

细说GCD（Grand Central Dispatch）如何用

戴铭 edited this page on 23 Feb 2017 · 5 revisions

Edit

New Page

文中较详细介绍GCD队列，各种GCD使用方法，实例如何使用Dispatch Source监听系统底层对象，分析不同锁的性能对比，实例GCD死锁情况。文中的Demo在这里<https://github.com/ming1016/GCDDemo> 对着文章试着来调demo体会更深哦，细细嚼消化好:)

GCD（Grand Central Dispatch）介绍

GCD属于系统级的线程管理，在Dispatch queue中执行需要执行的任务性能非常的高。GCD这块已经开源，地址<http://libdispatch.macosforge.org>。GCD中的FIFO队列称为dispatch queue，用来保证先进来的任务先得到执行。

GCD概要

- 和operation queue一样都是基于队列的并发编程API，他们通过集中管理大家协同使用的线程池。
- 公开的5个不同队列：运行在主线程中的main queue，3个不同优先级的后台队列（High Priority Queue，Default Priority Queue，Low Priority Queue），以及一个优先级更低的后台队列Background Priority Queue（用于I/O）
- 可创建自定义队列：串行或并列队列。自定义一般放在Default Priority Queue和Main Queue里。
- 操作是在多线程上还是单线程主要是看队列的类型和执行方法，并行队列异步执行才能多线程，并行队列同步执行就只会在这个并行队列在队列中被分配的那个线程执行。（[TorchLennon](#) 指出先前文中此句一处[错误](#)）

基本概念

- 系统标准两个队列

```
//全局队列，一个并行的队列
dispatch_get_global_queue
//主队列，主线程中的唯一队列，一个串行队列
dispatch_get_main_queue
```

- 自定义队列

```
//串行队列
dispatch_queue_create("com.starming.serialqueue", DISPATCH_QUEUE_SERIAL)
//并行队列
dispatch_queue_create("com.starming.concurrentqueue", DISPATCH_QUEUE_CONCURRENT)
```

- 同步异步线程创建

```
//同步线程
dispatch_sync(..., ^(block))
//异步线程
dispatch_async(..., ^(block))
```

队列（dispatch queue）

- Serial：又叫private dispatch queues，同时只执行一个任务。Serial queue常用于同步访问特定的资源或数据。当你创建多个Serial queue时，虽然各自是同步，但serial queue之间是并发执行。

▼ Pages 64

Find a Page...

Home

Auto Layout

Block

Camera

CFRunLoop

Cocoapods

Collection View动画

Core Animation

Core Data

Core Image

GPU处理图像

HTML 转原生 HTN 项目开发记录

iOS Background Tasks


iOS书籍推荐


iOS函数响应式编程以及ReactiveCocoa的使用

Show 49 more pages...

Clone this wiki locally

https://github.com/ming1016/study/wiki



 Clone in Desktop

- Main dispatch queue: 全局可用的serial queue, 在应用程序主线程上执行任务。
- Concurrent: 又叫global dispatch queue, 可以并发的执行多个任务, 但执行完成顺序是随机的。系统提供四个全局并发队列, 这四个队列有这对应的优先级, 用户是不能够创建全局队列的, 只能获取。

```
dispatch_queue_t queue;
queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0);
```

- user create queue: 创建自己定义的队列, 可以用dispatch_queue_create函数, 函数有两个参数, 第一个自定义的队列名, 第二个参数是队列类型, 默认NULL或者DISPATCH_QUEUE_SERIAL的是串行, 参数为DISPATCH_QUEUE_CONCURRENT为并行队列。

```
dispatch_queue_t queue
queue = dispatch_queue_create("com.starming.gcddemo.concurrentqueue", DISPATCH_QUEUE
```

- 自定义队列的优先级: 可以通过dispatch_queue_attr_make_with_qos_class或dispatch_set_target_queue方法设置队列的优先级

```
//dispatch_queue_attr_make_with_qos_class
dispatch_queue_attr_t attr = dispatch_queue_attr_make_with_qos_class(DISPATCH_QUEUE_
dispatch_queue_t queue = dispatch_queue_create("com.starming.gcddemo.qosqueue", attr

//dispatch_set_target_queue
dispatch_queue_t queue = dispatch_queue_create("com.starming.gcddemo.settargetqueue"
dispatch_queue_t referQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_LOW,
dispatch_set_target_queue(queue, referQueue); //设置queue和referQueue的优先级一样
```

- dispatch_set_target_queue: 可以设置优先级, 也可以设置队列层级体系, 比如让多个串行和并行队列在统一一个串行队列里串行执行, 如下

```
dispatch_queue_t serialQueue = dispatch_queue_create("com.starming.gcddemo.serialque
dispatch_queue_t firstQueue = dispatch_queue_create("com.starming.gcddemo.firstqueue
dispatch_queue_t secondQueue = dispatch_queue_create("com.starming.gcddemo.secondque

dispatch_set_target_queue(firstQueue, serialQueue);
dispatch_set_target_queue(secondQueue, serialQueue);

dispatch_async(firstQueue, ^{
    NSLog(@"1");
    [NSThread sleepForTimeInterval:3.f];
});
dispatch_async(secondQueue, ^{
    NSLog(@"2");
    [NSThread sleepForTimeInterval:2.f];
});
dispatch_async(secondQueue, ^{
    NSLog(@"3");
    [NSThread sleepForTimeInterval:1.f];
});
```

