- 尽可能少地使用System.Exception和常规catch块。几乎所有异常都是从System.Exception派生的。然而，处理某些System.Exception的最佳方式是不对它们进行处理，或者尽快以正常方式关闭应用程序。这些异常包括System.OutOfMemoryException和System.StackOverflowException等。在CLR 4中，这些异常默认为“不可恢复”。所以，如果捕捉它们但不重新引发它们，会造成CLR重新引发它们。这些异常是运行时异常，开发人员不能写代码从这些异常中恢复。所以，最佳对策就是关闭应用程序——在CLR 4和更高版本中，这是“运行时”会强制采取的操作。CLR 4之前的代码在捕捉这种异常后，也只应运行清理或紧急代码（比如保存任何易失的数据），然后马上关闭应用程序，或者使用throw;语句重新引发异常。
- 避免在调用栈较低的位置报告或记录异常。新手程序员倾向于异常一发生就记录它，或者向用户报告它。然而，由于当前正处在调用栈中较低的位置，而这些位置很少能够完整地处理异常，所以只好重新引发异常。像这样的catch块不应记录异常，也不应向用户报告。假如异常被记录，然后又被重新引发（调用栈中较高的调用者可能做同样的事情），就会造成重复出现的异常记录项。更糟的是，取决于应用程序的类型，向用户显示异常可能并不合适。例如，在Windows应用程序中使用System.Console.WriteLine()，用户永远看不到显示的内容。类似地，在无人值守的命令行进程中显示对话框，可能根本不会被人看到，而且可能使应用程序冻结在这个位置。日志记录和与异常相关的用户界面应该保留到调用栈中较高的位置。
- 在catch块中使用throw;而不是throw <异常对象>语句。可以在catch块中重新引发异常。例如，在catch(ArgumentNullException exception)的实现中，可以包含对throw exception的调用。然而，像这样重新引发异常，会将栈追踪重置为重新引发的位置，而不是重用原始引发位置。所以，只要不是重新引发不同的异常类型，或者不是要故意隐藏原始调用栈，就应该使用throw;语句，允许相同的异常在调用栈中向上传播。
- 重新引发不同异常时要小心。在catch块中重新引发不同的异常，那么不仅会重置引发点，还会隐藏原始异常。为了保留原始异常，需要设置新异常的InnerException属性（该属性通常可以通过构造器来赋值）。只有以下情况才可以重新引发不同的异常。

a) 更改异常类型可以更好地澄清问题。例如，在对Logon(User user)的一个调用中，假如遇到用户列表文件不能访问的情况，那么重新引发一个不同的异常类型要比传播System.IO.IOException更合适。

b) 私有数据是原始异常的一部分。在上面的例子中，假如文件路径包含在原始的System.IO.IOException中，就会暴露敏感的系统信息，所以应该使用其他异常类型来包装它。当然，前提是原始异常没有设置InnerException属性。有趣的

是，CLR v1的一个非常早的版本（比alpha还要早的一个版本）有一个异常会报告这样的消息：“安全异常：没有足够权限确定c:\temp\foo.txt的路径”。

c) 异常类型过于具体，以至于调用者不能恰当地处理。例如，不要引发数据库系统的专有异常。相反，可以使用一个较为泛化的异常，避免在调用栈较高的位置写数据库的专有代码。

规范

避免在调用栈较低的位置报告或记录异常。

不要捕捉不应该捕捉的异常。要允许异常在调用栈中向上传播，除非能非常清楚地知道如何通过程序准确地定位栈中较低位置的错误。

如果理解特定异常在给定的上下文中为何引发，并能通过程序响应错误，就**考虑**捕捉该异常。

避免捕捉System.Exception或 System.SystemException，除非是在顶层异常处理程序中在重新引发异常之前执行最后的清理操作。

要在catch块中使用throw;而不是throw <异常对象>语句。

重新引发不同的异常时**要**小心。

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

10.5 定义自定义异常

在必须引发异常时，首选的方案是使用.NET Framework的异常，因为它们得到了良好的构建，能够被很好地理解。例如，首选的不是引发自定义的“无效参数”异常，而是引发System.ArgumentException。然而，假如使用特定API的开发人员想要执行特殊操作（例如，处理自定义异常的逻辑有所不同），就可以考虑定义一个自定义异常。例如，假定某个地图API接收到邮政编码无效的一个地址，那么不是引发System.ArgumentException。相反，更好的做法也许是引发自定义的InvalidAddressException。这里的关键在于调用者是否愿意编写专门的InvalidAddressE xception catch块来进行特殊处理，而不是用一个常规的System.ArgumentException catch块。

定义一个自定义异常时，从System.Exception或者其他异常类型派生就可以了。代码清单10-5展示了一个例子。

代码清单10-5 创建自定义异常

```
class DatabaseException : System.Exception
{
  public DatabaseException(
      System.Data.SqlClient.SQLException exception)
  {
      InnerException = exception;
      // ...
```

```
    }

    public DatabaseException(
        System.Data.OracleClient.OracleException exception)
    {
        InnerException = exception;
        // ...
    }

    public DatabaseException()
    {
        // ...
    }

    public DatabaseException(string message)
    {
        // ...
    }

    public DatabaseException(
        string message, Exception innerException)
    {
        InnerException = innerException;
        // ...
    }
}
```

这个自定义异常包装了多个专门的数据库异常。例如，因为Oracle和SQL Server会为类似的错误引发不同的异常，所以可以在应用程序中定义一个自定义异常，将不同数据库特有的异常标准化到一个通用的异常封装器中，使应用程序能采取标准方式处理相似的异常。这样，不管应用程序使用的后端数据库是Oracle还是SQL Server，都可以在调用栈较高的位置用同一个catch块处理相似的错误。

自定义异常唯一的要求是必须从System.Exception或者它的某个子类派生。除此之外，在使用自定义异常的时候，还应遵照以下最佳实践。

- 所有异常都应该使用“Exception”后缀，这样一来，从名称上就很容易得知它们的用途。
- 通常，所有异常都应包含以下3个构造器：无参构造器、获取一个string参数的构造器以及同时获取一个字符串和一个内部异常作为参数的构造器。除此之外，由于异常通常是在引发它们的那个语句中构造的，所以也应允许其他任何异常数据成为构造器的一部分。（当然，假如特定的数据是必需的，而构造器回避了这个要求，则不能创建该构造器。）
- 避免使用深的继承层次结构（一般应该小于5级）。

重新引发一个与已捕捉到的异常不同的异常时，内部异常将发挥重要作用。例如，假定一个数据库调用引发了System.Data.SqlClient.SqlException，但这个异常在数据访问层捕捉到，并作为一个DatabaseException重新引发，那么获取SqlException（或者内部异常）的DatabaseException构造器会将原始SqlException保存到InnerException属性中。这样，在请求与原始异常有关的附加细节时，开发者就可以从InnerException属性中获取异常。

规范

如果异常不以有别于现有CLR异常的方式进行处理，就**不要**创建新异常。相反，应该引发现有的框架异常。

要创建新异常类型来描述特别的程序错误。这种错误用现有的CLR异常无法描述，而且能通过程序以不同于现有CLR异常类型的方式进行处理。

要为所有自定义异常类型提供无参构造器。还要提供获取消息和内部异常作为参数的构造器。

要为异常类的名称附加“Exception”后缀。

要使异常能由“运行时”序列化。

考虑提供异常属性，以便通过程序访问异常的额外信息。

避免使用过深的异常继承层次结构。

高级主题：可序列化异常

可序列化对象（serializable object）是“运行时”可以持久化成一个流（例如文件流），然后根据这个流来重新实例化的对象。分布式通信可能要求异常使用这个技术。为了支持序列化，异常声明应包含System.SerializableAttribute特性或者实现ISerializable接口。此外，必须包含一个构造器来获取System.Runtime.Serialization.SerializationInfo和System.Runtime.Serialization.StreamingContext作为参数。代码清单10-6展示了使用System.SerializableAttribute的一个例子。

代码清单10-6 定义可序列化异常

```
// Supporting serialization via an attribute
[Serializable]
class DatabaseException : System.Exception
{
  // ...

  // Used for deserialization of exceptions
  public DatabaseException(
      SerializationInfo serializationInfo,
      StreamingContext context)
  {
      //...
  }

}
```

这个DatabaseException例子演示了可序列化对象对于特性和构造器的要求。

10.6 封装异常并重新引发

有时，栈中较低位置引发的异常在较高处捕捉到时已经没有意义了。例如，假定服务器上因为系统的磁盘空间耗尽而引发的System.IO.IOException。捕捉到该异常的客户端理解不了为什么居然会有I/O活动。类似地，假定地理坐标请求API引发System.

UnauthorizedAccessException（和调用的API完全无关的异常），调用者理解不了API调用和安全性有什么关系 。从调用API的代码的角度看，这些异常带来的更多是困惑而不是帮助。所以不应该向客户端公开这些异常，而是捕捉异常，并引发一个不同的异常，比如InvalidOperationException（或者自定义异常），从而告诉用户系统处于无效状态。在这种情况下，一定要设置封装异常的InnerException属性（一般是通过构造器调用，例如new InvalidOperationException(String, Exception)）。这样，就有一个额外的上下文可以供那些更接近所调用框架的人进行诊断。

封装并重新引发异常时要记住一个重要细节，即原始的栈跟踪（提供了原始异常引发位置的上下文）将被替换成新的栈跟踪（提供了封装异常引发位置的上下文）。幸好，将原始异常嵌入封装异常，原始的栈跟踪仍然可用。

最后，记住异常的预期接收者是写代码调用（可能是以不正确的方式调用）你的API的程序员。所以，要提供尽可能多的信息来描述他哪里做错了以及如何修正（后者可能更重要）。异常类型是这个沟通机制中的重要一环。所以，要精心选择类型。

规范

如果低层引发的异常在高层操作的上下文中没有意义，考虑将低层异常封装到更恰当的异常中。

要在封装异常时设置内部异常属性。

要将开发人员作为异常的接收者，尽量说清楚问题和解决问题的机制。

要在重新引发相同的异常时使用空throw语句（throw;），而不是向throw传递异常作为参数。

初学者主题：checked和unchecked转换

正如第2章的“高级主题：checked和unchecked转换”讨论过的那样，C#提供了特殊的关键字来标识代码块，指出假如目标数据类型太小，以至于不能包含所赋的数据，那么“运行时”应该怎么办。默认情况下，假如目标数据类型容不下所赋的数据，那么数据会被截短。代码清单10-7展示了一个例子。

代码清单10-7 溢出一个整数值

```
using System;

public class Program
{
  public static void Main()
  {
      // int.MaxValue equals 2147483647
      int n = int.MaxValue;
      n = n + 1 ;
      System.Console.WriteLine(n);
  }
}
```

输出10-1展示了结果。

输出10–1

```
-2147483648
```

它会在控制台上显示值-2147483648。然而，将代码放到checked块中或者在编译时使用checked选项就会造成“运行时”引发System.OverflowException类型的异常。checked块使用关键字checked，它的语法如代码清单10-8所示。

代码清单10–8 checked块示例

```
using System;

public class Program
{
  public static void Main()
  {
      checked
      {
          // int.MaxValue equals 2147483647
          int n = int.MaxValue;
          n = n + 1 ;
          System.Console.WriteLine(n);
      }
  }
}
```

如果计算只涉及常量，那么计算默认就是checked的。输出10-2展示了代码清单10-8的结果。

输出10–2

```
Unhandled Exception: System.OverflowException: Arithmetic operation
resulted in an overflow. at Program.Main() in ...Program.cs:line 12
```

除此之外，依据Windows版本以及是否安装了调试器，可能会出现一个对话框，提示用户选择向微软发送错误信息、检查解决方案或者对应用程序进行调试。另外，只有在调试编译中才会出现位置信息（Program.cs:行号*X*）。为了进行调试编译，请使用微软的csc.exe编译器的/Debug选项。

在checked块内，如果在运行时发生溢出的赋值，就会引发异常。

C# 编译器提供了命令行选项将默认行为从unchecked改成checked。除此之外，C#还支持unchecked块，它会将数据截短，不会报告溢出错误，如代码清单10-9所示。

代码清单10–9 unchecked块示例

```
using System;

public class Program
{
  public static void Main()
  {
      unchecked
      {
          // int.MaxValue equals 2147483647
          int n = int.MaxValue;
```

```
                n = n + 1 ;
                System.Console.WriteLine(n);
            }
        }
    }
```

输出10-3展示了代码清单10-9的结果。

输出10–3

```
-2147483648
```

即使编译时打开了checked选项，上述代码在执行期间，unchecked关键字也会阻止“运行时”引发异常。

如果不允许使用语句（比如在初始化字段的时候），那么可以使用checked和unchecked表达式，如下例所示：

```
int _Number = unchecked(int.MaxValue + 1);
```

10.7 小结

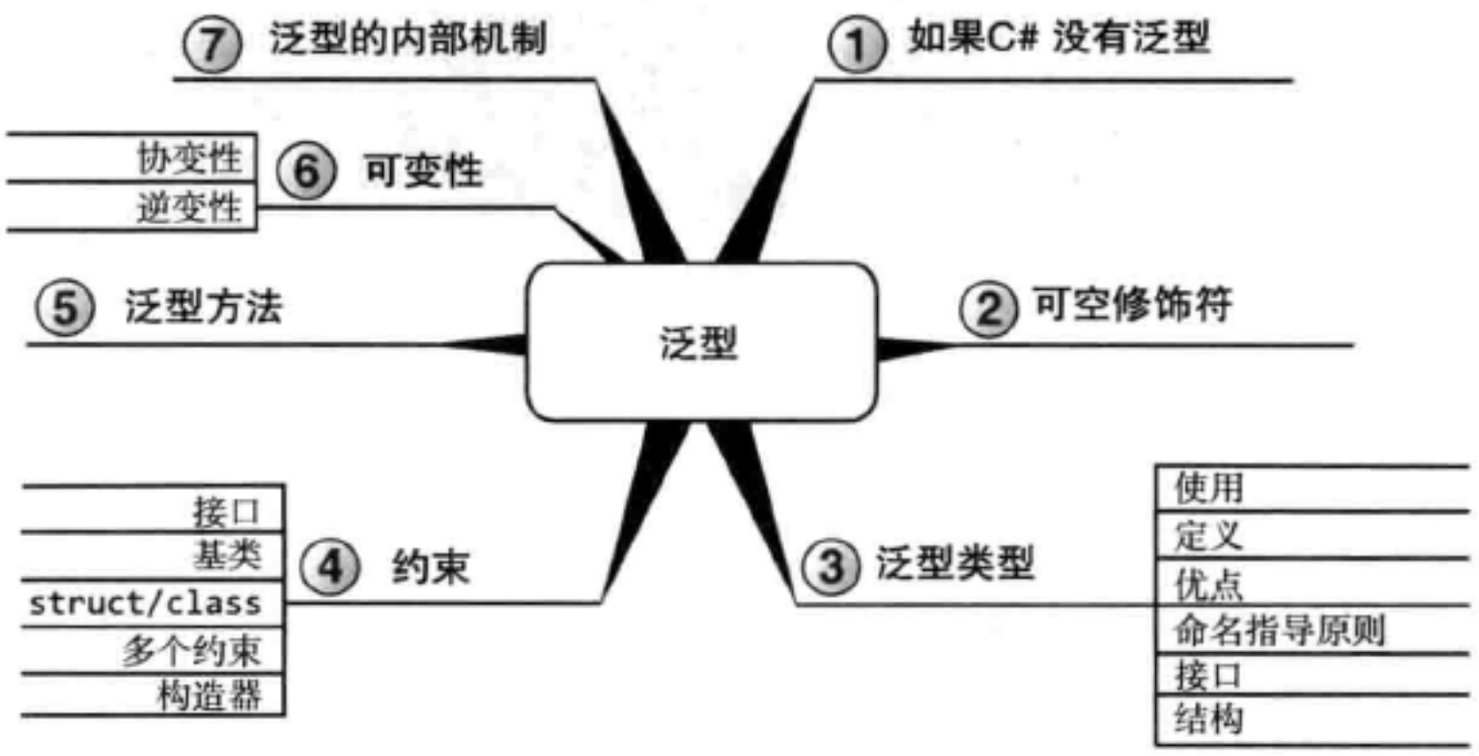
引发异常会显著影响性能。发生异常后，需要加载和处理大量额外的运行时栈信息，整个过程会花费可观的时间。正如第4章指出的那样，只应使用异常处理异常情况。API应提供相应的机制来检查是否引发了异常，而不是非要调用特定API来判断是否引发异常。

下一章将介绍泛型，这是从C# 2.0开始加入的功能，能显著增强用C# 1.0写的代码。事实上，它使System.Collections命名空间几乎完全失去了用处，而以前几乎每个项目都使用了这个命名空间。

第11章

泛　型

随着项目日趋复杂，需要用更好的方式重用和定制现有软件。C#通过泛型来促进代码重用，尤其是算法的重用。方法因为能获取参数而强大；类似地，类型和方法也会因为能获取类型参数而变得强大。



泛型在词义上等价于Java中的泛型类型和C++中的模板。在这三种语言中，该功能都使算法和模式只需实现一次，而不必为每个类型都实现一次。然而，与Java中的泛型和C++中的模板相比，C#中的泛型在实现细节和对类型系统的影响方面差异甚大。

注意，泛型是自C# 2.0起引入才添加到“运行时”和C#中的。

11.1　如果C# 没有泛型

开始讨论泛型之前，先看看一个没有使用泛型的类。这个类是System.Collections.Stack，用于表示一个对象集合，使加入集合的最后一项是从集合中获取的第一项（称为后进先出或LIFO）。Push()和Pop()是Stack类的两个主要方法，分别用于在栈中添加和移除数据项。代码清单11-1展示了Stack类的Pop()和Push()方法的声明。

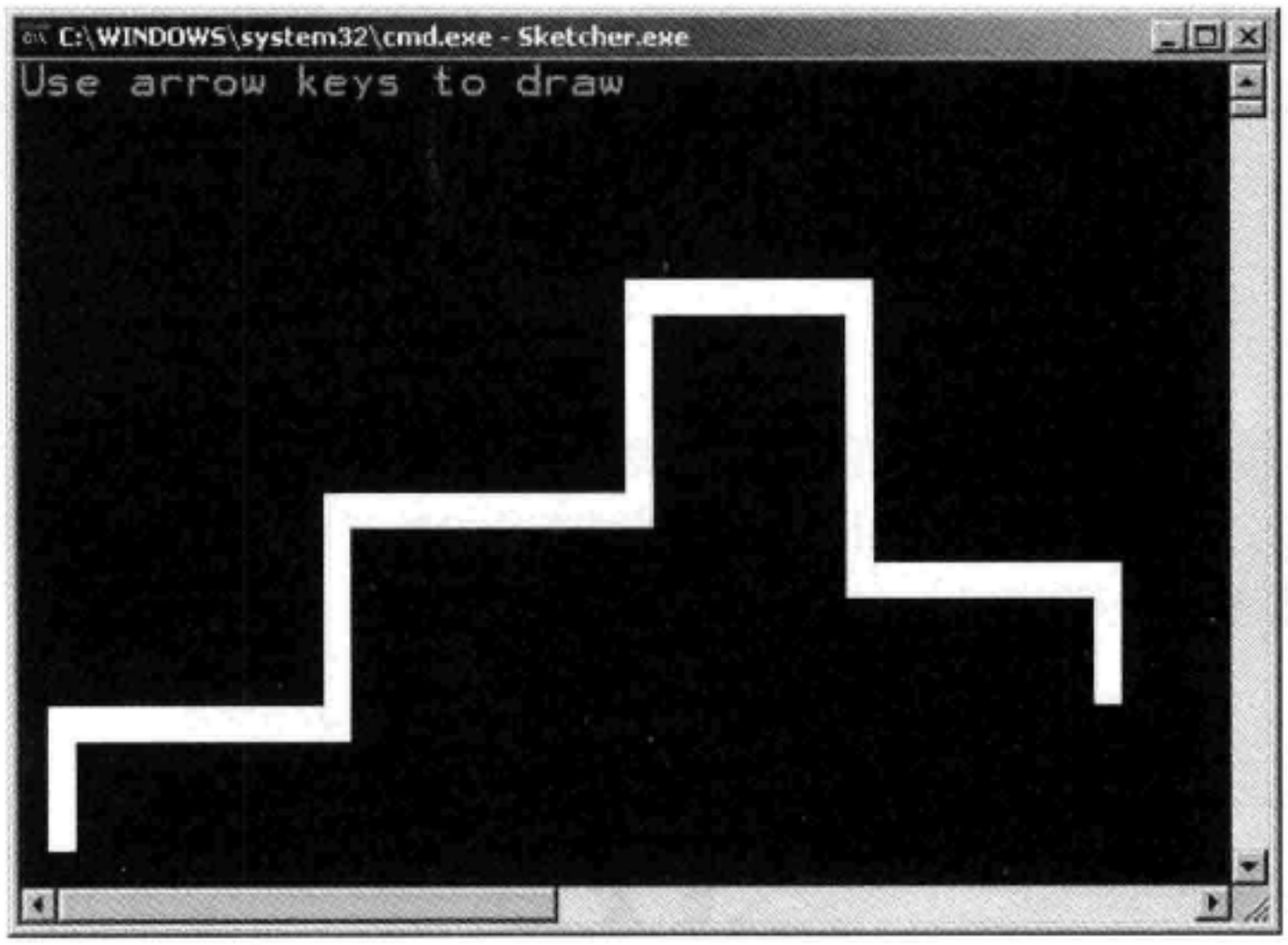
代码清单11-1 System.Collections.Stack类的方法签名

```
public class Stack
{
   public virtual object Pop() { ... }
   public virtual void Push(object obj) { ... }
   // ...
}
```

程序经常使用Stack类型的集合来实现多次撤销（undo）操作。例如，代码清单11-2利用Stack类在Etch A Sketch®游戏程序中执行撤销操作。

代码清单11-2的结果如输出11-1所示。

输出11-1



代码清单11-2 在类似Etch A Sketch游戏的程序中模拟撤销操作

```
using System;
using System.Collections;

class Program
{
  // ...

public void Sketch()
{
    Stack path = new Stack();
    Cell currentPosition;
    ConsoleKeyInfo key;  // Added in C# 2.0

    do
    {
        // Etch in the direction indicated by the
        // arrow keys that the user enters.
        key = Move();

        switch (key.Key)
```

```
{
    case ConsoleKey.Z:
        // Undo the previous Move.
        if (path.Count >= 1)
        {
            currentPosition = (Cell)path.Pop();
            Console.SetCursorPosition(
                currentPosition.X, currentPosition.Y);
            Undo();
        }
        break;

    case ConsoleKey.DownArrow:
    case ConsoleKey.UpArrow:
    case ConsoleKey.LeftArrow:
    case ConsoleKey.RightArrow:
        // SaveState()
        currentPosition = new Cell(
            Console.CursorLeft, Console.CursorTop);
        path.Push(currentPosition);
        break;

    default:
        Console.Beep();  // Added in C# 2.0
        break;
}

}
while (key.Key != ConsoleKey.X);  // Use X to quit.
  }
}

public struct Cell
{
    readonly public int X;
    readonly public int Y;
    public Cell(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

path是声明为System.Collections.Stack的一个变量。为了利用path来保存上一次移动，只需使用path.Push(currentPosition)方法，将一个自定义类型Cell传入Stack.Push()方法。如果用户输入Z（或者Ctrl+Z），表明需要撤销上一次移动。为此，程序使用一个Pop()方法获取上一次移动，将光标位置设为上一个位置，然后调用Undo()。虽然代码能够正常工作，但System.Collections.Stack类存在重大缺陷。如代码清单11-1所示，Stack类收集object类型的变量。由于CLR的每个对象都从object派生，所以Stack无法验证放到其中的元素是不是希望的类型。例如，传递的可能不是currentPosition而是string。在这个string中，X和Y坐标通过小数点拼接到一起。不过，编译器必须允许不一致的数据类型。因为Stack类被设计成获取任意对象，其中包括较具体类型的对象。

此外，使用Pop()方法从栈中获取数据时，必须将返回值转型为Cell。但是，假如

Pop()返回的不是Cell，就会引发异常。通过强制类型转换将类型检查推迟到运行时进行，程序变得更加脆弱。在不用泛型的情况下创建支持多种数据类型的类，根本的问题在于它们必须支持一个公共基类（或接口），通常是object。

为使用object的方法使用struct或整数这样的值类型，问题会变得更糟。将值类型的实例传给Stack.Push()方法，“运行时”将自动对它进行装箱。类似地，获取值类型的实例时需要显式对数据进行拆箱，将从Pop()获取的object引用转型为值类型。引用类型转换为基类或接口对性能的影响可忽略不计，但值类型装箱的开销较大，因为必须分配内存、复制值以及进行垃圾回收。

C#鼓励“类型安全”。许多类型错误（比如将整数赋值给string变量）能在编译时捕捉到。目前的根本问题是Stack类不是类型安全的。为了修改Stack类来确保类型安全，强迫它存储特定的数据类型，在不使用泛型的前提下，只能创建一个特殊的Stack类，如代码清单11-3所示。

代码清单11-3 定义特殊Stack类

