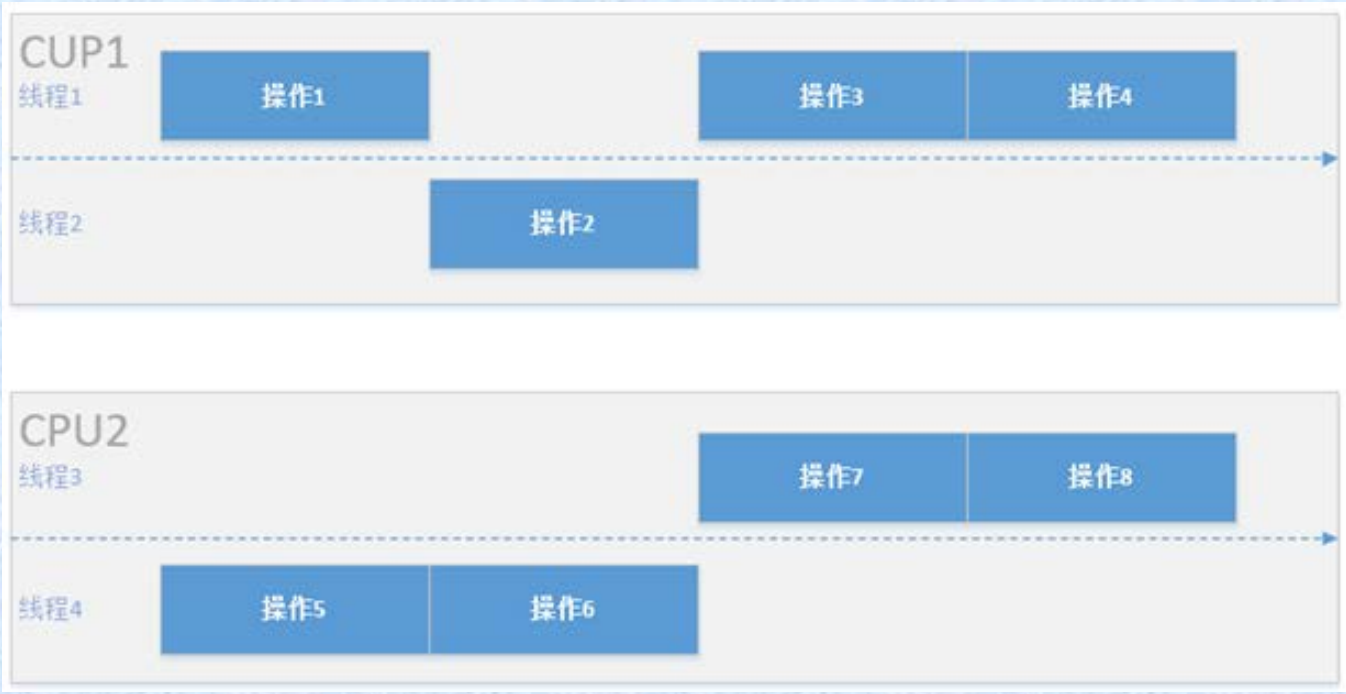
多线程

一、IOS中进程与线程

- 什么是进程？什么是线程？
- 进程与线程的区别？


CUP1
线程1
操作1
操作3
操作4
线程2
操作2
CPU2
线程3
操作7
操作8
线程4
操作5
操作6

二、IOS中的多线程方案

IOS有三种多线程解决方案：

- NSThread
- Cocoa NSOperation
- GCD（Grand Central Dispatch）

推荐使用GCD实现多线程。

二、GCD多线程开发

什么是GCD?

Grand Central Dispatch (GCD) comprises language features, runtime libraries, and system enhancements that provide systemic, comprehensive improvements to the support for concurrent code execution on **multicore hardware** in iOS and OS X.

The BSD subsystem, Core Foundation, and Cocoa APIs have all been extended to use these enhancements to help both the system and your application to run faster, more efficiently, and with improved responsiveness. Consider how difficult it is for a single application to use multiple cores effectively, let alone doing it on different computers with different numbers of computing cores or in an environment with multiple applications competing for those cores. GCD, operating at the **system level**, can better accommodate the needs of all running applications, matching them to the available system resources in a balanced fashion.

为什么推荐GCD?

- GCD是基于多核并行计算的解决方案，它可以自动利用更多的CPU内核。
- 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）,只需告诉GCD执行什么任务即可，不需要编写线程管理代码。
- 相较于其它两种方案，GCD更加简单高效，具有更好并发性能。

Grand Central Dispatch是低层API，提供了一种简单高效的方法来实现多线程的开发。GCD允许程序将任务切分为多个单一任务(BLOCK)然后提交至任务队列来并发地或者串行地执行。

关于GCD需要注意的几点：

● GCD不属于Cocoa框架，它是一组实现并发编程的C接口，因此我们在编写GCD相关代码的时面对的是函数，而不是对象中的方法。

● GCD的API很大程度上基于BLOCK，GCD可以脱离block来使用(例如：使用c中供函数指针）。但更多它是配合BLOCK来使用来发挥其最大能力。

三、任务和队列

什么是任务?

