[《你不常用的c#之一》:略谈unsafe](http://blog.csdn.net/robingaoxb/article/details/6199508)

标签： [c#](http://www.csdn.net/tag/c%23)[byte](http://www.csdn.net/tag/byte)[dst](http://www.csdn.net/tag/dst)[pointers](http://www.csdn.net/tag/pointers)[integer](http://www.csdn.net/tag/integer)[null](http://www.csdn.net/tag/null)

2011-02-22 11:15 3352人阅读 [评论](http://blog.csdn.net/robingaoxb/article/details/6199508#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/robingaoxb/article/details/6199508#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

不常用的C#系列（7） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

msdn里讲到:

“在 C# 中很少需要使用指针，但仍有一些需要使用的情况。例如，在下列情况中使用允许采用指针的不安全上下文是正确的：

* 处理磁盘上的现有结构
* 涉及内部包含指针的结构的高级 COM 或平台调用方案
* 性能关键代码

对于第一和第二点，主要是调win32的api。  
但是“性能关键代码”这个非常重要。我来举例引申一下。

我们都知道像飞信这种大型IM服务端，难免会面临大量的字符处理（协议报文）。  
如果同时在线100多万，而且大家都同时进行会话，服务端的程序如果对内存回收不好，那肯定会crash.  
飞信服务端是.net的，所以就拉扯一下这个例子。  
不过你可以大胆猜测，它服务端肯定用了unsafe，不然顶不住这档子活!!

还是msdn上的demo:

以下示例使用指针将一个字节数组从 src 复制到 dst。用 /unsafe 选项编译此示例。

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6199508) [copy](http://blog.csdn.net/robingaoxb/article/details/6199508)

1. // fastcopy.cs
2. // compile with: /unsafe
3. **using** System;
4. **class** Test
5. {
6. // The unsafe keyword allows pointers to be used within
7. // the following method:
8. **static** **unsafe** **void** Copy(**byte**[] src, **int** srcIndex,
9. **byte**[] dst, **int** dstIndex, **int** count)
10. {
11. **if** (src == **null** || srcIndex < 0 ||
12. dst == **null** || dstIndex < 0 || count < 0)
13. {
14. **throw** **new** ArgumentException();
15. }
16. **int** srcLen = src.Length;
17. **int** dstLen = dst.Length;
18. **if** (srcLen - srcIndex < count ||
19. dstLen - dstIndex < count)
20. {
21. **throw** **new** ArgumentException();
22. }
23. // The following fixed statement pins the location of
24. // the src and dst objects in memory so that they will
25. // not be moved by garbage collection.
26. **fixed** (**byte**\* pSrc = src, pDst = dst)
27. {
28. **byte**\* ps = pSrc;
29. **byte**\* pd = pDst;
30. // Loop over the count in blocks of 4 bytes, copying an
31. // integer (4 bytes) at a time:
32. **for** (**int** n =0 ; n < count/4 ; n++)
33. {
34. \*((**int**\*)pd) = \*((**int**\*)ps);
35. pd += 4;
36. ps += 4;
37. }
38. // Complete the copy by moving any bytes that weren’t
39. // moved in blocks of 4:
40. **for** (**int** n =0; n < count%4; n++)
41. {
42. \*pd = \*ps;
43. pd++;
44. ps++;
45. }
46. }
47. }
48. **static** **void** Main(**string**[] args)
49. {
50. **byte**[] a = **new** **byte**[100];
51. **byte**[] b = **new** **byte**[100];
52. **for**(**int** i=0; i<100; ++i)
53. a[i] = (**byte**)i;
54. Copy(a, 0, b, 0, 100);
55. Console.WriteLine(”The first 10 elements are:”);
56. **for**(**int** i=0; i<10; ++i)
57. Console.Write(b[i] + ” “);
58. Console.WriteLine(”/n”);
59. }
60. }

请注意使用了 unsafe 关键字，这允许在 Copy 方法内使用指针。  
fixed 语句用于声明指向源和目标数组的指针。它锁定 src 和 dst 对象在内存中的位置以便使其不会被垃圾回收移动。当 fixed 块完成后，这些对象将被解除锁定。  
通过略过数组界限检查，不安全代码可提高性能。

fixed 语句允许您获取指向字节数组使用的内存的指针，并且标记实例，以便垃圾回收器不会移动它。

在 fixed 块的末尾，将标记该实例以便可以移动它。此功能称为声明式锁定。锁定的好处是系统开销非常小，除非在 fixed 块中发生垃圾回收（但此情况不太可能发生）。

对头，fixed 内我只分配我自己的内存，用完就释放，从不霸占平民土地，不多征收平民余粮！！  
对于如果你要是等着GC来跟你处理，它寻根寻址还得点时候呢。。。。。

* 上一篇[字节流与结构、类之间的转换](http://blog.csdn.net/robingaoxb/article/details/6186353)
* 下一篇[《你不常用的c#之二》:略谈GCHandle](http://blog.csdn.net/robingaoxb/article/details/6199514)

[《你不常用的c#之二》:略谈GCHandle](http://blog.csdn.net/robingaoxb/article/details/6199514)

标签： [c#](http://www.csdn.net/tag/c%23)[callback](http://www.csdn.net/tag/callback)[interop](http://www.csdn.net/tag/interop)[object](http://www.csdn.net/tag/object)[class](http://www.csdn.net/tag/class)[reference](http://www.csdn.net/tag/reference)

2011-02-22 11:16 3913人阅读 [评论](http://blog.csdn.net/robingaoxb/article/details/6199514#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/robingaoxb/article/details/6199514#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

不常用的C#系列（7） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

我们在使用c#托管代码时，内存地址和GC回收那不是我们关心的，CLR已经给我们暗箱操作。  
但是如果我们在c#中调用了一个非托管代码，比如vc的DLL,而且他有个回调函数，需要引用c#中的某个对象并操作，  
这时候你就得要小心了。  
要是非托管代码中用到得托管代码那个对象被GC给回收了，这时候就会报内存错误。  
所以我们就要把那个对象“钉”住(pin)，让它的内存地址固定，而不被垃圾回收掉，然后最后我们自己管理，自己释放内存,这时候就需要GCHandle,来看个msdn上的例子:

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6199514) [copy](http://blog.csdn.net/robingaoxb/article/details/6199514)

1. **public** **delegate** **bool** CallBack(**int** handle, IntPtr param);
2. **public** **class** LibWrap
3. {
4. [DllImport("user32.dll")]
5. **public** **static** **extern** **bool** EnumWindows(CallBack cb, IntPtr param);
6. }
8. **class** Program
9. {
10. **static** **void** Main(**string**[] args)
11. {
13. TextWriter tw = System.Console.Out;
14. GCHandle gch = GCHandle.Alloc(tw);
15. CallBack cewp = **new** CallBack(CaptureEnumWindowsProc);
16. LibWrap.EnumWindows(cewp, (IntPtr)gch);
17. gch.Free();
18. Console.Read();
20. }
21. **private** **static** **bool** CaptureEnumWindowsProc(**int** handle, IntPtr param)
22. {
23. GCHandle gch = (GCHandle)param;
24. TextWriter tw = (TextWriter)gch.Target;
26. tw.WriteLine(handle);
27. **return** **true**;
28. }
30. }

对上面的代码，略加解释：gch 会钉住(pin)tw这个对象，使其不受GC管理，告诉它，以后你崩管我，我也不用给你上税,其实管理权已经给gch，通过free来释放内存。  
这种情况主要用在托管和非托管代码交互的时候，防止内存泄露来使用GCHandle。

另也可以使用GC.KeepAlive 方法(引用msdn)  
KeepAlive 方法的目的是确保对对象的引用存在，该对象有被垃圾回收器过早回收的危险。这种现象可能发生的一种常见情形是，当在托管代码或数据中已没有对该对象的引用，但该对象仍然在非托管代码（如 Win32 API、非托管 DLL 或使用 COM 的方法）中使用。  
下面是例子:

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6199514) [copy](http://blog.csdn.net/robingaoxb/article/details/6199514)

1. **using** System;
2. **using** System.Threading;
3. **using** System.Runtime.InteropServices;
5. // A simple class that exposes two static Win32 functions.
6. // One is a delegate type and the other is an enumerated type.
7. **public** **class** MyWin32
8. {
9. // Declare the SetConsoleCtrlHandler function
10. // as external and receiving a delegate.
11. [DllImport("Kernel32")]
12. **public** **static** **extern** Boolean SetConsoleCtrlHandler(HandlerRoutine Handler,
13. Boolean Add);
15. // A delegate type to be used as the handler routine
16. // for SetConsoleCtrlHandler.
17. **public** **delegate** Boolean HandlerRoutine(CtrlTypes CtrlType);
19. // An enumerated type for the control messages
20. // sent to the handler routine.
21. **public** **enum** CtrlTypes
22. {
23. CTRL\_C\_EVENT = 0,
24. CTRL\_BREAK\_EVENT,
25. CTRL\_CLOSE\_EVENT,
26. CTRL\_LOGOFF\_EVENT = 5,
27. CTRL\_SHUTDOWN\_EVENT
28. }
29. }
31. **public** **class** MyApp
32. {
33. // A private static handler function in the MyApp class.
34. **static** Boolean Handler(MyWin32.CtrlTypes CtrlType)
35. {
36. String message = "This message should never be seen!";
38. // A switch to handle the event type.
39. **switch** (CtrlType)
40. {
41. **case** MyWin32.CtrlTypes.CTRL\_C\_EVENT:
42. message = "A CTRL\_C\_EVENT was raised by the user.";
43. **break**;
44. **case** MyWin32.CtrlTypes.CTRL\_BREAK\_EVENT:
45. message = "A CTRL\_BREAK\_EVENT was raised by the user.";
46. **break**;
47. **case** MyWin32.CtrlTypes.CTRL\_CLOSE\_EVENT:
48. message = "A CTRL\_CLOSE\_EVENT was raised by the user.";
49. **break**;
50. **case** MyWin32.CtrlTypes.CTRL\_LOGOFF\_EVENT:
51. message = "A CTRL\_LOGOFF\_EVENT was raised by the user.";
52. **break**;
53. **case** MyWin32.CtrlTypes.CTRL\_SHUTDOWN\_EVENT:
54. message = "A CTRL\_SHUTDOWN\_EVENT was raised by the user.";
55. **break**;
56. }
58. // Use interop to display a message for the type of event.
59. Console.WriteLine(message);
61. **return** **true**;
62. }
64. **public** **static** **void** Main()
65. {
67. // Use interop to set a console control handler.
68. MyWin32.HandlerRoutine hr = **new** MyWin32.HandlerRoutine(Handler);
69. MyWin32.SetConsoleCtrlHandler(hr, **true**);
71. // Give the user some time to raise a few events.
72. Console.WriteLine("Waiting 30 seconds for console ctrl events");
74. // The object hr is not referred to again.
75. // The garbage collector can detect that the object has no
76. // more managed references and might clean it up here while
77. // the unmanaged SetConsoleCtrlHandler method is still using it.
79. // Force a garbage collection to demonstrate how the hr
80. // object will be handled.
81. GC.Collect();
82. GC.WaitForPendingFinalizers();
83. GC.Collect();
85. Thread.Sleep(10000);
87. // Display a message to the console when the unmanaged method
88. // has finished its work.
89. Console.WriteLine("Finished!");
91. // Call GC.KeepAlive(hr) at this point to maintain a reference to hr.
92. // This will prevent the garbage collector from collecting the
93. // object during the execution of the SetConsoleCtrlHandler method.
94. GC.KeepAlive(hr);
95. Console.Read();
96. }
97. }

* 上一篇[《你不常用的c#之一》:略谈unsafe](http://blog.csdn.net/robingaoxb/article/details/6199508)
* 下一篇[《你不常用的c#之三》:Action 之怪状](http://blog.csdn.net/robingaoxb/article/details/6199891)

[《你不常用的c#之三》:Action 之怪状](http://blog.csdn.net/robingaoxb/article/details/6199891)

标签： [action](http://www.csdn.net/tag/action)[c#](http://www.csdn.net/tag/c%23)[c](http://www.csdn.net/tag/c)[string](http://www.csdn.net/tag/string)[编译器](http://www.csdn.net/tag/%e7%bc%96%e8%af%91%e5%99%a8)[优化](http://www.csdn.net/tag/%e4%bc%98%e5%8c%96)

2011-02-22 14:05 2431人阅读 [评论](http://blog.csdn.net/robingaoxb/article/details/6199891#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/robingaoxb/article/details/6199891#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

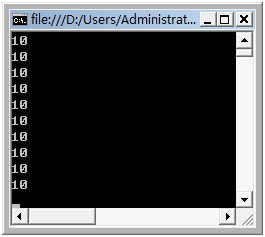
不常用的C#系列（7） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

例1：

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6199891) [copy](http://blog.csdn.net/robingaoxb/article/details/6199891)

1. **static** **void** Main(**string**[] args)
2. {
3. List<Action> ls = **new** List<Action>();
4. **for** (**int** i = 0; i < 10; i++)
5. { ls.Add(() => Console.WriteLine(i));
6. }
7. **foreach** (Action action **in** ls)
8. {
9. action();
10. }
11. System.Console.Read();
12. }

输出结果:

