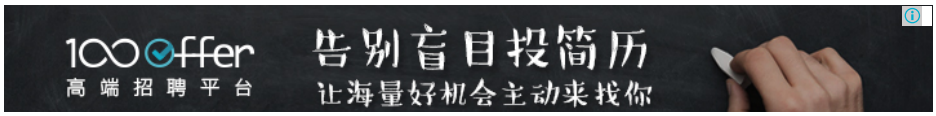


GCD 最佳实践指南

2016-05-11 13:22 编辑：cocopeng 分类：iOS开发 来源：chengway.in 2 1142

GCD

招聘信息：java高级软件工程师

英文原文：The GCD Handbook，译者：walkingway（微博）

Grand Central Dispatch 大中枢派发或俗称 GCD 是一件极其强大的武器。他为你提供了很多底层工具，比如队列和信号量，你可以组合这些工具来达成自己想要的多线程效果。不幸的是，这些基于 C 的 API 晦涩难懂，并且想要将低级工具组合起来实现高抽象层级 API 的效果（译者注：类似于 NSOperation）也不是一件容易的事。这篇文章，我来教大家如何利用 GCD 提供的工具来达成高抽象层级的行为。

在后台工作

这或许是 GCD 提供的最简单的工具了，他能让你在后台线程处理一些工作，然后返回主线程继续，就如同 UIKit 的操作只能在主线程中进行。

在本指南中，我将使用 doSomeExpensiveWork() 函数来表示一些长期运行的任务，并最终返回一个结果。

根据这一思路，我们的套路一般是：

```
let defaultPriority = DISPATCH_QUEUE_PRIORITY_DEFAULT
let backgroundQueue = dispatch_get_global_queue(defaultPriority, 0)
dispatch_async(backgroundQueue, {
    let result = doSomeExpensiveWork()
    dispatch_async(dispatch_get_main_queue(), {
        //use `result` somehow
    })
})
```

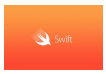
在实际项目中，除了 DISPATCH_QUEUE_PRIORITY_DEFAULT，我们几乎不使用其他的优先级选项。dispatch_get_global_queue() 将返回一个队列，支持数百个线程的执行。如果你总是需要在一个后台队列上执行开销庞大的操作，那么可以使用 dispatch_queue_create 创建自己的队列，dispatch_queue_create 带两个参数，第一个是需要指定的队列名，第二个说明是串行队列还是并发队列。

热门资讯


iOS开发——你真的会用SDWebImage?
点击量 10987


iOS本地数据存取，看这里就够了
点击量 8203


聊聊魔性的动画引擎pop
点击量 7703


10个惊艳的Swift单行代码
点击量 6588


神奇的 BlocksKit（一）
点击量 6210


两招让你成为很厉害的T型人才
点击量 5645


iOS网络层架构设计分享
点击量 5526


Swift 3.0 预告：将 Objc 库转换成更符合
点击量 5385


70%的人离职因领导这4宗罪
点击量 5338


【零基础必备】超实用的动效设计入门小手册
点击量 5313

综合评论

随便逛逛，长点见识。
646773295 评论了 牛叉的技术经理，也应当是牛叉的调试者...

最近有几次时间的确缩短了，隔天就出结果了，但也有好几天的
646773295 评论了 App Store应用审查缩至24小时...

自己举办，自己拿奖，不知道其他的参赛人员什么感受。。。
盛夏光年 评论了 48小时Cocos VR黑客松圆满落幕 五大奖项揭...

深深的不信，排队都要排一周
也嘉 评论了 App Store应用审查缩至24小时...

