Objective-C 高级编程读书笔记之 GCD – IOS – 伯乐在线

Objective-C 高级编程
iOS 与 OS X 多线程和内存管理

Grand Central Dispatch (GCD)

目录

- 什么是 GCD
- 什么是多线程, 并发
- GCD 的优势
- GCD 的 API 介绍
- GCD 的注意点
- GCD 的使用场景
- Dispatch Source

> - 总结

1. 什么是 GCD

GCD, Grand Central Dispatch, 可译为”强大的中枢调度器”, 基于 libdispatch, 纯 C 语言, 里面包含了许多多线程相关非常强大的函数. 程序员可以既不写一句线程管理的代码又能很好地使用多线程执行任务.

GCD 中有 **Dispatch Queue** 和 **Dispatch Source**. Dispatch Queue 是主要的, 而 Dispatch Source 比较次要. 所以这里主要介绍 Dispatch Queue, 而 Dispatch Source 在下面会简单介绍.

Dispatch Queue

苹果官方对 GCD 的说明如下 :

> 开发者要做的只是定义想执行的任务并追加到适当的 Dispatch Queue 中.

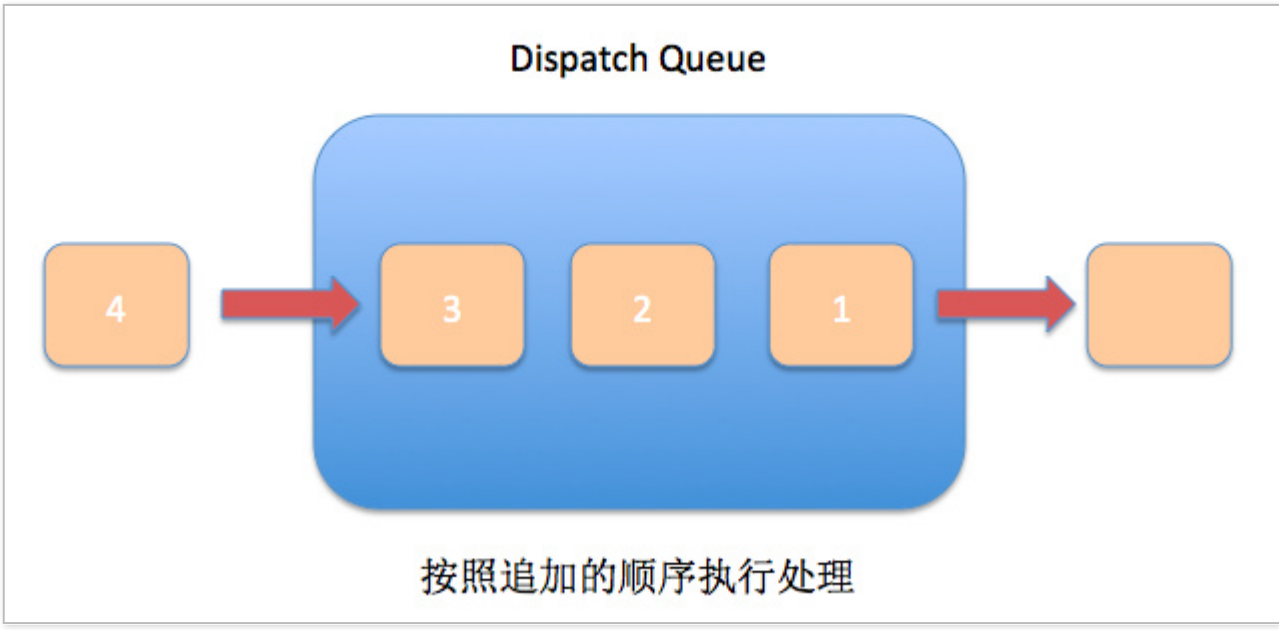
这句话用源代码表示如下

```Objective-C
dispatch_async(queue, ^{
    /*
     * 想执行的任务
     */
});
```

该源码用 block 的语法定义想执行的任务然后通过 dispatch_async 函数讲任务追加到赋值在变量 queue 的”Dispatch Queue”中.

Dispatch Queue 究竟是什么???