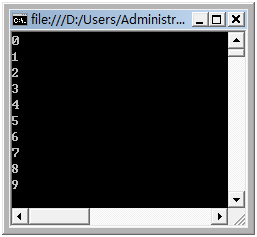


例2:

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6199891) [copy](http://blog.csdn.net/robingaoxb/article/details/6199891)

1. **static** **void** Main(**string**[] args)
2. {
3. List<Action> ls = **new** List<Action>();
4. **for** (**int** i = 0; i < 10; i++)
5. {
6. **int** tp=i;
7. ls.Add(() => Console.WriteLine(tp));
8. }
9. **foreach** (Action action **in** ls)
10. {
11. action();
12. }
13. System.Console.Read();
14. }

则结果为:



为啥呢？我们来用windbg分析下:

对于例1，我们观看堆上的Action对象(System.Action )只有一个，  
heap上的列表:

6c27166c 1 32 System.Action

你的foreach里其实都是循环的一个action  
所以最后的时候是10（i的最后值）

而例2中则有10个  
action对象在heap上的列表：

6c27166c 10 320 System.Action

10个，每个action里引用一个tp,  
heap列表:

Address MT Size  
01451790 6c27166c 32  
014517dc 6c27166c 32  
01451808 6c27166c 32  
01451834 6c27166c 32  
01451860 6c27166c 32  
014518bc 6c27166c 32  
014518e8 6c27166c 32  
**01451914** 6c27166c 32  
01451940 6c27166c 32  
014519bc 6c27166c 32

随便看一个Address,比如这个01451914 ，dump出来的值是

MT Field Offset Type VT Attr Value Name  
79102290 4000001 4 System.Int32 1 instance 7 **tp**

看到没，此时的tp还存在，值为7。

为了验证，我们可以

Action a1 = ls[0] as Action;  
Action a2 = ls[1] as Action;  
Console.WriteLine(object.ReferenceEquals(a1,a2));

第一个例子true,第二个(tp) 则为false。

看来我用windbg有点越俎代庖。

更确切说是VS编译器优化的问题，从MSIL代码可以看出

* 上一篇[《你不常用的c#之二》:略谈GCHandle](http://blog.csdn.net/robingaoxb/article/details/6199514)
* 下一篇[《你不常用的c#之四》:Array的小抽屉ArraySegment](http://blog.csdn.net/robingaoxb/article/details/6200060)

[《你不常用的c#之四》:Array的小抽屉ArraySegment](http://blog.csdn.net/robingaoxb/article/details/6200060)

标签： [c#](http://www.csdn.net/tag/c%23)[buffer](http://www.csdn.net/tag/buffer)[string](http://www.csdn.net/tag/string)[网络](http://www.csdn.net/tag/%e7%bd%91%e7%bb%9c)

2011-02-22 14:58 5480人阅读 [评论](http://blog.csdn.net/robingaoxb/article/details/6200060#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/robingaoxb/article/details/6200060#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

不常用的C#系列（7） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

**一:)略谈**  
ArraySegment顾名思义就是Array区块，数组的小抽屉，用于对该数组中元素的范围进行分隔。比如这个数组：

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6200060) [copy](http://blog.csdn.net/robingaoxb/article/details/6200060)

1. String[] myArr = { "Overred", "Medloy", "Xiaoguai", "Hare" };

我们用ArraySegment这个抽屉给装起来就是:

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6200060) [copy](http://blog.csdn.net/robingaoxb/article/details/6200060)

1. ArraySegment<String> arrSeg = **new** ArraySegment<String>(myArr);

只需for一下就可以得出这个小抽屉里的myArr值（不能foreach,ArraySegment无GetEnumerator方法实现）

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6200060) [copy](http://blog.csdn.net/robingaoxb/article/details/6200060)

1. **for** ( **int** i = arrSeg.Offset; i < (arrSeg.Offset + arrSeg.Count); i++ )  {
2. Console.WriteLine( "   [{0}] : {1}", i, arrSeg.Array[i] );
3. }

比如我们想取某个区域，可以:

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6200060) [copy](http://blog.csdn.net/robingaoxb/article/details/6200060)

1. ArraySegment<String> myArrSegMid = **new** ArraySegment<String>(myArr, 1, 3);

则取myArr的索引从1到3的值，

需要注意的是:  
**多个 ArraySegment 实例可以引用同一个原始数组，从而可能发生重叠。Array 属性返回整个原始数组，而不是该数组的副本；**

所以:

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6200060) [copy](http://blog.csdn.net/robingaoxb/article/details/6200060)

1. arrSeg.Array[2] = "xx";

则会导致myArr里的值也发生变化，此时的myArr值则为:

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6200060) [copy](http://blog.csdn.net/robingaoxb/article/details/6200060)

1. { "Overred", "Medloy", "xx", "Hare" };

**三）：应用**

应用方面：所以ArraySegment可以用在内存碎片（fragmentation）较多的地方，来减少内存碎片的产生，以便使内存更有序，提高我们的程序性能。比如Socket.BeginReceive异步接收频繁的时候，我们并不知道要接受的这个buffer长度，所以就预先设置一个固定值，这时候这块内存就会被pin住，一直等到网络硬件接收到数据完成为止，这往往比较容易产生碎片，如果你用windbg一看，!DumpHeap -type Free -stat会有很多objects但是值都是0.所以可以用ArraySegment+Stack来解决，我会拿出专门的一节来讲解。

上一篇[《你不常用的c#之三》:Action 之怪状](http://blog.csdn.net/robingaoxb/article/details/6199891)

* 下一篇[《你不常用的c#之五》:Thread与ThreadPool的内存之战](http://blog.csdn.net/robingaoxb/article/details/6200495)

[《你不常用的c#之五》:Thread与ThreadPool的内存之战](http://blog.csdn.net/robingaoxb/article/details/6200495)

标签： [thread](http://www.csdn.net/tag/thread)[c#](http://www.csdn.net/tag/c%23)[generation](http://www.csdn.net/tag/generation)[statistics](http://www.csdn.net/tag/statistics)[c](http://www.csdn.net/tag/c)[exception](http://www.csdn.net/tag/exception)

2011-02-22 17:06 1483人阅读 [评论](http://blog.csdn.net/robingaoxb/article/details/6200495#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/robingaoxb/article/details/6200495#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

不常用的C#系列（7） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

Thread与ThreadPool使用的时候在内存里对象是如何分布的呢？   
今天我们就从内存堆的角度分析下两者。   
先上小白鼠代码：

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6200495) [copy](http://blog.csdn.net/robingaoxb/article/details/6200495)

1. **static** **void** Main(**string**[] args)
2. {
3. **for** (**int** i = 0; i < 30; i++)
4. {
5. Thread t = **new** Thread(**new** ThreadStart(ThreadProc));
6. t.Name = "Overred\_" + i;
7. t.Start();
8. }
9. Console.Read();
10. }
11. **static** **void** ThreadProc()
12. {
13. **try**
14. {
15. **for** (**int** i = 0; i < 10; i++)
16. {
17. Console.WriteLine("{0}  Value:{1}",Thread.CurrentThread.Name,i);
18. }
19. }
20. **catch** (Exception ex)
21. {
22. Console.WriteLine(ex.Message);
23. }
24. }

以上代码非常简单，就是循环启动30个线程去执行同一个方法ThreadProc(),然后打印出结果。

**现在提出问题1：当Main里的30个线程都把ThreadProc()方法执行完毕后，这些Threads是自动消亡还是被GC回收，还是变成DeadThread？**

好，拿出我们的看家工具windbg，来debug一把。

首先启动我们的程序，然后打开windbg，然后F6，Attach我们的exe

1,加载mscorwks(.net 2.0或者以上)

0:003> .loadby sos mscorwks

2,查看该程序的线程情况

0:003> !Threads  
\*\*\* ERROR: Symbol file could not be found.  Defaulted to export symbols for   
C:/Windows/Microsoft.NET/Framework/v2.0.50727/mscorwks.dll -   
PDB symbol for mscorwks.dll not loaded  
**ThreadCount: 32**UnstartedThread: 0  
BackgroundThread: 1  
PendingThread: 0  
**DeadThread: 30**Hosted Runtime: no  
                                      PreEmptive   GC Alloc           Lock  
       ID OSID ThreadOBJ    State     GC       Context       Domain   Count APT Exception  
   0    1 25e4 00518858      a020 Enabled  013f878c:013f9fe8 00514818     1 MTA  
   2    2 24b8 00526f20      b220 Enabled  00000000:00000000 00514818     0 MTA (Finalizer)  
XXXX    3    0 00533028      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX    4    0 00536858      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX    5    0 005385c8      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX    6    0 005393d0      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX    7    0 00534fd8      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX    8    0 0053a5c0      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX    9    0 0053b3c8      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX    a    0 0053bfc0      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX    b    0 0053eba8      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX    c    0 00543370      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX    d    0 00543b38      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX    e    0 00544700      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX    f    0 00544ec8      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   10    0 00545690      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   11    0 00545ee0      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   12    0 005466c0      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   13    0 00546a88      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   14    0 00546e50      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   15    0 00547218      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   16    0 005475e0      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   17    0 005479a8      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   18    0 00547d70      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   19    0 00548138      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   1a    0 00548500      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   1b    0 005488c8      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   1c    0 00548c90      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   1d    0 00549058      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   1e    0 00549420      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   1f    0 005497e8      9820 Enabled  00000000:00000000 00514818     0 Ukn  
XXXX   20    0 00549bb0      9820 Enabled  00000000:00000000 00514818     0 Ukn

看红色加粗部分，我们总共有32个线程，而DeadThread为30个（其他2个为程序自身所有，其中一个BackgroundThread），先告诉你这30个死线程正式我们循环创建的线程，可以回答我提的第一个问题拉，没错，他们统统死拉，而且不会醒来，还占地方（不是永远占地方，待会我们用GC手动让它们消亡）。

3，然后我们继续看看内存堆上它们这些坏家伙如何分布：

0:003> !DumpHeap -type System.Threading -stat  
total 155 objects  
Statistics:  
      MT    Count    TotalSize Class Name  
79108930        1           32 System.Threading.ContextCallback  
790fe284        2          144 System.Threading.ThreadAbortException  
79124b74       30          600 System.Threading.ThreadHelper  
**79104de8       31         1116 System.Threading.ExecutionContext  
790fe704       31         1736 System.Threading.Thread**791249e8       60         1920 System.Threading.ThreadStart  
Total 155 objects

红色部分，31个Thread，对应着31个Context，每个线程在windows底层都是一个内核对象和一个栈空间，内核对象存放一些线程的统计信息，比如计数器以及一个上下文，就是我上次执行到那里等。而栈空间则是用来存放线程参数等。

4，我们来具体看下这些Thread们的MethodTable

0:003> !DumpHeap -MT 790fe704   
 Address       MT     Size  
013c1708 790fe704       56       
013c178c 790fe704       56       
013c235c 790fe704       56       
013c2474 790fe704       56       
013c258c 790fe704       56       
013c26a4 790fe704       56       
013c27bc 790fe704       56       
013c28d4 790fe704       56       
013c29ec 790fe704       56       
013c2b04 790fe704       56       
013c2c1c 790fe704       56       
013c2d34 790fe704       56       
013c2e54 790fe704       56       
013c2f74 790fe704       56       
013c3094 790fe704       56       
013c31b4 790fe704       56       
013c32d4 790fe704       56       
013c33f4 790fe704       56       
013c3514 790fe704       56       
013c3634 790fe704       56       
013c3754 790fe704       56       
013c3874 790fe704       56       
013c3994 790fe704       56       
013c3ab4 790fe704       56       
**013c3bd4**790fe704       56       
013c3cf4 790fe704       56       
013c3e14 790fe704       56       
013c3f34 790fe704       56       
013f8084 790fe704       56       
013f81a4 790fe704       56       
013f82c4 790fe704       56       
total 31 objects  
Statistics:  
      MT    Count    TotalSize Class Name  
790fe704       31         1736 System.Threading.Thread  
Total 31 objects

5，随便拿一个线程的Address来看看到底是谁占着我们的Thread而不让我们的GC回收掉

0:003> !GCRoot 013c3bd4

Note: Roots found on stacks may be false positives. Run "!help gcroot" for

more info.

