

```
// 在这里执行其他一些代码...
Monitor.Exit(t);
}
```

在上述代码中，执行 **SomeMethod** 的线程调用 **Monitor.Enter**，获取由 **Transaction** 对象公开的锁。线程池线程查询 **LastTransaction** 属性时，这个属性也调用 **Monitor.Enter** 来获取同一个锁，造成线程池线程阻塞，直到执行 **SomeMethod** 的线程调用 **Monitor.Exit**。有调试器可发现线程池线程在 **LastTransaction** 属性内部阻塞。但很难判断是另外哪个线程拥有锁。即使真的弄清楚了是哪个线程拥有锁，还必须弄清楚是什么代码造成它取得锁，这就更难了。更糟的是，即使历经千辛万苦，终于搞清楚了是什么代码造成线程取得锁，最后却发现那些代码不在你的控制范围之内，或者无法修改它们来修正问题。因此，我的建议是始终坚持使用私有锁。下面展示了如何修正 **Transaction** 类：

```
internal sealed class Transaction {
    private readonly Object m_lock = new Object(); // 现在每个 Transaction 对象都有私有锁
    private DateTime m_timeOfLastTrans;

    public void PerformTransaction() {
        Monitor.Enter(m_lock); // 进入私有锁
        // 以下代码拥有对数据的独占访问权...
        m_timeOfLastTrans = DateTime.Now;
        Monitor.Exit(m_lock); // 退出私有锁
    }

    public DateTime LastTransaction {
        get {
            Monitor.Enter(m_lock); // 进入私有锁
            // 以下代码拥有对数据的独占访问权...
            DateTime temp = m_timeOfLastTrans;
            Monitor.Exit(m_lock); // 退出私有锁
            return temp;
        }
    }
}
```

如果 **Transaction** 的成员是静态的，只需将 **m_lock** 字段也变成静态字段，即可确保静态成员的线程安全性。

通过以上讨论，一个很明显的结论是：**Monitor** 根本就不该实现成静态类；它应该像其他所有同步构造那样实现。也就是说，应该是一个可以实例化并在上面调用实例方法的类。事实上，正因为 **Monitor** 被设计成一个静态类，所以它还存在其他许多问题。下面对这些额外的问题进行了总结。

- 变量能引用一个代理对象——前提是变量引用的那个对象的类型派生自 **System.MarshalByRefObject** 类(参见第 22 章“CLR 寄宿和 AppDomain”)。调用 **Monitor** 的方法时，传递对代理对象的引用，锁定的是代理对象而不是代理引用的实际对象。
- 如果线程调用 **Monitor.Enter**，向它传递对类型对象的引用，而且这个类型对象是以

“AppDomain 中立”的方式加载的^①，线程就会跨越进程中的所有 AppDomain 在那个类型上获取锁。这是 CLR 一个已知的 bug，它破坏了 AppDomain 本应提供的隔离能力。这个 bug 很难在保证高性能的前提下修复，所以它一直没有修复。我的建议是，永远都不要向 **Monitor** 的方法传递类型对象引用。

- 由于字符串可以留用(参见 14.2.2 节“字符串是不可变的”), 所以两个完全独立的代码段可能在不知情的情况下获取对内存中的一个 **String** 对象的引用。如果将这个 **String** 对象引用传给 **Monitor** 的方法, 两个独立的代码段现在就会在不知情的情况下以同步方式执行^②。
- 跨越 AppDomain 边界传递字符串时, CLR 不创建字符串的副本; 相反, 它只是将对字符串的一个引用传给其他 AppDomain。这增强了性能, 理论上也是可行的, 因为 **String** 对象本来就不可变(不可修改)。但和其他所有对象一样, **String** 对象关联了一个同步块索引, 这个索引是可变的(可修改), 使不同 AppDomain 中的线程在不知情的情况下开始同步。这是 CLR 的 AppDomain 隔离存在的另一个 bug。我的建议是永远不要将 **String** 引用传给 **Monitor** 的方法。
- 由于 **Monitor** 的方法要获取一个 **Object**, 所以传递值类型会导致值类型被装箱, 造成线程在已装箱对象上获取锁。每次调用 **Monitor.Enter** 都会在一个完全不同的对象上获取锁, 造成完全无法实现线程同步。
- 向方法应用 **[MethodImpl(MethodImplOptions.Synchronized)]**特性, 会造成 JIT 编译器用 **Monitor.Enter** 和 **Monitor.Exit** 调用包围方法的本机代码。如果方法是实例方法, 会将 **this** 传给 **Monitor** 的这些方法, 锁定隐式公共的锁。如果方法是静态的, 对类型的类型对象的引用会传给这些方法, 造成锁定“AppDomain 中立”的类型。我的建议是永远不要使用这个特性。
- 调用类型的类型构造器时(参见 8.3 节“类型构造器”), CLR 要获取类型对象上的一个锁, 确保只有一个线程初始化类型对象及其静态字段。同样地, 这个类型可能以“AppDomain 中立”的方式加载, 所以会出问题。例如, 假定类型构造器的代码进入死循环, 进程中的所有 AppDomain 都无法使用该类型。我的建议是尽量避免使用类型构造器, 或者至少保持它们的短小和简单。

遗憾的是, 除了前面说的这些, 还可能出现更糟糕的情况。由于开发人员习惯在一个方法中获取一个锁, 做一些工作, 然后释放锁, 所以 C# 语言通过 **lock** 关键字来提供了一个简化的语法。假定你要写下面这样的方法:

```
private void SomeMethod() {
```

① 参考 22.2 节“AppDomain”。——译注

② 强调一下, 同步执行意味着不能同时访问一个资源, 只有在你用完了之后, 我才能接着用。在多线程编程中, “同步”(Synchronizing)的定义是: 当两个或更多的线程需要存取共同的资源时, 必须确定在同一时间点只有一个线程能存取共同的资源, 而实现这个过程就称为“同步”。——译注

```
lock (this) {
    // 这里的代码拥有对数据的独占访问权...
}
```

它等价于像下面这样写方法：

```
private void SomeMethod() {
    Boolean lockTaken = false;
    try {
        // 这里可能发生异常(比如 ThreadAbortException)...
        Monitor.Enter(this, ref lockTaken);
        // 这里的代码拥有对数据的独占访问权...
    }
    finally {
        if (lockTaken) Monitor.Exit(this);
    }
}
```

第一个问题是，C#团队认为他们在 **finally** 块中调用 **Monitor.Exit** 是帮了你一个大忙。他们的想法是，这样一来，就可确保锁总是得以释放，无论 **try** 块中发生了什么。但这只是他们一厢情愿的想法。在 **try** 块中，如果在更改状态时发生异常，这个状态就会处于损坏状态。锁在 **finally** 块中退出时，另一个线程可能开始操作损坏的状态。显然，更好的解决方案是让应用程序挂起，而不是让它带着损坏的状态继续运行。这样不仅结果难以预料，还有可能引发安全隐患。第二个问题是，进入和离开 **try** 块会影响方法的性能。有的 JIT 编译器不会内联含有 **try** 块的方法，造成性能进一步下降。所以最终结果是，不仅代码的速度变慢了，还会造成线程访问损坏的状态。^①我的建议是杜绝使用 C# 的 **lock** 语句。

现在终于可以开始讨论 **Boolean lockTaken** 变量了。下面是这个变量试图解决的问题。假定一个线程进入 **try** 块，但在调用 **Monitor.Enter** 之前退出(参见第 22 章)。现在，**finally** 块会得到调用，但它的代码不应退出锁。**lockTaken** 变量就是为了解决这个问题而设计的。它初始化为 **false**，假定现在还没有进入锁(还没有获得锁)。然后，如果调用 **Monitor.Enter**，而且成功获得锁，**Enter** 方法就会将 **lockTaken** 设为 **true**。**finally** 块通过检查 **lockTaken**，便知道到底要不要调用 **Monitor.Exit**。顺便说一句，**SpinLock** 结构也支持这个 **lockTaken** 模式。

30.3.3 ReaderWriterLockSlim 类

我们经常都希望让一个线程简单地读取一些数据的内容。如果这些数据被一个互斥锁(比如 **SimpleSpinLock**、**SimpleWaitLock**、**SimpleHybridLock**、**AnotherHybridLock**、**Mutex** 或者 **Monitor**)保护，那么当多个线程同时试图访问这些数据时，只有一个线程才会运行，其他所有线程都会阻塞。这会造成应用程序伸缩性和吞吐能力的急剧下降。如果所有线程都希望以只读方式访问数据，就根本没有必要阻塞它们；应该允许它们并发地访问数据。另一方面，如果一个线程希望修改数据，这个线程就需要对数据的独占式访问。

^① 顺便说一句，虽然仍然会对性能造成影响，但假如 **try** 块的代码只是对状态执行读取操作，而不是试图修改它，那么在 **finally** 块中释放锁是安全的。