```
public class CellStack
{
  public virtual Cell Pop();
  public virtual void Push(Cell cell);
  // ...
}
```

由于CellStack只能存储Cell类型的对象，所以这个解决方案要求对栈的各个方法进行自定义的实现，所以并不是理想的解决方案。例如，为了实现类型安全的整数栈，就需要另一个自定义实现。每一个实现看起来都差不多。最终将产生大量重复的、冗余的代码。

初学者主题：另一个例子——可空值类型

第2章讲过，在声明值类型的变量时，可以使用可空修饰符？ 声明允许包含null值的变量。C# 从2.0开始支持这个功能，它需要泛型才能正确实现。在引入泛型之前，程序员主要有两个选择。

第一个选择是为需要处理null值的每个值类型都声明可空数据类型，如代码清单11-4所示。

代码清单11-4 为各个值类型声明可以存储null的版本

```
struct NullableInt

{
    /// <summary>
    /// Provides the value when HasValue returns true.
    /// </summary>
    public int Value{ get; private set; }

    /// <summary>
    /// Indicates whether there is a value or whether
    /// the value is "null"
    /// </summary>
    public bool HasValue{ get; private set; }

    // ...
```

```
    }

    struct NullableGuid
{
    /// <summary>
    /// Provides the value when HasValue returns true.
    /// </summary>
    public Guid Value{ get; private set; }

    /// <summary>
    /// Indicates whether there is a value or whether
    /// the value is "null"
    /// </summary>
    public bool HasValue{ get; private set; }

    ...
}
...
```

代码清单11-4只显示了NullableInt和NullableGuid的实现。如果程序需要更多的可空值类型，就不得不创建更多的结构，并修改属性来使用所需的值类型。如果可空值类型的实现发生了改变（例如，为了支持一个用户定义的从基础类型向可空类型的隐式转换），就不得不修改所有可空类型声明。

第二个选择是声明可空类型，在其中包含object类型的Value属性，如代码清单11-5所示。

代码清单11-5 声明可空类型，其中包含object类型的Value属性

```
struct Nullable
{
    /// <summary>
    /// Provides the value when HasValue returns true.
    /// </summary>
    public object Value{ get; private set; }

    /// <summary>
    /// Indicates whether there is a value or whether
    /// the value is "null"
    /// </summary>
    public bool HasValue{ get; private set; }

    ...
}
```

虽然这个方案只需可空类型的一个实现，但“运行时”在设置Value属性时总是对值类型进行装箱。此外，从Value属性获取基础值需要进行一次强制类型转换，而这个操作在运行时可能无效。

以上两种方案都不理想。为了解决这个问题，C# 2.0引入了泛型的概念。事实上，可空类型是作为泛型类型Nullable<T>实现的。

11.2 泛型类型概述

可利用泛型创建一个数据结构，该数据结构能进行特化以处理特定的类型。程序员定义这种**参数化类型**，使泛型类型的每个变量都具有相同的内部算法，但数据类型和方法签

名可随类型参数而变。

为了减轻开发者的学习负担，C# 的设计者选择了与C++模板相似的语法。所以， C#中的泛型类和结构要求使用尖括号声明泛型类型参数以及指定泛型类型实参[①]。

11.2.1 泛型类的使用

代码清单11-6展示了如何指定泛型类使用的实际类型。为了指示path变量使用Cell类型，在实例化和声明语句中都要用尖括号表示法指定Cell。换言之，使用泛型数据类型声明变量（本例是path）时，C#要求指定泛型类型使用的类型实参。代码清单11-6展示了新的泛型Stack类。

代码清单11-6 使用泛型Stack类实现撤销

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

class Program
{
  // ...

  public void Sketch()
  {
        Stack<Cell> path;           // Generic variable declaration
        path = new Stack<Cell>();   // Generic object instantiation
        Cell currentPosition;
        ConsoleKeyInfo key;

        do
        {
            // Etch in the direction indicated by the
            // arrow keys entered by the user.
            key = Move();

            switch (key.Key)
            {
                case ConsoleKey.Z:
                    // Undo the previous Move.
                    if (path.Count >= 1)
                    {
                        // No cast required.
                        currentPosition = path.Pop();
                        Console.SetCursorPosition(
                            currentPosition.X, currentPosition.Y);
                        Undo();
                    }
                    break;

                case ConsoleKey.DownArrow:
                case ConsoleKey.UpArrow:
                case ConsoleKey.LeftArrow:
                case ConsoleKey.RightArrow:
                    // SaveState()
                    currentPosition = new Cell(
                        Console.CursorLeft, Console.CursorTop);
                    // Only type Cell allowed in call to Push().
```

① 本章很少区分类型参数和类型实参（实际上作者也不是特别严格），因为不影响理解。——译者注

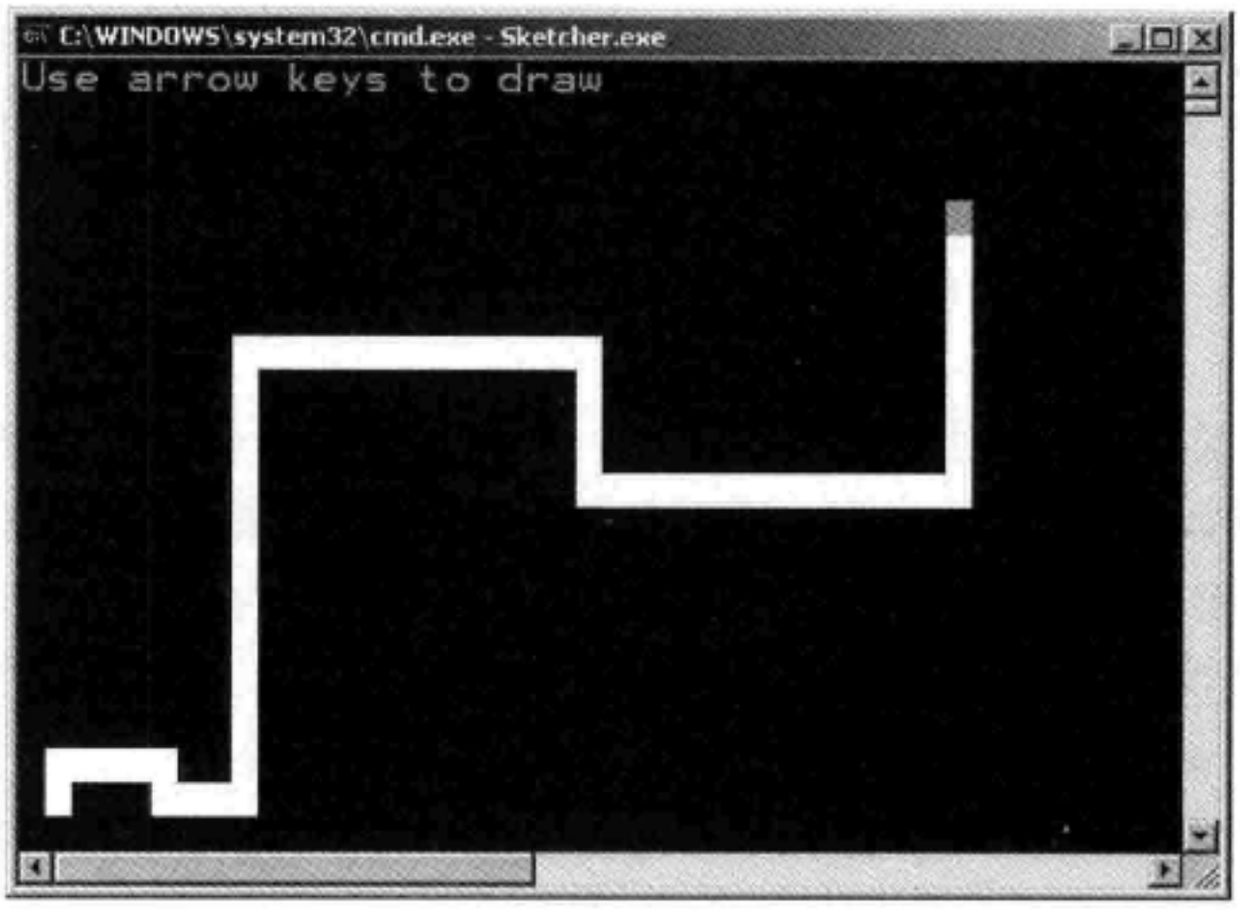
```
                    path.Push(currentPosition);
                    break;
                default:

                    Console.Beep();  // Added in C# 2.0
                    break;
            }

        } while (key.Key != ConsoleKey.X);  // Use X to quit.
    }
}
```

代码清单11-6的结果如输出11-2所示。

输出11-2



代码清单11-6中将path声明为System.Collections.Generic.Stack<Cell>类的变量，并用新实例初始化它。尖括号中指定栈中的元素类型为Cell。结果是添加到path以及从path取回的每个对象都是Cell类型。所以，不再需要对path.Pop()的返回值进行转型，也不需要在Push()方法中确保只有Cell类型的对象才能添加到path中。

11.2.2 简单泛型类的定义

泛型允许开发人员把精力放在创建算法和模式上，并确保代码能由不同数据类型重用。代码清单11-7创建了泛型Stack<T>类，它与代码清单11-6使用的System.Collections.Generic.Stack<T>类相似。在类名之后，需要在一对尖括号中指定**类型参数**（本例是T）。然后，可以向泛型Stack<T>提供类型实参，它将“替换”类中出现的每个T。这样，栈就可以存储指定的任何类型的数据项，不需要重复代码，也不需要将数据项转换成object。在代码清单11-7中可以看到将类型参数T用于内部Items数组、Push()方法的参数类型以及Pop()方法的返回类型。

代码清单11-7 声明泛型类Stack<T>

```
public class Stack<T>
{
    private T[] _Items;
```

```
        public void Push(T data)
        {
            ...
        }

        public T Pop()
        {
            ...
        }
    }
```

11.2.3 泛型的优点

使用泛型类而不是非泛型版本（比如使用System.Collections.Generic.Stack<T>类而不是原始的System.Collections.Stack类型）有以下几个方面的优点。

（1）泛型促进了类型安全。它确保在参数化的类中，只有成员明确希望的数据类型才可使用。在代码清单11-7中，参数化栈类限制为Stack<Cell>的所有实例使用Cell数据类型。例如，执行path.Push("garbage")会造成编译时错误，指出没有System. Collections. Generic.Stack<T>.Push(T)方法的哪一个重载版本可以处理字符串“garbage”，因为它不能被转换成Cell。

（2）编译时类型检查减小了在运行时发生InvalidCastException异常的几率。

（3）为泛型类成员使用值类型，不再造成到object的装箱转换。例如，path.Pop()和path.Push()不需要在添加一个项时装箱，或者在删除一个项时拆箱。

（4）C#泛型缓解了代码膨胀的情况。泛型类型保持了具体类版本的优势，但没有具体类版本的开销（例如，没有必要定义像CellStack这样的一个类）。

（5）性能得到了提高。一个原因是不再需要从object的强制类型转换，从而避免了类型检查。另一个原因是不再需要为值类型执行装箱。

（6）泛型减小了内存消耗。由于避免了装箱，因此减少了堆上的内存的消耗。

（7）代码的可读性更好。一个原因是转型检查次数变少了。另一个原因是现在需要较少的类型特定的实现。

（8）支持IntelliSense®的代码编辑器现在能直接处理来自泛型类的返回参数。没有必要为了使IntelliSense工作起来，而对返回数据执行转型。

最核心的是，泛型允许写代码来实现模式，并在以后出现这种模式的时候重用那个实现。模式描述了在代码中反复出现的问题，而泛型类型为这些反复出现的模式提供了单一的实现。

11.2.4 类型参数命名规范

和方法参数的命名相似，类型参数的命名应该尽量具有描述性。除此之外，为了强调它是类型参数，名称应包含T前缀。例如，在定义诸如EntityCollection<TEntity>的类时，使用的类型参数名称是TEntity。

唯一不需要使用描述性类型参数名称的时候是描述没有意义的时候。例如，在Stack<T>中使用T就够了，因为T足以说明问题——栈适合任意类型。

11.3节会介绍约束。使用约束描述类型名称是一种良好的编程习惯。例如，假定类型参数必须实现IComponent，则类型名称可以是“TComponent”。

规范

要为类型参数选择有意义的名称，并为名称附加“T”前缀。

考虑在类型名称中指明约束。

11.2.5 泛型接口和结构

C#支持在语言中全面地使用泛型，其中包括接口和结构。语法和类的语法完全相同。要声明包含类型参数的接口，将类型参数放到接口名称后面的一对尖括号中即可，比如代码清单11-8 中的IPair<T>。

代码清单11-8 声明泛型接口

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

该接口代表一对相似对象，比如一个点的平面坐标、一个人的生身父母，或者一个二叉树的节点，等等。pair中的两个数据项具有相同类型。

实现接口的语法与非泛型类的语法相同。注意，一个泛型的类型实参可以成为另一个泛型类型的类型参数。这既合法又普遍，如代码清单11-9所示。接口的类型实参是类声明的类型参数。除此之外，这个例子使用了结构而不是类，表明C# 支持自定义的泛型值类型。

代码清单11-9 实现泛型接口

```
public struct Pair<T>: IPair<T>
{
    public T First
    {
        get
        {
            return _First;
        }
        set
        {
            _First = value;
        }
    }
    private T _First;

    public T Second
    {
        get
        {
            return _Second;
        }
        set
        {
            _Second = value;
        }
```

```
        }
        private T _Second;
    }
```

对泛型接口的支持对于集合类尤其重要。使用泛型最多的地方就是集合类。假如没有泛型，开发者就要依赖于System.Collections命名空间中的一系列接口。和它们的实现类一样，这些接口只能使用object类型，因此，此接口要求进出这些集合类的所有访问都要执行转型。使用类型安全的泛型接口，就可以避免执行转型。

高级主题：在类中多次实现同一个接口

相同泛型接口的不同构造被看成是不同的类型，所以类或结构能多次实现“同一个”泛型接口。来看看代码清单11-10的例子。

代码清单11-10 在类中重复一个接口实现

```
public interface IContainer<T>
{
    ICollection<T> Items
    {
        get;
        set;
    }
}

public class Person: IContainer<Address>,
    IContainer<Phone>, IContainer<Email>
{
    ICollection<Address> IContainer<Address>.Items
    {
        get{...}
        set{...}
    }
    ICollection<Phone> IContainer<Phone>.Items
    {
        get{...}
        set{...}
    }
    ICollection<Email> IContainer<Email>.Items
    {
        get{...}
        set{...}
    }
}
```

在这个例子中，Items属性通过显式的接口实现多次出现，每次类型参数都有所不同。没有泛型这是不可能的。在没有泛型的情况下，编译器只允许一个显式的IContainer.Items属性。

然而，像这样实现“同一个”接口的多个版本不是好的编码风格，因为它会造成混淆（尤其是在接口允许协变或逆变转换的情况下）此外， Person类的设计似乎也有问题，因为一般不会认为人是“能提供一组电子邮件地址”的东西。与其实现同一个接口的三个版本，不如实现3个属性：EmailAddresses、PhoneNumbers和MailingAddresses，每个属性都返回泛型接口的相应构造。

规范

避免在类型中实现同一个泛型接口的多个构造。

11.2.6 构造器和终结器的定义

令人惊讶的是，泛型类或结构的构造器（和终结器）不要求类型参数。换言之，不要求写成Pair<T>(){...}这样的形式。在代码清单11-11的数对例子中，构造器声明为public Pair(T first, T second)。

代码清单11-11 声明泛型类型的构造器

```
public struct Pair<T>: IPair<T>
{
  public Pair(T first, T second)
  {
      _First = first;
      _Second = second;
  }

  public T First
  {
      get{ return _First; }
      set{ _First = value; }
  }
  private T _First;

  public T Second
  {
      get{ return _Second; }
      set{ _Second = value; }
  }
  private T _Second;
}
```

11.2.7 默认值的指定

在代码清单11-11中，构造器获取First和Second的初始值，并把它们赋给_First和_Second。由于Pair<T>是结构，因此提供的任何构造器都必须对全部字段进行初始化。但这会带来一个问题。假定有一个Pair<T>的构造器，它在实例化的时候，只对数对（pair）的一半进行初始化。

如代码清单11-12所示，定义这样的构造器会造成编译错误，因为在构造结束的时候，字段_Second仍然处于未初始化的状态。对_Second进行初始化有一个问题，因为不知道T的数据类型。如果是引用类型，那么可以使用null来初始化。然而，假如T是非空值类型，使用null进行初始化就行不通了。

代码清单11-12 不初始化所有字段，造成编译错误

```
public struct Pair<T>: IPair<T>
{
  // ERROR:  Field 'Pair<T>._second' must be fully assigned
  //         before control leaves the constructor
```

```
    // public Pair(T first)
    // {
    //     _First = first;
    // }

    // ...
}
```

为了应对这样的局面，C# 提供了default操作符，这个操作符最早是在第8章讨论的。第8章讲过，可以使用default(int)指定int的默认值。对于T，可以使用default(T)来初始化_Second，如代码清单11-13所示。

代码清单11-13 用default操作符初始化字段

```
public struct Pair<T>: IPair<T>
{
  public Pair(T first)
  {
      _First = first;
      _Second = default(T);
  }

  // ...
}
```

default操作符可提供任意类型的默认值，包括类型参数。

11.2.8 多个类型参数

泛型类型可以使用任意数量的类型参数。在前面的Pair<T>例子中，只包含一个类型参数。为了存储不同类型的两个对象，比如一个“名称/值”对，可以创建类型的新版本来声明两个类型参数，如代码清单11-14所示。

代码清单11-14 使用多个类型参数声明泛型

```
interface IPair<TFirst, TSecond>
{
    TFirst First { get; set; }
    TSecond Second { get; set; }
}

public struct Pair<TFirst, TSecond>: IPair<TFirst, TSecond>
{
    public Pair(TFirst first, TSecond second)
    {
        _First = first;
        _Second = second;
    }

    public TFirst First
    {
        get{ return _First; }
        set{ _First = value; }
    }
    private TFirst _First;

    public TSecond Second
    {
```

```
        get{ return _Second; }
        set{ _Second = value; }
    }
    private TSecond _Second;
}
```

使用Pair<TFirst, TSecond>类时，只需在声明和实例化语句的尖括号中指定多个类型参数。调用方法时，提供与方法参数匹配的类型。如代码清单11-15所示。

代码清单11-15 使用要获取多个类型参数的类型

```
Pair<int, string> historicalEvent =
    new Pair<int, string>(1914,
        "Shackleton leaves for South Pole on ship Endurance");
Console.WriteLine("{0}: {1}",
    historicalEvent.First, historicalEvent.Second);
```

类型参数的数量（或者称为元数，即arity）对类进行了唯一性的区分。因此，由于类型参数的数量不同，可以在同一个命名空间同时定义Pair<T>和Pair<TFirst, TSecond>。此外，由于在语义上的密切联系，仅元数不同的泛型应该放到同一个C#文件中。

规范

要将只是类型参数数量不同的多个泛型类放到同一个文件中。

11.2.9 元数

从C# 4.0开始，CLR团队定义了9个新的泛型类型，它们都叫做Tuple[①]。和Pair<...>一样，相同的名称可以重用，只要元数[②]不同（每个类都有不同数量的类型参数），如代码清单11-16所示。

代码清单11-16 通过元数的不同来重载类型定义

```
public class Tuple { ... }
public class Tuple<T1>:
  IStructuralEquatable, IStructuralComparable, IComparable {...}
public class Tuple<T1, T2>: ... {...}
public class Tuple<T1, T2, T3>: ... {...}
public class Tuple<T1, T2, T3, T4>: ... {...}
public class Tuple<T1, T2, T3, T4, T5>: ... {...}
public class Tuple<T1, T2, T3, T4, T5, T6>: ... {...}
public class Tuple<T1, T2, T3, T4, T5, T6, T7>: ... {...}
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>: ... {...}
```

这一组Tuple<...>类是出于和Pair<T>与Pair<TFirst,TSecond>类相同的目的而设计的，只是在不特殊处理的前提下，它们最多能同时处理7个类型参数。事实上，使用代码清单11-16的最后一个Tuple，可以在TRest中存储另一个Tuple。这样特殊处理后，元组的大小实际就是无限的。

① Tuple一词在不作为类名提及时，许多文档都把它翻译成“元组”。在英文中，这个词来源于对顺序的一个抽象：single，double，triple，quadruple，quintuple，*n*-tuple。——译者注

② 元数的英文是arity。在计算机编程中，一个函数或运算（操作）的元数是指函数获取的实参或操作数的个数。它源于像unary（arity=1）、binary（(arity=2）、ternary（(arity=2）这样的单词。——译者注

在这一组元组类中，无类型参数的Tuple类非常有趣。这个类有8个静态的“工厂”方法[①]可以实例化各种泛型元组类型。虽然每个泛型类型都可以用自己的构造器直接实例化，但Tuple类型的工厂方法允许推断类型参数。代码清单11-17演示了这个差异。

代码清单11-17 使用Tuple的Create()工厂方法

```
Tuple<string, Contact> keyValuePair;
keyValuePair =
  Tuple.Create(
      "555-55-5555", new Contact("Inigo Montoya"));
keyValuePair =
  new Tuple<string, Contact>(
      "555-55-5555", new Contact("Inigo Montoya"));
```

显然，当Tuple变大时，如果不用Create()工厂方法，要指定的类型参数的数量会变得非常恐怖。

从元组类型由框架库声明这一事实可以推断出，CLR类型系统并不支持所谓的“元数可变”泛型类型。方法可以使用“参数数组”获取任意数量的参数，但泛型类型没有对应的技术，每个泛型类型的元数都必须固定。