Scan Thread 0 OSTHread 25e4

Scan Thread 2 OSTHread 24b8

DOMAIN(00514818):HANDLE(WeakSh):241298:**Root:013c3bd4(System.Threading.Thread)**

结果另我们很失望，他自己就是根，并没被其他任何对象所引用，什么情况下会出现此情况呢？我们先来看看对象在内存中分布的几种方式,我们只需在windbg里执行如下命令则知：

0:003> !Help gcroot  
-------------------------------------------------------------------------------  
!GCRoot [-nostacks] <Object address>  
!GCRoot looks for references (or roots) to an object. These can exist in four  
places:  
   1. On the stack  
   2. Within a GC Handle  
   3. In an object ready for finalization  
   4. As a member of an object found in 1, 2 or 3 above.  
First, all stacks will be searched for roots, then handle tables, and finally  
the freachable queue of the finalizer. Some caution about the stack roots:   
!GCRoot doesn't attempt to determine if a stack root it encountered is valid   
or is old (discarded) data. You would have to use !CLRStack and !U to   
disassemble the frame that the local or argument value belongs to in order to   
determine if it is still in use.  
Because people often want to restrict the search to gc handles and freachable  
objects, there is a -nostacks option.

windbg已经很清楚的告诉我们，   
一个对象可以   
1，在栈上   
2，在一个GCHandle里（可以执行!GCHandles命令查看）   
3，**在FinalizeQueue里**   
4，是一个对象的成员   
难道对象就必定在以上的“四行”之中吗？答案是不一定，还有个Gchandleleaks，就是你在内存里看不到这个Handle，它已经leak。（这种也算在GCHandle里吧）。   
  
回头我们接着说他自己没被其他任何对象所引用，自己就是个根，但是GC却不搭理它，为何？那就是他在GCHandle里，

0:003> !GCHandles  
GC Handle Statistics:  
Strong Handles: 14  
Pinned Handles: 4  
Async Pinned Handles: 0  
Ref Count Handles: 0  
Weak Long Handles: 0  
Weak Short Handles: 31  
Other Handles: 0  
Statistics:  
      MT    Count    TotalSize Class Name  
790fd0f0        1           12 System.Object  
790fcc48        1           24 System.Reflection.Assembly  
790feba4        1           28 System.SharedStatics  
790fe17c        1           72 System.ExecutionEngineException  
790fe0e0        1           72 System.StackOverflowException  
790fe044        1           72 System.OutOfMemoryException  
790fed00        1          100 System.AppDomain  
79100a18        4          144 System.Security.PermissionSet  
790fe284        2          144 System.Threading.ThreadAbortException  
**790fe704       32         1792 System.Threading.Thread**  
7912d8f8        4         8736 System.Object[]  
Total 49 objects

而且在FinalizeQueue里也有它的踪影:

0:003> !FinalizeQueue  
SyncBlocks to be cleaned up: 0  
MTA Interfaces to be released: 0  
STA Interfaces to be released: 0  
----------------------------------  
generation 0 has 35 finalizable objects (00526658->005266e4)  
generation 1 has 0 finalizable objects (00526658->00526658)  
generation 2 has 0 finalizable objects (00526658->00526658)  
Ready for finalization 0 objects (005266e4->005266e4)  
Statistics:  
      MT    Count    TotalSize Class Name  
791037c0        1           20 Microsoft.Win32.SafeHandles.SafeFileMappingHandle  
79103764        1           20 Microsoft.Win32.SafeHandles.SafeViewOfFileHandle  
79101444        2           40 Microsoft.Win32.SafeHandles.SafeFileHandle  
**790fe704       31         1736 System.Threading.Thread**  
Total 35 objects

下面就来解释下什么才可以在FinalizeQueue里出现呢？答案就是有身份的人，很有身份的人，享受特殊待遇的哦！   
啥身份，就是自身实现拉析构函数。

啥待遇，就是GC两次才有可能把他们部分清理掉！为啥部分，是我们不知道windows到底何时去把所有的清理掉（赖皮阿）   
具体原理大家可以看.net框架去，我这里不多说。

**说到此，也就找到我们当初30个彪形大汉为啥赖着不走的原因拉，是在0代的第一次GC时候，他们被放进FinalizeQueue,等着第二次GC他们部分才会从内存堆上消亡。   
为证明我们的观点，我们可以修改程序为 ：**

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6200495) [copy](http://blog.csdn.net/robingaoxb/article/details/6200495)

1. **static void Main(string[] args)**
2. **{**
3. **for (int i = 0; i < 30; i++)**
4. **{**
5. **Thread t = new Thread(new ThreadStart(ThreadProc));**
6. **t.Name = "Overred\_" + i;**
7. **t.Start();**
8. **}**
9. **GC.Collect();**
10. **GC.Collect();**
11. **Console.Read();**
12. **}**

**首先声明一点就是当我们调用一次GC.Collect();时，并不是执行一次垃圾收集，只是告诉系统我要强制进行垃圾收集，系统听到这个命令后乖不乖那就不一定拉。   
当我们用Reflector查看mscorlib对Thread实现的使用也会发现他实现拉析构：**

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6200495) [copy](http://blog.csdn.net/robingaoxb/article/details/6200495)

1. **~Thread()**
2. **{**
3. **this.InternalFinalize();**
4. **}**

**来个虎头蛇尾吧，当我们把小白鼠程序使用ThreadPool修改为：**

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6200495) [copy](http://blog.csdn.net/robingaoxb/article/details/6200495)

1. **static void Main(string[] args)**
2. **{**
3. **for (int i = 0; i < 30; i++)**
4. **{**
5. **ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadProc));**
6. **}**
7. **Console.Read();**
8. **}**
9. **static  void ThreadProc(object o)**
10. **{**
11. **try**
12. **{**
13. **for (int i = 0; i < 10; i++)**
14. **{**
15. **Console.WriteLine(" Value:{0}",i);**
16. **}**
18. **}**
19. **catch (Exception ex)**
20. **{**
21. **Console.WriteLine(ex.Message);**
22. **}**
23. **}**

**再用windbg查看线程时则为:**

0:006> !Threads  
\*\*\* ERROR: Symbol file could not be found.  Defaulted to export symbols   
for C:/Windows/Microsoft.NET/Framework/v2.0.50727/mscorwks.dll -   
PDB symbol for mscorwks.dll not loaded  
ThreadCount: 4  
UnstartedThread: 0  
BackgroundThread: 3  
PendingThread: 0  
DeadThread: 0

而FinalizeQueue则为:

0:006> !FinalizeQueue  
SyncBlocks to be cleaned up: 0  
MTA Interfaces to be released: 0  
STA Interfaces to be released: 0  
----------------------------------  
generation 0 has 7 finalizable objects (00266658->00266674)  
generation 1 has 0 finalizable objects (00266658->00266658)  
generation 2 has 0 finalizable objects (00266658->00266658)  
Ready for finalization 0 objects (00266674->00266674)  
Statistics:  
      MT    Count    TotalSize Class Name  
791037c0        1           20 Microsoft.Win32.SafeHandles.SafeFileMappingHandle  
79103764        1           20 Microsoft.Win32.SafeHandles.SafeViewOfFileHandle  
79101444        2           40 Microsoft.Win32.SafeHandles.SafeFileHandle  
790fe704        3          168 System.Threading.Thread  
Total 7 objects

那现在又出现问题拉，既然ThreadPool这么好，那我们为啥还使用Thread呢？这个问题就是ThreadPool有个GetMaxThreads，可以通过GetMaxThreads(out int workerThreads, out int completionPortThreads);方法获取到，如果线程池满拉，则会死锁更严重！  
另：ThreadPool都为后台线程。  
究竟使用那个，根据情况而定，理解拉内在的东西，一切表象就简单拉。  
OK，到此吧。。。

希望本文能对你有所帮助，谢谢！

* 上一篇[《你不常用的c#之四》:Array的小抽屉ArraySegment](http://blog.csdn.net/robingaoxb/article/details/6200060)
* 下一篇[volatile修饰符](http://blog.csdn.net/robingaoxb/article/details/6284137)

[volatile修饰符](http://blog.csdn.net/robingaoxb/article/details/6284137)

标签： [多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[java](http://www.csdn.net/tag/java)[优化](http://www.csdn.net/tag/%e4%bc%98%e5%8c%96)[编译器](http://www.csdn.net/tag/%e7%bc%96%e8%af%91%e5%99%a8)[虚拟机](http://www.csdn.net/tag/%e8%99%9a%e6%8b%9f%e6%9c%ba)[float](http://www.csdn.net/tag/float)

2011-03-28 17:35 737人阅读 [评论](http://blog.csdn.net/robingaoxb/article/details/6284137#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/robingaoxb/article/details/6284137#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

（10） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

volatile 关键字指示一个字段可以由多个同时执行的线程修改。声明为 volatile 的字段不受编译器优化（假定由单个线程访问）的限制。这样可以确保该字段在任何时间呈现的都是最新的值。

volatile 修饰符通常用于由多个线程访问但不使用 [lock](http://msdn.microsoft.com/zh-cn/library/c5kehkcz(v=VS.90).aspx) 语句对访问进行序列化的字段。有关在多线程方案中使用 volatile 的示例，请参见[如何：创建和终止线程（C# 编程指南）](http://msdn.microsoft.com/zh-cn/library/7a2f3ay4(v=VS.90).aspx)。

volatile 关键字可应用于以下类型的字段：

* 引用类型。
* 指针类型（在不安全的上下文中）。请注意，虽然指针本身可以是可变的，但是它指向的对象不能是可变的。换句话说，您无法声明“指向可变对象的指针”。
* 整型，如 sbyte、byte、short、ushort、int、uint、char、float 和 bool。
* 具有整数基类型的枚举类型。
* 已知为引用类型的泛型类型参数。
* [IntPtr](http://msdn.microsoft.com/zh-cn/library/system.intptr(v=VS.90).aspx) 和 [UIntPtr](http://msdn.microsoft.com/zh-cn/library/system.uintptr(v=VS.90).aspx)。

可变关键字仅可应用于类或结构字段。不能将局部变量声明为 volatile。

**[c-sharp]** [view plain](http://blog.csdn.net/robingaoxb/article/details/6284137) [copy](http://blog.csdn.net/robingaoxb/article/details/6284137)

1. **class** VolatileTest
2. {
3. **public** volatile **int** i;
4. **public** **void** Test(**int** \_i)
5. {
6. i = \_i;
7. }
8. }

恐怕比较一下volatile和synchronized的不同是最容易解释清楚的。volatile是变量修饰符，而synchronized则作用于一段代码或方法；看如下三句get代码：

1. int i1;              int geti1() {return i1;}
2. volatile int i2;  int geti2() {return i2;}
3. int i3;              synchronized int geti3() {return i3;}

　　geti1()得到存储在当前线程中i1的数值。多个线程有多个i1变量拷贝，而且这些i1之间可以互不相同。换句话说，另一个线程可能已经改变了它线程内的i1值，而这个值可以和当前线程中的i1值不相同。事实上，[**Java**](http://lib.csdn.net/base/javaee)有个思想叫“主”内存区域，这里存放了变量目前的“准确值”。每个线程可以有它自己的变量拷贝，而这个变量拷贝值可以和“主”内存区域里存放的不同。因此实际上存在一种可能：“主”内存区域里的i1值是1，线程1里的i1值是2，线程2里的i1值是3——这在线程1和线程2都改变了它们各自的i1值，而且这个改变还没来得及传递给“主”内存区域或其他线程时就会发生。  
　　而geti2()得到的是“主”内存区域的i2数值。用volatile修饰后的变量不允许有不同于“主”内存区域的变量拷贝。换句话说，一个变量经volatile修饰后在所有线程中必须是同步的；任何线程中改变了它的值，所有其他线程立即获取到了相同的值。理所当然的，volatile修饰的变量存取时比一般变量消耗的资源要多一点，因为线程有它自己的变量拷贝更为高效。  
　　既然volatile关键字已经实现了线程间数据同步，又要synchronized干什么呢？呵呵，它们之间有两点不同。首先，synchronized获得并释放监视器——如果两个线程使用了同一个对象锁，监视器能强制保证代码块同时只被一个线程所执行——这是众所周知的事实。但是，synchronized也同步内存：事实上，synchronized在“主”内存区域同步整个线程的内存。因此，执行geti3()方法做了如下几步：  
1. 线程请求获得监视this对象的对象锁（假设未被锁，否则线程等待直到锁释放）  
2. 线程内存的数据被消除，从“主”内存区域中读入（Java虚拟机能优化此步。。。[后面的不知道怎么表达,汗]）  
3. 代码块被执行  
4. 对于变量的任何改变现在可以安全地写到“主”内存区域中（不过geti3()方法不会改变变量值）  
5. 线程释放监视this对象的对象锁  
　　因此volatile只是在线程内存和“主”内存间同步某个变量的值，而synchronized通过锁定和解锁某个监视器同步所有变量的值。显然synchronized要比volatile消耗更多资源。

# [改善C#程序的建议1：非用ICloneable不可的理由](http://www.cnblogs.com/luminji/archive/2011/02/02/1948826.html)

好吧，我承认，这是一个反标题，实际的情况是：我找不到一个非用ICloneable不可的理由。事实上，接口ICloneable还会带来误解，因为它只有一个Clone方法。

我们都知道，对象的拷贝分为：浅拷贝和深拷贝。ICloneable仅有一个Clone方法使我们无法从命名的角度去区分到底是哪个拷贝。

浅拷贝：将对象的字段复制到副本（新的对象）中，同时将字段的值也赋值过去，但是引用类型字段只复制引用，而不是引用类型本身。这意味着，源对象引用类型字段的值改变了，会影响到副本中对应的值也改变；

深拷贝：将对象的字段复制到副本（新的对象）中，无论是值类型还是引用类型字段，都会复制类型本身及类型的值。这意味着，源对象引用类型字段的值改变了，不会影响到副本中对应的值；

于是问题来了，如果类型继承了ICloneable接口，那么类型中的Clone是浅拷贝还是深拷贝。微软的解释是：你既可以在Clone方法中实现浅拷贝，也可以实现深拷贝。那么，为什么不直接提供两个方法呢？比如：DeepClone或者ShallowClone。还是，一般类型的创建，只要实现了浅拷贝就不需要再实现深拷贝（或者反之），所以我们没有必要提供两个方法。

