // Device / Resource busy.
EEXIST  = 17  // File exists.
```

```
EXDEV  = 18  // Cross-device link.
ENODEV  = 19  // Operation not supported by device.
ENOTDIR  = 20  // Not a directory.
EISDIR  = 21  // Is a directory.
EINVAL  = 22  // Invalid argument.
ENFILE  = 23  // Too many open files in system.
EMFILE  = 24  // Too many open files.
ENOTTY  = 25  // Inappropriate ioctl for device.
ETXTBSY  = 26  // Text file busy.
EFBIG  = 27  // File too large.
ENOSPC  = 28  // No space left on device.
ESPIPE  = 29  // Illegal seek.
EROFS  = 30  // Read-only file system.
EMLINK  = 31  // Too many links.
EPIPE  = 32  // Broken pipe.
EDOM  = 33  // math software. Numerical argument out of domain.
ERANGE  = 34  // Result too large.
EAGAIN  = 35  // non-blocking and interrupt i/o. Resource temporarily unavailable.
EWOULDBLOCK  = 35  // Operation would block.
EINPROGRESS  = 36  // Operation now in progress.
EALREADY  = 37  // Operation already in progress.
ENOTSOCK  = 38  // ipc/network software — argument errors. Socket operation on non-socket.
EDESTADDRREQ  = 39  // Destination address required.
EMSGSIZE  = 40  // Message too long.
EPROTOTYPE  = 41  // Protocol wrong type for socket.
ENOPROTOOPT  = 42  // Protocol not available.
EPROTONOSUPPORT  = 43  // Protocol not supported.
ESOCKTNOSUPPORT  = 44  // Socket type not supported.
ENOTSUP  = 45  // Operation not supported.
EPFNOSUPPORT  = 46  // Protocol family not supported.
EAFNOSUPPORT  = 47  // Address family not supported by protocol family.
EADDRINUSE  = 48  // Address already in use.
EADDRNOTAVAIL  = 49  // Can't assign requested address.
ENETDOWN  = 50  // ipc/network software — operational errors Network is down.
ENETUNREACH  = 51  // Network is unreachable.
ENETRESET  = 52  // Network dropped connection on reset.
ECONNABORTED  = 53  // Software caused connection abort.
ECONNRESET  = 54  // Connection reset by peer.
ENOBUFS  = 55  // No buffer space available.
EISCONN  = 56  // Socket is already connected.
ENOTCONN  = 57  // Socket is not connected.
ESHUTDOWN  = 58  // Can't send after socket shutdown.
ETOOMANYREFS  = 59  // Too many references: can't splice.
ETIMEDOUT  = 60  // Operation timed out.
ECONNREFUSED  = 61  // Connection refused.
ELOOP  = 62  // Too many levels of symbolic links.
ENAMETOOLONG  = 63  // File name too long.
EHOSTDOWN  = 64  // Host is down.
EHOSTUNREACH  = 65  // No route to host.
```

```
ENOTEMPTY  = 66  // Directory not empty.
EPROCLIM  = 67  // quotas & mush. Too many processes.
EUSERS  = 68  // Too many users.
EDQUOT  = 69  // Disc quota exceeded.
ESTALE  = 70  // Network File System. Stale NFS file handle.
EREMOTE  = 71  // Too many levels of remote in path.
EBADRPC  = 72  // RPC struct is bad.
ERPCMISMATCH  = 73  // RPC version wrong.
EPROGUNAVAIL  = 74  // RPC prog. not avail.
EPROGMISMATCH  = 75  // Program version wrong.
EPROCUNAVAIL  = 76  // Bad procedure for program.
ENOLCK  = 77  // No locks available.
ENOSYS  = 78  // Function not implemented.
EFTYPE  = 79  // Inappropriate file type or format.
EAUTH  = 80  // Authentication error.
ENEEDAUTH  = 81  // Need authenticator.
EPWROFF  = 82  // Intelligent device errors. Device power is off.
EDEVERR  = 83  // Device error e.g. paper out.
EOVERFLOW  = 84  // Value too large to be stored in data type.
EBADEXEC  = 85  // Program loading errors. Bad executable.
EBADARCH  = 86  // Bad CPU type in executable.
ESHLIBVERS  = 87  // Shared library version mismatch.
EBADMACHO  = 88  // Malformed Macho file.
ECANCELED  = 89  // Operation canceled.
EIDRM  = 90  // Identifier removed.
ENOMSG  = 91  // No message of desired type.
EILSEQ  = 92  // Illegal byte sequence.
ENOATTR  = 93  // Attribute not found.
EBADMSG  = 94  // Bad message.
EMULTIHOP  = 95  // Reserved.
ENODATA  = 96  // No message available on STREAM.
ENOLINK  = 97  // Reserved.
ENOSR  = 98  // No STREAM resources.
ENOSTR  = 99  // Not a STREAM.
EPROTO  = 100  // Protocol error.
ETIME  = 101  // STREAM ioctl timeout.
ENOPOLICY  = 103  // No such policy registered.
ENOTRECOVERABLE  = 104  // State not recoverable.
EOWNERDEAD  = 105  // Previous owner died.
EQFULL  = 106  // Interface output queue is full.
```

代码示例

示例 9.1：将两个大文件（通过压缩工具拆分的包）合并为一个文件。

实现思路：分别创建一个读、写通道，使用同一个串行队列处理数据，每读到一个数据块就提交一个写数据的任务，同时要保证按照读取的顺序提交写任务，在第一个文件读写完成后再开始第二个文件的读写操作。

测试文件地址：WWDC 2016–720，通过 Zip 压缩拆分为两个文件(Normal 方式)，设置按 350M 进行分割。注意测试时，建议使用模拟器，更方便读写 Mac 本地文件，后续类似例子相同。