11.2.10 嵌套泛型类型

嵌套类型自动获得包容类型的类型参数。例如，假如包容类型声明类型参数T，那么类型T也可以在嵌套类型中使用。如果嵌套类型包含了它自己的类型参数T，那么它会隐藏包容类型的同名类型参数。除此之外，嵌套类型中对T的任何引用都会引用嵌套的T类型参数。幸运的是，在嵌套类型中重用相同的类型参数名，会造成编译器报告一条警告消息，防止开发者因为不慎而使用了同名参数，如代码清单11-18所示。

代码清单11-18 嵌套的泛型类型

```
class Container<T, U>
{
  // Nested classes inherit type parameters.
  // Reusing a type parameter name will cause
  // a warning.
  class Nested<U>
  {
      void Method(T param0, U param1)
      {
      }
  }
}
```

包容类型的类型参数能在嵌套类型中访问，这类似于包容类型的成员能在嵌套类型中访问。规则很简单：在声明类型参数的类型主体的任何地方都能访问该类型参数。

规范

避免在嵌套类型中用同名参数隐藏外层类型的类型参数。

① “生产”对象的方法就是“工厂”方法。——译者注

11.3 约束

泛型允许为类型参数定义约束。这些约束强迫作为类型实参提供的类型遵守各种规则。来看看代码清单11-19的BinaryTree<T>类。

代码清单11-19 声明没有约束的BinaryTree<T>类

```
public class BinaryTree<T>
{
    public BinaryTree ( T item)
    {
        Item = item;
    }

    public T Item
    {
        get{ return _Item;    }
        set{ _Item = value; }
    }
    private T _Item;

    public Pair<BinaryTree<T>> SubItems
    {
        get{ return _SubItems; }
        set{ _SubItems = value; }
    }
    private Pair<BinaryTree<T>> _SubItems;
}
```

（一个有趣的地方是，BinaryTree<T>内部使用了Pair<T>。能这样做的原因很简单——Pair<T>是另一个类型。）

现在，假定当Pair<T>中的值赋给SubItems属性的时候，你希望二叉树对这些值进行排序。为了实现排序，SubItems的设置访问器（称为setter）使用当前所提供的键的CompareTo()方法，如代码清单11-20所示。

代码清单11-20 需要类型参数来支持一个接口

```
public class BinaryTree<T>
{
    // ...
    public Pair<BinaryTree<T>> SubItems
    {
        get{ return _SubItems; }
        set
        {
            IComparable<T> first;
            // ERROR: Cannot implicitly convert type...
            first = value.First;  // Explicit cast required

            if (first.CompareTo(value.Second) < 0)
            {
                // first is less than second.
                // ...
            }
            else
            {
                // first and second are the same or
```

```
                // second is Less than first.
                // ...
            }
            _SubItems = value;
        }
    }
    private Pair<BinaryTree<T>> _SubItems;
}
```

编译时，类型参数T是无约束的泛型。所以，编译器认为T当前只有从基类型object继承的成员，因为object是所有类型的终极基类（换言之，只能为类型参数T的实例调用像ToString()这样的已经由object定义的方法）。结果是编译器显示一个编译错误，因为object类型没有定义CompareTo()方法。

可以将T参数强制转换为IComparable<T>接口来访问CompareTo()方法，如代码清单11-21所示。

代码清单11-21 需要类型参数支持接口，否则会引发异常

```
public class BinaryTree<T>
{
    ...
    public Pair<BinaryTree<T>> SubItems
    {
        get{ return _SubItems; }
        set
        {
            IComparable<T> first;
            first = (IComparable<T>)value.First.Item;

            if (first.CompareTo(value.Second.Item) < 0)
            {
                // first is Less than second.
                ...
            }
            else
            {
                // second is Less than or equal to first.
                ...
            }
            _SubItems = value;
        }
    }
    private Pair<BinaryTree<T>> _SubItems;
}
```

遗憾的是，假如现在声明一个BinaryTree<SomeType>类的变量，但类型参数没有实现IComparable<SomeType>接口，就会发生运行时错误。具体地说，会引发InvalidCastException异常。这就失去了用泛型实现类型安全性的意义。

为了避免这个异常，代之以报告编译时错误，C# 允许为泛型类中声明的每个类型参数提供可选的**约束**列表。约束描述了泛型要求的类型参数的特征。为了声明一个约束，需要使用where关键字，后跟一对“参数:要求”。其中，“参数”必须是泛型类型中声明的一个参数，而“要求”描述了类型参数要能转换成的类或接口、是否必须有默认构造器，或者是引用类型还是值类型。

语言对比：C++——模板

一个有趣的地方是，微软在为C++提供的CLI支持中同时包含了泛型和C++模板，因为两者分属不同的体系。

C# 和CLR中的泛型有别于其他语言中的类似结构。虽然其他语言提供了类似的功能，但C# 在类型安全方面更出色。C# 的泛型既是语言特性，也是平台特性，底层的2.0“运行时”在其引擎中包含了对泛型的深入支持。

C++模板和C# 泛型存在显著区别；最根本的一点在于C++模板为提供的每个类型实参都要重新编译一次。它对模板化代码执行的是一种“查找并替换”分析方式。也就是说，先将类型参数替换成类型实参，再编译结果代码。另一方面，C#泛型将泛型类编译成CIL一次。C#利用了“泛型类型是CIL语言的一部分”这一事实。在运行时，如果泛型类型得到的是值类型的类型实参，CLR的JIT编译器会重新生成特化[①]的代码。如果类型实参是引用类型，则重用以前生成的代码。

由于C++模板在编译时针对每个类型实参都要重新分析一遍，所以泛型约束就没有必要了。当前已经提供了类型实参，所以不必向编译器证明模板化的代码还能兼容其他类型实参。

C++模板的一个优势是可以为类型实参使用操作符（+、-，等等）。用类型实参替换时，会重新分析代码以确保提供的类型支持操作符，否则会报错。（遗憾的是，错误中指出的出错位置的一般都不是问题类型实参的实际使用位置。）C# 不支持在类型参数上调用操作符，因为操作符是静态的——接口或者基类约束识别不了。

简单地说，C++模板错误仅在使用提供了有问题的类型实参的模板时才会被发现和报告，定义的时候不行。由于C# 泛型可以声明约束，所以编译器能在定义泛型时防范此类错误，从而更快地识别无效的假设。另外，如果使用类型实参违反了约束的泛型类型，错误信息会指向变量的使用位置，而不是泛型实现中的某个位置。

11.3.1 接口约束

为了确保二叉树的结点正确排序，需要使用BinaryTree类的CompareTo()方法。为此，最高效的做法是为T类型参数施加约束，规定T必须实现IComparable<T>接口。具体的语法如代码清单11-22所示。

代码清单11-22 声明接口约束

```
public class BinaryTree<T>
    where T: System.IComparable<T>
{
    ...
    public Pair<BinaryTree<T>> SubItems
    {
        get{ return _SubItems; }
```

① specialization，C#规范将这个词译为“专用化”。——译者注

```
            set
            {
                IComparable<T> first;
                // Notice that the cast can now be eliminated.
                first = value.First.Item;

                if (first.CompareTo(value.Second.Item) < 0)
                {
                    // first is less than second
                    ...
                }
                else
                {
                    // second is less than or equal to first.
                    ...
                }
                _SubItems = value;
            }
        }
        private Pair<BinaryTree<T>> _SubItems;
    }
```

在代码清单11-22中添加了接口约束之后，编译器会确保每次使用BinaryTree类的时候，所提供的类型参数都实现了IComparable<T>接口。此外，调用CompareTo()方法之前不需要先将变量显式转型为IComparable<T>接口。即使访问显式实现的接口成员也不需要转型（在其他情况下，如果不进行转型，会造成成员被隐藏起来）。如果传递的类型实参是值类型，那么在调用它的方法时，编译器会检查方法是否与在约束中指定的任何接口的任何方法相匹配。

如果试图使用System.Text.StringBuilder作为类型参数来创建一个BinaryTree<T>变量，会报告一个编译器错误，因为StringBuilder没有实现IComparable<T>。输出11-3展示了这样的一个错误。

输出11-3

```
error CS0311: The type 'System.Text.StringBuilder' cannot be used as type
parameter 'T' in the generic type or method 'BinaryTree<T>'. There is no
implicit reference conversion from 'System.Text.StringBuilder' to
'System.IComparable<System.Text.StringBuilder>'.
```

为了规定某个数据类型必须实现某个接口，需要声明一个**接口约束**。这种约束避免了非要转型才能调用一个显式的接口成员实现。

11.3.2 类类型约束

有时可能要求能将类型实参转换为特定的类类型。这是用**类类型约束**做到的，如代码清单11-23所示。

代码清单11-23 声明类类型约束

```
public class EntityDictionary<TKey, TValue>
    : System.Collections.Generic.Dictionary<TKey, TValue>
    where TValue : EntityBase
{
    ...
```

```
}
```

EntityDictionary <TKey, TValue>要求为类型参数TValue 提供的所有类型实参都能隐式转换为EntityBase类。这样一来，在泛型实现中，就可以为TValue类型的值使用EntityBase的成员，因为约束已确保所有类型实参都能隐式转换为EntityBase类。

类类型约束的语法与接口约束基本相同。但是，假如同时指定了多个约束，那么类类型约束必须第一个出现（就像在类声明中，基类必须先于所实现的接口出现）。然而，与接口约束不同，多个类类型约束是不允许的，因为不可能从多个不相关的类派生。类似地，类类型约束不能指定密封类或者不是类的类型。例如，C# 不允许将类型参数约束为string或者System.Nullable<T>，否则只有单一类型实参可供选择，那还有什么“泛型”可言？如果类型参数被约束为单一类型，类型参数就没有存在的必要了，直接使用那个类型即可。

11.3.3 struct/class约束

另一个重要的泛型约束是将类型参数限制为任何非可空的值类型或者任何引用类型。编译器不允许在约束中将System.ValueType指定成基类。相反，C# 提供了特殊的语法，这种语法同时适用于引用类型。在这种语法中，不是为T指定基类，而是只需使用关键字struct或者class，如代码清单11-24所示。

代码清单11-24 规定类型参数必须是值类型

```
public struct Nullable<T> :
    IFormattable, IComparable,
    IComparable<Nullable<T>>, INullable
    where T : struct
{
    // ...
}
```

注意，class约束不是将类型实参限制为类类型，而是限制为引用类型，所以任何类、接口、委托和数组类型都符合要求。

由于基类约束要求指定特定的基类，所以基类约束和struct或者class约束一起使用没有意义，也使人迷惑。因此，类类型约束和struct/class约束不能一起使用。

struct约束有一个非常特别的地方。可空值类型不符合约束，这是为什么呢？可空值类型是作为泛型类型Nullable<T>实现的，后者本身向T应用了struct约束。如果可空值类型符合该约束，就可以定义毫无意义的Nullable<Nullable<int>>类型。双重可空的整数当然没有意义。（如同预期的那样，快捷语法int??也是不允许的。）

11.3.4 多个约束

对于任何给定的类型参数，都可以指定任意数量的接口约束，但类类型约束只能指定一个，因为一个类可以实现任意数量的接口，但肯定只能从一个类继承。每个新约束都在一个以逗号分隔的列表中声明，约束列表跟在泛型类型名称和一个冒号之后。如果有多个类型参数，每个类型参数前面都要使用where关键字。在代码清单11-25中，EntityDictionary类声明了两个类型参数：TKey和TValue。TKey类型参数有两个接口约束，而TValue类型参数有一个类类型约束。

代码清单11-25 指定多个约束

```
public class EntityDictionary<TKey, TValue>
    : Dictionary<TKey, TValue>
    where TKey : IComparable<TKey>, IFormattable
    where TValue : EntityBase
{
  ...
}
```

在这个例子中，TKey有多个约束，而TValue有一个。为一个类型参数指定多个约束时，会假定存在一个AND关系。例如，假定提供类型C作为TKey的类型实参，那么C必须实现IComparable<C>和IFormattable。

注意，在两个where子句之间，并不存在逗号。

11.3.5 构造器约束

某些情况下，需要在泛型类中创建类型实参的实例。在代码清单11-26中，EntityDictionary<TKey, TValue>类的MakeValue()方法必须创建与类型参数TValue对应的类型实参的实例。

代码清单11-26 用约束规定必须有默认构造器

```
public class EntityBase<TKey>
{
    public TKey Key
    {
        get{ return _Key;  }
        set{ _Key = value; }
    }
    private TKey _Key;
}

public class EntityDictionary<TKey, TValue> :
    Dictionary<TKey, TValue>
    where TKey: IComparable<TKey>, IFormattable
    where TValue : EntityBase<TKey>, new()
{
    // ...

    public TValue MakeValue(TKey key)
    {
        TValue newEntity = new TValue();
        newEntity.Key = key;
        Add(newEntity.Key, newEntity);
        return newEntity;
    }

    // ...
}
```

并非所有对象都保证有公共默认构造器，所以编译器不允许为未约束的类型参数调用默认构造器。为了克服编译器的这一限制，要在指定了其他所有约束之后添加new()。这就是所谓的**构造器约束**，它要求类型实参必须有默认构造器。只能对默认构造器进行约束。不能为带参数的构造器指定约束。

11.3.6 约束继承

无论泛型类型参数，还是它们的约束，都不会被派生类继承，因为泛型类型参数不是成员。记住，类继承的特点在于派生类拥有基类的所有成员。经常采取的做法是创建新的泛型类型，让它从其他泛型类型继承。由于派生的泛型类型的类型参数现在是泛型基类的类型实参，所以类型参数必须具有与基类的那些相同（或更强）的约束。听起来是不是很拗口？来看看代码清单11-27的例子。

代码清单11-27 显式指定继承的约束

```
class EntityBase<T> where T : IComparable<T>
{
  // ...
}
```

```
// ERROR:
// The type 'U' must be convertible to
// 'System.IComparable<U>' in order to use it as parameter
// 'T' in the generic type or method.
// class Entity<U> : EntityBase<U>
// {
//     ...
// }
```

EntityBase<T>要求由基类指示符EntityBase<U>提供给T的类型实参U实现IComparable<U>，所以Entity<U>类要对U进行相同的约束，否则会造成编译时错误。这使程序员在使用派生类时能够注意到基类的约束，避免在使用派生类的时候发现了一个约束，却不知道该约束来自哪里。

本章要到后面才会讲到泛型方法。但简单地说，方法也可以是泛型的，也可以对类型参数应用约束。那么，当虚泛型方法被继承并重写时，约束是如何来处理的呢？和泛型类声明的类型参数相反，重写一个虚泛型方法时，或者创建一个显式接口方法实现[①]时，约束是隐式继承的，不可以重新声明。代码清单11-28对此进行了演示。

代码清单11-28 禁止虚成员重复继承的约束

```
class EntityBase
{
  public virtual void Method<T>(T t)
      where T : IComparable<T>
  {
    // ...
  }
}
```

```
class Order : EntityBase
{
  public override void Method<T>(T t)
  //     Constraints may not be repeated on overriding
  //     members
  //     where T : IComparable<T>
  {
```

① 如第7章所述，显式接口方法实现（Explicit Interface Methool Implementation, EIMI)是指在方法名前附加接口名前缀。——译者注

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

在泛型类继承的情况下，不仅可以保留基类本来的约束（这是必须的），还可以添加额外的约束，从而对派生类的类型参数进行进一步的限制。但是，重写虚泛型方法时，需要遵守和基类方法定义的约束完全一样的约束。额外的约束会破坏多态性，所以不允许新增约束，而且重写版本的类型参数约束是隐式继承的。

高级主题：约束的限制

对约束进行了一些适当的限制，以避免产生无意义的代码。例如，不能合并使用类类型约束和struct/class约束。另外，不能限制从一些特殊的类型继承，比如object、数组、System.ValueType、System.Enum (enum)、System.Delegate和System.MulticastDelegate。

下面讨论了另一些不支持的约束。

不支持操作符约束

不支持用约束限制类必须实现特定方法或操作符。只能通过接口约束（限制方法）和类类型约束（限制方法和操作符）来提供不完整的支持。正因为这个原因，代码清单11-29的泛型Add()方法是不能工作的。

代码清单11-29 约束表达式不能提出对操作符的要求

```
public abstract class MathEx<T>
{
    public static T Add(T first, T second)
    {
        // Error: Operator '+' cannot be applied to
        // operands of type 'T' and 'T'.
        // return first + second;
    }
}
```

在这个例子中，方法假定+操作符可以在为T提供的所有类型实参上使用。然而，没有约束能限制类型实参必须支持+操作符，所以发生了一个错误。这里只能使用一个类类型约束，限制从实现了+操作符的类继承。

另外，无法限制类型必须有一个静态方法。

不支持OR条件

假如为一个类型参数提供多个接口约束，编译器认为不同约束之间总是存在一个AND关系。例如，where TKey : IComparable<T>, IFormattable要求同时支持IComparable<T>和IFormattable。不能在约束之间指定OR关系。因此，代码清单11-30的写法是不允许的。

代码清单11-30 不允许使用OR来合并两个约束

```
public class BinaryTree<T>
    // Error:  OR is not supported.
    // where T: System.IComparable<T> || System.IFormattable
{
```

```
        ...
    }
```

如果支持OR，编译器就无法在编译时判断要调用哪一个方法。

委托和枚举类型的约束是无效的

委托类型、数组类型和枚举类型不能在基类约束中使用，因为它们实际全是“密封”类型。（委托的主题将在第12章讨论）它们的基类型System.Delegate、System.MultiCastDelegate、System.Array和System.Enum也不可在约束中使用。例如，针对代码清单11-31的类声明，编译器会报告一个错误。

代码清单11-31 继承约束不能是System.Delegate类型

```
// Error:  Constraint cannot be special class 'System.Delegate'
public class Publisher<T>
    where T : System.Delegate
{
    public event T Event;
    public void Publish()
    {
        if (Event != null)
        {
            Event(this, new EventArgs());
        }
    }
}
```

所有委托类型都被视为是不能指定为类型参数的特殊类。否则，编译时就无法对Event()调用进行验证，因为对于数据类型System.Delegate和System.MulticastDelegate来说，所触发事件的签名是未知的。同样的限制也适用于任何枚举类型。

构造器约束只针对默认构造器

代码清单11-26包含一个构造器约束，它要求TValue必须提供默认构造器（公共的无参数的构造器）。不能要求TValue必须提供一个带有参数的构造器。例如，你可能想约束TValue，要求它必须提供一个构造器来获取为TKey提供的类型实参，但这是不可能的。代码清单11-32进行了演示。

代码清单11-32 只能为默认构造器指定构造器约束

```
public TValue New(TKey key)
    {
        // Error: 'TValue': Cannot provide arguments
        // when creating an instance of a variable type.
        TValue newEntity = null;
        // newEntity = new TValue(key);
        Add(newEntity.Key, newEntity);
        return newEntity;
    }
```

为了克服这个限制，一个办法是提供一个工厂接口，让它包含一个方法对类型进行实例化。实现接口的工厂负责实例化实体，而不是EntityDictionary本身（参见代码清单11-33）。

代码清单11-33 使用工厂接口来代替构造器约束

```
public class EntityBase<TKey>
{
  public EntityBase(TKey key)
  {
      Key = key;
  }
  public TKey Key
  {
      get { return _key; }
      set { _key = value; }
  }
  private TKey _key;
}

public class EntityDictionary<TKey, TValue, TFactory> :
      Dictionary<TKey, TValue>
  where TKey : IComparable<TKey>, IFormattable
  where TValue : EntityBase<TKey>
  where TFactory : IEntityFactory<TKey, TValue>, new()
{
  ...
  public TValue New(TKey key)
  {
      TFactory factory = new TFactory();
      TValue newEntity = factory.CreateNew(key);
      Add(newEntity.Key, newEntity);
      return newEntity;
  }
  ...
}

public interface IEntityFactory<TKey, TValue>
{
    TValue CreateNew(TKey key);
}
...
```

像这样声明，就可将新key传给一个要获取参数的TValue工厂方法（而不是无参数的默认构造器）。这个例子不再为TValue使用构造器约束，因为TFactory负责实例化值（还可修改代码清单11-33来保存对工厂的引用。这样就可重用工厂，不必每次都重新实例化它）。

声明EntityDictionary<TKey, TValue, TFactory>类型的变量，会造成与代码清单11-34的Order实体类似的实体声明。

代码清单11-34 声明在EntityDictionary<...>中使用的实体

```
public class Order : EntityBase<Guid>
{
  public Order(Guid key) :
      base(key)
  {
      // ...
  }
}
```

```
public class OrderFactory : IEntityFactory<Guid, Order>
{
  public Order CreateNew(Guid key)
  {
      return new Order(key);
  }
}
```

11.4 泛型方法

通过前面的学习，你知道可以在泛型类中轻松添加方法来使用类型声明的泛型类型参数。在迄今为止的所有泛型类例子中，我们一直是这样做的。但这些不是泛型方法。

泛型方法要使用泛型类型参数，这一点和泛型类型一样。在泛型或非泛型类型中都能声明泛型方法。如果在泛型类型中声明泛型方法，其类型参数和泛型类型的类型参数是有区别的。为了声明泛型方法，要按照与泛型类型一样的方式指定泛型类型参数，也就是在方法名之后添加类型参数声明，例如代码清单11-35中的MathEx.Max<T>和MathEx.Min<T>。

代码清单11-35 定义泛型方法

```
public static class MathEx
{
  public static T Max<T>(T first, params T[] values)
      where T : IComparable<T>
  {
      T maximum = first;
      foreach (T item in values)
      {
          if (item.CompareTo(maximum) > 0)
          {
              maximum = item;
          }
      }
      return maximum;
}

public static T Min<T>(T first, params T[] values)
    where T : IComparable<T>
{
    T minimum = first;

      foreach (T item in values)
      {
          if (item.CompareTo(minimum) < 0)
          {
              minimum = item;
          }
      }
      return minimum;
  }
}
```

这个例子中的方法是静态的，但C#对此没有要求。

注意，泛型方法和泛型类型相似，可以包含多个类型参数。类型参数的数量（元数）是一种可用于区分方法签名的特性。换言之，两个方法可以有相同的名称和形参类型，只要类型参数的数量不同。

11.4.1 泛型方法类型推断

使用泛型类型时，是在类型名之后提供类型实参。类似地，调用泛型方法时，是在方法的类型名之后提供类型实参。代码清单11-36展示了用于调用Min<T>和Max<T>方法的代码。

代码清单11-36 显式指定类型参数

```
Console.WriteLine(
    MathEx.Max<int>(7, 490));
Console.WriteLine(
    MathEx.Min<string>("R.O.U.S.", "Fireswamp"));
```

输出11-4展示了代码清单11-36的结果。

输出11-4

```
490
Fireswamp
```

int和string是调用泛型方法时指定的类型实参。但指定类型实参是多余的，因为编译器能根据传给方法的实参推断出类型。为了避免多余的编码，可以在调用时不指定类型实参。这就是所谓的**类型推断**。代码清单11-37展示了一个例子，结果如输出11-5所示。

代码清单11-37 根据方法实参推断类型

```
Console.WriteLine(
    MathEx.Max(7, 490)); // No type arguments!
Console.WriteLine(
    MathEx.Min("R.O.U.S'", "Fireswamp"));
```

输出11-5

```
490
Fireswamp
```

类型推断要想成功，方法实参的类型必须与泛型方法的形参“匹配”以推断出正确的类型实参。一个有趣的问题是，假如推断自相矛盾怎么办？例如，使用MathEx.Max(7.0, 490)调用Max<T>，从第一个方法实参推断出类型实参是double，从第二个推断出是int，这是自相矛盾的。在C# 2.0中，这会造成一个错误。但进行更高级的分析，就知道这个矛盾是可以解决的，因为每个int都能转换成double，所以double是类型实参的最佳选择。从C# 3.0开始改进了方法类型推断算法，允许编译器进行这种更高级的分析。

