

C#多线程下的调优 - 包子wxi - 博客园

 cnblogs.com/wei325/p/16065342.html

[回到顶部](#)

一、原子操作

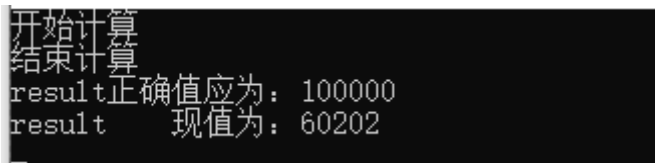
先看一段问题代码



```
/// <summary>
/// 获取自增
/// </summary>
public static void GetIncrement()
{
    long result = 0;
    Console.WriteLine("开始计算");
    //10个并发执行
    Parallel.For(0, 10, (i) =>
    {
        for (int j = 0; j < 10000; j++)
        {
            result++;
        }
    });
    Console.WriteLine("结束计算");
    Console.WriteLine($"result正确值应为: {10000 * 10}");
    Console.WriteLine($"result    现值为: {result}");
    Console.ReadLine();
}
```



这是多线程下，result的值不同步的原因。



```
开始计算
结束计算
result正确值应为: 100000
result    现值为: 60202
```

1.基于Lock实现

平时大家用的最多的应该就是加锁了，同一时间，只有一个线程进入代码块。

实现代码：



```

private static Object _obj = new object();

    /// <summary>
    /// 原子操作基于Lock实现
    /// </summary>
    public static void AtomicityForLock()
    {
        long result = 0;
        Console.WriteLine("开始计算");
        //10个并发执行
        Parallel.For(0, 10, (i) =>
        {
            //lock锁
            lock (_obj)
            {
                for (int j = 0; j < 10000; j++)
                {
                    result++;
                }
            }
        });
        Console.WriteLine("结束计算");
        Console.WriteLine($"result正确值应为: {10000 * 10}");
        Console.WriteLine($"result    现值为: {result}");
        Console.ReadLine();
    }
}

```



结果：

```

开始计算
结束计算
result正确值应为: 100000
result    现值为: 100000

```

2.基于CAS实现

CAS是一种有名的无锁算法。无锁编程，即不适用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步。

CAS在.NET中的实现类是Interlocked，内部提供很多原子操作的方法，最终都是调用Interlocked.CompareExchange(ref out,更新值，期望值) //基于内存屏障的方式操作（七个步骤）

说到线程安全，不要一下子就想到加锁，尤其是可能会调用频繁或者是要求高性能的场合。

- CAS（Compare And Swap）比较并替换，是线程并发运行时用到的一种技术
- CAS是原子操作，保证并发安全，而不能保证并发同步
- CAS是CPU的一个指令（需要JNI调用Native方法，才能调用CPU的指令）
- CAS是非阻塞的、轻量级的乐观锁

CAS的适用场景

读多写少：如果有大量的写操作，CPU开销可能会过大，因为冲突失败后会不断重试（自旋），这个过程中会消耗CPU

单个变量原子操作：CAS机制所保证的只是一个变量的原子操作，而不能保证整个代码块的原子性，比如需要保证三个变量共同进行原子性的更新，就不得不使用悲观锁了

Interlocked主要函数如下：

Interlocked.Increment 原子操作，递增指定变量的值并存储结果。
Interlocked.Decrement 原子操作，递减指定变量的值并存储结果。
Interlocked.Add 原子操作，添加两个整数并用两者的和替换第一个整数
Interlocked.Exchange 原子操作，赋值
Interlocked.CompareExchange(ref a, b, c); 原子操作，a参数和c参数比较，相等b替换a，不相等不替换。方法返回值始终是第一个参数的原值，也就是内存里的值

用Interlocked.Increment实现上面自增功能

代码：



```
/// <summary>
/// 自增CAS实现
/// </summary>
public static void AtomicityForInterLock()
{
    long result = 0;
    Console.WriteLine("开始计算");
    Parallel.For(0, 10, (i) =>
    {
        for (int j = 0; j < 10000; j++)
        {
            //自增
            Interlocked.Increment(ref result);
        }
    });
    Console.WriteLine($"结束计算");
    Console.WriteLine($"result正确值应为：{10000 * 10}");
    Console.WriteLine($"result    现值为：{result}");
    Console.ReadLine();
}
```



结果：

Interlocked下原子操作的方法最终都是调用Interlocked.CompareExchange(ref a, b, c)实现的，现在我们利用CompareExchange自己实现一个原子操作功能

实现“一个变量自增到10000，然后又初始化到1开始自增的功能”

代码：

```
开始计算
结束计算
result正确值应为: 100000
result    现值为: 100000
```



```

/// <summary>
/// 基于CAS原子操作自己写
/// </summary>
public static void AtomicityForMyCalc()
{
    long result = 0;
    Console.WriteLine("开始计算");

    Parallel.For(0, 10, (i) =>
    {
        long init = 0;
        long incrementNum = 0;
        for (int j = 0; j < 10000; j++)
        {
            do
            {
                init = result;
                incrementNum = result + 1;
                incrementNum = incrementNum > 10000 ? 1 : incrementNum; //自
增到10000后初始化成1

            }
            //如果result=init,则result的值被incrementNum替换,否则result不变,返
回的是result的原始值
            while (init != Interlocked.CompareExchange(ref result,
incrementNum, init));
            if(incrementNum==10000)
            {
                Console.WriteLine($"自增到达10000啦!值被初始化为1");
            }
        }
    });
    Console.WriteLine($"结束计算");

    Console.WriteLine($"result正确值应为: {10000}");
    Console.WriteLine($"result    现值为: {result}"

);

    Console.ReadLine();
}

```



结果：

[illegible]