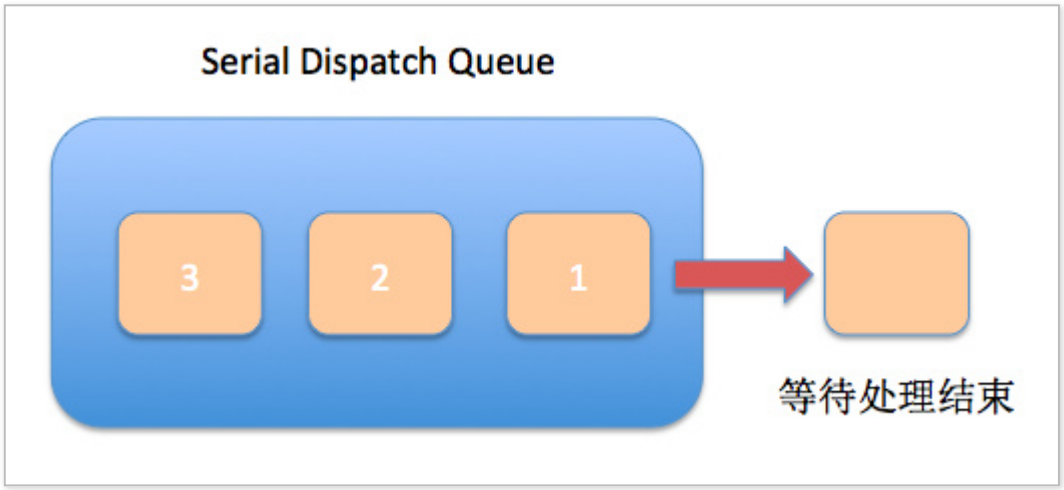
Dispatch Queue 是执行处理的等待队列, 按照先进先出 (FIFO, First-In-First-Out) 的顺序进行任务处理.



First-In-First-Out

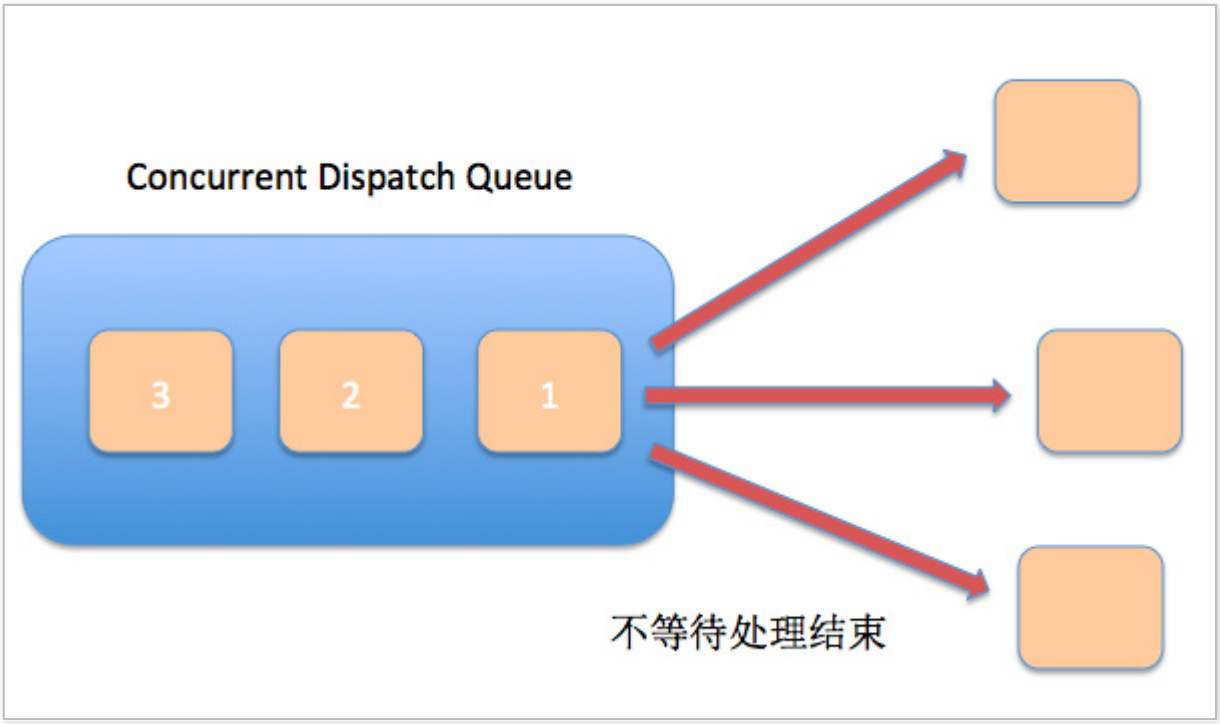
另外，队列分两种，一种是串行队列 (Serial Dispatch Queue)，一种是并行队列 (Concurrent Dispatch Queue).

