16.2.8 链表：LinkedList<T>

除此之外，System.Collections.Generic还支持一个链表集合，它允许正向和反向遍历。图16-7展示了类图。注意，没有对应的非泛型类型。

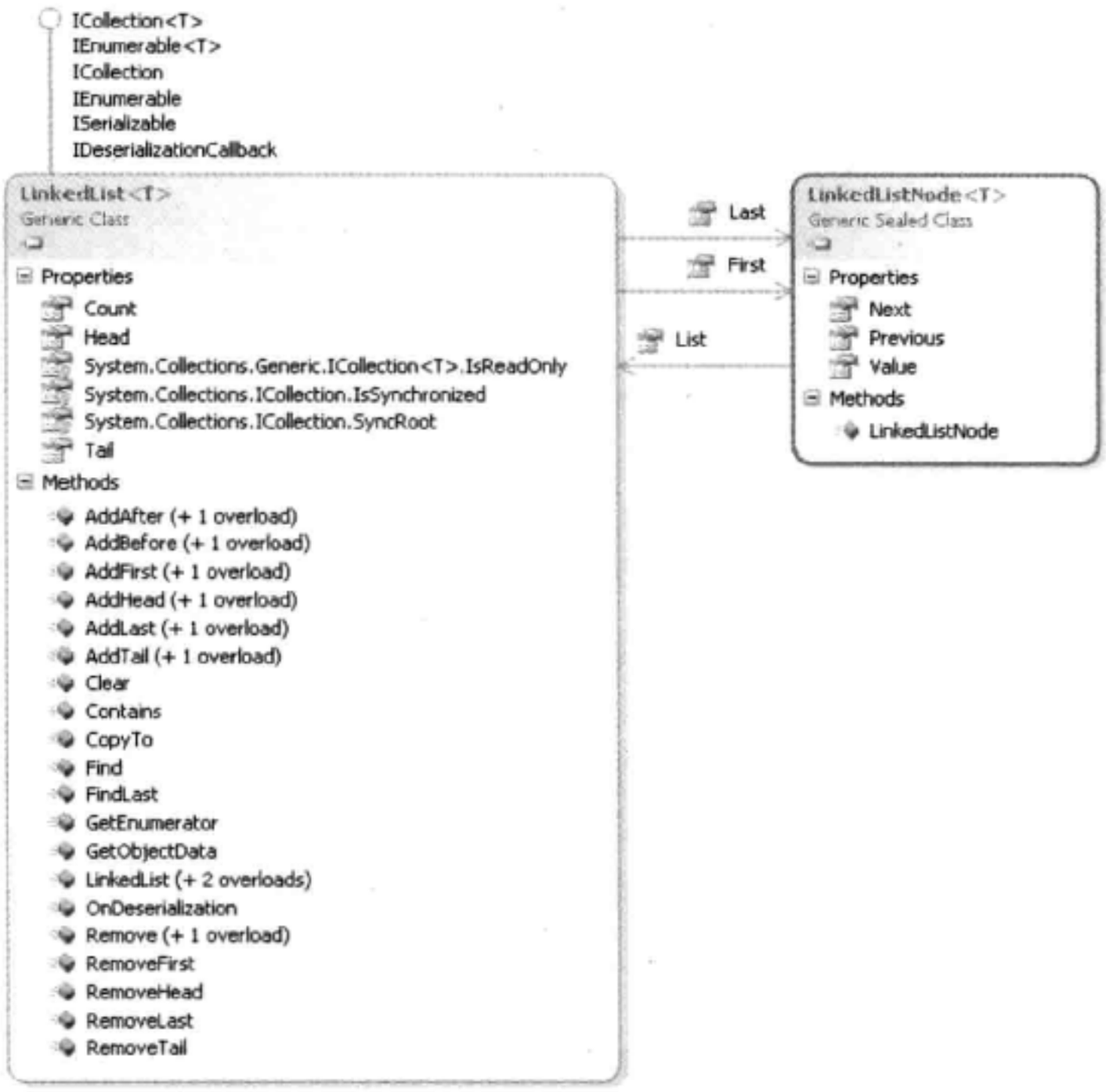


图16-7 LinkedList<T>和LinkedListNode<T>类图

16.3 提供索引器

数组、字典和列表都提供了**索引器**（indexer）以便根据键或索引来获取或者设置成员。如前所述，为了使用索引器，只需将索引放到集合后的方括号中。可定义自己的索引器，代码清单16-9用Pair<T>来展示了一个例子。

代码清单16-9 定义索引器

```
interface IPair<T>
{
  T First
  {
      get;
  }

  T Second
  {
      get;
  }

  T this[PairItem index]
  {
```

```
        get;
    }
}
```

```
public enum PairItem
{
    First,
    Second
}
```

```
public struct Pair<T> : IPair<T>
{
public Pair(T first, T second)
{
    _first = first;
    _second = second;
}
public T First
{
    get{ return _first; }
    private set{ _first = value; }
}
private T _first;
public T Second
{
    get{ return _second; }
    private set{ _second = value; }
}
private T _second;
```

```
public T this[PairItem index]
{
    get
    {
        switch (index)
        {
            case PairItem.First:
                return First;
            case PairItem.Second:
                return Second;
            default :
                throw new NotImplementedException(
                    string.Format(
                    "The enum {0} has not been implemented",
                    index.ToString()));
        }
    }
    set
    {
        switch (index)
        {
            case PairItem.First:
                First = value;
                break;
            case PairItem.Second:
                Second = value;
                break;
            default:
                throw new NotImplementedException(
                    string.Format(
```

```
                    "The enum {0} has not been implemented",
                    index.ToString()));
            }
        }
    }
}
```

索引器的声明和属性的声明很相似，但不是使用属性名，而是使用关键字this，后面是一个包含在方括号中的参数列表。主体也像一个带有get和set块的属性。如代码清单16-9所示，参数不一定是int。事实上，索引可获取多个参数，甚至可以重载。本例使用enum来防止调用者为不存在的项提供索引。

C#编译器为索引器创建的CIL代码是名为Item的特殊属性，它获取一个实参。接受实参的属性不可以在C#中显式地创建，所以Item属性在这个方面相当特殊。这是因为使用了Item这个标识符的其他任何成员，即使它有完全不同的签名，都会和编译器创建的成员产生冲突，所以不允许其他成员使用item标识符。

高级主题：使用IndexerName指定索引器属性名称

如前所述，索引器（[]）的CIL属性名称默认为Item。但是，可以使用IndexerNameAttibute来指定一个不同的名称。代码清单16-10将名称更改为"Entry"。

代码清单16-10 更改索引器的默认名称

```
[System.Runtime.CompilerServices.IndexerName("Entry")]
public T this[params PairItem[] branches]
{
    // ...
}
```

这个更改对于索引的C#调用者来说没有任何区别，但是，它为不直接支持索引器的语言指定了名称。

该特性只是一个编译器指令，它指示为索引器使用不同的名称。特性不会实际地进入编译器生成的元数据。所以，不能通过反射来获得这个名称。

高级主题：定义参数可变的索引器

索引器还可获取一个可变的参数列表。例如，代码清单16-11为第11章讨论（下一节也会讨论）的BinaryTree<T>定义了一个索引器。

代码清单16-11 定义接受可变参数的索引器

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

public class BinaryTree<T>:
    IEnumerable<T>
{

  // ...

  public T this[params PairItem[] branches]
  {
      get
```

```
{
    BinaryTree<T> currentNode = this;
    int totalLevels =
        (branches == null) ? 0 : branches.Length;
    int currentLevel = 0;

    while (currentLevel < totalLevels)
    {
        currentNode = currentNode.SubItems[
            branches[currentLevel]];
        if (currentNode == null)
        {
            // The binary tree at this location is null.
            throw new IndexOutOfRangeException();
        }
        currentLevel++;
    }

    return currentNode.Value;
}
set
{
    // ...
}
}

}
```

branches中的每一项都是一个PairItem，指出二叉树中向下导航到哪一个分支。

16.4 返回null或者空集合

返回数组或集合时，必须允许返回null，或者返回不包含任何数据项的集合实例，以便指出包含零个数据项的情况。通常，更好的选择是返回不包含数据项的集合实例。这样一来，可避免强迫调用者在遍历集合数据项之前检查null值。例如，假定现在有一个长度为零的IEnumerable<T>集合，调用者可以立即用一个foreach循环来安全地遍历集合，而不用担心生成的GetEnumerator()调用会引发NullReferenceException异常。考虑使用Enumerable.Empty<T>()方法来简单地生成给定类型的空集合。

但这个准则也有例外的时候，比如null被有意地用来表示有别于“零个项目”的情况。例如，网站用户名集合可能会是null，以此来表示出于某种原因未能获得最新集合，这在语义上有别于空集合。

规范

不要用null引用表示空集合。**考虑**改为使用Enumerable.Empty<T>()方法。

16.5 迭代器

第14章详细探讨了foreach循环的内部工作方式。本节将讨论如何利用**迭代器**为自定义集合实现自己的IEnumerator<T>、IEnumerable<T>和对应的非泛型接口。迭代器用清楚

的语法描述了如何循环遍历（也就是迭代）集合类中的数据，尤其是如何使用foreach来迭代。迭代器使集合的最终用户能遍历集合的内部结构，同时不必了解那个结构。

高级主题：迭代器的起源

1972年，麻省理工学院的Barbara Liskov和一组科学家开始研究新的编程方法，他们将重点放在对用户自定义数据的抽象上。为了检验他们的大多数工作，他们创建了一种名为CLU的语言。这种语言提出了一个名为“群集”（cluster，注意前三个字母是CLU）的概念，这是当今程序员使用的主要数据抽象概念——对象——的前身。作为其研究工作的一部分，团队意识到CLU语言虽然能从最终用户使用的类型中抽象出一些数据表示，但经常都不得不揭示数据的内部结构，以便其他人能正确地使用它。为了摆脱这方面的困扰，他们创建了一种名为迭代器的语言结构。（当年由CLU提出的许多概念最终都在面向对象编程中成为非常基本的东西。）

假如某个类希望支持用foreach进行迭代，就必须实现枚举器（enumerator）模式。如第14章所述，在C#中，foreach循环结构被编译器扩展成while循环结构，它以从IEnumerable<T>接口获取的IEnumerator<T>接口为基础。

枚举数模式的问题在于，以手工方式实现它比较麻烦，因为它需要维持对集合中的当前位置进行描述所需的全部状态。对于列表集合类型，这个内部状态可能比较简单，当前位置的索引就足够了。但对于需要以递归方式遍历的数据结构（比如二叉树），状态就可能相当复杂。为了应对实现这个模式时面临的挑战，C# 2.0引入了迭代器的概念，它使一个类可以更容易地描述foreach循环如何遍历它的内容。

16.5.1 迭代器的定义

迭代器是实现类的方法的一个途径，它们是更加复杂的枚举数模式的语法简化形式。C#编译器遇到迭代器时，会把它的内容扩展成实现了枚举数模式的CIL代码。因此，迭代器的实现对“运行时”没有特别的依赖。由于C#编译器在生成的CIL代码中仍然采用枚举数模式，所以在使用迭代器之后，并不会带来真正的运行时性能优势。不过，选择使用迭代器，而不是手动实现枚举数模式，能显著提高程序员的编程效率。先看看迭代器在代码中如何定义。

16.5.2 迭代器语法

迭代器提供了迭代器接口（也就是IEnumerable<T>和IEnumerator<T>接口的组合）的一个快捷实现。代码清单16-12通过创建一个GetEnumerator()方法，为泛型BinaryTree<T>类型声明了一个迭代器。接着，我们要添加对迭代器接口的支持。

代码清单16-12 迭代器接口模式

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

public class BinaryTree<T>:
    IEnumerable<T>
{
    public BinaryTree ( T value)
    {
```

```
        Value = value;
    }

    #region IEnumerable<T>
    public IEnumerator<T> GetEnumerator()
    {
        //...
    }
    #endregion IEnumerable<T>

    public T Value
    { get; private set; }
    public Pair<BinaryTree<T>> SubItems
    { get; private set; }
}

public struct Pair<T>
{
    public Pair(T first, T second) : this()
    {
        First = first;
        Second = second;
    }
    public T First { get; private set; }
    public T Second { get; private set; }
}
```

如代码清单16-12所示，我们要为GetEnumerator()方法提供一个实现。

16.5.3 从迭代器生成值

迭代器类似于函数，但它不是返回（return）一个值，而是生成（yield）一系列值。在BinaryTree<T>的情况下，迭代器生成的值的类型是提供给T的类型实参。假如使用的是IEnumerator的非泛型版本，生成的值就是object类型。

为了正确实现迭代器模式，需要维护一些内部状态，以便在枚举集合时跟踪记录当前位置。在BinaryTree<T>的情况下，要记录的是树中哪些元素已被枚举，哪些还没有被枚举。迭代器由编译器转换成“状态机”来跟踪记录当前位置，它还知道如何将自己移动到下一个位置。

迭代器每次遇到yield return语句都生成一个值，控制立即返回到请求数据项的调用者。然后，当调用者请求下一项时，会紧接在上一个yield return语句之后执行。在代码清单16-13中，顺序返回C#内建的数据类型关键字。

代码清单16-13 顺序地生成C#关键字

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

public class CSharpBuiltInTypes: IEnumerable<string>
{
  public IEnumerator<string> GetEnumerator()
  {
      yield return "object";
      yield return "byte";
      yield return "uint";
      yield return "ulong";
```

```
            yield return "float";
            yield return "char";
            yield return "bool";
            yield return "ushort";
            yield return "decimal";
            yield return "int";
            yield return "sbyte";
            yield return "short";
            yield return "long";
            yield return "void";
            yield return "double";
            yield return "string";
        }

            // The IEnumerable.GetEnumerator method is also required
            // because IEnumerable<T> derives from IEnumerable.
        System.Collections.IEnumerator
            System.Collections.IEnumerable.GetEnumerator()
        {
            // Invoke IEnumerator<string> GetEnumerator() above
            return GetEnumerator();
        }
    }

    public class Program
    {
        static void Main()
        {
            CSharpBuiltInTypes keywords =
                new CSharpBuiltInTypes();

            foreach (string keyword in keywords)
            {
                Console.WriteLine(keyword);
            }
        }
    }
```

代码清单16-13的结果如输出16-5所示。

输出16-5

```
object
byte
uint
ulong
float
char
bool
ushort
decimal
int
sbyte
short
long
void
double
string
```

上述代码清单的结果是所有C#内建类型的一个列表。

16.5.4 迭代器和状态

GetEnumerator()在foreach语句（比如代码清单16-13的foreach(string keyword in keywords)）中被首次调用时，会创建一个迭代器对象，其状态被初始化为特殊的“起始”状态，表示迭代器尚未执行代码，所以尚未生成任何值。只要foreach语句继续，迭代器就会一直维持其状态。循环每一次请求下一个值，控制就会进入迭代器，从上一次离开的位置继续。该位置是根据迭代器对象中存储的状态信息来判断的。foreach语句在调用点终止时，迭代器的状态就不再保存了。

总是可以安全地再次调用Get Enumerator()；如果有必要，会创建新的枚举器对象。

图16-8展示了所有事件的发生顺序。记住，MoveNext()方法是由IEnumerator<T>接口提供的。

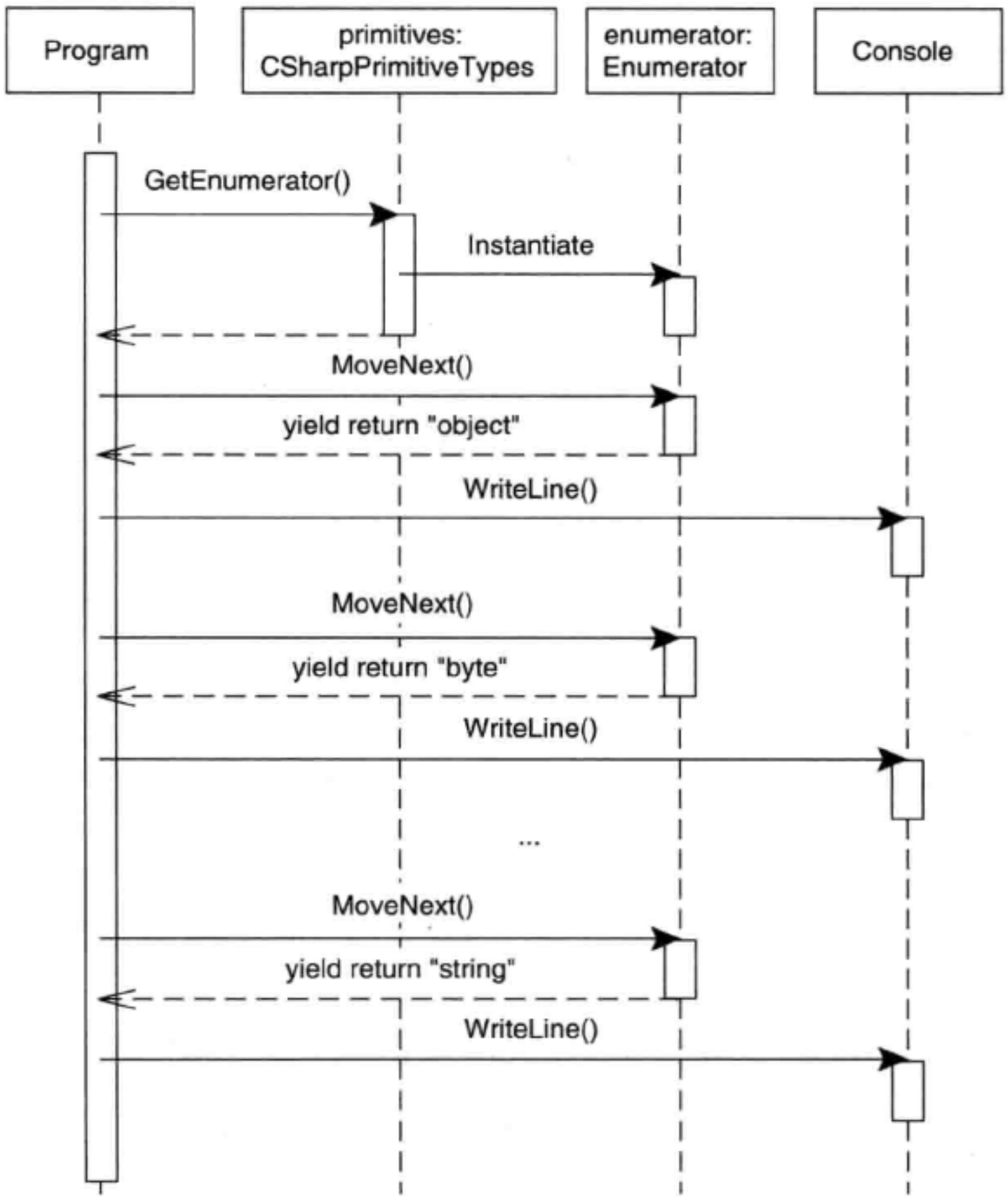


图16-8 yield return顺序图

在代码清单16-13中，foreach语句在名为keywords的CSharpBuiltInTypes实例上初始化对GetEnumerator()的一个调用。

获得迭代器实例（由iterator引用）之后，foreach的每次循环迭代都以一个MoveNext()调用开始。在迭代器内部生成一个值，并把它返回给位于调用点的foreach语句。执行了yield return语句之后，GetEnumerator()方法暂停并等待下一个MoveNext()请

求。现在回到循环主体，foreach语句在屏幕上显示生成的值。然后，它开始下一次循环迭代，再次在迭代器上调用MoveNext()。注意，第二次循环迭代的时候，控制是从第二个yield return语句开始执行的。同样地，foreach在屏幕上显示由CSharpBuiltInTypes生成的值，并开始下一次循环迭代。这个过程会一直继续下去，直至迭代器中没有更多的yield return语句。届时foreach循环将终止，因为MoveNext()返回false。

16.5.5 更多的迭代器例子

在修改BinaryTree<T>之前，必须修改Pair<T>，让它支持使用迭代器的IEnumerable<T>接口。代码清单16-14展示了如何生成Pair<T>中的每个元素（其实就两个元素，First和Second）。

代码清单16-14 使用yield来实现BinaryTree<T>

```
public struct Pair<T>: IPair<T>,
    IEnumerable<T>
{
  public Pair(T first, T second) : this()
  {
      First = first;
      Second = second;
  }
  public T First { get; private set; }
  public T Second { get; private set; }

  #region IEnumerable<T>
  public IEnumerator<T> GetEnumerator()
  {
      yield return First;
      yield return Second;
  }
#endregion IEnumerable<T>

  #region IEnumerable Members
  System.Collections.IEnumerator
      System.Collections.IEnumerable.GetEnumerator()
  {
      return GetEnumerator();
  }
  #endregion
}
```

在代码清单16-14中，Pair<T>数据类型被循环迭代了两次。第一次是通过yield return First，第二次是通过yield return Second。在GetEnumerator()中，每次遇到yield return语句，状态都会保存，而且执行似乎“跳出”了GetEnumerator()方法的上下文，并进入循环主体。第二次循环迭代开始的时候，GetEnumerator()从yield return Second语句恢复执行。

System.Collections.Generic.IEnumerable<T>是从System.Collections.IEnumerable继承的。因此，实现IEnumerable<T>还需实现IEnumerable。在代码清单16-14中，这个实现是显式进行的。不过这个实现很简单，就是调用了一下IEnumerable<T>的GetEnumerator()实现。由于IEnumerable<T>和IEnumerable之间的类型兼容性（通过继

承），所以从IEnumerable.GetEnumerator()中调用IEnumerable<T>.GetEnumerator()始终是合法的。由于IEnumerable和IEnumerator<T>之间的类型兼容性，从x向y的调用始终是合法的。由于两个GetEnumerator()的签名完全一致（返回类型不是签名的区分因素），所以要么一个实现是显式的，要么两个实现都是显式的。考虑到IEnumerable<T>的版本提供了额外的类型安全性，所以IEnumerable的实现选择显式地进行。

代码清单16-15使用了Pair<T>.GetEnumerator()方法，在连续的两行上显示"Inigo"和"Montoya"。

代码清单16-15 通过foreach来使用Pair<T>.GetEnumerator()

```
Pair<string> fullname = new Pair<string>("Inigo", "Montoya");
foreach (string name in fullname)
{
    Console.WriteLine(name);
}
```

注意，对GetEnumerator()的调用在foreach循环中是隐式进行的。

16.5.6 将yield return语句放到循环中

在前面的CSharpBuiltInTypes和Pair<T>中，是对每个yield return语句都进行硬编码。但实际并不需要如此。可以使用yield return语句从循环结构中返回值。在代码清单16-16中，GetEnumerator()中的foreach每次执行都会返回下一个值。

代码清单16-16 将yield return语句放到一个循环中

```
public class BinaryTree<T>: IEnumerable<T>
{
  // ...