可以看作需要执行的一段代码（用来实现某一任务）

什么是队列?

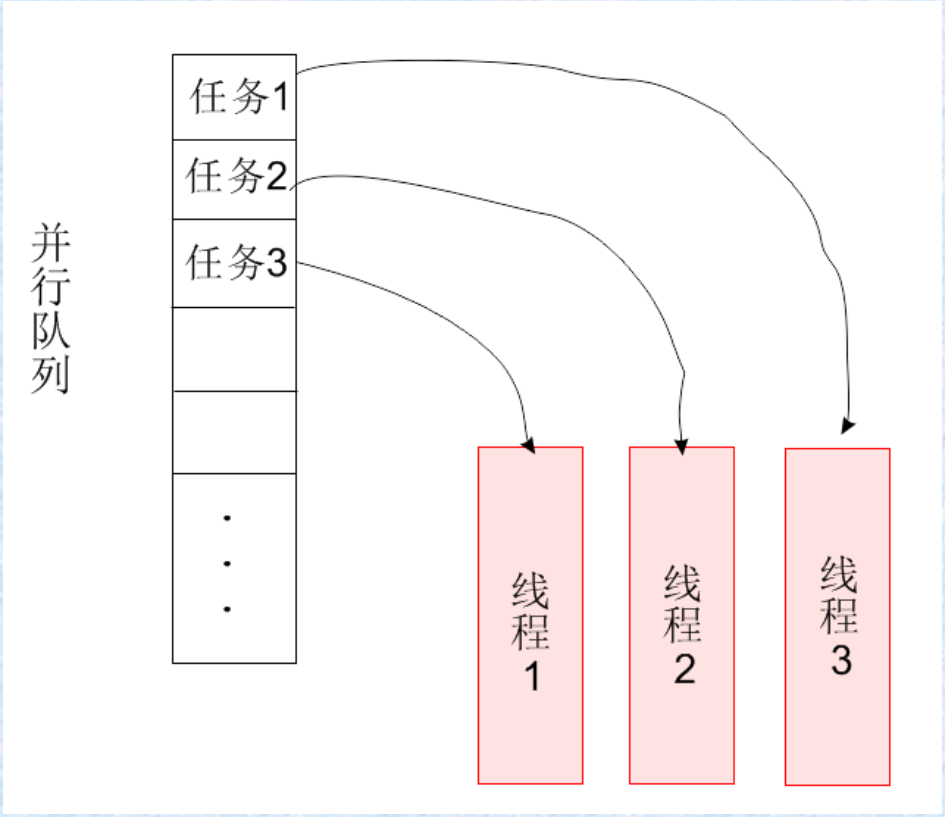
线性表，先进先出。

GCD如何工作？

只需将任务添加到队列中，GCD会自动将队列中的任务取出，放到对应线程中执行。

- 并行队列（Concurrent Dispatch Queue）

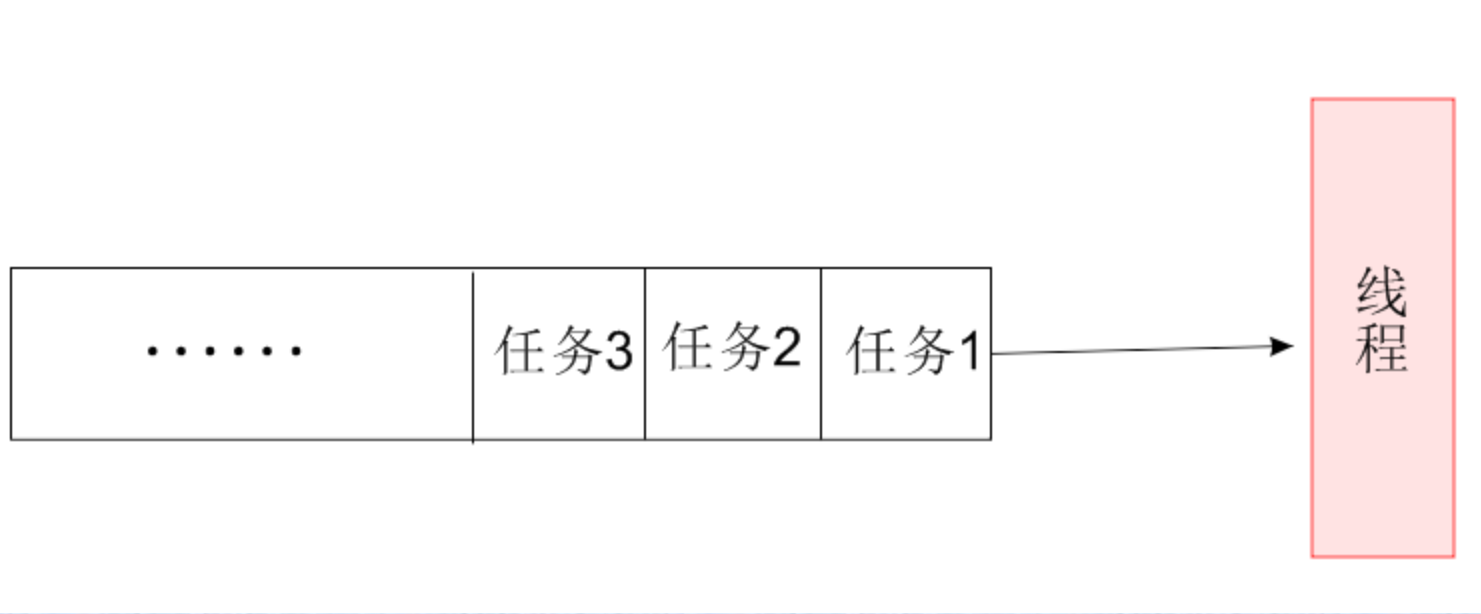
可以让多个任务并发执行（自动开启多个线程同时执行任务）。



注意：并发功能只有在异步（dispatch_async）函数下才有效

- 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）。



如何获得队列？

- 使用dispatch_queue_create函数创建串行队列

dispatch_queue_create

Creates a new dispatch queue to which blocks can be submitted.

Declaration

```
OBJECTIVE-C
dispatch_queue_t dispatch_queue_create ( const char *label, dispatch_queue_attr_t attr );
```

Parameters

label	A string label to attach to the queue to uniquely identify it in debugging tools such as Instruments, `sample`, stackshots, and crash reports. Because applications, libraries, and frameworks can all create their own dispatch queues, a reverse-DNS naming style (*com.example.myqueue*) is recommended. This parameter is optional and can be `NULL`.
attr	In OS X v10.7 and later or iOS 4.3 and later, specify `DISPATCH_QUEUE_SERIAL` (or `NULL`) to create a serial queue or specify `DISPATCH_QUEUE_CONCURRENT` to create a concurrent queue. In earlier versions, you must specify `NULL` for this parameter.

队列名称

Return Value

The newly created dispatch queue.

- 使用dispatch_get_main_queue()函数获得主队列

dispatch_get_main_queue

Returns the serial dispatch queue associated with the application's main thread.

Declaration

```
OBJECTIVE-C
dispatch_queue_t dispatch_get_main_queue ( void );
```

Return Value

Returns the main queue. This queue is created automatically on behalf of the main thread before `main` is called.

注意： 涉及到UI的代码必须放在主队列中。

- 使用dispatch_get_global_queue函数获得全局并发队列。

dispatch_get_global_queue

Returns a system-defined global concurrent queue with the specified quality of service class.

Declaration

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

Parameters

优先级

