```
        exception = exception.Flatten();
        try
        {
            exception.Handle(innerException =>
            {
                // Rethrowing rather than using
                // if condition on the type.
                ExceptionDispatchInfo.Capture(
                    exception.InnerException)
                    .Throw();
                return true;
            });
        }
        catch(WebException)
        {
            // ...
        }
        catch(IOException )
        {
            // ...
        }
        catch(NotSupportedException )
        {
            // ...
        }
    }
}

private static Task WriteWebRequestSizeAsync(
    string url)
{
    StreamReader reader = null;
    WebRequest webRequest =
         WebRequest.Create(url);

    Task task =
        webRequest.GetResponseAsync()
    .ContinueWith( antecedent =>
    {
        WebResponse response =
           antecedent.Result;

        reader =
            new StreamReader(
                response.GetResponseStream());
        return reader.ReadToEndAsync();
    })
        .Unwrap()
        .ContinueWith(antecedent =>
        {
            if(reader != null) reader.Dispose();
            string text = antecedent.Result;
            Console.WriteLine(
                FormatBytes(text.Length));
        });

        return task;
    }

    // ...
}
```

和代码清单18-13不同，代码清单18-14会在网页下载期间向控制台打印句点符号。所以不是简单地打印几个句点（ 如“……” ），而是在下载文件、从流中读取以及判断大小期间，一直地打印句点。

遗憾的是，异步操作的代价是复杂性的增加。代码中到处都是和TPL相关的代码，读起来比较费劲。不是直接在WebRequest.GetResponseAsync()调用之后就获取StreamReader并调用ReadToEndAsync()，这个异步版本要求使用ContinueWith()语句。 第一个ContinueWith()语句指出WebRequest.GetResponseAsync()之后要执行什么。注意在第一个ContinueWith()表达式中，return语句返回StreamReader.ReadToEndAsync()，后者返回另一个Task。

所以，如果没有Unwrap()调用，第二个ContinueWith()语句中的先驱任务（antecedent）就是一个Task<Task<string>>，光看这个就知道有多复杂了。这种情况必须调用Result两次，一次直接在antecedent上调用，一次在antecedent.Result返回的Task<string>.Result属性上调用，后者会一直阻塞，直至ReadToEnd()操作结束。为了避免复杂的Task<Task<TResult>>结构，可以在调用ContinueWith()之前调用Unwrap()，这样外层的Task就可以脱掉了，而且能恰当地处理任何错误或取消请求。

但是，不仅仅是Task和ContinueWith()在搅局，异常处理使局面变得更复杂。本章前面说过，TPL通常引发AggregateException异常，因为异步操作可能出现多个异常。但是，由于是从ContinueWith()块中调用Result属性，所以工作者线程中也可能引发AggregateException。

如本章前面所述，可用多种方式处理这些异常。

（1）用ContinueWith()方法为返回任务的所有*Async方法都添加延续任务。但这样就无法很流畅地一个接一个写ContinueWith()了。除此之外，还会被迫将错误处理逻辑嵌入控制流中，而不是简单地依赖异常处理。

（2）用try/catch块包围每个委托主体，这样任务中就没有异常会成为未处理异常。遗憾的是，这个方式也不甚理想。首先，有的异常（比如调用antecedent.Result所造成的）会引发AggregateException， 需要对InnerException(s)进行解包才能单独处理这些异常。解包时，要么重新引发以便捕捉特定类型的异常，要么单独用其他catch块（甚至针对同一个类型的多个catch块）条件性地检查异常类型。其次，每个委托主体都需要自己的try/catch处理程序，即使块和块之间的有些异常类型是相同的。最后，Main中的task.Wait()调用仍会引发异常，因为WebRequest.GetResponseAsync()可能引发异常，但没有办法用try/catch块来包围它。因此，没有办法在Main中消除围绕task.Wait()的try/catch块。

（3）在WriteWebRequesSizeAsync()中忽略所有异常处理，只依赖围绕Main的task.Wait()的try/catch块。由于异常必然是AggregateException，所以只捕捉它就好。catch块调用AggregateException.Handle()来处理异常，并用Exception DispatchInfo对象引发每个异常，以确保不会丢失原始的栈跟踪。这些异常随即由预期的异常处理程序捕捉和处理。但要注意的是，处理AggregateException的InnerException(s)之前要先调用AggregateException.Flatten()。这是因为在AggregateException中包装的内部异常也是AggregateException类型（以此类推）。调用Flatten()之后，所有异常都跑到第一级，包含的所有AggregateException都被删除。

如代码清单18-14所示，选项3或许是最好的，因为它在很大程度上使异常处理独立于控制流。虽然不能完全消除错误处理的复杂性，但却在很大程度上防止了在控制流中零散地嵌入异常处理。

代码清单18-14的异步版本和代码清单18-3的同步版本具有几乎完全一样的控制流逻辑，两个版本都试图从服务器下载资源，下载成功就返回结果，失败就检查异常类型来确定对策。然而，很明显异常版本较难阅读、理解和更改。同步版本使用标准的控制流语句，而异步处理被迫创建多个Lambda表达式，以委托的形式表达延续逻辑。

遗憾的是，这还是一个相当简单的例子！想象一下，假定要用循环来重试3次操作（如果失败的话）、要访问多个服务器、要获取资源集合而不是一个资源或者将所有这些功能都集成到一起。用同步代码实现这些功能很简单，但用异步代码来实现就非常复杂。为每个任务显式指定延续，从而将同步方法重写成异步方法，这个过程很快就会变得不堪其扰。

18.5.3 通过async和await实现基于任务的异步模式

幸好，写一个程序来帮助自己完成这些复杂的代码转换也不是太难。C#语言的设计者意识到了这一点，在C# 5.0编译器中添加了这个功能。现在可以使用**基于任务的异步模式**（Task-based Asynchronous Pattern，TAP）将同步程序轻松重写为异步程序；将由C#编译器负责将方法转换成一系列任务延续。代码清单18-15展示了如何将代码清单18-13的方法重写为异步版本，同时不像代码清单18-4那样对主结构进行大手术。

代码清单18-15 使用基于任务的异步模式来实现异步Web请求

```
using System;
using System.IO;
using System.Net;
using System.Linq;
using System.Threading.Tasks;

public class Program
{
  private static async Task WriteWebRequestSizeAsync(
      string url)
  {
      try
      {
          WebRequest webRequest =
              WebRequest.Create(url);
          WebResponse response =
              await webRequest.GetResponseAsync();
      using(StreamReader reader =
          new StreamReader(
              response.GetResponseStream()))
      {
          string text =
              await reader.ReadToEndAsync();
          Console.WriteLine(
              FormatBytes(text.Length));
      }
}
catch(WebException)
{
    // ...
}
catch(IOException ).
{
```

```
            // ...
        }
        catch(NotSupportedException )
        {
            // ...
        }
    }

    public static void Main(string[] args)
    {
        string url = "http://www.IntelliTect.com";
        if(args.Length > 0)
        {
            url = args[0];
        }

        Console.Write(url);

        Task task = WriteWebRequestSizeAsync(url);

        while(!task.Wait(100))
        {
            Console.Write(".");
        }
    }

    // ...

}
```

注意代码清单18-13和代码清单18-15的一些小区别。首先将Web请求功能的主体重构为新方法（WriteWebRequestSizeAsync()），并在方法声明中添加了新的上下文关键字async。用这个关键字修饰的方法必须返回Task、Task<T>或void。在本例中，由于方法主体无数据返回，但我们想把有关异步活动的信息返回给调用者，所以WriteWebRequestSizeAsync()返回Task。注意方法名使用了Async后缀，虽然并非必须，但根据约定，应该用这个后缀标识异步方法。最后，针对和同步方法等价的每个异步版本，都在调用异步版本之前插入了await关键字。

除了这些，代码清单18-13和18-15就没有其他区别了。 方法的异步版本表面上返回和以前一样的数据类型，但实际返回的是一个Task<T>。这不是通过隐式类型转换来实现的。GetResponseAsync()的声明如下：

```
public virtual Task<WebResponse> GetResponseAsync() { ... }
```

在调用点处，我们将返回值赋给WebResponse：

```
WebResponse response = await webRequest.GetResponseAsync()
```

这里的关键在于async上下文关键字，它指示编译器将表达式重写为一个状态机来代表如代码清单18-14所示的全部控制流（以及更多）。

注意，try/catch逻辑也比代码清单18-14好得多。在代码清单18-15中，根本没有必要捕捉AggregateException。catch子句直接捕捉确切的异常类型，不需要对内部异常进行解包。经编译器重写之后，任务中的AggregateExceptionin似乎被当作普通的、以同步方式引发的异常进行处理。但事实上，AggregateException（及其内部异常集合）的工作方式和以前一样，只是在等待任务时，编译器重写的版本会从集合中取出第一个异常并引发它。目的就是使异步代码块尽可能地像同步代码。

为了更好地理解控制流，表18-2展示了每个任务中的控制流。

表18-2 每个任务中的控制流

说明	Main()线程	GetResponseAsync()任务	ReadToEndAsync()任务
1. 执行正常进入Main,并一路执行到第一个Console.Write(url)语句。 2. 调用WriteWebRequestSize-Async()造成控制流正常进入那个方法。 3. WriteWebRequestSizeAsync()中的代码正常执行（仍然在Main()线程上），包括对WebRequest.Create(url)的调用。	`string url =` `  "http://www.IntelliTect.com";` `if(args.Length > 0)` `{` `  url = args[0];` `}` `Console.Write(url);` `Task task =` `WriteWebRequestSizeAsync(url);` `WebRequest webRequest =` `  WebRequest.Create(url);`		
4. 遇到第一个await修饰符生成新Task来执行GetResponseAsync()。假定该方法不是瞬间就执行完毕，控制流将返回Main()，开始执行while循环。 5. 一旦GetResponseAsync()任务完成,同一个任务中执行将继续，隐式地将任务的结果赋给response变量。然后，根据response的值来实例化StreamReader。	`while(!task.Wait(100))` `{` `  Console.Write(".");` `}`	`WebResponse response =` `  await webRequest.GetResponseAsync();` `StreamReader reader =` `  new StreamReader(` `  response.GetResponseStream());`	
6. 遇到另一个await时，创建另一个任务，这次是执行ReadToEndAsync()。（同时Main的while循环将继续执行。） 7. ReadToEndAsync()任务完成后,结果赋给text，在控制台显示它的Length。 8. 最后，task.Wait()返回true，继续执行while之后的代码。			`string text =` `  (await reader.` `ReadToEndAsync());` `Console.WriteLine(` `  FormatBytes(text.` `Length));`

这个表格很好地澄清了以下两个错误观念。

- **错误观念#1：用async关键字修饰的方法一旦调用，就会自动在一个工作者线程上执行。**这绝对是不成立的，方法在调用线程上正常执行。如果方法的实现不等待任何未完成的、可等待的任务，就会在同一个线程上同步地完成。是由方法的实现决定是否启动任何异步工作。仅仅使用async关键字，改变不了方法的代码在哪里执行。此外，从调用者的角度看，对async方法的调用没有任何特别之处，就是一个返回Task的方法，该方法正常调用，最后正常返回指定返回类型的对象。
- **错误观念#2：await关键字造成当前线程阻塞，直到被等待的任务完成。**这也绝对不成立。要阻塞当前线程直到任务完成，应该调用Wait()方法；事实上，Main线程在等待其他任务完成时一直都在做这件事情。但是，while(!task.Wait(100)) { }是和其他任务并发执行的——而不是同步。await关键字对它后面的表达式进行求值，该表达式一般是Task或Task<T>类型，为最终的任务添加延续，然后立即将控制返回给调用者。创建任务将开始异步工作；await关键字意味着开发人员希望在异步工作进行期间，方法的调用者在这个线程上继续执行它的工作。异步工作完成之后的某个时间，从await表达式之后的控制点恢复执行。

事实上，async关键字最主要的作用就是：（1）向阅读代码的人清楚说明它所修饰的方法将自动由编译器重写；（2）告诉编译器方法中的上下文关键字await要被视为异步控制流，不能当成普通的标识符。

18.5.4 异步Lambda

我们知道，转换成委托的Lambda表达式是声明普通方法的一种精简语法。类似地，C# 5.0允许包含await表达式的Lambda表达式转换成委托——为Lambda表达式附加async关键字前缀即可。代码清单18-16重写了代码清单18-15的GetResourceAsync()方法，把它从async方法转换成async Lambda。

代码清单18-16 作为Lambda表达式的异步客户端-服务器交互

```
using System;
using System.IO;
using System.Net;
using System.Linq;
using System.Threading.Tasks;

public class Program
{
    public static void Main(string[] args)
    {
        string url = "http://www.IntelliTect.com";
        if(args.Length > 0)
        {
            url = args[0];
        }

        Console.Write(url);

        Func<string, Task> writeWebRequestSizeAsync =
            async (string webRequestUrl) =>
```

```
        {
            // Error handling ommitted for
            // elucidation.
            WebRequest webRequest =
               WebRequest.Create(url);

            WebResponse response =
                await webRequest.GetResponseAsync();
            using(StreamReader reader =
                new StreamReader(
                    response.GetResponseStream()))
            {
                string text =
                    (await reader.ReadToEndAsync());
                Console.WriteLine(
                    FormatBytes(text.Length));
            }
        };

    Task task = writeWebRequestSizeAsync(url);

    while (!task.Wait(100))
    {
        Console.Write(".");
    }
}

// ...
}
```

async Lambda表达式具有和具名async方法一样的限制。

- async Lambda表达式必须转换成返回类型为void、Task或Task<T>的委托。
- Lambda进行了重写，使return语句成为“Lambda返回的任务已经完成并获得给定结果”的信号。
- Lambda表达式中的执行最初是同步进行的，直到遇到第一个针对“未完成的可等待任务”的await为止。
- await之后的指令作为被调用异步方法所返回的任务的延续而执行。但假如可等待任务已经完成，就以同步方式执行而不是作为延续。
- async Lambda表达式可用await调用（代码清单18-16未演示）。

高级主题：实现自定义异步方法

使用await关键字可以轻松实现异步方法，它依赖其他异步方法（后者依赖更多的异步方法）。但在调用层次结构的某个位置，需要写一个返回Task的“叶”异步方法。例如，假定要用一个异步方法运行命令行程序，最终目标是能访问程序的输出，那么可像下面这样声明它：

```
static public Task<Process> RunProcessAsync(string filename)
```

最简单的实现当然是再次依赖Task.Run()，并调用System.Diagnostics.Process的Start()和WaitForExit()方法。但是，如果被调用的进程有自己的线程集合，在当前进程创建一个额外的线程就是不必要的。为了实现RunProcessAsync()方法并在被调用的进

程结束时回到调用者的同步上下文中，可以依赖一个TaskCompletionSource<T>对象，如代码清单18-17所示。

代码清单18-17 实现自定义异步方法

```
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;
class Program
{
  static public Task<Process> RunProcessAsync(
      string fileName,
      string arguments = null,
      CancellationToken cancellationToken =
          default(CancellationToken))
{
      TaskCompletionSource<Process> taskCS =
          new TaskCompletionSource<Process>();

      Process process = new Process()
      {
          StartInfo = new ProcessStartInfo(fileName)
          {
              UseShellExecute = false,
              Arguments = arguments,
          },
          EnableRaisingEvents = true,
      };

      process.Exited += (sender, localEventArgs) =>
      {
          taskCS.SetResult(process);
      };

      cancellationToken
          .ThrowIfCancellationRequested();

      process.Start();

      cancellationToken.Register(() =>
      {
          process.CloseMainWindow();
      });

      return taskCS.Task;
  }

    // ...
}
```

暂时忽略突出显示的内容，重点关注如何用事件来获得关于进程结束的通知。由于System.Diagnostics.Process包含退出通知，所以登记并在回调方法中调用TaskCompletionSource.SetResult()。上述代码展示了不借助Task.Run()来创建异步方法的一个常用模式。

async方法的另一个重要特点是要求提供取消机制，TAP依赖的是和TPL一样的方法，即需要一个System.Threading.CancellationToken。上述代码突出显示了支持取消所需的代码。这个例子允许在进程开始前取消它并尝试关闭应用程序主窗口（如果有的话）。更激进的方式是调用Process.Kill()，但这可能造成正在执行的程序出问题。

注意是在进程开始后才登记取消事件。这是由于在极少数情况下，可能进程还没有开始就触发了取消。

最后要考虑支持的一个功能是进度更新。代码清单18-18 是包含了这个功能的完整RunProcessAsync()。

代码清单18-18 提供了进度支持的自定义异步方法

```
using System;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;
class Program
{
  static public Task<Process> RunProcessAsync(
      string fileName,
      string arguments = null,
      CancellationToken cancellationToken =
          default(CancellationToken),
      IProgress<ProcessProgressEventArgs> progress =
          null,
      object objectState = null)
  {
      TaskCompletionSource<Process> taskCS =
          new TaskCompletionSource<Process>();

      Process process = new Process()
      {
          StartInfo = new ProcessStartInfo(fileName)
          {
              UseShellExecute = false,
              Arguments = arguments,
              RedirectStandardOutput =
                  progress != null
          },
          EnableRaisingEvents = true,
      };

      process.Exited += (sender, localEventArgs) =>
      {
          taskCS.SetResult(process);
      };

      if(progress != null)
      {
          process.OutputDataReceived +=
              (sender, localEventArgs) =>
          {
              progress.Report(
                  new ProcessProgressEventArgs(
                      localEventArgs.Data,
                      objectState));
          };
      }

      if(cancellationToken.IsCancellationRequested)
      {
          cancellationToken
              .ThrowIfCancellationRequested();
      }

      process.Start();
```

```
        if(progress != null)
        {
            process.BeginOutputReadLine();
        }

        cancellationToken.Register(() =>
        {
            process.CloseMainWindow();
            cancellationToken
                .ThrowIfCancellationRequested();
        });

        return taskCS.Task;
    }

    // ...
}
```

高级主题：等待非Task<T>值

通常，await关键字后面的表达式是Task或Task<T>类型。在迄今为止的await例子中，关键字后面的表达式都是返回Task<T>。从语法的角度看，作用于Task类型的await相当于返回void的表达式。事 实上，编译器连任务是否有结果都不知道，更别说结果的类型了。所以，像这样的表达式相当于调用一个返回void的方法，只能在语句的上下文中使用它。代码清单18-19展示了作为语句表达式使用的await表达式。

代码清单18-19 事件表达式可以作为语句表达式使用

```
async Task<int> DoStuffAsync()
{
  await DoSomethingAsync();
  await DoSomethingElseAsync();
  return await GetAnIntegerAsync() + 1;
}
```

这里假定前两个方法返回Task而不是Task<T>。由于前两个任务没有关联任何结果值，所以等待它们不会产生任何值，表达式必须作为语句使用。第三个任务假定是Task<int>类型，它的值可用于计算DoStuffAsync()返回的任务的值。

这个高级主题以“通常”两个字开头。事实上，关于await所要求的返回类型，规则比单单Task或Task<T>宽泛得多。它只要求类型支持GetAwaiter。该方法生成一个对象，其中包含特定的属性和方法以便由编译器的重写逻辑使用。这使系统能由第三方进行扩展[①]；可以设计自己的、不是基于Task的异步系统，用其他类型来表示异步工作。与此同时，还是能使用await语法。

但要注意，async方法不能返回除void、Task或Task<T>之外的东西——不管方法内部等待的是什么类型。

精准地掌握async方法中发生的事情可能比较难，但相较于异步代码用显式的延续来写Lambda表达式，前者理解起来还是容易多了。注意下面这些要点：

① 根据签名来查找特定方法以实现第三方扩展， 这个技术在另外两个C#功能中也得到了运用。具体地说，LINQ查找Select()和Where()方法来实现select和where上下文关键字，而foreach循环不要求集合实现IEnumerable，只要求有恰当的GetEnumerator()方法。

- 控制抵达await关键字时，后面的表达式会生成一个任务[①]。控制随即返回调用者，在任务异步完成期间，继续做自己的事情。
- 任务完成后的某个时间，控制从await之后的位置恢复。如果等待的任务生成结果，就获取那个结果。出错则引发异常。
- async方法中的return语句造成与方法调用关联的任务变成“已完成”状态。如果return语句有一个值，返回值成为任务的结果。

18.5.5 任务调度器和同步上下文

本章偶尔提到任务调度器及其在为线程高效分配工作时所扮演的角色。从编程的角度看，任务调度器是System.Threading.Tasks.TaskScheduler的实例。该类默认用线程池调度任务，决定如何安全有效地执行它们——何时重用、何时进行资源清理（dispose）以及何时创建额外的线程。

从TaskScheduler派生出一个新类型，即可创建自己的任务调度器，从而对任务调度做出不同的选择。可以获取一个TaskScheduler，使用静态方法FromCurrentSynchronizationContext()将任务调度给当前线程（更准确地说，调度给和当前线程关联的**同步上下文**），而不是调度给不同的工作者线程[②]。

用于执行任务（进而执行延续任务）的同步上下文之所以重要，是因为正在等待的任务会查询同步上下文（如果有的话）才能高效和安全地执行。代码清单18-20（和输出18-7）与代码清单18-5相似，但它在显示消息时还打印线程ID。

代码清单18-20 调用Task.ContinueWith()

```
using System;
using System.Threading.Tasks;

public class Program
{
  public static void Main()
  {
      DisplayStatus("Before");
      Task taskA =
          Task.Run(() =>
              DisplayStatus("Starting..."))
          .ContinueWith( antecedent =>
              DisplayStatus("Continuing A..."));
      Task taskB = taskA.ContinueWith( antecedent =>
          DisplayStatus("Continuing B..."));
      Task taskC = taskA.ContinueWith( antecedent =>
          DisplayStatus("Continuing C..."));
      Task.WaitAll(taskB, taskC);
      DisplayStatus("Finished!");
  }

  private static void DisplayStatus(string message)
  {
      string text =
          string.Format("{0}: {1}",
```

① 从技术上说，是在“高级主题：等待非Task<T>值”中描述的一个可等待类型。
② 例子参考附录C的代码清单C-8。

```
                Thread.CurrentThread.ManagedThreadId,
                message);
            Console.WriteLine(text);
        }
    }
```

输出18–7

```
1: Before
3: Starting...
4: Continuing A...
3: Continuing C...
4: Continuing B...
1: Finished!
```

在输出中，注意事件ID时而改变，时而重复以前的。控制台应用程序的同步上下文（通过SynchronizationContext.Current属性访问）为null——造成由线程池处理线程分配。这就解释了为何在任务和任务之间线程ID会发生变化；有时线程池觉得使用新线程更有效，有时又觉得应该使用现有线程。