  #region IEnumerable<T>
  public IEnumerator<T> GetEnumerator()
  {
      // Return the item at this node.
      yield return Value;

      // Iterate through each of the elements in the pair.
      foreach (BinaryTree<T> tree in SubItems)
      {
          if (tree != null)
          {
              // Since each element in the pair is a tree,
              // traverse the tree and yield each
              // element.
              foreach (T item in tree)
              {
                  yield return item;
              }
          }
      }
  }
  #endregion IEnumerable<T>

  #region IEnumerable Members
  System.Collections.IEnumerator
      System.Collections.IEnumerable.GetEnumerator()
  {
```

```
            return GetEnumerator();
        }
        #endregion
    }
```

在代码清单16-16中，第一次循环迭代会返回二叉树的根元素。第二次循环迭代时，将遍历由两个子元素构成的一个pair。假如子元素pair包含非空的值，就进入子节点，并生成它的元素。注意，foreach(T item in tree)是对子节点的递归调用。

和CSharpBuiltInTypes和Pair<T>的情况一样，现在就可以使用foreach循环来遍历一个BinaryTree<T>。代码清单16-17对此进行了演示，输出16-6展示了结果。

代码清单16-17 将foreach用于BinaryTree<string>

```
// JFK
jfkFamilyTree = new BinaryTree<string>(
    "John Fitzgerald Kennedy");

jfkFamilyTree.SubItems = new Pair<BinaryTree<string>>(
    new BinaryTree<string>("Joseph Patrick Kennedy"),
    new BinaryTree<string>("Rose Elizabeth Fitzgerald"));

// Grandparents (Father's side)
jfkFamilyTree.SubItems.First.SubItems =
    new Pair<BinaryTree<string>>(
    new BinaryTree<string>("Patrick Joseph Kennedy"),
    new    BinaryTree<string>("Mary Augusta Hickey"));

// Grandparents (Mother's side)
jfkFamilyTree.SubItems.Second.SubItems =
    new Pair<BinaryTree<string>>(
    new BinaryTree<string>("John Francis Fitzgerald"),
    new BinaryTree<string>("Mary Josephine Hannon"));

foreach (string name in jfkFamilyTree)
{
    Console.WriteLine(name);
}
```

输出16-6

```
John Fitzgerald Kennedy
Joseph Patrick Kennedy
Patrick Joseph Kennedy
Mary Augusta Hickey
Rose Elizabeth Fitzgerald
John Francis Fitzgerald
Mary Josephine Hannon
```

高级主题：递归迭代器的危险性

代码清单16-16在遍历二叉树时创建新的“嵌套”迭代器。这意味着当值由一个节点生成时，值由节点的迭代器生成，再由父的迭代器生成，再由父的父的迭代器生成……直到最后由根的迭代器生成。如果一个值有*n*级深度，值就要在包含*n*个迭代器的一个链条中传递。对于很浅的二叉树，这一般不是问题。然而，不平衡的二叉树可能相当深，因而会造成递归迭代的高昂成本。

规范

考虑在迭代较深的数据结构时使用非递归算法。

初学者主题：struct与class

将Pair<T>定义成一个struct而不是class，会造成一个有趣的副作用：SubItems.First和SubItems.Second不能被直接赋值——即使赋值方法是public的。若将赋值方法修改为public，以下代码会造成编译错误，指出SubItems不能被修改，“因为它不是变量”。

```
jfkFamilyTree.SubItems.First =
    new BinaryTree<string>("Joseph Patrick Kennedy");
```

上述代码的问题在于，SubItems是Pair<T>类型的属性，而Pair<T>是一个struct。因此，当属性返回值时，会生成SubItems的一个副本，这个副本在语句执行完毕之后就会丢失。所以，对一个马上就要丢失的副本上的First进行赋值，显然会引起误解。幸好，C#编译器禁止这样做。

为了解决上述问题，可以选择不赋值（代码清单16-17用的就是这种方式），为Pair<T>使用class而不是struct，不创建SubItems属性而改为使用字段，或者在BinaryTree<T>中提供属性以直接访问SubItems的成员。

16.5.7 取消更多的迭代：yield break

有时需要取消更多的迭代。为此，可以包含一个if语句，不执行代码中余下的语句。然而，也可以使用yield break使MoveNext()返回false，使控制立即返回到调用者并终止循环。代码清单16-18展示了一个例子。

代码清单16-18 用yield break取消迭代

```
public System.Collections.Generic.IEnumerable<T>
    GetNotNullEnumerator()
{
  if((First == null) || (Second == null))
  {
      yield break;
  }
  yield return Second;
  yield return First;
}
```

在上述代码中，Pair<T>类中的任何一个元素为null，都会取消迭代。

yield break语句类似于在函数顶部放一个return语句，判断函数无事可做的时候就执行。这是一种可以从更深的迭代中退出的方法，而无需再使用if块来包围余下的所有代码。另外，它使设置多个出口成为可能，所以使用需谨慎，因为阅读代码时如果不小心，就可能错过某个靠前的出口。

高级主题：迭代器是如何工作的

C#编译器遇到一个迭代器时，会根据枚举器模式（enumerator pattern）将代码展开成恰当的CIL。在生成的代码中，C#编译器首先创建一个嵌套的私有类来实现IEnumerator<T>接口，以及它的Current属性和MoveNext()方法。Current属性返回与迭代器的返回类型对应的一个类型。在代码清单16-14中，Pair<T>包含的迭代器能返回一个T类型。C#编译器检查包含在迭代器中的代码，并在MoveNext方法和Current属性中创建必要的代码来模拟它的行为。对于Pair<T>迭代器，C#编译器生成的是大致一一对应的代码（参见代码清单16-19）。

代码清单16-19 与迭代器的CIL代码等价的C#代码

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

public class Pair<T> : IPair<T>, IEnumerable<T>
{
  // ...

  // The iterator is expanded into the following
  // code by the compiler
  public virtual IEnumerator<T> GetEnumerator()
  {
      __ListEnumerator result = new __ListEnumerator(0);
      result._Pair = this;
      return result;
  }
  public virtual System.Collections.IEnumerator
      System.Collections.IEnumerable.GetEnumerator()
  {
      return new GetEnumerator();
  }

  private sealed class __ListEnumerator<T> : IEnumerator<T>
  {
      public __ListEnumerator(int itemCount)
      {
          _ItemCount = itemCount;
      }

      Pair<T> _Pair;
      T _Current;
      int _ItemCount;

      public object Current
      {
          get
          {
              return _Current;
          }
      }

      public bool MoveNext()
      {
          switch (_ItemCount)
          {
              case 0:
                  _Current = _Pair.First;
                  _ItemCount++;
```

```
                return true;
            case 1:
                _Current = _Pair.Second;
                _ItemCount++;
                return true;
            default:
                return false;
        }
    }
}
```

因为编译器拿掉了yield return语句，而且生成的类和手工编写的类基本一致，所以C#迭代器的性能特征与手工实现枚举器模式的性能特征一致。虽然性能没有提升，但开发效率显著提高了。

高级主题：上下文关键字

许多C#关键字都是“保留”的，除非附加@前缀，否则不可以作为标识符使用。yield关键字是上下文关键字，不是保留的关键字。 可以合法地声明名为yield的局部变量（虽然这样做有时会令人混淆）。事实上，C# 1.0之后加入的所有关键字都是上下文关键字，这是为了防止升级老程序来使用新版本的语言时出现问题。

如果C#的设计者为迭代器选择使用yield value;语法而不是yield return value;，就会造成歧义。例如，yield(1+2);是生成值，还是将值作为实参传给一个名为yield的方法呢？

由于以前在return或break之前添加标识符yield是不合法的，所以C#编译器知道在这样的“上下文”中，yield必然是关键字而非标识符。

16.5.8 在单个类中创建多个迭代器

在之前的迭代器例子中，我们实现了IEnumerable<T>.GetEnumerator()。这是foreach要隐式寻找的方法。有时需要不同的迭代顺序，比如逆向迭代、筛选结果或者遍历对象投射等。为了在类中声明额外的迭代器，你可以把它们封装到返回IEnumerable<T>或IEnumerable的属性或方法中。例如，要想逆向遍历Pair<T>中的元素，可以提供如代码清单16-20所示的GetReverseEnumerator()方法。

代码清单16-20 在返回IEnumerable<T>的方法中使用yield return

```
public struct Pair<T>: IEnumerable<T>
{
    ...

    public IEnumerable<T> GetReverseEnumerator()
    {
        yield return Second;
        yield return First;
    }
    ...
}

public void Main()
{
    Pair<string> game = new Pair<string>("Redskins", "Eagles");
```

```
        foreach (string name in game.GetReverseEnumerator())
        {
            Console.WriteLine(name);
        }
    }
```

注意，这里返回的是IEnumerable<T>，而不是IEnumerator<T>。这有别于IEnumerable<T>.GetEnumerator()，后者返回的是IEnumerator<T>。Main()中的代码演示了如何使用foreach循环来调用GetReverseEnumerator()。

16.5.9 yield语句的要求

只有在返回IEnumerator<T>或者IEnumerable<T>类型（或者它们的非泛型版本）的成员中，才能使用yield return语句。主体包含yield return语句的成员不能包含一个简单return语句（也就是普通的、没有添加yield的return语句）。假如成员使用了yield return语句，C#编译器会生成必要的代码来维护迭代器的状态。相反，假如方法使用return语句而不是yield return，就要由程序员负责维护自己的状态机，并返回其中一个迭代器接口的实例。另外，我们知道，在有返回类型的方法中，所有代码路径都必须用一个return语句返回值（前提是代码路径中没有引发异常）。类似地，迭代器中的所有代码路径都必须包含一个yield return语句（如果这些代码路径要返回任何数据的话）。

yield语句的其他限制包括如下方面（违反这些限制将造成编译错误）。

- yield语句只能在方法、用户自定义操作符，或者索引器或属性的get访问器方法中出现。成员不可以获取任何ref或out参数。
- yield语句不能在匿名方法或Lambda表达式（参见第12章）中出现。
- yield语句不能在try语句的catch和finally子句中出现。除此之外，yield语句能够在try块中出现的前提是，这个try块没有catch块。

16.6 小结

自从在C# 2.0中引入泛型集合类和泛型接口之后，应该优先选择泛型而不是非泛型版本。由于避免了因为装箱而造成的开销，而且能在编译时强制遵守类型规则，所以这些泛型版本更快、更安全。除非必须维持与C# 1.0遗留代码的兼容性，否则应该放弃整个System.Collections命名空间（事实上，Silverlight和WinRT CLR已经完全把它放弃了）。换言之，不需要删除已经使用了这个命名空间的所有代码。相反，为新的代码使用System.Collections.Generics，并逐渐迁移现有的代码，以使用对应的泛型集合（这些集合包含了用来处理对象集合的接口和类）。

System.Collections.Generic命名空间并不是C# 2.0对集合进行的唯一改变。另一个引人注目的新特性是迭代器。迭代器涉及一个新的上下文关键字yield。C#利用这个关键字来生成底层的CIL代码，从而实现可由foreach循环使用的迭代器模式。

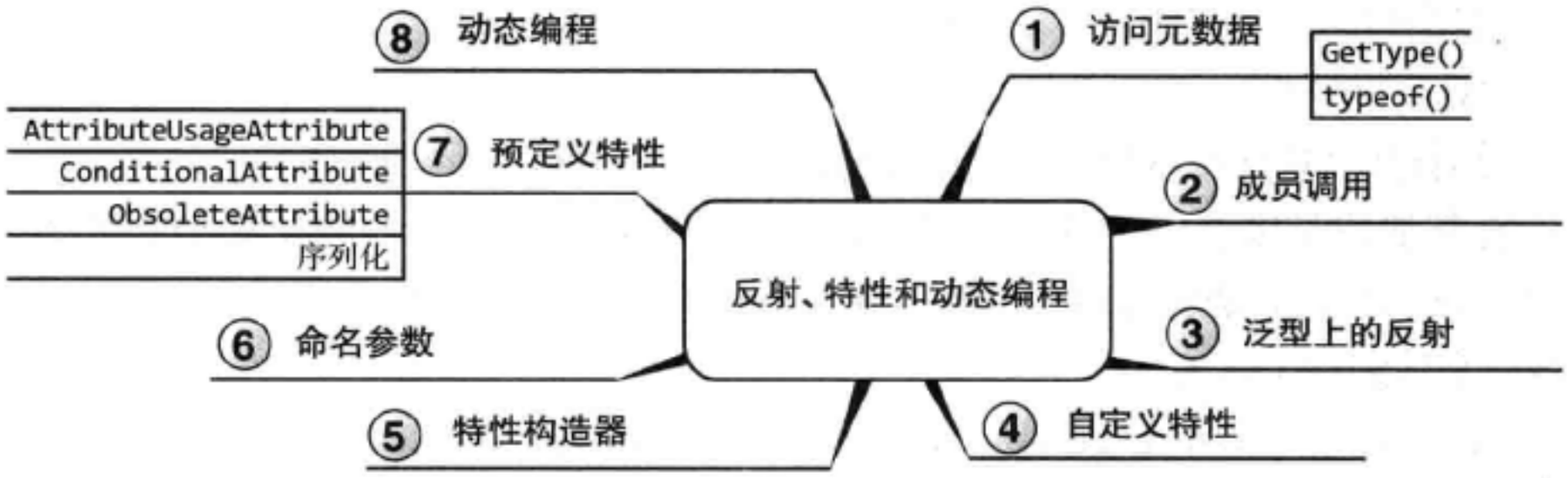
下一章要讨论反射。我们在之前简单地接触过它，但几乎没怎么解释。反射实现了在运行时检查CIL代码中的类型结构。

第17章

反射、特性和动态编程

特性（attribute）的作用是在程序集中插入额外的元数据，并将元数据同编程构造（比如类、方法或者属性）关联起来。本章探讨了特性的细节，并描述了如何创建自定义特性。自定义特性要用起来就必须能被识别。这是通过反射（reflection）来实现的。本章首先讨论了反射，其中包括如何用它在运行时实现动态绑定，以及如何在编译时利用一个成员的名称来调用该成员。在像代码生成器这样的工具中，会频繁地执行这个操作。除此之外，在执行时调用目标未知的情况下，也要用到反射。

本章最后讨论了动态编程，它是C#4.0新增的功能，可利用这个功能简化对动态数据的处理（而且这种动态数据要求执行时而不是编译时绑定）。



17.1 反射

可以利用反射做下面这些事情。

- 访问程序集中类型的元数据。其中包括像完整类型名和成员名这样的构造，以及对一个构造进行修饰的任何特性。
- 使用元数据，在运行时动态调用一个类型的成员，而不是执行编译时绑定。

反射是指对程序集中的元数据进行检查的过程。在以前，当代码编译成一种机器语言时，关于代码的所有元数据（比如类型和方法名）都会被丢弃。相反，当C#编译成CIL时，它会维持关于代码的大部分元数据。除此之外，可以利用反射枚举程序集中的所有类型，找出满足特定条件的那些。我们通过System.Type的实例来访问类型的元数据，该对象包含了对类型实例的成员进行枚举的方法。除此之外，可在被检查类型的特定对象上调用那些成员。

基于反射，人们发展出了一系列前所未有的编程模式。例如，反射允许枚举程序集中的所有类型及其成员。在这个过程中，可以创建对程序集API进行编档所需的存根（stub）。然后，可将通过反射获取的元数据与通过XML注释（使用/doc开关）创建的XML文档合并，从而创建API文档。类似地，程序员可以利用反射元数据来生成代码，从而将业务对象（business object）持久化（序列化）到数据库中。可以在显示对象集合的列表控件中使用反射。基于这个集合，列表控件可以利用反射来遍历集合中的一个对象的所有属性，并在列表中为每个属性都定义一个列。除此之外，通过调用每个对象的每个属性，列表控件可以使用对象中包含的数据来填充每一行和每一列——即使对象的数据类型在编译时是未知的。

.NET Framework所提供的XmlSerializer、ValueType和DataBinder类均在其实现中利用了反射技术。

17.1.1 使用System.Type访问元数据

读取类型的元数据，关键在于获得System.Type的一个实例，它代表了目标类型实例。System.Type提供了获取类型信息的所有方法。可以用它回答以下问题。

- 类型的名称是什么（Type.Name）？
- 类型是public的吗（Type.IsPublic）？
- 类型的基类型是什么（Type.BaseType）？
- 类型支持任何接口吗（Type.GetInterfaces()）？
- 类型是在哪个程序集中定义的（Type.Assembly）？
- 类型的属性、方法、字段是什么（Type.GetProperties()、Type.GetMethods()、Type.GetFields()）？
- 都有什么特性在修饰一个类型（Type.GetCustomAttributes()）？

还有其他成员未能一一列出，但总而言之，它们都提供了与特定类型有关的信息。很明显，现在的关键是获得对类型的Type对象的引用。我们主要通过object.GetType()和typeof()来达到这个目的。

> **注意**
>
> GetMethods()调用不能返回扩展方法，扩展方法只能作为实现类型的静态成员使用。

1. GetType()

object包含一个GetType()成员。因此，所有类型都包含该方法。调用GetType()可获得与原始对象对应的System.Type实例。代码清单17-1对此进行了演示，它使用来自DateTime的一个Type实例。输出17-1展示了结果。

代码清单17-1 使用Type.GetProperties()获取对象的public属性

```
DateTime dateTime = new DateTime();

Type type = dateTime.GetType();
foreach (
    System.Reflection.PropertyInfo property in
        type.GetProperties())
{
    Console.WriteLine(property.Name);
}
```

输出17-1

```
Date
Day
DayOfWeek
DayOfYear
Hour
Kind
Millisecond
Minute
Month
Now
UtcNow
Second
Ticks
TimeOfDay
Today
Year
```

在调用了GetType()之后，程序遍历从Type.GetProperties()返回的每个System.Reflection. PropertyInfo实例，并显示属性名。成功调用GetType()的关键在于获得一个对象实例。但在某些时候，这样的实例是无法获得的。例如，静态类是无法实例化的，所以没办法调用GetType()。

2. typeof()

获得Type对象的另一个办法是使用typeof表达式。typeof在编译时绑定到特定的Type实例，并直接获取类型作为参数。代码清单17-2演示了如何同时使用Enum.Parse()和typeof。

代码清单17-2 使用typeof()创建System.Type实例

```
using System.Diagnostics;
// ...
    ThreadPriorityLevel priority;
    priority = (ThreadPriorityLevel)Enum.Parse(
            typeof(ThreadPriorityLevel), "Idle");
// ...
```

Enum.Parse()获取标识了一个枚举的Type对象，然后将一个字符串转换成特定的枚举值。在本例中，它将"Idle"转换成System.Diagnostics.ThreadPriorityLevel.Idle。

17.1.2 成员调用

反射并非仅可以用来获取元数据。下一步是获取元数据，并动态调用它引用的成员。假定现在定义一个类，代表一个应用程序的命令行，并把它命名为CommandLineInfo。对于这个类来

说，最困难的地方在于如何在类中填充启动应用程序时的实际命令行数据。然而，利用反射，可以将命令行选项映射到属性名，并在运行时动态设置属性。代码清单17-3对此进行了演示。

代码清单17-3 动态调用成员

```
using System;
using System.Diagnostics;

public partial class Program
{
  public static void Main(string[] args)
  {
    string errorMessage;
    CommandLineInfo commandLine = new CommandLineInfo();
    if (!CommandLineHandler.TryParse(
        args, commandLine, out errorMessage))
    {
        Console.WriteLine(errorMessage);
        DisplayHelp();
    }

    if (commandLine.Help)
    {
        DisplayHelp();
    }
    else
    {
        if (commandLine.Priority !=
            ProcessPriorityClass.Normal)
        {
            // Change thread priority
        }

    }
    // ...

  }

  private static void DisplayHelp()
  {
    // Display the command-line help.
  }
}
```

```
using System;
using System.Diagnostics;

public partial class Program
{
  private class CommandLineInfo
  {
      public bool Help { get; set; }

      public string Out { get; set; }

      public ProcessPriorityClass Priority
      {
          get { return _Priority; }
          set { _Priority = value; }
```

```
        }
        private ProcessPriorityClass _Priority =
            ProcessPriorityClass.Normal;
    }

}
```

```
using System;
using System.Diagnostics;
using System.Reflection;

public class CommandLineHandler
{
  public static void Parse(string[] args, object commandLine)
  {
      string errorMessage;
      if (!TryParse(args, commandLine, out errorMessage))
      {
          throw new ApplicationException(errorMessage);
      }
  }

  public static bool TryParse(string[] args, object commandLine,
      out string errorMessage)
  {
      bool success = false;
      errorMessage = null;
      foreach (string arg in args)
      {
          string option;
          if (arg[0] == '/' || arg[0] == '-')
          {
              string[] optionParts = arg.Split(
                  new char[] { ':' }, 2);

              // Remove the slash|dash
              option = optionParts[0].Remove(0, 1);
              PropertyInfo property =
                  commandLine.GetType().GetProperty(option,
                      BindingFlags.IgnoreCase |
                      BindingFlags.Instance |
                      BindingFlags.Public);
              if (property != null)
              {
                  if (property.PropertyType == typeof(bool))
                  {
                      // Last parameters for handling indexers
                      property.SetValue(
                          commandLine, true, null);
                      success = true;
                  }
                  else if (
                      property.PropertyType == typeof(string))
                  {
                      property.SetValue(
                          commandLine, optionParts[1], null);
                      success = true;
                  }
                  else if (property.PropertyType.IsEnum)
                  {
                      try
```

```
                {
                    property.SetValue(commandLine,
                        Enum.Parse(
                            typeof(ProcessPriorityClass),
                            optionParts[1], true),
                        null);
                    success = true;
                }
                catch (ArgumentException )
                {
                    success = false;
                    errorMessage =
                        string.Format(
                            "The option '{0}' is " +
                            "invalid for '{1}'",
                        optionParts[1], option);
                }
            }
            else
            {
                success = false;
                errorMessage = string.Format(
                    "Data type '{0}' on {1} is not"
                    + " supported.",
                    property.PropertyType.ToString(),
                    commandLine.GetType().ToString());
            }
        }
                else
                {
                    success = false;
                    errorMessage = string.Format(
                        "Option '{0}' is not supported.",
                        option);
                }
            }
        }
        return success;
    }
}
```

虽然这是一个较长的程序，但代码结构是相当简单的。Main()首先实例化一个CommandLineInfo类。这个类型专门用来包含当前这个程序的命令行数据。每个属性都对应于程序的一个命令行选项，具体的命令行如输出17-2所示。

输出17-2

```
Compress.exe /Out:<file name> /Help
    /Priority:RealTime|High|AboveNormal|Normal|BelowNormal|Idle
```

