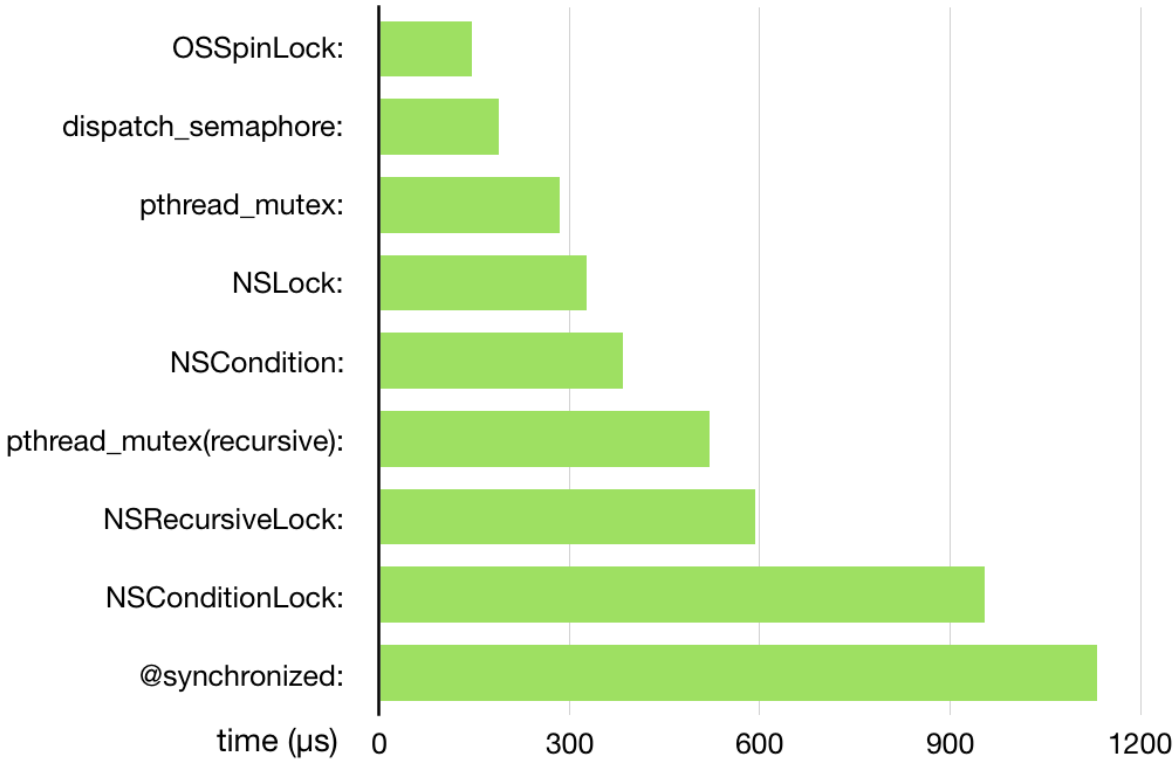
# iOS 锁

# 各种锁的加解锁性能

## 锁的性能对比图




# 基础知识

### 时间片轮转调度算法

时间片轮转调度算法
时间片轮转调度是一种最古老，最简单，最公平且使用最广的算法。每个进程被分配一时间段，称作它的时间片，即该进程允许运行的时间。
这是目前操作系统中大量使用的线程管理方式，大致就是操作系统会给每个线程分配一段时间片（通常100ms左右）这些线程都被放在一个队列中，cpu只需要维护这个队列，当队首的线程时间片耗尽就会被强制放到队尾等待，然后提取下一个队首线程执行。

### 原子操作

“原子”：一般指最小粒度，不可分割；原子操作也就是不可分割，不可中断的操作。我们最重要是知道这个概念，至于具体实现就不在本文的讨论范围。可以参考以下文章：
http://southpeak.github.io/2014/10/17/osatomic-operation/

狭义上的原子操作表示一条不可打断的操作，也就是说线程在执行操作过程中，不会被操作系统挂起，而是一定会执行完。在单处理器环境下，一条汇编指令显然是原子操作，因为中断也要通过指令来实现。

**然而在多处理器的情况下，能够被多个处理器同时执行的操作任然算不上原子操作。因此，真正的原子操作必须由硬件提供支持，比如 x86 平台上如果在指令前面加上 “LOCK” 前缀，对应的机器码在执行时会把总线锁住，使得其他 CPU不能再执行相同操作，从而从硬件层面确保了操作的原子性。**

### 优先级翻转

优先级翻转是当一个高优先级任务通过信号量机制访问共享资源时，该信号量已被一低优先级任务占有，因此造成高优先级任务被许多具有较低优先级任务阻塞，实时性难以得到保证。

