

🔗 branch: master ▾ dev-blog / 2014-04-19-grand-central-dispatch-in-depth-part-1.md

 100mango on 10 Jan 修改杂波为杂乱的工作

3 contributors 

588 lines (384 sloc) 37.164 kb

RawBlameHistory📄✎🗑

GCD 深入理解：第一部分

本文翻译自 <http://www.raywenderlich.com/60749/grand-central-dispatch-in-depth-part-1>

原作者：Derek Selander

译者：@nixzhu

虽然 GCD 已经出现过一段时间了，但不是每个人都明了其主要内容。这是可以理解的；并发一直很棘手，而 GCD 是基于 C 的 API，它们就像一组尖锐的棱角截进 Objective-C 的平滑世界。我们将分两个部分的教程来深入学习 GCD。

在这两部分的系列中，第一个部分的将解释 GCD 是做什么的，并从许多基本的 GCD 函数中找出几个来展示。在第二部分，你将学到几个 GCD 提供的高级函数。

什么是 GCD

GCD 是 libdispatch 的市场名称，而 libdispatch 作为 Apple 的一个库，为并发代码在多核硬件（跑 iOS 或 OS X）上执行提供有力支持。它具有以下优点：

- GCD 能通过推迟昂贵计算任务并在后台运行它们来改善你的应用的响应性能。
- GCD 提供一个易于使用的并发模型而不仅仅只是锁和线程，以帮助我们避开并发陷阱。
- GCD 具有在常见模式（例如单例）上用更高性能的原语优化你的代码的潜在能力。

本教程假设你对 Block 和 GCD 有基础了解。如果你对 GCD 完全陌生，先看看 [iOS 上的多线程](#)和[GCD 入门教程](#) 学习其要领。

GCD 术语

要理解 GCD，你要先熟悉与线程和并发相关的几个概念。这两者都可能模糊和微妙，所以在开始 GCD 之前先简要地回顾一下它们。

Serial vs. Concurrent 串行 vs. 并发

这些术语描述当任务相对于其它任务被执行，任务串行执行就是每次只有一个任务被执行，任务并发执行就是在同一时间可以有多个任务被执行。

虽然这些术语被广泛使用，本教程中你可以将任务设定为一个 Objective-C 的 Block。不明白什么是 Block？看看 [iOS 5 教程中的如何使用 Block](#)。实际上，你也可以在 GCD 上使用函数指针，但在大多数场景中，这实际上更难于使用。Block 就是更加容易些！

Synchronous vs. Asynchronous 同步 vs. 异步

在 GCD 中，这些术语描述当一个函数相对于另一个任务完成，此任务是该函数要求 GCD 执行的。一个同步函数只在完成

了它预定的任务后才返回。

一个异步函数，刚好相反，会立即返回，预定的任务会完成但不会等它完成。因此，一个异步函数不会阻塞当前线程去执行下一个函数。

注意——当你读到同步函数“阻塞（Block）”当前线程，或函数是一个“阻塞”函数或阻塞操作时，不要被搞糊涂了！动词“阻塞”描述了函数如何影响它所在的线程而与名词“代码块（Block）”没有关系。代码块描述了用 Objective-C 编写的一个匿名函数，它能定义一个任务并被提交到 GCD。

译者注：中文不会有这个问题，“阻塞”和“代码块”是两个词。

Critical Section 临界区

就是一段代码不能被并发执行，也就是，两个线程不能同时执行这段代码。这很常见，因为代码去操作一个共享资源，例如一个变量若能被并发进程访问，那么它很可能会变质（译者注：它的值不再可信）。

Race Condition 竞态条件

这种状况是指基于特定序列或时机的事件的软件系统以不受控制的方式运行的行为，例如程序的并发任务执行的确切顺序。竞态条件可导致无法预测的行为，而不能通过代码检查立即发现。

Deadlock 死锁

两个（有时更多）东西——在大多数情况下，是线程——所谓的死锁是指它们都卡住了，并等待对方完成或执行其它操作。第一个不能完成是因为它在等待第二个的完成。但第二个也不能完成，因为它在等待第一个的完成。

Thread Safe 线程安全

线程安全的代码能在多线程或并发任务中被安全的调用，而不会导致任何问题（数据损坏，崩溃，等）。线程不安全的代码在某个时刻只能在一个上下文中运行。一个线程安全代码的例子是 `NSDictionary`。你可以在同一时间在多个线程中使用它而不会有问題。另一方面，`NSMutableDictionary` 就不是线程安全的，应该保证一次只能有一个线程访问它。