队列类型

队列默认是串行的, 如果设置改参数为NULL会按串行处理, 只能执行一个单独的block, 队列也可以是并行的, 同一时间执行多个block

```
- (id)init;
{
    self = [super init];
    if (self != nil) {
        NSString *label = [NSString stringWithFormat:@"%s.isolation.%p", [self cla
        self.isolationQueue = dispatch_queue_create([label UTF8String], 0);

        label = [NSString stringWithFormat:@"%s.work.%p", [self class], self];
        self.workQueue = dispatch_queue_create([label UTF8String], 0);
    }
}
```

```

        return self;
    }

```

5种队列，主队列（main queue），四种通用调度队列，自己定制的队列。四种通用调度队列为

- QOS_CLASS_USER_INTERACTIVE：user interactive等级表示任务需要被立即执行提供好的体验，用来更新UI，响应事件等。这个等级最好保持小规模。
- QOS_CLASS_USER_INITIATED：user initiated等级表示任务由UI发起异步执行。适用场景是需要及时结果同时又可以继续交互的时候。
- QOS_CLASS_UTILITY：utility等级表示需要长时间运行的任务，伴有用户可见进度指示器。经常会用来做计算，I/O，网络，持续的数据填充等任务。这个任务节能。
- QOS_CLASS_BACKGROUND：background等级表示用户不会察觉的任务，使用它来处理预加载，或者不需要用户交互和对时间不敏感的任务。

示例：后台加载显示图片

```

override func viewDidLoad() {
    super.viewDidLoad()

    dispatch_async(dispatch_get_global_queue(Int(QOS_CLASS_USER_INITIATED.value), 0) {
        let overlayImage = self.faceOverlayImageFromImage(self.image)
        dispatch_async(dispatch_get_main_queue()) { // 新图完成，把一个闭包加入主线程用
            self.fadeInNewImage(overlayImage) // 更新UI
        }
    })
}

```

何时使用何种队列类型

- 主队列（顺序）：队列中有任务完成需要更新UI时，dispatch_after在这种类型中使用。
- 并发队列：用来执行与UI无关的后台任务，dispatch_sync放在这里，方便等待任务完成进行后续处理或和dispatch barrier同步。dispatch groups放在这里也不错。
- 自定义顺序队列：顺序执行后台任务并追踪它时。这样做同时只有一个任务在执行可以防止资源竞争。dispatch barriers解决读写锁问题的放在这里处理。dispatch groups也是放在这里。

可以使用下面的方法简化QoS等级参数的写法

```

var GlobalMainQueue: dispatch_queue_t {
    return dispatch_get_main_queue()
}
var GlobalUserInteractiveQueue: dispatch_queue_t {
    return dispatch_get_global_queue(Int(QOS_CLASS_USER_INTERACTIVE.value), 0)
}
var GlobalUserInitiatedQueue: dispatch_queue_t {
    return dispatch_get_global_queue(Int(QOS_CLASS_USER_INITIATED.value), 0)
}
var GlobalUtilityQueue: dispatch_queue_t {
    return dispatch_get_global_queue(Int(QOS_CLASS_UTILITY.value), 0)
}
var GlobalBackgroundQueue: dispatch_queue_t {
    return dispatch_get_global_queue(Int(QOS_CLASS_BACKGROUND.value), 0)
}

//使用起来就是这样，易读而且容易看出在使用哪个队列
dispatch_async(GlobalUserInitiatedQueue) {
    let overlayImage = self.faceOverlayImageFromImage(self.image)
    dispatch_async(GlobalMainQueue) {
        self.fadeInNewImage(overlayImage)
    }
}

```

dispatch_once用法

dispatch_once_t要是全局或static变量，保证dispatch_once_t只有一份实例

```

+ (UIColor *)boringColor;
{

```

```

static UIColor *color;
//只运行一次
static dispatch_once_t onceToken;
dispatch_once(&onceToken, ^{
    color = [UIColor colorWithRed:0.380f green:0.376f blue:0.376f alpha:1.000f];
});
return color;
}

```

dispatch_async

设计一个异步的API调用dispatch_async(), 这个调用放在API的方法或函数中做。让API的使用者设置一个回调处理队列

```

- (void)processImage:(UIImage *)image completionHandler:(void (^)(BOOL success))handler {
    dispatch_async(self.isolationQueue, ^(void){
        // do actual processing here
        dispatch_async(self.resultQueue, ^(void){
            handler(YES);
        });
    });
}

```

可以避免界面会被一些耗时的操作卡死, 比如读取网络数据, 大数据IO, 还有大量数据的数据库读写, 这时需要在另一个线程中处理, 然后通知主线程更新界面, GCD使用起来比NSThread和NSOperation方法要简单方便。

```

//代码框架
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    // 耗时的操作
    dispatch_async(dispatch_get_main_queue(), ^{
        // 更新界面
    });
});

//下载图片的示例
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    NSURL * url = [NSURL URLWithString:@"http://avatar.csdn.net/2/C/D/1_totogo2010"];
    NSData * data = [[NSData alloc] initWithContentsOfURL:url];
    UIImage *image = [[UIImage alloc] initWithData:data];
    if (data != nil) {
        dispatch_async(dispatch_get_main_queue(), ^{
            self.imageView.image = image;
        });
    }
});

