多线程

@M了个J

https://github.com/CoderMJLee

实力IT教育 www.520it.com

小码哥教育 SEEMYGO

面试题

- 你理解的多线程？
- iOS的多线程方案有哪几种？你更倾向于哪一种？
- 你在项目中用过 GCD 吗？
- GCD 的队列类型
- 说一下 OperationQueue 和 GCD 的区别，以及各自的优势
- 线程安全的处理手段有哪些？
- OC你了解的锁有哪些？在你回答基础上进行二次提问；
  - 追问一：自旋和互斥对比？
  - 追问二：使用以上锁需要注意哪些？
  - 追问三：用C/OC/C++，任选其一，实现自旋或互斥？口述即可！

面试题

- 请问下面代码的打印结果是什么？

```objc
dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
dispatch_async(queue, ^{
    NSLog(@"1");

    [self performSelector:@selector(test) withObject:nil afterDelay:.0];

    NSLog(@"3");
});
```

```objc
- (void)test
{
    NSLog(@"2");
}
```

- 打印结果是：1、3
- 原因
  - performSelector:withObject:afterDelay:的本质是往Runloop中添加定时器
  - 子线程默认没有启动Runloop

小码哥教育 SEEMYGO

面试题

- 请问下面代码的打印结果是什么？

```
- (void)test
{
    NSLog(@"2");
}

- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    NSThread *thread = [[NSThread alloc] initWithBlock:^{
        NSLog(@"1");
    }];
    [thread start];

    [self performSelector:@selector(test) onThread:thread withObject:nil waitUntilDone:YES];
}
```

小码哥教育 SEEMYGO

iOS中的常见多线程方案

技术方案	简介	语言	线程生命周期	使用频率
pthread	■ 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	■ 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	■ 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	■ 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

GCD的常用函数

- GCD中有2个用来执行任务的函数

用同步的方式执行任务

```
dispatch_sync(dispatch_queue_t queue, dispatch_block_t block);
```

✓ queue：队列

✓ block：任务

用异步的方式执行任务

```
dispatch_async(dispatch_queue_t queue, dispatch_block_t block);
```

- GCD源码：https://github.com/apple/swift-corelibs-libdispatch

GCD的队列

- GCD的队列可以分为2大类型

- 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

- 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）

容易混淆的术语

- 有4个术语比较容易混淆：同步、异步、并发、串行
- 同步和异步主要影响：能不能开启新的线程
  - ✓ 同步：在当前线程中执行任务，不具备开启新线程的能力
  - ✓ 异步：在新的线程中执行任务，具备开启新线程的能力

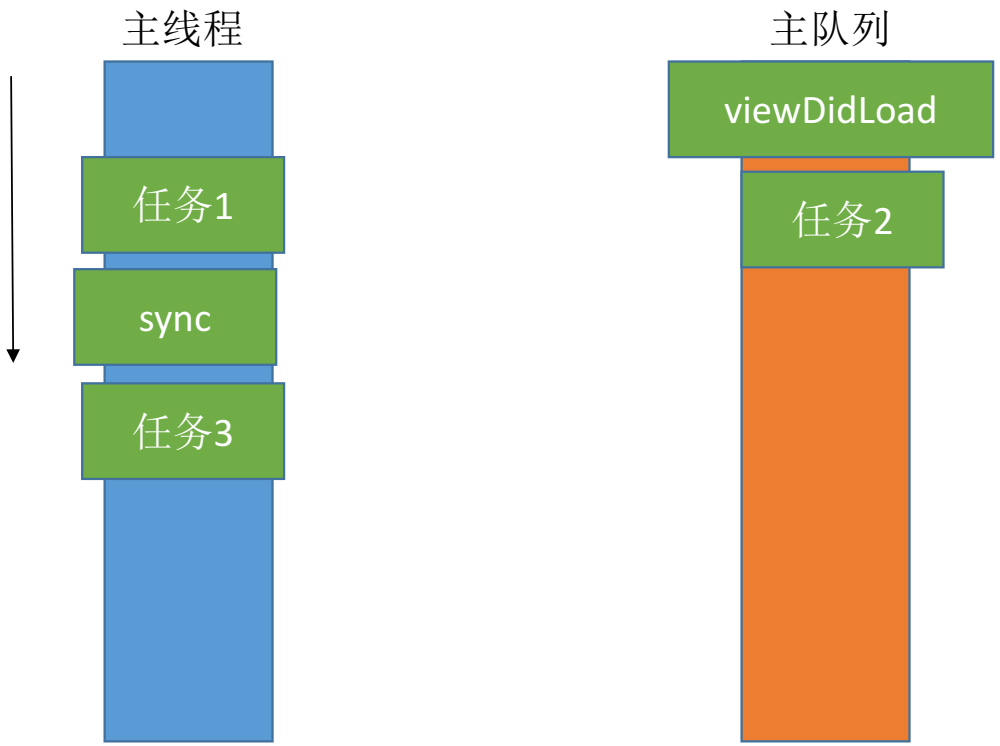
- 并发和串行主要影响：任务的执行方式
  - ✓ 并发：多个任务并发（同时）执行
  - ✓ 串行：一个任务执行完毕后，再执行下一个任务