如果分析仍然不够高级，不能推断出正确类型，那么要么对方法实参进行强制类型转换，向编译器澄清类型推断时应使用的参数类型，要么放弃类型推断，显式指定类型实参。

还要注意，推断时算法只考虑方法实参、实参类型以及形参类型。本来其他因素也可以考虑在内的，比如泛型方法的返回类型、方法返回值所赋给的变量的类型以及对泛型方法类型参数的约束。但这些都被算法忽视了。

11.4.2 约束的指定

泛型方法的类型参数也允许指定约束，其方式与在反省类型中指定类型参数的方式相同。例如，可以指定类型参数必须实现某个接口，或者必须能转换成某个类。约束在参数列表和方法主体之间指定，如代码清单11-38所示。

代码清单11-38 约束泛型方法的类型参数

```
public class ConsoleTreeControl
{
    // Generic method Show<T>
    public static void Show<T>(BinaryTree<T> tree, int indent)
        where T :  IComparable<T>
    {
        Console.WriteLine("\n{0}{1}",
            "+ --".PadLeft(5*indent, ' '),
            tree.Item.ToString());
        if (tree.SubItems.First != null)
            Show(tree.SubItems.First, indent+1);
        if (tree.SubItems.Second != null)
            Show(tree.SubItems.Second, indent+1);
    }
}
```

注意，Show<T>的实现没有直接使用IComparable<T>接口的任何成员，所以你可能奇怪为什么需要约束。记住，BinaryTree<T>类需要这个接口（参见代码清单11-39）。

代码清单11-39 BinaryTree<T>需要IComparable<T>类型参数

```
public class BinaryTree<T>
    where T: System.IComparable<T>
{
    ...
}
```

因为BinaryTree<T>类为T施加了这个约束，而Show<T>使用了BinaryTree<T>中T作为约束类型参数对应的类型实参，所以Show<T>也需要提供约束。

高级主题：泛型方法中的转型

有时应该避免使用泛型。例如，在使用它会造成一次转型操作被“掩盖”起来的时候。来看看下面这个方法，它的作用是将一个stream转换成给定类型的对象：

```
public static T Deserialize<T>(
   Stream stream, IFormatter formatter)
{
   return (T)formatter.Deserialize(stream);
}
```

formatter负责将数据从流中移除，并把它转换成object。为formatter调用Deserialize()，会返回object类型的数据。如果使用Deserialize()的泛型版本进行调用，那么会写出下面这样的代码：

```
string greeting =
    Deserialization.Deserialize<string>(stream, formatter);
```

上述代码的问题在于，对于方法的调用者来说，Deserialize<T>()似乎是类型安全的。然而，仍然会为调用者隐式（而非显式）执行一次转型。下面是非泛型的等价调用：

```
string greeting =
   (string)Deserialization.Deserialize(stream, formatter);
```

运行时转型可能失败；所以方法不像表面上那样类型安全。Deserialize<T>方法是纯泛型的，会向调用者隐藏转型动作，这存在一定风险。更好的做法是使方法成为非泛型的，返回object，使调用者注意到它不是类型安全的。开发者在泛型方法中执行转型时，假如没有约束来验证转型的有效性，那么一定要非常小心。

规范

避免用表面上看是类型安全的但实际并不是类型安全的泛型方法误导调用者。

11.5 协变性和逆变性

刚接触泛型类型的人经常会问一个问题：为什么不能将List<string>类型的表达式赋给List<object>类型的变量。既然string能转换成object，string列表也应兼容于object列表呀？但实际情况并非如此，这个赋值动作既不类型安全，也不合法。用不同类型参数声明同一个泛型类的两个变量，即使是将一个较具体的类型赋给一个较泛化的类型，这两个变量也不是类型兼容的。也就是说，它们不是**协变量**（covariant）。

“协变量”是借鉴自范畴论的术语，意思很容易明白。假定两个类型X和Y具有特殊关系，X类型的每个值都能转换成Y类型。如果I<X>和I<Y>总是具有相同的特殊关系，就说“I<T>对T协变”。使用仅一个类型参数的泛型类型时，可以简单地说“I<T>是协变的”。从I<X>向I<Y>的转换称为协变转换。

例如，同一个泛型类Pair<Contact>和Pair<PdaItem>的实例就不是类型兼容的，即使两者的类型实参兼容。换言之，编译器禁止隐式或显式地将Pair<Contact>转换为Pair<PdaItem>，即使Contact从PdaItem派生。类似地，从Pair<Contact>转换为接口类型IPair<PdaItem>也会失败，如代码清单11-40所示。

代码清单11–40 具有不同类型参数的泛型之间的转换

```
// ...
// Error: Cannot convert type ...
Pair<PdaItem> pair = (Pair<PdaItem>) new Pair<Contact>();
IPair<PdaItem> duple = (IPair<PdaItem>) new Pair<Contact>();
```

但是，为什么这样做不合法呢？为什么List<T>和Pair<T>不是协变的呢？代码清单11-41展示了假如C#语言允许不受限制的泛型协变性，那么会发生什么呢？

代码清单11–41 禁止协变性以维持同质性（homogeneity）

```
//...
Contact contact1 = new Contact("Princess Buttercup"),
Contact contact2 = new Contact("Inigo Montoya");
Pair<Contact> contacts = new Pair<Contact>(contact1, contact2);


// This gives an error: Cannot convert type ...
// But suppose it did not.
// IPair<PdaItem> pdaPair = (IPair<PdaItem>) contacts;
// This is perfectly legal, but not type safe.
// pdaPair.First = new Address("123 Sesame Street");
...
```

IPair<PdaItem>中可以包含地址，但Pair<Contact>对象只能包含联系人，不能包含地址。若允许不受限制的泛型协变性，类型安全将完全失去保障。

现在应该很清楚为什么字符串列表不能作为对象列表使用了。在字符串列表中不能插入整数，但在对象列表中可以插入整数，所以从字符串列表转型成对象列表一定要被视为非法，使编译器能预防错误。

11.5.1 在C# 4.0中使用out类型参数修饰符允许协变性

你或许已经注意到了，不限制协变性之所以会造成刚才描述的问题，是因为泛型Pair<T>和泛型List<T>都允许其内容写入。如果创建只读的IReadOnlyPair<T>接口，只允许T从接口中“出来”（换言之，作为方法或只读属性的返回类型使用），永远不“进入”接口（换言之，不作为形参或可写属性的类型使用），就不会出问题了，如代码清单11-42所示。

代码清单11-42 可能的协变性

```
interface IReadOnlyPair<T>
{
  T First { get; }
  T Second { get; }
}
```

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

```
public struct Pair<T> : IPair<T>, IReadOnlyPair<T>
{
  // ...
}
```

```
class Program
{
  static void Main()
  {
      // Error: Only theoretically possible without
      // the out type parameter modifier
      Pair<Contact> contacts =
          new Pair<Contact>(
              new Contact("Princess Buttercupt"),
              new Contact("Inigo Montoya") );
      IReadOnlyPair<PdaItem> pair = contacts;
      PdaItem pdaItem1 = pair.First;
      PdaItem pdaItem2 = pair.Second;
  }
}
```

通过限制泛型类型声明，让它只向接口的外部公开数据，编译器就没有理由禁止协变性了。在IReadOnlyPair<PdaItem>实例上进行的所有操作都将Contact（来自原始的Pair<Contact>对象）向上转换为基类PdaItem——这是完全有效的转换。没有办法将地址

“写入”实际只是一对联系人的对象，因为接口没有公开任何可写属性。

上面的代码仍然编译不了。但是，从C# 4开始加入了对安全协变性的支持。为了指出泛型接口应该对它的某个类型参数协变，就用out修饰符来修饰该类型参数。代码清单11-43展示了如何修改接口声明以指出它应该允许协变。

代码清单11-43 使用out类型参数修饰符实现协变

```
...
interface IReadOnlyPair<out T>
{
  T First { get; }
  T Second { get; }
}
```

用out修饰IReadOnlyPair<out T>接口的类型参数，会导致编译器验证T真的只用作“输出”，即只用于方法的返回类型和只读属性的返回类型，永远不用于形参或者属性的赋值方法。如果验证通过，编译器就会允许对接口的任何协变赋值操作了。对代码清单11-42进行了这个修改之后，程序就能成功编译和执行了。

协变转换有一些重要的限制。

- 只有泛型接口和泛型委托（第12章）才可以是协变的。泛型类和结构永远不是协变的。
- 提供给“来源”和“目标”泛型类型的类型实参必须是引用类型，不能是值类型。也就是说，一个 IReadOnlyPair<string>可以协变转换为IReadOnlyPair<object>，因为string和object都是引用类型。但一个IReadOnlyPair<int>不能转换成IReadOnlyPair<object>，因为int不是引用类型。
- 接口或委托必须声明为支持协变，编译器必须验证协变所针对的类型参数确实只用在“输出”位置。

11.5.2 在C# 4.0中使用in类型参数修饰符允许逆变性

协变性在反方向上称为**逆变性**（contravariance）。同样地，假定X和Y类型彼此相关，每个X类型的值都能转换成Y类型。如果I<X>和I<Y>类型总是具有相反的特殊关系，也就是说，I<Y>类型的每个值都能转换成I<X>类型——就说“I<T>对T逆变”。

大多数人都感觉逆变理解起来比协变难。“比较器”（comparer）是逆变的典型例子。假定有一个派生类型Apple（苹果）和一个基类型Fruit（水果）。它们明显具有特殊关系：Apple类型的每个值都能转换成Fruit。

现在，假定有一个接口ICompareThings<T>，它包含方法bool FirstIsBetter(T t1, T t2)来获取两个T，返回一个bool指明第一个是否比第二个好。

提供类型实参会发生什么？ ICompareThings<Apple>的方法获取两个Apple并比较它们。ICompareThings<Fruit>的方法获取两个Fruit并比较它们。由于所有Apple都是Fruit，所以凡是需要一个ICompareThings<Apple>的地方，都可以安全地使用ICompareThings<Fruit>类型的值。转换的方向变反了，这正是“逆变”一词的由来。

丝毫不奇怪，为了确保安全地进行逆变，对协变接口的限制必须反着来。对某个类型参数逆变的接口只能在“输入”位置使用那个类型参数。这种位置主要就是形参，极少见的是只读属性的类型。如代码清单11-44所示，用in修饰符声明类型参数，从而将接口标记为逆变。

代码清单11-44 使用in类型参数修饰符实现逆变

```
class Fruit {}
class Apple : Fruit {}
class Orange : Fruit {}
```

```
interface ICompareThings<in T>
{
  bool FirstIsBetter(T t1, T t2);
}
```

```
class Program
{
  class FruitComparer : ICompareThings<Fruit>
  { ... }
  static void Main()
  {
      // Allowed in C# 4.0
      ICompareThings<Fruit> fc = new FruitComparer();
      Apple apple1 = new Apple();
      Apple apple2 = new Apple();
      Orange orange = new Orange();
      // A fruit comparer can compare apples and oranges:
      bool b1 = fc.FirstIsBetter(apple1, orange);
      // or apples and apples:
      bool b2 = fc.FirstIsBetter(apple1, apple2);
      // This is legal because the interface is
      // contravariant:
      ICompareThings<Apple> ac = fc;
      // This is really a fruit comparer, so it can
      // compare two apples still.
      bool b3 = ac.FirstIsBetter(apple1, apple2);
  }
}
```

注意，与协变性相似，逆变性要求在声明接口的类型参数时使用修饰符in。它指示编译器核实T从未在属性的取值方法（get访问器方法）中出现，也没有作为方法的返回类型使用。如果检查无误，就启用接口的逆变转换。

逆变转换存在与协变转变相似的限制：只有泛型接口和委托类型才能是逆变的，发生变化的类型实参只能是引用类型，而且编译器必须能验证接口对于逆变转换是安全的。

接口可以对一个类型参数协变，对另一个逆变。如代码清单11-45所示，假定设备能将一样东西转换成ITransformer<in TSource, out TTarget>接口所描述的另一样东西。

代码清单11-45 在单个泛型类型中组合使用协变性和逆变性

```
class Food {}
class Pizza : Food {}
class Salad : Food {}
class Document {}
class ComputerProgram : Document {}
interface ITransformer<in TSource, out TTarget>
{
```

```
  TTarget Transform(TSource source);
}
// A computer programmer is a device which transforms
// food into computer programs:

class Programmer : ITransformer<Food, ComputerProgram>
{
  public ComputerProgram Transform(Food f) { ... }
}
class Program
{
  static void Main()
  {
    var programmer = new Programmer();
    ComputerProgram cp = programmer.Transform(new Salad());
    // A computer programmer may be converted with
    // both co- and contra-variant conversions. Because
    // a programmer can turn any food into a computer
    // program, it can be used as a device that turns pizza
    // into documents.
    ITransformer<Pizza, Document> transformer = programmer;
    Document d = transformer.Transform(new Pizza());
  }
}
```

最后要注意，编译器要在整个源代码的范围内检查协变性和逆变性类型参数修饰符的有效性。以代码清单11-46的PairInitializer<in T>接口为例。

代码清单11-46 可变性的编译器验证

```
// ERROR:  Invalid variance, the type parameter 'T' is not
//         invariantly valid
interface IPairInitializer<in T>
{
  void Initialize(IPair<T> pair);
}
```

```
// Suppose the code above were legal, and see what goes
// wrong:
class FruitPairInitializer : IPairInitializer<Fruit>
{
  // Let's initiaize our pair of fruit with an
  // apple and an orange.
  public void Initialize(IPair<Fruit> pair)
  {
    pair.First = new Orange();
    pair.Second = new Apple();
  }
}
```

```
// ... later ...
var f = new FruitPairInitializer();
// This would be legal if contravariance were legal:
IPairInitializer<Apple> a = f;
// And now we write an orange into a pair of apples:
a.Initialize(new Pair<Apple>());
```

如果对协变性和逆变性研究得不是很透彻，可能认为既然IPair<T>只是一个输入参数，所以在IPairInitializer上将T限制为in是有效的。然而，IPair<T>接口不能安全地变化，所以不能用可以变化的类型实参构造它。由于不是类型安全的，所以编译器一开始就不允许将IPairInitializer<T>接口声明为逆变的。

11.5.3 数组对不安全协变性的支持

之前，我们一直将协变性和逆变性描述成泛型类型的特点。在所有非泛型类型中，数组是最像泛型的。如同平时思考泛型“T的列表”或者泛型“T的数对”一样，也可以用相同的模式思考“T的数组”。由于数组同时支持读取和写入，基于刚学到的协变和逆变的知识，你可能以为数组既不能安全逆变，也不能安全协变，只有在从不写入时才能安全地协变，只有在从不读取时才能安全地逆变。这两个限制似乎都不现实。

遗憾的是，C#确实支持数组协变，即使这样并不是类型安全的。例如，Fruit[] fruits = new Apple[10]在C#中完全合法，但如果接着写fruits[0] = new Orange();，“运行时”会引发异常来报告违反了类型安全性。将Orange赋给Fruit数组并非总是合法（因为后者可能实际上是一个Apple数组），这对开发人员造成了极大的困扰。但这个问题并非只是C#才有。事实上，使用了“运行时”的数组实现的所有CLR语言都存在相同的问题。

尽是避免使用不安全的数组协变。每个数组都能转换成只读（进而安全协变）的IEnumerable<T>接口。换言之，IEnumerable<Fruit> fruits = new Apple[10]既是类型安全的又是合法的，因为在只有只读接口的前提下，是无法在数组中插入一个Orange的。

规范

避免不安全的数组协变。而是**考虑**将数组转换成只读接口IEnumerable<T>，以便通过协变转换来安全地转换。

11.6 泛型的内部机制

在之前的章节中，我们描述了对象在CLI类型系统中的广泛应用。所以，假如告诉你泛型也是对象，相信你一点儿都不会感到奇怪。事实上，泛型类的“类型参数”变成了元数据，“运行时”在需要的时候会利用它们构造恰当的类。所以，泛型支持继承、多态性以及封装。使用泛型，可以定义方法、属性、字段、类、接口以及委托。

为此，泛型需要来自底层“运行时”的支持。所以，将泛型引入C# 语言，既是编译器的一特性，也是平台的特性。例如，为了避免装箱，对于基于值的类型参数来说，其泛型实现和引用类型参数的泛型实现是不同的。

高级主题：泛型的CIL表示

泛型类编译后与普通类并无区别。编译的结果只有元数据和CIL。CIL是参数化的，接受在代码中别的地方由用户提供的类型。代码清单11-47中声明了一个简单的Stack类。

代码清单11-47 Stack<T>声明

```
public class Stack<T> where T : IComparable
```

```
{
    T[] items;
    // rest of the class here

}
```

编译类时，生成的CIL是参数化的，如代码清单11-48所示。

代码清单11-48 Stack<T>的CIL代码

```
.class private auto ansi beforefieldinit
    Stack'1<([mscorlib]System.IComparable)T>
    extends [mscorlib]System.Object
{
  ...
}
```

首先注意第二行Stack之后的'1。这是一个元数（参数数量）。它声明了泛型类要求的类型实参的数量。对于像EntityDictionary<TKey, TValue>这样的声明，元数是2。

除此之外，在生成的CIL中，第二行显示了施加在类上的约束。可以看出T类型参数有一个IComparable约束。

继续研究CIL代码，会发现类型T的items数组声明进行了修改，使用“感叹号表示法”包含了一个类型参数，这是自CIL开始支持泛型后引入的新特性。感叹号指出为类指定的第一个类型参数的存在，如代码清单11-49所示。

代码清单11-49 CIL用感叹号表示法来支持泛型

```
.class public auto ansi beforefieldinit
     'Stack'1'<([mscorlib]System.IComparable) T>
     extends [mscorlib]System.Object
{
     .field private !0[ ] items
     ...
}
```

除了在类的头部包含元数和类型参数，并在代码中用感叹号指出类型参数之外，泛型类和非泛型类的CIL代码并无多大区别。

11.6.1 基于值类型的泛型的实例化

用值类型作为类型参数首次构造一个泛型类型时，“运行时”会将指定的类型参数放到CIL中合适的位置，从而创建一个具体化的泛型类型。总之，“运行时”会针对每个新的“参数值类型”创建一个新的具体化的泛型类型。

例如，假定声明一个整数Stack，如代码清单11-50所示。

代码清单11-50 Stack<int>定义

```
Stack<int> stack;
```

第一次使用Stack<int>类型时，“运行时”会生成Stack类的一个具体化版本，并用int替换它的类型参数。以后，每当代码使用Stack<int>的时候，“运行时”都重用已生成

的具体化Stack<int>类。代码清单11-51中声明了Stack<int>的两个实例，两个实例都使用已由“运行时”为Stack<int>生成的代码。

代码清单11-51 声明Stack<T>类型的变量

```
Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

如果以后在代码中创建了另一个Stack，但用不同的值类型作为它的类型参数（如long或者用户自定义结构），“运行时”会生成泛型类型的另一个版本。使用具体化值类型的类，好处在于能获得较好的性能。除此之外，代码能避免转换和装箱，因为每个具体的泛型类都“原生地”（natively）包含值类型。

11.6.2 基于引用类型的泛型的实例化

对于引用类型，泛型的工作方式稍有不同。使用引用类型作为类型参数首次构造一个泛型类型时，“运行时”会在CIL代码中用object引用替换类型参数来创建一个具体化的泛型类型（而不是基于所提供的类型实参来创建一个具体化的泛型类型）。以后，每次用引用类型参数实例化一个构造好的类型，“运行时”都会重用以前生成好的泛型类型的版本——即使提供的引用类型与第一次不同。

例如，假定现在有两个引用类型，一个Customer类和一个Order类，而且创建了Customer类型的一个EntityDictionary，如下例所示：

```
EntityDictionary<Guid, Customer> customers;
```

访问这个类之前，“运行时”会生成EntityDictionary类的一个具体化版本。它不是将Customer作为指定的数据类型来存储，而是存储object引用。假定下一行代码创建Order类型的一个EntityDictionary：

```
EntityDictionary<Guid, Order> orders = new EntityDictionary<Guid, Order>();
```

和值类型不同，不会为使用Order类型的EntityDictionary创建EntityDictionary类的一个新的具体化版本。相反，会实例化使用object引用的EntityDictionary的一个实例，orders变量将引用这个实例。

为了确保类型安全性，针对用于替换类型参数的每个object引用，都会分配Order类型的一个内存区域，指针被设为指向那个内存区域。

假定随后用一行代码来实例化Customer类型的EntityDictionary：

