


iOS 底层原理之多线程 - 面壁者 LOGIC - CSDN 博客

 版权声明：不积跬步无以至千里，不积小流无以成江海！

https://blog.csdn.net/Bolted_snail/article/details/83062651

文章目录

[简介](#)

[进阶](#)

[GCD](#)

[多线程的安全隐患](#)

[多线程安全隐患的解决方案](#)

[iOS 中的线程同步方案](#)

[1.OSSpinLock](#)

[2. os_unfair_lock](#)

[3. pthread_mutex](#)

[pthread_mutex – 递归锁](#)

[pthread_mutex – 条件](#)

[4. NSLock、NSRecursiveLock、NSCondition、
NSConditionLock](#)

[NSLock](#)

[NSRecursiveLock](#)

[NSCondition](#)

[NSConditionLock](#)

[5.dispatch_semaphore](#)

[6.dispatch_queue](#)

[7.NSOperationQueue](#)

[8.@synchronized](#)

[iOS 线程同步方案性能比较](#)

[自旋锁、互斥锁比较](#)

[atomic](#)

[iOS 中的读写安全方案](#)

[1. pthread_rwlock 读写锁](#)

[2. dispatch_barrier_async 异步栅栏调用](#)

[面试题](#)

简介

在前面已经写过几篇文章详细的介绍了 pthread、NSThread、GCD、NSOperation 等：

[iOS 多线程简述](#)

[iOS 多线程 - pthread、NSThread](#)

[iOS 多线程 - GCD](#)

[iOS 多线程 - NSOperation, NSOperationQueue](#)

进阶

GCD

- 0GCD 的常用函数：

GCD 中有 2 个用来执行任务的函数

1. 用同步的方式执行任务

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

2. 用异步的方式执行任务

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

queue: 队列

block: 任务

- 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

1. 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

2. 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

1. 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

2. 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

- 发生死锁的前提条件: 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)。

- 特殊实例:

1. 下面代码打印顺序

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSLog(@"1---%@", [NSThread currentThread]);
        dispatch_sync(dispatch_get_global_queue(0, 0), ^{
            NSLog(@"2---%@", [NSThread currentThread]);
            dispatch_queue_t queue = dispatch_queue_create("serial", DISPATCH_
```

```

        dispatch_async(queue, ^{
            NSLog(@"3---%@", [NSThread currentThread]);
        });
    });
    NSLog(@"4---%@", [NSThread currentThread]);
}
return 0;
}

```

结果为 1, 2, 4, 3。并且在 3 的时候开启了子线程。异步开启了子线程，串行执行是指如果在 queue 这个串行队列中添加多个任务，里面的任务会串行执行，与外面无关，所以并不是 1, 2, 3, 4。

2. 下面打印结果，会不会产生死锁：

```

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        dispatch_queue_t queue = dispatch_queue_create("serial", DISPATCH_QUEUE_SERIAL);
        dispatch_async(queue, ^{
            NSLog(@"1");
            dispatch_sync(queue, ^{
                NSLog(@"2");
            });
        });
        NSLog(@"3");
    }
    return 0;
}

```

打印结果为 3, 1，会产生死锁，所以 2 打印不了。

3. 下面代码打印结果

```

- (void)viewDidLoad {
    [super viewDidLoad];
    dispatch_async(dispatch_get_global_queue(0, 0), ^{
        NSLog(@"1");
        // 这句代码的本质是往RunLoop中添加定时器
        [self performSelector:@selector(test) withObject:nil afterDelay:.0];
        // [self performSelector:@selector(test) withObject:nil]; //本周是me
        NSLog(@"3");
        // [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode beforeDate:[N

```

```

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

```

打印结果为 1, 3, 因为 `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, 所以无法执行 test 方法, 将 runloop 启动即可。

4. 下面代码打印结果

