读写锁的实现

需求分析

- 支持多个并发的写线程
- 可以支持多进程进行写访问（串行）
- 写线程具有具有更高的优先级，无**写饥饿**

实现

接口

- **AcquireReaderLock** & **ReleaseReaderLock** (配套使用)
- **AcquireWriterLock** & **ReleaseWriterLock** (配套使用)
- **AcquireWriterLock**(int timeout) & **AcquireWriterLock**(int timeout)
    - 同时读写锁的获取都有对应的**计时重载版本**，当超时便会放弃获取

实现细节

读锁

使用 Monitor 进行互斥，对于读锁获取而言如下：

```
public void AcquireLock() {
    Monitor.Enter(cond);
    //判断是否满足条件
    while(writeWaitingCnt > 0 || writing) {
        Monitor.Wait(cond);
    }
    // 更新状态
    readingCnt ++;
    Monitor.Exit(cond);
}
```

读锁获取锁成功的条件如下：

- 没有写者正在写
- 没有写者等待

读锁的释放如下：

```
public void ReleaseReaderLock() {
    Monitor.Enter(cond);

    // 更新状态，读者退出
    readingCnt --;
    Monitor.PulseAll(cond);// 唤醒所有人
    Monitor.Exit(cond);
}
```

写锁

获取死锁成功的条件如下：

- 没有写者正在写
- 没有读者正在读

风格与读锁类似，都是

- 用 Monitor 保护互斥区
- 检查是否满足获取锁的条件
    - 满足则获取，更新条件
    - 不满足则调用 wait 进入睡眠，等待唤醒

代码如下：

```
// 没有读者或写者占用锁时，才可以获取 lock
        public void AcquireWriterLock() {
            Monitor.Enter(cond);
            // 这样写者才不会饥饿
            writeWaitingCnt ++; // 当前有写者正在等待
            // 这样后续的读线程会注意到有写者正在等待，就不会一直读

            while(readingCnt > 0 || writing) {
                Monitor.Wait(cond);
            }

            writeWaitingCnt--; // 结束等待
            writing = true; // 接下来轮到当前写者了
            Monitor.Exit(cond);
        }
```

这里一个很重要的点是**避免写者饥饿**，写者饥饿出现的原因是：

- 有读者在读
- 接连不断的读者到来继续读

我们姑且称这种情况为**无限读者**，这样一来写者很可能会长期无法获取锁，我采用的方法是多了一个标志位 `writeWaitingCnt` ，该标志位表示正在等待的写者数量。注意到这一行代码：

```
writeWaitingCnt ++; // 当前有写者正在等待
```

在写者进入互斥区会递增这个变量，这样当出现上述情况（无限读者）时，后续的读线程会注意到有写者正在等待（下面代码为读锁获取的条件判断），就会调用 wait 进入睡眠：

```
// 摘自读锁的获取部分
while(writeWaitingCnt > 0 || writing) {
    Monitor.Wait(cond);
}
```

计时机制

为了方便测试，我编写了**超时打断锁获取**的机制。超时机制的实现思想如下：

- 开启一个 Task 尝试获取锁
- 主线程进入睡眠，在用户规定的 timeout(ms) 后醒来，如果还没完成，则抛出异常，告知用户，用户可自己处理异常。

这里为了复用代码，我采用了课堂上实践过的 delegate：

```
private delegate void AqLock();
```

该 AqLock 表示锁获取，可以是读锁也可以是写锁。这部分是读锁和死锁公用的逻辑：

```
// ------------ timeouts helper --------------- //
private delegate void AqLock();
private static void AcquireLockTimeoutBase(int timeout, AqLock aqLock) {
    //
    CancellationTokenSource tokenSource = new CancellationTokenSource();
    // CancellationToken token=new CancellationToken();

    bool isdone = false;
    // 直接调 timer 异常不是在 main thread 抛出的
    // 无法捕捉
    // var timer = new Timer(statusChecker.checkDone, null, timeout,
Timeout.Infinite);
    Task.Factory.StartNew(() =>
                          {
                              aqLock();
                              isdone = true;
                          },
                          tokenSource.Token);
    //
    Thread.Sleep(timeout);
    if(!isdone) {
        // 超时取消任务
        // 抛异常
        tokenSource.Cancel();
        throw new ApplicationException();
    }
}
```

这里一个比较微妙的点是，**异常应当在主线程抛出而不是放在新开的 Task 中抛出**，因为如果在 Task 抛出该异常无法捕获。

读写锁超时接口实现

读写锁只要简单包装 `AcquireLockTimeoutBase` 即可：

```
// ------------- timeouts version -------------- //
public void AcquireWriterLock(int timeout) {
    AcquireLockTimeoutBase(timeout, AcquireWriterLock);
}

public void AcquireReaderLock(int timeout) {
    AcquireLockTimeoutBase(timeout, AcquireReaderLock);
}
```

