[**第一节：复习委托，并且通过委托的异步调用开启一个新线程和异步回调、异步等待。**](https://www.cnblogs.com/yaopengfei/p/8094512.html)

一. 再谈委托

1. 委托是一个关键字为delegate的自定义类型，通过委托可以把方法以参数的形式传递给另外一个方法，实现插件式的开发模式；

    同时调用委托的时候，委托所包含的所有方法都会被实现。

2. 委托的发展历史：new实例化传递方法→直接等于方法名→delegate匿名方法→省略delegate→省略括号中的参数→当只有一个参数省略小括号

　　　　　　　　　 →当方法体只有一行，省略大括号

[**(详见：http://www.cnblogs.com/yaopengfei/p/6959141.html)**](http://www.cnblogs.com/yaopengfei/p/6959141.html)

3：常用的Action委托和Func委托

　　A. Action<>委托，无返回值，至少有一个参数的委托

　　B. Func<>委托，有返回值，可以无参数的委托(当然也可以有参数)

　　C. Action委托，无参数无返回值的委托

二. 委托的调用

委托的调用分为两种：

　　A. 同步调用：Invoke方法，方法参数为函数的参数。

　　B. 异步调用：BeginInvoke方法。

其中无论是哪类调用，都有两类写法：

　　①：利用Action<>(或Func<>)内置委托，调用的时候赋值。

　　②：利用Action委托，直接赋值，然后调用。

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 执行动作:耗时而已

3 /// </summary>

4 private void TestThread2(string threadName1, string threadName2)

5 {

6 Console.WriteLine("线程开始：线程名为：{2}和{3}，当前线程的id为:{0}，当前时间为：{1},", System.Threading.Thread.CurrentThread.ManagedThreadId, DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss:fff"), threadName1, threadName2);

7 long sum = 0;

8 for (int i = 1; i < 999999999; i++)

9 {

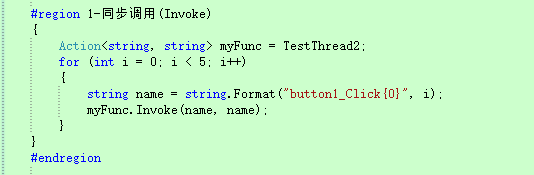
10 sum += i;

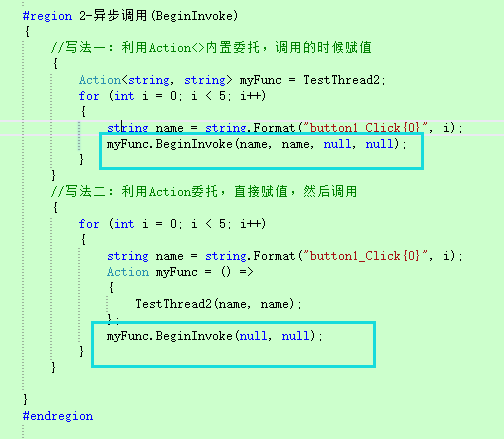
11 }

12 Console.WriteLine("线程结束：线程名为：{2}和{3}，当前线程的id为::{0}，当前时间为：{1}", System.Threading.Thread.CurrentThread.ManagedThreadId, DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss:fff"), threadName1, threadName2);

13 }

[复制代码](javascript:void(0);)





三. 深入剖析BeginInvoke方法

 　　首先需要明确，该方法参数个数不定, 最后两个参数含义固定，如果不使用的话，需要赋值null；该方法最少两个参数，**即方法无参数**，这种情况下BeginInvoke中只有两个参数。此外，赋值的方法有几个参数，BeginInvoke中从左开始，新增几个参数。

　　①. 倒数第二个参数：是有一个参数值无返回值的委托，它代表的含义为，该线程执行完毕后的回调。

　　②. 倒数第一个参数：向倒数第二个参数(即回调)中传值，需要用AsyncState来接受。

　　③. 其它参数：即为赋值方法的参数。

**注：BeginInvoke的返回值等价于异步回调中的t。**

[复制代码](javascript:void(0);)

1 private void button13\_Click(object sender, EventArgs e)

2 {

3 Stopwatch watch = new Stopwatch();

4 watch.Start();

5 Console.WriteLine("----------------- button1\_Click 开始 主线程id为：{0} --------------------------", Thread.CurrentThread.ManagedThreadId);

6

7 Action<string> myFunc = this.TestThread;

8 IAsyncResult asyncResult = null;

9 //参数说明：前面几个参数都是方法的参数值，倒数第二个为异步调用的回调函数，倒数第一个为传给回调函数的参数

10 for (int i = 0; i < 1; i++)

11 {

12 string name = string.Format("button1\_Click{0}", i);

13 asyncResult = myFunc.BeginInvoke(name, t =>

14 {

15 Console.WriteLine("我是线程{0}的回调", Thread.CurrentThread.ManagedThreadId);

16 //用 t.AsyncState 来获取回调传进来的参数

17 Console.WriteLine("传进来的参数为：{0}", t.AsyncState);

18

19 //测试一下异步返回值的结果

20 Console.WriteLine("异步返回值的结果：{0}", t.Equals(asyncResult));

21 }, "maru");

22 }

23

24 watch.Stop();

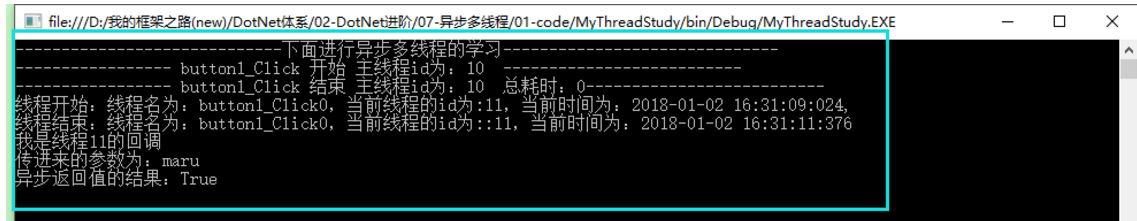
25 Console.WriteLine("----------------- button1\_Click 结束 主线程id为：{0} 总耗时：{1}--------------------------", Thread.CurrentThread.ManagedThreadId, watch.ElapsedMilliseconds);

26

27 }

[复制代码](javascript:void(0);)

**结果：**

****

四. 线程等待的三种方式

1. asyncResult.IsCompleted属性，该方式会存在时间上的误差。

2. WaitOne方法，可以控制一直等待or超时不再等待。

3. EndInvoke方法，官方推荐的线程等待的方式。

**以上三种方式的局限性：批量线程等待的时候，不灵活，需要for循环了。**

[复制代码](javascript:void(0);)

1 private void button14\_Click(object sender, EventArgs e)

2 {

3 Stopwatch watch = new Stopwatch();

4 watch.Start();

5 Console.WriteLine("----------------- button1\_Click 开始 主线程id为：{0} --------------------------", Thread.CurrentThread.ManagedThreadId);

6

7 IAsyncResult asyncResult = null;

8 Action<string> myFunc = this.TestThread;

9 string name = string.Format("button1\_Click{0}", 111);

10 asyncResult = myFunc.BeginInvoke(name, t =>

11 {

12 Console.WriteLine("我是线程{0}的回调", Thread.CurrentThread.ManagedThreadId);

13 //用 t.AsyncState 来获取回调传进来的参数

14 Console.WriteLine("传进来的参数为：{0}", t.AsyncState);

15 }, "maru");

16

17 //等待的方式1：会有时间上的误差

18 //while (!asyncResult.IsCompleted)

19 //{

20 // Console.WriteLine("正在等待中");

21 //}

22

23 // 等待的方式二:

24 //asyncResult.AsyncWaitHandle.WaitOne();//一直等待

25 //asyncResult.AsyncWaitHandle.WaitOne(-1);//一直等待

26 //asyncResult.AsyncWaitHandle.WaitOne(1000);//等待1000毫秒，超时就不等待了

27

28 //等待的方式三：

29 myFunc.EndInvoke(asyncResult);

30

31 watch.Stop();

32 Console.WriteLine("----------------- button1\_Click 结束 主线程id为：{0} 总耗时：{1}--------------------------", Thread.CurrentThread.ManagedThreadId, watch.ElapsedMilliseconds);

33

34 }

[复制代码](javascript:void(0);)

**下面是多个线程等待的情况：**

[复制代码](javascript:void(0);)

1 private void button15\_Click(object sender, EventArgs e)

2 {

3 Stopwatch watch = new Stopwatch();

4 watch.Start();

5 Console.WriteLine("----------------- button1\_Click 开始 主线程id为：{0} --------------------------", Thread.CurrentThread.ManagedThreadId);

6

7 List<IAsyncResult> list = new List<IAsyncResult>();

8

9 for (int i = 0; i < 5; i++)

10 {

11 string name = string.Format("button1\_Click{0}", i);

12 Action myFunc = () =>

13 {

14 TestThread2(name, name);

15 };

16 var asyncResult = myFunc.BeginInvoke(null, null);

17 list.Add(asyncResult);

18 }

19

20 //下面是线程等待

21 foreach (var item in list)

22 {

23 item.AsyncWaitHandle.WaitOne(-1);

24 }

25

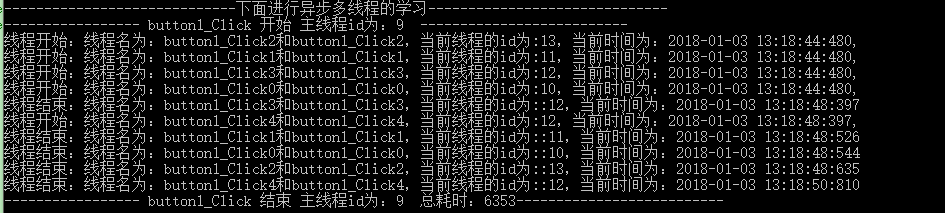
26 watch.Stop();

27 Console.WriteLine("----------------- button1\_Click 结束 主线程id为：{0} 总耗时：{1}--------------------------", Thread.CurrentThread.ManagedThreadId, watch.ElapsedMilliseconds);

28 }

[复制代码](javascript:void(0);)

结果：



[**第二节：深入剖析Thread的五大方法、数据槽、内存栅栏。**](https://www.cnblogs.com/yaopengfei/p/8135829.html)

一. Thread及其五大方法

 　　Thread是.Net最早的多线程处理方式，它出现在.Net1.0时代，虽然现在已逐渐被微软所抛弃，微软强烈推荐使用Task(后面章节介绍)，但从多线程完整性的角度上来说，我们有必要了解下N年前多线程的是怎么处理的，以便体会.Net体系中多线程处理方式的进化。

　　Thread中有五大方法，分别是：Start、Suspend、Resume、Intterupt、Abort

　　①.Start：开启线程

　　②.Suspend：暂停线程

　　③.Resume：恢复暂停的线程

　　④.Intterupt：中断线程(会抛异常，提示线程中断)

　　⑤.Abort：销毁线程

 　  这五大方法使用起来，也比较简单，下面贴一段代码，体会一下如何使用即可。



　　在这里补充一下，在该系列中，很多测试代码中看到TestThread0、TestThread、TestThread2，分别对应无参、一个参数、两个参数的耗时方法，代码如下：

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 执行动作:耗时而已

3 /// </summary>

4 private void TestThread0()

5 {

6 Console.WriteLine("线程开始：当前线程的id为:{0}，当前时间为：{1},", System.Threading.Thread.CurrentThread.ManagedThreadId, DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss:fff"));

7 long sum = 0;

8 for (int i = 1; i < 999999999; i++)

9 {

10 sum += i;

11 }

12 Console.WriteLine("线程结束：当前线程的id为::{0}，当前时间为：{1}", System.Threading.Thread.CurrentThread.ManagedThreadId, DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss:fff"));

13 }

14

15 /// <summary>

16 /// 执行动作:耗时而已

17 /// </summary>

18 private void TestThread(string threadName)

19 {

20 Console.WriteLine("线程开始：线程名为：{2}，当前线程的id为:{0}，当前时间为：{1},", System.Threading.Thread.CurrentThread.ManagedThreadId, DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss:fff"), threadName);

21 long sum = 0;

22 for (int i = 1; i < 999999999; i++)

23 {

24 sum += i;

25 }

26 Console.WriteLine("线程结束：线程名为：{2}，当前线程的id为::{0}，当前时间为：{1}", System.Threading.Thread.CurrentThread.ManagedThreadId, DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss:fff"), threadName);

27 }

28

29 /// <summary>

30 /// 执行动作:耗时而已

31 /// </summary>

32 private void TestThread2(string threadName1, string threadName2)

33 {

34 Console.WriteLine("线程开始：线程名为：{2}和{3}，当前线程的id为:{0}，当前时间为：{1},", System.Threading.Thread.CurrentThread.ManagedThreadId, DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss:fff"), threadName1, threadName2);

35 long sum = 0;

36 for (int i = 1; i < 999999999; i++)

37 {

38 sum += i;

39 }

40 Console.WriteLine("线程结束：线程名为：{2}和{3}，当前线程的id为::{0}，当前时间为：{1}", System.Threading.Thread.CurrentThread.ManagedThreadId, DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss:fff"), threadName1, threadName2);

41 }

[复制代码](javascript:void(0);)

View Code

二. 从源码角度分析Thread类

**(1)  分析Thread类的源码，发现其构造函数有两类，分别是ThreadStart和ParameterizedThreadStart类，**

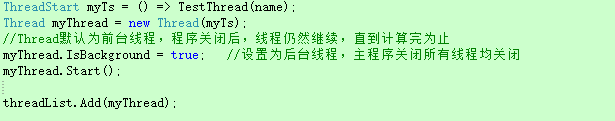
　其中

　　①：ThreadStart类，是无参无返回值的委托。

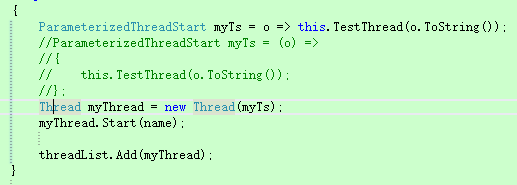
　　②：ParameterizedThreadStart类,是有一个object类型参数但无返回值的委托.

**使用方法：**

　　①：针对ThreadStart类， ThreadStart myTs = () => TestThread(name);  然后再把myTs传入Thread的构造函数中



　　②：针对ParameterizedThreadStart类，ParameterizedThreadStart myTs = o => this.TestThread(o.ToString());  然后再把myTs传入Thread的构造函数中



**注：该方式存在拆箱和装箱的转换问题，不建议这么使用**

**通用写法*：***

 　　　　Thread t = new Thread(() =>

　　　　{

 　　　　　　Console.Write("333");

　　　　});

　　　　t.Start();

无须考虑Thread的构造函数,也不需要考虑Start的时候传参，直接使用()=>{}的形式，解决一切问题。

**(二) 前台进程和后台进程（IsBackground属性）**

　　①：前台进程,Thread默认为前台线程，程序关闭后，线程仍然继续，直到计算完为止

　　②：后台进程,将IsBackground属性设置为true，即为后台进程，主线程关闭，所有子线程无论运行完否，都马上关闭

**(三) 线程等待（Join方法）**

　　利用Join方法实现主线程等待子线程,当多个子线程进行等待的时候，只能通过for循环来实现

**下面贴一下这三块设计到的代码：**

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

1 private void button3\_Click(object sender, EventArgs e)

2 {

3 Stopwatch watch = new Stopwatch();

4 watch.Start();

5 Console.WriteLine("-----------------Thread多线程 --------------------------");

6 Console.WriteLine("----------------- button\_Click 开始 主线程id为：{0} --------------------------", Thread.CurrentThread.ManagedThreadId);

7 List<Thread> threadList = new List<Thread>();

8 for (int i = 0; i < 5; i++)

9 {

10 string name = string.Format("button1\_Click{0}", i);

11

12 #region 方式一

13 {

14 ThreadStart myTs = () => TestThread(name);

15 Thread myThread = new Thread(myTs);

16 //Thread默认为前台线程，程序关闭后，线程仍然继续，直到计算完为止

17 myThread.IsBackground = true; //设置为后台线程，主程序关闭所有线程均关闭

18 myThread.Start();

19

20 threadList.Add(myThread);

21 }

22

23 #endregion

24

25 #region 方式二

26 //{

27 // ParameterizedThreadStart myTs = o => this.TestThread(o.ToString());

28 // //ParameterizedThreadStart myTs = (o) =>

29 // //{

30 // // this.TestThread(o.ToString());

31 // //};

32 // Thread myThread = new Thread(myTs);

33 // myThread.Start(name);

34

35 // threadList.Add(myThread);

36 //}

37

38 #endregion

39 }

40

41 #region Thread线程等待

42

43 //利用join方法进行线程等待

44 foreach (Thread thread in threadList)

45 {

46 thread.Join();

47 }

48 #endregion

49

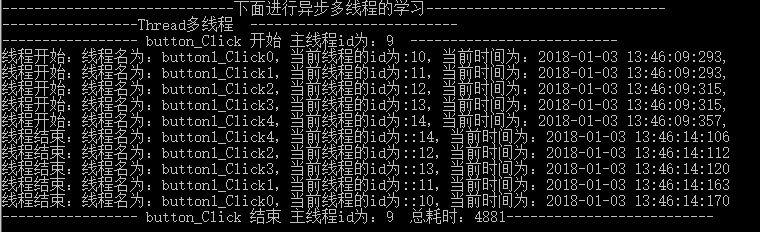
50 watch.Stop();

51 Console.WriteLine("----------------- button\_Click 结束 主线程id为：{0} 总耗时：{1}--------------------------", Thread.CurrentThread.ManagedThreadId, watch.ElapsedMilliseconds);

