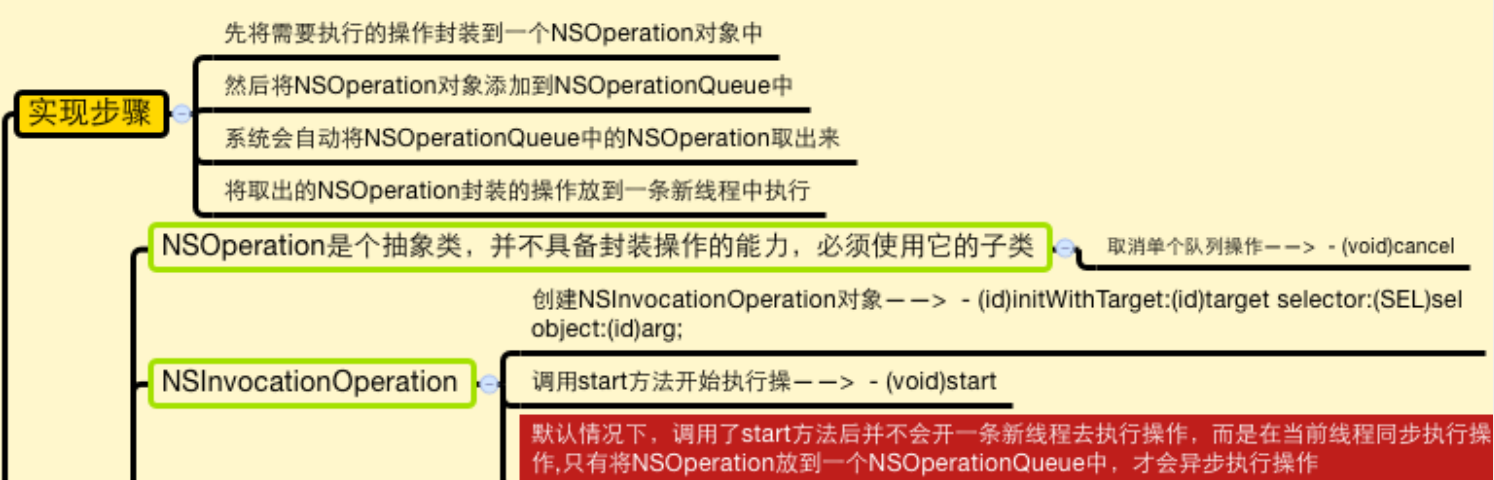
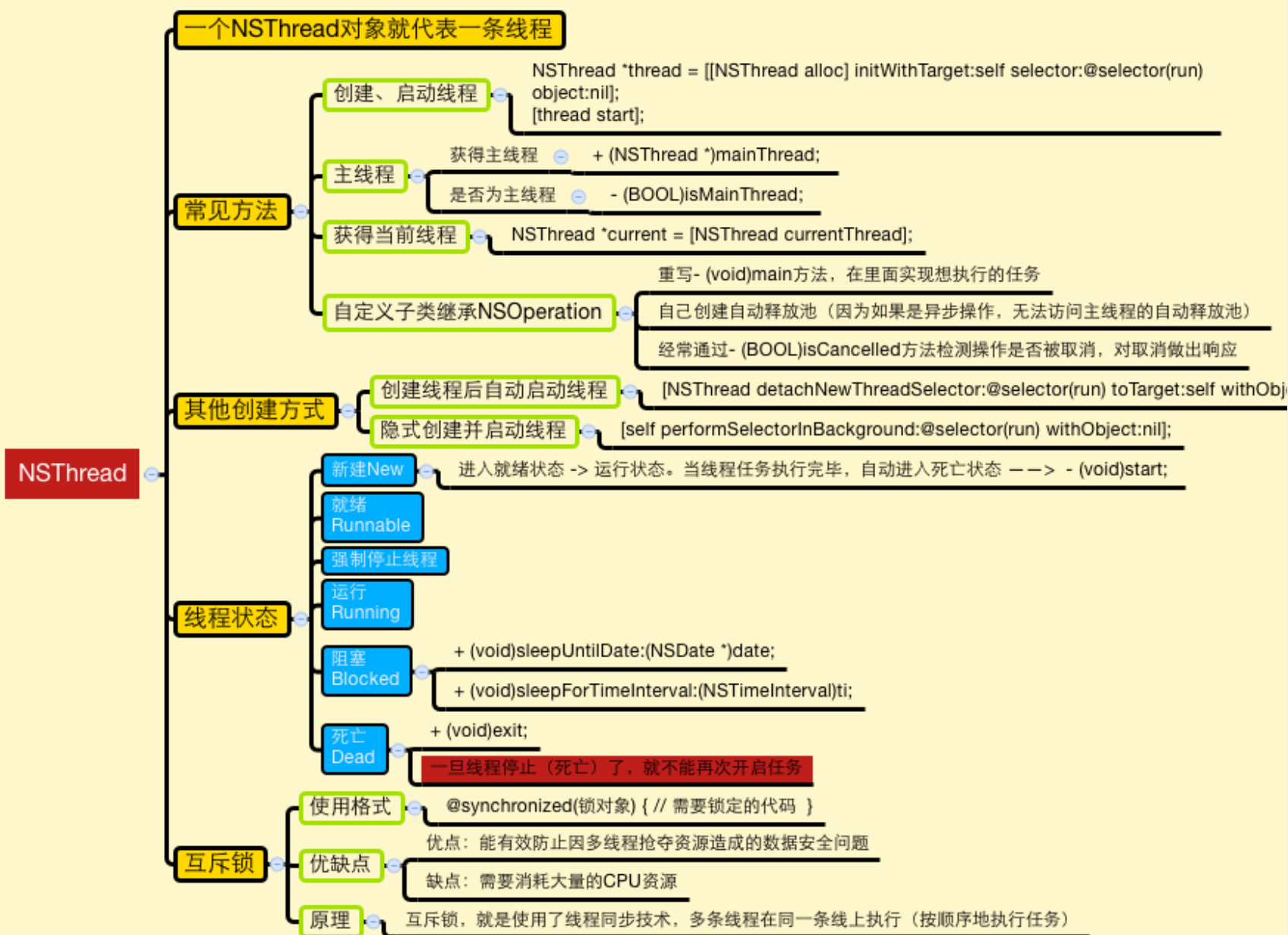
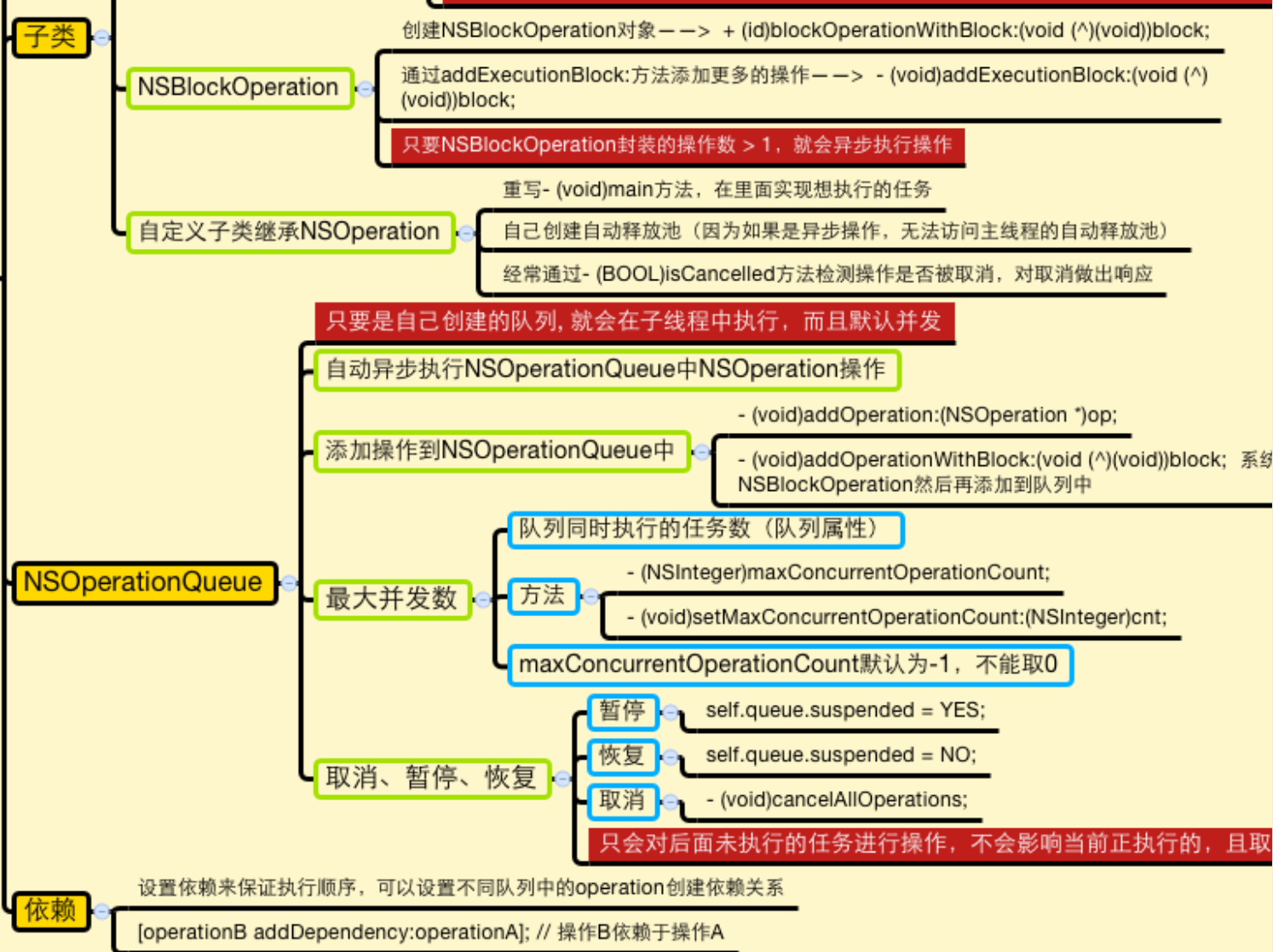




