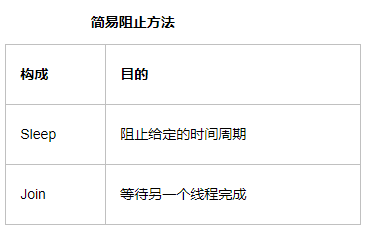
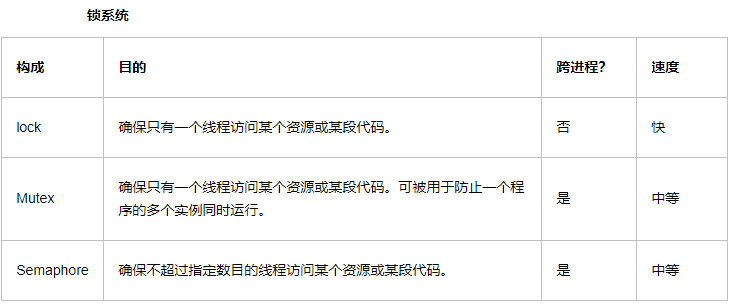
线程同步基础

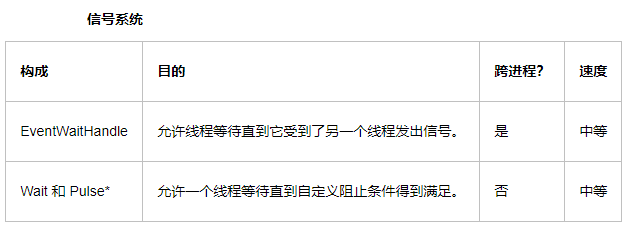
1. 同步

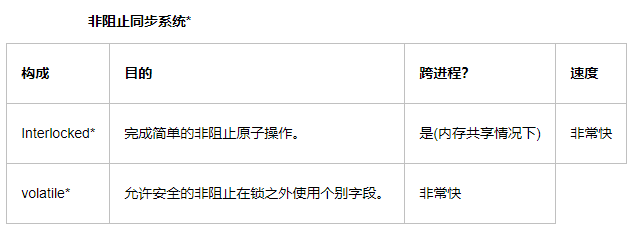
下面的表格列展了.NET对协调或同步线程动作的可用的工具：





同步情况下也提供自动锁。





Timeout类的用法

Timeout.Infinite作为sleep或join的参数，意为无限期超时间间隔

* 1. 阻止（blocking）线程被暂停,直到某一个条件发生才能停止被暂停

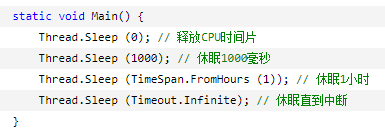
当一个线程通过上面所列的方式处于**等待或暂停**的状态，被称为被阻止。一旦被阻止，线程立刻放弃它被分配的CPU时间，将它的ThreadState属性添加为WaitSleepJoin状态，**不在安排时间**直到**停止阻止**。停止阻止在任意四种情况下发生：

* 阻止的条件已得到满足
* 操作超时
* 通过Thread.Interrupt中断了
* 通过Thread.Abort放弃了

当线程通过（不建议）Suspend方法暂停，不认为是被阻止了。

* 1. 休眠 和 轮询 线程被暂停,直到设置的时间到达后停止被暂停

调用Thread.Sleep阻止当前的线程指定的时间（或者直到中断）



更确切的说，Thread.Sleep放弃了占用CPU,请求不在被分配时间直到给定的时间经过.Thread.sleep(0)放弃CPU的时间刚好够其它在时间片队列里的活动线程被执行.

线程类同时也提供了一个SpinWait方法,它使用轮询CPU时间的方式,保持给定的迭代次数进行”无用地繁忙”.50迭代可能等同于停顿大约一微秒,虽然这将取决于CPU的速度和负载.从技术上讲,SpinWait并不是一个阻止的方法:一个处于spin\_wait的线程的ThreadState不是WaitSleepJoin状态,并且也不会被其他的线程过早的中断.SpinWait很少被使用,它的作用是等待一个在极短时间(可能小于一微秒)内准备好的可预期的资源.而不用调用Sleep方法阻止线程而浪费CPU时间.不过,这种技术的优势只有在多处理器计算机:对单一处理器的电脑,直到轮询的线程结束了它的时间片之前,一个资源没有机会改变状态,这有违它的初衷.并且调用SPinWait经常会花费较长的时间这本身就浪费了CPU时间.

**自我总结:**

(SpinWait方法主要是用于某些线程所需等待的资源很快就能实现,而使得这些线程处于自旋的状态(即假装很忙,而不放弃CPU的时间片),目的是不希望发生昂贵的上下文切换和内核切换带来的高开销.

SpinWait实质上会将处理器至于十分紧密的循环中,其循环计数有iteration参数指定.因此等待的时间取决于处理器的速度.

长时间的自旋不是很好的做法,因为自旋会阻塞更高级的线程极其相关的任务,还会阻塞垃圾回收机制.)

* 1. 使用join等待一个线程完成

**你可以通过join方法阻止线程直到另一个线程结束:(blocks the calling thread until a thread terminates)**

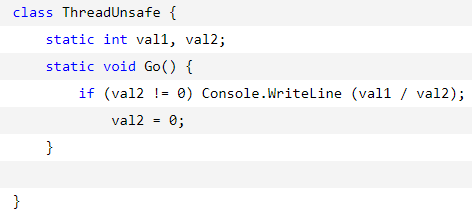
**当a thread使用join方法时，调用a thread的calling thread就会等待a thread 运行结束之后再继续运行**。



Join方法接收一个使用毫秒或用TimeSpan类的超时参数，当join超时则返回false，如果线程已终止，则返回true。

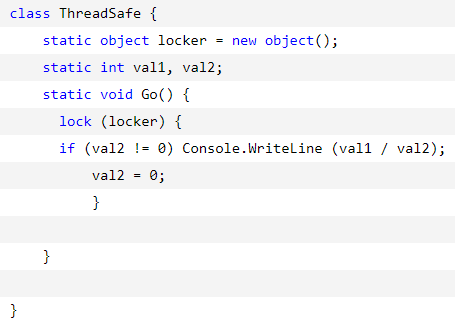
1. 锁和线程安全

锁实现互斥的访问，被用于确保在同一时刻只有一个线程可以进入特殊的代码片段。举例如下：



这不是线程安全的：如果Go方法同时被两个线程调用，可能会得到在某个线程中除数为零的错误，因为val2可能被一个线程设置为零，而另一个线程刚好执行到Console.Writeline语句。

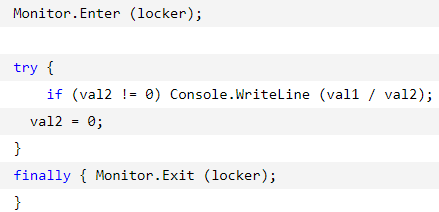
下面用lock来修正这个问题：



在同一时刻只有一个线程可以锁定同步对象（即locker），任何竞争的其他线程都会被阻止，直到这个锁被释放。如果有大于一个的线程竞争这个锁那么他们将形成就绪队列，以先到先得的方式授权锁。

一个等待竞争锁的线程被阻止将在ThreadState上为WaitSleepJoin状态。

C#的lock语句实际上是调用Monitor.Enter和Monitor.Exit,中间夹杂try-finally语句的简略版.



在同一个对象locker上,在调用第一个Monitor.Enter之前而先调用Monitor.Exit将引发异常.

* 1. 选择同步对象