identifier	The quality of service you want to give to tasks executed using this queue. Quality-of-service helps determine the priority given to tasks executed by the queue. You may specify the values QOS_CLASS_USER_INTERACTIVE, QOS_CLASS_USER_INITIATED, QOS_CLASS_UTILITY, or QOS_CLASS_BACKGROUND. Queues that handle user-interactive or user-initiated tasks have a higher priority than tasks meant to run in the background. You may also specify one of the dispatch queue priority values, which are found in dispatch_queue_priority_t. These values map to an appropriate quality-of-service class.
flags	Flags that are reserved for future use. Always specify 0 for this parameter.

Return Value

The requested global concurrent queue.

注意：GCD默认已经提供了全局的并发队列，供整个应用使用，不需要手动创建

优先级设置

```
#define DISPATCH_QUEUE_PRIORITY_HIGH 2 // 高
#define DISPATCH_QUEUE_PRIORITY_DEFAULT 0 // 默认（中）
#define DISPATCH_QUEUE_PRIORITY_LOW (-2) // 低
#define DISPATCH_QUEUE_PRIORITY_BACKGROUND INT16_MIN // 后台
```

四、同步和异步执行任务

同步和异步的区别是：同步是指在当前线程中执行。异步是指在不同的线程中执行。

1、用同步的方式执行任务。

dispatch_sync

Submits a block object for execution on a dispatch queue and waits until that block completes.

Declaration

```
OBJECTIVE-C
void dispatch_sync ( dispatch_queue_t queue, dispatch_block_t block );
```

Parameters

queue	The queue on which to submit the block. This parameter cannot be NULL.
block	The block to be invoked on the target dispatch queue. This parameter cannot be NULL.

四、同步和异步执行任务

2、用异步的方式执行任务。

dispatch_async

Submits a block for asynchronous execution on a dispatch queue and returns immediately.

Declaration

```
OBJECTIVE-C
void dispatch_async ( dispatch_queue_t queue, dispatch_block_t block );
```

Parameters

queue	The queue on which to submit the block. The queue is retained by the system until the block has run to completion. This parameter cannot be NULL.
block	The block to submit to the target dispatch queue. This function performs Block_copy and Block_release on behalf of callers. This parameter cannot be NULL.