```

dispatch_after延后执行

dispatch_after只是延时提交block, 不是延时立刻执行。

```

- (void)foo {
    double delayInSeconds = 2.0;
    dispatch_time_t popTime = dispatch_time(DISPATCH_TIME_NOW, (int64_t) (delayInSeconds * NSEC_PER_SEC));
    dispatch_after(popTime, dispatch_get_main_queue(), ^(void){
        [self bar];
    });
}

```

范例, 实现一个推迟出现弹出框提示, 比如说提示用户评价等功能。

```

func showOrHideNavPrompt() {
    let delayInSeconds = 1.0
    let popTime = dispatch_time(DISPATCH_TIME_NOW,
        Int64(delayInSeconds * Double(NSEC_PER_SEC))) // 在这里声明推迟的时间
    dispatch_after(popTime, GlobalMainQueue) { // 等待delayInSeconds将闭包异步到主队列

```

例子中的dispatch time的参数，可以先看看函数原型

第一个参数为DISPATCH_TIME_NOW表示当前。第二个参数的delta表示纳秒，一秒对应的纳秒为1000000000，系统提供了一些宏来简化

这样如果要表示一秒就可以这样写

dispatch_barrier_async使用Barrier Task方法Dispatch Barrier解决多线程并发读写同一个资源发生死锁

```
//创建队列
self.isolationQueue = dispatch_queue_create([label UTF8String], DISPATCH_QUEUE_CONCURRENT);
//改变setter
- (void)setCount:(NSUInteger)count forKey:(NSString *)key
{
    key = [key copy];
    //确保所有barrier都是async异步的
    dispatch_barrier_async(self.isolationQueue, ^(){
        if (count == 0) {
            [self.counts removeObjectForKey:key];
        } else {
            self.counts[key] = @(count);
        }
    });
}

- (void)dispatchBarrierAsyncDemo {
    //防止文件读写冲突，可以创建一个串行队列，操作都在这个队列中进行，没有更新数据读写并行，写用串行
    dispatch_queue_t dataQueue = dispatch_queue_create("com.starling.gcdemo.dataqueue", DISPATCH_QUEUE_SERIAL);
    dispatch_async(dataQueue, ^{
        [NSThread sleepForTimeInterval:2.f];
        NSLog(@"read data 1");
    });
    dispatch_async(dataQueue, ^{
        NSLog(@"read data 2");
    });
    //等待前面的都完成，在执行barrier后面的
    dispatch_barrier_async(dataQueue, ^{
        NSLog(@"write data 1");
        [NSThread sleepForTimeInterval:1];
    });
    dispatch_async(dataQueue, ^{
        [NSThread sleepForTimeInterval:1.f];
        NSLog(@"read data 3");
    });
}
```

```
});
dispatch_async(dataQueue, ^{
    NSLog(@"read data 4");
});
}
```

swift示例

```
//使用dispatch_queue_create初始化一个并发队列。第一个参数遵循反向DNS命名习惯，方便描述，第二个参
private let concurrentPhotoQueue = dispatch_queue_create(
    "com.raywenderlich.GooglyPuff.photoQueue", DISPATCH_QUEUE_CONCURRENT)

func addPhoto(photo: Photo) {
    dispatch_barrier_async(concurrentPhotoQueue) { // 将写操作加入到自定义的队列。开始执
        self._photos.append(photo) // barrier能够保障不会和其他任务同时进行。
        dispatch_async(GlobalMainQueue) { // 涉及到UI所以这个通知应该在主线程中，所以分派
            self.postContentAddedNotification()
        }
    }
}

//上面是解决了写可能发生死锁，下面是使用dispatch_sync解决读时可能会发生的死锁。
var photos: [Photo] {
    var photosCopy: [Photo]!
    dispatch_sync(concurrentPhotoQueue) { // 同步调度到concurrentPhotoQueue队列执行读操
        photosCopy = self._photos // 保存
    }
    return photosCopy
}
//这样读写问题都解决了。
```

都用异步处理避免死锁，异步的缺点在于调试不方便，但是比起同步容易产生死锁这个副作用还算是小的。

dispatch_apply进行快速迭代

类似for循环，但是在并发队列的情况下dispatch_apply会并发执行block任务。

```
for (size_t y = 0; y < height; ++y) {
    for (size_t x = 0; x < width; ++x) {
        // Do something with x and y here
    }
}
//因为可以并行执行，所以使用dispatch_apply可以运行的更快
- (void)dispatchApplyDemo {
    dispatch_queue_t concurrentQueue = dispatch_queue_create("com.starming.gcddemo.c
    dispatch_apply(10, concurrentQueue, ^(size_t i) {
        NSLog(@"%zu", i);
    });
    NSLog(@"The end"); //这里有个需要注意的是，dispatch_apply这个是会阻塞主线程的。这个log打
}
```

dispatch_apply能避免线程爆炸，因为GCD会管理并发

```
- (void)dealWiththreadWithMaybeExplode:(BOOL)explode {
    dispatch_queue_t concurrentQueue = dispatch_queue_create("com.starming.gcddemo.c
    if (explode) {
        //有问题的情况，可能会死锁
        for (int i = 0; i < 999 ; i++) {
            dispatch_async(concurrentQueue, ^{
                NSLog(@"wrong %d", i);
                //do something hard
            });
        }
    } else {
        //会优化很多，能够利用GCD管理
        dispatch_apply(999, concurrentQueue, ^(size_t i){
            NSLog(@"correct %zu", i);
            //do something hard
        });
    }
}
```

```
    }
}
```