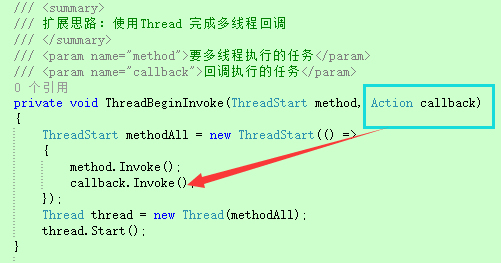
52

53 }

View Code



**(四). 扩展：Thread实现线程回调**



三. 数据槽-线程可见性

　　背景：为了解决多线程竞用共享资源的问题，引入数据槽的概念，即将数据存放到线程的环境块中，使该数据只能单一线程访问.（属于线程空间上的开销）

**下面的三种方式是解决多线程竞用共享资源的通用方式：**

**①：AllocateNamedDataSlot命名槽位和AllocateDataSlot未命名槽位 解决线程竞用资源共享问题。**

　　(PS:在主线程上设置槽位，使该数据只能被主线程读取，其它线程无法访问)

[复制代码](javascript:void(0);)

private void button10\_Click(object sender, EventArgs e)

{

#region 01-AllocateNamedDataSlot命名槽位

{

var d = Thread.AllocateNamedDataSlot("userName");

//在主线程上设置槽位，使该数据只能被主线程读取，其它线程无法访问

Thread.SetData(d, "ypf");

//声明一个子线程

var t1 = new Thread(() =>

{

Console.WriteLine("子线程中读取数据：{0}", Thread.GetData(d));

});

t1.Start();

//主线程中读取数据

Console.WriteLine("主线程中读取数据：{0}", Thread.GetData(d));

}

#endregion

#region 02-AllocateDataSlot未命名槽位

{

var d = Thread.AllocateDataSlot();

//在主线程上设置槽位，使该数据只能被主线程读取，其它线程无法访问

Thread.SetData(d, "ypf");

//声明一个子线程

var t1 = new Thread(() =>

{

Console.WriteLine("子线程中读取数据：{0}", Thread.GetData(d));

});

t1.Start();

//主线程中读取数据

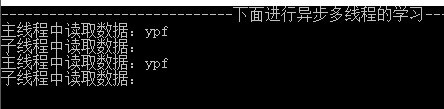
Console.WriteLine("主线程中读取数据：{0}", Thread.GetData(d));

}

#endregion

}

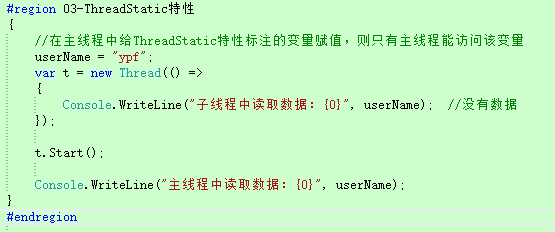
[复制代码](javascript:void(0);)



**②：利用特性[ThreadStatic] 解决线程竞用资源共享问题**

**(PS:在主线程中给ThreadStatic特性标注的变量赋值，则只有主线程能访问该变量)**

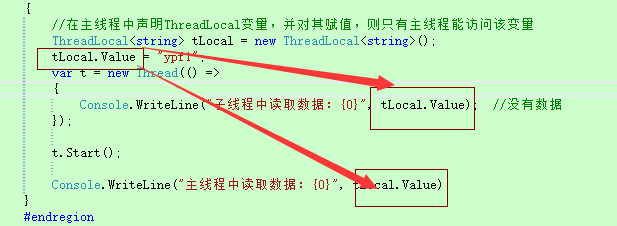
https://images2017.cnblogs.com/blog/1031302/201801/1031302-20180103140120581-152208974.png

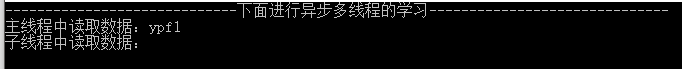


https://images2017.cnblogs.com/blog/1031302/201801/1031302-20180103140156987-457727881.png

**③：利用ThreadLocal线程的本地存储， 解决线程竞用资源共享问题（线程可见性）**

**(PS: 在主线程中声明ThreadLocal变量，并对其赋值，则只有主线程能访问该变量)**





四. 内存栅栏-线程共享资源

背景：当多个线程共享一个变量的时候，在Release模式的优化下，子线程会将共享变量加载的cup Cache中，导致主线程不能使用该变量而无法运行。

解决方案：

　　①：不要让多线程去操作同一个共享变量,从根本上解决这个问题。

　　②：利用MemoryBarrier方法进行处理,在此方法之前的内存写入都要及时从cpu cache中更新到 memory；在此方法之后的内存读取都要从memory中读取，而不是cpu cache。

　　③：利用VolatileRead/Write方法进行处理。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 private void button11\_Click(object sender, EventArgs e)

2 {

3 #region 01-默认情况(Release模式主线程不能正常运行)

4 //{

5 // var isStop = false;

6 // var t = new Thread(() =>

7 // {

8 // var isSuccess = false;

9 // while (!isStop)

10 // {

11 // isSuccess = !isSuccess;

12 // }

13 // Console.WriteLine("子线程执行成功");

14 // });

15 // t.Start();

16

17 // Thread.Sleep(1000);

18 // isStop = true;

19

20 // t.Join();

21 // Console.WriteLine("主线程执行结束");

22 //}

23 #endregion

24

25 #region 02-MemoryBarrier解决共享变量（Release模式下可以正常运行）

26 //{

27 // var isStop = false;

28 // var t = new Thread(() =>

29 // {

30 // var isSuccess = false;

31 // while (!isStop)

32 // {

33 // Thread.MemoryBarrier();

34

35 // isSuccess = !isSuccess;

36 // }

37 // Console.WriteLine("子线程执行成功");

38 // });

39 // t.Start();

40

41 // Thread.Sleep(1000);

42 // isStop = true;

43

44 // t.Join();

45 // Console.WriteLine("主线程执行结束");

46 //}

47 #endregion

48

49 #region 03-VolatileRead解决共享变量（Release模式下可以正常运行）

50 {

51 var isStop = 0;

52 var t = new Thread(() =>

53 {

54 var isSuccess = false;

55 while (isStop == 0)

56 {

57 Thread.VolatileRead(ref isStop);

58

59 isSuccess = !isSuccess;

60 }

61 Console.WriteLine("子线程执行成功");

62 });

63 t.Start();

64

65 Thread.Sleep(1000);

66 isStop = 1;

67

68 t.Join();

69 Console.WriteLine("主线程执行结束");

70 }

71 #endregion

72

73

74 }

[复制代码](javascript:void(0);)

View Code

# [第三节：ThreadPool的线程开启、线程等待、线程池的设置、定时功能](https://www.cnblogs.com/yaopengfei/p/8178039.html)

一. ThreadPool简介

　　ThreadPool简介：ThreadPool是一个线程池，当你需要开启n个线程时候，只需把这个指令抛给线程池，它将自动分配线程进行处理，它诞生于.Net 2.0时代。

　　ThreadPool与Thread的区别：

　　　　①：Thread每开启一个异步任务，就需要使用一个Thread，具有专一性，即使Thread已经死掉，仍然需要占用资源。

　　　　②：ThreadPool能实现n个线程处理n+m个异步任务，且没有死线程，默认都是初始化的。

二. 深究ThreadPool类

**1：QueueUserWorkItem方法，将方法排入队列以便开启异步线程，它有两个重载。**

　　a：QueueUserWorkItem(WaitCallback callBack)，WaitCallback是一个有一个object类型参数且无返回值的委托

　　b：QueueUserWorkItem(WaitCallback callBack, object state)，WaitCallback是一个有一个object类型参数且无返回值的委托，state即WaitCallback中需要的参数， 不推荐这么使用，存在拆箱装箱的转换问题，影响性能。

**使用方式：**

　　a：严格的卡定义的写法，第一个重载和第二个重载均是处理有一个参数的函数，大于一个参数需要封装一个实体类，所需参数在实体类的构造函数中引入，然后调用第一个重载即可.

**（不推荐这种写法）**

**通用写法*：*** (可以处理任何参数个数的方法)，这里不使用n即可。

 ThreadPool.QueueUserWorkItem((n) =>

{

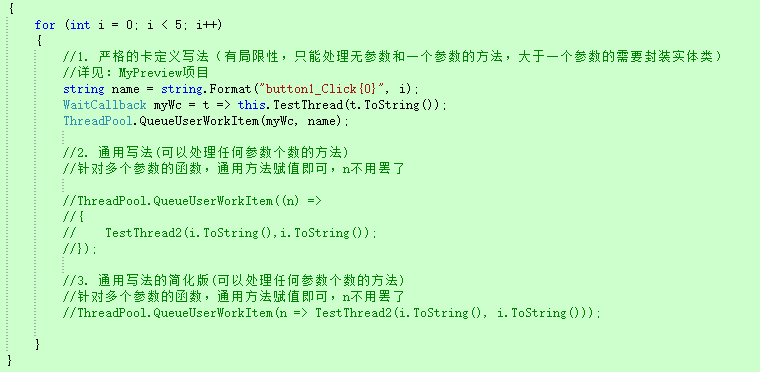
　　TestThread2(i.ToString(),i.ToString());

});

**通用写法的简化版：**

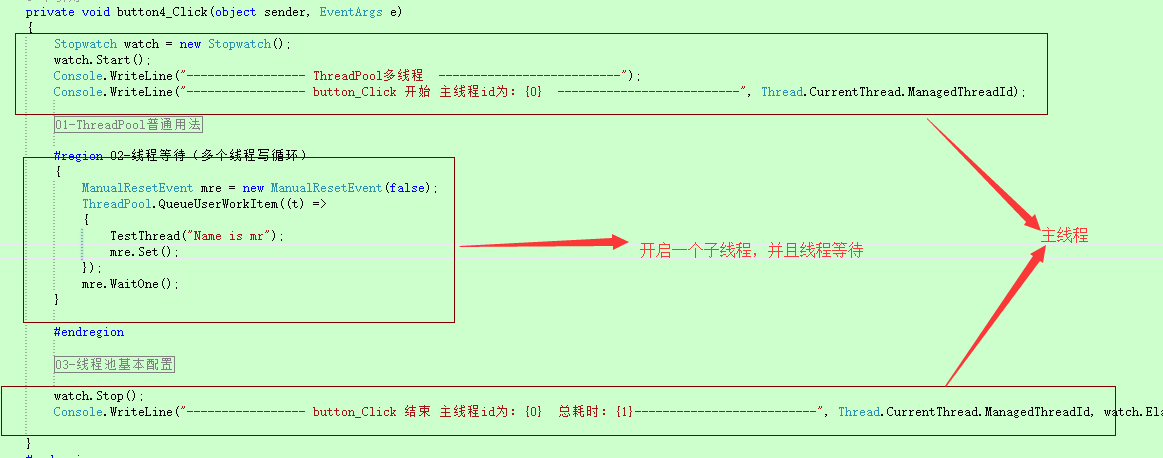
ThreadPool.QueueUserWorkItem(n => TestThread2(i.ToString(), i.ToString()));

 代码如下：

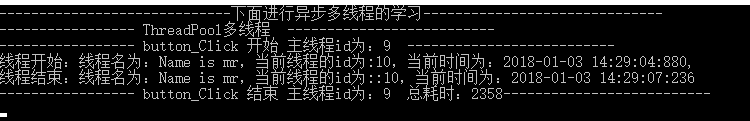


**2：线程等待（有局限性的）**

 　　利用ManualResetEvent类和WaitOne方法,多个线程的话需要写for循环。



**运行结果：**



**3：常用方法**

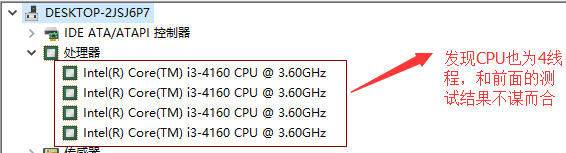
　　a：设置工作线程和IO线程的最大和最小值，SetMaxThreads和SetMinThreads。

　　b：获取工作线程和IO线程最大、最小、可用值，GetMaxThreads、GetMinThreads、GetAvailableThreads。



**测试结果：**





三. 扩展一个定时器功能

1：RegisterWaitForSingleObject类，但是不常用.（涉及到定时任务，建议使用Quartz.Net）

2：System.threading命名空间下的Thread类,通过查看源码，构造函数中有四个参数，第一个是object参数的委托，第二个是委托需要的值，

 　　第三个是调用 callback 之前延迟的时间量（以毫秒为单位）

 　　第四个是 调用 callback 的时间间隔（以毫秒为单位）

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 private void button12\_Click(object sender, EventArgs e)

2 {

3 Stopwatch watch = new Stopwatch();

4 watch.Start();

5 Console.WriteLine("----------------- button\_Click 开始 主线程id为：{0} --------------------------", Thread.CurrentThread.ManagedThreadId);

6

7 #region 01-RegisterWaitForSingleObject

8 //{

9 // //每隔3s开启一个线程执行业务逻辑

10 // ThreadPool.RegisterWaitForSingleObject(new AutoResetEvent(true), new WaitOrTimerCallback((obj, b) =>

11 // {

12 // //做逻辑判断，编写业务逻辑

13 // Console.WriteLine("obj={0}，tid={1}, datetime={2}", obj, Thread.CurrentThread.ManagedThreadId, DateTime.Now);

14

15 // }), "hello world", 3000, false);

16 //}

17 #endregion

18

19 #region 02-Timer类

20 {

21 //2秒后开启该线程，然后每隔4s调用一次

22 System.Threading.Timer timer = new System.Threading.Timer((n) =>

23 {

24 //书写业务逻辑

25 Console.WriteLine("我是子线程中的业务逻辑哦");

26 }, "1", 2000, 4000);

27 }

28 #endregion

29

30

31 watch.Stop();

32 Console.WriteLine("----------------- button\_Click 结束 主线程id为：{0} 总耗时：{1}--------------------------", Thread.CurrentThread.ManagedThreadId, watch.ElapsedMilliseconds);

33

34 }

[复制代码](javascript:void(0);)

# [第四节：Task的启动的四种方式以及Task、TaskFactory的线程等待和线程延续的解决方案](https://www.cnblogs.com/yaopengfei/p/8183530.html)

一. 背景

**揭秘：**

　　在前面的章节介绍过，Task出现之前，微软的多线程处理方式有：Thread→ThreadPool→委托的异步调用，虽然也可以基本业务需要的多线程场景，但它们在多个线程的等待处理方面、资源占用方面、线程延续和阻塞方面、线程的取消方面等都显得比较笨拙，在面对复杂的业务场景下，显得有点捉襟见肘了。

　　正是在这种背景下，Task应运而生。

　　Task是微软在.Net 4.0时代推出来的，也是微软极力推荐的一种多线程的处理方式，Task看起来像一个Thread，实际上，它是在ThreadPool的基础上进行的封装，Task的控制和扩展性很强，在线程的延续、阻塞、取消、超时等方面远胜于Thread和ThreadPool。

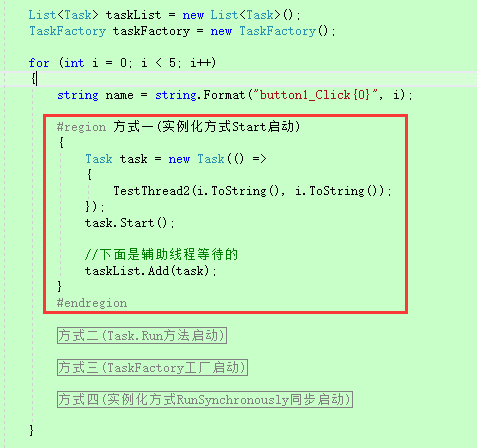
二. Task的4种启动方式

**概要：**

　　Task的启动有4种方式，其中3种异步启动开启一个新线程，1种同步启动的方式（有点和委托类似，BeginInvoke异步启动，Invoke同步启动），分别是：实例化的方式+Start方法启动、Task下Run方法启动、TaskFactory工厂的StartNew方法启动、Task下的同步方法RunSynchronously 启动。

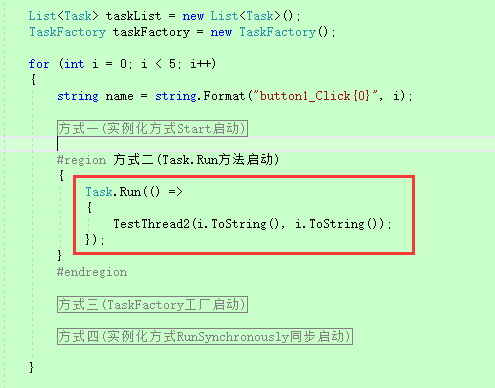
1. 实例化的方式启动,调用Start方法

 　　Task的构造函数中的参数是Action委托(注：不是Action<>多个重载)，所以直接使用 ()=>{   }的方式传参，简洁明了，然后调用Start方式启动。