CommandLineInfo对象被传给CommandLineHandler的TryParse()方法。该方法首先枚举每个选项，并分离出选项名（比如Help或Out）。在确定了名称之后，代码在CommandLineInfo对象上执行反射，查找同名的一个实例属性。如果找到这样的属性，就通过一个SetValue()调用，并指定与属性类型对应的数据，从而完成对属性的赋值。SetValue()的参数包括要设置值的对象、新值以及一个额外的index参数（除非属性是一个索引器，否则该参数就为null）。上述代码能处理3种属性类型：bool、string和枚举。在

枚举的情况下，我们要解析选项值，并将文本的枚举等价值赋给属性。如果TryParse()调用成功，方法会退出，而CommandLineInfo对象会使用来自命令行的数据进行初始化。

有趣的是，虽然CommandLineInfo是嵌套在Program中的一个private类，但CommandLineHandler在它上面执行反射没有任何问题，甚至可以调用它的成员。换言之，只要设置了恰当的代码访问安全性（code access security，CAS，参见第21章）权限，反射就可以绕过可访问性规则。例如，假定Out是private的，TryParse()方法仍然可以向其赋值。由于这一点，所以可以将CommandLineHandler转移到一个单独的程序集中，并在多个程序之间共享它，而且每个程序都有它们自己的Command LineInfo类。

在这个特定的例子中，是用PropertyInfo.SetValue()来调用CommandLineInfo的一个成员。此外，PropertyInfo还包含一个GetValue()方法，可以用它从属性中获取数据。然而，对于方法，有一个MethodInfo类可供利用，该类提供了一个Invoke()成员。MethodInfo和PropertyInfo都是从MemberInfo继承的（虽然并非直接），如图17-1所示。

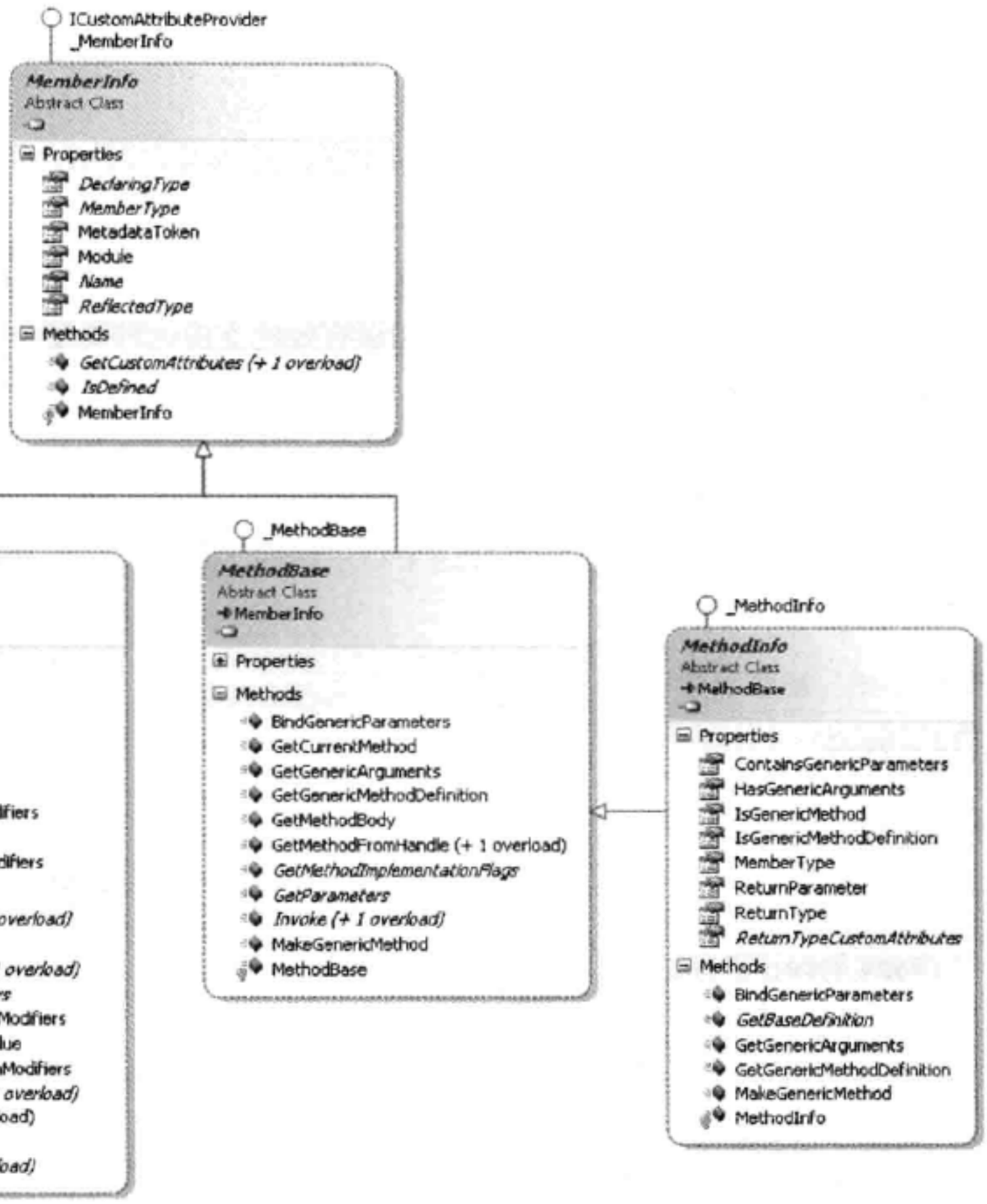


图17-1 MemberInfo的派生类

在这个例子中，之所以设置CAS权限来允许私有成员调用，是因为程序是从本地电脑中运行的。默认情况下，本地安装的程序是受信任区域的一部分，已被授予了恰当的权限。然而，从一个远程位置运行的程序需要被显式授予这样的权限。

17.1.3 泛型类型上的反射

从C#2.0开始，.NET Framework提供了在泛型类型上反射的能力。在泛型类型上执行运行时反射，可判断出类或方法是否包含泛型类型，以及它可能包含的任何类型参数。

1. 判断类型参数的类型

我们曾对非泛型类型使用typeof运算符，从而获取System.Type的实例。类似地，也可以对泛型类型或者泛型方法中的类型参数使用typeof运算符。代码清单17-4对Stack类的Add方法的类型参数应用了typeof运算符。

代码清单17-4 声明Stack<T>类

```
public class Stack<T>
{
    // ...
    public void Add(T i)
    {
        // ...
        Type t = typeof(T);
        // ...
    }
    // ...
}
```

获得类型参数的Type对象实例以后，就可以在类型参数上执行反射，从而判断它的行为，并针对具体类型来调整Add方法，使其能更有效地支持这种类型。

2. 判断类或方法是否支持泛型

在CLR 2.0加入的System.Type类中，新增了一系列方法来判断某个给定的类型是否支持泛型参数[①]。泛型实参是在实例化泛型类时提供的类型参数。如代码清单17-5所示，可以查询Type.ContainsGenericParameters这个布尔属性，判断类或方法是否包含尚未设置的泛型参数。

代码清单17-5 泛型反射

```
using System;

public class Program
{
  static void Main()
  {
      Type type;
      type = typeof(System.Nullable<>);
      Console.WriteLine(type.ContainsGenericParameters);
      Console.WriteLine(type.IsGenericType);

      type = typeof(System.Nullable<DateTime>);
      Console.WriteLine(!type.ContainsGenericParameters);
      Console.WriteLine(type.IsGenericType);
  }
}
```

① 泛型参数是“泛型形参”（generic parameter）和“泛型实参”（generic argument）的总称。本书仅在需要时才区分两者。——译者注

输出17-3展示了代码清单17-5的结果。

输出17-3

```
True
True
True
True
```

Type.IsGenericType是指示类型是否是泛型的布尔属性。

3. 为泛型类或方法获取类型参数

可以调用GetGenericArguments()方法从泛型类获取泛型实参（或类型参数）的列表。这样得到的是由System.Type实例构成的一个数组，这些System.Type实例的顺序就是它们作为泛型类的类型参数被声明的顺序。代码清单17-6反射一个泛型类型，获取它的每个类型参数。输出17-4展示了结果。

代码清单17-6 泛型类型反射

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

public partial class Program
{
  public static void Main()
  {
      Stack<int> s = new Stack<int>();

      Type t = s.GetType();

      foreach(Type type in t.GetGenericArguments())
      {
          System.Console.WriteLine(
              "Type parameter: " + type.FullName);
      }
      // ...
  }
}
```

输出17-4

```
Type parameter: System.Int32
```

17.2 特性

在讨论特性编程的细节之前，首先来研究一个演示其用途的例子。在代码清单17-3的CommandLineHandler例子中，是根据与属性名匹配的命令行选项来动态设置类的属性。然而，假如命令行选项是无效的属性名，这个办法就失效了。例如，/?就无法被支持。除此之外，使用这种方式无法指出选项是必需的还是可选的。

特性完美地解决了这个问题，它不依赖于选项名与属性名的完全匹配。利用特性，可以指定与被修饰的构造（本例就是命令行选项）有关的额外元数据。使用特性，可以将一个属性修饰为Required（必需的），并提供/?选项别名。换言之，特性是将额外数据关联

到属性（以及其他构造）的一种方式。

特性要放到它们修饰的那个构造之前的一对方括号中。例如，可以修改CommandLineInfo类来包含特性，如代码清单17-7所示。

代码清单17-7 用特性修饰属性

```
class CommandLineInfo
{
    [CommandLineSwitchAlias("?")]
    public bool Help
    {
        get { return _Help; }
        set { _Help = value; }
    }
    private bool _Help;

    [CommandLineSwitchRequired]
    public string Out
    {
        get { return _Out; }
        set { _Out = value; }
    }
    private string _Out;

    public System.Diagnostics.ProcessPriorityClass Priority
    {
        get { return _Priority; }
        set { _Priority = value; }
    }
    private System.Diagnostics.ProcessPriorityClass _Priority =
        System.Diagnostics.ProcessPriorityClass.Normal;
}
```

在代码清单17-7中，Help和Out属性均用特性进行了修饰。这些特性的目的是允许使用别名/?来取代/Help，以及指出/Out是必需的参数。我们的思路是从CommandLineHandler.TryParse()方法中启用对选项别名的支持。假定解析成功，可以检查是否指定了所有必要的开关。

有两个办法可以在同一个构造上合并多个特性。既可以在同一对方括号中以逗号分隔多个特性，也可以将每个特性放在它自己的一对方括号中，如代码清单17-8所示。

代码清单17-8 用多个特性来修饰一个属性

```
[CommandLineSwitchRequired]
[CommandLineSwitchAlias("FileName")]
public string Out
{
    get { return _Out; }
    set { _Out = value; }
}
```

```
[CommandLineSwitchRequired,
CommandLineSwitchAlias("FileName")]
public string Out
```

```
{
    get { return _Out; }
    set { _Out = value; }
}
```

除了修饰属性，还可以使用特性来修饰类、接口、结构、枚举、委托、事件、方法、构造器、字段、参数、返回值、程序集、类型参数和模块。对于其中大多数构造，都可以像代码清单17-8那样使用方括号语法来应用特性。但这个语法不适用于返回值、程序集和模块。

程序集的特性用于添加有关程序集的额外元数据。例如，Visual Studio的“项目向导”会生成一个AssemblyInfo.cs文件，其中包含了与程序集有关的大量特性。代码清单17-9展示了该文件的一个例子。

代码清单17-9 AssemblyInfo.cs中保存的程序集特性

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// General information about an assembly is controlled
// through the following set of attributes. Change these
// attribute values to modify the information
// associated with an assembly.
[assembly: AssemblyTitle("CompressionLibrary")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Michaelis.net")]
[assembly: AssemblyProduct("CompressionLibrary")]
[assembly: AssemblyCopyright("Copyright© Michaelis.net 2006-2012")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the types in this
// assembly not visible to COM components. If you need to
// access a type in this assembly from COM, set the ComVisible
// attribute to true on that type.
[assembly: ComVisible(false)]

// The following GUID is for the ID of the typelib if this project is exposed to COM
[assembly: Guid("417a9609-24ae-4323-b1d6-cef0f87a42c3")]

// Version information for an assembly consists
// of the following four values:
//
//      Major Version
//      Minor Version
//      Build Number
//      Revision
//
// You can specify all the values or you can
// default the Revision and Build Numbers
// by using the '*' as shown below:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

在这个例子中，一系列程序集特性（均附加assembly:前缀）定义了像公司、产品和程序集版本号这样的东西。与assembly:相似，为了指出一个特性作用于模块，需要为其附加module:前缀。对assembly:和module:特性的限制是它们必须出现在using指令之后，并在任何命名空间或类声明之前。上面列出的特性由Visual Studio项目向导生成，所有项目均应包含，以便使用与可执行文件或DLL的内容有关的信息来标记生成的二进制文件。

return:特性（如代码清单17-10所示）出现在方法声明之前，但使用相同的语法。

代码清单17-10 指定return:特性

```
[return: Description(
    "Returns true if the object is in a valid state.")]
public bool IsValid()
{
  // ...
  return true;
}
```

除了assembly:和return:，C#还允许显式指定module:、class:和method:等目标，分别对应于修饰模块、类和方法的特性。但是，正如前面展示的那样，class:和method:是可选的。

使用特性的一个方便之处在于，语言会自动考虑特性的命名规范问题，也就是在名称末尾添加Attribute后缀。前面的所有例子都没有主动添加该后缀——但事实上，所用的每个特性都符合命名规范。这是由于虽然在应用特性时允许使用全名（DescriptionAttribute、AssemblyVersionAttribute等），但C#规定后缀是可选的。应用特性时通常都不添加该后缀；只有在自己定义特性或者以内联方式使用特性的时候才要添加后缀，如typeof(DescriptionAttribute)。

规范

要向有公共类型的程序集应用AssemblyVersionAttribute。

考虑应用AssemblyFileVersionAttribute和AssemblyCopyrightAttribute以提供有关程序集的附加信息。

要应用以下程序集信息属性：

System.Reflection.AssemblyTitleAttribute

System.Reflection.AssemblyCompanyAttribute

System.Reflection.AssemblyProductAttribute

System.Reflection.AssemblyDescriptionAttribute

System.Reflection.AssemblyFileVersionAttribute

System.Reflection.AssemblyCopyrightAttribute

17.2.1 自定义特性

自定义特性很容易定义。特性是对象，所以要定义特性要定义类。从System.Attribute派生之后，一个普通的类就成为了特性。例如，可以创建一个CommandLineSwitchRequiredAttribute类，如代码清单17-11所示。

代码清单17-11 定义自定义特性

```
public class CommandLineSwitchRequiredAttribute : Attribute
{
}
```

有了这个简单的定义之后，就可以像代码清单17-7演示的那样使用这个特性。但目前还没有代码与这个特性对应。所以，包含这个特性的Out属性对命令行解析还不起作用。

规范

要为自定义特性类添加“Attribute”后缀。

17.2.2 查找特性

除了提供属性来返回类型成员之外，Type还提供了一些方法来获取对那个类型进行修饰的特性。类似地，所有反射类型（比如PropertyInfo和MethodInfo）都包含了一些成员，可以用来获取对类型进行修饰的特性列表。代码清单17-12定义了一个方法来返回命令行遗漏的必需的开关的列表。

代码清单17-12 获取自定义特性

```
using System;
using System.Collections.Specialized;
using System.Reflection;

public class CommandLineSwitchRequiredAttribute : Attribute
{
  public static string[] GetMissingRequiredOptions(
      object commandLine)
  {
      StringCollection missingOptions = new StringCollection();
      PropertyInfo[] properties =
          commandLine.GetType().GetProperties();

      foreach (PropertyInfo property in properties)
      {
          Attribute[] attributes =
               (Attribute[])property.GetCustomAttributes(
                  typeof(CommandLineSwitchRequiredAttribute),
                  false);
          if ((attributes.Length > 0) &&
              (property.GetValue(commandLine, null) == null))
          {
              missingOptions.Add(property.Name);
          }
      }
      return missingOptions.ToArray();
  }
}
```

用于查找特性的代码是相当简单的。给定一个PropertyInfo对象（通过反射来获取），你调用GetCustomAttributes()，指定要查找的特性，并指定是否检查任何重载的方法。另外，也可以调用GetCustomAttributes()方法而不指定特性类型，从而返回所有特性。

虽然可将查找CommandLineSwitchRequiredAttribute 特性的代码直接放到CommandLineHandler的代码中，但为了获得更好的对象封装，应该将代码放到CommandLineSwitchRequiredAttribute类自身中。这是自定义特性的常见模式。对于查找特性的代码来说，还有什么地方比特性类的静态方法中更好呢？

17.2.3 使用构造器来初始化特性

调用GetCustomAttributes()返回的是一个object数组，该数组能成功转型为Attribute数组。然而，由于本例中的特性没有任何实例成员，所以在返回的特性中，提供的唯一元数据信息就是它是否出现。然而，特性中还可以封装数据。代码清单17-13定义了一个CommandLineAliasAttribute 特性。这也是自定义特性，用于为命令行选项提供别名。例如，可以支持用户输入/Help或者/?作为缩写。类似地，可以将/S指定为/Subfolders的别名，指示命令应遍历所有子目录。

为了支持这个设计，需要为特性提供一个构造器。具体地说，针对别名，需要提供构造器来获取一个string参数。类似地，如果希望允许多个别名，就需要让构造器获取params string数组作为参数。

代码清单17-13 提供特性构造器

```
public class CommandLineSwitchAliasAttribute : Attribute
{
  public CommandLineSwitchAliasAttribute(string alias)
  {
      Alias = alias;
  }

  public string Alias
  {
      get { return _Alias; }
      private set { _Alias = value; }
  }
  private string _Alias;
}
```

```
class CommandLineInfo
{
  [CommandLineSwitchAlias("?")]
  public bool Help
  {
      get { return _Help; }
      set { _Help = value; }
  }
  private bool _Help;

  // ...
}
```

构造器唯一的限制在于，向某种构造应用特性时，只有字面值和类型（比如typeof(int)）才允许作为参数使用。这是为了确保它们能序列化到最终的CIL中。因此，无法在应用特性时调用静态方法。除此之外，提供构造器来获取System.DateTime类型的参数是没有多大意义的，因为没有System.DateTime字面量。

有了构造器调用之后，从PropertyInfo.GetCustomAttributes()返回的对象会使用指定的构造器参数来初始化，如代码清单17-14所示。

代码清单17-14 获取一个特定的特性，并检查它的初始化

```
PropertyInfo property =
    typeof(CommandLineInfo).GetProperty("Help");
CommandLineSwitchAliasAttribute attribute =
    (CommandLineSwitchAliasAttribute)
        property.GetCustomAttributes(
        typeof(CommandLineSwitchAliasAttribute), false)[0];
if(attribute.Alias == "?")
{
  Console.WriteLine("Help(?)");
};
```

除此之外，如代码清单17-15和代码清单17-16所示，可以在CommandLineAliasAttribute的GetSwitches()方法中使用类似的代码，返回由所有开关构成的一个字典集合（其中包括来自属性名的），并将每个名称同命令行对象的对应特性关联起来。

代码清单17-15 获取自定义特性实例

```
using System;
using System.Reflection;
using System.Collections.Generic;

public class CommandLineSwitchAliasAttribute : Attribute
{
  public CommandLineSwitchAliasAttribute(string alias)
  {
      Alias = alias;
  }

  public string Alias
  {
      get { return _Alias; }
      private set { _Alias = value; }
  }
  private string _Alias;

  public static Dictionary<string, PropertyInfo> GetSwitches(
      object commandLine)
  {
      PropertyInfo[] properties = null;
      Dictionary<string, PropertyInfo> options =
          new Dictionary<string, PropertyInfo>();

      properties = commandLine.GetType().GetProperties(
          BindingFlags.Public | BindingFlags.NonPublic |
          BindingFlags.Instance);
      foreach (PropertyInfo property in properties)
      {
          options.Add(property.Name.ToLower(), property);
          foreach (CommandLineSwitchAliasAttribute attribute in
              property.GetCustomAttributes(
              typeof(CommandLineSwitchAliasAttribute), false))
          {
              options.Add(attribute.Alias.ToLower(), property);
```

```
            }
        }
        return options;
    }
}
```

代码清单17-16 更新CommandLineHandler.TryParse()以处理别名

```
using System;
using System.Reflection;
using System.Collections.Generic;

public class CommandLineHandler
{
  // ...

  public static bool TryParse(
      string[] args, object commandLine,
      out string errorMessage)
  {
      bool success = false;
      errorMessage = null;

      Dictionary<string, PropertyInfo> options =
          CommandLineSwitchAliasAttribute.GetSwitches(
              commandLine);

      foreach (string arg in args)
      {
          PropertyInfo property;
          string option;
          if (arg[0] == '/' || arg[0] == '-')
          {
              string[] optionParts = arg.Split(
                  new char[] { ':' }, 2);
              option = optionParts[0].Remove(0, 1).ToLower();

              if (options.TryGetValue(option, out property))
              {
                  success = SetOption(
                      commandLine, property,
                      optionParts, ref errorMessage);
              }
              else
              {
                  success = false;
                  errorMessage = string.Format(
                      "Option '{0}' is not supported.",
                      option);
              }
          }
      }

      return success;
  }

  private static bool SetOption(
      object commandLine, PropertyInfo property,
      string[] optionParts, ref string errorMessage)
  {
      bool success;
```

```
if (property.PropertyType == typeof(bool))
{
    // Last parameters for handling indexers
    property.SetValue(
        commandLine, true, null);
    success = true;
}
else
{

    if ((optionParts.Length < 2)
    || optionParts[1] == ""
    || optionParts[1] == ":")
{
    // No setting was provided for the switch.
    success = false;
    errorMessage = string.Format(
        "You must specify the value for the {0} option.",
        property.Name);
}
else if (
    property.PropertyType == typeof(string))
{
    property.SetValue(
        commandLine, optionParts[1], null);
    success = true;
}
else if (property.PropertyType.IsEnum)
{
            success = TryParseEnumSwitch(
                commandLine, optionParts,
                property, ref errorMessage);
            }
            else
            {
                success = false;
                errorMessage = string.Format(
                    "Data type '{0}' on {1} is not supported.",
                    property.PropertyType.ToString(),
                    commandLine.GetType().ToString());
            }
        }
        return success;
    }
}
```

规范

要为必需的参数提供只能取值的属性（提供私有赋值函数）。

要提供构造器参数来初始化与必需的参数对应的属性。每个参数的名称都应该和对应的属性同名（大小写不同）。

避免提供构造器参数来初始化与可选参数对应的属性（因此，还要避免重载自定义属性构造器）。

17.2.4 System.AttributeUsageAttribute

大多数特性只修饰特定构造。例如，用CommandLineOptionAttribute修饰类或程序集是无意义的。为了避免不恰当地使用特性，可以用System.AttributeUsageAttribute修饰自定义特性。代码清单17-17演示了如何限制一个特性（即CommandLineOptionAttribute）的使用。

代码清单17-17 限制特性能修饰的构造