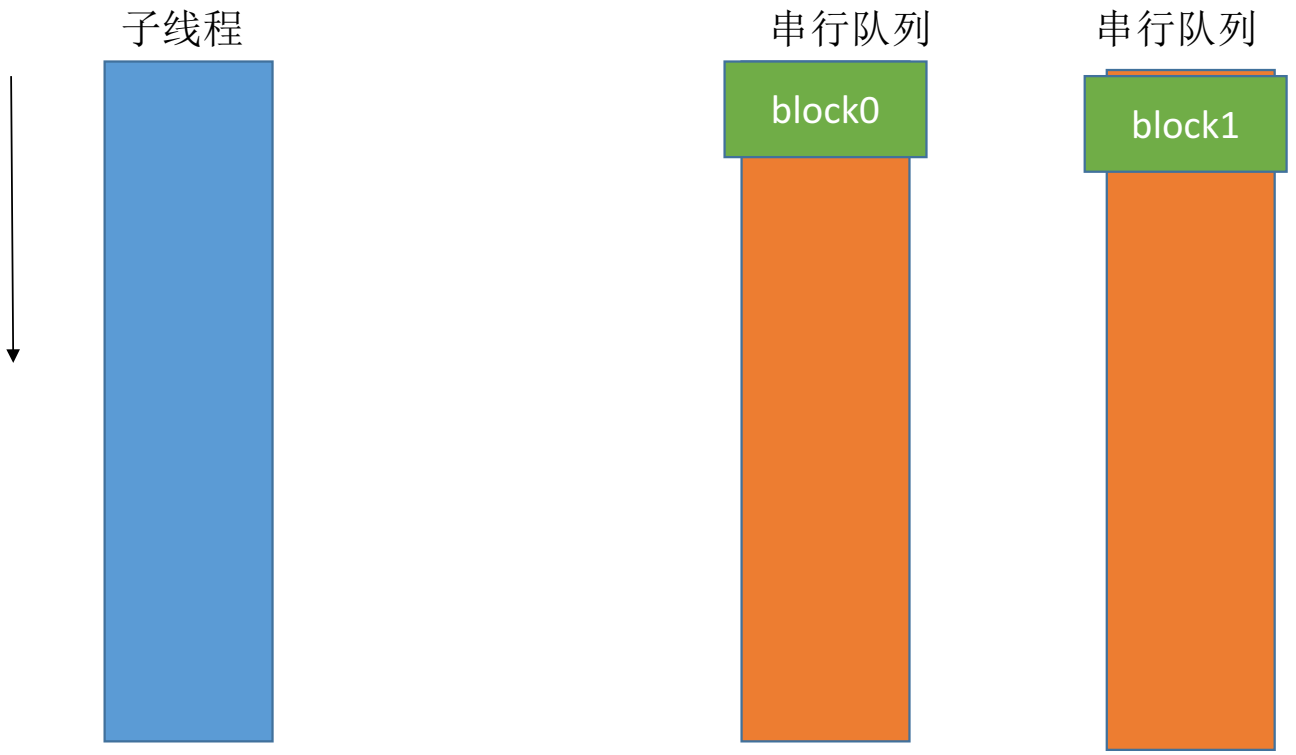
小码哥教育 SEEMYGO

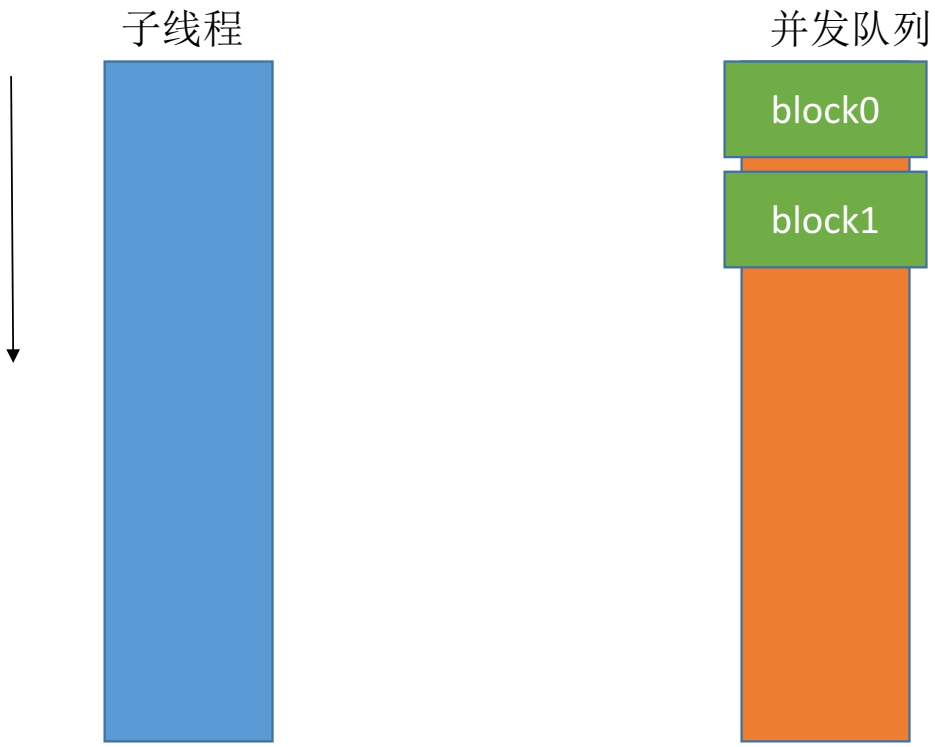
各种队列的执行效果

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

■ 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（产生死锁）


主线程
任务1
sync
任务3
主队列
viewDidLoad
任务2


子线程
串行队列
block0
串行队列
block1


子线程
并发队列
block0
block1

队列组的使用

- 思考：如何用gcd实现以下功能
- 异步并发执行任务1、任务2
- 等任务1、任务2都执行完毕后，再回到主线程执行任务3

```
dispatch_group_t group = dispatch_group_create();
dispatch_queue_t queue = dispatch_queue_create("myqueue", DISPATCH_QUEUE_CONCURRENT);
dispatch_group_async(group, queue, ^{
    NSLog(@"任务1111");
});
dispatch_group_async(group, queue, ^{
    NSLog(@"任务2222");
});
dispatch_group_notify(group, queue, ^{
    dispatch_async(dispatch_get_main_queue(), ^{
        NSLog(@"任务3333");
    });
});
```