Dispatch Queue 的种类	说明
Serial Dispatch Queue	等待现在执行中处理结束
Concurrent Dispatch Queue	不等待现在执行中处理结束



串行队列

串行队列：让任务一个接一个执行



并行队列

并发队列：让多个任务同时执行（自动开启多个线程执行任务）
并发功能只有在异步函数（dispatch_async）下才有效(想想看为什么?)

GCD 的 API 会在下面详细说明~

2. 什么是多线程, 并发

我们知道, 一个应用就相当于一个进程, 而一个进程可以同时分发几个线程同时处理任务. 而并发正是一个进程开启多个线程同时执行任务的意思, 主线程专门用来刷新 UI, 处理触摸事件等 而子线程呢, 则用来执行耗时的操作, 例如访问数据库, 下载数据等..

以前我们 CPU 还是单核的时候, 并不存在真正的线程并行, 因为我们只有一个核, 一次只能处理一个任务. 所以当时我们计算机是通过分时 也就是CPU地在各个进程之间快速切换，给人一种能同时处理多任务的错觉 来实现的, 而现在多核 CPU 计算机则能真真正正货真价实地办到同时处理多个任务.

3. GCD 的优势

说到优势, 当然有比较, 才能显得出优势所在. 事实上, iOS 中我们能使用的多线程管理技术有

- pthread
- NSThread
- GCD
- NSOperationQueue

pthread

来自 Clang, 纯 C 语言, 需要手动创建线程, 销毁线程, 手动进行线程管理. 而且代码极其恶心, 我保证你写一次不想写第二次... 不好意思我先去吐会 T~T

NSThread :

Foundation 框架下的 OC 对象, 依旧需要自己进行线程管理，线程同步。线程同步对数据的加锁会有一定的开销。

GCD :

两个字, 牛逼, 虽然是纯 C 语言, 但是它用难以置信的非常简洁的方式实现了极其复杂的多线程编程, 而且还支持 block 内联形式进行制定任务. 简洁! 高效! 而且我们再也不用手动进行线程管理了.

NSOperationQueue :

相当于 Foundation 框架的 GCD, 以面向对象的语法对 GCD 进行了封装. 效率一样高.

GCD 优势在哪里?

- GCD 会自动利用更多的 CPU 内核
- GCD 会自动管理线程的生命周期
- 使用方法及其简单

怎么样? 心动不, 迫不及待想要知道怎么使用 GCD 了吧, 那我们马上切入正题~

4. GCD 的 API 介绍

在介绍 GCD 的 API 之前, 我们先搞清楚四个名词: 串行, 并行, 同步, 异步

- 串行 : 一个任务执行完, 再执行下一个任务

- 并行：多个任务同时执行
- 同步：在**当前线程**中执行任务，**不具备**开启线程的能力
- 异步：在**新的线程**中执行任务，**具备**开启线程的能力

	并发队列	手动创建的串行队列	主队列
同步（sync）	◻ 没有开启新线程 ◻ 串行执行任务	◻ 没有开启新线程 ◻ 串行执行任务	◻ 没有开启新 ◻ 串行执行任
异步（async）	◻ 有开启新线程 ◻ 并发执行任务	◻ 有开启新线程 ◻ 串行执行任务	◻ 没有开启新 ◻ 串行执行任

串行，并行，同步，异步的关系

下面开始介绍 GCD 的 API

创建队列

```Objective-C
dispatch_queue_create(const char *label, dispatch_queue_attr_t attr)
```

手动创建一个队列.

- label：队列的标识符，日后可用来调试程序
- attr：队列类型
  DISPATCH_QUEUE_CONCURRENT：并发队列
  DISPATCH_QUEUE_SERIAL 或 NULL：串行队列

需要注意的是，通过 dispatch_queue_create 函数生成的 queue 在使用结束后需要通过 dispatch_release 函数来释放.(只有在 MRC 下才需要释放)

并不是什么时候都需要手动创建队列，事实上系统给我们提供 2 个很常用的队列.

主队列

```Objective-C
```

```
dispatch_get_main_queue();
```

该方法返回的是主线程中执行的同步队列. 用户界面的更新等一些必须在主线程中执行的操作追加到此队列中.