3.自旋锁SpinLock

自旋锁 (spinlock) :

是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环。

什么情况下使用自旋锁：

自旋锁非常有助于避免阻塞，但是如果预期有大量阻塞，由于旋转过多，您可能不应该使用自旋锁。当锁是细粒度的并且数量巨大（例如链接的列表中每个节点一个锁）时以及锁保持时间总是非常短时，旋转可能非常有帮助。

短时间锁定的情况下，自旋锁（spinlock）更快。（因为自旋锁本质上不会让线程休眠，而是一直循环尝试对资源访问，直到可用。所以自旋锁线程被阻塞时，不进行线程上下文切换，而是空转等待。对于多核CPU而言，减少了切换线程上下文的开销，从而提高了性能。）如果机器单核或锁定时间长的要避免使用，因为占有着逻辑核心会导致其他的线程也不可用。

SpinLock和Lock的区别：

SpinLock，自旋锁。尝试获取该锁的线程持续不断的check是否可以获得。此时线程仍然是激活状态，只是在空转，浪费cpu而已。但是spinlock避免了线程调度和上下文切换，如果锁的时间极短的话，使用该锁反而效率会高。

而lock是线程被block了。这将引起线程调度和上下文切换等行为。

示例：



```

//创建自旋锁
private static SpinLock spin = new SpinLock();
public static void Spinklock()
{
    Action action = () =>
    {
        bool lockTaken = false;
        try
        {
            //申请获取锁
            spin.Enter(ref lockTaken);
            //临界区
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine($"当前线程
{Thread.CurrentThread.ManagedThreadId.ToString()},输出:1");
            }
        }
        finally
        {
            //工作完毕，或者产生异常时，检测一下当前线程是否占有锁，如果有了锁释放它
            //避免出行死锁
            if(lockTaken)
            {
                spin.Exit();
            }
        }
    };

    Action action2 = () =>
    {
        bool lockTaken = false;
        try
        {
            //申请获取锁
            spin.Enter(ref lockTaken);
            //临界区
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine($"当前线程
{Thread.CurrentThread.ManagedThreadId.ToString()},输出:2");
            }
        }
        finally
        {
            //工作完毕，或者产生异常时，检测一下当前线程是否占有锁，如果有了锁释放它
            //避免出行死锁
            if (lockTaken)
            {
                spin.Exit();
            }
        }
    };

    //并行执行2个action
    Parallel.Invoke(action, action2);
}

```



结果：

申请锁下面的临界区保证是顺序执行的，不会因为多线程穿插输出。

```
当前线程4,输出:2
当前线程4,输出:2
当前线程4,输出:2
当前线程4,输出:2
当前线程4,输出:2
当前线程4,输出:2
当前线程4,输出:2
当前线程4,输出:2
当前线程4,输出:2
当前线程4,输出:2
当前线程4,输出:2
当前线程1,输出:1
当前线程1,输出:1
当前线程1,输出:1
当前线程1,输出:1
当前线程1,输出:1
当前线程1,输出:1
当前线程1,输出:1
当前线程1,输出:1
当前线程1,输出:1
当前线程1,输出:1
当前线程1,输出:1
当前线程1,输出:1
```

4.读写锁ReaderWriterLockSlim

- 读写锁是一个具有特殊用途的线程锁，适用于频繁读取且读取需要一定时间的场景，共享资源的读取操作通常是可以同时执行的，
- 普通的互斥锁不管是获取还是修改操作无法同时执行，如果多个线程为了读取操作而获取互斥锁，那么同一时间只有一个线程可以执行读取操作，
- 频繁读取的场景下会对吞吐量造成影响
- 读写锁把锁分为读取锁和写入锁，线程可以根据对共享资源的操作类型获取读取锁还是写入锁，读取锁可以被多个线程同时获取，写入锁不可以被多个线程同时获取，且读取锁和写入锁不可以被不同的线同时获取。

操作	读取锁状态	写入锁状态	获取锁是否需要等待
获取读取锁	未被获取	未被获取	无需等待
获取读取锁	已被其他线程获取	未获取	无需等待
获取读取锁	未被获取	已被其他线程获取	需要等待其他线程释放
获取写入锁	未被获取	未被获取	无需等待
获取写入锁	已被其他线程获取	未被获取	需要等待其他线程释放
获取写入锁	未被获取	已被其他线程获取	需要等待其他线程释放

代码示例：




```

//读写锁， //策略支持递归
private static ReaderWriterLockSlim rwl = new
ReaderWriterLockSlim(LockRecursionPolicy.SupportsRecursion);
private static int index = 0;
static void read()
{
    try
    {
        //进入读锁
        rwl.EnterReadLock();
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine($"线程id:
{Thread.CurrentThread.ManagedThreadId}, 读数据, 读到index:{index}");
        }
    }
    finally
    {
        //退出读锁
        rwl.ExitReadLock();
    }
}
static void write()
{
    try
    {
        //尝试获写锁
        while (!rwl.TryEnterWriteLock(50))
        {
            Console.WriteLine($"线程id:
{Thread.CurrentThread.ManagedThreadId}, 等待写锁");
        }
        Console.WriteLine($"线程id:{Thread.CurrentThread.ManagedThreadId},
获取到写锁");
        for (int i = 0; i < 5; i++)
        {
            index++;
            Thread.Sleep(50);
        }
        Console.WriteLine($"线程id:{Thread.CurrentThread.ManagedThreadId},
写操作完成");
    }
    finally
    {
        //退出写锁
        rwl.ExitWriteLock();
    }
}

/// <summary>
/// 执行多线程读写
/// </summary>
public static void test()
{
    var taskFactory = new TaskFactory(TaskCreationOptions.LongRunning,
TaskContinuationOptions.None);
    Task[] task = new Task[6];
    task[1] = taskFactory.StartNew(write); //写
    task[0] = taskFactory.StartNew(read); //读
}