ReaderWriterLockSlim 构造封装了解决这个问题的逻辑。具体地说, 这个构造像下面这样控制线程。

- 一个线程向数据写入时, 请求访问的其他所有线程都被阻塞。
- 一个线程从数据读取时, 请求读取的其他线程允许继续执行, 但请求写入的线程仍被阻塞。
- 向线程写入的线程结束后, 要么解除一个写入线程(**writer**)的阻塞, 使它能向数据写入, 要么解除所有读取线程(**reader**)的阻塞, 使它们能并发读取数据。如果没有线程被阻塞, 锁就进入可以自由使用的状态, 可供下一个 **reader** 或 **writer** 线程获取。
- 从数据读取的所有线程结束后, 一个 **writer** 线程被解除阻塞, 使它能向数据写入。如果没有线程被阻塞, 锁就进入可以自由使用的状态, 可供下一个 **reader** 或 **writer** 线程获取。

下面展示了这个类(未列出部分方法的重载版本):

```
public class ReaderWriterLockSlim : IDisposable {
    public ReaderWriterLockSlim(LockRecursionPolicy recursionPolicy);
    public void Dispose();

    public void EnterReadLock();
    public Boolean TryEnterReadLock(Int32 millisecondsTimeout);
    public void ExitReadLock();

    public void EnterWriteLock();
    public Boolean TryEnterWriteLock(Int32 millisecondsTimeout);
    public void ExitWriteLock();

    // 大多数应用程序从不查询以下任何属性
    public Boolean IsReadLockHeld { get; }
    public Boolean IsWriteLockHeld { get; }
    public Int32 CurrentReadCount { get; }
    public Int32 RecursiveReadCount { get; }
    public Int32 RecursiveWriteCount { get; }
    public Int32 WaitingReadCount { get; }
    public Int32 WaitingWriteCount { get; }
    public LockRecursionPolicy RecursionPolicy { get; }
    // 未列出和 reader 升级到 writer 有关的成员
}
```

以下代码演示了这个构造的用法:

```
internal sealed class Transaction : IDisposable {
    private readonly ReaderWriterLockSlim m_lock =
        new ReaderWriterLockSlim(LockRecursionPolicy.NoRecursion);
    private DateTime m_timeOfLastTrans;

    public void PerformTransaction() {
        m_lock.EnterWriteLock();
        // 以下代码拥有对数据的独占访问权...
        m_timeOfLastTrans = DateTime.Now;
        m_lock.ExitWriteLock();
    }
}
```

```

public DateTime LastTransaction {
    get {
        m_lock.EnterReadLock();
        // 以下代码拥有对数据的共享访问权...
        DateTime temp = m_timeOfLastTrans;
        m_lock.ExitReadLock();
        return temp;
    }
}

public void Dispose() { m_lock.Dispose(); }
}

```

这个构造有几个概念要特别留意。首先，**ReaderWriterLockSlim** 的构造器允许传递一个 **LockRecursionsPolicy** 标志，它的定义如下：

```
public enum LockRecursionPolicy { NoRecursion, SupportsRecursion }
```

如果传递 **SupportsRecursion** 标志，锁就支持线程所有权和递归行为。如同本章早些时候讨论的那样，这些行为对锁的性能有负面影响。所以，建议总是向构造器传递 **LockRecursionPolicy.NoRecursion**(就像本例)。**reader-writer** 锁支持线程所有权和递归的代价非常高昂，因为锁必须跟踪曾允许进入锁的所有 **reader** 线程，同时为每个线程都单独维护递归计数。事实上，为了以线程安全的方式维护所有这些信息，**ReaderWriterLockSlim** 内部要使用一个互斥的“自旋锁”，我不是在开玩笑！

ReaderWriterLockSlim 类提供了一些额外的方法(前面没有列出)允许一个 **reader** 线程升级为 **writer** 线程。以后，线程可以把自己降级回 **reader** 线程。设计者的思路是，一个线程刚开始的时候可能是读取数据。然后，根据数据的内容，线程可能想对数据进行修改。为此，线程要把它自己从 **reader** 升级为 **writer**。锁如果支持这个行为，性能会大打折扣。而且我完全不觉得这是一个有用的功能。线程并不是直接从 **reader** 变成 **writer** 的。当时可能还有其他线程正在读取，这些线程必须完全退出锁。在此之后，尝试升级的线程才允许成为 **writer**。这相当于先让 **reader** 线程退出锁，再立即获取这个锁以进行写入。



注意 FCL 还提供了一个 **ReaderWriterLock** 构造，它是在 Microsoft .NET Framework 1.0 中引入的。这个构造存在许多问题，所以 Microsoft 在 .NET Framework 3.5 中引入了 **ReaderWriterLockSlim** 构造。团队没有对原先的 **ReaderWriterLock** 构造进行改进，因为他们害怕失去和那些正在使用它的应用程序的兼容性。下面列举了 **ReaderWriterLock** 存在的几个问题。首先，即使不存在线程竞争，它的速度也非常慢。其次，线程所有权和递归行为是这个构造强加的，完全取消不了，这使锁变得更慢。最后，相比 **writer** 线程，它更青睐于 **reader** 线程，所以 **writer** 线程可能排起好长的队，却很少有机会获得服务，最终造成“拒绝服务”(DoS)问题。

30.3.4 OneManyLock 类

我自己创建了一个 reader-writer 构造，它的速度比 FCL 的 **ReaderWriterLockSlim** 类快。^① 该类名为 **OneManyLock**，因为它要么允许一个 writer 线程访问，要么允许多个 reader 线程访问。下面展示了这个类：

```
public sealed class OneManyLock : IDisposable {
    public OneManyLock();
    public void Dispose();

    public void Enter(Boolean exclusive);
    public void Leave();
}
```

现在，让我讲解一下这个类是如何工作的。在内部，类定义了一个 **Int64** 字段来存储锁的状态、一个供 reader 线程阻塞的 **Semaphore** 对象以及一个供 writer 线程阻塞的 **AutoResetEvent** 对象。**Int64** 状态字段分解成以下 4 个子字段。

- 4 位代表锁本身的状态。**0=Free**, **1=OwnedByWriter**, **2=OwnedByReaders**, **3=OwnedByReaders AndWriterPending**, **4=ReservedForWriter**。其他值未使用。
- 20 位(0~1048575 的一个数)代表锁当前允许进入的、正在读取的 reader 线程的数量(RR)。
- 20 位代表锁当前正在等待进入锁的 reader 线程的数量(RW)。这些线程在自动重置事件对象(**AutoResetEvent**)上阻塞。
- 20 位代表正在等待进入锁的 writer 线程的数量(WW)。这些线程在其他信号量对象(**Semaphore**)上阻塞。

由于与锁有关的全部信息都在单个 **Int64** 字段中，所以可以使用 **Interlocked** 类的方法来操纵这个字段，这就使锁的速度非常快，而且线程只有在存在竞争的时候才会阻塞。

下面说明了线程进入一个锁进行共享访问时发生的事情。

- 如果锁的状态是 **Free**：将状态设为 **OwnedByReaders**, **RR=1**，返回。
- 如果锁的状态是 **OwnedByReaders**：RR++，返回。
- 否则：**RW++**，阻塞 reader 线程。线程醒来时，循环并重试。

下面说明了进行共享访问的一个线程离开锁时发生的事情。

- **RR--**。
- 如果 **RR > 0**：返回。

^① 参见本书源代码文件 Ch30-1-HybridThreadSync.cs。

- 如果 **WW > 0**: 将状态设为 **ReservedForWriter**, **WW--**, 释放 1 个阻塞的 writer 线程, 返回。
- 如果 **RW==0 && WW==0**: 将状态设为 **Free**, 返回。

下面说明了一个线程进入锁进行独占访问时发生的事情。

- 如果锁的状态为 **Free**: 将状态设为 **OwnedByWriter**, 返回。
- 如果锁的状态为 **ReservedForWriter**: 将状态设为 **OwnedByWriter**, 返回。
- 如果锁的状态为 **OwnedByWriter**: **WW++**, 阻塞 writer 线程。线程醒来时, 循环并重试。
- 否则: 将状态设为 **OwnedByReadersAndWriterPending**, **WW++**, 阻塞 writer 线程。线程醒来时, 循环并重试。

下面说明了进行独占访问的一个线程离开锁时发生的事情。

- 如果 **WW==0 && RW==0**: 将状态设为 **Free**, 返回。
- 如果 **WW > 0**: 将状态设为 **ReservedForWriter**, **WW--**, 释放 1 个阻塞的 writer 线程, 返回。
- 如果 **WW==0 && RW>0**: 将状态设为 **Free**, **RW=0**, 唤醒所有阻塞的 reader 线程, 返回。

假定当前有一个线程(reader)正在锁中进行读取操作, 另一个线程(writer)想进入锁进行(独占的)写入操作。writer 线程首先检查锁是否为 **Free**, 由于不为 **Free**, 所以线程会继续执行下一项检查。然而, 在这个时候, reader 线程可能离开了锁, 而且在离开时发现 **RR** 和 **WW** 都是 0。所以, 线程会将锁的状态设为 **Free**。这便造成了一个问题, 因为 writer 线程已经执行过这个测试, 并且走开了。简单地说, 这里发生的事情是, reader 线程背着 writer 线程改变了 writer 线程访问的状态。我需要解决这个问题, 使锁能够正确工作。

为了解决这个问题, 所有这些位操作都要使用 29.3.4 节“Interlocked Anything 模式”描述的技术来执行。这个模式允许将任何操作转换成线程安全的原子操作。正是因为这个原因, 才使得这个锁的速度是如此之快, 而且其中维护的状态比其他 reader-writer 锁少。比较 **OneManyLock** 类与 FCL 的 **ReaderWriterLockSlim** 和 **ReaderWriterLock** 类的性能, 我得到以下结果:

Incrementing x in OneManyLock: 330	最快
Incrementing x in ReaderWriterLockSlim: 554	约慢 1.7 倍
Incrementing x in ReaderWriterLock: 984	约慢 3 倍

当然, 由于所有 reader-writer 锁都执行比互斥锁更多的逻辑, 所以它们的性能可能要稍差一些。但在比较时不要忘记这样一个事实: reader-writer 锁允许多个 reader 线程同时进入锁。

结束本节的讨论之前, 我想指出的是, 我的 Power Threading 库提供了这个锁的一个稍微不

同的版本,称为 **OneManyResourceLock**。这个锁和库中的其他锁提供了许多附加的功能,比如死锁检测,开启锁的所有权与递归行为(虽然要付出一定性能代价),全部锁的统一编程模型,以及观测锁的运行时行为。可供观测的行为包括:一个线程等待获取一个锁的最长时间和一个锁被占有的最短和最长时间。

30.3.5 CountdownEvent 类

下一个构造是 **System.Threading.CountdownEvent**。这个构造使用了一个 **ManualResetEventSlim** 对象。这个构造阻塞一个线程,直到它的内部计数器变成 0。从某种角度说,这个构造的行为和 **Semaphore** 的行为相反。(Semaphore 是在计数为 0 时阻塞线程。)下面展示了这个类(未列出部分方法的重载版本):

```
public class CountdownEvent : IDisposable {
    public CountdownEvent(Int32 initialCount);
    public void Dispose();
    public void Reset(Int32 count);           // 将 CurrentCount 设为 count
    public void AddCount(Int32 signalCount);  // 使 CurrentCount 递增 signalCount
    public Boolean TryAddCount(Int32 signalCount); // 使 CurrentCount 递增 signalCount
    public Boolean Signal(Int32 signalCount);  // 使 CurrentCount 递减 signalCount
    public Boolean Wait(Int32 millisecondsTimeout, CancellationToken cancellationToken);
    public Int32 CurrentCount { get; }
    public Boolean IsSet { get; }             // 如果 CurrentCount 为 0, 就返回 true
    public WaitHandle WaitHandle { get; }
}
```

一旦一个 **CountdownEvent** 的 **CurrentCount** 变成 0, 它就不能更改了。**CurrentCount** 为 0 时, **AddCount** 方法会抛出一个 **InvalidOperationException**。如果 **CurrentCount** 为 0, **TryAddCount** 直接返回 **false**。

30.3.6 Barrier 类

System.Threading.Barrier 构造用于解决一个非常稀有的问题,平时一般用不上。**Barrier** 控制的一系列线程需要并行工作,从而在一个算法的不同阶段推进。或许通过一个例子更容易理解:当 CLR 使用它的垃圾回收器(GC)的服务器版本时,GC 算法为每个内核都创建一个线程。这些线程在不同应用程序线程的栈中向上移动,并发标记堆中的对象。每个线程完成了它自己的那一部分工作之后,必须停下来等待其他线程完成。所有线程都标记好对象后,线程就可以并发地压缩(compact)堆的不同部分。每个线程都完成了对它的那一部分的堆的压缩之后,线程必须阻塞以等待其他线程。所有线程都完成了对自己那一部分的堆的压缩之后,所有线程都要在应用程序的线程的栈中上行,对根进行修正,使之引用因为压缩而发生了移动的对象的新位置。只有在所有线程都完成这个工作之后,垃圾回收器的工作才算真正完成,应用程序的线程现在可以恢复执行了。

使用 **Barrier** 类可轻松应付像这样的情形。下面展示了这个类(未列出部分方法的重载版本):

```
public class Barrier : IDisposable {
    public Barrier(Int32 participantCount, Action<Barrier> postPhaseAction);
    public void Dispose();
    public Int64 AddParticipants(Int32 participantCount); // 添加参与者
}
```

```

public void RemoveParticipants(Int32 participantCount); // 减去参与者
public Boolean SignalAndWait(Int32 millisecondsTimeout,
    CancellationToken cancellationToken);

public Int64 CurrentPhaseNumber { get; } // 指出进行到哪一个阶段(从0开始)
public Int32 ParticipantCount { get; } // 参与者数量
public Int32 ParticipantsRemaining { get; } // 需要调用 SignalAndWait 的线程数
}

```

构造 **Barrier** 时要告诉它有多少个线程准备参与工作，还可传递一个 **Action<Barrier>** 委托来引用所有参与者完成一个阶段的工作后要调用的代码。可以调用 **AddParticipant** 和 **RemoveParticipant** 方法在 **Barrier** 中动态添加和删除参与线程。但在实际应用中，人们很少这样做。每个线程完成它的阶段性工作后，应调用 **SignalAndWait**，告诉 **Barrier** 线程已经完成一个阶段的工作，而 **Barrier** 会阻塞线程(使用一个 **ManualResetEventSlim**)。所有参与者都调用了 **SignalAndWait** 后，**Barrier** 将调用指定的委托(由最后一个调用 **SignalAndWait** 的线程调用)，然后解除正在等待的所有线程的阻塞，使它们开始下一阶段。

30.3.7 线程同步构造小结

我的建议是，代码尽量不要阻塞任何线程。执行异步计算或 I/O 操作时，将数据从一个线程交给另一个线程时，应避免多个线程同时访问数据。如果不能完全做到这一点，请尽量使用 **Volatile** 和 **Interlocked** 的方法，因为它们的速度很快，而且绝不阻塞线程。遗憾的是，这些方法只能操作简单类型。但可以像 29.3.4 节“**Interlocked Anything** 模式”描述的那样在这些类型上执行丰富的操作。

主要在以下两种情况下阻塞线程。

- **线程模型很简单**

阻塞线程虽会牺牲一些资源和性能，但可顺序地写应用程序代码，无需使用回调方法。不过，C# 的异步方法功能现在提供了不阻塞线程的简化编程模型。

- **线程有专门用途**

有的线程是特定任务专用的。最好的例子就是应用程序的主线程。如果应用程序的主线程没有阻塞，它最终就会返回，造成整个进程终止。其他例子还有应用程序的 GUI 线程。Windows 要求一个窗口或控件总是由创建它的线程操作。因此，我们有时写代码阻塞一个 GUI 线程，直到其他某个操作完成。然后，GUI 线程根据需要对窗口和控件进行更新。当然，阻塞 GUI 线程会造成应用程序挂起，使用户体验变差。

要避免阻塞线程，就不要刻意地为线程打上标签。例如，不要创建一个拼写检查线程、一个语法检查线程、一个处理特定客户端请求的线程等。为线程打上标签，其实是在告诫自己该线程不能做其他任何事情。但由于线程是如此昂贵，所以不能把它们专门用于某个目的。相反，应通过线程池将线程出租短暂时间。所以正确方式是一个线程池线程开始拼写检查，再改为语法检查，再代表一个客户端请求执行工作，以此类推。

如果一定要阻塞线程，为了同步在不同 **AppDomain** 或进程中运行的线程，请使用内核对象

构造。要在一系列操作中原子性地操纵状态，请使用带有私有字段的 **Monitor** 类。^①另外，可以使用 reader-writer 锁代替 **Monitor**。reader-writer 锁通常比 **Monitor** 慢，但它们允许多个线程并发执行，这提升了总体性能，并将阻塞线程的机率降至最低。

此外，避免使用递归锁(尤其是递归的 reader-writer 锁)，因为它们会损害性能。但 **Monitor** 是递归的，性能也不错。^②另外，不要在 **finally** 块中释放锁，因为进入和离开异常处理块会招致性能损失。如果在更改状态时抛出异常，状态就会损坏，操作这个状态的其他线程会出现不可预料的行为，并可能引入安全隐患。

当然，如果写代码来占有锁，注意时间不要太长，否则会增大线程阻塞的机率。后面的 30.6 节“异步的同步构造”会展示如何利用集合类防止长时间占有锁。

最后，对于计算限制的工作，可以使用任务(参见第 27.5 节“任务”)避免使用大量线程同步构造。我喜欢的一个设计是，每个任务都关联一个或多个延续任务。某个操作完成后，这些任务将通过某个线程池线程继续执行。这比让一个线程阻塞并等待某个操作完成好得多。对于 I/O 限制的工作，调用各种 **XxxAsync** 方法将造成你的代码在 I/O 操作完成后继续；这其实类似于任务的延续任务。

30.4 著名的双检锁技术

双检锁(Double-Check Locking)是一个非常著名的技术，开发人员用它将单实例(singleton)对象的构造推迟到应用程序首次请求该对象时进行。这有时也称为延迟初始化(lazy initialization)。如果应用程序永远不请求对象，对象就永远不会构造，从而节省了时间和内存。但当多个线程同时请求单实例对象时就可能出问题。这个时候必须使用一些线程同步机制确保单实例对象只被构造一次。

该技术之所以有名，不是因为它非常有趣或有用，而是因为它曾经是人们热烈讨论的话题。该技术曾在 Java 中大量使用，后来有一天，一些人发现 Java 不能保证该技术在任何地方都能正确工作。这个使它出名的网页对问题进行了清楚的说明：<http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>。

