```
                LastName,
                Phone,
                Address
            };
        }
    }
```

```
    public static string[] Headers
    {
        get
        {
            return new string[] {
                "First Name", "Last Name    ",
                "Phone       ",
                "Address                          " };
        }
    }

    // ...
}
```

```
class Publication : IListable
{
  public Publication(string title, string author, int year)
  {
      Title = title;
      Author = author;
      Year = year;
  }

  public string Title { get; set; }
  public string Author { get; set; }
  public int Year { get; set; }
```

```
  public string[] ColumnValues
  {
      get
      {
          return new string[]
          {
              Title,
              Author,
              Year.ToString()
          };
      }
  }

  public static string[] Headers
  {
      get
      {
          return new string[] {
              "Title                                ",
              "Author              ",
              "Year" };
      }
  }

  // ...
```

```
}
```

```
class Program
{
  public static void Main()
  {
      Contact[] contacts = new Contact[6];
      contacts[0] = new Contact(
          "Dick", "Traci",
          "123 Main St., Spokane, WA  99037",
          "123-123-1234");
      contacts[1] = new Contact(
          "Andrew", "Littman",
          "1417 Palmary St., Dallas, TX 55555",
          "555-123-4567");
      contacts[2] = new Contact(
          "Mary", "Hartfelt",
          "1520 Thunder Way, Elizabethton, PA 44444",
          "444-123-4567");
      contacts[3] = new Contact(
          "John", "Lindherst",
          "1 Aerial Way Dr., Monteray, NH 88888",
          "222-987-6543");
      contacts[4] = new Contact(
          "Pat", "Wilson",
          "565 Irving Dr., Parksdale, FL 22222",
          "123-456-7890");
      contacts[5] = new Contact(
          "Jane", "Doe",
          "123 Main St., Aurora, IL 66666",
          "333-345-6789");

      // Classes are cast implicitly to
      // their supported interfaces
      ConsoleListControl.List(Contact.Headers, contacts);

      Console.WriteLine();

      Publication[] publications = new Publication[3] {
          new Publication("Celebration of Discipline",
              "Richard Foster", 1978),
          new Publication("Orthodoxy",
              "G.K. Chesterton", 1908),
          new Publication(
              "The Hitchhiker's Guide to the Galaxy",
              "Douglas Adams", 1979)
              };
          ConsoleListControl.List(
              Publication.Headers, publications);
  }
}
```

```
class ConsoleListControl
{
    public static void List(string[] headers, IListable[] items)
    {
        int[] columnWidths = DisplayHeaders(headers);

        for (int count = 0; count < items.Length; count++)
```

```
        {
            string[] values = items[count].ColumnValues;
            DisplayItemRow(columnWidths, values);
        }
    }

    /// <summary>Displays the column headers</summary>
    /// <returns>Returns an array of column widths</returns>
    private static int[] DisplayHeaders(string[] headers)
    {
        // ...
    }

    private static void DisplayItemRow(
        int[] columnWidths, string[] values)
    {
        // ...
    }
}
```

代码清单7-2的结果如输出7-1所示。

输出7-1

```
First Name  Last Name       Phone          Address
Dick        Traci           123-123-1234   123 Main St., Spokane, WA  99037
Andrew      Littman         555-123-4567   1417 Palmary St., Dallas, TX 55555
Mary        Hartfelt        444-123-4567   1520 Thunder Way, Elizabethton, PA
44444
John        Lindherst       222-987-6543   1 Aerial Way Dr., Monteray, NH 88888
Pat         Wilson          123-456-7890   565 Irving Dr., Parksdale, FL 22222
Jane        Doe             333-345-6789   123 Main St., Aurora, IL 66666

Title                                      Author              Year
Celebration of Discipline                  Richard Foster      1978
Orthodoxy                                  G.K. Chesterton     1908
The Hitchhiker's Guide to the Galaxy       Douglas Adams       1979
```

在代码清单7-2中，ConsoleListControl可以显示看似无关的类（Contact和Publication）。一个类是否可以显示，只取决于它是否实现了必须的接口。ConsoleListControl.List()方法依赖多态性正确显示传给它的对象集。每个类都有自己的ColumnValues实现，将类转换成IListable就可调用特定的实现。

7.3 接口实现

声明类以实现接口，类似于从基类派生——要实现的接口和基类名称以逗号分隔（基类在前，接口顺序任意）。类可实现多个接口，但只能从一个基类直接派生。代码清单7-3展示了一个例子。

代码清单7-3 实现接口

```
public class Contact : PdaItem, IListable, IComparable
{
  // ...

  #region IComparable Members
```

```
        /// <summary>
        ///
        /// </summary>
        /// <param name="obj"></param>
        /// <returns>
        /// Less than zero:      This instance is less than obj.
        /// Zero                 This instance is equal to obj.
        /// Greater than zero    This instance is greater than obj.
        /// </returns>
        public int CompareTo(object obj)
        {
            int result;
            Contact contact = obj as Contact;

            if (obj == null)
            {
                // This instance is greater than obj.
                result = 1;
            }
            else if (obj != typeof(Contact))
            {
                throw new ArgumentException("obj is not a Contact");
            }
            else if( Contact.ReferenceEquals(this, obj) )
            {
                result = 0;
            }
            else
            {
                result = LastName.CompareTo(contact.LastName);
                if (result == 0)
                {
                    result = FirstName.CompareTo(contact.FirstName);
                }
            }
            return result;
        }
        #endregion

        #region IListable Members
        string[] IListable.ColumnValues
        {
            get
            {
                return new string[]
                {
                    FirstName,
                    LastName,
                    Phone,
                    Address
                };
            }
        }
        #endregion
    }
```

一旦某个类声明自己要实现接口，接口的所有成员都必须实现。抽象类允许提供接口成员的抽象实现。非抽象实现可在方法主体中引发NotImplementedException异常，但是必须要为成员提供一个实现。

接口的一个重要特征是永远不能实例化；不能使用new创建接口。所以接口没有构造器或终结器。只有实例化实现接口的类型，才能使用接口实例。此外，接口不能包含静态成员。接口的重要目的是多态性，而假如没有实现接口的那个类型的实例，多态性是没有什么价值的。

每个接口成员都与抽象方法的行为相似，从而强迫派生类实现成员。所以，不可以为接口成员显式地使用abstract修饰符。

在类型中实现接口成员时有两种方式：**显式和隐式**。之前看到的只是隐式实现，是用类型的公共成员实现接口成员。

7.3.1 显式成员实现

显式实现的方法只能通过接口本身调用；为此，最典型的做法是将对象转型为接口。例如在代码清单7-4中，为了调用IListable.ColumnValues，首先必须将Contact对象转型为IListable，因为ColumnValues成员是显式实现的。

代码清单7-4 调用显式接口成员实现

```
string[] values;
Contact contact1, contact2;

// ...

// ERROR:  Unable to call ColumnValues() directly
//         on a contact.
// values = contact1.ColumnValues;

// First cast to IListable.
values = ((IListable)contact2).ColumnValues;
// ...
```

本例对ColumnValues的转型和调用在一个语句中进行。也可在调用ColumnValues之前将contact2赋给一个IListable变量。

声明显式接口成员实现要在成员名之前附加接口名前缀，如代码清单7-5所示。

代码清单7-5 显式接口实现

```
public class Contact : PdaItem, IListable, IComparable
{
  // ...

  public int CompareTo(object obj)
  {
      // ...
  }

  #region IListable Members
  string[] IListable.ColumnValues
  {
      get
      {
          return new string[]
          {
              FirstName,
              LastName,
```

```
                Phone,
                Address
            };
        }
    }
    #endregion
}
```

代码清单7-5通过附加IListable前缀来显式实现ColumnValues。此外，由于显式接口实现直接与接口关联，因此没有必要使用virtual、override或者public来修饰它们。事实上，这些修饰符是不被允许的。这些不被看成是类的公共成员，所以使其成为public有误导之嫌。

7.3.2 隐式成员实现

在代码清单7-5中，CompareTo()没有包含IComparable前缀，所以是隐式实现的。要隐式实现成员，只要求成员是公共的，而且签名与接口成员的签名相符。接口成员实现不需要override关键字或者其他任何表明该成员与接口关联的指示符。此外，由于成员是像其他类成员那样声明的，所以可以像调用其他类成员那样直接调用隐式实现的成员：

```
result = contact1.CompareTo(contact2);
```

换言之，隐式成员实现不要求执行转型，因为成员可以直接调用，没有在实现它的类型中被隐藏起来。

在隐式实现上，显式实现不允许的许多修饰符都是必需或可选的。例如，隐式成员实现必须是public的。还可选择virtual，具体取决于派生类是否可以重写实现。去掉virtual导致成员被密封。

7.3.3 显式接口实现与隐式接口实现的比较

对于隐式和显式实现的接口成员，关键区别不在于成员声明的语法，而在于通过类型的实例而不是接口访问成员的能力。

构建类层次结构时，有必要建模真实世界的“属于”（is a）关系。例如，长颈鹿“属于”哺乳动物。这些是“语义”（semantic）关系。接口常用于建模“机制”（mechanism）关系。PdaItem“不属于”一种“可比较”（comparable）的东西，但它仍然可以实现IComparable接口。这个接口与语义模型无关，只是实现机制的细节。显式接口实现是将“机制方面的考虑”与“模型方面的考虑”分隔开的一种技术。要求调用者先将对象转换为接口（如IComparable），然后才能认为对象是“可比较”的，可显式地区分你想在什么时候和模型沟通，以及想在什么时候处理实现机制。

一般来说，最好的做法是将一个类的公共层面限制成“全模型”，尽量少地涉及无关的机制。遗憾的是，有的机制在.NET中是不可避免的。不能获得长颈鹿的散列码，或者将长颈鹿转换成字符串。但是，可以获得Giraffe（长颈鹿）类的散列码（GetHashCode()），并把它转换成string（ToString()）。利用object作为通用基类，.NET混合了模型代码和机制代码——即使混合程度仍然比较有限。

以下是一些基本的设计原则，可以利用它们帮助自己选择显式还是隐式实现。

- 成员是不是核心的类功能？以Contact类的ColumnValues属性实现为例。该成

员并非Contact类型的一个密不可分的部分，它仅仅是辅助成员，可能只有ConsoleListControl类才会访问它。所以，没有必要把它设计成Contact对象的一个直接可见的成员，使本来就很庞大的成员列表变得更加拥挤。

再来看看IFileCompression.Compress()成员。在ZipCompression类中包含隐式的Compress()实现是一个非常合理的选择，因为Compress()是ZipCompression类的核心功能，所以应当能从ZipCompression类直接访问。

- *接口成员名作为类成员是否恰当？* 假定ITrace接口的Dump()成员将类的数据写入跟踪日志。在Person或者Truck（卡车）类中隐式实现Dump()，人们可能混淆该方法所做的事情[①]。所以，更好的选择是显式实现，确保只能通过ITrace数据类型调用Dump()，使这个方法不会产生歧义。总之，假如成员的用途在实现类中不明确，就考虑显式实现。
- *是否已经有相同签名的类成员？* 显式接口成员实现不会在类型的声明空间添加具名元素。所以，假如类型已经有一个潜在的与之相冲突的成员，那么只要另一个是显式的接口成员，就可与之同名。

具体选择隐式还是显式接口成员实现，大多数时候都是凭直觉。然而，在进行选择的时候，上述问题为需要考虑的事情提供了一些建议。由于从隐式变成显式会造成版本中断，因此较稳妥的做法是全部显式定义接口，使它们以后能安全地变成隐式。此外，因为隐式还是显式不需要在所有接口成员之间保持一致，所以完全可以将部分成员定义成显式的，将其他定义成隐式的。

规范

避免显式实现接口成员，除非有很好的理由。但如果不确定成员的用途，就先选择显式实现。

7.4 在实现类和接口之间转换

与派生类和基类的关系相似，从实现类型[②]向它的已实现接口的转换是隐式转换，不需要转型操作符。实现类型的实例总是包含接口的全部成员，所以对象总是能成功转换为接口类型。

虽然从实现类型向已实现的接口的转换总成功，但可能有多个类型都实现了同一个接口。所以，无法保证从接口向实现类型的向下转型能成功。因此，从接口转换为它的某个实现类型，需要执行一次显式的强制类型转换。

7.5 接口继承

一个接口可以从另一个接口派生，派生的接口将继承“基接口”的所有成员。如代码清单7-6所示，直接从IReadableSettingsProvider派生的接口是显式基接口。

---

① Dump在这里的意思是“转储”类的数据，但用于Truck类会把它同“卸货”联系起来，从而造成混淆。——译者注

② “实现类型”也就是实现接口的类。——译者注

代码清单7-6 从一个接口派生出另一个接口

```
interface IReadableSettingsProvider
{
    string GetSetting(string name, string defaultValue);
}
```

```
interface ISettingsProvider : IReadableSettingsProvider
{
    void SetSetting(string name, string value);
}
```

```
class FileSettingsProvider : ISettingsProvider
{
    #region ISettingsProvider Members
    public void SetSetting(string name, string value)
    {
        // ...
    }
    #endregion

    #region IReadableSettingsProvider Members
    public string GetSetting(string name, string defaultValue)
    {
        // ...
    }
    #endregion
}
```

在这个例子中，ISettingsProvider是从IReadableSettingsProvider派生的，所以会继承它的成员。假如IReadableSettingsProvider本身还有一个显式基接口，那么ISettingsProvider还会继承那个接口的成员。

有趣的是，假如显式实现GetSetting()，那么必须通过IReadableSettingsProvider进行。在代码清单7-7中，通过ISettingsProvider进行将无法通过编译。

代码清单7-7 未提供正确的包容接口的显式成员声明（错误）

```
// ERROR:  GetSetting() not available on ISettingsProvider
string ISettingsProvider.GetSetting(
  string name, string defaultValue)
{
  // ...
}
```

编译代码清单7-7，会获得如输出7-2所示的错误信息。

输出7-2

```
'ISettingsProvider.GetSetting' in explicit interface declaration
is not a member of interface.
```

伴随这个输出，还有一个错误信息指出IReadableSettingsProvider.GetSetting()尚未实现。在显式实现接口成员时使用的完全限定接口成员名称中，必须引用最初声明它的那个接口的名称。

即使类实现的是从基接口(IReadableSettingsProvider)派生的接口(ISettingsProvider)，仍然可以明确声明自己要实现这两个接口，如代码清单7-8所示。

代码清单7-8 在类声明中使用基接口

```
class FileSettingsProvider : ISettingsProvider,
      IReadableSettingsProvider
{
    #region ISettingsProvider Members
    public void SetSetting(string name, string value)
    {
        // ...
    }
    #endregion

    #region IReadableSettingsProvider Members
    public string GetSetting(string name, string defaultValue)
    {
        // ...
    }
    #endregion
}
```

在这个代码清单中，类的接口实现并没有改变。虽然类标题中附加的接口实现声明纯属多余，但它提供了更好的可读性。

提供多个接口，而非单独提供一个复合接口，这个决策在很大程度上依赖于接口设计者对于类的实现有什么要求。提供一个IReadableSettingsProvider接口，设计者告诉实现者：只需实现用于获取设置的“设置提供者”就可以了。实现者不需要考虑写入设置，从而减轻了实现者的负担。

相反，实现ISettingsProvider是基于这样一个前提：任何类都不可能只能写入而不能读取设置。因此，ISettingsProvider和IReadableSettingsProvider之间的继承关系强迫在FileSettingsProvider类上合并这两个接口。

最后要说的是，虽然“继承”这个词用得没错，但更准确的说法是接口代表契约，一份契约可指定另一份契约也必须遵守的条款。所以，ISettingsProvider:IReadableSettingsProvider从概念上是说ISettingsProvider契约还要求遵守IReadableSettingsProvider契约，而不是说ISettingsProvider“属于一种”IReadableSettingsProvider。话虽这么说，但为了和标准的C#术语保持一致，本章剩余部分仍会使用继承关系术语。

7.6 多接口继承

就像类能实现多个接口那样，接口也能从多个接口继承，而且语法与类的继承和实现语法一致，如代码清单7-9所示。

代码清单7-9 多接口继承

```
interface IReadableSettingsProvider
{
  string GetSetting(string name, string defaultValue);
}
```

```
interface IWriteableSettingsProvider
{
  void SetSetting(string name, string value);
}
```

```
interface ISettingsProvider : IReadableSettingsProvider,
    IWriteableSettingsProvider
{
}
```

接口一般很少没有成员。但是，假如两个接口必须一起实现，这种情况就再正常不过了。代码清单7-9和代码清单7-6的区别在于，现在可以在不提供任何读取功能的前提下实现IWriteableSettingsProvider。代码清单7-6的FileSettingsProvider不受影响，但假如它使用的是显式成员实现，就要以稍微不同的方式指定成员从属于哪个接口。

7.7 接口上的扩展方法

扩展方法的一个重要特点是除了能作用于类，还能作用于接口。语法和作用于类时一样。方法的第一个参数是要扩展的接口，该参数必须附加this修饰符。代码清单7-10展示了作用于IListable接口的一个扩展方法，它本身是在Listable类中声明的。

代码清单7-10 接口的扩展方法

```
class Program
{
  public static void Main()
  {
      Contact[] contacts = new Contact[6];
      contacts[0] = new Contact(
          "Dick", "Traci",
          "123 Main St., Spokane, WA  99037",
          "123-123-1234");
      // ...

      // Classes are implicitly converted to
      // their supported interfaces
      contacts.List(Contact.Headers);

      Console.WriteLine();

      Publication[] publications = new Publication[3] {
          new Publication("Celebration of Discipline",
              "Richard Foster", 1978),
          new Publication("Orthodoxy",
              "G.K. Chesterton", 1908),
          new Publication(
              "The Hitchhiker's Guide to the Galaxy",
              "Douglas Adams", 1979)
          };
      publications.List(Publication.Headers);
  }
}
```

```
static class Listable
{
  public static void List(
      this IListable[] items, string[] headers)
  {
      int[] columnWidths = DisplayHeaders(headers);

      for (int itemCount = 0; itemCount < items.Length; itemCount++)
      {
          string[] values = items[itemCount].ColumnValues;
```

```
            DisplayItemRow(columnWidths, values);
        }
    }
    // ...
}
```

要注意的是，在这个例子中，扩展方法扩展的不是一个IListable参数（虽然完全可以这样）。相反，它扩展的是一个IListable[]。这证明C#不仅允许为特定类型的实例添加扩展方法，还允许为那些对象的集合添加。对扩展方法的支持是实现LINQ的基础。IEnumerable是所有集合都要实现的基本接口。通过为IEnumerable定义扩展方法，为所有集合都添加了LINQ支持。这显著地改变了对象集合的编程方式，这个主题将在第14章详细讨论。

7.8 通过接口实现多继承

如代码清单7-3所示，虽然类只能从一个基类派生，但可以实现任意数量的接口。这在一定程度上解决了C#类不支持多继承的问题。为此，要像上一章讲述的那样使用聚合（参见6.1.5节），但可以稍微改变一下结构，在其中添加一个接口，如代码清单7-11所示。

代码清单7-11 使用接口和聚合来实现多继承

```
public class PdaItem
{
  // ...
}
```

```
interface IPerson
{
    string FirstName
    {
        get;
        set;
    }

    string LastName
    {
        get;
        set;
    }
}
```

```
public class Person : IPerson
{
  // ...
}
```

```
public class Contact : PdaItem, IPerson
{
  private Person Person
  {
      get { return _Person; }
      set { _Person = value; }
  }
  private Person _Person;

  public string FirstName
  {
```

```
        get { return _Person.FirstName; }
        set { _Person.FirstName = value; }
    }

    public string LastName
    {
        get { return _Person.LastName; }
        set { _Person.LastName = value; }
    }

    // ...
}
```

IPerson确保Person的成员和复制到Contact的成员具有一致的签名。然而，这个实现仍然没有做到与“多继承”真正同义，因为添加到Person的新成员不会同时添加到Contact。

如果被实现的成员是方法（而非属性），那么有一个办法可对此进行改进。具体就是为从第二个基类“派生”的附加功能定义接口扩展方法。例如，可为IPerson定义扩展方法VerifyCredentials()。这样，实现IPerson（即使IPerson接口没有成员，只有扩展方法）的所有类都会有VerifyCredentials()的默认实现。这之所以可行，完全是多态性和重写的功劳。之所以支持重写，是因为方法的任何实例实现都要优先于具有相同静态签名的扩展方法。

规范

考虑通过定义接口来获得和多继承相似的效果。

初学者主题：接口图示

UML图中[①]的接口可能有两种形式。第一种，可以将接口显示成与类继承相似的继承关系。在图7-1中，IPerson和IContact之间的关系就是这样的。第二种，可以使用小圆圈显示接口，一般将这种小圆圈称为“棒棒糖”，如图7-1的IPerson和IContact所示。