```
class DispatchIOTest {
    /// 利用很小的内存空间及同一队列读写方式合并文件
    static func combineFileWithOneQueue() {
        let files: NSArray = ["/Users/xxx/Downloads/gcd.mp4.zip.001",
                              "/Users/xxx/Downloads/gcd.mp4.zip.002"]
        let outFile: NSString = "/Users/xxx/Downloads/gcd.mp4.zip"
        let ioQueue = DispatchQueue(
            label: "com.sinkingsoul.DispatchQueueTest.serialQueue")
        let queueGroup = DispatchGroup()

        let ioWriteCleanupHandler: (Int32) -> Void = { errorNumber in
            print("写入文件完成 @\(Date())。")
        }

        let ioReadCleanupHandler: (Int32) -> Void = { errorNumber in
            print("读取文件完成。")
        }

        let ioWrite = DispatchIO(type: .stream,
                                 path: outFile.utf8String!,
                                 oflag: (O_RDWR | O_CREAT | O_APPEND),
                                 mode: (S_IRWXU | S_IRWXG),
                                 queue: ioQueue,
                                 cleanupHandler: ioWriteCleanupHandler)
        ioWrite?.setLimit(highWater: 1024*1024)

//        print("开始操作 @\(Date()).")

        files.enumerateObjects { fileName, index, stop in
            if stop.pointee.boolValue {
                return
            }
            queueGroup.enter()

            let ioRead = DispatchIO(type: .stream,
                                    path: (fileName as! NSString).utf8String!,
                                    oflag: O_RDONLY,
                                    mode: 0,
                                    queue: ioQueue,
                                    cleanupHandler: ioReadCleanupHandler)
            ioRead?.setLimit(highWater: 1024*1024)

            print("开始读取文件: \(fileName) 的数据")

            ioRead?.read(offset: 0, length: Int.max, queue: ioQueue) { doneReading, data, error in
                print("当前读线程：\(Thread.current)--->")
                if (error > 0 || stop.pointee.boolValue) {
```

```
                print("读取发生错误了，错误码：\(error)")
                ioWrite?.close()
                stop.pointee = true
                return
            }

            if (data != nil) {
                let bytesRead: size_t = data!.count
                if (bytesRead > 0) {
                    queueGroup.enter()
                    ioWrite?.write(offset: 0, data: data!, queue: ioQueue) {
                        doneWriting, data, error in
                        print("当前写线程：\(Thread.current)--->")
                        if (error > 0 || stop.pointee.boolValue) {
                            print("写入发生错误了，错误码：\(error)")
                            ioRead?.close()
                            stop.pointee = true
                            queueGroup.leave()
                            return
                        }
                        if doneWriting {
                            queueGroup.leave()
                        }
                        print("--->当前写线程：\(Thread.current)")
                    }
                }
            }

            if (doneReading) {
                ioRead?.close()
                if (files.count == (index+1)) {
                    ioWrite?.close()
                }
                queueGroup.leave()
            }
            print("--->当前读线程：\(Thread.current)")
        }
        _ = queueGroup.wait(timeout: .distantFuture)
    }
  }
}
```

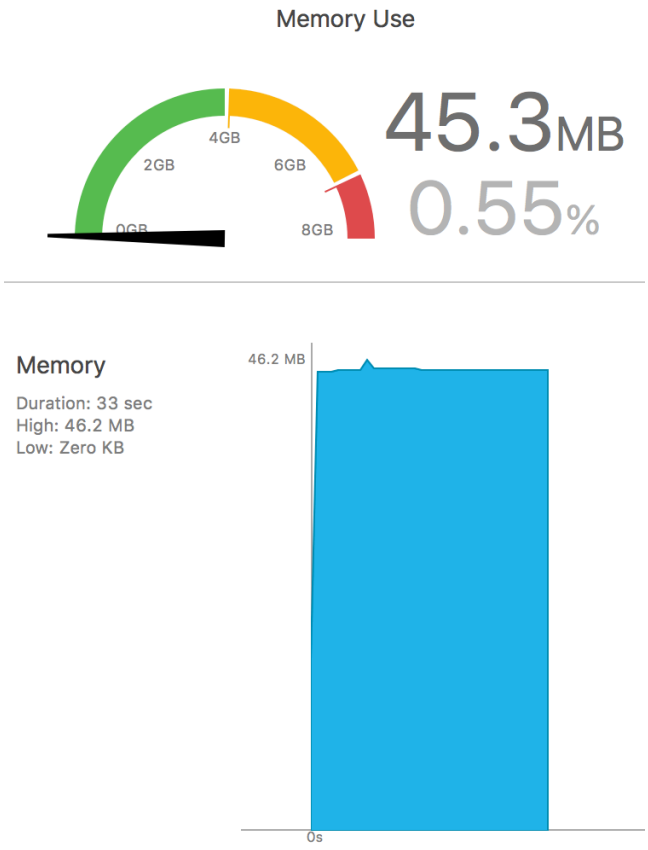
执行结果，可以看到串行队列利用了好几个线程来处理读写操作，但是细看同一时间只运行了一个线程，符合我们前面总结的串行队列的特点：

> 开始读取文件: /Users/xxx/Downloads/gcd.mp4.zip.001 的数据
> 当前读线程：<NSThread: 0x60400027e500>{number = 3, name = (null)}--->
> --->当前读线程：<NSThread: 0x60400027e500>{number = 3, name = (null)}
> 当前读线程：<NSThread: 0x6000006628c0>{number = 4, name = (null)}--->
> --->当前读线程：<NSThread: 0x6000006628c0>{number = 4, name = (null)}

```
当前写线程：<NSThread: 0x60400027e500>{number = 3, name = (null)}--->
--->当前写线程：<NSThread: 0x60400027e500>{number = 3, name = (null)}
当前读线程：<NSThread: 0x6000006628c0>{number = 4, name = (null)}--->
--->当前读线程：<NSThread: 0x6000006628c0>{number = 4, name = (null)}

......