无论如何，一个好消息是，CLR 很好地支持双检锁技术，这应该归功于 CLR 的内存模型以及 **volatile** 字段访问(参见第 29 章)。以下代码演示了如何用 C# 实现双检锁技术^③：

```
public sealed class Singleton {
    // s_lock 对象是实现线程安全所需要的。定义这个对象时，我们假设创建单实例对象的
    // 代价高于创建一个 System.Object 对象，并假设可能根本不需要创建单实例对象、
    // 否则，更经济、更简单的做法是在一个类构造器中创建单实例对象、
```

① 可用 **SpinLock** 代替 **Monitor**，因为 **SpinLock** 稍快一些。但 **SpinLock** 比较危险，因为它可能浪费 CPU 时间。而且在我看来，它还没有快到非用不可的地步。

② 部分是由于 **Monitor** 用本机代码(而非托管代码)来实现。

③ 两个 if 语句即是两次检查。——译注

```

private static Object s_lock = new Object();

// 这个字段引用一个单实例对象
private static Singleton s_value = null;

// 私有构造器阻止这个类外部的任何代码创建实例
private Singleton() {
    // 把初始化单实例对象的代码放在这里...
}

// 以下公共静态方法返回单实例对象 (如有必要就创建它)
public static Singleton GetSingleton() {
    // 如果单实例对象已经创建, 直接返回它 (这样速度很快)
    if (s_value != null) return s_value;

    Monitor.Enter(s_lock); // 还没有创建, 让一个线程创建它
    if (s_value == null) {
        // 仍未创建, 创建它
        Singleton temp = new Singleton();

        // 将引用保存到 s_value 中 (参见正文的详细讨论)
        Volatile.Write(ref s_value, temp);
    }
    Monitor.Exit(s_lock);

    // 返回对单实例对象的引用
    return s_value;
}
}

```

双检锁技术背后的思路在于, 对 **GetSingleton** 方法的一个调用可以快速地检查 **s_value** 字段, 判断对象是否创建。如果是, 方法就返回对它的引用。这里的妙处在于, 如果对象已经构造好, 就不需要线程同步; 应用程序会运行得非常快。另一方面, 如果调用 **GetSingleton** 方法的第一个线程发现对象还没有创建, 就会获取一个线程同步锁来确保只有一个线程构造单实例对象。这意味着只有线程第一次查询单实例对象时, 才会出现性能上的损失。

现在, 让我解释一下为什么这个模式在 Java 中出了问题。Java 虚拟机(JVM)在 **GetSingleton** 方法开始的时候将 **s_value** 的值读入 CPU 寄存器。然后, 对第二个 **if** 语句求值时, 它直接查询寄存器, 造成第二个 **if** 语句总是求值为 **true**, 结果就是多个线程都会创建 **Singleton** 对象。当然, 只有多个线程恰好同时调用 **GetSingleton** 才会发生这种情况。在大多数应用程序中, 发生这种情况的概率都是极低的。这正是该问题在 Java 中长时间都没有被发现的原因。

在 CLR 中, 对任何锁方法的调用都构成了一个完整的内存栅栏, 在栅栏之前写入的任何变量都必须在栅栏之前完成; 在栅栏之后的任何变量读取都必须在栅栏之后开始。对于 **GetSingleton** 方法, 这意味着 **s_value** 字段的值必须在调用了 **Monitor.Enter** 之后重新读取; 调用前缓存到寄存器中的东西作不了数。

GetSingleton 内部有一个 **Volatile.Write** 调用。下面让我解释一下它解决的是什么问题。假定第二个 **if** 语句中包含的是下面这行代码:

```
s_value = new Singleton(); // 你极有可能这样写
```

你的想法是让编译器生成代码为一个 **Singleton** 分配内存，调用构造器来初始化字段，再将引用赋给 **s_value** 字段。使一个值对其他线程可见称为发布(publishing)。但那只是你一厢情愿的想法，编译器可能这样做：为 **Singleton** 分配内存，将引用发布到(赋给)**s_value**，再调用构造器。从单线程的角度出发，像这样改变顺序是无关紧要的。但在将引用发布给 **s_value** 之后，并在调用构造器之前，如果另一个线程调用了 **GetSingleton** 方法，那么会发生什么？这个线程会发现 **s_value** 不为 **null**，所以会开始使用 **Singleton** 对象，但对象的构造器还没有结束执行呢！这是一个很难追踪的 bug，尤其是它完全是由于计时而造成的。

对 **Volatile.Write** 的调用修正了这个问题。它保证 **temp** 中的引用只有在构造器结束执行之后，才发布到 **s_value** 中。解决这个问题的另一个办法是使用 C# 的 **volatile** 关键字来标记 **s_value** 字段。这使向 **s_value** 的写入变得具有“易变性”。同样，构造器必须在写入发生前结束运行。但遗憾的是，这同时会使所有读取操作具有“易变性”^①，这是完全没必要的。因此，使用 **volatile** 关键字，会使性能无谓地受到损害。

本节开头指出双检锁技术无关有趣。在我看来，是开发人员把它捧得太高，在许多不该使用它的时候也在用它。大多数时候，这个技术实际会损害效率。下面是 **Singleton** 类的一个简单得多的版本，它的行为和上一个版本相同。这个版本没有使用“著名”的双检锁技术：

```
internal sealed class Singleton {
    private static Singleton s_value = new Singleton();

    // 私有构造器防止这个类外部的任何代码创建一个实例
    private Singleton() {
        // 将初始化单实例对象的代码放在这里...
    }

    // 以下公共静态方法返回单实例对象 (如有必要就创建它)
    public static Singleton GetSingleton() { return s_value; }
}
```

由于代码首次访问类的成员时，CLR 会自动调用类型的类构造器，所以首次有一个线程查询 **Singleton** 的 **GetSingleton** 方法时，CLR 就会自动调用类构造器，从而创建一个对象实例。此外，CLR 已保证了对类构造器的调用是线程安全的。我已在 8.3 节“类型构造器”对此进行了解释。这种方式的缺点在于，首次访问类的任何成员都会调用类型构造器。所以，如果 **Singleton** 类型定义其他静态成员，就会在访问其他任何静态成员时创建 **Singleton** 对象。有人通过定义嵌套类来解决这个问题。

下面展示了生成 **Singleton** 对象的第三种方式：

```
internal sealed class Singleton {
    private static Singleton s_value = null;

    // 私有构造器阻止这个类外部的任何代码创建实例
    private Singleton() {
        // 把初始化单实例对象的代码放在这里...
    }

    // 以下公共静态方法返回单实例对象 (如有必要就创建它)
```

① 所以对字段的读取也要同步了。——译注

```

public static Singleton GetSingleton() {
    if (s_value != null) return s_value;

    // 创建一个新的单实例对象，并把它固定下来(如果另一个线程还没有固定它的话)
    Singleton temp = new Singleton();
    Interlocked.CompareExchange(ref s_value, temp, null);

    // 如果这个线程竞争失败，新建的第二个单实例对象会被垃圾回收

    return s_value; // 返回对单实例对象的引用
}

```

如果多个线程同时调用 **GetSingleton**，这个版本可能创建两个(或更多)**Singleton** 对象。然而，对 **Interlocked.CompareExchange** 的调用确保只有一个引用才会发布到 **s_value** 字段中。没有通过这个字段固定下来的任何对象^①会在以后被垃圾回收。由在大多数应用程序都很少发生多个线程同时调用 **GetSingleton** 的情况，所以不太可能同时创建多个 **Singleton** 对象。

虽然可能创建多个 **Singleton** 对象，但上述代码有多方面的优势。首先，它的速度非常快。其次，它永不阻塞线程。相反，如果一个线程池线程在一个 **Monitor** 或者其他任何内核模式的线程同步构造上阻塞，线程池就会创建另一个线程来保持 CPU 的“饱和”。因此，会分配并初始化更多的内存，而且所有 DLL 都会收到一个线程连接通知。使用 **CompareExchange** 则永远不会发生这种情况。当然，只有在构造器没有副作用的时候才能使用这个技术。

FCL 有两个类型封装了本节描述的模式。下面是泛型 **System.Lazy** 类(有的方法没有列出):

```

public class Lazy<T> {
    public Lazy(Func<T> valueFactory, LazyThreadSafetyMode mode);
    public Boolean IsValueCreated { get; }
    public T Value { get; }
}

```

以下代码演示了它是如何工作的:

```

public static void Main() {
    // 创建一个“延迟初始化”包装器，它将 DateTime 的获取包装起来
    Lazy<String> s = new Lazy<String>(() =>
        DateTime.Now.ToString(), true);

    Console.WriteLine(s.IsValueCreated); // 还没有查询 Value，所以返回 false
    Console.WriteLine(s.Value);           // 现在调用委托
    Console.WriteLine(s.IsValueCreated); // 已经查询了 Value，所以返回 true
    Thread.Sleep(10000);                  // 等待 10 秒，再次显示时间
    Console.WriteLine(s.Value);           // 委托没有调用：显示相同结果
}