# OSSpinLock

### 自旋锁的总结

**原理，通过定义一个全局变量，用来表示锁的可用情况,同时while循环申请锁是原子操作。**
首选作为效率最优选择的OSSpinLock是自旋锁，在 ibireme 的再安全的OSSpinLock文中，已经指出潜在的bug：优先级反转，因此目前不建议使用。
自旋锁的目的是为了确保临界区只有一个线程可以访问，自旋锁的实现思路很简单，理论上来说只要定义一个全局变量，用来表示锁的可用情况即可，自旋锁是使用忙等机制。
伪代码如下:

```
bool lock = false; // 一开始没有锁上，任何线程都可以申请锁
do {
    while(test_and_set(&lock); // test_and_set 是一个原子操作
        Critical section  // 临界区
    lock = false; // 相当于释放锁，这样别的线程可以进入临界区
        Reminder section // 不需要锁保护的代码
}

bool test_and_set (bool *target) {
    bool rv = *target;
    *target = TRUE;
    return rv;
}
```

### 使用场景

如果临界区的执行时间过长，不建议使用自旋锁，因为在 while 循环中，线程处于忙等状态，白白浪费 CPU 时间，最终因为超时被操作系统抢占时间片。

线程在多种情况下会退出自己的时间片。其中一种是用完了时间片的时间，被操作系统强制抢占。除此以外，当线程进行 I/O 操作，或进入睡眠状态时，都会主动让出时间片。显然在 while 循环中，线程处于忙等状态，白白浪费 CPU 时间，最终因为超时被操作系统抢占时间片。如果临界区执行时间较长，比如是文件读写，这种忙等是毫无必要的。

## OSSpinLock 不再安全

主要原因发生在低优先级线程拿到锁时，高优先级线程进入忙等(busy-wait)状态，消耗大量 CPU 时间，从而导致低优先级线程拿不到 CPU 时间，也就无法完成任务并释放锁。这种问题被称为优先级反转。
为什么忙等会导致低优先级线程拿不到时间片？这还得从操作系统的线程调度说起。
现代操作系统在管理普通线程时，通常采用时间片轮转算法(Round Robin，简称 RR)。每个线程会被分配一段时间片(quantum)，通常在 10-100 毫秒左右。当线程用完属于自己的时间片以后，就会被操作系统挂起，放入等待队列中，直到下一次被分配时间片。

## OSSpinLock 代码例子

```
//设置票的数量为5
    _tickets = 5;
    //创建锁
    _pinLock = OS_SPINLOCK_INIT;
    //线程1
    dispatch_async(self.concurrentQueue, ^{
        [self saleTickets];
    });
    //线程2
    dispatch_async(self.concurrentQueue, ^{
        [self saleTickets];
    });

- (void)saleTickets {

        while (1) {
            [NSThread sleepForTimeInterval:1];
            //加锁
            OSSpinLockLock(&_pinLock);

            if (_tickets > 0) {
                _tickets--;
                NSLog(@"剩余票数= %ld, Thread:%@",_tickets,[NSThread curren
tThread]);

            } else {
                NSLog(@"票卖完了  Thread:%@",[NSThread currentThread]);
                break;
            }
            //解锁
            OSSpinLockUnlock(&_pinLock);
        }
}

#import <libkern/OSAtomic.h>
interface ViewController ()
{
    OSSpinLock spinlock;
```

```
}
end
implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    self.number = 10;
    spinlock = OS_SPINLOCK_INIT;
}
- (IBAction)test:(id)sender {
    for (int i = 0; i<10; i++) {
        dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
DEFAULT, 0), ^{
            [self sellTicket];
        });
    }
}
- (void)sellTicket {
    OSSpinLockLock(&spinlock);

    if (self.number > 0) {
        self.number--;
        NSLog(@"%@还剩%ld张票",[NSThread currentThread],self.number);
    }

    OSSpinLockUnlock(&spinlock);
}
end
```

# dispatch_semaphore 信号量

## 使用

```
dispatch_semaphore_create(1)：传入值必须>=0，若传入为0则阻塞线程并等待timeout,时间到后会执行其后的语句
dispatch_semaphore_wait(signal, overTime)：可以理解为lock,会使得signal值-1
dispatch_semaphore_signal(signal)：可以理解为unlock,会使得signal值+1
```

## 理解

```
停车场剩余4个车位，那么即使同时来了四辆车也能停的下。如果此时来了五辆车，那么就有一辆需要等待。
信号量的值（signal）就相当于剩余车位的数目，dispatch_semaphore_wait函数就相当于来了一辆车，dispatch_semaphore_signal就相当于走了一辆车。停车位的剩余数目在初始化的时候就已经指明了（dispatch_semaphore_create（long value））,调用一次 dispatch_semaphore_signal，剩余的车位就增加一个；调用一次dispatch_semaphore_wait 剩余车位就减少一个；当剩余车位为 0 时，再来车（即调用 dispatch_semaphore_wait）就只能等待。有
```

> 可能同时有几辆车等待一个停车位。有些车主没有耐心，给自己设定了一段等待时间，这段时间内等不到停车位就走了，如果等到了就开进去停车。而有些车主就像把车停在这，所以就一直等下去。