示例：

```
func downloadPhotosWithCompletion(completion: BatchPhotoDownloadingCompletionClosure) {
    var storedError: NSError!
    var downloadGroup = dispatch_group_create()
    let addresses = [OverlyAttachedGirlfriendURLString,
                     SuccessKidURLString,
                     LotsOfFacesURLString]

    dispatch_apply(UInt(addresses.count), GlobalUserInitiatedQueue) {
        i in
        let index = Int(i)
        let address = addresses[index]
        let url = NSURL(string: address)
        dispatch_group_enter(downloadGroup)
        let photo = DownloadPhoto(url: url!) {
            image, error in
            if let error = error {
                storedError = error
            }
            dispatch_group_leave(downloadGroup)
        }
        PhotoManager.sharedManager.addPhoto(photo)
    }

    dispatch_group_notify(downloadGroup, GlobalMainQueue) {
        if let completion = completion {
            completion(error: storedError)
        }
    }
}
```

Block组合Dispatch_groups

dispatch groups是专门用来监视多个异步任务。dispatch_group_t实例用来追踪不同队列中的不同任务。

当group里所有事件都完成GCD API有两种方式发送通知，第一种是dispatch_group_wait，会阻塞当前进程，等所有任务都完成或等待超时。第二种方法是使用dispatch_group_notify，异步执行闭包，不会阻塞。

第一种使用dispatch_group_wait的swift的例子：

```
func downloadPhotosWithCompletion(completion: BatchPhotoDownloadingCompletionClosure) {
    dispatch_async(GlobalUserInitiatedQueue) { // 因为dispatch_group_wait会阻塞当前进程
        var storedError: NSError!
        var downloadGroup = dispatch_group_create() // 创建一个dispatch_group

        for address in [OverlyAttachedGirlfriendURLString,
                        SuccessKidURLString,
                        LotsOfFacesURLString]
        {
            let url = NSURL(string: address)
            dispatch_group_enter(downloadGroup) // dispatch_group_enter是通知dispatch_group
            let photo = DownloadPhoto(url: url!) {
                image, error in
                if let error = error {
                    storedError = error
                }
                dispatch_group_leave(downloadGroup) // 保持和dispatch_group_enter
            }
            PhotoManager.sharedManager.addPhoto(photo)
        }

        dispatch_group_wait(downloadGroup, DISPATCH_TIME_FOREVER) // dispatch_group_wait
        dispatch_async(GlobalMainQueue) { // 这里可以保证所有图片任务都完成，然后在main
            if let completion = completion { // 执行闭包内容
                completion(error: storedError)
            }
        }
    }
}
```

```

    }
}
}

```

oc例子

```

- (void)dispatchGroupWaitDemo {
    dispatch_queue_t concurrentQueue = dispatch_queue_create("com.starming.gcdDemo.");
    dispatch_group_t group = dispatch_group_create();
    //在group中添加队列的block
    dispatch_group_async(group, concurrentQueue, ^{
        [NSThread sleepForTimeInterval:2.f];
        NSLog(@"1");
    });
    dispatch_group_async(group, concurrentQueue, ^{
        NSLog(@"2");
    });
    dispatch_group_wait(group, DISPATCH_TIME_FOREVER);
    NSLog(@"go on");
}

```

第二种使用dispatch_group_notify的swift的例子:

```

func downloadPhotosWithCompletion(completion: BatchPhotoDownloadingCompletionClosure) {
    // 不用加dispatch_async, 因为没有阻塞主进程
    var storedError: NSError!
    var downloadGroup = dispatch_group_create()

    for address in [OverlyAttachedGirlfriendURLString,
                    SuccessKidURLString,
                    LotsOfFacesURLString]
    {
        let url = NSURL(string: address)
        dispatch_group_enter(downloadGroup)
        let photo = DownloadPhoto(url: url!) {
            image, error in
            if let error = error {
                storedError = error
            }
        }
        dispatch_group_leave(downloadGroup)
        PhotoManager.sharedManager.addPhoto(photo)
    }

    dispatch_group_notify(downloadGroup, GlobalMainQueue) { // dispatch_group_notify
        if let completion = completion {
            completion(error: storedError)
        }
    }
}