```
[AttributeUsage(AttributeTargets.Property)]
public class CommandLineSwitchAliasAttribute : Attribute
{
  // ...
}
```

特性被不恰当地使用，如代码清单17-18所示，会导致编译时错误，如输出17-5所示。

代码清单17-18 AttributeUsageAttribute限制了特性能应用于的目标

```
// ERROR: The attribute usage is restricted to properties
[CommandLineSwitchAlias("?")]
class CommandLineInfo
{
}
```

输出17-5

```
...Program+CommandLineInfo.cs(24,17): error CS0592: Attribute
'CommandLineSwitchAlias' is not valid on this declaration type. It is
valid on 'property, indexer' declarations only.
```

AttributeUsageAttribute的构造器获取一个AttributesTargets标志。这个枚举提供了“运行时”允许特性修饰的所有目标的列表。例如，要允许用CommandLineSwitchAliasAttribute修饰字段，应该像代码清单17-19那样更新AttributeUsageAttribute类。

代码清单17-19 使用AttributeUsageAttribute限制特性的使用

```
// Restrict the attribute to properties and methods
[AttributeUsage(
  AttributeTargets.Field | AttributeTargets.Property)]
public class CommandLineSwitchAliasAttribute : Attribute
{
  // ...
}
```

规范

要对自定义特性应用AttributeUsageAttribute类。

17.2.5 命名参数

使用AttributeUsageAttribute除了限制特性能够修饰的东西，还可以用它指定是否允许特性在一个构造上进行多次复制。代码清单17-20展示了具体的语法。

代码清单17-20 使用命名参数

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple=true)]
public class CommandLineSwitchAliasAttribute : Attribute
{
  // ...
}
```

上述语法有别于先前讨论的构造器初始化语法。AllowMultiple是**命名参数**（named parameter），它类似于C# 4.0开始为方法参数添加的“命名参数”语法。命名参数在特性构造器调用中设置特定的公共属性和字段——即使构造器不包括对应的参数。命名参数虽然是可选的，但它允许对特性的额外实例数据进行设置，同时无需提供一个对应的构造器参数。在本例中，AttributeUsageAttribute包含一个名为AllowMultiple的公共成员。所以，在使用特性的时候，可以通过一次命名参数赋值来设置该成员。对命名参数的赋值只能放到构造器的最后一部分进行。任何显式声明的构造器参数都必须在它之前完成赋值。

有了命名参数之后，就可以直接对特性的数据进行赋值，而不必为特性属性的每一种组合都提供对应的构造器。由于一个特性的许多属性都是可选的，所以命名参数在大多数情况下都非常有用。

初学者主题：FlagsAttribute

第8章讲述了枚举，并用一个“高级主题”介绍了FlagsAttribute。这是.NET Framework所定义的一个特性，应用于包含了一组标志（flag）值的枚举。下面以一个“初学者主题”的形式重复第8章的内容。先来看看代码清单17-21的代码。

代码清单17-21 使用FlagsAttribute

```
// FileAttributes defined in System.IO.

[Flags]  // Decorating an enum with FlagsAttribute.
public enum FileAttributes
{
  ReadOnly =          1<<0,      // 0000000000000001
  Hidden =            1<<1,      // 0000000000000010
  // ...
}
```

```
using System;
using System.Diagnostics;
using System.IO;

class Program
{
  public static void Main()
  {
      // ...

      string fileName = @"enumtest.txt";
      FileInfo file = new FileInfo(fileName);

      file.Attributes = FileAttributes.Hidden |
          FileAttributes.ReadOnly;
```

```
            Console.WriteLine("\"{0}\" outputs as \"{1}\"",
                file.Attributes.ToString().Replace(",", " |"),
                file.Attributes);

            FileAttributes attributes =
                (FileAttributes)Enum.Parse(typeof(FileAttributes),
                file.Attributes.ToString());

            Console.WriteLine(attributes);

            // ...
        }
    }
```

输出17-6展示了代码清单17-21的结果。

输出17-6

```
"ReadOnly | Hidden" outputs as "ReadOnly, Hidden"
```

作为标志的枚举值可以组合使用。此外，它改变了ToString()和Parse()方法的行为。例如，为已用FlagsAttribute修饰了的枚举调用ToString()方法，会为已设置的每个枚举标志输出对应的字符串。在代码清单17-21中，file.Attributes.ToString()返回"ReadOnly, Hidden"。相反，如果没有FlagsAttribute标志，返回的就是3。如果两个枚举值相同，ToString()返回的就是第一个值。然而，正如以前说过的那样，使用需谨慎，这样转换得到的文本是无法本地化的。

将值从字符串解析成枚举也是可行的，只要每个枚举值标识符都以一个逗号分隔即可。

需要注意的是，FlagsAttribute并不会自动指派唯一的标志值，或检查它们是否具有唯一值。还是必须显式指派每个枚举项的值。

1. 预定义特性

AttributeUsageAttribute特性有一个特点是本书迄今为止创建的所有自定义特性都不具备的。这个特性会影响编译器的行为，造成编译器有时会报告错误。和早先用于获取CommandLineRequiredAttribute和CommandLineSwitchAliasAttribute的反射代码不同，AttributeUsageAttribute没有运行时代码，而是由编译器内建了对它的支持。

AttributeUsageAttribute是预定义特性。这种特性不仅提供了与它们修饰的构造有关的额外元数据，而且运行时和编译器在利用这种特性的功能时，行为也有所不同。AttributeUsageAttribute、FlagsAttribute、ObsoleteAttribute和ConditionalAttribute等都是预定义特性。它们都包含了只有CLI提供者或者编译器才能提供的特定行为，因为没有可用的扩展点用于额外的非自定义特性，而自定义特性又是完全被动的。代码清单17-21演示了两个预定义特性，第18章还会演示另外几个。

2. System.ConditionalAttribute

在一个程序集中，System.Diagnostics.ConditionalAttribute特性的行为有点儿像#if/#endif等预处理器标识符。然而，使用System.Diagnostics.ConditionalAttribute并不能从程序集中消除CIL代码。我们是利用它为一个调用赋予无操作（no-op）行为。换言之，使其成为一个什么都不做的指令。代码清单17-22演示了这个概念，输出17-7展示了结果。

代码清单17-22 使用System.ConditionalAttribute

```
#define CONDITION_A

using System;
using System.Diagnostics;

public class Program
{
  public static void Main()
  {
      Console.WriteLine("Begin...");
      MethodA();
      MethodB();
      Console.WriteLine("End...");
  }

  [Conditional("CONDITION_A")]
  static void MethodA()
  {
      Console.WriteLine("MethodA() executing...");
  }

  [Conditional("CONDITION_B")]
  static void MethodB()
  {
      Console.WriteLine("MethodB() executing...");
  }
}
```

输出17-7

```
Begin...
MethodA() executing...
End...
```

这个例子定义了CONDITION_A，所以MethodA()能正常执行。但是，没有使用#define或者csc.exe /Define选项来定义CONDITION_B。所以，在这个程序集中，对Program.MethodB()的所有调用都会“什么都不做”。

从功能上说，ConditionalAttribute类似于用一对#if/#endif把方法调用包围起来。然而，它的语法显得更清晰，因为开发者只需为目标方法添加ConditionalAttribute特性，无需对调用者本身进行任何修改。

注意，C#编译器在编译时会注意到被调用方法上的特性设置，假定预处理器标识符存在，它就会清除对方法的任何调用。还要注意的是，ConditionalAttibute不影响目标方法本身的已编译CIL代码（除了添加必要的特性元数据之外）。ConditionalAttibute会在编译期间通过移除调用的方式来影响调用点。这进一步澄清了跨越程序集进行调用时ConditionalAttribute和#if/#endif的区别。由于被这个特性修饰的方法仍会进行编译，并包含在目标程序集中，所以具体是否调用一个方法，不是取决于被调用者所在的程序集中的预处理器标识符，而是取决于调用者所在的程序集中的预处理器标识符。换句话说，假如创建了第二个程序集，并在其中定义了CONDITION_B，那么在第二个程序集中，对Program.MethodB()的任何调用都会执行。在许多需要进行跟踪和测试的情

形下，这都是一个非常有用的特点。事实上，对System.Diagnostics.Trace和System.Diagnostics.Debug的调用就是利用了这个特点（ConditionalAttribute和TRACE/DEBUG预处理器标识符配合使用）。

由于只要预处理器标识符没有定义，方法就不会执行，因此假如一个方法包含了out参数，或者返回类型不为void，就不能使用ConditionalAttribute，否则会造成一个编译时错误。之所以要进行这个限制，是因为如果不进行限定的话，在被这个特性修饰的方法中，有可能出现任何代码都不会执行的情况。在这种情况下，就不知道该将什么返回给调用者。类似地，属性不能用ConditionalAttribute来修饰。ConditionalAttribute的AttributeUsage（参见17.2.4节）是AttributeTargets.Class（自.NET 2.0起）和AttributeTargets.Method。这允许你将该特性用于方法或类。但将它用于类时比较特殊，因为只允许为派生于System.Attribute的类使用ConditionalAttribute。

用ConditionalAttribute修饰自定义特性时（自.NET 2.0起可以这样做），只有在调用程序集中定义了条件字符串的前提下，才能通过反射来获取那个自定义特性。没有这样的条件字符串，就不能通过反射来查找自定义特性。

3. System.ObsoleteAttribute

前面讲过，预定义特性会影响编译器和/或“运行时”的行为。ObsoleteAttribute是特性影响编译器行为的另一个例子。ObsoleteAttribute用于编制代码的版本，向调用者指出一个特定的成员或类型已过时。代码清单17-23展示了一个例子。如输出17-8所示，一个成员在使用ObsoleteAttribute进行了修饰之后，对调用它的代码进行编译，会造成编译器显示一条警告消息（也可选择报错）。

代码清单17-23 使用ObsoleteAttribute

```
class Program
{
  public static void Main()
  {
      ObsoleteMethod();
  }

  [Obsolete]
  public static void ObsoleteMethod()
  {
  }
}
```

输出17-8

```
c:\SampleCode\ObsoleteAttributeTest.cs(24,17): warning CS0612:
Program.ObsoleteMethod()' is obsolete
```

在这个例子中，ObsoleteAttribute只是显示一个警告。然而，这个特性还提供了另外两个构造器。第一个是ObsoleteAttribute(string message)，它能在编译器生成的报告过时的消息上附加额外的消息。在这个消息中，最好是告诉用户用什么来替代已过时的代码。第二个是ObsoleteAttribute(string,Boolean)，布尔参数指定是否强制将警告视为错误。

ObsoleteAttribute允许第三方向开发者通知已经过时的API。警告（而不是错误）允许原来的API继续发挥作用，直到开发者更新其调用代码为止。

4. 与序列化相关的特性

通过预定义特性，.NET Framework允许将对象序列化成一个流，使它们以后能反序列化为对象。这样一来，可以在关闭一个应用程序之前，方便地将一个文档类型的对象存盘。之后，文档可以反序列化，便于用户继续对它进行处理。

虽然对象可能相当复杂，而且可能链接到其他许多也需要序列化的对象类型，但序列化框架其实很容易使用。要想序列化对象，唯一需要做的就是包含一个System.SerializableAttribute。有了这个特性，一个formatter（格式化程序）类就可以对该序列化对象执行反射，并把它复制到一个流中，如代码清单17-24所示。

代码清单17-24 使用System.SerializableAttribute来保存文档

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

class Program
{
  public static void Main()
  {
      Stream stream;
      Document documentBefore = new Document();
      documentBefore.Title =
      "A cacophony of ramblings from my potpourri of notes";
      Document documentAfter;

      using (stream = File.Open(
          documentBefore.Title + ".bin", FileMode.Create))
      {
          BinaryFormatter formatter =
              new BinaryFormatter();
          formatter.Serialize(stream, documentBefore);
      }

      using (stream = File.Open(
          documentBefore.Title + ".bin", FileMode.Open))
      {
          BinaryFormatter formatter =
              new BinaryFormatter();
          documentAfter = (Document)formatter.Deserialize(
              stream);
      }

      Console.WriteLine(documentAfter.Title);
  }
}
```

```
// Serializable classes use SerializableAttribute.
[Serializable]
class Document
{

  public string Title = null;
  public string Data = null;
```

```
    [NonSerialized]
    public long _WindowHandle = 0;

    class Image
    {
    }
    [NonSerialized]
    private Image Picture = new Image();
}
```

输出17-9展示了代码清单17-24的结果。

输出17-9

```
A cacophony of ramblings from my potpourri of notes
```

代码清单17-24序列化并反序列化了一个Document对象。为了执行序列化，只需实例化一个formatter（本例使用System.Runtime.Serialization.Formatters.Binary.BinaryFormatter），然后为合适的流对象调用Serialization()。为了执行反序列化，只需调用formatter的Deserialize()方法，并指定包含了已序列化对象的流作为参数。然而，由于Deserialize()返回的是object类型，所以还需要把它转型为最初的类型。

注意，序列化是针对整个对象图（object graph，所有的项通过字段与已序列化对象[Document]关联）发生的。因此，对象图中的所有字段也必须是可序列化的。

- System.NonSerializable。不可序列化的字段应使用System.NonSerializable特性来修饰。它告诉序列化框架忽略这些字段。不应持久化的字段也应使用这个特性来修饰。例如，密码和Windows句柄就不应序列化。Windows句柄之所以不应序列化，是因为每次重新创建一个窗口时，它们就会发生改变。而密码之所以不应序列化，是因为序列化到一个流中的数据是没有加密的，可被轻松获取。在图17-2中，我们用“记事本”程序打开了一个已序列化的文档。

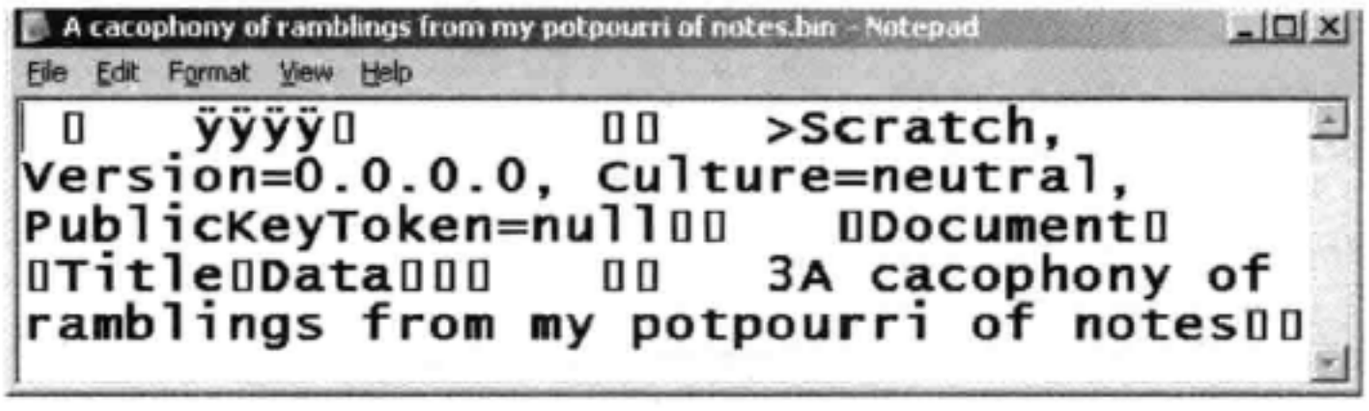


图17-2 BinaryFormatter没有对数据进行加密

代码清单17-24设置了Title字段，但最终生成的*.BIN文件包含的是明文。

- 提供自定义序列化。添加加密功能的一个办法是提供自定义序列化。先不管加密和解密的复杂性，为了提供自定义序列化，除了使用SerializableAttribute之外，还要实现ISerializable接口。这个接口只要求实现GetObjectData()方法。然而，这仅足以支持序列化。为了同时支持反序列化，需要提供一个构造器来获取System.Runtime.Serialization.SerializationInfo与System.Runtime.Serialization.StreamingContext类型的参数，如代码清单17-25所示。

代码清单17-25 实现System.Runtime.Serialization.ISerializable

```
using System;
using System.Runtime.Serialization;

[Serializable]
class EncryptableDocument :
    ISerializable
{
  public EncryptableDocument(){ }

  enum Field
  {
      Title,
      Data
  }
  public string Title;
  public string Data;

  public static string Encrypt(string data)
  {
      string encryptedData = data;
      // Key-based encryption ...
      return encryptedData;
  }

  public static string Decrypt(string encryptedData)
  {
      string data = encryptedData;
      // Key-based decryption...
      return data;
  }

#region ISerializable Members
      public void GetObjectData(
      SerializationInfo info, StreamingContext context)
  {
      info.AddValue(
          Field.Title.ToString(), Title);
      info.AddValue(
          Field.Data.ToString(), Encrypt(Data));
  }

  public EncryptableDocument(
      SerializationInfo info, StreamingContext context)
  {
      Title = info.GetString(
          Field.Title.ToString());
      Data = Decrypt(info.GetString(
          Field.Data.ToString()));
  }
#endregion
}
```

简单地说，System.Runtime.Serialization.SerializationInfo对象是由“名称/值”对构成的一个集合。在序列化的时候，GetObject()的实现会调用AddValue()。为了反转这个过程，需要调用某个Get*()成员。在本例中，我们在序列化和反序列化之前分别执行加密和解密。

- 序列化的版本控制。序列化框架还有一个问题需要注意：版本控制。像文档这样的对象可能用程序集的一个版本序列化，以后用一个新版本来反序列化。稍不注意，就会造成版本不兼容的问题。来看看表17-1描述的情形。

表17-1 新版本的反序列化引发异常

步 骤	描 述	代 码
1	定义类，并用System.SerializableAttribute修饰	`[Serializable]` `class Document` `{` `  public string Title;` `  public string Data;` `}`
2	添加任意可序列化类型的一个或两个（public或private）字段	
3	将对象序列化到名为*.v1.bin的文件中	`Stream stream;` `Document documentBefore = new Document();` `documentBefore.Title = "A cacophony of ramblings from my potpourri of notes";` `Document documentAfter;` `using (stream = File.Open(documentBefore.Title + ".bin", FileMode.Create))` `{` `  BinaryFormatter formatter = new BinaryFormatter();` `  formatter.Serialize(stream, documentBefore);` `}`
4	在可序列化类中添加一个新字段	`[Serializable]` `class Document` `{` `  public string Title;` `  public string Author;` `  public string Data;` `}`
5	将*v1.bin文件反序列化为新对象（Document）版本	`using (stream = File.Open(documentBefore.Title + ".bin", FileMode.Open))` `{` `  BinaryFormatter formatter = new BinaryFormatter();` `  documentAfter = (Document)formatter.Deserialize(stream);` `}`

令人惊讶的是，虽然唯一的改动就是添加了一个新字段，但对原始文件执行反序列化，就会引发一个System.Runtime.Serialization.SerializationException异常。这是由于formatter会在流中查找与新字段对应的数据。找不到当然会引发异常。

为了避免这个问题，.NET Framework 2.0及其后续版本添加了一个System.Runtime.Serialization.OptionalFieldAttribute。如果需要向后兼容性，就必须使用OptionalFieldAttribute来修饰序列化的字段——包括私有字段。当然，假如以后的某个

版本规定该字段是必需的而非可选的，就不要这样修饰了。

遗憾的是，在之前的.NET Framework版本中是不支持System.Runtime.Serialization.OptionalFieldAttribute的。相反，需要实现ISerializable（就像刚才为加密所做的那样），只保存并获取那些可用的字段。例如，假定新增了一个Author字段，那么必须像代码清单17-26那样做以确保向后兼容.NET Framework 2.0之前的版本。

代码清单17-26 如何确保向后兼容于2.0 Framework之前的版本

```
[Serializable]
public class VersionableDocument : ISerializable
{
  enum Field
  {
      Title,
      Author,
      Data,
  }

  public VersionableDocument()
  {
  }

  public string Title;
  public string Author;
  public string Data;

  #region ISerializable Members
  public void GetObjectData(
      SerializationInfo info, StreamingContext context)
  {
      info.AddValue(Field.Title.ToString(), Title);
      info.AddValue(Field.Author.ToString(), Author);
      info.AddValue(Field.Data.ToString(), Data);
  }
  public VersionableDocument(
      SerializationInfo info, StreamingContext context)
  {
      foreach(SerializationEntry entry in info)
      {
          switch ((Field)Enum.Parse(typeof(Field), entry.Name))
          {
              case Field.Title:
                  Title = info.GetString(
                        Field.Title.ToString());
                  break;
              case Field.Author:
                  Author = info.GetString(
                      Field.Author.ToString());
                  break;
              case Field.Data:
                  Data = info.GetString(
                      Field.Data.ToString());
                  break;
          }
      }
  }
  #endregion
}
```

GetObjectData()简单地对所有字段进行序列化（假定版本1不需要打开版本2生成的文档）。但反序列化的时候不能简单地调用GetString("Author")，因为假如不存在此项，就会引发异常。相反，应该遍历info中的所有项，并单独获取它们。

高级主题：System.SerializableAttribute和CIL

在许多方面，序列化特性的行为都和自定义特性相似。在运行时，formatter类搜索这些特性，如果特性存在，就对类进行适当的格式化。然而，System.SerializableAttribute之所以不是简单的自定义特性，是由于CIL为序列化类采用了特殊的header表示法。代码清单17-27展示了CIL代码中Person类header。

代码清单17-27 SerializableAttribute的CIL

```
class auto ansi serializable nested private
  beforefieldinit Person
  extends [mscorlib]System.Object
{
} // end of class Person
```

而特性（包括大多数预定义特性）通常出现在类定义的内部（参见代码清单17-28）。

代码清单17-28 普通特性的CIL