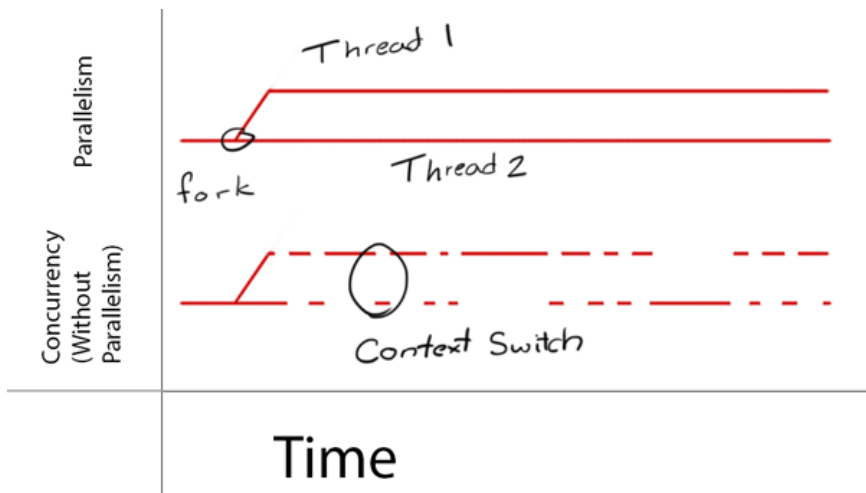
Context Switch 上下文切换

一个上下文切换指当你在单个进程里切换执行不同的线程时存储与恢复执行状态的过程。这个过程在编写多任务应用时很普遍，但会带来一些额外的开销。

Concurrency vs Parallelism 并发与并行

并发和并行通常被一起提到，所以值得花些时间解释它们之间的区别。

并发代码的不同部分可以“同步”执行。然而，该怎样发生或是否发生都取决于系统。多核设备通过并行来同时执行多个线程；然而，为了使单核设备也能实现这一点，它们必须先运行一个线程，执行一个上下文切换，然后运行另一个线程或进程。这通常发生地足够快以致给我们并发执行地错觉，如下图所示：



虽然你可以编写代码在 GCD 下并发执行，但 GCD 会决定有多少并行的需求。并行要求并发，但并发并不能保证并行。

更深入的观点是并发实际上是关于构造。当你在脑海中用 GCD 编写代码，你组织你的代码来暴露能同时运行的多个工作片段，以及不能同时运行的那些。如果你想深入此主题，看看 [这个由Rob Pike做的精彩的讲座](#)。

Queues 队列

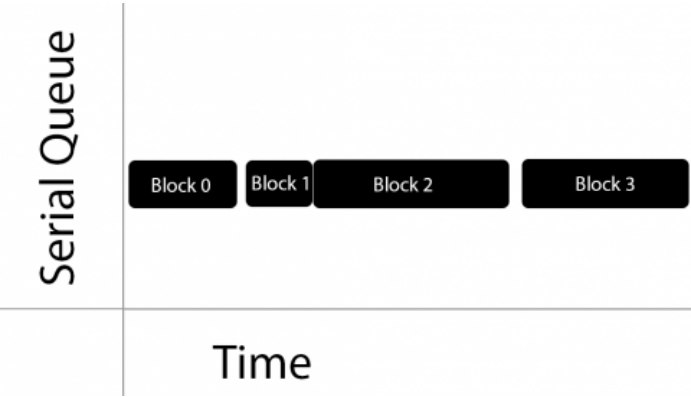
GCD 提供有 `dispatch queues` 来处理代码块，这些队列管理你提供给 GCD 的任务并用 FIFO 顺序执行这些任务。这就保证了第一个被添加到队列里的任务会是队列中第一个开始的任务，而第二个被添加的任务将第二个开始，如此直到队列的终点。

所有的调度队列（`dispatch queues`）自身都是线程安全的，你能从多个线程并行的访问它们。当你了解了调度队列如何为你自己代码的不同部分提供线程安全后，GCD 的优点就是显而易见的。关于这一点的关键是选择正确类型的调度队列和正确的调度函数来提交你的工作。

在本节你会看到两种调度队列，都是由 GCD 提供的，然后看一些描述如何用调度函数添加工作到队列的例子。

Serial Queues 串行队列

串行队列中的任务一次执行一个，每个任务只在前一个任务完成时才开始。而且，你不知道在一个 Block 结束和下一个开始之间的时间长度，如下图所示：



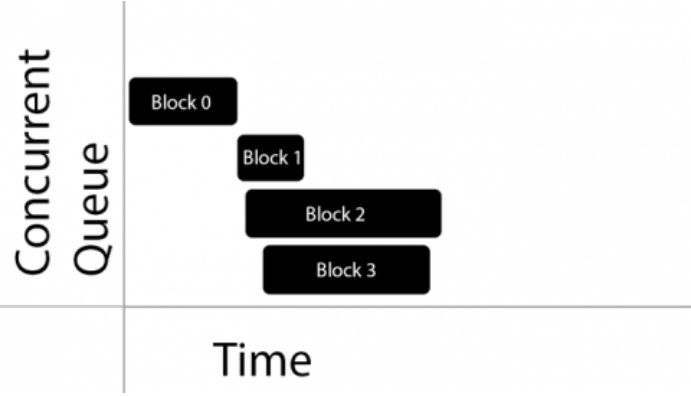
这些任务的执行时机受到 GCD 的控制；唯一能确保的事情是 GCD 一次只执行一个任务，并且按照我们添加到队列的顺序来执行。

由于在串行队列中不会有两个任务并发运行，因此不会出现同时访问临界区的风险；相对于这些任务来说，这就从竞态条件下保护了临界区。所以如果访问临界区的唯一方式是通过提交到调度队列的任务，那么你就不需要担心临界区的安全问题了。