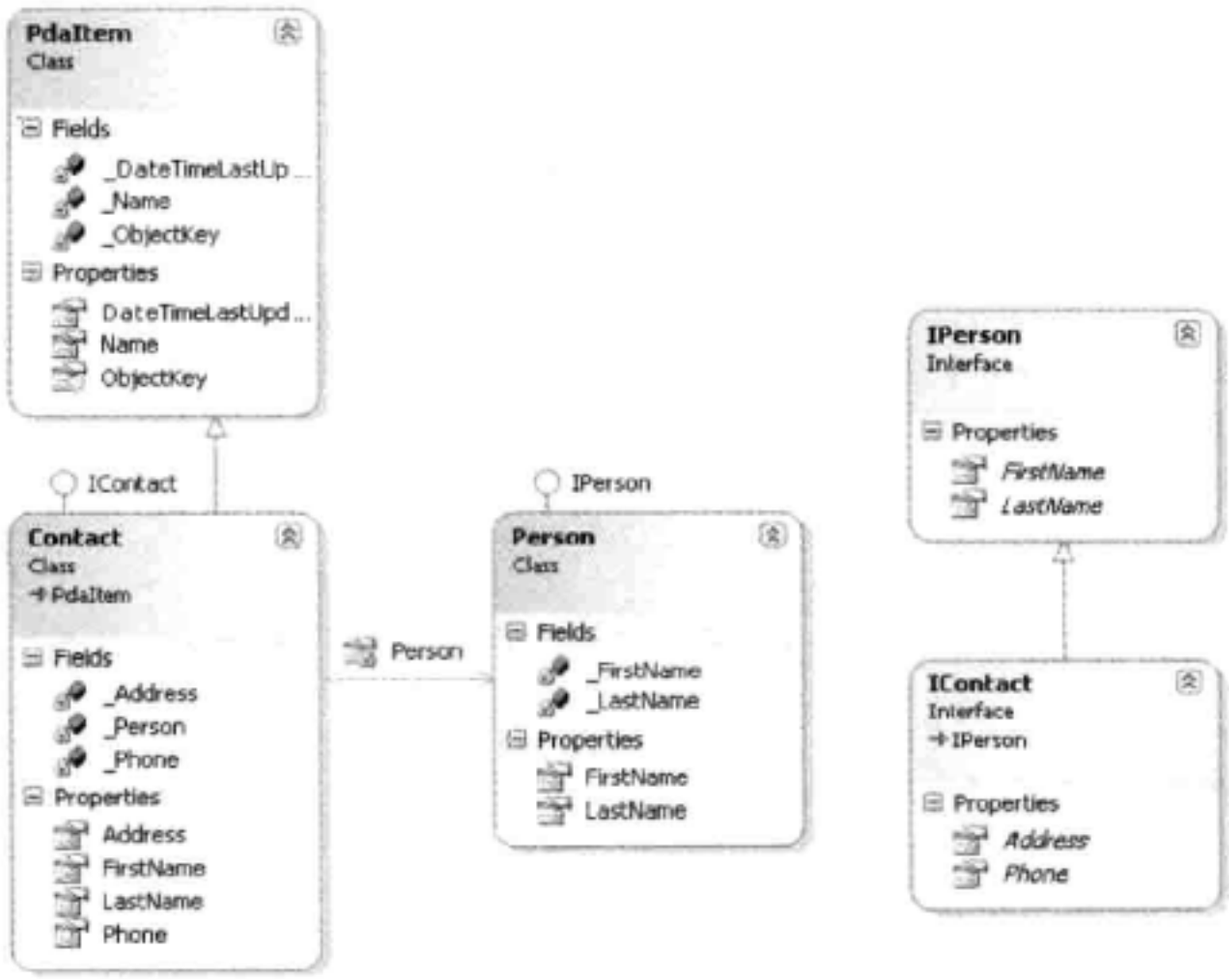


图7-1 接口图示

① UML是Unified Modeling Language的缩写，即“统一建模语言”，是使用图示来建模对象设计的一个标准规范。

在图7-1中，Contact从PdaItem派生，并实现了IContact。除此之外，它聚合了Person类，后者实现了IPerson。虽然Visual Studio 2005类设计器不支持，但接口有时也可使用与派生关系相似的箭头来显示。例如，在图7-1中，Person可以连接一条箭头线到IPerson，而不是画一个“棒棒糖”来表示。

7.9 版本控制

假定一个组件或应用程序供其他开发者使用，那么在创建它的新版本时不应改动接口。接口在负责实现的类和使用接口的类之间订立了契约，改动接口相当于改动契约，会使基于接口写的代码失效。

更改或删除特定接口成员的签名，明显会造成现有代码的中断，因为除非进行修改，否则对该成员的任何调用都不再能够编译。更改类的public或protected成员签名也会这样。但和类不同，在接口中添加成员，也有可能造成代码无法编译——除非进行额外的修改。这里的问题在于，实现接口的任何类都必须完整地实现，必须提供针所有成员的实现。在添加了新的接口成员后，编译器会要求开发者在实现接口的类中添加新的接口成员。

规范

不要为已交付的接口添加成员。

来看看代码清单7-12的IDistributedSettingsProvider接口，它很好地演示了如何以一种版本兼容的方式扩展现有接口。假定最开始只定义了ISettingsProvider接口（如代码清单7-6所示）。但在下个版本中，设计者提出了对每台机器的设置需求。为此创建了IDistributedSettingsProvider，它从ISettingsProvider派生。

代码清单7-12 从一个接口派生出另一个接口

```
interface IDistributedSettingsProvider : ISettingsProvider
{
  /// <summary>
  /// Get the settings for a particular machine.
  /// </summary>
  /// <param name="machineName">
  /// The machine name the setting is related to</param>
  /// <param name="name">The name of the setting</param>
  /// <param name="defaultValue">
/// The value returned if the setting is not found.</param>
/// <returns>The specified setting</returns>
string GetSetting(
    string machineName, string name, string defaultValue);

/// <summary>
/// Set the settings for a particular machine.
/// </summary>
/// <param name="machineName">
/// The machine name the setting is related to.</param>
/// <param name="name">The name of the setting.</param>
/// <param name="value">The value to be persisted.</param>
/// <returns>The specified setting</returns>
void SetSetting(
```

```
        string machineName, string name, string value);
}
```

这个设计的要点在于，其他程序员在自己的类中实现ISettingsProvider时，可选择升级实现来包含IDistributedSettingsProvider，也可选择忽略它。

但假如不是创建新接口，而是在现有的ISettingsProvider接口中添加与机器相关的方法，那么在新的接口定义下，实现该接口的类就不再能成功编译，这个改变造成版本中断。

在开发阶段更改接口显然是可以接受的，虽然开发者可能要为此付出相当多的劳动（在接口被广泛实现的前提下）。然而，一旦接口发布，就不应该更改它。相反，应该创建第二个接口，它可以从原始接口派生。

（代码清单7-12包含描述接口成员的XML注释。第9章将进一步讨论。）

7.10 接口与类的比较

接口引入了另一个类别的数据类型（是少数不对终极基类System.Object进行扩展的类型之一[①]）。但和类不同的是，接口永远不能实例化。只能通过对实现接口的一个对象的引用来访问接口实例。不能用new操作符创建接口实例；所以接口不能包含任何构造器或终结器。此外，接口中不允许静态成员。

接口近似于抽象类，两者具有一些共同的特点，比如都缺少实例化能力。表7-1对它们进行了比较。

表7-1 抽象类和接口的比较

抽 象 类	接 口
不能直接实例化，只能通过实例化一个派生类	不能直接实例化，只能通过实例化一个实现类型
派生类要么自己也是抽象的，要么必须实现所有抽象成员	实现类型必须实例化所有接口成员
可添加额外的非抽象成员，它们可由所有派生类继承，而不会破坏跨版本兼容性	为接口添加额外的成员会破坏版本兼容性
可声明属性和字段	可声明属性但不能声明字段
成员可以是实例、虚、抽象或静态的，而且非抽象成员可提供默认实现供派生类使用	所有成员都被自动看成是抽象成员，因此不能包含任何实现
派生类只能从一个基类派生（单继承）	实现类型可实现任意多的接口

抽象类和接口各有优缺点，必须根据表7-1的比较来做出最佳选择。

规范

一般要优先选择类而不是接口。用抽象类将契约（类型做什么）与实现细节（类型怎么做）分离开。

想在已从其他类型派生的类型上支持接口所定义的功能时，就考虑定义接口。

① 此外还有指针类型和类型参数类型。不过，每个接口类型都可转换为System.Object，而且可在接口的任何实例上调用System.Object的方法，所以这个区别或许有点儿吹毛求疵。

7.11 接口与特性的比较

有时用无任何成员（不管是不是继承的）的接口表示类型的信息。例如，可以创建名为IObsolete的标记接口（marker interface）指出某个类型已被另一类型取代。一般认为这是对接口机制的“滥用”；接口应该用于表示类型能提供的功能，而非陈述关于某个类型的事实。所以不要使用标记接口，而是使用特性（attributes）。详情参见第17章。

规范

避免使用无成员的标记接口，而是使用特性。

7.12 小结

接口是C#面向对象编程的关键元素。它提供了与抽象类相似的功能，但没有用掉单一继承选项，还支持实现多个接口。

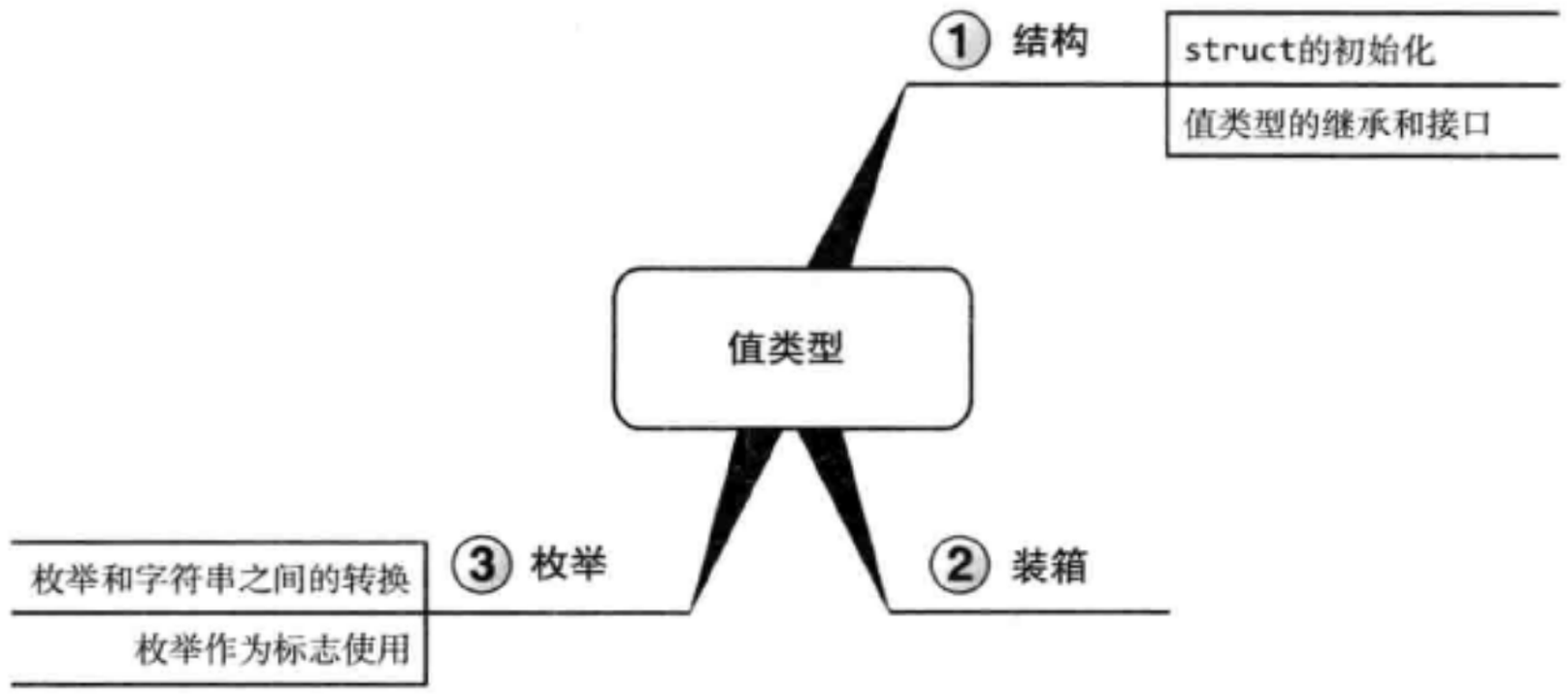
在C#中，接口的实现可以是显式或隐式的，具体取决于实现类是直接公开接口成员，还是通过到接口的强制转换来公开。此外，对于实现的显式和隐式的决定粒度还可以是在接口成员的级别上做出的——一个成员可以是隐式的，而同一接口的另一个成员可以是显式的。

下一章将探讨值类型，并讨论定义自定义值类型的重要性。同时会指出值类型可能带来的一些问题。

第8章

值类型

到目前为止，本书已经使用了大量值类型；例如，int就是值类型。本章不仅要讨论值类型的使用，还要讨论如何定义自定义值类型。有两种自定义值类型。第一种是结构。本章讨论了如何利用结构来定义新的值类型，使之具有与第2章讨论的大多数预定义类型相似的行为。这里的关键是，任何新定义的值类型都有它们自己的数据和方法。第二种是枚举。本章讨论了如何利用枚举定义常量值的集合。



8.1 结构

初学者主题：类型的分类

迄今为止讨论的所有类型都属于两个类别之一：引用类型和值类型。它们的区别源于复制策略的不同，后者又造成每种类型在内存中以不同的方式存储。为了巩固之前学习的知识，这里将再次介绍值类型和引用类型，便于大家温故而知新。

值类型

值类型直接包含值，如图8-1所示。换言之，变量引用的位置就是值在内存中实际存储的位置。因此，将一个原始变量的值赋给另一个变量，会在新变量的位置创建原始变量的值的一个内存副本。两个变量不可能引用同一个内存位置（除非其中一个或两个都是out或ref参数；根据定义，这种参数是另一个变量的别名），更改原始变量的值不会影响另一个变量的值，变量各有各的存储位置。因此，改变一个值类型变量的值不会影响其他任何值类型的值。

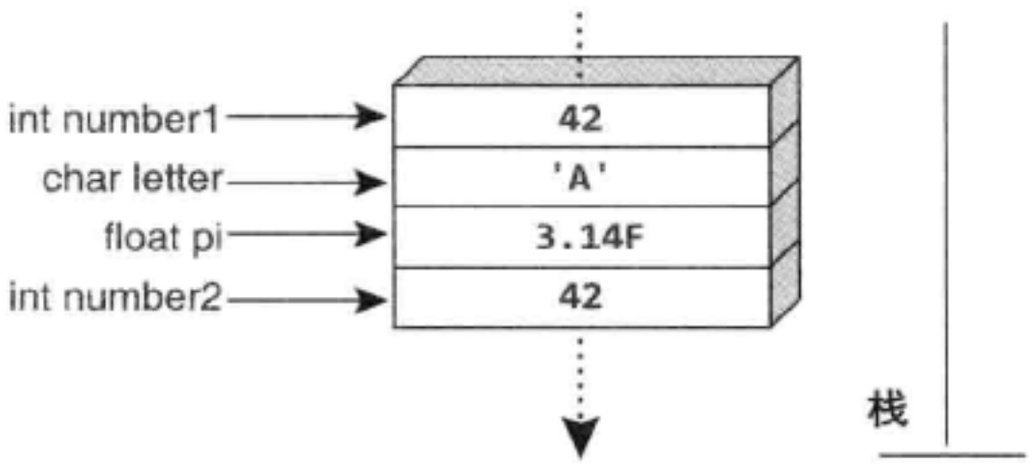


图8-1 值类型直接包含数据

值类型的变量就像一张上面写了数字的纸。如果要更改数字，可以擦除它并上写不同的数字。如果还有另一张纸，就可以把第一张纸上的数字抄写过来。但此后，这两张纸就是独立的了。在一张纸上面擦除和替换数字不会影响另一张纸上的数字。

类似地，将值类型的实例传给Console.WriteLine()这样的方法，也会生成一个内存副本。在方法内部对参数值进行任何修改都不会影响调用者中的原始值。由于值类型需要有一个内存副本，所以定义时通常不要让它们消耗太多内存（通常应该小于16字节）。

规范

不要创建消耗内存大于16字节的值类型。

值类型的值一般只是短时间存在。这样的值通常作为表达式的一部分，或用于激活方法。在这些情况下，值类型的变量和临时值经常是存储在称为**栈**的临时存储池中。（用词不太恰当。临时池并非一定要从栈中分配存储。事实上，作为一个实现细节，它经常选择从可用的寄存器中分配存储。）

临时池清理起来的代价低于需要进行垃圾回收的**堆**。不过，值类型要比引用类型更频繁地复制，这种复制操作会增加性能的开销。不要陷入“值类型更快是因为它们能在栈上分配存储”的误区。

引用类型

相反，对于**引用类型**的变量，它的值是对一个对象实例的引用，如图8-2所示。引用类型的变量存储的是引用（通常是内存地址），要去那个位置找到对象实例的数据。因此，为了访问数据，“运行时”要从变量中读取引用，然后对它进行“解引用[①]”，从而到达实际包含实例数据的内存位置。

① 引用（reference）引用一个资源，解引用（dereference）根据引用取得资源，也有人将后者翻译为“提领”、“用引”等。——译者注

所以，引用类型的变量关联了两个存储位置：直接和变量关联的存储位置，以及由变量中存储的值引用的存储位置。

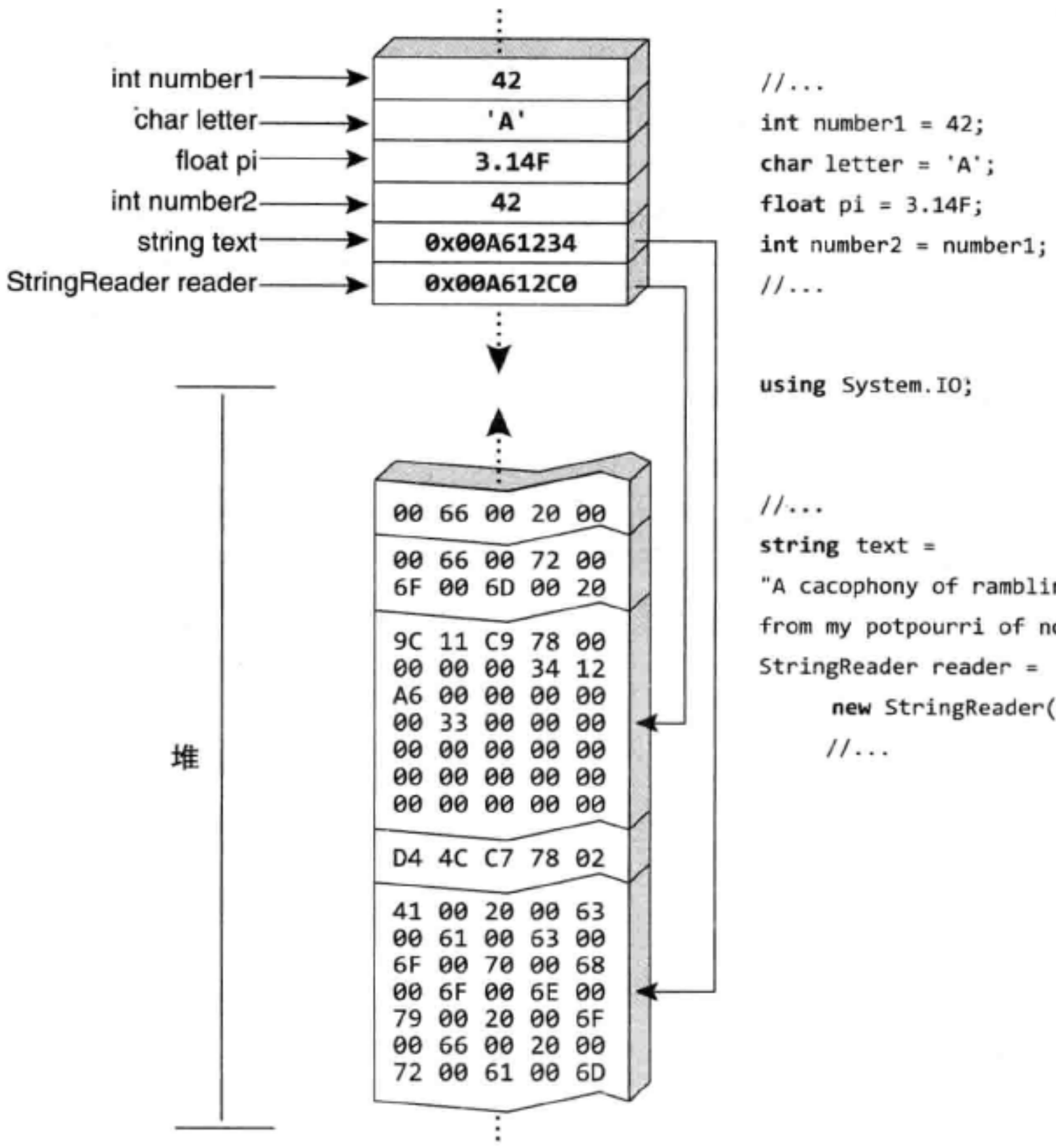


图8-2 引用类型指向堆

引用类型的变量也像是一张上面总是写了东西的纸。例如，假定一张纸上写了家庭地址“123 Sesame Street, New York City”。这张纸是一个变量，地址是对一幢建筑物的引用。纸和上面写的地址都不是建筑物本身，而且纸在哪里跟纸上的内容所指向的那幢建筑物在哪里没有任何关系。在另一张纸上复制这个引用，两张纸的内容都指向同一幢建筑物。以后将建筑物漆成绿色，可观察到两张纸引用的建筑物都变成绿色的了，因为引用指向的是同一个事物。