```

```

task[2] = taskFactory.StartNew(read); //读
task[3] = taskFactory.StartNew(write); //写
task[4] = taskFactory.StartNew(read); //读
task[5] = taskFactory.StartNew(read); //读

for (var i=0; i<6; i++)
{
    task[i].Wait();
}
}

```



```

线程id:4, 获取到写锁
线程id:7, 等待写锁
线程id:7, 等待写锁
线程id:7, 等待写锁
线程id:7, 等待写锁
线程id:7, 等待写锁
线程id:4, 写操作完成
线程id:7, 获取到写锁
线程id:7, 写操作完成
线程id:10, 读数据, 读到index:10
线程id:10, 读数据, 读到index:10
线程id:8, 读数据, 读到index:10
线程id:5, 读数据, 读到index:10
线程id:5, 读数据, 读到index:10
线程id:5, 读数据, 读到index:10
线程id:10, 读数据, 读到index:10
线程id:10, 读数据, 读到index:10
线程id:10, 读数据, 读到index:10
线程id:5, 读数据, 读到index:10
线程id:6, 读数据, 读到index:10
线程id:8, 读数据, 读到index:10
线程id:8, 读数据, 读到index:10
线程id:8, 读数据, 读到index:10
线程id:5, 读数据, 读到index:10
线程id:6, 读数据, 读到index:10
线程id:6, 读数据, 读到index:10
线程id:6, 读数据, 读到index:10
线程id:8, 读数据, 读到index:10
线程id:6, 读数据, 读到index:10

```

可以看到，在线程4写入期间，线程7是等待线程4写完再写，线程7写期间其它线程也没有操作，等写完后，读的操作是多个线程交叉的。

适合的场景举例：

- 多线程写文件，多线程并发写文件时，会报资源被占用错误，用这里的写锁就可以独占资源写完再到下一个线程写。
- 本地缓存的读写操作，几个缓存值写完才能读出来，防止读到不完整数据。

[回到顶部](#)

二、线程安全

1.线程安全集合

BlockingCollection：一个支持界限和阻塞的容器（线程安全集合），与队列结构相似，常用函数如下

Add：向容器中插入元素

TryTake：从容器中取出元素并删除

TryPeek：从容器中取出元素，但不删除。

CompleteAdding：告诉容器，添加元素完成。此时如果还想继续添加会发生异常。

IsCompleted：告诉消费线程，生产者线程还在继续运行中，任务还未完成。

普通用法示例：



```

/// <summary>
/// 线程安全集合用法
/// </summary>
public static void BC()
{
    //线程安全集合
    using (BlockingCollection<int> blocking = new BlockingCollection<int>
())
    {
        int NUMITEMS = 10000;

        for (int i = 1; i < NUMITEMS; i++)
        {
            blocking.Add(i);
        }
        //完成添加
        blocking.CompleteAdding();

        int outerSum = 0;

        // 定义一个委托方法取出集合元素
        Action action = () =>
        {
            int localItem;
            int localSum = 0;

            //取出并删除元素，先进先出
            while (blocking.TryTake(out localItem))
            {
                localSum += localItem;
            }
            //两数相加替换第一个值
            Interlocked.Add(ref outerSum, localSum);
        };

        //并行3个线程执行，多个线程同时取集合的数据
        Parallel.Invoke(action, action, action);

        Console.WriteLine($"0+...{NUMITEMS-1} = {((NUMITEMS * (NUMITEMS -
1)) / 2)},输出结果: {outerSum}");
        //此集合是否已标记为已完成添加且为空
        Console.WriteLine($"线程安全集合.IsCompleted=
{blocking.IsCompleted}");
    }
}

```



结果：

限制集合长度示例：



```

0+...9999 = 49995000,输出结果: 49995000
线程安全集合.IsCompleted=True

```

```

/// <summary>
/// 限制集合长度
/// </summary>
public static void BCLimtLength()
{
    //限制集合长度为5个，后面进的会阻塞等集合少于5个再进来
    BlockingCollection<int> blocking = new BlockingCollection<int>(5);

    var task1= Task.Run(() =>
    {
        for (int i = 0; i < 20; i++)
        {
            blocking.Add(i);
            Console.WriteLine($"集合添加:({i})");
        }

        blocking.CompleteAdding();
        Console.WriteLine("完成添加");
    });

    // 延迟500ms执行等待先生产数据
    var task2 = Task.Delay(500).ContinueWith((t) =>
    {
        while (!blocking.IsCompleted)
        {
            var n = 0;
            if (blocking.TryTake(out n))
            {
                Console.WriteLine($"取出:({n})");
            }
        }

        Console.WriteLine("IsCompleted = true");
    });

    Task.WaitAll(task1, task2);
}

```



结果：

在BlockingCollection中使用Stack(栈，先进后出)示例：



```

/// <summary>
/// 线程安全集合，先进后出
/// </summary>
public static void BCStack()
{
    //线程安全集合，参数表明栈标识，队列长度为5
    BlockingCollection<int> blocking = new
BlockingCollection<int>(new ConcurrentStack<int>(), 5);

    var task1 = Task.Run(() =>
    {
        for (int i = 0; i < 20; i++)
        {
            blocking.Add(i);
            Console.WriteLine($"集合添加:({i})");
        }

        blocking.CompleteAdding();
        Console.WriteLine("完成添加");
    });

    // 等待先生产数据
    var task2 = Task.Delay(500).ContinueWith((t)
=>
    {
        while (!blocking.IsCompleted)
        {
            var n = 0;
            if (blocking.TryTake(out n))
            {
                Console.WriteLine($"取出:({n})");
            }
        }

        Console.WriteLine("IsCompleted = true");
    });

    Task.WaitAll(task1, task2);
}

