帐号与登录那点事

起底多线程同步锁(iOS)

by SPRINGOX on 2016 年 01 月 14 日 · LEAVE A COMMENT · in 技术

iOS/MacOS为多线程、共享内存(变量)提供了多种的同步解决方案(即同步锁)，对于这些方案的比较，大都讨论了锁的用法以及锁操作的开销，然后就开销表现排个序。春哥以为，最优方案的选用还是看应用场景，高频接口PK低频接口、有限冲突PK激烈竞争、代码片段耗时的长短，以上都是正确选用的重要依据，不同方案在其适用范围表现各有不同。这些方案当中，除了熟悉的iOS/MacOS系统自有的同步锁，另外还有两个自研的读写锁，还有应用开发中常见的set/get访问接口的原子操作属性。

1、@synchronized(){}

Objective-C同步语法能够实现对block内的代码片段加锁， 可以指定任意一个Objective-C对象(id指针)作为锁“标记”，该语法将“标记”理解为token；

2、NSLock、NSRecursiveLock：

典型的面向对象的锁，即同步锁类，遵循Objective-C的NSLocking协议接口，前者支持tryLock，后者支持递归(可重入)；

3、NSCondition、NSConditionLock：

基于信号量方式实现的锁对象，前者提供单独的信号量管理接口，相比后者用法上可以更为灵活，而后者在接口上更为直接、实用；

4、ANReadWriteLock、ANRecursiveRWLock：

iOS/MacOS并没有提供读写锁，春哥尝试自己搞，Objective-C版的读写锁(ANLock)，遵循读写锁特性，前者写锁耗时较小，后者支持递归；

5、pthread_mutex：

POSIX标准的unix多线程库(pthread)中使用的互斥量，支持递归，需要特别说明的是信号机制pthread_cond_wait()同步方式也是依赖于该互斥量，pthread_cond_wait()本身并不具备同步能力；

6、dispatch_semaphore：

GCD用于控制多线程并发的信号量，允许通过wait/signal的信号事件控制并发执行的最大线程数，当最大线程数降级为1的时候则可当作同步锁使用，注意该信号量并不支持递归；

7、OSSpinLock：

iOS/MacOS自有的自旋锁，其特点是线程等待取锁时不进内核，线程因此不挂起，直接保持空转，这使得它的锁操作开销降得很低，OSSpinLock是不支持递归的；

8、atomic(property) set/get：

利用set/get接口的属性实现原子操作，进而确保“被共享”的变量在多线程中读写安全，这已经是能满足部分多线程同步要求；

基础表现-锁操作耗时：

近期文章

起底多线程同步锁(iOS)
帐号与登录那点事
攻城师的交互设计
漫谈互联网移动化
Cocoa开发优化之道
Objective-C与Runtime
流媒体服务与编码学习小结
重新定义NSUserDefault—— ANKeyValue

链接表

Fun In GitHub
极客程序员:华君

华君

十年果粉
青春有悔
一个山寨程序猿的成长故事（下）
一个山寨程序猿的成长故事（上）
我们究竟是为了维护正义，还是为了吃长粮?
经济学和人生观：用数学算给你看
你开始用Mac了么?
FIT团队收费APP第二枪：内置FIT输入法新浪微博客户端-FIT™随享
凭什么是我们
对不起，这次不是免费的——“FIT写字板”AppStore发布

SPRINGOX

SpringOx 微博主页

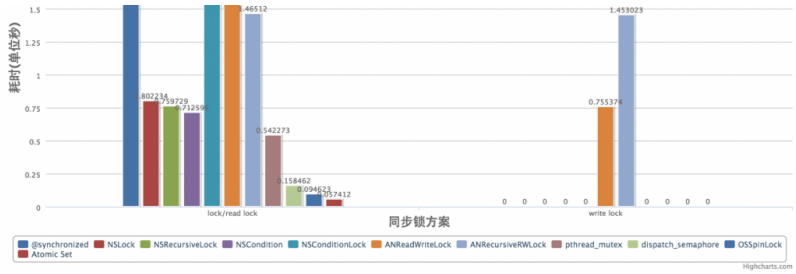
我录的这段怎么样？春晚吉祥物“猴赛雷”暴走 你经历过绝望吗？#腾讯小视频8秒也精彩#一起来看 http://t.cn/RbBu8TH (来自@腾讯视频)
1月30日 20:30

转，扩散

iOS大全：《@SpringOx：Objective-C 与 Runtime》消息派发是Objective-C函数调用的模式，而消息派发和动态绑定的实现机制为Runtime，但是Runtime并不仅仅为消息派发和动态绑定而work，它也是Objective-C面向对象、内存模型等特性的实现者。http://t.cn/RyJ84hi [图片]
9月27日 20:20