```
.class private auto ansi beforefieldinit Person
       extends [mscorlib]System.Object
{
  .custom instance void CustomAttribute::.ctor() =
      ( 01 00 00 00 )
} // end of class Person
```

在代码清单17-28中，CustomAttribute是负责修饰的特性的全名。

SerializableAttribute转换为元数据表中的一个设置位（set bit），这使其成为所谓的**伪特性**（pseudoattribute）——在元数据表中对位或字段进行设置的特性。

17.3 使用动态对象进行编程

随着动态对象在C# 4.0中的引入，许多编程情形都得到了简化，以前无法实现的一些编程情形现在也能实现了。从根本上说，使用动态对象进行编程，开发人员可以通过动态调度机制对设想的操作进行编码。“运行时”会在程序执行时对这个机制进行解析，而不是由编译器在编译时验证和绑定。

为什么要推出动态对象？从较高的级别上说，经常都有对象天生就不适合赋予一个静态类型。例子包括从XML/CSV文件、数据库表、Internet Explorer DOM或者COM的IDispatch接口加载数据，或者调用用动态语言写的代码（比如调用IronPython对象中的代码）。C# 4.0的动态对象提供了一个通用的解决方案与“运行时”环境对话。这种对象在编译时不一定有定义好的结构。在C# 4.0的动态对象的初始实现中，提供了以下4个绑定方式。

（1）针对底层CLR类型使用反射。

（2）调用自定义IDynamicMetaObjectProvider，它使一个DynamicMetaObject变得可用。

（3）通过COM的IUnknown和IDispatch接口来调用。

（4）调用由动态语言（如IronPython）定义的类型。

我们将讨论前两种方式。它们的基本原则可以很好地应用于剩余的情况——COM互操作性和动态语言互操作性。

17.3.1 使用dynamic调用反射

反射的关键功能之一就是动态查找和调用特定类型的成员。这要求在执行时识别成员名或其他特征，比如一个特性（参见代码清单17-3）。然而，C# 4.0新增的动态对象提供了更简单的办法来通过反射调用成员。但这个技术的限制在于，编译时需要知道成员名和签名（参数个数，以及指定的参数是否和签名类型兼容）。代码清单17-29（输出17-10）展示了一个例子。

代码清单17-29 使用“反射”的动态编程

```
using System;

// ...
dynamic data =
  "Hello!  My name is Inigo Montoya";
Console.WriteLine(data);
data = (double)data.Length;
data = data*3.5 + 28.6;
if(data == 2.4 + 112 + 26.2)
{
  Console.WriteLine(
      "{0} makes for a long triathlon.", data);
}
else
{
  data.NonExistentMethodCallStillCompiles();
}
// ...
```

输出17-10

```
Hello!  My name is Inigo Montoya
140.6 makes for a long triathlon.
```

这个例子没有显式的代码来判断对象类型，查找一个特定的MemberInfo实例并调用它。相反，data声明为dynamic类型，并直接在它上面调用方法。在编译时，不会检查指定的成员是否可用，甚至不会检查dynamic对象的基础类型是什么。所以，只要语法是有效的，编译时就可以发出任何调用。在编译时，是否有一个对应的成员是无关紧要的。

然而，类型安全性并没有被完全放弃。对于标准CLR类型（比如代码清单17-29使用的那些），一般在编译时用于非dynamic类型的类型检查器会在执行时为dynamic类型调用。所以，如果在执行时发现事实上没有这个成员，调用就会引发一个Microsoft.CSharp.RuntimeBinder.RuntimeBinderException。

注意，这个技术不如本章早些时候描述的反射技术灵活，虽然它的API无疑要简单一些。使用动态对象时，一个关键的区别在于需要在编译时识别签名，而不是在运行时再判断这些东西（如成员名），就像我们解析命令行实参时所做的那样。

17.3.2 dynamic的原则和行为

代码清单17-29和对它进行描述的文字揭示了dynamic数据类型的几个特征。

- dynamic是告诉编译器生成代码的指令。dynamic涉及一个解释机制。当“运行时”遇到一个dynamic调用时，它可以将请求编译成CIL，然后调用新编译的调用（参见稍后的“高级主题：dynamic揭秘”）。
  将类型指定成dynamic后，在概念上是“包装”（wrap）了原始类型。这样一来，便不会发生编译时验证。除此之外，在运行时调用一个成员时，“包装器”（wrapper）会解释调用，并相应地调度它（或拒绝它）。在dynamic对象上调用GetType()，可揭示出dynamic实例的基础类型——这个方法不会返回dynamic作为类型。
- 任何类型都会转换成dynamic。在代码清单17-29中，我们成功地将一个值类型（double）和一个引用类型（string）转型为dynamic。事实上，所有类型都能成功转换成dynamic对象。存在从任何引用类型到dynamic的一个隐式转换。类似地，存在从一个值类型到dynamic的一个隐式转换（一个装箱转换）除此之外，dynamic到dynamic也有一个隐式转换。这看起来可能很明显，但dynamic不是简单地将“指针”（地址）从一个位置复制到另一个位置，它要复杂得多。
- 从dynamic到一个替代类型的成功转换要依赖于基础类型的支持。从dynamic对象转换成一个标准CLR类型是一个显式转型，比如(double)data.Length。一点都不奇怪，如果目标类型是值类型，那么需要一次拆箱转换。如果基础类型支持向目标类型的转换，对dynamic执行的转换也会成功。
- dynamic类型的基础类型在每次赋值时都可能改变。隐式类型的变量（var）不能重新赋值成一个不同的类型。和它不同，dynamic涉及一个解释机制，要先编译再执行基础类型的代码。所以，可以将基础类型实例更换为一个完全不同的类型。这会出现另一个解释调用位置，它需要在调用前编译。
- 验证基础类型上是否存在指定的签名要推迟到运行时才进行——但至少会进行。以方法调用person.NonExistentMethodCallStillCompiles()为例，编译器不会验证dynamic类型是否真的是存在这样的一个操作[①]。这个验证要等到代码执行时，由“运行”时执行。如果代码永不执行，即使它的外层代码已经执行（如person.NonExistentMethodCallStillCompiles()的情况），也不会发生验证和对成员的绑定。
- 任何dynamic成员调用返回的都是一个dynamic对象（data = data*3.5 + 28.6）。对dynamic对象的任何成员进行调用都将返回一个dynamic对象。所以，像data.ToString()这样的调用会返回一个dynamic对象，而不是返回基础的string类型。但在执行时，如果在dynamic对象上调用GetType()，代码是编译好的，所以会返回编译好的类型。
- 如果指定的成员在运行时不存在，“运行时”会引发Microsoft.CSharp.RuntimeBinder.RuntimeBinderException异常。如果在执行时试图调用一个成员，“运行时”会验证成员调用是有效的（例如，在反射的情况下，签名是类型兼容的）。如果方法签名不兼容，“运行时”会引发Microsoft.CSharp.

① 这个方法是作者杜撰的，不存在的，但仍然能通过编译。——译者注

RuntimeBinder.RuntimeBinderException。

- 用dynamic来实现的反射不支持扩展方法。和使用System.Type实现的反射一样，用dynamic实现的反射不支持扩展方法。仍然只有在实现类型（如System.Linq.Enumerable）上才可以调用扩展方法，不能直接在扩展的类型上调用。
- 究其根本，dynamic是一个System.Object。由于任何对象都能成功转换成dynamic，而dynamic能显式转换成一个不同的对象类型，所以dynamic在行为上就像System.Object。类似于System.Object，它甚至会为它的默认值返回null（default(dynamic)），表明它是一个引用类型。dynamic特殊的动态行为只在调用时才会出现，这个行为是将它同一个System.Object区分开的关键。

高级主题：dynamic揭密

ILDASM揭示出在CIL中，dynamic类型实际是一个System.Object。事实上，如果没有任何调用，dynamic类型的声明和System.Object没有区别。然而，一旦调用它的一个成员，区别就变得明显了。为了调用成员，编译器要声明System.Runtime.CompilerServices.CallSite<T>类型的一个变量。T视成员签名而变化。但即使简单如ToString()这样的调用，也需要实例化以下类型：CallSite<Func<CallSite, object,string>>。除此之外，还会动态定义一个方法，该方法可通过参数CallSite site,object dynamicTarget和string result进行调用。其中，site是调用点本身。dynamicTarget是要在上面调用方法的对象，而result是ToString()方法调用的基础返回值。注意，不是直接实例化CallSite<Func<CallSite _site, object dynamicTarget, string result>>，而是通过一个Create()工厂方法来实例化它。（(Create()获取Microsoft.CSharp.RuntimeBinder.CSharpConvertBinder类型的一个参数。）在得到CallSite<T>的一个实例后，最后一步是调用CallSite<T>.Target()来调用实际的成员。

在执行时，.NET Framework在幕后会用“反射”来查找成员并验证签名是否匹配。然后，“运行时”生成一个表达式树，它代表由调用点定义的动态表达式。树表达式编译好以后，我们就得到了和本来应该由编译器生成的结果相似的CIL结果。然后，这些CIL代码被“注入”调用点，并通过一个委托调用来实际地触发调用。由于CIL现已注入调用点，所以后续的调用不会再产生反射和编译的开销。

17.3.3 为什么需要动态绑定

除了反射，还可定义自定义类型来进行动态调用。例如，假定需要通过动态调用来获取一个XML元素的值。可以不使用代码清单17-30的强类型语法，而是像代码清单17-31那样通过动态调用来调用person.FirstName和person.LastName。

代码清单17-30 不用dynamic在运行时绑定到XML元素

```
using System;
using System.Xml.Linq;

// ...
XElement person = XElement.Parse(
   @"<Person>
     <FirstName>Inigo</FirstName>
     <LastName>Montoya</LastName>
```

```
</Person>");

Console.WriteLine("{0} {1}",
  person.Descendants("FirstName").FirstOrDefault().Value,
  person.Descendants("LastName").FirstOrDefault().Value);
// ...
```

虽然代码清单17-30的代码看起来并不复杂，但和代码清单17-31相比就显得比较“难看”了。代码清单17-31使用动态类型的对象来达到和代码清单17-30相同的目的。

代码清单17-31 使用dynamic在运行时绑定到XML元素

```
using System;

// ...
dynamic person = DynamicXml.Parse(
  @"<Person>
      <FirstName>Inigo</FirstName>
      <LastName>Montoya</LastName>
  </Person>");

  Console.WriteLine("{0} {1}",
      person.FirstName, person.LastName);
// ...
```

优势是明显的，但这是不是说动态编程优于静态编译呢？

17.3.4 静态编译与动态编程的比较

代码清单17-31的功能和代码清单17-30是一样的，但有一个非常重要的区别。代码清单17-30是完全静态类型的。也就是说，所有类型及其成员签名在编译时都得到了验证。方法名必须匹配，而且所有参数都要通过类型兼容性检查。这是C#和我一直在强调的一个贯穿本书的关键功能。

相反，代码清单17-31几乎没有静态类型的代码，person变量是dynamic类型。所以，不会在编译时验证person是否真的有一个FirstName或LastName属性（或者其他成员）。此外，在IDE中写代码时，没有智能感知功能可帮助你判断person的任何成员。

由于类型变得不确定，所以似乎会造成功能的显著下降。作为C# 4.0新增的一个功能，为什么C#会允许出现这样的情况？让我们再次研究代码清单17-31。注意用于获取"FirstName"元素的调用：Element.Descendants("LastName").FirstOrDefault().Value。在这个代码清单中，是用一个字符串（"LastName"）标识元素名。然而，编译时没有验证这个字符串是否正确。如果大小写和元素名不一致，或者名称存在一个空格，编译仍会成功——即使调用Value属性时会引发一个NullReferenceException。除此之外，编译器根本不会验证是否真的存在一个"FirstName"元素。如果它不存在，还会引发NullReferenceException。换言之，虽然有编译时的类型安全性的好处，但在访问XML元素中存储的动态数据时，这种类型安全性并没有多大优势。

在编译时对获取的XML元素的验证方面，代码清单17-31并不比代码清单17-30好。如果真的有大小写不匹配，或者不存在FirstName元素的情况，仍会引发一个异常[①]。然而，

① 不能在FirstName属性调用中使用一个空格。但假如XML支持在元素名中使用空格，这就是动态调用的一个缺陷了。所以，让我们忽略这个事实。

将代码清单17-31中用于访问名字的调用（person.FirstName）和代码清单17-30中的调用进行比较，前者显然要简单得多。总之，在某些情况下，类型安全性不会——而且可能不能——进行特定的检查。在这种情况下，执行只在运行时验证的动态调用，而不是在编译时验证，代码会变得更易读、更简洁。当然，如果能在编译时验证，静态类型的编程就是首选的，因为这时也许能使用一些易读的、简洁的API。然而，当它的作用不大的时候，C# 4.0就允许你写更简单的代码，而不必刻意追求编译时的类型安全性。

17.3.5 实现自定义动态对象

代码清单17-31包含一个DynamicXml.Parse(...)方法调用，它本质上是DynamicXml的一个工厂方法调用。DynamicXml是一个自定义类型，而不是CLR Framework内建的一个类型。然而，DynamicXml没有实现FirstName或LastName属性。如若实现这两个属性，就会破坏在执行时从XML文件中获取数据的动态支持（我们的目的不是访问XML元素在编译时的实现）。换言之，DynamicXml不用反射来访问它的成员，而是根据XML内容动态绑定到值。

定义自定义动态类型的关键是实现System.Dynamic.IDynamicMetaObjectProvider接口。但是，不必从头实现接口。相反，首选的方案是从System.Dynamic.DynamicObject派生出自定义的动态类型。这样会为众多成员提供默认的实现，你只需重写那些不合适的。代码清单17-32展示了完整的实现。

代码清单17-32 实现自定义动态对象

```
using System;
using System.Dynamic;
using System.Xml.Linq;

public class DynamicXml : DynamicObject
{
  private XElement Element { get; set; }

  public DynamicXml(System.Xml.Linq.XElement element)
  {
      Element = element;
  }

  public static DynamicXml Parse(string text)
  {
      return new DynamicXml(XElement.Parse(text));
  }

  public override bool TryGetMember(
      GetMemberBinder binder, out object result)
  {
      bool success = false;
      result = null;
      XElement firstDescendant =
          Element.Descendants(binder.Name).FirstOrDefault();
      if (firstDescendant != null)
      {
          if (firstDescendant.Descendants().Count() > 0)
          {
              result = new DynamicXml(firstDescendant);
          }
          else
```

```
            {
                result = firstDescendant.Value;
            }
            success = true;
        }
        return success;
    }

    public override bool TrySetMember(
        SetMemberBinder binder, object value)
    {
        bool success = false;
        XElement firstDescendant =
            Element.Descendants(binder.Name).FirstOrDefault();
        if (firstDescendant != null)
        {
            if (value.GetType() == typeof(XElement))
            {
                firstDescendant.ReplaceWith(value);
            }
            else
            {
                firstDescendant.Value = value.ToString();
            }
            success = true;
        }
        return success;
    }
}
```

在这种情况下，要实现的核心动态方法是TryGetMember()和TrySetMember()（假定你还想对元素进行赋值）。为了支持调用动态取值和赋值属性，只需这两个方法实现就可以了。除此之外，它们的实现是相当简单的。首先，它们检查包含的XElement，查找与binder.Name（要调用的成员的名称）同名的一个元素。如果存在一个对应的XML元素，就取回（或设置）值。如果元素存在，返回值设为true，否则设为false。如果返回值为false，会自动导致“运行时”在进行动态成员调用的调用点处抛出一个Microsoft.CSharp.RuntimeBinder.RuntimeBinder Exception。

如果需要其他动态调用，可以利用System.Dynamic.DynamicObject支持的其他虚方法。代码清单17-33展示了所有可重写的成员。

代码清单17-33 System.Dynamic.DynamicObject的可重写成员

```
using System.Dynamic;

public class DynamicObject : IDynamicMetaObjectProvider
{
  protected DynamicObject();

  public virtual IEnumerable<string> GetDynamicMemberNames();
  public virtual DynamicMetaObject GetMetaObject(
      Expression parameter);
  public virtual bool TryBinaryOperation(
      BinaryOperationBinder binder, object arg,
          out object result);
  public virtual bool TryConvert(
```

