

读写锁的实现与应用

1753112欧阳桥梁

.NET体系结构与设计开发

2020.5.28

目录

[读写锁的理解 1](#_Toc41562539)

[读写锁的实现思路 1](#_Toc41562540)

[使用的原语 1](#_Toc41562541)

[实现思路 1](#_Toc41562542)

[调度策略 1](#_Toc41562543)

[代码结构 2](#_Toc41562544)

[读写锁的测试与应用 2](#_Toc41562545)

[优缺点与进一步优化 4](#_Toc41562546)

# 读写锁的理解

读写锁就是为了优化**读远远大于写**场景而创建出来的一种机制。与传统锁不同的是读写锁的规则是可以共享读，但只能一个写，即：读读不互斥，读写互斥，写写互斥，而一般的独占锁是：读读互斥，读写互斥，写写互斥，

一般情况下独占锁的效率低主要原因是高并发下临界区的激烈竞争导致线程上下文切换。但当并发不是很高的情况下，读写锁由于需要额外维护读锁的状态，可能还不如独占锁的效率高。因此读写锁适用的场景是读操作远远大于写，需要根据实际情况选择使用。

# 读写锁的实现

### 使用的原语

* Interlocked互锁函数族
* AutoRestEvent，ManualResetEvent事件
* Mutex互斥锁（用来测试对比）

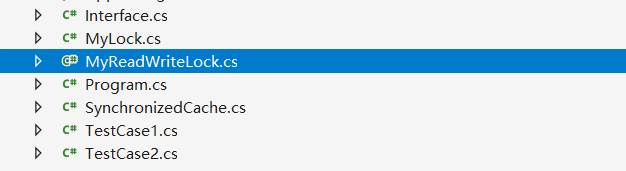
### 实现思路

1. **读写锁本身临界区的保护**：使用互锁函数族（Interlocked）实现的自旋锁（SpinLock）对读写锁本身临界区的保护。除此之外，使用互斥锁（Mutex）实现临界区保护做性能测试对比。
2. **读者/写者等待与唤醒**：使用事件（AutoRestEvent，ManualResetEvent）让读者/写者等待，避免忙等。事件触发来唤醒等待中的写者和读者

### 调度策略

1. **读者获取锁**：读者到来时，判断是否已有写者或等待中的写者，如果没有写者，则直接进入读模式，读计数+1。 如果有，则进入睡眠模式，等待所有写者完成工作，以此避免写饥饿。
2. **写者获取锁**：写者到来时,关闭读事件，后续读者均无法获取读锁，避免**写饥饿**。之后判断是否已有读者或写者正在工作，如果有，则写等待计数+1，写者进入睡眠模式。如果没有，则获取写锁，写者工作。
3. **读者释放锁**：释放锁时，判断自己是否是最后一个读者，如果还有其他读者正在读，则读计数-1，直接退出。如果是最后一位读者，就判断是否有等待中写者，有则触发写事件唤醒一位写者（AutoResetEvent）。
4. **写者释放锁**：写者释放锁时，判断是否有写者等待，如果有，触发写事件唤醒一位写者。如果没有等待的写者，就开启读事件（ManualResetEvent），唤醒等待的读者。
5. **注意**：事件触发仅仅用于唤醒等待的线程，以此避免忙等。但并不代表被唤醒后，线程一定能获取读写锁。线程被唤醒后，需要继续根据上述调度策略竞争读写锁。

