线程同步：当一个线程执行递增和递减操作时，其他线程需要依次等待，这种常见的问题被称为线程同步。

解决线程同步问题的方式：

1. 大多数时候可以通过重新设计程序移除共享状态，从而去掉复杂的同步构造。
2. 使用原子操作。一个操作只占用一个量子时间，一次就可以完成。
3. 引入上下文切换，将等待线程置于阻塞状态，线程处于阻塞状态。

Ps：上下文切换：操作系统使用线程调度器保存线程状态并切换到另一个线程，依次恢复等待线程的状态。

用户模式：当执行应用程序代码的时候是用户模式，且每个程序有自己的独立空间，其他程序一般无法访问。

内核模式：当调用系统api函数时（系统调用），会暂停执行应用程序代码，转而执行内核代码（内核也是程序），进行内核空间的访问，这就是内核模式。

就线程执行，阻塞该线程，进行上下文切换是内核模式，操作系统分配CPU时间片；只是让线程进行等待（比如Sleep）是用户模式，因为没有执行上下文切换。

Ps：混合模式：线程执行时，混合模式尝试使用用户模式进行等待，当等待足够长时间时切换到阻塞状态节省CPU资源，使用的是内核模式。

原子操作：如果一条语句在底层处理器上被当作一个独立不可分割的指令，那么它本质上是原子的（atomic）。严格的原子性可以阻止任何抢占的可能。

执行原子操作：

Interlocked类提供线程安全的原子操作静态方法：

文本, 表格

描述已自动生成

这些方法的具体签名和重载：

[Interlocked 类 (System.Threading) | Microsoft Learn](https://learn.microsoft.com/zh-cn/dotnet/api/system.threading.interlocked?view=net-6.0)

Mutex类（互斥锁）：

Mutex和lock语句的区别：

1. 使用范围：lock 关键字是一种语言级别的同步机制，只能在单个应用程序域（application domain）内的同一进程中使用。它是用于对共享资源进行线程同步的简单方法，通常在较小的范围内使用。Mutex 是一个系统级别的同步对象，可以跨越多个应用程序域和进程。它提供了更大的灵活性，但也因此具有更高的开销。

2. 性能：lock 的性能通常较好，因为它是在应用程序域内执行的，并且不涉及系统级别的内核对象。Mutex 通常比 lock 的开销更大，因为它需要涉及到内核模式的操作系统调用，涉及用户态和内核态的切换。

3.使用灵活性：lock 通常用于对单个方法或代码块进行同步，它是一种隐式的同步机制。当你使用 lock 时，C# 编译器会为你生成相应的代码，以确保在同一时刻只有一个线程能够访问被 lock 保护的代码块。Mutex 提供了更大的灵活性，可以用于跨越多个代码块和多个线程的同步。通过 Mutex 的等待句柄，你可以在一个线程等待另一个线程释放 Mutex。

4.异常处理：使用 lock 时，如果在被锁定的代码块中发生异常，锁会自动释放，确保不会出现死锁的情况。使用 Mutex 时，你需要小心处理异常，以确保在发生异常时释放 Mutex。否则，可能会导致 Mutex 永远无法释放。

5.跨进程同步：Mutex 可以用于在不同进程之间同步线程，因为它是一个系统级别的同步对象。lock 不能用于跨进程同步，因为它是在应用程序域内实现的，不具备进程间通信的能力。

图形用户界面, 文本

中度可信度描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

文档：[Mutex 类 (System.Threading) | Microsoft Learn](https://learn.microsoft.com/zh-cn/dotnet/api/system.threading.mutex?view=net-6.0)

SemaphoreSilm类（信号量Semaphore类的混合模式版本）：

Ps：SemaphoreSilm类使用的是内核模式，这个类有一个老版本Semaphore类，使用的是内核模式。因为SemaphoreSlim不使用内核信号量因此SemaphoreSlim不支持进程间同步，需要在不同进程间同步线程需要使用Semaphore类。

信号量和互斥锁的区别：

1.信号量用于控制对多个相同资源的并发访问，而不仅仅是一个。信号量维护一个可用资源的计数，多个线程可以同时访问，但受到信号量计数的限制。。互斥锁只有两个状态表示资源是否被占用。

2.互斥锁适用临界区访问，信号量不适用于临界区访问。互斥锁操作是原子的。而信号量操作不是原子的。



图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 应用程序

描述已自动生成

文档：[SemaphoreSlim 类 (System.Threading) | Microsoft Learn](https://learn.microsoft.com/zh-cn/dotnet/api/system.threading.semaphoreslim?view=net-6.0)

AutoResetEvent类：

采用内核模式，使用该类来从一个线程向另一个线程发送通知，表示线程同步事件在一个等待线程释放后收到信号时自动重置。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 应用程序, Word