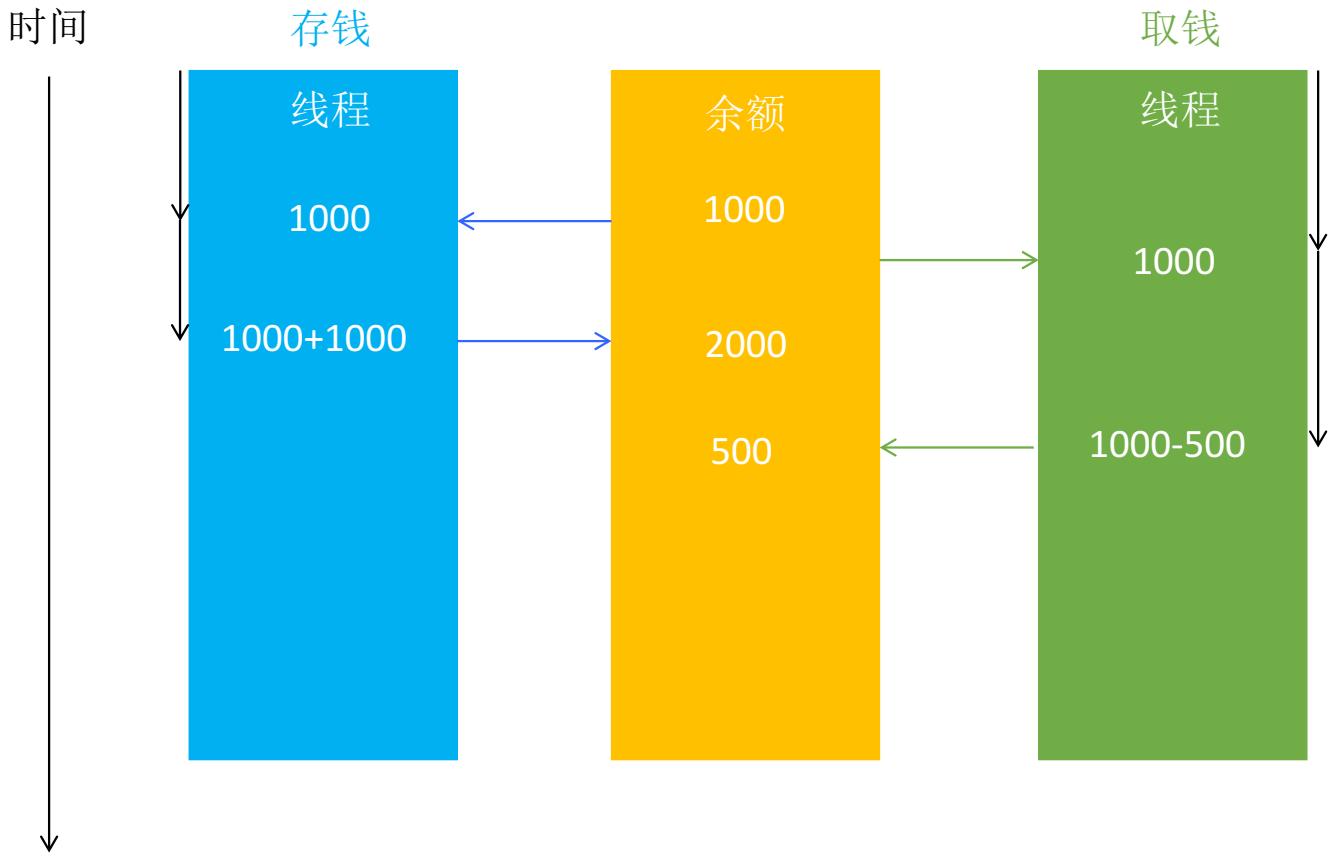
小码哥教育 SEEMYGO

多线程的安全隐患

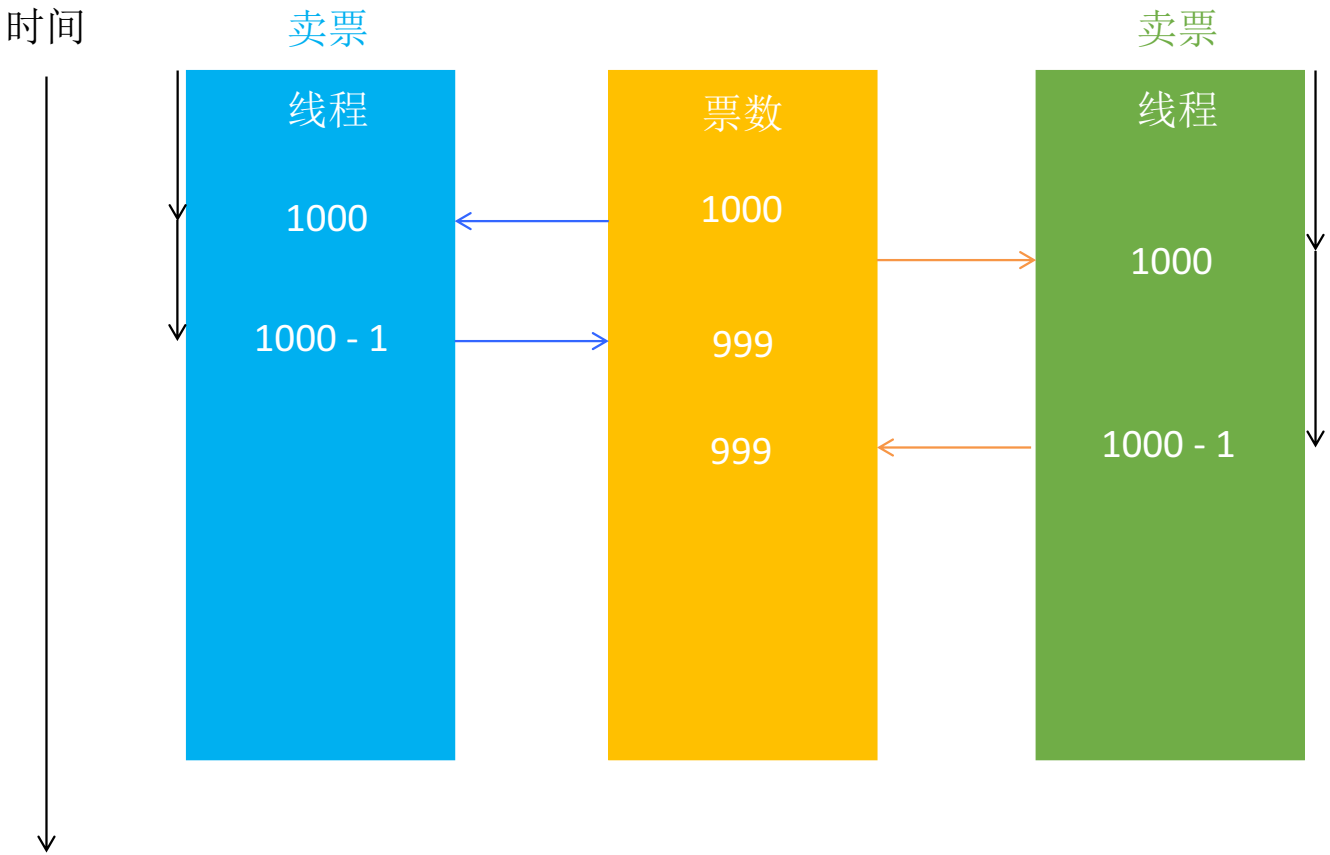
- 资源共享
  - 1块资源可能会被多个线程共享，也就是多个线程可能会访问同一块资源
  - 比如多个线程访问同一个对象、同一个变量、同一个文件