目测李荣浩有望接棒纠结伦呐，台湾小子加油，顶
9月24日 13:33

#中国好声音周杰伦#你见证了我的青春，我陪你惊艳这个夏天。金罐加多宝#中国好声音#为你而来，



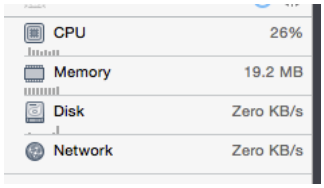
上图是常规的锁操作性能测试(iOS7.0SDK，iPhone6模拟器，Yosemite 10.10.5)，垂直方向表示耗时，单位是秒，总耗时越小越好，水平方向表示不同类型锁的锁操作，具体又分为两部分，左边的常规lock操作(比如NSLock)或者读read操作(比如ANReadWriteLock)，右边则是写write操作，图上仅有ANReadWriteLock和ANRecursiveRWLock支持，其它不支持的则默认为0，图上看出，单从性能表现，原子操作是表现最佳的(0.057412秒)，@synchronized则是最耗时的(1.753565秒) (测试代码)

正如前文所述，不同方案各有侧重，适用于不用的场景，不能唯性能论高低：

原子操作虽然性能很好，但仅限于set/get，比如对列表的插入移除操作需要做同步则无能为力，支持不到，所以适用于一些实例成员变量的读写同步；

得益于不进内核不挂起的方式，OSSpinLock有着优异的性能表现，然而在高并发执行(冲突概率大，竞争激烈)的时候，又或者代码片段比较耗时(比如涉及内核执行文件io、socket、thread等)，就容易引发CPU占有率暴涨的风险，因此更适用于一些简短低耗时的代码片段；

```
//主线程中
__block OSSpinLock spinlock = OS_SPINLOCK_INIT;

//线程1
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    OSSpinLockLock(&spinlock);
    [self threadMethod1];
    sleep(10);
    OSSpinLockUnlock(&spinlock);
});

for (int i=0; i<10; i++) {
    //线程2
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        sleep(1);
        OSSpinLockLock(&spinlock);
        [self threadMethod2];
        OSSpinLockUnlock(&spinlock);
    });
}
```



上图为OSSpinLock等待取锁时的耗时测试用例代码，下图为测试结果，图中可以看到，等待取锁时，如果异步线程比较耗时，CPU占有率会有一个飙升 (测试代码)

dispatch_semaphore的性能表现出乎意料之外的好，也没有OSSpinLock的CPU占有率暴涨的问题，然而原本是用于GCD的多线程并发控制，也是信号量机制，是否适用于常规同步锁有待实践验证，春哥这里仅提供选择，不做推荐；

```
- (void)test_dispatch_semaphore
{
    //主线程中
    _semaphore = dispatch_semaphore_create(1);

    //线程1
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
```

周董我们宣你！

7月18日 11:23

赞！ 分享单曲http://t.cn/R2dFSlJ（@网易云音乐）

6月24日 21:49

分类目录

产品
技术
综合

Search

```
209         dispatch_semaphore_wait(_semaphore, DISPATCH_TIME_FOREVER);
210         [self threadMethod1];
211         sleep(10);
212         dispatch_semaphore_signal(_semaphore);
213     });
214
215     //线程2
216     dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
217         sleep(1);
218         dispatch_semaphore_wait(_semaphore, DISPATCH_TIME_FOREVER);
219         [self threadMethod2];
220         dispatch_semaphore_signal(_semaphore);
221     });
222 }
```

上图为dispatch_semaphore测试用例

pthread_mutex是pthread经典的基于互斥量机制的同步锁，特性、性能以及稳定各方面都已被大量项目所验证，也是春哥比较推荐作为常规同步锁首选；

```
112     //pthread_mutex
113     pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
114     /*
115      pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
116      pthread_mutexattr_t attr;
117      pthread_mutexattr_init(&attr);
118      //设置锁的属性为可递归
119      pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
120      pthread_mutex_init(&mutex, &attr);
121      pthread_mutexattr_destroy(&attr);
122      */
123     timeBefore = CFAbsoluteTimeGetCurrent();
124     for(i=0; i<count; i++){
125         pthread_mutex_lock(&mutex);
126         pthread_mutex_unlock(&mutex);
127     }
128     timeCurrent = CFAbsoluteTimeGetCurrent();
129     printf("pthread_mutex used : %f\n", timeCurrent-timeBefore);
```

上图为pthread_mutex用法举例

读写锁的在锁操作耗时上明显不占优势，读写锁的主要性能优势在于多线程高并发量的场景，这时候锁竞争可能会非常激烈，使用一般的锁这时候并发性能都会明显下降，读写锁对于所有读操作能够把同步放开，进而保持并发性能不受影响；以pthread_mutex和ANRecursiveRWLock为例，假设mutex的lock耗时为lk，则rw的read lock耗时为2.7lk(从性能测试图表数据得出)，read操作耗时为rd，1000次的多线程接口访问：

mutex总耗时 = 1000*lk + 1000*rd

rw总耗时 = 1000*2.7*lk + 1000/c*rd

其中c表示应用的并发数，根据开发文档和技术资料，iOS第二条线程起stack为512KB，而单个应用useable memory size在50MB以内，即c<=100；