全局并发队列

```Objective-C
dispatch_get_global_queue(long identifier, unsigned long flags);
```

该方法返回的是全局并发队列. 使用十分广泛.

- identifier：优先级
  DISPATCH_QUEUE_PRIORITY_HIGH：高优先级
  DISPATCH_QUEUE_PRIORITY_DEFAULT：默认优先级
  DISPATCH_QUEUE_PRIORITY_LOW：低优先级
  DISPATCH_QUEUE_PRIORITY_BACKGROUND：后台优先级
- flags：暂时用不上, 传 **0** 即可

注意：对 Main Dispatch Queue 和 Global Dispatch Queue 执行 dispatch_release 和 dispatch_retain 没有任何问题. (MRC)

同步函数

```Objective-C
dispatch_sync(dispatch_queue_t queue, ^(void)block);
```

在参数 queue 队列下同步执行 block

异步函数

```Objective-C
dispatch_async(dispatch_queue_t queue, ^(void)block);
```

在参数 queue 队列下异步执行 block(开启新线程)

时间

```Objective-C
dispatch_time(dispatch_time_t when, int64_t delta);
```

根据传入的时间 (when) 和延迟 (delta) 计算出一个未来的时间

- when：
  DISPATCH_TIME_NOW：现在
  DISPATCH_TIME_FOREVER：永远 (别传这个参数, 否则该时间很大)
- delta：该参数接收的是纳秒, 可以用一个宏 **NSEC_PER_SEC** 来进行转换, 例如你要延迟 3 秒, 则为 3 * NSEC_PER_SEC.

延迟执行

```Objective-C
dispatch_after(dispatch_time_t when, dispatch_queue_t queue, ^(void)bl
```

有了上述获取时间的函数, 则可以直接把时间传入, 然后定义该延迟执行的 block 在哪一个 queue 队列中执行.

苹果还给我们提供了一个在主队列中延迟执行的代码块, 如下

```Objective-C
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(delayInSecor
            code to be executed after a specified delay
        });
```

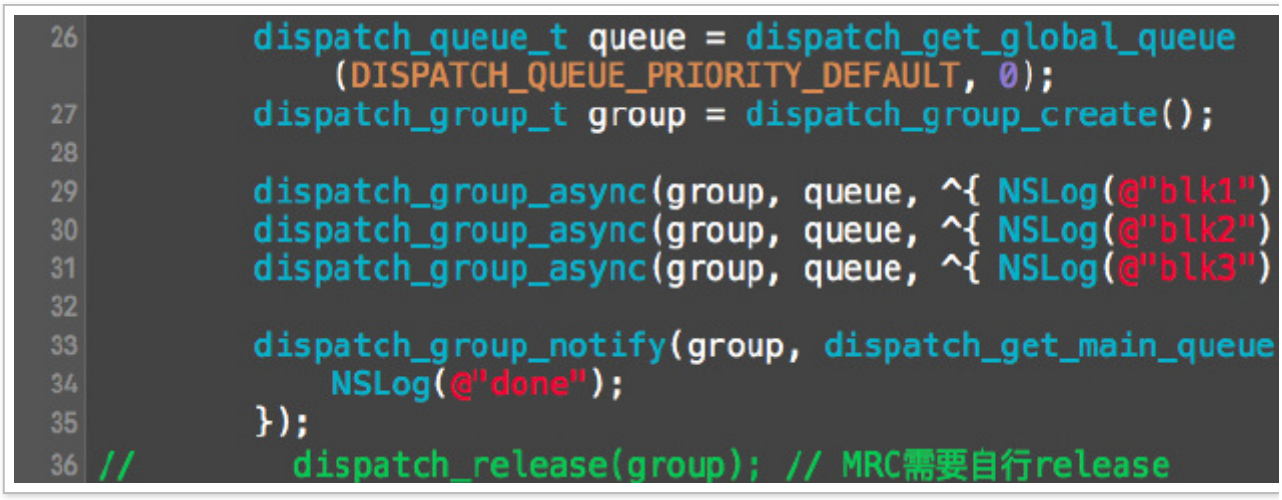
我们只需要传入需要延迟的秒数 (delayInSeconds) 和执行的任务 block 就可以直接调用了, 方便吧~

注意：延迟执行不是在指定时间后执行任务处理, 而是在指定时间后将处理追加到队列中, 这个是要分清楚的

队列组

```Objective-C
dispatch_group_create();
```

有时候我们想要在队列中的多个任务都处理完毕之后做一些事情, 就能用到这个 Group. 同队列一样, Group 在使用完毕也是需要 dispatch_release 掉的 (MRC). 上代码



group

组异步函数

```Objective-C
dispatch_group_async(dispatch_group_t group, dispatch_queue_t queue,
```

分发 Group 内的并发异步函数

组通知

```Objective-C
dispatch_group_notify(dispatch_group_t group, dispatch_queue_t queue,
```

监听 group 的任务进度, 当 group 内的任务全部完成, 则在 queue 队列中执行 block.