四、同步和异步执行任务

3、在主队列中执行任务。

dispatch_main

Executes blocks submitted to the main queue.

Declaration

```
OBJECTIVE-C
void dispatch_main ( void );
```

Return Value

This function never returns.

同步、异步、并行和串行

	全局并发队列	手动创建串行队列	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

代码示例

异步

并行队列

```
dispatch_queue_t queue =  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_async(queue, ^{
     NSLog(@"异步线程1----%@",[NSThread currentThread]);
   });
dispatch_async(queue, ^{
     NSLog(@"异步线程2----%@",[NSThread currentThread]);
   });
dispatch_async(queue, ^{
        NSLog(@"异步线程3---%@",[NSThread currentThread]);
    });
NSLog(@"主线程----%@",[NSThread mainThread]);
```

```
主线程----<NSThread: 0x7f94e0d20240>{number = 1, name = main}
异步线程2----<NSThread: 0x7f94e0d20a70>{number = 4, name = (null)}
异步线程1----<NSThread: 0x7f94e0e127e0>{number = 2, name = (null)}
异步线程3---<NSThread: 0x7f94e0c0ef50>{number = 3, name = (null)}
```

同步　　并行队列

```
dispatch_queue_t queue =  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_sync(queue, ^{
     NSLog(@"同步线程1----%@",[NSThread currentThread]);
   });
dispatch_sync(queue, ^{
     NSLog(@"同步线程2----%@",[NSThread currentThread]);
   });
dispatch_sync(queue, ^{
        NSLog(@"同步线程3---%@",[NSThread currentThread]);
    });
NSLog(@"主线程----%@",[NSThread mainThread]);
```

注意执行次序

```
同步线程1----<NSThread: 0x7f9213d10a30>{number = 1, name = main}
同步线程2----<NSThread: 0x7f9213d10a30>{number = 1, name = main}
同步线程3---<NSThread: 0x7f9213d10a30>{number = 1, name = main}
主线程----<NSThread: 0x7f9213d10a30>{number = 1, name = main}
```

异步　　串行队列

```
dispatch_queue_t queue =  dispatch_queue_create("NewQueue", NULL);
dispatch_async(queue, ^{
     NSLog(@"异步线程1----%@",[NSThread currentThread]);
   });
dispatch_async(queue, ^{
     NSLog(@"异步线程2----%@",[NSThread currentThread]);
   });
dispatch_async(queue, ^{
        NSLog(@"异步线程3---%@",[NSThread currentThread]);
    });
NSLog(@"主线程----%@",[NSThread mainThread]);
```

注意执行次序

```
主线程----<NSThread: 0x7f9b08e10c10>{number = 1, name = main}
异步线程1----<NSThread: 0x7f9b08d15670>{number = 2, name = (null)}
异步线程2----<NSThread: 0x7f9b08d15670>{number = 2, name = (null)}
异步线程3---<NSThread: 0x7f9b08d15670>{number = 2, name = (null)}
```

同步　　串行队列

```
dispatch_queue_t queue =  dispatch_queue_create("NewQueue", NULL);
dispatch_sync(queue, ^{
     NSLog(@"同步线程1----%@",[NSThread currentThread]);
   });
dispatch_sync(queue, ^{
     NSLog(@"同步线程2----%@",[NSThread currentThread]);
   });
dispatch_sync(queue, ^{
        NSLog(@"同步线程3---%@",[NSThread currentThread]);
    });
NSLog(@"主线程----%@",[NSThread mainThread]);
```

注意执行次序

```
同步线程1----<NSThread: 0x7fc630c13e00>{number = 1, name = main}
同步线程2----<NSThread: 0x7fc630c13e00>{number = 1, name = main}
同步线程3---<NSThread: 0x7fc630c13e00>{number = 1, name = main}
主线程----<NSThread: 0x7fc630c13e00>{number = 1, name = main}
```

异步　　主队列

```
dispatch_queue_t queue =  dispatch_get_main_queue();
dispatch_async(queue, ^{
     NSLog(@"异步主线程1----%@",[NSThread currentThread]);
   });
dispatch_async(queue, ^{
     NSLog(@"异步主线程2----%@",[NSThread currentThread]);
   });
dispatch_async(queue, ^{
        NSLog(@"异步主线程3---%@",[NSThread currentThread]);
    });
NSLog(@"主线程----%@",[NSThread mainThread]);
```