假设线程数取中值c=50(严格来说，线程数不等于冲突计数，冲突计数很可能会比线程数小得多，线程同步运行不代表就即刻会发生冲突)，当 mutex总耗时 > rw总耗时：

mutex总耗时 > rw总耗时 =》 50*lk + 50*rd > 50*2.7lk + rd =》 49*rd > 85*lk =》 rd > 1.73*lk

可以看出，只要read操作耗时超过锁操作耗时的1.7倍(这其实很容易达到的)，读写锁的性能就会占优势

假设线程数c=2(如上述，这里是假设了两个线程之间是竞争了，发生冲突，实际未必)：

mutex总耗时 > rw总耗时 =》 2*lk + 2*rd > 5.4*lk + rd =》 rd > 3.4lk

即使只有两个并发线程，只要read操作耗时超过锁操作耗时的3.4倍，读写锁的性能还会占优势

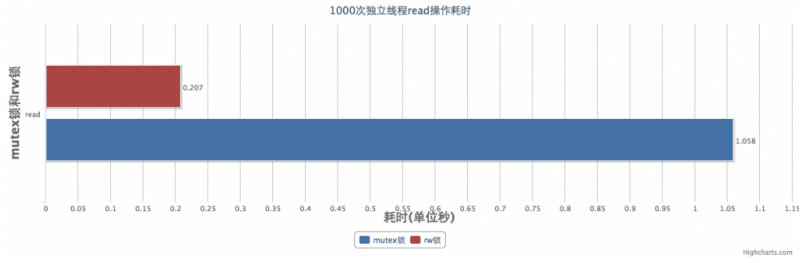
假设线程数c=1：

mutex总耗时 > rw总耗时 =》 0 > 1.7lk

这显然不成立，说明当单个线程的时候，rw的性能不可能有优势。这也好理解，这时候的mutex和rw的读操作都相当完全同步，不论是mutex还是rw，性能完全取决于锁操作本身，而rw在锁操作耗时上就不占优势，所以mutex总耗时总是要小于rw总耗时的

```
- (void)testMutexReadPerformance
{
    pthread_mutexattr_t attr;
    pthread_mutexattr_init(&attr);
    //设置锁的属性为可递归
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_NORMAL);
    pthread_mutex_init(&_mutex, &attr);
    pthread_mutexattr_destroy(&attr);

    int count = 1000;
    for (int i=0; i<count; i++) {
        [NSThread detachNewThreadSelector:@selector(mutexReadTest)
                                 toTarget:self
                               withObject:nil];
    }
}

- (void)mutexReadTest
{
    pthread_mutex_lock(&_mutex);
    usleep(100);
    NSLog(@"%@ flag %d",NSStringFromSelector(_cmd), _readWriteFlag);
    pthread_mutex_unlock(&_mutex);
}
```

```
- (void)testRWReadPerformance
{
    _rwLock = [[ANRecursiveRWLock alloc] init];

    int count = 1000;
    for (int i=0; i<count; i++) {
        [NSThread detachNewThreadSelector:@selector(rwReadTest)
                                 toTarget:self
                               withObject:nil];
    }
}

- (void)rwReadTest
{
    [_rwLock readLock];
    usleep(100);
    NSLog(@"%@ flag %d",NSStringFromSelector(_cmd), _readWriteFlag);
    [_rwLock readUnlock];
}
```



上图是mutex锁和rw锁read操作的耗时测试用例，下图为测试结果，read操作设置为100微秒，mutex锁的总耗时是rw锁的5倍多，read操作的耗时远比锁操作大许多(2k倍)，根据上述恒等式计算可以得出实际的冲突计数c=5 (测试代码)

其它方案的讨论：

a、NSCondition和NSConditionLock实际使用的性能表现并任何优势，然而条件锁的意义在于对信号量做了面向对象封装；

b、NSLock和NSRecursiveLock在性能表现上与mutex算比较接近，用法上也并无二致，因此，常规情况，NSRecursiveLock和mutex之间的选择，春哥以为更多是习惯和偏好的问题；

c、@synchronized似乎是这些方案当中性能表现最不佳的，那是不是应该完全抛弃呢？春哥倒不这么认为，@synchronized最大的特点在于“快捷”，同步语法仅仅需要一个对象(id指针)作为互斥量，而且还不限于实例对象，类对象也能够支持，这就使得类方法中做同步变得简单不少，block用法也使得代码更紧凑，内存管理更稳健，非常适合一些低频而又不得不同步的

逻辑，比如单例初始化、启动加载等等；

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；

原创文章，禁止转载[握手]

Tagged with: NSCondition • NSLock • NSRecursiveLock • OSSpinLock • pthread_mutex • synchronized • 同步锁 • 性能 • 耗时

(pthread)中使用的互斥量，支持递归，需要特别说明的是信号机制pthread_cond_wait()同步方式也是依赖于该互斥量，pthread_cond_wait()本身并不具备同步能力； [...]