下面是一个既实现了浅拷贝也实现深拷贝的例子：

[复制代码](javascript:void(0);)

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif代码

[Serializable]  
class Employee : ICloneable  
{  
publicstring IDCode { get; set; }  
publicint Age { get; set; }  
public Department Department { get; set; }  
  
#region ICloneable 成员  
  
publicobject Clone()  
{  
returnthis.MemberwiseClone();  
}  
  
#endregion  
  
public Employee DeepClone()  
{  
using (Stream objectStream =new MemoryStream())  
{  
IFormatter formatter =new BinaryFormatter();  
formatter.Serialize(objectStream, this);  
objectStream.Seek(0, SeekOrigin.Begin);  
return formatter.Deserialize(objectStream) as Employee;  
}  
}  
  
public Employee ShallowClone()  
{  
return Clone() as Employee;  
}  
}

[复制代码](javascript:void(0);)

实际上，ICloneable还带来一个问题（该问题Bill Wagner在Effcitive c#中曾经论述过），那就是：如果类型继承自ICloneable，但是同时它不是一个Sealed类型的话，它们的子类的默认Clone方法会带来BUG（子类的Clone方法会返回父类的副本，而不是子类本身）。这会逼迫所有的子类都重写Clone方法；

ICloneable的Clone方法的另一个问题是：它不是类型安全的，它返回的是Object，使用它的时候还设计到转型的问题，而我们自己实现的Clone方法却可以规避掉这个问题（如上文代码）。

综上所述，类型确实没必要继承ICloneable接口，如果类型本身需要实现拷贝功能，直接公开方法就行。如果在应用中你觉得确实必须实现这个接口的，来指正我吧。

# [改善C#程序的建议2：C#中dynamic的正确用法](http://www.cnblogs.com/luminji/archive/2011/02/18/1957484.html)

dynamic是FrameWork4.0的新特性。dynamic的出现让C#具有了弱语言类型的特性。编译器在编译的时候不再对类型进行检查，编译期默认dynamic对象支持你想要的任何特性。比如，即使你对GetDynamicObject方法返回的对象一无所知，你也可以像如下那样进行代码的调用，编译器不会报错：

dynamic dynamicObject = GetDynamicObject();  
 Console.WriteLine(dynamicObject.Name);  
 Console.WriteLine(dynamicObject.SampleMethod());

说到正确用法，那么首先应该指出一个错误用法：

常有人会拿var这个关键字来和dynamic做比较。实际上，var和dynamic完全是两个概念，根本不应该放在一起做比较。var实际上是编译期抛给我们的“语法糖”，一旦被编译，编译期会自动匹配var 变量的实际类型，并用实际类型来替换该变量的申明，这看上去就好像我们在编码的时候是用实际类型进行申明的。而dynamic被编译后，实际是一个object类型，只不过编译器会对dynamic类型进行特殊处理，让它在编译期间不进行任何的类型检查，而是将类型检查放到了运行期。

这从visual studio的编辑器窗口就能看出来。以var声明的变量，支持“智能感知”，因为visual studion能推断出var类型的实际类型，而以dynamic声明的变量却不支持“智能感知”，因为编译器对其运行期的类型一无所知。对dynamic变量使用“智能感知”，会提示“此操作将在运行时解析”。

关于dynamic变量是一个object变量这一点，可以通过IL代码得到验证，这里不再贴出IL代码。当然，编译器也对dynamic声明进行了处理，以区别直接object变量。

dynamic是做为简化互操作性而被MSDN中大肆渲染，我感觉正是基于这一点，才被部分开发人员误解：因为很多开发人员不会接触COM+、OFFICE二次开发之类的编码，所以急需要一个dynamic的应用理由。那么，在日常开发中，我认为dynamic很有价值的一点是：

**dynamic可以简化反射**。

以前我们这样使用反射：

[复制代码](javascript:void(0);)

public class DynamicSample  
 {  
 public string Name { get; set; }  
  
 public int Add(int a, int b)  
 {  
 return a + b;  
 }  
 }  
 DynamicSample dynamicSample = new DynamicSample(); //create instance为了简化演示，我没有使用反射  
 var addMethod = typeof(DynamicSample).GetMethod("Add");  
 int re = (int)addMethod.Invoke(dynamicSample, new object[] { 1, 2 });

[复制代码](javascript:void(0);)

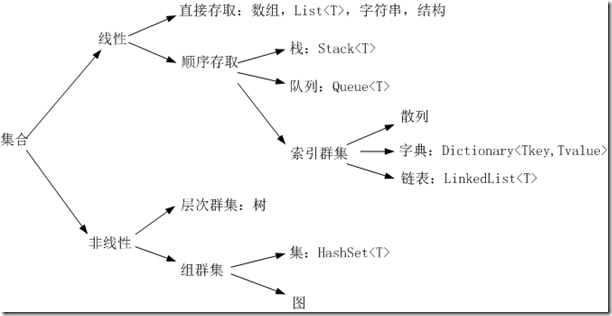
现在，我们有了简化的写法：

dynamic dynamicSample2 = new DynamicSample();  
 int re2 = dynamicSample2.Add(1, 2);

我们可能会对这样的简化不以为然，毕竟看起来代码并没有减少多少，但是，如果考虑到效率兼优美两个特性，那么dynamic的优势就显现出来了。编译器对dynamic进行了优化，比没有经过缓存的反射效率快了很多。如果非要比较，可以将上面两者的代码（调用Add方法部分）运行1000000就可以得出结论。

# [改善C#程序的建议3：在C#中选择正确的集合进行编码](http://www.cnblogs.com/luminji/archive/2011/03/24/1993393.html)

要选择正确的集合，我们首先要了解一些数据结构的知识。所谓数据结构，就是相互之间存在一种或多种特定关系的数据元素的集合。结合下图，我们看一下对集合的分类。

[](http://images.cnblogs.com/cnblogs_com/luminji/201103/201103240912184442.png)

集合分类

在上图中，可以看到，集合总体上分为线性集合和非线性集合。线性集合指元素具有唯一的前驱和后驱的数据结构类型。非线性集合是指具有多个前驱或后驱的数据结构类型，如：树、图。在FCL中，非线性集合实现的比较少，所以我们将会更多的讨论线性集合。

[clip_image004](http://images.cnblogs.com/cnblogs_com/luminji/201103/201103240912197723.gif) 注意：由于类型安全、转型效率等方面的原因，本建议将只讨论泛型集合。

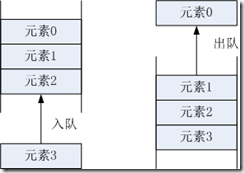
线性集合按存储方式，又分为直接存储和顺序存储。所谓直接存储是指：该类型的集合数据元素可以直接通过下标（也即index）来访问，在C#中有三种形式：Array（包括数组和List<T>），string，struct。直接存储结构的优点是：向数据结构中添加元素是很高效的，只要直接放在数据末尾的第一个空位上就可以了。它的缺点是：向集合插入元素将会变得低效，它需要给插入的元素腾出位置并顺序移动后面的元素。

string和structs虽然是直接存储结构，但它们与一般的集合定义有很大的不同，所以也不在本建议讨论之中。在直接存储的数据结构中，需要区分的是数组和List<T>的选择。再次强调一下：如果集合的数目固定并且不涉及到转型，使用数组效率高，否则就使用List<T>。

顺序存储结构，也即线性表。线性表的大小可动态的扩大和缩小，它在一片连续的区域中存储数据元素。线性表不能按照索引进行查找，它通过对地址的引用来搜索元素，为了找到某个元素，它必须遍历所有元素，直到找到对应的元素为止。所以线性表的优点是插入和删除数据效率高，而缺点是查找的效率相对来说低一些。

线性表又可以分为队列、栈以及索引群集，在C#中，分别表现为：Queue<T>，Stack<T>，索引群集又进一步泛化为字典类型Dictionary< TKey, TValue >和双向链表LinkedList<T>。

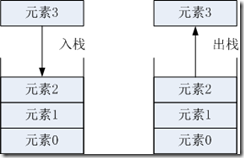
队列Queue<T>遵循的是先入先出模式，它在集合末尾添加元素，在集合起始删除元素，如图：

[](http://images.cnblogs.com/cnblogs_com/luminji/201103/201103240912206262.png)

队列操作

根据队列的特点，可以用来处理并发命令等场景：将所有客户端的命令先入队，由专门的工作线程来执行队列的命令。在分布式中的消息队列就是一个典型的队列应用实例。

栈Stack<T>遵循的是后入先出模式，它在集合末尾添加元素，同时也在集合末尾删除元素，如图2-3：

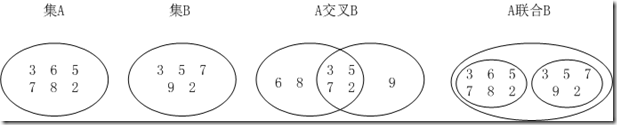
[](http://images.cnblogs.com/cnblogs_com/luminji/201103/201103240912208181.png)

栈操作

字典Dictionary<TKey, TValue>存储的是键值对，值在基于键的散列码的基础上进行存储。字典类对象由包含集合元素的存储桶组成，每一存储桶与基于该元素的键的哈希值关联。如果需要根据键进行值的查找，使用Dictionary<TKey, TValue>将会使搜索和检索更会快捷。

双向链表LinkedList<T>是一个类型为LinkedListNode的元素对象的集合。当我们在集合中觉得插入和删除数据很慢的时候，我们可以考虑使用链表。如果我们使用LinkedList<T>，我们会发现此类型并没有其它集合普遍具有的Add方法，取而代之的是AddAfter、AddBefore、AddFirst、AddLast等方法。双向链表中的每个节点都向前指向Previous节点，向后指向Next节点。

以上讨论了线性集合，在FCL中，非线性集合实现的不多。非线性集合分为层次集合和组集合。层次集合，如树，在FCL中就没有实现。组集合，又分为集和图。集在FCL中实现为HashSet<T>，而图在FCL中也没有对应实现。集的概念在本意上是指存放在集合中的元素是无序的且不能重复的。下图演示了集的用途：

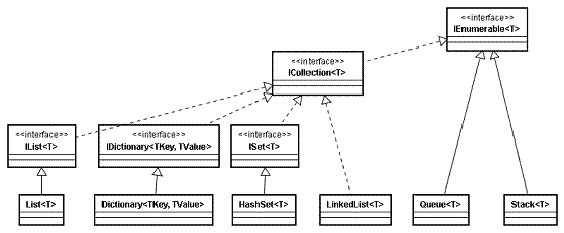
[](http://images.cnblogs.com/cnblogs_com/luminji/201103/201103240912219543.png)

集操作

除了上面我们提到的集合类型，还有其他几个要掌握的集合类型，它们是在实际应用中发展出来的对以上基础类型的扩展：SortedList<T>，SortedDictionary<TKey, TValue>，SortedSet<T>。它们所扩展的对应类为List<T>，Dictionary<TKey,TValue>，HashSet<T>，作用是将原本无序排列的元素，变为有序排列。

除了排序上的需求增加了上面3个集合类，在命名空间System.Collections.Concurrent下，还涉及几个多线程集合类。它们主要是：ConcurrentBag<T>对应List<T>，ConcurrentDictionary<TKey, TValue>对应Dictionary<TKey, TValue>，ConcurrentQueue<T>对应Queue<T>，ConcurrentStack<T>对应Stack<T>。如果我们的集合被用于多线程应用中，可以使用这几个集合类型。关于集合的线程安全性，可以进一步查看MSDN。

本建议到此为止已经介绍了FCL中的大部分泛型集合类，为了对它们有更好的了解，最后我们给出一个主要集合类的类图。实际工作中，应该根据需要选择合适的集合类。

[](http://images.cnblogs.com/cnblogs_com/luminji/201103/201103240912213098.gif)

FCL集合类图

[Creative Commons License](http://creativecommons.org/licenses/by/2.5/cn/)本文基于[Creative Commons Attribution 2.5 China Mainland License](http://creativecommons.org/licenses/by/2.5/cn/)发布，欢迎转载，演绎或用于商业目的，但是必须保留本文的署名<http://www.cnblogs.com/luminji>（包含链接）。如您有任何疑问或者授权方面的协商，请给我留言。

将本文分享到： QQ空间 新浪微博 人人网 开心网 搜狐微博 MSN 谷歌 [更多](http://www.jiathis.com/share?uid=1501904)

分类: [Efficient C#](http://www.cnblogs.com/luminji/category/268612.html),[算法和数据结构](http://www.cnblogs.com/luminji/category/282676.html)

[**好文要顶**](javascript:void(0);) [**关注我**](javascript:void(0);) [**收藏该文**](javascript:void(0);) **[http://common.cnblogs.com/images/icon_weibo_24.png](javascript:void(0);)**

[http://pic.cnblogs.com/face/123061/20131127142331.png](http://home.cnblogs.com/u/luminji/)

[Luminji](http://home.cnblogs.com/u/luminji/)  
[关注 - 45](http://home.cnblogs.com/u/luminji/followees)  
[粉丝 - 1669](http://home.cnblogs.com/u/luminji/followers)