- 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

多线程安全隐患示例01 – 存钱取钱

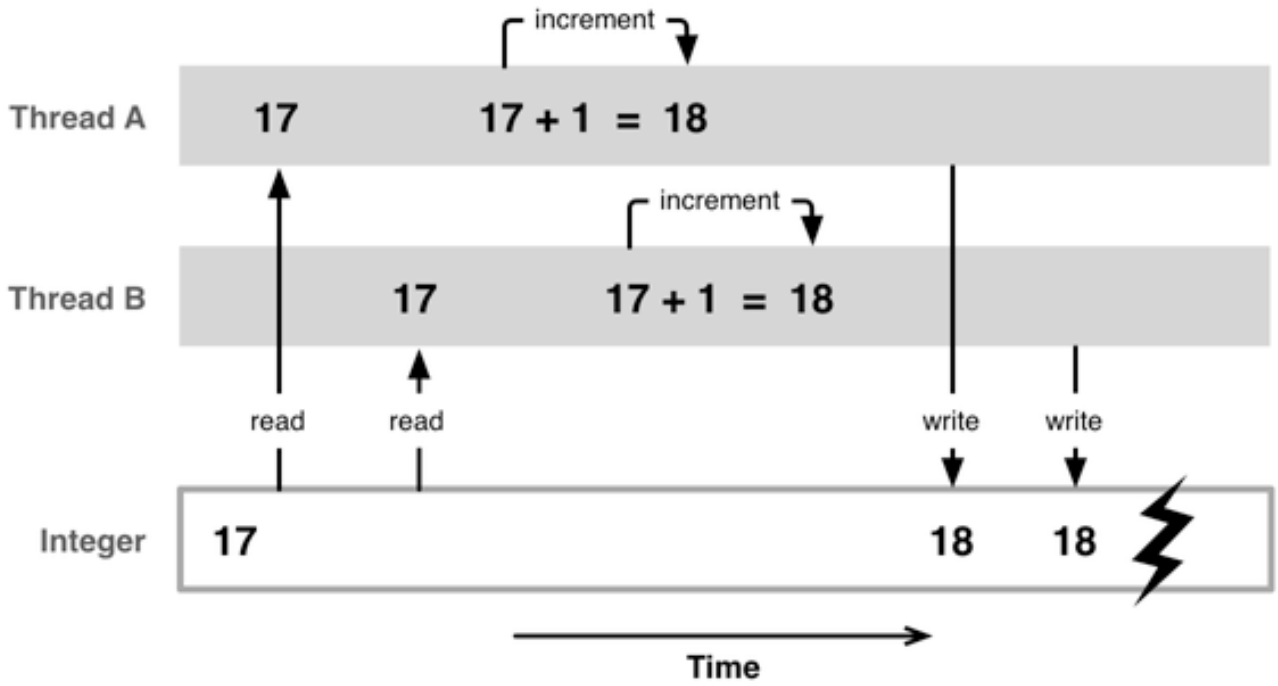


多线程安全隐患示例02 – 卖票



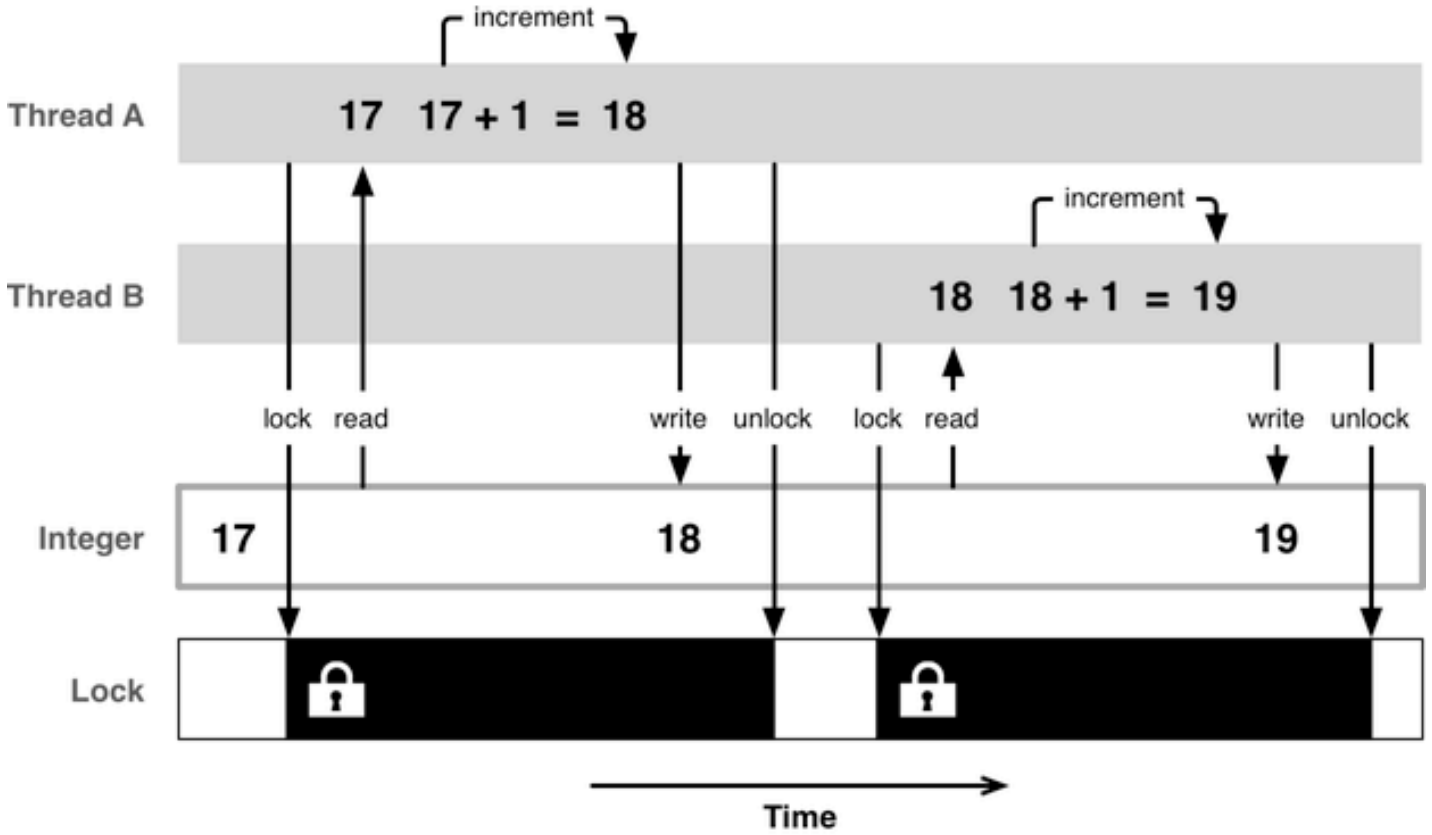
小码哥教育 SEEMYGO

多线程安全隐患分析



多线程安全隐患的解决方案

- 解决方案：使用线程同步技术（同步，就是协同步调，按预定的先后次序进行）
- 常见的线程同步技术是：加锁



小码哥教育 SEEMYGO

iOS中的线程同步方案

- OSSpinLock
- os_unfair_lock
- pthread_mutex
- dispatch_semaphore
- dispatch_queue(DISPATCH_QUEUE_SERIAL)
- NSLock
- NSRecursiveLock
- NSCondition
- NSConditionLock
- @synchronized