```

oc例子

```

//dispatch_group_notify
- (void)dispatchGroupNotifyDemo {
    dispatch_queue_t concurrentQueue = dispatch_queue_create("com.starming.gcdDemo.");
    dispatch_group_t group = dispatch_group_create();
    dispatch_group_async(group, concurrentQueue, ^{
        NSLog(@"1");
    });
    dispatch_group_async(group, concurrentQueue, ^{
        NSLog(@"2");
    });
    dispatch_group_notify(group, dispatch_get_main_queue(), ^{
        NSLog(@"end");
    });
    NSLog(@"can continue");
}

```

```

//dispatch_group_wait
- (void)dispatchGroupWaitDemo {
    dispatch_queue_t concurrentQueue = dispatch_queue_create("com.starming.gcdDemo.");

```



```
dispatch_group_t group = dispatch_group_create();
//在group中添加队列的block
dispatch_group_async(group, concurrentQueue, ^{
    [NSThread sleepForTimeInterval:2.f];
    NSLog(@"1");
});
dispatch_group_async(group, concurrentQueue, ^{
    NSLog(@"2");
});
dispatch_group_wait(group, DISPATCH_TIME_FOREVER);
NSLog(@"can continue");
}
```

如何对现有API使用dispatch_group_t

```
//给Core Data的performBlock:添加groups。组合完成任务后使用dispatch_group_notify来运行一个
- (void)withGroup:(dispatch_group_t)group performBlock:(dispatch_block_t)block
{
    if (group == NULL) {
        [self performBlock:block];
    } else {
        dispatch_group_enter(group);
        [self performBlock:^(void){
            block();
            dispatch_group_leave(group);
        }]];
    }
}

//NSURLConnection也可以这样做
+ (void)withGroup:(dispatch_group_t)group
    sendAsynchronousRequest:(NSURLRequest *)request
    queue:(NSOperationQueue *)queue
    completionHandler:(void (^)(NSURLResponse*, NSData*, NSError*))handler
{
    if (group == NULL) {
        [self sendAsynchronousRequest:request
            queue:queue
            completionHandler:handler];
    } else {
        dispatch_group_enter(group);
        [self sendAsynchronousRequest:request
            queue:queue
            completionHandler:^(NSURLResponse *response, NSData *data, NSError
            handler(response, data, error);
            dispatch_group_leave(group);
        }]];
    }
}
```

注意事项

- dispatch_group_async等价于dispatch_group_enter() 和 dispatch_group_leave()的组合。
- dispatch_group_enter() 必须运行在 dispatch_group_leave() 之前。
- dispatch_group_enter() 和 dispatch_group_leave() 需要成对出现的

Dispatch Block

队列执行任务都是block的方式,

- 创建block

```
- (void)createDispatchBlock {
    //normal way
    dispatch_queue_t concurrentQueue = dispatch_queue_create("com.starming.gcddemo.c
    dispatch_block_t block = dispatch_block_create(0, ^{
        NSLog(@"run block");
    });
    dispatch_async(concurrentQueue, block);

    //QOS way
    dispatch_block_t qosBlock = dispatch_block_create_with_qos_class(0, QOS_CLASS_US
```

```

        NSLog(@"run qos block");
    });
    dispatch_async(concurrentQueue, qosBlock);
}

```

- `dispatch_block_wait`: 可以根据dispatch block来设置等待时间, 参数 `DISPATCH_TIME_FOREVER`会一直等待block结束

```

- (void)dispatchBlockWaitDemo {
    dispatch_queue_t serialQueue = dispatch_queue_create("com.starming.gcdDemo.serial", &dispatch_queue_attr_t);
    dispatch_block_t block = dispatch_block_create(0, ^{
        NSLog(@"star");
        [NSThread sleepForTimeInterval:5.f];
        NSLog(@"end");
    });
    dispatch_async(serialQueue, block);
    //设置DISPATCH_TIME_FOREVER会一直等到前面任务都完成
    dispatch_block_wait(block, DISPATCH_TIME_FOREVER);
    NSLog(@"ok, now can go on");
}

```

- `dispatch_block_notify`: 可以监视指定dispatch block结束, 然后再加入一个block到队列中。三个参数分别为, 第一个是需要监视的block, 第二个参数是需要提交执行的队列, 第三个是待加入到队列中的block

```

- (void)dispatchBlockNotifyDemo {
    dispatch_queue_t serialQueue = dispatch_queue_create("com.starming.gcdDemo.serial", &dispatch_queue_attr_t);
    dispatch_block_t firstBlock = dispatch_block_create(0, ^{
        NSLog(@"first block start");
        [NSThread sleepForTimeInterval:2.f];
        NSLog(@"first block end");
    });
    dispatch_async(serialQueue, firstBlock);
    dispatch_block_t secondBlock = dispatch_block_create(0, ^{
        NSLog(@"second block run");
    });
    //first block执行完才在serial queue中执行second block
    dispatch_block_notify(firstBlock, serialQueue, secondBlock);
}

```

- `dispatch_block_cancel`: iOS8后GCD支持对dispatch block的取消

```

- (void)dispatchBlockCancelDemo {
    dispatch_queue_t serialQueue = dispatch_queue_create("com.starming.gcdDemo.serial", &dispatch_queue_attr_t);
    dispatch_block_t firstBlock = dispatch_block_create(0, ^{
        NSLog(@"first block start");
        [NSThread sleepForTimeInterval:2.f];
        NSLog(@"first block end");
    });
    dispatch_block_t secondBlock = dispatch_block_create(0, ^{
        NSLog(@"second block run");
    });
    dispatch_async(serialQueue, firstBlock);
    dispatch_async(serialQueue, secondBlock);
    //取消secondBlock
    dispatch_block_cancel(secondBlock);
}