荣誉：[推荐博客](http://www.cnblogs.com/expert/)

[+加关注](javascript:void(0);)

24

0

[«](http://www.cnblogs.com/luminji/archive/2011/03/02/1967934.html)上一篇：[C#高效编程话题集1（每期10话题）](http://www.cnblogs.com/luminji/archive/2011/03/02/1967934.html)  
[»](http://www.cnblogs.com/luminji/archive/2011/03/25/1994097.html)下一篇：[C#高效编程话题集2（每期10话题）](http://www.cnblogs.com/luminji/archive/2011/03/25/1994097.html)

# [改善C#程序的建议4：C#中标准Dispose模式的实现](http://www.cnblogs.com/luminji/archive/2011/03/29/1997812.html)

需要明确一下C#程序（或者说.NET）中的资源。简单的说来，C#中的每一个类型都代表一种资源，而资源又分为两类：

托管资源：由CLR管理分配和释放的资源，即由CLR里new出来的对象；

非托管资源：不受CLR管理的对象，windows内核对象，如文件、数据库连接、套接字、COM对象等；

毫无例外地，如果我们的类型使用到了非托管资源，或者需要显式释放的托管资源，那么，就需要让类型继承接口IDisposable。这相当于是告诉调用者，该类型是需要显式释放资源的，你需要调用我的Dispose方法。

不过，这一切并不这么简单，一个标准的继承了IDisposable接口的类型应该像下面这样去实现。这种实现我们称之为Dispose模式：

[复制代码](javascript:void(0);)

publicclass SampleClass : IDisposable  
{  
//演示创建一个非托管资源  
private IntPtr nativeResource = Marshal.AllocHGlobal(100);  
//演示创建一个托管资源  
private AnotherResource managedResource =new AnotherResource();  
privatebool disposed =false;  
  
///<summary>  
/// 实现IDisposable中的Dispose方法  
///</summary>  
publicvoid Dispose()  
{  
//必须为true  
Dispose(true);  
//通知垃圾回收机制不再调用终结器（析构器）  
GC.SuppressFinalize(this);  
}  
  
///<summary>  
/// 不是必要的，提供一个Close方法仅仅是为了更符合其他语言（如C++）的规范  
///</summary>  
publicvoid Close()  
{  
Dispose();  
}  
  
///<summary>  
/// 必须，以备程序员忘记了显式调用Dispose方法  
///</summary>  
~SampleClass()  
{  
//必须为false  
Dispose(false);  
}  
  
///<summary>  
/// 非密封类修饰用protected virtual  
/// 密封类修饰用private  
///</summary>  
///<param name="disposing"></param>  
protectedvirtualvoid Dispose(bool disposing)  
{  
if (disposed)  
{  
return;  
}  
if (disposing)  
{  
// 清理托管资源  
if (managedResource !=null)  
{  
managedResource.Dispose();  
managedResource =null;  
}  
}  
// 清理非托管资源  
if (nativeResource != IntPtr.Zero)  
{  
Marshal.FreeHGlobal(nativeResource);  
nativeResource = IntPtr.Zero;  
}  
//让类型知道自己已经被释放  
disposed =true;  
}  
  
publicvoid SamplePublicMethod()  
{  
if (disposed)  
{  
thrownew ObjectDisposedException("SampleClass", "SampleClass is disposed");  
}  
//省略  
}  
}

[复制代码](javascript:void(0);)

在Dispose模式中，几乎每一行都有特殊的含义。

在标准的Dispose模式中，我们注意到一个以~开头的方法：

[复制代码](javascript:void(0);)

///<summary>  
/// 必须，以备程序员忘记了显式调用Dispose方法  
///</summary>  
~SampleClass()  
{  
//必须为false  
Dispose(false);  
}

[复制代码](javascript:void(0);)

这个方法叫做类型的终结器。提供终结器的全部意义在于：我们不能奢望类型的调用者肯定会主动调用Dispose方法，基于终结器会被垃圾回收器调用这个特点，终结器被用做资源释放的补救措施。

一个类型的Dispose方法应该允许被多次调用而不抛异常。鉴于这个原因，类型内部维护了一个私有的布尔型变量disposed：

        private bool disposed = false;

在实际处理代码清理的方法中，加入了如下的判断语句：

if (disposed)  
{  
return;  
}  
//省略清理部分的代码，并在方法的最后为disposed赋值为true  
disposed =true;

这意味着类型如果被清理过一次，则清理工作将不再进行。

应该注意到：在标准的Dispose模式中，真正实现IDisposable接口的Dispose方法，并没有实际的清理工作，它实际调用的是下面这个带布尔参数的受保护的虚方法：

[复制代码](javascript:void(0);)

///<summary>  
/// 非密封类修饰用protected virtual  
/// 密封类修饰用private  
///</summary>  
///<param name="disposing"></param>  
protectedvirtualvoid Dispose(bool disposing)  
{  
//省略代码  
}

[复制代码](javascript:void(0);)

之所以提供这样一个受保护的虚方法，是为了考虑到这个类型会被其他类继承的情况。如果类型存在一个子类，子类也许会实现自己的Dispose模式。受保护的虚方法用来提醒子类必须在实现自己的清理方法的时候注意到父类的清理工作，即子类需要在自己的释放方法中调用base.Dispose方法。

还有，我们应该已经注意到了真正撰写资源释放代码的那个虚方法是带有一个布尔参数的。之所以提供这个参数，是因为我们在资源释放时要区别对待托管资源和非托管资源。

在供调用者调用的显式释放资源的无参Dispose方法中，调用参数是true：

publicvoid Dispose()  
{  
//必须为true  
Dispose(true);  
//其他省略  
}

这表明，这个时候代码要同时处理托管资源和非托管资源。

在供垃圾回收器调用的隐式清理资源的终结器中，调用参数是false：

~SampleClass()  
{  
//必须为false  
Dispose(false);  
}

这表明，隐式清理时，只要处理非托管资源就可以了。

那么，为什么要区别对待托管资源和非托管资源。在认真阐述这个问题之前，我们需要首先弄明白：托管资源需要手动清理吗？不妨先将C#中的类型分为两类，一类继承了IDisposable接口，一类则没有继承。前者，我们暂时称之为非普通类型，后者我们称之为普通类型。非普通类型因为包含非托管资源，所以它需要继承IDisposable接口，但是，这个包含非托管资源的类型本身，它是一个托管资源。所以说，托管资源需要手动清理吗？这个问题的答案是：托管资源中的普通类型，不需要手动清理，而非普通类型，是需要手动清理的（即调用Dispose方法）。

Dispose模式设计的思路基于：如果调用者显式调用了Dispose方法，那么类型就该按部就班为自己的所以资源全部释放掉。如果调用者忘记调用Dispose方法，那么类型就假定自己的所有托管资源（哪怕是那些上段中阐述的非普通类型）全部交给垃圾回收器去回收，而不进行手工清理。理解了这一点，我们就理解了为什么Dispose方法中，虚方法传入的参数是true，而终结器中，虚方法传入的参数是false。

注意：我们提到了需要及时释放资源，却并没有进一步细说是否需要及时让引用等于null这一点。有一些人认为等于null可以帮助垃圾回收机制早点发现并标识对象是垃圾。其他人则认为这没有任何帮助。下一篇“[**引用类型赋值为null与加速垃圾回收**](http://www.cnblogs.com/luminji/archive/2011/04/07/2007205.html)”我们再细说这一点。

# [改善C#程序的建议5：引用类型赋值为null与加速垃圾回收](http://www.cnblogs.com/luminji/archive/2011/04/07/2007205.html)

在标准的Dispose模式中（见前一篇博客“[C#中标准Dispose模式的实现](http://www.cnblogs.com/luminji/archive/2011/03/29/1997812.html)”），提到了需要及时释放资源，却并没有进一步细说让引用等于null是否有必要。

有一些人认为等于null可以帮助垃圾回收机制早点发现并标识对象是垃圾。其他人则认为这没有任何帮助。是否赋值为null的问题首先在方法的内部被人提起。现在，为了更好的阐述提出的问题，我们来撰写一个Winform窗体应用程序。如下：

[复制代码](javascript:void(0);)

privatevoid button1\_Click(object sender, EventArgs e)  
{  
Method1();  
Method2();  
}  
  
privatevoid button2\_Click(object sender, EventArgs e)  
{  
GC.Collect();  
}  
  
privatevoid Method1()  
{  
SimpleClass s =new SimpleClass("method1");  
s =null;  
//其它无关工作代码（这条注释源于回应回复的朋友的质疑）  
}  
privatevoid Method2()  
{  
SimpleClass s =new SimpleClass("method2");  
}  
}  
  
class SimpleClass  
{  
string m\_text;  
  
public SimpleClass(string text)  
{  
m\_text = text;  
}  
  
~SimpleClass()  
{  
MessageBox.Show(string.Format("SimpleClass Disposed, tag:{0}", m\_text));  
}  
}

[复制代码](javascript:void(0);)

先点击按钮1，再点击按钮2释放，我们会发现：

q 方法Method2中的对象先被释放，虽然它在Method1之后被调用；

q 方法Method2中的对象先被释放，虽然它不像Method1那样为对象引用赋值为null；

在CLR托管应用程序中，存在一个“根”的概念，类型的静态字段、方法参数以及局部变量都可以作为“根”存在（值类型不能作为“根”，只有引用类型的指针才能作为“根”）。

上面的两个方法中各自的局部变量，在代码运行过程中会在内存中各自创建一个“根”.在一次垃圾回收中，垃圾回收器会沿着线程栈上行检查“根”。检查到方法内的“根”时，如果发现没有任何一个地方引用了局部变量，则不管是否为变量赋值为null，都意味着该“根”已经被停止掉。然后垃圾回收器发现该根的引用为空，同时标记该根可被释放，这也表示着Simple类型对象所占用的内存空间可被释放。所以，在上面的这个例子中，为s指定为null丝毫没有意义（方法的参数变量也是这种情况）。

**更进一步的事实是，JIT编译器是一个经过优化的编译器，无论我们是否在方法内部为局部变量赋值为null，该语句都会被忽略掉**：

|  |
| --- |
| s = null; |

在我们将项目设置为Release模式下，上面的这行代码将根本不会被编译进运行时内。

**正式由于上面这样的分析，很多人认为为对象赋值为null完全没有必要。但是，在另外一种情况下，却要注意及时为变量赋值为null。那就是类型的静态字段。**为类型对象赋值为null，并不意味着同时为类型的静态字段赋值为null：

[复制代码](javascript:void(0);)

privatevoid button1\_Click(object sender, EventArgs e)  
{  
SimpleClass s =new SimpleClass("test");  
}  
  
privatevoid button2\_Click(object sender, EventArgs e)  
{  
GC.Collect();  
}  
}  
  
class SimpleClass  
{  
static AnotherSimpleClass asc =new AnotherSimpleClass();  
string m\_text;  
  
public SimpleClass(string text)  
{  
m\_text = text;  
}  
  
~SimpleClass()  
{  
//asc = null;  
MessageBox.Show(string.Format("SimpleClass Disposed, tag:{0}", m\_text));  
}  
}  
  
class AnotherSimpleClass  
{  
~AnotherSimpleClass()  
{  
MessageBox.Show("AnotherSimpleClass Disposed");  
}  
}

[复制代码](javascript:void(0);)

以上代码运行的结果使我们发现，当执行垃圾回收，当类型SampleClass对象被回收的时候，类型的静态字段asc并没有被回收。

必须要将SimpleClass的终结器中注释的那条代码启用。

字段asc才能被正确释放（注意，要点击两次释放按钮。这是因为一次垃圾回收会仅仅首先执行终结器）。之所以静态字段不被释放（同时赋值为null语句也不会像局部变量那样被运行时编译器优化掉），是因为类型的静态字段一旦被创建，该“根”就一直存在。所以垃圾回收器始终不会认为它是一个垃圾。非静态字段不存在这个问题。将asc改为非静态，再次运行上面的代码，会发现asc随着类型的释放而被释放。

上文代码的例子中，让asc=null是在终结器中完成的，实际工作中，一旦我们感觉到自己的静态引用类型参数占用内存空间比较大，并且使用完毕后不再使用，则可以立刻将其赋值为null。这也许并不必要，但这绝对是一个好习惯。**试想一下在一个大系统中，那些时不时在类型中出现的静态变量吧，它们就那样静静地呆在内存里，一旦被创建，就永远不离开，越来越多，越来越多……**。

微信扫一扫，关注最课程（[www.zuikc.com](http://www.zuikc.com/)），获取更多我的文章，获取软件开发每日一练

[Creative Commons License](http://creativecommons.org/licenses/by/2.5/cn/)本文基于[Creative Commons Attribution 2.5 China Mainland License](http://creativecommons.org/licenses/by/2.5/cn/)发布，欢迎转载，演绎或用于商业目的，但是必须保留本文的署名<http://www.cnblogs.com/luminji>（包含链接）。如您有任何疑问或者授权方面的协商，请给我留言。

将本文分享到： QQ空间 新浪微博 人人网 开心网 搜狐微博 MSN 谷歌 [更多](http://www.jiathis.com/share?uid=1501904)

分类: [Efficient C#](http://www.cnblogs.com/luminji/category/268612.html)