```

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

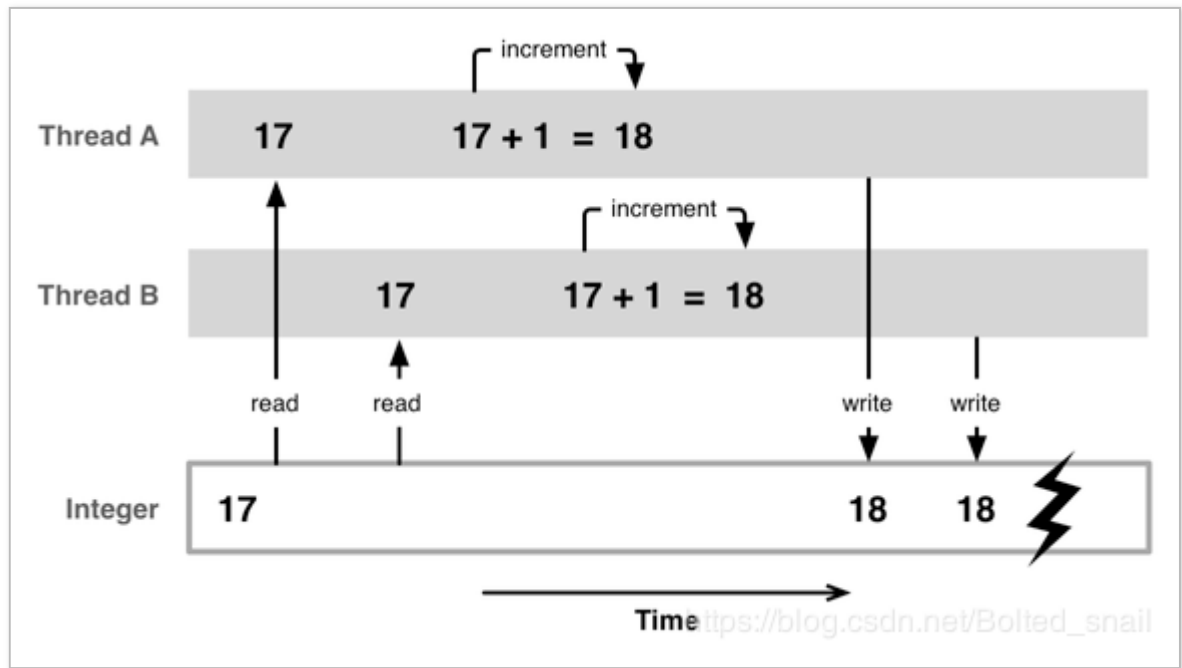
//        [[NSRunLoop currentRunLoop] addPort:[NSPort alloc] init] forMode:NSDefa
//        [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode beforeDate:[NSD
    ]];
    [thread start];
    [self performSelector:@selector(test) onThread:thread withObject:nil waitUntilDon
}
- (void)test
{
    NSLog(@"2");
}

```

打印结构为 1, 因为线程启动后会执行以下 block 中的方法, 执行完后就销毁了, 因为没有启动 runloop。

多线程的安全隐患

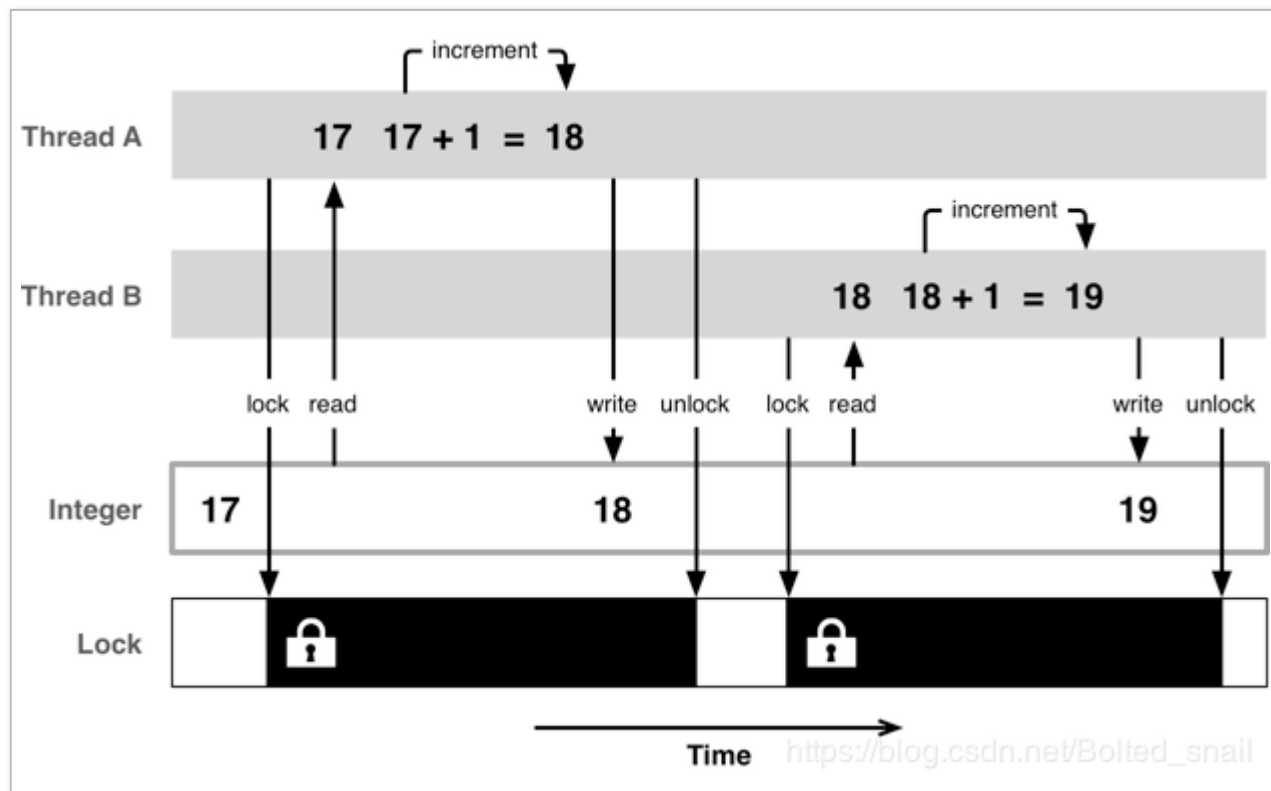
- 资源共享, 一块资源可能会被多个线程共享, 也就是 **多个线程可能会访问同一块资源** (比如多个线程访问同一个对象、同一个变量、同一个文件), 当多个线程访问同一块资源时, 很容易引发 **数据错乱和数据安全** 问题。



分析：公共资源初始值为 17，Thread A，Thread B 这时候都给公共资源加 1，但是都拿到的值是 17，加 1 返回的结果都是 18，而正确的值应该是 19 才对。

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。



分析：当 Thread A 要访问操作公共资源时，先给加锁，锁住这一块公共资源，这时候别的线程就必须等待；等 Thread A 操作完成后，将这一块资源解锁，这时候 Thread B 才能进来操作公共资源，B 也要给公共资源加锁，操作完成后解锁，以此类推。

主要：给公共资源加的锁必须是同一把锁。

- 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。

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

1.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>`，起使用步骤如下