对于直接与变量（或临时值）关联的存储位置和与值类型变量关联的存储位置，它们的处理方式没有区别。如果已知变量仅在短时间存在，就在临时存储池中分配它。对于引用类型的变量，它的值要么是null，要么是对需要进行垃圾回收的堆上的一个存储位置的引用。

值类型的变量直接存储实例的数据；相反，要进行一次额外的“跳转”才能访问到与引用关联的数据。首先要对引用进行“解引用”来找到实际数据的存储位置，然后才能读取或写入数据。复制引用类型的值时，复制的只是引用，这个引用非常小。（一个引用的大小就是处理器的“bit size”；32位机器是4字节的引用，64位机器是8字节的引

用，以此类推。）复制值类型的值会复制所有的数据，这些数据可能很大。所以，有时复制引用类型的效率更高。这正是编码规范要求值类型不得大于16字节的原因。如果复制值类型的代价比作为引用复制时高出4倍，就应该把它设计成引用类型了。

由于引用类型只复制对数据的引用，因此两个变量可引用相同的数据。除此之外，利用一个变量来更改数据，用另一个变量访问数据时将看到更改结果。赋值和方法调用都会如此。

还是前面的例子，如果将建筑物的地址传给方法，那么就相当于制作了包含引用的那张纸的副本，并将这个副本传给方法。方法无法改变原始纸张的内容来引用不同的建筑物。但是，如果方法将引用的建筑物漆成别的颜色，当方法返回时，调用者能观察到这个变化。

除了string和object是引用类型，所有C#“内建”类型（如bool和decimal）都是值类型。除此之外，.NET框架还提供了其他大量值类型。开发者甚至能定义自己的值类型。

定义自定义值类型使用的是和定义类及接口相似的语法。区别在于值类型使用关键字struct，如代码清单8-1所示。该值类型表示的是高精度角，即用度、分、秒来表示的角度。（1 度=60分=3600秒。该系统在导航系统中使用，在赤道海平面上沿子午线航行1分所走过的弧线正好是1海里。）

代码清单8-1 声明结构

```
// Use keyword struct to declare a value type.
struct Angle
{
public Angle(int degrees, int minutes, int seconds)
{
    _Degrees = degrees;
    _Minutes = minutes;
    _Seconds = seconds;
}

public int Degrees
{
    get { return _Degrees; }
}
private int _Degrees;

public int Minutes
{
    get { return _Minutes; }
}
private int _Minutes;

public int Seconds
{
    get { return _Seconds; }
}
private int _Seconds;

public Angle Move(int degrees, int minutes, int seconds)
{
    return new Angle(
        Degrees + degrees,
        Minutes + minutes,
        Seconds + seconds)
```

```
  }

}

// Declaring a class - a reference type
// (declaring it as a struct would create a value type
// larger than 16 bytes.)
class Coordinate
{
  public Angle Longitude
  {
      get { return _Longitude; }
      set { _Longitude = value; }
  }
  private Angle _Longitude;

  public Angle Latitude
  {
      get { return _Latitude; }
      set { _Latitude = value; }
  }
  private Angle _Latitude;

}
```

上述代码将Angle定义为值类型，用于存储角度（无论经度还是纬度）的度、分、秒。这样生成的C#类型称为**结构**。

注意

虽然语言本身未作要求，但对于使用值类型的一种良好的规范是确保值类型是不可变的。换言之，一旦实例化值类型，实例就不能修改。要修改，应该创建新实例。代码清单8-1提供了一个Move()方法，它不修改Angle的实例，而是返回一个全新实例。

该规范是出于两方面的考虑。首先，值类型表示的是值。应该没有人会想着将两个整数相加到一起，而这两个数中的任何一个都是不断变化的。所以，两个加数应该是不可变的，生成第三个值作为结果。

其次，因为值类型是按值复制的，而不是按引用复制，所以会很容易地迷惑或者错误地认为，一个值类型变量的变化也会被观察到引起了另一个变量的变化，就像引用类型的行为一样。

规范

要创建不可变的值类型。

8.1.1 结构的初始化

除了属性和字段，结构还可包含方法和构造器，但不可包含用户定义的默认（无参）构造器。每个结构自动获得一个无参构造器将所有字段初始化为各自的默认值。被自动初始化了的结构变量（比如结构类型的一个数组的元素）会自动初始化变量，就好像调用了

这个默认构造器一样。由于编译器会将声明时的实例字段赋值放到类型的默认构造器中进行，所以C#还禁止了在声明时对结构的实例字段进行赋值，如代码清单8-2所示。

代码清单8-2 声明的同时初始化结构的字段会造成错误

```
struct Angle
{
  // ...
  // ERROR:  Fields cannot be initialized at declaration time
  // int _Degrees = 42;
  // ...
}
```

但这不是说不需要对字段进行初始化。除非所有字段都初始化，否则结构类型的局部变量无法使用。

幸好C#支持有参构造器，而且它们提出了一个有趣的初始化要求。为了确保值类型的局部变量被完全初始化，每个构造器都必须初始化结构中的所有字段。不这样做会造成编译错误。例如，代码清单8-3的构造器初始化了属性但没有初始化字段，所以造成编译错误。

代码清单8-3 不初始化字段就访问属性

```
// ERROR:  The 'this' object cannot be used before
  //         all of its fields are assigned to
  // public Angle(int degrees, int minutes, int seconds)
  // {
  //     Degrees = degrees;     // Shorthand for this.Hours = hours;
  //     Minutes = minutes;     // Shorthand for this.Minutes = ...;
  //     Seconds = seconds;     // Shorthand for this.Seconds = ...;
  // }
```

错误消息指出，方法和属性（Hours暗指this.Hours）是在初始化好所有字段之前访问的。要解决这个问题，需要直接初始化字段，如代码清单8-1所示。

规范

要确保结构的默认值有效，总是可以获得结构默认的“全零”值。

高级主题：将new用于值类型

为引用类型使用new操作符，会造成“运行时”在堆上创建对象的新实例，并将所有字段初始化为默认值，再调用构造器，将对该实例的引用以this传给构造器。结果是对最终的实例的引用，该引用将被复制到最终的目的地。相反，为值类型使用new操作符，会造成“运行时”在临时存储池中创建对象的新实例，并将所有字段初始化为默认值，再调用构造器（将临时存储位置作为ref变量以this传给构造器），结果还是临时存储位置的值，该值将被复制到最终的目的地。

和类不同，结构不支持终结器。结构以值的形式复制，不像引用类型那样具有“引用同一性”，所以难以知道在什么时候能安全执行终结器并释放结构占用的非托管资源。垃圾回收器知道在什么时候不再有对一个引用类型实例的“活动”引用，可在此之后的任何

时间为这个引用类型运行终结器。但是，“运行时”没有任何机制能跟踪值类型在特定时刻有多少个副本。

语言对比：C++——struct定义带公共成员的类型

在C++中，对于用struct和class声明的类型，区别在于默认的可访问性是公共还是私有。两者在C#中的区别则要大得多，在于复制类型的实例时复制的是值还是引用。

8.1.2 default操作符的使用

如前所述，所有值类型都有自动定义的无参构造器将值类型的实例初始化成默认状态。所以，任何值类型都可合法地使用new S()这样的语法。除此之外，还可使用default操作符生成结构的默认值。代码清单8-4为Angle结构添加另一个构造器，向之前声明的带有3个参数的构造器传递default(int)作为实参。表达式default(int)、new int()和0全都生成一样的值。

代码清单8-4 用default操作符来获取类型的默认值

```
// Use keyword struct to declare a value type.
struct Angle
{
  public Angle(int degrees, int minutes)
      : this( degrees, minutes, default(int) )
  {
  }

  // ...
}
```

8.1.3 值类型的继承和接口

所有值类型都隐式密封。除此之外，除了枚举之外的所有值类型都派生自System.ValueType。这意味着结构的继承链总是从object到System.ValueType到结构。

值类型也能实现接口。.NET Framework内建的许多值类型都实现了IComparable和IFormattable等接口。

System.ValueType规定了值类型的行为，但没有包含任何附加成员。System.ValueType重写了object的所有虚成员。在结构中重写基类方法的规则与类基本一样（参见第9章），但一个区别在于，对于值类型，GetHashCode()的默认实现是将调用转发给结构中的第一个非空字段。此外，Equals()大量利用了反射。所以，假如一个值类型在集合中频繁使用，尤其是使用了散列码的字典类型的集合，那么值类型应该同时包含对Equals()和GetHashCode()的重写，以确保能获得较好的性能。详情参见第9章。

规范

如果值类型的相等性有意义，要重载值类型的相等性操作符（Equals()、==和!=）。还要考虑实现IEquatable<T>接口。

8.2 装箱

我们知道，值类型的变量直接包含它们的数据，但引用类型的变量包含对另一个存储位置的引用。那么，当值类型转换成它实现的某个接口或者object时会发生什么？转换的结果必然是对一个存储位置的引用。该变量表面上包含引用类型的实例，但实际上包含值类型的值。这种转换称为**装箱**（boxing）转换，它具有特殊的行为。从值类型的变量（直接引用其数据）转换成引用类型（引用堆上的一个位置）时，会涉及以下几个步骤。

（1）首先在堆上分配好内存。它将用于存放值类型的数据以及少许额外开销（SyncBlockIndex和方法表指针）。这些额外开销使对象看起来像引用类型的托管对象实例。

（2）接着发生一次内存复制动作，当前存储位置的值类型数据复制到堆上分配好的位置。

（3）最后，转换结果是对堆上的新存储位置的引用。

相反的过程称为**拆箱**（unboxing）。拆箱转换先检查已装箱的值的类型，以确保其能兼容于要拆箱成的值的类型，然后复制堆中存储的值。

装箱和拆箱之所以重要，是因为装箱会影响性能和行为。除了学习如何在C#代码中识别它们之外，开发者还应该通过查看CIL，在一个特定的代码片段中统计box/unbox指令的数量。如表8-1所示，每个操作都有对应的指令。

表8-1 CIL中的装箱代码

C# 代 码	CIL 代 码
`static void Main()` `{` `int number;` `object thing;` `number = 42;` `// Boxing` `thing = number;` `// Unboxing` `number = (int)thing;` `return;` `}`	`.method private hidebysig` `static void Main() cil managed` `{` `.entrypoint` `// Code size 21 (0x15)` `.maxstack 1` `.locals init ([0] int32 number,` `[1] object thing)` `IL_0000: nop` `IL_0001: ldc.i4.s 42` `IL_0003: stloc.0` `IL_0004: ldloc.0` `IL_0005: box [mscorlib]System.Int32` `IL_000a: stloc.1` `IL_000b: ldloc.1` `IL_000c: unbox.any [mscorlib]System.Int32` `IL_0011: stloc.0` `IL_0012: br.s IL_0014` `IL_0014: ret` `} // end of method Program::Main`

如果装箱和拆箱进行得不是很频繁，那么性能问题不大。但有的时候，装箱很容易被忽视，而且会非常频繁地发生，这就可能大幅影响性能。代码清单8-5和输出8-1展示了一个例子。ArrayList类型维护的是对象引用列表，所以在列表中添加整数或浮点数会造成对值进行装箱以获取引用。

代码清单8-5 容易忽视的box和unbox指令

```
        class DisplayFibonacci

{
  static void Main()
  {

      int totalCount;
      System.Collections.ArrayList list =
          new System.Collections.ArrayList();
      Console.Write("Enter a number between 2 and 1000:");
      totalCount = int.Parse(Console.ReadLine());

      // Execution-time error:
      // list.Add(0);  // Cast to double or 'D' suffix required
                       // Whether cast or using 'D' suffix,
                       // CIL is identical.
      list.Add((double)0);
      list.Add((double)1);
      for (int count = 2; count < totalCount; count++)
      {
          list.Add(
              ((double)list[count - 1] +
              (double)list[count - 2]) );
      }

      foreach (double count in list)
      {
          Console.Write("{0}, ", count);
      }
  }
}
```

输出8-1

```
Enter a number between 2 and 1000:42
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584,
4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229,
832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817,
39088169, 63245986, 102334155, 165580141,
```

代码清单8-5的代码在编译之后，会在最终的CIL中生成5个box指令和3个unbox指令。

（1）前两个box指令是在对list.Add()的初始调用中发生的。Add方法的签名是int Add(object value)。所以，传给该方法的任何值类型都会被装箱。

（2）接着是在for循环内部Add()调用中的两个unbox指令。ArrayList的索引操作符返回的总是object，因为那正是ArrayList所包含的。但为了将两个值加到一起，需要将它们转换回double。从object的引用向值类型的转型作为unbox调用来实现。

（3）现在要获取加法运算的结果，并把它放到ArrayList实例中，这再次造成了一次box操作。注意，前两个unbox指令和这个box指令是在一次循环中发生的。

（4）foreach遍历ArrayList中的每一项，并把它们的值赋给count。但ArrayList包含的是object的引用，所以每一项赋给double变量时都要执行unbox操作。

（5）对于foreach循环中调用的Console.WriteLine()方法，它的签名是void

Console.Write (string format, object arg)。因此，对它的每个调用都涉及从double到object的box操作。

每个装箱操作都涉及内存分配和复制，每个拆箱操作都涉及类型检查和复制。如果用已拆箱的类型做同样的事情，内存分配和类型检查就可以避免了。显然，可以通过避免许多装箱操作来提升代码的性能。例如在上例的foreach循环中，可以使用object而不是double来改进。另一处改进是将ArrayList数据类型更改为泛型集合（参见第11章）。但这里的重点在于，有的装箱操作很容易被人忽视，开发者要特别留意那些可能反复发生装箱并大幅影响性能的情形。

还有一个问题是运行时的装箱。假如要更改最开始的两个Add()调用，不使用强制类型转换（或者double字面值），就必须在数组列表中插入整数。由于int会隐式转型为double，因此这似乎是一个无关紧要的修改。然而，假如先在for循环中转型为double，再在foreach循环的count赋值过程中转型为double，就会出问题。这是因为在unbox操作之后，紧接着执行从已装箱的int向double的内存复制。这个操作要想成功，首先必须转型为一个int，否则代码会在执行时引发InvalidCastException异常。代码清单8-6展示了注释掉的一个类似的错误，在它后面是正确的转型方式。

代码清单8-6 必须拆箱为基础类型

```
// ...
int number;
object thing;
double bigNumber;

number = 42;
thing = number;
// ERROR: InvalidCastException
// bigNumber = (double)thing;
bigNumber = (double)(int)thing;
// ...
```

高级主题：lock语句中的值类型

C#支持用于同步代码的lock语句。该语句实际会编译成System.Threading.Monitor的Enter()和Exit()方法。两个方法必须成对调用。Enter()记录由其唯一的引用型参数传递的一个lock，这样一来，使用相同的引用调用Exit()的时候，就可以释放该lock。值类型的问题在于装箱。每次调用Enter()或者Exit()时，都会在堆上创建新值。将一个副本的引用同另一个副本的引用进行比较，总是返回false。所以，无法将Enter()与对应的Exit()钩到一起。因此，不允许在lock()语句中使用值类型。

代码清单8-7揭示了更多的运行时装箱问题，输出8-2展示了结果。

代码清单8-7 容易忽视的装箱问题

```
interface IAngle
{
    void MoveTo(int degrees, int minutes, int seconds);
}
```

```
struct Angle : IAngle
{
```

```
    // ...

    // NOTE:  This makes Angle mutable, against the general
    //        guideline
    public void MoveTo(int degrees, int minutes, int seconds)
    {
            _Degrees = degrees;
            _Minutes = minutes;
            _Seconds = seconds;
    }
}
```

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

      Angle angle = new Angle(25, 58, 23);
      object objectAngle = angle;  // Box
      Console.Write( ((Angle)objectAngle).Degrees);

      // Unbox, modify unboxed value, and discard value
      ((Angle)objectAngle).MoveTo(26, 58, 23);
      Console.Write(", " +  ((Angle)objectAngle).Degrees);

      // Box, modify boxed value, and discard reference to box
      ((IAngle)angle).MoveTo(26, 58, 23);
      Console.Write(", " + ((Angle)angle).Degrees);

      // Modify boxed value directly
      ((IAngle)objectAngle).MoveTo(26, 58, 23);
      Console.WriteLine(", " + ((Angle)objectAngle).Degrees);

      // ...
  }
}
```

输出8-2

```
25, 25, 25, 26
```

代码清单8-7使用了来自代码清单8-1的Angle结构和IAngle接口。还要注意，IAngle.MoveTo()使Angle成为可变（mutable）值类型。这带来了一些问题。通过演示这些问题，你将理解使值类型不可变（immutable）的重要性。

前两行初始化angle并把它装箱到objectAngle变量中。接着调用move将Hours更改为26。然而，正如输出演示的那样，这一次不会发生实际的改变。这里的问题是，为了调用MoveTo()，编译器要对objectAngle进行拆箱，并且（根据定义）创建值的副本。值类型根据值进行复制；这正是它们被称为值类型的原因。虽然结果值在执行时被成功修改，但值的这个副本会被丢弃。objectAngle引用的堆位置处没有发生任何改变。

前面说过，值类型的变量就像上面了写了值的纸。对值进行装箱相当于对纸进行复印，并把复印件放到箱子里。对值进行拆箱，相当于对箱子里的纸进行复印。编辑第二份复印件不会影响箱子里的副本。

在第三种情况下，即标记为“Box, modify boxed value, and discard reference to box”的部分，类似的问题在反方向上发生。不是直接调用MoveTo()，而是将值转型为IAngle。转换为接口类型会对值进行装箱，所以“运行时”将angle中的数据复制到堆上，并提供对那个箱子的引用。接着，方法调用会修改被引用的箱子中的值。存储在angle变量中的值保持未修改状态。

在最后一种情况下，向IAngle的转型是引用转换而不是装箱转换。值已通过向object的转换装箱了。所以，这一次转换不会发生值的复制。对MoveTo()的调用将更新箱子中存储的_Hours值，代码的行为符合预期。

从这个例子可以看出，可变的值类型很容易使人迷惑，因为往往修改的是值的副本，而不是真正想要修改的存储位置。如果在第一时间避免使用可变的值类型，就可以消除这种形式的迷惑了。

规范

避免可变的值类型。

高级主题：如何在方法调用期间避免装箱

在接收者是值类型的情况下调用方法，接收者（在方法主体中用this表示）必须是变量而不是值，因为方法可能尝试修改接收者。显然，它必须修改接收者的存储位置，而不是修改接收者值的副本再丢弃该副本。代码清单8-7的第二种和第四种情况演示了在已装箱值类型上调用方法时，这一事实对于性能的影响。

在第二种情况下，拆箱转换逻辑上生成已装箱的值，而不是生成对包含已装箱副本的“堆上存储位置”（箱子）的引用。那么，哪个存储位置作为this传给方法调用呢？不可能是堆上的箱子的存储位置，因为拆箱转换生成的是那个值的副本而不是对存储位置的引用。

在需要值类型的变量但只有一个值可用的情况下，会采取以下两种策略之一。要么是C#编译器生成代码来创建一个新的临时存储位置，并将值从箱子复制到新位置，使临时存储位置成为所需的变量；要么C#编译器不允许该操作并报错。本例采用了第一个策略。新的临时存储位置成为该调用的接收者。MoveTo()方法完成修改后，该临时存储位置被丢弃。

每次进行“拆箱后调用”，不管方法是否真的要修改变量，都会重复下述过程：对已装箱的值进行类型检查，拆箱以生成已装箱的值的存储位置，分配临时变量，将值从箱子复制到临时变量，再调用方法并传递临时存储位置。很明显，如果方法不修改变量，好多工作都是可以避免的。但是，C#编译器不知道一个方法是否要修改接收者，所以它只好“宁肯错杀一千也不肯放过一个”。

在已装箱的值类型上调用接口方法，所有这些开销都可以避免。调用接口方法时，预期接收者是箱子中的存储位置；如果接口方法要修改存储位置，修改的是箱子的位置。这样就避免了执行类型检查、分配新存储位置和生成副本的开销。相反，在调用结构的方法时，“运行时”将箱子中的存储位置作为this传给方法。

代码清单8-8调用int值类型实现的IFormattable接口中的获取2个实参的ToString()方法。在这个例子中，调用的接收者是已装箱的值类型，但在调用接口方法时不会拆箱。

代码清单8-8 避免拆箱和复制

```
int number;
object thing;
number = 42;
// Boxing
thing = number;
// No unboxing conversion.
string text = ((IFormattable)thing).ToString(
    "X", null);
Console.WriteLine(text);
```

你或许会问，如果将值类型的实例作为接收者来调用object声明的虚方法ToString()会发生什么？实例会装箱、拆箱还是做什么？取决于细节，可能有多种不同的情况。

- 如果接收者已拆箱，而且结构重写了ToString()，将直接调用重写的方法。没有必要以虚方式调用，因为方法不能被更深的派生类重写了。所有值类型是自动地密封的。
- 如果接收者已拆箱，而且结构没有重写ToString()，就必须调用基类的实现，该实现预期的接收者是一个对象引用。所以，接收者被装箱。
- 如果接收者已装箱，而且结构重写了ToString()，就将箱子的存储位置传给重写的方法，不对其进行拆箱。
- 如果接收者已装箱，而且结构没有重写ToString()，就将对箱子的引用传给基类的实现，该实现预期的正是一个引用。

8.3 枚举

对比如代码清单8-9所示的两个代码片段。

代码清单8-9 将整数switch与枚举switch进行比较