[**好文要顶**](javascript:void(0);) [**关注我**](javascript:void(0);) [**收藏该文**](javascript:void(0);) **[http://common.cnblogs.com/images/icon_weibo_24.png](javascript:void(0);)**

[http://pic.cnblogs.com/face/123061/20131127142331.png](http://home.cnblogs.com/u/luminji/)

[Luminji](http://home.cnblogs.com/u/luminji/)  
[关注 - 45](http://home.cnblogs.com/u/luminji/followees)  
[粉丝 - 1669](http://home.cnblogs.com/u/luminji/followers)

荣誉：[推荐博客](http://www.cnblogs.com/expert/)

[+加关注](javascript:void(0);)

6

1

[«](http://www.cnblogs.com/luminji/archive/2011/04/02/2003276.html)上一篇：[VS2010下如何调试Framework源代码（即FCL）](http://www.cnblogs.com/luminji/archive/2011/04/02/2003276.html)  
[»](http://www.cnblogs.com/luminji/archive/2011/04/19/2019597.html)下一篇：[C#高效编程话题集3（每期10话题）](http://www.cnblogs.com/luminji/archive/2011/04/19/2019597.html)

# [改善C#程序的建议6：在线程同步中使用信号量](http://www.cnblogs.com/luminji/archive/2011/05/03/2034890.html)

所谓线程同步，就是多个线程之间在某个对象上执行等待（也可理解为锁定该对象），直到该对象被解除锁定。C#中对象的类型分为引用类型和值类型。CLR在这两种类型上的等待是不一样的。我们可以简单的理解为在CLR中，值类型是不能被锁定的，也即：不能在一个值类型对象上执行等待。而在引用类型上的等待机制，则分为两类：锁定和信号同步。

锁定，使用关键字lock和类型Monitor。两者没有实质区别，前者其实是后者的语法糖。这是最常用的同步技术；

本建议我们讨论的是信号同步。信号同步机制中涉及的类型都继承自抽象类WaitHandle，这些类型有EventWaitHandle（类型化为AutoResetEvent、ManualResetEvent）和Semaphore以及Mutex。见类图6-3：

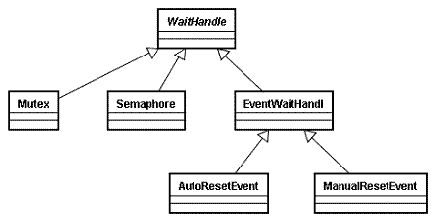
[](http://images.cnblogs.com/cnblogs_com/luminji/201105/201105030848062301.gif)

图 同步功能类类图

EventWaitHandle（子类为AutoResetEvent、ManualResetEvent）和Semaphore以及Mutex都继承自WaitHandle，所以它们底层的原理是一致的，维护的都是一个系统内核句柄。不过我们仍需简单的区分下这三类类型。

EventWaitHandle，维护一个由内核产生的布尔类型对象（我们称之为“阻滞状态”），如果其值为false，那么在它上面等待的线程就阻塞。可以调用类型的Set方法将其值设置为true，解除阻塞。EventWaitHandle类型的两个子类AutoResetEvent和ManualResetEvent，它们的区别并不大，本建议接下来会针对它们阐述如何正确使用信号量。

Semaphore，维护一个由内核产生的整型变量，如果其值为0，则在它上面等待的线程就阻塞，其值大于0，就解除阻塞，同时，每解除阻塞一个线程，其值就减1。

EventWaitHandle和Semaphore提供的都是单应用程序域内的线程同步功能，Mutex则不同，它为我们提供了跨应用程序域阻塞和解除阻塞线程的能力。

**1：使用信号机制提供线程同步的一个简单的例子**

使用信号机制提供线程同步的一个简单的例子如下：

[复制代码](javascript:void(0);)

AutoResetEvent autoResetEvent = new AutoResetEvent(false);  
  
 private void buttonStartAThread\_Click(object sender, EventArgs e)  
 {  
 Thread tWork = new Thread(() =>  
 {  
 label1.Text = "线程启动..." + Environment.NewLine;  
 label1.Text += "开始处理一些实际的工作" + Environment.NewLine;  
 //省略工作代码  
 label1.Text += "我开始等待别的线程给我信号，才愿意继续下去" + Environment.NewLine;   
 autoResetEvent.WaitOne();  
 label1.Text += "我继续做一些工作，然后结束了！";  
 //省略工作代码  
 });  
 tWork.IsBackground = true;  
 tWork.Start();  
 }  
  
 private void buttonSet\_Click(object sender, EventArgs e)  
 {  
 //给在autoResetEvent上等待的线程一个信号  
 autoResetEvent.Set();  
 }

[复制代码](javascript:void(0);)

这是一个简单的Winform窗体程序，其中一个按钮负责开启一个新的线程，还有一个按钮负责给刚开启的那个线程发送信号。现在详细解释这里面发生的事情。

AutoResetEvent autoResetEvent = new AutoResetEvent(false);

这段代码创建了一个同步类型对象autoResetEvent，它设置自己的默认阻滞状态是false。这意味着任何在它上面进行等待的线程将会被阻滞。所谓进行等待，就是在线程中应用：

autoResetEvent.WaitOne();

这说明tWork开始在autoResetEvent上等待任何其它地方给它的信号。信号来了，则tWork开始继续工作，否则就一直等着（即阻滞）。接下来我们看到在主线程中（本例中即UI线程，它相对线程tWork来说，就是一个“另外的线程”）：

autoResetEvent.Set();

主线程通过上面这句代码负责向在autoResetEvent上等待的线程tWork上下文发送信号，即将tWork的阻滞状态设置为true。tWork接收到这个信号，开始继续工作。

这个例子相当简单，但是已经完整说明了信号机制的工作原理。

**2：AutoResetEvent和ManualResetEvent的区别**

AutoResetEvent和ManualResetEvent有这样的区别：前者在发送信号完毕后（即调用Set方法），自动将自己的阻滞状态设置为false，而后者需要进行手动设定。可以通过一个例子来说明这种区别：

[复制代码](javascript:void(0);)

AutoResetEvent autoResetEvent = new AutoResetEvent(false);  
  
 private void buttonStartAThread\_Click(object sender, EventArgs e)  
 {  
 StartThread1();  
 StartThread2();  
 }  
  
 private void StartThread1()  
 {  
 Thread tWork1 = new Thread(() =>  
 {  
 label1.Text = "线程1启动..." + Environment.NewLine;  
 label1.Text += "开始处理一些实际的工作" + Environment.NewLine;  
 //省略工作代码  
 label1.Text += "我开始等待别的线程给我信号，才愿意继续下去" + Environment.NewLine;  
 autoResetEvent.WaitOne();  
 label1.Text += "我继续做一些工作，然后结束了！";  
 //省略工作代码  
 });  
 tWork1.IsBackground = true;  
 tWork1.Start();  
 }  
  
 private void StartThread2()  
 {  
 Thread tWork2 = new Thread(() =>  
 {  
 label2.Text = "线程2启动..." + Environment.NewLine;  
 label2.Text += "开始处理一些实际的工作" + Environment.NewLine;  
 //省略工作代码  
 label2.Text += "我开始等待别的线程给我信号，才愿意继续下去" + Environment.NewLine;  
 autoResetEvent.WaitOne();  
 label2.Text += "我继续做一些工作，然后结束了！";  
 //省略工作代码  
 });  
 tWork2.IsBackground = true;  
 tWork2.Start();  
 }  
  
 private void buttonSet\_Click(object sender, EventArgs e)  
 {  
 //给在autoResetEvent上等待的线程一个信号  
 autoResetEvent.Set();  
 }

[复制代码](javascript:void(0);)

这个例子的本意是要让新起的两个工作线程tWork1和tWork2都阻滞起来，直到收到主线程的信号再继续工作。结果程序运行的结果是，只有一个工作线程继续工作，另外一个工作线程则继续保持阻滞状态。我想原因大家都已经想到了。由于AutoResetEvent在发送信号完毕就在内核中自动将自己的状态设置回false了，所以另外一个工作线程相当于根本没有收到主线程的信号。

要修正这个问题，可以使用ManualResetEvent。大家可以换成ManualResetEvent试一下。

**3：应用实例**

最后，再举一个需要用到线程同步的实际例子：模拟网络通信。客户端在运行过程中，服务器每隔一段的时间会给客户端发送心跳数据。实际工作中服务器和客户端会是网络中两台不同的终端，在这个例子中我们进行了简化。工作线程tClient模拟客户端，主线程（UI线程）模拟服务器端。客户端每3秒检测是否收到服务器的心跳数据，如果没有心跳数据，则显示网络连接断开。代码如下：

[复制代码](javascript:void(0);)

AutoResetEvent autoResetEvent = new AutoResetEvent(false);  
  
 private void buttonStartAThread\_Click(object sender, EventArgs e)  
 {  
 Thread tClient = new Thread(() =>  
 {  
 while (true)  
 {  
 //等3秒，3秒没有信号，显示断开  
 //有信号，则显示更新  
 bool re = autoResetEvent.WaitOne(3000);  
 if (re)  
 {  
 label1.Text = string.Format("时间：{0}，{1}", DateTime.Now.ToString(), "保持连接状态");  
 }  
 else  
 {  
 label1.Text = string.Format("时间：{0}，{1}", DateTime.Now.ToString(), "断开，需要重启");  
 }  
 }  
 });  
 tClient.IsBackground = true;  
 tClient.Start();  
 }  
  
 private void buttonSet\_Click(object sender, EventArgs e)  
 {  
 //模拟发送心跳数据  
 autoResetEvent.Set();  
 }

[复制代码](javascript:void(0);)

备注：由本问题带来一个Winform跨线程控件赋值和操作的问题。由于在本示例中不影响上面代码的运行，所以没有涉及，但是回复中有人提出来，所以提前简述一下Winform的线程模型：

**在Winform框架中，有一个ISynchronizeInvoke接口，所有的UI元素（表现为Control）都继承了该接口。其中，接口中的InvokdRequired属性表示了当前线程是否是创建它的线程。接口中的Invoke和BeginInvoke方法负责将消息发送到消息队列中，这样，UI线程就能够正确处理它。**

具体到代码中，对于夸线程控件赋值，可以采用下面的方法：

**this.label1.BeginInvoke(new Action(()=>  
{  
 this.label1.Text = "跨线程中赋值";  
}));**

**之前的话题：**

#### [改善C#程序的建议5：引用类型赋值为null与加速垃圾回收](http://www.cnblogs.com/luminji/archive/2011/04/07/2007205.html)

#### [改善C#程序的建议4：C#中标准Dispose模式的实现](http://www.cnblogs.com/luminji/archive/2011/03/29/1997812.html)

#### [改善C#程序的建议3：在C#中选择正确的集合进行编码](http://www.cnblogs.com/luminji/archive/2011/03/24/1993393.html)

#### [改善C#程序的建议2：C#中dynamic的正确用法](http://www.cnblogs.com/luminji/archive/2011/02/18/1957484.html)

#### [改善C#程序的建议1：非用ICloneable不可的理由](http://www.cnblogs.com/luminji/archive/2011/02/02/1948826.html)

[Creative Commons License](http://creativecommons.org/licenses/by/2.5/cn/)本文基于[Creative Commons Attribution 2.5 China Mainland License](http://creativecommons.org/licenses/by/2.5/cn/)发布，欢迎转载，演绎或用于商业目的，但是必须保留本文的署名<http://www.cnblogs.com/luminji>（包含链接）。如您有任何疑问或者授权方面的协商，请给我留言。

将本文分享到： QQ空间 新浪微博 人人网 开心网 搜狐微博 MSN 谷歌 [更多](http://www.jiathis.com/share?uid=1501904)

分类: [Efficient C#](http://www.cnblogs.com/luminji/category/268612.html)

[**好文要顶**](javascript:void(0);) [**关注我**](javascript:void(0);) [**收藏该文**](javascript:void(0);) **[http://common.cnblogs.com/images/icon_weibo_24.png](javascript:void(0);)**

[http://pic.cnblogs.com/face/123061/20131127142331.png](http://home.cnblogs.com/u/luminji/)

[Luminji](http://home.cnblogs.com/u/luminji/)  
[关注 - 45](http://home.cnblogs.com/u/luminji/followees)  
[粉丝 - 1669](http://home.cnblogs.com/u/luminji/followers)

荣誉：[推荐博客](http://www.cnblogs.com/expert/)

[+加关注](javascript:void(0);)

20

0

[«](http://www.cnblogs.com/luminji/archive/2011/04/28/2031972.html)上一篇：[WPF快速指导15：动画](http://www.cnblogs.com/luminji/archive/2011/04/28/2031972.html)  
[»](http://www.cnblogs.com/luminji/archive/2011/05/08/2040170.html)下一篇：[改善C#程序的建议7：正确停止线程](http://www.cnblogs.com/luminji/archive/2011/05/08/2040170.html)