小码哥教育 SEEMYGO

GNUstep

- GNUstep是GNU计划的项目之一，它将Cocoa的OC库重新开源实现了一遍
- 源码地址：http://www.gnustep.org/resources/downloads.php
- 虽然GNUstep不是苹果官方源码，但还是具有一定的参考价值

OSSpinLock

- OSSpinLock叫做”自旋锁”，等待锁的线程会处于忙等（busy-wait）状态，一直占用着CPU资源
- 目前已经不再安全，可能会出现优先级反转问题
- 如果等待锁的线程优先级较高，它会一直占用着CPU资源，优先级低的线程就无法释放锁
- 需要导入头文件#import <libkern/OSAtomic.h>

```
// 初始化
OSSpinLock lock = OS_SPINLOCK_INIT;
// 尝试加锁（如果需要等待就不加锁，直接返回false；如果不需要等待就加锁，返回true）
bool result = OSSpinLockTry(&lock);
// 加锁
OSSpinLockLock(&lock);
// 解锁
OSSpinLockUnlock(&lock);
```

os_unfair_lock

- os_unfair_lock用于取代不安全的OSSpinLock ，从iOS10开始才支持
- 从底层调用看，等待os_unfair_lock锁的线程会处于休眠状态，并非忙等
- 需要导入头文件#import <os/lock.h>

```
// 初始化
os_unfair_lock lock = OS_UNFAIR_LOCK_INIT;
// 尝试加锁
os_unfair_lock_trylock(&lock);
// 加锁
os_unfair_lock_lock(&lock);
// 解锁
os_unfair_lock_unlock(&lock);
```

pthread_mutex

- mutex叫做”互斥锁”，等待锁的线程会处于休眠状态
- 需要导入头文件#import <pthread.h>

```
// 初始化锁的属性
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_NORMAL);
// 初始化锁
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, &attr);
// 尝试加锁
pthread_mutex_trylock(&mutex);
// 加锁
pthread_mutex_lock(&mutex);
// 解锁
pthread_mutex_unlock(&mutex);
// 销毁相关资源
pthread_mutexattr_destroy(&attr);
pthread_mutex_destroy(&mutex);
```

```
/*
 * Mutex type attributes
 */
#define PTHREAD_MUTEX_NORMAL        0
#define PTHREAD_MUTEX_ERRORCHECK    1
#define PTHREAD_MUTEX_RECURSIVE     2
#define PTHREAD_MUTEX_DEFAULT       PTHREAD_MUTEX_NORMAL
```

小码哥教育 SEEMYGO

pthread_mutex – 递归锁

```
// 初始化锁的属性
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
// 初始化锁
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, &attr);
```

小码哥教育 SEEMYGO

pthread_mutex – 条件

```
// 初始化锁
pthread_mutex_t mutex;
// NULL代表使用默认属性
pthread_mutex_init(&mutex, NULL);
// 初始化条件
pthread_cond_t condition;
pthread_cond_init(&condition, NULL);
// 等待条件（进入休眠，放开mutex锁；被唤醒后，会再次对mutex加锁）
pthread_cond_wait(&condition, &mutex);
// 激活一个等待该条件的线程
pthread_cond_signal(&condition);
// 激活所有等待该条件的线程
pthread_cond_broadcast(&condition);
// 销毁资源
pthread_mutex_destroy(&mutex);
pthread_cond_destroy(&condition);
```

NSLock、NSRecursiveLock

- NSLock是对mutex普通锁的封装

```
@interface NSLock : NSObject <NSLocking> {
- (BOOL)tryLock;
- (BOOL)lockBeforeDate:(NSDate *)limit;
@end
```

```
// 初始化锁
NSLock *lock = [[NSLock alloc] init];
```

```
@protocol NSLocking

- (void)lock;
- (void)unlock;

@end
```

- NSRecursiveLock也是对mutex递归锁的封装，API跟NSLock基本一致

小码哥教育 SEEMYGO

NSCondition

- NSCondition是对mutex和cond的封装

```
@interface NSCondition : NSObject <NSLocking> {
- (void)wait;
- (BOOL)waitUntilDate:(NSDate *)limit;
- (void)signal;
- (void)broadcast;
@end
```