```

使用dispatch block object（调度块）在任务执行前进行取消

dispatch block object可以为队列中的对象设置 示例, 下载图片中途进行取消

```

func downloadPhotosWithCompletion(completion: BatchPhotoDownloadingCompletionClosure) {
    var storedError: NSError!
    let downloadGroup = dispatch_group_create()
    var addresses = [OverlyAttachedGirlfriendURLString,
                     SuccessKidURLString,
                     LotsOfFacesURLString]
    addresses += addresses + addresses // 扩展address数组, 复制3份
}

```

```

var blocks: [dispatch_block_t] = [] // 一个保存block的数组

for i in 0 ..< addresses.count {
    dispatch_group_enter(downloadGroup)
    let block = dispatch_block_create(DISPATCH_BLOCK_INHERIT_QOS_CLASS) { // {
        let index = Int(i)
        let address = addresses[index]
        let url = NSURL(string: address)
        let photo = DownloadPhoto(url: url!) {
            image, error in
            if let error = error {
                storedError = error
            }
            dispatch_group_leave(downloadGroup)
        }
        PhotoManager.sharedManager.addPhoto(photo)
    }
    blocks.append(block)
    dispatch_async(GlobalMainQueue, block) // 把这个block放到GlobalMainQueue上异
}

for block in blocks[3 ..< blocks.count] {
    let cancel = arc4random_uniform(2) // 随机返回一个整数，会返回0或1
    if cancel == 1 {
        dispatch_block_cancel(block) // 如果是1就取消block，这个只能发生在block还
        dispatch_group_leave(downloadGroup) // 因为已经dispatch_group_enter了，
    }
}

dispatch_group_notify(downloadGroup, GlobalMainQueue) {
    if let completion = completion {
        completion(error: storedError)
    }
}
}

```

Dispatch IO 文件操作

dispatch io读取文件的方式类似于下面的方式，多个线程去读取文件的切片数据，对于大的数据文件这样会比单线程要快很多。

```

dispatch_async(queue, ^{ /*read 0-99 bytes*/ });
dispatch_async(queue, ^{ /*read 100-199 bytes*/ });
dispatch_async(queue, ^{ /*read 200-299 bytes*/ });

```

- dispatch_io_create: 创建dispatch io
- dispatch_io_set_low_water: 指定切割文件大小
- dispatch_io_read: 读取切割的文件然后合并。

苹果系统日志API里用到了这个技术，可以在这里查看：<https://github.com/Apple-FOSS-Mirror/Libc/blob/2ca2ae74647714acfc18674c3114b1a5d3325d7d/gen/asl.c>

```

pipe_q = dispatch_queue_create("PipeQ", NULL);
//创建
pipe_channel = dispatch_io_create(DISPATCH_IO_STREAM, fd, pipe_q, ^(int err){
    close(fd);
});

*out_fd = fdpair[1];
//设置切割大小
dispatch_io_set_low_water(pipe_channel, SIZE_MAX);

dispatch_io_read(pipe_channel, 0, SIZE_MAX, pipe_q, ^(bool done, dispatch_data_t pip
    if (err == 0)
    {
        size_t len = dispatch_data_get_size(pipedata);
        if (len > 0)
        {
            //对每次切块数据的处理
            const char *bytes = NULL;
            char *encoded;

```

```
uint32_t eval;

dispatch_data_t md = dispatch_data_create_map(pipedata, (const void **)&
encoded = asl_core_encode_buffer(bytes, len);
asl_msg_set_key_val(aux, ASL_KEY_AUX_DATA, encoded);
free(encoded);
eval = _asl_evaluate_send(NULL, (aslmsg)aux, -1);
_asl_send_message(NULL, eval, aux, NULL);
asl_msg_release(aux);
dispatch_release(md);
}
}

if (done)
{
    //semaphore +1使得不需要再等待继续执行下去。
    dispatch_semaphore_signal(sem);
    dispatch_release(pipe_channel);
    dispatch_release(pipe_q);
}
});
```

Dispatch Source 用GCD监视进程

Dispatch Source用于监听系统的底层对象，比如文件描述符，Mach端口，信号量等。主要处理的事件如下表

方法	说明
DISPATCH_SOURCE_TYPE_DATA_ADD	数据增加
DISPATCH_SOURCE_TYPE_DATA_OR	数据OR
DISPATCH_SOURCE_TYPE_MACH_SEND	Mach端口发送
DISPATCH_SOURCE_TYPE_MACH_RECV	Mach端口接收
DISPATCH_SOURCE_TYPE_MEMORYPRESSURE	内存情况
DISPATCH_SOURCE_TYPE_PROC	进程事件
DISPATCH_SOURCE_TYPE_READ	读数据
DISPATCH_SOURCE_TYPE_SIGNAL	信号
DISPATCH_SOURCE_TYPE_TIMER	定时器
DISPATCH_SOURCE_TYPE_VNODE	文件系统变化
DISPATCH_SOURCE_TYPE_WRITE	文件写入