```

在我的机器上运行，得到以下结果:

```

False
2:40:42 PM
True
2:40:42 PM ← 注意，10 秒之后，时间未发生改变

```

① 称为不可达对象。——译注

上述代码构造 **Lazy** 类的实例，并向它传递某个 **LazyThreadSafetyMode** 标志。下面总结了这些标志：

```
public enum LazyThreadSafetyMode {
    None, // 完全没有线程安全支持 (适合 GUI 应用程序)
    ExecutionAndPublication // 使用双检锁技术
    PublicationOnly, // 使用 Interlocked.CompareExchange 技术
}
```

内存有限时可能不想创建 **Lazy** 类的实例。这时可调用 **System.Threading.LazyInitializer** 类的静态方法。下面展示了这个类：

```
public static class LazyInitializer {
    // 这两个方法在内部使用了 Interlocked.CompareExchange:
    public static T EnsureInitialized<T>(ref T target) where T: class;
    public static T EnsureInitialized<T>(ref T target, Func<T> valueFactory) where T: class;

    // 这两个方法在内部将同步锁(syncLock)传给 Monitor 的 Enter 和 Exit 方法
    public static T EnsureInitialized<T>(ref T target, ref Boolean initialized,
        ref Object syncLock);
    public static T EnsureInitialized<T>(ref T target, ref Boolean initialized,
        ref Object syncLock, Func<T> valueFactory);
}
```

另外，为 **EnsureInitialized** 方法的 **syncLock** 参数显式指定同步对象，可以用同一个锁保护多个初始化函数和字段。下例展示了如何使用这个类的方法：

```
public static void Main() {
    String name = null;
    // 由于 name 为 null，所以委托执行并初始化 name
    LazyInitializer.EnsureInitialized(ref name, () => "Jeffrey");
    Console.WriteLine(name); // 显示"Jeffrey"

    // 由于 name 已经不为 null，所以委托不运行：name 不改变
    LazyInitializer.EnsureInitialized(ref name, () => "Richter");
    Console.WriteLine(name); // 还是显示"Jeffrey"
}
```

30.5 条件变量模式

假定一个线程希望在一个复合条件为 **true** 时执行一些代码。一个选项是让线程连续“自旋”，反复测试条件。但这会浪费 CPU 时间，也不可能对构成复合条件的多个变量进行原子性的测试。幸好，有一个模式允许线程根据一个复合条件来同步它们的操作，而且不会浪费资源。这个模式称为条件变量(condition variable)模式。我们通过 **Monitor** 类中定义的以下方法来使用该模式：

```
public static class Monitor {
    public static Boolean Wait(Object obj);
    public static Boolean Wait(Object obj, Int32 millisecondsTimeout);

    public static void Pulse(Object obj);
    public static void PulseAll(Object obj);
}
```

下面展示了这个模式：

```

internal sealed class ConditionVariablePattern {
    private readonly Object m_lock = new Object();
    private Boolean m_condition = false;

    public void Thread1() {
        Monitor.Enter(m_lock); // 获取一个互斥锁

        // 在锁中，“原子性”地测试复合条件
        while (!m_condition) {
            // 条件不满足，就等待另一个线程更改条件
            Monitor.Wait(m_lock); // 临时释放锁，使其他线程能获取它
        }

        // 条件满足，处理数据...

        Monitor.Exit(m_lock); // 永久释放锁
    }

    public void Thread2() {
        Monitor.Enter(m_lock); // 获取一个互斥锁

        // 处理数据并修改条件...
        m_condition = true;

        // Monitor.Pulse(m_lock); // 锁释放之后唤醒一个正在等待的线程
        Monitor.PulseAll(m_lock); // 锁释放之后唤醒所有正在等待的线程

        Monitor.Exit(m_lock); // 释放锁
    }
}

```

在上述代码中，执行 **Thread1** 方法的线程进入一个互斥锁，然后对一个条件进行测试。在这里，我只是检查一个 **Boolean** 字段，但它可以是任意复合条件。例如，可以检查是不是三月的一个星期二，同时一个特定的集合对象是否包含 10 个元素。如果条件为 **false**，你希望线程在条件上“自旋”。但自旋会浪费 CPU 时间，所以线程不是自旋，而是调用 **Wait**。**Wait** 释放锁，使另一个线程能获得它并阻塞调用线程。

Thread2 方法展示了第二个线程执行的代码。它调用 **Enter** 来获取锁的所有权，处理一些数据，造成一些状态的改变，再调用 **Pulse** 或 **PulseAll**，从而解除一个线程因为调用 **Wait** 而进入的阻塞状态。注意，**Pulse** 只解除等待最久的线程(如果有的话)的阻塞，而 **PulseAll** 解除所有正在等待的线程(如果有的话)的阻塞。但所有未阻塞的线程还没有醒来。执行 **Thread2** 的线程必须调用 **Monitor.Exit**，允许锁由另一个线程拥有。另外，如果调用的是 **PulseAll**，其他线程不会同时解除阻塞。调用 **Wait** 的线程解除阻塞后，它成为锁的所有者。由于这是一个互斥锁，所以一次只能有一个线程拥有它。其他线程只有在锁的所有者调用了 **Wait** 或 **Exit** 之后才能得到它。

执行 **Thread1** 的线程醒来时，它进行下一次循环迭代，再次对条件进行测试。如果条件仍为 **false**，它就再次调用 **Wait**。如果条件为 **true**，它就处理数据，并最终调用 **Exit**。这样就会将锁释放，使其他线程能得到它。这个模式的妙处在于，可以使用简单的同步逻辑(只是一个锁)来测试构成一个复合条件的几个变量，而且多个正在等待的线程可以全部解除阻塞，而不会造成任何逻辑错误。唯一的缺点就是解除线程的阻塞可能浪费一些 CPU 时间。

下面展示了一个线程安全的队列，它允许多个线程在其中对数据项(item)进行入队和出队操

作。注意，除非有了一个可供处理的数据项，否则试图出队一个数据项的线程会一直阻塞。

```
internal sealed class SynchronizedQueue<T> {
    private readonly Object m_lock = new Object();
    private readonly Queue<T> m_queue = new Queue<T>();

    public void Enqueue(T item) { // 入队
        Monitor.Enter(m_lock);

        // 一个数据项入队后，就唤醒任何/所有正在等待的线程
        m_queue.Enqueue(item);
        Monitor.PulseAll(m_lock);

        Monitor.Exit(m_lock);
    }

    public T Dequeue() { // 出队
        Monitor.Enter(m_lock);

        // 队列为空(这是条件)就一直循环
        while (m_queue.Count == 0)
            Monitor.Wait(m_lock);

        // 使一个数据项出队，返回它供处理
        T item = m_queue.Dequeue();
        Monitor.Exit(m_lock);
        return item;
    }
}
```

30.6 异步的同步构造

任何使用了内核模式基元的线程同步构造，我都不是特别喜欢。因为所有这些基元都会阻塞一个线程的运行。创建线程的代价很大。创建了不用，这于情于理都不通。下面这个例子能很好地说明这个问题。

假定客户端向网站发出请求。客户端请求到达时，一个线程池线程开始处理客户端请求。假定这个客户端想以线程安全的方式修改服务器中的数据，所以它请求一个 reader-writer 锁来进行写入(这使线程成为一个 writer 线程)。假定这个锁被长时间占有。在锁被占有期间，另一个客户端请求到达了，所以线程池为这个请求创建新线程。然后，线程阻塞，尝试获取 reader-writer 锁来进行读取(这使线程成为一个 reader 线程)。事实上，随着越来越多的客户端请求到达，线程池会创建越来越多的线程，所有这些线程都傻傻地在锁上面阻塞。服务器把它的所有时间都花在创建线程上面，而目的仅仅是让它们停止运行！，这样的服务器完全没有伸缩性可言。

更糟的是，当 writer 线程释放锁时，所有 reader 线程都同时解除阻塞并开始执行。现在，又变成了大量线程试图在相对数量很少的 CPU 上运行。所以，Windows 开始在线程之间不停地进行上下文切换。由于上下文切换产生了大量开销，所以真正的工作反正没有得到很快的处理。

观察本章介绍的所有构造，你会发现这些构造想要解决的许多问题其实最好都是用第 27 章讨论的 **Task** 类来完成。拿 **Barrier** 类来说：可以生成几个 **Task** 对象来处理一个阶段。然

后,当所有这些任务完成后,可以用另外一个或多个 **Task** 对象继续。和本章展示的大量构造相比,任务具有下述许多优势。

- 任务使用的内存比线程少得多,创建和销毁所需的时间也少得多。
- 线程池根据可用 CPU 数量自动伸缩任务规模。
- 每个任务完成一个阶段后,运行任务的线程回到线程池,在那里能接受新任务。
- 线程池是站在整个进程的高度观察任务。所以,它能更好地调度这些任务,减少进程中的线程数,并减少上下文切换。

锁很流行,但长时间拥有会带来巨大的伸缩性问题。如果代码能通过异步的同步构造指出它想要一个锁,那么会非常有用。在这种情况下,如果线程得不到锁,可直接返回并执行其他工作,而不必在那里傻傻地阻塞。以后当锁可用时,代码可恢复执行并访问锁所保护的资源。我当年在为客户端解决一个重大的伸缩性问题时有了这个思路,并于 2009 年将专利权卖给了 Microsoft,专利号是 7603502。

SemaphoreSlim 类通过 **WaitAsync** 方法实现了这个思路,下面是该方法的最复杂的重载版本的签名。

```
public Task<Boolean> WaitAsync(Int32 millisecondsTimeout, CancellationToken cancellationToken);
```