```

```

集合添加:(0)
集合添加:(1)
集合添加:(2)
集合添加:(3)
集合添加:(4)
集合取出:(0)
集合添加:(5)
集合取出:(1)
集合取出:(2)
集合取出:(3)
集合添加:(6)
集合添加:(7)
集合添加:(8)
集合添加:(9)
集合取出:(4)
集合取出:(5)
集合取出:(6)
集合取出:(7)
集合添加:(10)
集合取出:(8)
集合添加:(11)
集合取出:(9)
集合添加:(12)
集合添加:(13)
集合添加:(14)
集合添加:(15)

```



```
集合添加:(0)
集合添加:(1)
集合添加:(2)
集合添加:(3)
集合添加:(4)
取出:(4)
集合添加:(5)
取出:(5)
集合添加:(6)
取出:(6)
取出:(7)
取出:(3)
取出:(2)
取出:(1)
取出:(0)
集合添加:(7)
取出:(8)
集合添加:(8)
取出:(9)
集合添加:(9)
```

一开始入了0-4，从最后的4开始取。

2.线程安全字典

ConcurrentDictionary :这个比较好理解，普通字典多线程并发时添加时会报错，而这个则是线程安全的，不会报错。

普通字典示例：



```

//普通字典
private static IDictionary<string, string> Dictionaries { get; set; } =
new Dictionary<string, string>();
/// <summary>
/// 字典增加值
/// </summary>
public static void AddDictionaries()
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    //并发1000个线程写
    Parallel.For(0, 1000, (i) =>
    {
        var key = $"key-{i}";
        var value = $"value-{i}";

        // 不加锁会报错
        // lock (Dictionaries)
        // {
            Dictionaries.Add(key, value);
        // }
    });
    sw.Stop();
    Console.WriteLine("Dictionaries 当前数据量为: {0}",
Dictionaries.Count);
    Console.WriteLine("Dictionaries 执行时间为: {0} ms",
sw.ElapsedMilliseconds);
}

```



不加锁时结果：

```

/// <summary>
/// 字典增加值
/// </summary>
1 个引用
public static void AddDictionaries()
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    //并发100个线程写
    Parallel.For(0, 1000, (i) =>
    {
        var key = $"key-{i}";
        var value = $"value-{i}";

        // 不加锁会报错
        // lock (Dictionaries)
        // {
            Dictionaries.Add(key, value);
        // }
    });
    sw.Stop();
    Console.WriteLine("Dictionaries 当前数
    Console.WriteLine("Dictionaries 执行时
}
/// <summary>
/// 线程安全字典增加值
/// </summary>

```

用户未处理的异常

System.ArgumentException: "Destination array was not long enough. Check the destination index, length, and the array's lower bounds."

[查看详细信息](#) | [复制详细信息](#) | [启动 Live Share 会话...](#)

▶ 异常设置

加锁后：

线程安全字典示例：



```
//线程安全字典
private static
IDictionary<string, string>
ConcurrentDictionaries { get; set;
} = new
ConcurrentDictionary<string,
string>();

/// <summary>
/// 线程安全字典添加值
/// </summary>
public static void
AddConcurrentDictionaries()
{
    Stopwatch sw = new
Stopwatch();
    sw.Start();
    //并发1000个线程写
    Parallel.For(0, 1000,
(i) =>
    {
        var key = $"key-
{i}";
        var value =
$"value-{i}";

        // 无须加锁

ConcurrentDictionaries.Add(key,
value);

    });
    sw.Stop();

Console.WriteLine("ConcurrentDictic
    当前数据量为： {0}",
ConcurrentDictionaries.Count);

Console.WriteLine("ConcurrentDictic
    执行时间为： {0} ms",
sw.ElapsedMilliseconds);
}
```



```
ConcurrentDictionaries 当前数据量为： 1000
ConcurrentDictionaries 执行时间为： 89 ms
```

可以看到，线程安全字典比普通字典性能略好，且线程安全字典无需加锁。

[回到顶部](#)

三、线程池

1.通过QueueUserWorkItem启动工作者线程

ThreadPool线程池中有两个重载的静态方法可以直接启动工作者线程：

- ThreadPool.QueueUserWorkItem(waitCallback)；
- ThreadPool.QueueUserWorkItem(waitCallback,Object)；

先把WaitCallback委托指向一个带有Object参数的无返回值方法，再使用ThreadPool.QueueUserWorkItem(WaitCallback)就可以一步启动此方法，此时异步方法的参数被视为null。

示例1：



```
public class ThreadLoopDemo
{
    /// <summary>
    /// 回调方法
    /// </summary>
    /// <param name="obj"></param>
    static void CallMethod(object state)
    {
        Console.WriteLine("RunWorkerThread开始工作");
        Console.WriteLine("工作者线程启动成功!");
    }

    public static void Test()
    {
        //工作者线程最大数目，I/O线程的最大数目
        ThreadPool.SetMaxThreads(1000, 1000);

        //启动工作者线程
        ThreadPool.QueueUserWorkItem(new WaitCallback(CallMethod!));
        Console.ReadKey();
    }
}
```