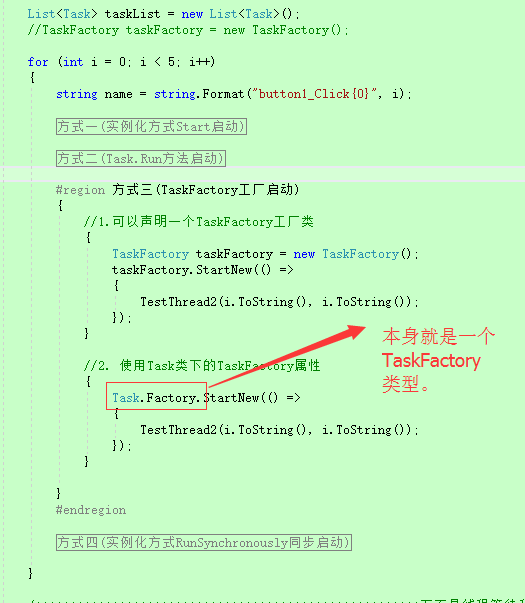
2. 调用Task类下的静态方法Run，进行启动

 　　使用该方式启动，更加简洁，不需要实例化，也不需要调用Start方法，Run方法直接通过Action委托的方式进行传参即可（即:  ()=>{} ）。



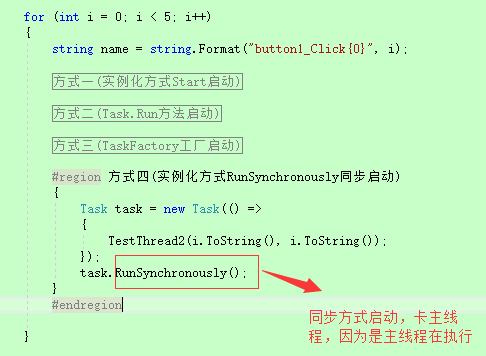
3. TaskFactory工厂启动

 　　使用TaskFactory工厂的StartNew方法启动，其中TaskFactory工厂可以直接实例化，或者 Task.Factory (推荐)。



4. 实例化方式RunSynchronously同步启动

 　　Task实例化的方式，然后调用同步方法RunSynchronously ，进行线程启动。(PS: 类似委托开启线程，BeginInvoke是异步，而Invoke是同步)



三. Task的线程等待和延续

**揭秘：**

　　线程等待和延续通常情况放在一起来说，在同步方法中，即在单线程中，业务代码块按照从上往下的顺序执行，下面的代码块必须要等上面的代码块执行完毕后才能继续执行，这本身就是一种等待和延续，只不过是单线程内的等待和延续。

　　同理，来到多线程领域，这里的等待就不单单局限于代码块之间的等待和延续了，而是上升到某个线程 要等待 另外一个线程执行完毕后方能执行，这里特别说明一下，前面的章节提到线程等待基本上都是主线程在等子线程，当然，完全可能是子线程之间的相互等待和延续(实际上，这种情况更多)。

　　Task下的线程等待和延续主要以下几类：

　　①. Wait：针对单个Task的实例，可以task1.wait进行线程等待.  <Task的实例方法>

　　②. WaitAny：执行的线程等待其中任何一个线程执行完毕即可执行(如果主线程执行,则卡主线程)  <Task的静态方法>

　　③. WaitAll：执行的线程等待其中所有线程执行完毕方可执行(如果主线程执行,则卡主线程)       <Task的静态方法>

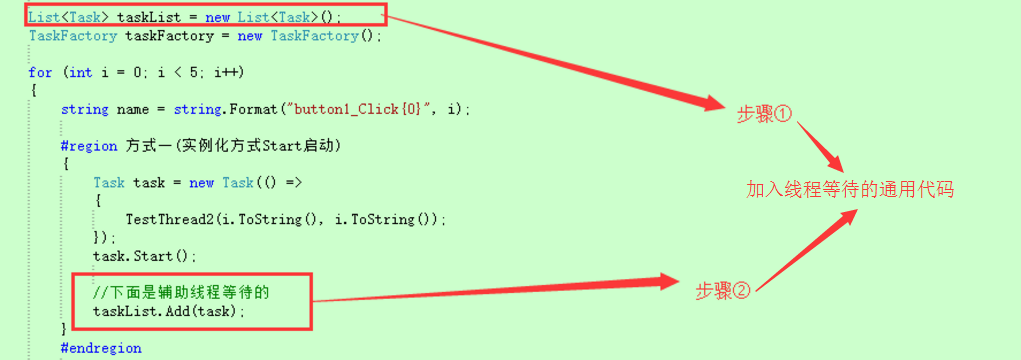
　　④. WhenAny：与下面ContinueWith配合执行,当传入的线程中任何一个线程执行完毕，继续执行ContinueWith中的任务(属于开启新线程，不卡主线程)   <Task的静态方法>

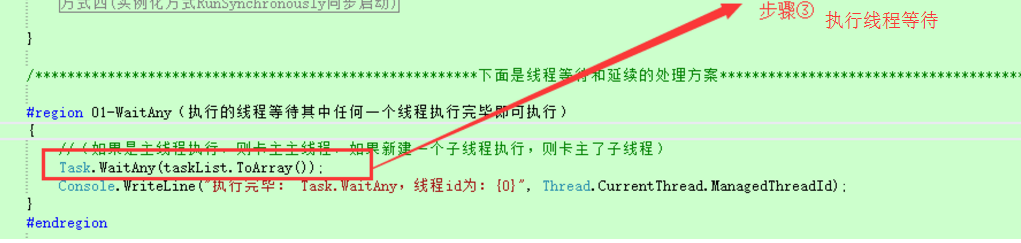
　　⑤. WhenAll：与下面ContinueWith配合执行,当传入的线程中所有线程执行完毕，继续执行ContinueWith中的任务(属于开启新线程，不卡主线程)     <Task的静态方法>

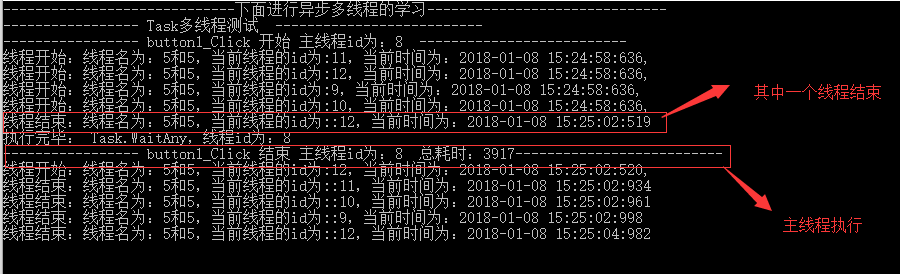
　　⑥. ContinueWith：和上面WhenAny和WhenAll配合使用    <Task的实例方法>

1. WaitAny（执行的线程等待其中任何一个线程执行完毕即可执行）

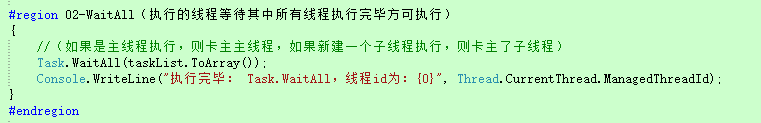
 　　这里给出线程等待加入集合中的代码，下面的线程等待通用这一部分代码，将不再列出。







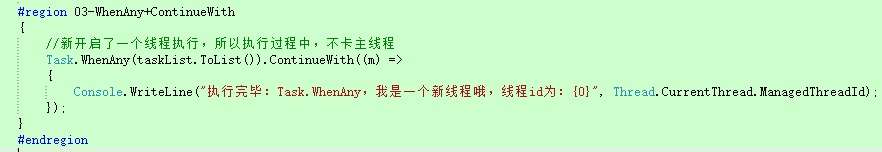
2. WaitAll（执行的线程等待其中所有线程执行完毕方可执行）





3. WhenAny+ContinueWith

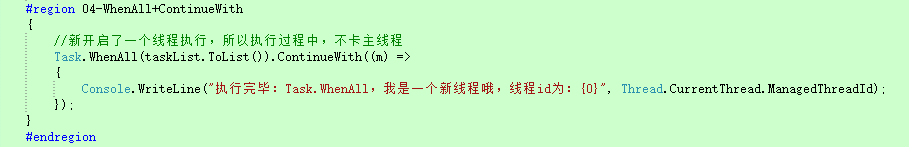
 　　　当其中一个线程执行完成后，新开启了一个线程执行，继续执行新业务，所以执行过程中，不卡主线程。





4. WhenAll+ContinueWith

 　　当其中所有线程执行完成后，新开启了一个线程执行，继续执行新业务，所以执行过程中，不卡主线程。



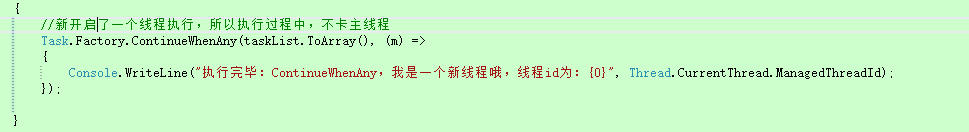
四. TaskFactory的线程等待

**说明：** TaskFactory可以开启线程，当然也对应的线程的等待和延续。

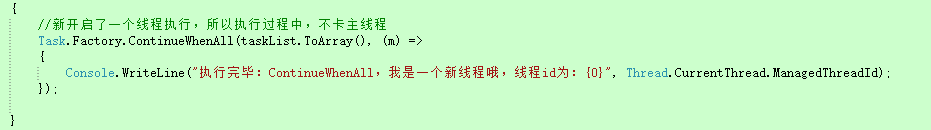
　　①：ContinueWhenAny：等价于Task的WhenAny+ContinueWith

　　②：ContinueWhenAll：等价于Task的WhenAll+ContinueWith

1. ContinueWhenAny



2. ContinueWhenAll



[**第五节：Task构造函数之TaskCreationOptions枚举处理父子线程之间的关系。**](https://www.cnblogs.com/yaopengfei/p/8205337.html)

一. 整体说明

**揭秘：**

通过F12查看Task类的源码(详见下面的截图)，发现Task类的构造函数有有一个参数为：TaskCreationOptions类型，本章节可以算作是一个扩展章节，主要就来研究TaskCreationOptions类的作用。

　　该类主要用来处理父子线程之间的关系，重要的几个参数如下：

　　①.AttachedToParent:指定将任务附加到任务层次结构中的某个父级,父任务必须等待所有子任务执行完毕才能执行

**（下面的例子task线程必须等task1和task2线程执行完毕才能执行）**

　　②. DenyChildAttach: 不允许子任务附加到父任务上

**(下面例子task不再等待task1和task2,和00的默认效果相同)**

　　③. HideScheduler： 子任务不使用父类Task的Scheduler，而是使用默认的 (不进行测试)

　　④. LongRunning：当已知是长时间运行的任务，可以使用该选项 (不进行测试)

　　⑤. PreferFairness：类似于队列的感觉，尽可能公平的方式安排任务 (不进行测试)

 源码如下：

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 // 摘要:

2 // 指定可控制任务的创建和执行的可选行为的标志。

3 [Serializable]

4 [Flags]

5 public enum TaskCreationOptions

6 {

7 // 摘要:

8 // 指定应使用默认行为。

9 None = 0,

10 //

11 // 摘要:

12 // 提示 System.Threading.Tasks.TaskScheduler 以一种尽可能公平的方式安排任务，这意味着较早安排的任务将更可能较早运行，而较晚安排运行的任务将更可能较晚运行。

13 PreferFairness = 1,

14 //

15 // 摘要:

16 // 指定某个任务将是运行时间长、粗粒度的操作。 它会向 System.Threading.Tasks.TaskScheduler 提示，过度订阅可能是合理的。

17 LongRunning = 2,

18 //

19 // 摘要:

20 // 指定将任务附加到任务层次结构中的某个父级。

21 AttachedToParent = 4,

22 //

23 // 摘要:

24 // 如果尝试附有子任务到创建的任务，指定 System.InvalidOperationException 将被引发。

25 DenyChildAttach = 8,

26 //

27 // 摘要:

28 // 防止环境计划程序被视为已创建任务的当前计划程序。 这意味着像 StartNew 或 ContinueWith 创建任务的执行操作将被视为 System.Threading.Tasks.TaskScheduler.Default

29 // 当前计划程序。

30 HideScheduler = 16,

31 }

[复制代码](javascript:void(0);)

View Code

二. 实际测试

 　　这里我们主要通过代码来比较默认情况下、AttachedToParent、DenyChildAttach之间的效果， task线程内部有task1和task2线程，并且在task内部开启。

**1. 默认情况**

[复制代码](javascript:void(0);)

1 {

2 Stopwatch watch = new Stopwatch();

3 watch.Start();

4 Console.WriteLine("----------------- Task多线程测试 --------------------------");

5 Console.WriteLine("----------------- button1\_Click 开始 主线程id为：{0} --------------------------", Thread.CurrentThread.ManagedThreadId);

6

7 #region 00-默认

8 {

9 Task task = new Task(() =>

10 {

11 Task task1 = new Task(() =>

12 {

13 Thread.Sleep(1000);

14 Console.WriteLine("我是task1线程");

15 });

16 Task task2 = new Task(() =>

17 {

18 Thread.Sleep(1000);

19 Console.WriteLine("我是task2线程");

20 });

21

22 task1.Start();

23 task2.Start();

24 });

25

26 task.Start();

27 task.Wait(); //单个线程的等待

28 Console.WriteLine("------------------我是主线程--------------------");

29 }

30 #endregion

31

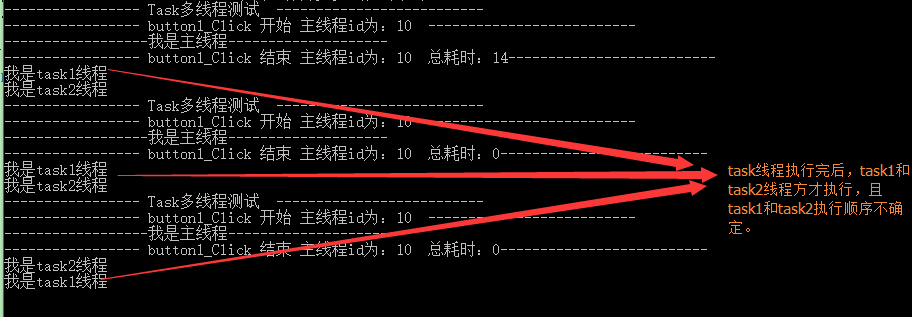
32 watch.Stop();

33 Console.WriteLine("----------------- button1\_Click 结束 主线程id为：{0} 总耗时：{1}--------------------------", Thread.CurrentThread.ManagedThreadId, watch.ElapsedMilliseconds);

34 }

[复制代码](javascript:void(0);)

**多次执行上述代码看效果：发现task线程执行完后，task1和task2才无序的执行。**

****

**2. AttachedToParent**

作用：指定将任务附加到任务层次结构中的某个父级,父任务必须等待所有子任务执行完毕才能执行

[复制代码](javascript:void(0);)

1 {

2 Task task = new Task(() =>

3 {

4 Task task1 = new Task(() =>

5 {

6 Thread.Sleep(3000);

7 Console.WriteLine("我是task1线程");

8 }, TaskCreationOptions.AttachedToParent);

9 Task task2 = new Task(() =>

10 {

11 Thread.Sleep(3000);

12 Console.WriteLine("我是task2线程");

13 }, TaskCreationOptions.AttachedToParent);

14

15 task1.Start();

16 task2.Start();

17 });

18

19 task.Start();

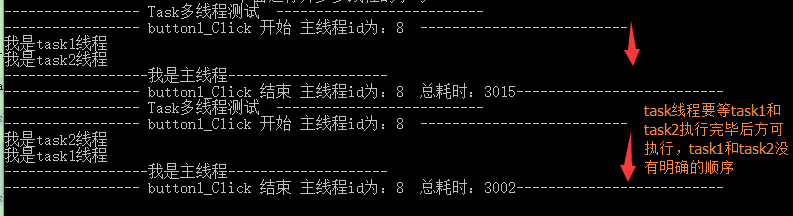
20 task.Wait(); //单个线程的等待

21 Console.WriteLine("------------------我是主线程--------------------");

22 }

[复制代码](javascript:void(0);)

**多次执行上述代码看效果：发现task线程必须等task1和task2执行完毕后才能执行（印证了AttachedToParent的作用），task1和task2无先后顺序。**



**3. DenyChildAttach**

 　　作用：不允许子任务附加到父任务上。

[复制代码](javascript:void(0);)

1 {

2 Task task = new Task(() =>

3 {

4 Task task1 = new Task(() =>

5 {

6 Thread.Sleep(3000);

7 Console.WriteLine("我是task1线程");

8 }, TaskCreationOptions.AttachedToParent);

9 Task task2 = new Task(() =>

10 {

11 Thread.Sleep(3000);

12 Console.WriteLine("我是task2线程");

13 }, TaskCreationOptions.AttachedToParent);

14

15 task1.Start();

16 task2.Start();

17 }, TaskCreationOptions.DenyChildAttach);

18

19 task.Start();

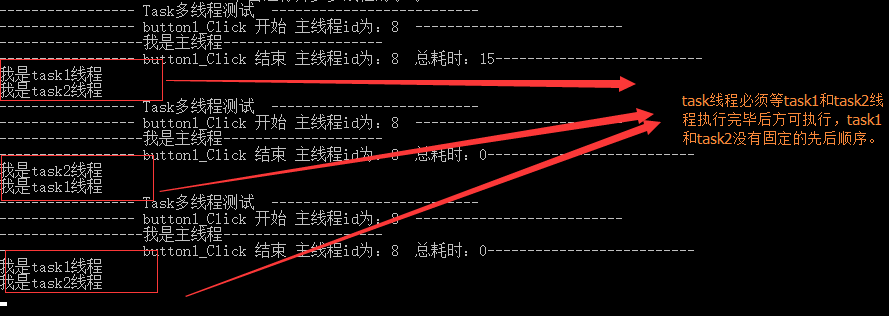
20 task.Wait(); //单个线程的等待

21 Console.WriteLine("------------------我是主线程--------------------");

22 }

[复制代码](javascript:void(0);)