小码哥教育 SEEMYGO

NSConditionLock

- NSConditionLock是对NSCondition的进一步封装，可以设置具体的条件值

```
@interface NSConditionLock : NSObject <NSLocking> {
- (instancetype)initWithCondition:(NSInteger)condition;
@property (readonly) NSInteger condition;
- (void)lockWhenCondition:(NSInteger)condition;
- (BOOL)tryLock;
- (BOOL)tryLockWhenCondition:(NSInteger)condition;
- (void)unlockWithCondition:(NSInteger)condition;
- (BOOL)lockBeforeDate:(NSDate *)limit;
- (BOOL)lockWhenCondition:(NSInteger)condition beforeDate:(NSDate *)limit;
@end
```

dispatch_semaphore

- semaphore叫做”信号量”
- 信号量的初始值，可以用来控制线程并发访问的最大数量
- 信号量的初始值为1，代表同时只允许1条线程访问资源，保证线程同步

```
// 信号量的初始值
int value = 1;
// 初始化信号量
dispatch_semaphore_t semaphore = dispatch_semaphore_create(value);
// 如果信号量的值<=0，当前线程就会进入休眠等待（直到信号量的值>0）
// 如果信号量的值>0，就减1，然后往下执行后面的代码
dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
// 让信号量的值加1
dispatch_semaphore_signal(semaphore);
```

小码哥教育 SEEMYGO

dispatch_queue

- 直接使用GCD的串行队列，也是可以实现线程同步的

```
dispatch_queue_t queue = dispatch_queue_create("lock_queue", DISPATCH_QUEUE_SERIAL);
dispatch_sync(queue, ^{
    // 任务
});
```

@synchronized

- @synchronized是对mutex递归锁的封装
- 源码查看：objc4中的objc-sync.mm文件
- @synchronized(obj)内部会生成obj对应的递归锁，然后进行加锁、解锁操作

```
@synchronized(obj) {
    // 任务
}
```

iOS线程同步方案性能比较

- 性能从高到低排序
  - os_unfair_lock
  - OSSpinLock
  - dispatch_semaphore
  - pthread_mutex
  - dispatch_queue(DISPATCH_QUEUE_SERIAL)
  - NSLock
  - NSCondition
  - pthread_mutex(recursive)
  - NSRecursiveLock
  - NSConditionLock
  - @synchronized

小码哥教育 SEEMYGO

自旋锁、互斥锁比较

- 什么情况使用自旋锁比较划算？
  - 预计线程等待锁的时间很短
  - 加锁的代码（临界区）经常被调用，但竞争情况很少发生
  - CPU资源不紧张
  - 多核处理器

- 什么情况使用互斥锁比较划算？
  - 预计线程等待锁的时间较长
  - 单核处理器
  - 临界区有IO操作
  - 临界区代码复杂或者循环量大
  - 临界区竞争非常激烈

小码哥教育 SEEMYGO

atomic

- atomic用于保证属性setter、getter的原子性操作，相当于在getter和setter内部加了线程同步的锁
- 可以参考源码objc4的objc-accessors.mm
- 它并不能保证使用属性的过程是线程安全的

iOS中的读写安全方案

- 思考如何实现以下场景
  - 同一时间，只能有1个线程进行写的操作
  - 同一时间，允许有多个线程进行读的操作
  - 同一时间，不允许既有写的操作，又有读的操作

- 上面的场景就是典型的“多读单写”，经常用于文件等数据的读写操作，iOS中的实现方案有
  - `pthread_rwlock`：读写锁
  - `dispatch_barrier_async`：异步栅栏调用

小码哥教育 SEEMYGO

pthread_rwlock

- 等待锁的线程会进入休眠

```
// 初始化锁
pthread_rwlock_t lock;
pthread_rwlock_init(&lock, NULL);
// 读-加锁
pthread_rwlock_rdlock(&lock);
// 读-尝试加锁
pthread_rwlock_tryrdlock(&lock);
// 写-加锁
pthread_rwlock_wrlock(&lock);
// 写-尝试加锁
pthread_rwlock_trywrlock(&lock);
// 解锁
pthread_rwlock_unlock(&lock);
// 销毁
pthread_rwlock_destroy(&lock);
```

dispatch_barrier_async

- 这个函数传入的并发队列必须是自己通过dispatch_queue_cretate创建的
- 如果传入的是一个串行或是一个全局的并发队列，那这个函数便等同于dispatch_async函数的效果

```
// 初始化队列
dispatch_queue_t queue = dispatch_queue_create("rw_queue", DISPATCH_QUEUE_CONCURRENT);

// 读
dispatch_async(queue, ^{

});

// 写
dispatch_barrier_async(queue, ^{

});
```

小码哥教育
SEEMYGO