幸好，同步上下文会根据应用程序的类型自动设置。例如，如果创建任务的代码在ASP.NET创建的线程中运行，线程将关联AspNetSynchronizationContext类型的一个同步上下文。如果代码在WPF应用程序创建的线程中运行，线程将关联DispatcherSynchronizationContext的一个实例。（控制台应用程序默认没有同步上下文。）由于TPL要查询同步上下文，而同步上下文会随执行环境而变化，所以TPL能调度延续任务，使其在高效和安全的上下文中执行。

要修改代码来利用同步上下文，首先要设置同步上下文，再用async/await关键字使同步上下文得以查询[①]。

也可以定义自己的同步上下文，或者修改现有同步上下文来提升特定情况下的性能，但具体如何做超出了本书的范围。

18.5.6 async/await和Windows UI

同步在UI和Web编程中特别重要。例如在Windows UI的情况下，是由一个消息泵处理鼠标点击/移动等事件消息。另外，UI是单线程的，和任何UI组件（如文本框）的交互都肯定在同一个UI线程中发生。async/await模式的一个重要优势在于它利用了同步上下文，使延续工作（await语句后的工作）总是在调用await语句的那个同步任务上执行。这一点非常重要，因为它避免了要显式切换回UI线程来更新控件。

为了体会这一点，来考虑如代码清单18-21所示的WPF中的按钮点击UI事件。

代码清单18–21 WPF中的同步高延迟调用

```
using System;

private void PingButton_Click(
  object sender, RoutedEventArgs e)
{
  StatusLabel.Content = "Pinging…";
```

① 要了解如何设置线程的同步上下文，以及如何使用任务调度器将任务调度给那个线程，请参考参考附录C的代码清单C-8。

```
    Ping ping = new Ping();
    PingReply pingReply =
        ping.Send("www.IntelliTect.com");
    StatusLabel.Content = pingReply.Status.ToString();
}
```

假定StatusLabel是一个WPF System.Windows.Label控件，PingButton_Click()事件处理程序更新Content属性两次。合理的假设是第一个“Pinging…”会一直显示，直到Ping.Send()返回。随即，标签再次更新，显示Send()的应答。但正如WPF经验人士知道的那样，实情并非如此。相反，是一条消息被发布到Windows消息泵，要求用“Pinging…”更新内容，但由于UI线程正忙于执行PingButton_Click()方法，所以Windows消息泵不会得到处理。等UI线程有空检查Windows消息泵时，第二个Content属性更新请求又加入队列，造成用户只看到最后一个应答状态。

为了用TAP修正这个问题，要像代码清单18-22突出显示的那样修改代码。

代码清单18-22 WPF中使用await的同步高延迟调用

```
using System;
async private void PingButton_Click(
  object sender, RoutedEventArgs e)
{
  StatusLabel.Content = "Pinging…";
  Ping ping = new Ping();
  PingReply pingReply =
      await ping.SendPingAsync("www.IntelliTect.com");
  StatusLabel.Content = pingReply.Status.ToString();
}
```

像这样修改有两方面的好处。第一，ping调用的异步本质解放了调用者线程，使其能返回Windows消息泵调用者的同步上下文，处理对StatusLabel.Content的更新，向用户显示“Pinging…”。其次，在等待ping.SendTaskAsync()完成期间，它总是在和调用者相同的同步上下文中执行。同步上下文特别适合WPF，它是单线程的，所以总是返回到同一个线程——UI线程。换言之，TPL不是立即执行延续任务，而是查询同步上下文，将关于延续工作的消息发送给消息泵。第二，UI线程监视消息泵，获得延续工作消息后，它调用await调用之后的代码。（结果是延续代码在和处理消息泵的调用者一样的线程上调用。）

TAP语言模式内建了一个关键的代码可读性增强。注意在代码清单18-22中，pingReply.Status调用自然地出现在await之后，清楚地指明它会紧接着上一个语句执行。然而，如果真要自己从头写，恐怕代码是许多人看不懂的。

18.5.7 await操作符

方法中的await数量不限。事实上，它们并非只能一个接一个地写。相反，可以将await放到循环中连续处理，从而遵循一个自然的控制流。来看看代码清单18-23的例子。

代码清单18-23 迭代await操作

```
using System;
async private void PingButton_Click(
  object sender, RoutedEventArgs e)
{
```

```
List<string> urls = new List<string>()
    {
        "www.habitat-spokane.org",
        "www.partnersintl.org",
        "www.iassist.org",
        "www.fh.org",
        "www.worldvision.org"
    };
IPStatus status;

Func<string, Task<IPStatus>> func =
    async (localUrl) =>
    {
        Ping ping = new Ping();
        PingReply pingReply =
            await ping.SendPingAsync(localUrl);
        return pingReply.Status;
    };

StatusLabel.Content = "Pinging…";

foreach(string url in urls)
{
    status = await func(url);
    StatusLabel.Content =
        string.Format("{0}: {1} ({2})",
            url, status.ToString(),
            Thread.CurrentThread.ManagedThreadId);
}
}
```

注意，无论将await放到循环中，还是单独地写，它们都会连续地、一个接一个地执行，而且顺序和从调用线程中调用的顺序一样。底层实现是用语义上等价于Task.ContinueWith()的方式把它们串接到一起，只是await操作符之间的代码全都在调用者的同步上下文中执行。

从UI中支持TAP，这是TAP的关键应用情形之一。另一个情形是在服务器上，来自客户端的请求要查询整个表来获取特定数据。由于查询数据可能相当花时间，所以应该创建新线程，而不是使用线程池数量有限的线程。但这个方式的问题在于，数据库查询完全在另一台机器上执行。由于线程一般都不活动，所以没必要阻塞整个线程。

总之，TAP的目的是解决以下关键问题。

- 需要在不阻塞UI线程的前提下，允许进行长时间运行的活动。
- 为非CPU密集型的工作创建新线程（或Task）代价相当高，因为线程唯一做的事情就是傻等活动完成。
- 活动完成时（不管是使用新线程还是通过回调），经常都需要执行一次线程同步上下文切换，以返回当初发起活动的调用者。
- TAP提供了一个崭新的模式，同时支持CPU密集型和非CPU密集型的异步调用。所有.NET语言都支持这种模式。

18.6 并行迭代

来考虑以下for循环语句以及相关代码（代码清单18-24和输出18-8）。它调用一个方法来计算pi的一个区段（section）。方法的第一个参数是要计算的位数（BatchSize），第二个参数是要计算的起始位（i * BatchSize）。实际如何计算跟当前的讨论关系不大。这

个计算重点在于，它是“极度可并行”（embarrassingly parallelizable）。也就是说，很容易将大型任务（比如计算pi的100万位）分解成任意数量的小任务，而且所有小任务都可并行运行。这种形式的计算通过并行来进行最容易不过了。

代码清单18-24 for循环同步计算pi的各个区段

```
using System;
using AddisonWesley.Michaelis.EssentialCSharp.Shared;

class Program
{
  const int TotalDigits = 100;
  const int BatchSize = 10;

  static void Main()
  {
      string pi = null;
      int iterations = TotalDigits / BatchSize;
      for(int i = 0; i < iterations; i++)
      {
          pi += PiCalculator.Calculate(
              BatchSize, i * BatchSize);
      }

      Console.WriteLine(pi);
  }
}
```

```
using System;

class PiCalculator
{
  public static string Calculate(
      int digits, int startingAt)
  {
     // ...
  }

   // ...
}
```

输出18-8

```
>3.14159265358979323846264338327950288419716939937510582097494459230781640628620
99862803482534211706798214808651328230664709384460955058223172535940812848111745028
41027019385211055596446229489549303819644288109756659334461284756482337867831652712
01909145648566923460348610454326648213393607260249141273724587006606315588174881520
92096282925409171536436789259036001133053054882046652138414695194151160943305727
03657595919530921861173819326117931051185480744623799627495673518857527248912279
3818301194912
```

for循环同步且顺序地执行每一次迭代。然而，由于pi的算法是将pi的计算分解成独立的部分，所以不需要顺序完成每一部分的计算——只要计算结果顺序连接即可。所以，我们希望不同的迭代能同时进行，每个处理器都负责一个迭代，并和正在执行其他迭代的处理器并行执行这个迭代。如果迭代能同时执行，那么执行时间将依据处理器的数量成比例地减少。

TPL提供了一个辅助方法Parallel.For()来做这件事情。代码清单18-25展示了如何修改顺序的、单线程的程序来使用这种并行for功能。

代码清单18-25 for循环分区段、并行地计算pi

```
using System;
using System.Threading.Tasks;
using AddisonWesley.Michaelis.EssentialCSharp.Shared;

// ...

class Program
{
  static void Main()
  {
        string pi = null;
        int iterations = TotalDigits / BatchSize;
        string[] sections = new string[iterations];
        Parallel.For(0, iterations, (i) =>
        {
            sections[i] = PiCalculator.Calculate(
                BatchSize, i * BatchSize);
        });
        pi = string.Join("", sections);
        Console.WriteLine(pi);
}
```

代码清单18-25的输出和输出18-8一样，但执行速度快多了（前提是有多个CPU；如果没有，速度可能更慢）。Parallel.For() API看起来和标准for循环相似。第一个参数是fromInclusive值，第二个是toExclusive值，最后一个是要作为循环主体执行的Action<int>。为操作使用表达式Lambda，代码看起来和for循环差不多，只是现在每一次迭代都可以并行执行。和for循环一样，除非迭代都已完成，否则Parallel.For()调用不会结束。也就是说，当执行到string.Join()语句时，pi的所有区段都已经计算好了。

要注意的一个重点是，将pi的各个区段合并起来的代码不再出现在迭代（action）的内部。由于pi的区段计算极有可能不是顺序完成的，所以假如每完成一次迭代，就连接一个区段，极有可能造成区段顺序的紊乱。即使顺序不是问题，仍有可能遇到竞态条件，因为+=操作符不是原子性的。为了解决这两个问题，pi的每个区段都要存储到一个数组中，而且不能有两个或多个迭代同时访问一个数组元素的情况。只有在pi的所有区段都计算好之后，string.Join()才把它们合并到一起。换言之，要将区段的连接操作推迟到Parallel.For()循环结束之后。这就避免了由尚未计算好的区段或者顺序错误的区段造成的竞态条件。

TPL使用和任务调度一样的线程池技术来确保并行循环的良好性能，即确保CPU不被过度调度或者调度不足。

规范

要在很容易将一个计算分解成大量相互独立的、处理器受限的小计算，而且这些小计算能在任何线程上以任何顺序执行时，使用并行循环。

TPL还提供了foreach循环的并行版本，如代码清单18-26所示。

代码清单18-26 foreach循环的并行执行

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;

class Program
{
    // ...
    static void EncryptFiles(
        string directoryPath, string searchPattern)
    {
        IEnumerable<string> files = Directory.GetFiles(
            directoryPath, searchPattern,
            SearchOption.AllDirectories);

        Parallel.ForEach(files, (fileName) =>
        {
            Encrypt(fileName);
        });
    }
    // ...
}
```

这个例子调用一个方法来并行加密files集合中的每个文件。TPL会判断同时执行多少个线程效率最高。

高级主题：TPL如何调整自己的性能

TPL的默认调度器面向的是线程池，这造成要进行大量试探以确保任何时候执行的都是正确数量的线程。它采用的两种试探法是爬山（hill climbing）和工作窃取（work stealing）。

爬山算法要求创建线程来运行任务，监视任务性能来找出添加线程使性能不升反降的点。一旦找到这个点，线程数可以降回保持最佳性能的数量。

TPL不将等待执行的“顶级”任务和特定线程关联。但是，如果任务在创建另一个任务的线程上运行，新创建的任务自动和该线程关联。新的“子”任务最终被调度运行时，它通常在创建它的那个任务的线程上运行。工作窃取算法能识别工作量特别大或特别小的线程，任务太少的线程有时会从任务太多的线程上“窃取”一些尚未执行的任务。

这些算法的关键作用是使TPL能动态调整自己的性能来缓解处理器被过度调度和调度不足的情况，从而在可用的处理器之间平衡工作量。

TPL通常能很好地调整自己的性能，但也可以帮它更上一层楼。例如，之前在“长时间运行的计算”一节中，就指定了TPL的TaskCreationOptions.LongRunning选项。还可显式告诉任务调度器一个并行循环的最佳线程数是多少，详情参见“高级主题：并行循环选项”。

初学者主题：使用AggregateException进行并行异常处理

之前说过，TPL在一个AggregateException中捕捉和保存与任务关联的异常，因为一个给定的任务可能从它的多个子任务获取多个异常。并行循环的情况也是如此，每次迭代都可能产生异常，同样要用集合异常收集这些异常，如代码清单18-27和输出18-9所示。

代码清单18-27 在进行并行迭代时如何处理未处理的异常

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

class Program
{
  // ...
  static void EncryptFiles(
      string directoryPath, string searchPattern)
  {
      IEnumerable<string> files = Directory.GetFiles(
          directoryPath, searchPattern,
          SearchOption.AllDirectories);
      try
      {
          Parallel.ForEach(files, (fileName) =>
          {
              Encrypt(fileName);
          });
      }
      catch(AggregateException exception)
      {
          Console.WriteLine(
              "ERROR: {0}:",
              exception.GetType().Name);
          foreach(Exception item in
              exception.InnerExceptions)
          {
              Console.WriteLine("  {0} - {1}",
                  item.GetType().Name, item.Message);
          }
      }
  }
  // ...
}
```

输出18-9

```
ERROR: AggregateException:
  UnauthorizedAccessException - Attempted to perform an unauthorized
operation.
  UnauthorizedAccessException - Attempted to perform an unauthorized
operation.
  UnauthorizedAccessException - Attempted to perform an unauthorized
operation.
```

> 输出18-9表明在执行Parallel.ForEach<T>(...)循环期间发生了3个异常。但在代码中，只有一个catch块在捕捉System.AggregationException。几个UnauthorizedAccessException从AggregationException的InnerExceptions属性获取。Parallel.ForEach<T>()循环的每一次迭代都可能引发异常，所以由方法调用引发的System.AggregationException会在其InnerExceptions属性中包含每个这样的异常。

取消并行循环

任务需要显式调用才能阻塞直至完成。而并行循环以并行方式执行迭代，但除非整个并行循环完成（全部迭代都完成），否则它本身不会返回。所以，为了取消并行循环，调用取消请求的那个线程通常不能是正在执行并行循环的那个线程。代码清单18-28使用Task.Run()来调用Parallel.ForEach<T>()。采取这种方式，不仅查询以并行方式执行，而且还异步执行，允许代码提示用户“Push ENTER to exit”（按ENTER键退出）。

代码清单18-28 取消并行循环

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO
using System.Threading;
using System.Threading.Tasks;

public class Program
{
    // ...

  static void EncryptFiles(
      string directoryPath, string searchPattern)
  {

      string stars =
          "*".PadRight(Console.WindowWidth-1, '*');

      IEnumerable<string> files = Directory.GetFiles(
         directoryPath, searchPattern,
         SearchOption.AllDirectories);

      CancellationTokenSource cts =
          new CancellationTokenSource();
      ParallelOptions parallelOptions =
          new ParallelOptions
          { CancellationToken = cts.Token };
cts.Token.Register(
    () => Console.WriteLine("Cancelling..."));

Console.WriteLine("Push ENTER to exit.");

// Use Task.Factory.StartNew<string>() for
// TPL prior to .NET 4.5
Task task = Task.Run(() =>
    {
        try
        {
            Parallel.ForEach(
                files, parallelOptions,
```

```
                (fileName, loopState) =>
                    {
                        Encrypt(fileName);
                    });
        }
        catch(OperationCanceledException){}
    });

// Wait for the user's input
Console.Read();

        // Cancel the query
        cts.Cancel();
        Console.Write(stars);
        task.Wait();
    }
}
```

并行循环使用了和任务一样的取消标志模式。从一个CancellationTokenSource获取的标志通过调用ForEach()方法的一个重载版本与并行循环关联。该重载版本要获取一个ParallelOptions类型的参数（该对象包含取消标志）。

注意，如果取消并行循环操作，尚未开始的任何迭代都会通过检查IsCancellationRequested属性而被禁止开始。当前正在执行的迭代会运行至各自的终止点。除此之外，即使是在所有迭代都结束之后，调用Cancel()仍会造成登记的取消事件（通过cts.Token.Register()）执行。

另外，ForEach()方法要知道循环是否被取消，唯一的途径就是通过OperationCanceledException。由于本例的取消是意料之中的，所以异常会被捕捉并忽略，允许应用程序在退出前显示“Canceling ...”，后跟一行星号。

高级主题：并行循环选项

虽然比较少见，但也许能通过Parallel.For()和Parallel.ForEach<T>()循环重载版本的ParallelOptions参数来控制最大并行度（即调度同时运行的线程数）。在一些特定的情况下，开发人员对具体的算法或情况有更好的了解。这时就可以考虑更改最大并行度。这些情况包括：

- 有时想禁止并行以简化调试或分析。这时可将最大并行度设为1，确保循环迭代不会并发运行。
- 有时知道算法的性能不能超过一个特定的上限——例如，算法受其他硬件条件的限制，比如可用的USB端口的数目。
- 有时迭代主体长时间（以分钟或小时计）阻塞。线程池区分不了长时间运行的迭代和被阻塞的操作，所以可能引入许多新线程，所有这些线程都被for循环使用。随着时间的推移，线程可能越来越多，造成进程中出现巨量的线程。

为了控制最大并行度，请使用ParallelOptions对象的MaxDegreeOfParallelism属性。

还可使用ParallelOptions对象的TaskScheduler属性，用自定义任务调度器来调度与每个迭代关联的任务。例如，假定当前用一个异步事件处理程序来响应用户点击“Next”按钮的操作。如果用户连续点击按钮几次，可以考虑使用自定义的任务调度器来优先最近创建的任务，而不是优先等待时间最久的任务。（因为用户或许只想看他请

求的最后一个屏幕。）利用任务调度器，可以指定一个任务如何相对于其他任务执行。

ParallelOptions对象还有一个CancellationToken属性，它提供了一个机制和不应继续迭代的循环进行通信。另外，迭代主体可监视取消标志，以判断是否需要提早从迭代中退出。

高级主题：中断并行循环

和标准for循环相似，Parallel.For循环也允许中断（break）循环，不开始后续的任何迭代。但在并行for的执行上下文中，中断循环意味着不要开始当前迭代之后的新迭代，当前正在执行的迭代还是会继续运行直至完成的。

要中断并行循环，可以提供一个取消标志，并在另一个线程上取消它。这已经在前面的高级主题中描述过了。还可使用Parallel.For()方法的一个重载版本，它的主体委托获取两个参数：索引和一个ParallelLoopState对象。如果一个迭代想要“中断”循环，可以在传给委托的循环状态对象上调用Break()或Stop()方法。Break()指出索引值比当前值大的迭代都不需要开始了。Stop()则指出任何未开始的迭代都不要开始了。

例如，假定一个Parallel.For()循环并行执行10个迭代，其中一些比另一些运行得更快，任务调度器不保证任何特定的顺序。假定第一个迭代完成，迭代3，5，7和9正在4个不同的线程上运行。这时，迭代5和7都调用了Break()。在这种情况下，迭代6和8永远不会开始，但迭代2和4仍然会得到调度并开始运行。与此同时，迭代3和9会运行完成，因为在中断发生时它们已经在运行了。

Parallel.For()和Parallel.ForEach<T>()方法返回对一个ParallelLoopResult对象的引用，对象中包含有关循环期间所发生事情的有用信息。这个结果对象提供以下属性。

- IsCompleted：返回一个Boolean，指出是否所有迭代都已开始。
- LowestBreakIteration：指出执行了一个中断的、索引最低的迭代。值是long?类型；null表明没有遇到break语句。

回到10次迭代的例子，IsCompleted属性会返回false，LowestBreakIteration会返回5。

18.7 并行执行LINQ查询

就像可以使用Parallel.For()并行执行循环，还可以使用并行LINQ API（简称PLINQ）并行执行LINQ查询。代码清单18-29演示了简单的非并行LINQ表达式，代码清单18-30修改它来并行运行。

代码清单18-29 LINQ Select()

```
using System.Collections.Generic;
using System.Linq;

class Cryptographer
{
  // ...
  public List<string>
    Encrypt(IEnumerable<string> data)
  {
      return data.Select(
```

```
            item => Encrypt(item)).ToList();
    }
    // ...
}
```

在代码清单18-29中，LINQ查询用标准Select()查询操作符加密一个字符串序列中的每个字符串，将结果序列转换成列表。这看起来像是一个“极度并行”操作，每个操作都可能是处理器受限的、高延迟的操作，可交由另一个CPU上的工作者线程执行。

代码清单18-30展示了如何修改代码清单19-29，以并行方式加密字符串。

代码清单18-30 并行LINQ Select()

```
using System.Linq;

class Cryptographer
{
  // ...
  public List<string>
    Encrypt (IEnumerable<string> data)
  {
      return data.AsParallel().Select(
          item => Encrypt(item)).ToList();
  }
  // ...
}
```

如代码清单18-30所示，要实现并行支持，只需做很小的更改。只需使用.NET Framework 4引入的标准查询操作符AsParallel()，它是由静态类System.Linq.ParallelEnumerable提供的。这个简单的扩展方法告诉“运行时”可以并行执行查询。如果计算机有多个处理器，查询的完成速度将快很多。

System.Linq.ParallelEnumerable包含由System.Linq.Enumerable提供的操作符的一个超集。所有常用查询操作符的性能都提高了，包括用于排序、筛选（Where()）、投射（Select()）、连接、分组和聚合的操作符。代码清单18-31展示了如何进行并行排序。

代码清单18-31 使用标准查询操作符的并行LINQ

```
// ...
      OrderedParallelQuery<string> parallelGroups =
          data.AsParallel().OrderBy(item => item);

      // Show the total count of items still
      // matches the original count
      System.Diagnostics.Trace.Assert(
          data.Count == parallelGroups.Sum(
              item => item.Count()));
// ...
```

如代码清单18-31所示，为了调用并行版本，只需调用AsParallel()扩展方法。注意，标准查询操作符的并行版本返回的结果类型是ParallelQuery<T>或OrderedParallelQuery<T>，它告诉编译器应继续使用标准查询操作的并行版本（如果可用的话）。

由于查询表达式不过是查询方法调用的“语法糖”，所以也可为表达式形式使用AsParallel()。代码清单18-32展示了如何使用查询表达式语法来并行执行一个分组操作。

代码清单18-32 用查询表达式执行并行LINQ