当前读线程：<NSThread: 0x600000662840>{number = 6, name = (null)}--->
--->当前读线程：<NSThread: 0x600000662840>{number = 6, name = (null)}
当前写线程：<NSThread: 0x600000662ac0>{number = 7, name = (null)}--->
--->当前写线程：<NSThread: 0x600000662ac0>{number = 7, name = (null)}

......

读取文件完成。

......

当前写线程：<NSThread: 0x60400027e500>{number = 3, name = (null)}--->
--->当前写线程：<NSThread: 0x60400027e500>{number = 3, name = (null)}
写入文件完成。
```

关闭 `print` 后的内存占用情况见下图，可以看到在读写过程中只额外占用了 1M 左右内存，用时 1s 左右，非常的棒。

```
开始操作 @2018-xx-xx 13:51:52 +0000.
开始读取文件: /Users/xxx/Downloads/gcd.mp4.zip.001 的数据
读取文件完成。
开始读取文件: /Users/xxx/Downloads/gcd.mp4.zip.002 的数据
读取文件完成。
写入文件完成 @2018-xx-xx 13:51:53 +0000。
```



示例 9.2：利用多个队列将两个大文件合并为一个文件。

这个例子在上面例子的基础上，各使用两个队列来进行读、写操作，验证利用地址偏移的方式多线程同时读写文件的效率。

这里对读写文件时的偏移量 `offset` 再做个简单说明：文件开头的偏移量是 0，后续逐渐递增，直到文件末尾的偏移量是 (按字节计算的文件大小 – 1)。

```
/// 利用很小的内存空间及双队列读写方式合并文件
static func combineFileWithMoreQueues() {
    let files: NSArray = ["/Users/xxx/Downloads/gcd.mp4.zip.001",
                          "/Users/xxx/Downloads/gcd.mp4.zip.002"]
//        真机运行时可使用以下地址（需手动将文件放入工程中）
//        let files: NSArray = [Bundle.main.path(forResource: "gcd.mp4.zip", ofType: "001")!,
//                              Bundle.main.path(forResource: "gcd.mp4.zip", ofType: "002")!]
    var filesSize = files.map {
        return (try! FileManager.default.attributesOfItem(atPath: $0 as! String)[FileAttributeKey.size]
as! NSNumber).int64Value
    }
    let outFile: NSString = "/Users/xxx/Downloads/gcd.mp4.zip"
//        真机运行时可使用以下地址（需手动将文件放入工程中）
//        let outFile: NSString = "\(NSTemporaryDirectory())/gcd.mp4.zip" as NSString

    // 每个分块文件各一个读、写队列
    let ioReadQueue1 = DispatchQueue(
        label: "com.sinkingsoul.DispatchQueueTest.serialQueue1")
    let ioReadQueue2 = DispatchQueue(
        label: "com.sinkingsoul.DispatchQueueTest.serialQueue2")
    let ioWriteQueue1 = DispatchQueue(
        label: "com.sinkingsoul.DispatchQueueTest.serialQueue3")
    let ioWriteQueue2 = DispatchQueue(
        label: "com.sinkingsoul.DispatchQueueTest.serialQueue4")

    let ioReadQueueArray = [ioReadQueue1, ioReadQueue2]
    let ioWriteQueueArray = [ioWriteQueue1, ioWriteQueue2]
    let ioWriteCleanupHandler: (Int32) -> Void = { errorNumber in
        print("写入文件完成 @\(Date())。")
    }
    let ioReadCleanupHandler: (Int32) -> Void = { errorNumber in
        print("读取文件完成 @\(Date())。")
    }

    let queueGroup = DispatchGroup()

    print("开始操作 @\(Date()).")

    let ioWrite = DispatchIO(type: .random, path: outFile.utf8String!, oflag: (O_RDWR | O_CREAT |
O_APPEND), mode: (S_IRWXU | S_IRWXG), queue: ioWriteQueue1, cleanupHandler:
ioWriteCleanupHandler)
```

```
    ioWrite?.setLimit(highWater: 1024 * 1024)
    ioWrite?.setLimit(lowWater: 1024 * 1024)

    filesSize.insert(0, at: 0)
    filesSize.removeLast()

    for (index, file) in files.enumerated() {
        DispatchQueue.global().sync {
            queueGroup.enter()

            let ioRead = DispatchIO(type: .stream, path: (file as! NSString).utf8String!, oflag:
O_RDONLY, mode: 0, queue: ioReadQueue1, cleanupHandler: ioReadCleanupHandler)
            ioRead?.setLimit(highWater: 1024 * 1024)
            ioRead?.setLimit(lowWater: 1024 * 1024)

            var writeOffsetTemp = filesSize[0...index].reduce(0) { offset, size in
                return offset + size
            }

            ioRead?.read(offset: 0, length: Int.max, queue: ioReadQueueArray[index]) {
                doneReading, data, error in
//                    print("读取文件: \(file)，线程：\(Thread.current)--->")
                if (error > 0) {
                    print("读取文件: \(file) 发生错误了，错误码：\(error)")
                    return
                }

                if (doneReading) {
                    ioRead?.close()
                    queueGroup.leave()
                }

                if (data != nil) {
                    let bytesRead: size_t = data!.count
                    if (bytesRead > 0) {
                        queueGroup.enter()
                        ioWrite?.write(offset: writeOffsetTemp, data: data!, queue:
ioWriteQueueArray[index]) {
                            doneWriting, writeData, error in
//                                print("写入文件: \(file), 线程：\(Thread.current)--->")
                            if (error > 0) {
                                print("写入文件: \(file) 发生错误了，错误码：\(error)")
                                ioRead?.close()
                                return
                            }
                            if doneWriting {
                                queueGroup.leave()
                            }
//                                print("--->写入文件: \(file), 线程：\(Thread.current)")
```