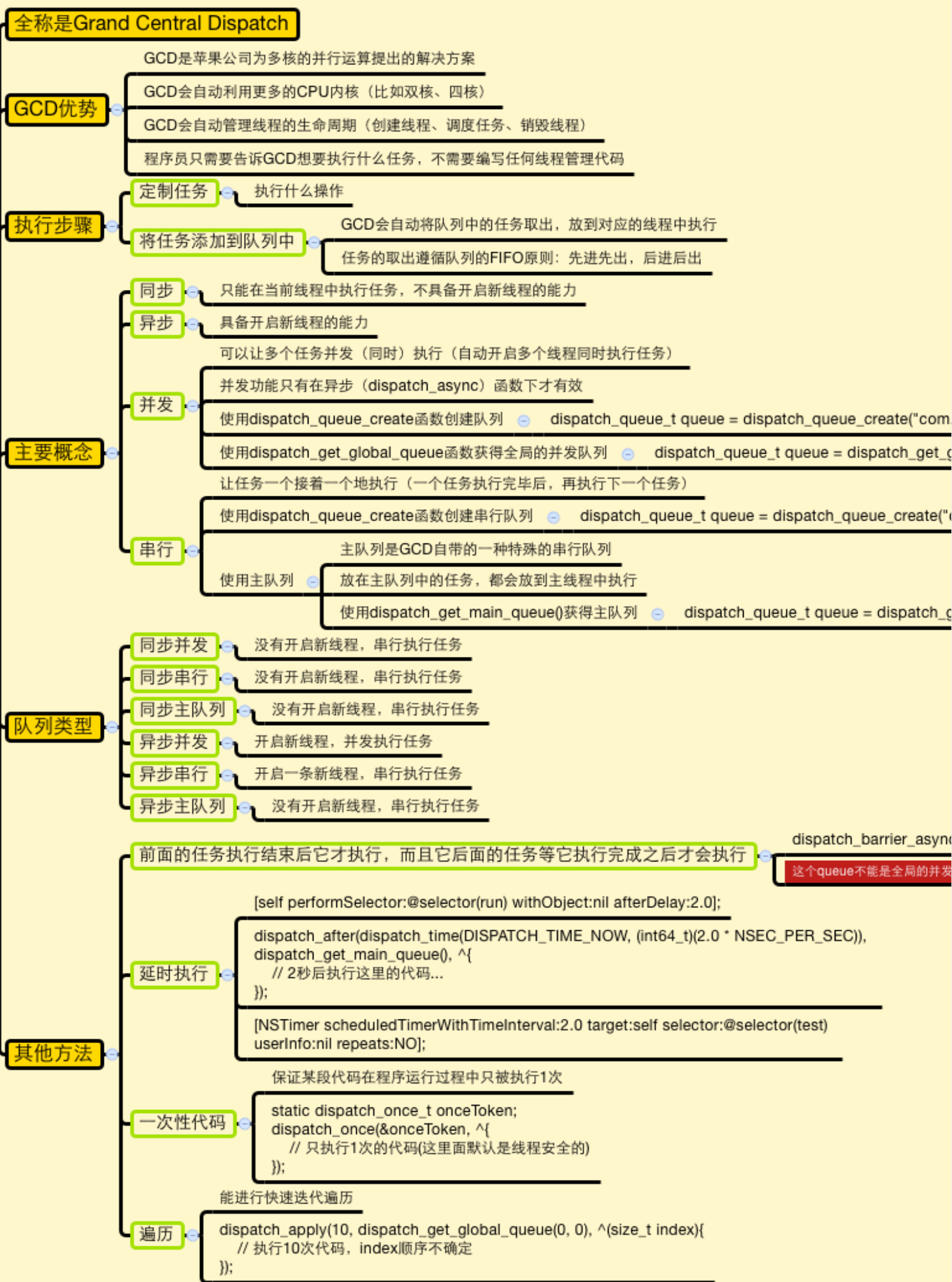
iOS多线程-概念



NSOperation



GCD



一. 多线程基础

1. 进程

进程是指在系统中正在运行的一个应用程序
每个进程之间是独立的，每个进程均运行在其专用且受保护的内存空间内

2.线程

1个进程要想执行任务，必须得有线程（每1个进程至少要有1条线程，称为主线程）
一个进程（程序）的所有任务都在线程中执行

3. 进程和线程的比较

- 1.线程是CPU调用(执行任务)的最小单位。
- 2.进程是CPU分配资源的最小单位。
- 3.一个进程中至少要有一个线程。
- 4.同一个进程内的线程共享进程的资源。

4. 线程的串行

1个线程中任务的执行是串行的
如果要在1个线程中执行多个任务，那么只能一个一个地按顺序执行这些任务
也就是说，在同一时间内，1个线程只能执行1个任务

5. 多线程

1个进程中可以开启多条线程，每条线程可以并行（同时）执行不同的任务
多线程技术可以提高程序的执行效率

6. 多线程原理

同一时间，**CPU只能处理1条线程**，只有**1条线程在工作（执行）**，多线程并发（同时）执行，其实是**CPU快速地在多条线程之间调度（切换）**，如果CPU调度线程的时间足够快，就造成了多线程并发执行的假象。
那么如果线程非常非常多，会发生什么情况？
CPU会在N多线程之间调度，CPU会累死，消耗大量的CPU资源，同时每条线程被调度执行的频次也会会降低（线程的执行效率降低）。
因此我们一般只开3-5条线程。

7. 多线程优缺点

多线程的优点
能适当提高程序的执行效率
能适当提高资源利用率（CPU、内存利用率）

多线程的缺点
创建线程是有开销的，iOS下主要成本包括：内核数据结构（大约1KB）、栈空间（子线程512KB、主线程1MB，也可以使用-setStackSize:设置，但必须是4K的倍数，而且最小是16K），创建线程大约需要90毫秒的创建时间
如果开启大量的线程，会降低程序的性能，线程越多，CPU在调度线程上的开销就越大。
程序设计更加复杂：比如线程之间的通信、多线程的数据共享等问题。