```
customers = new EntityDictionary<Guid, Customer>();
```

和之前使用Order类型来创建EntityDictionary类一样，现在会实例化使用object引用的EntityDictionary的另一个实例，其中包含的指针会指向Customer类型的内存区域。因为编译器为引用类型的泛型类创建的具体化类被减少到了一个，所以泛型极大地减少了代码量。

引用类型的类型参数发生变化时，“运行时”使用的是相同的内部泛型类型定义。但是，假如类型参数是值类型，就不是这个行为了。例如，Dictionary<int, Customer>，Dictionary<Guid, Order>和Dictionary<long, Order>将要求不同的内部类型定义。

语言对比：Java——泛型

Sun完全是在编译器中为Java实现泛型，而不是在JVM（Java虚拟机）中实现的。Sun这样做是为了防止因为使用了泛型而需要分发新的JVM。

Java的实现使用了与C++中的“模板”和C# 中的“泛型”相似的语法，其中包括类型参数和约束。但是，由于它不区分对待值类型和引用类型，所以未修改的JVM不能为值类型支持泛型。正因为如此，Java中的泛型不具有C# 那样的执行效率。Java编译器需要返回数据的时候，都会插入来自指定约束的自动向下转型（如果声明了这样的一个转型的话），或者插入基本Object类型（如果没有声明的话）。除此之外，Java编译器在编译时生成一个具体化类型，它随即用于实例化任何已构造的类型。最后，由于JVM没有提供对泛型的原生支持，所以在执行时无法确定一个泛型类型实例的类型参数，“反射”的其他应用也受到了严格限制。

11.7 小结

从C# 2.0开始引入的泛型类型和泛型方法从根本上改变了C#开发人员的编码风格。在C# 1.0代码中，凡是使用了object的地方，在C# 2.0和更高的版本中都最好用泛型来代替。至少，集合问题应考虑用泛型来解决。避免转型对类型安全性的提升、避免装箱对性能的促进以及重复代码的减少为泛型赋予了无穷的魅力。

第16章将讨论最常用的泛型命名空间之一——System.Collections.Generic。该命名空间几乎完全由泛型类型构成。它清晰地演示了如何将最初使用object的类型转换为使用泛型。但在深入接触这些主题之前，我们先来探讨一下Lambda表达式。作为C# 3.0（和以后版本）最引人注目的一项增强，它极大地改进了操作集合的方式。

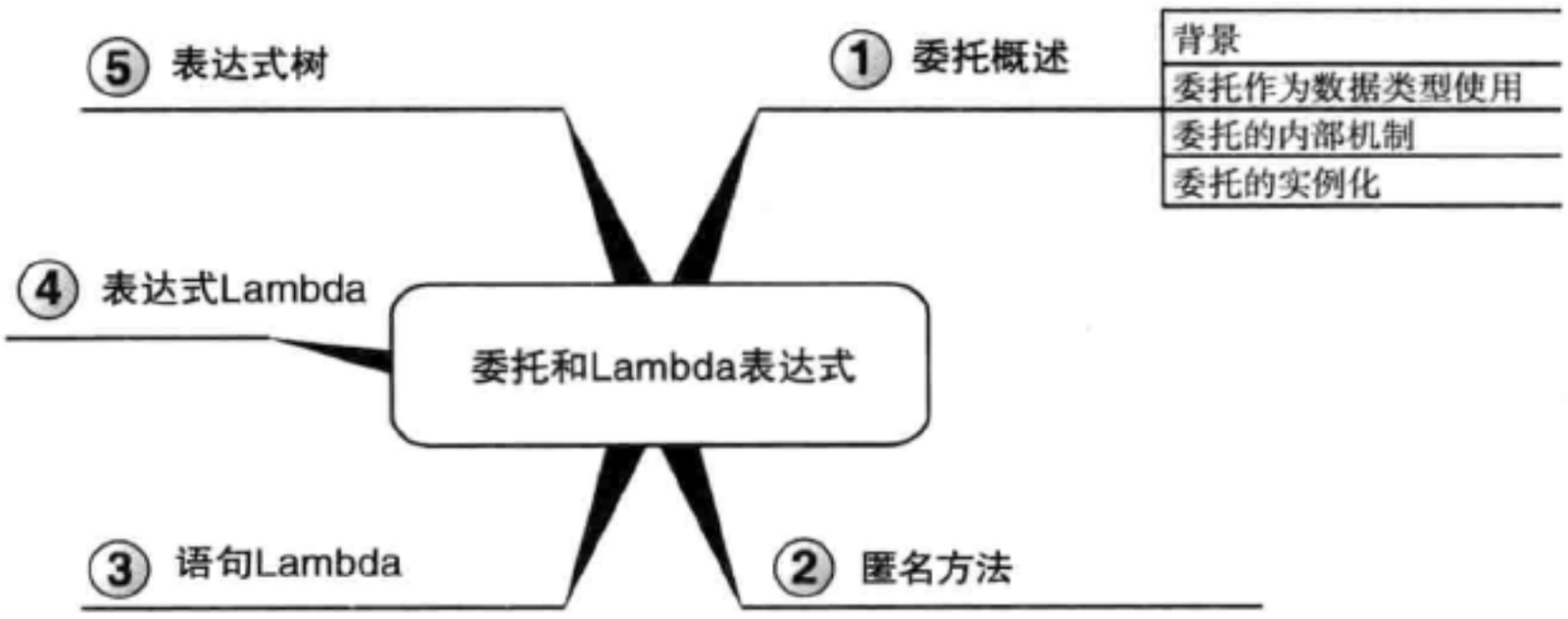
第12章

委托和Lambda表达式

在前几章中，我们全面讨论了如何创建类来封装数据以及对数据操作。随着创建的类越来越多，你会发现类和类的关系存在着一些常见的模式。一种常见的模式是向方法传递对象，该方法再调用对象的一个方法。例如，向方法传递一个IComparer<int>引用，被调用的方法本身可以在提供的对象上调用Compare()方法。在这种情况下，接口的作用只是向最终被调用的方法传递一个引用。所以，似乎不需要每次传递一个方法时都定义新接口。在本章中，我们讲述如何创建和使用被称为**委托**（delegate）的特殊类，它允许像处理其他任何数据那样处理对方法的引用。然后，我们展示如何使用Lambda表达式快速和简单地创建自定义委托。

Lambda表达式是从C# 3.0开始加入的。C# 2.0支持用**匿名方法**（anonymous method）这样一种不太优雅的语法来创建自定义委托。C#2.0之后的每个C#版本都支持匿名方法以保持向后兼容，但新写的代码应该弃用它，代之以Lambda表达式。本章将通过“高级主题”来描述如何使用匿名方法。只有要使用遗留的C# 2.0代码时才需要了解这些主题，否则完全可以忽略这些补充内容。

本章最后讨论了表达式树，它允许编译器在执行时对Lambda表达式进行分析。



12.1 委托概述

长期以来，经验丰富的C和C++程序员利用“函数指针”将对方法的引用作为实参传给另一个方法。C#使用**委托**提供相同的功能。委托允许捕捉对方法的引用，并像传递其他对象那样传递这个引用，像调用其他方法那样调用这个被捕捉的方法。来看看下面的例子。

12.1.1 背景

虽然效率不高，但冒泡排序或许是最简单的排序例程了。代码清单12-1展示了BubbleSort()方法。

代码清单12-1 BubbleSort()方法

```
static class SimpleSort1
{
  public static void BubbleSort(int[] items)
  {
      int i;
      int j;
      int temp;

      if(items==null)
      {
        return;
      }

      for (i = items.Length - 1; i >= 0; i--)
      {
          for (j = 1; j <= i; j++)
          {
              if (items[j - 1] > items[j])
              {
                  temp = items[j - 1];
                  items[j - 1] = items[j];
                  items[j] = temp;
              }
          }
      }
  }
  // ...
}
```

该方法对整数数组执行升序排序。

为了能选择升级或降序来排序整数，有两个方案可选：一是复制上述代码，然后将大于操作符替换成小于操作符。但是复制这么多代码只是改变了一个操作符，这似乎不是一个好主意；二是传递一个附加参数，指出如何排序，如代码清单12-2所示。

代码清单12-2 BubbleSort()方法，升序或降序

```
class SimpleSort2
{
  public enum SortType
  {
      Ascending,
      Descending
  }
```

```
    public static void BubbleSort(int[] items, SortType sortOrder)
    {
        int i;
        int j;
        int temp;

        if(items==null)
        {
          return;
        }

        for (i = items.Length - 1; i >= 0; i--)
        {
            for (j = 1; j <= i; j++)
            {
                bool swap = false;
                switch (sortOrder)
                {
                    case SortType.Ascending :
                        swap = items[j - 1] > items[j];
                        break;

                    case SortType.Descending :
                        swap = items[j - 1] < items[j];
                        break;
                }
                if (swap)
                {
                    temp = items[j - 1];
                    items[j - 1] = items[j];
                    items[j] = temp;
                }
            }
        }
    }
    // ...
}
```

然而，上述代码只是照顾到了两种可能的排序方式。假如想按字典顺序排序（即1，10，11，12，2，20，…），或者按其他方式排序， SortType值以及对应的switch分支的数量很快就会变得非常“恐怖”。

12.1.2 委托数据类型

为了增强灵活性和减少重复代码，可以将比较方法作为参数传给BubbleSort()方法。为了能将方法作为参数传递，必须要有一个能够表示方法的数据类型。这个数据类型就是**委托**，因为它“委托”调用对象所引用的方法。代码清单12-3对BubbleSort()方法进行了修改，它现在能获取一个委托参数。在本例中，委托数据类型是ComparisonHandler。

代码清单12-3 带有委托参数的BubbleSort()方法

```
class DelegateSample
{
// ...

    public static void BubbleSort(
```

```
        int[] items, ComparisonHandler comparisonMethod)
    {
        int i;
        int j;
        int temp;

        if(comparisonMethod == null)
        {
            throw new ArgumentNullException("comparisonMethod");
        }

        if(items==null)
        {
          return;
        }

        for (i = items.Length - 1; i >= 0; i--)
        {
            for (j = 1; j <= i; j++)
            {
                if (comparisonMethod(items[j - 1], items[j]))
                {
                  temp = items[j - 1];
                  items[j - 1] = items[j];
                  items[j] = temp;
                }
            }
        }
    }
    // ...
}
```

ComparisonHandler是委托类型，代表对两个整数进行比较的方法。在BubbleSort()方法中，可以使用comparisonMethod参数所引用的ComparisonHandler的实例来判断哪个整数更大。由于comparisonMethod代表一个方法，所以调用它的语法与直接调用方法的语法完全一致。在这个例子中，comparisonMethod委托获取两个整数参数，返回一个布尔值来指出第一个整数是否大于第二个。

要注意的是，ComparisonHandler委托是强类型的，它代表返回一个bool值，而且正好接受两个整数参数的方法。和其他方法调用一样，对委托的调用是强类型的。假如数据类型不匹配，C#编译器会报错。

12.1.3 委托类型的声明

前面描述了如何定义使用委托的方法，并介绍了如何将委托变量当作方法，从而简单地发出对委托的调用。然而，还必须学习如何声明委托类型。为了声明委托类型，要使用delegate关键字，后面跟着像是方法声明的东西。这个方法的签名是委托所引用的方法的签名。正常方法声明中方法名称的位置要替换成委托类型的名称。代码清单12-4展示了如何声明ComparisonHandler委托类型来要求两个整数并返回一个布尔值。

代码清单12-4 声明一个委托类型

```
public delegate bool ComparisonHandler (
  int first, int second);
```

就像类能嵌套在其他类中一样，委托也能嵌套在类中。假如委托声明出现在另一个类的内部，委托类型就会成为嵌套类型，如代码清单12-5所示。

代码清单12-5 声明嵌套的委托数据类型

```
class DelegateSample
{
  public delegate bool ComparisonHandler (
    int first, int second);
}
```

在这个例子中，声明的委托数据类型是DelegateSample.ComparisonHandler，因为它被定义成DelegateSample中的嵌套类型。

12.1.4 委托的实例化

在使用委托来实现BubbleSort()方法的最后一步中，你将学习如何调用方法并传递委托实例——具体地说，传递ComparisonHandler类型的实例。为了实例化委托，需要一个和委托类型自身的签名匹配的方法。对于ComparisonHandler，这个方法应获取两个整数，并返回bool值。方法名称无关紧要，但是方法的签名的剩余部分（参数和返回值）必须兼容委托的签名。代码清单12-6展示了与委托类型兼容的GreaterThan()方法。

代码清单12-6 声明与ComparisonHandler兼容的方法

```
public delegate bool ComparisonHandler (
  int first, int second);

class DelegateSample
{
  public static void BubbleSort(
    int[] items, ComparisonHandler comparisonMethod)
  {
      // ...
  }

  public static bool GreaterThan(int first, int second)
  {
    return first > second;
  }
  // ...
}
```

定义好方法之后，就可以调用BubbleSort()，并提供由委托捕捉到的方法的名称作为实参，如代码清单12-7所示。

代码清单12-7 使用方法名作为实参

```
public delegate bool ComparisonHandler (
   int first, int second);

class DelegateSample
{
  public static void BubbleSort(
```

```
        int[] items, ComparisonHandler comparisonMethod)
{
    // ...
}

public static bool GreaterThan(int first, int second)
{
    return first > second;
}

static void Main()
{
    int i;
    int[] items = new int[5];

    for (i=0; i < items.Length; i++)
    {
        Console.Write("Enter an integer: ");
        items[i] = int.Parse(Console.ReadLine());
    }

    BubbleSort(items, GreaterThan);

    for (int i = 0; i < items.Length; i++)
    {
        Console.WriteLine(items[i]);
    }

}
```

注意，ComparisonHandler委托是引用类型，但不必用new实例化它。从C# 2.0开始，从方法组（为方法命名的表达式）向委托类型的转换会自动创建一个新的委托对象。

高级主题：C# 1.0中的委托实例化

在代码清单12-7中，调用BubbleSort()时传递所需方法的名称（GreaterThan）作为一个实参就可以实例化委托。C#的第一个版本要求使用如代码清单12-8所示的较复杂的语法来实例化委托。

代码清单12-8 C# 1.0中将委托作为参数传递

```
BubbleSort(items,
           new ComparisonHandler(GreaterThan));
```

以后的版本支持上述两种语法。本书剩余的部分只使用更现代的、简洁的语法。

高级主题：委托的内部机制

委托实际是特殊的类。虽然C#标准没有确切规定类的层次结构应该是怎样的，但委托必须直接或间接地派生自System.Delegate。事实上，.NET中的委托类型总是派生自System.MulticastDelegate，后者又从System.Delegate派生，如图12-1所示。

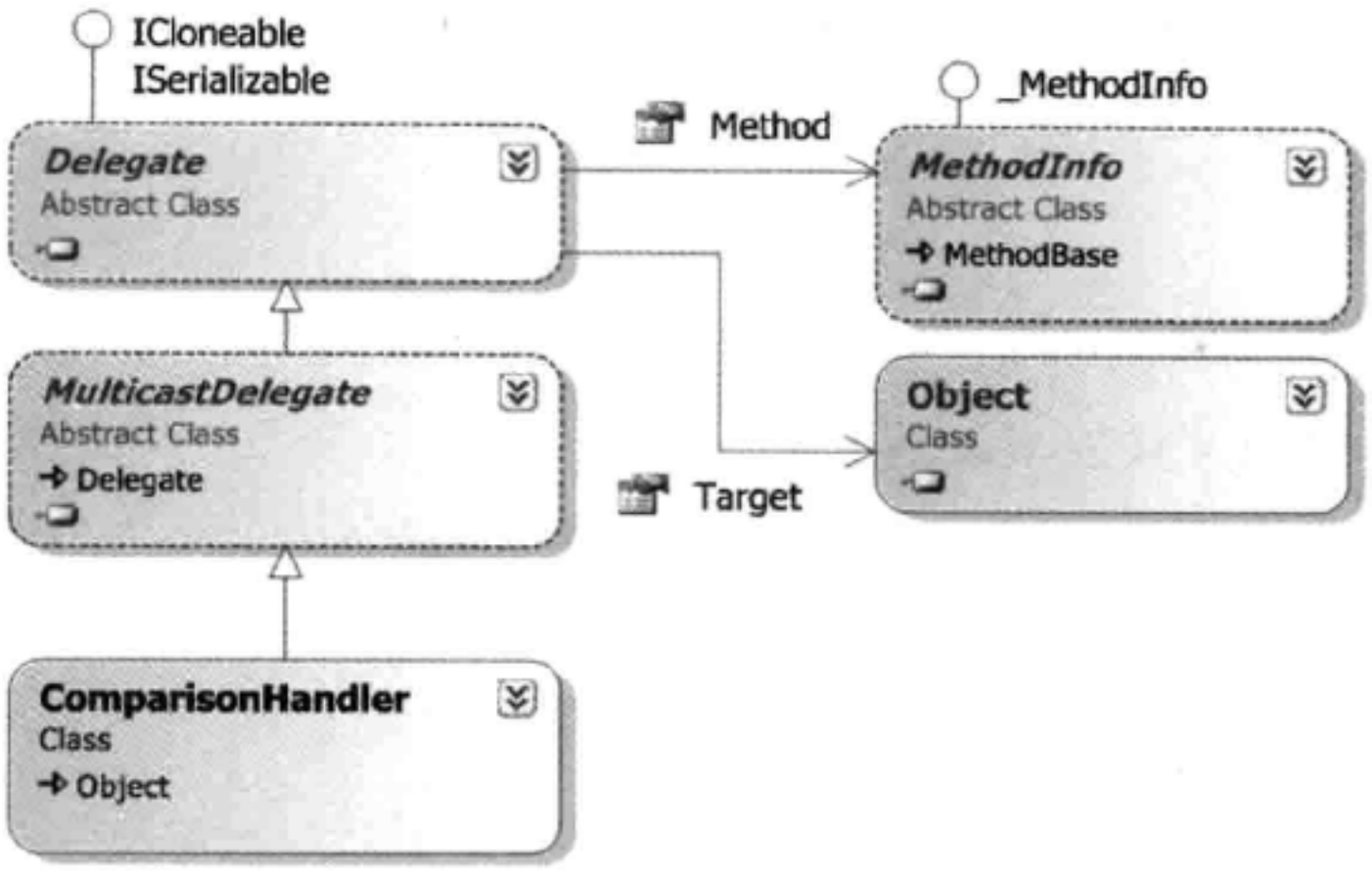


图12-1 委托类型的对象模型

第一个属性属于(System.Reflection.MethodInfo)类型，它是第17章要讨论的主题。MethodInfo描述了特定方法的签名，包括方法名称、参数和返回类型。除了MethodInfo，委托还需要一个对象实例，其中包含了要调用的方法。这正是第二个属性Target的作用。在静态方法的情况下，Target对应于类型自身。至于MulticastDelegate类的作用，将在下一章详细描述。

注意，所有委托都是不可变的。委托一旦创建好就无法更改。如果变量包含了委托的引用，还想引用其他不同的方法，那就必须创建一个新委托，并把它指派给这个变量。

虽然所有委托数据类型都是间接地从System.Delegate派生的，但C#编译器不允许声明直接或间接从System.Delegate或者System.MulticastDelegate派生的类。代码清单12-9的代码是无效的。

代码清单12-9 System.Delegate不能显式地作为基类

```
 // ERROR: 'ComparisonHandler' cannot

// inherit from special class 'System.Delegate'
// public class ComparisonHandler: System.Delegate
{
  // ...
}
```

通过传递委托来指定排序方式显然要比本章开头的方式灵活得多。例如，为了改为按字母排序，只需添加一个附加的委托，在比较过程中将整数转换为字符串。代码清单12-10提供了实现按字母排序的完整代码，输出12-1展示了结果。

代码清单12-10 使用其他与ComparisonHandler兼容的方法

```
using System;
class DelegateSample
{

  public delegate bool ComparisonHandler(int first, int second);

  public static void BubbleSort(
```

```
        int[] items, ComparisonHandler comparisonMethod)
    {
        int i;
        int j;
        int temp;

        for (i = items.Length - 1; i >= 0; i--)
        {
            for (j = 1; j <= i; j++)
            {
                if (comparisonMethod(items[j - 1], items[j]))
                {
                    temp = items[j - 1];
                    items[j - 1] = items[j];
                    items[j] = temp;
                }
            }
        }
    }

    public static bool GreaterThan(int first, int second)
    {
        return first > second;
    }

    public static bool AlphabeticalGreaterThan(
        int first, int second)
    {
        int comparison;
        comparison = (first.ToString().CompareTo(
            second.ToString()));

        return comparison > 0;
    }

    static void Main(string[] args)
    {
        int i;
        int[] items = new int[5];

        for (i=0; i<items.Length; i++)
        {
            Console.Write("Enter an integer: ");
            items[i] = int.Parse(Console.ReadLine());
        }

        BubbleSort(items, AlphabeticalGreaterThan);

        for (i = 0; i < items.Length; i++)
        {
            Console.WriteLine(items[i]);
        }
    }
}
```

输出12-1

```
Enter an integer: 1
Enter an integer: 12
Enter an integer: 13
Enter an integer: 5
Enter an integer: 4
1
12
13
4
5
```

按字母排序与按数值排序的结果不同。可以看到，和本章开头描述的方式相比，现在添加一个附加的排序机制是多么简单！

要想按字母排序，唯一要做的就是添加AlphabeticalGreaterThan方法，然后在调用BubbleSort()的时候传递该方法。

12.2 Lambda表达式

代码清单12-7和代码清单12-10展示了如何将表达式GreaterThan和AlphabeticalGreaterThan转换成委托类型，这些具名方法的参数类型和返回类型与委托类型兼容。你可能已经注意到了，GreaterThan方法的声明（public static bool GreaterThan(int first, int second)）比主体（return first > second;）冗长多了。这么简单的方法居然需要如此复杂的准备，这实在说不过去，而这些准备的目的只是为了能转换成委托类型。

为了解决问题，C# 2.0引入了非常精简的语法创建委托，C# 3.0则引入了更精简的语法。C# 2.0的相关特性称为**匿名方法**，C# 3.0的相关特性称为**Lambda表达式**。这两种语法统称为**匿名函数**（anonymous function）。这两种语法都合法，但新代码应优先使用Lambda表达式。除非要专门讲述C# 2.0匿名方法，否则本书都使用Lambda表达式。

Lambda表达式本身划分为两种类型：**语句Lambda**和**表达式Lambda**。图12-2展示了这些术语的关系。

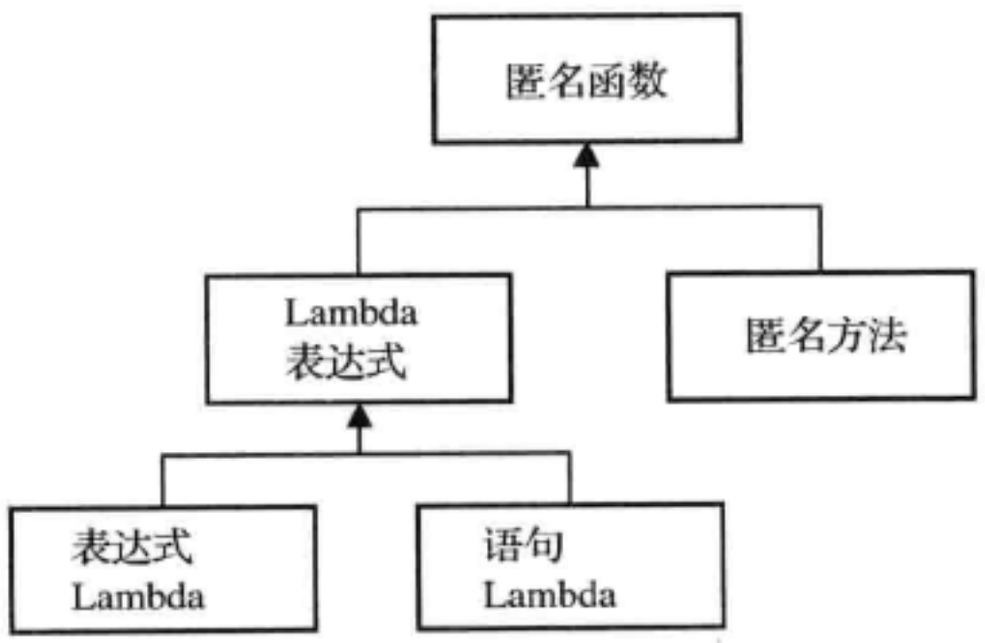


图12-2 与匿名函数有关的术语

12.2.1 语句Lambda

Lambda表达式的目的是在需要基于很简单的方法生成委托时，避免声明全新成员的麻烦。Lambda表达式有几种不同的形式。**语句Lambda**由形参列表，后跟Lambda操作符=>，然后跟一个代码块构成。代码清单12-11展示了和代码清单12-7的BubbleSort调用等价的功

能，只是用语句Lambda代表了比较方法，而不是创建完整的GreaterThan方法。如你所见，语句Lambda包含了GreaterThan方法声明中的大多数信息：形参声明和代码块都是相同的，但方法名称和修饰符没有了。

代码清单12-11 使用语句Lambda创建委托

```
// ...

BubbleSort(items,
    (int first, int second) =>
    {
        return first < second;
    }
);

// ...
```

查看含有Lambda操作符的代码时，可以在自己的脑海中将这个操作符替换成“用于”（go/goes to）。例如，在代码清单12-11中，可以将BubbleSort()的第二个参数理解成“整数first和second用于返回first小于second的结果”。

你可能已经注意到了，代码清单12-11的语法和代码清单12-7几乎完全一样，只是比较方法现在合理地出现在转换成委托类型的地方，而不是随便出现在别的地方，只能按名称来查找。方法名不见了，这正是这种方法被称为“匿名函数”的原因。返回类型不见了，但编译器知道这个Lambda表达式是要转换成要求bool返回类型的委托。编译器检查语句Lambda的代码块，验证每个return语句的表达式在要求返回bool的方法中是合法的。public修饰符不见了，因为方法不再是包容类的一个可访问的成员，因此没有必要描述它的可访问性。类似地，static修饰符也不需要了。围绕方法进行的准备工作已经大大减少了。

然而，语法还是冗长了一点。从委托类型就可以推断出Lambda表达式必然返回bool，同样可推断出两个参数必然是int类型，如代码清单12-12所示。

代码清单12-12 在语句Lambda中省略参数类型

```
// ...