组等待

```Objective-C
dispatch_group_wait(dispatch_group_t group, dispatch_time_t timeout)
```

- timeout：等待的时间
  DISPATCH_TIME_NOW：现在
  DISPATCH_TIME_FOREVER：永远

该函数会一直等待组内的异步函数任务全部执行完毕才会返回. 所以该函数会卡住当前线程. 若参数 timeout 为 DISPATCH_TIME_FOREVER, 则只要

group 内的任务尚未执行结束, 就会一直等待, 中途不能取消.

栅栏

```Objective-C
dispatch_barrier_async(dispatch_queue_t queue, ^(void)block)
```

在访问数据库或文件时, 为了提高效率, 读取操作放在并行队列中执行. 但是写入操作必须在串行队列中执行 (避免资源抢夺问题). 为了避免麻烦, 此时 dispatch_barrier_async 函数作用就出来了, 在这函数里进行写入操作, 写入操作会等到所有读取操作完毕后, 形成一道栅栏, 然后进行写入操作, 写入完毕后再把栅栏移除, 同时开放读取操作. 如图

dispatch_barrier_async

快速迭代

```Objective-C
dispatch_apply(10, dispatch_get_global_queue(0, 0), ^(size_t index){
    // code here
});
```

执行 10 次代码, index 顺序不确定. dispatch_apply 会等待全部处理执行结束才会返回. 意味着 dispatch_apply 会阻塞当前线程. 所以 dispatch_apply 一般用于异步函数的 block 中.

一次性代码

```Objective-C
static dispatch_once_t onceToken;
dispatch_once(&amp;onceToken, ^{
// 只执行 1 次的代码（这里面默认是线程安全的）
});
```

该代码在整个程序的生命周期中只会执行一次.

挂起和恢复

```Objective-C
dispatch_suspend(queue)
```

挂起指定的 queue 队列, 对已经执行的没有影响, 追加到队列中尚未执行的停止执行.

```Objective-C
dispatch_resume(queue)
```

恢复指定的 queue 队列, 使尚未执行的处理继续执行.

5. GCD 的注意点

因为在 ARC 下, 不需要我们释放自己创建的队列, 所以 GCD 的注意点就剩下**死锁**

死锁

```Objective-C
NSLog(@"111");
dispatch_sync(dispatch_get_main_queue(), ^{
    NSLog(@"222");
});
NSLog(@"333");
```

以上三行代码将输出什么?

```
111
```

```
222
```

```
333
```
？

还是

```
111
```

```
333
```
？

其实都不对，输出结果是

```
111
```

为什么？看下图

死锁

毫无疑问会先输出 111，然后在当前队列下调用 dispatch_sync 函数，dispatch_sync 函数会把 block 追加到当前队列上，然后等待 block 调用完毕该函数才会返回，不巧的是，block 在队列的尾端，而队列正在执行的是 dispatch_sync 函数. 现在的情况是，block 不执行完毕，dispatch_sync 函数就不能返回，dispatch_sync 不返回，就没机会执行 block 函数. 这种你等我，我也等你的情况就是**死锁**，后果就是大家都执行不了，当前线程卡死在这里.