```
// ...
    ParallelQuery<IGrouping<char, string>> parallelGroups;
    parallelGroups =
        from text in data.AsParallel()
         orderby text
         group text by text[0];

    // Show the total count of items still
    // matches the original count
    System.Diagnostics.Trace.Assert(
        data.Count == parallelGroups.Sum(
            item => item.Count()));
// ...
```

从前面的例子可以看出，很容易就可以转换查询或者循环迭代使其并行执行。但有一点需要注意。如下一章要深入讨论的那样，务必小心不要让多个线程不恰当地同时访问并修改相同的内存。这样做会造成竞态条件。

本章前面说过，Parallel.For()和Parallel.ForEach<T>()方法会收集并行迭代期间引发的所有异常，并引发包含所有原始异常的一个集合异常。 PLINQ操作也可能因为完全相同的原因而返回多个异常（对每个元素并行执行查询逻辑时，在每个元素上运行的代码可能独立引发异常）。毫不奇怪，PLINQ处理这种情况的机制与并行循环和TPL是一样的：并行查询期间引发的异常可以通过AggregateException的InnerExceptions属性访问。因此，将PLINQ查询包装到try/catch块中，并捕捉System.AggregateException类型的异常，就可以成功地处理每一次迭代中任何未处理的异常。

取消PLINQ查询

毫不奇怪，PLINQ查询也支持取消请求模式。代码清单18-33（和输出18-10）展示了一个例子。和并行循环一样，取消的PLINQ查询会引发System.OperationCanceledException，而且PLINQ查询是在调用线程上执行的同步操作。因此，常用的技术是将并行查询包装到在另一个线程上运行的任务中，使当前线程能在必要时取消它——和代码清单18-28的解决方案一样。

代码清单18-33 取消PLINQ查询

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

public class Program
{

  public static List<string> ParallelEncrypt(
      List<string> data,
      CancellationToken cancellationToken)
  {
```

```
    return data.AsParallel().WithCancellation(
        cancellationToken).Select(
            (item) => Encrypt(item)).ToList();
}

public static void Main()
{
    List<string> data = Utility.GetData(1000000).ToList();

    CancellationTokenSource cts =
        new CancellationTokenSource();

    Console.WriteLine("Push ENTER to exit.");

    // Use Task.Factory.StartNew<string>() for
    // TPL prior to .NET 4.5
    Task task = Task.Run(() =>
        {
            data = ParallelEncrypt(data, cts.Token);
        }, cts.Token);

    // Wait for the user's input
    Console.Read();

    cts.Cancel();
    Console.Write(stars);
    try{ task.Wait(); }
        catch(AggregateException){}
    }

    // ...
}
```

输出18-10

```
ERROR: The operation was canceled.
```

和并行循环/任务一样，取消PLINQ查询需要一个CancellationToken，它由CancellationTokenSource.Token属性提供。然而，不是重载每一个PLINQ查询来支持取消标志，而是IEnumerable的AsParallel()方法返回的ParallelQuery<T>对象包含一个WithCancellation()扩展方法，它获取一个CancellationToken参数。因此，在CancellationTokenSource对象上调用Cancel()就会请求并行查询取消——因为它会检查CancellationToken的IsCancellationRequested属性。

如前所述，取消PLINQ查询会引发异常而不是返回完整结果。因此，为了处理可能取消的PLINQ查询，一个常用的技术是用try块包装查询并捕捉OperationCanceledException。第二个常用的技术是将CancellationToken传给ParallelEncrypt()（如代码清单18-33所示），并作为Run()的第二个参数传递。这会造成task.Wait()引发一个AggregateException，它的InnerException属性会被设为一个TaskCanceledException。然后就可以捕捉集合异常，和捕捉并行操作的其他任何异常一样。

18.8 小结

本章首先讨论了多线程程序的基本组成部分：Thread类，它代表程序中独立的“控制点”；以及ThreadPool，它能将线程高效地分配给多个CPU。然而，这些 API都很低级，难以直接操纵。.NET Framework 4.0提供了并行扩展库，其中包括Task Parallel Library（TPL）和Parallel LINQ（PLINQ）。两者都提供了新的API方便你创建和调度Task对象所代表的工作单元、使用Parallel.For()和Parallel.ForEach()并行执行循环以及使用AsParallel()自动执行并行LINQ查询。

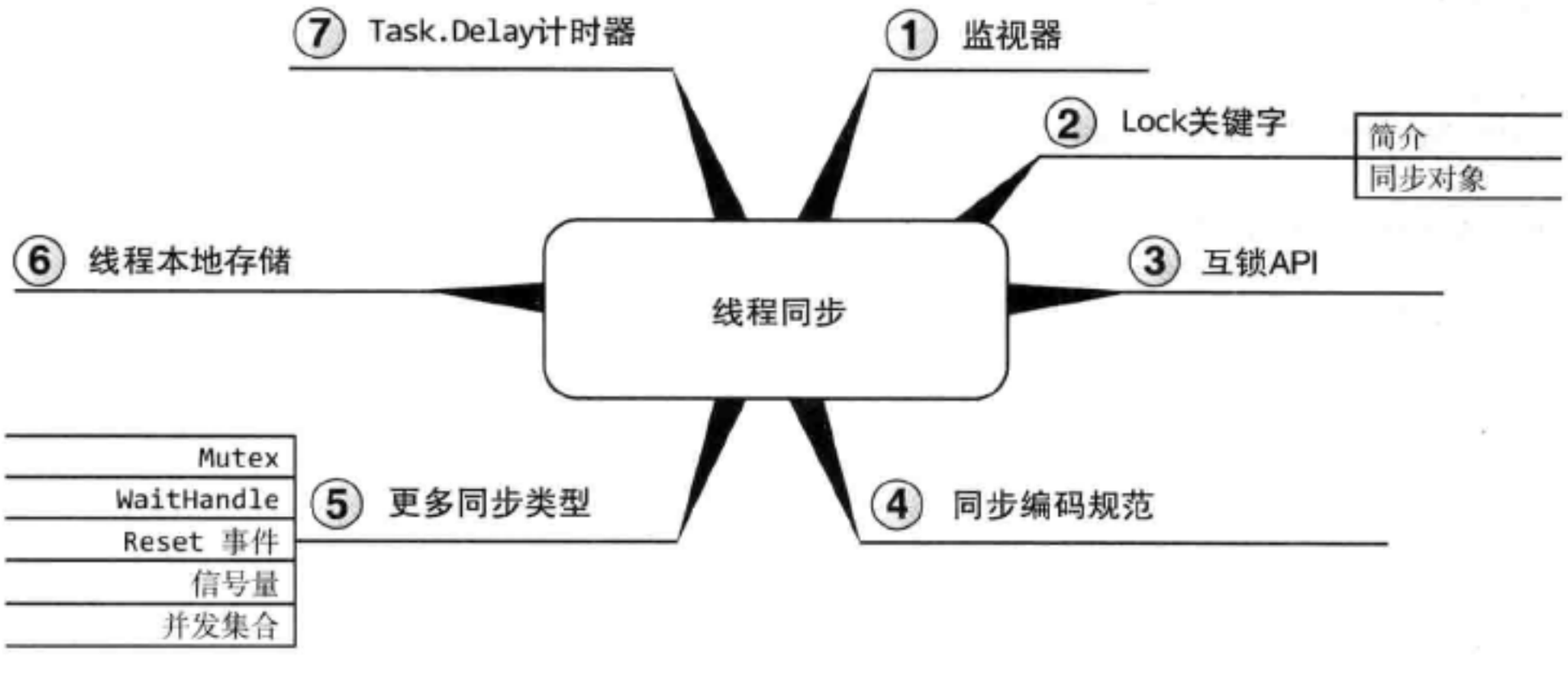
本章还讨论了C# 5.0如何自动重写程序来管理延续的“连接”，将较小的任务合并成较大的任务。

本章开头简单提及了编写多线程程序时面临的困难：原子性、死锁以及为多线程程序带来不确定性和错误行为的其他“竞态条件”，指出避免这些问题的标准方式是小心地写代码，用“锁”来同步以共享资源的访问，这将是下一章的主题。

第19章

线程同步

上一章讨论了使用TPL和PLINQ进行多线程编程的细节。然而，我们刻意避开了线程同步的主题。它的作用是在避免死锁的同时防止出现竞态条件。线程同步是本章的主题。



先来看一个访问共享数据时不进行线程同步的多线程例子。这会造成竞态条件，数据完整性会被破坏。从这个例子出发，我们解释了为何需要线程同步。然后，介绍了进行线程同步的各种机制和最佳实践。

在本书前几版中，本章还用大量篇幅讨论了额外的多线程处理模式，并讨论了各种计时器回调机制。但由于现在有了async/await模式，这些机制基本都被替换掉了，除非要针对C# 5.0/.NET 4.5之前的框架进行编程。为了方便查阅，这些内容被转移到了附录C和附录D中。

注意，本章使用的都是TPL，所以这些例子在.NET Framework 4之前的版本中都无法编译。但是，除非特别标识为.NET Framework 4 API，否则限定.NET Framework 4唯一的原因

是使用了System.Threading.Tasks.Task类来执行异步操作。所以，只需修改代码来实例化一个System.Threading.Thread，再调用Thread.Join()来等待线程执行，本章中的大多数例子就都可以在早期的.NET Framework 上编译了。

不过，本章用来开始任务的API是.NET4.5特有的System.Threading.Tasks.Task.Run()。如上一章所述，该方法比System.Threading.Tasks.Task.Factory.StartNew()好，因为它更简单，而且能满足大多数情况的需要。如果必须使用.NET 4，请将Task.Run()改为Task.Factory.StartNew()，其他任何地方都不需要修改。（正是因为只用了这一个方法，所以本章不专门强调代码是“.NET 4.5专用”的。）

除此之外，微软发布了用于.NET的Reactive Extensions（Rx），它是一个独立的下载，能在.NET 3.5中增加对TPL和PLINQ的支持[①]。这个框架还包含了本章要介绍的并发和同步类型。因此，通过Rx和对System.Threading.dll程序集的引用，在.NET Framework 3.5中也能使用本章那些依赖于Task的代码清单（或者引入了C# 4.0同步类的代码清单）。

总之，本章的大多数例子只要求少量修改即可支持.NET 4.5之前的版本—要么使用Task.Factory.StartNew()，要么依赖Thread.Start()（如果TPL不可用的话）。

19.1 为什么要同步

运行一个新线程是相当简单的编程任务。多线程编程的复杂性在于识别多个线程能同时访问的数据。程序必须对这种数据进行同步，通过防止同时访问来实现“安全”。来看看代码清单19-1的例子。

代码清单19-1 未同步的状态

```
using System;
using System.Threading.Tasks;

public class Program
{
    const int _Total = int.MaxValue;
    static long _Count = 0;

    public static void Main()
    {
        // Use Task.Factory.StartNew for .NET 4.0
        Task task = Task.Run(()=>Decrement());

        // Increment
        for(int i = 0; i < _Total; i++)
        {
            _Count++;
        }

        task.Wait();
        Console.WriteLine("Count = {0}", _Count);
    }

    static void Decrement()
    {
```

① 参见http://bit.ly/Rx3point5。

```
            // Decrement
            for(int i = 0; i < _Total; i++)
            {
                _Count--;
            }
        }
    }
```

输出19-1展示了代码清单19-1的一个可能的输出结果。

输出19–1

```
Count = 113449949
```

注意，代码清单19-1的输出不是0。如果Decrement()是直接（顺序）调用的，输出就是0。然而，异步调用Decrement()会发生竞态条件，因为_Count++和_Count--语句中单独的步骤会发生交错（第18章开头的“初学者主题： 多线程术语”讲过，一个C#语句可能涉及好几个步骤。）。来看看表19-1的示例执行情况。

表19-1展示了并行执行的情况。从一列中的指令切换到另一列，就会发生线程上下文切换。一行结束之后的_Count值在最后一列中显示。在这个示例执行中，_Count++执行了两次，而_Count--只执行了一次。然而，最终的_Count值是0，而不是1。将结果复制回_Count相当于取消同一个线程自从读取_Count以来对_Count值进行的任何修改。

表19–1 示范伪代码执行

Main线程	Decrement线程	-Count
...	...	...
从_Count中复制值0		0
复制的值（0）递增，结果值是1		0
将结果值（1）复制到_Count中		1
从_Count中复制值1		1
	从_Count中复制值1	1
复制的值（1）递增，结果值是2		1
将结果值（2）复制到_Count中		2
	复制的值（1）递减，结果值是0	2
	将结果值（0）复制到_Count中	0
...	...	...

代码清单19-1的问题在于它造成了一个竞态条件。多个线程同时访问相同的数据元素时，就会出现这个情形。正如这个示例执行过程展示的那样，多个线程同时访问相同的数据元素，可能破坏数据的完整性，即使是在一台单处理器的电脑上。为了解决这个问题，代码必须围绕数据进行同步。若能同步多个线程对代码或数据的并发访问，就说这些代码和数据是**线程安全**的。

关于变量读写的原子性，有一个重点需要注意。假如类型的大小不超过一个本机（指针大小的）整数，“运行时”就保证该类型不会被部分性地读取或写入。所以，64位操作系统保证能够原子性地读写一个long（64位）。然而，128位变量（比如decimal）的读写

就不保证是原子性的。所以，通过写操作来更改一个decimal变量时，可能会在仅仅复制了32位之后被打断，造成以后读取一个不正确的值，这称为一次torn read（被撕裂的读取）。

初学者主题：多个线程和局部变量

注意，局部变量没有必要同步。局部变量加载到栈上，而每个线程都有自己的逻辑栈。所以，针对每个方法调用，每个局部变量都有自己的实例。在不同的方法调用之间，局部变量默认是不共享的。因此，它们在多个线程之间也是不共享的。

然而，这并不是说局部变量完全没有并发性问题，因为代码可能轻易向多个线程公开局部变量[①]。例如，在迭代之间共享局部变量的并行for循环就会公开变量，使其可能被并发访问，从而造成一个竞态条件，如代码清单19-2所示。

代码清单19-2 未同步的局部变量

```
using System;
using System.Threading.Tasks;

public class Program
{
  public static void Main()
  {
      int x = 0;
      Parallel.For(0, int.MaxValue, i =>
          {
              x++;
              x--;
          });
      Console.WriteLine("Count = {0}", x);
  }
}
```

这个例子在一个并行for循环中访问x（一个局部变量），使多个线程可能同时修改它，造成和代码清单19-1非常相似的竞态条件。即使x新增和递减相同的次数，输出也不一定是0。

初学者主题：不用await返回Task

注意在代码清单19-1中，虽然Task.Run(()=>Decrement())返回一个Task，但并未使用await操作符。原因是只有在用async修饰的方法中才能使用await，而Main()不支持使用async。重构Main()外部的代码，就可以简单地利用await/async模式，如代码清单19-3所示。

代码清单19-3 未同步的局部变量

```
using System;
using System.Threading.Tasks;

public class Program
{
  // ...

  public static async void CountAsync()
```

① 虽然在C#的层级上是局部变量，但在IL的层级上是字段，而字段能从多个线程访问。

```
        {
            // Use Task.Factory.StartNew for .NET 4.0
            Task task = Task.Run(()=>Decrement());

            // Increment
            for(int i = 0; i < _Total; i++)
            {
                _Count++;
            }

            await task;
            Console.WriteLine("Count = {0}", _Count);
        }
    }
```

19.1.1 使用Monitor来同步

为了同步多个线程，防止它们同时执行特定的代码段，需要用**监视器**（monitor）来阻止第二个线程进入受保护的代码段，直到第一个线程退出那个代码段。监视器功能由System.Threading.Monitor类提供。为了标识受保护代码段的开始和结束位置，需要分别调用静态方法Monitor.Enter()和Monitor.Exit()。

代码清单19-4演示了显式使用Monitor类来进行同步。要记住的一个重点是，在Monitor.Enter()和Monitor.Exit()这两个调用之间，所有代码都要用一个try/finally块包围起来。否则，在受保护的代码段内可能发生一个异常，造成Monitor.Exit()永远无法调用，从而长时间地阻塞其他线程。

代码清单19-4 显式使用监视器来进行同步

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Program
{
    readonly static object _Sync = new object();
    const int _Total = int.MaxValue;
    static long _Count = 0;

    public static void Main()
    {
        // Use Task.Factory.StartNew for .NET 4.0
        Task task = Task.Run(()=>Decrement());

        // Increment
        for(int i = 0; i < _Total; i++)
        {
            bool lockTaken = false;
            try
            {
                Monitor.Enter(_Sync, ref lockTaken);
                _Count++;
            }
            finally
            {
```

```
                if (lockTaken)
                {
                    Monitor.Exit(_Sync);
                }
            }
        }

        task.Wait();
        Console.WriteLine("Count = {0}", _Count);
    }

    static void Decrement()
    {
        for(int i = 0; i < _Total; i++)
        {
            bool lockTaken = false;
            try
            {
                Monitor.Enter(_Sync, ref lockTaken);
                _Count--;
            }
            finally
            {
                if(lockTaken)
                {
                    Monitor.Exit(_Sync);
                }
            }
        }
    }
```

输出19-2展示了代码清单19-4的结果。

输出19–2

```
Count = 0
```

对Monitor.Enter()和Monitor.Exit()的调用是如何关联起来的呢？注意，在上述代码中，同一个对象引用（本例是_Sync）被作为参数传给了每个方法。

获取lockTaken的Monitor.Enter()重载方法是.NET 4.0新增的。在此之前没有lockTaken参数，没有办法可靠地捕捉Monitor.Enter()和try块之间发生的异常。让try块紧跟在Monitor.Enter()调用的后面，在发布（release）代码中是非常可靠的，因为JIT禁止出现像这样的任何异步异常。然而，紧跟在Monitor.Enter()后面的其他任何除try块以外的东西，包括编译器可能在调试（debug）代码中注入的任何指令，都有可能妨碍JIT可靠地从try块中返回。所以，如果真的发生异常，它会造成锁的泄漏（锁一直保持已获取的状态），而不是执行final块并释放锁。另一个线程试图获取锁的时候，就可能造成死锁。总之，在.NET 4.0之前，总是在Monitor.Enter()之后跟随一个try/finally{Monitor.Exit(_Sync)}块。

Monitor还支持Pulse()方法，它允许线程进入“就绪队列”（ready queue），指出下一个就轮到它获得锁（并可以开始执行）。这是同步生产者—消费者模式的一种常见方

式，目的是保证除非有“生产”，否则就没有“消费”。拥有监视器（通过调用Monitor.Enter()）的生产者线程调用Monitor.Pulse()通知消费者线程（它可能已经调用了Monitor.Enter()），一个项（item）已准备好供消费，所以请“准备好”（get ready）。一个Pulse()调用只允许一个线程（本例的消费者）进入就绪队列。生产者线程调用Monitor.Exit()时，消费者线程将取得锁（Monitor.Enter()结束）并进入关键执行区域[①]以开始“消费”那个项。消费者处理好等待处理的项以后，就调用Exit()，从而允许生产者（当前正由Monitor.Enter()阻塞）再次生产其他项。在本例中，一次只有一个线程进入就绪队列，确保没有“生产”就没有“消费”，反之亦然。

19.1.2 使用lock关键字

由于多线程代码中要频繁地使用Monitor来进行同步，同时try/finally块很容易被人遗忘，所以C#提供了特殊的关键字lock来处理这种锁定同步模式。代码清单19-5演示了如何使用lock关键字，输出19-3展示了结果。

代码清单19-5 使用lock关键字来同步

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Program
{
    readonly static object _Sync = new object();
    const int _Total = int.MaxValue;
    static long _Count = 0;

    public static void Main()
    {
        // Use Task.Factory.StartNew for .NET 4.0
        Task task = Task.Run(()=>Decrement());

        // Increment
        for(int i = 0; i < _Total; i++)
        {
            lock(_Sync)
            {
                _Count++;
            }
        }

        task.Wait();
        Console.WriteLine("Count = {0}", _Count);
    }

    static void Decrement()
    {
```

① 即critical execution region，文档中翻译成“关键执行区域”，也有人称为“临界执行区域”。关键区域是指线程终止或未处理异常的影响可能不限于当前任务的区域。相反，非关键代码区域中的终止或失败只对出现错误的任务有影响。——译者注

```
            for(int i = 0; i < _Total; i++)
            {
                lock(_Sync)
                {
                    _Count--;
                }
            }
        }
    }
```

输出19-3

```
Count = 0
```

锁定了要访问_Count的代码段（使用lock或者Monitor），Main()和Decrement()方法就是线程安全的。换言之，可以从多个线程中安全地同时调用它们。（C# 4.0之前的概念是一样的，只是编译器生成的代码要依赖于没有lockTaken参数的Monitor.Enter()方法，而且Monitor.Enter()在try块之前调用。）

同步是以牺牲性能为代价的。例如，代码清单19-5的执行时间比代码清单19-1长得多，这证明与直接递增和递减_Count相比，lock的速度相当慢。

即使为了同步的需要可以忍受lock的速度，也不要在多处理器计算机中不假思索地添加同步来避免死锁和不必要的同步（也许本来可以并行执行的）。对象设计的“最佳实践”是对可变的静态状态进行同步（永远不变的东西不必同步），但不要同步实例数据。如果允许多个线程访问一个特定的对象，那么必须为对象提供同步。任何类如果要显式地和线程打交道，通常应该保证实例在某种程度上是线程安全的。

19.1.3 lock对象的选择

无论是使用lock关键字，还是显式使用Monitor类，都必须小心地选择lock对象。

在前面的例子中，同步变量_Sync被声明为私有和只读。声明为只读是为了确保在Monitor.Enter()和Monitor.Exit()调用之间，其值不会发生改变。这就在同步块的进入和退出之间建立了关联。

类似地，将_Sync声明为私有，是为了确保类外的同步块不能同步同一个对象实例，这会造成代码阻塞。

假如数据是公共的，那么同步对象可能是公共的，造成其他类能用同一个同步对象实例来进行同步。这将使得死锁更难以避免。幸好，对这个模式的需求很少。对于公共数据，更好的做法是完全在类的外部进行同步，允许调用代码为它自己的同步对象获取锁。