```
    ConvertBinder binder, out object result);
public virtual bool TryCreateInstance(
    CreateInstanceBinder binder, object[] args,
          out object result);
public virtual bool TryDeleteIndex(
    DeleteIndexBinder binder, object[] indexes);
public virtual bool TryDeleteMember(
    DeleteMemberBinder binder);
public virtual bool TryGetIndex(
    GetIndexBinder binder, object[] indexes,
          out object result);
public virtual bool TryGetMember(
    GetMemberBinder binder, out object result);
public virtual bool TryInvoke(
    InvokeBinder binder, object[] args, out object result);
public virtual bool TryInvokeMember(
    InvokeMemberBinder binder, object[] args,
          out object result);
public virtual bool TrySetIndex(
    SetIndexBinder binder, object[] indexes, object value);
public virtual bool TrySetMember(
    SetMemberBinder binder, object value);
public virtual bool TryUnaryOperation(
    UnaryOperationBinder binder, out object result);
}
```

如代码清单17-33所示，几乎一切都有相应的成员实现——从转型和各种运算，一直到索引调用。除此之外，还有一个方法用于获取所有可能的成员名，这方法就是GetDynamicMemberNames()。

17.4 小结

本章讨论了如何利用反射来读取已编译成CIL的元数据。利用反射，你可以执行所谓的晚期绑定，也就是在执行时而不是编译时定义要调用的代码。虽然完全可以利用反射来部署一个动态系统，但相较于静态链接的（在编译时链接）、定义好的代码，它的速度要慢得多。因此，它更适合在开发工具中使用。

利用反射，还可以获取以特性的形式对各种构造进行修饰的附加元数据。通常，自定义特性是使用反射来查找的。你可以定义自己的特性，将自选的附加元数据插入CIL中。在运行时，可以获取这些元数据，并在编程逻辑中使用它们。

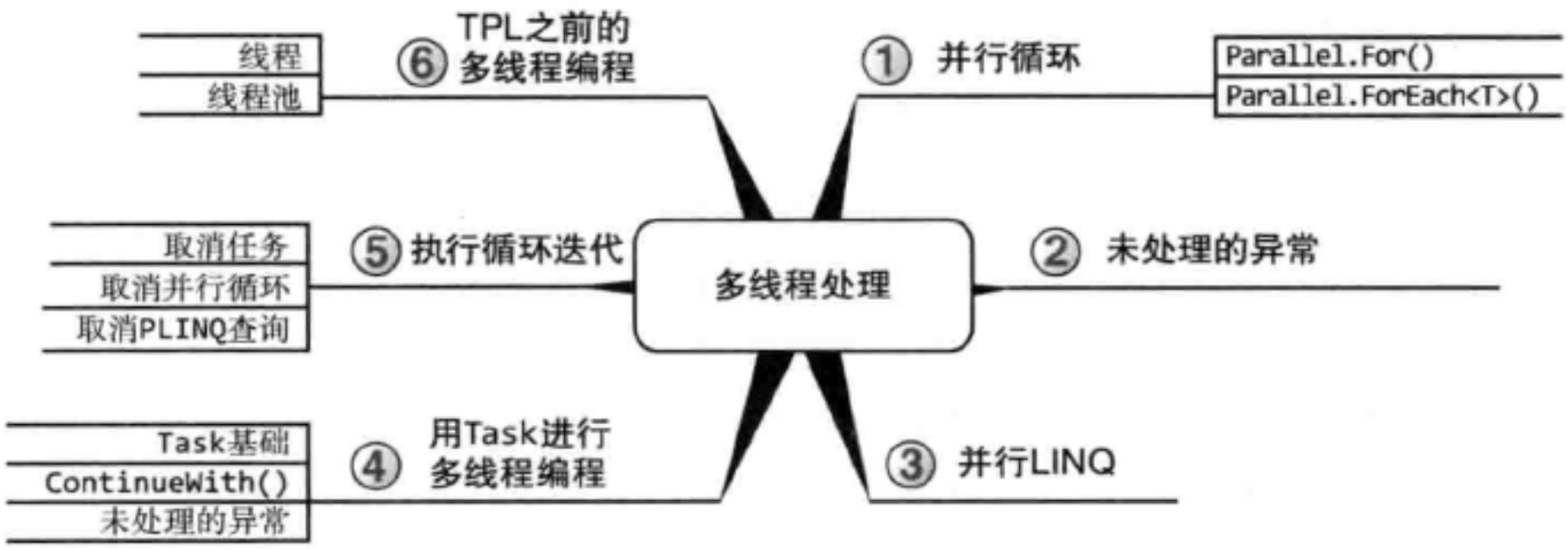
许多人都认为，正是特性的出现，才使“面向方面编程”（aspect-oriented programming）的概念变得清晰起来。在这种编程模型中，我们使用像特性这样的构造来添加功能，而不是在需要的时候手动实现一种功能。要在C#中看到真正的“方面”，尚需假以时日。然而，特性为我们指明了一个清晰的方向，朝这个方向前进，可以在不损害语言稳定性的同时，享受各种各样的新功能。

本章最后讲解了C# 4.0开始引入的功能——使用新的dynamic类型进行动态编程。这一节讨论了静态绑定在处理动态数据时存在的局限（不过，在API是强类型的前提下，静态绑定仍是首选）。

下一章将讨论多线程处理。在多线程处理中，我们将特性用于线程的同步。

第18章 多线程处理

过去十年有两个重要的趋势对软件开发产生了巨大影响。第一，不再通过时钟速度和晶体管密度来降低计算成本（如图18.1所示）。相反，现在是通过制造包含多个CPU的硬件来降低成本。



其次，现在的计算会遇到各种延迟。简单地说，延迟是获得结果所需的时间。延迟主要有两个原因。处理复杂的计算任务时产生处理器受限延迟（Processor-bound latency）；假定一个计算需要执行120亿次算术运算，而总共的处理能力是每秒6亿次，那么从请求结果到获得结果至少有2秒钟的处理器受限延迟。相反，I/O受限延迟（I/O-bound latency）是从外部来源（如磁盘驱动器、Web服务器等）获取数据所产生的延迟。任何计算要从远程Web服务器获取数据，至少都会产生相当于几百万个处理器周期的延迟。

这两种趋势为软件开发人员带来了巨大的挑战。既然现在计算能力大增，如何在有效利用它们快速获得结果的同时不损害用户体验？如何防止创建不科学的UI，在执行高延迟的操作时发生卡顿或冻结？此外，如何在多个处理器之间分配CPU受限的工作来缩短计算时间？

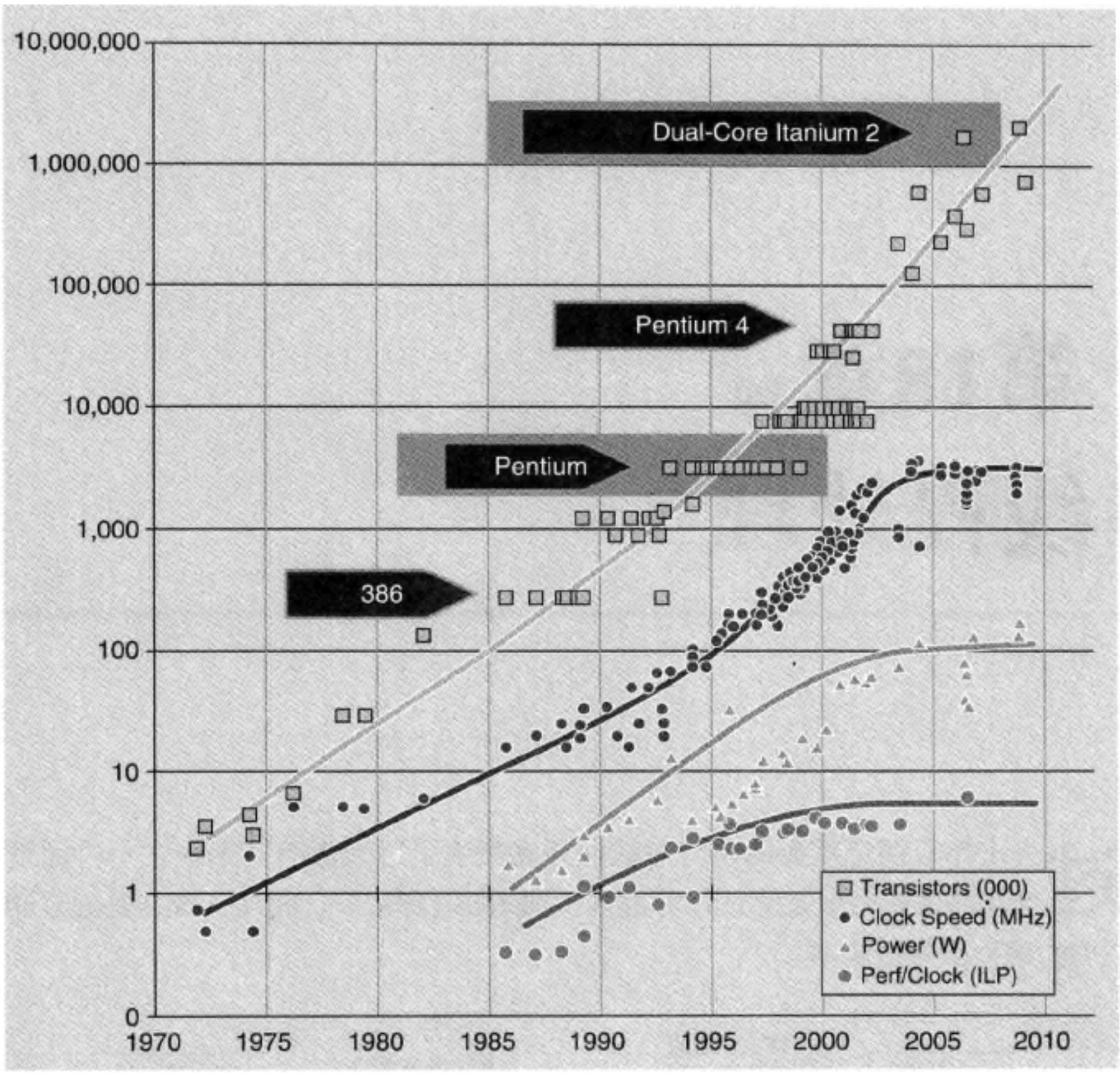


图18-1 时钟速度发展历史（本图由Herb Sutter制作。使用已获授权。来自www.gotw.ca）

为了保证UI响应迅速，同时高效利用CPU，标准技术是编写**多线程**程序，“并行”执行多个计算。遗憾的是，多线程逻辑很难写好，我们将用两章的篇幅讨论多线程处理的困难性，以及如何使用高级抽象和新的语言功能来减轻负担。

在要讨论的高级抽象中，首先是随同.NET 4.0发布的并行扩展库的两个基本组件[①]：TPL（Task Parallel Library，任务并行库）和PLINQ（Parallel LINQ，并行LINQ）。其次是TAP（Task-based Asynchronous Pattern，基于任务的异步模式）以及配套的C# 5.0语言支持。虽然我们强烈建议使用这些高级抽象，但在本章和附录C、D中，仍会花一些篇幅讲解老版本.NET“运行时”的低级线程处理API。除此之外，可以从本书的配套网站下载本书前几版（《C#本质论（第2版）》和《C#本质论（第3版）》）的多线程处理章节，网址是http://IntelliTect.com/EssentialCSharp。老的内容在今天仍有意义，仍然重要，因为许多人都享受不到只为最新版本的.NET Framework写代码的“奢侈”仍然要兼顾理解之前的代码。

本章首先是面向多线程编程的新手提供的一系列初学者主题。然后简单讨论了如何在不用并行扩展库的前提下进行“传统”的线程处理，从而对线程处理有一个基本理解（下一章会更全面地讨论线程处理）。最后，本章的大部分篇幅都用来讲解TPL、TAP和PLINQ。

---

① 要在.NET 3.5中使用这些库，可以下载Reactive Extensions Library for.NET 3.5，但它没有得到官方支持。

18.1 多线程基础

初学者主题：多线程术语

多线程处理术语太多，容易混淆，所以我们先把它们定义好。

CPU（中央处理器）或者**内核**[①]是实际执行给定程序的硬件单元。每台机器至少有一个CPU；如今多CPU机器也不少见。许多现代CPU都支持**同时多线程**（Intel称为**超线程**），使一个CPU能表现为多个“虚拟”CPU。

进程是给定程序当前正在执行的实例；操作系统的一个基本功能就是管理进程。每个进程都包含一个或多个线程。一个进程由System.Diagnostics命名空间的Process类的一个实例表示。

C#编程在语句和表达式的级别上根本就是在描述**控制流**。本书到目前为止一直假设程序仅有一个“控制点”。可以想象在启动程序之后，控制点像“光标”（cursor）一样进入Main方法，然后随着各种条件、循环、方法等的执行在程序中移动。**线程**就是这个控制点。一个线程由System.Threading.Thread类的一个实例表示。Thread类和操纵Thread的API都在System.Threading命名空间中。

单线程程序是仅包含一个线程的进程。**多线程**程序的进程则包含两个或更多的线程。

在多线程程序中运行时具有正确的表现，就说代码是**线程安全**的。代码的**线程处理模型**是指代码向调用者提出的一系列要求，只有满足这些要求才能保障线程安全。例如，许多类的线程处理模型都是“可以从任意线程调用静态方法，但实例方法仅能从分配实例的线程调用”。

任务是可能有高延迟的工作单元，目的是生成一个结果值，或者产生想要的效果。任务和线程的区别是：一个任务代表需要执行的一个作业，而一个线程代表做这个作业的工作者。一个任务由Task类的一个实例表示。生成给定类型的值的任务由Task<T>类表示，后者从非泛型Task类型派生。它们都在System.Threading.Tasks命名空间中。

线程池是多个线程的集合，也是决定如何向线程分配工作的逻辑。有任务要执行时，它分配池中的一个工作者线程执行任务，并在任务结束后解除分配。

初学者主题：多线程处理的目的和方式

多线程处理主要用于两个方面：实现多任务和解决延迟。

用户随随便便就会同时运行几十乃至上百个进程。他们可能一边编辑演示文稿和电子表格，一边在网上浏览、听音乐、接收即时消息和电子邮件通知，还不时看一看角落里的小时钟。每个进程都在干活——谁都不是机器唯一关注的任务。这种多任务处理通常是在进程级实现的，但有时也要在一个进程中进行这样的多任务处理。

考虑到本书的目的，我们主要用多线程技术解决延迟问题。例如，为了在导入大文件时允许用户点击“取消”，开发者可以创建一个额外的线程来执行导入。这样，用户随时都能点击“取消”，而不是冻结UI直至导入完成。

如果有足够的内核使得每一个线程都能分配到一个内核，那么每一个线程就都使

---

① 从技术上说，“CPU”总是指物理芯片，而“内核”可以指物理CPU，也可以指虚拟CPU。在本书中两者的区别并不重要，两个词都可以使用。

用它们各自的CPU。然而，即使今天已经有更多的多核机器，线程数仍然大于内核的数目，每个进程要执行多个线程。

为了解决粥（CPU内核）少僧（线程）多的矛盾，操作系统通过称为**时间分片**（time slicing）的机制来模拟多个线程并发运行。操作系统以极快的速度从一个线程切换到另一个线程，给人的感觉就是所有线程都在同时执行。处理器在切换到下一个线程之前，执行一个特定线程的时间周期称为**时间片**（time slice）或**量子**（quantum）。在一个给定的内核中改换执行线程的动作称为**上下文切换**。

这个技术的效果和光纤电话线相似。光纤线相当于处理器，每个通话都相当于一个线程。一条（单模式）光纤电话线每次只能发送一个信号，但同时可以有许多人通过同一条电话线打电话。由于光纤信道的速度极快，所以能在不同的通话之间快速切换，使人感觉不到自己的通话有中断的现象。类似地，一个多线程进程所包含的每个线程都能“不中断”地并行运行。

无论是真正的多核并行运行，还是使用时间分片技术来模拟，我们说“一起”进行的两个操作是**并发**的。为了实现这种并发的操作，需要以异步方式调用它，被调用操作的开始与结束都独立于调用它的控制流。异步分配的工作与当前控制流并行执行，就实现了并发性。**并行编程**是指将一个问题分解成较小的部分，并**异步**地发起对每个部分的处理，使它们能并发地得到处理。

初学者主题：性能问题

执行I/O受限操作的线程平时会被操作系统忽略，直至I/O子系统返回结果。所以，在等待时切换到一个不同的线程能提高执行效率，避免处理器无意义地闲置。

然而，上下文切换是有代价的。必须将CPU当前的内部状态保存到内存，还必须加载与新线程关联的状态。如果线程太多，切换开销就会开始显著影响性能。添加更多的线程会进一步降低性能。处理器的大量时间被花在从一个线程切换到另一个线程上，而不是主要花在线程的执行上。

即使忽略上下文切换的开销，时间分片本身对性能也有巨大的影响。例如，假定有两个处理器受限的高延迟任务，分别计算10亿个数字的平均值。假定处理器每秒能执行10亿次运算。如果两个任务分别和一个线程关联，而且两个线程分别有自己的内核，那么显然能在1秒钟内获得两个结果。

但是，如果一个处理器由两个线程共享，时间分片将在一个线程上执行几十万次操作，再切换到另一个线程，再切换回来，以此类推。每个任务都要消耗1秒钟的处理器时间，所以两个结果要在2秒钟之后才能获得，造成平均完成时间是2秒。（同样地，我们在这里忽略了上下文切换的开销）。

如果分配两个任务都由一个线程执行，而且严格按前后顺序执行，则第一个任务的结果在1秒后获得，第二个在第2秒后获得，造成平均完成时间是1.5秒。

规范

不要以为多线程总是会使代码更快。

要在通过多线程来更快解决处理器受限问题时，谨慎地衡量性能。

初学者主题：线程处理的问题

前面说过，多线程的程序写起来既复杂又困难，但未曾提及原因。其实根本原因在于单线程程序中一些成立的假设在多线程程序中变得不成立了。问题包括缺乏原子性、竞态条件、复杂的内存模型以及死锁。

1. 大多数操作都不是原子性的

原子操作要么尚未开始，要么已经完成。从外部看，其状态永远不会是“进行中”。例如以下代码：

```
if (bankAccounts.Checking.Balance >= 1000.00m)
{
  bankAccounts.Checking.Balance -= 1000.00m;
  bankAccounts.Savings.Balance += 1000.00m;
}
```

上述代码检查银行账户余额，如符合条件就从一个账户中取钱，向另一个账户存钱。这个操作必须是原子性的。换言之，为了使代码能正确执行，永远不能发生操作只是部分完成的情况。例如，假定两个线程同时运行，可能两个都验证账户有足够的余额，所以两个都执行转账，而剩余的资金其实只够进行一次转账。事实上，局面会变得更糟。在上述代码中，没有任何一个操作是原子性的。就连复合加/减（或者读/写目标类型的属性）在C#中都不属于原子操作。因此，它们在多线程环境中全都属于“部分完成”——部分增大或减小。因为部分完成的非原子操作而造成的不一致状态是**竞态条件**的一种特例。

2. 竞态条件所造成的不确定性

如前所述，并发性一般通过时间分片来模拟。在缺少下一章要详细讨论的特殊控制流语句的情况下，操作系统会在它认为合适的任何时间，在任何两个线程之间切换上下文。这意味着当两个线程访问同一个对象时，无法预测哪个线程“竞争胜出”并抢先运行。例如，假定有两个线程运行上述代码段，可能一个胜出并一路运行到尾，第二个线程甚至还没有开始。也可能在第一个执行余额检查时发生上下文切换，第二个胜出，一路运行到尾。

对于包含竞态条件的代码，它的行为取决于上下文切换时机。这造成了程序执行的不确定性。一个线程中的指令相对于另一个线程中的指令，两者的执行顺序是未知的。最糟的情况是包含竞态条件的代码99.9%的时间都具有正确行为 。1000次中只有那么1次，另一个线程在竞争中胜出。正是这种不确定性使多线程编程显得很难。

由于竞态条件难以复制，所以多线程代码的品质保证主要基于长期的压力测试、专门的代码分析工具以及专家对代码进行的大量分析和检查。

下一章会介绍一些用于应对竞态条件的技术。

3. 内存模型的复杂性

竞态条件（两个控制点以无法预测而且不一致的速度“竞争”代码的执行）本来就已经很糟了，但还有更糟的。假定两个线程在两个不同的进程中运行，但都要访问同一个对象中的字段。现代处理器不会在每次要用一个变量的时候都去访问主内存。相反，是在处理器的“高速缓存”中生成一个本地副本。这个缓存会定时与主内存同步。这意味着在两个不同的处理器上，两个线程以为自己在读写相同的位置，其实看到的不是对方对那个位置的实时更新。两者获得的结果不一致。简单地说，这里是因为处理器同步缓存的时机而产生了竞态条件。

4. 锁定造成死锁

显然，肯定有什么机制能将非原子的操作转变成原子操作，要求操作系统对线程进行调度以防止竞态条件，并确保处理器的高速缓存在必要时进行同步。C#程序解决所有这些问题的主要机制是lock语句。该语句允许开发者将一部分代码设为“关键”代码，一次只有一个线程能执行它。如果多个线程试图进入临界区，操作系统只允许一个进入，其他将被挂起。操作系统还确保在遇到锁的时候处理器高速缓存能正确同步。

然而，锁自身也有问题。最容易想到的是，假如不同线程以不同的顺序来获取锁，就可能发生死锁（deadlock）。这个时候，线程会被冻结，彼此等待对方释放它们的锁。

例如：

时间 ↓

线程A	线程B
获得a上的锁	获得b上的锁
请求b上的锁	请求a上的锁
死锁，等待b	死锁，等待a

此时，每个线程都只有在对方释放了锁之后才能继续运行，线程阻塞，造成该代码执行彻底死锁。

我们将在第19章讨论各种锁定技术。

规范

不要无根据地以为普通代码中原子性的操作在多线程代码中也是原子性的。
不要以为所有线程看到的都是一致的共享内存。
要确保同时拥有多个锁的代码总是以相同的顺序获取它们。
避免所有竞态条件，即程序行为不能受操作系统调度线程的方式的影响。

18.2 使用System.Threading

并行扩展库相当有用，因为它允许使用更高级的抽象——任务，而不必直接和线程打交道。但有的时候，要处理的代码是在TPL和PLINQ问世之前（.NET 4.0之前）写的。也有可能某个编程问题不能直接用它们解决。本节简单地讨论一下用于直接操纵线程的API。

18.2.1 使用System.Threading.Thread进行异步操作

操作系统实现线程并提供各种非托管API来创建和管理线程。CLR封装这些非托管线程，在托管代码中通过System.Threading.Thread类来公开它们。该类的实例代表程序中的一个“控制点”。如前所述，可以将线程想象成一名“工作者”，它独立地按照你的程序指令工作。

代码清单18-1提供了一个例子。独立的控制点由同时运行的一个Thread实例表示。线程需要知道启动时运行什么代码，所以它的构造器获取对要执行代码的一个委托。本例是将方法组DoWork转换成恰当的委托类型ThreadStart。然后调用Start()启动线程。新线程运行时，主线程在控制台上打印1000个连字符。 然后调用Join()告诉主线程等候工作者线程完成。结果如输出18-1所示。

代码清单18-1 使用System.Threading.Thread启动一个方法

```
using System;
using System.Threading;

public class RunningASeparateThread
{
  public const int Repetitions = 1000;

  public static void Main()
  {
      ThreadStart threadStart = DoWork;
      Thread thread = new Thread(threadStart);
      thread.Start();
      for(int count = 0; count < Repetitions; count++)
      {
          Console.Write('-');
      }
      thread.Join();
  }

  public static void DoWork()
  {
      for(int count = 0; count < Repetitions; count++)
      {
          Console.Write('+');
      }
  }
}
```

输出18-1

```
++++++++++++++++++++++++++++++++----------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++----------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++----------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++
```

如输出所示，线程是在轮流执行，分别打印几百个字符，然后发生上下文切换。两个循环“并行”运行，而不是第一个运行完了第二个才开始（委托若同步而不是异步执行便会如此）。

代码为了在不同线程的上下文中运行，需要ThreadStart或者ParameterizedThreadStart类型的一个委托来标识要执行的代码（后者允许单个object类型的参数）。给定用ThreadStart委托构造器创建的一个Thread实例，可调用thread.Start来启动线程。（代码清单18-1显式创建ThreadStart类型的一个变量，目的是在源代码中显示委托类型。但方法组DoWork实际可以直接传给线程构造器。）调用Thread.Start()是告诉操作系统开始并发地执行一个新线程；然后主线程中的控制立即返回，开始执行Main()方法中的for循环。两个线程现在独立运行，不会等待对方，直到调用Join()。

18.2.2 线程管理

线程包含大量方法和属性，用于管理它们的执行。下是一些基本的。

- Join()。如代码清单18-1所示，可以调用Join()使一个线程等待另一个线程。它告诉操作系统暂停执行当前线程，直到另一个线程终止。Join()方法的重载版本允许获取一个int或者一个TimeSpan作为参数，指定最多等待thread执行多长时间，过期不候。
- IsBackGround。新线程默认为“前台”线程；操作系统将在进程的所有前台线程完成后终止进程。可将thread.IsBackGround属性设为true，从而将线程标记为“后台”线程。这样，即使后台线程仍在运行，操作系统也允许进程终止。不过，最好还是不要半路中止（abort）[①]任何线程，而是在进程退出之前显式退出每个线程。更多细节可参考稍后关于线程中止的小节。
- Priority。每个线程都关联了优先级，可将Priority属性设为新的ThreadPriority枚举值（Lowest、BelowNormal、Normal、AboveNormal或Highest），从而增大或减小线程的优先级。操作系统倾向于将时间片调拨给高优先级线程。但要注意，如果优先级设置不当，可能会出现“饥饿”情况，即一个高优先级线程快乐地运行，而其他许多低优先级线程只能眼睁睁看着它。
- ThreadState。如果只是想知道一个线程是还“活着”，还是已完成了所有工作，可以使用布尔属性IsAlive。更全面的线程状态可通过ThreadState属性访问。ThreadState枚举值包括Aborted、AbortRequested、Background、Running、Stopped、StopRequested、Suspended、SuspendRequested、Unstarted和WaitSleepJoin。这些都是标志（flag），有的可以组合。

有两个常用（而且经常被滥用）的方法是Sleep()和Abort()，值得用专门的小节讨论。

18.2.3 在生产代码中不要让线程进入睡眠

静态方法Thread.Sleep()使当前线程进入睡眠——其实就是告诉操作系统在指定时间内不要为该线程调度任何时间片。参数（毫秒数或者TimeSpan）指定了操作系统要等待它多长时间。等待期间，操作系统当然可以将时间片调度给其他线程。这个设计表面合理，

---

① abort在英语里面很好理解，有一种被野蛮终止的感觉。但中文里面中断、中止、终止、取消等各种翻译。都有。MSDN文档将abort翻译成“中止”。——译者注

实则不然，值得好好推敲一下。

线程睡眠的目的通常是和其他线程就某个事件进行同步。但是，操作系统不保证计时的精确度。也就是说，如果指定："睡眠123毫秒"，操作系统至少让线程睡眠123毫秒，但这个时间可能更长。线程从进入睡眠到它被唤醒，所经历的实际时间无法确定，可能为任意长度。不要将Thread.Sleep()作为高精度计时器使用，因为它不是。

更糟的是，Thread.Sleep()经常作为"穷人的同步系统"使用。也就是说，如果有一些异步工作要做，而当前线程必须等这个工作完成才能继续，你可能试图让线程睡眠，寄希望于当前线程醒来后异步工作已完成。但这是一个糟糕的想法，异步工作花费的时间可能超出你的想象。下一章要描述正确的线程同步机制。（代码清单18-2会给出一个例子。）

线程睡眠是不好的编程实践，因为睡眠的线程是不运行代码的。如果Windows应用程序的主线程睡眠，就不再处理来自UI的消息，感觉就像挂起了一样。另外，之所以分配像线程这样的昂贵资源，肯定是想物尽其用。没人花钱聘请员工来睡觉，所以不要分配昂贵的线程，目的只是让它睡眠大量的处理器周期。

不过，Thread.Sleep()还是有一些用处的。首先，将线程睡眠时间设为零，相当于告诉操作系统："当前线程剩下的时间片就送给其他线程了。"然后，该线程会被正常调度，不会发生更多的延迟。其次，测试代码经常用Thread.Sleep()模拟高延迟操作，同时不必让处理器真的去做一些无意义的运算。除了这些，在生产代码中使用该方法都应仔细地检查，研究是否有其他替代方案。

在C# 5基于任务的异步编程中，可以为Task.Delay()方法的结果使用await操作符，在不阻塞当前线程的前提下引入异步延迟。详情参见下一章关于计时器的小节。

规范

避免在生产代码中调用Thread.Sleep()。

18.2.4 在生产代码中不要中止线程

Thread对象的Abort()方法一旦执行，就会尝试销毁线程。它造成"运行时"在线程中引发ThreadAbortException异常。该异常可被捕获，但即使被捕获并被忽略，都会自动重新引发以确保线程事实上被销毁。不要中止线程的原因是多方面的，下面罗列了一部分。

- 方法只是承诺尝试中止线程；不保证成功。例如，如果线程的控制点当前在finally块中，"运行时"不会试图引发ThreadAbortException（因为当前可能在运行关键的清理代码，不应被打断）。在非托管代码中也不会引发，否则会破坏CLR本身。相反，CLR会推迟到控制离开finally块或者回到托管代码之后才引发异常。但这也是不保证的，被中止的线程可能在finally块中包含无限循环。（讽刺的是，恐怕正是因为线程中有无限循环才想中止。）
- 被中止的线程可能正在执行由lock语句保护的关键代码（详情参见下一章）。和finally不同，lock阻止不了异常。关键代码会因为异常而中断，lock对象会自动释放，允许正在等待这个锁的其他代码进入关键区域，并查看执行到一半的代码的状

态。锁的原意就是要防止这种情况的发生。所以，中止线程会造成貌似线程安全的代码，它实际是危险和不正确的。

- 线程中止时，CLR保证自己的内部数据结构不会损坏，但BCL没有做出这个保证。在错误的时间引发异常，中止线程可能使你的数据结构或者BCL的数据结构处于损坏状态。在其他线程或者中止线程的finally块中运行的代码可能看到损坏的状态，要么崩溃，要么行为错误。

简单地说，开发人员应将Abort()作为不得已而为之的“终极招术”。理想情况是，在整个AppDomain或者整个进程被摧毁的时候，才将中止线程作为一组更全面的紧急关闭策略的一部分。幸好，基于任务的异步机制采用了更健壮、更安全的协作式取消模式来终止结果已经不再需要的“线程”。详情参见18.3节。

规范

避免在生产代码中中止线程，因为可能发生不可预测的结果，使程序不稳定。

18.2.5 线程池处理

之前的“初学者主题：性能问题”说过，线程太多会对性能造成负面影响。线程是昂贵的资源，线程上下文切换不是免费的，而且通过时间分片来模拟两个作业“并行”运行，可能比一个接一个运行还要慢。

为了缓解这种状况，BCL提供了线程池。开发人员不是直接分配线程，而是告诉线程池想要完成什么工作。工作结束后，线程不是中断并被销毁，而是回到池中，从而节省了当更多的工作来临时还要分配新线程的开销。代码清单18-2做的是和代码清单18-1一样的事情，但使用了线程池中的线程。

代码清单18-2 使用ThreadPool而不是显式实例化线程