## 信号量锁机制原理说明

GCD提供一种信号的机制，dispatch_semaphore是gcd中通过信号量来实现共享数据的数据安全，不是使用忙等，而是阻塞线程并睡眠，需要进行上下文切换。

参考介绍 GCD 底层实现的文章中简单描述了信号量 dispatch_semaphore_t 的实现原理，它最终会调用到 sem_wait 方法，这个方法在 glibc 中被实现如下:

```
int sem_wait (sem_t *sem) {
  int *futex = (int *) sem;
  if (atomic_decrement_if_positive (futex) > 0)
    return 0;
  int err = lll_futex_wait (futex, 0);
    return -1;
}
```

首先会把信号量的值减一，并判断是否大于零。如果大于零，说明不用等待，所以立刻返回。具体的等待操作在 lll_futex_wait 函数中实现，lll 是 low level lock 的简称。这个函数通过汇编代码实现，调用到 SYS_futex 这个系统调用，使线程进入睡眠状态，主动让出时间片，这个函数在互斥锁的实现中，也有可能被用到。

主动让出时间片并不总是代表效率高。让出时间片会导致操作系统切换到另一个线程，这种上下文切换通常需要 10 微秒左右，而且至少需要两次切换。如果等待时间很短，比如只有几个微秒，忙等就比线程睡眠更高效。

可以看到，自旋锁和信号量的实现都非常简单，这也是两者的加解锁耗时分别排在第一和第二的原因。再次强调，加解锁耗时不能准确反应出锁的效率(比如时间片切换就无法发生)，它只能从一定程度上衡量锁的实现复杂程度。

## 优缺点

缺点
在时间较短的操作，没有自旋锁高效，会有上下文切换的成本。
优点
效率高

## 具体代码实例

```
// 实例类person
Person *person = [[Person alloc] init];

// 创建并设置信量
dispatch_semaphore_t semaphore = dispatch_semaphore_create(1);
```

```
// 线程A
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
    dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
    [person personA];
    [NSThread sleepForTimeInterval:5];
    dispatch_semaphore_signal(semaphore);
});

// 线程B
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
    dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
    [person personB];
    dispatch_semaphore_signal(semaphore);
});
```

# pthread_mutex互斥锁

pthread 表示 POSIX thread，定义了一组跨平台的线程相关的 API，pthread_mutex 表示互斥锁。互斥锁的实现原理与信号量非常相似，不是使用忙等，而是阻塞线程并睡眠，需要进行上下文切换。

## 常见用法

互斥锁的常见用法如下:

```
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_NORMAL);  // 定义锁的属性

pthread_mutex_t mutex;
pthread_mutex_init(&mutex, &attr) // 创建锁

pthread_mutex_lock(&mutex); // 申请锁
    // 临界区
pthread_mutex_unlock(&mutex); // 释放锁
```

## 互斥锁的实现

互斥锁在申请锁时，调用了 `pthread_mutex_lock` 方法，它在不同的系统上实现各有不同，有时候它的内部是使用信号量来实现，即使不用信号量，也会调用到 `lll_futex_wait` 函数，从而导致线程休眠。

上文说到如果临界区很短，忙等的效率也许更高，所以在有些版本的实现中，会首先尝试一定次数(比如 `1000` 次)的 `testandtest`，这样可以在错误使用互斥锁时提高性能。

对于 pthread_mutex 来说，它的用法和之前没有太大的改变，比较重要的是锁的类型，可以有 PTHREAD_MUTEX_NORMAL、PTHREAD_MUTEX_ERRORCHECK、PTHREAD_MUTEX_RECURSIVE 等等，具体的特性就不做解释了，网上有很多相关资料。

一般情况下，一个线程只能申请一次锁，也只能在获得锁的情况下才能释放锁，多次申请锁或释放未获得的锁都会导致崩溃。假设在已经获得锁的情况下再次申请锁，线程会因为等待锁的释放而进入睡眠状态，因此就不可能再释放锁，从而导致死锁。

然而这种情况经常会发生，比如某个函数申请了锁，在临界区内又递归调用了自己。辛运的是 pthread_mutex 支持递归锁，也就是允许一个线程递归的申请锁，只要把 attr 的类型改成 PTHREAD_MUTEX_RECURSIVE 即可。

注意：必须在头文件导入：#import <pthread.h>

```
// 实例类person
Person *person = [[Person alloc] init];

// 创建锁对象
__block pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);

// 线程A
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
    pthread_mutex_lock(&mutex);
    [person personA];
    [NSThread sleepForTimeInterval:5];
    pthread_mutex_unlock(&mutex);
});

// 线程B
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
    pthread_mutex_lock(&mutex);
    [person personB];
    pthread_mutex_unlock(&mutex);
});
```

pthread_mutex 互斥锁

```
__block pthread_mutex_t mutex;
    pthread_mutex_init(&mutex, NULL);

    //线程1
    dispatch_async(self.concurrentQueue), ^{
        pthread_mutex_lock(&mutex);
        NSLog(@"任务1");
        sleep(2);
```

```
        pthread_mutex_unlock(&mutex);
    });

    //线程2
    dispatch_async(self.concurrentQueue), ^{
        sleep(1);
        pthread_mutex_lock(&mutex);
        NSLog(@"任务2");
        pthread_mutex_unlock(&mutex);
    });
```