Concurrent Queues 并发队列

在并发队列中的任务能得到的保证是它们会按照被添加的顺序开始执行，但这就是全部的保证了。任务可能以任意顺序完成，你不会知道何时开始运行下一个任务，或者任意时刻有多少 Block 在运行。再说一遍，这完全取决于 GCD。

下图展示了一个示例任务执行计划，GCD 管理着四个并发任务：



注意 Block 1, 2 和 3 都立马开始运行，一个接一个。在 Block 0 开始后，Block 1 等待了好一会儿才开始。同样，Block 3 在 Block 2 之后才开始，但它先于 Block 2 完成。

何时开始一个 Block 完全取决于 GCD。如果一个 Block 的执行时间与另一个重叠，也是由 GCD 来决定是否将其运行在另一个不同的核心上，如果那个核心可用，否则就用上下文切换的方式来执行不同的 Block。

有趣的是，GCD 提供给你至少五个特定的队列，可根据队列类型选择使用。

Queue Types 队列类型

首先，系统提供给你一个叫做 主队列 (main queue) 的特殊队列。和其它串行队列一样，这个队列中的任务一次只能执行一个。然而，它能保证所有的任务都在主线程执行，而主线程是唯一可用于更新 UI 的线程。这个队列就是用于发生消息给 UIView 或发送通知的。

系统同时提供给你好几个并发队列。它们叫做 全局调度队列 (Global Dispatch Queues)。目前的四个全局队列有着不同的优先级：background、low、default 以及 high。要知道，Apple 的 API 也会使用这些队列，所以你添加的任何任务都不会是这些队列中唯一的任务。

最后，你也可以创建自己的串行队列或并发队列。这就是说，至少有五个队列任你处置：主队列、四个全局调度队列，再加上任何你自己创建的队列。

以上是调度队列的大框架！

GCD 的“艺术”归结为选择合适的队列来调度函数以提交你的工作。体验这一点的最好方式是走一遍下边的例子，我们沿途会提供一些一般性的建议。

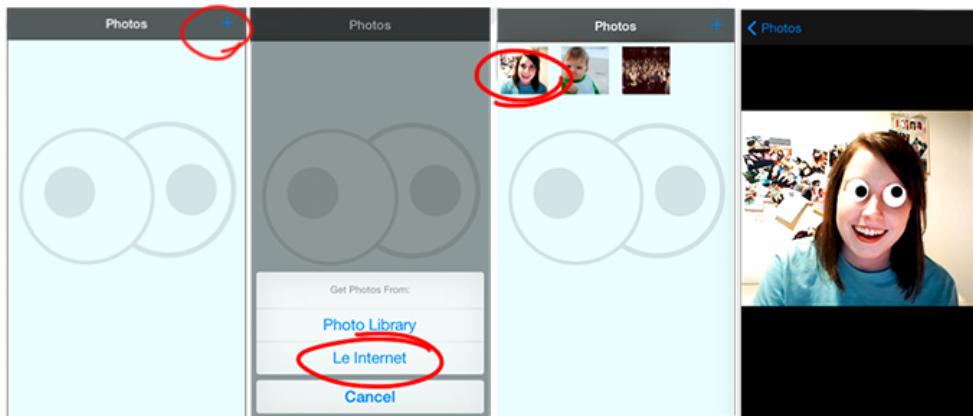
入门

既然本教程的目标是优化且安全的使用 GCD 调用来自不同线程的代码，那么你将从一个近乎完成的叫做 GooglyPuff 的项目入手。

GooglyPuff 是一个没有优化，线程不安全的应用，它使用 Core Image 的人脸检测 API 来覆盖一对曲棍球眼睛到被检测到的人脸上。对于基本的图像，可以从相机胶卷选择，或用预设好的 URL 从互联网下载。

[点击此处下载项目](#)

完成项目下载之后，将其解压到某个方便的目录，再用 Xcode 打开它并编译运行。这个应用看起来如下图所示：



注意当你选择 Le Internet 选项下载图片时，一个 UIAlertView 过早地弹出。你将在本系列教程地第二部分修复这个问题。

这个项目中有四个有趣的类：

- PhotoCollectionViewController：它是应用开始的第一个视图控制器。它用缩略图展示所有选定的照片。
- PhotoDetailViewController：它执行添加曲棍球眼睛到图像上的逻辑，并用一个 UIScrollView 来显示结果图片。
- Photo：这是一个类簇，它根据一个 NSURL 的实例或一个 ALAsset 的实例来实例化照片。这个类提供一个图像、缩略图以及从 URL 下载的状态。
- PhotoManager：它管理所有 Photo 的实例。

用 dispatch_async 处理后台任务

回到应用并从你的相机胶卷添加一些照片或使用 `Le Internet` 选项下载一些。

注意在按下 `PhotoCollectionViewController` 中的一个 `UICollectionViewCell` 到生成一个新的 `PhotoDetailViewController` 之间花了多久时间；你会注意到一个明显的滞后，特别是在比较慢的设备上查看很大的图。