```
int connectionState;
// ...
switch (connectionState)
{
    case 0:
        // ...
        break;
    case 1:
        // ...
        break;
    case 2:
        // ...
        break;
    case 3:
        // ...
        break;
}
```

```
ConnectionState connectionState;
// ...
switch (connectionState)
{
```

```
        case ConnectionState.Connected:
            // ...
            break;
        case ConnectionState.Connecting:
            // ...
            break;
        case ConnectionState.Disconnected:
            // ...
            break;
        case ConnectionState.Disconnecting:
            // ...
            break;
    }
```

显然，两者在可读性上区别巨大。在第二个代码片段中，各个case让人一目了然。不过，两者在运行时的性能完全一样，因为第二个代码片段在每个case中使用了**枚举值**（enum value）。

枚举是可由开发者声明的值类型。枚举的关键特征是在编译时声明了一组可以通过名称来引用的常量值，这使代码更易读。代码清单8-10是一个典型的枚举声明语法。

代码清单8-10 定义枚举

```
enum ConnectionState

{
  Disconnected,
  Connecting,
  Connected,
  Disconnecting
}
```

注意

用枚举替代布尔值能改善可读性。例如，形如 SetState(DeviceState.On)的方法调用的可读性要好于SetState(true)。

要使用枚举值需要为其附加枚举名称前缀。例如，使用Connected值的语法是ConnectionState.Connected。枚举值名称不要包含枚举名称，避免出现像ConnectionState.ConnectionStateConnected这样啰嗦的写法。根据约定，除了位标志（将于8.3.3节讨论）之外的枚举名称应该使用单数形式。例如，应该使用ConnectionState而不是ConnectionStates。

枚举值实际是作为整数常量实现的。默认第一个枚举值是0，后续每一项都递增1。然而，可以显式地为枚举赋值，如代码清单8-11所示。

代码清单8-11 定义枚举类型

```
enum ConnectionState : short
{
  Disconnected,
  Connecting = 10,
  Connected,
  Joined = Connected,
```

```
    Disconnecting
}
```

Disconnected仍然具有默认值0，Connecting则被显式赋值为10，所以它后面的Connected会被赋值为11。随后，Joined也被赋值为11，也就是赋给Connected的值（将Connected引用的值赋给Joined时，不需要为Connected附加枚举名称前缀，因为目前正处在枚举的作用域内）。最后，Disconnecting自动递增1，所以值是12。

枚举总是具有一个基础类型，这可以是除了char之外的任意整型。事实上，枚举类型的性能完全取决于基础类型的性能。默认基础类型是int，但可以使用继承语法指定其他类型。例如在代码清单8-11中，使用的不是int而是short。为了保持一致性，这里的语法模拟了继承的语法，但并没有真正建立继承关系。所有枚举的基类都是System.Enum，后者从System.ValueType派生。除此之外，这些类都是密封的，不能从现有枚举类型派生以添加额外成员。

规范

考虑使用默认的32位整型作为枚举的基础类型。只有出于互操作性或者性能方面的考虑才使用较小的类型，只有创建标志（flag）数超过32个的标志枚举时才使用较大的类型。

枚举只不过是基础类型上的一组名称，对于枚举类型的变量，它的值并不限于声明中命名的值。例如，整数42能转型为short，也能转型为ConnectionState类型，即使没有对应的ConnectionState枚举值。只要值能转换成基础类型，就能转换成枚举类型。

在没有对应的枚举值的前提下也允许转型，这个设计的优点在于可在未来的API版本中为枚举添加新值，同时不会破坏早期的版本。除此之外，枚举值为已知的值提供了名称，同时仍然允许在运行时分配未知的值。这个设计的麻烦之处在于，开发者在编码时必须谨慎，要主动考虑到未命名值的可能性。例如，假如将case ConnectionState.Disconnecting替换成default，并认定default case唯一可能的值就是ConnectionState.Disconnecting，那么就是不明智的。相反，应显式处理Disconnecting这一case，而让default case报告一个错误，或者执行其他无害的行为。然而，正如前面讲到的，从枚举转换为基础类型以及从基础类型转换为枚举类型都涉及显式转型，而不是隐式转型。例如，假定方法签名是void ReportState(ConnectionState state)，就不能调用ReportState(10)。唯一的例外是传递0值，因为0能隐式转换为任何枚举。

虽然允许在代码将来的版本中为枚举添加额外的值，但在这样做时要小心。在枚举中部插入枚举值，会使其后的所有枚举值发生顺移。（例如，在Connected之前添加Flooded或Locked值，会造成Connected值改变）。这会影响根据新枚举来重新编译的所有版本。不过，任何基于旧枚举编译的代码都会继续使用旧值。除了在列表末尾插入新的枚举值，避免旧枚举值改变的另一个办法是显式赋值。

规范

考虑在现有枚举中添加新成员，但要注意兼容性风险。

避免创建代表“不完整”值（如版本号）集合的枚举。

避免在枚举中创建“保留给将来使用”的值。

避免包含单个值的枚举。要为简单枚举提供0值（代表无）。注意若不显式地进行初始化，就默认从0开始。

枚举和其他值类型稍有不同，因为枚举的继承链是从System.ValueType到System.Enum，再到枚举。

8.3.1 枚举之间的类型兼容性

C#不支持不同枚举数组之间的直接转型。但CLR允许，前提是两个枚举具有相同的基础类型。为了避开C#的限制，一个可用的技巧是先转型为System.Array，如代码清单8-12末尾所示。

代码清单8-12 枚举数组之间的转型

```
enum ConnectionState1
{
  Disconnected,
  Connecting,
  Connected,
  Disconnecting
}
```

```
enum ConnectionState2
{
  Disconnected,
  Connecting,
  Connected,
  Disconnecting
}
```

```
class Program
{
  static void Main()
  {
      ConnectionState1[] states =
          (ConnectionState1[])(Array)new ConnectionState2[42];
  }
}
```

这个技巧利用了CLR的赋值兼容性比C#宽松这一事实。（还可用同样的技巧进行非法转换，如int[]转换成uint[]。）但使用这个技巧时务必慎重，因为C#规范没有说这个技巧在不同CLR实现中都能正常发挥作用。

8.3.2 枚举和字符串之间的转换

枚举的一个好处是ToString()方法（该方法由System.Console.WriteLine()这样的方法调用）会输出枚举值标识符：

```
System.Diagnostics.Trace.WriteLine(string.Format(
    "The connection is currently {0}.",
    ConnectionState.Disconnecting));
```

上述代码会将输出8-3的文本写入跟踪缓冲区（trace buffer）。

输出8-3

```
The connection is currently Disconnecting.
```

从字符串向枚举的转换刚开始比较难掌握，因为它涉及由System.Enum基类提供的一个静态方法。代码清单8-13展示了在不利用泛型（参见第11章）的前提下如何转换的例子，输出8-4展示了结果。

代码清单8-13 使用Enum.Parse()将字符串转换为枚举

```
ThreadPriorityLevel priority = (ThreadPriorityLevel)Enum.Parse(
  typeof(ThreadPriorityLevel), "Idle");
Console.WriteLine(priority);
```

输出8-4

```
Idle
```

Enum.Parse()的第一个参数是类型，用关键字typeof()指定。这是在编译时判断类型的一种方式，可把它看成类型值的字面值（参见第17章）。

.NET Framework 4之前，Enum没有TryParse()方法，所以之前的代码应包含恰当的异常处理机制，以防范字符串和枚举值标识符不匹配的情况。.NET Framework 4的TryParse<T>()方法使用了泛型，但类型参数可以推断出来。所以，可以像代码清单8-14那样转换。

代码清单8-14 使用Enum.TryParse<T>()将字符串转换成枚举

```
System.Diagnostics.ThreadPriorityLevel priority;
if(Enum.TryParse("Idle", out priority))
{
  Console.WriteLine(priority);
}
```

这个技术的优点在于，不必使用异常处理机制来防范转换不成功的情况。相反，只需检查TryParse<T>()返回的布尔结果。

不管是用“Parse”还是“TryParse”模式来编码，从字符串向枚举的转换都是不可本地化的。所以，在本地化是一项硬性要求的前提下，只有对那些不公开给用户的消息，开发人员才可进行这种形式的转型。

规范

如果字符串必须本地化成用户语言，**避免**枚举和字符串之间的直接转换。

8.3.3 枚举作为标志使用

许多时候，开发者不仅希望枚举值是独一无二的，还希望能对其进行组合以表示复合值。以System.IO.FileAttributes为例，如代码清单8-15所示，这个枚举用于表示文件的各种属性：只读、隐藏、存档等。在前面定义的ConnectionState枚举中，每个值都是互斥的。而FileAttributes枚举值则不同，它们能自由组合。例如，一个文件可能同时只读和隐藏。为此，每个枚举值都必须是不重复的一位。

代码清单8-15 枚举作为标志使用

```
[Flags] public enum FileAttributes
{
  None =              0,        // 0000000000000000
  ReadOnly =          1<<0,     // 0000000000000001
  Hidden =            1<<1,     // 0000000000000010
  System =            1<<2,     // 0000000000000100
  Directory =         1<<4,     // 0000000000010000
  Archive =           1<<5,     // 0000000000100000
  Device =            1<<6,     // 0000000001000000
  Normal =            1<<7,     // 0000000010000000
  Temporary =         1<<8,     // 0000000100000000
  SparseFile =        1<<9,     // 0000001000000000
  ReparsePoint =      1<<10,    // 0000010000000000
  Compressed =        1<<11,    // 0000100000000000
  Offline =           1<<12,    // 0001000000000000
  NotContentIndexed = 1<<13,    // 0010000000000000
  Encrypted =         1<<14,    // 0100000000000000
}
```

注意，位标志枚举名称通常是复数，因为它的值代表一组标志。

使用按位OR操作符联结枚举值，使用按位AND操作符测试特定位是否存在，如代码清单8-16所示。

代码清单8-16 为标志枚举值使用按位OR和AND操作符

```
using System;
using System.IO;

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

string fileName = @"enumtest.txt";

System.IO.FileInfo file =
    new System.IO.FileInfo(fileName);

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

Console.WriteLine("{0} | {1} = {2}",
    FileAttributes.Hidden, FileAttributes.ReadOnly,
    (int)file.Attributes);

if ( (file.Attributes & FileAttributes.Hidden) !=
```

```
    FileAttributes.Hidden)
{
    throw new Exception("File is not hidden.");
}

if (( file.Attributes & FileAttributes.ReadOnly) !=
    FileAttributes.ReadOnly)
{
    throw new Exception("File is not read-only.");
}

// ...
}
```

输出8-5展示了代码清单8-16的结果。

输出8–5

```
Hidden | ReadOnly = 3
```

可用按位OR操作符将文件同时设为只读和隐藏。此外，可用按位AND操作符核实特定设置。

枚举中的每个值不一定只对应一个标志。完全可以为常用的标志组合定义额外的枚举值。代码清单8-17展示了一个例子。

代码清单8–17　为常用组合定义枚举值

```
[Flags] enum DistributedChannel
{
  None = 0,
  Transacted = 1,
  Queued = 2,
  Encrypted = 4,
  Persisted = 16,
  FaultTolerant =
      Transacted | Queued | Persisted
}
```

一个好习惯是在标志枚举中包含值为0的None成员，因为无论是枚举类型的字段，还是枚举类型的数组的元素，其初始默认值都是0。避免让枚举值对应像Maximum这样的东西，因为Maximum可能被解释成一个有效的枚举值。要检查枚举是否包含某个值，请使用System.Enum.IsDefined()方法。

规范

要用FlagsAttribute指出枚举包含标志。

要为所有标志枚举提供等于0的None值。

避免标志枚举中的零值是除了“所有标志都未设置”之外的其他意思。

考虑为常用标志组合提供特殊值。

不要包含“哨兵”值（如Maximum），这种值会使用户感到困惑。

要用2的幂确保所有标志组合都不重复。

高级主题：FlagsAttribute

如果决定使用位标志枚举，枚举的声明应该用FlagsAttribute进行标记。这个特性应包含在一对方括号中（具体参见第17章），并放在枚举声明之前，如代码清单8-18所示。

代码清单8-18 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.Open(FileMode.Create).Close();

      FileAttributes startingAttributes =
          file.Attributes;

      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);

      File.SetAttributes(fileName,
          startingAttributes);
      file.Delete();
  }
}
```

输出8-6展示了代码清单8-18的结果。

输出8-6

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

这个特性指出枚举值可以组合。此外，它改变了ToString()和Parse()方法的行为。例如，为已经用FlagsAttribute修饰的枚举调用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8-18中，file.Attributes.ToString()返回ReadOnly, Hidden。而如果没有用FlagsAttributes修饰，返回的就是3。如果两个枚举值相同，ToString()返回第一个值。但如前所述，使用需谨慎，因为这样转换得到的文本无法本地化。

将值从字符串解析成枚举也是可行的。每个枚举值标识符都以逗号分隔。

注意，FlagsAttribute不会自动分配唯一的标志值，也不会核实它们有唯一的值。那样做也没有意义，因为经常都需要重复值和组合值。相反，应显式分配每个枚举项的值。

8.4 小结

本章首先讨论如何创建自定义值类型。由于修改值类型时很容易写出令人困惑或者含有bug的代码，而且由于值类型一般用于建模不可变的值，所以最好使值类型不可变。还讨论了如何对值类型进行“装箱”，作为引用类型以多态的形式对待。

装箱具有一些容易被人忽视的特性，其中大多数都会造成在执行时（而不是编译时）出问题。虽然需要了解这些特性以避免问题，但有时过于关注潜在的缺陷会遮盖掉值类型在有用性和性能上的优势。程序员不要过度地关注使用值类型。值类型渗透到本书的几乎每一章，容易出问题的时候并不多。虽然罗列了值类型可能发生的问题，并用代码进行了演示，但现实中其实很少遇到相同的情形。记住“不要创建可变的值类型”这一准则，就可以避免其中的多数问题。这正是你在内建值类型中没有遇到这些问题的原因。

或许最容易出的问题就是循环内的反复装箱操作。不过，使用泛型显著地减少了装箱。而且即使没有泛型，该问题对性能造成的影响也是微乎其微的，除非你确定某个特定的算法因为装箱而成为性能瓶颈。

另外，自定义值类型（struct）平时用得比较少。虽然它们在C# 开发中扮演了重要角色，但相较于自定义类的数量，自定义结构的数量还是非常少的——只有在需要与托管代码进行互操作的时候，才需要大量地用到自定义结构。

规范

不要定义结构，除非它在逻辑上代表单个值，消耗16字节或更少的存储空间、不可变，而且很少装箱。

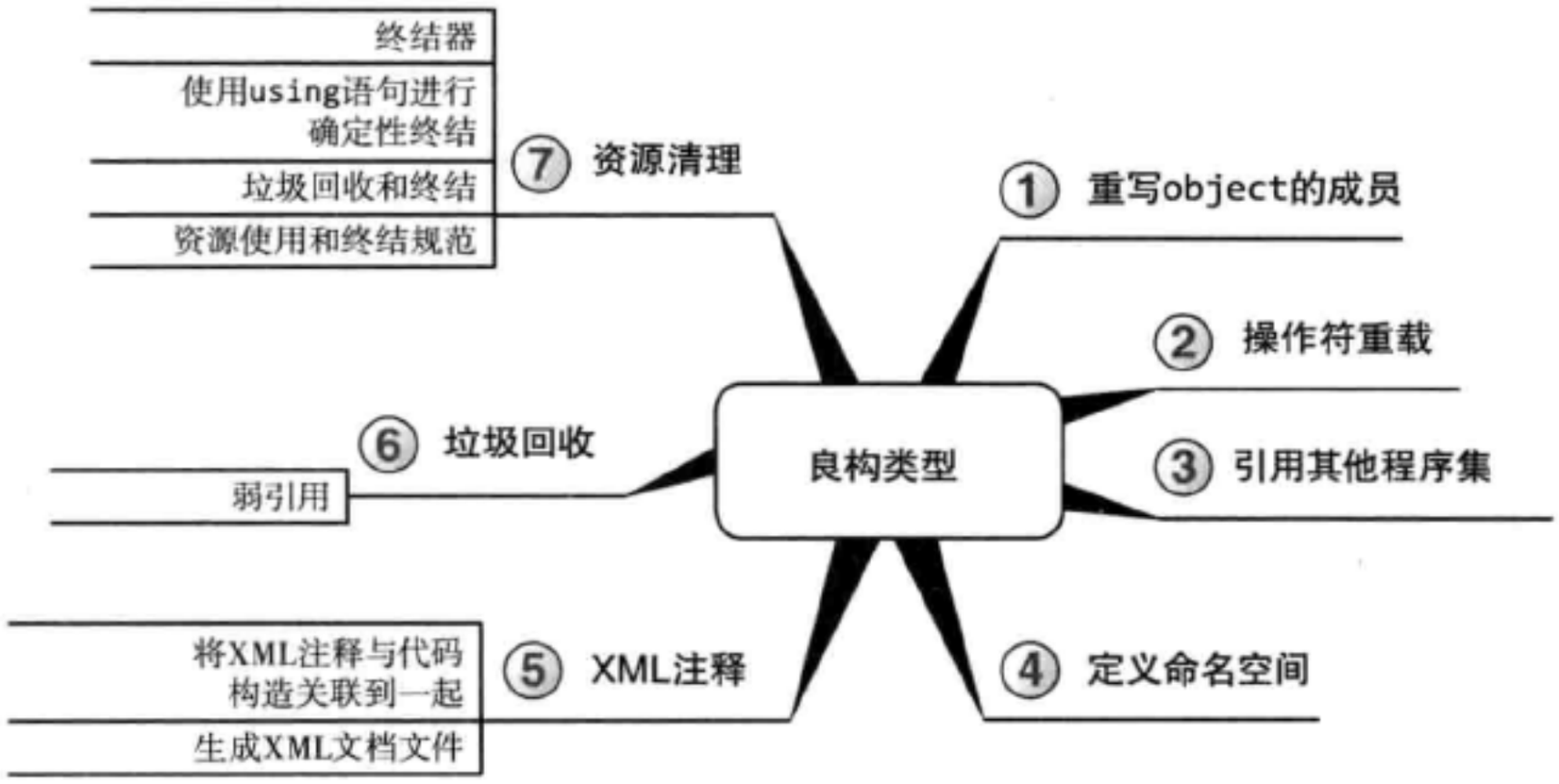
本章还介绍了枚举。枚举类型是大多数编程语言都有的标准构造。在改进API可用性与代码可读性方面，枚举功不可没。

下一章将提供更多的规范帮助你创建“良构”（well-formed）的值类型和引用类型。首先讨论如何重写对象的虚成员，以及如何定义操作符重载方法。这两个主题同时适用于结构和类，但它们更重要的意义在于完善结构定义，使其做到“良构”。

第9章

良构类型[①]

前面讲述了用于定义类和结构的大多数构造。然而，目前的类型并不是最完美的，还缺少一些合用且完备的功能。本章要描述如何完善类型声明。



9.1 重写object的成员

第6章说过，object是终极基类，所有类和结构都从它派生。此外，还讨论了object提供的每一个方法，并提到其中一些是虚方法。本节将讨论对虚方法进行重写的细节。

9.1.1 重写ToString()

默认情况下，在任何对象上调用ToString()会返回类的完全限定名称。例如，在一个System.IO.FileStream对象上调用ToString()，会返回字符串System.IO.FileStream。

---

① “良构”的原文是“well-formed”，也有翻译为“格式良好”或者“格式正确”，或者“合式”。——译者注

但对某些类来说，ToString()应返回更有意义的结果。以string类为例，ToString()应返回字符串值本身。类似地，返回一个Contact的姓名显然更有意义。代码清单9-1重写了ToString()，返回Coordinate（坐标）的字符串表示。

代码清单9-1 重写ToString()

```
public struct Coordinate

{
  public Coordinate(Longitude longitude, Latitude latitude)
  {
      _Longitude = longitude;
      _Latitude = latitude;
  }

  public Longitude Longitude { get { return _Longitude; } }
  private readonly Longitude _Longitude;

  public Latitude Latitude { get { return _Latitude; } }
  private readonly Latitude _Latitude;

  public override string ToString()
  {
      return string.Format("{0} {1}", Longitude, Latitude);
  }

  // ...
}
```

Console.WriteLine()和System.Diagnostics.Trace.Write()等方法会调用对象的ToString()方法，所以可重写ToString()输出比默认实现更有意义的信息。总之，如果能输出更能说明问题的诊断信息（特别是当目标用户是开发者的时候），就要考虑重写ToString()方法，因为object.ToString()默认只是输出类型名称，对用户不太友好。在IDE中调试时，或者向日志文件写入时，ToString()相当有用。考虑到这个原因，字符串不要太长（最好在一屏之内），以免被截断。不过，由于缺乏本地化和其他高级格式化功能，所以它不太适合一般性的用户文本显示。

规范

要重写ToString()以返回有用的、面向开发人员的诊断字符串。

要使ToString()返回的字符串简短。

不要从ToString()返回空字符串代表“空”（null）。

避免ToString()引发异常或造成可观察到的副作用（改变对象状态）。

如果返回值与语言文化相关或需要格式化（如DateTime），就**要**重载ToString(string format)或实现IFormattable。

考虑从ToString()返回独一无二的字符串以标识对象实例。

9.1.2 重写GetHashCode()