深深的不信，排队都要排一周
也嘉 评论了 App Store应用审查缩至

注意每次调用使用的是 dispatch_async 而不是 dispatch_sync。dispatch_async 将在 block 执行前立即返回，而 dispatch_sync 则会等到 block 执行完毕后才返回。内部的调用可以使用 dispatch_sync（因为不在乎什么时候返回），但是外部的调用必须是 dispatch_async（否则主线程会被阻塞）。

创建单例

dispatch_once 这个 API 可以用来创建单例。不过这种方式在 Swift 已不再重要，Swift 有更简单的方法来创建单例。我这里就只贴 OC 的实现：

```
+ (instancetype) sharedInstance {
    static dispatch_once_t onceToken;
    static id sharedInstance;
    dispatch_once(&onceToken, ^{
        sharedInstance = [[self alloc] init];
    });
    return sharedInstance;
}
```

将 completion block 扁平化

至此我们的 GCD 之旅开始变得有趣起来。我们可以使用信号量来阻塞一个线程任意时长，直到一个信号从另一个线程发出。信号量和 GCD 的其他部分一样也是线程安全的，而且能够从任意位置被触发。

信号量适用于当你有一个异步 API 需要同步执行的情形，但你不能修改它

```
// on a background queue
dispatch_semaphore_t semaphore = dispatch_semaphore_create(0)
doSomeExpensiveWorkAsynchronously(completionBlock: {
    dispatch_semaphore_signal(semaphore)
})
dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER)
//the expensive asynchronous work is now done
```

调用 dispatch_semaphore_wait 将会阻塞线程直到 dispatch_semaphore_signal 被调用。这就意味着 signal 必须从不同的线程被调用，因为当前线程是被完全阻塞的。更进一步，你从来都不该在主线程中调用 wait，而是只从后台线程中调用。

你可以在调用 dispatch_semaphore_wait 时设置一个超时时间，但是我总是趋向使用 DISPATCH_TIME_FOREVER

为什么在已经有 completion block 的情况下，还想要摊平代码？因为方便呀，我能想到的一种场景是执行一组异步程序，但他们必须串行执行（即只有前一个任务执行完成，才会继续执行下一个任务）。现在把上述想法简单地抽象成一个 AsyncSerialWorker 示例：

```
typealias DoneBlock = () -> ()
typealias WorkBlock = (DoneBlock) -> ()
class AsyncSerialWorker {
    private let serialQueue = dispatch_queue_create("com.khanlou.serial.queue", DISPATC
    func enqueueWork(work: WorkBlock) {
        dispatch_async(serialQueue) {
            let semaphore = dispatch_semaphore_create(0)
            work({
                dispatch_semaphore_signal(semaphore)
            })
            dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER)
        }
    }
}
```

上面这个简短的类创建了一个串行队列，允许你将 work 的入队列操作放进 block 中。在 WorkBlock 需要一个 DoneBlock 作为参数，而 DoneBlock 会在当前工作结束时被执行，我们通过将 DoneBlock 设置为 {dispatch_semaphore_signal(semaphore)} 来调整信号量，从而让串行队列继续执行下去。

译者注：既然已经使用了 DISPATCHQUEUESERIAL，那么队列中 work 的执行顺序不应该是先进先出的吗？确实是这样，但如果我们把 work 看成是一个耗时的网络操作，其内部是提交到其他线程并发去执行（dispatch_async），也就是每次执行到 work 就立刻返回了，即使最终结果可能还未返回。那么我们想要保证队列中的 work 等到前一个 work 执行返回结果后才执行，就需要 semaphore。说了这么多还是举个例子吧，打开 Playground：

```
import UIKit
import XCPlayground
typealias DoneBlock = () -> ()
typealias WorkBlock = (DoneBlock) -> ()
class AsyncSerialWorker {
```

24小时...

依旧看不明白
向宏simida 评论了 GCD 最佳实践指南

路过
feifei_Smile 评论了 48小时Cocos VR 黑客松圆满落幕 五大奖项揭...

受益匪浅，thanks a lot
starmo2004 评论了 iOS应用架构谈view层的组织和调用方案...

请问怎么打开啊?
兰云阁 评论了 Masonry介绍与使用实践：快速上手Autola...

这标题……
zagger 评论了 App Store应用审查缩至24小时...