# NSLock

NSLock 是 Objective-C 以对象的形式暴露给开发者的一种锁，它的实现非常简单，通过宏，定义了 lock 方法:

```
#define    MLOCK \
- (void) lock\
{\
  int err = pthread_mutex_lock(&_mutex);\
  // 错误处理 ……
}
```

NSLock 只是在内部封装了一个 pthread_mutex，属性为 PTHREAD_MUTEX_ERRORCHECK，它会损失一定性能换来错误提示。

这里使用宏定义的原因是，OC 内部还有其他几种锁，他们的 lock 方法都是一模一样，仅仅是内部 pthread_mutex 互斥锁的类型不同。通过宏定义，可以简化方法的定义。

NSLock 比 pthread_mutex 略慢的原因在于它需要经过方法调用，同时由于缓存的存在，多次方法调用不会对性能产生太大的影响。

**NSLock 互斥锁 不能多次调用 lock方法,会造成死锁**
在Cocoa程序中NSLock中实现了一个简单的互斥锁。
所有锁（包括NSLock）的接口实际上都是通过NSLocking协议定义的，它定义了lock和unlock方法。你使用这些方法来获取和释放该锁。
NSLock类还增加了tryLock和lockBeforeDate:方法。
tryLock试图获取一个锁，但是如果锁不可用的时候，它不会阻塞线程，相反，它只是返回NO。
lockBeforeDate:方法试图获取一个锁，但是如果锁没有在规定的时间内被获得，它会让线程从阻塞状态变为非阻塞状态（或者返回NO）

# NSCondition

NSCondition 的底层是通过条件变量(condition variable) pthread_cond_t 来实现的。条件变量有点像信号量，提供了线程阻塞与信号机制，因此可以用来阻塞某个线程，并等待某个数据就绪，随后

唤醒线程，比如常见的生产者-消费者模式

## 使用条件变量

很多介绍 pthread_cond_t 的文章都会提到，它需要与互斥锁配合使用:

```
void consumer () { // 消费者
    pthread_mutex_lock(&mutex);
    while (data == NULL) {
        pthread_cond_wait(&condition_variable_signal, &mutex); // 等待数据
    }
    // --- 有新的数据，以下代码负责处理 ↓↓↓↓↓↓
    // temp = data;
    // --- 有新的数据，以上代码负责处理 ↑↑↑↑↑↑
    pthread_mutex_unlock(&mutex);
}

void producer () {
    pthread_mutex_lock(&mutex);
    // 生产数据
    pthread_cond_signal(&condition_variable_signal); // 发出信号给消费者，告
诉他们有了新的数据
    pthread_mutex_unlock(&mutex);
}
```

自然我们会有疑问:“如果不用互斥锁，只用条件变量会有什么问题呢？”。问题在于，temp = data; 这段代码不是线程安全的，也许在你把 data 读出来以前，已经有别的线程修改了数据。因此我们需要保证消费者拿到的数据是线程安全的。
wait 方法除了会被 signal 方法唤醒，有时还会被虚假唤醒，所以需要这里 while 循环中的判断来做二次确认。

## NSCondition的实现原理

NSCondition 其实是封装了一个互斥锁和条件变量。NSCondition 的底层是通过条件变量(condition variable) pthread_cond_t 来实现的。条件变量有点像信号量，提供了线程阻塞与信号机制，因此可以用来阻塞某个线程，并等待某个数据就绪，随后唤醒线程。它仅仅是控制了线程的执行顺序。

NSCondition 其实是封装了一个互斥锁和条件变量， 它把前者的 lock 方法和后者的 wait/signal 统一在 NSCondition 对象中，暴露给使用者。NSCondition的加解锁过程与 NSLock 几乎一致，理论上来说耗时也应该一样(实际测试也是如此)。在图中显示它耗时略长，有可能是测试者在每次加解锁的前后还附带了变量的初始化和销毁操作。

**互斥锁提供线程安全，条件变量提供线程阻塞与信号机制。**
它的基本用法和NSLock一样，这里说一下NSCondition的特殊用法。
NSCondition提供更高级的用法，方法如下:

```
- (void)wait; //阻塞当前线程 直到等待唤醒
- (BOOL)waitUntilDate:(NSDate *)limit;  //阻塞当前线程到一定时间 之后自动唤醒
```

```
- (void)signal; //唤醒一条阻塞线程
- (void)broadcast; //唤醒所有阻塞线程
```