注意执行次序

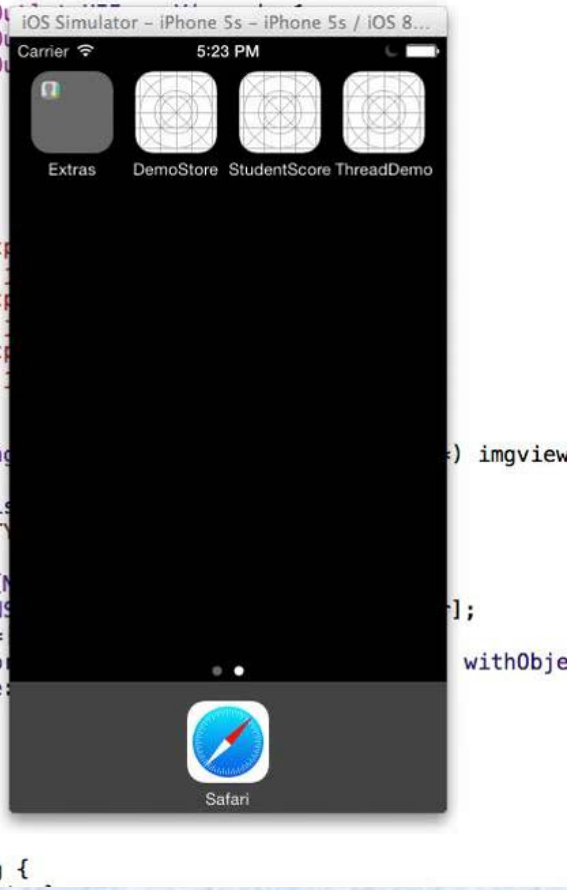
```
主线程----<NSThread: 0x7fc5eac1f0f0>{number = 1, name = main}
异步主线程1----<NSThread: 0x7fc5eac1f0f0>{number = 1, name = main}
异步主线程2----<NSThread: 0x7fc5eac1f0f0>{number = 1, name = main}
异步主线程3---<NSThread: 0x7fc5eac1f0f0>{number = 1, name = main}
```

同步　　主队列

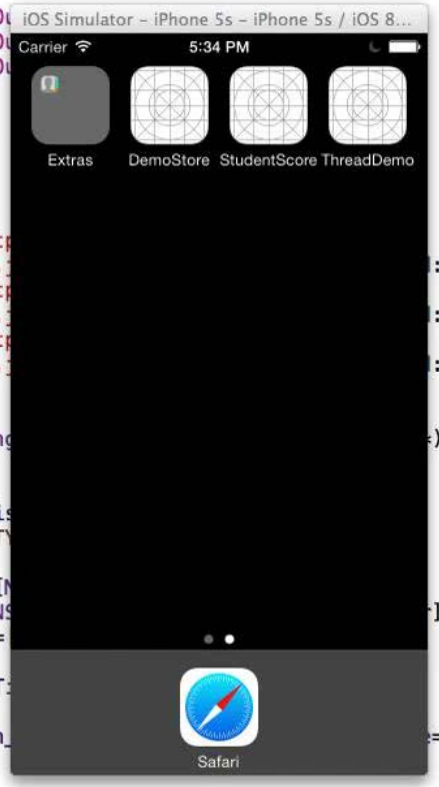
```
dispatch_queue_t queue =  dispatch_get_main_queue();
dispatch_sync(queue, ^{
     NSLog(@"同步主线程1----%@",[NSThread currentThread]);
   });
dispatch_sync(queue, ^{
     NSLog(@"同步主线程2----%@",[NSThread currentThread]);
   });
dispatch_sync(queue, ^{
        NSLog(@"同步主线程3---%@",[NSThread currentThread]);
    });
NSLog(@"主线程----%@",[NSThread mainThread]);
```

注意：使用同步函数，在主线程中执行主队列中得任务，会发生死循环，任务无法往下执行。

```objc
@interface ViewController ()
@property (weak, nonatomic) IBOutlet UIImageView *img1;
@property (weak, nonatomic) IBOutlet UIImageView *img2;
@property (weak, nonatomic) IBOutlet UIImageView *img3;
@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    [self imageLoadInPath:@"http://vi.ustc.edu.cn/view/201107/
        W020110714443895379885.jpg" ImageView:self.img1];
    [self imageLoadInPath:@"http://vi.ustc.edu.cn/view/201107/
        W020110714437503402403.jpg" ImageView:self.img2];
    [self imageLoadInPath:@"http://vi.ustc.edu.cn/view/201107/
        W020110714443895379885.jpg" ImageView:self.img3];
}

-(void)imageLoadInPath:(NSString *) path ImageView:(UIImageView *) imgview
{
    dispatch_queue_t queue = dispatch_get_global_queue
        (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_async(queue, ^{
                NSURL *urlstr=[NSURL URLWithString:path];
                NSData *data=[NSData dataWithContentsOfURL:urlstr];
                UIImage *image=[UIImage imageWithData:data];
        [imgview performSelectorOnMainThread:@selector(setImage:) withObject:
            image waitUntilDone:NO];

    });

}
```



```objc
@interface ViewController ()
@property (weak, nonatomic) IBOutlet UIImageView *img1;
@property (weak, nonatomic) IBOutlet UIImageView *img2;
@property (weak, nonatomic) IBOutlet UIImageView *img3;
@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    [self imageLoadInPath:@"http://vi.ustc.edu.cn/view/201107/
        W020110714443895379885.jpg" ImageView:self.img1 isSleeped:NO];
    [self imageLoadInPath:@"http://vi.ustc.edu.cn/view/201107/
        W020110714437503402403.jpg" ImageView:self.img2 isSleeped:YES];
    [self imageLoadInPath:@"http://vi.ustc.edu.cn/view/201107/
        W020110714443895379885.jpg" ImageView:self.img3 isSleeped:NO];
}

-(void)imageLoadInPath:(NSString *) path ImageView:(UIImageView *) imgview
    isSleeped:(BOOL) tag
{
    dispatch_queue_t queue = dispatch_get_global_queue
        (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_async(queue, ^{
                NSURL *urlstr=[NSURL URLWithString:path];
                NSData *data=[NSData dataWithContentsOfURL:urlstr];
                UIImage *image=[UIImage imageWithData:data];
        if (tag) {
            [NSThread sleepForTimeInterval:3.0];
        }
        dispatch_async(dispatch_get_main_queue(), ^{imgview.image=image;});

    });

}
```