8. 多线程的应用

主线程的主要作用
显示\刷新UI界面
处理UI事件（比如点击事件、滚动事件、拖拽事件等）
主线程的使用注意
别将比较耗时的操作放到主线程中

耗时操作会卡住主线程，严重影响UI的流畅度，给用户一种“卡”的坏体验

将耗时操作放在子线程中执行，提高程序的执行效率

二. 多线程实现方案

技术方案	简介	语言	线程生命周期	使用频率
pthread	<ul style="list-style-type: none">一套通用的多线程API适用于Unix\Linux\Windows等系统跨平台\可移植使用难度大	C	程序员管理	几乎不用
NSThread	<ul style="list-style-type: none">使用更加面向对象简单易用，可直接操作线程对象	OC	程序员管理	偶尔使用
GCD	<ul style="list-style-type: none">旨在替代NSThread等线程技术充分利用设备的多核	C	自动管理	经常使用
NSOperation	<ul style="list-style-type: none">基于GCD（底层是GCD）比GCD多了一些更简单实用的功能使用更加面向对象	OC	自动管理	经常使用

多线程实现的四种方案

1. pthread的简单使用（了解）






```
-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    //创建线程
    pthread_t thread;
    /*
    第一个参数pthread_t *restrict:线程对象
    第二个参数const pthread_attr_t *restrict:线程属性
    第三个参数void *(*)(void *) :指向函数的指针
    第四个参数void *restrict:函数的参数
    */
    pthread_create(&thread, NULL,run ,NULL);
}
//void *(*)(void *)
void *run(void *param)
{
    for (NSInteger i =0 ; i<10000; i++) {
        NSLog(@"%zd--%@",i,[NSThread currentThread]);
    }
    return NULL;
}
```






2. NSThread的使用

2.1 创建线程





```
// 方法一：创建线程，需要自己开启线程
NSThread *thread = [[NSThread alloc]initWithTarget:self selector:@selector(run)
object:nil];
// 开启线程
[thread start];

// 方法二：创建线程后自动启动线程
```

```
[NSThread detachNewThreadSelector:@selector(run) toTarget:self withObject:nil];
```

```
// 方法三：隐式创建并启动线程
[self performSelectorInBackground:@selector(run) withObject:nil];
```



后面两种方法都不用我们开启线程，相对方便快捷，但是没有办法拿到子线程对象，没有办法对子线程进行更详细的设置，例如线程名字和优先级等。

2.2 NSThread的属性



```
// 获取当前线程
+ (NSThread *)currentThread;
// 创建启动线程
+ (void)detachNewThreadSelector:(SEL)selector toTarget:(id)target withObject:
(id)argument;
// 判断是否是多线程
+ (BOOL)isMultiThreaded;
// 线程休眠 NSDate 休眠到什么时候
+ (void)sleepUntilDate:(NSDate *)date;
// 线程休眠时间
+ (void)sleepForTimeInterval:(NSTimeInterval)ti;
// 结束/退出当前线程
+ (void)exit;
// 获取当前线程优先级
+ (double)threadPriority;
// 设置线程优先级 默认为0.5 取值范围为0.0 - 1.0
// 1.0优先级最高
// 设置优先级
+ (BOOL)setThreadPriority:(double)p;
// 获取指定线程的优先级
- (double)threadPriority NS_AVAILABLE(10_6, 4_0);
- (void)setThreadPriority:(double)p NS_AVAILABLE(10_6, 4_0);

// 设置线程的名字
- (void)setName:(NSString *)n NS_AVAILABLE(10_5, 2_0);
- (NSString *)name NS_AVAILABLE(10_5, 2_0);

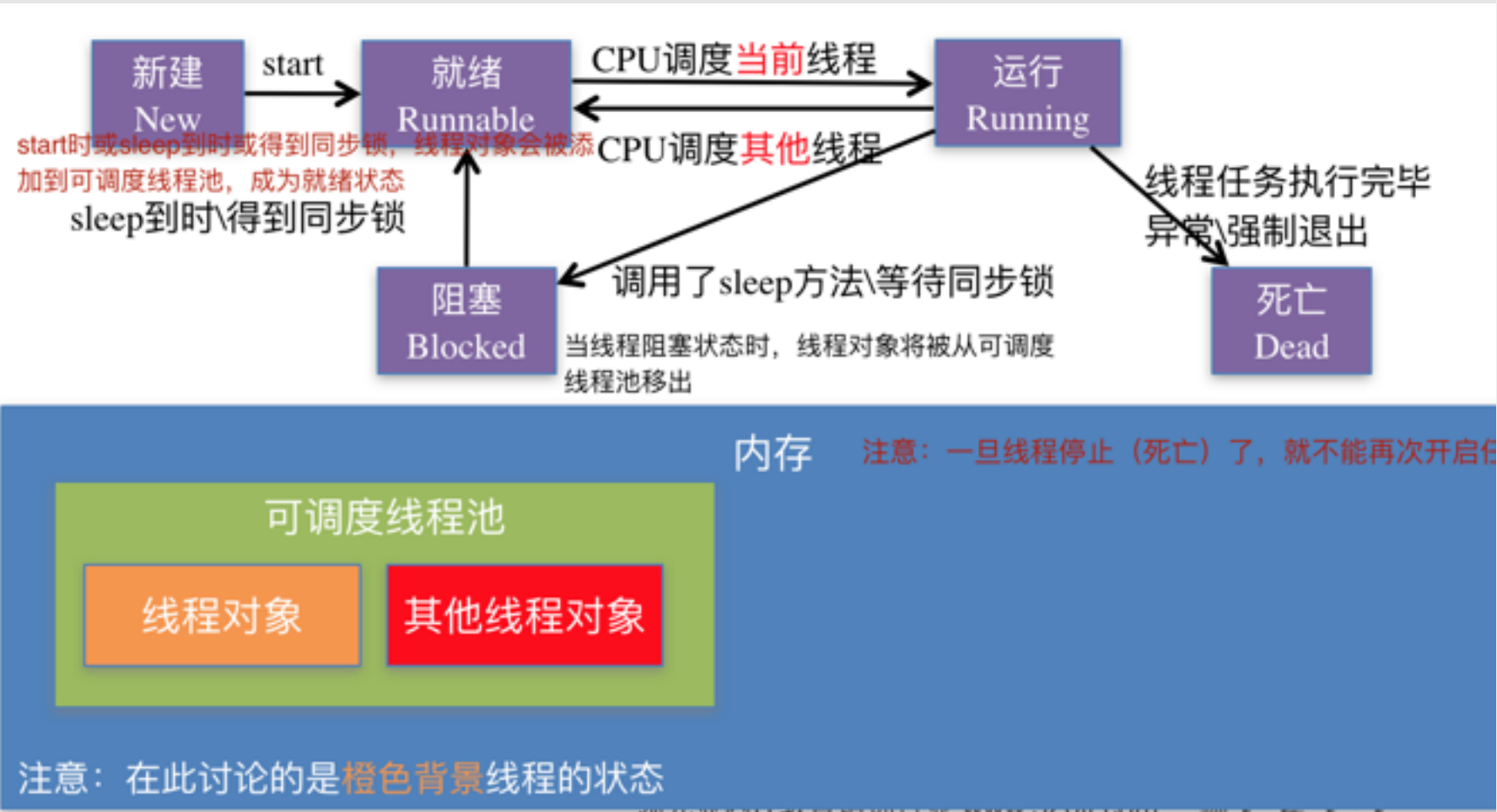
// 判断指定的线程是否是 主线程
- (BOOL)isMainThread NS_AVAILABLE(10_5, 2_0);
// 判断当前线程是否是主线程
+ (BOOL)isMainThread NS_AVAILABLE(10_5, 2_0); // reports whether current thread is main
// 获取主线程
+ (NSThread *)mainThread NS_AVAILABLE(10_5, 2_0);

- (id)init NS_AVAILABLE(10_5, 2_0); // designated initializer
// 创建线程
- (id)initWithTarget:(id)target selector:(SEL)selector object:(id)argument
NS_AVAILABLE(10_5, 2_0);
// 指定线程是否在执行
- (BOOL)isExecuting NS_AVAILABLE(10_5, 2_0);
// 线程是否完成
- (BOOL)isFinished NS_AVAILABLE(10_5, 2_0);
// 线程是否被取消 （是否给当前线程发过取消信号）
- (BOOL)isCancelled NS_AVAILABLE(10_5, 2_0);
// 发送线程取消信号的 最终线程是否结束 由 线程本身决定
- (void)cancel NS_AVAILABLE(10_5, 2_0);
// 启动线程
- (void)start NS_AVAILABLE(10_5, 2_0);

// 线程主函数 在线程中执行的函数 都要在-main函数中调用，自定义线程中重写-main方法
- (void)main NS_AVAILABLE(10_5, 2_0); // thread body metho
```