可用它异步地同步对一个资源的访问(不阻塞任何线程)。

```
private static async Task AccessResourceViaAsyncSynchronization(SemaphoreSlim asyncLock)
{
    // TODO: 执行你想要的任何代码...

    await asyncLock.WaitAsync(); // 请求获得锁来获得对资源的独占访问
    // 执行到这里,表明没有别的线程正在访问资源
    // TODO: 独占地访问资源...

    // 资源访问完毕就放弃锁,使其他代码能访问资源
    asyncLock.Release();

    // TODO: 执行你想要的任何代码...
}
```

SemaphoreSlim 的 **WaitAsync** 方法很有用,但它提供的是信号量语义。一般创建最大计数为 1 的 **SemaphoreSlim**,从而对 **SemaphoreSlim** 保护的资源进行互斥访问。所以,这和使用 **Monitor** 时的行为相似,只是 **SemaphoreSlim** 不支持线程所有权和递归语义(这是好事)。

但 reader-writer 语义呢? .NET Framework 提供了 **ConcurrentExclusiveSchedulerPair** 类。

```
public class ConcurrentExclusiveSchedulerPair {
    public ConcurrentExclusiveSchedulerPair();

    public TaskScheduler ExclusiveScheduler { get; }
    public TaskScheduler ConcurrentScheduler { get; }

    // 未列出其他方法...
}
```

这个类的实例带有两个 **TaskScheduler** 对象,它们在调度任务时负责提供 reader/writer 语义。只要当前没有运行使用 **ConcurrentScheduler** 调度的任务,使用 **ExclusiveScheduler** 调度的任何任务将独占式地运行(一次只能运行一个)。另外,只要当前没有运行使用 **ExclusiveScheduler** 调度的任务,使用 **ConcurrentScheduler** 调度的任务就可同时运行(一次运行多个)。以下代码演示了该类的使用。

```
private static void ConcurrentExclusiveSchedulerDemo() {
    var cesp = new ConcurrentExclusiveSchedulerPair();
    var tfExclusive = new TaskFactory(cesp.ExclusiveScheduler);
    var tfConcurrent = new TaskFactory(cesp.ConcurrentScheduler);

    for (Int32 operation = 0; operation < 5; operation++) {
        var exclusive = operation < 2; // 出于演示的目的,我建立了 2 个独占和 3 个并发

        (exclusive ? tfExclusive : tfConcurrent).StartNew(() => {
            Console.WriteLine("{0} access", exclusive ? "exclusive" : "concurrent");
            // TODO: 在这里进行独占写入或并发读取操作...
        });
    }
}
```

遗憾的是, .NET Framework 没有提供具有 reader-writer 语义的异步锁。但我构建了这样的一个类,称为 **AsyncOneManyLock**。它的用法和 **SemaphoreSlim** 一样。下面是一个例子。

```
private static async Task AccessResourceViaAsyncSynchronization(AsyncOneManyLock
asyncLock) {
    // TODO: 执行你想要的任何代码...

    // 为想要的并发访问传递 OneManyMode.Exclusive 或 OneManyMode.Shared
    await asyncLock.AcquireAsync(OneManyMode.Shared); // 要求共享访问
    // 如果执行到这里,表明没有其他线程在向资源写入;可能有其他线程在读取
    // TODO: 从资源读取...

    // 资源访问完毕就放弃锁,使其他代码能访问资源
    asyncLock.Release();

    // TODO: 执行你想要的任何代码...
}
```

下面展示了 **AsyncOneManyLock** 类。

```
public enum OneManyMode { Exclusive, Shared }

public sealed class AsyncOneManyLock {
    #region 锁的代码
    private SpinLock m_lock = new SpinLock(true); // 自旋锁不要用 readonly
    private void Lock() { Boolean taken = false; m_lock.Enter(ref taken); }
    private void Unlock() { m_lock.Exit(); }
    #endregion

    #region 锁的状态和辅助方法
    private Int32 m_state = 0;
    private Boolean IsFree { get { return m_state == 0; } }
    private Boolean IsOwnedByWriter { get { return m_state == -1; } }
    private Boolean IsOwnedByReaders { get { return m_state > 0; } }
    private Int32 AddReaders(Int32 count) { return m_state += count; }
    private Int32 SubtractReader() { return --m_state; }
    private void MakeWriter() { m_state = -1; }
    private void MakeFree() { m_state = 0; }
    #endregion
}
```

```

// 目的是在非竞态条件时增强性能和减少内存消耗
private readonly Task m_noContentionAccessGranter;

// 每个等待的 writer 都通过它们在这里排队的 TaskCompletionSource 来唤醒
private readonly Queue<TaskCompletionSource<Object>> m_qWaitingWriters =
    new Queue<TaskCompletionSource<Object>>();

// 一个 TaskCompletionSource 收到信号, 所有等待的 reader 都唤醒
private TaskCompletionSource<Object> m_waitingReadersSignal =
    new TaskCompletionSource<Object>();
private Int32 m_numWaitingReaders = 0;

public AsyncOneManyLock() {
    m_noContentionAccessGranter = Task.FromResult<Object>(null);
}

public Task WaitAsync(OneManyMode mode) {
    Task accessGranter = m_noContentionAccessGranter; // 假定无竞争

    Lock();
    switch (mode) {
        case OneManyMode.Exclusive:
            if (IsFree) {
                MakeWriter(); // 无竞争
            } else {
                // 有竞争: 新的 writer 任务进入队列, 并返回它使 writer 等待
                var tcs = new TaskCompletionSource<Object>();
                m_qWaitingWriters.Enqueue(tcs);
                accessGranter = tcs.Task;
            }
            break;

        case OneManyMode.Shared:
            if (IsFree || (IsOwnedByReaders && m_qWaitingWriters.Count == 0)) {
                AddReaders(1); // 无竞争
            } else { // 有竞争
                // 竞争: 递增等待的 reader 数量, 并返回 reader 任务使 reader 等待
                m_numWaitingReaders++;
                accessGranter = m_waitingReadersSignal.Task.ContinueWith(t => t.Result);
            }
            break;
    }
    Unlock();

    return accessGranter;
}

public void Release() {
    TaskCompletionSource<Object> accessGranter = null; // 假定没有代码被释放

    Lock();
    if (IsOwnedByWriter) MakeFree(); // 一个 writer 离开
    else SubtractReader(); // 一个 reader 离开

    if (IsFree) {
        // 如果自由, 唤醒 1 个等待的 writer 或所有等待的 readers
        if (m_qWaitingWriters.Count > 0) {
            MakeWriter();
            accessGranter = m_qWaitingWriters.Dequeue();
        } else if (m_numWaitingReaders > 0) {
            AddReaders(m_numWaitingReaders);
            m_numWaitingReaders = 0;
            accessGranter = m_waitingReadersSignal;
        }

        // 为将来需要等待的 readers 创建一个新的 TCS
    }
}

```

```

        m_waitingReadersSignal = new TaskCompletionSource<Object>();
    }
}
Unlock();

// 唤醒锁外面的 writer/reader, 减少竞争机率以提高性能
if (accessGranter != null) accessGranter.SetResult(null);
}
}

```

如同我说过的一样, 上述代码永远不会阻塞线程。原因是我在内部没有使用任何内核构造。这里确实使用了一个 **SpinLock**, 它在内部使用了用户模式的构造。但第 29 章讨论自旋锁的时候说过, 只有执行时间很短的代码段才可以用自旋锁来保护。查看我的 **WaitAsync** 方法, 会发现我用锁保护的只是一些整数计算和比较, 以及构造一个 **TaskCompletionSource** 并把它添加到队列的动作。这花不了多少时间, 所以能保证锁只是短时间被占有。

类似地, 查看我的 **Release** 方法, 会发现做的事情不外乎一些整数计算、一个比较以及将一个 **TaskCompletionSource** 出队或者构造一个 **TaskCompletionSource**。这同样花不了多少时间。使用一个 **SpinLock** 来保护对 **Queue** 的访问, 我感觉非常自在。线程在使用这种锁时永远不会阻塞, 使我能构建响应灵敏的、可伸缩的软件。

30.7 并发集合类

FCL 自带 4 个线程安全的集合类, 全部在 **System.Collections.Concurrent** 命名空间中定义。它们是 **ConcurrentQueue**, **ConcurrentStack**, **ConcurrentDictionary** 和 **ConcurrentBag**。以下是它们最常用的一些成员。

```

// 以先入先出 (FIFO) 的顺序处理数据项
public class ConcurrentQueue<T> : IProducerConsumerCollection<T>,
    IEnumerable<T>, ICollection, IEnumerable {

    public ConcurrentQueue();
    public void Enqueue(T item);
    public Boolean TryDequeue(out T result);
    public Int32 Count { get; }
    public IEnumerator<T> GetEnumerator();
}

```

```

// 以后入先出 (LIFO) 的方式处理数据项
public class ConcurrentStack<T> : IProducerConsumerCollection<T>,
    IEnumerable<T>, ICollection, IEnumerable {

    public ConcurrentStack();
    public void Push(T item);
    public Boolean TryPop(out T result);
    public Int32 Count { get; }
    public IEnumerator<T> GetEnumerator();
}