重写GetHashCode()远比重写ToString()复杂。但无论如何，重写Equals()就要重

写GetHashCode()。不这样做编译器会显示警告。将类作为散列表集合（比如System.Collections.Hashtable和System.Collections.Generic.Dictionary）的键（key）使用时，也应重写GetHashCode()。

散列码（hash code）的作用是生成与对象值对应的数字，从而高效地平衡散列表[①]。要获得良好的GetHashCode()实现，请参照以下实现原则（“必须”是指必须满足的要求，“性能”是指为了增强性能而需要采取的措施，“安全性”是指为了保障安全性而需要采取的措施）。

- 必须：相等的对象必然有相等的散列码（若a.Equals(b)，则a.GetHashCode() == b.GetHashCode()）。
- 必须：在特定对象的生存期内，GetHashCode()始终返回相同的值，即使对象的数据发生了改变。许多时候应缓存方法的返回值，从而确保这一点。
- 必须：GetHashCode()不应引发任何异常，GetHashCode()总是成功返回一个值。
- 性能：散列码应尽可能唯一。但由于散列码只是返回一个int，所以只要一种对象包含的值比一个int能够容纳得多，那么散列码肯定存在重复。一个很容易想到的例子是long，因为long的取值范围大于int，所以假如规定每个int值都只能标识一个不同的long值，那么肯定会剩下大量long值无法标识。
- 性能：可能的散列码值应当在int的范围内平均分布。例如，在创建散列码的时候，假如没有考虑到字符串在拉丁语言中的分布主要集中在初始的128个ASCII字符上，就会造成字符串值的分布非常不平均，所以不能算是可靠的GetHashCode()算法。
- 性能：GetHashCode()的性能应该优化。GetHashCode()通常在Equals()实现中用于“短路”一次完整的相等性比较（散列码都不同，当然没必要进行完整的相等性比较了）。所以，当类型作为字典集合中的键类型使用时，会频繁地调用这个方法。
- 性能：两个对象的细微差异应造成散列码值的极大差异。理想情况下，1位的差异应造成散列码平均16位的差异。这有助于确保不管散列表如何对散列值进行“装桶”（bucketing），也能保持良好的平衡性。
- 安全性：攻击者应该难以伪造具有特定散列码的对象。攻击的手法是向散列表中填写大量散列成同一个值的数据。如果散列表的实现不高效，就易于受到DOS（拒绝服务）攻击。

当然，许多规范和原则是相互对立的。很难有一种散列算法既快捷又能满足所有这些要求。和任何设计问题一样，好的解决方案必然兼顾到了各方面的利益。

代码清单9-2展示了如何为Coordinate类型实现GetHashCode()。

代码清单9-2 实现GetHashCode()

```
public struct Coordinate

{
  public Coordinate(Longitude longitude, Latitude latitude)
  {
      _Longitude = longitude;
      _Latitude = latitude;
  }
```

① 也就是要提供良好的随机分布，使散列表获得最佳性能。——译者注

```
    public Longitude Longitude { get { return _Longitude; } }
    private readonly Longitude _Longitude;

    public Latitude Latitude { get { return _Latitude; } }
    private readonly Latitude _Latitude;

    public override int GetHashCode()
    {
        int hashCode = Longitude.GetHashCode();
        // As long as the hash codes are not equal
        if(Longitude.GetHashCode() != Latitude.GetHashCode())
        {
            hashCode ^= Latitude.GetHashCode();  // eXclusive OR
        }
        return hashCode;
    }

    // ...
}
```

通常采取的做法是向相关类型的散列码应用XOR操作符，并确保XOR的操作数不相近或相等，否则结果会全为零。在操作数相近或相等的情况下，考虑改为使用移位和加法操作。其他备选的操作符——AND和OR——具有类似的限制，但这些限制会发生得更加频繁。多次应用AND，会逐渐变成全为0；而多次应用OR，会逐渐变成全为1。

为了进行更细致的控制，应该使用移位操作符来分解比int大的类型。例如，假定有一个名为value的long类型，它的GetHashCode()方法可以像下面这样实现。

```
int GetHashCode() { return ((int)value ^ (int)(value >> 32)) };
```

另外，假如基类不是object，应该在XOR赋值中包含base.GetHashCode()。

最后，Coordinate没有对散列码值进行缓存。执行散列码计算时，每个字段都是只读的，所以值不会改变。然而，假如计算得到的值可能改变，或者在将值缓存之后能显著优化性能，就应该对散列码进行缓存。

9.1.3 重写Equals()

重写Equals()而不重写GetHashCode()，会出现如输出9-1所示的一个警告。

输出9-1

```
warning CS0659: '<Class Name>' overrides Object.Equals(object o) but
does not override Object.GetHashCode()
```

一些程序员认为重写Equals()是再简单不过的事情。但事实上，其中存在着大量容易被人忽视的细节，必须通盘考虑和测试。

1. “对象同一性”和“相等的对象值”

两个引用假如引用同一个实例，就说这两个引用是同一的。object（因而延展到所有派生类型）包含名为ReferenceEquals()的静态方法，它能显式地检查对象同一性，如图9-1所示。

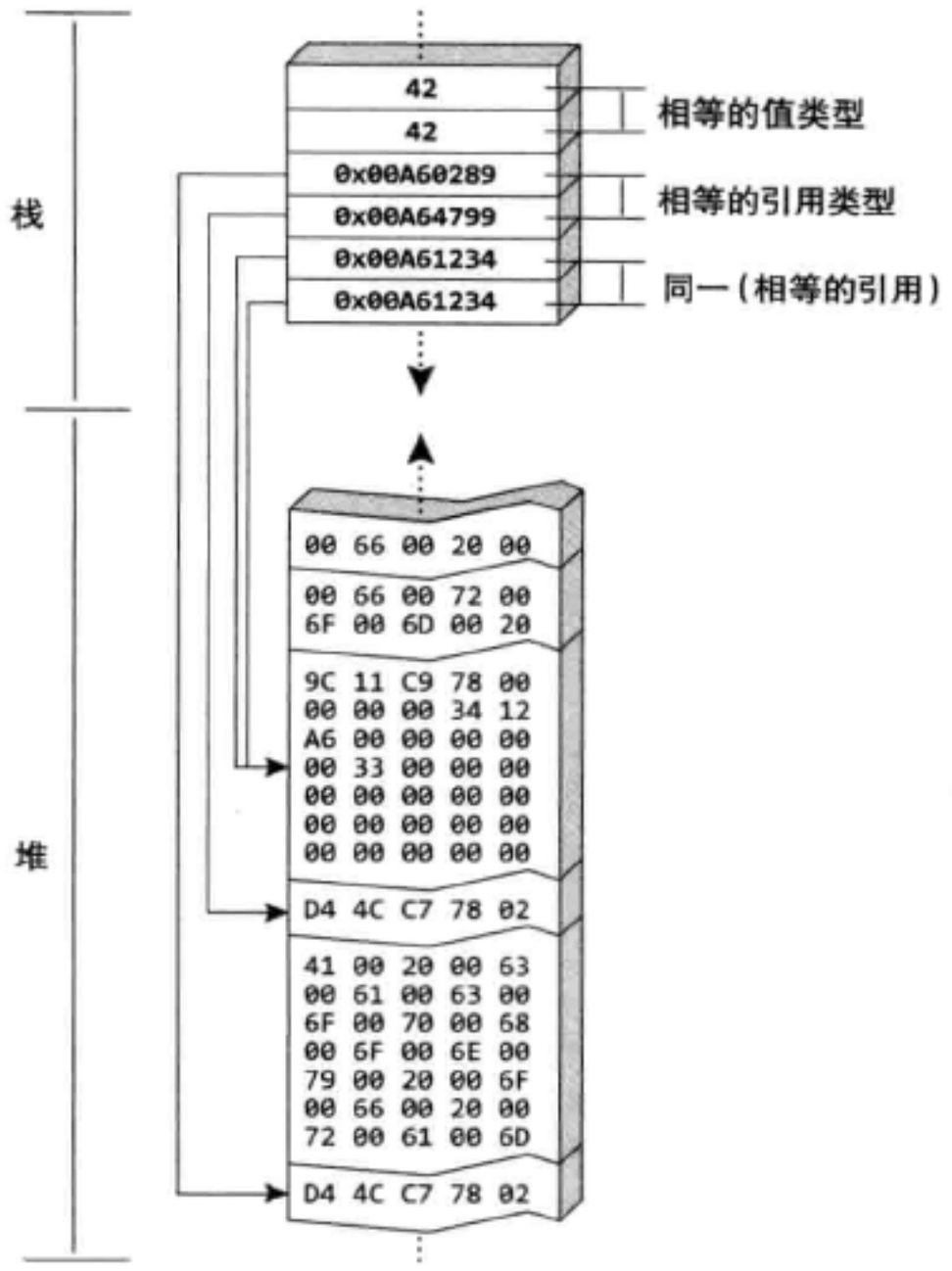


图9-1 同一性

然而，引用的相等性并不是唯一“相等性”。两个对象实例的成员值部分或全部相等，也可以说它们相等。来看代码清单9-3中对两个ProductSerialNumber的比较。

代码清单9-3 重写相等性操作符

```
public sealed class ProductSerialNumber
{
    public ProductSerialNumber(
        string productSeries, int model, long id)
    {
        ProductSeries = productSeries;
        Model = model;
        Id = id;
    }

    public readonly string ProductSeries;
    public readonly int Model;
    public readonly long Id;

    public override int GetHashCode()
    {
        int hashCode = ProductSeries.GetHashCode();
        hashCode ^= Model;  // Xor (eXclusive OR)
        hashCode ^= Id.GetHashCode();  // Xor (eXclusive OR)
        return hashCode;
    }

    public override bool Equals(object obj)
    {
        if (obj == null)
        {
            return false;
```

```
        }
        if (ReferenceEquals(this, obj))
        {
            return true;
        }
        if (this.GetType() != obj.GetType())
        {
            return false;
        }
        return Equals((ProductSerialNumber)obj);
    }

    public bool Equals(ProductSerialNumber obj)
    {
        // STEP 3: Possibly check for equivalent hash codes
        // if (this.GetHashCode() != obj.GetHashCode())
        // {
        //    return false;
        // }

        // STEP 4: Check base.Equals if base overrides Equals()
        // System.Diagnostics.Debug.Assert(
        //     base.GetType() != typeof(object));
        // if ( base.Equals(obj) )
        // {
        //    return false;
        // }

        // STEP 1: Check for null
        return ((obj != null)
            // STEP 5: Compare identifying fields for equality.
            && (ProductSeries == obj.ProductSeries) &&
            (Model == obj.Model) &&
            (Id == obj.Id));
    }

    public static bool operator ==(
        ProductSerialNumber leftHandSide,
        ProductSerialNumber rightHandSide)
    {

        // Check if leftHandSide is null.
        // (operator== would be recursive)
        if (ReferenceEquals(leftHandSide, null))
        {
            // Return true if rightHandSide is also null
            // but false otherwise.
            return ReferenceEquals(rightHandSide, null);
        }

        return (leftHandSide.Equals(rightHandSide));
    }

    public static bool operator !=(
        ProductSerialNumber leftHandSide,
        ProductSerialNumber rightHandSide)
    {
        return !(leftHandSide == rightHandSide);
    }
}
```

```
class Program
{
  static void Main()
  {
      ProductSerialNumber serialNumber1 =
          new ProductSerialNumber("PV", 1000, 09187234);
      ProductSerialNumber serialNumber2 = serialNumber1;
      ProductSerialNumber serialNumber3 =
          new ProductSerialNumber("PV", 1000, 09187234);

      // These serial numbers ARE the same object identity.
      if(!ProductSerialNumber.ReferenceEquals(serialNumber1,
          serialNumber2))
      {
          throw new Exception(
              "serialNumber1 does NOT " +
              "reference equal serialNumber2");
      }
      // and, therefore, they are equal
      else if(!serialNumber1.Equals(serialNumber2))
      {
          throw new Exception(
              "serialNumber1 does NOT equal serialNumber2");
      }
      else
      {
          Console.WriteLine(
              "serialNumber1 reference equals serialNumber2");
          Console.WriteLine(
              "serialNumber1 equals serialNumber2");
      }


      // These serial numbers are NOT the same object identity.
      if (ProductSerialNumber.ReferenceEquals(serialNumber1,
              serialNumber3))
      {
          throw new Exception(
              "serialNumber1 DOES reference " +
              "equal serialNumber3");
      }
      // but they are equal (assuming Equals is overloaded).
      else if(!serialNumber1.Equals(serialNumber3) ||
          serialNumber1 != serialNumber3)
      {
          throw new Exception(
              "serialNumber1 does NOT equal serialNumber3");
          }

          Console.WriteLine( "serialNumber1 equals serialNumber3" );
      }
  }
```

代码清单9-3的结果如输出9-2所示。

输出9-2

```
serialNumber1 reference equals serialNumber2
serialNumber1 equals serialNumber3
```

正如最后一个ReferenceEquals()断言所演示的，serialNumber1和serialNumber3不是相同的引用。然而，两者使用相同的值来构造，而且逻辑上与同一个物理产品关联。假如一个实例基于数据库中的数据创建，另一个实例基于人工输入的数据创建，两个实例就应该相等，从而确保产品不会在数据库中重复（被重复录入）。两个同一的引用显然是相等的，然而，两个引用不相等的对象也可能是相等的对象。对象标识不同，不一定标识数据不同。

只有引用类型才可能引用相等，因此提供了对同一性概念的支持。为值类型调用ReferenceEquals()总是返回false，因为值类型转换成object时要装箱。即使向ReferenceEquals()的两个参数传递同一个变量，结果也是false，因为两个值会独立地装箱。代码清单9-4演示了这个行为。每个实参都“装到不同的箱子”中，所以永远不会引用相等。

> **注意**
>
> 为值类型调用ReferenceEquals()将总是返回false。

代码清单9-4 值类型本身不可能引用相等

```
public struct Coordinate
{
  public Coordinate(Longitude longitude, Latitude latitude)
  {
      _Longitude = longitude;
      _Latitude = latitude;
  }

  public Longitude Longitude { get { return _Longitude; } }
  private readonly Longitude _Longitude;

  public Latitude Latitude { get { return _Latitude; } }
  private readonly Latitude _Latitude;

  // ...
}
```

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

Coordinate coordinate1 =
    new Coordinate( new Longitude(48, 52),
                    new Latitude(-2, -20));

// Value types will never be reference equal.
if ( Coordinate.ReferenceEquals(coordinate1,
    coordinate1) )
{
    throw new Exception(
        "coordinate1 reference equals coordinate1");
}
```

```
        Console.WriteLine(
            "coordinate1 does NOT reference equal itself" );
    }
}
```

第8章将Coordinate定义成引用类型。本例将它定义成值类型（struct），因为经纬度数据的组合在逻辑上被认为是一个值，而且大小不超过16字节（第8章的Coordinate聚合了Angle，而不是Longitude和Latitude）。将Coordinate声明为值类型的另一个理由在于，它是支持特定运算的复数（complex number）。相反，像Employee这样的引用类型不是一个能以数值方式操作的值，而是对实际存在的一个对象的引用。

2. 实现Equals()

判断两个对象是否相等（包含相同的标识数据）是使用对象的Equal()方法。在object中，这个虚方法只是用ReferenceEquals()判断相等性。这显然并不充分，所以一般都有必要用更恰当的实现重写Equals()。

> **注意**
>
> object.Equals()的实现只是简单地调用了一下ReferenceEquals()。

两个对象要相等，它们的标识数据（identifying data）必须相等。例如，对于ProductSerialNumber对象，ProductSeries、Model和Id必须相同。然而，对于Employee对象，可能比较EmployeeId就足够了。只有重写才能解决object.Equals()实现不充分的问题。例如，值类型就重写了Equals()实现，使用类型包含的字段进行比较。

重写Equals()的步骤如下。

（1）检查是否为null。

（2）如果是引用类型，就检查引用是否相等。

（3）检查数据类型是否相同。

（4）一个指定了具体类型的辅助方法，它能将操作数视为要比较的类型，而不是将其视为对象（参见代码清单9-5中的Equals(Coordinate obj)方法）。

（5）可能要检查散列码是否相等。如果散列码都不相等，就没必要继续执行一次全面的、逐个字段的比较。（相等的两个对象不可能散列码不同。）

（6）如果基类重写了Equals()，就检查base.Equals()。

（7）比较每一个标识字段，判断是否相等。

（8）重写GetHashCode()。

（9）重写==和!=操作符（参见下一节）。

代码清单9-5展示了Equals()的一个示例实现。

代码清单9-5 重写Equals()

```
public struct Longitude
{
  // ...
}
```

```
public struct Latitude
{
  // ...
}
```

```
public struct Coordinate
{
  public Coordinate(Longitude longitude, Latitude latitude)
  {
      _Longitude = longitude;
      _Latitude = latitude;
  }

public Longitude Longitude { get { return _Longitude; } }
private readonly Longitude _Longitude;

public Latitude Latitude { get { return _Latitude; } }
private readonly Latitude _Latitude;

public override bool Equals(object obj)
{
    // STEP 1: Check for null
    if (obj == null)
    {
        return false;
    }
    // STEP 3: equivalent data types
    // Can be avoided if type is sealed.
    if (this.GetType() != obj.GetType())
    {
        return false;
    }
    return Equals((Coordinate)obj);
}
public bool Equals(Coordinate obj)
{
    // STEP 1: Check for null if a reference type
    // (e.g., a reference type)
    // if (obj == null)
    // {
    //     return false;
    // }

    // STEP 2: Check for ReferenceEquals if this
    // is a reference type
    // if ( ReferenceEquals(this, obj))
    // {
    //   return true;
    // }

// STEP 4: Possibly check for equivalent hash codes
// if (this.GetHashCode() != obj.GetHashCode())
// {
//    return false;
// }

// STEP 5: Check base.Equals if base overrides Equals()
// System.Diagnostics.Debug.Assert(
//     base.GetType() != typeof(object) );
// if ( !base.Equals(obj) )
```

```
        // {
        //     return false;
        // }

        // STEP 6: Compare identifying fields for equality
        //         using an overload of Equals on Longitude.
        return ( (Longitude.Equals(obj.Longitude)) &&
            (Latitude.Equals(obj.Latitude)) );
    }

    // STEP 7: Override GetHashCode.
    public override int GetHashCode()
    {
        int hashCode = Longitude.GetHashCode();
        hashCode ^= Latitude.GetHashCode(); // Xor (eXclusive OR)
        return hashCode;
    }

}
```

在这个实现中，前两个检查肯定要进行。但注意如果类型是密封的，那么步骤3可以省略。

步骤4～6在Equals()的重载版本中进行，该版本获取Coordinate数据类型的对象作为参数。这样在比较两个Coordinate对象时，就可以完全避免执行Equals(object obj)及其GetType()检查。

由于GetHashCode()没有缓存，而且效率比不上步骤5，因此GetHashCode()比较代码被设为注释，没有被实际执行。类似地，也没有使用base.Equals()，因为基类没有重写Equals()（断言中检查base是不是object类型，但没有检查基类是否重写了Equals()。而调用base.Equals()要求基类重写Equals()）。总之，由于GetHashCode()并非一定返回“独一无二”的值（它只能表明操作数不同），所以不能仅仅依赖它来判断两个对象是否相等。

类似于GetHashCode()，Equals()永远不应引发任何异常。对象相互之间应该能够随意比较，这样做永远都不应该造成异常。

规范

要一起实现GetHashCode()、Equals()、==操作符和!=操作符，缺一不可。

要用相同的算法实现Equals()、==和!=。

避免在GetHashCode()、Equals()、==和!=的实现中引发异常。

避免重载可变的引用类型的相等性操作符（如果重载的实现速度过慢，也不要重载）。

要在实现IComparable时，实现与相等性有关的所有方法。

9.2 操作符重载

上一节讨论了如何重写Equals()，并指出类还应实现==和!=。实现操作符的过程称为**操作符重载**。本节的内容不只适用于==和!=操作符，还适用于C#支持的其他操作符。

例如，string支持用“+”操作符拼接两个字符串。这或许并不奇怪，string毕竟是预定义类型，它可能获得了特殊的编译器支持。然而，C#允许为任何类或结构添加“+”操作符支持。事实上，所有操作符都支持重载，除了x.y、f(x)、new、typeof、default、checked、unchecked、delegate、is、as、=和=>之外。尤其注意不能实现赋值操作符；无法改变=操作符的行为。

决定进行操作符重载之前，要意识到这种操作符无法通过IntelliSense发现。除非目的是使类型表现得像是一种基元类型（如数值类型），否则就不要去重载操作符。

9.2.1 比较操作符

一旦重写了Equals()，就可能出现不一致的情况。对两个对象执行Equals()可能返回true。但==操作符可能返回false，因为==默认也是执行引用相等性检查。为了解决这个问题，有必要重载相等（==）和不相等（!=）操作符。

从很大程度上说，这些操作符的实现都可以将逻辑委托给Equals()。反之亦然。然而，首先要执行一些初始的null检查，如代码清单9-6所示。

代码清单9-6 实现==和!=操作符

```
public sealed class Coordinate
{

  // ...

  public static bool operator ==(
      Coordinate leftHandSide,
      Coordinate rightHandSide)
  {