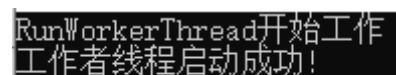


执行Test方法，结果：

示例2：

使用第二个重载方法

ThreadPool.QueueUserWorkItem(WaitCallback,object)方法可以把object对象作为参数传送到回调函数中。



```

public class ThreadLoopDemo
{
    /// <summary>
    /// 回调方法
    /// </summary>
    /// <param name="obj"></param>
    static void CallMethod(object state)
    {
        Console.WriteLine("RunWorkerThread开始工作");
        Order order=state as Order;
        Console.WriteLine($"orderName:{order.orderName},price:{order.price}");
        Console.WriteLine("工作者线程启动成功!");
    }

    public static void Test()
    {
        //工作者线程最大数目，I/O线程的最大数目
        ThreadPool.SetMaxThreads(1000, 1000);
        Order order = new Order()
        {
            orderName = "冰箱",
            price = 1888
        };
        //启动工作者线程
        ThreadPool.QueueUserWorkItem(new WaitCallback(CallMethod!),order);
        Console.ReadKey();
    }
}

public class Order
{
    public string orderName { get; set; }
    public decimal price { get; set; }
}

```



执行Test方法，结果：

通过ThreadPool.QueueUserWork启动工作者线程非常方便，但是WaitCallback委托指向的必须是一个带有object参数的无返回值方法。

```

RunWorkerThread开始工作
orderName:冰箱,price:1888
工作者线程启动成功!

```

线程池还可以重用线程，比喻可以吧线程池大小设为5个，去执行一批任务，防止大量创建新线程损耗大量cpu。

所以这个方法启动的工作者线程仅仅适合于带单个参数和无返回值的情况。

2.线程池等待（信号量）

ThreadPool并没有Thread的Join等待接口，那么想让ThreadPool等待要这么做呢？

ManualResetEvent:通知一个或多个正在等待的线程已发生的事件，相当于发送了一个信号

mre.WaitOne:卡住当前主线程，一直等到信号修改为true的时候，才会接着往下跑



```
public class ThreadLoopDemo
{
    /// <summary>
    /// 执行
    /// </summary>
    public static void Test()
    {
        //用来控制线程等待, false默认为关闭状态
        ManualResetEvent mre = new ManualResetEvent(false);
        ThreadPool.QueueUserWorkItem(p =>
        {
            Console.WriteLine("线程1开始");
            Thread.Sleep(1000);
            Console.WriteLine($"线程1结束, 带参数,
{Thread.CurrentThread.ManagedThreadId}");
            //通知线程, 修改信号为true
            mre.Set();
        });

        //阻止当前线程, 直到等到信号为true在继续执行
        mre.WaitOne();

        //关闭线程, 相当于设置成false
        mre.Reset();
        Console.WriteLine("信号被关闭了");

        ThreadPool.QueueUserWorkItem(p =>
        {
            Console.WriteLine("线程2开始");
            Thread.Sleep(2000);
            Console.WriteLine("线程2结束");
            mre.Set();
        });

        mre.WaitOne();
        Console.WriteLine("主线程结束");
    }
}
```



执行Test, 结果:

```
线程1开始
线程1结束, 带参数, 4
信号被关闭了
线程2开始
线程2结束
主线程结束
```

3.Task

`Task.Factory.StartNew`:创建一个新的线程，`Task`的线程也是从线程池中拿的（`ThreadPool`），官方建议用`Task`而不是直接使用`ThreadPool`，因为`Task`是对`ThreadPool`的封装和优化。

`Task.WaitAny`:等待一群线程中的其中一个完成，这里是卡主线程，一直等到一群线程中的最快的一个完成，才能继续往下执行，打个简单的比方：从三个地方获取配置信息（数据库，`config`，`IO`），同时开启三个线程来访问，谁快谁来执行。

`Task.WaitAll`:等待所有线程完成，这里也是卡主线程，一直等待所有子线程完成任务，才能继续往下执行。

`Task.WhenAll`:等待所有线程完成，这里不卡主线程，一直等待所有子线程完成任务，才能继续往下执行。

`Task.ContinueWhenAny`:回调形式的，任意一个线程完成后执行的后续动作，这个就跟`WaitAny`差不多，只是是回调形式的。

`Task.ContinueWhenAll`:回调形式的，所有线程完成后执行的后续动作，理解同上

代码示例：



```

public class TaskDemo
{
    /// <summary>
    /// 一个比较耗时的方法,循环1000W次
    /// </summary>
    /// <param name="name"></param>
    public static void DoSomething(string name)
    {
        int iResult = 0;
        for (int i = 0; i < 10000000000; i++)
        {
            iResult += i;
        }

        Console.WriteLine($"{name}, 线程Id:
{Thread.CurrentThread.ManagedThreadId}, {DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss
ffff")}" + Environment.NewLine);
    }

    public static void Test()
    {
        //线程容器
        List<Task> taskList = new List<Task>();
        Stopwatch watch = new Stopwatch();
        watch.Start();
        Console.WriteLine("*****任务开始*****");

        //启动5个线程
        for (int i = 0; i < 5; i++)
        {
            string name = $"Task:{i}";
            Task task = Task.Factory.StartNew(() =>
            {
                DoSomething(name);
            });

            taskList.Add(task);
        }

        //回调形式的,任意一个完成后执行的后续动作
        Task any = Task.Factory.ContinueWhenAny(taskList.ToArray(), task =>
        {
            Console.WriteLine($"ContinueWhenAny, 当前线程Id:
{Thread.CurrentThread.ManagedThreadId}, 一个线程执行完的回调, 继续执行后面动作,
{DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss ffff")}" + Environment.NewLine);
        });

        //回调形式的,全部任务完成后执行的后续动作
        Task all = Task.Factory.ContinueWhenAll(taskList.ToArray(), tasks =>
        {
            Console.WriteLine($"ContinueWhenAll, 当前线程Id:
{Thread.CurrentThread.ManagedThreadId}, 全部线程执行完的回调, 线程数: {tasks.Length},
{DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss ffff")}" + Environment.NewLine);
        });

        //把两个回调也放到容器里面, 包含回调一起等待
        taskList.Add(any);
        taskList.Add(all);
    }
}