```

```

// 一个无序数据项集合, 允许重复
public class ConcurrentBag<T> : IProducerConsumerCollection<T>,
    IEnumerable<T>, ICollection, IEnumerable {

    public ConcurrentBag();
    public void Add(T item);
    public Boolean TryTake(out T result);
    public Int32 Count { get; }
}

```

```

    public IEnumerator<T> GetEnumerator();
}

// 一个无序 key/value 对集合
public class ConcurrentDictionary<TKey, TValue> : IDictionary<TKey, TValue>,
    ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>,
    IDictionary, ICollection, IEnumerable {

    public ConcurrentDictionary();
    public Boolean TryAdd(TKey key, TValue value);
    public Boolean TryGetValue(TKey key, out TValue value);
    public TValue this[TKey key] { get; set; }
    public Boolean TryUpdate(TKey key, TValue newValue, TValue comparisonValue);
    public Boolean TryRemove(TKey key, out TValue value);
    public TValue AddOrUpdate(TKey key, TValue addValue,
        Func<TKey, TValue> updateValueFactory);
    public TValue GetOrAdd(TKey key, TValue value);
    public Int32 Count { get; }
    public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator();
}

```

所有这些集合类都是“非阻塞”的。换言之，如果一个线程试图提取一个不存在的元素(数据项)，线程会立即返回；线程不会阻塞在那里，等着一个元素的出现。正是由于这个原因，所以如果获取了一个数据项，像 **TryDequeue**、**TryPop**、**TryTake** 和 **TryGetValue** 这样的方法全都返回 **true**；否则返回 **false**。

一个集合“非阻塞”，并不意味着它就不需要锁了。**ConcurrentDictionary** 类在内部使用了 **Monitor**。但是，对集合中的项进行操作时，锁只被占有极短的时间。**ConcurrentQueue** 和 **ConcurrentStack** 确实不需要锁；它们两个在内部都使用 **Interlocked** 的方法来操纵集合。一个 **ConcurrentBag** 对象(一个 bag)由大量迷你集合对象构成，每个线程一个。线程将一个项添加到 bag 中时，就用 **Interlocked** 的方法将这个项添加到调用线程的迷你集合中。一个线程试图从 bag 中提取一个元素时，bag 就检查调用线程的迷你集合，试图从中取出数据项。如果数据项在那里，就用一个 **Interlocked** 方法提取这个项。如果不在，就在内部获取一个 **Monitor**，以便从另一个线程的迷你集合提取一个项。这称为一个线程从另一个线程“窃取”一个数据项。

注意，所有并发集合类都提供了 **GetEnumerator** 方法，它一般用于 C# 的 **foreach** 语句，但也可用于 LINQ。对于 **ConcurrentStack**、**ConcurrentQueue** 和 **ConcurrentBag** 类，**GetEnumerator** 方法获取集合内容的一个“快照”，并从这个快照中返回元素；实际集合的内容可能在使用快照枚举时发生改变。**ConcurrentDictionary** 的 **GetEnumerator** 方法不获取它的内容的快照。因此，在枚举字典期间，字典的内容可能改变；这一点务必注意。还要注意的，**Count** 属性返回的是查询时集合中的元素数量。如果其他线程同时正在集合中增删元素，这个计数可能马上就变得不正确了。

ConcurrentStack、**ConcurrentQueue** 和 **ConcurrentBag** 这三个并发集合类都实现了 **IProducerConsumerCollection** 接口。接口定义如下：

```

public interface IProducerConsumerCollection<T> : IEnumerable<T>, ICollection, IEnumerable {
    Boolean TryAdd(T item);
    Boolean TryTake(out T item);
    T[] ToArray();
}

```

```
void CopyTo(T[] array, Int32 index);
}
```

实现了这个接口的任何类都能转变成一个阻塞集合。如果集合已满，那么负责生产(添加)数据项的线程会阻塞；如果集合已空，那么负责消费(移除)数据项的线程会阻塞。当然，我会尽量不使用这种阻塞集合，因为它们生命的全部意义就是阻塞线程。要将非阻塞的集合转变成阻塞集合，需要构造一个 **System.Collections.Concurrent.BlockingCollection** 类，向它的构造器传递对非阻塞集合的引用。**BlockingCollection** 类看起来像下面这样(有的方法未列出)：

```
public class BlockingCollection<T> : IEnumerable<T>, ICollection, IEnumerable, IDisposable {
    public BlockingCollection(IProducerConsumerCollection<T> collection,
        Int32 boundedCapacity);

    public void Add(T item);
    public Boolean TryAdd(T item, Int32 msTimeout, CancellationToken cancellationToken);
    public void CompleteAdding();

    public T Take();
    public Boolean TryTake(out T item, Int32 msTimeout, CancellationToken cancellationToken);

    public Int32 BoundedCapacity { get; }
    public Int32 Count { get; }
    public Boolean IsAddingCompleted { get; } // 如果调用了 CompleteAdding, 则为 true
    public Boolean IsCompleted { get; } // 如果 IsAddingCompleted 为 true, 而且 Count==0, 则返回 true

    public IEnumerable<T> GetConsumingEnumerable(CancellationToken cancellationToken);

    public void CopyTo(T[] array, int index);
    public T[] ToArray();
    public void Dispose();
}
```

构造一个 **BlockingCollection** 时，**boundedCapacity** 参数指出你想在集合中最多容纳多少个数据项。在基础集合已满的时候，如果一个线程调用 **Add**，生产线程就会阻塞。如果愿意，生产线程可调用 **TryAdd**，传递一个超时值(以毫秒为单位)和/或一个 **CancellationToken**，使线程一直阻塞，直到数据项成功添加、超时到期或者 **CancellationToken** 被取消(对 **CancellationToken** 类的讨论请参见第 27 章)。

BlockingCollection 类实现了 **IDisposable** 接口。调用 **Dispose** 时，这个 **Dispose** 会调用基础集合的 **Dispose**。它还会对类内部用于阻塞生产者和消费者的两个 **SemaphoreSlim** 对象进行清理。

生产者不再向集合添加更多的项时，生产者应调用 **CompleteAdding** 方法。这会向消费者发出信号，让它们知道不会再生产更多的项了。具体地说，这会造成正在使用 **GetConsumingEnumerable** 的一个 **foreach** 循环终止。下例演示了如何设置一个生产者/消费者环境，以及如何在完成数据项的添加之后发出通知：

```
public static void Main() {
    var bl = new BlockingCollection<Int32>(new ConcurrentQueue<Int32>());

    // 由一个线程池线程执行“消费”
```

```

ThreadPool.QueueUserWorkItem(ConsumeItems, bl);

// 在集合中添加 5 个数据项
for (Int32 item = 0; item < 5; item++) {
    Console.WriteLine("Producing: " + item);
    bl.Add(item);
}

// 告诉消费线程(可能有多个这样的线程), 不会在集合中添加更多的项了
bl.CompleteAdding();

Console.ReadLine(); // 这一行代码是出于测试的目的(防止 Main 返回)
}

private static void ConsumeItems(Object o) {
    var bl = (BlockingCollection<Int32>)o;
    // 阻塞, 直到出现一个数据项。出现后就处理它
    foreach (var item in bl.GetConsumingEnumerable()) {
        Console.WriteLine("Consuming: " + item);
    }

    // 集合空白, 没有更多的项进入其中
    Console.WriteLine("All items have been consumed");
}

```

在我自己的机器上执行上述代码, 得到以下输出:

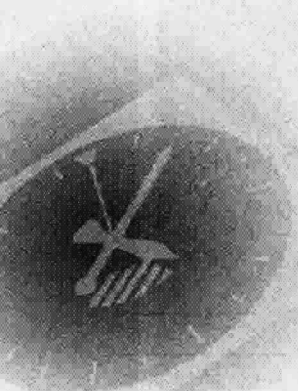
```

Producing: 0
Producing: 1
Producing: 2
Producing: 3
Producing: 4
Consuming: 0
Consuming: 1
Consuming: 2
Consuming: 3
Consuming: 4
All items have been consumed

```

你自己执行上述代码时, “Producing” 和 “Consuming” 行可能交错出现。但 “All items have been consumed” 这一行必然最后出现。

BlockingCollection 类还提供了静态 **AddToAny**, **TryAddToAny**, **TakeFromAny** 和 **TryTakeFromAny** 方法。所有这些方法都获取一个 **BlockingCollection<T>[]**, 以及一个数据项、一个超时值以及一个 **CancellationToken**。**(Try)AddToAny** 方法遍历数组中的所有集合, 直到发现因为容量还没有达到(还没有满), 而能够接受数据项的一个集合。**(Try)TakeFromAny** 方法则遍历数组中的所有集合, 直到发现一个能从中移除一个数据项的集合。



译者后记

从事软件开发的人，都是耐得住寂寞的人。Jeffrey 不仅耐得住寂寞，还在自己的专业领域取得了很高的造诣。取得了很高的造诣不说，他还愿意将自己的心得与大家分享。愿意和大家分享不说，他还非常实诚，真心想把自己的全部知识都清楚地交待给读者。字里行间，全是殷殷叮嘱。无浮夸之文字，倾心血而写就，近十年之所悟，尽呈现于本书。