相关帖子

非常诡异的问题:_NSZombie_XXXX（想挑战一下的进来）

自定义NSLog 无法打印 怎么破

ios app强制更新破解

如何将APP上传到第三方市场

数据

添加沙箱技术测试者 邮件验证失败

iOS 实现手指涂抹,涂抹位置形成马赛克

使用 GCDAsyncSocket 怎么接收服务器主动发过来的消息

求大神帮忙 正态分布图！！

微博

CocoaChina

加关注

［源码推荐(05.06)］1.一句代码搞定毛玻璃特效。2.自定义弹出视图。3.相册/图片浏览。以上源码下载：http://t.cn/RqRbhPH

5月6日 16:18 转发(69) | 评论(25)

[为设计师精心打造的14款进阶移动端应用]越来越强大的智能手机有时候会让我们的电脑相形见绌，也许一些大型复杂的设计项目非得在电脑上完成，但是许多日常的、简单的甚至是专业的设计工作，手机会比电脑更便捷

```
    private let serialQueue = dispatch_queue_create("com.khanlou.serial.queue", DISPATC
    func enqueueWork(work: WorkBlock) {
        dispatch_async(serialQueue) {
            let semaphore = dispatch_semaphore_create(0)
            work({
                dispatch_semaphore_signal(semaphore)
            })
            dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER)
        }
    }
}
let a = AsyncSerialWorker()
for i in 1...5 {
    a.enqueueWork { doneBlock in
        dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)) {
            sleep(arc4random_uniform(4)+1)
            print(i)
            doneBlock()
        }
    }
}
XCPlaygroundPage.currentPage.needsIndefiniteExecution = true
```

此时的输出结果为：1，2，3，4，5，如果将关于 semaphore 的代码都注释掉，结果就不会是按顺序输出了。

dispatch_semaphore_create(0) 当两个线程需要协调处理某个事件时，我们在这里传入 0；内部其实是维护了一个计数器，我们下面会说到。

限制并发的数量

在上面的例子中，信号量被用来当做是一个简单的标志，但它也可以当成一个有限资源的计数器。如果你想针对某些特定的资源限制连接数，你可以这样做：

```
class LimitedWorker {
    private let concurrentQueue = dispatch_queue_create("com.khanlou.concurrent.queue",
    private let semaphore: dispatch_semaphore_t
    init(limit: Int) {
        semaphore = dispatch_semaphore_create(limit)
    }
    func enqueueWork(work: () -> ()) {
        dispatch_async(concurrentQueue) {
            dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER)
            work()
            dispatch_semaphore_signal(semaphore)
        }
    }
}
```

这个例子来自于苹果官方的多线程编程指南，官方给出的解释如下：

在创建信号量的时候，可以限定资源的可用数。这个可用数（long 类型）会随信号量初始化时作为参数传入。每次等待信号量时，dispatch_semaphore_wait 都会消耗一次这个可用数，如果结果为负，函数会告诉内核阻断你的线程。另一方面，dispatch_semaphore_signal 函数每次执行都会将该可用计数 + 1，以此来表明已经释放了资源。如果此刻有因为等待可用资源而被阻隔的任务，系统会从等待的队列中解锁一个任务来执行。

这个效果类似于 NSOperationQueue 的 maxConcurrentOperationCount。如果你使用原生的 GCD 队列而不是 NSOperationQueue，你就能使用信号量来限制并发任务的数量。

值得注意是：每次调用 enqueueWork 都会将 work 提交到一个并发队列，而该并发队列收到任务就会丢出去执行，直到触碰到信号量数量耗尽的天花板（work 入队列的速度太快，dispatch_semaphore_wait 已经消耗完了所有的数量，而之前的 work 还未执行完毕，dispatch_semaphore_signal 不能增加信号量的可用数量）