## 为什么要使用条件变量

介绍条件变量的文章非常多，但大多都对一个一个基本问题避而不谈:“为什么要用条件变量？它仅仅是控制了线程的执行顺序，用信号量或者互斥锁能不能模拟出类似效果？”

网上的相关资料比较少，我简单说一下个人看法。信号量可以一定程度上替代 condition，但是互斥锁不行。在以上给出的生产者-消费者模式的代码中， pthread_cond_wait 方法的本质是锁的转移，消费者放弃锁，然后生产者获得锁，同理，pthread_cond_signal 则是一个锁从生产者到消费者转移的过程。

如果使用互斥锁，我们需要把代码改成这样:

```
void consumer () { // 消费者
    pthread_mutex_lock(&mutex);
    while (data == NULL) {
        pthread_mutex_unlock(&mutex);
        pthread_mutex_lock(&another_lock)  // 相当于 wait 另一个互斥锁
        pthread_mutex_lock(&mutex);
    }
    pthread_mutex_unlock(&mutex);
}
```

## NSCondition 的做法

NSCondition 其实是封装了一个互斥锁和条件变量， 它把前者的 lock 方法和后者的 wait/signal 统一在 NSCondition 对象中，暴露给使用者:

```
- (void) signal {
  pthread_cond_signal(&_condition);
}

// 其实这个函数是通过宏来定义的，展开后就是这样
- (void) lock {
  int err = pthread_mutex_lock(&_mutex);
}
```

它的加解锁过程与 NSLock 几乎一致，理论上来说耗时也应该一样(实际测试也是如此)。在图中显示它耗时略长，我猜测有可能是测试者在每次加解锁的前后还附带了变量的初始化和销毁操作。

## 使用POSIX（条件锁）创建锁

```
// 实例类person
Person *person = [[Person alloc] init];
```

```
// 创建互斥锁
__block pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
// 创建条件锁
__block pthread_cond_t cond;
pthread_cond_init(&cond, NULL);

// 线程A
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
    pthread_mutex_lock(&mutex);
    pthread_cond_wait(&cond, &mutex);
    [person personA];
    pthread_mutex_unlock(&mutex);
});

// 线程B
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
    pthread_mutex_lock(&mutex);
    [person personB];
    [NSThread sleepForTimeInterval:5];
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
});
```

效果：程序会首先调用线程B，在5秒后再调用线程A。因为在线程A中创建了等待条件锁，线程B有激活锁，只有当线程B执行完后会激活线程A。
pthread_cond_wait方法为等待条件锁。
pthread_cond_signal方法为激动一个相同条件的条件锁。

# pthread_mutex(recursive) 递归锁

一般情况下，一个线程只能申请一次锁，也只能在获得锁的情况下才能释放锁，多次申请锁或释放未获得的锁都会导致崩溃。假设在已经获得锁的情况下再次申请锁，线程会因为等待锁的释放而进入睡眠状态，因此就不可能再释放锁，从而导致死锁。
然而这种情况经常会发生，比如某个函数申请了锁，在临界区内又递归调用了自己，由此也就引出了递归锁：允许同一个线程在未释放其拥有的锁时反复对该锁进行加锁操作。

递归锁的使用和pthread_mutex很类似，主要就是要设置锁的类型为PTHREAD_MUTEX_RECURSIVE即可。

> 由于 pthread_mutex 有多种类型，可以支持递归锁等，因此在申请加锁时，需要对锁的类型加以判断，这也就是为什么它和信号量的实现类似，但效率略低的原因。

# NSRecursiveLock

上文已经说过，递归锁也是通过 pthread_mutex_lock 函数来实现，在函数内部会判断锁的类型，如果显示是递归锁，就允许递归调用，仅仅将一个计数器加一，锁的释放过程也是同理。

NSRecursiveLock 与 NSLock 的区别在于内部封装的 pthread_mutex_t 对象的类型不同，前者的类型为 PTHREAD_MUTEX_RECURSIVE。

多次调用不会阻塞已获取该锁的线程，不会死锁。
实际使用：

```
// 实例类person
Person *person = [[Person alloc] init];
// 创建锁对象
NSRecursiveLock *theLock = [[NSRecursiveLock alloc] init];

// 创建递归方法
static void (^testCode)(int);
testCode = ^(int value) {
    [theLock tryLock];
    if (value > 0) {
        [person personA];
        [NSThread sleepForTimeInterval:1];
        testCode(value - 1);
    }
    [theLock unlock];
};

//线程A
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
    testCode(5);
});

//线程B
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
    [theLock lock];
    [person personB];
    [theLock unlock];
});
```

如果我们把NSRecursiveLock类换成NSLock类，那么程序就会死锁。因为在此例子中，递归方法会造成锁被多次锁定（Lock），所以自己也被阻塞了。而使用NSRecursiveLock类，则可以避免这个问题。