**多次执行上述代码看效果：发现task线程执行完后，task1和task2才无序的执行。（和上述的默认情况是一致的）**



[**第六节：深入研究Task实例方法ContinueWith的参数TaskContinuationOptions**](https://www.cnblogs.com/yaopengfei/p/8213263.html)

一. 整体说明

**揭秘：**

　　该章节的性质和上一个章节类似，也是一个扩展的章节，主要来研究Task类下的实例方法ContinueWith中的参数TaskContinuationOptions。

　　通过F12查看TaskContinuationOptions的源码，知道主要有这么几个参数：

　　①. LazyCancellation：在延续取消的情况下，防止延续的完成直到完成先前的任务。

**(下面的例子task2取消，原先的延续关系不复存在，task1和task3可以并行执行)**

　　②. ExecuteSynchronously：希望执行前面那个task的thread也在执行本延续任务

**(下面的例子执行task2的Thread和执行task1的Thread是同一个，所有二者的线程id相同)**

　　③. NotOnRanToCompletion和OnlyOnRanToCompletion

　　　　NotOnRanToCompletion：延续任务必须在前面task非完成状态才能执行

　　　　OnlyOnRanToCompletion：延续任务必须在前面task完成状态才能执行

**(下面例子：注释掉异常的这句代码task2不能执行，task3能执行；不注释，task2能执行，task3不能执行)**

**源码如下：**

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 [Serializable]

2 [Flags]

3 public enum TaskContinuationOptions

4 {

5 // 摘要:

6 // Default = "Continue on any, no task options, run asynchronously" 指定应使用默认行为。

7 // 默认情况下，完成前面的任务之后将安排运行延续任务，而不考虑前面任务的最终 System.Threading.Tasks.TaskStatus。

8 None = 0,

9 //

10 // 摘要:

11 // 提示 System.Threading.Tasks.TaskScheduler 以一种尽可能公平的方式安排任务，这意味着较早安排的任务将更可能较早运行，而较晚安排运行的任务将更可能较晚运行。

12 PreferFairness = 1,

13 //

14 // 摘要:

15 // 指定某个任务将是运行时间长、粗粒度的操作。 它会向 System.Threading.Tasks.TaskScheduler 提示，过度订阅可能是合理的。

16 LongRunning = 2,

17 //

18 // 摘要:

19 // 指定将任务附加到任务层次结构中的某个父级。

20 AttachedToParent = 4,

21 //

22 // 摘要:

23 // 如果尝试附有子任务到创建的任务，指定 System.InvalidOperationException 将被引发。

24 DenyChildAttach = 8,

25 //

26 // 摘要:

27 // 防止环境计划程序被视为已创建任务的当前计划程序。 这意味着像 StartNew 或 ContinueWith 创建任务的执行操作将被视为 System.Threading.Tasks.TaskScheduler.Default

28 // 当前计划程序。

29 HideScheduler = 16,

30 //

31 // 摘要:

32 // 在延续取消的情况下，防止延续的完成直到完成先前的任务。

33 LazyCancellation = 32,

34 //

35 // 摘要:

36 // 指定不应在延续任务前面的任务已完成运行的情况下安排延续任务。 此选项对多任务延续无效。

37 NotOnRanToCompletion = 65536,

38 //

39 // 摘要:

40 // 指定不应在延续任务前面的任务引发了未处理异常的情况下安排延续任务。 此选项对多任务延续无效。

41 NotOnFaulted = 131072,

42 //

43 // 摘要:

44 // 指定只应在延续任务前面的任务已取消的情况下才安排延续任务。 此选项对多任务延续无效。

45 OnlyOnCanceled = 196608,

46 //

47 // 摘要:

48 // 指定不应在延续任务前面的任务已取消的情况下安排延续任务。 此选项对多任务延续无效。

49 NotOnCanceled = 262144,

50 //

51 // 摘要:

52 // 指定只应在延续任务前面的任务引发了未处理异常的情况下才安排延续任务。 此选项对多任务延续无效。

53 OnlyOnFaulted = 327680,

54 //

55 // 摘要:

56 // 指定只应在延续任务前面的任务已完成运行的情况下才安排延续任务。 此选项对多任务延续无效。

57 OnlyOnRanToCompletion = 393216,

58 //

59 // 摘要:

60 // 指定应同步执行延续任务。 指定此选项后，延续任务将在导致前面的任务转换为其最终状态的相同线程上运行。 如果在创建延续任务时已经完成前面的任务，则延续任务将在创建此延续任务的线程上运行。

61 // 只应同步执行运行时间非常短的延续任务。

62 ExecuteSynchronously = 524288,

63 }

[复制代码](javascript:void(0);)

View Code

二. 实际测试

 　　下面通过代码来说明默认情况下、LazyCancellation、ExecuteSynchronously、NotOnRanToCompletion和OnlyOnRanToCompletion的作用和效果。

1. 默认情况

 　　默认情况下，task1执行完后→task2→task2执行完后→task3。

[复制代码](javascript:void(0);)

1 {

2 Task task1 = new Task(() =>

3 {

4 Thread.Sleep(1000);

5 Console.WriteLine("task1 tid={0}， dt={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

6 });

7

8 var task2 = task1.ContinueWith(t =>

9 {

10 Console.WriteLine("task2 tid={0}， dt={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

11 });

12

13 var task3 = task2.ContinueWith(t =>

14 {

15 Console.WriteLine("task3 tid={0}， dt={1} {2}", Thread.CurrentThread.ManagedThreadId, DateTime.Now, task2.Status);

16 });

17

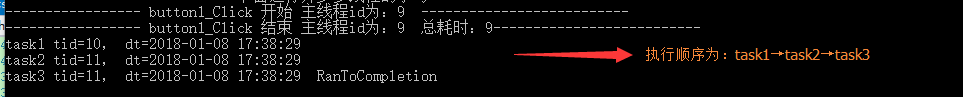
18 task1.Start();

19

20 }

[复制代码](javascript:void(0);)

**运行结果： task1执行完后→ task2执行→task2执行完后→ task3执行。**



2. LazyCancellation

**作用：取消该线程，该线程的前一个线程和后一个线程并行执行。**

[复制代码](javascript:void(0);)

1 {

2 CancellationTokenSource source = new CancellationTokenSource();

3 source.Cancel();

4

5 Task task1 = new Task(() =>

6 {

7 Thread.Sleep(1000);

8 Console.WriteLine("task1 tid={0}， dt={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

9 });

10

11 var task2 = task1.ContinueWith(t =>

12 {

13 Console.WriteLine("task2 tid={0}， dt={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

14 }, source.Token, TaskContinuationOptions.LazyCancellation, TaskScheduler.Current);

15

16 var task3 = task2.ContinueWith(t =>

17 {

18 Console.WriteLine("task3 tid={0}， dt={1} {2}", Thread.CurrentThread.ManagedThreadId, DateTime.Now, task2.Status);

19 });

20

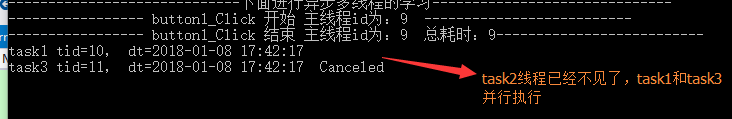
21 task1.Start();

22

23 }

[复制代码](javascript:void(0);)

**运行结果： task2线程已经被取消，task1线程和task2线程并行执行。**



3. ExecuteSynchronously

**作用：希望执行前面那个task的thread也在执行本延续任务。**

[复制代码](javascript:void(0);)

1 {

2 Task task1 = new Task(() =>

3 {

4 Thread.Sleep(1000);

5 Console.WriteLine("task1 tid={0}， dt={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

6 });

7

8 var task2 = task1.ContinueWith(t =>

9 {

10 Console.WriteLine("task2 tid={0}， dt={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

11 }, TaskContinuationOptions.ExecuteSynchronously);

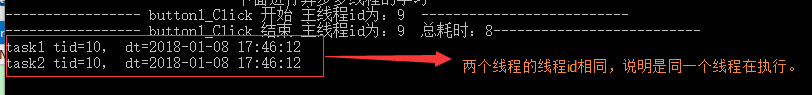
12

13 task1.Start();

14 }

[复制代码](javascript:void(0);)

**结果：task1和task2线程的线程id相同。**



4. NotOnRanToCompletion和OnlyOnRanToCompletion

　　NotOnRanToCompletion：延续任务必须在前面task非完成状态才能执行。

　　OnlyOnRanToCompletion：延续任务必须在前面task完成状态才能执行。

[复制代码](javascript:void(0);)

{

Task task1 = new Task(() =>

{

Thread.Sleep(1000);

Console.WriteLine("task1 tid={0}， dt={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

//手动制造异常，表示不能执行完毕

//（注释掉这句话task2不能执行，task3能执行）

//不注释，task2能执行，task3不能执行

//throw new Exception("hello world");

});

var task2 = task1.ContinueWith(t =>

{

Console.WriteLine("task2 tid={0}， dt={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

}, TaskContinuationOptions.NotOnRanToCompletion);

var task3 = task1.ContinueWith(t =>

{

Console.WriteLine("task3 tid={0}， dt={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

}, TaskContinuationOptions.OnlyOnRanToCompletion);

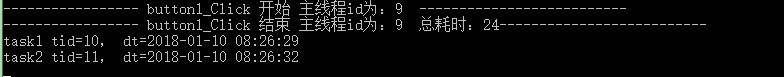
task1.Start();

}

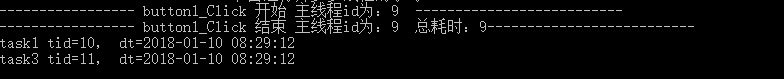
[复制代码](javascript:void(0);)

**分析：task2和task3均为task的延续线程，当task1报错时候，task2执行，task3不能执行；当task1正常时候，task2不能执行，task3能执行。**

**task1报错时的运行结果：**



**task2正常时的运行结果：**



[**第七节：利用CancellationTokenSource实现任务取消和利用CancellationToken类检测取消异常。**](https://www.cnblogs.com/yaopengfei/p/8229175.html)

一. 传统的线程取消

 　　所谓的线程取消，就是线程正在执行的过程中取消线程任务。

 　　传统的线程取消，是通过一个变量来控制，但是这种方式，在release模式下，被优化从cpu高速缓存中读取，而不是从内存中读取，会造成主线程无法执行这一个bug。

[复制代码](javascript:void(0);)

1 {

2 var isStop = false;

3 var thread = new Thread(() =>

4 {

5 while (!isStop)

6 {

7 Thread.Sleep(100);

8 Console.WriteLine("当前thread={0} 正在运行", Thread.CurrentThread.ManagedThreadId);

9 }

10 });

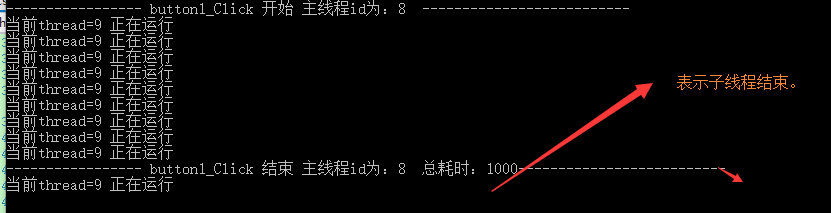
11 thread.Start();

12 Thread.Sleep(1000);

13 isStop = true;

14 }

[复制代码](javascript:void(0);)



**PS：** 通过上面的代码看可以看出来，传统模式的线程取消，在排除release模式bug的情况下，局限性还是很明显的。比如：当子线程任务取消的那一刻，我想执行另外一项任务；我想延时取消一个线程任务；线程取消的时候抛异常。

　　上述这几种情况，我们都要借助单独的类来处理。

二. CancellationTokenSource实现任务取消

1. 取消任务的同时触发一个函数

 　　利用Cancel方法、Register注册、source.Token标记取消位来实现。

[复制代码](javascript:void(0);)

{

CancellationTokenSource source = new CancellationTokenSource();

//注册一个线程取消后执行的逻辑

source.Token.Register(() =>

{

//这里执行线程被取消后的业务逻辑.

Console.WriteLine("-------------我是线程被取消后的业务逻辑---------------------");

});

Task.Run(() =>

{

while (!source.IsCancellationRequested)

{

Thread.Sleep(100);

Console.WriteLine("当前thread={0} 正在运行", Thread.CurrentThread.ManagedThreadId);

}

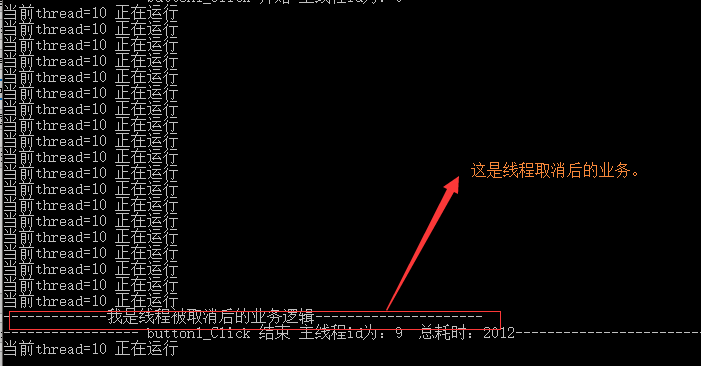
}, source.Token);

Thread.Sleep(2000);

source.Cancel();

}

[复制代码](javascript:void(0);)



2. 延时取消

线程的延时取消有两种方式：

　　方案一：CancelAfter方法。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 #region 方案一：CancelAfter方法

2 {

3 CancellationTokenSource source = new CancellationTokenSource();

4 //注册一个线程取消后执行的逻辑

5 source.Token.Register(() =>

6 {

7 //这里执行线程被取消后的业务逻辑.

8 Console.WriteLine("-------------我是线程被取消后的业务逻辑---------------------");

9 });

10

11 Task.Run(() =>

12 {

13 while (!source.IsCancellationRequested)

14 {

15 Thread.Sleep(100);

16 Console.WriteLine("当前thread={0} 正在运行", Thread.CurrentThread.ManagedThreadId);

17 }

18 }, source.Token);

19

20 Thread.Sleep(2000);

21 //4s后自动取消

22 source.CancelAfter(new TimeSpan(0, 0, 0, 4));

23 }

24 #endregion

[复制代码](javascript:void(0);)

View Code

　　方案二：CancellationTokenSource构造函数（不再需要Cancel方法了）。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 {

2 //4s后自动取消

3 CancellationTokenSource source = new CancellationTokenSource(4000);

4 //注册一个线程取消后执行的逻辑

5 source.Token.Register(() =>

6 {

7 //这里执行线程被取消后的业务逻辑.

8 Console.WriteLine("-------------我是线程被取消后的业务逻辑---------------------");

9 });

10

11 Task.Run(() =>

12 {

13 while (!source.IsCancellationRequested)

14 {

15 Thread.Sleep(100);

16 Console.WriteLine("当前thread={0} 正在运行", Thread.CurrentThread.ManagedThreadId);

17 }

18 }, source.Token);

19

20 Thread.Sleep(2000);

21 }

[复制代码](javascript:void(0);)

View Code

3. 组合取消

　　 利用CreateLinkedTokenSource构建CancellationTokenSource的组合体，其中任何一个体取消，则组合体就取消。

[复制代码](javascript:void(0);)

{

CancellationTokenSource source1 = new CancellationTokenSource();

//source1.Cancel();

CancellationTokenSource source2 = new CancellationTokenSource();

source2.Cancel();

var combineSource = CancellationTokenSource.CreateLinkedTokenSource(source1.Token, source2.Token);

Console.WriteLine("s1={0} s2={1} s3={2}", source1.IsCancellationRequested,

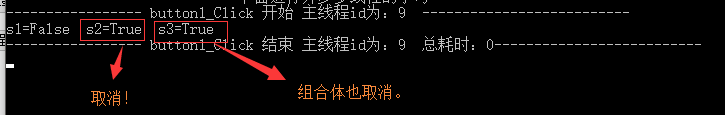
source2.IsCancellationRequested,

combineSource.IsCancellationRequested);

}

[复制代码](javascript:void(0);)

　　上述代码，source1和source2中的任何一个取消，combineSource就会被取消。



三. CancellationToken类监控取消

 　　CancellationToken类下ThrowIfCancellationRequested属性，等价于if (XXX.IsCancellationRequested){throw new Exception("报错了");}

**只要取消就报错。**

[复制代码](javascript:void(0);)

1 {

2 CancellationTokenSource source1 = new CancellationTokenSource();

3 CancellationTokenSource source2 = new CancellationTokenSource();

4 var combineSource = CancellationTokenSource.CreateLinkedTokenSource(source1.Token, source2.Token);

5 source1.Cancel();

6

7 //if (combineSource.IsCancellationRequested)

8 //{

9 // throw new Exception("报错了");

10 //}

11

12 //等价于上面那句话

13 try

14 {

15 combineSource.Token.ThrowIfCancellationRequested();

16 }

17 catch (Exception)

18 {

19 Console.WriteLine("报错了");

20 }

21

22

23 Console.WriteLine("s1={0} s2={1} s3={2}", source1.IsCancellationRequested,

24 source2.IsCancellationRequested,

25 combineSource.IsCancellationRequested);

26 }

[复制代码](javascript:void(0);)

[**第八节：Task的各类Task<TResult>返回值以及通用线程的异常处理方案。**](https://www.cnblogs.com/yaopengfei/p/8256760.html)