BubbleSort(items,
    (first, second) =>
    {
        return first < second;
    }
);

// ...
```

通常，只要编译器能从Lambda表达式所转换成的委托推断出类型，所有Lambda表达式都不需要显式声明参数类型。然而，若指定类型能使代码更易读，C#也允许这样做。在不能推断出类型的情况下，C#要求显式地指定Lambda参数类型。只要显式指定了一个Lambda参数类型，所有参数类型都必须被指定，而且必须和委托参数类型完全一致。

规范

考虑在Lambda形参列表中省略类型，只要类型对于读者是显而易见的，或者是无关紧要的细节。

语法也许还能进一步精简，如代码清单12-13所示。当只有单个参数，而且类型可以推断时，这种Lambda表达式可省略围绕参数列表的圆括号。如果Lambda没有参数，或者有不止一个参数，或者显式指定了类型的单个参数，那么就必须将参数列表放到圆括号中。

代码清单12-13 只有一个输入参数的语句Lambda

```
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
  // ...
  IEnumerable<Process> processes = Process.GetProcesses().Where(
      process => { return process.WorkingSet64 > 1000000000; });
  // ...
```

在代码清单12-13中，Where()返回的是对物理内存占用超过1 GB的进程的一个查询。

代码清单12-14包含了一个无参数的语句Lambda。空参数列表要求圆括号。还要注意，在代码清单12-14中，语句Lambda的主体包含了多个语句。虽然语句Lambda允许包含任意数量的语句，但在它的语句块中，一般只使用两三条语句（泛型Func委托类型将在12.4节讨论）。

代码清单12-14 无参数的语句Lambda

```
using System;
  // ...
  Func<string> getUserInput =
      () =>
      {
          string input;
          do
          {
              input = Console.ReadLine();
          }
          while(input.Trim().Length == 0);
          return input;
      };
  // ...
```

12.2.2 表达式Lambda

语句Lambda的语法比完整的方法声明简单得多，可以不指定方法名、可访问性和返回类型，有时甚至可以不指定参数类型。表达式Lambda则更进一步。在代码清单12-12、代码清单12-13和代码清单12-14中，语句Lambda的代码块都只由一个return语句构成。其实在这种Lambda块中，唯一需要的就是准备返回的表达式，其他都可以省略，即表达式Lambda只有要返回的表达式，完全没有语句块。代码清单12-15等价于代码清单12-11，只是它使用了表达式Lambda而不是语句Lambda。

代码清单12-15 使用一个表达式Lambda来传递委托

```
// ...
BubbleSort(items, (first, second) => first < second );
// ...
```

通常，可以像阅读语句Lambda那样，将表达式Lambda中的Lambda操作符=>理解成“用于”。但有时候，把这个运算符理解成“变成”则更容易明白。在这个BubbleSort()调用的例子中，表达式Lambda是一个断言（返回一个布尔值）。在这种情况下，将Lambda操作符理解成“满足……条件”（such that或者where）会更清楚一些。所以，在阅读代码清单12-15的语句Lambda时，可以理解为：“first和second满足first小于second的条件”。

和null字面量相似，匿名函数不与任何类型关联。它的类型由它将要转换成的类型决定。也就是说，前面出现的所有Lambda表达式并非天生就是ComparisonHandler类型。它们只是与那个类型兼容，能转换成它。所以，不能对一个匿名方法使用typeof()操作符（参见第17章）。另外，只有在将匿名方法转换成一个特定类型后才能调用GetType()。

表12-1总结了关于Lambda表达式的其他注意事项。

表12-1 Lambda表达式的注意事项和例子

注意事项	例　子
Lambda表达式本身没有类型。所以，没有能直接从Lambda表达式中访问的成员，就连object的方法也不行	// 错误：'.'操作符不适用于 // 'lambda表达式类型的操作数 string s = ((int x) => x).ToString();
由于Lambda表达式没有类型，所以不能出现在is操作符的左侧	// 错误：'is'或'as'操作符的第一个操作数 // 不能是一个Lambda表达式或者匿名方法 bool b = ((int x) => x) is Func<int, int>;
Lambda表达式只能转换成兼容的委托类型。在例子中，返回int的Lambda表达式不能转换成代表“返回bool的方法”的委托类型	// 错误：Lambda表达式不兼容于 // Func<int, bool>类型 Func<int, bool> f = (int x) => x;
Lambda表达式没有类型，所以不能用于推断局部变量的类型	// 错误：不能将Lambda表达式赋给一个 // 隐式类型的局部变量 var v = x => x;
如果目的地在Lambda表达式的外部，C#就不允许在匿名函数内部使用跳转语句（break、goto、continue），反之亦然。在例子中，Lambda中的break语句试图跳转到Lambda外部的switch语句末尾	// 错误：控制不能离开匿名方法 // 或Lambda表达式主体 string[] args; Func<string> f; switch(args[0]) { case "/File": f = () => { if (!File.Exists(args[1])) break; return args[1]; }; // ... }

续表

注意事项	例　子
对于Lambda表达式引入的参数和局部变量，其作用域仅限于Lambda主体	// 错误：名称'first'在当前 // 上下文中不存在 Func<int, int, bool> expression = (first, second) => first > second; first++;
编译器的确定性赋值分析机制在Lambda表达式内部检测不到对外部局部变量进行初始化的情况	int number; Func<string, bool> f = text => int.TryParse(text, out number); if (f("1")) { // ERROR: Use of unassigned local variable System.Console.Write(number); }
	int number; Func<int, bool> isFortyTwo = x => 42 == (number = x); if (isFortyTwo(42)) { // ERROR: Use of unassigned local variable System.Console.Write(number); }

12.3 匿名方法

C# 2.0不支持Lambda表达式，而是使用称为**匿名方法**的语法。匿名方法很像语句Lambda，但缺少许多使Lambda变得简洁的特性。匿名方法必须显式指定每个参数的类型，而且必须有一个语句块。参数列表和代码块之间不使用Lambda操作符=>，而是在参数列表前面添加关键字 delegate，以强调匿名方法必须转换成一个委托类型。代码清单12-16展示了如何重写代码清单12.7、代码清单12-12和代码清单12-15来使用一个匿名方法。

代码清单12-16　在C# 2.0中传递一个匿名方法

```
// ...
BubbleSort(items,
    delegate(int first, int second)
    {
        return first < second;
    }
);
// ...
```

所以，在C# 3.0和以后的版本中，可以用两种非常相似的方式定义匿名函数，这多少有一点令人遗憾。

规范

避免在新代码中使用匿名方法语法，应该优先使用更简洁的Lambda表达式语法。

然而，有一个小特性是匿名方法支持而Lambda表达式不支持的。匿名方法在某些情况下可以彻底省略参数列表。

高级主题：无参数的匿名方法

和Lambda表达式不同，匿名方法允许彻底省略参数列表，前提是主体中不使用任何参数，而且委托类型只要求“值”参数（也就是说，不要求将参数标记为out或ref）。例如，对于以下匿名方法表达式：

```
delegate { return Console.ReadLine() != ""; }
```

它可以转换成任意要求返回bool的委托类型，而不管委托需要多少个参数。这一特性较少使用，但在阅读遗留代码时可能会遇到。

高级主题：“Lambda”表达式源起

“匿名方法”挺好理解。因为它们看起来和普通的方法声明相似，但没有方法名称，因此被称为匿名方法。但是，“Lambda表达式”中的“Lambda”是怎么来的呢？

Lambda表达式的概念来自阿隆佐•邱奇（Alonzo Church），他于20世纪30年代发明了用于函数研究的λ演算（lambda calculus）系统。如果函数要获取参数*x*，而其最终得到一个表达式是*y*，用邱奇的记号法，就可以在整个表达式的前面以小写的希腊字母λ作为前缀，再用点号分隔参数和值来表示这个函数。所以，C#的Lambda表达式x=>y用邱奇的记号法应该写成λx.y。由于在C#代码中不便输入希腊字母，而且点号在C#中有太多含义，所以C#的设计者选择使用“fat arrow”记号法（=>）。“Lambda表达式”这种叫法提醒人们匿名函数的理论基础是λ演算，即使希腊字母λ并没有实际地出现在文本中。

12.4 通用的委托：System.Func和System.Action

为了减少自定义委托类型的必要，.NET 3.5“运行时”库（对应C# 3.0）包含了一组通用的委托，其中大多数都是泛型。System.Func系列委托代表有返回值的方法，而System.Action系列委托代表返回void的方法。代码清单12-1展示了这些委托的签名。（in/out类型修饰符是C# 4.0才加入的，详情稍后讨论。）

代码清单12-17 Func和Action委托声明

```
public delegate void Action ();
public delegate void Action<in T>(T arg)
public delegate void Action<in T1, in T2>(
    in T1 arg1, in T2 arg2)
public delegate void Action<in T1, in T2, in T3>(
    T1 arg1, T2 arg2, T3 arg3)
public delegate void Action<in T1, in T2, in T3, in T4(
    T1 arg1, T2 arg2, T3 arg3, T4 arg4)
...
public delegate void Action<
```

```
    in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8,
    in T9, in T10, in T11, in T12, in T13, in T14, in T16(
        T1 arg1, T2 arg2, T3 arg3, T4 arg4,
        T5 arg5, T6 arg6, T7 arg7, T8 arg8,
        T9 arg9, T10 arg10, T11 arg11, T12 arg12,
        T13 arg13, T14 arg14, T15 arg15, T16 arg16)

public delegate TResult Func<out TResult>();
public delegate TResult Func<in T, out TResult>(T arg)
public delegate TResult Func<in T1, in T2, out TResult>(
    in T1 arg1, in T2 arg2)
public delegate TResult Func<in T1, in T2, in T3, out TResult>(
    T1 arg1, T2 arg2, T3 arg3)
public delegate TResult Func<in T1, in T2, in T3, in T4,
    out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)
...
public delegate TResult Func<
    in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8,
    in T9, in T10, in T11, in T12, in T13, in T14, in T16,
    out TResult>(
        T1 arg1, T2 arg2, T3 arg3, T4 arg4,
        T5 arg5, T6 arg6, T7 arg7, T8 arg8,
        T9 arg9, T10 arg10, T11 arg11, T12 arg12,
        T13 arg13, T14 arg14, T15 arg15, T16 arg16)
```

由于代码清单12-17中的委托定义是泛型的，因此可以直接使用它们而不用自定义一个委托。Func委托的最后一个类型参数总是委托的返回类型。其他类型参数依次对应于委托参数的类型。代码清单12-3中的BubbleSort方法要求的是一个能返回bool并获取两个int参数的委托。所以，不用声明并使用ComparisonHandler委托类型。相反，可以像下面这样声明BubbleSort方法：

```
void BubbleSort(int[] items,
    Func<int, int, bool> comparisonMethod) { ... }
```

在许多情况下，NET 3.5 Framework添加的Func委托都能完全避免定义自己的委托类型。然而，如果要想显著增强代码的可读性，还是应该声明自己的委托类型。例如，ComparisonHandler显式表明了委托的用途>而Func<int, int, bool>显式表明了委托的形参和返回类型。

规范

考虑定义自己的委托类型对于可读性的提升是否比使用预定义泛型委托类型所带来的便利性来得重要。

12.4.1 委托没有结构相等性

.NET委托类型不具备结构相等性（structural equality）。也就是说，不能将某个委托类型的对象引用转换成不相关的委托类型，即使这两个委托类型的形参和返回类型完全一致。例如，不能将一个ComparisonHandler引用直接赋给一个Func<int, int, bool>变量，即使两个委托类型都代表获取两个int并返回一个bool的方法。遗憾的是，在需要结构一致但不相关的委托类型的情况下，为了使用给定类型的委托，唯一的办法是创建一个新委

托，让它引用旧委托的Invoke方法。例如，假定有一个ComparisonHandler类型的变量c，需要把c的值赋给Func<int, int, bool>类型的变量f，那么可以写成 f = c.Invoke;。

然而，通过C# 4.0添加的对可变性的支持，现在可以在某些引用类型之间进行引用转换。来考虑以下逆变的例子：由于void Action<in T>(T arg)有一个in类型参数修饰符，所以可以将Action<object>类型的委托引用赋给Action<string>类型的变量。

许多人都觉得委托的逆变性不好理解。只需记住，适合任何对象的行动必定适合任何字符串。反之则不然，只适合字符串的行动不适合所有对象。类似地，Func系列委托类型对它的返回类型协变，这是通过TResult的out类型参数修饰符来指示的。所以，Func<string>类型的委托引用可以赋给Func<object>类型的变量。代码清单12-8展示了委托的协变性和逆变性。

代码清单12-18 为委托使用可变性

```
// Contravariance
Action<object> broadAction =
  (object data) =>
  {
      Console.WriteLine(data);
  };
Action<string> narrowAction = broadAction;

// Covariance
Func<string> narrowFunction =
  () =>Console.ReadLine();
Func<object> broadFunction = narrowFunction;

// Contravariance & Covariance Combined
Func<object, string> func1 =
  (object data) => data.ToString();
Func<string, object> func2 = func1;
```

代码清单的最后一部分在一个例子中合并了两种可变性的概念，演示了假如同时涉及in和out类型参数，协变性和逆变性是如何同时发生的。

实现泛型委托类型的引用转换，这是C# 4.0添加协变和逆变转换的关键原因之一。（另一个原因是提供IEnumerable<out T>的协变性支持。）

高级主题：Lambda表达式和匿名方法的内部机制

Lambda表达式（和匿名方法）并非CLR内部的固有构造。相反，当编译器遇到匿名方法时，会把它转换成特殊的隐藏类、字段和方法，从而实现你希望的语义。也就是说，C#编译器会为这个模式生成实现代码，这样就不需要开发人员自己去实现。例如，给定代码清单12-11、代码清单12-12、代码清单12-15或者代码清单12-16，C#编译器将生成类似于代码清单12-19所示的CIL代码。

代码清单12-19 与Lambda表达式的CIL代码等价的C#代码

```
class DelegateSample
{
  // ...
  static void Main(string[] args)
  {
```

```
        int i;
        int[] items = new int[5];

        for (i=0; i<items.Length; i++)
        {
            Console.Write("Enter an integer:");
            items[i] = int.Parse(Console.ReadLine());
        }


        BubbleSort(items,
            DelegateSample.__AnonymousMethod_00000000);

        for (i = 0; i < items.Length; i++)
        {
            Console.WriteLine(items[i]);
        }

    }

    private static bool __AnonymousMethod_00000000(
        int first, int second)
    {
        return first < second;
    }

}
```

在这个例子中，编译器将匿名函数转换成一个单独的声明的静态方法。这个静态方法再实例化成一个委托，并作为参数被传递。毫不奇怪，编译器生成的代码看上去有点儿像代码清单12-7中的原始代码，也就是后来应用匿名函数语法进行简化的代码。但是，编译器执行的代码转换要复杂得多，在涉及“外部变量”的时候，不只是将匿名函数重写成一个静态方法那样简单。

12.4.2 外部变量

在Lambda表达式外部声明的局部变量（包括包容方法的参数）称为那个Lambda表达式的**外部变量**。（this引用虽然从技术上说不是一个变量，但也被视为外部变量。）当Lambda主体使用一个外部变量的时候，就说该变量被这个Lambda表达式**捕捉**。代码清单12-20利用外部变量统计BubbleSort()执行了多少次比较。输出12-2展示了结果。

代码清单12-20 在Lambda表达式中使用外部变量

```
class DelegateSample
{

  // ...

  static void Main(string[] args)
  {

      int i;
      int[] items = new int[5];
      int comparisonCount=0;

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

```
        {
            Console.Write("Enter an integer:");
            items[i] = int.Parse(Console.ReadLine());
        }

        BubbleSort(items,
            (int first, int second) =>
            {
                comparisonCount++;
                return first < second;
            }
        );

        for (i = 0; i < items.Length; i++)
        {
            Console.WriteLine(items[i]);
        }

        Console.WriteLine("Items were compared {0} times.",
                          comparisonCount);
    }
}
```

输出12-2

```
Enter an integer:5
Enter an integer:1
Enter an integer:4
Enter an integer:2
Enter an integer:3
5
4
3
2
1
Items were compared 10 times.
```

要注意的是，comparisonCount出现在Lambda表达式的外部，在其内部递增。在调用了BubbleSort()方法之后，在控制台上输出comparisonCount的值。

局部变量的生存期一般都和它的作用域绑定；一旦控制离开作用域，变量的存储位置就不再有效。但是，如果Lambda表达式捕捉了外部变量，根据该表达式创建的委托可能具有比局部变量在一般情况下更长（或更短）的生存期。委托每次被调用时，都必须能安全地访问外部变量。在这种情况下，被捕捉的变量的生存期被延长了。这个生存期至少与存活时间最长的委托对象一样长。（也许更长；编译器如何生成代码来延长外部变量生存期是一种实现细节，可能会随不同的编译器实现发生变化。）

总之，是由C#编译器负责生成CIL代码，在匿名方法和声明它的方法之间共享comparisonCount。

高级主题：外部变量的CIL实现

C#编译器为捕捉外部变量的匿名方法生成的CIL代码要比为一个什么都不捕捉的简单匿名方法生成的CIL代码复杂。代码清单12-21展示了与实现代码清单12-20的外部变量的CIL代码对应的C#代码。

代码清单12-21 与外部变量CIL代码对应的C#代码

```
class DelegateSample
{
  // ...
  private sealed class __LocalsDisplayClass_00000001
  {
      public int comparisonCount;
      public bool __AnonymousMethod_00000000(
          int first, int second)
      {
          comparisonCount++;
          return first < second;
      }
  }
  // ...
  static void Main(string[] args)
  {
      int i;
      __LocalsDisplayClass_00000001 locals =
          new __LocalsDisplayClass_00000001();
      locals.comparisonCount=0;
      int[] items = new int[5];

      for (i=0; i<items.Length; i++)
      {
          Console.Write("Enter an integer:");
          items[i] = int.Parse(Console.ReadLine());
      }

      BubbleSort(items, locals.__AnonymousMethod_00000000);
      for (i = 0; i < items.Length; i++)
      {
          Console.WriteLine(items[i]);
      }

      Console.WriteLine("Items were compared {0} times.",
                        locals.comparisonCount);
  }
}
```

注意，被捕捉的局部变量永远不会被“传递”到别的地方，也永远不会被“复制”到别的地方。相反，被捕捉的局部变量（comparisonCount）作为一个实例字段（而非局部变量）实现，从而延长了其生存期。所有使用局部变量的地方都被重写为使用那个字段。

生成的__LocalsDisplayClass类称为**闭包**（closure），它是一个数据结构（一个C#类），其中包含一个表达式以及对表达式进行求值所需的变量（C#中的公共字段）。

高级主题：不小心捕捉循环变量

思考一下代码清单12-22的输出是什么？

代码清单12-22 在C# 5.0中捕捉循环变量

```
class CaptureLoop
{
  static void Main()
  {
```

```
        var items = new string[] { "Moe", "Larry", "Curly" };
        var actions = new List<Action>();
        foreach (string item in items)
        {
            actions.Add( ()=> { Console.WriteLine(item); } );
        }
        foreach (Action action in actions)
        {
            action();
        }
    }
}
```

大多数人都觉得结果应该如输出12-3所示，在C# 5.0中也确实如此。然而，在之前的C#版本中，结果却如输出12-4所示。

输出12–3 C# 5.0的输出

```
Moe
Larry
Curly
```

输出12–4 C# 4.0的输出

```
Curly
Curly
Curly
```

Lambda表达式捕捉变量并总是使用其最新的值，而不是捕捉并保留变量在委托创建时的值。这通常正是你希望的行为。例如，代码清单12-20捕捉了变量comparisonCount，目的正是确保递增时使用其最新的值。对于循环变量没什么两样。捕捉一个循环变量时，每个委托都捕捉同一个循环变量。当这个循环变量发生变化时，捕捉它的每个委托都看到了变化。所以，无法指责C# 4.0的行为——虽然这几乎肯定不是代码作者想要的。

C# 5.0对此进行了更改，认为每一次循环迭代，foreach循环的循环变量都应该是“新”变量。所以，每次创建委托，捕捉的都是不同的变量，而不再共享同一个变量。然而，需要注意的是，这个更改不适用于for循环。用for循环写类似的代码，for语句头中声明的任何循环变量在被捕捉时都被看成是同一个外部变量。如果你要编写在C# 5.0和之前的版本中行为一致的代码，请使用如代码清单12-23所示的模式。

代码清单12–23 在C# 5.0之前捕捉循环变量的解决方案

```
class DoNotCaptureLoop
{
  static void Main()
  {
      var items = new string[] { "Moe", "Larry", "Curly" };
      var actions = new List<Action>();
      foreach (string item in items)
      {
          string _item = item;
          actions.Add(
              ()=> { Console.WriteLine(_item); } );
```

```
        }
        foreach (Action action in actions)
        {
            action();
        }
    }
}
```

这样就明确了每次循环迭代都有一个新变量，每一个委托捕捉的都是一个不同的变量。

规范

避免在匿名函数中捕捉循环变量。

12.4.3 表达式树

Lambda表达式提供了一种简洁的语法来定义代码中“内联”的方法，使其能转换成委托类型。表达式Lambda（不是语句Lambda和匿名方法）也能转换成**表达式树**（expression tree）。委托是一个对象，允许像传递其他任何对象那样传递一个方法，并在任何时候调用该方法。表达式树也是一个对象，允许传递编译器对Lambda主体的分析。但是，这个分析有什么用呢？显然，编译器的分析在生成CIL时对编译器有用。但为什么说在程序执行时，开发人员拥有代表这种分析的一个对象是有用的呢？下面看一个例子。

1. Lambda表达式作为数据使用

来看看以下代码中的Lambda表达式：

```
persons.Where(
    person => person.Name.ToUpper() == "INIGO MONTOYA");
```

假定persons是一个Person数组，和Lambda表达式实参对应的Where方法形参具有委托类型Func<Person, bool>。编译器生成一个方法来包含Lambda表达式主体中的代码。编译器生成代码来创建一个委托用以代表所生成的方法，并将这个委托传给Where方法。Where方法返回一个查询对象；一旦执行查询，就将委托应用于数组的每个成员来判断查询结果。

现在假定persons不是Person[]类型，而是代表远程数据库表的对象，表中包含数百万人的数据。表中每一行的信息都可以从服务器传输到客户端，客户端可以创建一个Person对象来代表那一行。调用Where将返回一个代表查询的对象。那么，在客户端上执行查询，如何判断结果？

一个技术是将几百万行数据从服务器传输到客户端。为每一行都创建一个Person对象，根据Lambda创建一个委托，再针对每个Person执行这个委托。这在概念上与数组的情况一致，但代价过于高昂。

第二个技术则要好得多，它是将Lambda的含义（过滤掉姓名不是Inigo Montoya的每一行）发送给服务器。数据库服务器本来就很擅长快速执行这种筛选。然后，服务器只将符合条件的少数几行传输到客户端；而不是先创建几百万个Person对象，再把它们几乎全部丢弃。客户端只创建那些经过服务器判断与查询相匹配的对象。但是，怎样将Lambda的含义发送给服务器呢？

这正是要在语言中添加**表达式树**这一概念的动机。转换成表达式树的Lambda表达式

对象代表的是对Lambda表达式进行描述的数据，而不是编译好的、用于实现匿名函数的代码。由于表达式树代表数据而非编译好的代码，所以能在执行时分析Lambda，用分析得到的数据构造查询，并针对数据库执行。如代码清单12-24所示，由Where()方法接收的表达式树可以转换成SQL查询，这个查询将传给数据库。