**NSRecursiveLock 递归锁**
使用锁最容易犯的一个错误就是在递归或循环中造成死锁
如下代码中，因为在线程1中的递归block中，锁会被多次的lock，所以自己也被阻塞了

```
    //创建锁
    _mutexLock = [[NSLock alloc]init];

    //线程1
    dispatch_async(self.concurrentQueue, ^{
        static void(^TestMethod)(int);
        TestMethod = ^(int value)
        {
            [_mutexLock lock];
            if (value > 0)
            {
                [NSThread sleepForTimeInterval:1];
                TestMethod(value--);
            }
            [_mutexLock unlock];
        };

        TestMethod(5);
    });
```

此处将NSLock换成NSRecursiveLock，便可解决问题。
NSRecursiveLock类定义的锁可以在同一线程多次lock，而不会造成死锁。
递归锁会跟踪它被多少次lock。每次成功的lock都必须平衡调用unlock操作。
只有所有的锁住和解锁操作都平衡的时候，锁才真正被释放给其他线程获得。

```
//创建锁
    _rsLock = [[NSRecursiveLock alloc] init];

   //线程1
    dispatch_async(self.concurrentQueue, ^{
        static void(^TestMethod)(int);
        TestMethod = ^(int value) {
            [_rsLock lock];
            if (value > 0) {
                [NSThread sleepForTimeInterval:1];
                TestMethod(value--);
            }
            [_rsLock unlock];
        };

        TestMethod(5);
    });
```

# NSConditionLock

NSConditionLock 借助 NSCondition 来实现，它的本质就是一个生产者-消费者模型。“条件被满足”可以理解为生产者提供了新的内容。NSConditionLock 的内部持有一个 NSCondition 对象，以

及 _condition_value 属性，在初始化时就会对这个属性进行赋值:

```
// 简化版代码
- (id) initWithCondition: (NSInteger)value {
    if (nil != (self = [super init])) {
        _condition = [NSCondition new]
        _condition_value = value;
    }
    return self;
}
```

它的 lockWhenCondition 方法其实就是消费者方法:

```
- (void) lockWhenCondition: (NSInteger)value {
    [_condition lock];
    while (value != _condition_value) {
        [_condition wait];
    }
}
```

对应的 unlockWhenCondition 方法则是生产者，使用了 broadcast 方法通知了所有的消费者:

```
- (void) unlockWithCondition: (NSInteger)value {
    _condition_value = value;
    [_condition broadcast];
    [_condition unlock];
}
```

使用此方法可以创建一个指定开锁的条件，只有满足条件，才能开锁。

```
// 实例类person
Person *person = [[Person alloc] init];
// 创建条件锁
NSConditionLock *conditionLock = [[NSConditionLock alloc] init];

// 线程A
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
    [conditionLock lock];
    [person personA];
    [NSThread sleepForTimeInterval:5];
    [conditionLock unlockWithCondition:10];
});

// 线程B
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
```

```
        [conditionLock lockWhenCondition:10];
        [person personB];
        [conditionLock unlock];
    });
```

线程A使用的是lock方法，因此会直接进行锁定，并且指定了只有满足10的情况下，才能成功解锁。
unlockWithCondition:方法，创建条件锁，参数传入“整型”。lockWhenCondition:方法，则为解锁，也是传入一个“整型”的参数。

```
    //主线程中
    NSConditionLock *theLock = [[NSConditionLock alloc] init];

    //线程1
    dispatch_async(self.concurrentQueue, ^{
        for (int i=0;i<=3;i++)
        {
            [theLock lock];
            NSLog(@"thread1:%d",i);
            sleep(1);
            [theLock unlockWithCondition:i];
        }
    });

    //线程2
    dispatch_async(self.concurrentQueue, ^{
        [theLock lockWhenCondition:2];
        NSLog(@"thread2");
        [theLock unlock];
    });
```

在线程1中的加锁使用了lock，是不需要条件的，所以顺利的就锁住了。
unlockWithCondition:在开锁的同时设置了一个整型的条件 2 。
线程2则需要一把被标识为2的钥匙，所以当线程1循环到 i = 2 时，线程2的任务才执行。
NSConditionLock也跟其它的锁一样，是需要lock与unlock对应的，只是lock,lockWhenCondition:与unlock，unlockWithCondition:是可以随意组合的，当然这是与你的需求相关的。

# @synchronized（互斥锁）

显然，这是我们最熟悉的加锁方式，因为这是OC层面的为我们封装的，使用起来简单粗暴。使用时 @synchronized 后面需要紧跟一个 OC 对象，它实际上是把这个对象当做锁来使用。这是通过一个哈希表来实现的，OC 在底层使用了一个互斥锁的数组(也就是锁池)，通过对对象去哈希值来得到对应的互斥锁。

这其实是一个 OC 层面的锁， 主要是通过牺牲性能换来语法上的简洁与可读。
我们知道 @synchronized 后面需要紧跟一个 OC 对象，它实际上是把这个对象当做锁来使用。这是