如何避免死锁？

不要在当前队列使用同步函数，在队列嵌套的情况下也不允许. 如下图，

```
        dispatch_queue_t queue1 = dispatch_queue_create("queue1",
        dispatch_queue_t queue2 = dispatch_queue_create("queue2",

        dispatch_sync(queue1, ^{
            NSLog(@"111");

            dispatch_sync(queue2, ^{
                NSLog(@"222");

                dispatch_sync(queue1, ^{
                    NSLog(@"333");
                });
            });
        });
```

队列嵌套调用同步函数引发死锁

大家可以想象, 队列 1 执行完 NSLog 后到队列 2 中执行 NSLog, 队列 2 执行完后又跳回队列 1 中执行 NSLog, 由于都是同步函数, 所以最内层的 NSLog(“333”); 追加到队列 1 中, 实际上最外层的 dispatch_sync 是还没返回的, 所以它没有执行的机会. 也形成死锁. 运行程序, 果不其然, 打印如下 :

```
111

222
```

6. GCD 的使用场景

线程间的通信

这是 GCD 最常用的使用场景了, 如下代码

```Objective-C
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAU
    // 执行耗时操作
    dispatch_async(dispatch_get_main_queue(), ^{
        // 回到主线程作刷新 UI 等操作
    });
});
```

为了不阻塞主线程, 我们总是在后台线程中发送网络请求, 处理数据, 然后再回到主线程中刷新 UI 界面

单例

单例也就是在程序的整个生命周期中, 该类有且仅有一个实例对象, 此时为了保证只有一个实例对象, 我们这里用到了 dispatch_once 函数

```Objective-C
static XXTool <em>_instance;
+ (instancetype)allocWithZone:(struct _NSZone </em>)zone
{
    static dispatch_once_t onceToken;
    dispatch_once(&amp;onceToken, ^{
        _instance = [self allocWithZone:zone];
    });
    return _instance;
}
+ (instancetype)sharedInstance
{
    static dispatch_once_t onceToken;
    dispatch_once(&amp;onceToken, ^{
        _instance = [[self alloc] init];
    });
    return _instance;
}
- (id)copy
{
    return _instance;
}
- (id)mutableCopy
{
    return _instance;
}
```

因为 alloc 内部会调用 allWithZone, 所以我们重写 allocWithZone 方法就行了. 通过以上代码可以保证程序只能创建一个实例对象, 并且该实例对象永远存在程序中.

同步队列和锁

我们知道, 属性中有 atomic 和 nonatomic 属性

- atomic : setter 方法线程安全, 需要消耗大量的资源
- nonatomic : setter 方法非线程安全, 适合内存小的移动设备

为了实现属性线程安全, 避免资源抢夺的问题, 我们也许会这样写

```Objective-C
- (NSString *)setMyString:(NSString *)myString
{
    @synchronized(self) {
        _myString = myString;
    }
}
```

这种方法没错是可以达到该属性线程安全的需求, 但是试想一下, 如果一个对象中有许多个属性都需要保证线程安全, 那么就会在 self 对象上频繁加锁, 那么两个毫无关系的 setter 方法就有可能执行一个 setter 方法需要等待另一个 setter 方法执行完毕解锁之后才能执行, 这样做毫无必要. 那么你有可能会说, 在每个方法内部创建一个锁对象就好啦, 不过你不觉得这样会浪费资源吗?

那么能不能利用队列, 实现 getter 方法可以并发执行, 而 setter 方法串行执行并且 setter 和 getter 不能并发执行呢??? 没错, 我们这里用到了 dispatch_barrier_async 函数.

```Objective-C
- (NSString <em>)myString
{
    __block NSString </em>localMyString = nil;
    dispatch_sync(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DE
        localMyString = self.myString;
    });
    return localMyString;
}
- (void)setMyString:(NSString *)myString
{
    dispatch_barrier_async(dispatch_get_global_queue(DISPATCH_QUEUE_PR
        _myString = myString;
    });
}
```

这里利用了栅栏块必须单独执行, 不能与其他块并行的特性, 写入操作就必须等当前的读取操作都执行完毕, 然后单独执行写入操作, 等待写入操作执行完毕后再继续处理读取.

7. Dispatch Source

它是 BSD 系内核惯有功能 kqueue 的包装. kqueue 的 CPU 负荷非常小, 可以说是应用程序处理 XNU 内核中发生的各种事件的方法中最优秀的一种.

但是由于 Dispatch Source 实在是太少人用了, 所以这里不再介绍. 感兴趣的朋友们可以自行 Google.

8. 总结

- GCD 可进行线程间通信
- GCD 可以办到线程安全
- GCD 可用于延迟执行

- GCD 需要注意死锁问题 (不要在当前队列调用同步函数)

想再往深了解并发编程, 可以看看这篇文章

[并发编程：API 及挑战]

附上另外两篇文章的链接

[Objective-C 高级编程读书笔记之内存管理]

[Objective-C 高级编程读书笔记之 blocks]

全文完

本文由 简悦 SimpRead 优化，用以提升阅读体验。