```
                    }
                    writeOffsetTemp = writeOffsetTemp + Int64(data!.count)
                }
            }
//                  print("--->读取文件: \(file)，线程：\(Thread.current)")
        }
    }
}
_ = queueGroup.wait(timeout: .distantFuture)
ioWrite?.close()

}
```

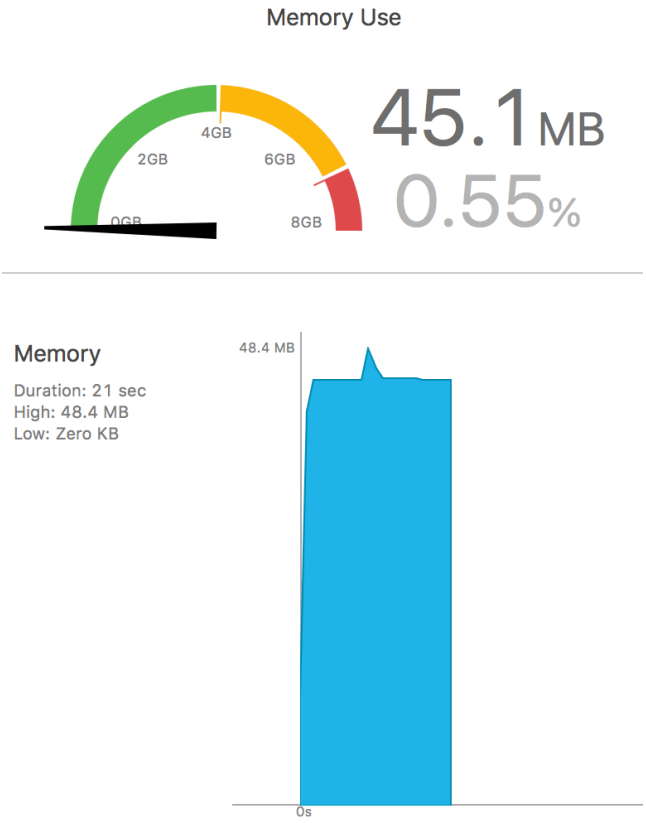
执行结果，可以看到 4 个串行队列同时都在运行：

```
开始操作 @2018-04-03 03:57:00 +0000.
读取文件: /Users/xxx/Downloads/gcd.mp4.zip.001，线程：<NSThread: 0x60400047d940>{number = 3, name = (null)}--->
读取文件: /Users/xxx/Downloads/gcd.mp4.zip.002，线程：<NSThread: 0x60400047d9c0>{number = 4, name = (null)}--->
--->读取文件: /Users/xxx/Downloads/gcd.mp4.zip.002 ，线程：<NSThread: 0x60400047d9c0>{number = 4, name = (null)}
--->读取文件: /Users/xxx/Downloads/gcd.mp4.zip.001 ，线程：<NSThread: 0x60400047d940>{number = 3, name = (null)}

......

写入文件: /Users/xxx/Downloads/gcd.mp4.zip.001, 线程：<NSThread: 0x60400047d940>{number = 3, name = (null)}--->
写入文件: /Users/xxx/Downloads/gcd.mp4.zip.002, 线程：<NSThread: 0x600000672980>{number = 5, name = (null)}--->
--->写入文件: /Users/xxx/Downloads/gcd.mp4.zip.001, 线程：<NSThread: 0x60400047d940>{number = 3, name = (null)}
--->写入文件: /Users/xxx/Downloads/gcd.mp4.zip.002, 线程：<NSThread: 0x600000672980>{number = 5, name = (null)}

......

写入文件: /Users/xxx/Downloads/gcd.mp4.zip.001, 线程：<NSThread: 0x6000006720c0>{number = 9, name = (null)}--->
--->写入文件: /Users/xxx/Downloads/gcd.mp4.zip.001, 线程：<NSThread: 0x6000006720c0>{number = 9, name = (null)}
读取文件完成 @2018-04-03 03:57:04 +0000。
写入文件: /Users/xxx/Downloads/gcd.mp4.zip.001, 线程：<NSThread: 0x6000006720c0>{number = 9, name = (null)}--->
--->写入文件: /Users/xxx/Downloads/gcd.mp4.zip.001, 线程：<NSThread: 0x6000006720c0>{number = 9, name = (null)}
写入文件: /Users/xxx/Downloads/gcd.mp4.zip.001, 线程：<NSThread: 0x6000006720c0>{number = 9, name = (null)}--->
--->写入文件: /Users/xxx/Downloads/gcd.mp4.zip.001, 线程：<NSThread: 0x6000006720c0>{number = 9, name = (null)}
写入文件: /Users/xxx/Downloads/gcd.mp4.zip.001, 线程：<NSThread: 0x6000006720c0>{number = 9, name = (null)}--->
读取文件完成 @2018-04-03 03:57:04 +0000。
```

```
--->写入文件: /Users/xxx/Downloads/gcd.mp4.zip.001, 线程：<NSThread: 0x6000006720c0>{number = 9, name = (null)}
写入文件: /Users/xxx/Downloads/gcd.mp4.zip.001, 线程：<NSThread: 0x6000006720c0>{number = 9, name = (null)}--->
--->写入文件: /Users/xxx/Downloads/gcd.mp4.zip.001, 线程：<NSThread: 0x6000006720c0>{number = 9, name = (null)}

.....