```

```

        //WaitAny:线程等待，当前线程等待其中一个线程完成
        Task.WaitAny(taskList.ToArray());
        Console.WriteLine($"WaitAny,当前线程Id:
{Thread.CurrentThread.ManagedThreadId},其中一个完成已完成,
{DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss ffff")}" + Environment.NewLine);

        //WaitAll:线程等待，当前线程等待所有线程的完成
        Task.WaitAll(taskList.ToArray());
        Console.WriteLine($"WaitAll,当前线程Id:
{Thread.CurrentThread.ManagedThreadId},全部线程完成,{DateTime.Now.ToString("yyyy-
MM-dd HH:mm:ss ffff")}" + Environment.NewLine);

        Console.WriteLine($"*****任务结束*****耗时:
{watch.ElapsedMilliseconds}ms,{Thread.CurrentThread.ManagedThreadId},
{DateTime.Now}");
    }
}

```



结果：

```

*****任务开始*****
Task:1,线程Id:4,2022-04-01 23:41:47 2183
Task:0,线程Id:6,2022-04-01 23:41:47 2396
WaitAny,当前线程Id:1,其中一个完成已完成,2022-04-01 23:41:47 2581
ContinueWhenAny,当前线程Id:4,一个线程执行完的回调,继续执行后面动作,2022-04-01 23:41:47 2584
Task:2,线程Id:5,2022-04-01 23:41:47 2150
Task:3,线程Id:7,2022-04-01 23:41:47 3722
Task:4,线程Id:6,2022-04-01 23:41:50 0284
ContinueWhenAll,当前线程Id:4,全部线程执行完的回调,线程数: 5,2022-04-01 23:41:50 0299
WaitAll,当前线程Id:1,全部线程完成,2022-04-01 23:41:50 0302
*****任务结束*****耗时: 6113ms,1,2022/4/1 星期五 23:41:50

```

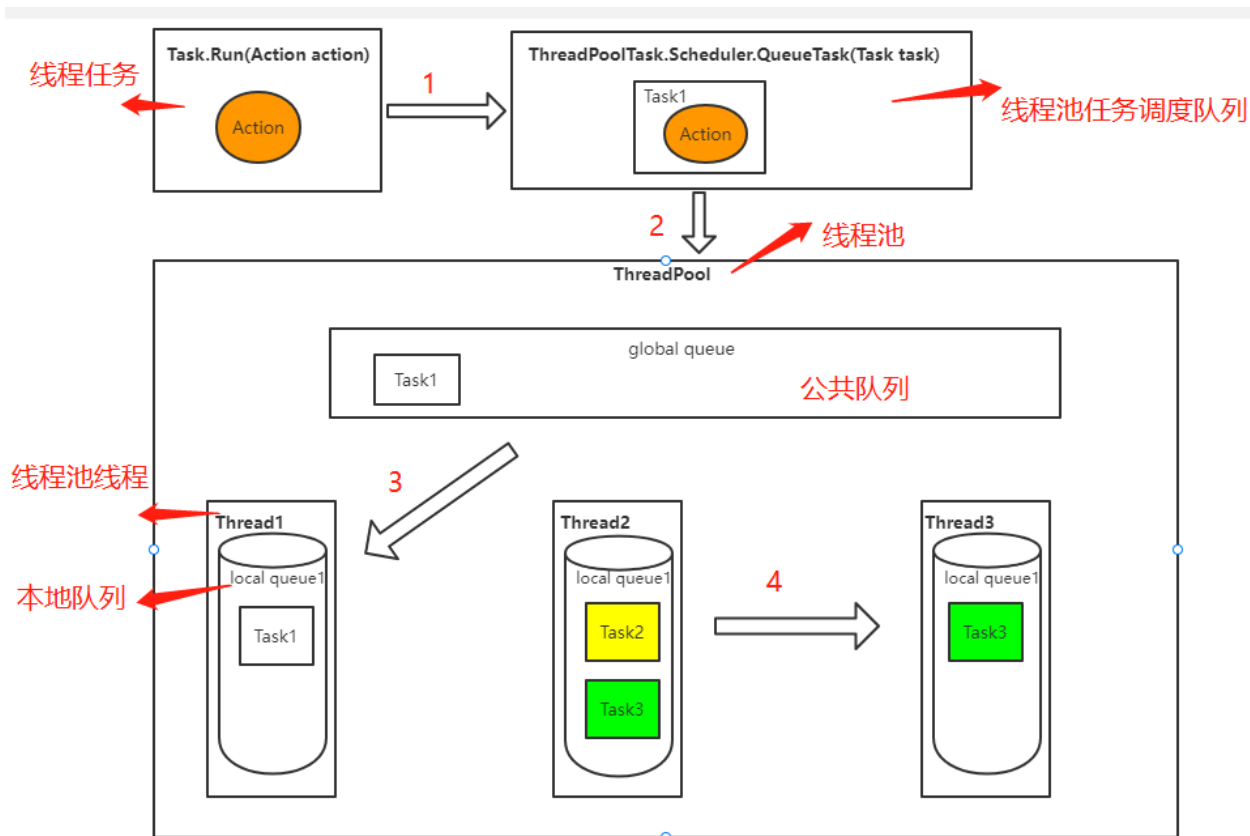
4.线程池调度原理

根据 TaskCreationOptions 的不同，出现了三个分支

- LongRunning：独立线程，和线程池无关
- 包含 PreferFairness时：preferLocal=false，进入全局队列
- 不包含 PreferFairness时：preferLocal=true，进入本地队列

