多线程

参考简书博客

GCD

GCD简介

Grand Central Dispatch，用于优化应用程序以支持多核处理器以及其他对称多处理系统，在线程池模式的基础上执行并发任务。

- 可用于多核并行运算
- 自动利用更多的CPU内核
- 自动管理线程的生命周期（创建线程、调度、销毁线程等）
- 程序员只需要关心GCD执行什么任务，不需要关心线程管理

GCD任务和队列

- 任务

在线程中执行的那段代码。在GCD中是放在block中的。可以[同步执行]和[异步执行]，两者主要区别是：是否等待队列的任务执行结束，以及是否具备开启新线程的能力。

  - 同步执行
    - 同步添加任务到指定的队列中，在添加的任务执行结束之前，会一直等待，直到队列里面的任务完成之后再继续执行
    - 不具备开启新线程的能力
  - 异步执行
    - 异步添加任务到指定的队列中，不会做任何等待，可以继续执行任务
    - 具备开启新线程的能力

- 队列

这里的队列指的是执行任务的等待队列，用来存放任务的队列。采用FIFO的原则。在GCD中有两种队列，[串行队列]和[并发队列]。两者的主要区别是：执行顺序的不同，以及开启线程数不同。

  - 串行队列
    - 每次只有一个任务被执行，任务一个接着一个执行
  - 并发队列
    - 可以多个任务同时执行（开启多个线程，同时执行任务，只在异步方法下才有效）

GCD的使用步骤

- 创建一个队列
  - 使用dispatch_queue_create方法创建队列

```
// 串行队列的创建方法
dispatch_queue_t queue = dispatch_queue_create("net.bujige.testQueue", DISPATCH_QUEUE_SERIAL);
// 并发队列的创建方法
dispatch_queue_t queue = dispatch_queue_create("net.bujige.testQueue", DISPATCH_QUEUE_CONCURRENT);

```

  - 对于串行队列，默认提供主队列：Main Dispatch Queue
    - 所有放在主队列中的任务，都会放到主线程执行，使用dispatch_get_main_queue()获得主队列
  - 对于并发队列，默认提供全局并发队列：Global Dispatch Queue
    - dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,0);
- 将任务添加到等待队列中，然后系统就会根据任务类型执行任务

```
// 同步执行任务创建方法
dispatch_sync(queue, ^{});
// 异步执行任务创建方法
dispatch_sync(queue, ^{});
```

- 不同队列 + 不同任务
  - 在主线程，同步执行 + 主队列 = 死锁。主队列中追加的同步任务，和主线程本身的任务两者之间相互等待

GCD之间的通信

在iOS中，把UI刷新放在主线程里，把一些耗时的操作放在其他线程，如图片下载、文件上传等。而当我们完成了耗时操作，回到主线程，就用到了线程通信。

```
/**
 * 线程间通信
 */
- (void)communication {
    // 获取全局并发队列
```

```
    dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    // 获取主队列
    dispatch_queue_t mainQueue = dispatch_get_main_queue();

    dispatch_async(queue, ^{
        // 异步追加任务 1
        [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
        NSLog(@"1---%@",[NSThread currentThread]);      // 打印当前线程

        // 回到主线程
        dispatch_async(mainQueue, ^{
            // 追加在主线程中执行的任务
            [NSThread sleepForTimeInterval:2];              // 模拟耗时操作
            NSLog(@"2---%@",[NSThread currentThread]);      // 打印当前线程
        });
    });
}

```

GCD的其他方法

- 栅栏方法 dispatch_barrier_async
- 延时执行方法 dispatch_after
- 一次性代码 dispatch_once
- 快速迭代方法 dispatch_apply 按照指定次数将指定任务追加到指定队列
- 队列组 dispatch_group
- 信号量 dispatch_semaphore

线程安全

简书博客

为避免多个线程同时访问同一块资源产生的数据错误和数据不安全问题，使用线程锁。

@synchronized（互斥锁）

```
NSObject *obj = [[NSObject alloc] init];
dispatch_async(... , ^{
    @synchronized(obj)
});
```