写入文件: /Users/xxx/Downloads/gcd.mp4.zip.001, 线程：<NSThread: 0x6000006720c0>{number = 9, name = (null)}--->
--->写入文件: /Users/xxx/Downloads/gcd.mp4.zip.001, 线程：<NSThread: 0x6000006720c0>{number = 9, name = (null)}
写入文件完成 @2018-04-03 03:57:04 +0000。
```

关闭 `print` 后的内存占用情况见下图，可以看到在读写过程中额外占用了 3M 左右内存，用时 2s 左右。这个结果中，内存占用比单队列大（这个比较好理解），但速度还更慢了，性能瓶颈很有可能是在磁盘读写上。所以涉及文件写操作时，并不是线程越多越快，要考虑传输速度、文件大小等因素。

```
开始操作 @2018-04-03 04:05:44 +0000.
读取文件完成 @2018-04-03 04:05:45 +0000。
读取文件完成 @2018-04-03 04:05:46 +0000。
写入文件完成 @2018-04-03 04:05:46 +0000。
```



10. DispatchData

`DispatchData` 对象可以管理基于内存的数据缓冲区。这个数据缓冲区对外表现为连续的内存区域，但内部可能由多个独立的内存区域组成。

`DispatchData` 对象很多特性类似于 `Data` 对象，且 `Data` 对象可以转换为 `DispatchData` 对象，而通过 `DispatchIO` 的 `read` 方法获得的数据也是封装为 `DispatchData` 对象的。

下面再看个示例，通过 `Data`、`DispatchData`、`DispatchIO` 这三种类型结合，完成内存占用更小也同样快速的文件读写操作。

代码示例

示例 10.1：将两个大文件合并为一个文件（与示例 9.1 类似）。

实现思路：首先将两个文件转换为 `Data` 对象，再转换为 `DispatchData` 对象，然后拼接两个对象为一个 `DispatchData` 对象，最后通过 `DispatchIO` 的 `write` 方法写入文件中。看起来有多次的转换过程，实际上 `Data` 类型读取文件时支持虚拟隐射的方式，而 `DispatchData` 类型更是支持多个数据块虚拟拼接，也不占用什么内存。

> 实际上完全使用 `Data` 类型也能完成文件合并，利用 `append`、`write` 方法即可，但是 `append` 方法是要占用比文件大小稍大的内存，`write` 方法也要占用额外内存空间。即使使用 `NSMutableData` 类型不占用内存的 `append` 方法通过虚拟隐射方式读文件（即读文件、拼接数据都不占用内存），但是 `NSMutableData` 类型的 `write` 方法还是要占用额外内存，虽然要比 `Data` 类型内存少很多，但是也不少了。因此 `DispatchData` 类型在内存占用上更有优势。

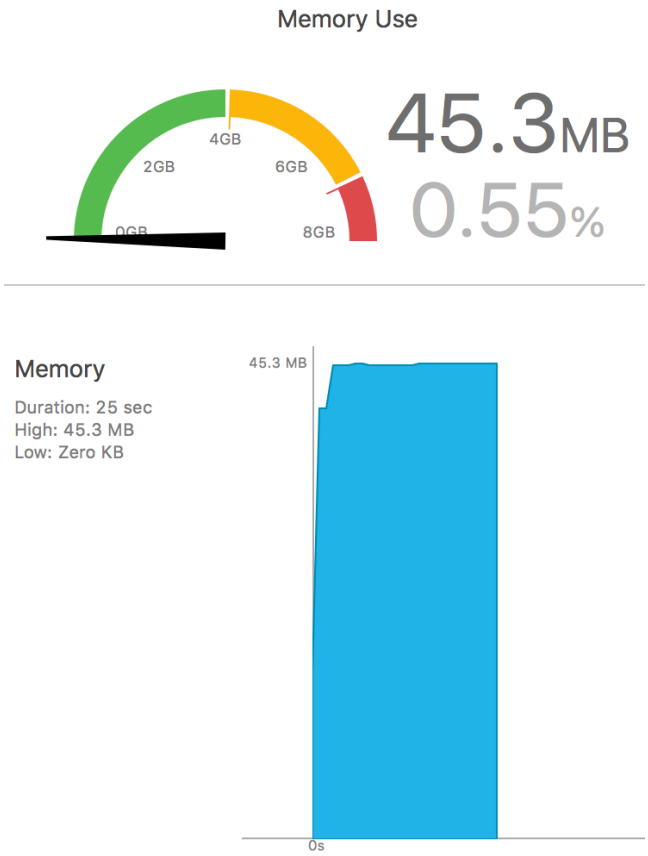
```
/// 利用 DispatchData 类型快速合并文件
static func combineFileWithDispatchData() {
    let filePathArray = ["/Users/xxx/Downloads/gcd.mp4.zip.001",
                         "/Users/xxx/Downloads/gcd.mp4.zip.002"]
    let outputFilePath: NSString = "/Users/xxx/Downloads/gcd.mp4.zip"
    let ioWriteQueue = DispatchQueue(
        label: "com.sinkingsoul.DispatchQueueTest.serialQueue")

    let ioWriteCleanupHandler: (Int32) -> Void = { errorNumber in
        print("写入文件完成 @\(Date()).")
    }
    let ioWrite = DispatchIO(type: .stream,
                             path: outputFilePath.utf8String!,
                             oflag: (O_RDWR | O_CREAT | O_APPEND),
                             mode: (S_IRWXU | S_IRWXG),
                             queue: ioWriteQueue,
                             cleanupHandler: ioWriteCleanupHandler)
    ioWrite?.setLimit(highWater: 1024*1024*2)

    print("开始操作 @\(Date()).")