```
using System;
using System.Threading;

public class Program
{
  public const int Repetitions = 1000;
  public static void Main()
  {
      ThreadPool.QueueUserWorkItem(DoWork, '+');
      for(int count = 0; count < Repetitions; count++)
      {
          Console.Write('-');
      }

      // Pause until the thread completes
      // This is for illustrative purposes; do not
      // use Thread.Sleep for synchronization in
      // production code.
      Thread.Sleep(1000);
  }
  public static void DoWork(object state)
  {
```

```
            for(int count = 0; count < Repetitions; count++)
            {
                Console.Write(state);
            }
        }
    }
```

程序结果和输出18-1相似，都是 “+” 和 “-” 字符混合。如果多个不同的作业异步执行，线程池能在单处理器和多处理器计算机上获得更好的执行效率。效率是通过重用线程（而不是每个异步调用都重新构造线程）来获得的。遗憾的是，线程池并非没有缺点，有一些性能和同步问题需要考虑。

为了有效利用处理器，线程池假定你在线程池上调度的所有工作都能及时结束，使线程能回到池中并为其他任务所用。线程池还假定所有工作的运行时间都较短（耗费以毫秒或秒计的处理器时间，而非以小时或天计）。基于这些假设，就可保证每个处理器都全力以赴完成任务，而不是无效率地通过时间分片来进行多个任务。线程池通过确保线程的创建“刚刚好”，没有一个处理器因为运行太多线程而超出负荷，从而防止过度的时间分片。但这也意味着一旦用完池中的所有线程，正在排队的工作就只好延迟执行了。如果池中的所有线程都被长时间运行或者I/O受限的工作占用了，正在排队的工作必然会被延迟。

有别于能够直接操纵的Thread与Task对象，线程池不提供对正在执行给定工作的线程的引用。这就阻止了发出调用的线程通过前面描述的线程管理功能与工作者线程同步（或者控制工作者线程）。代码清单18-2使用了前面描述的“穷人的同步”；这在生产代码中是不可取的，因为不知道工作要花多长时间完成。

简单地说，线程池能很好地完成作业，但该作业中不包括处理长时间运行的作业，或者处理需要与其他线程（包括主线程）同步的作业。开发人员真正要做的是构建高级抽象，将线程和线程池作为实现细节使用。这种抽象由任务并行库（Task Parallel Library，TPL）实现，本章剩余的大部分内容都会围绕这个主题展开。

要了解.NET 4之前如何管理工作者线程，请参见附录D。

规范

要用线程池向处理器受限任务高效地分配处理器时间。

避免将池中的工作者线程分配给I/O受限或者长时间运行的任务，而是改为使用TPL。

18.3 异步任务

多线程编程的复杂性主要反映在以下几个方面。

（1）监视异步操作的状态，知道它在何时完成。为了判断一个异步操作于何时完成，最好不要采取轮询线程状态的办法，也不要采取阻塞并等待的办法。

（2）线程池。线程池避免了启动和终止线程的巨大开销。此外，线程池避免了创建太多的线程，防止系统将大多数时间花在线程的切换上而不是线程的运行上。

（3）避免死锁：在避免死锁的同时，防止数据同时被两个不同的线程访问。

（4）为不同的操作提供原子性并同步数据访问。为不同的操作组（指令序列）提供同步，可以确保将一系列操作作为一个整体来执行，并可由另一个线程恰当地中断。锁定机制防止两个不同的线程同时访问数据。

此外，任何时候只要有方法需要长时间运行，就很可能需要多线程编程，即异步调用该方法。随着编写的多线程代码越来越多，开发人员总结出了一系列常见的情形以及对这些情形进行处理的编程模式。

C# 5.0利用来自.NET 4.0的TPL，并通过新增语言构造的方式，对其中一个称为TAP的模式进行了增强，改进了它的可编程性。本节和下一节将详细讨论TPL，然后讨论如何利用上下文关键字async和await简化TAP编程。第19章后半部分将讨论其他多线程编程模式。如果不允许使用TPL和C# 5.0，或者要求使用不是基于TPL的API，这些编程模式就很重要了。

18.3.1 从Thread到Task

创建线程代价高昂，而且每个线程都要占用大量虚拟内存（默认1 MB）。前面说过，更有效的做法是使用线程池，需要时分配线程，为线程分配异步工作，运行直至结束，再为后续的异步工作重用线程，而不是在工作结束后销毁再重新创建线程。

在.NET Framework 4中，TPL不是每次开始异步工作时都创建一个线程，而是创建一个Task，并告诉**任务调度器**有异步工作要执行。此时任务调度器可能采取多种策略，但默认是从**线程池**请求一个工作者线程。线程池会自行判断怎么做最高效。可能在当前任务结束后再运行新任务，也可能将新任务的工作者线程调度到特定处理器上。线程池还会判断是创建一个全新线程，还是重用一个之前已结束运行的现有线程。

通过将异步工作的概念抽象到Task对象中，TPL提供了一个能代表异步工作的对象，还提供了相应的API以便与工作进行交互。通过提供代表工作单元的对象，TPL使我们能通过编程将小任务合并成大任务，从而建立起一个工作流程，详情请参见稍后的讨论。

任务是对象，其中封装了以异步方式执行的工作。这听起来有点儿耳熟，委托不也是封装了代码的对象吗？区别在于，委托是同步的，而任务是异步的。如果执行一个委托（例如一个Action），当前线程的控制点会立即转移到委托的代码；除非委托结束，否则控制不会返回调用者。相反，启动一个任务，控制几乎立即返回调用者，无论任务要执行多少工作。任务在另一个线程上异步执行（本章稍后会讲到，可以只用一个线程来异步执行任务，而且这样做还有一些好处）。简单地说，任务将委托从同步执行模式转变成异步。

18.3.2 理解异步任务

之所以知道在当前线程上执行的委托于何时完成，是由于除非委托完成，否则调用者什么也做不了。那么，怎样知道任务于何时完成，怎样获得结果呢（如果有的话）？下面来看一个将同步委托转换成异步任务的例子。它做的是和代码清单18-1（用线程）与代码清单18-2（用线程池）相同的事情，但这一次用的是任务。工作者线程向控制台写入加号，主线程写入连字号。

启动任务将从线程池获取一个新线程，创建第二个“控制点”，并在那个线程上执行委托。主线程上的控制点和平常一样，从启动任务的调用（Task.Run()）之后继续。代码清单18-3的结果总和输出18-1差不多。

代码清单18-3 调用异步任务

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

public class Program
{
  public static void Main()
  {
      const int Repetitions = 10000;
      // Use Task.Factory.StartNew<string>() for
      // TPL prior to .NET 4.5
      Task task = Task.Run(() =>
          {
              for(int count = 0;
                  count < Repetitions; count++)
              {
                  Console.Write('-');
              }
          });
      for(int count = 0; count < Repetitions; count++)
      {
          Console.Write('+');
      }

      // Wait until the Task completes
      task.Wait();
  }
}
```

新线程要运行的代码由传给Task.Run()方法的委托（本例是Action类型）来定义。这个委托（以Lambda表达式的形式）在控制台上反复打印虚线。主线程的循环几乎完全一样，只是它打印句点。

注意，调用Task.Run()之后，作为参数传递的Action几乎立即开始执行。这称为“热”任务，即已触发并开始执行。“冷”任务则相反，它需要在显式触发之后才开始异步工作。

虽然一个Task可通过Task构造器实例化成“冷”状态，但这一般只应作为返回“热”任务的API的内部实现细节。

注意，在调用Run()之后，“热”任务的确切状态是不确定的。相反，这个状态由操作系统来决定，操作系统选择是立即运行任务的工作者线程，还是推迟到有更多资源的时候。事实上，轮到调用线程再次执行时，说不定“热”任务已经完成了。调用Wait()将强迫主线程等待分配给任务的所有工作结束执行。这相当于代码清单18-1在工作者线程上调用Join()。

目前的例子只有一个任务，但理所当然可能有多个任务以异步方式运行。一个常见的情况是要等待一组任务完成，或者等待其中一个任务完成，当前线程才能继续。这是用Task.WaitAll()和Task.WaitAny()方法来实现的。

前面描述了任务如何获取一个Action并以异步方式运行它。如果任务中执行的工作要返回结果又该怎么办呢？可以使用Task<T>类型来异步运行一个Func<T>。以同步方式执行委托时，除非获得结果，否则控制不会返回。异步执行Task<T>时，可以从一个线程中轮询

它，看它是否完成，若完成就获取结果[①]。代码清单18-4 演示了如何在控制台应用程序中做这件事情。注意这个例子用到了PiCalculator.Calculate()方法，该方法的详情将在后面的“并行执行循环迭代”小节中讲述。

代码清单18-4 轮询一个Task<T>

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

public class Program
{
  public static void Main()
  {
      // Use Task.Factory.StartNew<string>() for
      // TPL prior to .NET 4.5
      Task<string> task =
          Task.Run<string>(
              () => PiCalculator.Calculate(100));

      foreach(
          char busySymbol in Utility.BusySymbols())
      {
          if(task.IsCompleted)
          {
              Console.Write('\b');
              break;
          }
          Console.Write(busySymbol);
      }

      Console.WriteLine();

      Console.WriteLine(task.Result);
      System.Diagnostics.Trace.Assert(
          task.IsCompleted);
  }
}
```

```
public class PiCalculator
{
  public static string Calculate(int digits = 100)
  {
      \\ ...
  }
}
```

```
public class Utility
{
  public static IEnumerable<char> BusySymbols()
  {
      string busySymbols = @"-\|/-\|/";
      int next = 0;
      while(true)
```

① 使用轮询技术要小心。如果像当前这个例子那样根据一个委托来创建任务，任务将被安排到线程池的一个工作者线程上运行。这意味着当前线程会一直循环，直到工作者线程上的工作结束为止。虽然这个技术可行，但可能无谓地消耗CPU资源。如果不将任务安排到工作者线程上执行，而是安排由当前线程在未来的某个时间执行，轮询技术变会变得非常危险。由于当前线程在一个对任务进行轮询的循环中，所以会一直循环下去，因为除非当前线程退出循环，否则任务是不会结束的。

```
        {
            yield return busySymbols[next];
            next = (next + 1) % busySymbols.Length;
            yield return '\b';
        }
    }
}
```

在这个代码清单中，任务的数据类型是Task<string>。泛型类型包含一个Result属性，可从中获取由Task<string>执行的Func<string>的返回值。

注意代码清单18-4没有调用Wait()。相反，读取Result属性自动造成当前线程等待，直到结果可用（如果结果还不可用的话）。在这个例子中，我们知道在获取结果时，结果肯定已经准备好了。

除了Task<T>的IsCompleted和Result属性，还有其他几件事情需要注意。

- 任务结束后，IsCompleted属性被设true——不管是正常结束，还是出错（即引发异常并终止）。更详细的任务状态信息可通过读取Status属性来获得，该属性返回TaskStatus类型的值，可能的值包括Created、WaitingForActivation、WaitingToRun、Running、WaitingForChildrenToComplete、RanToCompletion、Canceled和Faulted。任何时候只要Status为RanToCompletion、Canceled或者Faulted，IsCompleted就为true。当然，如果任务在另一个线程上运行，读取的状态值是“正在运行”，那么任何时候状态都可能变成“已完成”，包括刚好在读取了属性值之后。这一点对于其他许多状态都是成立的——就连Created都可能改变（如果另一个线程把它启动的话）。只有RanToCompletion、Canceled和Faulted可被视为最终状态，不会再改变。
- 任务可用Id属性的值来唯一性地标识。静态属性Task.CurrentId返回当前正在执行的Task（发出Task.CurrentId调用的那个任务）的标识符。这些属性对于调试特别有用。
- 可用AsyncState为任务关联额外的数据。例如，假定一个List<T>中的值要由多个任务完成。为此，每个任务都将值的索引包含到AsyncState属性中。这样一来，当任务结束后，代码可使用AsyncState（先转型成int）访问列表中的特定索引位置[①]。

还有一些有用的属性将在本章稍后关于任务取消的一节进行讨论。

18.3.3 任务延续

之前多次提到程序的“控制流”，但一直没有把控制流最根本的地方说出来：控制流决定了接着要发生什么。对于Console.WriteLine(x.ToString());这样的一个简单的控制流，它指出如果ToString正常结束，接着发生的事情就是调用WriteLine，将刚才的返回值作为实参传给它。“接着发生的事情”就是一个延续（continuation）。控制流中的每个控制点都有一个延续。在我们的例子中，ToString的延续是WriteLine（WriteLine的延续是下一个语句中运行的代码）。延续的概念对于C#编程来过于平常，大多数程序员根本没有想过它，只是在不知不觉中就用了。C#编程其实就是在延续的基础上构造延续，直到整个程序的控制流结束。

注意，在普通的C#程序中，给定代码的延续会在那个代码完成后立即执行。一旦

① 用任务异步修改集合要小心。任务可能在不同的工作者线程上运行，而集合可能不是线程安全的。更安全的做法是任务结束后在主线程填充集合。

ToString()返回，当前线程的控制点立即执行对WriteLine的同步调用。还要注意，任何给定的代码实际上都有两个可能的延续：“正常”延续和“异常”延续。如果当前代码引发异常，执行的就是后者。

异步方法调用（比如开始一个Task）会为控制流添加一个新的维度。执行异步Task调用，控制流立即转到Task.Start()之后的语句。与此同时，Task委托的主体也开始执行了。换言之，在涉及异步任务的时候，“接着发生的事情”是多维的。发生异常时的延续仅仅是一个不同的执行路径。相反，异步延续是多了一个并行的执行路径。

异步任务使我们能将较小的任务合并成较大的任务，只需描述好异步延续就可以了。和普通控制流一样，任务可以用多个不同的延续来处理错误情形，而通过操纵延续，可以将多个任务合并到一起。有几个技术可以做到这一点，最显而易见的就是ContinueWith()方法，如代码清单18-5和输出18-2所示。

代码清单18-5 调用Task.ContinueWith()

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

public class Program
{
  public static void Main()
  {
      Console.WriteLine("Before");
      // Use Task.Factory.StartNew<string>() for
      // TPL prior to .NET 4.5
      Task taskA =
          Task.Run( () =>
              Console.WriteLine("Starting..."))
          .ContinueWith(antecedent =>
              Console.WriteLine("Continuing A..."));
      Task taskB = taskA.ContinueWith( antecedent =>
          Console.WriteLine("Continuing B..."));
      Task taskC = taskA.ContinueWith( antecedent =>
          Console.WriteLine("Continuing C..."));
      Task.WaitAll(taskB, taskC);
      Console.WriteLine("Finished!");
  }
}
```

输出18-2

```
Before
Starting...
Continuing A...
Continuing C...
Continuing B...
Finished!
```

可用ContinueWith()“链接”两个任务，这样当**先驱任务**完成后，第二个任务（**延续任务**）自动以异步方式开始。例如在代码清单18-5中，Console.WriteLine("Starting...")是先驱任务的主体，而Console.WriteLine("Continuing A...")是它的延续任务主体。延续任务获取一个Task作为实参（antecedent），这样才能从延续任务的代码中访问先驱任务的完成状态。先驱任务完成时，延续任务自动开始，异步执行第二个委托，刚才完成的

先驱任务作为实参传给那个委托。此外，由于ContinueWith()方法也返回一个Task，所以当然可以作为另一个Task的先驱使用。以此类推，便可以建立起任意长度的连续任务链。

为同一个先驱任务调用两次ContinueWith()（例如在代码清单18-5中，taskB和taskC都是taskA的延续任务），先驱任务（taskA）就有两个延续任务。先驱任务完成时，两个延续任务将异步执行。注意，同一个先驱的多个延续任务的执行顺序在编译时不确定的。输出18-2只是恰巧taskC先于taskB执行。再次执行程序，就不一定是这个顺序了。不过，taskA总是先于taskB和taskC执行，因为后两者是taskA的延续任务。因此，它们不可能在taskA完成前开始。类似地，肯定是Console.WriteLine("Starting...")委托先执行完成，再执行taskA(Console.WriteLine("Continuing A..."))，因为后者是前者的延续任务。此外，“Finished!”总是最后显示，因为对Task.WaitAll(taskB,taskC)的调用阻塞了控制流，直到taskB和taskC都完成才能继续。

ContinueWith()有许多重载版本，有的要获取一个TaskContinuationOptions值，以便对延续链的行为进行调整。这些值都是位标志，可用逻辑OR操作符（|）合并。表18-1列出了部分标志值，详情参见MSDN文档[①]。

表18-1 TaskContinuationOptions枚举值列表

枚 举 值	说 明
None	这是默认行为；先驱任务完成后，不管任务状态是什么，延续任务都将执行
PreferFairness	如果两个任务一前一后异步开始，那么并不保证先开始的先运行。这个标志告诉任务调度器尽量公平地调度任务。换言之，较早调度的任务可能运行得更早，而较晚调度的任务可能运行得更晚。如果两个任务是从不同的线程池线程创建的，这个技术就特别有用了
LongRunning	告诉任务调度器这可能是一个I/O受限的高延迟任务，调度器可处理队列中的其他工作，不至于因为等待长时间运行的工作而“饥饿”。该选项要少用
AttachedToParent	指定任务连接到任务层次结构中的一个父任务
DenyChildAttach（.NET 4.5）	试图创建子任务将引发异常。 如果延续任务的代码试图使用AttachedToParent，会表现得就像没有父任务一样
NotOnRanToCompletion*	指定如果延续任务的先驱任务成功完成，就不应调度该延续任务。这个选项对多任务延续无效
NotOnFaulted*	指定如果延续任务的先驱任务引发一个未处理的异常，就不应调度该延续任务。这个选项对多任务延续无效
OnlyOnCanceled*	指定只有在延续任务的先驱任务被取消的情况下，才调度该延续任务。这个选项对多任务延续无效
NotOnCanceled*	指定如果延续任务的先驱任务被取消，就不应调度该延续任务。这个选项对多任务延续无效
OnlyOnFaulted*	指定只有在延续任务的先驱任务引发一个未处理的异常的情况下，才调度该延续任务。这个选项对多任务延续无效
OnlyOnRanToCompletion*	指定只有在延续任务的先驱任务成功完成的情况下，才应调度该延续任务 。这个选项对多任务延续无效
ExecuteSynchronously	指定延续任务应同步执行。若指定这个选项，延续任务会在造成先驱任务转变成它的最终状态的那个线程上运行[①]。如果在创建延续任务时，先驱任务已经完成，就在创建延续任务的那个线程上运行延续任务

① MSDN .NET Framework Developer Center，http://msdn.microsoft.com/en-us/library/system.threading.tasks.taskcontinuationoptions(v=vs.110).aspx。

续表

枚 举 值	说 明
HideScheduler（.NET 4.5）	在创建的任务中隐藏线程调度器。这意味着在创建好的任务中，Run/StartNew和ContinueWith等操作看到的当前线程调度器是TaskScheduler.Default(null)。如果延续任务应该在特定的调度器上运行，但调用了额外的代码，后者不应该由同一个调度器调度，这个技术就非常有用
LazyCancellation（.NET 4.5）	造成延续任务将监视取消标志的时间推迟到先驱任务结束之后。假定有任务t1，t2和t3，后者是前者的延续。如果t2在t1完成前取消，那么t3可能在t1完成前开始。设置LazyCancellation可避免这一点

* 指出在什么情况下延续任务。

带星号（*）的项指出延续任务在什么情况下执行。可利用它们创建类似于“事件处理程序”的延续任务，根据先驱任务的行为来进行操作。代码清单18-6演示了如何为先驱任务准备多个延续任务，根据先驱任务的完成情况来有条件地执行。

代码清单18-6 用ContinueWith()“登记”通知

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

public class Program
{
public static void Main()
{
    // Use Task.Factory.StartNew<string>() for
    // TPL prior to .NET 4.5
    Task<string> task =
        Task.Run<string>(
            () => PiCalculator.Calculate(10));

    Task faultedTask = task.ContinueWith(
        (antecedentTask) =>
        {
            Trace.Assert(task.IsFaulted);
            Console.WriteLine(
                "Task State: Faulted");
        },
        TaskContinuationOptions.OnlyOnFaulted);

Task canceledTask = task.ContinueWith(
    (antecedentTask) =>
    {
        Trace.Assert(task.IsCanceled);
        Console.WriteLine(
            "Task State: Canceled");
    },
    TaskContinuationOptions.OnlyOnCanceled);

Task completedTask = task.ContinueWith(
    (antecedentTask) =>
    {
        Trace.Assert(task.IsCompleted);
        Console.WriteLine(
```

```
                    "Task State: Completed");
            },  TaskContinuationOptions.
                    OnlyOnRanToCompletion);

        completedTask.Wait();
    }
}
```

这个代码清单实际是在登记有关先驱任务的“事件”。事件（任务正常或异常完成）一旦发生，就执行特定的“侦听”任务。这是一个很强大的功能，尤其是对于那些fire-and-forget[①]的任务——任务开始，和延续任务关联，然后就可以不管了。注意，代码清单18-6没有等待原始先驱任务。相反，等待的是用于处理完成事件的延续任务。完全可以丢弃对原始任务的引用；任务以异步方式开始执行，不需要任何后续的代码来检查状态。本例是改为调用completedTask.Wait()，确保主线程不在输出“任务完成”信息前退出，如输出18-3所示。

输出18-3