# [改善C#程序的建议7：正确停止线程](http://www.cnblogs.com/luminji/archive/2011/05/08/2040170.html)

开发者总尝试对自己的代码有更多的控制。“让那个还在工作的线程马上停止下来”就是诸多要求中的一种。然而事与愿违，这里面至少存在两个问题：

第一个问题是：正如线程不能立即启动一样，线程也并不能说停就停。无论采用何种方式通知工作线程需要停止，工作线程都会忙完手头最紧要的活，然后在它觉得合适的时候退出。以最传统的Thread.Abort方法为例，如果线程当前正在执行的是一段非托管代码，那么CLR就不会抛出ThreadAbortException，只有当代码继续回到CLR中时，才会引发ThreadAbortException。当然，即便是在CLR环境中，ThreadAbortException也不会立即引发。

其次，正确停止线程，不在于调用者采取了什么行为（如最开始的Thread.Abort()方法），而更多依赖于工作线程是否能主动响应调用者的停止请求。大体机制是，如果线程需要被停止，那么线程自身就得负责开放给调用者这样的接口：Cancled，然后线程在工作的同时，还得以某种频率检测Cancled标识，若检测到Cancled，线程自己负责退出。

FCL现在为我们提供了标准的取消模式：协作式取消（Cooperative Cancellation）。协作式取消的机制就是上文提到的机制。下面是一个最基础的协作式取消的样例：

[复制代码](javascript:void(0);)

CancellationTokenSource cts =new CancellationTokenSource();  
Thread t =new Thread(() =>  
{  
while (true)  
{  
if (cts.Token.IsCancellationRequested)  
{  
Console.WriteLine("线程被终止！");  
break;  
}  
Console.WriteLine(DateTime.Now.ToString());  
Thread.Sleep(1000);  
}  
});  
t.Start();  
Console.ReadLine();  
cts.Cancel();

[复制代码](javascript:void(0);)

调用者使用CancellationTokenSource的Cancle方法通知工作线程退出。工作线程则以大致1000毫秒的频率一边工作，一边检查是否有外界传入进来的Cancel信号。若有这样的信号，则负责退出。可以看到，在正确停止线程的机制中，真正起到主要作用的是线程本身。样例中的工作代码比较简单，不过也足以说明问题。更复杂的计算式的工作，也应该以这样的一种方式，妥善而正确地处理退出。

协作式取消中的关键类型是CancellationTokenSource。它有一个关键属性Token，Token是一个名为CancellationToken的值类型。CancellationToken继而进一步提供了布尔值的属性IsCancellationRequested作为需要取消工作的标识。CancellationToken还有一个方法尤其值得注意，那就是Register方法。它负责传递一个Action委托，在线程停止的时候被回调，使用方法如：

cts.Token.Register(() =>  
{  
Console.WriteLine("工作线程被终止了。");  
});

本建议中的例子使用Thread进行了演示，使用ThreadPool也是一样的模式，这里就不再赘述。后面我们还会讲到任务Task，它依赖于CancellationTokenSource和CancellationToken完成了所有的取消控制。

 微信扫一扫，关注最课程（[www.zuikc.com](http://www.zuikc.com/)），获取更多我的文章，获取软件开发每日一练

**之前的话题：**

##### [改善C#程序的建议6：在线程同步中使用信号量](http://www.cnblogs.com/luminji/archive/2011/05/03/2034890.html) [改善C#程序的建议5：引用类型赋值为null与加速垃圾回收](http://www.cnblogs.com/luminji/archive/2011/04/07/2007205.html) [改善C#程序的建议4：C#中标准Dispose模式的实现](http://www.cnblogs.com/luminji/archive/2011/03/29/1997812.html) [改善C#程序的建议3：在C#中选择正确的集合进行编码](http://www.cnblogs.com/luminji/archive/2011/03/24/1993393.html) [改善C#程序的建议2：C#中dynamic的正确用法](http://www.cnblogs.com/luminji/archive/2011/02/18/1957484.html) [改善C#程序的建议1：非用ICloneable不可的理由](http://www.cnblogs.com/luminji/archive/2011/02/02/1948826.html)

[Creative Commons License](http://creativecommons.org/licenses/by/2.5/cn/)本文基于[Creative Commons Attribution 2.5 China Mainland License](http://creativecommons.org/licenses/by/2.5/cn/)发布，欢迎转载，演绎或用于商业目的，但是必须保留本文的署名<http://www.cnblogs.com/luminji>（包含链接）。如您有任何疑问或者授权方面的协商，请给我留言。

将本文分享到： QQ空间 新浪微博 人人网 开心网 搜狐微博 MSN 谷歌 [更多](http://www.jiathis.com/share?uid=1501904)

分类: [Efficient C#](http://www.cnblogs.com/luminji/category/268612.html)

[**好文要顶**](javascript:void(0);) [**关注我**](javascript:void(0);) [**收藏该文**](javascript:void(0);) **[http://common.cnblogs.com/images/icon_weibo_24.png](javascript:void(0);)**

[http://pic.cnblogs.com/face/123061/20131127142331.png](http://home.cnblogs.com/u/luminji/)

[Luminji](http://home.cnblogs.com/u/luminji/)  
[关注 - 45](http://home.cnblogs.com/u/luminji/followees)  
[粉丝 - 1669](http://home.cnblogs.com/u/luminji/followers)

荣誉：[推荐博客](http://www.cnblogs.com/expert/)

[+加关注](javascript:void(0);)

3

0

[«](http://www.cnblogs.com/luminji/archive/2011/05/03/2034890.html)上一篇：[改善C#程序的建议6：在线程同步中使用信号量](http://www.cnblogs.com/luminji/archive/2011/05/03/2034890.html)  
[»](http://www.cnblogs.com/luminji/archive/2011/05/09/2040563.html)下一篇：[改善C#程序的建议8：避免锁定不恰当的同步对象](http://www.cnblogs.com/luminji/archive/2011/05/09/2040563.html)

# [改善C#程序的建议8：避免锁定不恰当的同步对象](http://www.cnblogs.com/luminji/archive/2011/05/09/2040563.html)

在C#中让线程同步的另一种编码方式就是使用线程锁。所谓线程锁，就是锁住一个资源，使得应用程序只能在此刻有一个线程访问该资源。可以用下面这句不是那么贴切的话来理解线程锁的作用：锁，就是让多线程变成单线程。在C#中，可以将被锁定的资源理解成new出来的普通对象。

既然需要锁定的资源就是一个C#中的对象，我们就该仔细思考，到底什么样的对象能够成为一个锁对象（也叫同步对象）？在选择同步对象的时候，应当始终注意以下几点：

θ    q同步对象在需要同步的多个线程中是可见的、同一个对象；

θ    q非静态方法中，静态变量不应作为同步对象；

θ    q值类型对象不能作为同步对象；

θ    q避免将字符串作为同步对象。

θ    q降低同步对象的可见性。

**第一点，需要锁定的对象在多个线程中是可见的、同一个对象**

“可见的”这是显而易见的，如果对象不可见，就不能被锁定。“同一个对象”，这理解起来也很好理解，如果锁定的不是同一个对象，那又如何来同步两个对象呢？可是，不见得我们在这上面不会犯错误。为了阐述本建议，我们先模拟一个必须使用到锁的场景：在遍历一个集合的过程中，同时在另外一个线程中删除集合中的某项。下面的这个例子中，如果没有lock语句，将会抛出异常InvalidOperationException：“集合已修改；可能无法执行枚举”：

[复制代码](javascript:void(0);)

public partial class FormMain : Form  
{  
public FormMain()  
{  
InitializeComponent();  
}  
  
AutoResetEvent autoSet = new AutoResetEvent(false);  
List<string> tempList = new List<string>() { "init0", "init1", "init2" };  
  
private void buttonStartThreads\_Click(object sender, EventArgs e)  
{  
object syncObj = new object();  
  
Thread t1 = new Thread(() =>  
{  
//确保等待t2开始之后才运行下面的代码  
autoSet.WaitOne();  
lock (syncObj)  
{  
foreach (var item in tempList)  
{  
Thread.Sleep(1000);  
}  
}  
});  
t1.IsBackground = true;  
t1.Start();  
  
Thread t2 = new Thread(() =>  
{  
//通知t1可以执行代码  
autoSet.Set();  
//沉睡1秒是为了确保删除操作在t1的迭代过程中  
Thread.Sleep(1000);  
lock (syncObj)  
{  
tempList.RemoveAt(1);  
}  
});  
t2.IsBackground = true;  
t2.Start();  
}  
}

[复制代码](javascript:void(0);)

这是一个Winform窗体应用程序，我们需要演示的功能在按钮的点击事件中。对象syncObj对于线程t1和t2来说，在CLR中肯定是同一个对象。所以上面的示例运行没有问题。

现在，我们将以上示例重构一下。将实际的工作代码移到一个类型SampleClass中去，该示例要在多个SampleClass实例间操作一个静态字段：

[复制代码](javascript:void(0);)

private void buttonStartThreads\_Click(object sender, EventArgs e)  
{  
SampleClass sample1 = new SampleClass();  
SampleClass sample2 = new SampleClass();  
sample1.StartT1();  
sample2.StartT2();  
}  
  
class SampleClass  
{  
public static List<string> TempList = new List<string>() { "init0", "init1", "init2" };  
static AutoResetEvent autoSet = new AutoResetEvent(false);  
object syncObj = new object();  
  
public void StartT1()  
{  
Thread t1 = new Thread(() =>  
{  
//确保等待t2开始之后才运行下面的代码  
autoSet.WaitOne();  
lock (syncObj)  
{  
foreach (var item in TempList)  
{  
Thread.Sleep(1000);  
}  
}  
});  
t1.IsBackground = true;  
t1.Start();  
}  
  
public void StartT2()  
{  
Thread t2 = new Thread(() =>  
{  
//通知t1可以执行代码  
autoSet.Set();  
//沉睡1秒是为了确保删除操作在t1的迭代过程中  
Thread.Sleep(1000);  
lock (syncObj)  
{  
TempList.RemoveAt(1);  
}  
});  
t2.IsBackground = true;  
t2.Start();  
}  
}

[复制代码](javascript:void(0);)

该示例运行起来抛出异常InvalidOperationException：“集合已修改；可能无法执行枚举”。查看类型SampleClass的方法StartT1和StartT2，方法内部锁定的是SampleClass的实例变量syncObject。实例变量意味着每创建一个SampleClass的实例都会生成一个syncObject对象。在本例中，调用者一共创建了两个SampleClass实例，继而分别调用：

sample1.StartT1();  
sample2.StartT2();

以上代码锁定的是两个不同的syncObject，这等于完全没有达到两个线程锁定同一个对象的目的。要修正以上的错误，只要将syncObject变成static就可以了。

另外，思考一下lock(this)，我们同样不建议在代码中编写这样的代码。如果两个对象的实例分别执行了锁定的代码，实际锁定的也是两个对象，完全不能达到同步的目的。

**第二个注意事项：非静态方法中，静态变量不应作为同步对象**

我们刚说完，要修正第一点中的示例，需要将syncObject变成static。这似乎和本注意事项有矛盾。实际上，第一点中的示例代码仅出于演示的目的。我们强烈建议你不要在实际应用中编写此类代码，在编写多线程代码时，要遵循这样的一个原则：类型的静态方法应当保证线程安全，非静态方法不需实现线程安全。FCL中的绝大部分类，都遵循了这个原则。如果将syncObject变成static，就相当于让非静态方法具备线程安全性，这带来的一个问题是，如果应用程序中该类型存在多个实例，在遇到这个锁的时候，都会产生同步，而这可能不是开发者原先所愿意看到的。

**第三点：值类型对象不能作为同步对象**

值类型在传递另一个线程的时候，会创建一个副本，这相当于每个线程锁定的也是两个对象。故，值类型对象不能作为同步对象。第二点实际也可以归结到第一点中。

**第四点，锁定字符串是完全没有必要，而且相当危险的**

这整个过程看上去和值类型正好相反。字符串在CLR中会被暂存到内存里，如果有两个变量被分配了相同内容的字符串，那么这两个引用会被指向同一块内存。所以，如果有两个地方同时使用了lock(“abc”)，那么它们实际锁定的是同一个对象，导致整个应用程序被阻滞。

**第五点：降低同步对象的可见性**

可见范围最广的一种同步对象是typeof(SampleClass)。typeof方法所返回的结果，也就是类型的type，是SampleClass的所有实例所共有的，即：所有实例的type都指向typeof方法的结果。这样一来，如果我们lock(typeof(SampeClass))，当前应用程序中的所有SampleClass的实例的线程，将会全部被同步。这样编码是完全没有必要的，这样的同步对象太开放了。

另外，同步对象一般来说，也不应该是一个公共变量或属性。在FCL的早期版本中，一些常用的集合类型，如ArrayList，提供了公共属性SyncRoot，让我们锁定以便进行一些线程安全的操作。所以你一定会觉得我们刚才的结论不正确。其实不然，ArrayList的操作，大部分的应用场景不涉及到多线程同步，所以它的方法更多的是单线程应用场景。线程同步是一个非常耗时（也就是低效）的操作。若ArrayList的所有非静态方法都要考虑线程安全，那么ArrayList完全可以将这个SyncRoot变成静态私有。现在它将SyncRoot变为公开的，是让调用者自己去决定操作是否需要线程安全。在我们自己编写的大部分代码中，除非也有这样的要求，否则就应该始终考虑降低同步对象的可见性，将我们的同步对象藏起来，只开放给自己或自己的子类就够了（需要开放给子类的情况其实也不多见）。