一. Task的各种返回值-Task<TResult>

**PS：** 在前面章节，我们介绍了Task类开启线程、线程等待、线程延续的方式，但我们并没有关注这些方式的返回值，其实他们都是有返回值的Task<TResult>，然后可以通过Task的实例调用Result属性来获取这个返回值。

　　下面我们分三类来介绍：

　　①：线程开启类的返回值, 使用Task<TResult>接受，或者直接使用Task接受，通过 实例.Result 来获取返回值。这里的线程开启类有多种，eg: Task.Run()、 task.start()、 Task.Factory.StartNew() 等。

　　②：线程延续类的返回值. eg：ContinueWith。

　　③：线程条件延续类的返回值. eg：WhenAll和WhenAny。

**1. 线程开启类的返回值**

[复制代码](javascript:void(0);)

1 {

2 Task<string> task1 = Task.Factory.StartNew(() =>

3 {

4 Console.WriteLine("我是子线程哦");

5 return "ok";

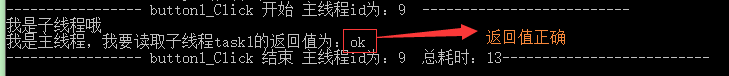
6 });

7 task1.Wait();

8 Console.WriteLine("我是主线程，我要读取子线程task1的返回值为：{0}", task1.Result);

9 }

[复制代码](javascript:void(0);)



**2.  线程延续类的返回值**

[复制代码](javascript:void(0);)

1 {

2 Task<int> task1 = Task.Run(() =>

3 {

4 Console.WriteLine("我是子线程1哦");

5 return 2;

6 });

7

8 var task2 = task1.ContinueWith((t) =>

9 {

10 Console.WriteLine("我是子线程2哦");

11

12 //这里的t代表 task1

13 var num = t.Result + 2;

14 return num.ToString();

15 });

16

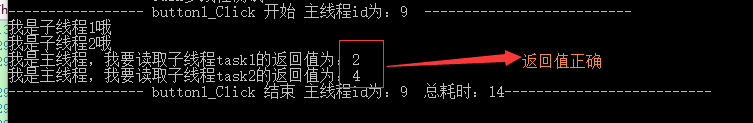
17 task2.Wait();

18 Console.WriteLine("我是主线程，我要读取子线程task1的返回值为：{0}", task1.Result);

19 Console.WriteLine("我是主线程，我要读取子线程task2的返回值为：{0}", task2.Result);

20 }

[复制代码](javascript:void(0);)



**3. 线程条件延续类**

[复制代码](javascript:void(0);)

1 {

2 Task<int> task1 = Task.Run(() =>

3 {

4 Console.WriteLine("我是子线程1哦");

5 return 1;

6 });

7 Task<int> task2 = Task.Run(() =>

8 {

9 Console.WriteLine("我是子线程2哦");

10 return 2;

11 });

12

13 var task = Task.WhenAny(new Task<int>[2] { task1, task2 });

14 task.Wait();

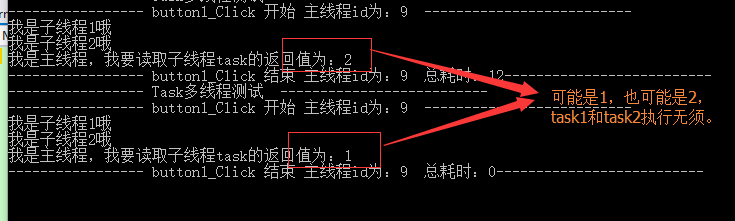
15

16 //下面的值可能是1，也可能是2

17 Console.WriteLine("我是主线程，我要读取子线程task的返回值为：{0}", task.Result.Result);

18 }

[复制代码](javascript:void(0);)



二. 通用线程异常处理方案

1.  背景：我们想达到一个目的，当同时开启多个线程的时候，其中一个线程报错，不影响其他线程的执行，并且能把错误记下来。

2.   解决方案：多重try-catch，整个外侧主线程一个try-catch，然后线程执行业务再用一个try-catch包裹起来。

**常规方式捕获异常：**

[复制代码](javascript:void(0);)

1 {

2 try

3 {

4 for (int i = 0; i < 5; i++)

5 {

6 string name = string.Format("name{0}", i);

7 var task = Task.Run(() =>

8 {

9 try

10 {

11 //模拟某个线程出错

12 if (name == "name2")

13 {

14 throw new Exception(string.Format("线程执行失败，i={0}", name));

15 }

16 else

17 {

18 Console.WriteLine(string.Format("线程执行执行成功，i={0}", name));

19 }

20 }

21 catch (Exception ex)

22 {

23 Console.WriteLine(ex.Message);

24 }

25

26 });

27 taskList.Add(task);

28 }

29 Task.WaitAll(taskList.ToArray());

30 }

31 catch (Exception ex)

32 {

33 Console.WriteLine(ex.Message);

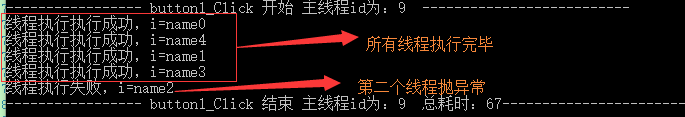
34

35 }

36 }

[复制代码](javascript:void(0);)

**运行结果：我们发现所有的线程均执行完毕，且name2执行失败，并捕获。**



补充一下：**通过 AggregateException 类来捕获异常。**

[复制代码](javascript:void(0);)

1 {

2 try

3 {

4 for (int i = 0; i < 5; i++)

5 {

6 string name = string.Format("name{0}", i);

7 var task = Task.Run(() =>

8 {

9 throw new Exception(string.Format("线程执行失败，i={0}", name));

10 });

11 taskList.Add(task);

12 }

13 Task.WaitAll(taskList.ToArray());

14 }

15 catch (AggregateException aes)

16 {

17 foreach (var item in aes.InnerExceptions)

18 {

19 Console.WriteLine(item.Message);

20 }

21 }

22 }

[复制代码](javascript:void(0);)

[**第九节：深究并行编程Parallel类中的三大方法 (For、ForEach、Invoke）和几大编程模型(SPM、APM、EAP、TAP）**](https://www.cnblogs.com/yaopengfei/p/8279848.html)

一. 并行编程

**1. 区分串行编程和串行编程**

　　①. 串行编程：所谓的串行编程就是单线程的作用下，按顺序执行。(典型代表for循环 下面例子从1-100按顺序执行)

　　②. 并行编程：充分利用多核cpu的优势，同时开启多个线程并行执行。(典型代表Parallel.For循环 下面例子从1-100无序执行)

**代码实践：**

按 Ctrl+C 复制代码



按 Ctrl+C 复制代码

**结论：串行的代码按顺序依次输出，并行的代码无顺序输出。**

**2. 深究Parallel类中的方法 （For方法、ForEach方法、Invoke方法 这三个方法都是用来开启线程的）**

(1). Invoke方法

　　a. 该方法的作用就是用来同时开启多个线程的。

　　b. 该方法有两个重载，主要涉及到两个参数，用来配置最大并行数(即线程数)和一个可变的Action委托数组（详见源码）。

**案例一： 开启五个不同的线程调用五个方法**

我们发现一个现象，主线程等着这五个子线程执行完毕后才执行，但是我们并没有写线程等待的代码，所以我们可以总结：

①：并行计算，开启多个线程后，不需要再开辟线程等待，直接是主线程完成后续操作。

②：而普通多线程执行后，需要单独再开辟一个线程等待，然后主线程在执行。

**代码实践：**

[复制代码](javascript:void(0);)

1 {

2 Parallel.Invoke(() => this.TestThread("bct1")

3 , () => this.TestThread("bct2")

4 , () => this.TestThread("bct3")

5 , () => this.TestThread("bct4")

6 , () => this.TestThread("bct5")

7 );

8 }

[复制代码](javascript:void(0);)



**案例二： 指定最大并行数进行线程调用**

我们发现，五个任务中的四个任务同时由不同线程开启，当其中一个任务结束时，第五个任务开启，并由刚结束的任务的线程来执行。

[复制代码](javascript:void(0);)

{

//设置最大的线程并行数

ParallelOptions p = new ParallelOptions();

p.MaxDegreeOfParallelism = 4;

Parallel.Invoke(p, () => this.TestThread("bct1")

, () => this.TestThread("bct2")

, () => this.TestThread("bct3")

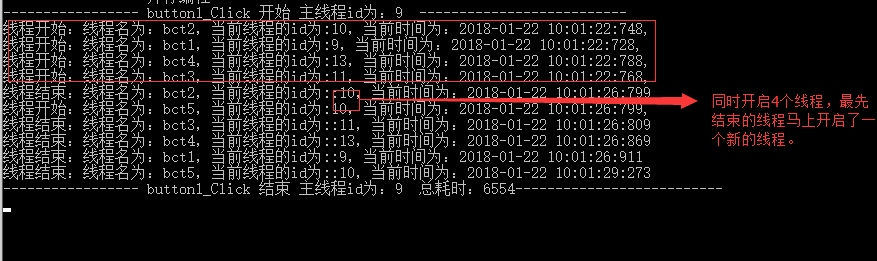
, () => this.TestThread("bct4")

, () => this.TestThread("bct5")

);

}

[复制代码](javascript:void(0);)



(2). For方法 (前两个参数之间的差代表任务的个数)

　　这里介绍一个简单重载： public static ParallelLoopResult For(int fromInclusive, int toExclusive, Action<int> body);

　　fromInclusive:开始索引（含）.

　　toExclusive:结束索引（不含）.

　　body:将为每个迭代调用一次的委托.

当然该方法中的其他重载中也有很丰富的功能，比如也可以配置最大线程数。

**代码实践：**

[复制代码](javascript:void(0);)

1 {

2 //案例一：前两个参数之间的差，就为并行计算线程的个数

3 {

4 Parallel.For(5, 10, t =>

5 {

6 //这里的t分别为：5,6,7,8,9 五个数

7 string name = string.Format("bct{0}", t);

8 this.TestThread(name);

9 });

10 }

11 //案例二: 配置最大并行数

12 //结果：同时最多5个线程执行，但是还是要执行9个任务，(6,7,8,9,10,11,12,13,14),后面四个任务等前面的执行完后，再执行

13 {

14 ParallelOptions po = new ParallelOptions()

15 {

16 MaxDegreeOfParallelism = 5 //表示最大线程数为5，后面即使配置超过5，也无效

17 };

18 Parallel.For(6, 15, po, (t, state) =>

19 {

20 string name = string.Format("bct{0}", t);

21 this.TestThread(name);

22 //state.Break(); //退出单次循环(没看到实际作用)

23 // state.Stop(); //退出全部循环(没看到实际作用)

24 //return;

25 });

26 }

27 }

[复制代码](javascript:void(0);)

(3). ForEach方法

　　这里也是介绍一个简单的重载：int数组中的个数代表需要进行并行任务的个数，但并不一定所有任务同时执行，也不一定每个任务都是一个新线程执行。

该方法当然也可以配置最大并行数。

**代码实践：**

[复制代码](javascript:void(0);)

{

//数组里的个数，就为并行进行并行任务数

Parallel.ForEach(new int[] { 3, 5, 44, 55, 100 }, t =>

{

//这里的t分别为：3, 5, 44, 55, 100五个数

string name = string.Format("bct{0}", t);

this.TestThread(name);

}

}

[复制代码](javascript:void(0);)

二. 常见的编程模型

1.同步编程模型(SPM)：单线线程、串行开发模式。

2.异步编程模型(APM)：xxxbegin、xxxend的模式。

3.基于事件的编程模型(EAP)： xxAsync这样的事件模式。 eg：WebClient。

4.基于Task的编程模型(TAP)： APM和EAP都可以使用Task来实现，微软的初衷就是想通过Task大一统异步编程领域。

**下面分享两段代码，不做深入研究了。**

[复制代码](javascript:void(0);)

1 {

2 FileStream fs = new FileStream(Environment.CurrentDirectory + "//1.txt", FileMode.Open);

3 var bytes = new byte[fs.Length];

4 var task = Task.Factory.FromAsync(fs.BeginRead, fs.EndRead, bytes, 0, bytes.Length, string.Empty);

5

6 var nums = task.Result;

7

8 Console.WriteLine(nums);

9 }

10 {

11 FileStream fs = new FileStream(Environment.CurrentDirectory + "//1.txt", FileMode.Open);

12

13 var bytes = new byte[fs.Length];

14

15 fs.BeginRead(bytes, 0, bytes.Length, (aysc) =>

16 {

17 var nums = fs.EndRead(aysc);

18

19 Console.WriteLine(nums);

20

21 }, string.Empty);

22

23 Console.Read();

24 }

[复制代码](javascript:void(0);)

[**第十节：利用async和await简化异步编程模式的几种写法**](https://www.cnblogs.com/yaopengfei/p/8283956.html)

一. async和await简介

PS：简介

1. async和await这两个关键字是为了简化异步编程模型而诞生的，使的异步编程跟简洁，它本身并不创建新线程，但在该方法内部开启多线程，则另算。

2. 这两个关键字适用于处理一些文件IO操作。

3. 好处：代码简介，把异步的代码写成了同步的形式，提高了开发效率。

　坏处：如果使用同步思维去理解，容易出问题，返回值对不上。

二. 几种用法

情况1：当只有async，没有await时，方法会有个警告，和普通的多线程方法没有什么区别，不存在线程等待的问题。

**代码实践：**

[复制代码](javascript:void(0);)

1 private static async void Test1()

2 {

3 //主线程执行

4 Console.WriteLine("主线程{0}开始：", Thread.CurrentThread.ManagedThreadId);

5 //启动新线程完成任务

6 Task task = Task.Run(() =>

7 {

8 Console.WriteLine("子线程{0}开始：", Thread.CurrentThread.ManagedThreadId);

9 Thread.Sleep(3000);

10 Console.WriteLine("子线程{0}结束：", Thread.CurrentThread.ManagedThreadId);

11 });

12 //主线程执行

13 Console.WriteLine("主线程{0}结束：", Thread.CurrentThread.ManagedThreadId);

14 }

[复制代码](javascript:void(0);)

**代码结果：**



情况2：不推荐void返回值，使用Task来代替Task和Task<T>能够使用await, Task.WhenAny, Task.WhenAll等方式组合使用，async Void 不行。

**代码实践：**

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 不推荐void返回值，使用Task来代替

3 /// Task和Task<T>能够使用await, Task.WhenAny, Task.WhenAll等方式组合使用。async Void 不行

4 /// </summary>

5 private static async void Test2()

6 {

7 //主线程执行

8 Console.WriteLine("主线程{0}开始：", Thread.CurrentThread.ManagedThreadId);

9 //启动新线程完成任务

10 Task task = Task.Run(() =>

11 {

12 Console.WriteLine("子线程{0}开始：", Thread.CurrentThread.ManagedThreadId);

13 Thread.Sleep(3000);

14 Console.WriteLine("子线程{0}结束：", Thread.CurrentThread.ManagedThreadId);

15 });

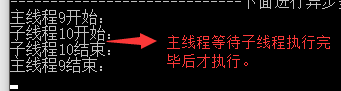
16 await task; //等待子线程执行完毕，方可执行后面的语句

17 Console.WriteLine("主线程{0}结束：", Thread.CurrentThread.ManagedThreadId);

18 }

[复制代码](javascript:void(0);)

**代码结果：**



情况3：async Task == async void。 区别：Task和Task<T>能够使用await, Task.WhenAny, Task.WhenAll等方式组合使用，async Void 不行。

**代码实践：**

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 无返回值 async Task == async void

3 /// Task和Task<T>能够使用await, Task.WhenAny, Task.WhenAll等方式组合使用,async Void 不行

4 /// </summary>

5 private static async Task Test3()

6 {

7 //主线程执行

8 Console.WriteLine("主线程{0}开始：", Thread.CurrentThread.ManagedThreadId);

9 //启动新线程完成任务

10 Task task = Task.Run(() =>

11 {

12 Console.WriteLine("子线程{0}开始：", Thread.CurrentThread.ManagedThreadId);

13 Thread.Sleep(3000);

14 Console.WriteLine("子线程{0}结束：", Thread.CurrentThread.ManagedThreadId);

15 });

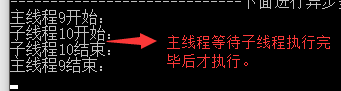
16 await task; //等待子线程执行完毕，方可执行后面的语句

17 Console.WriteLine("主线程{0}结束：", Thread.CurrentThread.ManagedThreadId);

18 }

[复制代码](javascript:void(0);)

**代码结果：**



情况4和情况5：说明要使用子线程中的变量，一定要等子线程执行结束后再使用。

**代码实践：**

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 带返回值的Task，要使用返回值，一定要等子线程计算完毕才行

3 /// </summary>

4 /// <returns></returns>

5 private static async Task<long> Test4()

6 {

7 //主线程执行

8 Console.WriteLine("主线程{0}开始：", Thread.CurrentThread.ManagedThreadId);

9 long result = 0;

10 //启动新线程完成任务

11 Task task = Task.Run(() =>

12 {

13 for (long i = 0; i < 100; i++)

14 {

15 result += i;

16 }

17 });

18 await task; //等待子线程执行完毕，方可执行后面的语句

19 Console.WriteLine("主线程{0}结束：", Thread.CurrentThread.ManagedThreadId);

20 Console.WriteLine("result：{0}", result);

21 return result;

22 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 带返回值的Task，要使用返回值，一定要等子线程计算完毕才行

3 /// 与情况四形成对比，没有等待，最终结果不准确

4 /// </summary>

5 /// <returns></returns>

6 private static Task<long> Test5()

7 {

8 //主线程执行

9 Console.WriteLine("主线程{0}开始：", Thread.CurrentThread.ManagedThreadId);

10 long result = 0;