```
//初始化锁
OSSpinLock lock = OS_SPINLOCK_INIT;
//尝试枷锁（如果需要等待就不要加锁，直接返回返回false，如果不需要等到就枷锁，返回true）
bool result = OSSpinLockTry(&lock);
// 加锁
OSSpinLockLock(&lock);
// 解锁
OSSpinLockUnlock(&lock);
```

示例 1：卖票

```
#import "ViewController.h"
#import <libkern/OSAtomic.h>
@interface ViewController ()
@property (assign, nonatomic) int ticketsCount;
@property (assign, nonatomic) OSSpinLock lock;
@end
```

```
@implementation ViewController
```

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // 初始化锁
    self.lock = OS_SPINLOCK_INIT;
    [self ticketTest];
}
```

```
/*
 异步并发卖票演示
*/
```

```
- (void)ticketTest
{
```



```

self.ticketsCount = 15;

dispatch_queue_t queue = dispatch_get_global_queue(0, 0);

dispatch_async(queue, ^{
    for (int i = 0; i < 5; i++) {
        [self saleTicket];
    }
});

dispatch_async(queue, ^{
    for (int i = 0; i < 5; i++) {
        [self saleTicket];
    }
});

dispatch_async(queue, ^{
    for (int i = 0; i < 5; i++) {
        [self saleTicket];
    }
});

/**
 卖1张票
 */
- (void)saleTicket
{
    //    if (OSSpinLockTry(&_lock)) {
    //        int oldTicketsCount = self.ticketsCount;
    //        sleep(.2);
    //        oldTicketsCount--;
    //        self.ticketsCount = oldTicketsCount;
    //        NSLog(@"还剩%d张票 - %@", oldTicketsCount, [NSThread currentThread]);
    //
    //        OSSpinLockUnlock(&_lock);
    //    }

    // 加锁
    OSSpinLockLock(&_lock);

    int oldTicketsCount = self.ticketsCount;
    sleep(.2);
    oldTicketsCount--;
    self.ticketsCount = oldTicketsCount;
    NSLog(@"还剩%d张票 - %@", oldTicketsCount, [NSThread currentThread]);

    // 解锁

```

```
    OSSpinLockUnlock(&_lock);  
}
```

示例 2. 存钱取钱

```
//前面代码一样省略  
/**  
    存钱、取钱演示  
    */  
- (void)moneyTest  
{  
    self.money = 100;  
  
    dispatch_queue_t queue = dispatch_get_global_queue(0, 0);  
  
    dispatch_async(queue, ^{  
        for (int i = 0; i < 10; i++) {  
            [self saveMoney]; //存钱  
        }  
    });  
  
    dispatch_async(queue, ^{  
        for (int i = 0; i < 10; i++) {  
            [self drawMoney]; //取钱  
        }  
    });  
}  
  
/**  
    存钱  
    */  
- (void)saveMoney  
{  
    // 加锁  
    OSSpinLockLock(&_lock1);  
  
    int oldMoney = self.money;  
    sleep(.2);  
    oldMoney += 50;  
    self.money = oldMoney;  
    NSLog(@"存50, 还剩%d元 - %@", oldMoney, [NSThread currentThread]);  
    // 解锁  
    OSSpinLockUnlock(&_lock1);  
}  
  
/**
```

```

取钱
*/
- (void)drawMoney
{
    // 加锁
    OSSpinLockLock(&_lock1);

    int oldMoney = self.money;
    sleep(.2);
    oldMoney -= 20;
    self.money = oldMoney;

    NSLog(@"取20, 还剩%d元 - %@", oldMoney, [NSThread currentThread]);
    // 解锁
    OSSpinLockUnlock(&_lock1);
}

```

注意：不管是买票还存钱取钱，只要同时访问的是同一块资源就要，加要加同一把锁，也就说锁变量是一个全局变量或静态变量，只初始化一次。

2. 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>**，使用类似于 **OSSpinLock**，这里不再举例。

```

//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);

```

3. 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>**。

```

// 初始化属性
pthread_mutexattr_t attr;