进入全局队列的任务能够公平地被各个线程池中的线程领取执行。

下图中 Task1先进入全局队列，随后被 Thread1 领走。Thread3 通过 WorkStealing 机制窃取了 Thread2 中的 Task3。



- 1、线程任务1先去到线程池调度队列。
- 2、线程池队列根据参数去到公共队列或线程的本地队列。
- 3、线程池线程1从公共线程池中取Task1到本地队列执行
- 4、Thead3发现自己的队列为空，公共队列也为空，会从其它的线程中窃取任务执行。

[回到顶部](#)

四、并行

Parallel :是并行编程，在Task的基础上做了封装，.NET Framework 4.5之后的版本可用，调用 **Parallel** 线程参与执行任务。

与Task区别 :使用Task开启子线程的时候，主线程是属于空闲状态，并不参与执行;Parallel开启子线程的时候，主线程也会参与计算

示例1：




```

/// <summary>
/// 一个比较耗时的方法,循环1000W次
/// </summary>
/// <param name="name"></param>
public static void DoSomething(string name)
{
    int iResult = 0;
    for (int i = 0; i < 1000000000; i++)
    {
        iResult += i;
    }

    Console.WriteLine($"{name},线程Id:
{Thread.CurrentThread.ManagedThreadId},{DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss
ffff")}" + Environment.NewLine);
}

public static void Test1()
{
    //并行编程
    Console.WriteLine($"并行编程开始,主线程Id:
{Thread.CurrentThread.ManagedThreadId}");
    Console.WriteLine("【示例1】");
    //示例1:
    //一次性执行1个或多个线程,效果类似:Task WaitAll,只不过Parallel的主线程也参与
    Parallel.Invoke(
        () => { DoSomething("并行1-1"); },
        () => { DoSomething("并行1-2"); },
        () => { DoSomething("并行1-3"); },
        () => { DoSomething("并行1-4"); },
        () => { DoSomething("并行1-5"); });
    Console.WriteLine("*****并行结束*****");
    Console.ReadLine();
}

```



执行Test1结果：

```

并行编程开始,主线程Id:1
【示例1】
并行1-1,线程Id:1,2022-04-02 23:48:06 3343
并行1-3,线程Id:5,2022-04-02 23:48:06 3647
并行1-2,线程Id:7,2022-04-02 23:48:06 3848
并行1-5,线程Id:6,2022-04-02 23:48:06 4081
并行1-4,线程Id:4,2022-04-02 23:48:06 4403
*****并行结束*****

```

示例2：



```

public static void Test2()
{
    //并行编程
    Console.WriteLine($"并行编程开始，主线程Id:
{Thread.CurrentThread.ManagedThreadId}");
    Console.WriteLine("【示例2】");
    //示例2：
    //定义要执行的线程数量
    Parallel.For(0, 5, t =>
    {
        int a = t;
        DoSomething($"并行2-{a}");

    });
    Console.WriteLine("*****并行结束*****");
    Console.ReadLine();
}

```



结果：

```

并行编程开始，主线程Id:1
【示例2】
并行2-1, 线程Id:4, 2022-04-02 23:54:54 2478
并行2-2, 线程Id:5, 2022-04-02 23:54:54 2885
并行2-0, 线程Id:1, 2022-04-02 23:54:54 3017
并行2-3, 线程Id:6, 2022-04-02 23:54:54 3729
并行2-4, 线程Id:7, 2022-04-02 23:54:54 4282
*****并行结束*****

```

示例3：



```

public static void Test3()
{
    //并行编程
    Console.WriteLine($"并行编程开始，主线程Id:
{Thread.CurrentThread.ManagedThreadId}");
    Console.WriteLine("【示例3】");
    ParallelOptions options = new ParallelOptions()
    {
        MaxDegreeOfParallelism = 3//执行线程的最大并发数量,执行完成一个，就接着开
启一个
    };

    //遍历集合，根据集合数量执行线程数量
    Parallel.ForEach(new List<string> { "a", "b", "c", "d", "e", "f", "g"
}, options, (t, status) =>

    {
        //status.Break();//这一次结束。
        //status.Stop();//整个Parallel结束掉，Break和Stop不可以共存
        DoSomething($"并行4-{t}");
    });
}

```



结果：分成3次并行

```

并行编程开始，主线程Id:1
【示例3】
并行4-c, 线程Id:4, 2022-04-02 23:58:13 6214
并行4-e, 线程Id:5, 2022-04-02 23:58:13 6647
并行4-a, 线程Id:1, 2022-04-02 23:58:13 7164
并行4-d, 线程Id:4, 2022-04-02 23:58:16 3696
并行4-b, 线程Id:1, 2022-04-02 23:58:16 4085
并行4-f, 线程Id:5, 2022-04-02 23:58:16 4109
并行4-g, 线程Id:4, 2022-04-02 23:58:18 6994

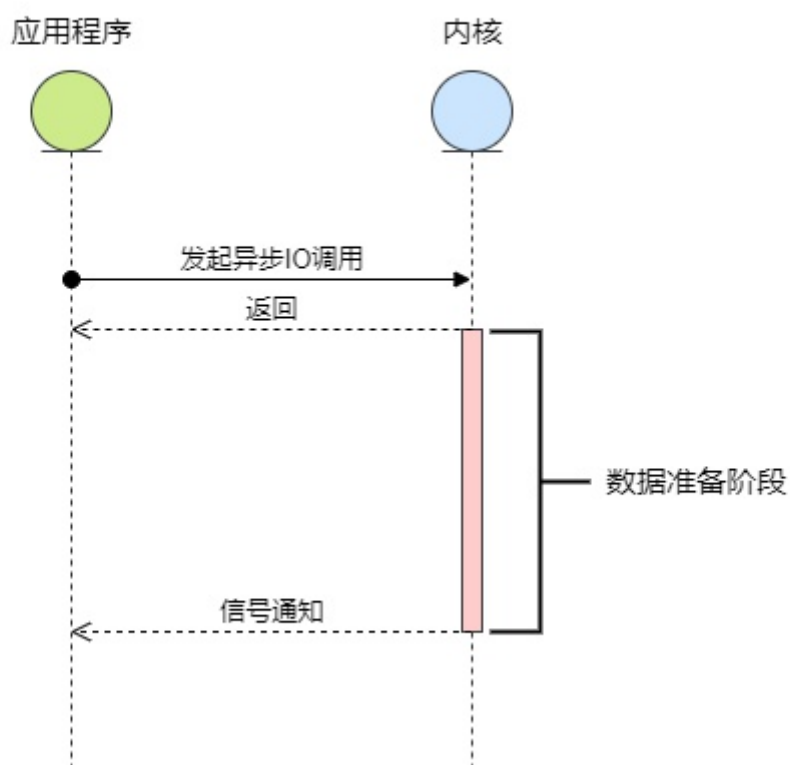
```