在重载 `UIViewController` 的 `viewDidLoad` 时容易加入太多杂乱的工作（too much clutter），这通常会引起视图控制器出现前更长的等待。如果可能，最好是卸下一些工作放到后台，如果它们不是绝对必须要运行在加载时间里。

这听起来像是 `dispatch_async` 能做的事情！

打开 `PhotoDetailViewController` 并用下面的实现替换 `viewDidLoad`：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    NSAssert(!_image, @"Image not set; required to use view controller");
    self.photoImageView.image = _image;

    //Resize if necessary to ensure it's not pixelated
    if (_image.size.height <= self.photoImageView.bounds.size.height &&
        _image.size.width <= self.photoImageView.bounds.size.width) {
        [self.photoImageView setContentMode:UIViewContentModeCenter];
    }

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{ // 1
        UIImage *overlayImage = [self faceOverlayImageFromImage:_image];
        dispatch_async(dispatch_get_main_queue(), ^{ // 2
            [self fadeInNewImage:overlayImage]; // 3
        });
    });
}
```

下面来说明上面的新代码所做的事：

1. 你首先将工作从主线程移到全局线程。因为这是一个 `dispatch_async()`，Block 会被异步地提交，意味着调用线程地执行将会继续。这就使得 `viewDidLoad` 更早地在主线程完成，让加载过程感觉起来更加快速。同时，一个人脸检测过程会启动并将在稍后完成。
2. 在这里，人脸检测过程完成，并生成了一个新的图像。既然你要使用此新图像更新你的 `UIImageView`，那么你就添加一个新的 Block 到主线程。记住——你必须总是在主线程访问 `UIKit` 的类。
3. 最后，你用 `fadeInNewImage:` 更新 UI，它执行一个淡入过程切换到新的曲棍球眼睛图像。

编译并运行你的应用；选择一个图像然后你会注意到视图控制器加载明显变快，曲棍球眼睛稍微在之后就加上了。这给应用带来了不错的效果，和之前的显示差别巨大。

进一步，如果你试着加载一个超大的图像，应用不会在加载视图控制器上“挂住”，这就使得应用具有很好伸缩性。

正如之前提到的，`dispatch_async` 添加一个 Block 到队列就立即返回了。任务会在之后由 GCD 决定执行。当你需要在后台执行一个基于网络或 CPU 紧张的任务时就使用 `dispatch_async`，这样就不会阻塞当前线程。

下面是一个关于在 `dispatch_async` 上如何以及何时使用不同的队列类型的快速指导：

- 自定义串行队列：当你想串行执行后台任务并追踪它时就是一个好选择。这消除了资源争用，因为你知道一次只有一个任务在执行。注意若你需要来自某个方法的数据，你必须内联另一个 Block 来找回它或考虑使用 `dispatch_sync`。
- 主队列（串行）：这是在一个并发队列上完成任务后更新 UI 的共同选择。要这样做，你将在一个 Block 内部编写另一个 Block。以及，如果你在主队列调用 `dispatch_async` 到主队列，你能确保这个新任务将在当前方法完成后的某个时间执行。
- 并发队列：这是在后台执行非 UI 工作的共同选择。

使用 `dispatch_after` 延后工作

稍微考虑一下应用的 UX。是否用户第一次打开应用时会困惑于不知道做什么？你是这样吗？:]

如果用户的 `PhotoManager` 里还没有任何照片，那么显示一个提示会是个好主意！然而，你同样要考虑用户的眼睛会如何

在主屏幕上浏览：如果你太快的显示一个提示，他们的眼睛还徘徊在视图的其它部分上，他们很可能会错过它。

显示提示之前延迟一秒钟就足够捕捉到用户的注意，他们此时已经第一次看过了应用。

添加如下代码到 PhotoCollectionViewController.m 中 showOrHideNavPrompt 的废止实现里：

```
- (void)showOrHideNavPrompt
{
    NSUInteger count = [[PhotoManager sharedManager] photos].count;
    double delayInSeconds = 1.0;
    dispatch_time_t popTime = dispatch_time(DISPATCH_TIME_NOW, (int64_t)(delayInSeconds * NSEC_PER_SEC)); // 1
    dispatch_after(popTime, dispatch_get_main_queue(), ^(void){ // 2
        if (!count) {
            [self.navigationItem setPrompt:@"Add photos with faces to Googlyify them!"];
        } else {
            [self.navigationItem setPrompt:nil];
        }
    });
}
```

showOrHideNavPrompt 在 viewDidLoad 中执行，以及 UICollectionView 被重新加载的任何时候。按照注释数字顺序看看：

1. 你声明了一个变量指定要延迟的时长。
2. 然后等待 delayInSeconds 给定的时长，再异步地添加一个 Block 到主线程。

编译并运行应用。应该有一个轻微地延迟，这有助于抓住用户的注意力并展示所要做的事情。

dispatch_after 工作起来就像一个延迟版的 dispatch_async。你依然不能控制实际的执行时间，且一旦 dispatch_after 返回也就不能再取消它。

不知道何时适合使用 dispatch_after ？

- 自定义串行队列：在一个自定义串行队列上使用 dispatch_after 要小心。你最好坚持使用主队列。
- 主队列（串行）：是使用 dispatch_after 的好选择；Xcode 提供了一个不错的自动完成模版。
- 并发队列：在并发队列上使用 dispatch_after 也要小心；你会这样做就比较罕见。还是在主队列做这些操作吧。

让你的单例线程安全

单例，不论喜欢还是讨厌，它们在 iOS 上的流行情况就像网上的猫。:]

一个常见的担忧是它们常常不是线程安全的。这个担忧十分合理，基于它们的用途：单例常常被多个控制器同时访问。

单例的线程担忧范围从初始化开始，到信息的读和写。PhotoManager 类被实现为单例——它在目前的状态下就会被这些问题所困扰。要看看事情如何很快地失去控制，你将在单例实例上创建一个控制好的竞态条件。

导航到 PhotoManager.m 并找到 sharedManager；它看起来如下：