同步对象不能是值类型，这一点很重要。如果非要在值类型上使用lock关键字，编译器会报错（然而，如果显式访问System.Threading.Monitor类，而不是通过lock，那么编译时不会报错。相反，代码会在调用Monitor.Exit()时引发异常，指出没有对应的Monitor.Enter()调用）。使用值类型的时候，“运行时”会创建值的一个副本，把它放到堆中（发生装箱），并将装箱的值传给Monitor.Enter()。类似地，Monitor.Exit()会接收到原始变量的一个已装箱的副本。结果是Monitor.Enter()和Monitor.Exit()接收到了不同的同步对象实例，所以两个调用失去了关联性。

19.1.4 为什么要避免锁定this、typeof(type)和string

一个常见的模式是锁定代表类中实例数据的this关键字，以及锁定从typeof(type)获取的类型实例（例如typeof(MyType)）上锁定。如果使用this，可以为与一个特定对象实例关联的所有状态提供一个同步目标；如果使用typeof(type)，则可以为一个类型的所有静态数据提供一个同步目标。但这样做的问题在于，在另一个完全不相干的代码块中，可能创建一个完全不同的同步块，而这个同步块的同步目标可能就是this（或者typeof(type)）所指向的同步目标。换言之，虽然只有实例自身内部的代码可以使用this关键字来阻塞，但创建实例的调用者仍可将那个实例传给一个同步锁。

结果就变成了对两套不同的数据进行同步的两个同步块可能互相阻塞对方。虽然看起来不太可能，但是共享同一个同步目标可能影响性能，极端的时候甚至会造成死锁。所以，请不要在this或者typeof(type)上锁定。更好的做法是定义一个私有只读字段，除了能访问它的那个类之外，没有谁能在这个字段上阻塞。

要避免的另一个锁定类型是string，这是因为要考虑到字符串留用[①]（string interning）问题。如果同一个字符串常量在多个位置出现，那么所有位置都可能引用同一个实例，使锁定的范围大于预期。

总之，锁定的目标应该是object类型的单位同步上下文实例（per-synchronization context instance，例如前面说的私有只读字段）。

规范

避免锁定this、typeof()或者字符串。

要为同步目标声明object类型的一个单独的只读同步变量。

高级主题：避免用MethodImplAttribute同步

.NET 1.0引入的一个同步机制是MethodImplAttribute。和MethodImplOptions.Synchronized方法配合使用，这个特性能将一个方法标记为已同步，确保每次只有一个线程执行方法。为了达到这个目的，JIT编译器本质上会将方法看成是被lock(this)包围，如果是静态方法，则看成是已在类型上锁定。像这样的实现意味着，方法以及在同一个类中的其他所有方法，只要用相同的特性和枚举参数进行了修饰，它们事实上就是同步的，而不是“各顾各”[②]。换言之，在同一个类中，如果两个或多个方法都用特性进行了修饰，那么每次只能执行其中一个。一个方法执行时，除了会阻塞其他线程对它自己的调用，还会阻塞对类中具有相同修饰的其他任何方法的调用。除此之外，由于同步是在this（或者更糟，是在类型上）进行的，所以上一节讲过的lock(this)（或者更糟，针对静态数据，在从typeof(type)获取的类型上锁定）存在的问题在这里同样会发生。因此，最佳实践是完全避免使用这个特性。

---

① 文档中将interning翻译成“拘留”，专门供字符串留用的表称为“拘留池”。本书采用“留用”这一译法。——译者注

② 例如，只阻塞其他线程对它们自己的调用。——译者注

> **规范**
>
> 避免用MethodImplAttribute同步。

19.1.5 将字段声明为volatile

某些时候，编译器和/或CPU会对代码进行优化，使指令不按照它们编码的顺序执行，或者干脆拿掉一些无用的指令。若代码只在一个线程上执行，像这样的优化无伤大雅。然而，对于多个线程，像这样进行优化，就可能造成出乎意料的结果。这是因为优化可能造成两个线程对同一个字段的读写顺序发生错乱。

为了解决这个问题，一个办法是使用volatile关键字声明字段。这个关键字强迫对volatile字段的所有读写操作都在代码指示的位置发生，而不是在通过优化而生成的其他某个位置发生。volatile修饰符指出字段容易被硬件、操作系统或者另一个线程修改。所以，这种数据是“易变的”（volatile），编译器和“运行时”要更加严谨地处理它。

一般很少使用volatile修饰符。即便使用，也可能因为疏忽而错误地使用。lock比volatile更好，除非对volatile的用法有绝对的把握。

19.1.6 使用System.Threading.Interlocked类

迄今为止讨论的互斥（排他）模式提供了在一个进程（应用程序域）中处理同步的一套基本工具。然而，用System.Threading.Monitor进行同步代价非常高。除了使用Monitor之外，还有一个备选的方案，它通常直接由处理器支持，而且面向特定的同步模式。

代码清单19-6将_Data设为新值——只要它之前的值是null。正如方法名（CompareExchange）揭示的那样，这是一个比较/交换（Compare/Exchange）模式。在这里，不需要手动锁定具有等价行为的比较和交换代码。相反，Interlocked.CompareExchange()方法内建了同步机制，它同样会检查null值，如果值等于第二个参数的值，就更新第一个参数。表19-2展示了Interlocked类支持的其他同步方法。

代码清单19-6 使用System.Threading.Interlocked进行同步

```
public class SynchronizationUsingInterlocked
{
    private static object _Data;

    // Initialize data if not yet assigned.
    static void Initialize(object newValue)
    {
        // If _Data is null then set it to newValue.
        Interlocked.CompareExchange(
            ref _Data, newValue, null);
    }

    // ...
}
```

表19-2 Interlock提供的与同步相关的方法

方法签名	描述
`public static T CompareExchange<T>( T location, T value, T comparand );`	检查location的值是不是comparand。如果两个值相等，就将location设为value，并返回location中存储的原始数据
`public static T Exchange<T>( T location, T value );`	将value赋给location，并返回之前的值
`public static int Decrement( ref int location );`	location减1。等价于--操作符，只是Decrement是线程安全的
`public static int Increment( ref int location );`	location增1。等价于++操作符，只是Increment是线程安全的
`public static int Add( ref int location, int value );`	将value加到location上，并将结果赋给location。等价于+=操作符
`public static long Read( ref long location );`	在一次原子操作中返回一个64位值

其中大多数方法都进行了重载，支持其他数据类型（例如long）签名。表19-2提供了常规签名和描述。

注意，Increment()和Decrement()可以在代码清单19-6中替换已同步的“++”和“--”操作符，这样可以获得更好的性能。还要注意，假如一个不同的线程使用一个非Interlocked提供的方法来访问location，那么两个访问将不会得到正确的同步。

19.1.7 多个线程时的事件通知

开发者容易忽视触发事件时的同步问题。代码清单19-7展示了非线程安全的事件发布代码。

代码清单19-7 触发一个事件通知

```
// Not thread-safe
if(OnTemperatureChanged != null)
{
  // Call subscribers
  OnTemperatureChanged(
      this, new TemperatureEventArgs(value) );
}
```

只要这个方法和修改事件的代码之间没有竞态条件，这段代码就是有效的。然而，这段代码不是原子性的，所以多个线程可能造成竞态条件。从检查OnTemperatureChange是否为null到实际触发事件这段时间里，OnTemperatureChange可能被设为null，从而引发一个NullReferenceException。换言之，如果委托可能由多个线程同时访问，就需要对委托的赋值和触发进行同步。

幸好，用于添加和删除侦听器的操作符是线程安全的，而且是静态的（操作符重载通过静态方法完成）。为了修正代码清单19-7，并使它成为线程安全的，需要为一个副本赋值，检查副本是否为null，再触发副本（参见代码清单19-8）。

代码清单19-8 线程安全的事件通知

```
// ...
TemperatureChangedHandler localOnChange =
    OnTemperatureChanged;
if(localOnChanged != null)
{
  // Call subscribers
  localOnChanged(
      this, new TemperatureEventArgs(value) );
}
// ...
```

由于委托是引用类型，所以有人会觉得奇怪：为什么赋值一个局部变量，再触发那个局部变量，就足以使null检查成为线程安全的操作？由于localOnChange和OnTemperatureChange指向同一位置，所以有人会认为：OnTemperatureChange中的任何改变都会在localOnChange中反映出来。

但实情并非如此。这是由于对OnTemperatureChange += <*listener*>的任何调用都不会为OnTemperatureChange添加新委托。相反，会为它赋予一个全新的多播委托，而不会对原始的多播委托（localOnChange也指向这个委托）产生任何影响。这就使代码成为线程安全的，因为只有一个线程会访问localOnChange实例。添加或删除侦听器，OnTemperatureChange会成为一个全新的实例。

19.1.8 同步设计最佳实践

考虑到多线程编程的复杂性，有几条最佳设计实践可供学习。

1. 避免死锁

同步的引入带来了死锁的可能。两个或更多的线程都在等待对方释放一个同步锁，就会发生死锁。例如，线程1请求_Sync1上的锁，并在释放_Sync1锁之前，请求_Sync2上的锁。与此同时，线程2请求_Sync2上的锁，并在释放_Sync2锁之前，请求_Sync1上的锁。这就埋下了发生死锁的隐患。假如线程1和线程2都在获得第二个锁之前成功获得了它们请求的第一个锁（分别是_Sync1和_Sync2），就会实际地发生死锁。

死锁的发生必须满足以下4个基本条件。

（1）排他或互斥（Mutual exclusion）：一个线程（ThreadA）独占一个资源，没有其他线程（ThreadB）能获取相同的资源。

（2）占有并等待（Hold and wait）：互斥的一个线程（ThreadA）请求获取另一个线程（ThreadB）占有的资源。

（3）不可抢先（No preemption）：一个线程（ThreadA）占有的资源不能被强制拿走（只能等待ThreadA主动释放它锁定的资源）。

（4）循环等待条件（Circular wait condition）：两个或多个线程构成一个循环等待链，它们锁定两个或多个相同的资源，每个线程都在等待链中的下一个线程占有的资源。

移除其中任何一个条件，都能阻止死锁的发生。

有可能造成死锁的一个情形是，两个或多个线程请求独占对相同的两个或多个同步目标（资源）的所有权，而且以不同的顺序请求锁。如果开发人员小心一些，保证多个锁总是以相同的顺序获得，就可避免这个情形。发生死锁的另一个原因是不可重入（reentrant）的锁。如果来自一个线程的锁可能阻塞同一个线程（换言之，它重新请求同一个锁），这个锁就是不可重入的。例如，假定ThreadA获取一个锁，然后重新请求同一个锁，但由于锁已被拥有，所以发生了阻塞，那么这个锁就是不可重入的，额外的请求会造成死锁。所以，不可重入的锁只在单线程的时候发生。

lock关键字（通过底层的Monitor类）生成的代码是可重入的。然而，如同19.1.9节“更多的同步类型”要讲到的那样，还有一些类型的锁是不可重入的。

2. 何时提供同步

前面讲过，所有静态数据都应该确保线程安全。所以，同步需要围绕静态数据来进行。通常，这意味着程序员应该声明私有静态变量，并提供公共方法来修改数据。这些方法在内部应处理好同步问题。

相反，实例数据不需要包含同步机制。同步会显著降低性能，并增大争夺锁或者死锁的概率。除了显式设计成由多个线程访问的类之外，程序员在多个线程中共享对象时，应当针对要共享的数据，解决好它们自己的同步问题。

3. 避免不必要的锁定

在不破坏数据完整性的前提下，同步能避免的就要尽量避免。例如，在线程之间使用不可变的类型，避免对同步的需要（这个方式的价值在F#这样的函数式编程语言中得到了证明）。类似地，避免锁定本来就是线程安全的操作，比如对小于本机（指针大小）整数的值的读写，这种操作本来就是原子性的。

规范

不要以不同的顺序请求对相同两个或更多同步目标的排他所有权。

要确保同时持有多个锁的代码总是以相同的顺序获得这些锁。

要将可变的静态数据封装到具有同步逻辑的公共API中。

避免对不大于本机（指针大小）整数的值的简单读写操作进行同步，这种操作本来就是原子性的。

19.1.9 更多的同步类型

除了System.Threading.Monitor和System.Threading.Interlocked之外，还存在着其他几种同步技术。

1. System.Threading.Mutex

System.Threading.Mutex在概念上和System.Threading.Monitor类几乎完全一致（没有Pulse()方法支持），只是lock关键字用的不是它，而且可以命名不同的Mutex来支持多个进程之间的同步。可用Mutex类同步对文件或者其他跨进程资源的访问。由于Mutex是一个跨进程资源，所以.NET 2.0开始允许通过一个System.Security.AccessControl.

MutexSecurity对象来设置访问控制。Mutex类的一个用处是限制应用程序不能同时运行多个实例，如代码清单19-9所示。

代码清单19-9 创建单实例应用程序

```
using System;
using System.Threading;
using System.Reflection;

public class Program
{
public static void Main()
{
    // Indicates whether this is the first
    // application instance
    bool firstApplicationInstance;

    // Obtain the mutex name from the full
    // assembly name.
    string mutexName =
        Assembly.GetEntryAssembly().FullName;

    using(Mutex mutex = new Mutex(false, mutexName,
        out firstApplicationInstance))
    {

        if(!firstApplicationInstance)
        {
            Console.WriteLine(
                "This application is already running.");
        }
        else
        {
            Console.WriteLine("ENTER to shutdown");
            Console.ReadLine();
        }
    }
  }
}
```

运行应用程序的第一个实例，会得到输出19-4的结果。

输出19-4

```
ENTER to shutdown
```

保持第一个实例的运行状态，再运行应用程序的第二个实例，会得到输出19-5的结果。

输出19-5

```
This application is already running.
```

在上例中，应用程序在整个计算机上只能运行一次，即使它由不同的用户启动。要限制每个用户最多只能运行一个实例，需要为Assembly.GetEntryAssembly().FullName附加System.Windows.Forms.Application.UserAppDataPath.Replace("\\", "+")前缀。这需要

对System.Windows.Forms程序集的引用。

Mutex派生自System.Threading.WaitHandle，所以它包含WaitAll()、WaitAny()和SignalAndWait()方法，可以自动获取多个锁（这是Monitor类不支持的）。

2. WaitHandle

Mutex的基类是System.Threading.WaitHandle。后者是由Mutex、EventWaitHandle和Semaphore等同步类使用的一个基础同步类。WaitHandle的关键方法是WaitOne()，它有多个重载版本。这些方法会阻塞当前线程，直到WaitHandle实例收到信号或者被设置（调用Set()）。WaitOne()的几个重载版本：void WaitOne()，无限期等待， bool WaitOne(int milliseconds) 等待指定的毫秒；bool WaitOne(TimeSpan timeout)，等待一个TimeSpan指定的时间。对于那些返回Boolean的版本，只要WaitHandle在超时之前收到信号，就会返回一个true值。

除了WaitHandle实例方法，还有两个关键的静态成员：WaitAll()和WaitAny()。和它们的实例版本相似，这两个静态成员也支持超时。除此之外，它们要获取一个WaitHandle集合（以一个数组的形式），使它们能响应来自集合中的任何WaitHandle的信号。

关于WaitHandle最后要注意的一点是，它包含了一个实现了IDisposable的SafeWaitHandle类型的句柄。所以，在WaitHandle不再需要的时候，注意要对它们进行资源清理（dispose）。

3. 重置事件类：ManualResetEvent和ManualResetEventSlim

前面说过，假如不进行控制，一个线程的指令相对于另一个线程中的指令的执行时机是不确定的。对这种不确定性进行控制的另一个办法是使用重置事件（reset event）。虽然名称中有事件一词，但重置事件和C#的委托以及事件没有任何关系。重置事件用于强迫代码等候另一个线程的执行，直到获得事件已经发生的通知。它们尤其适合用来测试多线程代码，因为有时需要先等待一个特定的状态，才能对结果进行验证。

重置事件类型包括System.Threading.ManualResetEvent和.NET Framework 4新增的轻量级版本System.Threading.ManualResetEventSlim。（如同稍后的“高级主题”讲述的那样，还有第三个类型，即System.Threading.AutoResetEvent，但程序员应避免用它，尽量用前两个类型。）它们提供的关键方法是Set()和Wait ()（ManualResetEvent提供的称为WaitOne()）。调用Wait()方法，会阻塞一个线程的执行，直到一个不同的线程调用Set()，或者直到设定的等待时间结束（超时）。代码清单19-10演示了具体过程，输出19-6展示了结果。

代码清单19-10 等待ManualResetEventSlim

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Program
{
  static ManualResetEventSlim MainSignaledResetEvent;
  static ManualResetEventSlim DoWorkSignaledResetEvent;

  public static void DoWork()
  {
```

```
        Console.WriteLine("DoWork() started....");
        DoWorkSignaledResetEvent.Set();
        MainSignaledResetEvent.Wait();
        Console.WriteLine("DoWork() ending....");
    }

    public static void Main()
    {
        using(MainSignaledResetEvent =
            new ManualResetEventSlim())
        using (DoWorkSignaledResetEvent =
        new ManualResetEventSlim())
    {
        Console.WriteLine(
            "Application started....");
        Console.WriteLine("Starting task....");

        // Use Task.Factory.StartNew for .NET 4.0
        Task task = Task.Run(()=>DoWork());

        // Block until DoWork() has started.
        DoWorkSignaledResetEvent.Wait();
        Console.WriteLine(
            " Waiting while thread executes...");
        MainSignaledResetEvent.Set();
        task.Wait();
        Console.WriteLine("Thread completed");
        Console.WriteLine(
            "Application shutting down....");
        }
    }
}
```

输出19-6

```
Application started....
Starting thread....
DoWork() started....
Waiting while thread executes...
DoWork() ending....
Thread completed
Application shutting down....
```

代码清单19-10首先实例化并启动一个新的Task。表19-3展示了执行路径，其中每一列都代表一个线程。如果代码出现在同一行，表明不确定哪一边先执行。

表19-3 采用ManualResetEvent同步的执行路径

Main()	DoWork()
...	
Console.WriteLine("Application started.....");	
Task task = new Task(DoWork);	

续表

Main()	DoWork()
Console.WriteLine("Starting thread....");	
task.Start();	
DoWorkSignaledResetEvent.Wait();	Console.WriteLine("DoWork() started....");
	DoWorkSignaledResetEvent.Set();
Console.WriteLine("Thread executing...");	MainSignaledResetEvent.Wait();
MainSignaledResetEvent.Set();	
task.Wait();	Console.WriteLine("DoWork() ending....");
Console.WriteLine("Thread completed");	
Console.WriteLine("Application exiting....");	

调用重置事件的Wait()方法（或者调用ManualResetEvent的WaitOne()方法）会阻塞当前的调用线程，直到另一个线程向其发出信号，并允许被阻塞的线程继续。但是，除了阻塞不确定的时间之外，Wait()/WaitOne()还有一些重载版本，允许阻塞当前线程确定性的时间。它们允许用一个参数来指定最大的阻塞时间。该参数要么以毫秒为单位指定，要么作为一个TimeSpan对象来指定。如果指定了超时期限，那么假如在重置事件收到信号之前超时，WaitOne()就会返回false值。ManualResetEvent.Wait()还有一个版本可以获取一个取消标志，从而允许取消请求（参见上一章的描述）。

ManualResetEventSlim和ManualResetEvent的区别在于，后者默认使用核心同步，而前者进行了优化—除非万不得已，否则会尽量避免使用核心机制。因此，ManualResetEventSlim的性能更好，虽然它可能会使用更多的CPU周期。所以，一般情况下应该使用ManualResetEventSlim，除非需要等待多个事件，或者需要跨越多个进程。

注意，重置事件实现了IDisposable，所以不需要的时候，应该释放它们。代码清单19-9是用一个using语句来做这件事情。（CancellationTokenSource包含一个ManualResetEvent，这正是它也实现了IDisposable的原因。）

System.Threading.Monitor包含了Wait()和Pulse()方法，它们在某些情况下提供了与重置事件类似的功能，虽然两者并不完全相同。

高级主题：尽量用ManualResetEvent和信号量而不要用AutoResetEvent

还有第三个重置事件，即System.Threading.AutoResetEvent。与ManualResetEvent相似，它允许线程A通知线程B（通过一个Set()调用）线程A已经抵达代码中的一个特定的位置。区别在于，AutoResetEvent只解除一个线程的Wait()调用所造成的阻塞，因

> 为在线程A在通过自动重置的大门后，它会恢复锁定状态。然而，如果使用自动重置事件，很容易在编写生产者线程时发生失误，造成它的迭代次数多于消费者线程。因此，一般情况下最好使用Monitor的Wait()/Pulse()模式，或者使用一个信号量（如果少于*n*个线程能参与一个特定的阻塞）。
>
> 和AutoResetEvent相反，除非显式调用Reset()，否则ManualResetEvent不会恢复到未收到信号之前[①]状态。

4. Semaphore/SemaphoreSlim和CountdownEvent

Semaphore/SemaphoreSlim在性能上的差异和ManualResetEvent/ManualResetEventSlim一样。ManualResetEvent/ManualResetEventSlim提供了一个要么打开要么关闭的锁（就像一道门）。和它们不同的是，信号量（semaphore）限制的只是在一个关键执行区域中同时通过的调用。信号量本质上是保持了对资源池的一个计数。计数为0时，就阻止对资源池更多访问，直到其中的一个资源返回。有可用的资源之后，就可以把它拿给队列中的下一个已阻塞的请求。

CountdownEvent和信号量相似，只是它实现的是反向同步。不是阻止对已经枯竭的一个资源池的访问，而是只有在计数为0时才允许访问。例如，假定一个并行操作是下载多个股票价格。只有在所有价格都下载完毕之后，才能执行一个特定的搜索算法。在这种情况下，可以用CountdownEvent对搜索算法进行同步，每下载一个股票，就使计数递减1。计数为0时，就允许搜索开始。

注意，SemaphoreSlim和CountdownEvent是在.NET Framework 4中引入的。在 .NET 4.5中，前者包含了一个SemaphoreSlim.WaitAsync()方法，这样就可以在等待进入信号量时使用TAP。

5. 并发集合类

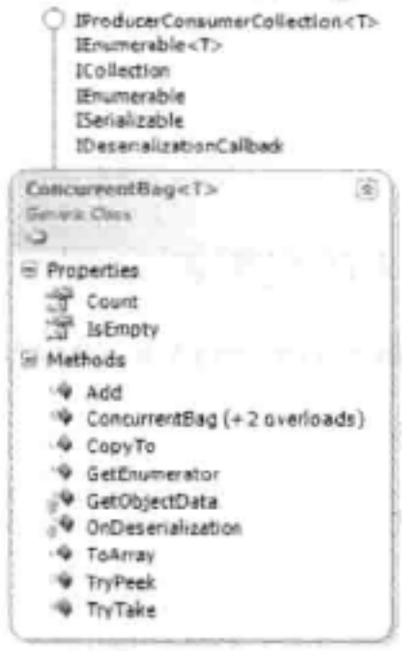
.NET Framework 4新引入的一系列类是并发集合类。这些类专门用来包含内建的同步代码，使它们能支持多个线程同时访问而不必关心竞态条件。表19-4总结了并发集合类。

表19-4 并发集合类

集合类	描述
BlockingCollection<T> 	提供一个阻塞集合，允许在生产者/消费者模式中，生产者向集合中写入数据，同时消费者从集合中读取数据。这个类提供了一个泛型集合类型，支持同步添加和删除操作，而不必关心后端存储（可以是队列、栈、列表等）。BlockingCollection<T>为实现了IProducer Consumer Collection<T>接口的集合提供了阻塞和bounding支持

① 调用Set()允许线程继续（收到信号），Reset()则使线程阻塞（恢复到没有收到信号的状态）。——译者注

续表

集合类	描述
*ConcurrentBag<T> 	一个线程安全的无序集合，由T类型的对象构成
ConcurrentDictionary <TKey, TValue> 	一个线程安全的字典，由键/值对构成的集合
*ConcurrentQueue<T> 	一个线程安全的队列，支持T类型对象的先进先出（FIFO）语义
*ConcurrentStack<T> 	一个线程安全的栈，支持T类型对象的先进后出（FIFO）语义

* 实现IProducerConsumerCollection<T>的集合类。

利用并发集合，可以实现的一个常见的模式是生产者和消费者的线程安全的访问。实现了IProducerConsumerCollection<T>的类（表19-4中用*标注）是专门为了支持这个模式而设计的。它允许一个或多个类将数据写入集合，而一个不同的集合将其读出并删除。数据添加和删除的顺序由实现了IProducerConsumerCollection<T>接口的单独的集合类决定。

19.1.10 线程本地存储

某些情况下，使用同步锁可能导致让人无法接受的性能，并对伸缩性造成限制。另一些情况下，围绕特定数据元素提供同步可能过于复杂，尤其是在以前写好的原始代码的基础上进行修补。

同步的一个替代方案是隔离，而实现隔离的一个办法就是使用**线程本地存储**。利用线程本地存储，线程就有了专属的变量的实例。这样就没有同步的必要了，因为对只在单个线程的上下文中发生的数据进行同步是没有意义的。线程本地存储的实现有两个例子，分别是ThreadLocal<T>和ThreadStaticAttribute。

1. ThreadLocal<T>

为了使用.NET Framework 4提供的线程本地存储，需要声明ThreadLocal<T>类型的一个字段（在编译器生成的闭包类[①]的前提下，则是一个变量）。这样一来，每个线程都有字段的一个不同实例。代码清单19-11和输出19-7对此进行了演示。注意，虽然字段是静态的，但却有一个不同的实例。

代码清单19-11 用ThreadLocal<T>实现线程本地存储