    // 将所有文件合并为一个 DispatchData 对象
    let dispatchData = filePathArray.reduce(DispatchData.empty) { data, filePath in
        // 将文件转换为 Data
        let url = URL(fileURLWithPath: filePath)
        let fileData = try! Data(contentsOf: url, options: .mappedIfSafe)
        var tempData = data
        // 将 Data 转换为 DispatchData
        let dispatchData = fileData.withUnsafeBytes {
            (u8Ptr: UnsafePointer<UInt8>) -> DispatchData in
```

```
            let rawPtr = UnsafeRawPointer(u8Ptr)
            let innerData = Unmanaged.passRetained(fileData as NSData)
            return DispatchData(bytesNoCopy:
                UnsafeRawBufferPointer(start: rawPtr, count: fileData.count),
                            deallocator: .custom(nil, innerData.release))
        }
        // 拼接 DispatchData
        tempData.append(dispatchData)
        return tempData
    }

    //将 DispatchData 对象写入结果文件中
    ioWrite?.write(offset: 0, data: dispatchData, queue: ioWriteQueue) {
        doneWriting, data, error in
        if (error > 0) {
            print("写入发生错误了，错误码：\(error)")
            return
        }

        if data != nil {
//              print("正在写入文件，剩余大小：\(data!.count) bytes.")
        }

        if (doneWriting) {
            ioWrite?.close()
        }
    }
}
```

执行结果：

```
开始操作 @2018-xx-xx 13:32:37 +0000.
正在写入文件，剩余大小：640096267 bytes.
正在写入文件，剩余大小：639047691 bytes.
.....
正在写入文件，剩余大小：464907 bytes.
写入文件完成 @2018-xx-xx 13:32:40 +0000.
```

关闭 `print` 后的内存占用情况见下图，可以看到在整个读写过程中几乎没有额外占用内存，速度很快在 1s 左右，这个读写方案堪称完美，这要归功于 `DispatchData` 的虚拟拼接和 `DispatchIO` 的分块读写大小控制。这里顺便提一下 `DispatchIO` 数据阀值上限 `highWater`，经过测试，如果设置为 1M，将耗时 4s 左右，设为 2M 及以上时，耗时均为 1s 左右，非常快速，而所有阀值的内存占用都很少。所以设置合理的阀值，对性能的改善也是有帮助的。



11. 信号量

`DispatchSemaphore`，通常称作信号量，顾名思义，它可以通过计数来标识一个信号，这个信号怎么用呢，取决于任务的性质。通常用于对同一个资源访问的任务数进行限制。

例如，控制同一时间写文件的任务数量、控制端口访问数量、控制下载任务数量等。

信号量的使用非常的简单：

- 首先创建一个初始数量的信号对象
- 使用 `wait` 方法让信号量减 1，再安排任务。如果此时信号量仍大于或等于 0，则任务可执行，如果信号量小于 0，则任务需要等待其他地方释放信号。
- 任务完成后，使用 `signal` 方法增加一个信号量。
- 等待信号有两种方式：永久等待、可超时的等待。

下面看个简单的例子

代码示例

示例 11.1：限制同时运行的任务数。

```
/// 信号量测试类
class DispatchSemaphoreTest {

    /// 限制同时运行的任务数
    static func limitTaskNumber() {
```

```
        let queue = DispatchQueue(
            label: "com.sinkingsoul.DispatchQueueTest.concurrentQueue",
            attributes: .concurrent)
        let semaphore = DispatchSemaphore(value: 2) // 设置数量为 2 的信号量

        semaphore.wait()
        queue.async {
            task(index: 1)
            semaphore.signal()
        }

        semaphore.wait()
        queue.async {
            task(index: 2)
            semaphore.signal()
        }

        semaphore.wait()
        queue.async {
            task(index: 3)
            semaphore.signal()
        }
    }

    /// 任务
    static func task(index: Int) {
        print("Begin task \(index) --->")
        Thread.sleep(forTimeInterval: 2)
        print("Sleep for 2 seconds in task \(index).")
        print("--->End task \(index).")
    }

}
```

执行结果，示例中设置了同时只能运行 2 个任务，可以看到任务 3 在前两个任务完成后才开始运行：

```
Begin task 2 --->
Begin task 1 --->
Sleep for 2 seconds in task 2.
Sleep for 2 seconds in task 1.
--->End task 2.
--->End task 1.
Begin task 3 --->
Sleep for 2 seconds in task 3.
--->End task 3.
```

12. 任务对象

在队列和任务组中，任务实际上是被封装为一个 `DispatchWorkItem` 对象的。任务封装最直接的好处就是可以取消任务。

前面提到的栅栏任务就是通过封装任务对象实现的。

创建任务

先看看它的创建，其中 `qos` 、`flags` 参数都有默认值，可以不填：