11 //启动新线程完成任务

12 TaskFactory taskFactory = new TaskFactory();

13 Task<long> task = taskFactory.StartNew<long>(() =>

14 {

15 for (long i = 0; i < 100; i++)

16 {

17 result += i;

18 }

19 return 1;

20 });

21 Console.WriteLine("主线程{0}结束：", Thread.CurrentThread.ManagedThreadId);

22 Console.WriteLine("result：{0}", result);

23 return task;

24 }

[复制代码](javascript:void(0);)

**代码结果：**

**以上两种情况，第一种情况含有线程等待的结果为4950，第二个情况么有线程等待，结果不准确（即共享变量竞用问题）。**

[**第十一节：深究用户模式锁的使用场景(异变结构、互锁、旋转锁)**](https://www.cnblogs.com/yaopengfei/p/8290228.html)

一. 锁机制的背景介绍

　　本章节，将结合多线程来介绍锁机制， 那么问题来了，什么是锁呢？ 为什么需要锁？ 为什么要结合多线程来介绍锁呢？锁的使用场景又是什么呢？ DotNet中又有哪些锁呢？

　　在接下来的几个章节中，将陆续解答这些问题。

**PS：**

**多个线程对一个共享资源进行使用的时候，会出问题, 比如实际的业务场景，入库和出库操作同时进行，库存量就会存在并发问题。所以锁就是用来解决多线程资源竞用的问题。**

**Net领域中，锁机制非常多，比如：时间锁、信号量、互斥锁、读写锁、互锁、异变结构，主要我们可以把他们划分为三大类：**

**①.用户模式锁：就是通过一些cpu指令或者一个死循环,来达到达到线程的等待和休眠。**  
**②.内核模式锁：就是调用win32底层的代码，来实现thread的各种操作。**  
**③.混合锁：用户模式+内核模式**

　　其中用户模式锁又分为这么几类：异变结构、互锁和旋转锁。

二. 异变结构

背景：一个线程读，一个线程写，在release模式下会出现bug，导致主线程无法执行，原因在前面章节已经介绍过了。

　　方式一：利用MemoryBarrier方法进行处理 。（[前面章节已介绍](http://www.cnblogs.com/yaopengfei/p/8135829.html)）

　　方式二：利用VolatileRead/Write方法进行处理。 ([前面章节已介绍](http://www.cnblogs.com/yaopengfei/p/8135829.html))

　　方式三：volatile关键字进行处理，我的read和write都是从memrory中读取，读取的都是最新的。（下面的案例使用volatile关键字后，主线程可以执行）

**代码实践：**

[复制代码](javascript:void(0);)

1 public static volatile bool isStop = false;  
 2 //使用Volatile关键字处理

3 var t = new Thread(() =>

4 {

5 var isSuccess = false;

6 while (!isStop)

7 {

8 isSuccess = !isSuccess;

9 }

10 });

11 t.Start();

12 Thread.Sleep(1000);

13 isStop = true;

14 t.Join();

15 Console.WriteLine("主线程执行结束！");

16 Console.ReadLine();

[复制代码](javascript:void(0);)

**代码结论：使用volatile关键字进行修饰，解决共享资源的竞用问题。**

三. 互锁

　　互锁结构（Interlocked类），常用的方法有：

　　　　\* Increment：自增操作

　　　　\* Decrement：自减操作

　　　　\* Add： 增加指定的值

　　　　\* Exchange： 赋值

　　　　\* CompareExchange： 比较赋值

**代码实践：**

[复制代码](javascript:void(0);)

1 {

2 //1. 自增

3 {

4 int a = 1;

5 Interlocked.Increment(ref a);

6 Console.WriteLine("自增后的数据为:{0}", a);

7 }

8 //2. 自减

9 {

10 int b = 2;

11 Interlocked.Decrement(ref b);

12 Console.WriteLine("自减后的数据为:{0}", b);

13 }

14 //3. 增加操作

15 {

16 int c = 3;

17 Interlocked.Add(ref c, 4);

18 Console.WriteLine("增加后的数据为:{0}", c);

19

20 }

21 //4. 赋值操作

22 {

23 int d = 4;

24 Interlocked.Exchange(ref d, 55);

25 Console.WriteLine("赋值后的数据为:{0}", d);

26

27 }

28 //5. 比较赋值

29 {

30 //Interlocked.CompareExchange(ref num1, sum, num2); // num1==num2 ; num1=sum;

31 int ee = 5;

32 Interlocked.CompareExchange(ref ee, 15, 5);

33 Console.WriteLine("比较赋值后的数据为:{0}", ee);

34

35 Interlocked.CompareExchange(ref ee, 100, 15);

36 Console.WriteLine("比较赋值后的数据为:{0}", ee);

37

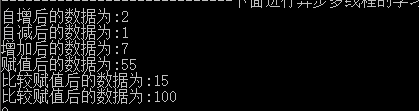
38 }

39

40 }

[复制代码](javascript:void(0);)

**代码结果：**



四. 旋转锁

　　旋转锁(SpinLock)， 特殊的业务逻辑让thread在用户模式下进行自选，欺骗cpu当前thread正在运行中。

 　　SpinLock类有两个核心方法，分别是：Enter和Exit方法。

**代码实践：**

[复制代码](javascript:void(0);)

1 {

2 //下面代码的结果：num从0-249，且是有序的。

3 //如果把旋转锁去掉，num将没有任何顺序

4 for (int i = 0; i < 5; i++)

5 {

6 Task.Factory.StartNew(() =>

7 {

8 for (int j = 0; j < 50; j++)

9 {

10 try

11 {

12 var b = false;

13 sl.Enter(ref b);

14 Console.WriteLine(num++);

15 }

16 catch (Exception ex)

17 {

18 Console.WriteLine(ex.Message);

19 }

20 finally

21 {

22 sl.Exit();

23 }

24 }

25 });

26 }

27 }

[复制代码](javascript:void(0);)

**代码结果：下面代码的结果：num从0-249，且是有序的；如果将旋转锁的代码去掉，num的输出将没有任何顺序可言。**

[**第十二节：深究内核模式锁的使用场景(自动事件锁、手动事件锁、信号量、互斥锁、读写锁、动态锁)**](https://www.cnblogs.com/yaopengfei/p/8303045.html)

一. 整体介绍

**温馨提示：内核模式锁，在不到万不得已的情况下，不要使用它，因为代价太大了，有很多种替代方案。**

**内核模式锁包括：**

**①：事件锁**

**②：信号量**

**③：互斥锁**

**④：读写锁**

**⑤：动态锁**

二. 事件锁

 事件锁包括：

A. 自动事件锁(AutoResetEvent)

　　使用场景：可以用此锁实现多线程环境下某个变量的自增.

　　现实场景： 进站火车闸机，我们用火车票来实现进站操作.

　　true: 表示终止状态,闸机中没有火车票

　　false: 表示费终止状态，闸机中此时有一张火车票

B.手动事件锁(ManualResetEvent)

　　现实场景：有人看守的铁道栅栏(和自动事件锁不一样，不能混用)

　　true： 栅栏没有合围，没有阻止行人通过铁路

　　false：栅栏合围了， 阻止行人通过

\* 下面案例发现，锁不住，自增仍然是无序的输出了.

**\* 核心方法：WaitOne和Set**

**代码实践-自动事件锁：**

[复制代码](javascript:void(0);)

1 static AutoResetEvent autoResetLock1 = new AutoResetEvent(true);

2 static AutoResetEvent autoResetLock2 = new AutoResetEvent(false);

3 static int num2 = 0;

4 {

5 //1. 能输出

6 {

7 autoResetLock1.WaitOne();

8 Console.WriteLine("autoResetLock1检验通过，可以通行");

9 autoResetLock1.Set();

10 }

11

12 //2. 不能输出

13 {

14 autoResetLock2.WaitOne();

15 Console.WriteLine("autoResetLock2检验通过，可以通行");

16 autoResetLock2.Set();

17 }

18

19 //3.下面代码的结果：num从0-249，有序的发现可以锁住。

20 {

21 for (int i = 0; i < 5; i++)

22 {

23 Task.Factory.StartNew(() =>

24 {

25 for (int j = 0; j < 50; j++)

26 {

27 try

28 {

29 autoResetLock1.WaitOne();

30 Console.WriteLine(num2++);

31 autoResetLock1.Set();

32 }

33 catch (Exception ex)

34 {

35 Console.WriteLine(ex.Message);

36 }

37

38 }

39 });

40 }

41 }

42 }

[复制代码](javascript:void(0);)

**代码实践-手动事件锁：**

[复制代码](javascript:void(0);)

1 　　　　　　　　static int num2 = 0;

2 　　　　　　　　static ManualResetEvent mreLock = new ManualResetEvent(true);

3 　　　　　　　　//下面代码锁不住，仍然是无序的输出了

4 {

5 for (int i = 0; i < 5; i++)

6 {

7 Task.Factory.StartNew(() =>

8 {

9 for (int j = 0; j < 50; j++)

10 {

11 try

12 {

13 mreLock.WaitOne();

14 Console.WriteLine(num2++);

15 mreLock.Set();

16 }

17 catch (Exception ex)

18 {

19 Console.WriteLine(ex.Message);

20 }

21

22 }

23 });

24 }

25 }

[复制代码](javascript:void(0);)

三. 信号量

信号量：

　　\* 核心类：Semaphore，通过int数值来控制线程个数。

　　\* 通过观察构造函数 public Semaphore(int initialCount, int maximumCount);：

　　\* initialCount: 可以同时授予的信号量的初始请求数。

　　\* maximumCount: 可以同时授予的信号量的最大请求数。

　　\* **static Semaphore seLock = new Semaphore(1, 1);** //**表示只允许一个线程通过**

\* 下面的案例可以有序的输出。

\* 核心方法：WaitOne和Release

**代码实践：**

[复制代码](javascript:void(0);)

1  static Semaphore seLock = new Semaphore(1, 1); //只允许一个线程通过   
 2 //下面代码锁住了，可以有序的输出

3 {

4 for (int i = 0; i < 5; i++)

5 {

6 Task.Factory.StartNew(() =>

7 {

8 for (int j = 0; j < 50; j++)

9 {

10 try

11 {

12 seLock.WaitOne();

13 Console.WriteLine(num2++);

14 seLock.Release();

15 }

16 catch (Exception ex)

17 {

18 Console.WriteLine(ex.Message);

19 }

20

21 }

22 });

23 }

24 }

[复制代码](javascript:void(0);)

四. 互斥锁

互斥锁：

　　核心方法：WaitOne和ReleaseMutex

　　下面案例可以锁住，有序输出

**总结以上三种类型的锁，都有一个WaitOne方法，观察源码可知，都继承于WaitHandle类。**

**代码实践：**

[复制代码](javascript:void(0);)

1 　　　　　　static Mutex mutex = new Mutex();

2 　　　　 //下面代码锁住了，可以有序的输出

3 {

4 for (int i = 0; i < 5; i++)

5 {

6 Task.Factory.StartNew(() =>

7 {

8 for (int j = 0; j < 50; j++)

9 {

10 try

11 {

12 mutex.WaitOne();

13 Console.WriteLine(num2++);

14 mutex.ReleaseMutex();

15 }

16 catch (Exception ex)

17 {

18 Console.WriteLine(ex.Message);

19 }

20

21 }

22 });

23 }

24 }

[复制代码](javascript:void(0);)

五. 读写锁

  读写锁(ReaderWriterLock)：

　　背景：多个线程读，一个线程写，如果写入的时间太久，此时读的线程会被卡死，这个时候就要用到读写锁了。

　　锁读的两个核心方法：AcquireReaderLock和ReleaseReaderLock。

　　锁写的两个核心方法：AcquireWriterLock和ReleaseWriterLock。

**代码实践：**

[复制代码](javascript:void(0);)

1 static ReaderWriterLock rwlock = new ReaderWriterLock();

2 private void button24\_Click(object sender, EventArgs e)

3 {

4 #region 01-读写锁

5 {

6 //开启5个线程执行读操作

7 for (int i = 0; i < 5; i++)

8 {

9 Task.Run(() =>

10 {

11 Read();

12 });

13 }

14 //开启1个线程执行写操作

15 Task.Factory.StartNew(() =>

16 {

17 Write();

18 });

19 }

20 #endregion

21

22 }

23 /// <summary>

24 /// 线程读

25 /// </summary>

26 static void Read()

27 {

28 while (true)

29 {

30 Thread.Sleep(10);

31 rwlock.AcquireReaderLock(int.MaxValue);

32 Console.WriteLine("当前 t={0} 进行读取 {1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

33 rwlock.ReleaseReaderLock();

34 }

35 }

36 /// <summary>

37 /// 线程写

38 /// </summary>

39 static void Write()

40 {

41 while (true)

42 {

43 Thread.Sleep(300);

44 rwlock.AcquireWriterLock(int.MaxValue);

45 Console.WriteLine("当前 t={0} 进行写入 {1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

46 rwlock.ReleaseWriterLock();

47 }

48 }

[复制代码](javascript:void(0);)

六. 动态锁

动态锁(CountdownEvent)：

　　\* 作用：限制线程数的一个机制。

　　\* 业务场景：有Orders、Products、Users表，我们需要多个线程从某一张表中读取数据。

　　\* 比如：Order表10w，10个线程读取。(每个线程读1w)

　　　　　  Product表5w，5个线程读取。(每个线程读1w)

　　　　     User表2w，2个线程读取。(每个线程读1w)

三个核心方法：

　　①.Reset方法：重置当前的线程数量上限。（初始化的时候，默认设置一个上限）

　　②.Signal方法：将当前的线程数量执行减1操作。（使用一个thread，这个线程数量就会减1操作，直到为0后，继续下一步）

　　③.Wait方法：相当于我们的Task.WaitAll方法。

**代码实践：**

[复制代码](javascript:void(0);)

1 //初始化线程数量上限为10.

2 static CountdownEvent cdLock = new CountdownEvent(10);

3 private void button25\_Click(object sender, EventArgs e)

4 {

5 //加载Orders搞定

6 cdLock.Reset(10);

7 for (int i = 0; i < 10; i++)

8 {

9 Task.Factory.StartNew(() =>

10 {

11 LoadOrder();

12 });

13 }

14 cdLock.Wait();

15 Console.WriteLine("所有的Orders都加载完毕。。。。。。。。。。。。。。。。。。。。。");

16

17 //加载Product搞定

18 cdLock.Reset(5);

19 for (int i = 0; i < 5; i++)

20 {

21 Task.Run(() =>

22 {

23 LoadProduct();

24 });

25 }

26 cdLock.Wait();

27 Console.WriteLine("所有的Products都加载完毕。。。。。。。。。。。。。。。。。。。。。");

28

29 //加载Users搞定

30 cdLock.Reset(2);

31 for (int i = 0; i < 2; i++)

32 {

33 Task.Factory.StartNew(() =>

34 {

35 LoadUser();

36 });

37 }

38 cdLock.Wait();

39 Console.WriteLine("所有的Users都加载完毕。。。。。。。。。。。。。。。。。。。。。");

40

41 Console.WriteLine("所有的表数据都执行结束了。。。恭喜恭喜。。。。");

42 Console.Read();

43 }

44 static void LoadOrder()

45 {

46 //书写具体的业务逻辑

47 Console.WriteLine("当前LoadOrder正在加载中。。。{0}", Thread.CurrentThread.ManagedThreadId);

48 //线程数量减1

49 cdLock.Signal();

50

51 }

52 static void LoadProduct()

53 {

54 //书写具体的业务逻辑

55 Console.WriteLine("当前LoadProduct正在加载中。。。{0}", Thread.CurrentThread.ManagedThreadId);

56 //线程数量减1

57 cdLock.Signal();

58 }

59 static void LoadUser()

60 {

61 //书写具体的业务逻辑

62 Console.WriteLine("当前LoadUser正在加载中。。。{0}", Thread.CurrentThread.ManagedThreadId);

63 //线程数量减1

64 cdLock.Signal();

65 }

[复制代码](javascript:void(0);)

# [第十三节：实际开发中使用最多的监视锁Monitor、lock语法糖的扩展、混合锁的使用(ManualResetEvent、SemaphoreSlim、ReaderWriterLockSlim)](https://www.cnblogs.com/yaopengfei/p/8315212.html)

一. 监视锁(Monitor和lock)

1. Monitor类，限定线程个数的一把锁，两个核心方法：

　　Enter：锁住某个资源。

　　Exit：退出某一个资源。

测试案例：开启5个线程同时对一个变量进行自增操作，结果变量有序的输出，说明该锁同时只允许一个线程访问。

但是写法很麻烦，每次都要try-catch-finally，还要声明bool变量。这个时候lock语法糖就很好的解决了这个问题。

**代码实践：**

[复制代码](javascript:void(0);)

1 static object lockMe = new object();

2 {

3 for (int i = 0; i < 5; i++)

4 {

5 Task.Factory.StartNew(() =>

6 {

7 for (int j = 0; j < 100; j++)

8 {

9 var b = false;

10 try

11 {

12 Monitor.Enter(lockMe, ref b);

13 Console.WriteLine(num++);

14 }

15 catch (Exception)

16 {

17

18 throw;

19 }

20 finally

21 {

22 if (b)

23 {

24 Monitor.Exit(lockMe);

25 }

26 }

27

28 }

29

30 });

31 }

32 }

[复制代码](javascript:void(0);)

2. lock语法糖

　　使用很简单，声明一个静态的object类型变量，调用lock语法糖，将共享变量放入其中，即可保证lock内同时只能一个线程访问。

**代码实践：**

[复制代码](javascript:void(0);)