```
using System;
using System.Threading;

public class Program
{
  static ThreadLocal<double> _Count =
      new ThreadLocal<double>(() => 0.01134);

  public static double Count
  {
      get { return _Count.Value; }
      set { _Count.Value = value; }
  }

  public static void Main()
  {
      Thread thread = new Thread(Decrement);
      thread.Start();

      // Increment
      for(double i = 0; i < short.MaxValue; i++)
      {
          Count++;
      }

      thread.Join();
      Console.WriteLine("Main Count = {0}", Count);
```

① 闭包（closure）是由编译器生成的一个数据结构（一个C#类），其中包含一个表达式以及对表达式进行求值所需的变量（C#中的公共字段）。变量允许在不改变表达式签名的前提下，将数据从表达式的一次调用传递到下一次调用。——译者注

续表

```
}

static void Decrement()
{
    Count = -Count;
    for (double i = 0; i < short.MaxValue; i++)
    {
        Count--;
    }
    Console.WriteLine(
        "Decrement Count = {0}", Count);
    }
}
```

输出19-7

```
Decrement Count = -32767.01134
Main Count = 32767.01134
```

如输出19-7所示，在执行Main()的线程中，Count的值永远不会被执行Decrement()的线程递减。对于Main()的线程，初始值是0.01134，终值是32767.01134。Decrement()具有类似的值，只是它们是负的。由于Count是基于ThreadLocal<T>类型的静态字段，所以运行Main()的线程和运行Decrement()的线程在_Count.Value中存储有独立的值。

2. 用ThreadStaticAttribute提供线程本地存储

用ThreadStaticAttribute修饰静态字段（如代码清单19-12所示）是指定静态变量每线程一个实例的第二个办法。虽然和ThreadLocal<T>相比，这个技术有一些小缺点，但它的优点在于对.NET Framework 4之前的版本也支持。（另外，由于ThreadLocal<T>基于ThreadStaticAttribute，所以如果涉及大量重复的、小的迭代处理，后者消耗的内存会少一些，性能也会好一些。）

代码清单19-12 用ThreadStaticAttribute实现线程本地存储

```
using System;
using System.Threading;

public class Program
{
    [ThreadStatic]
    static double _Count = 0.01134;
    public static double Count
    {
        get { return Program._Count; }
        set { Program._Count = value; }
    }

    public static void Main()
    {
        Thread thread = new Thread(Decrement);
        thread.Start();

        // Increment
        for(int i = 0; i < short.MaxValue; i++)
        {
            Count++;
```

```
        }

        thread.Join();
        Console.WriteLine("Main Count = {0}", Count);
    }

    static void Decrement()
    {
        for(int i = 0; i < short.MaxValue; i++)
        {
            Count--;
        }
        Console.WriteLine("Decrement Count = {0}", Count);
    }
}
```

代码清单19-12的结果如输出19-8所示。

输出19-8

```
Decrement Count = -32767
Main Count = 32767.01134
```

如上述代码清单所示，在执行Main()的线程中，Count的值永远不会被执行Decrement()的线程递减。对于Main()的线程，初始值是negative_Total，终值是0。在用ThreadStaticAttribute修饰了之后，每个线程的Count值都是那个线程专属的，不可跨线程访问。

注意，和代码清单19-11不同，为“Decrement Count”显示的值没有任何小数位，意味着它一直没有被初始化为0.01134。虽然_Count在声明时赋了值（本例是static double _Count = 0.01134），但只有和“正在运行静态构造器的线程”关联的线程静态实例（也就是线程本地存储变量）才会被初始化。在代码清单19-12中，只有在执行Main()的那个线程中，才有一个线程本地存储变量被初始化为0.01134。由Decrement()递减的_Count总是被初始化为0（(default(double)，因为_Count是一个double)。类似地，如果构造器初始化一个线程本地存储字段，只有调用那个线程的构造器才会初始化线程本地存储实例。因此，好的编程实践是在每个线程最初调用的方法内部，对一个线程本地存储字段进行初始化。但这样做并非总是合理的，尤其是在涉及async的时候。在这种情况下，计算的不同部分可能在不同的线程上运行，造成每一部分不同的线程本地存储值。

决定是否使用线程本地存储时，需要进行一番性价比分析。例如，可以考虑为一个数据库连接使用线程本地存储。取决于数据库管理系统，数据库连接可能是相当昂贵的，为每个线程都创建连接可能不现实。另一方面，如果锁定一个连接来同步所有数据库调用，会造成可伸缩性的急剧下降。每个模式都有利与弊，具体如何抉择在很大程度上取决于单独的实现。

使用线程本地存储的另一个原因是将经常需要的上下文信息提供给其他方法使用，同时不显式地通过参数来传递数据。例如，假如调用栈中的多个方法都需要用户安全信息，就可以使用线程本地存储字段而不是参数来传递数据。这样可以使API更简洁，同时仍然能以线程安全的方式将信息传给方法。这要求你保证总是设置线程本地数据，在Task或其他线程池线程上，这一点尤其重要，因为基础线程是重用的。

19.2 计时器

有时需要将代码的执行推后一段时间，或者在指定时间后发出通知。例如，可能要以固定周期刷屏，而不是每当数据有变化时就刷新。实现计时器的一个方式是利用C# 5.0的async/await模式和.NET 4.5加入的Task.Delay()方法。如上一章所述， TAP的一个关键功能就是async调用之后执行的代码会在支持的线程上下文中继续，从而避免了UI跨线程问题。代码清单19-13是使用Task.Delay()方法的一个例子。

代码清单19-13 Task.Delay()作为计时器使用

```
using System;
using System.Threading.Tasks;

public class Pomodoro
{
  // ...

  private static async Task TickAsync(
      System.Threading.CancellationToken token)
  {
      for(int minute = 0; minute < 25; minute++)
      {
          DisplayMinuteTicker(minute);
          for(int second = 0; second < 60; second++)
          {
              await Task.Delay(1000);
              if(token.IsCancellationRequested) break;
              DisplaySecondTicker();
          }
          if(token.IsCancellationRequested) break;
      }
  }
}
```

对Task.Delay(1000)的调用会设置一个倒计时器，它1秒后触发并执行之后的延续代码。

在C# 5.0中，TAP专门使用同步上下文解决了UI跨线程问题。在此之前，必须使用UI线程安全（或者可以配置成这样）的特殊计时器类。System.Windows.Forms.Timer、System.Windows.Threading.DispatcherTimer和System.Timers.Timer（要专门配置）都是UI线程友好的。其他计时器（如System.Threading.Timer）是为性能而优化的。

和C# 5.0 async/await之前的计时器有关的其他细节和示例代码请参考附录D。

高级主题：使用STAThreadAttribute控制COM线程处理模型

使用COM，4个不同的单元线程处理模型决定了与COM对象之间的调用有关的线程处理规则。幸好，只要程序没有调用COM组件，这些规则以及随之而来的复杂性就从.NET中消失了。处理COM Interop（COM互操作）的常规方式是将所有.NET组件都放到一个主要的、单线程的单元中，具体做法就是使用System.STAThreadAttribute来修饰进程的Main方法。这样一来，在调用大多数COM组件的时候，就不必跨越单元的边界。除此之外，除非执行COM Interop调用，否则不会发生单元的初始化。这种方式的缺点在于，其他所有线程（包括Task的那些）都默认使用一个多线程单元（Multithreaded Apartment，MTA）。因此，从除了主线程之外的其他线程调用COM组件时，一定要非常小心。

> COM Interop不一定是由开发者显式执行的。微软在实现.NET Framework中的许多组件时，采取的做法是创建一个运行时可调用包装器（runtime callable wrapper，RCW），而不是用托管代码重写所有COM功能。因此，经常都会不知不觉地发出COM调用。为了确保这些调用始终都是从单线程的单元中发出，最好的办法就是使用System.STAThreadAttribute来修饰所有Windows Form可执行文件的Main方法。

19.3 小结

本章首先探讨了各种同步机制，以及如何利用各种类来避免出现竞态条件。讨论了lock关键字，它在幕后利用了System.Threading.Monitor。其他同步类包括System.Threading.Interlocked、System.Threading.Mutex、System.Threading.WaitHandle、重置事件、信号量和并发集合类。

虽然多线程编程方式一直在改进，多线程程序的同步仍然容易出问题。为此需要遵守许多最佳编程实践，包括坚持按相同顺序获取同步目标，以及用同步逻辑封装静态成员。

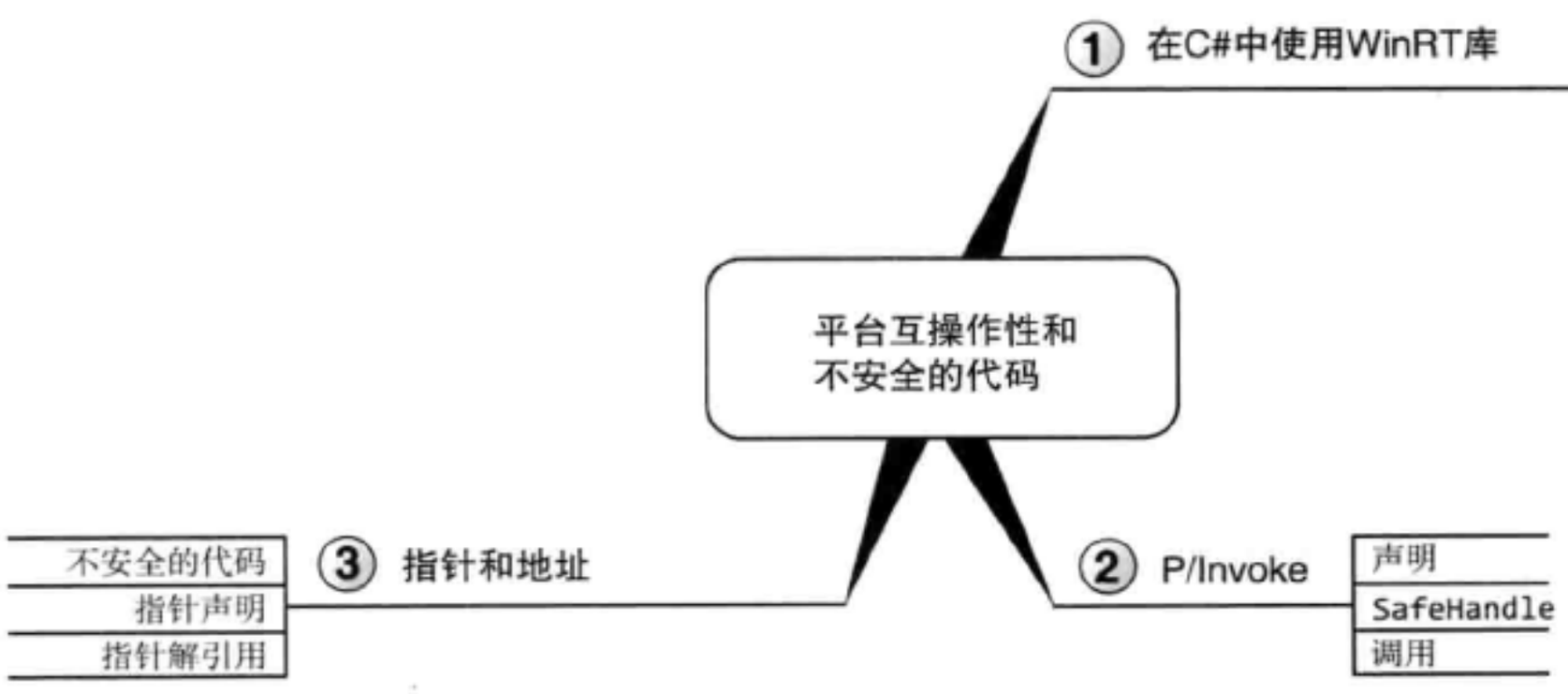
本章最后讨论了Task.Delay()方法，这个.NET 4.5 API在TAP的基础上实现了计时器。

下一章将探讨另一项相当复杂的.NET技术：如何使用P/Invoke将调用从.NET封送到非托管代码中。另外还讨论了不安全代码的概念；我们将利用这一概念来直接访问内存指针，如同在非托管代码（比如C++）中所做的那样。

第20章

平台互操作性和不安全的代码

C#功能强大，尤其是在和.NET库配合使用的时候。但它有时仍然“不给力”，造成只能放弃它所提供的所有安全性，退回到内存地址和指针的世界。C#通过4种方式对此提供支持。第一种方式是通过Windows Runtime（WinRT） API，它正在公开越来越多的操作系统函数，可直接在C# 5.0中使用这些函数。第二种方式是通过平台调用（Platform Invoke，P/Invoke）来调用非托管DLL所公开的API。第三种方式是通过**不安全的代码**（unsafe code），它允许我们访问内存指针和地址。第四种方式是通过本书不打算详述的COM Interop（COM互操作）。



本章首先简单讨论了WinRT存在的一些问题，开发人员在使用它之前务必注意。然后，讨论了与非托管代码的交互，以及不安全代码的使用。本章最后提供了一个小程序来判断计算机是不是虚拟机。代码要求执行以下操作。

（1）调用一个操作系统DLL，并请求分配一部分内存来执行指令。

（2）将一些汇编指令写入已分配的内存区域。

（3）将一个地址位置注入汇编指令。

（4）执行汇编程序代码。

这个例子演示了C#的强大能力，证明从C#和托管代码中仍然是可以访问非托管代码的。

20.1 在C#中使用WinRT库

Windows RT是Windows 8操作系统的一个版本，它只支持沉浸式Metro风格的应用程序，不支持传统桌面应用程序。 用于支持沉浸式应用程序的操作系统API是Windows Runtime，简称为WinRT。

虽然WinRT API本质上是非托管COM API，但却是用和.NET“运行时”一样的元数据格式来描述的。所以，WinRT不仅支持用非托管语言来写沉浸式Windows应用程序，还允许用C#这样的托管语言来写，这样就不需要用到本章剩余部分要讲述的各种P/Invoke技巧了。

由于进行了精心设计，C#开发人员看到WinRT API时会倍感亲切。但是，在写面向WinRT的C#程序时，仍有一些细微的地方需要注意。

20.1.1 具有自定义Add/Remove处理程序的WinRT事件

有多种方式可以实现“观察者”模式。如同之前讨论过的那样，C#中的事件通常作为多播委托类型的字段来实现。换言之，有一个委托类型的字段，那个委托可引用多个不同的方法。事件发生时就调用委托（所引用的）方法。要增删事件处理程序，只需新建一个多播委托，将字段值替换成新委托即可。

通过为事件使用+=和-=操作符，所有这些机制都会自动实现。C#还允许在类的用户增删事件处理程序时运行自定义代码，这是通过add和remove事件访问器方法来实现的。

从使用者的角度出发，WinRT事件没什么不同。可以在C#程序中和往常一样使用+=和-=从WinRT对象中增删事件处理程序；C#编译器在生成代码时会自动采用正确的WinRT机制。但是，针对自定义访问器，WinRT采用了和传统的C#程序稍有不同的机制，这影响到在C#中为WinRT类型编写自定义访问器的方式。

对于普通的C#事件，从事件删除一个委托时，委托作为remove访问器的隐藏value实参传递。add和remove访问器都不返回值。具有自定义访问器的WinRT事件则采用了稍微不同的机制：向事件添加委托时，add访问器返回一个“标志”（token）。从事件删除委托则要向remove访问器传递该标志（而不是委托）。为WinRT事件编写自定义的访问器时，必须遵循WinRT事件模式。

幸好，WinRT库提供了一个特殊的辅助类来跟踪这些“标志”及其对应的委托。代码清单20-1展示了这个模式。

代码清单20–1 WinRT事件模式

```
using System;
class WinRTEvent
{
  EventRegistrationTokenTable<EventHandler> table = null;
  public event EventHandler MyEvent
```

```
{
  add
  {
    return EventRegistrationTokenTable<EventHandler>
      .GetOrCreateEventRegistrationTokenTable(ref table)
      .AddEventHandler(value);
  }
  remove
  {
    return EventRegistrationTokenTable<EventHandler>
      .GetOrCreateEventRegistrationTokenTable(ref table)
      .RemoveEventHandler(value);
    }
  }
  void OnMyEvent()
  {
    EventHandler handler =
      EventRegistrationTokenTable<EventHandler>
        .GetOrCreateEventRegistrationTokenTable(ref table)
        .InvocationList;
      if (handler != null)
        handler(this, new EventArgs());
  }
}
```

如你所见，每次向事件添加、删除或调用处理程序都会创建一个表（如果没有的话），注意每个事件只有一个表变量。这个表负责管理从add访问器返回的“标志”与表中的多播委托之间的关系。写自己的程序时，只需将EventHandler类型替换成你的事件需要的委托类型，再为add和remove访问器添加想要执行的代码。

20.1.2 自动变换的接口

WinRT和普通.NET调用代码的另一个区别是某些常用接口在WinRT中有了稍有不同的名称和成员。C#编译器和.NET运行时了解这些情况，会自动生成代码将一个接口“变换”为另一个，从而最大程度地减轻开发人员的负担。最引人注目的两个例子是IEnumerable<T>，它在WinRT 中称为IIterable<T>；以及IDisposable，它在WinRT中称为ICloseable。

由于能自动变换，所以可以在需要IDisposable的任何地方（比如using语句）使用返回ICloseable的方法。类似地，无论使用C#标准接口还是WinRT版本，序列和集合的行为是一样的。

20.1.3 基于任务的异步

WinRT API不使用Task<T>来代表异步工作。（参见第18章详细了解如何使用Task<T>和C# 5 await操作符。）相反，它使用的是IAsyncAction<T>接口。该类型具有许多与Task<T>相同的功能。例如，它也支持取消、进度报告等。

C# 5 await操作符能很好地支持IAsyncAction<T>类型的操作数。但是，包含await操作符的C# 5 async方法仍然必须返回Task、Task<T>或者void。async方法不能返回IAsyncAction<T>。要将IAsyncAction<T>转换成Task<T>，只要在它上面调用AsTask()方法即可。

WinRT的其他问题主要涉及API的变化，对它们的详细讨论超出了本书的范围。但请记住一个重点，在WinRT中，.NET 4.5和更早版本中可用的所有高延迟同步方法都被放弃，剩下的只有*Async异步版本。

20.2 平台调用

无论开发人员尝试调用其现有非托管代码库，尝试访问操作系统的未由任何托管API公开的非托管代码，还是尝试避免类型检查和垃圾回收的运行时开销，从而发挥出一个特定算法的最大性能，最终都必然会调用到非托管代码。CLI通过P/Invoke提供了这个功能，它允许对非托管DLL所导出的函数执行API调用。

本节调用的所有API都是Windows API。虽然其他平台上没有提供相同的API，但开发人员仍可为其平台上的原生API使用P/Invoke，或者用P/Invoke调用自己的DLL（中的函数）。基本原则和语法都是一样的。

20.2.1 外部函数的声明

确定了要调用的目标函数以后，P/Invoke的下一步便是用托管代码声明函数。和类的所有普通方法一样，必须在类的上下文中声明目标API，但是要为它添加extern修饰符，从而把它声明为**外部函数**。代码清单20-2演示了具体做法。

代码清单20-2 声明一个外部方法