```

```

pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_DEFAULT); //默认模式
// 初始化锁
pthread_mutex_t mutex1;
pthread_mutex_init(&mutex1, &attr);
//尝试加锁
pthread_mutex_trylock(&mutex1);
//加锁
pthread_mutex_lock(&mutex1);
//解锁
pthread_mutex_unlock(&mutex1);
// 销毁属性
pthread_mutexattr_destroy(&attr);
// 销毁锁
pthread_mutex_destroy(&mutex1);

//属性相关模式
#define PTHREAD_MUTEX_NORMAL 0
#define PTHREAD_MUTEX_ERRORCHECK 1
#define PTHREAD_MUTEX_RECURSIVE 2 //递归模式
#define PTHREAD_MUTEX_DEFAULT PTHREAD_MUTEX_NORMAL //默认模式

```

- 示例:

```

#import "MutexDemo.h"
#import <pthread.h>

@interface MutexDemo()
@property (assign, nonatomic) pthread_mutex_t ticketMutex;
@property (assign, nonatomic) pthread_mutex_t moneyMutex;
@end

@implementation MutexDemo

- (instancetype)init
{
    if (self = [super init]) {
        [self __initMutex:&ticketMutex];
        [self __initMutex:&moneyMutex];
    }
    return self;
}

- (void)__initMutex:(pthread_mutex_t *)mutex
{
    // 静态初始化

```

```

// pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
// 初始化属性
// pthread_mutexattr_t attr;
// pthread_mutexattr_init(&attr);
// pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_DEFAULT); //默认模式
// 初始化锁
// pthread_mutex_init(mutex, &attr);
pthread_mutex_init(mutex, NULL); //传NULL就是默认模式
}

//卖票
- (void)__saleTicket
{
    //加锁
    pthread_mutex_lock(&_ticketMutex);
    [super __saleTicket];
    //解锁
    pthread_mutex_unlock(&_ticketMutex);
}

//存钱
- (void)__saveMoney
{
    pthread_mutex_lock(&_moneyMutex);
    [super __saveMoney];
    pthread_mutex_unlock(&_moneyMutex);
}

//取钱
- (void)__drawMoney
{
    pthread_mutex_lock(&_moneyMutex);
    [super __drawMoney];
    pthread_mutex_unlock(&_moneyMutex);
}

//销毁属性和锁
- (void)dealloc
{
    pthread_mutex_destroy(&_moneyMutex);
    pthread_mutex_destroy(&_ticketMutex);
}

@end

```

pthread_mutex – 递归锁

有时候我们可能会对一块资源递归访问操作，如果用上面默认类型的互斥锁就会产生死锁的情况，这时候用递归锁就可以解决，只需要把属性类型设置为 **PTHREAD_MUTEX_RECURSIVE**。

- 示例：死锁

```
#import "MutexDemo2.h"
#import <pthread.h>
@interface MutexDemo2()
@property (assign, nonatomic) pthread_mutex_t mutex;
@end
@implementation MutexDemo2
- (instancetype)init
{
    if (self = [super init]) {
        [self __initMutex:&_mutex];
    }
    return self;
}
- (void)__initMutex:(pthread_mutex_t *)mutex
{
    // 递归锁：允许同一个线程对一把锁进行重复加锁

    // 初始化属性
    pthread_mutexattr_t attr;
    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_DEFAULT);
    // 初始化锁
    pthread_mutex_init(mutex, &attr);
    // 销毁属性
    pthread_mutexattr_destroy(&attr);
}
- (void)otherTest
{
    pthread_mutex_lock(&_mutex);
    NSLog(@"%s", __func__);
    static int count = 0;
    if (count < 10) {
        count++;
        [self otherTest];
    }
    pthread_mutex_unlock(&_mutex);
}
- (void)dealloc
{
    pthread_mutex_destroy(&_mutex);
}
@end
```

分析：上面 `otherTest` 方法第一次进来会加锁，然后递归调自己，这时候第二次进来又要加锁，但这时候第一次加的锁还没有打开就加不了锁，也就是说第一次的锁没有打开，第二次又要加锁是加不上的，第二次线程就一直处在水面状态，就一直卡在那。

解决方案：只需将属性类型设置为递归锁， `pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);`

pthread_mutex – 条件

- 有时候我们需要在 **A** 线程做完一部分事情后去 **B** 线程做事情，等 **B** 做完事情后又回到 **A** 做事情（生产者 - 消费者模式），这时候就需要用到 `pthread_mutex – 条件`，也称为条件锁。

//初始化锁

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
//初始化条件
pthread_cond_t condition;
pthread_cond_init(&condition, NULL);
//等待条件（线程进入休眠，放开mutex锁，被唤醒后会再次mutex加锁）
pthread_cond_wait(&condition, &mutex);
//激活一个等待条件的线程（发信号）
pthread_cond_signal(&condition);
//激活所有等待条件的线程（发广播）
pthread_cond_broadcast(&condition);
//销毁线程
pthread_mutex_destroy(&mutex);
//销毁条件
pthread_cond_destroy(&condition);
```

- 示例：

```
#import "MutexDemo3.h"
#import <pthread.h>
@interface MutexDemo3()
@property (assign, nonatomic) pthread_mutex_t mutex;
@property (assign, nonatomic) pthread_cond_t cond;
@property (strong, nonatomic) NSMutableArray *data;
@end
@implementation MutexDemo3
- (instancetype)init
{
    if (self = [super init]) {
        // 初始化属性
        pthread_mutexattr_t attr;
```

登录后复制