代码清单12-24 将表达式树转换成SQL Where子句

```
persons.Where( person => person.Name.ToUpper() == "INIGO MONTOYA");
select * from Person where upper(Name) = 'INIGO MONTOYA';
```

传给Where()的表达式树指出Lambda实参由以下几部分构成：

- 对Person的Name属性的读取；
- 对string的ToUpper()方法的调用；
- 一个常量值"INIGO MONTOYA"；
- 一个相等性操作符==。

Where()方法获取这些数据，检查数据并构造一个SQL查询字符串，从而把这些数据转换成一个SQL where子句。然而，表达式树并非只能转换成SQL语句。可以构造一个表达式树计算程序（evaluator），将表达式转换成任意查询语言。

2. 表达式树作为对象图使用

在执行时，一个转换成表达式树的Lambda成为一个对象图，其中包含了来自System.Linq.Expressions命名空间的对象。图中的“根”对象代表Lambda本身，这个对象引用代表参数、一个返回类型和一个主体表达式的对象，如图12-3所示。对象图包含编译器根据Lambda推断出来的所有信息。执行时可利用这些信息创建一个查询；另外，根Lambda表达式有一个Compile方法，它能“动态”生成CIL并创建一个实现了指定Lambda的委托。

图12-4展示了一个Lambda主体中的一元表达式和二元表达式的对象图中的类型。

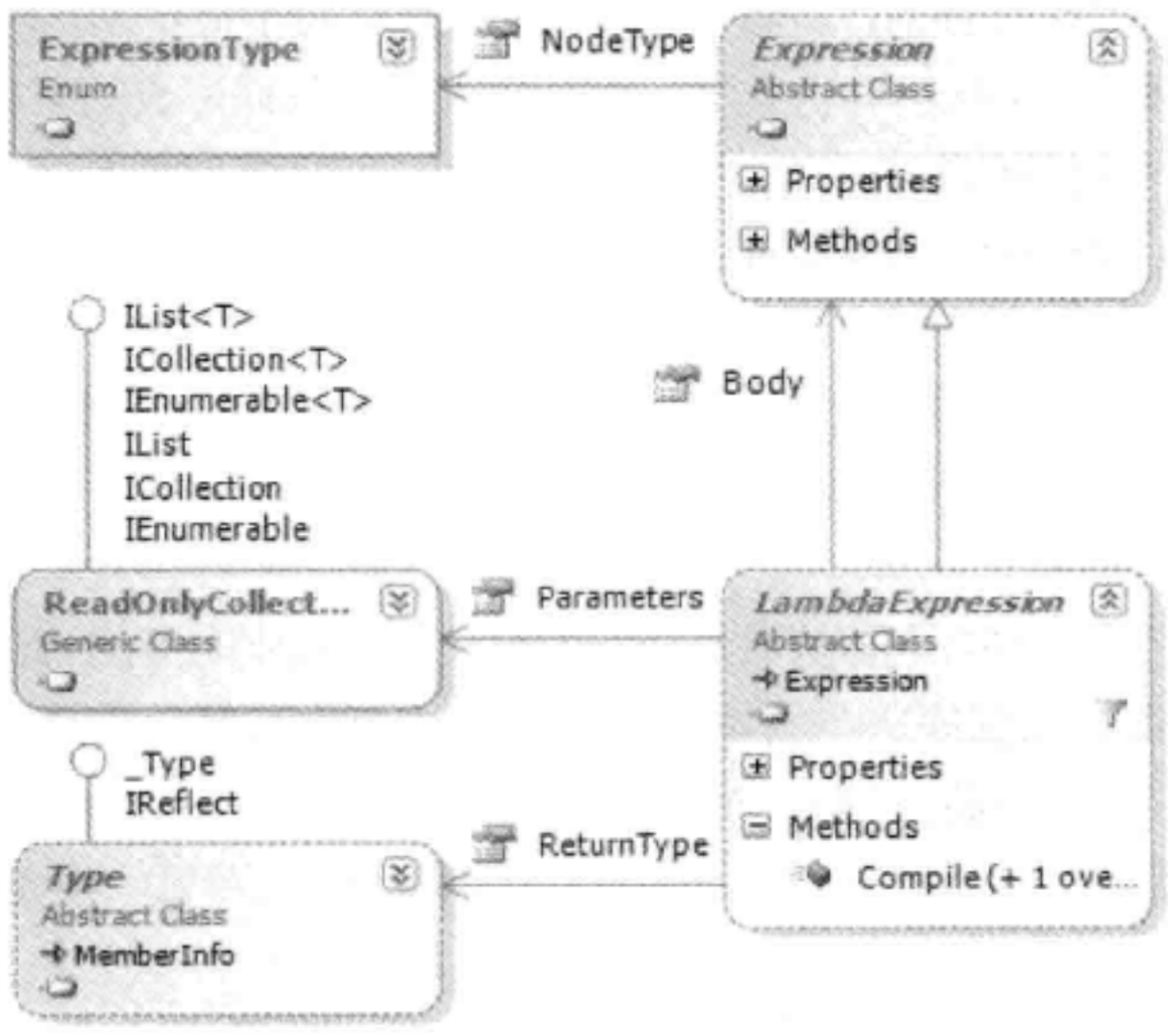


图12-3 Lambda表达式树类型

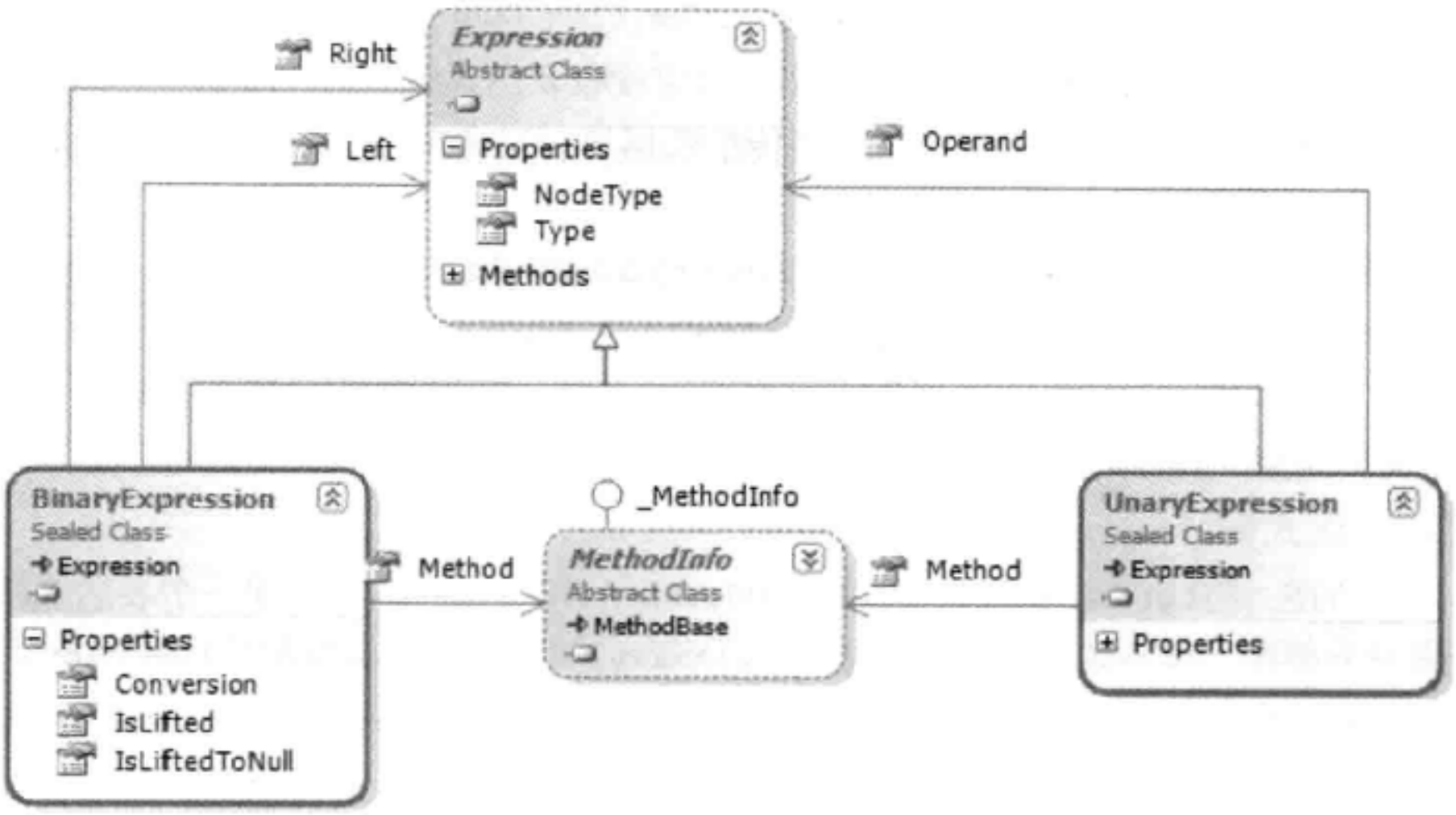


图12-4 一元和二元表达式树类型

UnaryExpression代表一个形如-count这样的表达式。它具有Expession类型的单个子操作数（Operand）。BinaryExpression有两个子表达式，Left 和Right。两个类型都通过NodeType属性标识具体的运算符。两者都从基类Expression派生。还有其他约30种表达式类型，比如NewExpression、ParameterExpression、MethodCallExpression、LoopExpression等，它们能表示C#和Visual Basic中的几乎每一种可能的表达式。

3. 委托和表达式树的比较

不管是转换成一个委托还是转换成一个表达式树，一个Lambda表达式都会在编译时进行全面的语义分析，从而验证其有效性。转换成委托的Lambda使编译器将这个Lambda作为一个方法生成，并生成代码在执行时创建对那个方法的委托。转换成表达式树的Lambda会使编译器生成代码，在执行时创建LambdaExpression的一个实例。但是在使用LINQ时，编译器是如何知道是应该生成委托，或是在本地执行查询，还是生成一个表达式树将查询信息发送给远程数据库服务器呢？

像Where()这样的用于构建LINQ查询的方法是扩展方法。扩展了IEnumerable<T>接口的方法获取委托参数，扩展了IQueryable<T>接口的方法获取表达式树参数。所以，编译器能够根据查询的集合类型来判断应该根据作为实参提供的Lambda创建委托还是创建表达式树。例如以下代码中的Where()方法：

```
persons.Where( person => person.Name.ToUpper() =="INIGO MONTOYA");
```

在System.Linq.Enumerable类中声明的扩展方法签名是：

```
public IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> collection,
    Func<TSource, bool> predicate);
```

在System.Linq.Queryable类中声明的扩展方法签名是：

```
public IQueryable<TSource> Where<TSource>(
    this IQueryable<TSource> collection,
    Expression<Func<TSource, bool>> predicate);
```

编译器根据persons在编译时的类型决定使用哪个扩展方法；如果是一个能转换成

IQueryable<Person>的类型，就选择来自System.Linq.Queryable的方法。它将Lambda转换成一个表达式树。执行时，persons引用的对象接收表达式树数据，用那些数据构造一个SQL查询，在请求查询结果时将这个查询传给数据库。当请求查询结果时，将查询发送给数据库并生成结果，调用Where的结果是一个对象。

如果persons不能隐式地转换成IQueryable<Person>，但能隐式地转换成IEnumerable<Person>，那么就选择来自System.Linq.Enumerable的方法，Lambda被转换成一个委托。当请求查询结果时，将生成的委托作为断言应用于集合的每个成员，并生成与断言匹配的结果，调用Where的结果是一个对象。

4. 解析表达式树

如前所述，将Lambda表达式转换成Expression<TDelegate>将创建一个表达式树而不是创建一个委托。本章前面讲过如何将(x,y)=>x>y这样的Lambda转换成Func<int, int, bool>这样的委托类型。要将同样的Lambda转换成一个表达式树，只需要把它转换成Expression<Func<int, int, bool>>，如代码清单12-25所示。然后就可以检查生成的对象，显示和它的结构相关的信息，而且可以显示更复杂的表达式树的信息。

要注意的是，将表达式树的实例传给Console.WriteLine()方法，会自动将表达式树转换成一个描述性的字符串形式。为表达式树生成的对象都重写了ToString()，以便程序员在调试时能一眼看出表达式树的内容。

代码清单12-25 解析表达式树

```
using System;
using System.Linq.Expressions;

class Program
{
    static void Main()
    {
        Expression<Func<int, int, bool>> expression;
        expression = (x, y) => x > y;
        Console.WriteLine("-------------{0}-------------",
            expression);
        PrintNode(expression.Body, 0);
        Console.WriteLine();
        Console.WriteLine();
        expression = (x, y) => x * y > x + y;
        Console.WriteLine("-------------{0}-------------",
            expression);
        PrintNode(expression.Body, 0);
        Console.WriteLine();
        Console.WriteLine();
    }
public static void PrintNode(Expression expression,
    int indent)
{
    if (expression is BinaryExpression)
        PrintNode(expression as BinaryExpression, indent);
    else
        PrintSingle(expression, indent);
}
private static void PrintNode(BinaryExpression expression,
  int indent)
```

```
{
    PrintNode(expression.Left, indent + 1);
    PrintSingle(expression, indent);
    PrintNode(expression.Right, indent + 1);
}
private static void PrintSingle(
    Expression expression, int indent)
{
    Console.WriteLine("{0," + indent * 5 + "}{1}",
      "", NodeToString(expression));
}
private static string NodeToString(Expression expression)
{
    switch (expression.NodeType)
    {
        case ExpressionType.Multiply:
            return "*";
        case ExpressionType.Add:
            return "+";
        case ExpressionType.Divide:
            return "/";
        case ExpressionType.Subtract:
            return "-";
        case ExpressionType.GreaterThan:
            return ">";
        case ExpressionType.LessThan:
            return "<";
        default:
            return expression.ToString() +
                " (" + expression.NodeType.ToString() + ")";
        }
    }
}
```

在输出12-5中，我们可以看到，Main()内部的 Console.WriteLine()语句将表达式树的主体作为文本打印出来。

输出12-5

```
------------- (x, y) => x > y -------------
    x (Parameter)
>
    y (Parameter)


------------- (x, y) => (x * y) > (x + y) -------------
        x (Parameter)
    *
        y (Parameter)
>
        x (Parameter)
    +
        y (Parameter)
```

要注意的重点在于，表达式树是一个数据集合，可以通过遍历数据将其转换成另一种格式。在本例中，我们是将表达式树转换成描述性字符串。但是，表达式树也可以被转换成另一种查询语言中的表达式。

通过使用递归PrintNode()方法，证明了一个表达式树是由零个或者多个其他表达式

树构成的。代表Lambda的“根”树通过其Body属性引用Lambda的主体。每个表达式树节点都包含枚举类型ExpressionType的一个NodeType属性，描述了它是哪一种表达式。有多个表达式类型，例如BinaryExpression、ConditionalExpression、LambdaExpression、MethodCallExpression、ParameterExpression和ConstantExpression。每个类型都是从Expression派生的。

注意，虽然现在表达式树库包含的对象能表示C#和Visual Basic的大多数语句，但两种语言都不支持将语句Lambda转换成表达式树。只有表达式Lambda才能转换成表达式树。

12.5 小结

本章首先讨论了委托以及如何把它作为方法的引用或者回调来使用。可利用委托传递一组能在不同位置调用的指令。这些指令不是立即调用的，而是在完成编码之后，再在其他位置调用。

本章讨论了C# 3.0引入的Lambda表达式的概念。Lambda表达式语法取代（但不是消除）了C# 2.0的匿名方法语法。不管哪种语法，程序员都可以利用它们直接将一组指令赋给一个变量，而不必显式地定义一个包含这些指令的方法。这使得程序员可以在方法内部动态地编写指令——这是一个很强大的概念，它通过一个称为LINQ（语言集成查询）的API大幅简化了集合编程。

最后，本章讨论了表达式树的概念，描述了它们如何编译成对象来表示对一个Lambda表达式的语义分析，而不是表示委托的具体实现。利用这一特性，LINQ to SQL和LINQ to XML等库能解释表达式树，并在CIL之外的上下文中使用它。

“Lambda表达式”这个术语兼具“语句Lambda”和“表达式Lambda”的意思 。换言之，语句Lambda和表达式Lambda都是Lambda表达式的类型。

本章提到但未详述的一个概念是多播委托。下一章将详细讨论多播委托，并解释如何利用它们来实现事件的“发布—订阅”模式。

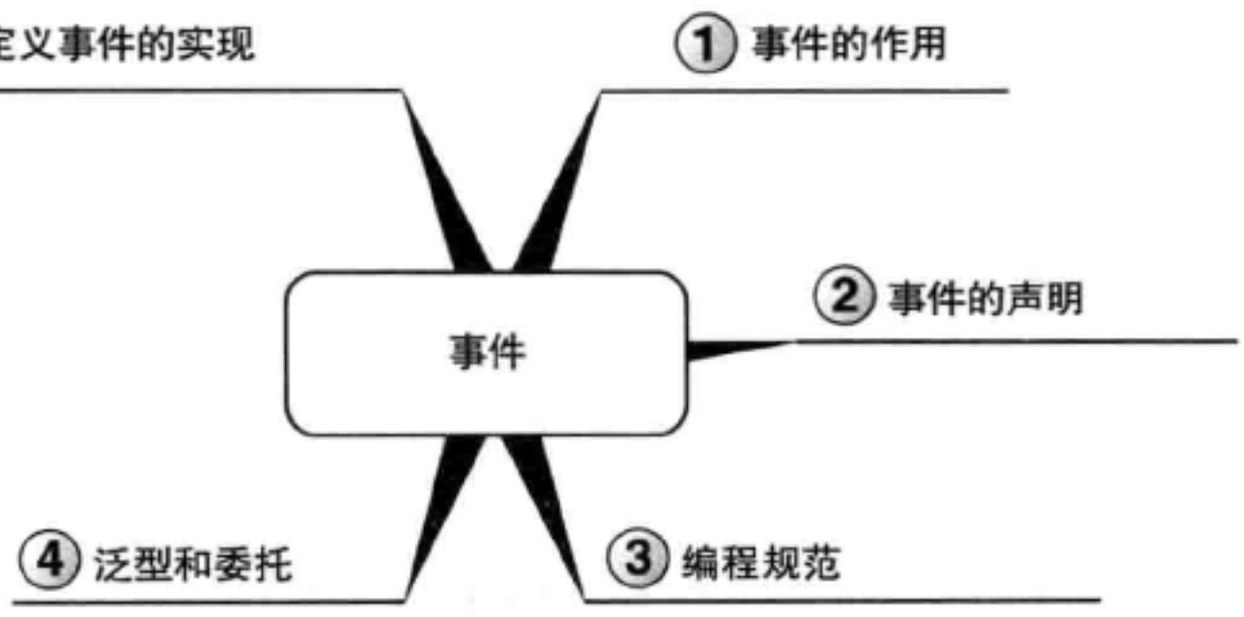
第13章

事　件

上一章讲述了如何用委托类型的实例引用方法，并通过委托来调用那个方法。委托本身又是一个更大的模式（pattern）的基本单位，这个模式称为publish-subscribe（发布—订阅）。委托的使用及其对publish-subscribe模式的支持是本章的重点。本章描述的所有内容几乎都可以单独使用委托来实现。然而，本章所着眼的事件构造提供了额外的“封装性”，使publish-subscribe模式更容易实现，更不容易出错。

在上一章中，所有委托都只引用一个方法。然而，一个委托值是可以引用一系列方法的，这些方法将顺序调用。这样的委托称为**多播委托**（multicast delegate）。这样一来，单一事件（比如对象状态的改变）的通知就可以发布给多个订阅者。

虽然事件在C# 1.0中就有了，但C# 2.0对泛型的引入极大地改变了编码规范，因为使用泛型委托数据类型意味着不再需要为每种可能的事件签名声明一个委托。所以，本章的最低起点是C# 2.0。但是，仍在使用C# 1.0的读者也不是不能使用事件，只是必须声明自己的委托数据类型（参见第12章）。



13.1 使用多播委托来编码Observer模式[①]

来考虑一个温度控制的例子。在这个假想的情形中，一个加热器（Heater）和一个冷却器（Cooler）连接到同一个自动调温器。为了控制加热器和冷却器的打开和关闭，要向它们通知温度的变化。自动调温器将温度的变化发布给多个订阅者——也就是加热器和冷却器。下一节将研究具体的代码[②]。

13.1.1 定义订阅者方法

首先定义Heater和Cooler对象，如代码清单13-1所示。

代码清单13-1 Heater和Cooler事件订阅者的实现

```
class Cooler
{
  public Cooler(float temperature)
  {
      Temperature = temperature;
  }

  public float Temperature
  {
      get{return _Temperature;}
      set{_Temperature = value;}
  }
  private float _Temperature;

  public void OnTemperatureChanged(float newTemperature)
  {
      if (newTemperature > Temperature)
      {
          System.Console.WriteLine("Cooler: On");
      }
      else
      {
          System.Console.WriteLine("Cooler: Off");
      }
  }
}

class Heater
{
  public Heater(float temperature)
  {
      Temperature = temperature;
  }

  public float Temperature
  {
      get{return _Temperature;}
      set{_Temperature = value;}
  }
```

① 在C#中，多播委托的实现是一个通用的模式，目的是避免大量的手工编码。这个模式称为Observer（观察者）或者publish-subscribe模式，它要应对的是这样一种情形：需要将单一事件的通知（如对象状态发生的一个变化）广播给多个订阅者（subscriber）。——译者注

② 在这个例子中，我们使用的是自动调温器（thermostat）这个词，因为人们习惯于它在加热或冷却系统中使用。但从技术上说，温度计（thermometer）一词更适合这个例子。

```
private float _Temperature;

public void OnTemperatureChanged(float newTemperature)
{
    if (newTemperature < Temperature)
    {
        System.Console.WriteLine("Heater: On");
    }
    else
    {
        System.Console.WriteLine("Heater: Off");
    }
  }
}
```

除了温度比较，两个类几乎完全一致（事实上，如果在OnTemperatureChanged方法中使用对一个比较方法的委托，两个类还可以再减少一个）。每个类都存储了启动设备所需的温度。此外，两个类都提供了OnTemperatureChanged()方法。调用OnTemperatureChanged()方法的目的是向Heater和Cooler类指出温度已发生改变。在方法的实现中，用newTemperature同存储好的触发温度进行比较，从而决定是否让设备启动。

两个OnTemperatureChanged()方法都是订阅者方法。作为订阅者方法，很重要的一点在于，它们的参数和返回类型必须与来自Thermostat类的委托匹配，Thermostat类的详情将在下一节讨论。

13.1.2 定义发布者

Thermostat类负责向heater和cooler对象实例报告温度变化。代码清单13-2展示了Thermostat类。

代码清单13-2 定义事件发布者Thermostat

```
public class Thermostat
{
  // Define the event publisher
  public Action<float> OnTemperatureChange { get; set; }

  public float CurrentTemperature
  {
      get{return _CurrentTemperature;}
      set
      {
          if (value != CurrentTemperature)
          {
              _CurrentTemperature = value;
          }
      }
  }
  private float _CurrentTemperature;
}
```

Thermostat包含一个名为OnTemperatureChange的属性，它具有Action<float>委托类型。OnTemperatureChange存储了订阅者列表。注意，只需一个委托字段即可存储所有订阅者。换言之，来自同一个发布者的温度变化通知会同时被Cooler和Heater类接收。