```
using System;
using System.Runtime.InteropServices;
class VirtualMemoryManager
{
  [DllImport("kernel32.dll", EntryPoint="GetCurrentProcess")]
  internal static extern IntPtr GetCurrentProcessHandle();
}
```

本例的类是VirtualMemoryManager，它将包含与内存管理相关的函数（函数直接由System.Diagnostics.Processor类提供，所以真正进行编码时没必要声明）。注意，方法返回的是一个IntPtr；该类型将在下一节进行解释。

extern方法永远不包含任何主体，而且几乎总是静态方法。是由附加在方法声明之前的DllImport特性（而不是方法主体）指向实现。该特性至少需要定义了函数的DLL的名称。“运行时”根据方法名来判断函数名。然而，也可以用EntryPoint命名参数来重写此默认行为，明确地提供一个函数名。.NET平台会自动尝试调用API的Unicode（...W）或者ASCII（...A）版本。

在本例中，外部函数GetCurrentProcess()获取当前进程的一个“伪句柄”（pseudohandle）。在调用中会使用这个伪句柄进行虚拟内存分配。以下是非托管声明：

```
HANDLE GetCurrentProcess();
```

20.2.2 参数的数据类型

假设开发人员已经确定了目标DLL和导出的函数，那么最困难的一步是标识或创建与外

部函数中的非托管数据类型对应的托管数据类型[①]。代码清单20-3展示了一个更难的API。

代码清单20-3 VirtualAllocEx() API

```
LPVOID VirtualAllocEx(
    HANDLE hProcess,   // The handle to a process. The
                       // function allocates memory within
                       // the virtual address space of this
                       // process.
    LPVOID lpAddress,  // The pointer that specifies a
                       // desired starting address for the
                       // region of pages that you want to
                       // allocate. If lpAddress is NULL,
                       // the function determines where to
                       // allocate the region.
    SIZE_T dwSize,     // The size of the region of memory to
                       // allocate, in bytes. If lpAddress
                       // is NULL, the function rounds dwSize
                       // up to the next page boundary.
    DWORD flAllocationType, // The type of memory allocation.
    DWORD flProtect);  // The type of memory allocation.
```

VirtualAllocEx()分配操作系统特别为代码执行或数据指定的虚拟内存。为了调用它，托管代码中要为每一种数据类型提供对应的定义。不过，在Win32编程中，HANDLE、LPVOID、SIZE_T和DWORD在CLI托管代码中通常是未定义的。因此，VirtualAllocEx()的C#定义如代码清单20-4所示。

代码清单20-4 在C#中声明VirtualAllocEx() API

```
using System;
using System.Runtime.InteropServices;
class VirtualMemoryManager
{
  [DllImport("kernel32.dll")]
  internal static extern IntPtr GetCurrentProcess();

  [DllImport("kernel32.dll", SetLastError = true)]
  private static extern IntPtr VirtualAllocEx(
      IntPtr hProcess,
      IntPtr lpAddress,
      IntPtr dwSize,
      AllocationType flAllocationType,
      uint flProtect);
}
```

托管代码的一个显著特征是，像int这样的基本数据类型不会因处理器而改变大小。不管是16位、32位还是64位处理器，int始终是32位。不过在非托管代码中，内存指针会随处理器而变化。因此，不要将HANDLE和LPVOID等类型映射到int，而应该把它们映射到System.IntPtr，其大小将依据处理器的内存布局而变化。本例还使用了一个AllocationType枚举类型，详情参见本章后面的20.1.8节。

代码清单20-3的一个有趣的地方在于，IntPtr不仅能用于存储指针，还能用于存储其

① 有关Win32 API声明的一个非常有用的网上资源是www.pinvoke.net。这个网站为许多API提供了一个很好的起点。开发人员从头编写一个外部API调用的时候，这个网站有助于避免一些容易被忽视的问题。

他东西，比如数量。IntPtr并非只能表示“存储在整数中的指针”，它还能表示“指针大小的整数”。IntPtr并非一定要包含指针，它只需包含指针大小的内容即可。许多东西都只有指针大小，但并非就是指针。

20.2.3 使用ref而不是指针

很多情况下，非托管代码会为传引用（pass-by-reference）参数使用指针。在这些情况下，P/Invoke不要求你在托管代码中将数据类型映射到指针。相反，应该将对应的参数映射为ref（或out），具体取决于参数是输入/输出，还是仅输出。在代码清单20-5中，参数lpflOldProtect便是一个例子，它的数据类型是PDWORD，返回的是一个指针，该指针指向一个变量，该变量接收的是“指定页区域的第一页上一个访问保护”。

代码清单20-5 使用ref和out而不是指针

```
class VirtualMemoryManager
{
  // ...
  [DllImport("kernel32.dll", SetLastError = true)]
  static extern bool VirtualProtectEx(
      IntPtr hProcess, IntPtr lpAddress,
      IntPtr dwSize, uint flNewProtect,
      ref uint lpflOldProtect);
}
```

文档中lpflOldProtect被定义为[out]参数（虽然签名没有强行要求），但在随后的描述中，又指出该参数必须指向一个有效的变量，而不能是NULL。文档中出现这种自相矛盾的说法，难免会让人产生迷惑，但这是一个很常见的情况。针对这种情况，我们的指导原则是为P/Invoke类型的参数使用ref，而不是out，因为被调用者总是能忽略随同ref传递的数据，反之则不然。

其他参数与VirtualAllocEx()差不多，唯一例外的是lpAddress，它是从VirtualAllocEx()返回的地址。此外，flNewProtect指定了确切的内存保护类型：page execute、page read-only等。

20.2.4 为顺序布局使用StructLayoutAttribute

有些API涉及的类型没有对应的托管类型。要调用这些API，需要用托管代码重新声明类型。例如，可以使用托管代码来声明非托管的COLORREF结构，如代码清单20-6所示。

代码清单20-6 从非托管的struct中声明类型

```
[StructLayout(LayoutKind.Sequential)]
struct ColorRef
{
  public byte Red;
  public byte Green;
  public byte Blue;
  // Turn off warning about not accessing Unused.
  #pragma warning disable 414
  private byte Unused;
  #pragma warning restore 414
```

```
    public ColorRef(byte red, byte green, byte blue)
    {
        Blue = blue;
        Green = green;
        Red = red;
        Unused = 0;
    }
}
```

微软的Windows所有与颜色相关的API都用COLORREF来表示RGB颜色（红、绿、蓝）。以上声明的关键之处在于StructLayoutAttribute。默认情况下，托管代码可以优化类型的内存布局，所以，内存布局可能不是从一个字段到另一个字段顺序存储。为了强制顺序布局，使类型能够直接映射，而且可以在托管和非托管代码之间逐位地复制，你需要添加StructLayout Attribute，并指定LayoutKind.Sequential枚举值。（从文件流读写数据时，如果要求一个顺序布局，那么也应该这样指定。）

由于struct的非托管（C++）定义没有映射到C#定义，所以在非托管结构和托管结构之间不存在直接的映射关系。开发人员应遵循常规的C#指导原则来操作，即类型在行为上是像值类型还是像引用类型，以及大小是否很小（约小于16字节）。

20.2.5 错误处理

Win32 API编程的一个不便之处在于，错误经常以不一致的方式来报告。例如，有的API返回一个值（0、1、false等）来指示错误，有的API则以某种方式来设置一个out参数。除此之外，为了了解错误的细节，还需要额外调用GetLastError() API，再调用FormatMessage()来获取对应的错误消息。总之，非托管代码中的Win32错误报告很少通过异常来生成。

幸好，P/Invoke设计者为此提供了相应的处理机制。要启用该机制，DllImport特性的SetLastError命名参数要设为true。这样就可实例化一个System.ComponentModel.Win32Exception()。在P/Invoke调用之后，会自动用Win32错误数据来初始化它，如代码清单20-7所示。

代码清单20-7 Win32错误处理

```
class VirtualMemoryManager
{
  [DllImport("kernel32.dll", ", SetLastError = true)]
  private static extern IntPtr VirtualAllocEx(
      IntPtr hProcess,
      IntPtr lpAddress,
      IntPtr dwSize,
      AllocationType flAllocationType,
      uint flProtect);

// ...
[DllImport("kernel32.dll", SetLastError = true)]
static extern bool VirtualProtectEx(
      IntPtr hProcess, IntPtr lpAddress,
      IntPtr dwSize, uint flNewProtect,
      ref uint lpflOldProtect);
```

```
[Flags]
private enum AllocationType : uint
{
    // ...
}

[Flags]
private enum ProtectionOptions
{
    // ...
}

[Flags]
private enum MemoryFreeType
{
    // ...
}

public static IntPtr AllocExecutionBlock(
    int size, IntPtr hProcess)
{
    IntPtr codeBytesPtr;
    codeBytesPtr = VirtualAllocEx(
        hProcess, IntPtr.Zero,
        (IntPtr)size,
        AllocationType.Reserve | AllocationType.Commit,
        (uint)ProtectionOptions.PageExecuteReadWrite);

    if (codeBytesPtr == IntPtr.Zero)
    {
        throw new System.ComponentModel.Win32Exception();
    }

    uint lpflOldProtect = 0;
    if (!VirtualProtectEx(
        hProcess, codeBytesPtr,
        (IntPtr)size,
        (uint)ProtectionOptions.PageExecuteReadWrite,
        ref lpflOldProtect))
    {
        throw new System.ComponentModel.Win32Exception();
    }
    return codeBytesPtr;
}

public static IntPtr AllocExecutionBlock(int size)
{
    return AllocExecutionBlock(
        size, GetCurrentProcessHandle());
}
}
```

这样一来，开发人员就可以提供每个API使用的自定义错误检查，同时仍然可以通过标准方式报告错误。

代码清单20-2和代码清单20-4将P/Invoke方法声明为内部或私有。除了最简单的API，通常应该将方法封装到公共包装器中，从而降低P/Invoke API调用的复杂性。这

样可以增强API的可用性，同时更有利于转向面向对象的类型结构。代码清单20-7的AllocExecutionBlock()声明就是一个很好的例子。

规范

要围绕非托管方法创建公共托管包装器；这种非托管方法使用了托管代码的约定，比如结构化的异常处理。

20.2.6 使用SafeHandle

P/Invoke经常涉及资源，比如一个窗口句柄。在用完此类资源之后，代码需要清理它们。但是，不要强迫开发人员记住这一点，并每次都手动编写代码，而是应该提供实现IDisposable接口和终结器的类。例如，在代码清单20-8中，VirtualAllocEx()和VirtualProtectEx()会返回一个地址，该资源需调用VirtualFreeEx()进行清理。为了提供内建的支持，需要定义一个VirtualMemoryPtr类，该类派生自System.Runtime.InteropServices.SafeHandle。

代码清单20-8 使用SafeHandle的托管资源

```
public class VirtualMemoryPtr :
  System.Runtime.InteropServices.SafeHandle
{
  public VirtualMemoryPtr(int memorySize) :
      base(IntPtr.Zero, true)
  {
      ProcessHandle =
          VirtualMemoryManager.GetCurrentProcessHandle();
      MemorySize = (IntPtr)memorySize;
      AllocatedPointer =
          VirtualMemoryManager.AllocExecutionBlock(
          memorySize, ProcessHandle);
      Disposed = false;
  }

  public readonly IntPtr AllocatedPointer;
  readonly IntPtr ProcessHandle;
  readonly IntPtr MemorySize;
  bool Disposed;

  public static implicit operator IntPtr(
      VirtualMemoryPtr virtualMemoryPointer)
  {
      return virtualMemoryPointer.AllocatedPointer;
  }

  // SafeHandle abstract member
  public override bool IsInvalid
  {
      get
      {
          return Disposed;
      }
  }
```

```
// SafeHandle abstract member
protected override bool ReleaseHandle()
{
    if (!Disposed)
    {
        Disposed = true;
        GC.SuppressFinalize(this);
        VirtualMemoryManager.VirtualFreeEx(ProcessHandle,
            AllocatedPointer, MemorySize);
    }
    return true;
  }
}
```

System.Runtime.InteropServices.SafeHandle包含两个抽象成员：IsInvalid和ReleaseHandle()。在后者中，可以放入对资源进行清理的代码；前者则指出是否已经执行了资源清理代码。

有了VirtualMemoryPtr之后，内存的分配就变得很简单。只需实例化类型，然后指定所需的内存分配即可。

高级主题：显式使用IDisposable来代替SafeHandle

在C# 1.0中，System.Runtime.InteropServices.SafeHandle是不可用的。取而代之的是IDisposable的一个自定义实现，如代码清单20-9所示。

代码清单20-9 没有SafeHandle，但使用了IDisposable的托管资源

```
public struct VirtualMemoryPtr : IDisposable
{
  public VirtualMemoryPtr(int memorySize)
  {
      ProcessHandle =
          VirtualMemoryManager.GetCurrentProcessHandle();
      MemorySize = (IntPtr)memorySize;
      AllocatedPointer =
          VirtualMemoryManager.AllocExecutionBlock(
          memorySize, ProcessHandle);
      Disposed = false;
  }

  public readonly IntPtr AllocatedPointer;
  readonly IntPtr ProcessHandle;
  readonly IntPtr MemorySize;
  bool Disposed;

  public static implicit operator IntPtr(
      VirtualMemoryPtr virtualMemoryPointer)
  {
      return virtualMemoryPointer.AllocatedPointer;
  }

  #region IDisposable Members
  public void Dispose()
  {
```

```
            if (!Disposed)
            {
                Disposed = true;
                GC.SuppressFinalize(this);
                VirtualMemoryManager.VirtualFreeEx(ProcessHandle,
                    AllocatedPointer, MemorySize);
            }
        }
        #endregion
    }
```

为了使VirtualMemoryPtr具有值类型的行为，需要将其实现为struct。然而，这样做的后果是可以没有终结器，因为值类型不由垃圾回收器进行管理。这意味着，使用此类型的开发人员必须自己记住去清理资源。如果忘记，将没有任何补救措施。

第二个限制是不要将实例传递或复制到方法外部。这是用IDisposable实现类型时的常规导原则。它们的作用域应限制在using语句内，而且不应作为参数传给其他方法，因为那些方法可能会延长它们的生存期，从而超过using的范围。

20.2.7 外部函数的调用

声明好P/Invoke函数以后，就可以像调用其他任何类成员一样调用它们。然而，关键之处在于，导入的DLL必须在路径内，其中包括可执行文件的目录，以使其能成功加载。代码清单20-7和代码清单20-8已经对此进行了演示。不过，它们要依赖于某些常量。

由于flAllocationType和flProtect是标志（flag），所以最好的做法是为它们提供常量或枚举。但是，不要期待由调用者定义这些东西。应该将它们作为API声明的一部分来提供。如代码清单20-10所示。

代码清单20-10 将API封装到一起

```
class VirtualMemoryManager
{
  // ...

  /// <summary>
  /// The type of memory allocation. This parameter must
  /// contain one of the following values.
  /// </summary>
  [Flags]
  private enum AllocationType : uint
  {
  /// <summary>
  /// Allocates physical storage in memory or in the
  /// paging file on disk for the specified reserved
  /// memory pages. The function initializes the memory
  /// to zero.
  /// </summary>
  Commit = 0x1000,
  /// <summary>
  /// Reserves a range of the process's virtual address
  /// space without allocating any actual physical
  /// storage in memory or in the paging file on disk.
  /// </summary>
  Reserve = 0x2000,
  /// <summary>
```

```
        /// Indicates that data in the memory range specified by
        /// lpAddress and dwSize is no longer of interest. The
        /// pages should not be read from or written to the
        /// paging file. However, the memory block will be used
        /// again later, so it should not be decommitted. This
        /// value cannot be used with any other value.
        /// </summary>
        Reset = 0x80000,
        /// <summary>
        /// Allocates physical memory with read-write access.
        /// This value is solely for use with Address Windowing
        /// Extensions (AWE) memory.
        /// </summary>
        Physical = 0x400000,
        /// <summary>
        /// Allocates memory at the highest possible address.
        /// </summary>
        TopDown = 0x100000,
    }

    /// <summary>
    /// The memory protection for the region of pages to be
    /// allocated.
    /// </summary>
    [Flags]
    private enum ProtectionOptions : uint
    {
        /// <summary>
        /// Enables execute access to the committed region of
        /// pages. An attempt to read or write to the committed
        /// region results in an access violation.
        /// </summary>
        Execute = 0x10,
        /// <summary>
        /// Enables execute and read access to the committed
        /// region of pages. An attempt to write to the
        /// committed region results in an access violation.
        /// </summary>
        PageExecuteRead = 0x20,
        /// <summary>
        /// Enables execute, read, and write access to the
        /// committed region of pages.
        /// </summary>
        PageExecuteReadWrite = 0x40,
        // ...
    }

    /// <summary>
    /// The type of free operation
    /// </summary>
    [Flags]
    private enum MemoryFreeType : uint
    {
        /// <summary>
        /// Decommits the specified region of committed pages.
        /// After the operation, the pages are in the reserved
        /// state.
        /// </summary>
        Decommit = 0x4000,
```

```
        /// <summary>
        /// Releases the specified region of pages. After this
        /// operation, the pages are in the free state.
        /// </summary>
        Release = 0x8000
    }

    // ...
}
```

枚举的优势在于它们将每个值都组合到一起。另外，它们还将作用域严格限制在这些值之内。

20.2.8 用包装器简化API调用

无论错误处理、结构还是常量值，优秀的API开发人员都应该提供一个简化的托管API将底层的 Win32 API包装起来。例如，代码清单20-11用简化了调用的公共版本重载了VirtualFreeEx()。

代码清单20-11 包装底层API

```
class VirtualMemoryManager
{
  // ...

  [DllImport("kernel32.dll", SetLastError = true)]
  static extern bool VirtualFreeEx(
      IntPtr hProcess, IntPtr lpAddress,
      IntPtr dwSize, IntPtr dwFreeType);
public static bool VirtualFreeEx(
      IntPtr hProcess, IntPtr lpAddress,
      IntPtr dwSize)
{
      bool result = VirtualFreeEx(
          hProcess, lpAddress, dwSize,
          (IntPtr)MemoryFreeType.Decommit);
      if (!result)
      {
          throw new System.ComponentModel.Win32Exception();
      }
      return result;
}
public static bool VirtualFreeEx(
      IntPtr lpAddress, IntPtr dwSize)
{
      return VirtualFreeEx(
          GetCurrentProcessHandle(), lpAddress, dwSize);
}

[DllImport("kernel32", SetLastError = true)]
static extern IntPtr VirtualAllocEx(
      IntPtr hProcess,
      IntPtr lpAddress,
      IntPtr dwSize,
      AllocationType flAllocationType,
      uint flProtect);

// ...
```

```
}
```

20.2.9 函数指针映射到委托

P/Invoke的最后一个要点是非托管代码中的函数指针映射到托管代码中的委托。例如，为了设置一个Microsoft Windows计时器，需要提供一个一旦过期能由计时器回调的函数指针。具体地说，需要传递一个与回调签名匹配的委托实例。

20.2.10 编码规范

鉴于P/Invoke的特点，写这种代码时应谨记以下编码规范。

规范

不要无谓地重复现有的、能执行非托管API功能的托管类。

要将外部方法声明为私有或内部。

要提供使用了托管约定的公共包装器方法，包括结构化的异常处理、为特殊值使用枚举等。

要为不必要的参数选择默认值来简化包装器方法。

要用SetLastErrorAttribute将使用SetLastError错误码的API转换成引发Win32Exception的方法。

要扩展SafeHandle或实现IDisposable并创建终结器来确保非托管资源被高效率地清理。

要在非托管API需要函数指针的时候，使用和所需方法的签名匹配的委托类型。

要尽量使用ref参数而不是指针类型。

20.3 指针和地址

有时，开发人员希望能够直接访问和操纵内存，以及直接用指针来定位内存。这对特定的操作系统交互以及某些对时间敏感的算法来说是必要的。为了提供这方面的支持，C#要求使用“不安全的代码”结构。

20.3.1 不安全的代码

C#的一个突出优点在于它是强类型的，而且支持运行时的类型检查。然而，你仍然可以绕过这个机制，直接操纵内存和地址。例如，在操作内存映射设备的时候，或者想要实现一个对时间敏感的算法的时候，就需要这样做。为此，只需将代码区域指定为unsafe（不安全）。

不安全的代码是一个显式的代码块和编译选项，如代码清单20-12所示。unsafe修饰符对生成的CIL代码本身没有影响。它只是一个预编译指令，作用是向编译器指出允许在不安全的代码块内操纵指针和地址。此外，不安全并不意味着非托管。

代码清单20-12 为不安全的代码指定方法

```
class Program
{
  unsafe static int Main(string[] args)
```

```
    {
        // ...
    }
}
```

可将unsafe用作类型或者类型内部的特定成员的修饰符。

除此以外，C#允许用unsafe标记代码块，指出其中允许不安全的代码，如代码清单20-13所示。

代码清单20-13 指定不安全代码块

```
class Program
{
  static int Main(string[] args)
  {
      unsafe
      {
          // ...
      }
  }
}
```

unsafe块中的代码可以包含指针之类的不安全构造。

> **注意**
>
> 必须向编译器显式指明要支持不安全的代码。

在命令行上，这要求使用/unsafe开关。例如，编译上述代码需要使用如输出20-1所示的命令。

输出20-1

```
csc.exe /unsafe Program.cs
```

使用Visual Studio时，可在项目属性窗口的“生成”选项卡中勾选“允许不安全的代码”。

之所以需要使用/unsafe开关，是因为不安全的代码使缓冲区溢出成为可能，同时也可能暴露其他安全漏洞。/unsafe开关意味着能直接操纵内存和执行非托管的指令。因此，需要显式指定/unsafe开关，表明自己认同了随之而来的安全风险。

20.3.2 指针的声明

代码块标记为unsafe之后，接着要知道如何写不安全的代码。首先，不安全的代码允许声明指针。来看看下面的例子。