[回到顶部](#)

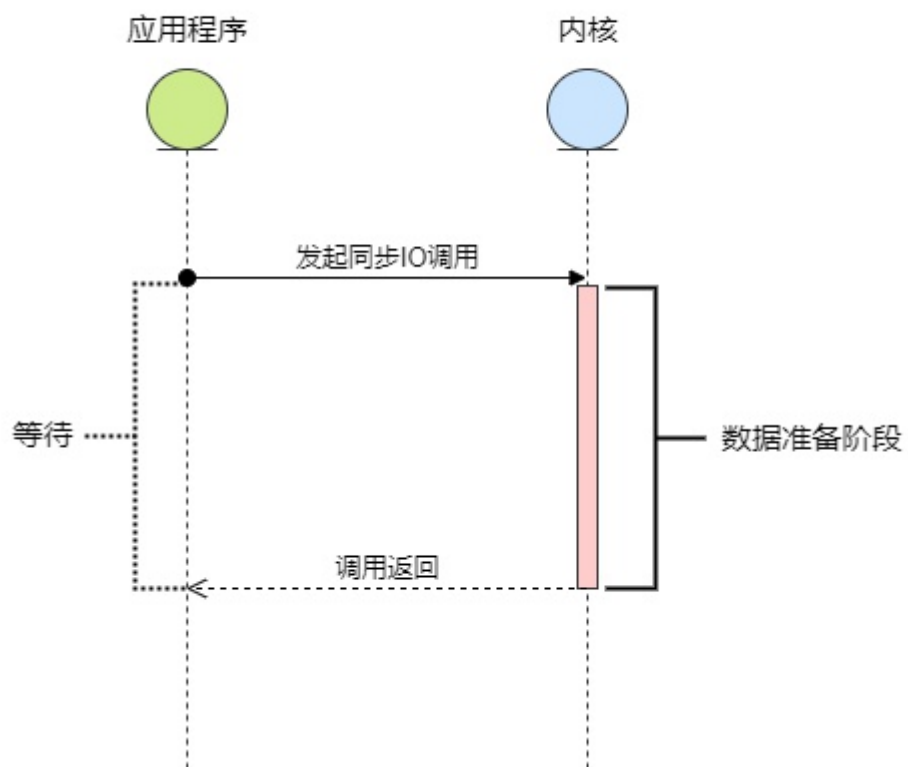
五、异步IO

1.异步IO于同步IO比较

异步IO



同步IO



异步IO在数据准备阶段不会阻塞主线程，而同步IO则会阻塞主线程。

2.异步读写文件

这里使用 `FileStream` 类，它带有一个参数 `useAsync`，可以避免在许多情况下阻塞线程池的线程。可以通过 `useAsync = true` 来进行启用或在构造函数中进行参数调用。

但是我们不能对 `StreamReader` 和 `StreamWriter` 中的参数进行设置。但是，如果你想使用该参数 `useAsync`，则需要自己新建一个 `FileStream` 对象。

请注意，异步调用是在 UI 中的，即使线程池线程阻塞，在 `await` 期间，用户界面线程也不会被阻塞。

异步写入文本



```
/// <summary>
/// 异步写入文件
/// </summary>
/// <returns></returns>
public async Task WriteTextAsync()
{
    var path = "temp.txt"; //文件名
    var content = Guid.NewGuid().ToString(); //写入内容

    using (var fs = new FileStream(path, FileMode.OpenOrCreate,
        FileAccess.ReadWrite, FileShare.None, bufferSize: 4096, useAsync: true))
    {
        var buffer = Encoding.UTF8.GetBytes(content);
        await fs.WriteAsync(buffer, 0, buffer.Length);
    }
}
```



执行完查看根目录文件结果：

异步读取文件



temp.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

5835a40e-f32a-4632-9ac1-a22aaded7992

```

/// <summary>
/// 异步读取文本
/// </summary>
/// <returns></returns>
public static async Task ReadTextAsync()
{
    var fileName = "temp.txt"; //文件名
    using (var fs = new FileStream(fileName,
                                   FileMode.OpenOrCreate, FileAccess.Read,
FileShare.None, bufferSize: 4096, useAsync: true))
    {
        var sb = new StringBuilder();
        var buffer = new byte[4096];
        var readLength = 0;

        while ((readLength = await fs.ReadAsync(buffer, 0, buffer.Length))
!= 0)
        {
            var text = Encoding.UTF8.GetString(buffer, 0, readLength);
            sb.Append(text);
        }

        Console.WriteLine("读取文件内容:"+sb.ToString());
    }
}

```



执行结果：

```

读取文件内容:5835a40e-f32a-4632-9ac1-a22aaded7992

```