2.3 NSThread线程的状态（了解）



线程的状态

启动线程

```
- (void) start;
```

// 进入就绪状态 -> 运行状态。当线程任务执行完毕, 自动进入死亡状态

阻塞 (暂停) 线程

```
+ (void) sleepUntilDate: (NSDate *) date;
```

```
+ (void) sleepForTimeInterval: (NSTimeInterval) ti;
```

// 进入阻塞状态

强制停止线程

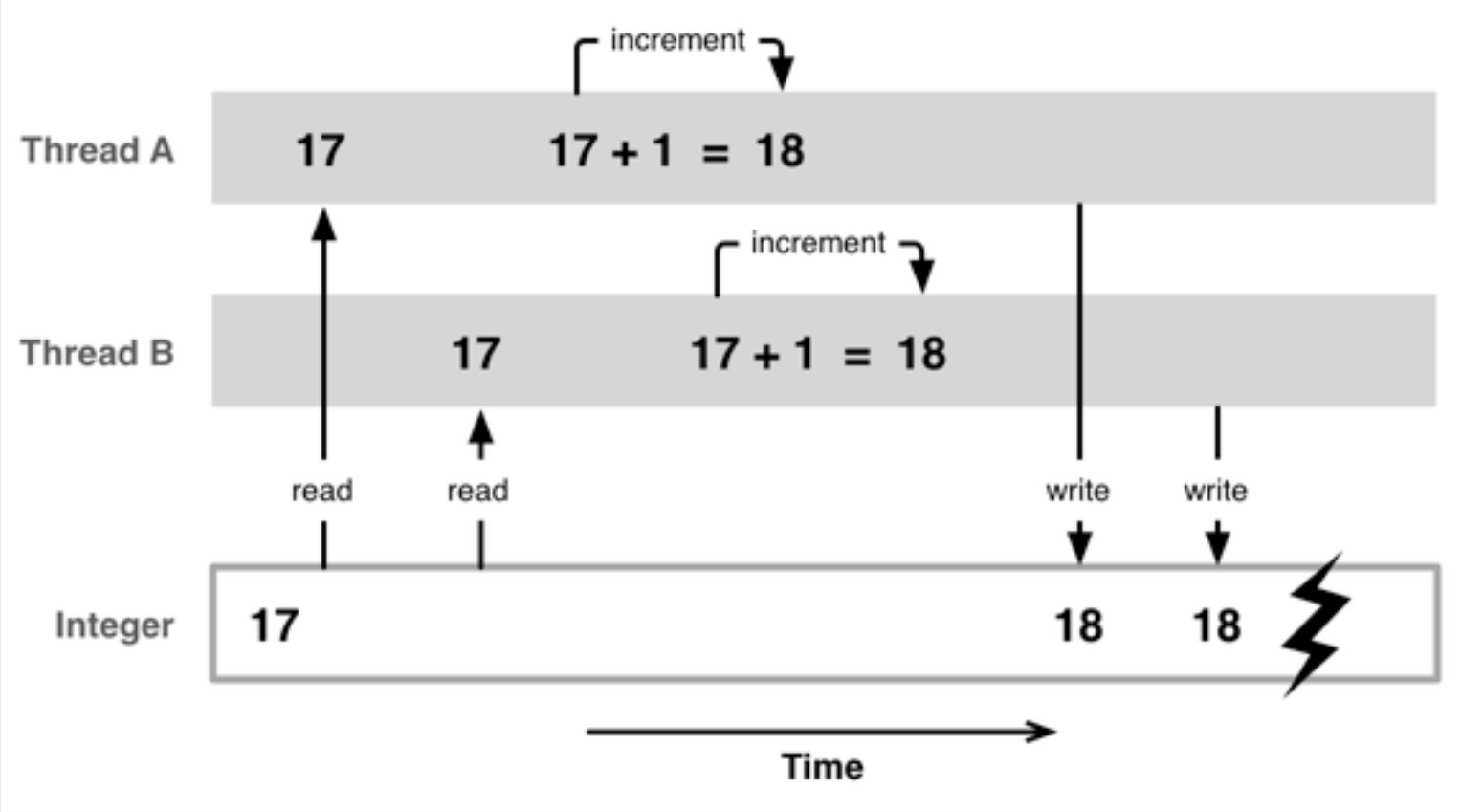
```
+ (void) exit;
```

// 进入死亡状态

2.4 NSThread多线程安全隐患

多线程安全隐患的原因：1块资源可能会被多个线程共享，也就是多个线程可能会访问同一块资源，比如多个线程访问同一个对象、同一个变量、同一个文件。

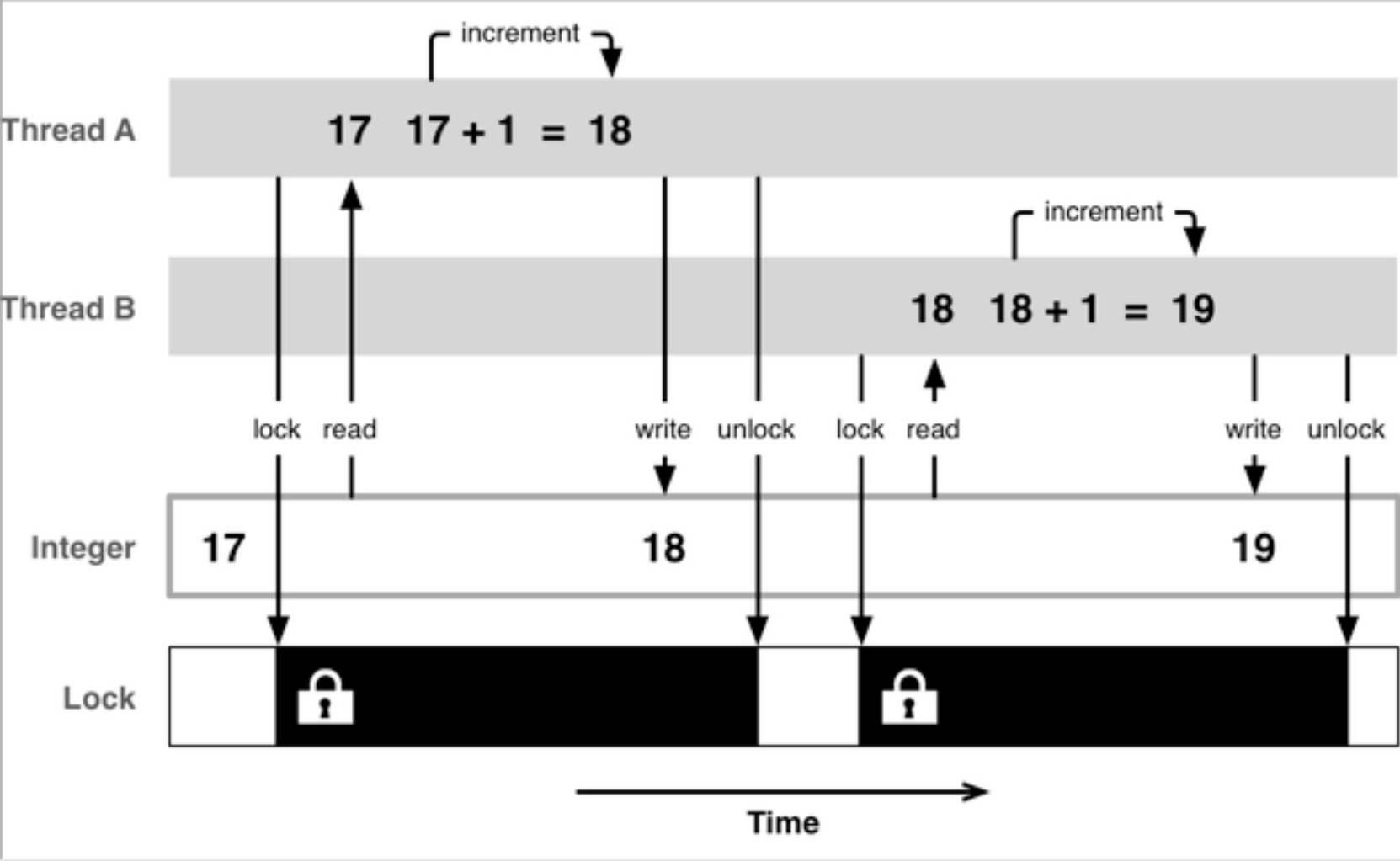
那么当多个线程访问同一块资源时，很容易引发数据错乱和数据安全问题。



安全隐患分析

通过上图我们发现，当线程A访问数据并对数据进行操作的同时，线程B访问的数据还是没有更新的数据，线程B同样对数据进行操作，当两个线程结束返回时，就会发生数据错乱的问题。