```

    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
    // 初始化锁
    pthread_mutex_init(&mutex, &attr);
    // 销毁属性
    pthread_mutexattr_destroy(&attr);

    // 初始化条件
    pthread_cond_init(&cond, NULL);

    self.data = [NSMutableArray array];
}
return self;
}
- (void)otherTest
{
    //开启两个子线程执行删除和添加元素操作
    [[NSThread alloc] initWithTarget:self selector:@selector(__remove) object:nil] s
    [[NSThread alloc] initWithTarget:self selector:@selector(__add) object:nil] s
}
// 生产者-消费者模式
// 线程1
// 删除数组中的元素
- (void)__remove
{
    pthread_mutex_lock(&mutex);
    NSLog(@"__remove - begin");

    if (self.data.count == 0) {
        // 等待，这时候就会解锁，线程2就可以加锁执行其添加操作
        pthread_cond_wait(&cond, &mutex);
    }

    [self.data removeLastObject];
    NSLog(@"删除了元素");

    pthread_mutex_unlock(&mutex);
}
// 线程2
// 往数组中添加元素
- (void)__add
{
    NSLog(@"__add - begin");
    pthread_mutex_lock(&mutex);
    sleep(5);
    [self.data addObject:@"Test"];
    NSLog(@"添加了元素");
    // 发信号激活线程1的等待条件

```



```

    pthread_cond_signal(&_cond);
    // 广播
//    pthread_cond_broadcast(&_cond);
    //当线程2的锁解开后，线程1就会加锁，执行等待条件后面的代码
    pthread_mutex_unlock(&_mutex);
}
- (void)dealloc
{
    pthread_mutex_destroy(&_mutex);
    pthread_cond_destroy(&_cond);
}
@end

```

结果及分析：不管哪个子线程先执行，“添加元素”总是先于“删除元素自行”。

注意：`pthread_cond_signal(&_cond);`当这句代码执行发射信号激活子线程 1 的条件时，并不是立马就去在线程 1 中去加锁，而是等到线程 2 中的锁解开后采取线程 1 中加锁，也就是说信号发出后，线程 1 一直在等待线程 2 的解锁。

4. NSLock、NSRecursiveLock、NSCondition、NSConditionLock

NSLock

- **NSLock** 是对 **mutex** 普通锁的封装;

```

//尝试加锁
- (BOOL)tryLock;
//尝试在某个日期之前加锁
- (BOOL)lockBeforeDate:(NSDate *)limit;
//加锁解锁的操作在 NSLocking协议里面
@protocol NSLocking
//加锁
- (void)lock;
//解锁
- (void)unlock;
@end

```

示例：

```

#import "NSLockDemo.h"

@interface NSLockDemo()
@property (strong, nonatomic) NSLock *ticketLock;

```

```

@property (strong, nonatomic) NSLock *moneyLock;
@end

@implementation NSLockDemo
- (instancetype)init
{
    if (self = [super init]) {
        //初始化锁
        self.ticketLock = [[NSLock alloc] init];
        self.moneyLock = [[NSLock alloc] init];
    }
    return self;
}

//卖票
- (void)__saleTicket
{
    [self.ticketLock lock];

    [super __saleTicket];

    [self.ticketLock unlock];
}

//存钱
- (void)__saveMoney
{
    [self.moneyLock lock];
    [super __saveMoney];
    [self.moneyLock unlock];
}

//取钱
- (void)__drawMoney
{
    [self.moneyLock lock];
    [super __drawMoney];
    [self.moneyLock unlock];
}
@end

```

NSRecursiveLock

- **NSRecursiveLock** 也是对 **mutex递归锁** 的封装，API 跟 **NSLock** 基本一致；
- **NSRecursiveLock** 示例：

```

#import "NSRecursiveLockDemo.h"
@interface NSRecursiveLockDemo()
@property (strong, nonatomic) NSRecursiveLock *lock;
@end
@implementation NSRecursiveLockDemo
- (instancetype)init
{
    if (self = [super init]) {
        //初始化锁
        self.lock = [[NSRecursiveLock alloc]init];
    }
    return self;
}
//递归锁
-(void)otherTest{
    [self.lock lock];
    static int count = 0;
    if (count < 10) {
        NSLog(@"%d",count);
        count++;
        [self otherTest];
    }
    [self.lock unlock];
}
@end

```

NSCondition

- **NSCondition** 是对 **mutex** 和 **cond** 的封装。

```

//条件等待
- (void)wait;
//尝试在某个日子加锁
- (BOOL)waitUntilDate:(NSDate *)limit;
//发射信号激活条件
- (void)signal;
//发射广播激活条件
- (void)broadcast;

```

- 示例：