通过一个哈希表来实现的，OC 在底层使用了一个互斥锁的数组(你可以理解为锁池)，通过对对象去哈希值来得到对应的互斥锁。

若是在self对象上频繁加锁，那么程序可能要等另一段与此无关的代码执行完毕，才能继续执行当前代码，这样做其实并没有必要。
使用场景：创建单例时使用。

优点：使用@synchronized关键字可以很方便地创建锁对象，而且不用显式的创建锁对象。

缺点：会隐式添加一个异常处理来保护代码，该异常处理会在异常抛出的时候自动释放互斥锁。而这种隐式的异常处理会带来系统的额外开销，为优化资源，你可以使用锁对象。

具体的实现原理可以参考这篇文章: 关于 @synchronized，这儿比你想知道的还要多

```
// 实例类person
Person *person = [[Person alloc] init];

// 线程A
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
    @synchronized(person) {
        [person personA];
        [NSThread sleepForTimeInterval:3]; // 线程休眠3秒
    }
});

// 线程B
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
    @synchronized(person) {
        [person personB];
    }
});
```

关键字@synchronized的使用，锁定的对象为锁的唯一标识，只有标识相同时，才满足互斥。如果线程B锁对象person改为self或其它标识，那么线程B将不会被阻塞。你是否看到@synchronized(self)，也是对的。它可以锁任何对象，描述为@synchronized(anObj)。

```
@synchronized(这里添加一个OC对象，一般使用self) {
        这里写要加锁的代码
  }
  注意点
    1.加锁的代码尽量少
    2.添加的OC对象必须在多个线程中都是同一对象
    3.优点是不需要显式的创建锁对象，便可以实现锁的机制。
    4. @synchronized块会隐式的添加一个异常处理例程来保护代码，该处理例程会在异常抛出的时候自动的释放互斥锁。所以如果不想让隐式的异常处理例程带来额外的开销，你可以考虑使用锁对象。
```

二、“Object－C”语言

1）NSLock（互斥锁）
2）NSRecursiveLock（递归锁）
条件锁，递归或循环方法时使用此方法实现锁，可避免死锁等问题。
3）NSConditionLock（条件锁）
使用此方法可以指定，只有满足条件的时候才可以解锁。
4）NSDistributedLock（分布式锁）
在IOS中不需要用到，也没有这个方法，因此本文不作介绍，这里写出来只是想让大家知道有这个锁存在。
如果想要学习NSDistributedLock的话，你可以创建MAC OS的项目自己演练，方法请自行Google，谢谢。

三、C语言
1）pthread_mutex_t（互斥锁）
2）GCD－信号量（“互斥锁”）
3）pthread_cond_t（条件锁）

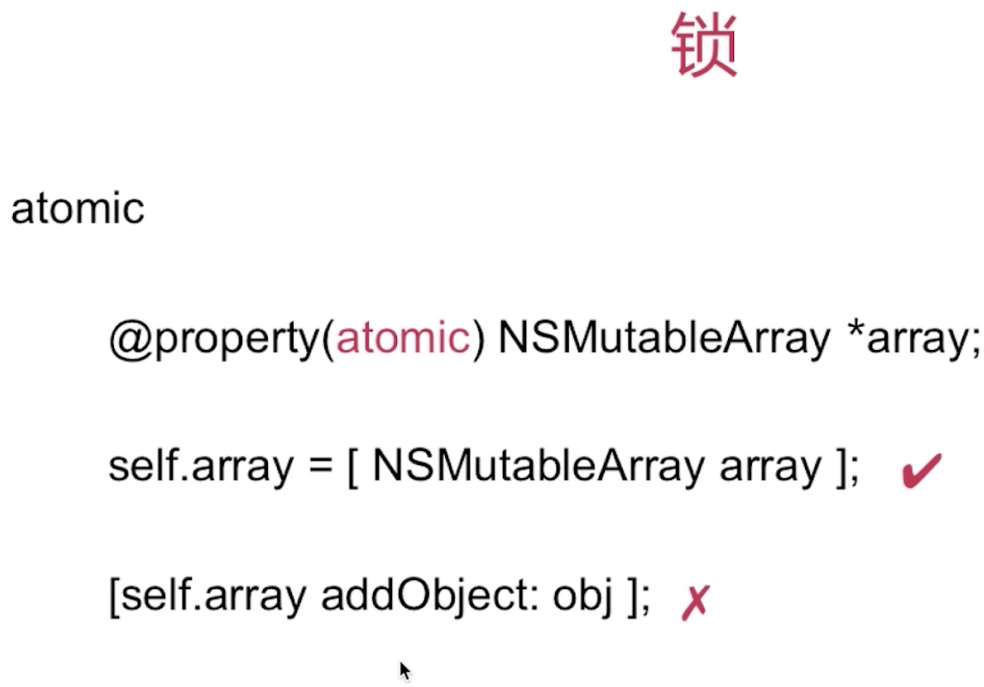
综合上述分析与讨论，总结有以下几点原则：

```
1、总的来看，推荐pthread_mutex作为实际项目的首选方案；
2、对于耗时较大又易冲突的读操作，可以使用读写锁代替pthread_mutex；
3、如果确认仅有set/get的访问操作，可以选用原子操作属性；
4、对于性能要求苛刻，可以考虑使用OSSpinLock，需要确保加锁片段的耗时足够小；
5、条件锁基本上使用面向对象的NSCondition和NSConditionLock即可；
6、@synchronized则适用于低频场景如初始化或者紧急修复使用；
1.自旋锁：OSSpinLock 在ios中已经不是线程安全的了，如果共享数据已经有其他线程加锁了，线程会以死循环的方式等待锁，一旦被访问的资源被解锁，则等待资源的线程会立即执行。（效率最高，如果一直等不到锁会较占用cpu资源）
2.信号量：dispatch_semaphore是gcd中通过信号量来实现共享数据的数据安全。（效率第二）
3.互斥锁：pthread_mutex ，nslock ，synchronized都是互斥锁。如果共享数据已经有其他线程加锁了，线程会进入休眠状态等待锁。一旦被访问的资源被解锁，则等待资源的线程会被唤醒。（synchronized效率最低）
4.递归锁：pthread_mutex(recursive)与NSRecursiveLock ， 多次调用不会阻塞已获取该锁的线程。
5.条件锁：nsconditionlock 满足一定的条件的加锁和解锁，可以实现依赖关系。nsconditionlock条件锁，也是通过信号来解锁，主要用来实现生产者消费者模式。
```