      // Check if leftHandSide is null.
      // (operator== would be recursive)
      if (ReferenceEquals(leftHandSide, null))
      {
          // Return true if rightHandSide is also null
          // but false otherwise.
          return ReferenceEquals(rightHandSide, null);
      }

      return (leftHandSide.Equals(rightHandSide));
  }

  public static bool operator !=(
      Coordinate leftHandSide,
      Coordinate rightHandSide)
  {
      return !(leftHandSide == rightHandSide);
  }

}
```

在本例中，Coordinate是类而不是结构。还要注意，在检查是否为null时，不能用null执行相等性检查，比如leftHandSide == null。否则，就会递归地调用方法，造成只有栈溢出才会终止的死循环。为了避免这个问题，应该调用ReferenceEquals()来检查是否为null。

9.2.2 二元操作符

可以将一个Arc加到一个Coordinate上。但是，迄今为止的代码都没有提供对加法操作符的支持。相反，需要自己定义这样的一个方法，如代码清单9-7所示。

代码清单9-7 新增操作符

```
struct Arc
{
  public Arc(
      Longitude longitudeDifference,
      Latitude latitudeDifference)
  {
      _LongitudeDifference = longitudeDifference;
      _LatitudeDifference = latitudeDifference;
  }

  public Longitude LongitudeDifference
  {
      get
      {
          return _LongitudeDifference;
      }
  }
  private readonly Longitude _LongitudeDifference;

  public Latitude LatitudeDifference
  {
      get
      {
          return _LatitudeDifference;
      }
  }
  private readonly Latitude _LatitudeDifference;
}
```

```
struct Coordinate
{
  // ...
  public static Coordinate operator +(
      Coordinate source, Arc arc)
  {
      Coordinate result = new Coordinate(
          new Longitude(
              source.Longitude + arc.LongitudeDifference),
          new Latitude(
              source.Latitude + arc.LatitudeDifference));
      return result;
  }
}
```

+、-、*、/、%、&、|、^、<<和>>操作符都被实现成二元静态方法，其中至少有一个参数的类型是包容类型（当前正在实现该操作符的类型）。方法名由operator加操作符构成。如代码清单9-8所示，在定义好一和+二元操作符之后，就可以在Coordinate上加减Arc。

注意，Longitude和Latitude也需要实现+操作符，它们由source.Longitude + arc.LongitudeDifference和source.Latitude + arc.LatitudeDifference调用。

代码清单9-8 调用-和+二元操作符

```
public class Program
{
  public static void Main()
  {
      Coordinate coordinate1,coordinate2;
      coordinate1 = new Coordinate(
          new Longitude(48, 52), new Latitude(-2, -20));
      Arc arc = new Arc(new Longitude(3), new Latitude(1));

      coordinate2 = coordinate1 + arc;
      Console.WriteLine(coordinate2);

      coordinate2 = coordinate2 - arc;
      Console.WriteLine(coordinate2);

      coordinate2 += arc;
      Console.WriteLine(coordinate2);
  }
}
```

代码清单9-8的运行结果如输出9-3所示。

输出9-3

```
51° 52' 0 E  -1° -20' 0 N
48° 52' 0 E  -2° -20' 0 N
51° 52' 0 E  -1° -20' 0 N
```

Coordinate的-和+操作符会在加减Arc之后返回坐标位置。这就允许将多个操作符和操作数串联起来，如result = ((coordinate1 + arc1) + arc2) + arc3。另外，通过让Arc支持相同的操作符（参见代码清单9-9），就连圆括号都可以省略。

之所以允许这样写，是因为第一个操作（coordinate1 + arc1）的结果也是Coordinate，可以把它加到下一个Arc或Coordinate类型的操作数上。

相反，假定让操作符-获取两个Coordinate作为参数，并返回double值来表示两个坐标之间的距离。由于没有定义Coordinate和double相加，所以不能串联多个操作符和操作数。操作符返回不同类型有违直觉，所以像这样定义操作符时要小心。

9.2.3 赋值与二元操作符的结合

如前所述，不支持对赋值操作符的重载。然而，只要重载了二元操作符，就自动重载了赋值操作符与二元操作符的结合（+=、-=、*=、/=、%=、&=、|=、^=、<<=和>>=）。定义好一个二元操作符后，C#会自动允许将赋值操作符与二元操作符结合起来使用。所以，可以直接使用下面这样的代码：

```
coordinate += arc;
```

它等价于：

```
coordinate = coordinate + arc;
```

9.2.4 条件逻辑操作符

与赋值操作符相似，条件逻辑操作符不能显式重载。但由于逻辑操作符&和|可以重载，而条件操作符由逻辑操作符构成，因此实际上完全有可能重载条件操作符。x && y可

以作为x & y处理，其中y必须求值为true。类似地，在x求值为false的时候，x || y可以作为x | y处理。要允许将类型求值为true或false（比如在if语句中），就需要重载true/false一元操作符。

9.2.5 一元操作符

一元操作符的重载非常类似于二元操作符的重载，只是它们只获取一个参数，该参数也必须是包容类型（正在重载操作符的类型）。代码清单9-9为Longitude和Latitude重载了+和–操作符。然后，在Arc中重载相同的操作符时使用了这些操作符。

代码清单9-9 重载-和+一元操作符

```
public struct Latitude
{
  // ...
  public static Latitude operator -(Latitude latitude)
  {
      return new Latitude(-latitude.DecimalDegrees);
  }
  public static Latitude operator +(Latitude latitude)
  {
      return latitude;
  }
}
```

```
public struct Longitude
{
  // ...
  public static Longitude operator -(Longitude longitude)
  {
      return new Longitude(-longitude.DecimalDegrees);
  }
  public static Longitude operator +(Longitude longitude)
  {
      return longitude;
  }
}
```

```
public struct Arc
{
  // ...
  public static Arc operator -(Arc arc)
  {
      // Uses unary - operator defined on
      // Longitude and Latitude
      return new Arc(-arc.LongitudeDifference,
                     -arc.LatitudeDifference);
  }
  public static Arc operator +(Arc arc)
  {
      return arc;
  }
}
```

和数值类型一样，代码清单9-9中的+操作符没有任何实际效果，提供它只是为了保持对称。

重载true和false时多了一个要求，即它们都必须进行重载。它们的签名和其他操作符重载相同，但是，返回的必须是一个bool值，如代码清单9-10所示。

代码清单9-10 重载true和false操作符

```
public static bool operator false(IsValid item)
{
    // ...
}
public static bool operator true(IsValid item)
{
    // ...
}
```

重载了true和false操作符的类型可以在if、do、while和for语句的控制表达式中使用。

9.2.6 转换操作符

目前，Longitude、Latitude和Coordinate还不支持其他类型转换。例如，没有办法将double转换为Longitude或者Latitude实例。类似地，也不支持使用string向Coordinate赋值。幸好，C#允许定义方法来处理一种类型向另一种类型的转型。除此之外，允许在方法声明中指定这个转换是隐式的还是显式的。

高级主题：转型操作符

从技术上说，实现显式和隐式转换操作符并不是对转型操作符（()）进行重载。但由于效果一样，所以一般都将“实现显式或隐式转换”说成“定义转型操作符”。

定义转换操作符在形式上类似于定义其他操作符，只是“operator”成了转换的结果类型。除此之外，operator要放在表示隐式或显式转换的implicit或explicit关键字后面，如代码清单9-11所示。

代码清单9-11 在Latitude和double之间提供隐式转换

```
public struct Latitude
{
  // ...

  public Latitude(double decimalDegrees)
  {
      _DecimalDegrees = Normalize(decimalDegrees);
  }

  public double DecimalDegrees
  {
      get { return _DecimalDegrees; }
  }
  private readonly double _DecimalDegrees;

  // ...

  public static implicit operator double(Latitude latitude)
  {
      return latitude.DecimalDegrees;
  }
  public static implicit operator Latitude(double degrees)
  {
      return new Latitude(degrees);
  }
```

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

定义好这些转换操作符之后，就可以将double隐式转型为Latitude对象，或者将Latitude对象隐式转型为double。假定为Longitude定义了类似的转换，那么为了创建Coordinate对象，可以像下面这样写：

```
coordinate = new Coordinate(43, 172);
```

注意

实现转换操作符时，为了保证封装性，要么返回值，要么参数必须是封闭类型。C#不允许在被转换类型的作用域之外指定转换。[①]

9.2.7 转换操作符的规范

定义隐式和显式转换操作符的差别主要在于，后者能防止不小心执行隐式转换，这些隐式转换可能造成你不希望的行为。使用显式转换操作符通常是出于两方面的考虑。首先，会引发异常的转换操作符始终都应该是显式的。例如，从string转换为Coordinate时，提供的string并非一定具有正确的格式（这是极有可能发生的）。由于转换可能失败，所以应该将转换操作符定义为显式，从而明确要进行转换的意图，并要求用户确保格式正确，或者由用户提供代码来处理可能发生的异常。通常采用的转换模式是：在一个方向上（string到Coordinate）显式，相反方向（Coordinate到string）隐式。

第二个要考虑的是某些转换可能丢失信息。例如，虽然完全可以从float(4.2)转换成int，但前提是用户知道float小数部分会丢失这一事实。任何转换只要会丢失数据，而且不能成功转换回原始类型，就应该定义成显式转换。如果显式转换出乎意料持地丢失了信息，或者转换无效，考虑引发System.InvalidCastException异常。

规范

不要为有损转换提供隐式转换操作符。

不要从隐式转换中引发异常。

9.3 引用其他程序集

C#和底层CLI平台不是将所有代码都放到一个二进制文件中，而是允许将代码分散到多个程序集中。这样就可以在多个可执行文件中重用程序集。

初学者主题：类库

HelloWorld.exe程序是你写过的最简单的程序之一。现实世界的程序要复杂得多，而且随着复杂性的增加，有必要将程序分解成多个小的部分，便于控制复杂性。为此，开发者可以将程序的不同部分转移到单独的编译单元中，这些单元称为**类库**，或者简称为

① 换言之，转换操作符只能从它的封闭类型转换为其他某个类型，或从其他某个类型转换为它的封闭类型。——译者注

库。然后，程序可以引用和依赖类库来提供自己的一部分功能。这样一来，两个程序就可以依赖同一个类库，从而在两个程序中共享那个类库的功能，并减少所需的编码量。

换言之，特定功能只需编码一次，并放到类库中。然后，多个程序可引用同一个类库来提供该功能。以后，如果开发者修正了bug，或者在类库中添加了新功能，所有程序都能获得增强的功能，因为它们现在引用的是改善过的类库。

为了重用其他程序集中的代码，要在运行C#编译器时引用程序集。通常，引用的程序集是一个类库。为了创建类库，要用/target选项指定和默认不同的目标。

9.3.1 更改程序集目标

编译器允许通过/target选项创建下面4种不同的程序集类型。

- 控制台可执行程序：这是默认的程序集类型。本书到目前为止都是编译成这种类型（省略/target或者指定/target:exe，都将创建一个控制台可执行程序）。
- 类库：在多个可执行程序之间共享的类通常在类库中定义（/target:library）。
- Windows可执行程序：Windows可执行程序设计在Windows操作系统中运行，不是使用控制台界面（/target:winexe）。
- 模块：为了在程序集中集成多种语言，代码可以编译成模块，多个模块可以合并为程序集（/target:module）。

要在多个应用程序中共享的程序集通常编译成类库。例如，为了将Coordinate、Longitude和Latitude类编译到专门提供经纬度坐标功能的库中，可以使用如输出9-4所示的命令。

输出9-4

```
>csc /target:library /out:Coordinates.dll Coordinate.cs IAngle.cs
Latitude.cs Longitude.cs Arc.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.20506.1
Copyright (C) Microsoft Corporation. All rights reserved.
```

假定正在使用.NET，而且C#编译器在路径中，就会生成名为Coordinates.dll的程序集（库）。

9.3.2 引用程序集

为了访问不同程序集中的代码，C#编译器允许开发者在命令行上引用程序集。具体的选项是/reference（/r是缩写），后跟一个引用列表。代码清单9-8的Program类使用了Coordinate类，如果要将这些代码放在一个独立的可执行文件中，那么为了引用Coordinates.dll文件，你要像输出9-5那样执行编译命令。

输出9-5

```
csc.exe /R:Coordinates.dll Program.cs
```

对应的Mono命令行如输出9-6所示。

输出9-6

```
msc.exe /R:Coordinates.dll Program.cs
```

9.3.3 类型封装

类封装了一系列相关的行为和数据，程序集则封装了一系列相关的类型。开发者可以将一个系统分解成多个程序集，然后在多个应用程序中共享那些程序集，或者将它们与第三方提供的程序集集成。

默认情况下，没有任何访问修饰符的类被定义成internal[①]。结果是该类无法从程序集外部访问。即使另一个程序集引用了该类所在的程序集，被引用程序集中的所有internal类都是无法访问的。

类似于为类成员使用private和protected访问修饰符来指定不同的封装级别，C#允许为类使用访问修饰符，从而控制类在程序集中的封装级别。可用的访问修饰符是public和internal。类要在程序集外部可见，必须标记成public。所以，在编译Coordinates.dll程序集之前，需要像代码清单9-12那样，将类型声明修改为public。

代码清单9-12 使类在程序集外部可用

```
public struct Coordinate
{
  // ...
}
```

```
public struct Latitude
{
  // ...
}
```

```
public struct Longitude
{
  // ...
}
```

```
public struct Arc
{
  // ...
}
```

类似地，像class和enum这样的声明也可以是public或者internal的。

高级主题：其他类访问修饰符

嵌套类可以使用适用于其他类成员的任何访问修饰符（如private）来修饰。但是，在类的作用域之外，唯一可用的访问修饰符只有public和internal。

internal访问修饰符并非仅适用于类型声明。它还适用于类型的成员。所以，可以将某个类型标记为public，但将其中的一些方法标记为internal，确保成员只在程序集内部可用。成员的可访问性无法超过它所在的类型的可用性。例如，将类声明为internal，它的public成员就只能从程序集中访问。

protected internal是另一种类型成员访问修饰符。这种成员可从包容程序集的任何位置以及类型的派生类中访问（即使派生类不在同一个程序集中）。由于默认是private，所以随便指定别的一个访问修饰符（public除外），成员的可见性都会稍微提高。类似地，添加两个修饰符，可访问性会复合到一起，变得更大。

① 嵌套类型除外，它们默认为private。

注意

protected internal成员可以从包容程序集的任何位置以及类型的派生类中访问（即使派生类不在同一个程序集中）。

初学者主题：类型成员的可访问性修饰符

表9-1提供了完整的访问修饰符列表。

表9-1 可访问性修饰符

修饰符	描 述
public	凡是能访问类型的地方，都能访问类型的成员。如果类是internal的，则成员只在程序集内部可见。如果包容类型是public的，public成员可以从程序集外部访问
internal	成员只能从当前程序集中访问
private	成员可以从包容类型中访问，但其他任何地方都不能访问
protected	成员可以从包容类型以及任何派生类型中访问，即使派生类型在不同程序集中
protected internal	成员可以从包容程序集中的任何地方访问，还可以从包容类型的任何派生类中访问，即使派生类在不同程序集中

9.4 定义命名空间

第2章讲过，任何数据类型都用命名空间与名称的组合来标识。事实上，CLR对“命名空间”一无所知。CLR中的类型名称都是完全限定的。早先定义的类没有显式声明命名空间。这种类自动声明为默认的全局命名空间的成员。但这极有可能造成名称冲突。试图定义两个同名的类时，冲突就会发生。一旦开始引用第三方的程序集，名称冲突的概率会变得更大。

更重要的是，.NET Framework内部有成千上万个类型，外部的类型则更多。所以，为特定问题寻找适当的类型可能十分费劲。

解决这些问题的方案是组织所有的类型，用**命名空间**对它们进行逻辑分组。例如，System命名空间外部的类通常应该放到与公司名、产品名或者“公司名+产品名”对应的命名空间中。例如，来自Addison-Wesley [①]的类应该放到Awl或AddisonWesley命名空间中，而来自Microsoft的类（System中的类除外）应该放到Microsoft命名空间中。命名空间的第二级应该是不会随着版本升级而改变的稳定产品名称。事实上，稳定性在所有级别上都是关键的。改变命名空间的名称会影响版本的兼容性，所以应该避免这样做。有鉴于此，应避免在一个命名空间的名称中使用容易变化的名称（例如组织层次结构、短期品牌等）。

命名空间应使用Pascal大小写风格，但如果你的品牌使用非传统大小写风格，也可以使用和品牌相符的大小写形式。（一致性是关键，所以如果面临两难选择——是采用Pascal大小写风格还是采用品牌名称，就使用一致性更好的风格。）用namespace关键字创建命名空间，并将类分配给该命名空间，如代码清单9-13所示。

代码清单9-13 定义命名空间

```
// Define the namespace AddisonWesley
```

① 出版社名称。——译者注

```
namespace AddisonWesley
{
  class Program
  {
      // ...
   }
}
// End of AddisonWesley namespace declaration
```

命名空间大括号之间的所有内容都从属于该命名空间。在代码清单9-13中，Program被放到命名空间AddisonWesley中，所以它的全名是AddisonWesley.Program。

> **注意**
>
> CLR中没有“命名空间”这种东西。类型名称必然完全限定。

和类相似，命名空间也可以嵌套。这样就可以对类进行层次化的组织。例如，与网络API有关的所有系统类都放到System.Net命名空间中，而与Web相关的系统类都放到System.Web中。

有两个办法嵌套命名空间。第一个办法是逐级嵌套（和类一样），如代码清单9-14所示。

代码清单9-14 逐渐嵌套命名空间

```
// Define the namespace AddisonWesley
namespace AddisonWesley
{
  // Define the namespace AddisonWesley.Michaelis
  namespace Michaelis
  {
    // Define the namespace
    // AddisonWesley.Michaelis.EssentialCSharp
    namespace EssentialCSharp
    {
      // Declare the class
      // AddisonWesley.Michaelis.EssentialCSharp.Program
      class Program
      {
          // ...
      }
    }
  }
}
// End of AddisonWesley namespace declaration
```

这样嵌套会将Program类分配给AddisonWesley.Michaelis.EssentialCSharp命名空间。

第二个办法是在namespace声明中使用完整的命名空间名称，每个标识符都以句点分隔，如代码清单9-15所示。

代码清单9-15 使用句点来分隔每个标识符，从而嵌套命名空间

```
// Define the namespace AddisonWesley.Michaelis.EssentialCSharp
namespace AddisonWesley.Michaelis.EssentialCSharp
{
  class Program
  {
      // ...
```

```
    }
}
// End of AddisonWesley namespace declaration
```

无论像代码清单9-14或代码清单9-15，还是采取两者的组合方式来声明命名空间，最终的CIL代码都完全一致。同一个命名空间可以多次出现，可以在多个文件中出现，甚至可以跨越多个程序集。例如，约定是一个类对应一个文件。所以，可以为每个类都定义一个文件，并用同一个命名空间声明把类包容起来。

由于命名空间是对类型进行组织的关键，所以使用命名空间来组织所有的类文件通常都是有益的。有鉴于此，可以为每个命名空间都创建一个文件夹，将AddisonWesley.Fezzik.Services.Registration这样的类放到和名称对应的文件夹层次结构中。

使用Visual Studio项目时，如果项目名称是AddisonWesley.Fezzik，就创建子文件夹Services，将RegistrationService.cs放到其中。然后创建另一个子文件夹Data，在其中放入与程序中的实体有关的类，比如RealestateProperty、Buyer和Seller。

规范

要为命名空间名称附加公司名前缀，防止不同公司的命名空间使用相同的名称。

要为命名空间名称中的二级名称使用稳定的、不随版本升级而变化的产品名称。

不要定义没有明确放到一个命名空间中的类型。

考虑创建与命名空间层次结构相匹配的文件夹结构。

命名空间别名限定符

命名空间旨在处理有可能发生的大量命名冲突。但有时（虽然很少见）会因为命名空间与类重名而发生冲突。为了解决这个问题，C# 2.0编译器开始允许使用/reference选项提供别名。例如，假定程序集CoordinatesPlus.dll和Coordinates.dll都有Arc类型，那么在命令行上同时引用这两个程序集时，可以为一个或两个引用指定**命名空间别名限定符**（namespace alias qualifier），从而进一步区分两个类。输出9-7展示了具体的命令。

输出9–7

```
csc.exe /R:CoordPlus=CoordinatesPlus.dll /R:Coordinates.dll Program.cs
```

仅在编译时添加别名还不够。为了引用别名程序集中的类，需要提供一个extern指令，它声明命名空间别名限定符是在外部提供给源代码的，如代码清单9-16所示。

代码清单9–16 使用extern别名指令

```
// extern must precede all other namespace elements
extern alias CoordPlus;

using System;
using CoordPlus::AddisonWesley.Michaelis.EssentialCSharp
// Equivalent also allowed
// using CoordPlus.AddisonWesley.Michaelis.EssentialCSharp
```