五、线程同步

多线程操作过程中往往多个线程是并发执行的，同一个资源可能被多个线程同时访问，造成资源抢夺，所以多线程应用中必须引入锁机制。

三种实现锁的方法：

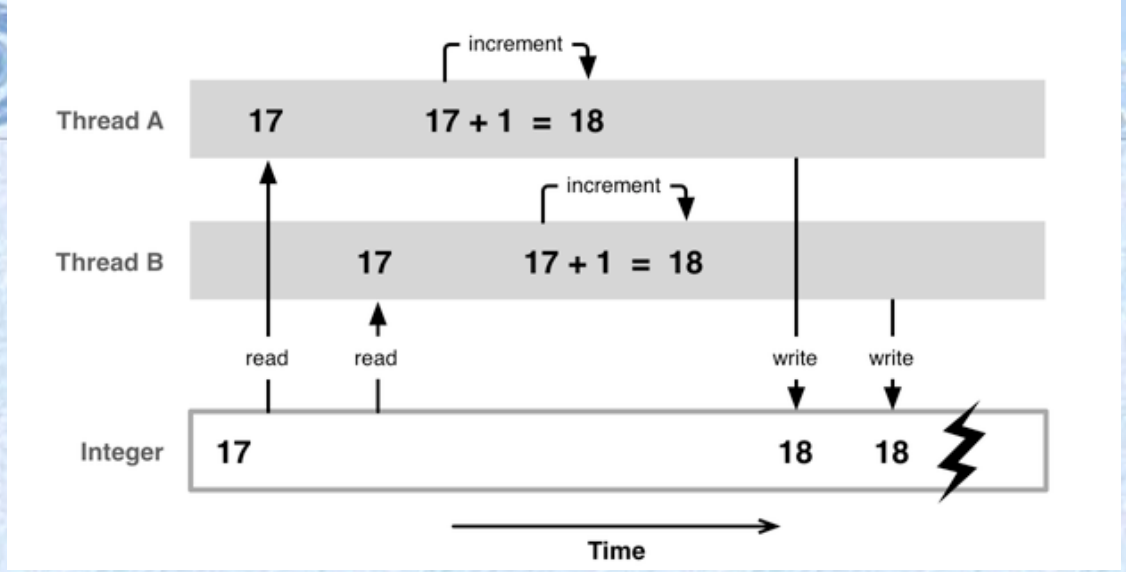
- NSLock同步锁。
- 使用@synchronized代码块。
- 控制线程通信