等待许多并发任务去完成

如果你有许多 blocks 任务要去执行，你需要在他们全部完成时得到通知，你可以使用 group。dispatch_group_async 允许你在队列中添加任务（这些任务应该是同步执行的），而且你会追踪有多少被添加的任务。注意：同一个 dispatch group 能够添加不同队列上的任务，并且能保持对所有组内任务的追踪。当所有被追踪的任务完成时，一个传递给 dispatch_group_notify 的 block 会被触发执行，有点类似于 completion block

```
dispatch_group_t group = dispatch_group_create()
for item in someArray {
    dispatch_group_async(group, backgroundQueue, {
        performExpensiveWork(item: item)
    })
}
```

苹果Genius考什么证书?
ACSP-苹果认证工程师证书 立即申请免费的苹果ACSP认证考试资料

```
dispatch_group_notify(group, dispatch_get_main_queue(), {
    // all the work is complete
}
```

对于摊平一个拥有 completion block 的函数来说，下面是一个绝佳的示例。分发组（The dispatch group）认为整个 block 将在返回时完成，所以 block 会一直等到任务完成时才会返回

还有更多手动使用 dispatch groups 的姿势，尤其有很多异步执行的大开销任务：

```
// must be on a background thread
dispatch_group_t group = dispatch_group_create()
for item in someArray {
    dispatch_group_enter(group)
    performExpensiveAsyncWork(item: item, completionBlock: {
        dispatch_group_leave(group)
    })
}
dispatch_group_wait(group, DISPATCH_TIME_FOREVER)
// all the work is complete
```

这段代码更加复杂一些，但是我们逐行去看还是可以理解的。如同信号量，groups 同样保持着一个线程安全的、可以操控的内部计数器。你可以使用这个计数器来确保在 completion block 执行前，多个大开销任务都已执行完毕。使用 enter 来增加计数器，使用 leave 来减少计数器。dispatch_group_async 已为你处理了这些细节，所以尽情地享受即可。

代码片段的最后一行是 wait 调用：他会阻断当前线程并且等待计数器达到 0 才会继续执行。注意尽管你使用了 enter/leave API，但你还是能够通过 dispatch_group_notify 将 block 提交到队列中。反过来也是成立的：如果你用了 dispatch_group_async API，也是能够使用 dispatch_group_wait 的。

dispatch_group_wait 和 dispatch_semaphore_wait 一样都接收一个超时参数。再次重申，我更喜欢 DISPATCH_TIME_FOREVER，同样的不要在主线程中调用 dispatch_group_wait。

上面两段代码最大的不同在于使用 notify 可以完全从主线程上被调用，而使用 wait 的只能在后台线程上使用（至少是 wait 部分，因为他会完全阻塞当前线程）

隔离队列

Swift 中的字典（和数组）都是值类型，当他们被修改时，他们的引用会被一个新的副本所替代。但是，因为更新 Swift 对象的实例变量操作并不是原子性的，所以这些操作也不是线程安全的。如果两个线程同一时间更新一个字典（比如都添加一个值），而且这两个操作都尝试写同一块内存，这就会导致内存崩坏，我们可以使用隔离队列来达成线程安全的目的。

先来构建一个标识映射Identity Map，一个标识映射是一个字典，代表着从 ID 到 model 对象的映射

Identity Map（标识映射）模式是通过将所有已加载对象放在一个映射中确保所有对象只被加载一次，并且在引用这些对象时使用该映射来查找对象。在处理数据并发访问时，要有一种策略让多个用户共同影响同一个业务实体，这个固然很重要。同样重要的是，单个用户在一个长运行事务或复杂事务中始终使用业务实体的一致版本。Identity Map 模式提供的功能；为事务中使用所有的业务对象均保存一个版本，如果一个实体被请求两次，返回同一个实体。

```
class IdentityMap {
    var dictionary = Dictionary()
    func object(forID ID: String) -> T? {
        return dictionary[ID] as T?
    }
    func addObject(object: T) {
        dictionary[object.ID] = object
    }
}
```