```
byte* pData;
```

假设pData不为null，那么它的值指向的是包含一个或多个连续字节的内存位置。pData的值代表这些字节的内存地址。符号*之前指定的类型是**被引用物**（referent）的类型，或者说是指针指向的那个位置存储的值的类型。在本例中，pData是指针，而byte是被引用物类型，如图20-1所示。

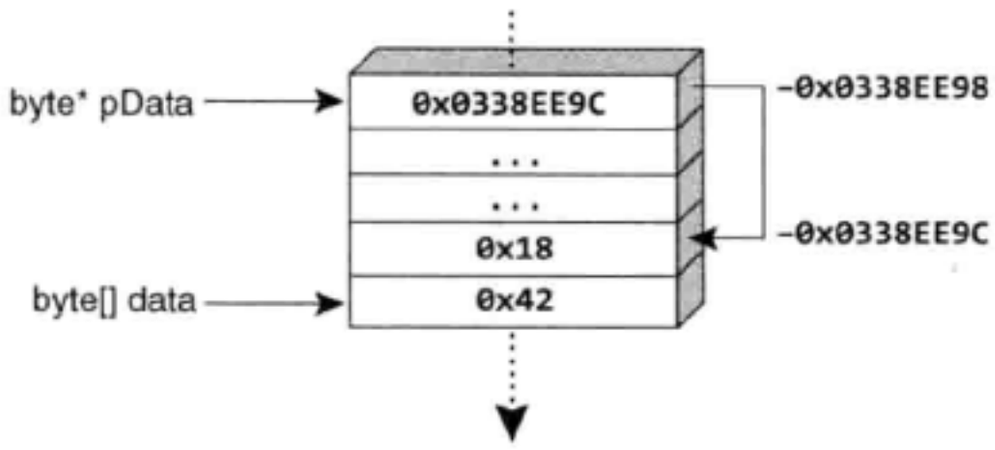


图20-1 指针包含的只是实际数据所在的地址

由于指针恰好是指向内存地址的整数，所以不会被垃圾回收。C#不允许非托管类型以外的被引用物类型。换言之，不能是引用类型，不能是泛型类型，而且内部不能包含引用类型。所以，以下声明是无效的：

```
string* pMessage;
```

以下声明也不正确：

```
ServiceStatus* pStatus;
```

其中，ServiceStatus的定义如代码清单20-14所示。问题仍然是ServiceStatus中包含了一个string字段。

代码清单20-14 无效的被引用物类型示例

```
struct ServiceStatus
{
  int State;
  string Description;  // Description is a reference type
}
```

除了只包含非托管类型的自定义结构，有效的被引用物类型还包括枚举、预定义值类型（sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal和bool）以及指针类型（如byte**）。void*指针也是有效的，它代表指向未知类型的指针。

语言对比：C/C++——指针声明

在C/C++中，要像下面这样一次声明多个指针：

```
int *p1, *p2;
```

注意p2之前的*，它使p2成为一个int*，而不是一个int。相比之下，C#总是把*和数据类型放在一块儿，如下所示：

```
int* p1, p2;
```

结果是两个int*类型的变量。在两种语言中，在一个语句中声明多个数组的语法是一致的：

```
int[] array1, array2;
```

指针是一种全新的类型。有别于结构、枚举和类，指针的终极基类不是System.Object，甚至不能转换成System.Object。相反，它们能转换成System.IntPtr（后者能转换成System.Object）。

20.3.3 指针的赋值

代码定义好指针后，在访问它之前必须为它赋值。就像其他引用类型一样，指针可以包含null值，这也是它们的默认值。指针保存的是一个位置的地址。因此，为了对指针进行赋值，首先必须获得数据的地址。可以显式将一个int或long转换为指针。但是，除非有办法在执行时获得一个特定的数据值的地址，否则很少能够这样做。相反，需要使用地址操作符（&）来获取值类型的地址，如下所示：

```
byte* pData = &bytes[0]; // 编译错误
```

问题在于托管环境中的数据可能发生移动，因而导致地址无效。编译器显示的错误消息是："只能获取固定语句初始值设定项内的未固定表达式的地址"[①]。在本例中，被引用的字节出现在一个数组内，而数组是引用类型（在内存中可能移动的类型）。引用类型出现在内存堆（heap）上，可以被垃圾回收或者重新分配。对一个可移动类型中的值类型字段进行引用，会发生类似的问题：

```
int* a = &"message".Length;
```

无论哪种方式，为了将数据的地址赋给指针，要求如下。

- 数据必须属于一个变量。
- 数据必须是非托管类型。
- 变量需要用fixed固定，不能移动。

如果数据是一个非托管变量类型，但是不固定，就使用固定语句固定可移动的变量。

1. 固定数据

要获取可移动数据项的地址，首先必须把它固定下来，如代码清单20-15所示。

代码清单20-15 固定语句

```
byte[] bytes = new byte[24];
fixed (byte* pData = &bytes[0])  // pData = bytes also allowed
{
  // ...
}
```

在fixed限定的代码块中，赋值的数据不会再移动。在本例中，bytes会固定不动（至少在fixed语句结束之前如此）。

fixed语句要求在其作用域内声明指针变量。这样可以防止当数据不再固定的时候，访问到fixed语句外部的变量。然而，最终是由程序员来保证不会将指针赋给在fixed语句的范围之外也能生存的变量（API调用中可能有这样的变量）。不安全代码的"不安全"是有原因的。要确保安全地使用指针，不要依赖"运行时"来帮助你确保安全性。类似地，对于在方法调用之后就不会生存的数据来说，使用ref或out参数也会出现问题。由于string是无效的被引用物类型，所以定义string指针似乎是无效的。然而，和C++一样，在内部，string本质上就是指针，它指向字符数组的第一个字符。而且可以使用char*来声明字符指针。因此，C#允许在fixed语句中声明char*类型的指针，并可以把它赋给一个string。fixed语句防止字符串在指针生存期内移动。类似地，它允许任何可移动的类型隐式转换为其他类型的指针，只要是在一个fixed语句之内。

可以使用缩写的bytes来取代冗长的&bytes[0]赋值，如代码清单20-16所示。

① 英文版显示"You can only take the address of [an] unfixed expressio inside a fixed statement initializer"。——译者注

代码清单20-16 没有地址或数组索引值的固定语句

```
byte[] bytes = new byte[24];
fixed (byte* pData = bytes)
{
  // ...
}
```

取决于执行频率和执行的时机，固定语句可能会导致内存堆中出现碎片（fragmentation），这是由于垃圾回收器不能压缩[①]已固定的对象。为缓解这个问题，最好的做法是在执行前期就固定好代码块，而且宁可固定较少的几个大块，也不要固定许多的小块。遗憾的是，这又和另一个原则发生了冲突，即“应该尽量缩短固定时间，降低在数据固定期间发生垃圾回收的几率”。.NET 2.0（及更高版本）添加了一些能避免过多碎片的代码，从而在某种程度上缓解了这方面的问题。

有时需要在方法主体中固定一个对象，并保持固定直至调用另一个方法。用fixed语句做不到这一点。这时可以使用GCHandle对象提供的方法来不确定地固定对象。但若非绝对必要，否则不应这样做。长时间固定对象很容易造成垃圾回收器不能高效地压缩内存。

2. 在栈上分配

应该为一个数组使用fixed语句，防止垃圾回收器移动数据。然而，另一种做法是在调用栈上分配数组。栈分配的数据不会被垃圾回收，也不会被终结器清理。和引用类型一样，要求stackalloc（栈分配）数据是非托管类型的数组。例如，你可以不在堆上分配一个byte数组，而是把它放在调用栈上，如代码清单20-17所示。

代码清单20-17 在调用栈上分配数据

```
byte* bytes = stackalloc byte[42];
```

由于数据类型是非托管类型的数组，所以“运行时”可以为该数组分配一个固定大小的缓冲区，并在指针越界的时候回收该缓冲区。具体地说，它会分配sizeof(T) * E，其中的E是数组大小，T是引用类型。由于只能为非托管类型的数组使用stackalloc，所以“运行时”为了回收缓冲区并把它返还给系统，只需对栈执行一次展开，从而避免了遍历f-reachable队列（参见第9章的垃圾回收和终结器部分）并对reachable（可达）数据进行压缩的复杂性。因此，没有办法显式地释放stackalloc数据。

注意，栈是一种宝贵的资源。耗尽栈空间会造成程序崩溃。应尽一切努力防止把它耗尽。如果程序真的耗尽了栈空间，最理想的情况是程序立即关闭/崩溃。一般情况下，程序只有不到1 MB的栈空间（实际可能更少）。所以，注意不要在栈上分配随意大小的缓冲区。

20.3.4 指针的解引用

为了访问指针引用的一个类型的值，要求你解引用指针，即在指针类型之前添加一个间接寻址操作符*。例如以下语句：

```
byte data = *pData;
```

它的作用是解引用出pData引用的byte所在的位置，并返回那个位置上的一个byte。

① 此压缩非彼压缩，这里只是约定俗成地将compact翻译成“压缩”。不要以为“压缩”后内存会增多。相反，这里的“压缩”更接近于“碎片整理”。事实上，compact正确的意思是“变得更紧凑”。但从20世纪80年代起，人们就把它看成是compress的近义词而翻译成“压缩”，以讹传讹至今。——译者注

在不安全的代码中这样做会使本来“不可变”的字符串变得可以被修改，如代码清单20-18所示。虽然不建议这样做，但它确实揭示了执行低级内存处理的可能性。

代码清单20-18 修改“不可变”字符串

```
string text = "S5280ft";
Console.Write("{0} = ", text);
unsafe  // Requires /unsafe switch.
{
  fixed (char* pText = text)
  {
      char* p = pText;
      *++p = 'm';
      *++p = 'i';
      *++p = 'l';
      *++p = 'e';
      *++p = ' ';
      *++p = ' ';
  }
}
Console.WriteLine(text);
```

代码清单20-18的结果如输出20-2所示。

输出20-2

```
S5280ft = Smile
```

这个例子获取初始地址，并使用前递增操作符，使这个地址递增被引用物类型的大小（sizeof(char)）。接着，使用间接寻址操作符*解引用出地址，并为这个地址指派一个不同的字符。类似地，对指针使用“+”和“-”操作符，会使地址增大或减小sizeof(T)的量，其中T是被引用物的类型。

类似地，比较操作符（==、!=、<、>、<=和=>）也可以用于对指针进行比较，它们实际会转变成地址位置值的比较。

不能对void*类型的指针应用解引用操作符。void*数据类型代表的是指向一个未知类型的指针。由于数据类型未知，所以不能解引用到另一种类型。相反，为了访问void*引用的数据，必须把它转换成其他任何指针类型的变量，然后对后一种类型执行解引用。

可以使用索引操作符而不是间接寻址操作符来实现与代码清单20-18相同的行为，如代码清单20-19所示。

代码清单20-19 在不安全的代码中用索引操作符修改“不可变”字符串

```
string text;
text = "S5280ft";
Console.Write("{0} = ", text);

unsafe  // Requires /unsafe switch.
{
  fixed (char* pText = text)
  {
      pText[1] = 'm';
      pText[2] = 'i';
      pText[3] = 'l';
      pText[4] = 'e';
```

```
            pText[5] = ' ';
            pText[6] = ' ';
        }
    }
    Console.WriteLine(text);
```

代码清单20-19的结果如输出20-3所示。

输出20-3

```
S5280ft = Smile
```

代码清单20-18和代码清单20-19中的修改会导致不可预期的行为。例如，假定在Console. WriteLine()语句之后，重新为text赋值"S5280ft"，然后重新显示text，那么输出结果仍是Smile，这是由于两个相同的字符串字面量的地址会优化成由两个变量共同引用的一个字符串字面量。在代码清单20-18的不安全代码之后，即使明确执行以下赋值：

```
text = "S5280ft";
```

但在内部，字符串赋值实际是一次地址赋值，所赋的地址仍然是修改过的"S5280ft"位置，所以text永远不会被设置成你希望的值。

20.3.5 访问被引用物类型的成员

指针解引用将生成指针基础类型的变量。然后可以使用成员访问“点”操作符来访问基础类型的成员。然而，根据操作符优先级规则，*x.y等价于*(x.y)，而这可能不是你所希望的。如果x是一个指针，正确的代码就是(*x).y。当然，这个语法并不好看。为了更容易地访问解引用的指针的成员，C#提供了特殊的成员访问修饰符：x->y是(*x).y的简化形式，如代码清单20-20所示。

代码清单20-20 直接访问被引用物类型的成员

```
unsafe
{
  Angle angle = new Angle(30, 18, 0);
  Angle* pAngle = &angle;
  System.Console.WriteLine("{0}° {1}' {2}\"",
      pAngle->Hours, pAngle->Minutes, pAngle->Seconds);
}
```

代码清单20-20的输出结果如输出20-4所示。

输出20-4

```
30° 18' 0
```

20.3.6 通过委托执行不安全的代码

本章最后提供一个完整的例子，它代表了用C#所能做的最“不安全”的事情：获取内存块指针，用机器码字节填充它，让委托引用新代码，并执行委托。这段汇编代码判断机器是在虚拟机还是真正的计算机上执行。如果是虚拟机，就显示“Inside Matrix!”，如代码清单 20-21所示。

初学者主题：什么是虚拟机

虚拟机通过在主机操作系统上运行并同主机硬件进行交互的一个软件来虚拟或仿真运行。例如，虚拟机软件（比如VMWare Workstation和Microsoft Virtual PC）可以安装到运行较新版本Windows的计算机上。安装好后，用户可以在软件中配置一台客户计算机，启动它，并在上面安装操作系统，就好像使用的是一台真正的计算机。

代码清单20-21 指定不安全代码块

```
using System.Runtime.InteropServices;

class Program
{
  unsafe static int Main(string[] args)
  {
      // Assign redpill
      byte[] redpill = {
          0x0f, 0x01, 0x0d,       // asm SIDT instruction
          0x00, 0x00, 0x00, 0x00, // placeholder for an address
          0xc3};                  // asm return instruction

      unsafe
      {
          fixed (byte* matrix = new byte[6],
              redpillPtr = redpill)
      {
          // Move the address of matrix immediately
          // following the SIDT instruction of memory.
          *(uint*)&redpillPtr[3] = (uint)&matrix[0];

          using (VirtualMemoryPtr codeBytesPtr =
              new VirtualMemoryPtr(redpill.Length))
          {
              Marshal.Copy(
                  redpill, 0,
                  codeBytesPtr, redpill.Length);

              MethodInvoker method =
                  (MethodInvoker)Marshal.GetDelegateForFunctionPointer(
                  codeBytesPtr, typeof(MethodInvoker));

              method();
          }
          if (matrix[5] > 0xd0)
          {
              Console.WriteLine("Inside Matrix!\n");
              return 1;
          }
          else
          {
              Console.WriteLine("Not in Matrix.\n");
              return 0;
```

```
                    }
                } // fixed
            } // unsafe
        }
    }
```

代码清单20-21的结果如输出20-5所示。

输出20–5

```
Inside Matrix!
```

20.4 小结

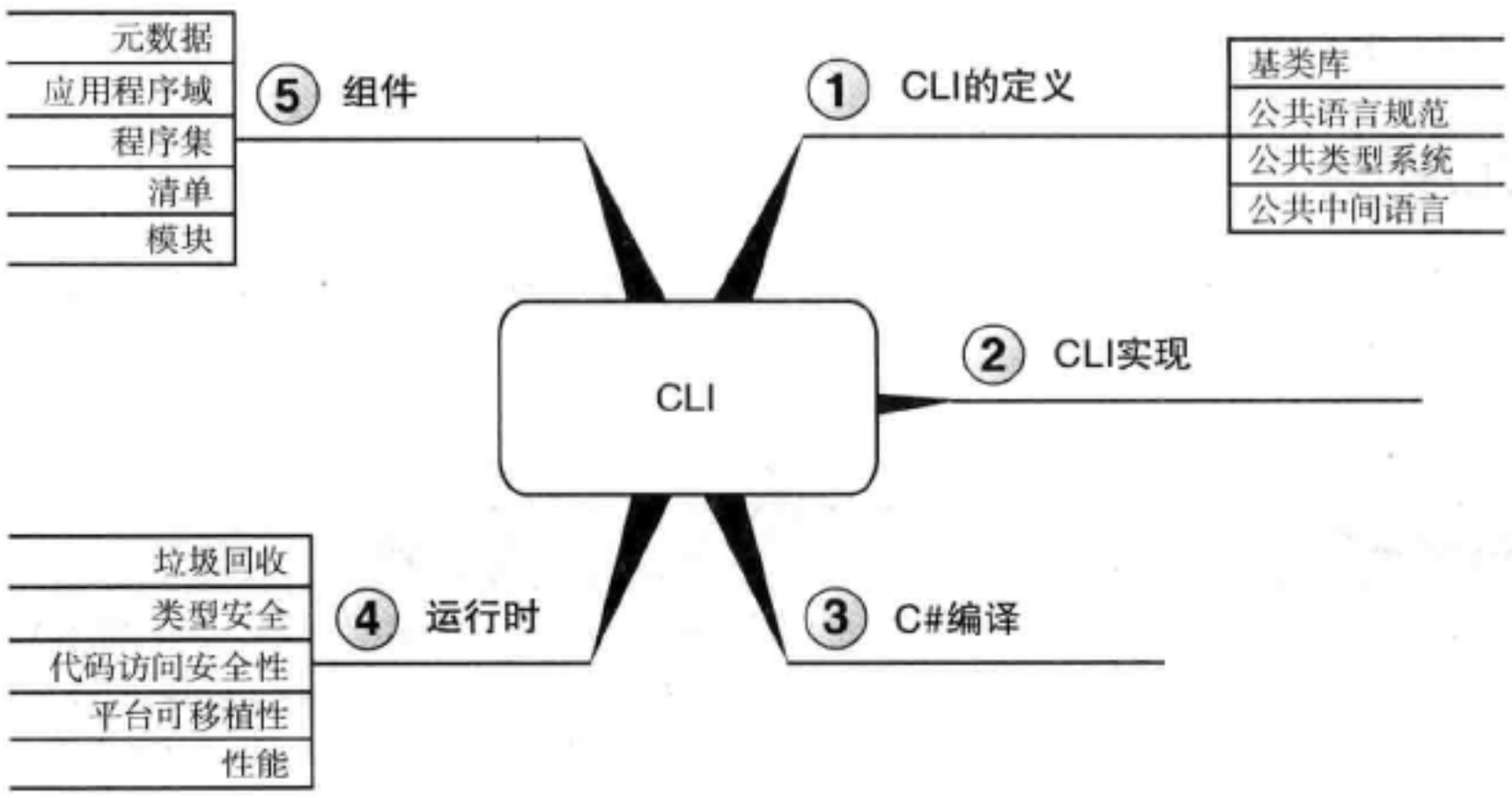
本书之前已经展示了C#语言的强大功能、灵活性、一致性以及精妙的结构。本章则证明虽然语言提供了如此高级的编程功能，但还是能执行一些很底层的操作。

结束本书之前，还将用一章的篇幅来简单描述底层执行平台，并把重心从C#语言本身转向C#程序所依托的一个更宽广的平台上。

第21章

CLI

除了语法本身，C#程序员还必须关心C#程序的执行环境。本章要讨论C#如何处理内存分配和回收，如何执行类型检查，如何与其他编程语言互操作，如何跨平台执行，以及如何支持元数据的编程。换句话说,本章要研究C#语言编译时和执行时都要依赖的公共语言基础结构（Common Language Infrastructure，CLI）。本章描述了在运行时管理C#程序的执行引擎，介绍了C#如何与同一个执行引擎管辖的各种语言相适应。由于C#语言与这个基础结构密切相关，所以这个基础结构的大多数功能都可以在C#中使用。



21.1 CLI的定义

C#生成的不是处理器能直接解释的指令，而是一种中间语言指令。这种中间语言就是**公共中间语言**（Common Intermediate Language，CIL）。第二个编译步骤通常在执行时发生。在这个步骤中，CIL被转换为处理器能理解的**机器码**（machine code）。然而，代码要

想执行，仅仅转换为机器码是不够的。C#程序还需要在一个代理的上下文中执行。负责管理C#程序执行的代理就是**虚拟执行系统**（Virtual Execution System，VES），它的一个更常见、更通俗的称呼是“**运行时**”[①]，“运行时”负责加载和运行程序，并在程序执行时提供额外的服务（比如安全保障、垃圾回收等）。

CIL和“运行时”规范包含在一项国际性的标准中，即所谓的**公共语言基础结构**（Common Language Infrastructure，CLI）。这是理解C#程序的执行环境以及C#如何与其他程序和库（甚至是用其他语言编写的）进行无缝交互的一个重要规范。注意，CLI没有规定标准具体如何实现，但它描述了一个CLI平台在符合标准的前提下应该具有什么行为。这为CLI实现者提供了足够大的灵活性，放手让他们在必要的情况下大胆创新，但同时又能提供足够多的结构，使一个平台创建的程序能在另一个不同的CLI实现上运行，甚至能在另一个不同的操作系统上运行。

> **注意**
>
> CIL和CLI这两个缩写词一定要仔细区分。充分理解它们的含义，避免混淆。

CLI标准包含以下更详细的规范：

- 虚拟执行系统（VES，即常说的“运行时”）；
- 公共中间语言（CIL）；
- 公共类型系统（Common Type System，CTS）；
- 公共语言规范（Common Language Specification，CLS）；
- 元数据（Metadata）；
- 框架（Framework）。

本章将进一步拓宽你对C#的认识，让你能从CLI的角度看问题。CLI是决定C#程序如何运行以及如何与其他程序和操作系统进行交互的关键。

21.2 CLI实现

目前有7种主流的CLI实现（4种来自微软），各自都实现了一个C#编译器。表21-1对这些实现进行了总结。

表21-1 主要的C#编译器

编译器名称	说 明
Microsoft Visual C# .NET Compiler	微软的.NET C#编译器是业内最主要的编译器，但只能在Windows操作系统上运行。它作为Microsoft .NET Framework SDK的一部分免费提供给用户下载和使用。网址为：http://msdn.microsoft.com/en-us/netframework/default.aspx
Microsoft Silverlight	这是CLI的一个跨平台的实现，能同时在Windows和Macintosh操作系统上运行。要想更多地了解在这个平台上进行开发的知识，请访问http://silverlight.net/getstarted

① “运行时”或者说runtime在这里并不是指“在运行的时候”。如果说到时间，我不会加引号，或者会直接说执行时。加了引号的“运行时”特指“虚拟执行系统”这个代理，它负责管理C#程序的执行。——译者注

续表

编译器名称	说　明
Microsoft Compact Framework	这是.NET Framework的一个精简的实现，设计成在PDA和手机上运行
Microsoft XNA	这是游戏开发人员使用的一个CLI实现，面向的是Xbox和Windows Vista。欲知详情，请访问www.xna.com
Mono Project	Mono Project是一个开源的实现，由Ximian赞助，提供了CLI规范及C#编译器的Windows、Linux和UNIX兼容版本。源代码及二进制文件可从以下网址获得：www.go-mono.com
DotGNU	侧重于创建平台可移植应用程序，使其能够同时在CLI的.NET 和DotGNU.Portable.NET实现下运行。该实现文件可从以下网址获得：www.dotgnu.org。支持的操作系统包括GNU/Linux *BSD、Cygwin/Mingw32、Mac OS X、Solaris、AIX和PARISC。DotGNU和Mono在不同的时期都使用了对方的一部分库
Rotor	Rotor程序，也称为Shared Source CLI，是微软开发出来在Windows、Mac OS X和FreeBSD上运行的一个CLI实现。实现文件和源代码可从以下网址免费下载：http://msdn.microsoft. com/en-us/library/ms973880.aspx。注意，尽管源代码是可以下载的，但微软并没有授权用Rotor开发商业应用程序，它只是一个学习工具

虽然这些平台和编译器在处理第1章的源代码时没有丝毫问题，但请注意，每个CLI和C#编译器实现和规范相符的程度是不一的。例如，有的实现不能编译全部的新语法。然而，所有实现都符合C# 1.0的ECMA-334[①]规范和CLI 1.2的ECMA-335规范[②]。此外，许多实现还包含一些原型功能，你可以想象它们是在最终成为标准之前的雏形。

21.3 C#编译成机器码

第1章的HelloWorld代码清单显然是C#代码，因此，执行此程序需要用C#编译器编译它。然而，处理器仍然不能直接解释编译好的C#代码。还需要另外一个编译步骤，将C#编译结果转换为机器码。此外，执行时还涉及一个代理，此代理负责为C#程序添加额外的服务，这些服务是无需显式编码的。

所有计算机语言都为程序设计定义了语法和语义。由于C和C++之类的语言会直接编译成机器码，所以这些语言的平台是底层操作系统与机器指令集，即Microsoft Windows、Linux以及Unix等。但C#不同，它的底层平台是“运行时”（或VES）。

CIL是C#编译器的编译结果。之所以称为 “公共中间语言”（CIL），因为还需要一个额外的步骤将CIL转换为处理器能理解的东西（图21-1显示了这个过程）。

换言之，C#编译需要两个步骤。

（1）C#编译器将C#转换为CIL。

（2）将CIL转换为处理器能够执行的指令。

---

① 可以通过电子邮件免费获得，或者从以下网址免费下载：www.ecma-international.org/publications/standards/Ecma-334.htm。

② 可以通过电子邮件免费获得，或者从以下网址免费下载：www.ecma-international.org/publications/standards/Ecma-335.htm.。

C#代码