```

#import "NSConditionDemo.h"
@interface NSConditionDemo()
@property (strong, nonatomic) NSCondition *condition;
@property (strong, nonatomic) NSMutableArray *data;
@end
@implementation NSConditionDemo
- (instancetype)init
{
    if (self = [super init]) {
        //初始化锁和条件
        self.condition = [[NSCondition alloc] init];
        self.data = [NSMutableArray array];
    }
    return self;
}
- (void)otherTest
{
    [[NSThread alloc] initWithTarget:self selector:@selector(__remove) object:nil] s
    [[NSThread alloc] initWithTarget:self selector:@selector(__add) object:nil] s
}
// 生产者-消费者模式
// 线程1
// 删除数组中的元素
- (void)__remove
{
    [self.condition lock];
    NSLog(@"__remove - begin");
    if (self.data.count == 0) {
        // 等待
        [self.condition wait];
    }
    [self.data removeLastObject];
    NSLog(@"删除了元素");
    [self.condition unlock];
}
// 线程2
// 往数组中添加元素
- (void)__add
{
    [self.condition lock];
    sleep(1);
    [self.data addObject:@"Test"];
    NSLog(@"添加了元素");
    // 发射信号激活条件
    [self.condition signal];
    // 广播
    //[self.condition broadcast];
}

```

```
        [self.condition unlock];
    }
@end
```

NSConditionLock

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

```
//初始化
- (instancetype)initWithCondition:(NSInteger)condition NS_DESIGNATED_INITIALIZER;
//获取条件
@property (readonly) NSInteger condition;
//在条件值为几的时候加锁
- (void)lockWhenCondition:(NSInteger)condition;
//尝试加锁
- (BOOL)tryLock;
//尝试在某个条件值时加锁
- (BOOL)tryLockWhenCondition:(NSInteger)condition;
//解锁并重写设置锁条件
- (void)unlockWithCondition:(NSInteger)condition;
//尝试在某个日期前加锁
- (BOOL)lockBeforeDate:(NSDate *)limit;
//尝试在某个条件下某个日期前加锁
- (BOOL)lockWhenCondition:(NSInteger)condition beforeDate:(NSDate *)limit;
```

- 示例：

```
#import "NSConditionLockDemo.h"
@interface NSConditionLockDemo()
@property (strong, nonatomic) NSConditionLock *conditionLock;
@end
@implementation NSConditionLockDemo
- (instancetype)init
{
    if (self = [super init]) {
        //初始化锁和条件，并设置条件为1
        self.conditionLock = [[NSConditionLock alloc] initWithCondition:1];
        //默认条件为0
        self.conditionLock = [[NSConditionLock alloc] init];
    }
    return self;
}
```

```

- (void)otherTest
{
    //创建三个线程执行三个任务
    [[NSThread alloc] initWithTarget:self selector:@selector(__one) object:nil] s
    [[NSThread alloc] initWithTarget:self selector:@selector(__two) object:nil] s
    [[NSThread alloc] initWithTarget:self selector:@selector(__three) object:nil]
}
- (void)__one
{
    [self.conditionLock lock];
    //在条件为1时加锁
    // [self.conditionLock lockWhenCondition:1];
    NSLog(@"__one");
    sleep(1);
    //解开当前所，并设置条件为为2
    [self.conditionLock unlockWithCondition:2];
}
- (void)__two
{
    //在条件为2时加锁
    [self.conditionLock lockWhenCondition:2];
    NSLog(@"__two");
    sleep(1);
    //解开当前所，并设置条件为3
    [self.conditionLock unlockWithCondition:3];
}
- (void)__three
{
    //在条件为3时加锁
    [self.conditionLock lockWhenCondition:3];
    NSLog(@"__three");
    //解锁
    [self.conditionLock unlock];
}
@end

```

结果及分析：不管开启的三条子线程执行任务的顺序是什么都会按 **__one** 、 **__two** 、 **three** 的顺序打印，因为当三天线程开启执行任务时，因为初始化的条件为 1，但是没有找到就会去执行 **lock** 方法来加锁，然后依次执行到 **unlockWithCondition**：解锁并重新设置条件，就会给对应条件下的代码加锁并执行里面的代码。（也就是说会个加个条件标签，通过判断标签为几就去加锁并执行里面的代码，如果不是对应的标签都需要睡眠等待）。

5.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。

```

//初始化信号量
dispatch_semaphore_create(long value);
    // 如果信号量的值 > 0, 就让信号量的值减1, 然后继续往下执行代码
// 如果信号量的值 <= 0, 就会休眠等待, 直到信号量的值变成>0, 就让信号量的值减1, 然后继续往下执行代码
dispatch_semaphore_wait(dispatch_semaphore_t dsema, dispatch_time_t timeout);
// 让信号量的值+1
dispatch_semaphore_signal(dispatch_semaphore_t dsema);

```

- 示例:

```

#import "SemaphoreDemo.h"
@interface SemaphoreDemo()
@property (strong, nonatomic) dispatch_semaphore_t semaphore;
@property (strong, nonatomic) dispatch_semaphore_t ticketSemaphore;
@property (strong, nonatomic) dispatch_semaphore_t moneySemaphore;
@end
@implementation SemaphoreDemo
- (instancetype)init
{
    if (self = [super init]) {
        //初始化, 最大并发值为5
        self.semaphore = dispatch_semaphore_create(5);
        self.ticketSemaphore = dispatch_semaphore_create(1);
        self.moneySemaphore = dispatch_semaphore_create(1);
    }
    return self;
}
//取钱
- (void)__drawMoney
{
    dispatch_semaphore_wait(self.moneySemaphore, DISPATCH_TIME_FOREVER);
    [super __drawMoney];
    dispatch_semaphore_signal(self.moneySemaphore);
}
//存钱
- (void)__saveMoney
{
    dispatch_semaphore_wait(self.moneySemaphore, DISPATCH_TIME_FOREVER);
    [super __saveMoney];
    dispatch_semaphore_signal(self.moneySemaphore);
}
//卖票
- (void)__saleTicket
{
    dispatch_semaphore_wait(self.ticketSemaphore, DISPATCH_TIME_FOREVER);

```

```

        [super __saleTicket];
        dispatch_semaphore_signal(self.ticketSemaphore);
    }
    - (void)otherTest
    {
        //开启20条线程
        for (int i = 0; i < 20; i++) {
            [[NSThread alloc] initWithTarget:self selector:@selector(test) object:nil];
        }
    }
    //信号量设置为5，所以最多是5条线程同时执行任务
    - (void)test
    {
        // 如果信号量的值 > 0，就让信号量的值减1，然后继续往下执行代码
        // 如果信号量的值 <= 0，就会休眠等待，直到信号量的值变成>0，就让信号量的值减1，然后继续往下执行
        dispatch_semaphore_wait(self.semaphore, DISPATCH_TIME_FOREVER);
        sleep(2);
        NSLog(@"test - %@", [NSThread currentThread]);
        // 让信号量的值+1
        dispatch_semaphore_signal(self.semaphore);
    }
@end

```

6.dispatch_queue

- 直接使用 GCD 的串行队列，也是可以实现线程同步的。

```

//创建串行队列
dispatch_queue_t queue = dispatch_queue_create("lockQueue", DISPATCH_QUEUE_SERIAL)
//执行队列中的任务
dispatch_sync(queue, ^{
    //执行任务
});

```

示例：

```

#import "SerialQueueDemo.h"
@interface SerialQueueDemo()
@property (strong, nonatomic) dispatch_queue_t ticketQueue;
@property (strong, nonatomic) dispatch_queue_t moneyQueue;
@end
@implementation SerialQueueDemo
- (instancetype)init
{

```



```

    if (self = [super init]) {
        self.ticketQueue = dispatch_queue_create("ticketQueue", DISPATCH_QUEUE_SER
        self.moneyQueue = dispatch_queue_create("moneyQueue", DISPATCH_QUEUE_SERIA
    }
    return self;
}
//取钱
- (void)__drawMoney
{
    dispatch_sync(self.moneyQueue, ^{
        [super __drawMoney];
    });
}
//存钱
- (void)__saveMoney
{
    dispatch_sync(self.moneyQueue, ^{
        [super __saveMoney];
    });
}
//卖票
- (void)__saleTicket
{
    dispatch_sync(self.ticketQueue, ^{
        [super __saleTicket];
    });
}
@end

```

注意，这里也可以用异步执行函数 `dispatch_async`，只要保证要访问的同一块资源的任务放在同一个串行队列中就可以按顺序同步执行了。

7.NSOperationQueue

- 可以通过设置 `NSOperationQueue` 的最大并发数来实现线程同步。

```

//初始化
NSOperationQueue * queue = [[NSOperationQueue alloc] init];
//设置最大并发数
queue.maxConcurrentOperationCount = 1;
//添加操作并执行队列中的任务
[queue addOperationWithBlock:^(
//任务
}]);

```

- 示例：

```
#import "NSOperationQueueDemo.h"
@interface NSOperationQueueDemo()
@property(strong, nonatomic)NSOperationQueue * moneyQueue;
@property(strong, nonatomic)NSOperationQueue * ticketQueue;
@end
@implementation NSOperationQueueDemo
-(instancetype)init{
    if (self = [super init]) {
        self.moneyQueue = [[NSOperationQueue alloc]init];
        self.moneyQueue.maxConcurrentOperationCount = 1;
        self.ticketQueue = [[NSOperationQueue alloc]init];
        self.ticketQueue.maxConcurrentOperationCount = 1;
    }
    return self;
}
- (void)__saveMoney{

    [self.moneyQueue addOperationWithBlock:^(
        [super __saveMoney];
    )];

}
- (void)__drawMoney{
    [self.moneyQueue addOperationWithBlock:^(
        [super __drawMoney];
    )];
}
- (void)__saleTicket{
    [self.ticketQueue addOperationWithBlock:^(
        [super __saleTicket];
    )];
}
@end
```

开启了线程，但是认为是穿行执行的，这样就保证的线程同步。

注意：访问同一块资源的的任务要添加到同一个队列中。

8.@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  
}
```