那么我们看下图的解决方法：添加互斥锁。



安全隐患解决

我们可以看出，当线程A访问数据并对数据进行操作的时候，数据被加上一把锁，这个时候其他线程都无法访问数据，知道线程A结束返回数据，线程B此时在访问数据并修改，就不会造成数据错乱了。

下面我们来看一下互斥锁的使用：

互斥锁使用格式

```
@synchronized(锁对象) {  
    // 需要锁定的代码  
}
```

- 互斥锁的使用前提：多条线程抢夺同一块资源时
- 注意：锁定1份代码只用1把锁，用多把锁是无效的
- 互斥锁的优缺点
 - 优点：能有效防止因多线程抢夺资源造成的数据安全问题
 - 缺点：需要消耗大量的CPU资源

下面通过一个售票实例来看一下线程安全的重要性

```
#import "ViewController.h"  
  
@interface ViewController ()  
  
@property(nonatomic, strong)NSThread *thread01;  
@property(nonatomic, strong)NSThread *thread02;  
@property(nonatomic, strong)NSThread *thread03;  
@property(nonatomic, assign)NSInteger numTicket;  
  
//@property(nonatomic, strong)NSObject *obj;  
@end  
  
@implementation ViewController  
  
- (void)viewDidLoad {  
    [super viewDidLoad];  
    // 总票数为30  
}
```



```
self.numTicket = 30;
self.thread01 = [[NSThread alloc]initWithTarget:self selector:@selector(saleTicket)
object:nil];
self.thread01.name = @"售票员01";
self.thread02 = [[NSThread alloc]initWithTarget:self selector:@selector(saleTicket)
object:nil];
self.thread02.name = @"售票员02";
self.thread03 = [[NSThread alloc]initWithTarget:self selector:@selector(saleTicket)
object:nil];
self.thread03.name = @"售票员03";
}
-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    [self.thread01 start];
    [self.thread02 start];
    [self.thread03 start];
}
// 售票
-(void)saleTicket
{
    while (1) {
        // 创建对象
        // self.obj = [[NSObject alloc]init];
        // 锁对象，本身就是一个对象，所以self就可以了
        // 锁定的时候，其他线程没有办法访问这段代码
        @synchronized (self) {
            // 模拟售票时间，我们让线程休息0.05s
            [NSThread sleepForTimeInterval:0.05];
            if (self.numTicket > 0) {
                self.numTicket -= 1;
                NSLog(@"%@卖出了一张票，还剩下%d张票",[NSThread
currentThread].name,self.numTicket);
            }else{
                NSLog(@"票已经卖完了");
                break;
            }
        }
    }
}

@end
```





当没有加互斥锁的时候我们看一下输出

```
2016-08-24 23:56:02.616 线程安全[7797:1596749] 售票员03卖出了一张票，还剩下29张票
2016-08-24 23:56:02.616 线程安全[7797:1596747] 售票员01卖出了一张票，还剩下29张票
2016-08-24 23:56:02.616 线程安全[7797:1596748] 售票员02卖出了一张票，还剩下29张票
2016-08-24 23:56:02.671 线程安全[7797:1596749] 售票员03卖出了一张票，还剩下27张票
2016-08-24 23:56:02.671 线程安全[7797:1596747] 售票员01卖出了一张票，还剩下27张票
2016-08-24 23:56:02.671 线程安全[7797:1596748] 售票员02卖出了一张票，还剩下27张票
```

没有加互斥锁的输出

我们发现第29张，第27张都被销售了3次，这显然是不允许的，这就是数据错乱，那么当我们加上互斥锁时，其锁定的时候其他线程没有办法访问锁定的内容，等其访问完毕之后，其他线程才可以访问，我们爱来看一下输出

```
2016-08-25 00:01:23.871 线程安全[7817:1602237] 售票员02卖出了一张票，还剩下29张票
2016-08-25 00:01:23.926 线程安全[7817:1602238] 售票员03卖出了一张票，还剩下28张票
2016-08-25 00:01:23.979 线程安全[7817:1602236] 售票员01卖出了一张票，还剩下27张票
2016-08-25 00:01:24.033 线程安全[7817:1602237] 售票员02卖出了一张票，还剩下26张票
2016-08-25 00:01:24.086 线程安全[7817:1602238] 售票员03卖出了一张票，还剩下25张票
2016-08-25 00:01:24.141 线程安全[7817:1602236] 售票员01卖出了一张票，还剩下24张票
2016-08-25 00:01:24.194 线程安全[7817:1602237] 售票员02卖出了一张票，还剩下23张票
2016-08-25 00:01:24.247 线程安全[7817:1602238] 售票员03卖出了一张票，还剩下22张票
```

加上互斥锁的输出

此时就不会出现同一张票被多次出售的数据错乱的情况了。

2.5 NSThread线程之间的通信

什么叫做线程间通信


在**1**个进程中，线程往往不是孤立存在的，多个线程之间需要经常进行通信，例如我们在子线程完成下载图片后，回到主线程刷新**UI**显示图片

线程间通信的体现


1个线程传递数据给另**1**个线程

在**1**个线程中执行完特定任务后，转到另**1**个线程继续执行任务

线程间通信常用的方法



```
// 返回主线程
- (void)performSelectorOnMainThread:(SEL)aSelector withObject:(id)arg waitUntilDone:(BOOL)wait;
// 返回指定线程
- (void)performSelector:(SEL)aSelector onThread:(NSThread *)thr withObject:(id)arg waitUntilDone:(BOOL)wait;
```



下面我们通过一个实例看一下线程之间的通信





```
#import "ViewController.h"
@interface ViewController ()
@property (weak, nonatomic) IBOutlet UIImageView *imageView;
@end
@implementation ViewController
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    [NSThread detachNewThreadSelector:@selector(donwLoadImage) toTarget:self withObject:nil];
}
- (void)donwLoadImage
{
    // 获取图片url地址 http://www.itunes123.com/uploadfile/2016/0421/20160421014340186.jpg
    NSURL *url = [NSURL
URLWithString:@"http://www.itunes123.com/uploadfile/2016/0421/20160421014340186.jpg"];
    // 下载图片二进制文件
    NSData *data = [NSData dataWithContentsOfURL:url];
    // 将图片二进制文件转化为image;
    UIImage *image = [UIImage imageWithData:data];
    // 参数 waitUntilDone 是否等@selector(showImage:) 执行完毕以后再执行下面的操作 YES :等
NO:不等
    // 返回主线程显示图片
    // [self performSelectorOnMainThread:@selector(showImage:) withObject:image waitUntilDone:YES];
    // self.imageView 也可以直接调用这个方法 直接选择 setImage方法，传入参数image即可
    // [self.imageView performSelectorOnMainThread:@selector(setImage:) withObject:image waitUntilDone:YES];
    // 返回特定的线程, [NSThread mainThread] 获得主线程
    [self performSelector:@selector(showImage:) onThread:[NSThread mainThread]
withObject:image waitUntilDone:YES];
}
- (void)showImage:(UIImage *)image
{
    self.imageView.image = image;
}
@end
```