```
+ (instancetype)sharedManager
{
    static PhotoManager *sharedPhotoManager = nil;
    if (!sharedPhotoManager) {
        sharedPhotoManager = [[PhotoManager alloc] init];
        sharedPhotoManager->_photosArray = [NSMutableArray array];
    }
    return sharedPhotoManager;
}
```

当前状态下，代码相当简单；你创建了一个单例并初始化一个叫做 photosArray 的 NSMutableArray 属性。

然而，if 条件分支不是线程安全的；如果你多次调用这个方法，有一个可能性是在某个线程（就叫它线程A）上进入 if 语句块并可能在 sharedPhotoManager 被分配内存前发生一个上下文切换。然后另一个线程（线程B）可能进入 if，分配单例实例的内存，然后退出。

当系统上下文切换回线程A，你会分配另外一个单例实例的内存，然后退出。在那个时间点，你有了两个单例的实例——很明显这不是你想要的（译者注：这还能叫单例吗？）！

要强制这个（竞态）条件发生，替换 PhotoManager.m 中的 sharedManager 为下面的实现：

```
+ (instancetype)sharedManager
{
    static PhotoManager *sharedPhotoManager = nil;
    if (!sharedPhotoManager) {
        [NSThread sleepForTimeInterval:2];
        sharedPhotoManager = [[PhotoManager alloc] init];
        NSLog(@"Singleton has memory address at: %@", sharedPhotoManager);
        [NSThread sleepForTimeInterval:2];
        sharedPhotoManager->_photosArray = [NSMutableArray array];
    }
    return sharedPhotoManager;
}
```

上面的代码中你用 NSThread 的 sleepForTimeInterval: 类方法来强制发生一个上下文切换。

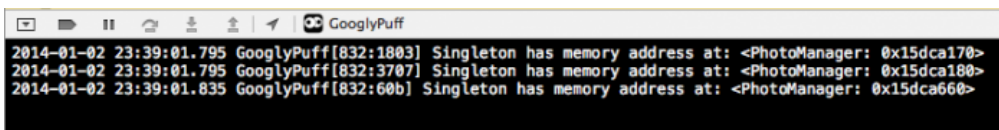
打开 AppDelegate.m 并添加如下代码到 application:didFinishLaunchingWithOptions: 的最开始处：

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{
    [PhotoManager sharedManager];
});

dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{
    [PhotoManager sharedManager];
});
```

这里创建了多个异步并发调用来实例化单例，然后引发上面描述的竞态条件。

编译并运行项目；查看控制台输出，你会看到多个单例被实例化，如下所示：



```
2014-01-02 23:39:01.795 GooglyPuff[832:1803] Singleton has memory address at: <PhotoManager: 0x15dca170>
2014-01-02 23:39:01.795 GooglyPuff[832:3707] Singleton has memory address at: <PhotoManager: 0x15dca180>
2014-01-02 23:39:01.835 GooglyPuff[832:60b] Singleton has memory address at: <PhotoManager: 0x15dca660>
```

注意到这里有好几行显示着不同地址的单例实例。这明显违背了单例的目的，对吧？:]

这个输出向你展示了临界区被执行多次，而它只应该执行一次。现在，固然是你自己强制这样的状况发生，但你可以想像一下这个状况会怎样在无意间发生。

注意：基于其它你无法控制的系统事件，NSLog 的数量有时会显示多个。线程问题极其难以调试，因为它们往往难以重现。

要纠正这个状况，实例化代码应该只执行一次，并阻塞其它实例在 if 条件的临界区运行。这刚好就是 dispatch_once 能做的事。

在单例初始化方法中用 dispatch_once 取代 if 条件判断，如下所示：

```
+ (instancetype)sharedManager
{
    static PhotoManager *sharedPhotoManager = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        [NSThread sleepForTimeInterval:2];
        sharedPhotoManager = [[PhotoManager alloc] init];
        NSLog(@"Singleton has memory address at: %@", sharedPhotoManager);
        [NSThread sleepForTimeInterval:2];
        sharedPhotoManager->_photosArray = [NSMutableArray array];
    });
    return sharedPhotoManager;
}
```

编译并运行你的应用；查看控制台输出，你会看到有且仅有一个单例的实例——这就是你对单例的期望！:]