Thermostat的最后一个成员是CurrentTemperature属性。它负责设置和获取由Thermostat类报告的当前温度值。

13.1.3 连接发布者和订阅者

最后，将所有这些东西都放到一个Main()方法中。代码清单13-3展示了一个示例Main()。

代码清单13-3 连接发布者和订阅者

```
class Program
{
  public static void Main()
  {
      Thermostat thermostat = new Thermostat();
      Heater heater = new Heater(60);
      Cooler cooler = new Cooler(80);
      string temperature;

      thermostat.OnTemperatureChange +=
          heater.OnTemperatureChanged;
      thermostat.OnTemperatureChange +=
          cooler.OnTemperatureChanged;

      Console.Write("Enter temperature: ");
      temperature = Console.ReadLine();
      thermostat.CurrentTemperature = int.Parse(temperature);
  }
}
```

上述代码使用+=操作符来直接赋值，向OnTemperatureChange委托注册了两个订阅者，即heater.OnTemperatureChanged和cooler.OnTemperatureChanged。

通过获取用户输入的温度值，可以设置thermostat（自动调温器）的CurrentTemperature（当前温度）。然而，目前还没有写任何代码将温度变化发布给订阅者。

13.1.4 调用委托

Thermostat类的CurrentTemperature属性每次发生变化时，你都希望**调用委托**来通知订阅者（heater和cooler）温度的变化。为此，需要修改CurrentTemperature属性来保存新值，并向每个订阅者发出一个通知，如代码清单13-4所示。

代码清单13-4 调用委托（不带有检查null值部分）

```
public class Thermostat
{
  ...
  public float CurrentTemperature
  {
      get{return _CurrentTemperature;}
      set
      {
          if (value != CurrentTemperature)
          {
              _CurrentTemperature = value;

              // INCOMPLETE:  Check for null needed
```

```
                // Call subscribers
                OnTemperatureChange(value);
            }
        }
    }
    private float _CurrentTemperature;
}
```

现在，对CurrentTemperature的赋值包含了一些特殊的逻辑，可以向订阅者通知CurrentTemperature发生的变化。为了向所有订阅者发出通知，只需执行一个简单的C#语句，即OnTemperatureChange(value);。这个语句将温度的变化发布给cooler和heater对象。在此，只需执行一个调用，即可向多个订阅者发出通知——这正是将委托更明确地称为“多播委托”的原因。

13.1.5 检查null值

在代码清单13-4中，遗漏了事件发布代码的一个重要部分。假如当前没有订阅者注册接收通知，则OnTemperatureChange为null，执行OnTemperatureChange(value)语句就会引发一个NullReferenceException异常。为了避免这个问题，有必要在触发事件之前检查null值。代码清单13-5演示了具体如何做。

代码清单13-5 调用一个委托

```
public class Thermostat
{
...
public float CurrentTemperature
{
    get{return _CurrentTemperature;}
    set
    {
        if (value != CurrentTemperature)
        {
            _CurrentTemperature = value;
            // If there are any subscribers
            // then notify them of changes in
            // temperature
            Action<float> localOnChange =
                OnTemperatureChange;
            if(localOnChange != null)
            {
                // Call subscribers
                localOnChange(value);
            }
        }
    }
    private float _CurrentTemperature;
}
```

我们并不是一开始就检查null值，而是首先将OnTemperatureChange赋给另一个委托变量localOnChange。这个简单的修改可确保在检查null值和发送通知之间，假如所有OnTemperatureChange订阅者都（由一个不同的线程）移除，那么不会引发NullReferenceException异常。

规范

要在调用委托前检查它的值是不是null值。

高级主题：将“-=”操作符应用于委托会返回一个新实例

既然委托是引用类型，那么肯定有人会觉得奇怪：为什么赋值给一个局部变量，再用那个局部变量就可以保证null检查的线程安全性？由于localOnChange指向的位置就是OnTemperatureChange指向的位置，所以很自然的结论就是：OnTemperatureChange中发生的任何变化都会在localOnChange中反映出来。

但实情并非如此。事实上，对OnTemperatureChange -= <listener>的任何调用都不会从OnTemperatureChange删除一个委托而使它的委托比之前少一个。相反，会赋值一个全新的多播委托，原始的多播委托不受任何影响（localOnChange也指向那个原始的多播委托）。

高级主题：线程安全的委托调用

如前所述，由于订阅者可以由不同的线程从委托中增加或删除，所以在进行null值检查前有必要将委托引用复制到一个局部变量中。但是，这虽然能防止调用空委托，却不能防止所有可能的竞态条件。例如，一个线程进行复制，另一个线程将这个委托重置为null，然后原始线程可以调用委托的前一个值，借此通知一个已经不再在订阅者列表中的订阅者。在多线程程序中，订阅者应确保在这种情况下的健壮性。一个“过气”的订阅者随时都可能被调用。

13.1.6 委托操作符

为了合并Thermostat例子中的两个订阅者，要使用+=操作符。这样会获取第一个委托，并将第二个委托添加到委托链中。第一个委托的方法返回后，会调用第二个委托。从委托链中删除委托，则要使用-=操作符，如代码清单13-6所示。

代码清单13-6 使用+=和-=委托操作符

```
// ...
Thermostat thermostat = new Thermostat();
Heater heater = new Heater(60);
Cooler cooler = new Cooler(80);

Action<float> delegate1;
Action<float> delegate2;
Action<float> delegate3;

delegate1 = heater.OnTemperatureChanged;
delegate2 = cooler.OnTemperatureChanged;

Console.WriteLine("Invoke both delegates:");
delegate3 = delegate1;
delegate3 += delegate2;
delegate3(90);

Console.WriteLine("Invoke only delegate2");
delegate3 -= delegate1;
```

```
delegate3(30);
// ...
```

代码清单13-6的结果如输出13-1所示。

输出13–1

```
Invoke both delegates:
Heater: Off
Cooler: On
Invoke only delegate2
Cooler: Off
```

除此之外，还可以使用+和-操作符来合并委托，如代码清单13-7所示。

代码清单13–7 使用+和-委托操作符

```
// ...
Thermostat thermostat = new Thermostat();
Heater heater = new Heater(60);
Cooler cooler = new Cooler(80);

Action<float> delegate1;
Action<float> delegate2;
Action<float> delegate3;

// Note: Use new Action(
//       cooler.OnTemperatureChanged) for C# 1.0 syntax.
delegate1 = heater.OnTemperatureChanged;
delegate2 = cooler.OnTemperatureChanged;

Console.WriteLine("Combine delegates using + operator:");
delegate3 = delegate1 + delegate2;
delegate3(60);

Console.WriteLine("Uncombine delegates using - operator:");
delegate3 = delegate3 - delegate2;
delegate3(60);
// ...
```

使用赋值操作符会清除之前的所有订阅者，并允许用新订阅者替换它们。这是委托很容易让人犯错的一个设计，因为在本来应该使用+=操作符的时候，很容易就会错误地写成=。解决这个问题的良方是事件，详情将在稍后的13.2节讲述。

应注意的是，无论+、-还是它们的复合赋值版本（+=和-=），在内部都是使用静态方法System.Delegate.Combine()和System.Delegate.Remove()来实现的。两个方法都获取delegate类型的两个参数。第一个方法Combine()会连接两个参数，将两个委托的调用列表按顺序连接到一起。第二个方法Remove()则搜索由第一个参数指定的委托链，删除由第二个参数指定的委托。

对于Combine()方法，一个有趣的地方在于，它的两个参数都可以为null。如果其中任何一个参数为null，Combine()就返回非空的那个。如果两个都为null，则Combine()返回null。这就解释了为什么可以调用thermostat.OnTemperatureChange += heater.OnTemperatureChanged;而不引发异常（即使thermostat.OnTemperatureChange尚未赋值）。

13.1.7 顺序调用

图13-1展示了heater和cooler的顺序通知。

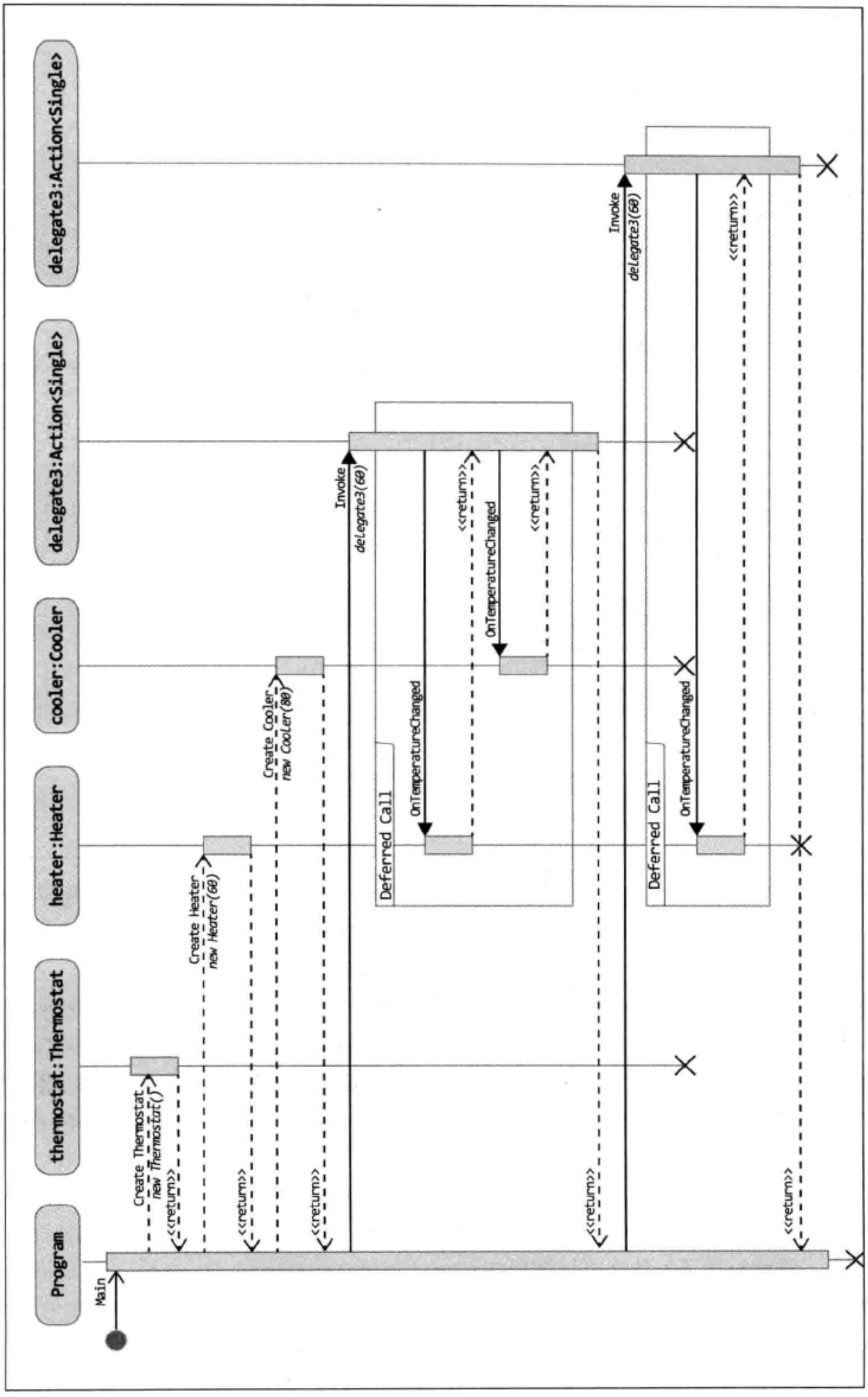


图13-1 委托调用顺序图

虽然代码中只包含一个简单的OnTemperatureChange()调用，但这个调用会广播给两个订阅者，使cooler和heater都会收到温度发生变化的通知。假如添加更多的订阅者，它们也会收到OnTemperatureChange()的通知。

虽然一个OnTemperatureChange()调用造成每个订阅者都收到通知，但它们仍然是顺序调用的，而不是同时调用，因为它们全都在一个执行线程上调用。

高级主题：多播委托的内部机制

为了理解事件是如何工作的，你需要回顾第12章中我们第一次探讨System.Delegate类型的内部机制的部分。delegate关键字是派生自System.MulticastDelegate的一个类型的别名。System.MulticastDelegate则是从System.Delegate派生的，后者由一个对象引用（以满足非静态方法的需要）和一个方法引用构成。创建委托时，编译器自动使用System.MulticastDelegate类型而不是System.Delegate类型。MulticastDelegate类包含一个对象引用和一个方法引用，这和它的Delegate基类一样。但除此之外，它还包含对另一个System.MulticastDelegate对象的引用。

向多播委托添加方法时，MulticastDelegate类会创建委托类型的一个新实例，在新实例中为新增的方法存储对象引用和方法引用，并在委托实例列表中添加新的委托实例作为下一项。实际上，MulticastDelegate类维护着一个Delegate对象链表。从概念上讲，可以像图13-2那样表示Thermostat的例子。

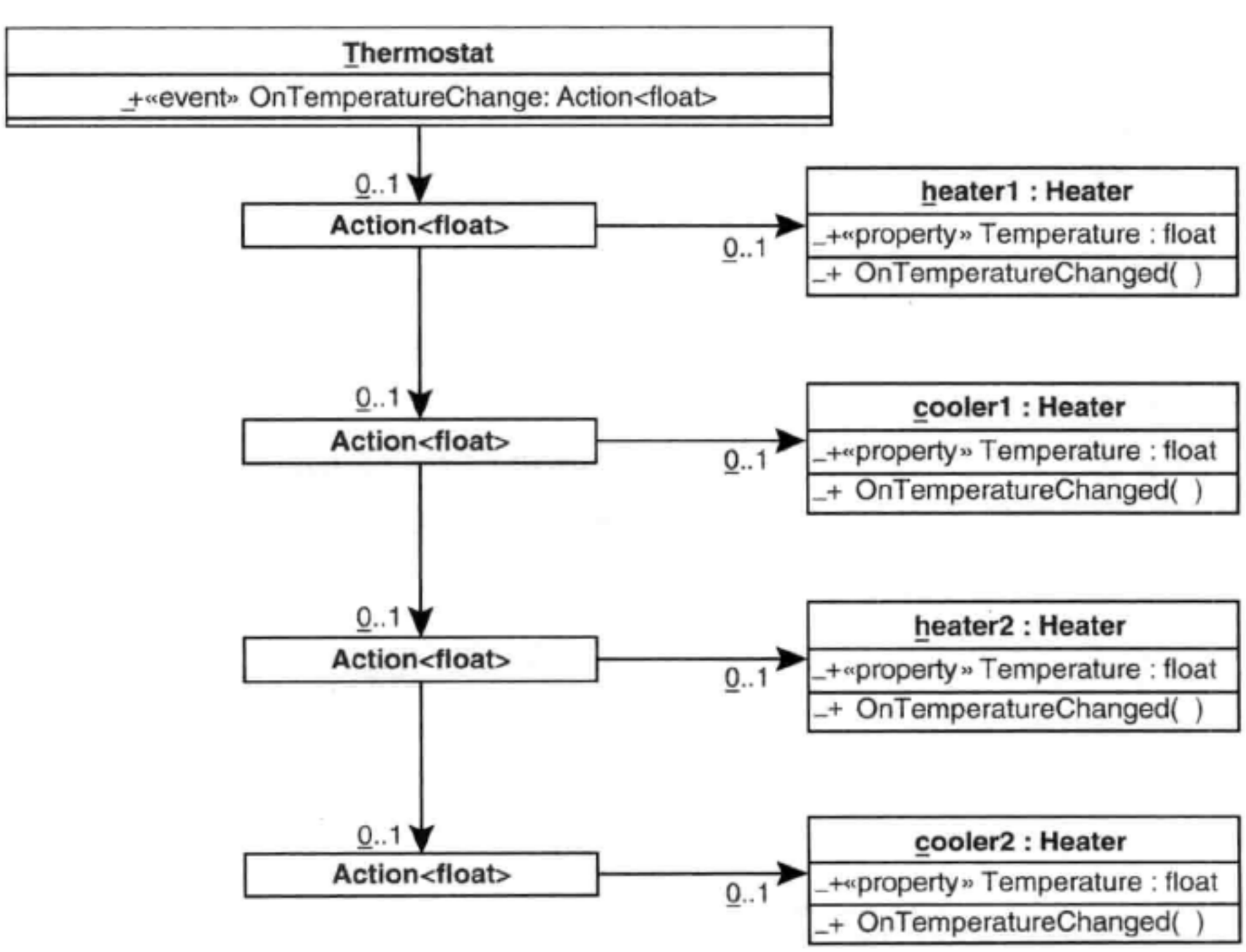


图13-2 链接到一起的多播委托

调用多播委托时，链表中的委托实例会被顺序调用。通常，委托是按照它们添加时的顺序调用的，但CLI规范并未对此做出硬性规定，而且这个顺序可能被覆盖。所以，程序员不应依赖于一个特定的调用顺序。

13.1.8 错误处理

错误处理凸显了顺序通知可能造成的问题。假如一个订阅者引发了异常，链中的后续订阅者就收不到通知。例如，假定修改Heater的OnTemperatureChanged()方法，使它引发异常，那么会发生什么？如代码清单13-8所示。

代码清单13-8 OnTemperatureChanged()引发异常

```
class Program
{
  public static void Main()
  {
      Thermostat thermostat = new Thermostat();
      Heater heater = new Heater(60);
      Cooler cooler = new Cooler(80);
      string temperature;

      thermostat.OnTemperatureChange +=
          heater.OnTemperatureChanged;
      // Using C# 3.0.  Change to anonymous method
      // if using C# 2.0
      thermostat.OnTemperatureChange +=
          (newTemperature) =>
              {
                  throw new InvalidOperationException();
              };
      thermostat.OnTemperatureChange +=
          cooler.OnTemperatureChanged;

      Console.Write("Enter temperature: ");
      temperature = Console.ReadLine();
      thermostat.CurrentTemperature = int.Parse(temperature);
  }
}
```

图13-3是更新过的顺序图。

虽然cooler和heater已进行了订阅来接收消息，但Lambda表达式异常会使链发生中断，造成cooler对象收不到通知。

为了避免这个问题，使所有订阅者都能收到通知（不管之前的订阅者有过什么行为），必须手动遍历订阅者列表，并单独调用它们。代码清单13-9展示了需要在CurrentTemperature属性中进行的更新。结果如输出13-2所示。

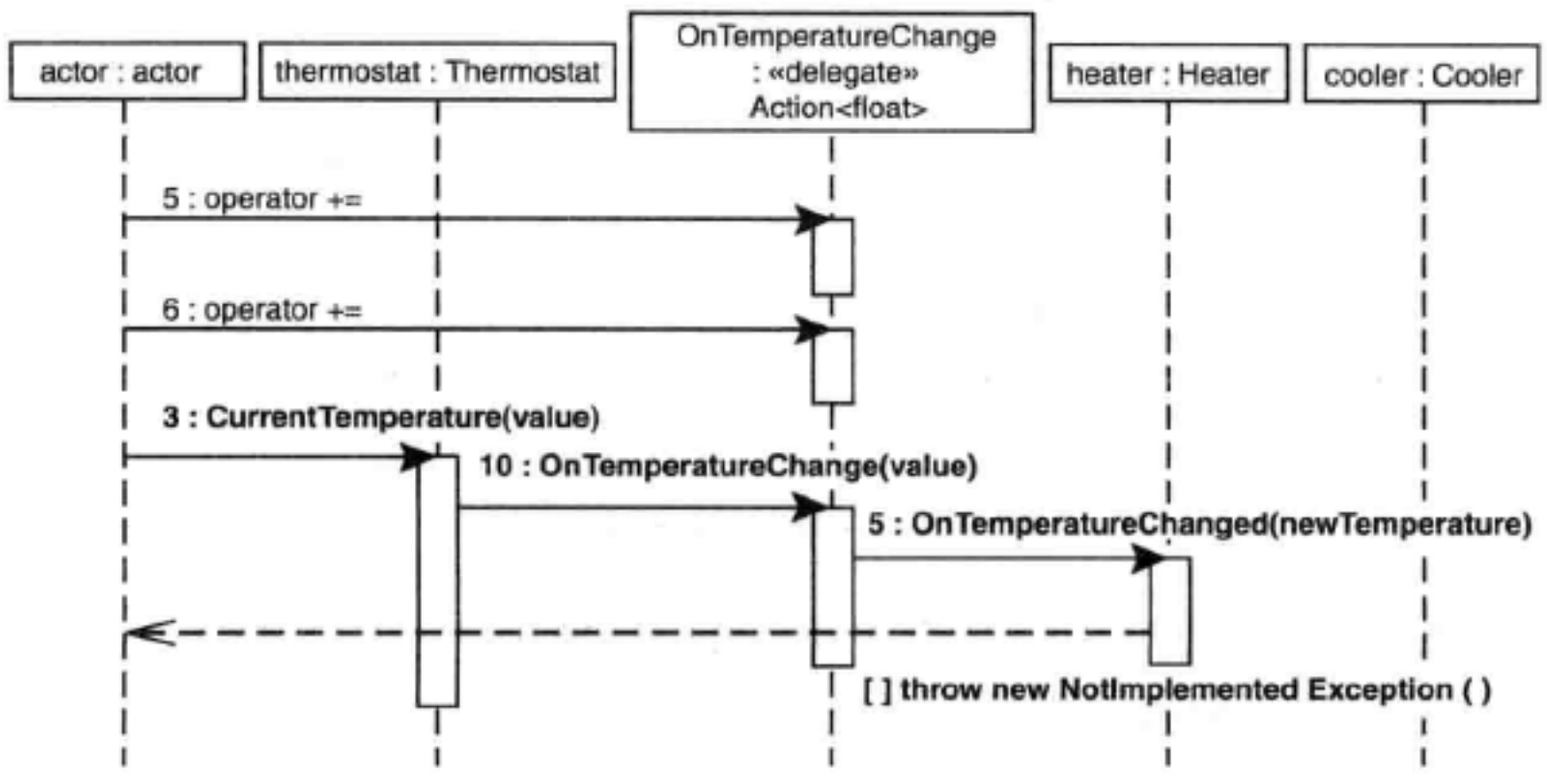


图13-3 委托调用顺序图（已添加异常）

代码清单13-9 处理来自订阅者的异常

```
public class Thermostat
{
  // Define the event publisher
  public Action<float> OnTemperatureChange;

  public float CurrentTemperature
  {
      get{return _CurrentTemperature;}
      set
  {
      if (value != CurrentTemperature)
      {
          _CurrentTemperature = value;
          if(OnTemperatureChange != null)
          {
              List<Exception> exceptionCollection =
                  new List<Exception>();
              foreach (
                  Action<float> handler in
                  OnTemperatureChange.GetInvocationList())
              {
                  try
                  {
                      handler(value);
                  }
                  catch (Exception exception)
                  {
                      exceptionCollection.Add(exception);
                  }
              }
              if (exceptionCollection.Count > 0)
              {
                  throw new AggregateException(
                      "There were exceptions thrown by
OnTemperatureChange Event subscribers.",
                      exceptionCollection);
              }
          }
      }
  }
  private float _CurrentTemperature;
}
```

输出13-2

```
Enter temperature: 45
Heater: On
Error in the application
Cooler: Off
```

这个代码清单演示了你可以从委托的GetInvocationList()方法获得一份订阅者列表。枚举该列表中的每一项，可以返回给单独的订阅者。如果随后将每个订阅者调用都放到一个try/catch块中，就可以先处理好任何出错的情形，再继续循环迭代。在这个例子中，尽管委托侦听者（delegate listener）引发了一个异常，但cooler仍会接收到温度发生改变