3. GCD的使用（重点）

GCD的全称是**Grand Central Dispatch**，是纯C语言，提供了非常多强大的函数

GCD的优势

GCD是苹果公司为多核的并行运算提出的解决方案

GCD会自动利用更多的CPU内核（比如双核、四核）

GCD会自动管理线程的生命周期（创建线程、调度任务、销毁线程）

程序员只需要告诉GCD想要执行什么任务，不需要编写任何线程管理代码

3.1 任务和队列

GCD中有2个核心概念：**任务和队列**

任务：执行什么操作，任务有两种执行方式：**同步函数** 和 **异步函数**，他们之间的区别是

同步：只能在当前线程中执行任务，不具备开启新线程的能力，任务立刻马上执行，会阻塞当前线程并等待 **Block**中的任务执行完毕，然后当前线程才会继续往下运行

异步：可以在新的线程中执行任务，具备开启新线程的能力，但不一定会开新线程，当前线程会直接往下执行，不会阻塞当前线程

队列：用来存放任务，分为**串行队列** 和 **并行队列**

串行队列（Serial Dispatch Queue）

让任务一个接着一个地执行（一个任务执行完毕后，再执行下一个任务）

并发队列（Concurrent Dispatch Queue）

可以让多个任务并发（同时）执行（自动开启多个线程同时执行任务）


并发功能只有在异步（dispatch_async）函数下才有效

GCD的使用就2个步骤


1. 定制任务
- 确定想做的事情
2. 将任务添加到队列中
- GCD会自动将队列中的任务取出，放到对应的线程中执行
- 任务的取出遵循队列的FIFO原则：先进先出，后进后出

3.2 GCD的创建


1. 队列的创建




```
// 第一个参数const char *label : C语言字符串，用来标识
// 第二个参数dispatch_queue_attr_t attr : 队列的类型
// 并发队列:DISPATCH_QUEUE_CONCURRENT
// 串行队列:DISPATCH_QUEUE_SERIAL 或者 NULL
dispatch_queue_t queue = dispatch_queue_create(const char *label, dispatch_queue_attr_t attr);
```




创建并发队列




```
dispatch_queue_t queue = dispatch_queue_create("com.xxcc", DISPATCH_QUEUE_CONCURRENT);
```




创建串行队列



```
dispatch_queue_t queue = dispatch_queue_create("com.xxcc", DISPATCH_QUEUE_SERIAL);
```



GCD默认已经提供了全局并发队列，供整个应用使用，可以无需手动创建





```
/**
    第一个参数:优先级 也可直接填后面的数字
```

```
#define DISPATCH_QUEUE_PRIORITY_HIGH 2 // 高
#define DISPATCH_QUEUE_PRIORITY_DEFAULT 0 // 默认
#define DISPATCH_QUEUE_PRIORITY_LOW (-2) // 低
#define DISPATCH_QUEUE_PRIORITY_BACKGROUND INT16_MIN // 后台
第二个参数：预留参数  0
*/

dispatch_queue_t ququel =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
```

获得主队列

```
dispatch_queue_t  queue = dispatch_get_main_queue();
```

- 2. 任务的执行
- 队列在**queue**中，任务在**block**块中
- 开启同步函数 同步函数：要求立刻马上开始执行

```
/*
第一个参数:队列
第二个参数:block,在里面封装任务
*/
dispatch_sync(queue, ^{

});
```

开启异步函数 异步函数：等主线程执行完毕之后，回过头开线程执行任务

```
dispatch_async(queue, ^{

});
```


- 3. 任务和队列的组合
- 任务：同步函数 异步函数
- 队列：串行 并行
- 异步函数+并发队列：会开启新的线程,并发执行

```
dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
dispatch_async(queue, ^{
    NSLog(@"---download1---%@", [NSThread currentThread]);
});
```


异步函数+串行队列：会开启一条线程,任务串行执行

```
dispatch_queue_t queue =  dispatch_queue_create("com.xxcc", DISPATCH_QUEUE_SERIAL);
dispatch_async(queue, ^{
    NSLog(@"---download1---%@", [NSThread currentThread]);
});
```

同步函数+并发队列：不会开线程,任务串行执行



```
dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
dispatch_sync(queue, ^{
    NSLog(@"---download1---%@", [NSThread currentThread]);
});
```



同步函数+串行队列：不会开线程,任务串行执行

```
dispatch_queue_t queue = dispatch_queue_create("com.xxcc",
DISPATCH_QUEUE_SERIAL);
dispatch_sync(queue, ^{
    NSLog(@"---download1---%@", [NSThread currentThread]);
});
```


异步函数+主队列:不会开线程,任务串行执行

使用主队列（跟主线程相关联的队列）


主队列是GCD自带的一种特殊的串行队列，放在主队列中的任务，都会放到主线程中执行




```
//1.获得主队列
dispatch_queue_t queue = dispatch_get_main_queue();
//2.异步函数
dispatch_async(queue, ^{
    NSLog(@"---download1---%@", [NSThread currentThread]);
});
```



同步函数+主队列:死锁



```
//1.获得主队列
dispatch_queue_t queue = dispatch_get_main_queue();
//2.同步函数
dispatch_sync(queue, ^{
    NSLog(@"---download1---%@", [NSThread currentThread]);
});
```



因为这个方法在主线程中，给主线程中添加任务，而同步函数要求立刻马上执行，因此就会相互等待而发生死锁。将这个方法放入子线程中，则不会发生死锁，任务串行执行。

总结：


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

注意
使用sync函数往当前串行队列中添加任务，会卡住当前的串行队列

任务队列组合总结

4. 同步函数和异步函数的执行顺序
- 同步函数：立刻马上执行，会阻塞当前线程





```
-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
```

```
{
    [self syncConcurrent];
}
//同步函数+并发队列:不会开线程,任务串行执行
-(void) syncConcurrent
{
    dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
    NSLog(@"--syncConcurrent--start-");
    dispatch_sync(queue, ^{
        NSLog(@"---download1---%@", [NSThread currentThread]);
    });
    dispatch_sync(queue, ^{
        NSLog(@"---download2---%@", [NSThread currentThread]);
    });
    dispatch_sync(queue, ^{
        NSLog(@"---download3---%@", [NSThread currentThread]);
    });
    dispatch_sync(queue, ^{
        NSLog(@"---download4---%@", [NSThread currentThread]);
    });
    NSLog(@"--syncConcurrent--end-");
}

```

我们看一下输出

```
--syncConCurrent--start-
---download1---<NSThread: 0x7f916d201620>{number = 1, name = main}
---download2---<NSThread: 0x7f916d201620>{number = 1, name = main}
---download3---<NSThread: 0x7f916d201620>{number = 1, name = main}
---download4---<NSThread: 0x7f916d201620>{number = 1, name = main}
--syncConCurrent--end-
```

同步函数会阻塞线程，立即执行

异步函数：当前线程会直接往下执行，不会阻塞当前线程

```

-(void) touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    [self syncConcurrent];
}

```

```

//异步函数+并发队列:会开启新的线程,并发执行

```

```

-(void) asyncCONCURRENT
{
    NSLog(@"--asyncCONCURRENT--start-");
    dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
    dispatch_async(queue, ^{
        NSLog(@"---download1---%@", [NSThread currentThread]);
    });
    dispatch_async(queue, ^{
        NSLog(@"---download2---%@", [NSThread currentThread]);
    });
    dispatch_async(queue, ^{
        NSLog(@"---download3---%@", [NSThread currentThread]);
    });
    dispatch_async(queue, ^{
        NSLog(@"---download4---%@", [NSThread currentThread]);
    });
    NSLog(@"--asyncCONCURRENT--end-");
}

```



我们来看一下输出

```
--asyncConCurrent--start-
--asyncConCurrent--end-
---download2---<NSThread: 0x7fed4b51e3f0>{number = 2, name = (null)}
---download3---<NSThread: 0x7fed4b460570>{number = 4, name = (null)}
---download4---<NSThread: 0x7fed4b600940>{number = 5, name = (null)}
---download1---<NSThread: 0x7fed4d2287f0>{number = 3, name = (null)}
```

异步函数不会阻塞当前线程

注意：GCD中开多少条线程是由系统根据CUP繁忙程度决定的，如果任务很多，GCD会开启适当的子线程，并不会让所有任务同时执行。

3.3 GCD线程间的通信

我们同样通过一个实例来看

```
#import "ViewController.h"
@interface ViewController ()
@property (weak, nonatomic) IBOutlet UIImageView *imageView;
@end
@implementation ViewController

-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
    dispatch_async(queue, ^{
        // 获得图片URL
        NSURL *url = [NSURL URLWithString:@"//upload-images.jianshu.io/upload_images/2301429-d5cc0a007447e469.jpg?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240"];
        // 将图片URL下载为二进制文件
        NSData *data = [NSData dataWithContentsOfURL:url];
        // 将二进制文件转化为image
        UIImage *image = [UIImage imageWithData:data];
        NSLog(@"%@", [NSThread currentThread]);
        // 返回主线程 这里用同步函数不会发生死锁，因为这个方法在子线程中被调用。
        // 也可以使用异步函数
        dispatch_async(dispatch_get_main_queue(), ^{
            self.imageView.image = image;
            NSLog(@"%@", [NSThread currentThread]);
        });
    });
}
@end
```