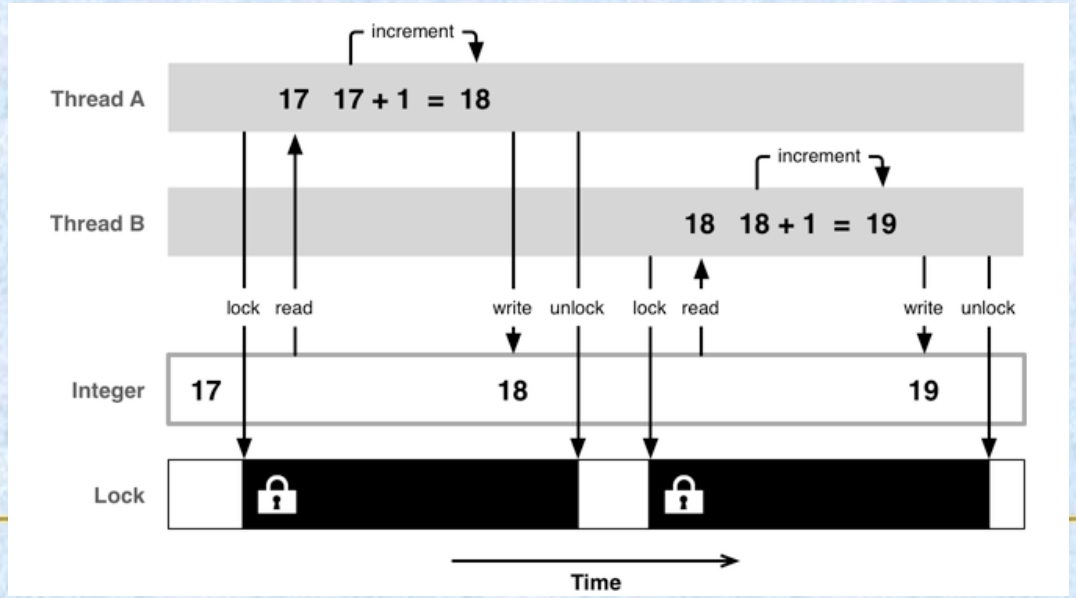
```
@interface ViewController : UIViewController
{
    __block int result;
 }
@end
```

```
result=1;
dispatch_queue_t queue =dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_async(queue, ^{
    NSLog(@"线程1 加1---- %d", [self addResult]); });
dispatch_async(queue, ^{
   NSLog(@"线程2 加1---- %d",[self addResult]);
        });
```

```
-(int)addResult
{
    result++;
    [NSThread sleepForTimeInterval:0.02];
    return result;

}
```

```
] 线程2 加1---- 3
] 线程1 加1---- 3
```


increment
Thread A
17
17 + 1 = 18
increment
Thread B
17
17 + 1 = 18
read
read
write
write
Integer
17
18
18
Time


increment
Thread A
17
17 + 1 = 18
increment
Thread B
18
18 + 1 = 19
lock
read
write
unlock
lock
read
write
unlock
Integer
17
18
19
Lock
Time

1、 NSLock同步锁。

IOS中可使用NSLock来解决资源抢占问题，使用时把需要加锁的代码放到NSLock的lock和unlock之间，一个线程A进入加锁代码之后由于已经加锁，另一个线程B就无法访问，只有等待前一个线程A执行完加锁代码后解锁，B线程才能访问加锁代码。

注意：lock和unlock之间的仅仅应该放对*“抢占资源的读取和修改”*代码，不要将其他操作代码放入其中，否则一个线程执行的时候另一个线程就一直在等待，就无法发挥多线程的作用了。

1、 NSLock同步锁。

```
@interface ViewController : UIViewController
{
    __block int result;
    NSLock *lock;
}
@end
```

```
result=1;
lock=[[NSLock alloc] init];
dispatch_queue_t queue =dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_async(queue, ^{
    NSLog(@"线程1 加1---- %d", [self addResult]); });
dispatch_async(queue, ^{
   NSLog(@"线程2 加1---- %d",[self addResult]);
        });
```

```
-(int)addResult
{
    [lock lock];
    result++;
    [NSThread sleepForTimeInterval:0.02];
    [lock unlock];
    return result;

}
```

```
线程1 加1---- 2
线程2 加1---- 3
```

2、使用@synchronized 。

使用@synchronized解决线程同步问题相比较NSLock要简单一些（推荐使用此方法）。首先选择一个对象作为同步对象（一般使用self），然后将”加锁代码”（争夺资源的读取、修改代码）放到代码块中。

@synchronized中的代码执行时先检查同步对象是否被另一个线程占用，如果占用该线程就会处于等待状态，直到同步对象被释放。

2、使用@synchronized 。

```
@interface ViewController : UIViewController
{
    __block int result;
 }
@end
```

```
result=1;
dispatch_queue_t queue =dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_async(queue, ^{
    NSLog(@"线程1 加1---- %d", [self addResult]); });
dispatch_async(queue, ^{
   NSLog(@"线程2 加1---- %d",[self addResult]);
        });
```

```
-(int)addResult
{
    @synchronized(self)
    {
    result++;
    [NSThread sleepForTimeInterval:0.02];
    return result;
    }}
```

```
线程2 加1---- 2
线程1 加1---- 3
```

3、控制线程通信。

在GCD中提供了一种信号机制，也可以解决资源抢占问题。GCD中信号量是dispatch_semaphore_t类型，支持信号通知和信号等待。每当发送一个信号通知，则信号量+1；每当发送一个等待信号时信号量-1,；如果信号量为0则信号会处于等待状态，直到信号量大于0开始执行。根据这个原理我们可以初始化一个信号量变量，默认信号量设置为1，每当有线程进入“加锁代码”之后就调用信号等待命令（此时信号量为0）开始等待，此时其他线程无法进入，执行完后发送信号通知（此时信号量为1），其他线程开始进入执行，如此一来就达到了线程同步目的。

```
@interface ViewController : UIViewController
{
    __block int result;
    dispatch_semaphore_t _semaphore;
}
@end
```