描述已自动生成

1.关于Reset方法：在没有线程被阻塞的情况下释放出了Set信号，下个线程在用到Waitone方法时将不会被阻塞，用此方法可以将状态置为阻塞，初始化为不阻塞，也可使用此方法重设。Reset 方法不会唤醒等待的线程。它仅仅是将 AutoResetEvent 的状态设置为未发信号的状态，等待的线程将继续等待直到另一个线程调用 Set 方法为止。

2.关于Set方法：发出一个释放信号，给最先被此对象阻塞的线程。只会释放一个被阻塞的线程

Ps：

1.AutoResetEvent初始化时，参数设置表示其第一次的使用是否已经是阻塞状态，true表示线程不等待，false表示线程需要等待。

2.每次调用AutoResetEvent的set方法将只会释放一个被阻止的线程，不会同时释放所有被此阻塞的线程。

3.AutoResetEvent的waitone方法不带参数将无休止等待一个被释放的信号，直到收到信号才继续执行。

文档：[AutoResetEvent 类 (System.Threading) | Microsoft Learn](https://learn.microsoft.com/zh-cn/dotnet/api/system.threading.autoresetevent?view=net-6.0)

AutoResetEvent例子：

文本

描述已自动生成

用Task Factory创建了一个线程，它调用了GetDataFromServer方法。调用该方法后，调用AutoResetEvent的WaitOne方法将主线程变为等待状态。在调用GetDataFromServer方法时，调用了AutoResetEvent对象的Set方法，它释放了主线程，并控制台打印输出dataFromServer方法返回的数据。

ManualResetEventSlim类：

ManualResetEventSlim和AutoResetEvent的区别：

1. ManualResetEventSlim是混合模式，AutoResetEvent是内核模式。前者性能更好。

2. ManualResetEvent 调用一次Set()后将允许恢复所有被阻塞线程，但必须手动调用Reset变为unsignal状态。AutoResetEvent只恢复一个线程，但会自动恢复unsignal状态。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

文档：[ManualResetEventSlim 类 (System.Threading) | Microsoft Learn](https://learn.microsoft.com/zh-cn/dotnet/api/system.threading.manualreseteventslim?view=net-6.0)

例子：

图形用户界面, 文本

描述已自动生成

CountDownEvent类：在收到一定次数的信号之后，将会解除对其等待线程的锁定。设置一个数量，每次调用Signal()方法减1，在信号数不为0时对Wait的调用会阻塞。

图形用户界面, 文本, 应用程序

描述已自动生成 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

文档：[CountdownEvent.Dispose 方法 (System.Threading) | Microsoft Learn](https://learn.microsoft.com/zh-cn/dotnet/api/system.threading.countdownevent.dispose?view=net-6.0" \l "system-threading-countdownevent-dispose)

例子：

图形用户界面, 文本

描述已自动生成

Barrier类：

作用：使多个任务能够采用并行方式依据某种算法在多个阶段中协同工作。一组任务通过一系列阶段进行协作，其中组中的每个任务都表示已到达 Barrier 给定阶段，并隐式等待其他人到达。同一 Barrier 个阶段可用于多个阶段。当指定数量的方法都调用SignalAndWait后执行依次回调方法