```
Task State: Completed.
```

注意在代码清单18-6中，千万不能在canceledTask或faultedTask上调用Wait()。那些延续任务只有在先驱任务取消或者引发异常时才运行。由于那些事件在本程序中不可能发生，所以任务永远运行不了。如果等待它们完成，将引发异常。代码清单18-6的延续选项恰好是互斥的，所以当先驱任务运行完成，而且与completedTask关联的任务开始执行时，任务调度器会自动取消与canceledTask和faultedTask关联的任务。取消的任务的最终状态是Canceled。所以，在这两种任务上调用Wait()或者其他会导致当前线程等待任务完成的方法，会引发一个异常，指出它们已经取消。

18.3.4 用AggregateException处理Task上的未处理异常

同步调用的方法可以包装到try块中，用catch子句 告诉编译器发生异常时应该执行什么代码。但异步调用不能这么做。不能用try块包装Start()调用来捕捉异常，因为控制会立即从调用返回，然后控制会离开try块，而这时距离工作者线程发生异常可能还有好久呢。一个解决方案是将任务的委托主体包装到try/catch块中。引发并被工作者线程捕捉的异常不会造成问题，因为try块在工作者线程上能正常地工作。但未处理的异常就麻烦了，工作者线程不捕捉它们。

自CLR 2.0起[②]，任何线程上的未处理的异常都被视为严重错误，会触发“Windowds错误报告”对话框，并造成应用程序异常中止。所有线程上的所有异常都必须被捕捉，否则应用程序不允许继续。（要了解处理未处理异常的高级技术，请参见稍后的“高级主题：处理未处理异常”。）幸好，异步任务中的未处理异常有所不同。在这种情况下，任务调度器会用一个“catchall”异常处理程序来包装委托。如果任务引发了未处理的异常，catchall处理程序会捕捉并记录异常细节，防止CLR自动终止进程。

---

① 本来是一个军事术语，指导弹发射后便不理会，它会自动寻找目标的。——译者注

② 在CLR1.0中，工作者线程上的未处理异常会终止线程，但不终止应用程序。这样一来，有bug的应用程序可能所有工作者线程都死了，主线程却还在运行——即使程序已经不再做任何工作。这为用户带来了困扰。更好的策略是通知用户应用程序处于不良状态，并在它造成损害前将其终止。

如代码清单18-6所示，为了处理出错的任务，一个技术是显式创建延续任务作为那个任务的“错误处理程序”。检测到先驱任务引发未处理的异常，任务调度器会自动调度延续任务。但是，如果没有这种处理程序，同时在出错的任务上执行Wait()（或其他试图获取Result的动作），就会引发一个AggregateException，如代码清单18-7和输出18-4所示。

代码清单18–7 处理任务的未处理异常

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

public class Program
{
  public static void Main()
  {
      // Use Task.Factory.StartNew<string>() for
      // TPL prior to .NET 4.5
      Task task = Task.Run(() =>
      {
          throw new InvalidOperationException();
      });

      try
      {
          task.Wait();
      }
      catch(AggregateException exception)
      {
          exception.Handle(eachException =>
          {
              Console.WriteLine(
                  "ERROR: {0}",
                  eachException.Message);
              return true;
          });
      }
  }
}
```

输出18–4

```
ERROR: Operation is not valid due to the current state of the object.
```

之所以叫集合异常（AggregateException），是因为它可能包含从一个或多个出错的任务收集到的异常。例如，异步执行10个任务，其中5个引发异常。为了报告所有5个异常，并用一个catch块处理它们，框架用AggregateException收集异常，并把它们当作一个异常来报告。除此之外，由于编译时不知道工作者任务将引发一个还是多个异常，所以未处理的出错任务总是引发一个AggregateException，代码清单18-7和输出18-4对此进行了演示。虽然工作者线程上引发的未处理异常是InvalidOperationException类型，主线程捕捉的仍是一个AggregateException。另外，捕捉异常需要一个AggregateException catch块。

AggregateException包含的异常列表通过InnerExceptions属性来获取。可以遍历该属性来检查每个异常并采取相应的对策。另外，如代码清单18-7所示，可以使用AggregateException.Handle()方法，为AggregateException中的每个异常都指定一个要执行的表达式。Handle()方法的重要特点在于它是一个断言。针对Handle()委托成功处理的

任何异常，断言应返回true。任何异常处理调用若为一个异常返回false，Handle()方法将引发新的AggregateException，其中包含了由这种异常构成的列表。

还可查看任务的Exception属性来了解出错任务的状态，这样不会造成在当前线程上重新引发异常。代码清单18-8通过等待任务的一个已知会引发异常的延续[①]来进行演示。

代码清单18-8 使用ContinueWith()观察未处理的异常

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

public class Program
{
  public static void Main()
  {
      bool parentTaskFaulted = false;
      Task task = new Task(() =>
          {
              throw new InvalidOperationException();
          });
      Task continuationTask = task.ContinueWith(
          (antecedentTask) =>
          {
              antecedentTaskIsFaulted =
                  antecedentTask.IsFaulted;
          }, TaskContinuationOptions.OnlyOnFaulted);
      task.Start();
      continuationTask.Wait();
      Trace.Assert(antecedentTaskIsFaulted);
      if (!task.IsFaulted)
      {
          task.Wait();
      }
      else
      {
          task.Exception.Handle(eachException =>
          {
              Console.WriteLine(
                  "ERROR: {0}",
                  eachException.Message);
              return true;
          });
      }
  }
}
```

注意，为了获取原始任务上的未处理异常，我们使用了Exception属性。结果和输出18-4一样。如果任务中发生的异常完全没有被观察到，那么a)它不会在任务中被捕捉到。b)永远观察不到任务完成（通过Wait()、Result或者访问Exception属性。c)出错的ContinueWith()永远观察不到。这可能造成异常完全未被处理，最终成为进程级别的未处理异常。在.NET 4.0中，像这样的异常会由终结器线程重新引发并可能造成进程崩溃。但在.NET 4.5 中，这个崩溃被阻止了（虽然可以配置CLR还是像以前那样崩溃）。

① 如前所述，一般都不等待有错的延续，因为大多数时候都不会安排它运行。这些代码仅供演示。

无论哪种情况，都可以通过TaskScheduler.UnobservedTaskException事件来登记未处理的任务异常。

高级主题：处理Thread上的未处理异常

如前所述，任何线程上的未处理异常默认都造成应用程序终止。未处理异常代表严重的、事先没有注意的bug，而且异常可能因为关键数据结构损坏而发生。由于不知道程序在发生异常后可能做什么，所以最安全的对策是立即关闭整个程序。

理想情况是程序在任何线程上都不引发未处理的异常。否则程序有bug，最好在交付客户前修复。但是，有些人认为在发生未处理异常时，不是尽快关闭程序，而是保存完好的数据，并且/或者记录异常以便进行错误报告和调试。这要求用一个机制来登记未处理异常通知。

每个AppDomain都提供了这样的一个机制，为了观察AppDomain中发生的未处理异常，必须添加一个UnhandledException事件来处理程序。无论主线程还是工作者线程，AppDomain中的线程发生的所有未处理异常都会触发UnhandledException事件 。

注意，这个机制的目的是通知，不允许应用程序从未处理异常中恢复并继续执行。事件处理程序运行完毕，应用程序会显示“Windows错误报告”对话框并退出。（如果是控制台应用程序，异常详细信息还会在控制台上显示。）

代码清单18-9展示了如何创建另一个线程来引发异常，并由AppDomain的未处理异常事件处理程序进行处理。出于演示目的，为了确保不受到线程计时问题的干扰，我们使用Thread.Sleep插入了一些人工延迟。结果如输出18-5所示。

代码清单18-9 登记未处理异常

```
using System;
using System.Diagnostics;
using System.Threading;

public class Program
{
  public static Stopwatch clock = new Stopwatch();
  public static void Main()
  {
    try
    {
      clock.Start();
      // Register a callback to receive notifications
      // of any unhandled exception.
      AppDomain.CurrentDomain.UnhandledException +=
        (s, e) =>
        {
          Message("Event handler starting");
          Delay(4000);
        };

      Thread thread = new Thread(() =>
      {
        Message("Throwing exception.");
        throw new Exception();
      });
      thread.Start();
```

```
      Delay(2000);
    }
    finally
    {
      Message("Finally block running.");
    }
  }

  static void Delay(int i)
  {
    Message(string.Format("Sleeping for {0} ms", i));
    Thread.Sleep(i);
    Message("Awake");
  }

  static void Message(string text)
  {
    Console.WriteLine("{0}:{1:0000}:{2}",
      Thread.CurrentThread.ManagedThreadId,
      clock.ElapsedMilliseconds,
      text);
  }
}
```

输出18-5

```
3:0047:Throwing exception.
3:0052:Unhandled exception handler starting.
3:0055:Sleeping for 4000 ms
1:0058:Sleeping for 2000 ms
1:2059:Awake
1:2060:finally block running.
3:4059:Awake
Unhandled Exception: System.Exception: Exception of type 'System.
Exception' was thrown.
```

如输出18-5所示，新线程被赋予线程ID 3，主线程被赋予线程ID 1。操作系统调度线程3运行一会儿，未处理异常被引发，调用事件来处理程序，然后进入睡眠。很快，操作系统意识到线程1可以调度了，但它的代码是立即睡眠。线程1先醒来并运行finally块，两秒钟后线程3醒来，未处理线程终于使进程崩溃。

执行事件处理程序，进程会等到处理程序完成后再崩溃，这是很典型的一个顺序，但却无法保证。一旦程序发生未处理异常，一切都会变得难以预料；程序现在处于一种未知的、不稳定的状态，其行为可能出乎意料。在这个例子中，CLR允许主线程继续运行并执行它的finally块，即使它知道当控制进入finally块时，另一个线程正处于AppDomain的未处理异常事件处理程序中。为了更深刻地理解这个事实，可试着修改延迟，使主线程睡眠得比事件处理程序长一些。这样finally块将不会运行！线程1被唤醒之前，进程就会因为未处理的异常而销毁了。取决于引发异常的线程是不是由线程池创建的，还可能得到不同的结果。因此，最佳实践就是避免所有未处理异常，不管在工作者线程中，还是在主线程中。

这跟任务有什么关系？如果关机时有未完成的任务将系统挂起怎么办？下一节将讨论任务取消。

规范

避免程序在任何线程上产生未处理异常。
考虑登记“未处理异常”事件处理程序以进行调试、记录和紧急关闭。
要取消未完成的任务而不要在程序关闭期间允许其运行。

18.4 取消任务

本章之前描述了为什么野蛮中止线程来取消正由该线程执行的任务不是一个好的选择。TPL使用的是**协作式取消**（cooperative cancellation），一种得体的、健壮的和可靠的技术来安全地取消不再需要的任务。支持取消的任务要监视一个CancellationToken对象（位于System.Threading命名空间）。任务定期轮询它，检查是否发出了取消请求。代码清单18-10演示了取消请求和对请求的响应。输出18-6展示了结果。

代码清单18-10 使用CancellationToken取消任务

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

public class Program
{
    public static void Main()
    {
        string stars =
            "*".PadRight(Console.WindowWidth-1, '*');
        Console.WriteLine("Push ENTER to exit.");

        CancellationTokenSource cancellationTokenSource=
            new CancellationTokenSource();
        // Use Task.Factory.StartNew<string>() for
        // TPL prior to .NET 4.5
        Task task = Task.Run(
            () =>
                WritePi(cancellationTokenSource.Token),
                    cancellationTokenSource.Token);

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

        cancellationTokenSource.Cancel();
        Console.WriteLine(stars);
        task.Wait();
        Console.WriteLine();
    }

    private static void WritePi(
        CancellationToken cancellationToken)
    {
        const int batchSize = 1;
        string piSection = string.Empty;
        int i = 0;

        while(!cancellationToken.IsCancellationRequested
            || i == int.MaxValue)
```

```
        {
            piSection = PiCalculator.Calculate(
                batchSize, (i++) * batchSize);
            Console.Write(piSection);
        }
    }
}
```

输出18-6

```
Push ENTER to exit.
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745O
**********
2
```

启动任务后，一个Console.Read()会阻塞主线程。与此同时，任务继续执行，计算并打印pi的下一位。用户按Enter键以后，执行就会遇到一个CancellationTokenSource.Cancel()调用。在代码清单18-10中，对task.Cancel()的调用和对task.Wait()的调用是分开的，两者之间会打印一行星号。目的是证明在观察到取消标志之前，极有可能发生一次额外的迭代。如输出18-6所示，星号后面多输出了一个2。之所以会显示这个2，是因为CancellationTokenSource.Cancel()不是“野蛮”地中止正在执行的Task。任务会继续运行，直到它检查标志，发现标志的所有者已请求取消任务，这时才会得体地关闭任务。

调用Cancel()实际会在从CancellationTokenSource.Token复制的所有取消标志上设置IsCancellationRequested属性。在这一句话中，有以下几个地方需要注意。

- 提供给异步任务的是CancellationToken（而不是CancellationTokenSource）。CancellationToken使我们能轮询取消请求，而CancellationTokenSource负责提供标志，并在取消时发出通知，如图18-2所示。

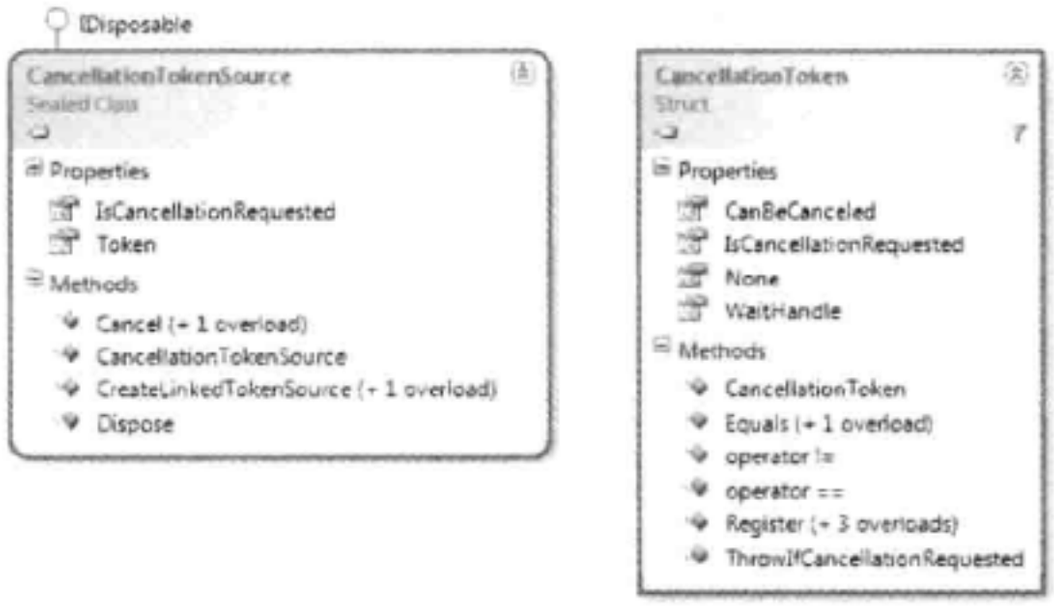


图18-2 CancellationTokenSource和CancellationToken类图

- CancellationToken是结构，所以能复制值。CancellationTokenSource.Token返回的是标志的副本。

为了监视IsCancellationRequested属性，CancellationToken的一个实例（从Cancellation TokenSource.Token获取）传给并行任务。代码清单18-9在计算出每一位之后，都检查CancellationToken参数上的IsCancellationRequested属性。如果属性返回true，while循环就退出。线程中止（abort）是在一个随机位置引发异常，而这里是使用正

常的控制流退出循环。通过频繁的轮询来保证代码能积极地响应取消请求。

关于CancellationToken，要注意的另一个要点是重载的Register()方法。通过这个方法，可以登记一个操作，在标志为已取消时调用。换言之，调用Register()方法将登记对应CancellationTokenSource的Cancel()上的一个侦听器委托。

在这个程序中，由于在任务完成前取消任务是预料之中的行为，所以代码清单18-9没有引发System.Threading.Tasks.TaskCanceledException。因此，task.Status会返回Task Status. RanToCompletion，不报告任务的工作实际已经取消。虽然本例不需要这样的报告，但需要的话TPL也能提供。如果取消任务会在某些方面造成破坏（例如，造成无法返回一个有效的结果），那么TPL为了报告这个问题，采用的模式就是引发一个TaskCanceledException（派生自System.OperationCanceledException）。不是显式引发异常，而是由CancellationToken包含一个ThrowIfCancellationRequested()方法，从而更容易地报告异常，前提是有一个可用的CancellationToken的实例。

在引发TaskCanceledException的任务上调用Wait()（或获取Result），结果和在任务中引发其他任何异常一样。这个调用会引发AggregateException。该异常标志着任务的执行状态可能不完整。在成功完成的任务中，所有预期的工作都成功执行。相反，被取消的任务可能有工作只是部分完成——工作的状态不可信。

这个例子演示了一个长时间运行的处理器受限操作（pi可以无限地计算下去）如何监视一个取消请求，并相应做出响应。但某些情况下，即便目标任务没有显式编码也能取消。例如，本章稍后讨论的Parallel类就默认提供了这样的行为。

18.4.1 Task.Run()是Task.Factory.StartNew()的简化形式

在.NET 4.0中，获取任务的一般方式是调用Task.Factory.StartNew()。.NET 4.5提供了更简单的调用方式Task.Run()。

Task.Factory.StartNew()用于调用一个要求创建额外线程的CPU密集型方法。而在.NET 4.5中，应该默认使用Task.Run()，除非它满足不了一些特殊要求。例如，要用TaskCreationOptions控制任务，要指定其他调度器，或者出于性能的考虑要传递对象状态，就可以考虑Task.Factory.StartNew()。只有需要将创建和调度分开（这种情况很少见），才可考虑在构造器调用之后添加一个Start()调用。

代码清单18-11是使用Task.Factory.StartNew()的一个例子。

代码清单18-11 使用Task.Factory.StartNew()

```
public Task<string> CalculatePiAsync(int digits)
{
  return Task.Factory.StartNew<string>(
      () => CalculatePi(digits));
}

private string CalculatePi(int digits)
{
    // ...
}
```

18.4.2 长时间运行的任务

前面在讨论代码清单18-2时指出，线程池假定所有工作都是处理器受限的，而且运行

时间都较短。这些假设的目的是控制创建的线程数量，防止因为过多分配昂贵的线程资源以及超额预订处理器而造成过于频繁的上下文切换和时间分片。

但是，如果开发人员知道一个任务要长时间运行，会长时间“霸占”一个底层线程资源，就可以通知调度器任务不会太快结束工作。这个通知有两方面的作用。首先，它提醒调度器或许应该为该任务创建专用线程（而不是使用来自线程池的）。其次，它提醒调度器可能应该调度比平时更多的线程。这会造成更多的时间分片，但这是一件好事。我们不希望长时间运行的任务霸占整个处理器，让其他短时间的任务没法运行。短时间运行的任务利用分配到的时间片，能在短时间内完成大部分工作，而长时间运行的任务基本注意不到因为和其他任务共享处理器而产生的些许延迟。为此要在调用StartNew()时使用TaskCreationOptions.LongRunning选项（Task.Run()不支持TaskCreationOptions参数），如代码清单18-12所示。

代码清单18-12 协作执行长时间运行的任务

```
using System.Threading.Tasks;

// ...

      Task task = Task.Factory.StartNew(
          () =>
              WritePi(cancellationTokenSource.Token),
                  TaskCreationOptions.LongRunning);
// ...
```

规范

要告诉任务工厂新建的任务可能长时间运行，使其能恰当地管理它。

要尽量少用TaskCreationOptions.LongRunning。

18.4.3 对任务进行资源清理

注意，Task还支持IDisposable。这是必然的，因为Task可能在等待完成时分配一个WaitHandle。由于WaitHandle支持IDisposable，所以Task根据最佳实践也要支持IDisposable。然而，读者会注意到在前面的代码示例中，既没有包含一个Dispose()调用，也没有通过using语句来隐式调用。相反，代码是依赖程序退出时的一个自动的WaitHandle终结器调用。

这造成了两个结果。首先，句柄存活时间变长了，因此会消耗更多资源。其次，垃圾回收器的效率变低了，因为被终结的对象存活到了下一代。但在Task的情况下，除非要终结大量任务，否则这两方面的问题都不大。因此，虽然从技术上说所有代码都应该对任务进行dispose（资源清理），但除非对性能的要求极为严格，而且做起来很容易（换言之，确定任务已经结束，而且没有其他代码在使用它们），否则就不必麻烦了。

18.5 C# 5.0基于任务的异步模式

如前所述，处理异步工作时，任务提供了比线程更好的抽象。任务自动调度为恰当数

量的线程，而且大型任务可由多个小任务组成，就和大量程序由多个小方法组成一样。

但任务有自己的缺点。其中最麻烦的是它“颠倒”了程序逻辑。为了演示这个问题，先来考虑一个同步方法，它因为一个Web请求（I/O受限的、高延迟的操作）而阻塞。然后，将它和C# 5.0之前的异步版本和**基于任务的异步模式**（Task-based Asynchronous Pattern，TAP）进行比较。最后，使用C# 5.0和async/await上下文关键字来实现同一个例子。

18.5.1 以同步方式调用高延迟操作

代码清单18-13使用WebRequest来下载网页并显示大小。操作失败将引发异常。

代码清单18-13 同步的Web请求

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

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

      try
      {
          Console.Write(url);
          WebRequest webRequest =
              WebRequest.Create(url);

          WebResponse response =
              webRequest.GetResponse();

          Console.Write(".....");

          using(StreamReader reader =
              new StreamReader(
                  response.GetResponseStream()))
          {
              string text =
                  reader.ReadToEnd();
              Console.WriteLine(
                  FormatBytes(text.Length));
          }
      }
      catch(WebException)
      {
          // ...
      }
      catch(IOException )
      {
          // ...
      }
      catch(NotSupportedException )
      {
          // ...
      }
```

```
}

static public string FormatBytes(long bytes)
{
    string[] magnitudes =
        new string[] { "GB", "MB", "KB", "Bytes" };
    long max =
        (long)Math.Pow(1024, magnitudes.Length);

    return string.Format("{1:##.##} {0}",
        magnitudes.FirstOrDefault(
            magnitude =>
                bytes > (max /= 1024)) ?? "0 Bytes",
        (decimal)bytes / (decimal)max).Trim();
  }
}
```

代码清单18-13的逻辑很简单，就是使用try/catch块和return语句等基本的东西来描述控制流。给定一个WebRequest，在它上面调用GetResponse()即可下载网页。要对网页进行流访问，调用GetResponseStream()并将结果赋给一个StreamReader。最后，用ReadToEnd()读到流的末尾，以便判断网页大小并打印结果。

这个方式的问题在于调用线程会被阻塞，直至I/O操作结束。异步工作进行期间，线程被白白浪费了，它本可做一些更有用的工作（如显示进度）。

18.5.2 使用TPL异步调用高延迟操作

代码清单18-14通过TPL来使用基于任务的异步模式，从而解决了上述问题。

代码清单18-14 异步的Web请求

```
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);

Task task = WriteWebRequestSizeAsync(url);

try
{
    while(!task.Wait(100))
    {
        Console.Write(".");
    }
}
catch(AggregateException exception)
{
```