### 代码结构



Interface.cs: 读写锁的接口，方便比较不同实现方式的读写锁

MyLock.cs: 保护读写锁临界区，给出了两种实现方式Mutex实现，自旋锁实现

MyReadWriteLock.cs：读写锁实现

Program.cs：程序入口处，执行测试函数

SynchronizedCache.cs: 使用读写锁实现了一个简单的同步缓存

TestCase1.cs:测试用例1，使用同步缓存测试读写锁正确性

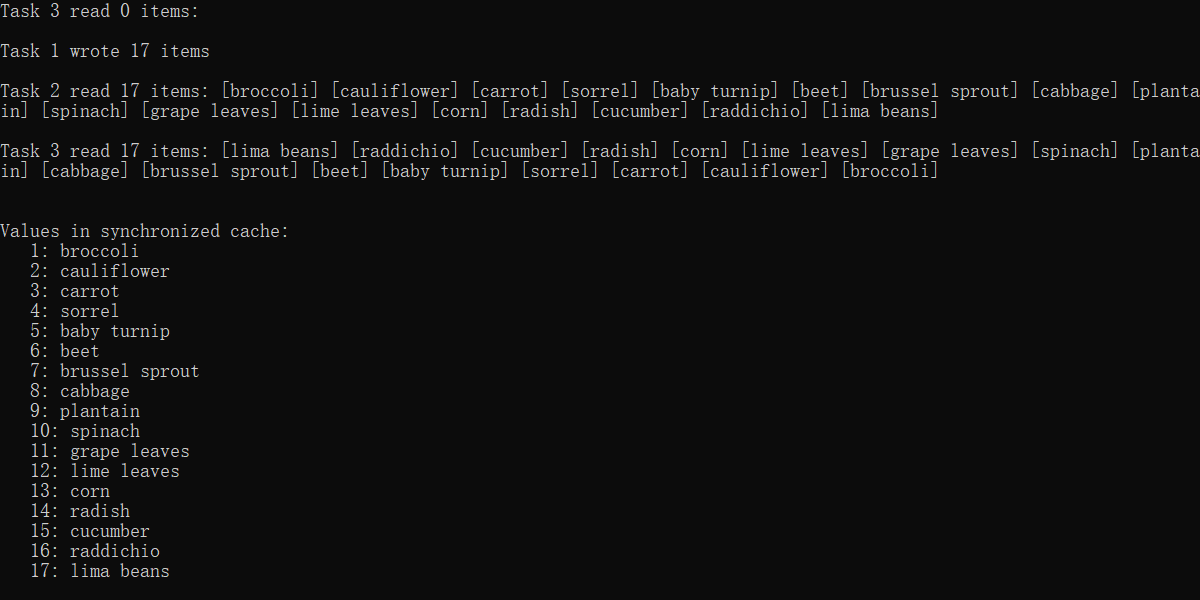
TestCase2.cs：测试用例2，比较不同实现方式的读写锁，测试其性能

# 读写锁的测试与应用

为了测试读写锁的正确性和性能，编写了两个测试用例。

TestCase1 使用读写锁实现了一个简单的**同步缓存**（SynchronizedCache），其中包含带有整数键的字符串。 使用读写锁同步多线程对用作内部缓存的Dictionary <TKey，TValue>的访问。该同步缓存包括添加记录，删除记录以及查询记录的简单方法。之后模拟多个线程对同步缓存进行读写，测试读写锁的正确性。

代码运行结果如下



可以发现线程输出结果符合预期，并未出现漏读，跳读，多读等错误，初步验证读写锁的正确性。

TestCase2 开启40个读线程，4个写线程对共享变量进行访问。写线程对变量进行加一操作，读线程则每次读取变量，判断变量是否达到计数结果，如果达到则退出线程。使用读写锁实现对共享变量的同步访问。对写者等待，读者等待，任务完成时间进行测量。

使用TestCase2对ReaderWriterLockSlim（C#官方实现），MyReadWriteLock（自旋锁实现），Mutex\_MyReadWriteLock（Mutex实现）三种读写锁实现进行测试比较，测试的结果如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| 计数\时间ms | ReaderWriterLockSlim | MyReadWriteLock | Mutex\_MyReadWriteLock |
| 100000 | 101 | 115 | 980 |
| 1000000 | 743 | 824 | 9328 |
| 10000000 | 5037 | 6316 | - |

可以发现使用Mutex实现读写锁性能最差，原因在于Mutex需要频繁的从用户态切换到内核态，十分浪费时间。由于临界区操作代码短执行快，读写锁临界区冲突可能性较小， 因此自旋锁可以减少用户态到内核态的开销，极大的提升性能。此外，本自旋锁在自旋达到最大次数后，会主动放弃线程执行，避免忙等。

对比ReaderWriterLockSlim 与 本读写锁，ReaderWriterLockSlim性能大概较本实现提升10%。参考C# ReaderWriterLockSlim 源码实现，可以发现，其读者写者冲突时，并不是立即让线程睡眠，而是先自旋一定次数， 每次自旋中持续竞争读写锁。直到达到最大自旋次数还未获取读写锁，才让线程睡眠等待唤醒事件。因此可以从这个方面进一步去优化读写锁。

# 优缺点与进一步优化

1. 不可**重入**

重入是指获取读写锁的线程重复获取读写锁，常见在递归程序。在本读写锁实现中，读者写者不可重复获取读写锁，不然会导致死锁

1. 不支持**锁升级**和**锁降级**

锁升级指的是线程获取到了读锁，在没有释放读锁的前提下，又获取写锁。锁降级指的是线程获取到了写锁，在没有释放写锁的情况下，又获取读锁。本读锁实现不支持锁升级锁降级。写者必须释放写锁才能获取读锁，读者也必须释放读锁才能获取写锁，不然会导致死锁。

1. **自旋锁**实现临界区保护

由于临界区操作代码短执行快，读写锁临界区冲突可能性较小，采用自旋锁实现能有效提升性能。相较于Mutex可以减少用户态到内核态的开销，以及线程上下文切换的浪费。此外，本自旋锁在自旋达到最大次数后，会主动放弃线程执行，避免忙等。

1. 进一步优化：

参考C# ReaderWriterLockSlim 源码实现，可以发现，其读者写者冲突时，并不是立即让线程睡眠，而是先自旋一定次数， 每次自旋中持续竞争读写锁。直到达到最大自旋次数还未获取读写锁，才让线程睡眠等待唤醒事件。 因此可以从这个方面进一步去优化读写锁。