1 {

2 for (int i = 0; i < 5; i++)

3 {

4 Task.Factory.StartNew(() =>

5 {

6 for (int j = 0; j < 100; j++)

7 {

8 lock (lockMe)

9 {

10 Console.WriteLine(num++);

11 }

12 }

13 });

14 }

15 }

[复制代码](javascript:void(0);)

二. 混合锁

1. 简介：混合锁=用户模式锁+内核模式锁，先在用户模式下内旋，如果超过一定的阈值，会切换到内核锁，在内旋模式下，我们会看到大量的Sleep(0),Sleep(1),Yield等语法。

　　Thread.Sleep(1) 让线程休眠1ms

　　Thread.Sleep(0) 让线程放弃当前的时间片，让本线程更高或者同等线程得到时间片运行。

　　Thread.Yield() 让线程立即放弃当前的时间片，可以让更低级别的线程得到运行，当其他thread时间片用完，本thread再度唤醒。

　混合锁包括以下三种：ManualResetEventSlim、SemaphoreSlim、ReaderWriterLockSlim,这三种混合锁，要比他们对应的内核模式锁 (ManualResetEvent、Semaphore、ReaderWriterLock),的性能高的多。

2. ManualResetEventSlim

　　构造函数默认为false，可以使用Wait方法替代WaitOne方法，支持任务取消. (详细的代码同内核版本类似，这里不做测试了)

3. SemaphoreSlim

　　用法和内核版本类似，使用Wait方法代替WaitOne方法，Release方法不变。(详细的代码同内核版本类似，这里不做测试了)

4. ReaderWriterLockSlim

　　用法和内核版本类似，但是四个核心方法换成了：

　　锁读的两个核心方法：EnterReadLock、ExitReadLock。

　　锁写的两个核心方法：EnterWriteLock、ExitWriteLock。

　　(详细的代码同内核版本类似，这里不做测试了)

[**第十四节: 介绍四大并发集合类并结合单例模式下的队列来说明线程安全和非安全的场景及补充性能调优问题。**](https://www.cnblogs.com/yaopengfei/p/8322016.html)

一. 四大并发集合类

背景：我们目前使用的所有集合都是线程不安全的 。

**A. ConcurrentBag**：就是利用线程槽来分摊Bag中的所有数据，链表的头插法,0代表移除最后一个插入的值.

　　(等价于同步中的List)

**B. ConcurrentStack**：线程安全的Stack是使用Interlocked来实现线程安全, 而没有使用内核锁.

　　(等价于同步中的数组)

**C. ConcurrentQueue**: 队列的模式，先进先出

　　(等价于同步中的队列)

**D. ConcurrentDictionary**: 字典的模式

　　(等价于同步中的字典)

以上四种安全的并发集合类，也可以采用同步版本+Lock锁(或其它锁)来实现

**代码实践：**

[复制代码](javascript:void(0);)

01-ConcurrentBag

{

Console.WriteLine("---------------- 01-ConcurrentBag ---------------------");

ConcurrentBag<int> bag = new ConcurrentBag<int>();

bag.Add(1);

bag.Add(2);

bag.Add(33);

//链表的头插法,0代表移除最后一个插入的值

var result = 0;

//flag为true，表示移除成功，并且返回被移除的值

var flag = bag.TryTake(out result);

Console.WriteLine("移除的值为：{0}", result);

}

#endregion

02-ConcurrentStack

{

Console.WriteLine("---------------- 02-ConcurrentStack ---------------------");

ConcurrentStack<int> stack = new ConcurrentStack<int>();

stack.Push(1);

stack.Push(2);

stack.Push(33);

//链表的头插法,0代表移除最后一个插入的值

var result = 0;

//flag为true，表示移除成功，并且返回被移除的值

var flag = stack.TryPop(out result);

Console.WriteLine("移除的值为：{0}", result);

}

#endregion

03-ConcurrentQueue

{

Console.WriteLine("---------------- 03-ConcurrentQueue ---------------------");

ConcurrentQueue<int> queue = new ConcurrentQueue<int>();

queue.Enqueue(1);

queue.Enqueue(2);

queue.Enqueue(33);

//队列的模式，先进先出，0代表第一个插入的值

var result = 0;

//flag为true，表示移除成功，并且返回被移除的值

var flag = queue.TryDequeue(out result);

Console.WriteLine("移除的值为：{0}", result);

}

#endregion

04-ConcurrentDictionary

{

Console.WriteLine("---------------- 04-ConcurrentDictionary ---------------------");

ConcurrentDictionary<int, int> dic = new ConcurrentDictionary<int, int>();

dic.TryAdd(1, 10);

dic.TryAdd(2, 11);

dic.TryAdd(3, 12);

dic.ContainsKey(3);

//下面是输出字典中的所有值

foreach (var item in dic)

{

Console.WriteLine(item.Key + item.Value);

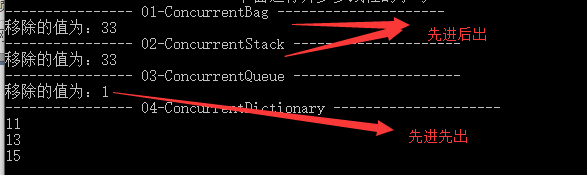
}

}

#endregion

[复制代码](javascript:void(0);)

**代码结果：**



二. 队列的综合案例

 　　上面介绍了四大安全线程集合类和与其对应的不安全的线程集合类，可能你会比较疑惑，到底怎么安全了，那些不安全的集合类怎么能变成安全呢，下面以队列为例，来解决这些疑惑。

　　1. 测试Queue队列并发情况下是不安全的(存在资源竞用的问题)，ConcurrentQueue队列在并发情况下是安全的。

　　2. 利用Lock锁+Queue队列，实现多线程并发情况下的安全问题，即等同于ConcurrentQueue队列的效果。

　　　　经典案例测试：开启100个线程进行入队操作，正常所有的线程执行结束后，队列中的个数应该为100.

**①. Queue不加锁的情况：结果出现99、98、100，显然是出问题了。**

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

{

Queue queue = new Queue();

object o = new object();

int count = 0;

List<Task> taskList = new List<Task>();

for (int i = 0; i < 100; i++)

{

var task = Task.Run(() =>

{

queue.Enqueue(count++);

});

taskList.Add(task);

}

Task.WaitAll(taskList.ToArray());

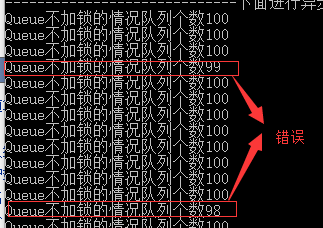
//发现队列个数在不加锁的情况下 竟然不同 有100，有99

Console.WriteLine("Queue不加锁的情况队列个数" + queue.Count);

}

[复制代码](javascript:void(0);)

View Code



**②. Queue加锁的情况：结果全是100，显然是正确的。**

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 {

2 Queue queue = new Queue();

3 object o = new object();

4 int count = 0;

5 List<Task> taskList = new List<Task>();

6 for (int i = 0; i < 100; i++)

7 {

8 var task = Task.Run(() =>

9 {

10 lock (o)

11 {

12 queue.Enqueue(count++);

13 }

14 });

15 taskList.Add(task);

16 }

17

18 Task.WaitAll(taskList.ToArray());

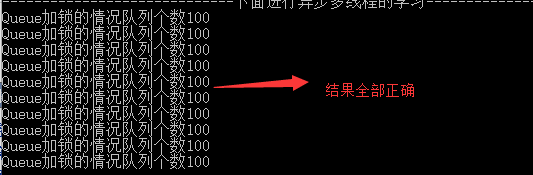
19 //发现队列个数在全是100

20 Console.WriteLine("Queue加锁的情况队列个数" + queue.Count);

21 }

[复制代码](javascript:void(0);)

View Code



**③. ConcurrentQueue不加锁的情况：结果全是100，显然是正确，同时证明ConcurrentQueue队列本身就是线程安全的。**

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 {

2 ConcurrentQueue<int> queue = new ConcurrentQueue<int>();

3 object o = new object();

4 int count = 0;

5 List<Task> taskList = new List<Task>();

6

7 for (int i = 0; i < 100; i++)

8 {

9 var task = Task.Run(() =>

10 {

11 queue.Enqueue(count++);

12 });

13 taskList.Add(task);

14 }

15 Task.WaitAll(taskList.ToArray());

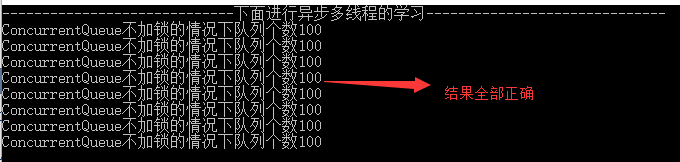
16 //发现队列个数不加锁的情形=也全是100，证明ConcurrentQueue是线程安全的

17 Console.WriteLine("ConcurrentQueue不加锁的情况下队列个数" + queue.Count);

18 }

[复制代码](javascript:void(0);)

View Code



　　3. 在实际项目中，如果使用队列来实现一个业务，该队列需要是全局的，这个时候就需要使用单例(**ps：单例是不允许被实例化的，可以通过单例类中的属性或者方法的形式来获取这个类**)，同时，队列的入队和出队操作，如果使用Queue队列，需要配合lock锁，来解决多线程下资源的竞用问题。

　　经典案例：开启100个线程对其进行入队操作，然后主线程输入队列的个数，并且将队列中的内容输出.

　　结果：队列的个数为100，输出内容1-100依次输出。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 单例类

3 /// </summary>

4 public class QueueUtils

5 {

6 /// <summary>

7 /// 静态变量：由CLR保证，在程序第一次使用该类之前被调用，而且只调用一次

8 /// </summary>

9 private static readonly QueueUtils \_QueueUtils = new QueueUtils();

10

11 /// <summary>

12 /// 声明为private类型的构造函数，禁止外部实例化

13 /// </summary>

14 private QueueUtils()

15 {

16

17 }

18 /// <summary>

19 /// 声明属性，供外部调用，此处也可以声明成方法

20 /// </summary>

21 public static QueueUtils instanse

22 {

23 get

24 {

25 return \_QueueUtils;

26 }

27 }

28

29

30 //下面是队列相关的

31 Queue queue = new Queue();

32

33 private static object o = new object();

34

35 public int getCount()

36 {

37 return queue.Count;

38 }

39

40 /// <summary>

41 /// 入队方法

42 /// </summary>

43 /// <param name="myObject"></param>

44 public void Enqueue(object myObject)

45 {

46 lock (o)

47 {

48 queue.Enqueue(myObject);

49 }

50 }

51 /// <summary>

52 /// 出队操作

53 /// </summary>

54 /// <returns></returns>

55 public object Dequeue()

56 {

57 lock (o)

58 {

59 if (queue.Count > 0)

60 {

61 return queue.Dequeue();

62 }

63 }

64 return null;

65 }

66

67 }

[复制代码](javascript:void(0);)

单例类

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 {

2 int count = 1;

3 List<Task> taskList = new List<Task>();

4 for (int i = 0; i < 100; i++)

5 {

6 var task = Task.Run(() =>

7 {

8 QueueUtils.instanse.Enqueue(count++);

9 });

10 taskList.Add(task);

11 }

12

13 Task.WaitAll(taskList.ToArray());

14 //发现队列个数在全是100

15 Console.WriteLine("单例模式下队列个数" + QueueUtils.instanse.getCount());

16

17 //下面是出队相关的业务

18 while (QueueUtils.instanse.getCount() > 0)

19 {

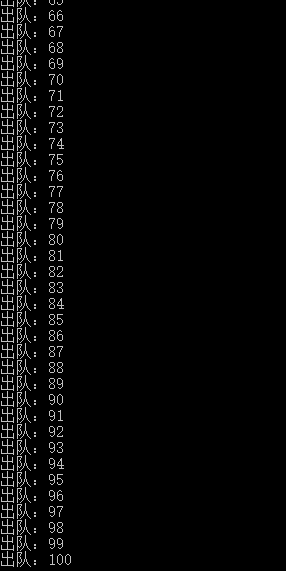
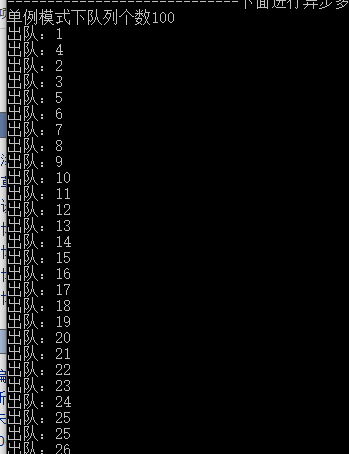
20 Console.WriteLine("出队：" + QueueUtils.instanse.Dequeue());

21 }

22 }

[复制代码](javascript:void(0);)

出入队操作



三. 常见的几类性能调优

PS：

1. 常见的一级事件：CPU占用过高、死锁问题、内存爆满  
　　a. CPU过高：查看是否while(true)中的业务过于复杂，导致cpu一直在高负荷运行。  
　　b. 死锁问题：乱用lock，千万不要lock中再加lock，多个lock重叠  
　　c. 内存爆满：字符串的无限增长，全局的静态变量过多。  
2. 补充几个常用的性能调优的方式  
　　a. 使用字典类型Dictionary<T,T>，代替只有两个属性的对象或匿名对象。  
　　b. 使用数组代替只有两个属性的对象或匿名对象。  
　　比如：  
　　　　index：存放id  
　　　　value：存放数量或其他属性  
3. 返璞归真，使用最原始的代码代替简洁漂亮的代码。  
4. 合理的使用多线程，业务复杂的尽可能的并发执行(或者异步)。  
5. 运用设计模式，使代码简洁、易于扩展。

[**第十五节：深入理解async和await的作用及各种适用场景和用法**](https://www.cnblogs.com/yaopengfei/p/9249390.html)

一. 同步VS异步

**1.   同步 VS 异步 VS 多线程**

**同步方法：调用时需要等待返回结果，才可以继续往下执行业务**

**异步方法：调用时无须等待返回结果，可以继续往下执行业务**

**开启新线程：在主线程之外开启一个新的线程去执行业务**

**同步方法和异步方法的本质区别： 调用时是否需要等待返回结果才能继续执行业务**

**2. 常见的异步方法（都以Async结尾）**

　　① HttpClient类：PostAsync、PutAsync、GetAsync、DeleteAsync

　　② EF中DbContext类：SaveChangesAsync

　　③ 文件相关中的：WriteLineAsync



**3. 引入异步方法的背景**

　　比如我在后台要向另一台服务器中获取中的2个接口获取信息，然后将两个接口的信息拼接起来，一起输出，接口1耗时3s，接口2耗时5s，

**① 传统的同步方式：**

　　需要的时间大约为：3s + 5s =8s, 如下面 **【案例1】**

先分享一个同步请求接口的封装方法，下同。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 public class HttpService

2 {

3 /// <summary>

4 /// 后台跨域请求发送代码

5 /// </summary>

6 /// <param name="url">eg:http://ac.guojin.org/jeesite/regist/saveAppAgentAccount </param>

7 ///<param name="postData"></param>

8 /// 参数格式（手拼Json） string postData = "{\"name\":\"" + vip.comName + "\",\"shortName\":\"" + vip.shortName + + "\"}";

9 /// <returns></returns>

10 public static string PostData(string postData, string url)

11 {

12 HttpWebRequest req = (HttpWebRequest)WebRequest.Create(url);//后台请求页面

13 Encoding encoding = Encoding.GetEncoding("utf-8");//注意页面的编码，否则会出现乱码

14 byte[] requestBytes = encoding.GetBytes(postData);

15 req.Method = "POST";

16 req.ContentType = "application/json";

17 req.ContentLength = requestBytes.Length;

18 Stream requestStream = req.GetRequestStream();

19 requestStream.Write(requestBytes, 0, requestBytes.Length);

20 requestStream.Close();

21 HttpWebResponse res = (HttpWebResponse)req.GetResponse();

22 StreamReader sr = new StreamReader(res.GetResponseStream(), System.Text.Encoding.GetEncoding("utf-8"));

23 string backstr = sr.ReadToEnd();//可以读取到从页面返回的结果，以数据流的形式。

24 sr.Close();

25 res.Close();

26

27 return backstr;

28 }

[复制代码](javascript:void(0);)

View Code

然后在分享服务上的耗时操作，下同。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 耗时方法 耗时3s

3 /// </summary>

4 /// <returns></returns>

5 public ActionResult GetMsg1()

6 {

7 Thread.Sleep(3000);

8 return Content("GetMsg1");

9

10 }

11

12 /// <summary>

13 /// 耗时方法 耗时5s

14 /// </summary>

15 /// <returns></returns>

16 public ActionResult GetMsg2()

17 {

18 Thread.Sleep(5000);

19 return Content("GetMsg2");

20

21 }

[复制代码](javascript:void(0);)

View Code

下面是案例1代码

[复制代码](javascript:void(0);)

1 　　　　　　#region 案例1(传统同步方式 耗时8s左右)

2 {

3 Stopwatch watch = Stopwatch.StartNew();

4 Console.WriteLine("开始执行");

5

6 string t1 = HttpService.PostData("", "http://localhost:2788/Home/GetMsg1");

7 string t2 = HttpService.PostData("", "http://localhost:2788/Home/GetMsg2");

8

9 Console.WriteLine("我是主业务");

10 Console.WriteLine($"{t1},{t2}");

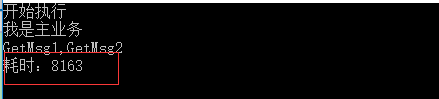
11 watch.Stop();