GCD线程间的通信非常简单，使用同步或异步函数，传入主队列即可。

3.4 GCD其他常用函数

1. 栅栏函数（控制任务的执行顺序）

```
dispatch_barrier_async(queue, ^{
    NSLog(@"--dispatch_barrier_async-");
});
```





我们来看一下栅栏函数的作用



```
-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    [self barrier];
}
-(void)barrier
{
    //1.创建队列(并发队列)
    dispatch_queue_t queue = dispatch_queue_create("com.xxccqueue",
DISPATCH_QUEUE_CONCURRENT);
    dispatch_async(queue, ^{
        for (NSInteger i = 0; i<3; i++) {
            NSLog(@"%zd-download1--%@",i,[NSThread currentThread]);
        }
    });
    dispatch_async(queue, ^{
        for (NSInteger i = 0; i<3; i++) {
            NSLog(@"%zd-download2--%@",i,[NSThread currentThread]);
        }
    });
    //栅栏函数
    dispatch_barrier_async(queue, ^{
        NSLog(@"我是一个栅栏函数");
    });
    dispatch_async(queue, ^{
        for (NSInteger i = 0; i<3; i++) {
            NSLog(@"%zd-download3--%@",i,[NSThread currentThread]);
        }
    });
    dispatch_async(queue, ^{
        for (NSInteger i = 0; i<3; i++) {
            NSLog(@"%zd-download4--%@",i,[NSThread currentThread]);
        }
    });
}
```



我们来看一下输出

```
0-download2--<NSThread: 0x7ff7cbd1e890>{number = 3, name = (null)}
1-download1--<NSThread: 0x7ff7ce100fc0>{number = 2, name = (null)}
1-download2--<NSThread: 0x7ff7cbd1e890>{number = 3, name = (null)}
2-download1--<NSThread: 0x7ff7ce100fc0>{number = 2, name = (null)}
2-download2--<NSThread: 0x7ff7cbd1e890>{number = 3, name = (null)}
我是一个栅栏函数
0-download3--<NSThread: 0x7ff7cbd1e890>{number = 3, name = (null)}
0-download4--<NSThread: 0x7ff7ce100fc0>{number = 2, name = (null)}
1-download3--<NSThread: 0x7ff7cbd1e890>{number = 3, name = (null)}
1-download4--<NSThread: 0x7ff7ce100fc0>{number = 2, name = (null)}
2-download3--<NSThread: 0x7ff7cbd1e890>{number = 3, name = (null)}
2-download4--<NSThread: 0x7ff7ce100fc0>{number = 2, name = (null)}
```

栅栏函数

栅栏函数可以控制任务执行的顺序，栅栏函数之前的执行完毕之后，执行栅栏函数，然后在执行栅栏函数之后的

2. 延迟执行（延迟·控制在哪个线程执行）





```
// 2s之后调用run方法
[self performSelector:@selector(run) withObject:nil afterDelay:2.0];
// repeats: YES 是否重复
[NSTimer scheduledTimerWithTimeInterval:2.0 target:self selector:@selector(run)
userInfo:nil repeats:YES];
```



4. 快速迭代（开多个线程并发完成迭代操作）



5. 队列组（同栅栏函数）





```
// 创建队列组
dispatch_group_t group = dispatch_group_create();
// 创建并行队列
dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
// 执行队列组任务
dispatch_group_async(group, queue, ^{
});
//队列组中的任务执行完毕之后，执行该函数
dispatch_group_notify(group, queue, ^{
});
```



下面看一了实例使用group下载两张图片然后合成在一起



```
#import "ViewController.h"
@interface ViewController ()
@property (weak, nonatomic) IBOutlet UIImageView *imageView;
@property (nonatomic, strong) UIImage *image1; /**< 图片1 */
@property (nonatomic, strong) UIImage *image2; /**< 图片2 */
@end
@implementation ViewController
-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    [self group];
}
-(void)group
{
    //下载图片1
    //创建队列组
    dispatch_group_t group = dispatch_group_create();
    //1.开子线程下载图片
    //创建队列(并发)
    dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
    dispatch_group_async(group, queue, ^{
        //1.获取url地址
        NSURL *url = [NSURL
URLWithString:@"http://www.huabian.com/uploadfile/2015/0914/20150914014032274.jpg"];
        //2.下载图片
        NSData *data = [NSData dataWithContentsOfURL:url];
        //3.把二进制数据转换成图片
        self.image1 = [UIImage imageWithData:data];
        NSLog(@"1---%@", self.image1);
    });
    //下载图片2
    dispatch_group_async(group, queue, ^{
        //1.获取url地址
        NSURL *url = [NSURL
URLWithString:@"http://img1.3lian.com/img2011/w12/1202/19/d/88.jpg"];
        //2.下载图片
        NSData *data = [NSData dataWithContentsOfURL:url];
        //3.把二进制数据转换成图片
        self.image2 = [UIImage imageWithData:data];
        NSLog(@"2---%@", self.image2);
    });
    //合成，队列组执行完毕之后执行
    dispatch_group_notify(group, queue, ^{
        //开启图形上下文
        UIGraphicsBeginImageContext(CGSizeMake(200, 200));
        //画1
        [self.image1 drawInRect:CGRectMake(0, 0, 200, 100)];
        //画2
        [self.image2 drawInRect:CGRectMake(0, 100, 200, 100)];
        //根据图形上下文拿到图片
        UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
        //关闭上下文
        UIGraphicsEndImageContext();
```



```
//回到主线程刷新UI
dispatch_async(dispatch_get_main_queue(), ^{
    self.imageView.image = image;
    NSLog(@"%@--刷新UI",[NSThread currentThread]);
});
});
}
```

4. NSOperation的使用（重点）

NSOperation 是苹果公司对 GCD 的封装，完全面向对象，并比GCD多了一些更简单实用的功能，所以使用起来更加方便易于理解。NSOperation 和NSOperationQueue 分别对应 GCD 的 任务 和 队列。

NSOperation和NSOperationQueue实现多线程的具体步骤

1.将需要执行的操作封装到一个NSOperation对象中

2.将NSOperation对象添加到NSOperationQueue中

系统会自动将NSOperationQueue中的NSOperation取出来，并将取出的NSOperation封装的操作放到一条新线程中执行

4.1 NSOperation的创建

NSOperation是个抽象类，并不具备封装操作的能力，必须使用它的子类
使用NSOperation子类的方式有3种

1. NSInvocationOperation


```
/*
    第一个参数:目标对象
    第二个参数:选择器,要调用的方法
    第三个参数:方法要传递的参数
*/
NSInvocationOperation *op = [[NSInvocationOperation alloc]initWithTarget:self
selector:@selector(download) object:nil];
//启动操作
[op start];
```

2. NSBlockOperation（最常用）

```
//1.封装操作
NSBlockOperation *op = [NSBlockOperation blockOperationWithBlock:^(
    //要执行的操作，在主线程中执行
    NSLog(@"1-----%@",[NSThread currentThread]);
)];
//2.追加操作，追加的操作在子线程中执行，可以追加多条操作
[op addExecutionBlock:^(
    NSLog(@"---download2--%@",[NSThread currentThread]);
)];
[op start];
```

3. 自定义子类继承NSOperation，实现内部相应的方法

```
// 重写自定义类的主方法实现封装操作
```





```
- (void)main
{
    // 要执行的操作
}
// 实例化一个自定义对象，并执行操作
CLOperation *op = [[CLOperation alloc] init];
[op start];
```