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

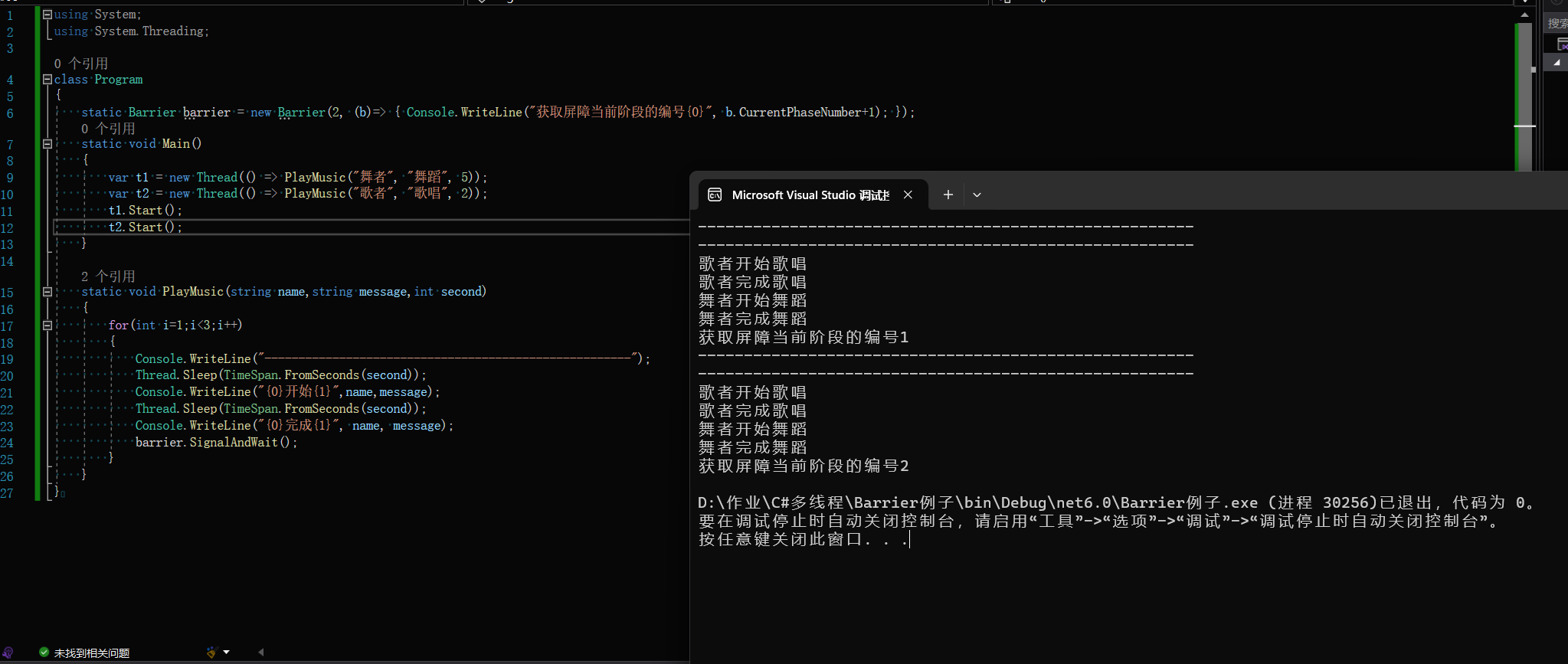


图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

文档：[Barrier 类 (System.Threading) | Microsoft Learn](https://learn.microsoft.com/zh-cn/dotnet/api/system.threading.barrier?view=net-6.0)

例子：



ReaderWriterLockSlim类：

读写锁作用：在多线程中对一个集合进行读写操作，其中读操作允许多个线程同时读取，但写操作只允许一个线程操作写入。当某个线程进入读取模式时，此时其他线程依然能进入读取模式，假设此时一个线程要进入写入模式，那么他不得不被阻塞直到读取模式退出为止。同样的，如果某个线程进入了写入模式，那么其他线程无论是要写入还是读取，都是会被阻塞的。

[如何使用C#读写锁ReaderWriterLockSlim\_c# 加锁 多线程 同时读和写\_lfw2019的博客-CSDN博客](https://blog.csdn.net/ABC13222880223/article/details/121711991)

图形用户界面, 文本, 表格

中度可信度描述已自动生成

图形用户界面, 应用程序

描述已自动生成

应用程序

中度可信度描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

文档：[ReaderWriterLockSlim 类 (System.Threading) | Microsoft Learn](https://learn.microsoft.com/zh-cn/dotnet/api/system.threading.readerwriterlockslim?view=net-6.0)

升级锁（Lock Upgrade）：

升级锁指的是将当前线程持有的低级别锁（例如读锁）升级为更高级别锁（例如写锁）。这个过程通常是在当前线程已经获取了低级别锁的情况下，发现需要执行一些写操作时才会进行。升级锁的问题在于可能引起死锁。如果一个线程在持有读锁的情况下试图升级为写锁，并且在此过程中其他线程持有写锁，就会发生死锁。因此，某些锁的实现可能不支持升级锁操作，而采用降级锁的方式。

降级锁（Lock Downgrade）：

降级锁是指将当前线程持有的高级别锁（例如写锁）降级为更低级别锁（例如读锁）。这通常发生在当前线程已经获取了写锁的情况下，发现在一些操作后可以释放写锁并获取读锁，从而允许其他线程也能够访问共享资源。降级锁的主要优势在于减小了锁的粒度，提高了并发性能。这样，其他线程可以在读锁的保护下访问共享资源，而不必等待写锁释放。

读写锁的递归计数：写锁（ReaderWriterLockSlim 类）具有递归计数的特性。递归计数是指允许同一个线程多次获取写锁或读锁，而不会造成死锁，并且需要相同数量的释放操作才能解锁。

读写锁的递归计数存在的意义是什么？什么情况下需要用到递归计数的读写锁？

SpinWait结构：

System.Threading.SpinWait 是一个轻量同步类型，可以在低级别方案中使用它来避免内核事件所需的高开销的上下文切换和内核转换。

使用SpinWait好处：

对于只需要短暂等待的场景，SpinWait在用户模式下就能结束等待，然后继续运行，这时线程不会因为短暂等待而发生上下文切换（因为等待的线程不会让出cpu的时间片），这样就可以提高cpu的利用率，因为有时候上下文切换所需的时间比等待的时间还要久；

对于有些确实是需要长时间等待的场景，SpinWait如果在用户模式下还没有完成等待，就会切换到内核模式进行等待，这时线程会让出cpu的时间片给其他线程，会强制触发上下文切换，这样就不会因为长时间的等待而白白占用cpu。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

文档：[SpinWait 结构 (System.Threading) | Microsoft Learn](https://learn.microsoft.com/zh-cn/dotnet/api/system.threading.spinwait?view=net-6.0)