方法

- dispatch_source_create：创建dispatch source，创建后会处于挂起状态进行事件接收，需要设置事件处理handler进行事件处理。
- dispatch_source_set_event_handler：设置事件处理handler
- dispatch_source_set_cancel_handler：事件取消handler，就是在dispatch source释放前做些清理的事。
- dispatch_source_cancel：关闭dispatch source，设置的事件处理handler不会被执行，已经执行的事件handler不会取消。

```
NSRunningApplication *mail = [NSRunningApplication runningApplicationsWithBundleId:@"com.apple.mail"];
if (mail == nil) {
    return;
}
pid_t const pid = mail.processIdentifier;
self.source = dispatch_source_create(DISPATCH_SOURCE_TYPE_PROC, pid, DISPATCH_PROC_ALL, dispatch_queue_t main);
dispatch_source_set_event_handler(self.source, ^(){
    NSLog(@"Mail quit.");
});
//在事件源传到你的事件处理前需要调用dispatch_resume()这个方法
dispatch_resume(self.source);
```

监视文件夹内文件变化

```

NSURL *directoryURL; // assume this is set to a directory
int const fd = open([[directoryURL path] fileSystemRepresentation], O_EVTONLY);
if (fd < 0) {
    char buffer[80];
    strerror_r(errno, buffer, sizeof(buffer));
    NSLog(@"Unable to open \"%@\": %s (%d)", [directoryURL path], buffer, errno);
    return;
}
dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_VNODE, fd,
DISPATCH_VNODE_WRITE | DISPATCH_VNODE_DELETE, DISPATCH_TARGET_QUEUE_DEFAULT);
dispatch_source_set_event_handler(source, ^(){
    unsigned long const data = dispatch_source_get_data(source);
    if (data & DISPATCH_VNODE_WRITE) {
        NSLog(@"The directory changed.");
    }
    if (data & DISPATCH_VNODE_DELETE) {
        NSLog(@"The directory has been deleted.");
    }
});
dispatch_source_set_cancel_handler(source, ^(){
    close(fd);
});
self.source = source;
dispatch_resume(self.source);
//还要注意需要用DISPATCH_VNODE_DELETE 去检查监视的文件或文件夹是否被删除，如果删除了就停止监听

```

NSTimer在主线程的runloop里会在runloop切换其它模式时停止，这时就需要手动在子线程开启一个模式为NSRunLoopCommonModes的runloop，如果不想开启一个新的runloop可以用不跟runloop关联的dispatch source timer，如下。

```

dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER, 0, 0, [
dispatch_source_set_event_handler(source, ^(){
    NSLog(@"Time flies.");
});
dispatch_time_t start
dispatch_source_set_timer(source, DISPATCH_TIME_NOW, 5ull * NSEC_PER_SEC, 100ull * NS
self.source = source;
dispatch_resume(self.source);

```

Dispatch Semaphore和的介绍

另外一种保证同步的方法。使用dispatch_semaphore_signal加1dispatch_semaphore_wait减1，为0时等待的设置方式来达到线程同步的目的和同步锁一样能够解决资源抢占的问题。

```

//dispatch semaphore
- (void)dispatchSemaphoreDemo {
    //创建semaphore
    dispatch_semaphore_t semaphore = dispatch_semaphore_create(0);
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        NSLog(@"start");
        [NSThread sleepForTimeInterval:1.f];
        NSLog(@"semaphore +1");
        dispatch_semaphore_signal(semaphore); //+1 semaphore
    });
    dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
    NSLog(@"continue");
}

```

锁

这里简单介绍下iOS中常用的各种锁和他们的性能。

- NSRecursiveLock：递归锁，可以在一个线程中反复获取锁不会造成死锁，这个过程会记录获取锁和释放锁的次数来达到何时释放的作用。
- NSDistributedLock：分布锁，基于文件方式的锁机制，可以跨进程访问。

- NSConditionLock：条件锁，用户定义条件，确保一个线程可以获取满足一定条件的锁。因为线程间竞争会涉及到条件锁检测，系统调用上下切换频繁导致耗时是几个锁里最长的。
- OSSpinLock：自旋锁，不进入内核，减少上下文切换，性能最高，但抢占多时会占用较多cpu，好点多，这时使用pthread_mutex较好。
- pthread_mutex_t：同步锁基于C语言，底层api性能高，使用方法和其它的类似。
- @synchronized：更加简单。

dispatch_suspend和dispatch_resume挂起和恢复队列

dispatch_suspend这里挂起不会暂停正在执行的block，只是能够暂停还没执行的block。

dispatch_set_context和dispatch_get_context

GCD深入操作

- 缓冲区：dispatch_data_t基于零碎的内存区域，使用dispatch_data_apply来遍历，还可以用dispatch_data_create_subrange来创建一个不做任何拷贝的子区域
- I/O调度：使用GCD提供的dispatch_io_read，dispatch_io_write和dispatch_io_close
- 测试：使用dispatch_benchmark小工具
- 原子操作：libkern/OSAtomic.h里可以查看那些函数，用于底层多线程编程。