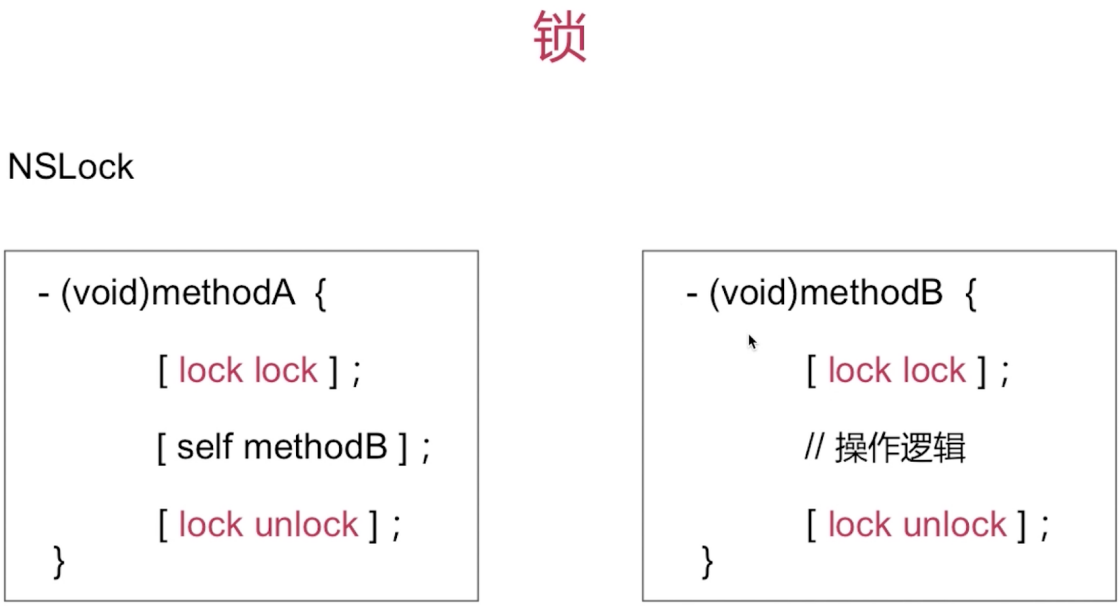
# 我们平时使用的：

1. synchronized，一般用在创建单例的时候。

2. atomic修饰属性的关键字，它不是绝对安全的。




3. 一般使用NSLock即可，但是如果方法会有递归调用则会死锁




这时我们使用递归锁：

# 锁

NSRecursiveLock

```
- (void)methodA {
        [ recursiveLock lock ] ;
        [ self methodB ] ;
        [recursiveLock unlock ] ;
}
```

```
- (void)methodB {
        [recursiveLock lock ] ;
        // 操作逻辑
        [recursiveLock unlock ] ;
}
```

4.OSSpinLock自旋锁，轮询的方式，用于轻量级的数据操作+1/-1。
5.信号量：dispatch_semaphore是gcd中通过信号量来实现共享数据的数据安全。（效率第二）

## 知识总结

1.有事你喊我，397027757 wechat or qq，好用给一个星
2.已经做了一个代码测试，但是结果有时候不固定，大致趋势是一致的
3.测试代码：ZgtiOSLock