```
let workItem = DispatchWorkItem(qos: .default, flags: DispatchWorkItemFlags()) {
    // Do something
}
```

`qos` 前面提到过了，这里说一下 `DispatchWorkItemFlags`，它有以下几个静态属性（详细解释可参考 官方源码）：

- assignCurrentContext: 标记应该为任务分配创建它时的上下文属性（例如：QoS、os_activity_t、可能存在的当前 IPC 请求属性）。如果直接调用任务，任务对象将在它的持续时间内在调用线程中应用这些属性。如果提交任务至队列中，则会替换提交任务时的上下文属性默认值。
- barrier: 标记任务为栅栏任务，提交至并行队列时生效，如果直接运行该任务对象则无此效果。
- detached: 标记任务在执行时应该剥离当前执行上下文属性（例如：QoS、os_activity_t、可能存在的当前 IPC 请求属性）。如果直接调用任务，任务对象将在它的持续时间内从调用线程中删除这些属性（如果存在属性，且应用于任务之前）。如果提交任务至队列中，将使用队列的属性（或专门分配给任务对象的任何属性）进行执行。如果创建任务时指定了 QoS，则该 QoS 将优先于 flag 对应的 QoS 值。
- enforceQoS: 标记任务提交至队列执行时，任务对象被分配的 QoS （提交任务时的值）应优先于队列的 QoS，这样做不会降低 QoS。当任务提交至队列同步执行时，或则直接执行任务时，这个 flag 是默认值。
- inheritQoS: 标记任务提交至队列执行时，队列的 QoS 应优先于任务对象被分配的 QoS （提交任务时的值），后一个 QoS 值只会在队列的 QoS 有问题时才会采用，这样做会导致 QoS 不会低于继承自队列的 QoS。当任务提交至队列异步执行时，这个 flag 是默认值，且直接执行任务时该标志无效。
- noQoS: 标记任务不应指定 QoS，如果直接执行，将以调用线程的 QoS 执行。如果提交至队列，则会替换提交任务时的 QoS 默认值。

执行任务

执行任务时，调用任务项对象的 `perform()` 方法，这个调用是同步执行的：

```
workItem.perform()
```

或则在队列中执行：

```
let queue = DispatchQueue.global()
queue.async(execute: workItem)
```

取消任务

在任务未实际执行之前可以取消任务，调用 `cancel()` 方法，这个调用是异步执行的：

```
workItem.cancel()
```

取消任务将会带来以下结果：

- 取消将导致 **任何** 将来的任务在执行时立即返回，但不会影响已在执行的任务。
- 与任务对象关联的任何资源的释放都会延迟，直到下一次尝试执行任务对象（或者任何正在进行中的执行已完成）。因此需要注意确保可能被取消的任务对象不要捕获任何需要实际执行才能释放的资源，例如使用 `malloc(3)` 进行内存分配，而在任务中调用 `free(3)` 释放。 如果由于取消而从未执行任务，则会导致内存泄露。

任务通知

任务对象也有一个通知方法，在任务执行完成后可以向指定队列发送一个异步调用闭包：

```
workItem.notify(queue: queue) {
    // Do something
}
```

这个通知方法有一些地方需要注意：

- 任务不支持在被多次调用结束后再发出通知，运行时将会报错，通知只能响应一次完整的调用（如果在发出通知时，还有另一次执行未完成，这种情况也视为只有一次调用）。需要在多次执行结束后发出通知，使用任务组的通知更合适。
- 可以多次发出通知，但通知执行的顺序是不确定的。
- 任务只要提交至队列中，即使调用 `cancel()` 方法被取消了，通知也可以生效。

任务等待

任务对象支持等待方法，类似于任务组的等待，也是阻塞型的，需要等待已有的任务完成才能继续执行，也可以指定等待时间：

```
workItem.perform()
workItem.wait()
workItem.wait(timeout: DispatchTime) // 指定等待时间
workItem.wait(wallTimeout: DispatchWallTime) // 指定等待时间
// 等待任务完成
// do something
```

下面看个完整的例子：

代码示例

示例 12.1：任务对象测试。

```
/// 任务对象测试
@IBAction func dispatchWorkItemTestButtonTapped(_ sender: Any) {
    DispatchWorkItemTest.workItemTest()
}

/// 任务对象测试类
```

```
class DispatchWorkItemTest {
    static func workItemTest() {
        var value = 10
        let workItem = DispatchWorkItem {
            print("workItem running start.--->")
            value += 5
            print("value = ", value)
            print("--->workItem running end.")
        }
        let queue = DispatchQueue.global()

        queue.async(execute: workItem)

        queue.async {
            print("异步执行 workItem")
            workItem.perform()
            print("任务2取消了吗：\(workItem.isCancelled)")
            workItem.cancel()
            print("异步执行 workItem end")
        }

        workItem.notify(queue: queue) {
            print("notify 1: value = ", value)
        }

        workItem.notify(queue: queue) {
            print("notify 2: value = ", value)
        }

        workItem.notify(queue: queue) {
            print("notify 3: value = ", value)
        }

        queue.async {
            print("异步执行2 workItem")
            Thread.sleep(forTimeInterval: 2)
            print("任务3取消了吗：\(workItem.isCancelled)")
            workItem.perform()
            print("异步执行2 workItem end")
        }
    }

}
```

执行结果，可以看到任务第一次执行完成后，发出了 3 次通知，而且未按照代码的顺序。在发出通知前，任务还有一次执行未完成，并未造成通知报错。第二次执行任务后，取消了任务，因此任务第三次未正常执行：

```
workItem running start.--->
异步执行 workItem
异步执行2 workItem
value = 15
workItem running start.--->
value = 20
--->workItem running end.
任务2取消了吗：false
异步执行 workItem end
notify 2: value = 20
notify 3: value = 20
notify 1: value = 20
--->workItem running end.
任务3取消了吗：true
异步执行2 workItem end
```

附：时间相关的结构体说明

DispatchTime

它通过时间间隔的方式来表示一个时间点，初始时间从系统最近一次开机时间开始计算，而且在系统休眠时暂停计时，等系统恢复后继续计时，精确到纳秒（1/1000,000,000 秒）。可以直接使用 + 运算符设定延时，如果使用变量延时药使用 TimeInterval 类型：