```
using global::AddisonWesley.Michaelis.EssentialCSharp
// Equivalent NOT allowed
// using global.AddisonWesley.Michaelis.EssentialCSharp

public class Program
{
  // ...
}
```

指定CoordPlus是外部别名以后，只需写下CoordPlus，在后面写两个冒号或者一个句点就可以引用它所代表的命名空间了。

要在全局命名空间中查找类型，C# 2.0允许使用global::限定符（但不允许使用global.，否则会被误会成一个真正的、名为global的命名空间）。

9.5 XML注释

第1章介绍了注释。但XML注释更胜一筹，它不仅仅是向审阅代码的其他程序员显示的注解。XML注释遵循随着Java开始流行起来的实践准则。虽然C#编译器在最终生成的可执行文件中忽略所有注释，但开发者可以利用命令行选项，指示编译器[①]将XML注释提取到单独的XML文件中。这样就可根据XML注释生成API文档。除此之外，C#编辑器能解析代码中的XML注释，并对其进行分区显示（例如，可以使用有别于其他代码的一种颜色），或者解析XML注释数据元素并向开发者显示。

图9-2演示了IDE如何利用XML注释向开发者提示如何编码。

```
/// <summary>
/// Display the text specified
/// </summary>
/// <param name="text">The text to be displayed in the console.</param>
private static void Display(string text)
{
    Console.WriteLine(text);
}

static void Main()
{
    Display(
}
```

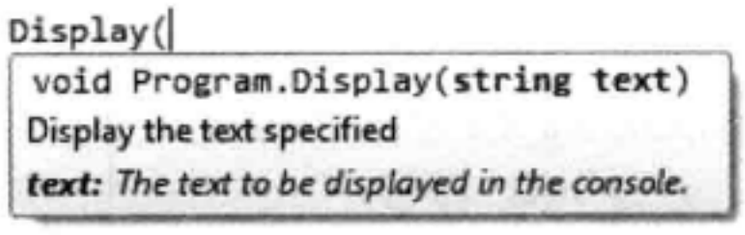


图9-2 以提示形式在Visual Studio IDE中显示的XML注释

这些编码提示能在大型程序中提供重要的帮助，尤其是在多个开发者共享代码的时候。然而，为了让它起作用，开发者必须花一些时间在代码中输入XML注释内容，然后指示编译器创建XML文件。下一节解释具体如何做。

9.5.1 将XML注释与代码构造关联到一起

来看代码清单9-17展示的DataStorage类。

① C#标准没有硬性规定要由C#编译器还是实用程序来负责提取XML数据。然而，所有主流C#编译器都通过一个编译开关提供了这个功能，而不是要求使用额外的实用程序。

代码清单9-17 使用XML注释添加代码注释

```
        /// <summary>
        /// DataStorage is used to persist and retrieve
        /// employee data from the files.
        /// </summary>
        class DataStorage
    {
        /// <summary>
        /// Save an employee object to a file
        /// named with the Employee name.
        /// </summary>
        /// <remarks>
        /// This method uses
        /// <seealso cref="System.IO.FileStream"/>
        /// in addition to
        /// <seealso cref="System.IO.StreamWriter"/>
        /// </remarks>
        /// <param name="employee">
        /// The employee to persist to a file</param>
        /// <date>January 1, 2000</date>
        public static void Store(Employee employee)
        {
            // ...
        }

        /**  <summary>
         *  Loads up an employee object
         *  </summary>
         *  <remarks>
         *  This method uses
         *  <seealso cref="System.IO.FileStream"/>
         *  in addition to
         *  <seealso cref="System.IO.StreamReader"/>
         *  </remarks>
         *  <param name="firstName">
         *  The first name of the employee</param>
         *  <param name="lastName">
         *  The last name of the employee</param>
         *  <returns>
         *  The employee object corresponding to the names
         *  </returns>
         *  <date>January 1, 2000</date>**/
        public static Employee Load(
            string firstName, string lastName)
        {
            // ...
        }
    }

    class Program
    {
        // ...
    }
```

XML单行注释

XML带分隔符的注释（C#2.0）

代码清单9-17既使用了跨越多行的XML带分隔符的注释，也使用了每行都要求单独使用一个///分隔符的XML单行注释。

由于XML注释用于提供API文档，所以一般只和C#声明（比如代码清单9-17中的类或方法）配合使用。如果试图将XML注释嵌入代码，同时不和一个声明关联，就会造成编译器显示一条警告消息。编译器判断是否关联的依据很简单——如果XML注释恰好在一个声明之前，两者就是关联的。

虽然C#允许在注释中使用任意XML标记，但C#标准明确定义了一组可以使用的标记。`<seealso cref="System.IO.StreamWriter"/>`是使用seealso标记的一个例子。该标记在文本和System.IO.StreamWriter类之间创建了一个链接。

9.5.2 生成XML文档文件

编译器会检查XML注释是不是良构的。不是会显示警告。生成XML文件需要在编译时使用/doc选项，如输出9-8所示。

输出9-8

```
>csc /doc:Comments.xml DataStorage.cs
```

/doc选项指示编译器使用冒号后的名称创建XML文件。使用代码清单9-17展示的DataStorage类以及这里的编译器选项，会生成如代码清单9-18所示的Comments.xml文件。

代码清单9-18 Comments.xml

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>DataStorage</name>
    </assembly>
    <members>
        <member name="T:DataStorage">
            <summary>
            DataStorage is used to persist and retrieve
            employee data from the files.
            </summary>
        </member>
        <member name="M:DataStorage.Store(Employee)">
            <summary>
            Save an employee object to a file
            named with the Employee name.
            </summary>
            <remarks>
            This method uses
            <seealso cref="T:System.IO.FileStream"/>
            in addition to
            <seealso cref="T:System.IO.StreamWriter"/>
            </remarks>
            <param name="employee">
            The employee to persist to a file</param>
            <date>January 1, 2000</date>
</member>
<member name="M:DataStorage.Load(
```

```
            System.String,System.String)">
        <summary>
        Loads up an employee object
        </summary>
        <remarks>
        This method uses
        <seealso cref="T:System.IO.FileStream"/>
        in addition to
        <seealso cref="T:System.IO.StreamReader"/>
        </remarks>
        <param name="firstName">
        The first name of the employee</param>
        <param name="lastName">
        The last name of the employee</param>
        <returns>
        The employee object corresponding to the names
        </returns>
        <date>January 1, 2000</date>*
      </member>
    </members>
  </doc>
```

文件只包含最基本的、将元素与C#声明关联所需的元数据。这一点需要注意，因为为了生成任何有意义的文档，一般都有必要将XML输出和生成的程序集配合起来使用。幸好，可利用一些工具（如免费的GhostDoc[①]和开源项目NDoc[②]）来生成文档。

规范

如果API签名不能完全说明问题，要为公共API提供XML注释，其中包括成员说明、参数说明和API调用示例。

9.6 垃圾回收

垃圾回收是“运行时”的核心功能，作用是回收不再被引用的对象所占用的内存。这句话的重点是“内存”和“引用”。垃圾回收器只回收内存，不处理其他资源，比如数据库连接、句柄（文件、窗口等）、网络端口以及硬件设备（比如串口）。此外，垃圾回收器根据是否存在任何引用来决定要清理什么。这暗示垃圾回收器处理的是引用对象，只回收堆上的内存。另外，还意味着假如维持对一个对象的引用，就会阻止垃圾回收器重用对象所用的内存。

高级主题：.NET中的垃圾回收

垃圾回收器的许多细节都依赖于CLI实现，所以各不相同。本节讨论最常见的.NET的实现。

.NET的垃圾回收器采用mark-and-compact算法[③]。在一次垃圾回收周期开始的时候，它识别对象的所有**根引用**。根引用是来自静态变量、CPU寄存器以及局部变量或者

① 请访问http://submain.com，更多地了解这个工具。
② 请访问http://ndoc.sourceforge.net，更多地了解这个工具。
③ 所谓mark-and-compact算法，mark是指先确定所有可达对象，compact是指移动这些对象，使它们紧挨着存放。整个过程有点儿像磁盘碎片整理。注意MSDN文档中把compact不恰当地翻译成“压缩”。——译者注

参数实例（以及本节稍后会讲到的f-reachable对象）的任何引用。基于这个列表，垃圾回收器可以遍历每个根引用所标识的树形结构，并递归确定所有根引用指向的对象。这样，垃圾回收器就可以识别出所有可达对象。

执行垃圾回收时，垃圾回收器不是枚举所有访问不到的对象；相反，它将所有可达对象紧挨着放到一起，从而覆盖不可访问的对象（也就是垃圾，或者不可达对象）所占用的内存。

为了定位和移动所有可达对象，系统要在垃圾回收器运行期间维持状态的一致性。为此，进程中的所有托管线程都会在垃圾回收期间暂停。这显然会造成应用程序出现短暂的停顿。不过，除非垃圾回收周期特别长，否则这个停顿是不太引人注意的。为了尽量避免在不恰当的时间执行垃圾回收，`System.GC`对象包含一个`Collect()`方法。可在执行关键代码之前调用它（执行这些代码时不希望GC运行）。这样做不会阻止垃圾回收器运行，但会显著减小它运行的可能性——前提是关键代码执行期间不会发生内存被大量消耗的情况。

.NET垃圾回收的特别之处在于，并非所有垃圾都一定会在一个垃圾回收周期中清除。研究对象的生存期，发现相较于长期存在的对象，最近创建的对象更有可能需要垃圾回收。为此，.NET垃圾回收器支持“代”（generation）的概念，它会以更快的频率尝试清除生存时间较短的对象（新生对象）。而那些已在一次垃圾回收中“存活”下来的对象（老对象）会以较低的频率清除。具体地说，总共有3代对象。一个对象每次在一个垃圾回收周期中存活下来，它都会移动到下一代，直至最终移动到第二代（从第零代开始）。相较于第二代对象，垃圾回收器会以更快的频率对第零代的对象执行垃圾回收。

在.NET的beta测试阶段，一些舆论认为它的性能比不上非托管代码，但时间已经证明，.NET的垃圾回收机制是相当高效的。更重要的是，它带来了开发过程总体效率的提高——虽然在极少数情况下，因为要对特定的算法进行优化，托管代码的性能会出现打折扣的情况。

弱引用

迄今为止讨论的所有引用都是**强引用**，因为它们维持着对象的可访问性，禁止垃圾回收器清除对象所占用的内存。除此之外，.NET Framework还支持**弱引用**。弱引用不禁止对对象进行垃圾回收，但会维持一个引用。这样，如果对象尚未被垃圾回收器清除，就可以重用。

弱引用是为创建起来代价较高（开销很大），而且维护开销特别大的对象而设计的。例如，假定一个很大的对象列表要从一个数据库中加载，并向用户显示。在这种情况下，加载这个列表的代价是很高的。一旦列表被用户关闭，就应该可以进行垃圾回收。但是，假如用户多次请求这个列表，那么每次都要执行代价高昂的加载动作。解决方案是使用弱引用。这样就可使用代码检查列表是否清除。如果尚未清除，就重新引用同一个列表。这样，弱引用就相当于对象的一个内存缓存。缓存中的对象可被快速获取。但是，假如垃圾回收器已回收了这些对象的内存，就还是要重新创建它们。

如果认为对象（或对象集合）应该进行弱引用，就把它赋给`System.WeakReference`（参见代码清单9-19）。

代码清单9-19 使用弱引用

```
// ...
```

```
    private WeakReference Data;

    public FileStream GetData()
    {
        FileStream data = (FileStream)Data.Target;
        if (data != null)
        {
            return data;
        }
        else
        {
            // Load data
            // ...

            // Create a weak reference
            // to data for use later.
            Data.Target = data;
        }
        return data;
    }

    // ...
```

创建弱引用（Data）之后，可查看弱引用是否为null来检查垃圾回收。然而，这里的关键是先将弱引用赋给一个强引用（FileStream data = Data），避免在“检查null值”和“访问数据”这两个动作之间，垃圾回收器运行并清除弱引用。强引用明显会阻止垃圾回收器清除对象，所以它必须先被赋值（而不是先检查Target是不是为null）。

9.7 资源清理

垃圾回收是“运行时”的重要职责。但要注意，垃圾回收旨在提高内存利用率，而非清理文件句柄、数据库连接字符串、端口或者其他有限的资源。

9.7.1 终结器

终结器（finalizer）允许程序员编写代码来清理类的资源。但是，和使用new显式调用构造器不同，终结器不能从代码中显式调用。没有与new对应的操作符（比如像delete这样的操作符）。相反，是垃圾回收器负责为对象实例调用终结器。因此，开发者不能在编译时确定执行终结器的时间。唯一确定的是终结器会在对象最后一次使用之后，并在应用程序正常关闭前的某个时间运行。（如果进程异常终止，终结器将不会运行。例如，计算机断电或者进程被强行终止，就会阻止终结器的运行。）

> **注意**
>
> 编译时不能确定终结器的确切运行时间。

终结器的声明与C++析构器的语法完全一致。如代码清单9-20所示，终结器声明要求在类名之前添加一个“~”字符。

代码清单9-20 定义终结器

```
using System.IO;
class TemporaryFileStream
{
  public TemporaryFileStream()
  {
      _File = new FileInfo(Path.GetTempFileName());
      _Stream = new FileStream(
          File.FullName, FileMode.OpenOrCreate,
          FileAccess.ReadWrite);
  }

  // Finalizer
  ~TemporaryFileStream()
  {
      Close();
  }

  public FileStream Stream
  {
      get { return _Stream; }
  }
  readonly private FileStream _Stream;

  public FileInfo File
  {
      get { return _File; }
  }
  readonly private FileInfo _File =
      new FileInfo(Path.GetTempFileName());

  public void Close()
  {
      if(Stream != null)
      {
        Stream.Close();
      }
      if(File != null)
      {
        File.Delete();
      }
  }
}
```

终结器不允许传递任何参数，结果是终结器不能重载。此外，终结器不能显式调用。调用终结器的只能是垃圾回收器。因此，为终结器添加访问修饰符没有意义（也不支持）。基类中的终结器作为对象终结调用的一部分被自动调用。

> **注意**
>
> 终结器不能显式调用，只有垃圾回收器才能调用终结器。

由于垃圾回收器负责所有内存管理工作，所以终结器不负责回收内存。它们主要负责释放像数据库连接和文件句柄这样的资源，这些资源需要通过一次显式的行动来进行清理，而垃圾回收器不知道具体如何采取这些行动。

注意，终结器在自己的线程中执行，这使它们的执行变得更不确定。这种不确定性使

终结器中（调试器外）的一个未处理的异常变得难以诊断，因为造成异常的情况是不明朗的。从用户的角度看，未处理异常的引发时机显得相当随机，跟用户当时执行的任何操作都没有太大关系。有鉴于此，一定要避免在终结器中出现异常。为此需要采取一些防卫性编程技术，比如空值检查（参见代码清单9-20）。

9.7.2 使用using语句进行确定性终结

终结器的问题在于不支持**确定性终结**（deterministic finalization，也就是预知终结器在何时运行）。相反，终结器是对资源进行清理的备用机制。假如类的使用者忘记显式调用必要的清理代码，就得依赖终结器清理资源。

例如，假定TemporaryFileStream不仅包含终结器，还包含Close()方法。该类使用了一个可能大量占用磁盘空间的文件资源。使用TemporaryFileStream的开发者可以显式调用Close()回收磁盘空间。

很有必要提供进行确定性终结的方法，避免依赖终结器不确定的计时行为。即使开发者忘记显式调用Close()，终结器也会调用它。虽然终结器运行得会晚一些（相较于显式调用Close()），但该方法肯定会得到调用。

由于确定性终结的重要性，基类库为这个使用模式包含了特殊接口，而且C#已将这个模式集成到语言中。IDisposable接口用名为Dispose()的方法定义了该模式的细节。开发者可针对资源类调用该方法，从而dispose[①]当前占用的资源。代码清单9-21演示了IDisposable接口以及调用它的一些代码。

代码清单9-21 使用IDisposable清理资源

```
using System;
using System.IO;

class Program
{
  // ...
  static void Search()
  {
      TemporaryFileStream fileStream =
          new TemporaryFileStream();

      // Use temporary file stream;
      // ...
      fileStream.Dispose();

      // ...
  }
}

class TemporaryFileStream : IDisposable
{
```

① dispose在本书没有翻译。在英语语境中，它的意思是“摆脱”或“除去”（get rid of）一个东西，尤其是在这个东西很难除去的情况下。MSDN文档把它翻译成“释放”，意思是显式释放或清理对象包装的资源。之所以认为“释放”不恰当，除了和release一词冲突之外，还因为dispose强调了“清理资源”，而且在完成（它包装的）资源的清理之后，对象本身的内存并不会释放。所以，“dispose一个对象”或者“close一个对象”真正的意思是：清理对象中包装的资源（比如它的字段所引用的对象），然后等待垃圾回收器自动回收该对象本身占用的内存。——译者注

```
public TemporaryFileStream()
{
    _File = new FileInfo(Path.GetTempFileName());
    _Stream = new FileStream(
        File.FullName, FileMode.OpenOrCreate,
        FileAccess.ReadWrite);
}

~TemporaryFileStream()
{
    Dispose(false);
}

public FileStream Stream
{
    get { return _Stream; }
}
readonly private FileStream _Stream;

public FileInfo File
{
    get { return _File; }
}
readonly private FileInfo _File;

public void Close()
{
    Dispose();
}

#region IDisposable Members
public void Dispose()
{
    Dispose(true);

    // Turn off calling the finalizer
    System.GC.SuppressFinalize(this);
}
#endregion
public void Dispose(bool disposing)
{
    // Do not dispose of an owned managed object (one with a
    // finalizer) if called by member finalize
    // as the owned managed objects finalize method
    // will be (or has been) called by finalization queue
    // processing already
    if (disposing)
    {
        if (Stream != null)
        {
            Stream.Close();
        }
    }
    if(File != null)
    {
      File.Delete();
    }
  }
}
```

Program.Search()在用完TemporaryFileStream之后显式调用Dispose()。Dispose()负责清理和内存无关的资源（本例中是一个文件），这些资源不会由垃圾回收器隐式清理。但在执行过程中，仍有一个漏洞会阻止Dispose()执行。实例化TemporaryFileStream后，调用Dispose()前，有可能发生一个异常。这造成Dispose()得不到调用，资源清理不得不依赖于终结器。为避免这个问题，调用者需要实现一个try/finally块。但是，开发者不需要显式地写一个这样的块。因为C#提供了using语句来用于这个目的。最终代码如代码清单9-22所示。

代码清单9-22 执行using语句

```
class Program
{
  // ...

  static void Search()
  {
      using (TemporaryFileStream fileStream1 =
          new TemporaryFileStream(),
          fileStream2 = new TemporaryFileStream())
      {
          // Use temporary file stream;
      }
  }
}
```

最终生成的CIL代码与写一个显式的try/finally块完全一样。如果自己写try/ finally块，那么fileStream.Dispose()应该在finally块中调用。总之，using语句只是提供了try/finally块的语法快捷方式。

可在using语句内实例化多个变量，只需用逗号分隔每个变量即可。关键是所有变量都具有相同的类型，都实现了IDisposable。为了强制使用相同类型，数据类型只能指定一次，而不是在每个变量声明之前都指定一次。

9.7.3 垃圾回收、终结和IDisposable

代码清单9-21还有另外几个值得注意的地方。首先，IDisposable.Dispose()方法包含对System.GC.SuppressFinalize()的调用，作用是从终结（f-reachable）队列中移除TemporaryFileStream类实例。这是因为所有清理都在Dispose()方法中完成了，而不是等着终结器执行。

不调用SuppressFinalize()，对象的实例会被包括到f-reachable队列中。该队列中的对象已差不多准备好了进行垃圾回收，只是它们还有终结方法没有运行。“运行时”只有在对象的终结方法被调用之后，才能对这个对象进行垃圾回收。然而，垃圾回收器本身不会调用终结方法。相反，对这种对象的引用会添加到f-reachable队列中，并由一个额外的线程根据执行上下文，挑选合适的时间进行处理。讽刺的是，这样做推迟了托管资源的垃圾回收——而许多这样的资源本来应更早一些被清理。推迟是由于f-reachable队列是“引用”列表。所以，对象只有在它的终结方法得到调用，而且对象引用从f-reachable队列中删除之后，才会真正变成“垃圾”。

注意

有终结器的对象如果不显式地释放，其生存期会被延长，因为即使对它的所有显式引用都不存在了，f-reachable队列中仍然包含着对它的引用，使对象一直生存，直至f-reachable队列处理完毕。

正是由于这个原因，Dispose()才调用System.GC.SuppressFinalize，从而告诉“运行时”不要将该对象添加到f-reachable队列，而是允许垃圾回收器在对象没有任何引用（包括任何f-reachable引用）时清除对象。

其次，Dispose()调用了Dispose(bool disposing)方法，并传递实参true。结果是为Stream调用Dispose()方法（清理它的资源并阻止终结）。接着，临时文件在调用Dispose()之后立即删除。这个重要的调用避免了一定要等待终结队列处理完毕才能清理潜在的贵重资源。

再次，终结器现在不是调用Close()，而是调用Dispose(bool disposing)，并传递实参false。结果是即使文件被删除，Stream也不会关闭（disposed）。原因是从终结器中调用Dispose(bool disposing)时，Stream实例本身还在等待终结（或者已经终结，系统会以任意顺序终结对象）。所以，在执行终结器时，拥有托管资源的对象不应该被清理，那应该是终结队列的职责。

最后，同时创建Close()和Dispose()方法需谨慎。只看API并不知道Close()会调用Dispose()。所以开发人员搞不清楚是否要显示调用Close()和Dispose()。