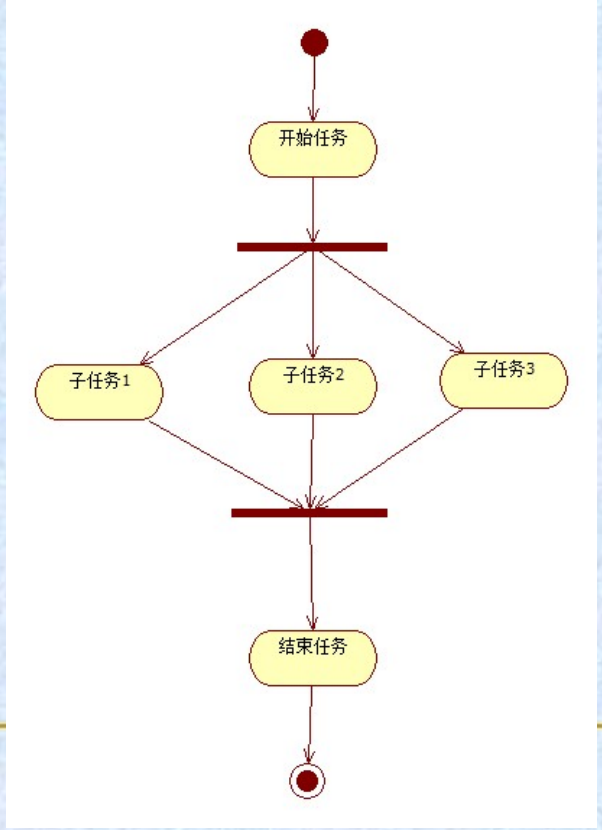
```
result=1;
_semaphore=dispatch_semaphore_create(1);
dispatch_queue_t queue =dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_async(queue, ^{
    NSLog(@"线程1 加1---- %d", [self addResult]); });
dispatch_async(queue, ^{
   NSLog(@"线程2 加1---- %d",[self addResult]);
        });
```

```
-(int)addResult
{
    dispatch_semaphore_wait(_semaphore, DISPATCH_TIME_FOREVER);
    result++;
    [NSThread sleepForTimeInterval:0.02];
    dispatch_semaphore_signal(_semaphore);
    return result;
}
```

```
线程1 加1---- 2
线程2 加1---- 3
```

五、队列组

使用队列组可以使得组中的所有任务都执行完毕后再回到主线程执行其他操作。



```
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_group_t group = dispatch_group_create();
dispatch_group_async(group, queue, ^{
   [NSThread sleepForTimeInterval:1];
   NSLog(@“第一个任务");
});
dispatch_group_async(group, queue, ^{
   [NSThread sleepForTimeInterval:2];
   NSLog(@“第二个任务");
});
dispatch_group_async(group, queue, ^{
   [NSThread sleepForTimeInterval:3];
   NSLog(@“第三个任务");
});
dispatch_group_notify(group, dispatch_get_main_queue(), ^{
   NSLog(@“所有任务完成");
});
```

六、其它一些常用函数

dispatch_apply():重复执行某个任务，这个方法没有办法异步执行（为不阻塞线程可以使用dispatch_async()包装一下再执行）。

dispatch_once():单次执行一个任务，此方法中的任务只会执行一次，重复调用也没办法重复执行。

dispatch_time()：延迟一定的时间后执行。

dispatch_barrier_async()：使用此方法创建的任务首先会查看队列中有没有别的任务要执行，如有则会等待已有任务执行完毕再执行；同时在此方法后添加的任务必须等待此方法中任务执行后才能执行。

```
dispatch_queue_t queue = dispatch_queue_create(“barrier_async",
DISPATCH_QUEUE_CONCURRENT);
dispatch_async(queue, ^{
    [NSThread sleepForTimeInterval:2];
    NSLog(@"dispatch_async1");
});
dispatch_async(queue, ^{
    [NSThread sleepForTimeInterval:4];
    NSLog(@"dispatch_async2");
});
dispatch_barrier_async(queue, ^{
    NSLog(@"dispatch_barrier_async");
    [NSThread sleepForTimeInterval:4];

});
dispatch_async(queue, ^{
    [NSThread sleepForTimeInterval:1];
    NSLog(@"dispatch_async3");
});
```

```
dispatch_async1
dispatch_async2
dispatch_barrier_async
dispatch_async3
```

七、原子和非原子属性

在Object C中定义属性时有nonatomic和atomic两种选择。

- atomic：原子属性,会为setter方法加锁（默认为atomic）
- nonatomic：非原子属性，不会为setter方法加锁

原子属性是线程安全，但需要消耗大量的资源。非原子属性非线程安全，但适合内存小的移动设备。

```
@property (assign, atomic) int age;
 - (void)setAge:(int)age
 {
    @synchronized(self) {
        _age = age;
     }
}
```

注意：在IOS开发中，所有属性都应声明为非原子属性，尽量避免多线程抢夺同一块资源，应该将加锁、资源抢夺的业务逻辑交给服务器端处理，减小移动客户端的压力