12 Console.WriteLine($"耗时：{watch.ElapsedMilliseconds}");

13 }

14 #endregion

[复制代码](javascript:void(0);)



**② 开启新线程分别执行两个耗时操作**

　　需要的时间大约为：Max(3s,5s) = 5s ,如下面**【案例2】**

[复制代码](javascript:void(0);)

1 　　　　　　#region 案例2(开启新线程分别执行两个耗时操作 耗时5s左右)

2 {

3 Stopwatch watch = Stopwatch.StartNew();

4 Console.WriteLine("开始执行");

5

6 var task1 = Task.Run(() =>

7 {

8 return HttpService.PostData("", "http://localhost:2788/Home/GetMsg1");

9 });

10

11 var task2 = Task.Run(() =>

12 {

13 return HttpService.PostData("", "http://localhost:2788/Home/GetMsg2");

14 });

15

16 Console.WriteLine("我是主业务");

17 //主线程进行等待

18 Task.WaitAll(task1, task2);

19 Console.WriteLine($"{task1.Result},{task2.Result}");

20 watch.Stop();

21 Console.WriteLine($"耗时：{watch.ElapsedMilliseconds}");

22 }

23 #endregion

[复制代码](javascript:void(0);)



　　既然②方式可以解决同步方法串行耗时间的问题，但这种方式存在一个弊端，一个业务中存在多个线程，且需要对线程进行管理，相对麻烦，从而引出了异步方法。

**这里的异步方法 我 特指：系统类库自带的以async结尾的异步方法。**

**③ 使用系统类库自带的异步方法**

　　需要的时间大约为：Max(3s,5s) = 5s ,如下面**【案例3】**

[复制代码](javascript:void(0);)

1 　　　　　　#region 案例3(使用系统类库自带的异步方法 耗时5s左右)

2 {

3 Stopwatch watch = Stopwatch.StartNew();

4 HttpClient http = new HttpClient();

5 var httpContent = new StringContent("", Encoding.UTF8, "application/json");

6 Console.WriteLine("开始执行");

7 //执行业务

8 var r1 = http.PostAsync("http://localhost:2788/Home/GetMsg1", httpContent);

9 var r2 = http.PostAsync("http://localhost:2788/Home/GetMsg2", httpContent);

10 Console.WriteLine("我是主业务");

11

12 //通过异步方法的结果.Result可以是异步方法执行完的结果

13 Console.WriteLine(r1.Result.Content.ReadAsStringAsync().Result);

14 Console.WriteLine(r2.Result.Content.ReadAsStringAsync().Result);

15

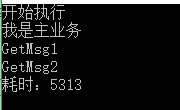
16 watch.Stop();

17 Console.WriteLine($"耗时：{watch.ElapsedMilliseconds}");

18 }

19 #endregion

[复制代码](javascript:void(0);)



**PS：通过 .Result 来获取异步方法执行完后的结果。**

二. 利用async和await封装异步方法

**1. 首先要声明几点：**

　　① async和await关键字是C# 5.0时代引入的，它是一种异步编程模型

　　② 它们本身并不创建新线程，但我可以在自行封装的async中利用Task.Run开启新线程

　　③ 利用async关键字封装的方法中如果写全部都是一些串行业务, 且不用await关键字，那么即使使用async封装，也并没有什么卵用，并起不了异步方法的作用。

　　 需要的时间大约为：3s + 5s =8s, 如下面 **【案例4】**，并且封装的方法编译器会提示：“缺少关键字await，将以同步的方式调用，请使用await运算符等待非阻止API或Task.Run的形式”（PS：非阻止API指系统类库自带的以Async结尾的异步方法）

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 //利用async封装同步业务的方法

2 private static async Task<string> NewMethod5Async()

3 {

4 Thread.Sleep(3000);

5 //其它同步业务

6 return "Msg1";

7 }

8 private static async Task<string> NewMethod6Async()

9 {

10 Thread.Sleep(5000);

11 //其它同步业务

12 return "Msg2";

13 }

[复制代码](javascript:void(0);)

View Code

[复制代码](javascript:void(0);)

1 #region 案例4(async关键字封装的方法中如果写全部都是一些串行业务 耗时8s左右)

2 {

3 Stopwatch watch = Stopwatch.StartNew();

4

5 Console.WriteLine("开始执行");

6

7 Task<string> t1 = NewMethod5Async();

8 Task<string> t2 = NewMethod6Async();

9

10 Console.WriteLine("我是主业务");

11 Console.WriteLine($"{t1.Result},{t2.Result}");

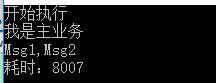
12 watch.Stop();

13 Console.WriteLine($"耗时：{watch.ElapsedMilliseconds}");

14 }

15 #endregion

[复制代码](javascript:void(0);)



**观点结论1：从上面③中可以得出一个结论，async中必须要有await运算符才能起到异步方法的作用，且await 运算符只能加在 系统类库默认提供的异步方法或者新线程（如：Task.Run）前面。**

　　 如：下面**【案例5】** 和 **【案例6】**需要的时间大约为：Max(3s,5s) = 5s

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 // 将系统类库提供的异步方法利用async封装起来

2 private static async Task<String> NewMethod1Async()

3 {

4 HttpClient http = new HttpClient();

5 var httpContent = new StringContent("", Encoding.UTF8, "application/json");

6 //执行业务

7 var r1 = await http.PostAsync("http://localhost:2788/Home/GetMsg1", httpContent);

8 return r1.Content.ReadAsStringAsync().Result;

9 }

10 private static async Task<String> NewMethod2Async()

11 {

12 HttpClient http = new HttpClient();

13 var httpContent = new StringContent("", Encoding.UTF8, "application/json");

14 //执行业务

15 var r1 = await http.PostAsync("http://localhost:2788/Home/GetMsg2", httpContent);

16 return r1.Content.ReadAsStringAsync().Result;

17 }

18

19 //将await关键字加在新线程的前面

20 private static async Task<string> NewMethod3Async()

21 {

22 var msg = await Task.Run(() =>

23 {

24 return HttpService.PostData("", "http://localhost:2788/Home/GetMsg1");

25 });

26 return msg;

27 }

28 private static async Task<string> NewMethod4Async()

29 {

30 var msg = await Task.Run(() =>

31 {

32 return HttpService.PostData("", "http://localhost:2788/Home/GetMsg2");

33 });

34 return msg;

35 }

[复制代码](javascript:void(0);)

View Code

[复制代码](javascript:void(0);)

1 　　　　　　#region 案例5(将系统类库提供的异步方法利用async封装起来 耗时5s左右)

2 //并且先输出“我是主业务”，证明t1和t2是并行执行的，且不阻碍主业务

3 {

4 Stopwatch watch = Stopwatch.StartNew();

5

6 Console.WriteLine("开始执行");

7 Task<string> t1 = NewMethod1Async();

8 Task<string> t2 = NewMethod2Async();

9

10 Console.WriteLine("我是主业务");

11 Console.WriteLine($"{t1.Result},{t2.Result}");

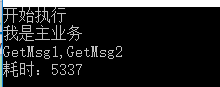
12 watch.Stop();

13 Console.WriteLine($"耗时：{watch.ElapsedMilliseconds}");

14 }

15 #endregion

[复制代码](javascript:void(0);)



[复制代码](javascript:void(0);)

1 　　　　#region 案例6(将新线程利用async封装起来 耗时5s左右)

2 //并且先输出“我是主业务”，证明t1和t2是并行执行的，且不阻碍主业务

3 {

4 Stopwatch watch = Stopwatch.StartNew();

5

6 Console.WriteLine("开始执行");

7 Task<string> t1 = NewMethod3Async();

8 Task<string> t2 = NewMethod4Async();

9

10 Console.WriteLine("我是主业务");

11 Console.WriteLine($"{t1.Result},{t2.Result}");

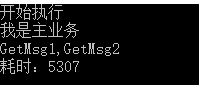
12 watch.Stop();

13 Console.WriteLine($"耗时：{watch.ElapsedMilliseconds}");

14 }

15 #endregion

[复制代码](javascript:void(0);)



**2. 几个规则和约定**

　　① async封装的方法中，可以有多个await，这里的await代表等待该行代码执行完毕。

　　② 我们通常自己封装的方法也要以Async结尾，方便识别

　　③ 异步返回类型主要有三种：Task<T> 、Task、Void

**3. 测试得出其他几个结论**

① 如果async封装的异步方法里既有同步业务又有异步业务(开启新线程或者系统类库提供异步方法)，那么同步方法那部分的时间在调用的时候是会阻塞主线程的，即主线程要等待这部分同步业务执行完才能往下执行。

　　如**【案例7】** 耗时：同步操作之和 2s+2s + Max(3s,5s)=9s;

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 //同步耗时操作和异步方法同时封装

2 private static async Task<String> NewMethod7Async()

3 {

4 //调用异步方法之前还有一个耗时操作

5 Thread.Sleep(2000);

6

7 //下面的操作耗时3s

8 HttpClient http = new HttpClient();

9 var httpContent = new StringContent("", Encoding.UTF8, "application/json");

10 //执行业务

11 var r1 = await http.PostAsync("http://localhost:2788/Home/GetMsg1", httpContent);

12 return r1.Content.ReadAsStringAsync().Result;

13 }

14 private static async Task<String> NewMethod8Async()

15 {

16 //调用异步方法之前还有一个耗时操作

17 Thread.Sleep(2000);

18

19 //下面的操作耗时5s

20 HttpClient http = new HttpClient();

21 var httpContent = new StringContent("", Encoding.UTF8, "application/json");

22 //执行业务

23 var r1 = await http.PostAsync("http://localhost:2788/Home/GetMsg2", httpContent);

24 return r1.Content.ReadAsStringAsync().Result;

25 }

[复制代码](javascript:void(0);)

View Code

[复制代码](javascript:void(0);)

1 　　　　 #region 案例7(既有普通的耗时操作，也有系统本身的异步方法,耗时9s左右)

2 //且大约4s后才能输出 “我是主业务”，证明同步操作Thread.Sleep(2000); 阻塞主线程

3 {

4 Stopwatch watch = Stopwatch.StartNew();

5

6 Console.WriteLine("开始执行");

7 Task<string> t1 = NewMethod7Async();

8 Task<string> t2 = NewMethod8Async();

9

10 Console.WriteLine("我是主业务");

11 Console.WriteLine($"{t1.Result},{t2.Result}");

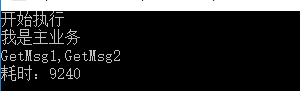
12 watch.Stop();

13 Console.WriteLine($"耗时：{watch.ElapsedMilliseconds}");

14 }

15 #endregion

[复制代码](javascript:void(0);)



**证明：async封装的异步方法里的同步业务的时间会阻塞主线程，再次证明 await只能加在 非阻止api和开启新线程的前面**

② 如果封装的异步方法中存在等待的问题，而且不能阻塞主线程(不能用Thread.Sleep) , 这个时候可以用Task.Delay，并在前面加await关键字

　　如**【案例8】** 耗时：Max（2+3 , 5+2）=7s

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 //利用Task.Delay(2000);等待

2 private static async Task<String> NewMethod11Async()

3 {

4 //调用异步方法之前需要等待2s

5 await Task.Delay(2000);

6

7 //下面的操作耗时3s

8 HttpClient http = new HttpClient();

9 var httpContent = new StringContent("", Encoding.UTF8, "application/json");

10 //执行业务

11 var r1 = await http.PostAsync("http://localhost:2788/Home/GetMsg1", httpContent);

12 return r1.Content.ReadAsStringAsync().Result;

13 }

14

15 private static async Task<String> NewMethod12Async()

16 {

17 //调用异步方法之前需要等待2s

18 await Task.Delay(2000);

19

20 //下面的操作耗时5s

21 HttpClient http = new HttpClient();

22 var httpContent = new StringContent("", Encoding.UTF8, "application/json");

23 //执行业务

24 var r1 = await http.PostAsync("http://localhost:2788/Home/GetMsg2", httpContent);

25 return r1.Content.ReadAsStringAsync().Result;

26 }

[复制代码](javascript:void(0);)

View Code

[复制代码](javascript:void(0);)

1 　　　　　　 #region 案例8(利用Task.Delay执行异步方法的等待操作)

2 //结果是7s，且马上输出“我是主业务”，说明Task.Delay(),不阻塞主线程。

3 {

4 Stopwatch watch = Stopwatch.StartNew();

5 Console.WriteLine("开始执行");

6 Task<string> t1 = NewMethod11Async();

7 Task<string> t2 = NewMethod12Async();

8

9 Console.WriteLine("我是主业务");

10 Console.WriteLine($"{t1.Result},{t2.Result}");

11 watch.Stop();

12 Console.WriteLine($"耗时：{watch.ElapsedMilliseconds}");

13 }

14 #endregion

[复制代码](javascript:void(0);)



三. 异步方法返回类型

1. Task<T>, 处理含有返回值的异步方法，通过 .Result 等待异步方法执行完，且获取到返回值。

2. Task：调用方法不需要从异步方法中取返回值，但是希望检查异步方法的状态，那么可以选择可以返回 Task 类型的对象。不过，就算异步方法中包含 return 语句，也不会返回任何东西。

　　如**【案例9】**

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1

2 //返回值为Task的方法

3 private static async Task NewMethod9Async()

4 {

5

6 //下面的操作耗时3s

7 HttpClient http = new HttpClient();

8 var httpContent = new StringContent("", Encoding.UTF8, "application/json");

9 //执行业务

10 var r1 = await http.PostAsync("http://localhost:2788/Home/GetMsg1", httpContent);

11 Console.WriteLine("NewMethod9Async执行完成");

12 }

[复制代码](javascript:void(0);)

View Code

[复制代码](javascript:void(0);)

1 　　　　　　#region 案例9(返回值为Task的异步方法)

2 //结果是5s，说明异步方法和主线程的同步方法 在并行执行

3 {

4 Stopwatch watch = Stopwatch.StartNew();

5

6 Console.WriteLine("开始执行");

7 Task t = NewMethod9Async();

8

9 Console.WriteLine($"{nameof(t.Status)}: {t.Status}"); //任务状态

10 Console.WriteLine($"{nameof(t.IsCompleted)}: {t.IsCompleted}"); //任务完成状态标识

11 Console.WriteLine($"{nameof(t.IsFaulted)}: {t.IsFaulted}"); //任务是否有未处理的异常标识

12

13 //执行其他耗时操作，与此同时NewMethod9Async也在工作

14 Thread.Sleep(5000);

15

16 Console.WriteLine("我是主业务");

17

18 t.Wait();

19

20 Console.WriteLine($"{nameof(t.Status)}: {t.Status}"); //任务状态

21 Console.WriteLine($"{nameof(t.IsCompleted)}: {t.IsCompleted}"); //任务完成状态标识

22 Console.WriteLine($"{nameof(t.IsFaulted)}: {t.IsFaulted}"); //任务是否有未处理的异常标识

23

24 Console.WriteLine($"所有业务执行完成了");

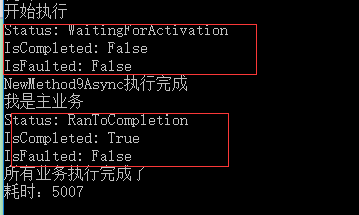
25 watch.Stop();

26 Console.WriteLine($"耗时：{watch.ElapsedMilliseconds}");

27 }

28 #endregion

[复制代码](javascript:void(0);)

****

**PS：对于Task返回值的异步方法，可以调用Wait(),等 待该异步方法执行完，他和await不同，await必须出现在async关键字封装的方法中。**

3. void：调用异步执行方法，不需要做任何交互

　　如**【案例10】**

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 //返回值是Void的方法

2 private static async void NewMethod10Async()

3 {

4 //下面的操作耗时5s

5 HttpClient http = new HttpClient();

6 var httpContent = new StringContent("", Encoding.UTF8, "application/json");

7 //执行业务，假设这里主需要请求，不需要做任何交互

8 var r1 = await http.PostAsync("http://localhost:2788/Home/GetMsg1", httpContent);

9 Console.WriteLine("NewMethod10Async执行完成");

10 }

[复制代码](javascript:void(0);)

View Code

[复制代码](javascript:void(0);)

1 　　　　　　#region 案例10(返回值为Void的异步方法)

2 //结果是5s，说明异步方法和主线程的同步方法 在并行执行

3 {

4 Stopwatch watch = Stopwatch.StartNew();

5

6 Console.WriteLine("开始执行");

7 NewMethod10Async();

8

9 //执行其他耗时操作，与此同时NewMethod9Async也在工作

10 Thread.Sleep(5000);

11

12 Console.WriteLine("我是主业务");

13

14

15 Console.WriteLine($"所有业务执行完成了");

16 watch.Stop();

17 Console.WriteLine($"耗时：{watch.ElapsedMilliseconds}");

18 }

19 #endregion

[复制代码](javascript:void(0);)



四. 几个结论

**1. 异步方法到底开不开起新线程？**

**异步和等待关键字不会导致其他线程创建。 因为异步方法本身并不会运行的线程，异步方法不需要多线程。 只有 + 当方法处于活动状态，则方法在当前同步上下文中运行并使用在线程的时间。 可以使用**[**Task.Run**](https://msdn.microsoft.com/zh-cn/library/system.threading.tasks.task.run(v=vs.110).aspx)**移动 CPU 工作移到后台线程，但是，后台线程不利于等待结果变得可用处理。（来自MSDN原话）**

**2. async和await是一种异步编程模型，它本身并不能开启新线程，多用于将一些非阻止API或者开启新线程的操作封装起来，使其调用的时候像同步方法一样使用。**

**下面补充博客园dudu的解释，方便大家理解。**

****