现在你已经明白了防止竞态条件的重要性，从 AppDelegate.m 中移除 dispatch_async 语句，并用下面的实现替换 PhotoManager 单例的初始化：

```
+ (instancetype)sharedManager
{
    static PhotoManager *sharedPhotoManager = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedPhotoManager = [[PhotoManager alloc] init];
        sharedPhotoManager->_photosArray = [NSMutableArray array];
    });
    return sharedPhotoManager;
}
```

dispatch_once() 以线程安全的方式执行且仅执行其代码块一次。试图访问临界区（即传递给 dispatch_once 的代码）的不同的线程会在临界区已有一个线程的情况下被阻塞，直到临界区完成为止。



需要记住的是，这只是让访问共享实例线程安全。它绝对没有让类本身线程安全。类中可能还有其它竞态条件，例如任何操纵内部数据的情况。这些需要用其它方式来保证线程安全，例如同步访问数据，你将在下面几个小节看到。

处理读者与写者问题

线程安全实例不是处理单例时的唯一问题。如果单例属性表示一个可变对象，那么你就需要考虑是否那个对象自身线程安全。

如果问题中的这个对象是一个 Foundation 容器类，那么答案是——“很可能不安全”！Apple 维护一个有用且有些心寒的列表，众多的 Foundation 类都不是线程安全的。NSMutableArray，已用于你的单例，正在那个列表里休息。

虽然许多线程可以同时读取 NSMutableArray 的一个实例而不会产生问题，但当一个线程正在读取时让另外一个线程修改数组就是不安全的。你的单例在目前的情况下不能预防这种情况的发生。

要分析这个问题，看看 PhotoManager.m 中的 addPhoto:，转载如下：

```
- (void)addPhoto:(Photo *)photo
{
    if (photo) {
        [_photosArray addObject:photo];
        dispatch_async(dispatch_get_main_queue(), ^{
            [self postContentAddedNotification];
        });
    }
}
```

这是一个写方法，它修改一个私有可变数组对象。

现在看看 photos，转载如下：


```
- (NSArray *)photos
{
    return [NSArray arrayWithArray:_photosArray];
}
```

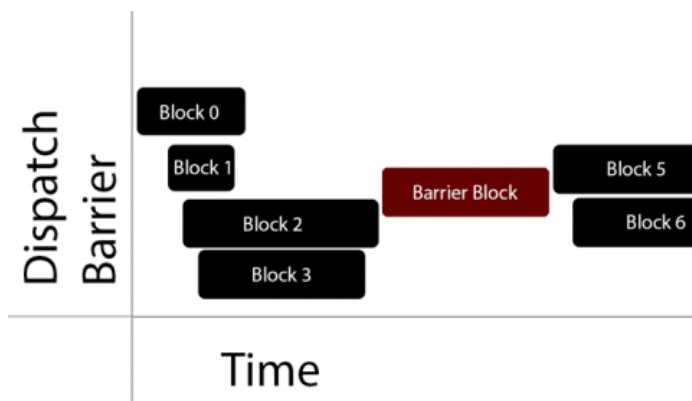
这是所谓的 读 方法，它读取可变数组。它为调用者生成一个不可变的拷贝，防止调用者不当地改变数组，但这不能提供任何保护来对抗当一个线程调用读方法 `photos` 的同时另一个线程调用写方法 `addPhoto:`。

这就是软件开发中经典的 读者写者问题。GCD 通过用 `dispatch barriers` 创建一个 读者写者锁 提供了一个优雅的方案。

Dispatch barriers 是一组函数，在并发队列上工作时扮演一个串行式的瓶颈。使用 GCD 的障碍（barrier）API 确保提交的 Block 在那个特定时间上是指定队列上唯一被执行的条目。这就意味着所有的先于调度障碍提交到队列的条目必能在这个 Block 执行前完成。

当这个 Block 的时机到达，调度障碍执行这个 Block 并确保在那个时间里队列不会执行任何其它 Block。一旦完成，队列就返回到它默认的实现状态。GCD 提供了同步和异步两种障碍函数。

下图显示了障碍函数对多个异步队列的影响：



注意到正常部分的操作就如同一个正常的并发队列。但当障碍执行时，它本质上就如同一个串行队列。也就是，障碍是唯一在执行的事物。在障碍完成后，队列回到一个正常并发队列的样子。

下面是你何时会——和不会——使用障碍函数的情况：

- 自定义串行队列：一个很坏的选择；障碍不会有任何帮助，因为不管怎样，一个串行队列一次都只执行一个操作。
- 全局并发队列：要小心；这可能不是最好的主意，因为其它系统可能在使用队列而且你不能垄断它们只为你自己的目的。
- 自定义并发队列：这对于原子或临界区代码来说是极佳的选择。任何你在设置或实例化的需要线程安全的事物都是使用障碍的最佳候选。

由于上面唯一像样的选择是自定义并发队列，你将创建一个你自己的队列去处理你的障碍函数并分开读和写函数。且这个并发队列将允许多个操作同时进行。

打开 `PhotoManager.m`，添加如下私有属性到类扩展中：

```
@interface PhotoManager ()
@property (nonatomic, strong, readonly) NSMutableArray *photosArray;
@property (nonatomic, strong) dispatch_queue_t concurrentPhotoQueue; ///< Add this
@end
```