读完这本书，你的心灵会受到极大的震撼。原因很简单，以前许多似懂非懂的概念，现在变得清晰明了；以前自以为是的做法，现在得到彻底纠正；以前艰苦摸索的编程技巧，现在如同 1+1 一样简单。

Jeffrey 最擅长的就是把最基本的东西讲清楚。你以前或许知道 1+1 等于 2，但他会把 1+1 为什么等于 2 讲得明明白白。最终你会有一种顿悟的感觉，然后自动地知道 1+2 等于几，2+2 等于几。不需要翻阅其他书籍来查询结果。

本书基于《CLR via C#》第 3 版，在保持上一版精华内容的同时进行了大量修订，并添加了大量关于 C# 5.0 新特性的内容。翻译这一版的过程中，我对上一版也进行了精校和勘误，许多勘误和改进意见都是由热心读者提交的。在此对他们表示衷心感谢！他们包括但不限于：56xue Castor DamnnnSure JeffFan Jeffrey You Kenny Wang LuckyDogFace, Neil, Titus, charlesgeorge, dream-like, fanfan, frank, hongkunjiang, huihui, kman, liu888bingsong, negy_liang, ooooo84, quark, sd, skyiv, wu, wukun0502, wzhou, yeahe83, 北纬 49° 的雨, 笨笨, 不世出的 CLR 小白, 大支猪与小苹果, 何苦, 来客心动, 飘荡, 飘飘白云, 气万象千, 千年一梦, 如水的人生, 施博, 酸菜鱼, 永远的阿哲

没有后期维护的书不算是好书。即使是本书英文原版，也维护了一份很长的勘误表，我本人也为其贡献良多。本书中文版将延续我一直以来坚持的风格，建立专门的页面进行维护，提供资源下载和勘误等服务。请大家继续前往我的博客(<http://transbot.blog.163.com>)，发表关于本书的意见和建议。

本书翻译过程中，参与人员有柏红羽、陈绍云、成荣静、文天山、刁灿群、黄洁、江卫、蒋昌友、李建、刘其星、刘琼、刘勇、毛越龙、莫永红、文瑞、施玉梅、叶昌元、游美波、张攀飞和周建超。尤其要感谢我的乖女儿周子衿。人生的足迹，终于来到高中这一重要阶段，且行且珍惜。

最后，如同往常一样，我要说所有的功劳都要归于作者，所有的错误都要归于译者。欢迎大家批评指正。本书最后将列出我所采用的术语，以帮助大家进一步理解。

下表列出本书使用的一些术语，有异于 MSDN 文档(以后简称“文档”)的会专门指出。

术语	说明
(S)Byte	等同于“SByte 和 Byte”，这是作者喜欢的说法。类似的还有 (U)Int16, (U)Int32, (U)IntPtr 等
AppDomain	(保留原文)
Compute-Bound 和 I/O-Bound	计算限制和 I/O 限制(一个操作如果因为处理器和 I/O 的限制而不得不等待，就称为计算限制或 I/O 限制的操作)
Windows Store app	Windows Store 应用(文档中翻译成“Windows 应用商店应用”，显得过于冗长)
action method	操作方法
antecedent task 和 continuation task	前置任务和延续任务
arity	元数。在计算机编程中，一个函数或运算(操作)的元数是指函数获取的实参或操作数的个数。它源于像 unary(arity=1)、binary(arity=2)、ternary(arity=3)这样的单词
asynchronously synchronization	异步地同步(同步对资源的访问，但以异步方式进行，即不阻塞线程)
atomic	原子性(读或写都是一次完成，别的线程看不到中间状态，就说这种读写是原子性的)
attribute	特性(以文档为准)
awaiter	等待者(调用 GetAwaiter 所返回的对象)
bit flag	位标志
block	阻塞(停下来等着)
callback	回调(回调方法简称为“回调”)
calling thread	调用线程(发出调用的线程，也称主调线程)
capture	捕捉(文档中主要用“捕捉”，偶尔用“捕获”)
cast	转型(不用文档的“强制类型转换”是因为太冗长)
compact	压缩(但此压缩非彼压缩，这里只是按照约定俗成的方式将 compact 翻译成“压缩”。不要以为“压缩”后内存会增多。相反，这里的“压缩”更接近于“碎片整理”。事实上，compact 正确的意思是“变得更紧凑”。但事实上，从上个世纪 80 年开始，人们就把它看成是 compress 的近义词而翻译成“压缩”，以讹传讹至今)
contract	文档的翻译非常混乱，包括协定、协议、合约、约定和契约等；本书使用“协定”
covariance 和 contravariance	协变和逆变(协变是指在要求使用一个类型的地方，能改为使用它的基类；逆变则是指在要求使用一个类型的地方，能改为使用它的派生类。C#用关键字 in 表示逆变量，用在输入位置；用 out 表示协变量，用在输出位置。详情参见 12.5 节)

术语	说明
culture	语言文化(而不是文档中的“区域性”)
cyclical reference	循环引用(我引用你, 你引用我)
declarative	声明性(文档如此, 个人更喜欢“宣告式”)
dispose	文档翻译成“释放”。但“dispose 一个对象”真正的意思是“清理或处置对象中包装的资源(比如它的字段引用的对象), 然后等着在一次垃圾回收之后回收该对象占用的托管堆内存(此时才释放)。”为避免误解, 本书将 dispose 翻译成“清理”, 偶尔也会保留原文。
entry	记录项(而不是“条目”、“入口”)
flush	文档翻译成“刷新”, 本书保留原文。其实 flush 在技术文档中的意思和日常生活中一样, 即“冲洗(到别处)”。例如, 我们会说“冲水”, 不会说“刷新水”
formatter	格式化器(文档是“格式化程序”)
get accessor method	get 访问器方法(取值函数或 getter)
guideline	设计规范
handler	处理程序。文档如此, 个人不喜欢“程序”二字。
helper method	辅助方法
host	寄宿(动词)或宿主(名词)
invoke 和 call	都翻译成“调用”, 但两者是有区别的。执行一个所有信息都已知的方法时, 用 call 比较恰当。但在需要先“唤出”某个东西来帮你调用一个信息不明的方法时, 用 invoke 就比较恰当。阅读关于委托和反射的章节时, 可以更好地体会两者的区别
literal	直接在代码中书写的值就是 literal 值, 比如字符串值和数值("Hello"和 123)。翻译成什么的都有, 包括直接量、字面值、文字常量、常值(台译)等。但实际最容易理解的还是英文原文。本书采用“字面值”
marshal	封送
metadata	元数据
mutex	互斥体
native method	本机方法(其实就是非托管方法)
native	本机(文档如此, 个人更喜欢“原生”, 比如原生类库、原生 C/C++ 代码、原生堆。一切非托管的, 就是 native 的)
operand	操作数(要操作/运算的目标)
operator	操作符(而不是文档中的“运算符”)
overload 和 override	重载和重写
preempt	抢占

术语	说明
primitive types	基元类型(文档如此, 不是“基本类型”。可以在代码中使用的最简单的构造就称为“基元”, 其他构造都是它们复合而成的)
provider	提供程序(文档如此, 个人不喜欢“程序”二字)
raise an event	引发事件
recursion count 和 recursive lock	递归计数和递归锁(可重入的锁就是递归锁, 重入的次数就是递归计数)
scalability	伸缩性(在少量时间里做更多工作的能力, 就是所谓的“伸缩性”。作为一个伸缩性好的服务器, 理论上应该 CPU 越多, 一个耗时操作所需的时间就越短。通俗地说, 在多个 CPU 之间并行执行, 执行时间将根据 CPU 的数量成比例地缩短)
self-hosted	自寄宿(应用程序的进程自己容纳 CLR, 就是所谓的自寄宿)
semaphore	信号量
set accessor method	set 访问器方法(赋值函数或 setter)
side effect	副作用
singleton	单实例(例如, 如果某类型在每个 AppDomain 中只能有一个实例, 它就是单实例类型)
spinning	自旋(线程不是阻塞, 而是原地“打转”, 浪费 CPU 时间。但在用于保护执行得非常快的代码区域时性能比较好)
string interning	字符串留用(而不是文档中的“字符串拘留”)
synchronous 和 asynchronous	同步和异步(同步意味着一个操作开始后必须等待它完成; 异步则意味着不用等它完成, 可以立即返回做其他事情。不要将“同步”理解成“同时”)
tap(点击), press and hold(长按), slide(滑动), swipe(轻扫), turn(转动), pinch(收缩)和 stretch(拉伸)	Windows 8 的各种触摸“手势”
throw an exception	抛出异常(而不是文档中的“引发异常”)
unwind	一般翻译成“展开”, 但这并不是一个很好的翻译。wind 和 unwind 源于生活。把线缠到线圈上称为 wind; 从线圈上松开称为 unwind。同样地, 调用方法时压入栈帧, 称为 wind; 方法执行完毕, 弹出栈帧, 称为 unwind
volatile	易变(文档将 volatile 翻译为“可变”。其实它是“短暂存在”、“易变”的意思, 因为可能有多个线程都对这种字段进行修改, 本书采用“易变”)
work item 和 worker thread	工作项和工作者线程(线程池术语。工作项是指要由一个线程池线程调用的方法, 代表线程实际要做的工作; 处理工作项的线程称为工作者线程。工作项被放到一个队列中, 工作者线程将工作项从队列中取出并处理)