这个对象基本就是一个字典封装器，如果有多个线程在同一时刻调用函数 addObject，内存将会崩坏，因为线程会操作相同的引用。这也是操作系统中的经典的读者-写者问题，简而言之，我们可以在同一时刻有多个读者，但同一时刻只能有一个线程可以写入。

幸运的是 GCD 针对在该场景下同样拥有强力武器，有如下四种 APIs 供我们选用：

- dispatch_sync
- dispatch_async

- dispatch_barrier_sync
- dispatch_barrier_async

我们的想法是读操作可以支持同步和异步，而写操作也能支持异步写入，并且必须确保是写入的是同一个引用。GCD 的 barrier 集合 APIs 提供了解决方案：他们将会一直等到队列中的任务清空，才会继续执行 block。使用 barrier APIs 可以用来限制我们对字典对象的写入，并且确保我们不会在同一时刻进行多个写操作，以及正在写操作时同时进行读操作。

```
class IdentityMap {
    var dictionary = Dictionary()
    let accessQueue = dispatch_queue_create("com.khanlou.isolation.queue", DISPATCH_QUE
    func object(withID ID: String) -> T? {
        var result: T? = nil
        dispatch_sync(accessQueue, {
            result = dictionary[ID] as T?
        })
        return result
    }
    func addObject(object: T) {
        dispatch_barrier_async(accessQueue, {
            dictionary[object.ID] = object
        })
    }
}
```

dispatch_sync 将会分发 block 到我们的隔离队列上然后等待其执行完毕。通过这种方式，我们就实现了同步读操作（如果我们搞成异步读取，getter 方法就需要一个 completion block）。因为 accessQueue 是并发队列，这些同步读取操作可以并发执行，也就是允许同时读。

dispatch_barrier_async 将分发 block 到隔离队列上，async 异步部分意味着会立即返回，并不会等待 block 执行完毕。这对性能是有好处的，但是在一个写操作后立即执行一个读操作会导致读到一个半成品的数据（因为可能写操作还未完成就开始读了）。

dispatch_barrier_async 中的 barrier 部分意味着：只要 barrier block 进入队列，并不会立即执行，而是会等待该队列其他 block 执行完毕后再执行。所以在这点上就保证了我们的 barrier block 每次都只有它自己在执行。而所有在他之后提交的 block 也会一直等待这个 barrier block 执行完再执行。

传入 dispatch_barrier_async() 函数的 queue，必须是用 dispatch_queue_create 创建的并发 queue。如果是串行 queue 或者是 global concurrent queues，这个函数就会变成 dispatch_async() 了

完结

GCD 是一个具备底层特性的框架，通过他们，我们可以构建高层级的抽象行为。如果还有一些我没提到的高层级的行为可以用 GCD 来构建，欢迎来交流。

微信扫一扫

订阅每日移动开发及APP推广热点资讯

公众号：CocoaChina

我要投稿 收藏文章 分享到：

小笨狼漫谈多线程：GCD(一)
谈iOS多线程(NSThread、NSOperation、GCD)编程
知其然亦知其所以然--NSOperation并发编程
Grand Central Dispatch 基础教程：Part 2/2
GCD使用经验与技巧浅谈
iOS并行开发：从NSOperation和调度队列开始
GCD使用三部曲之：基本用法
深入理解dispatch_queue
Grand Central Dispatch 基础教程：Part 1/2
为GCD队列绑定NSObject类型上下文数据-利用

苹果Genius考什么证书?
ACSP-苹果认证工程师证书 立即申请免费的苹果ACSP认证考试资料

我来说两句

您还没有登录！请 登录 或 注册

所有评论（2）

向宏simida 刚刚
依旧看不明白
0 0 回复

asdasdaa 刚刚
一个愿意留下字的路人
0 0 回复

关于我们 商务合作 联系我们 合作伙伴
北京触控科技有限公司版权所有
©2016 Chukong Technologies,Inc.
京ICP备 11006519号 京ICP证 100954号 京公网安备11010502020289 京网文[2012]0426-138号