GCD死锁

当前串行队列里面同步执行当前串行队列就会死锁，解决的方法就是将同步的串行队列放到另外一个线程就能够解决。

```
- (void)deadLockCase1 {
    NSLog(@"1");
    //主队列的同步线程，按照FIFO的原则（先入先出），2排在3后面会等3执行完，但因为同步线程，3又要等
    dispatch_sync(dispatch_get_main_queue(), ^{
        NSLog(@"2");
    });
    NSLog(@"3");
}
- (void)deadLockCase2 {
    NSLog(@"1");
    //3会等2，因为2在全局并行队列里，不需要等待3，这样2执行完回到主队列，3就开始执行
    dispatch_sync(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{
        NSLog(@"2");
    });
    NSLog(@"3");
}
- (void)deadLockCase3 {
    dispatch_queue_t serialQueue = dispatch_queue_create("com.starming.gcddemo.serial", DISPATCH_QUEUE_SERIAL);
    NSLog(@"1");
    dispatch_async(serialQueue, ^{
        NSLog(@"2");
        //串行队列里面同步一个串行队列就会死锁
        dispatch_sync(serialQueue, ^{
            NSLog(@"3");
        });
        NSLog(@"4");
    });
    NSLog(@"5");
}
- (void)deadLockCase4 {
    NSLog(@"1");
    dispatch_async(dispatch_get_global_queue(0, 0), ^{
        NSLog(@"2");
        //将同步的串行队列放到另外一个线程就能够解决
        dispatch_sync(dispatch_get_main_queue(), ^{
            NSLog(@"3");
        });
        NSLog(@"4");
    });
    NSLog(@"5");
}
```

```
- (void)deadLockCase5 {
    dispatch_async(dispatch_get_global_queue(0, 0), ^{
        NSLog(@"1");
        //回到主线程发现死循环后面就没法执行了
        dispatch_sync(dispatch_get_main_queue(), ^{
            NSLog(@"2");
        });
        NSLog(@"3");
    });
    NSLog(@"4");
    //死循环
    while (1) {
        //
    }
}
```

GCD实际使用

FMDB如何使用dispatch_queue_set_specific和dispatch_get_specific来防止死锁

作用类似objc_setAssociatedObject跟objc_getAssociatedObject

```
static const void * const kDispatchQueueSpecificKey = &kDispatchQueueSpecificKey;
//创建串行队列，所有数据库的操作都在这个队列里
_queue = dispatch_queue_create([NSString stringWithFormat:@"fmdb.%@", self] UTF8String, NULL);
//标记队列
dispatch_queue_set_specific(_queue, kDispatchQueueSpecificKey, (__bridge void *)self);

//检查是否是同一个队列来避免死锁的方法
- (void)inDatabase:(void (^)(FMDatabase *db))block {
    FMDatabaseQueue *currentSyncQueue = (__bridge id)dispatch_get_specific(kDispatchQueueSpecificKey);
    assert(currentSyncQueue != self && "inDatabase: was called reentrantly on the same queue");
    [currentSyncQueue synchronousQueue];
    block();
}
```

DTCoreText使用GCD加快解析速度

DTCoreText采用的是SAX解析，iOS自带了XML/HTML的解析引擎libxml，提供了两个解析接口，DOM解析和SAX解析，前者使用简单但是占用内存多，SAX解析由于不会返回一个dom树，采用的是查到一个标签比如回调startElement方法碰到内容就回调_characters碰到类似就回调_endElement这样的方式。

根据这种解析方式DTCoreText使用多线程解析能够更快的解析，DTHTMLAttributedStringBuilder使用三个dispatch_queue

- _dataParsingQueue：解析html的
- _treeBuildingQueue：生成dom树的
- _stringAssemblyQueue：组装NSAttributedString的 获取三个队列全部完成采用了dispatch_group的dispatch_group_wait这种阻塞同步方式来返回结果。

iOS系统版本新特性

iOS8

iOS8新加了一个功能叫Quality of Service(QoS)，里面提供了一下几个更容易理解的枚举名来使用user interactive，user initiated，utility和background。下面的表做了对比

Global queue	Corresponding QoS class	说明
--------------	-------------------------	----

Global queue	Corresponding QoS class	说明
Main thread	NSQualityOfServiceUserInteractive	UI 相关，交互等
DISPATCH_QUEUE_PRIORITY_HIGH	NSQualityOfServiceUserInitiated	用户发起需要马上得到结果进行后续任务
DISPATCH_QUEUE_PRIORITY_DEFAULT	NSQualityOfServiceDefault	默认的不应该使用这个设置任务
DISPATCH_QUEUE_PRIORITY_LOW	NSQualityOfServiceUtility	花费时间稍多比如下载，需要几秒或几分钟的
DISPATCH_QUEUE_PRIORITY_BACKGROUND	NSQualityOfServiceBackground	不可见后台的操作可能需要几分钟至小时的

参考资料

WWDC

- Building Responsive and Efficient Apps with GCD：
<https://developer.apple.com/videos/play/wwdc2015-718/>

文档

- 官方文档：

https://developer.apple.com/library/prerelease/ios/documentation/Performance/Reference/GCD_libdispatch_Ref/