规范

要只为使用了稀缺或昂贵资源的对象实现终结器方法，即使终结会推迟垃圾回收。

要为有终结器的类实现IDisposable接口以支持确定性终结。

要为实现了IDisposable的类实现终结器方法，以防Dispose()没有被显式调用。

要重构终结方法来调用与IDisposable相同的代码，可能就是调用一下Dispose()方法。

不要在终结器方法中引发异常。

要从Dispose()中调用System.GC.SuppressFinalize()，使垃圾回收更快地发生，并避免重复性的资源清理。

要保证Dispose()可以重入（可以被多次调用）。

要保持Dispose()的简单性，把重点放在终结所要求的资源清理上。

避免为自己拥有的、带终结器的对象调用Dispose()。相反，依赖终结队列清理实例。

避免在终结方法中引用未被终结的其他对象。

要在重写Dispose()时调用基类的实现。

考虑在调用Dispose()之后将对象状态设为不可用。对象被dispose之后，调用除Dispose()之外的方法都应该引发ObjectDisposedException异常。（Dispose()应该能多次调用。）

要为含有可dispose字段（或属性）的类型实现IDisposable接口，并dispose这些字段引用的对象。

语言对比：C++——确定性析构

虽然终结器类似于C++析构器，但由于编译时无法确定终结器的执行时间，因此两者实际存在相当大的区别。垃圾回收器调用C#终结器是在对象最后一次使用之后、应用程序关闭之前的某个时间。相反，只要对象（而非指针）超出作用域，C++析构器就会自动调用。

虽然运行垃圾回收器可能代价相当高，但事实上垃圾回收器拥有足够的判断力，会一直等到处理器占用率较低的时候才运行。在这一点上，我们认为它优于确定性析构器。确定性析构器会在编译时定义的位置运行——即使当前处理器处于重负荷下。

高级主题：从构造器传播的异常

即使有异常从构造器传播出来，对象仍会实例化，只是没有新实例从new操作符返回。如果类型定义了一个终结器，对象一旦准备好进行垃圾回收，就会运行该方法（即使只构造了一部分的对象，也会提供额外的动机以确保终结方法能够运行）。另外要注意，如果构造器过早共享它的this引用，即使构造器引发异常，也能访问该引用。不要让这种情况发生。

高级主题：对象复活

调用对象的终结方法时，对该对象的引用都已消失。垃圾回收前唯一剩下的步骤就是运行终结代码。然而，完全可能无意之间重新引用一个待终结的对象。这样，被重新引用的对象就不再是不可访问的，所以不能当作垃圾被回收掉。然而，假如对象的终结方法已经运行，那么除非显式标记为要进行终结（使用GC.ReRegisterFinalize()方法），否则终结方法不一定会再次运行。

显然，像这样的对象复活是非常罕见的，而且通常应该避免这样做。终结代码应该简单，只清理它引用的资源。

9.8 推迟初始化

上一节讨论了如何用using语句确定性地释放（dispose）对象，以及在没有进行确定性dispose的时候终结队列如何清理资源。

与此相关的模式称为**推迟初始化**（lazy initialization）或者**推迟加载**（lazy loading）。使用延迟初始化，可以在需要时才创建（或获取）对象，而不是提前创建好——尤其是它们永远都不使用的前提下。来考虑代码清单9-23中的FileStream属性。

代码清单9-23 推迟加载属性

```
using System.IO;


class DataCache
{
  // ...

  public TemporaryFileStream FileStream
```

```
{
    get
    {
        if (_FileStream == null)
        {
            _FileStream = new TemporaryFileStream();
        }
        return _FileStream;
    }
}
private TemporaryFileStream _FileStream = null;

// ...
}
```

在FileStream属性中，仅当调用属性的get访问器方法时才实例化TemporaryFileStream对象。get访问器方法永不调用，TemporaryFileStream对象永不实例化。这样就可省下进行这个实例化所需的时间。当然，如果实例化的代价微不足道，或者必然要实例化（而且推迟这个必然要做的事情显得不合适），就应该直接在声明时赋值，或者在构造器中赋值。将对象的初始化推迟到需要这个对象时进行，就称为**推迟初始化**。

高级主题：为泛型和Lambda表达式使用推迟加载

从.NET Framework 4.0开始，CLR添加了一个新类来帮助进行推迟初始化，这个类就是System.Lazy<T>。代码清单9-25演示了如何使用它。

代码清单9-24 使用System.Lazy<T>推迟加载属性

```
using System.IO;


class DataCache
{
  // ...

  public string FileStreamName { get; set; }

  public DataCache()
  {
      _FileStream = new Lazy<TemporaryFileStream>(
          () => new TemporaryFileStream(FileStreamName));
  }

  public TemporaryFileStream FileStream
  {
      get
      {
          return _FileStream.Value;
      }
  }
  private Lazy<TemporaryFileStream> _FileStream;

  // ...
}
```

System.Lazy<T>获取一个类型参数（T），这个参数标识了System.Lazy<T>的Value属性要返回的类型。不是将一个完全构造好的TemporaryFileStream赋给_FileStream字

段。相反，赋给它的是Lazy<TemporaryFileStream>的一个实例（一个轻量级的调用），将TemporaryFileStream本身的实例化推迟到访问Value属性（进而访问FileStream属性）的时候才进行。

如果除了类型参数（泛型）还使用了委托，甚至可以提供一个函数来指定在访问Value属性值时如何初始化对象。代码清单9-25演示了如何将委托（本例是一个Lambda表达式）传给System.Lazy<T>的构造器。

注意，Lambda表达式本身，即() => new TemporaryFileStream(FileStreamName)，是直到调用Value时才执行的。Lambda表达式提供了一种方式为将来发生的事情传递指令，但除非显式请求，否则那些指令是不会执行的。

9.9 小结

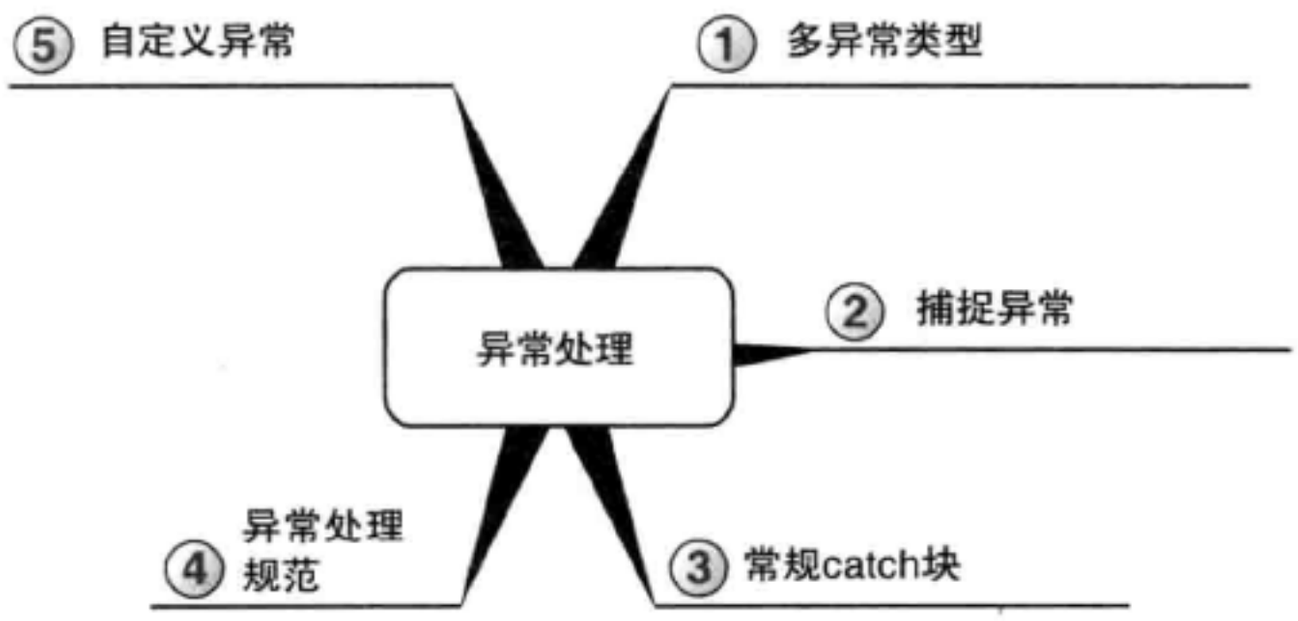
本章围绕如何构建可靠的类库展开了全面讨论。虽然所有主题都和内部开发相关，但它们对于构建可靠的类来说尤为关键。最终目标是构建可靠性和可编程性更好的API。同时涉及可靠性和可编程性的主题有“命名空间”和“垃圾回收”。涉及可编程性的其他主题还有“重写object的虚成员”、“操作符重载”以及“用XML注释来进行文档化”。

异常处理通过定义一个异常层次结构，并强迫自定义异常进入这个层次结构，从而充分地利用了继承。此外，C#编译器利用继承来验证catch块顺序。下一章将解释继承为什么是异常处理的核心部分。

第10章

异常处理

第4章讨论了如何使用try/catch/finally块执行标准异常处理。在那一章中，catch块总是捕捉System.Exception类型的异常。本章描述了异常处理的更多细节，具体包括其他异常类型、定义自定义异常类型以及用于处理每种异常类型的多个catch块。本章还详细描述了异常对继承的依赖。



10.1 多异常类型

代码清单10-1引发System.ArgumentException异常，而不是如第4章所述的System.Exception类型的异常。C#允许代码引发从System.Exception派生（无论直接还是间接）的任何异常类型。

代码要引发任何异常，只需为要引发的异常实例附加关键字throw作为前缀。具体选择的异常类型应该能够最好地说明异常发生的背景。

下面以代码清单10-1的TextNumberParser.Parse()方法为例。

代码清单10-1 引发异常

```
public sealed class TextNumberParser
{
  public static int Parse(string textDigit)
  {
      string[] digitTexts =
          { "zero", "one", "two", "three", "four",
            "five", "six", "seven", "eight", "nine" };

      int result = Array.IndexOf(
          digitTexts, textDigit.ToLower());

      if (result < 0)
      {
          throw new ArgumentException(
              "The argument did not represent a digit",
              "textDigit");
      }

      return result;
  }
}
```

程序不是引发System.Exception，而是引发更合适的ArgumentException，因为类型本身指出什么地方出错（参数异常），并包含了特殊的参数来指出具体是哪一个参数出错。

两个类似的异常是ArgumentNullException和NullReferenceException。前者应该在错误传递了null时引发。null是无效参数的特例。如果不为null，无效参数所引发的异常是ArgumentException或ArgumentOutOfRangeException。NullReferenceException一般只有在底层“运行时”解引用null值（想调用对象的成员，但发现对象的值为null）时引发。开发人员不要自己引发NullReferenceException。相反，应该在访问参数前检查它们是否为null，并在参数为null的前提下引发ArgumentNullException，从而提供更具体的上下文信息，如参数名等。

还有其他几个仅供“运行时”引发的、直接或间接从System.SystemException派生的异常，其中包括System.StackOverflowException、System.OutOfMemoryException、System.Runtime.InteropServices.COMException、System.ExecutionEngineException和System.Runtime.InteropServices.SEHException。不要自己引发这些异常。类似地，不要引发System.Exception或System.ApplicationException。它们过于泛泛，对指明问题的起因和问题的解决提供不了太多帮助。相反，要引发符合场景的、最具体的派生的异常。虽然开发人员应避免创建可能造成系统错误的API，但假如代码的执行达到一种再执行就会不安全或者不可恢复的状态，就应该果断地调用System.Environemnt.FailFast()。这样做会向Windows Application事件日志写入一条消息，然后立即终止进程。如果用户事先进行了选择，消息还会发送给“Windows错误报告”。

规范

要在向成员传递了错误参数时引发ArgumentException或者它的某个子类型。引发尽可能具体的异常（如ArgumentNullException）。

要在引发ArgumentException或者它的某个子类时设置ParamName属性。

要引发能说明问题的、最具体的异常（派生得最远的异常）。

不要引发NullReferenceException。相反，在值意外为空时引发ArgumentNullException。

不要引发System.SystemException或者它的派生类型。

不要引发System.Exception或者System.ApplicationException。

考虑在程序继续执行会变得不安全时调用System.Environment.FailFast()来终止进程。

10.2 捕捉异常

引发特定的异常类型，可以通过异常类型本身来识别（并解决）问题。换言之，不需要捕捉异常并使用一个switch语句，根据异常消息来决定要采取的操作。相反，C#允许使用多个catch块，每个块都面向一个具体的异常类型，如代码清单10-2所示。

代码清单10-2 捕捉不同的异常类型

```
using System;

public sealed class Program
{
  public static void Main(string[] args)
  {
      try
      {
              // ...
              throw new InvalidOperationException(
                  "Arbitrary exception");
              // ...
      }
      catch (NullReferenceException exception)
      {
          // Handle NullReferenceException
      }
      catch (ArgumentException exception)
      {
          // Handle ArgumentException
      }
      catch (InvalidOperationException exception)
      {
         bool exceptionHandled=false;
          // Handle InvalidOperationException
         // ...
          if(!exceptionHandled)
          {
```

```
                throw;
            }
        }
        catch (SystemException)
        {
            // Handle SystemException
        }
        catch (Exception exception)
        {
            // Handle Exception
            throw;
        }
        finally
        {
            // Handle any cleanup code here as it runs
            // regardless of an exception or not.
        }
    }
}
```

代码清单10-2总共有5个catch块，每个块都处理不同的异常类型。发生异常时，会跳转到与异常类型最匹配的catch块执行。匹配度由继承链决定。例如，即使引发的是System.Exception类型的异常，由于System.InvalidOperationException派生自System.Exception，这个“属于”关系就在继承链中发生了。因此InvalidOperationException与引发的异常匹配度最高。最终，将由catch(InvalidOperationException...)捕捉到异常，而不是由catch(Exception ...)块捕捉。

catch块必须按照从最具体到最常规的顺序排列，这样才能避免编译错误。例如，将catch (Exception ...)块移到其他任何一种异常的前面，都会造成编译错误。因为之前的所有异常都直接或间接从System.Exception派生。

如catch (SystemException){}这个catch块所示，catch块并非一定需要一个命名的参数。事实上，如下一节所述，最后一个catch甚至连类型参数都可以不要。

语言对比：Java——异常指示符

C#没有与Java的异常指示符相对应的东西。通过异常指示符，Java编译器可以验证一个函数（或者函数的调用层次结构）中引发的所有可能的异常要么已被捕捉到，要么被声明为可以重新引发。C#团队考虑过这种设计，但认为它的好处不足以抵消维护所带来的负担。所以，最终的决定是没有必要维持一个特定调用栈上所有可能异常的列表。当然，同时也不能轻松判断所有可能的异常。事实上，Java也做不到这一点。由于可能调用虚方法或者使用晚期绑定（比如反射），所以编译时不可能完整分析一个方法可能引发的异常。

注意，在捕捉InvalidOperationException的catch块中，有一个未注明要引发什么异常的throw语句（throw自己为一行语句）——虽然当前在catch块的范围内有一个异常实例（exception）可供重新引发。如果引发一个具体的异常，会更新所有栈信息来匹配

新的引发位置。这会造成指示异常最初发生的调用位置的所有栈信息丢失，使问题变得更难诊断。有鉴于此，C#支持不带有显式异常引用的throw语句，前提是只能在catch块中使用。这样，代码可以检查异常，判断是否能完整地处理该异常；如果不能，就重新引发异常（虽然没有显式指定这个动作）。结果是异常似乎从未被捕捉，也没有任何栈信息被替换。

高级主题：引发现有异常而不替换栈信息

C# 5.0新增了一个机制，允许引发从前引发的异常而不丢失原始异常中的栈跟踪信息。这样，即使在catch块外部也能重新引发异常，而不是像以前那样只能在catch块内部使用throw;语句来重新引发。虽然很少需要这样做，但偶尔需要包装或保存异常，直到catch块外部。例如，多线程代码可能用AggregateException包装异常。.NET 4.5 Framework提供了System.Runtime.ExceptionServices.ExceptionDispatchInfo类来专门处理这种情况。你需要使用它的静态方法Catch()和实例方法Throw()。代码清单10-3演示了如何在不重置栈跟踪信息或者不使用空throw语句的前提下重新引发异常。

代码清单10-3 使用ExceptionDispatchInfo重新引发异常

```
using System
using System.Runtime.ExceptionServices;

using System.Threading.Tasks;
Task task = WriteWebRequestSizeAsync(url);
try
{
  while (!task.Wait(100))
  {
      Console.Write(".");
  }
}
catch(AggregateException exception)
{
  exception = exception.Flatten();
  ExceptionDispatchInfo.Capture(
      exception.InnerException).Throw();
}
```

ExeptionDispatchInfo.Throw()值得注意的一个地方是，编译器不认为它是像普通throw语句那样的一个返回语句。例如，如果方法签名返回一个值，但没有值从ExceptionDispatchInfo.Throw()所在的代码路径返回，编译器就会报错，指出没有返回值。所以，开发人员有时不得不在ExceptionDispatchInfo.Throw()后面跟随一个return语句，即使这样一条语句在运行时永远都不会执行（相反是引发异常）。

10.3 常规catch块

C#要求代码引发的任何对象都必须从System.Exception派生。然而，并不是所有语言都有同样的要求。例如，C/C++允许引发任意对象类型，其中包括不是从System.Exception派生的托管异常。从C# 2.0开始，所有异常（不管是不是从System.Exception派生）在进入

程序集之后，都会被“包装”成从System.Exception派生。结果是捕捉System.Exception的catch块现在可以捕捉之前的块不能捕捉的所有异常。

C#还支持常规catch块，即catch{}，它在行为上和catch(System.Exception exception)块完全一致，只是没有类型名或变量名。除此之外，常规catch块必须是所有catch块的最后一个。由于常规catch块在功能上完全等价于catch(System.Exception exception)块，而且必须放在最后，所以在同一个try/catch语句中，假如这两个catch块同时出现，编译器就会显示一条警告消息，因为常规catch块永远都得不到调用。（参见下面的“高级主题：C# 1.0中的常规catch块”，更多地了解常规catch块。）

高级主题：C# 1.0中的常规catch块

在C# 1.0中，如果从一个方法调用中（该方法在一个不是用C#写的程序集中）引发不是从System.Exception派生的异常，这个异常不会被catch(System.Exception)捕捉到。例如，假如一个不同的语言引发一个string，异常就会进入未被处理的状态。为了避免这个问题，C#允许写一个无参数的catch块。C#团队将这个catch块称为常规catch块，如代码清单10-4所示。

代码清单10-4 捕捉任意异常类型

```
using System

public sealed class Program
{
  public static void Main()
  {
      try
      {
              // ...
              throw new InvalidOperationException (
                  "Arbitrary exception");
              // ...
      }
      catch (NullReferenceException exception)
      {
          // Handle NullReferenceException
      }
      catch (ArgumentException exception)
      {
          // Handle ArgumentException
      }
      catch (InvalidOperationException exception)
      {
          // Handle ApplicationException
       }
      catch (SystemException exception)
      {
          // Handle SystemException
      }
      catch (Exception exception)
      {
          // Handle Exception
      }
```

```
        catch
        {
            // Any unhandled exception
        }
        finally
        {
            // Handle any cleanup code here as it runs
            // regardless of an exception or not.
        }
    }
}
```

常规catch块捕捉先前的catch块没有捕捉到的所有异常，无论它们是不是从System.Exception派生。这个catch块的缺点在于没有一个可供访问的异常实例，所以没有办法确定最佳对策。另外，甚至没有办法确定像这样的一个异常是不是有害的。最好的对策就是在关闭应用程序之前，用一些清理代码去处理异常。例如，在关闭应用程序或者重新引发异常之前，常规catch块应该保存任何易失的数据。

高级主题：空catch块的内部原理

事实上，与空catch块对应的CIL代码是一个catch(object)块。这意味着不管引发什么类型，空catch块都能捕捉到它。有趣的是，不能在C#代码中显式声明catch(object)块。所以，没有办法捕捉一个非System.Exception派生的异常，并拿一个异常实例仔细检查问题。

具有讽刺意味的是，来自C++等语言的非托管异常通常会造成System.Runtime.InteropServices.SEHException类型的异常，它从System.Exception派生。所以，常规catch块（空catch块）不仅能捕捉非托管类型的异常，还能捕捉非System.Exception托管类型的异常，比如string类型。

10.4 异常处理的规范

异常处理为它之前的错误处理机制提供了急需的基本结构。然而，若使用不当，它仍有可能造成一些令人不快的后果。以下规范是异常处理的最佳实践。

- 只捕捉能处理的异常。通常，一些类型的异常可以处理，但另一些不能。例如，试图打开正在使用的文件进行独占式读/写访问，会引发一个System.IO.IOException，因为文件已经在使用中了。通过捕捉这种类型的异常，代码可以向用户报告该文件正在使用，并允许用户选择取消或者重试。只有那些已知操作的异常才应捕捉。其他异常类型应留给栈中较高的调用者去处理。
- 不要隐藏你不能完全处理的异常。新手程序员常犯的一个错误是捕捉所有异常，然后假装什么都没有发生，而不向用户报告未处理的异常。这有可能导致严重的系统问题逃过检测。除非代码执行显式的操作来处理一个异常，或者显式地确定一个异常无害，否则catch块应当重新引发异常，而不是在捕捉了异常之后在调用者面前隐藏它们。尤其要注意的是，catch(System.Exception)和常规catch块应放在调用栈中较高的位置，除非决定在块中重新引发异常。