```
class HelloWorld
{
    static void  Main()
    {
       System.Console.WriteLine(
          "Hello. My name is Inigo Montoya");
    }
}
```

C#编译器

CIL代码

```
.method private hidebysig static void Main() cil
managed
{
    .entrypoint
    //Code size       11 (0xb)
    .maxstack 8
    IL_0000: ldstr      "Hello. My name is Inigo Montoya"
    IL_0005: call       void
[mscorlib]System.Console::WriteLine(string)
    IL_000a: ret
} // end of method HelloWorld::Main
```

运行时

机器码

```
00000000  push        ebp
00000001  mov         ebp,esp
00000003  sub         esp,28h
00000006  mov         dword ptr [ebp-4],0
0000000d  mov         dword ptr [ebp-0Ch],0
00000014  cmp         dword ptr ds:[001833E0h],0
0000001b  je          00000022
0000001d  call        75F9C9E0
00000022  mov         ecx,dword ptr ds:[01C31418h]
00000028  call        dword ptr ds: [03C8E854h]
0000002e  nop
0000002f  mov         esp,ebp
00000031  pop         ebp
00000032  ret
```

图21-1 C#编译成机器码

“运行时”能理解CIL语句，并能将它们编译为机器码。通常，要由“运行时”内部的一个组件来执行从CIL到机器码的编译。这个组件称为**即时编译器**（just-in-timecompiler，即JIT）。程序安装或执行时，便可能发生JIT编译，或者说**即时编译**（jitting）。大多数CLI实现都倾向于执行时编译CIL，但CLI本身并没有规定应该在什么时候编译。事实上，CLI甚至允许CIL解释执行，而不是编译执行，就像许多脚本程序那样。此外，.NET包含一个NGEN工具，此工具能在实际运行程序之前将代码编译成机器码。这个执行前的编译动作必须要在实际运行程序的计算机上进行，因为它会评估机器特性（处理器、内存等），以便生成更高效的代码。在程序安装时（或者在它执行前的任何时候）使用NGEN，好处在于可以避免在程序启动时才执行JIT编译，以缩短程序的启动时间。

21.4 运行时

即使“运行时”将CIL代码转换为机器码，并开始执行它之后，它也在继续管理着代码的执行。在“运行时”这样的一个代理环境下执行的代码称为**托管代码**，在“运行时”控制下的执行过程则称为**托管执行**。对执行的控制转向数据，数据就成为了**托管数据**，因为数据所需的内存是由“运行时”自动分配和回收的。

略有微词的是，公共语言运行时（CLR）这个术语从技术上讲并不是CLI的一个专业术语。CLR更像是微软专门针对.NET平台实现的“运行时”。不管怎样，CLR正在逐渐成为运行时的一个常用代名词，而技术上更准确的术语虚拟执行系统很少在CLI规范之外的地方用到。

由于代理控制着程序的执行，所以能将额外的服务注入程序，即使程序员没有显式地指定需要这些服务。因此，托管代码提供了一些信息来允许附加这些服务。例如，托管代码允许定位与类型成员有关的元数据，支持异常处理，允许访问安全信息，并允许遍历栈。在本节剩余的部分中，还要描述通过“运行时”和托管执行来提供的一些附加服务。CLI没有明确要求提供所有这些服务，但目前的CLI平台都实现了它们。

21.4.1 垃圾回收

垃圾回收是根据程序的需要自动分配和回收内存的过程。对于没有自动系统来做这件事的语言来说，这是一个严重的编程问题。没有垃圾回收器，程序员就必须记住亲自回收他们分配的任何内存。如果忘记这样做，或者对同一个内存分配反复这样做，就会在程序中造成内存泄漏或损坏，尤其是对于Web服务器这样长时间运行的程序。由于“运行时”内建了垃圾回收支持，所以程序员可以将精力集中在程序特性上，而不是为内存管理疲于奔命。

> **语言对比：C++——确定性析构**
>
> 垃圾回收器具体如何工作并不是CLI规范的一部分。因此，每个实现所采取的方法略有不同（事实上，垃圾回收器不是CLI明确要求的）。C++程序员要逐渐习惯的一个重要概念是：被垃圾回收的对象不一定会进行确定性回收。所谓**确定性**（deterministically）回收，是指在良好定义的、编译时知道的位置进行回收。事实上，对象可以在它们最后一次被访问和程序关闭之间的任何时间进行垃圾回收。这包括在超出作用域之前回收，或者等到一个对象实例能由代码访问之后回收。

应该注意，垃圾回收器只负责内存管理。它没有提供一个自动的系统来管理与内存无关的资源。因此，如果需要采取显式的行动来释放资源（除内存之外的其他资源），使用该资源的程序员应该使用特殊的CLI兼容编程模式来帮助清理这些资源（详情参见第9章）。

21.4.2 .NET的垃圾回收

.NET平台实现的垃圾回收机制使用一个分代的（generational）、支持压缩的（compacting）以及基于mark-and-sweep（标记并清除）的算法。之所以说它是分代的，是因为只存活过短暂时间的对象与已经在垃圾回收时存活下来（原因是对象仍在使用）的对象相比，前者会被更早地清理掉。这一点符合内存分配的常规模式：已经存活过较长时间的对象，会比最近才实例化的对象存活得更久一些。

此外，.NET垃圾回收器使用了一个mark-and-sweep算法。在每次执行垃圾回收期间，它都会标记出将要回收的对象，并将剩余的对象压缩到一起，确保它们之间没有“脏”空间。使用压缩机制来填充由回收的对象腾出来的空间，通常会使新对象能以更快的速度实例化（与非托管代码相比），这是因为不必搜索内存为一次新的分配寻找空间。与此同时，这个算法还降低了执行分页处理的概率，因为同一个页中能存储更多的对象，这也有利于性能的提升。

垃圾回收器会考虑到机器上的资源以及执行时对那些资源的需求。例如，假定计算机的内存尚剩余大量空间，垃圾回收器就很少运行，并很少花时间去清理那些资源。相比之下，不基于垃圾回收的平台和语言很少进行这样的优化。

21.4.3 类型安全

“运行时”提供的关键优势之一就是检查类型之间的转换。我们把它称为“运行时”的**类型检查**能力。通过类型检查，“运行时”防止了程序员不慎引入可能造成缓冲区溢出安全漏洞的非法类型转换。此类安全漏洞是最常见的计算机入侵方式之一。因此，让“运行时”自动杜绝此类漏洞，对于安全性来说是很有利的。运行时提供的类型检查可以确保以下几点。

- 变量和变量引用的数据都是有类型的，而且变量的类型兼容于它所引用的数据。
- 可以局部分析一个类型（而不必分析使用了该类型的所有代码），确定需要什么权限来执行该类型的成员。
- 每个类型都有一组编译时定义的方法和数据。“运行时”强制性地规定什么类能访问那些方法和数据。例如，标记为“private”的方法只能由包容它的类型访问。

高级主题：绕过封装和访问修饰符

只要有相应的权限，就可以通过**反射**机制绕过封装和访问修饰符。反射机制提供了晚期绑定功能，允许浏览类型的成员，在对象的元数据中查找特定构造的名称，并可调用类型的成员。

21.4.4 代码访问安全性

“运行时”可以在程序执行时执行安全检查，根据权限来允许或禁止特定类型的操作。执行特定功能的权限并不限于正在运行程序的用户。“运行时”还根据程序创建者以及他们是不是受信任的提供者来控制执行。类似地，“代码访问安全性”（Code Access Security，CAS）还根据代码的位置来应用安全策略。默认情况下，本地计算机上安装的代码比来自LAN的代码更受信任，后者比来自Internet的代码更受信任。权限是可以精细调节的，所以部分受信任的提供者可以在磁盘上受到控制的位置读写文件，但他们不可以访问其他未授权位置（比如来自一个电子邮件程序的邮件地址）。一个提供者的身份标识由提供者编译代码时嵌入程序的证书来决定。

21.4.5 平台可移植性

“运行时”理论上允许C#代码和最终的程序具有**平台可移植性**。换言之，程序能在多种操作系统上运行，而且不管具体的CLI实现是什么。这里所说的可移植性并不仅仅是指代码需要重新编译才能在另一个平台上运行。针对一个平台编译好的CLI模块应该能在任何一个CLI兼容平台上运行，而不需要重新编译。为了获得这种程度的可移植性，代码的移植工

作必须由“运行时”的实现来完成，而不是由应用程序的开发者来完成。

当然，为了实现这种理想化的可移植性，前提是不能使用某个平台特有的API。也正是因为这个限制，许多开发人员放弃了CLI的与平台无关代码，换用平台特有的API来访问底层的平台功能，而不愿意花时间从头写这些API。

取决于平台开发人员的目标，.NET、WinRT、DotGNU、Rotor和Mono提供的平台可移植性各不相同。

- .NET最初定位于在微软系列的操作系统上运行。
- 同样由微软出品的Rotor主要用于教学和推动未来的CLI开发。它对FreeBSD的支持证明了CLI的可移植潜力。.NET包含的一些库（如WinForms、ASP.NET、ADO.NET等）作为专利技术未被纳入Rotor。
- DotGNU和Mono最初是为Linux设计的，但已被移植到多种操作系统（包括iOS和Android）。此外，这些CLI的目标是提供一种方法将.NET应用程序移植到不受微软控制的其他操作系统。因此，.NET和Mono/DotGNU有很大一部分API是重叠的。
- WinRT由微软进行了全面重制，通过牺牲兼容性来换取API的增强。不仅.NET从中受益，对底层操作系统的访问也被增强了。例如，以前要花50毫秒以上的.NET/BCL API被取消了，换成了异步版本。Win32 访问也被放弃，代之以大幅增强的WinRT API。

遗憾的是，仅仅是基类库（BCL）自己的不一致（即便只是在Mirosoft开发的CLI平台中），就使移植变得困难重重（有时甚至根本不可能移植）。就连WinRT、桌面.NET Framework和Silverlight之间的API都是不同的。要实现可移植性，最好的选择就是用Visual Studio 2012创建“可移植类库”项目（Visual Studio 2010要用到Portable Library Tools，请访问http://msdn.microsoft.com/en-us/library/gg597391.aspx）。但是，这个方案也不完美，因为它会妨碍使用一些最新的C#语言功能，包括`await`和`dynamic`关键字。不过，如果向后兼容的优先级高于新语言功能的使用，这还是可以接受的。

21.4.6 性能

许多习惯于写非托管代码的程序员会一语道破天机：托管环境为应用程序带来了额外的开销，无论它有多简单。这要求开发者做出取舍：是不是可以牺牲运行时的一些性能，换取更高的开发效率以及托管代码中更少的bug数量？事实上，从汇编程序转向C这样的高级语言，以及从结构化编程转向面向对象开发时，我们都在进行同样的取舍。大多时候都选择了开发效率的提升，尤其是在硬件速度越来越快但价格越来越便宜的今天。将较多的时间花在架构设计上，相较于穷于应付低级开发平台的各种复杂性，前者更有可能带来大幅的性能提升。另外，考虑到缓冲区溢出可能引发安全漏洞，托管执行变得更有吸引力了。

毫无疑问，特定的开发情形（比如设备驱动程序）是不适合托管执行的。然而，随着托管执行的功能越来越强，对其性能的担心会变得越来越少。最终，只有在要求精确控制的场合，或者要求必须拿掉“运行时”的场合，才需要用到非托管执行[①]。

此外，“运行时”的一些特别设计可以使程序的性能优于本机编译。例如，由于到机

① 事实上，微软已经明确指出，托管开发将成为未来开发Windows应用程序的主流方式。即使那些和操作系统集成的应用程序，也主要采用这种开发方式。

器码的转换是在目标机器上发生的，所以生成的编译代码能够与那台机器的处理器和内存布局完美匹配。相反，非JIT编译的语言是无法获得这一性能优势的。另外，“运行时”能灵活响应执行时的一些突发状况。相反，已直接编译成机器码的程序是无法照顾到这些情况的。例如，在目标机器上的内存非常富余的时候，非托管语言仍然会刻板地执行既定计划，在编译时定义的位置回收内存（确定性析构）。相反，支持JIT编译的语言只有在运行速度变慢或者程序关闭的时候才会回收内存。虽然JIT编译为执行过程添加了一个额外的编译步骤，但JIT编译器能带来代码执行效率的大幅提升，使程序的最终性能仍然优于直接编译成机器码的程序。总之，CLI程序不一定比非CLI程序快，但性能是有优势的。

21.5 应用程序域

通过在程序和操作系统之间引入一个层，可以实现称为**应用程序域**（app domain或AppDomain）的虚拟进程或者应用程序。应用程序域的行为类似于一个操作系统进程。换言之，不同的应用程序域相互之间是隔离的。例如，应用程序域有自己的虚拟内存分配，而且应用程序域之间的通信需要采用分布式通信模式，就像两个操作系统进程之间进行通信那样。类似地，静态数据不能在应用程序域之间共享，所以静态构造器是为每个应用程序域运行的，而且假定每个应用程序域一个线程，就没有必要同步静态数据，因为每个应用程序有其自己的数据实例。此外，每个应用程序域有其自己的线程，而且和操作系统进程一样，线程不能跨越应用程序域的边界。

之所以要使用应用程序域，是由于进程被视为相当昂贵的资源。有了应用程序域，就能在一个进程内运行多个应用程序域，从而避免额外的花费。例如，可以使用一个进程来容纳一系列网站。然而，可以把这些网站放在它们自己的应用程序域内，使唤这些网站相互隔离。总之，应用程序域代表的是一个虚拟进程，它在操作系统进程和线程之间的一个层上运行。

21.6 程序集、清单和模块

CLI规定了一个源语言编译器的CIL输出规范。编译器输出的通常是一个程序集。除了CIL指令本身，程序集还包含一个清单（manifest），它由下面这些内容构成：

- 程序集定义和导入的类型；
- 程序集本身的版本信息；
- 程序集依赖的其他文件；
- 程序集的安全权限。

清单本质上是程序集的一个标头（header），提供了与程序集的构成有关的所有信息，另外还有对这个程序集进行唯一性标识的信息。

程序集可以是类库，也可以是可执行文件本身。而且，一个程序集能引用其他程序集（后者又可以引用更多的程序集）。由此而构建的一个应用程序是由许多组件构成的，而不是一个巨大的、单一的程序。这是现代编程平台要考虑的一个重要特性，因为它能显著提高程序的可维护性，并允许一个组件在多个应用程序中共享。

除了清单，程序集还将CIL代码包含到一个或者多个模块中。通常，程序集和清单被合并成一个单独的文件，就像第1章的HelloWorld.exe一样。然而，也可以将模块单独放到它们自己的文件中，然后使用程序集链接器（al.exe）来创建一个程序集文件，其中包括

对每个模块进行引用的清单[①]。这不仅提供了另一种手段将程序分解成单独的组件，还实现了使用多种不同源语言来开发一个程序集。

偶尔，术语模块和程序集可以互换着使用。然而，在谈及CLI兼容程序或库的时候，首选的术语是程序集。图21-2解释了不同的组件术语。

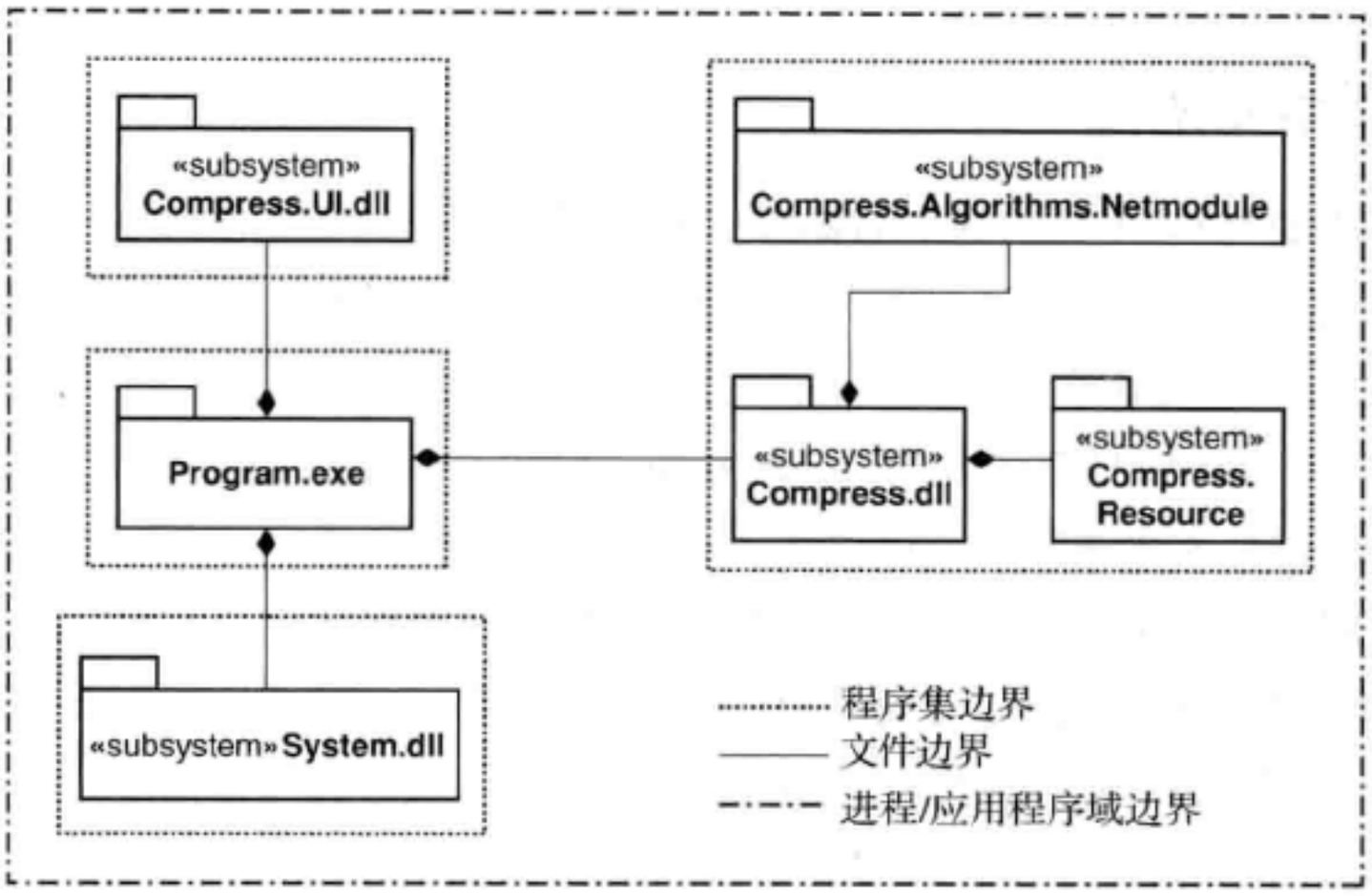


图21-2 带有模块的程序集以及它们引用的文件

注意，程序集和模块都能引用文件，比如本地化为特定语言的资源文件。尽管很少见，但两个不同的程序集也可以引用同一个模块或文件。

虽然程序集可以包含多个模块和文件，但整个文件组只有一个版本号，而且这个版本号被放在程序集的清单中。因此，在一个应用程序内，最小的可版本化的组件就是程序集，即使那个程序集由多个文件构成。如果在不更新程序集清单的情况下更改了引用的任何一个文件，甚至是发布了一个补丁，都会违背清单以及整个程序集本身的完整性。考虑到这方面的原因，程序集成为了一个逻辑性的组件结构，或者一个部署单元。

注意

程序集是可以版本化和安装的最小单元。构成程序集的单独模块则不是最小单元。

虽然程序集（逻辑构造）可以由多个模块构成，但大多数程序集都只包含一个模块。此外，微软现在提供了`ILMerge.exe`实用程序，能将多个模块及其清单合并成单文件的程序集。

由于清单包含对程序集所有依赖文件的引用，所以可以根据清单判断程序集的依赖性。此外，在执行时，“运行时”只需检查清单，就可以确定它需要什么文件。只有发行供多个应用程序共享的库的工具供应商（比如微软）才需要在部署时注册那些文件。这使部署变得非常容易。通常，人们将基于CLI的应用程序的部署过程称为xcopy部署。这个名字来源于Windows著名的xcopy命令，它的作用是直接将文件复制到指定的目的地。

---

① 部分原因在于主流CLI IDE——Visual Studio .NET——缺乏相应的功能来处理由多个模块构成的程序集。当前的Visual Studio .NET没有提供集成的工具来构建由多个模块构成的程序集。在使用此类程序集的时候，“智能感知”不能完全发挥作用。