**之前内容：**  
[**改善C#程序的建议7：正确停止线程**](http://www.cnblogs.com/luminji/archive/2011/05/08/2040170.html)[**改善C#程序的建议6：在线程同步中使用信号量**](http://www.cnblogs.com/luminji/archive/2011/05/03/2034890.html)[**改善C#程序的建议5：引用类型赋值为null与加速垃圾回收**](http://www.cnblogs.com/luminji/archive/2011/04/07/2007205.html)[**改善C#程序的建议4：C#中标准Dispose模式的实现**](http://www.cnblogs.com/luminji/archive/2011/03/29/1997812.html)[**改善C#程序的建议3：在C#中选择正确的集合进行编码**](http://www.cnblogs.com/luminji/archive/2011/03/24/1993393.html)[**改善C#程序的建议2：C#中dynamic的正确用法**](http://www.cnblogs.com/luminji/archive/2011/02/18/1957484.html)[**改善C#程序的建议1：非用ICloneable不可的理由**](http://www.cnblogs.com/luminji/archive/2011/02/02/1948826.html)

[Creative Commons License](http://creativecommons.org/licenses/by/2.5/cn/)本文基于[Creative Commons Attribution 2.5 China Mainland License](http://creativecommons.org/licenses/by/2.5/cn/)发布，欢迎转载，演绎或用于商业目的，但是必须保留本文的署名<http://www.cnblogs.com/luminji>（包含链接）。如您有任何疑问或者授权方面的协商，请给我留言。

将本文分享到： QQ空间 新浪微博 人人网 开心网 搜狐微博 MSN 谷歌 [更多](http://www.jiathis.com/share?uid=1501904)

分类: [Efficient C#](http://www.cnblogs.com/luminji/category/268612.html)

[**好文要顶**](javascript:void(0);) [**关注我**](javascript:void(0);) [**收藏该文**](javascript:void(0);) **[http://common.cnblogs.com/images/icon_weibo_24.png](javascript:void(0);)**

[http://pic.cnblogs.com/face/123061/20131127142331.png](http://home.cnblogs.com/u/luminji/)

[Luminji](http://home.cnblogs.com/u/luminji/)  
[关注 - 45](http://home.cnblogs.com/u/luminji/followees)  
[粉丝 - 1669](http://home.cnblogs.com/u/luminji/followers)

荣誉：[推荐博客](http://www.cnblogs.com/expert/)

[+加关注](javascript:void(0);)

8

0

[«](http://www.cnblogs.com/luminji/archive/2011/05/08/2040170.html)上一篇：[改善C#程序的建议7：正确停止线程](http://www.cnblogs.com/luminji/archive/2011/05/08/2040170.html)  
[»](http://www.cnblogs.com/luminji/archive/2011/05/13/2044801.html)下一篇：[改善C#程序的建议9：使用Task代替ThreadPool和Thread](http://www.cnblogs.com/luminji/archive/2011/05/13/2044801.html)

# [改善C#程序的建议9：使用Task代替ThreadPool和Thread](http://www.cnblogs.com/luminji/archive/2011/05/13/2044801.html)

**一：Task的优势**

ThreadPool相比Thread来说具备了很多优势，但是ThreadPool却又存在一些使用上的不方便。比如：

1: ThreadPool不支持线程的取消、完成、失败通知等交互性操作；

2: ThreadPool不支持线程执行的先后次序；

以往，如果开发者要实现上述功能，需要完成很多额外的工作，现在，FCL中提供了一个功能更强大的概念：Task。Task在线程池的基础上进行了优化，并提供了更多的API。在FCL4.0中，如果我们要编写多线程程序，Task显然已经优于传统的方式。

以下是一个简单的任务示例：

[复制代码](javascript:void(0);)

staticvoid Main(string[] args)  
{  
Task t =new Task(() =>  
{  
Console.WriteLine("任务开始工作……");  
//模拟工作过程  
Thread.Sleep(5000);  
});  
t.Start();  
t.ContinueWith((task) =>  
{  
Console.WriteLine("任务完成，完成时候的状态为：");  
Console.WriteLine("IsCanceled={0}\tIsCompleted={1}\tIsFaulted={2}", task.IsCanceled, task.IsCompleted, task.IsFaulted);  
});  
Console.ReadKey();  
}

[复制代码](javascript:void(0);)

**二：Task的完成状态**

任务Task有这样一些属性，让我们查询任务完成时的状态：

1: IsCanceled，因为被取消而完成；

2: IsCompleted，成功完成；

3: IsFaulted，因为发生异常而完成

需要注意的是，任务并没有提供回调事件来通知完成（像BackgroundWorker一样），它通过启用一个新任务的方式来完成类似的功能。ContinueWith方法可以在一个任务完成的时候发起一个新任务，这种方式天然就支持了任务的完成通知：我们可以在新任务中获取原任务的结果值。

       下面是一个稍微复杂一点的例子，同时支持完成通知、取消、获取任务返回值等功能：

[复制代码](javascript:void(0);)

staticvoid Main(string[] args)  
{  
CancellationTokenSource cts =new CancellationTokenSource();  
Task<int> t =new Task<int>(() => Add(cts.Token), cts.Token);  
t.Start();  
t.ContinueWith(TaskEnded);  
//等待按下任意一个键取消任务  
Console.ReadKey();  
cts.Cancel();  
Console.ReadKey();  
}  
  
staticvoid TaskEnded(Task<int> task)  
{  
Console.WriteLine("任务完成，完成时候的状态为：");  
Console.WriteLine("IsCanceled={0}\tIsCompleted={1}\tIsFaulted={2}", task.IsCanceled, task.IsCompleted, task.IsFaulted);  
Console.WriteLine("任务的返回值为：{0}", task.Result);  
}  
  
staticint Add(CancellationToken ct)  
{  
Console.WriteLine("任务开始……");  
int result =0;  
while (!ct.IsCancellationRequested)  
{  
result++;  
Thread.Sleep(1000);  
}  
return result;  
}

[复制代码](javascript:void(0);)

在任务开始后大概3秒钟的时候按下键盘，会得到如下的输出：

任务开始……  
任务完成，完成时候的状态为：  
IsCanceled=False IsCompleted=True IsFaulted=False  
任务的返回值为：3

你也许会奇怪，我们的任务是通过Cancel的方式处理，为什么完成的状态IsCanceled那一栏还是False。这是因为在工作任务中，我们对于IsCancellationRequested进行了业务逻辑上的处理，并没有通过ThrowIfCancellationRequested方法进行处理。如果采用后者的方式，如下：

[复制代码](javascript:void(0);)

staticvoid Main(string[] args)  
{  
CancellationTokenSource cts =new CancellationTokenSource();  
Task<int> t =new Task<int>(() => AddCancleByThrow(cts.Token), cts.Token);  
t.Start();  
t.ContinueWith(TaskEndedByCatch);  
//等待按下任意一个键取消任务  
Console.ReadKey();  
cts.Cancel();  
Console.ReadKey();  
}  
  
staticvoid TaskEndedByCatch(Task<int> task)  
{  
Console.WriteLine("任务完成，完成时候的状态为：");  
Console.WriteLine("IsCanceled={0}\tIsCompleted={1}\tIsFaulted={2}", task.IsCanceled, task.IsCompleted, task.IsFaulted);  
try  
{  
Console.WriteLine("任务的返回值为：{0}", task.Result);  
}  
catch (AggregateException e)  
{  
e.Handle((err) => err is OperationCanceledException);  
}  
}  
  
staticint AddCancleByThrow(CancellationToken ct)  
{  
Console.WriteLine("任务开始……");  
int result =0;  
while (true)  
{  
ct.ThrowIfCancellationRequested();  
result++;  
Thread.Sleep(1000);  
}  
return result;  
}

[复制代码](javascript:void(0);)

那么输出为：

任务开始……  
任务完成，完成时候的状态为：  
IsCanceled=True IsCompleted=True IsFaulted=False

在任务结束求值的方法TaskEndedByCatch中，如果任务是通过ThrowIfCancellationRequested方法结束的，对任务求结果值将会抛出异常OperationCanceledException，而不是得到抛出异常前的结果值。这意味着任务是通过异常的方式被取消掉的，所以可以注意到上面代码的输出中，状态IsCancled为True。

再一次，我们注意到取消是通过异常的方式实现的，而表示任务中发生了异常的IsFaulted状态却还是等于False。这是因为ThrowIfCancellationRequested是协作式取消方式类型CancellationTokenSource的一个方法，CLR进行了特殊的处理。CLR知道这一行程序开发者有意为之的代码，所以不把它看作是一个异常（它被理解为取消）。要得到IsFaulted等于True的状态，我们可以修改While循环，模拟一个异常出来：

[复制代码](javascript:void(0);)

while (true)  
{  
//ct.ThrowIfCancellationRequested();  
if (result ==5)  
{  
thrownew Exception("error");  
}  
result++;  
Thread.Sleep(1000);  
}

[复制代码](javascript:void(0);)

模拟异常后的输出为：

任务开始……  
任务完成，完成时候的状态为：  
IsCanceled=False IsCompleted=True IsFaulted=True

**三：任务工厂**

Task还支持任务工厂的概念。任务工厂支持多个任务之间共享相同的状态，如取消类型CancellationTokenSource就是可以被共享的。通过使用任务工厂，可以同时取消一组任务：

[复制代码](javascript:void(0);)

staticvoid Main(string[] args)  
{  
CancellationTokenSource cts =new CancellationTokenSource();  
//等待按下任意一个键取消任务  
TaskFactory taskFactory =new TaskFactory();  
Task[] tasks =new Task[]  
{  
taskFactory.StartNew(() => Add(cts.Token)),  
taskFactory.StartNew(() => Add(cts.Token)),  
taskFactory.StartNew(() => Add(cts.Token))  
};  
//CancellationToken.None指示TasksEnded不能被取消  
taskFactory.ContinueWhenAll(tasks, TasksEnded, CancellationToken.None);  
Console.ReadKey();  
cts.Cancel();  
Console.ReadKey();  
}  
  
staticvoid TasksEnded(Task[] tasks)  
{  
Console.WriteLine("所有任务已完成！");  
}

[复制代码](javascript:void(0);)

以上代码输出为：

任务开始……  
任务开始……  
任务开始……  
所有任务已完成（取消）！

本建议演示了Task（任务）和TaskFactory（任务工厂）的使用方法。Task甚至进一步优化了后台线程池的调度，加快了线程的处理速度。在FCL4.0时代，使用多线程，我们理应更多地使用Task。

# [改善C#程序的建议10：用Parallel简化Task](http://www.cnblogs.com/luminji/archive/2011/05/19/2050692.html)

在命名空间System.Threading.Tasks下，有一个静态类Parallel简化了在同步状态下的Task的操作。Parallel主要提供了3个有用的方法：For、ForEach、Invoke。

For方法，主要用于处理针对数组元素的并行操作，如下：

[复制代码](javascript:void(0);)

staticvoid Main(string[] args)  
{  
int[] nums =newint[] { 1, 2, 3, 4 };  
Parallel.For(0, nums.Length, (i) =>  
{  
Console.WriteLine("针对数组索引{0}对应的那个元素{1}的一些工作代码……",i, nums[i]);  
});  
Console.ReadKey();  
}

[复制代码](javascript:void(0);)

输出为：

针对数组索引0对应的那个元素1的一些工作代码……  
针对数组索引2对应的那个元素3的一些工作代码……  
针对数组索引1对应的那个元素2的一些工作代码……  
针对数组索引3对应的那个元素4的一些工作代码……

可以看到，工作代码并不按照数组的索引次序进行遍历。显而易见，这是因为我们的遍历是并行的，不是顺序的。所以这里也可以引出一个小建议：如果我们的输出必须是同步的或者说必须是顺序输出的，则不应使用Parallel的方式。

Foreach方法，主要用于处理泛型集合元素的并行操作，如下：

[复制代码](javascript:void(0);)

staticvoid Main(string[] args)  
{  
List<int> nums =new List<int> { 1, 2, 3, 4 };  
Parallel.ForEach(nums, (item) =>  
{  
Console.WriteLine("针对集合元素{0}的一些工作代码……", item);  
});  
Console.ReadKey();  
}

[复制代码](javascript:void(0);)

输出为：

针对集合元素1的一些工作代码……  
针对集合元素4的一些工作代码……  
针对集合元素3的一些工作代码……  
针对集合元素2的一些工作代码……

使用For和Foreach方法，Parallel类型自动为我们分配Task完成针对元素的一些工作。当然我们也可以直接使用Task，但是上面的这种形式，在语法上看上去更简洁了。

Parallel的Invoke方法，则为我们简化了启动一组并行操作，它隐式启动的就是Task。该方法接受Params Action[]参数，如下：

[复制代码](javascript:void(0);)

staticvoid Main(string[] args)  
{  
Parallel.Invoke(() =>  
{  
Console.WriteLine("任务1……");  
},  
() =>  
{  
Console.WriteLine("任务2……");  
},  
() =>  
{  
Console.WriteLine("任务3……");  
});  
Console.ReadKey();  
}

[复制代码](javascript:void(0);)

输出为：

任务2……  
任务3……  
任务1……

同样，由于所有的任务都是并发的，所以它不保证先后次序。