- 示例:

```
#import "SynchronizedDemo.h"  
@implementation SynchronizedDemo  
//取钱  
- (void)__drawMoney  
{  
    @synchronized([self class]) {  
        [super __drawMoney];  
    }  
}  
//存钱  
- (void)__saveMoney  
{  
    @synchronized([self class]) { // objc_sync_enter  
        [super __saveMoney];  
    } // objc_sync_exit  
}  
//卖票  
- (void)__saleTicket  
{  
    static NSObject *lock;  
    static dispatch_once_t onceToken;  
    dispatch_once(&onceToken, ^{  
        lock = [[NSObject alloc] init];  
    });  
  
    @synchronized(lock) {  
        [super __saleTicket];  
    }  
}  
//递归锁  
- (void)otherTest  
{  
    @synchronized([self class]) {  
        static int a = 0;  
        if (a < 10) {  
            a++;  
            NSLog(@"%d",a);  
            [self otherTest];  
        }  
    }  
}
```

```
}  
}  
@end
```

注意：传进取的 `objc` 可以是任意的实例对象和类对象，访问统一块资源必须使用同一把锁（窜进去的必须是同一个对象）。

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)  
NSOperationQueue(queue.maxConcurrentOperationCount = 1)  
NSLock  
NSCondition  
pthread_mutex(recursive)  
NSRecursiveLock  
NSConditionLock  
@synchronized
```

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；

临界区有 IO（读写）操作；

临界区代码复杂或者循环量大；

临界区竞争非常激烈；
单核处理器。

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。

```
- (void)setName:(NSString *)name
{
    // 加锁
    _name = name;
    // 解锁
}
```

- 为什么说 `atomic` 不能保证使用属性的过程是线程安全的？
例如一颗可变数组 `NSMutableArray`，在调用 `setter` 、 `getter` 是线程安全的，但是在添加 `addObject` 或删除 `removeObject` 元素时并没有添加锁，所以不能保证在使用的时候是线程安全的。

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

1. pthread_rwlock 读写锁

等待锁的线程会进入休眠。需导入 `#import <pthread.h>`。

```

//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);

```

- 示例

```

#import "ViewController.h"
#import <pthread.h>
@interface ViewController ()
@property (assign, nonatomic) pthread_rwlock_t lock;
@end
@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    // 初始化锁
    pthread_rwlock_init(&_amp;lock, NULL);
    dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
    for (int i = 0; i < 100; i++) {
        dispatch_async(queue, ^{
            [self read];
        });
        dispatch_async(queue, ^{
            [self write];
        });
    }
}

//读
- (void)read {
    pthread_rwlock_rdlock(&_amp;lock);
    sleep(1);
    NSLog(@"%s---%@", __func__, [NSThread currentThread]);
}

```

```

        pthread_rwlock_unlock(&_lock);
    }
    //写
    - (void)write
    {
        pthread_rwlock_wrlock(&_lock);
        sleep(1);
        NSLog(@"%s--%@", __func__, [NSThread currentThread]);
        pthread_rwlock_unlock(&_lock);
    }
    - (void)dealloc
    {
        pthread_rwlock_destroy(&_lock);
    }
@end

```

写总是单个打印，读可能是多个同时打印，注意并发对弈必须是同一个队列。

2. dispatch_barrier_async 异步栅栏调用

- 这个函数传入的并发队列必须是自己通过 **dispatch_queue_create** 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。

```

//手动创建并发队列
dispatch_queue_t queue = dispatch_queue_create("rw_queue", DISPATCH_QUEUE_CONCURRE
//读操作
dispatch_async(queue, ^{
    //执行读任务
});
//写操作
dispatch_barrier_async(queue, ^{
    //执行写任务
});

```

- 示例：

```

#import "ViewController.h"
@interface ViewController ()
@end
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    dispatch_queue_t queue = dispatch_queue_create("rw_queue", DISPATCH_QUEUE_CONC

```

```

    for (int i = 0; i < 10; i++) {
        dispatch_async(queue, ^{
            [self read];
        });

        dispatch_async(queue, ^{
            [self read];
        });

        dispatch_async(queue, ^{
            [self read];
        });

        dispatch_barrier_async(queue, ^{
            [self write];
        });
        dispatch_barrier_async(queue, ^{
            [self write];
        });
    }
}

- (void)read {
    sleep(2);
    static int a = 0;
    a++;
    NSLog(@"read:%d次",a);
}

- (void)write
{
    sleep(2);
    static int a = 0;
    a++;
    NSLog(@"write:%d次",a);
}

@end

```

不管是读写锁还是栅栏函数，在读的时候是可以多线程并发执行，但在写的时候只能同步执行。

面试题

- 你理解的多线程？

开启多条线程同时执行不同的任务，可以提升任务执行的效率，把耗时操作放到子线程执行，可以让主线程

- iOS 的多线程方案有哪几种？你更倾向于哪一种？
`pthread`、`NSThread`、`GCD`、`NSOperation`，更倾向于后面两种，因为使用简单，系统自己管理内存。
- 你在项目中用过 GCD 吗？
用过，比如 GCD 定时器，GCD 处理网络请求数据。
- GCD 的队列类型
串行队列，并发队列，主队列。
- 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？
互斥锁，自旋锁，GCD 可以设置信号量位 1，串行队列，OperationQueue 可以设置最大并发锁为 1 等等都可以达到线程同步，从而保证了线程安全。
- 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！
- 所有示例源码：

[线程同步方案](#)

[读写安全](#)

全文完

本文由 简悦 SimpRead 优化，用以提升阅读体验。