```
DispatchTime.now() // 表示当前时间与开机时间的间隔

let twoSecondAfter = DispatchTime.now() + 2.1 // 当前时间之后 2.1 秒
```

DispatchWallTime

它表示一个绝对时间的时间戳，可以直接使用字面量表示延时，也可以借用 `timespec` 结构体来表示，以微秒为单位（1/1000,000 秒）。

```
// 使用字面量设置
var wallTime = DispatchWallTime.now() + 2.0 // 表示从当前时间开始后 2 秒，数字字面量也可以改为使用 TimeInterval 类型变量

// 获取当前时间，以 timeval 结构体的方式表示
var getTimeval = timeval()
gettimeofday(&getTimeval, nil)

// 转换为 timespec 结构体
let time = timespec(tv_sec: __darwin_time_t(getTimeval.tv_sec), tv_nsec: Int(getTimeval.tv_usec * 1000))

// 转换为 DispatchWallTime
let wallTime = DispatchWallTime(timespec: time)
```

如何通过字符串字面量创建 DispatchWallTime 时间戳

首先需要做一些扩展：

```
extension Date {
    /// 通过字符串字面量创建 DispatchWallTime 时间戳
    ///
    /// - Parameter dateString: 时间格式字符串，如："2016-10-05 13:11:12"
    /// - Returns: DispatchWallTime 时间戳
    static func getWallTime(from dateString: String) -> DispatchWallTime? {
        let dateformatter = DateFormatter()
        dateformatter.dateFormat = "YYYY-MM-dd HH:mm:ss"
        dateformatter.timeZone = TimeZone(secondsFromGMT: 0)

        var newDate = dateformatter.date(from: dateString)

        guard let timeInterval = newDate?.timeIntervalSince1970 else {
            return nil
        }
        var time = timespec(tv_sec: __darwin_time_t(timeInterval), tv_nsec: 0)
        return DispatchWallTime(timespec: time)
    }
}
```

下面通过字符串即可创建时间戳：

```
let time = Date.getWallTime(from: "2018-03-08 13:30:00")
```

timespec

这是 `Darwin` 内核中的一个结构体，用于表示一个绝对时间点，它描述的是从格林威治时间 `1970年1月1日零点` 开始指定时间间隔后的时间点，精确到**纳秒**，结构如下：

```
struct timespec {
    __darwin_time_t tv_sec; // 表示时间的秒数
    long tv_nsec; // 表示时间的1秒内的部分（相当于小数部分），以纳秒为 1 个单位计数。
};

let time = timespec(tv_sec: __darwin_time_t(86400), tv_nsec: 10) // 表示 1970-1-2 号第 10 纳秒
```

timeval

这是 `Darwin` 内核中的一个结构体，也用于表示一个绝对时间点，它描述的是从格林威治时间 `1970年1月1日零点` 开始指定时间间隔后的时间点，精确到**微秒**，结构如下：

```
struct timeval {
    __darwin_time_t tv_sec; // 表示时间的秒数
    __darwin_suseconds_t tv_usec; // 表示时间的1秒内的部分（相当于小数部分），以微秒为 1 个单位计数。
};
```

gettimeofday

这是 `Unix` 系统中的一个获取当前时间的方法，它接收两个指针参数，执行后将修改指针对应的结构体值，一个参数为 `timeval` 类型的时间结构体指针，另一个为时区结构体指针（时区在此方法中已不再使用，设为 nil 即可）。方法返回 0 时表示获取成功，返回 -1 时表示获取失败：

```
var getTimeval = timeval() // 原始时间
let time = gettimeofday(&getTimeval, nil) // 再次读取 getTimeval 即为当前时间
```

问答习题

最后留下几个问题给大家思考。

队列与任务特性

1. 主队列只能使用主线程吗?
2. 串行队列可以使用多个线程吗?如果可以，可以同时使用多个线程吗?
3. 向主队列中提交同步任务会导致死锁吗?
4. 向串行队列中提交同步任务会导致死锁吗?

5. 向并行队列中提交同步任务会导致死锁吗?

扩展阅读

源码

官方 GCD Swift 源码

官方 Operation Swift 源码 (推荐看一下，更易懂好用的 Operation 类原来封装起来这么简单。)

鸣谢

本教程在撰写过程中，参考或从以下文章中获得灵感，感谢以下文章及作者的帮助：

- 行走的少年郎: iOS多线程：『GCD』详尽总结
- onevcat: Swifter – Swift 开发者必备 Tips – GCD 和延时调用
- nekno: Large file copy with GCD – Dispatch IO consumes large amounts of memory
- THE GRAND CENTRAL DISPATCH: SOURCES – EXAMPLES (TIMER)
- 戴铭: 细说GCD（Grand Central Dispatch）如何用
- bestswifter: 深入了解GCD
- 酸菜Amour:
    - GCD源码的分析
    - dispatch_sync 的分析
    - dispatch_async 的分析
- Sindri Lin: 奇怪的GCD
- GABRIEL THEODOROPOULOS(译者：小锅): Swift 3 中的 GCD 与 Dispatch Queue
- Florian Kugler(译者：破船): 并发编程：API 及挑战
- Mike Ash: Intro to Grand Central Dispatch, Part IV: Odds and Ends
- Dave Rahardja: GCD Target Queues