测试

针对读写锁，我编写了几个测试用例。

Test case 1

测试读写锁的互斥性是否能够满足，主要是开启多个写进程，看是否能达到互斥：

```
public static void BaseTest() {
    //
    Console.WriteLine("============== BaseTest ==============");
    rWLock = new RWLock();
    resource = 0;

    Thread[] t = new Thread[numThreads+1];
    t[0] = new Thread(new ThreadStart(ReaderProc)); t[0].Start();
    //
    for (int i = 1; i <= numThreads; i++){
        t[i] = new Thread(new ThreadStart(WriterProc));
        t[i].Start();
    }
    for(int i=0;i<numThreads+1;i++) t[i].Join();
    Console.WriteLine("Expected: {0}, Actual: {1}\n", numThreads*1000,
resource);
}
private static void ReaderProc() {
    // 读 100 次
    for(int i=0;i<500;i++) {
        rWLock.AcquireReaderLock();
        // Thread.Sleep(2);
        rWLock.ReleaseReaderLock();
    }
}

private static void WriterProc() {
    for(int i=0;i<1000;i++) {
        rWLock.AcquireWriterLock();
        resource ++;
        rWLock.ReleaseWriterLock();
    }
}
```

测试结果（多次）：

```
============== BaseTest ==============
Expected: 26000, Actual: 26000s
```

可以看出互斥性是可以保证的。

Test case 2

这个测试用例主要是为了验证是否可以避免读者饥饿，开启多读一写，并为写者设置超时参数：

```
// TEST CASE 2
// 多读者，看是否会发生写饥饿
// 当写者写完后，让读者也终止，程序结束
// 考虑到实际，这里开启了多个写进程并发写
// 同时开启一个写线程，测试是否写线程会超过等待时间，即是否会发生写饥饿

// TODO: 实践中发现这种 多读一写 效率很低
// 感觉主要是为了避免写饥饿，过于限制读线程
// 多个线程被唤醒后继续 sleep
```

```
public static void InfiniteReader() {
    rWLock = new RWLock();
    resource = 0;

    Console.WriteLine("============== InfiniteReader ==============");
    Thread[] t = new Thread[numThreads+1];
    // 读者开始
    int readerThread = 5;
    for (int i = 1; i <= readerThread; i++){
        t[i] = new Thread(new ThreadStart(ReaderProc));
        t[i].Start();
    }
    // 写者开始
    t[0] = new Thread(new ThreadStart(WriterProcWithTimeout));
    t[0].Start();
    for(int i=0;i<readerThread+1;i++) t[i].Join();
    Console.WriteLine("Expected: {0}, Actual: {1}", 1*50, resource);
}

private static void WriterProcWithTimeout() {
    int timeout = 500;
    for(int i=0;i<50;i++) {
        // 不要超时
        try {
            rWLock.AcquireWriterLock(timeout);
            resource++;
            rWLock.ReleaseWriterLock();
        }
        catch (ApplicationException) {
            // 超时
            Console.WriteLine("Hungry Writer TIMEOUT {0}", timeout);
        }
        // try block end
    }
}
```

测试结果（多次）：

```
============== InfiniteReader ==============
Expected: 50, Actual: 50
```

这个测试用例耗时较长，经过分析，我认为主要是：

- 锁粒度过大，用一把大锁保护，唤醒时只能暴力地唤醒所有线程
- 为了避免死饥饿，**当有写者在等待后，读者线程被唤醒后又进入 wait 等待**，其实是十分耗时地

Test Case 3

我尝试修改了官网 的部分示例代码，做了简单的性能测试，因为我们的读写锁只实现了标准库的部分接口，故不能很好进行比较，然而可以比较明显地看出自己的读写锁实现性能比较一般， Test case 2 已经分析过，这里不再赘述。

```
public static void PMain()
{
    Console.WriteLine("============== PerformanceTest ==============");
    // 开启多线程，性能测试入口
```

```
    Thread[] t = new Thread[numThreads];
    for (int i = 0; i < numThreads; i++){
        t[i] = new Thread(new ThreadStart(PerformanceThreadProc));
        t[i].Name = new String(Convert.ToChar(i + 65), 1);
        t[i].Start();
        if (i >= numThreads/2)
            Thread.Sleep(300);
    }

    // 设置标志位，终止还在 run 的读写线程
    running = false;
    for (int i = 0; i < numThreads; i++)
        t[i].Join();

    // Display
    Console.WriteLine("\n{0} reads, {1} writes, {2} reader time-outs, {3} writer
time-outs.",
                      reads, writes, readerTimeouts, writerTimeouts);
    Console.Write("Press ENTER to exit... ");
    Console.ReadLine();
}
```