找到 `addPhoto:` 并用下面的实现替换它：

```
- (void)addPhoto:(Photo *)photo
{
    if (photo) { // 1
        dispatch_barrier_async(self.concurrentPhotoQueue, ^{ // 2
            [_photosArray addObject:photo]; // 3
            dispatch_async(dispatch_get_main_queue(), ^{ // 4
```

```

        [self postContentAddedNotification];
    });
    });
}
}
}

```

你新写的函数是这样工作的：

1. 在执行下面所有的工作前检查是否有合法的相片。
2. 添加写操作到你的自定义队列。当临界区在稍后执行时，这将是你的队列中唯一执行的条目。
3. 这是添加对象到数组的实际代码。由于它是一个障碍 Block，这个 Block 永远不会同时和其它 Block 一起在 `concurrentPhotoQueue` 中执行。
4. 最后你发送一个通知说明完成了添加图片。这个通知将在主线程被发送因为它将会做一些 UI 工作，所以在此为了通知，你异步地调度另一个任务到主线程。

这就处理了写操作，但你还需要实现 `photos` 读方法并实例化 `concurrentPhotoQueue`。

在写者打扰的情况下，要确保线程安全，你需要在 `concurrentPhotoQueue` 队列上执行读操作。既然你需要从函数返回，你就不能异步调度到队列，因为那样在读者函数返回之前不一定运行。

在这种情况下，`dispatch_sync` 就是一个绝好的候选。

`dispatch_sync()` 同步地提交工作并在返回前等待它完成。使用 `dispatch_sync` 跟踪你的调度障碍工作，或者当你需要等待操作完成后才能使用 Block 处理过的数据。如果你使用第二种情况做事，你将不时看到一个 `__block` 变量写在 `dispatch_sync` 范围之外，以便返回时在 `dispatch_sync` 使用处理过的对象。

但你需要很小心。想像如果你调用 `dispatch_sync` 并放在你已运行着的当前队列。这会导致死锁，因为调用会一直等待直到 Block 完成，但 Block 不能完成（它甚至不会开始！），直到当前已经存在的任务完成，而当前任务无法完成！这将迫使你自觉于你正从哪个队列调用——以及你正在传递进入哪个队列。

下面是一个快速总览，关于在何时以及何处使用 `dispatch_sync`：

- 自定义串行队列：在这个状况下要非常小心！如果你正运行在一个队列并调用 `dispatch_sync` 放在同一个队列，那你就百分百地创建了一个死锁。
- 主队列（串行）：同上面的理由一样，必须非常小心！这个状况同样有潜在的导致死锁的情况。
- 并发队列：这才是做同步工作的好选择，不论是通过调度障碍，或者需要等待一个任务完成才能执行进一步处理的情况。

继续在 `PhotoManager.m` 上工作，用下面的实现替换 `photos`：

```

- (NSArray *)photos
{
    __block NSArray *array; // 1
    dispatch_sync(self.concurrentPhotoQueue, ^{ // 2
        array = [NSArray arrayWithArray:_photosArray]; // 3
    });
    return array;
}

```

这就是你的读函数。按顺序看看编过号的注释，有这些：

1. `__block` 关键字允许对象在 Block 内可变。没有它，`array` 在 Block 内部就只是只读的，你的代码甚至不能通过编译。
2. 在 `concurrentPhotoQueue` 上同步调度来执行读操作。
3. 将相片数组存储在 `array` 内并返回它。

最后，你需要实例化你的 `concurrentPhotoQueue` 属性。修改 `sharedManager` 以便像下面这样初始化队列：

```

+ (instancetype)sharedManager
{
    static PhotoManager *sharedPhotoManager = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{

```

```

sharedPhotoManager = [[PhotoManager alloc] init];
sharedPhotoManager->_photosArray = [NSMutableArray array];

// ADD THIS:
sharedPhotoManager->_concurrentPhotoQueue = dispatch_queue_create("com.selander.GooglyPuff.photoQueue",
                                                                    DISPATCH_QUEUE_CONCURRENT);

});

return sharedPhotoManager;
}

```

这里使用 `dispatch_queue_create` 初始化 `concurrentPhotoQueue` 为一个并发队列。第一个参数是反向DNS样式命名惯例；确保它是描述性的，将有助于调试。第二个参数指定你的队列是串行还是并发。

注意：当你在网上搜索例子时，你会经常看人们传递 `0` 或者 `NULL` 给 `dispatch_queue_create` 的第二个参数。这是一个创建串行队列的过时方式；明确你的参数总是更好。

恭喜——你的 `PhotoManager` 单例现在是线程安全的了。不论你在何处或怎样读或写你的照片，你都有这样的自信，即它将以安全的方式完成，不会出现任何惊吓。