自定义类封装性高，复用性高。

4.2 NSOperationQueue的使用

NSOperation中的两种队列

主队列：通过mainQueue获得，凡是放到主队列中的任务都将在主线程执行


非主队列：直接alloc init出来的队列。非主队列同时具备了并发和串行的功能，通过设置最大并发数属性来控制任务是并发执行还是串行执行

NSOperationQueue的作用


NSOperation可以调用start方法来执行任务，但默认是同步执行的

如果将NSOperation添加到NSOperationQueue（操作队列）中，系统会自动异步执行NSOperation中的操作

添加操作到NSOperationQueue中



```
- (void)addOperation:(NSOperation *)op;
- (void)addOperationWithBlock:(void (^)(void))block;
```



注意：将操作添加到**NSOperationQueue**中，就会自动启动，不需要再自己启动了**addOperation**内部调用 **start**方法
start方法 内部调用 **main**方法

4.3 NSOperation和NSOperationQueue结合使用创建多线程





```
注：这里使用NSBlockOperation示例，其他两种方法一样
// 1. 创建非主队列 同时具备并发和串行的功能，默认是并发队列
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
//NSBlockOperation 不论封装操作还是追加操作都是异步并发执行
// 2. 封装操作
NSBlockOperation *op1 = [NSBlockOperation blockOperationWithBlock:^(
    NSLog(@"download1 -- %@", [NSThread currentThread]);
}];
// 3. 将封装操作加入主队列
// 也可以不获取封装操作对象 直接添加操作到队列中
//[queue addOperationWithBlock:^(
// 操作
//});
[queue addOperation:op1];
```





4.4 NSOperation和NSOperationQueue的重要属性和方法

NSOperation

1. NSOperation的依赖 – (void)addDependency:(NSOperation *)op;



```
// 操作op1依赖op5，即op1必须等op5执行完毕之后才会执行
// 添加操作依赖,注意不能循环依赖，如果循环依赖会造成两个任务都不会执行
// 也可以夸队列依赖，依赖别的队列的操作

[op1 addDependency:op5];
```



2. NSOperation操作监听void (^completionBlock)(void)



```
// 监听操作的完成
// 当op1线程完成之后，立刻就会执行block块中的代码
// block中的代码与op1不一定在一个线程中执行，但是一定在子线程中执行
op1.completionBlock = ^{
    NSLog(@"op1已经完成了---%@", [NSThread currentThread]);
};
```



NSOperationQueue

1. maxConcurrentOperationCount



```
//1.创建队列
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
/*
    默认是并发队列,如果最大并发数>1,并发
    如果最大并发数==1,串行队列
    系统的默认是最大并发数-1 ,表示不限制
    设置成0则不会执行任何操作
*/
queue.maxConcurrentOperationCount = 1;
```



2. suspended



```
//当值为YES的时候暂停,为NO的时候是恢复
queue.suspended = YES;
```



3. -(void)cancelAllOperations;



```
//取消所有的任务，不再执行，不可逆
[queue cancelAllOperations];
```



注意：暂停和取消只能暂停或取消处于等待状态的任务，不能暂停或取消正在执行中的任务，必须等正在执行的任务执行完毕之后才会暂停，如果想要暂停或者取消正在执行的任务，可以在每个任务之间即每当执行完一段耗时操作之后，判断是否任务是否被取消或者暂停。如果想要精确的控制，则需要将判断代码放在任务之中，但是不建议这么做，频繁的判断会消耗太多时间

4.5 NSOperation和NSOperationQueue的一些其他属性和方法

NSOperation



```
// 开启线程
```



```
- (void) start;
- (void) main;
// 判断线程是否被取消
@property (readonly, getter=isCancelled) BOOL cancelled;
// 取消当前线程
- (void) cancel;
//NSOperation任务是否在运行
@property (readonly, getter=isExecuting) BOOL executing;
//NSOperation任务是否已结束
@property (readonly, getter=isFinished) BOOL finished;
// 添加依赖
- (void) addDependency:(NSOperation *)op;
// 移除依赖
- (void) removeDependency:(NSOperation *)op;
// 优先级
typedef NS_ENUM(NSInteger, NSOperationQueuePriority) {
    NSOperationQueuePriorityVeryLow = -8L,
    NSOperationQueuePriorityLow = -4L,
    NSOperationQueuePriorityNormal = 0,
    NSOperationQueuePriorityHigh = 4,
    NSOperationQueuePriorityVeryHigh = 8
};
// 操作监听
@property (nullable, copy) void (^completionBlock)(void) NS_AVAILABLE(10_6, 4_0);
// 阻塞当前线程，直到该NSOperation结束。可用于线程执行顺序的同步
- (void) waitUntilFinished NS_AVAILABLE(10_6, 4_0);
// 获取线程的优先级
@property double threadPriority NS_DEPRECATED(10_6, 10_10, 4_0, 8_0);
// 线程名称
@property (nullable, copy) NSString *name NS_AVAILABLE(10_10, 8_0);
@end
```



NSOperationQueue



```
// 获取队列中的操作
@property (readonly, copy) NSArray<__kindof NSOperation *> *operations;
// 队列中的操作数
@property (readonly) NSUInteger operationCount NS_AVAILABLE(10_6, 4_0);
// 最大并发数，同一时间最多只能执行三个操作
@property NSInteger maxConcurrentOperationCount;
// 暂停 YES:暂停 NO:继续
@property (getter=isSuspended) BOOL suspended;
// 取消所有操作
- (void) cancelAllOperations;
// 阻塞当前线程直到此队列中的所有任务执行完毕
- (void) waitUntilAllOperationsAreFinished;
```



4.6 NSOperation线程之间的通信

NSOperation线程之间的通信方法



```
// 回到主线程刷新UI
[[NSOperationQueue mainQueue] addOperationWithBlock:^(
    self.imageView.image = image;
}];
```



我们同样使用下载多张图片合成综合案例





```
#import "ViewController.h"
@interface ViewController ()
@property (weak, nonatomic) IBOutlet UIImageView *imageView;
@property (nonatomic, strong) UIImage *image1;
@property (nonatomic, strong) UIImage *image2;
@end
@implementation ViewController

- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    // 创建非住队列
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
    // 下载第一张图片
    NSBlockOperation *download1 = [NSBlockOperation blockOperationWithBlock:^(
        NSURL *url = [NSURL URLWithString:@"http://img2.3lian.com/2014/c7/12/d/77.jpg"];
        NSData *data = [NSData dataWithContentsOfURL:url];
        self.image1 = [UIImage imageWithData:data];
    )];
    // 下载第二张图片
    NSBlockOperation *download2 = [NSBlockOperation blockOperationWithBlock:^(
        NSURL *url = [NSURL URLWithString:@"http://img2.3lian.com/2014/c7/12/d/77.jpg"];
        NSData *data = [NSData dataWithContentsOfURL:url];
        self.image2 = [UIImage imageWithData:data];
    )];
    // 合成操作
    NSBlockOperation *combie = [NSBlockOperation blockOperationWithBlock:^(
        // 开启图形上下文
        UIGraphicsBeginImageContext(CGSizeMake(375, 667));
        // 绘制图片1
        [self.image1 drawInRect:CGRectMake(0, 0, 375, 333)];
        // 绘制图片2
        [self.image2 drawInRect:CGRectMake(0, 334, 375, 333)];
        // 获取合成图片
        UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
        // 关闭图形上下文
        UIGraphicsEndImageContext();
        // 回到主线程刷新UI
        [[NSOperationQueue mainQueue] addOperationWithBlock:^(
            self.imageView.image = image;
        )];
    )];
    // 添加依赖，合成图片需要等图片1，图片2都下载完毕之后合成
    [combie addDependency:download1];
    [combie addDependency:download2];
    // 添加操作到队列
    [queue addOperation:download1];
    [queue addOperation:download2];
    [queue addOperation:combie];
}
@end
```



注意：子线程执行完操作之后就会立即释放，即使我们使用强引用引用子线程使子线程不被释放，也不能给子线程再次添加操作，或者再次开启。

声明：本文非原创，仅仅整理一些开发技能知识文章，以作存档学习用
参考

<http://www.jianshu.com/p/6e6f4e005a0b>

<http://www.jianshu.com/p/f28a50f72bb1>

<http://www.jianshu.com/p/6e74f5438f2c>

少而好学，如日出之阳； 壮而好学，如日中之光； 老而好学，如炳烛之明。

最新新闻：

- 算法根据步态识别情绪
- 重新思考引力