`synchronized`指令使用的obj为该锁的唯一标识，只有当标识相同时，才满足互斥。

GCD - 信号量 dispatch_semaphore

```
dispatch_semaphore_t signal = dispatch_semaphore_create(1);
dispatch_time_t overTime = dispatch_time(DISPATCH_TIME_NOW, 3 * NSEC_PER_SEC);
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    dispatch_semaphore_wait(signal, overTime);
        NSLog(@"需要线程同步的操作1 开始");
        sleep(2);
        NSLog(@"需要线程同步的操作1 结束");
    dispatch_semaphore_signal(signal);
});

```

dispatch_semaphore_create(long value)传入参数为long，返回一个值为value的信号量，这里的value必须大于等于零，否则会返回NULL。
dispatch_semaphore_wait(dsema,timeout)是，如果dsema的信号量值大于0，那么自动执行接下来的语句，否则会阻塞，直到dsema的信号量大于0。或者等到timeout的时候，自动执行后面的语句。
dispatch_semaphore_signal(dsema)作用是给dsema的值加1。

OC 锁

NSLock

```
NSLock *lock = [[NSLock alloc] init];
  dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        //[lock lock];
        [lock lockBeforeDate:[NSDate date]];// 1
            NSLog(@"需要线程同步的操作1 开始"); // 1
```

```
            sleep(2);
            NSLog(@"需要线程同步的操作1 结束"); // 3
        [lock unlock]; // 3

    });

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        sleep(1);
        if ([lock tryLock]) {//尝试获取锁，如果获取不到返回NO，不会阻塞该线程
            NSLog(@"锁可用的操作");
            [lock unlock];
        }else{
            NSLog(@"锁不可用的操作"); // 2
        }

        NSDate *date = [[NSDate alloc] initWithTimeIntervalSinceNow:3];
        if ([lock lockBeforeDate:date]) {//尝试在未来的3s内获取锁，并阻塞该线程，如果3s内获取不到恢复线程，返回NO,不会阻塞该线程
            NSLog(@"没有超时，获得锁"); // 4
            [lock unlock];
        }else{
            NSLog(@"超时，没有获得锁");
        }

    });

```

NSLock是Cocoa提供的最基本的锁对象，除lock和unlock方法，还提供了tryLock和lockBeforeData两个方法，前一个方法会尝试加锁，如果锁不可用，并不会阻塞。而lockBeforeDate会在指定date之前尝试加锁并阻塞，如果在指定时间内不能加锁，就返回NO。
注意：**lock和unlock必须配对使用**。

NSRecursiveLock 递归锁

```
    NSRecursiveLock *lock = [[NSRecursiveLock alloc] init];
```

```
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        static void (^RecursiveMethod)(int);
        RecursiveMethod = ^(int value) {
            [lock lock];
            if (value > 0) {
                NSLog(@"value = %d", value);
                sleep(1);
                RecursiveMethod(value - 1);
            }
            [lock unlock];
        };
        RecursiveMethod(5);
    });

```

递归锁，可以被同一个线程多次请求，不会引起死锁，主要用在循环或递归操作中。

NSConditionLock 条件锁

```
    NSConditionLock *lock = [[NSConditionLock alloc] init];
    NSMutableArray *products = [NSMutableArray array];
    NSInteger HAS_DATA = 1;
    NSInteger NO_DATA = 0;
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        while (1) {
            [lock lockWhenCondition:NO_DATA];
            [products addObject:[[NSObject alloc] init]];
            NSLog(@"produce a product,总量:%zi",products.count);
            [lock unlockWithCondition:HAS_DATA];
            sleep(1);
        }
    });
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
```

```
        while (1) {
            NSLog(@"wait for product");
            [lock lockWhenCondition:HAS_DATA];
            [products removeObjectAtIndex:0];
            NSLog(@"custome a product");
            [lock unlockWithCondition:NO_DATA];
        }
    });
```