A Visual Review of Queueing 队列的虚拟回顾

依然没有 100% 地掌握 GCD 的要领？确保你可以使用 GCD 函数轻松地创建简单的例子，使用断点和 `NSLog` 语句保证自己明白当下发生的情况。

我在下面提供了两个 GIF 动画来帮助你巩固对 `dispatch_async` 和 `dispatch_sync` 的理解。包含在每个 GIF 中的代码可以提供视觉辅助；仔细注意 GIF 左边显示代码断点的每一步，以及右边相关队列的状态。

dispatch_sync 回顾

```

- (void)viewDidLoad
{
    [super viewDidLoad];

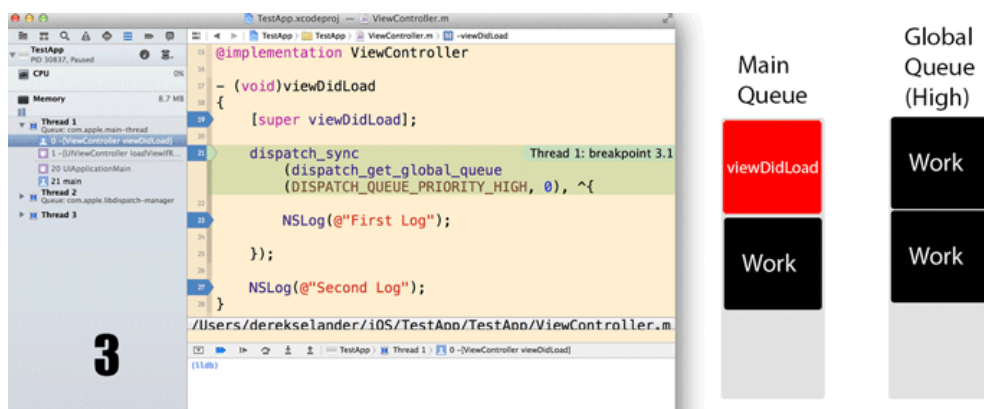
    dispatch_sync(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{

        NSLog(@"First Log");

    });

    NSLog(@"Second Log");
}

```



下面是图中几个步骤的说明：

1. 主队列一路按顺序执行任务——接着是一个实例化 `UIViewController` 的任务，其中包含了 `viewDidLoad`。
2. `viewDidLoad` 在主线程执行。
3. 主线程目前在 `viewDidLoad` 内，正要到达 `dispatch_sync`。

4. `dispatch_sync` Block 被添加到一个全局队列中，将在稍后执行。进程将在主线程挂起直到该 Block 完成。同时，全局队列并发处理任务；要记得 Block 在全局队列中将按照 FIFO 顺序出列，但可以并发执行。
5. 全局队列处理 `dispatch_sync` Block 加入之前已经出现在队列中的任务。
6. 终于，轮到 `dispatch_sync` Block。
7. 这个 Block 完成，因此主线程上的任务可以恢复。
8. `viewDidLoad` 方法完成，主队列继续处理其他任务。

`dispatch_sync` 添加任务到一个队列并等待直到任务完成。`dispatch_async` 做类似的事情，但不同之处是它不会等待任务的完成，而是立即继续“调用线程”的其它任务。

dispatch_async 回顾

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{

        NSLog(@"First Log");

    });

    NSLog(@"Second Log");
}
```



1. 主队列一路按顺序执行任务——接着是一个实例化 `UIViewController` 的任务，其中包含了 `viewDidLoad`。
2. `viewDidLoad` 在主线程执行。
3. 主线程目前在 `viewDidLoad` 内，正要到达 `dispatch_async`。
4. `dispatch_async` Block 被添加到一个全局队列中，将在稍后执行。
5. `viewDidLoad` 在添加 `dispatch_async` 到全局队列后继续进行，主线程把注意力转向剩下的任务。同时，全局队列并发地处理它未完成地任务。记住 Block 在全局队列中将按照 FIFO 顺序出列，但可以并发执行。
6. 添加到 `dispatch_async` 的代码块开始执行。
7. `dispatch_async` Block 完成，两个 `NSLog` 语句将它们的输出放在控制台上。

在这个特定的实例中，第二个 `NSLog` 语句执行，跟着是第一个 `NSLog` 语句。并不总是这样——着取决于给定时刻硬件正在做的事情，而且你无法控制或知晓哪个语句会先执行。“第一个” `NSLog` 在某些调用情况下会第一个执行。

下一步怎么走？

在本教程中，你学习了如何让你的代码线程安全，以及在执行 CPU 密集型任务时如何保持主线程的响应性。

你可以下载 [GooglyPuff 项目](#)，它包含了目前所有本教程中编写的实现。在本教程的[第二部分](#)，你将继续改进这个项目。

如果你计划优化你自己的应用，那你应该用 Instruments 中的 Time Profile 模版分析你的工作。对这个工具的使用超出了本教程的范围，你可以看看 [如何使用 Instruments](#) 来得到一个很好的概述。

同时请确保在真实设备上分析，而在模拟器上测试会对程序速度产生非常不准确的印象。

在教程的下一部分，你将更加深入到 GCD 的 API 中，做一些更 Cool 的东西。

如果你有任何问题或评论，可自由地加入下方的讨论！

译者注：欢迎转载，但请一定注明出处！ <https://github.com/nixzhu/dev-blog>

欢迎转发此条微博 <http://weibo.com/2076580237/B0tjrEQr9> 以分享给更多人！

如果你认为这篇翻译不错，也有闲钱，那你可以用支付宝随便捐助一点，以慰劳译者的辛苦：

